Project 1: Covid Testing Waiting Line Simulator

DUE: Sunday, _September 13 at 11:59pm

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, example readme.txt provided).
- Have a style and pass the automatic style checker (see P0).
- Comment your code well in JavaDoc style AND pass the automatic JavaDoc checker (see P0).
- Have code that compiles with the command: **javac *.java** in your user directory

You may:

- Add additional methods and variables, however these methods **must be private**.
- Use any arrays anywhere in your program (including the data field)

You may NOT:

- Make your program part of a package.
- Add additional public methods or variables
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, HashSet, etc.).
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add any additional import statements (or use the "fully qualified name" to get around this requirement).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the `project1.zip` and unzip it. This will create a folder `section-yourGMUUserName-p1`;
- Rename the folder replacing `section` with the `001`, `002`, `005` etc. based on the lecture section you are in;
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address;
- After renaming, your folder should be named something like: `001-krusselc-p1`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p1.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
001-krusselc-p1.zip --> 001-krusselc-p1 --> JavaFile1.java
                                            JavaFile2.java
```

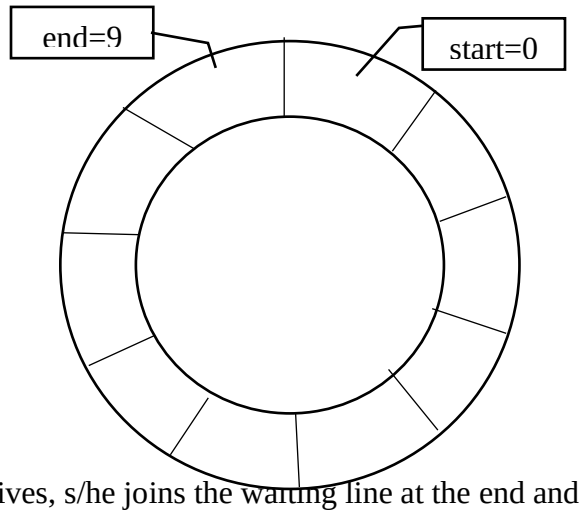
- Submit to blackboard.

Grading Rubric

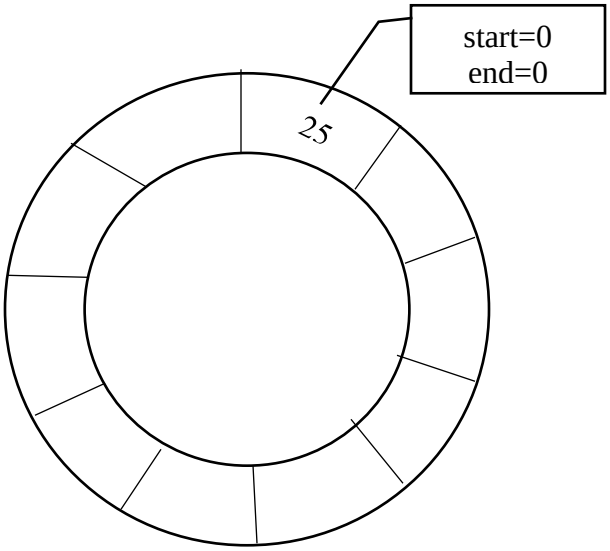
Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

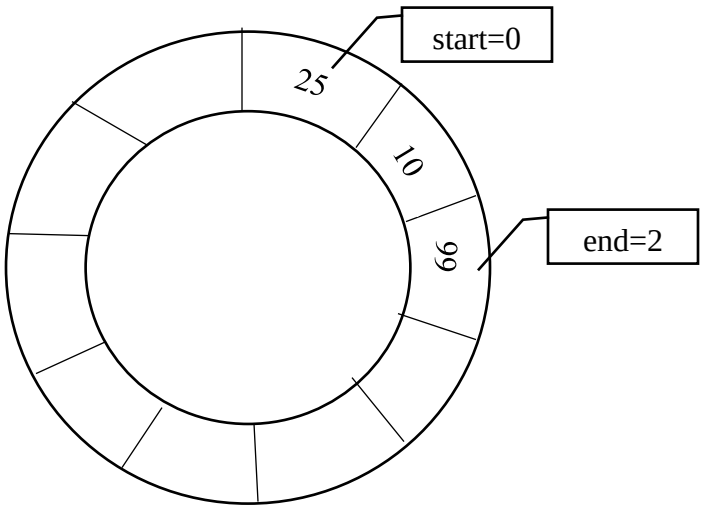
Covid-19 testing has been a pressing issue in the nation for the past few months. We have witnessed long lines of people waiting to be tested. In this project we will develop a simple array-based data structure that can be used to simulate these waiting lines. The data structure will be generic in such a way that it can represent any waiting lines. This Covid testing waiting line has special properties. The data structure has one entry point (called **end**) and one exit point (called **start**). A person who needs testing can only enter the waiting line via the entry (end) and can only be get the testing service when s/he reaches the exit point (start). Individuals can get the testing service based on the order of their arrival; the first one to arrive will be the first one to get tested (i.e., exit the waiting line through the exit point (start)). The waiting line has a circular shape with a mark on the **end** and **start**. Consider the waiting line as an array and initially (empty waiting line) has the last index as end and index 0 as a start. Take a look at the following diagram for an empty waiting line with a capacity of 10.



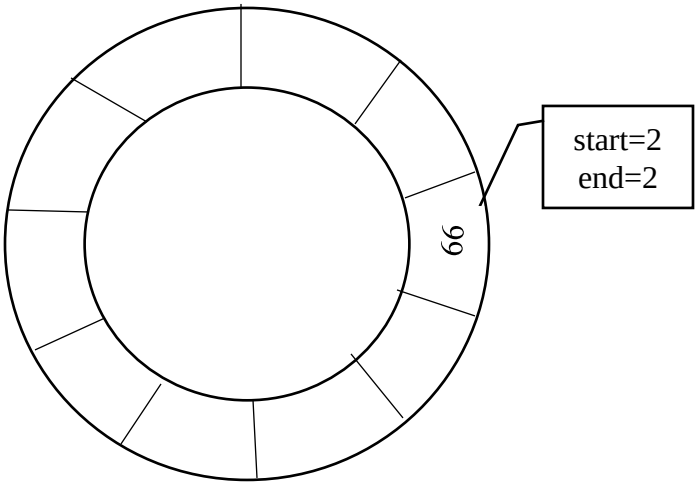
When the first individual arrives, s/he joins the waiting line at the end and **end** will be incremented and point to that new arrival spot. Let see how new arrivals are added and how tested individuals leave the waiting line using integers. Take a look at the following diagram when the first element is added. Note the increment of **end**.



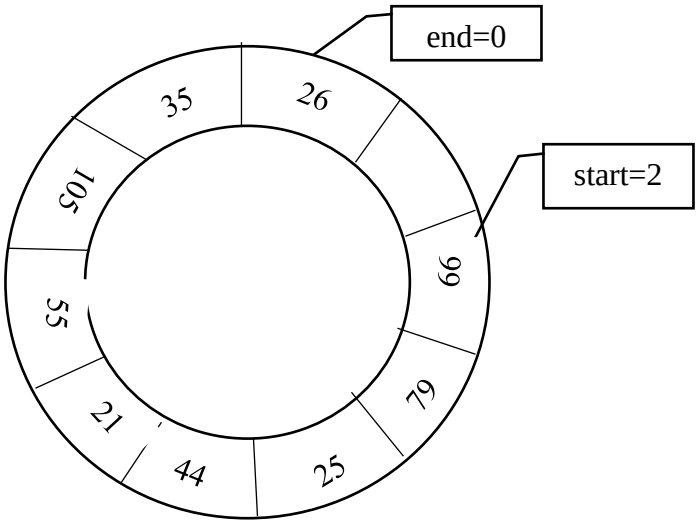
After two more additions:



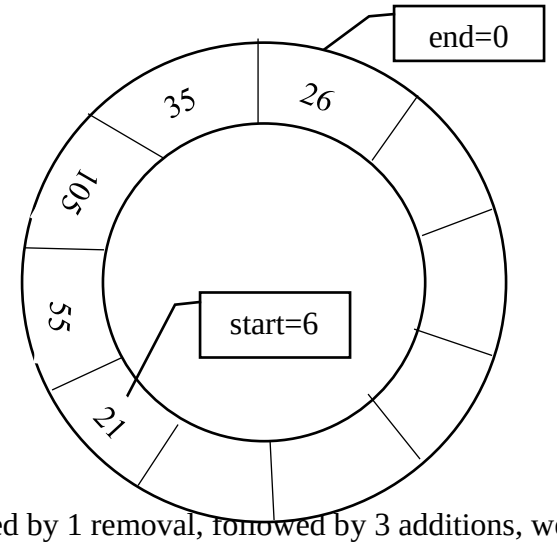
When it is time to be tested, the tester will start from the start and move towards the end, testing one individual at a time. The individual who got tested will leave the waiting line. The following diagram shows, after two individuals are tested (using the integer example).



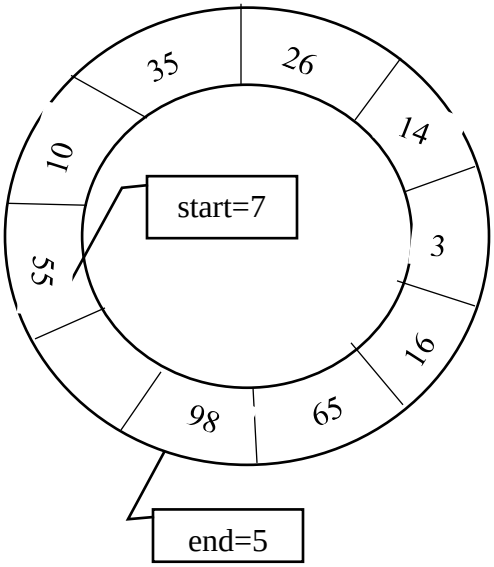
After 8 new individuals join the testing waiting line:



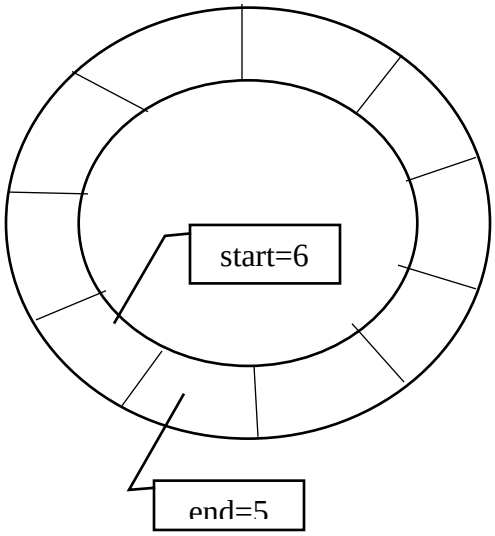
In order to enforce social distancing, we will leave one spot unoccupied. That makes the above waiting line full. In order for a new individual to join the line, somebody should get tested and leave the line or we have to resize the waiting line. The following is after 4 removals.



After 2 additions followed by 1 removal, followed by 3 additions, we will have a full waiting line again.



When we remove 9 elements we will have:



In order to accommodate a surge in testing need, the waiting line can extend its capacity by doubling the number of slots. For example, in this specific example it can extend itself to accommodate up to 19 individuals (double the size and one empty slot requirement).

Implementation/Classes

This project will be built using a number of classes/interfaces representing the component pieces of the waiting line described in the previous section. Here we provide a description of these classes.

INTERFACE (CircularLineInterface): **`CircularLineInterface.java`**

This interface represents all the functionalities the waiting line supposed to have. Since we want the waiting line to be used to simulate any kind of waiting line (not only Covid-Testing lines), all the operations are generic. The return type and/or parameter types of some of the following methods are purposely omitted.

Operations

+insert(newData): void - This method adds a new data to the waiting line. If the waiting line is full, it will double the size to accommodate the new data. Note that the data type of the newData is not specified.

+remove(): - this method removes and returns the first data in the waiting line. For our specific example, it removes the individual who is in the line the longest. If the waiting line is empty, it throws a NoElementException.

+removeAll(): void - removes all elements in the waiting line. It throws NoElementException, if the waiting line is empty.

+getFront() - returns the first element currently waiting in the waiting line (without removing the element) . If nothing is in the waiting line it will throw NoElementException.

+getBack() - returns the last element currently waiting in the line(without removing the element). If nothing is in the waiting line it will throw NoElementException.

+getCapacity():int – returns the maximum storage capacity. For example, it returns the maximum number of individuals that the waiting line can accommodate.

+size():int - returns the number of data/elements at the time. For example, it returns the number of individuals that are waiting in the waiting line currently.

+isEmpty():boolean – checks if the waiting line empty.

+isFull():boolean – checks if the waiting line is full.

Note that this interface should be generic, i.e., it should be able to accommodate any type of object. The Covid testing is just an example.

CLASSNAME (CircularLine): CircularLine.java

This class implements the CircularLineInterface using the waiting line that behaves as described in the previous section. The waiting line uses an array as an implementation data structure. Note that the underline structure uses **start** and **end** to keep track of the first and last elements in the waiting line. The class has the following additional behaviors:

Constructors

+CircularLine(): – creates an empty circularLine with an default capacity of 50.

+CircularLine(capacity:int): - creates an empty CircularLine with a capacity of capacity.

+doubleCapacity():void – This method doubles the size of the CircularLine, if it is full. The already existing elements should not be lost during the resizing.

+getStart():int – returns the position of **start.**

+getEnd():int – returns the position of **end.**

+toString():String – returns the string represntatioon of the data in the waiting line separarted by comma. For example, if the data in the waiting line are Brown, Minilik, Adams, Lawson, the method returns [**Brown,Minilik,Adams,Lawson].** If the waiting line is empty, it will return **[].**

The big-Oh of your methods' implementations should be as follows:

Method Name	Big-Oh	Remark
insert()	O(n)	O(1) amortized complexity
remove()	O(1)	
removeAll	O(n)	
getFront()	O(1)	
getBack()	O(1)	
getCapacity()	O(1)	
size()	O(1)	
isEmpty()	O(1)	
isFull()	O(1)	
doubleCapacity()	O(n)	
toString()	O(n)	
getStart()	O(1)	
getEnd()	O(1)	

CLASSNAME (PriorityCircularLine): PriorityCircularLine.java

This class implements the CircularLineInterface. The class uses some priority for its operations. For instance, for the Covid testing, certain individuals (such as senior citizens) will have priorities to get tested, irrespective of their arrival time. In general, the waiting line will serve/process a high priority element before it serves a low priority element. Note that to compare two objects of a class in Java, the class need to implement a specific built-in Java interface.

Constructors

+PriorityCircularLine(): – creates an empty circularLine with an default capacity of 50.

+PriorityCircularLine(capacity:int): – creates an empty CircularLine with a capacity of capacity.

+doubleCapacity():void – This method doubles the size of the CircularLine, if it is full. The already existing elements should not be lost during the resizing.

+getStart():int – returns the position of **start.**

+getEnd():int – returns the position of **end.**

+toString():String – returns the string represntatioon of the data in the waiting line separarted by comma. For example, if the data in the waiting line are Brown, Minilik, Adams, Lawson, the method returns [**Adams,Brown,Lawson,Minilik] (Note the ordering).** If the waiting line is empty, it will return **[].**

The big-Oh of your method implementation should be as follows:

Method Name	Big-Oh	Remark
insert()	O(n)	
remove()	O(1)	
removeAll	O(n)	
getFront()	O(1)	
getBack()	O(1)	
getCapacity()	O(1)	
size()	O(1)	
isEmpty()	O(1)	
isFull()	O(1)	
doubleCapacity()	O(n)	
toString()	O(n)	
getStart()	O(1)	
getEnd()	O(1)	

CLASSNAME (NoElementException): NoElementException.java

Some of the operations in the classes, throw exception. For example, we cannot test an empty waiting line (remove a person from an empty waiting line). NoElementException handles this error by returning "No element to process" string. The class extends Java's RuntimeException class.

Requirements

An overview of the requirements are listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes and interface
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods. Check provided classes for example JavaDoc comments.
- **Big-O** – The method in each class should be implements according the REQUIRED Big-O runtime shown in the tables above. Your implementation of those methods should NOT have a higher Big-O.

Testing

The main methods provided in the template files contain useful code to test your project as you work. You can use command like "**`java PriorityCircularMainTester/ java CircularLineMainTester`**" to run the testing defined in **`main()`**. You could also edit **`main()`** to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.