Lab 6. Queues

Theme. In this lab, you will:

–use a circular array to implement a queue collection

–use a queue to simulate a customer waiting queue

Key concepts: queue operations, circular queues, typical application of queues

Required file(s): lab6.zip

---

1 Getting started...

1. In a web browser, download the archive lab6.zip from Blackboard and extract its contents into your default eclipse workspace for this module.

2. Start up eclipse.

3. Making use of the contents of your extracted archive, create a new Java project named **post-office-counters** in your eclipse workspace using the contents of your extracted archive.

2 Implementing Circular Queue

So far, our implementations of collection classes use an instance variable count to keep track of the actual number of elements within a collection object (e.g. a queue, a stack, a linked list, a set). The use of count improve the efficiency of some operations on a collection object whose internal structure is backed by a linked list of LinearNode objects. Arguably, the use of count is not mandatory. However, depending on your algorithm, using the variable count may improve efficiency.

A *circular queue* is an implementation of a queue using an array that conceptually *loops around on itself*. The cells in an array is ordered linearly which begins with index 0 and ends at index $n-1$ (where n is the size of the array). However, we can make our program to "pretend" that an array is a *loop* in which the cell following index $n-1$ is index 0.
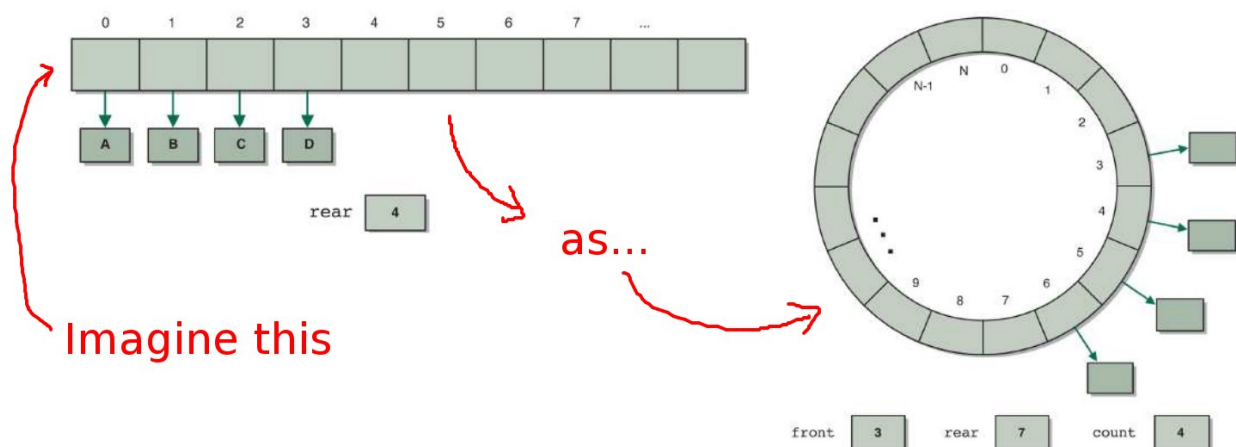

Figure 1: A circular array queue is an array implementation of queue which "loops around".

This lab. session is designed to gain practical experience on working with circular queues.

The archive `lab6.zip` contains two packages:

- `jss2` includes a partial implementation for the class `CircularArrayQueue` and its implementing interface `QueueADT`;

- `ticketCounter` has two classes `TicketCounter` and `Customer` that are designed for modelling a ticket counter simulation (i.e. a typical application for using queues).

Apart from the classes `CircularArrayQueue` and `TicketCounter`, all of the given classes have been fully implemented. Your task is to complete the given partial implementation for these two classes.

> **Hint**: The rough locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for **block comments** that include a sequence of *four* exclamation marks, i.e.:
>
> $$/* \; !!!! \; ... \qquad ... \; */$$
>
> Pay particular attention on the given Java code with **block comments** that include a sequence of *four* plus signs, i.e.:
> $$/* \; ++++ \; ... \qquad ... \; */$$

1. In the class `CircularArrayQueue`, complete the implementation for the method `first`.

> **Hint**: Only one Java statement is needed to complete this task.

2. Complete the implementation for the method `isEmpty` in `CircularArrayQueue`.

> **Hint**: Only one Java statement is needed to complete this task.

3. Complete the implementation for the method `size` in `CircularArrayQueue` that will enable this method to be executed in *constant time* (i.e. O(1)). This method should return the number of elements in the circular queue.

> **Hint**: To work out how to "calculate" the size of a circular array queue, you need to note the various states which the underlying array can be in. See Figure 2 for details.
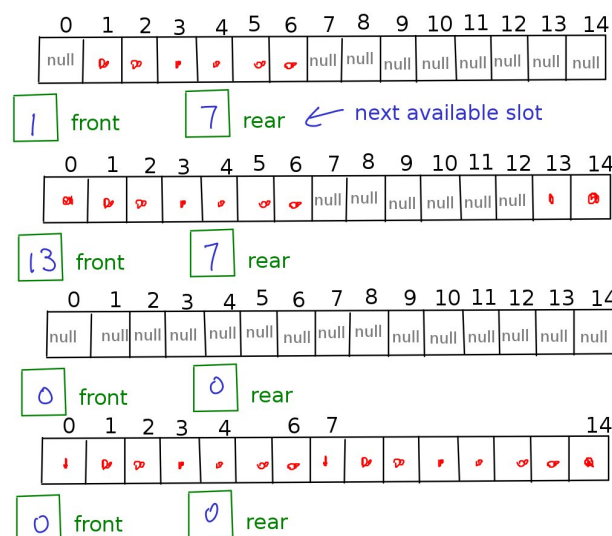

Figure 2: Various states that a circular array queue can be in during the runtime of a sytem.

4. Complete the implementation for the method `toString` in `CircularArrayQueue`. This method should return a `String` representation of all elements in the queue.

5. Complete the implementation for the method `dequeue` in `CircularArrayQueue`. This method should remove the first element from the circular queue and return it as a result.

6. In class `TicketCounter`, create a suitable object whose object reference is to be kept in the `QueueADT` variable `customerQueue`.

7. In class `TicketCounter`, write out an iteration for pre-loading the customer queue with appropriate customer arrival information.

3   Testing

Testing your circular queue implementation

To test your `CircularArrayQueue` implementation, you may use the given `main` method. A successful implementation of this class should produce the following output:

```
Enqueued: a
Enqueued: b
Enqueued: c
Enqueued: d
Enqueued: e
Enqueued: f
Enqueued: g
Enqueued: h
Enqueued: i
Size: 9
Dequeued: a
Dequeued: b
The queue is empty: false
Enqueued: A
Size: 8
Enqueued: B
Size: 9
Enqueued: C
Size: 10
Enqueued: D
Size: 11
All elements in the queue: c
d
e
f
g
h
i
A
B
C
D

Draining the queue...
The queue is empty: true
```

Testing the post office counter simulation

To test the complete implementation, use the given `main(String[])` method in the class `TicketCounter`. If your implementation is correct, the output of the program should be:

```
Number of cashiers: 1
Average time: 5317

Number of cashiers: 2
Average time: 2325

Number of cashiers: 3
Average time: 1332

Number of cashiers: 4
Average time: 840

Number of cashiers: 5
Average time: 547

Number of cashiers: 6
Average time: 355

Number of cashiers: 7
Average time: 219

Number of cashiers: 8
Average time: 120

Number of cashiers: 9
Average time: 120

Number of cashiers: 10
Average time: 120
```

4   Results Analysis

Try plotting the results of the simulation on a spreadsheet.

1. What is the highest number of cashiers the ticket counter should employ in order to maintain a optimal performance?

2. How many cashiers does the ticket counter need to employ in order to ensure that each customer is served within 5 mins?

5 The Ultimate Challenge: Questions to think about...

1. Without the use of the instance variable `count`, we need to explicitly count the elements kept in the array every time when we need to reveal the size of the queue. One implementation of the methods `expandCapacity` and `toString` requires the knowledge of the size of the queue to complete its operation. There are two ways to supply this information to these methods:

   (a) In each of these methods, use a local variable `size` to keep the size of the queue *before* using an iteration to go through the elements in the queue. The continuity of the iteration is dependent on the value of `size`.

   (b) When determining if the iteration should continue, call the method `size()` to find out the current size of the queue.

   Both approaches have pros and cons. Can you describe the circumstance when each of these approaches would be more suitable than the other?

2. With our current implementation, is it possible for the `toString` method to go through the circular array using a *for-each* loop (aka *enhanced for* loop)? Explain why.