# ITI 1121. Introduction to Computing II

Implementing a queue using a **circular array**

by

Marcel Turcotte

Version March 8, 2020

# Preamble

# Preamble

## Overview

# Overview

**Implementing a queue using a circular array**

We consider here the implementation of a queue using an array. As with stacks, we expect the size of the array to grow dynamically, depending on the needs of the application. However, we will discover that the implementation is not as simple as it seems.

**General objective :**

- This week, you will be able to implement a queue using a circular array.

# Preamble

## Learning objectives

# Learning objectives

- **Implement** a queue using a circular array.
- **Compare** implementations using arrays and linked elements of a queue.

**Readings:**

- Pages 189-194 of E. Koffman and P. Wolfgang.

# Preamble

## Plan

# Plan

# Reminder

# Definitions

A **queue** is a linear **abstract data type** such that adding information is done at one end, the **rear** of the queue, and removing it at the other, the **front**.
These data structures are referred to as first-in-first-out (FIFO) structures.

$$\texttt{enqueue()} \Rightarrow \text{Queue} \Rightarrow \texttt{dequeue()}$$

The two basic operations are:

**enqueue:** **adding** an item to the **rear** of the queue,

**dequeue:** the **removal** of an item from the **front** of the queue.

$\Rightarrow$ The queues are therefore data structures similar to the queues at the supermarket, bank, cinema, etc.

# Interface Queue

```
public interface Queue<E> {
    boolean isEmpty();
    void enqueue(E o);
    E dequeue();
}
```

# Implementation

# Implementation

## Instance variables

# Implementation using an array

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;

    public boolean isEmpty() { ... }
    public void enqueue(E o) { ... }
    public E dequeue() { ... }

}
```

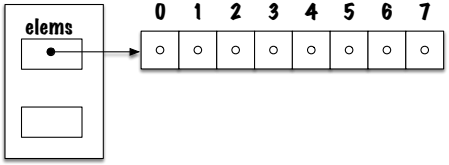
- **Any suggestions** for the additional **instance variable** or **instance variables**?

# Implementation

## Implementation 1

# Implementing a queue using an array

**Implementation 1.** The **front** of the queue is **fixed**, at position 0, for example, and we use a variable that points to the **rear** of the queue.




$\Rightarrow$ Unlike implementing stacks, implementing queues using arrays will cause **some problems**.

# Implementing a queue using an array

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;
    private int rear;

    public boolean isEmpty() { ... }
    public void enqueue(E o) { ... }
    public E dequeue() { ... }

}
```

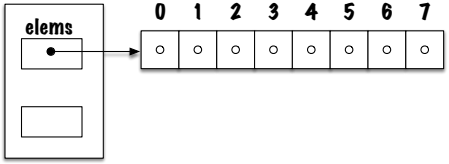
# Remarks

- Just like we did with the stacks, we could:
  - Make the value of **rear** point to the **first free cell or the cell where the rear element is;**
  - **When removing an element, put the value null in the array cell to avoid memory leaks.**
  - **Use the dynamic array technique.**

$\Rightarrow$ However, the **conclusions** to be drawn regarding the performance of the insertion and removal algorithms would remain the same.

# Insersion (enqueue)

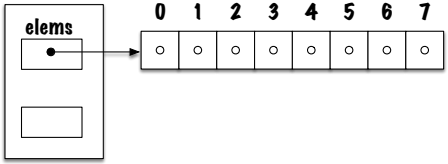
Inserting a new element into a queue, when the front of the queue is fixed and the back moves, is similar to adding an element to a stack.




$\Rightarrow$ **(i)** The variable **rear** is increased by 1, then **(ii)** the value is put at the position **rear** of the array.

# Removal (dequeue)

- What about the removal of an item?



# Removal (dequeue)

After removing the element, one must **move all elements one position to the left** so that the front of the queue remains in a fixed position, 0.

1. **Save** in a temporary variable the value at the front of the queue;
2. From **i** equals 1 to **rear** move the value at position **i** to position **i-1**;
3. **Initialize** the **rear** cell of the array;
4. **Decrement** the variable **rear** by one;
5. **Return** the saved value.

# Removal (dequeue)

- This means that for each removal, one must **move n-1** values, if there were **n** values before the removal.
- **The more items** in the queue, **the more moves** to be made. If we double the number of elements in the queue, removing an element will require twice as many moves.
- This is not the case for insertions, an insertion is always made at the first free position of the array. Inserting an element in a queue containing twice as much data does not require more work (operations).

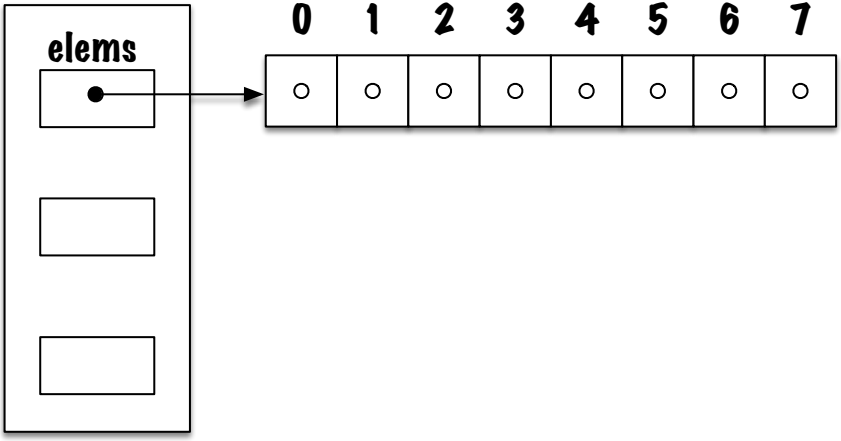
$\Rightarrow$ **Can we do better?**

# Implementation

## Implementation 2

# Implémenter une file à l'aide d'un tableau

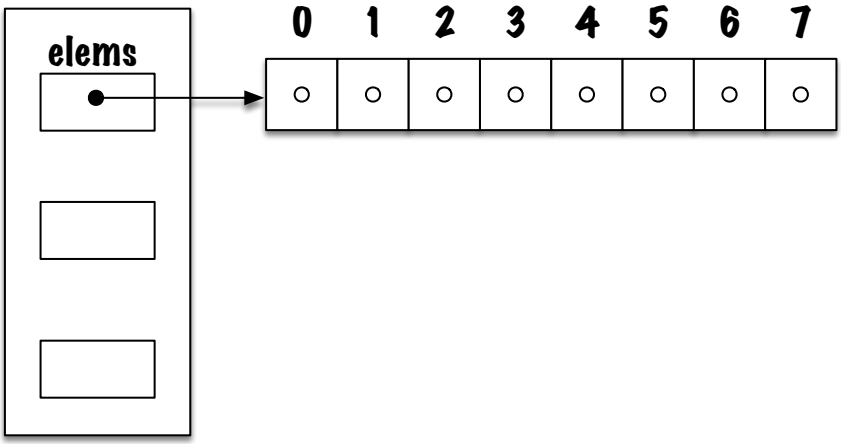
**Implementation 2.** The **front** and the **rear** of the queue are **moving**, so you have to use a variable that points at the **front** element, and another that points to the **rear**.



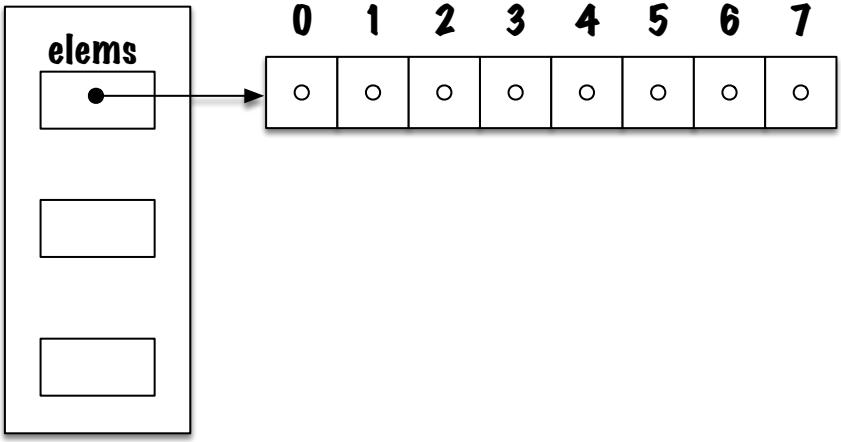
# Implementation using an array

```
public class ArrayQueue<E> implements Queue<E> {

    private E[] elems;
    private int front;
    private int rear;

    public boolean isEmpty() { ... }
    public void enqueue(E o) { ... }
    public E dequeue() { ... }

}
```

# enqueue



# dequeue



# Retrait (dequeue)

- In order to **remove** an element, we now have to **(i)** save the value at the front in a temporary variable, **(ii)** increment front and **(iii)** return the saved value.
- The **removal** of an element is now always done in constant time, i.e. no matter how many elements are present in the queue, there are only the three operations to do (and possibly initialize the free cell to **null**, if dealing with reference values).
- **Mission accomplished?**

# Insertion (enqueue)

It's just like before:

1. **Increase** the value of the variable **rear**;
2. **Insert** the new value at position **rear**.

# Discussion

- There's a **fundamental problem** with the implementation.
  - **What is it?**

# Discussion

- This new implementation allows us to **remove** an element in a way that is **efficient**, i.e. the time required no longer depends on the number of elements in the queue.
- However, the **price to pay** is that when the value of **rear** reaches the limit of the array (but **front** is not 0, i.e. the queue is not full, it has just moved to the right) then it must be **repositioned to the left** of the array, i.e. all its elements must be moved.

# Discussion

- **The more elements the queue contains** at the time of insertion, **the more elements there are to be shifted**. If there are twice as many elements in the queue, then twice as many elements will need to be moved.

# Discussion

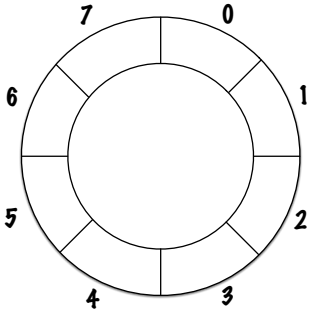
- Under **implementation 1**, adding elements is effective, but removing them is slow.
- Under **implementation 2**, the opposite is true, adding elements is expensive (when the queue has moved to the right), but removing them is efficient.
- **Is it possible to have both an efficient removal and insertion?**

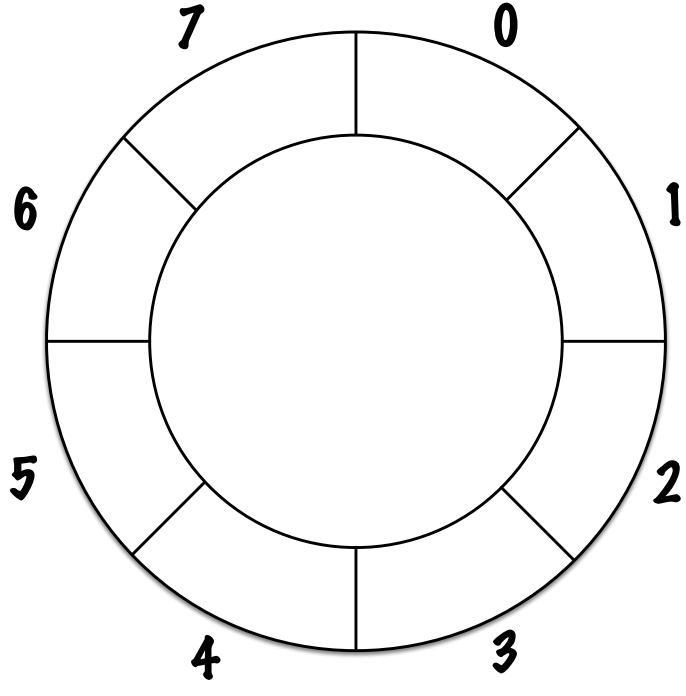
# Implementation

## Implementation 3

# Implementing a queue using an array

**Implementation 3.** In order to make the 2 basic operations, removal and insertion, efficient, we will use a **circular array**. Both the front and the rear of the queue will move, so we must use a variable that points at the front, **front**, and another that points at the rear, **rear**.



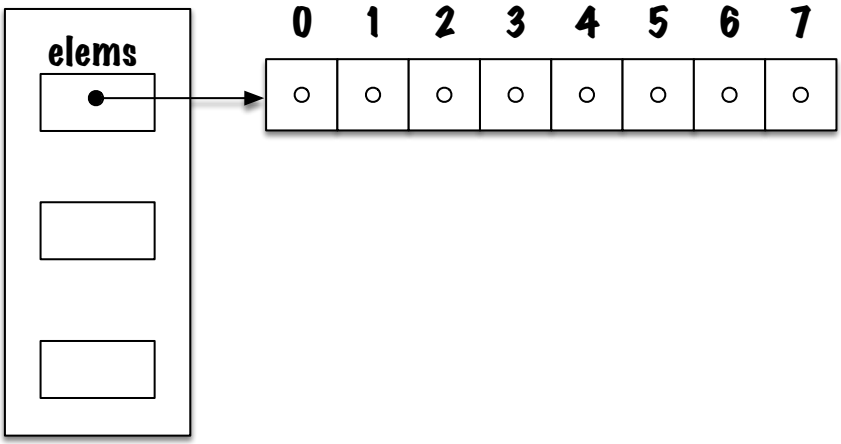

7
0
6
1
5
2
4
3

# Implementing a queue using an array

- So, in order to **insert** a value, one has to: **(i)** increase the value **rear**, **(ii)** then insert the new value at the position **rear**.
- Similarly, in order to **remove** an element one must: save the value found at the position **front**, re-initialize this position of the array, increase the value of **front** and return the saved value.

# Circular array

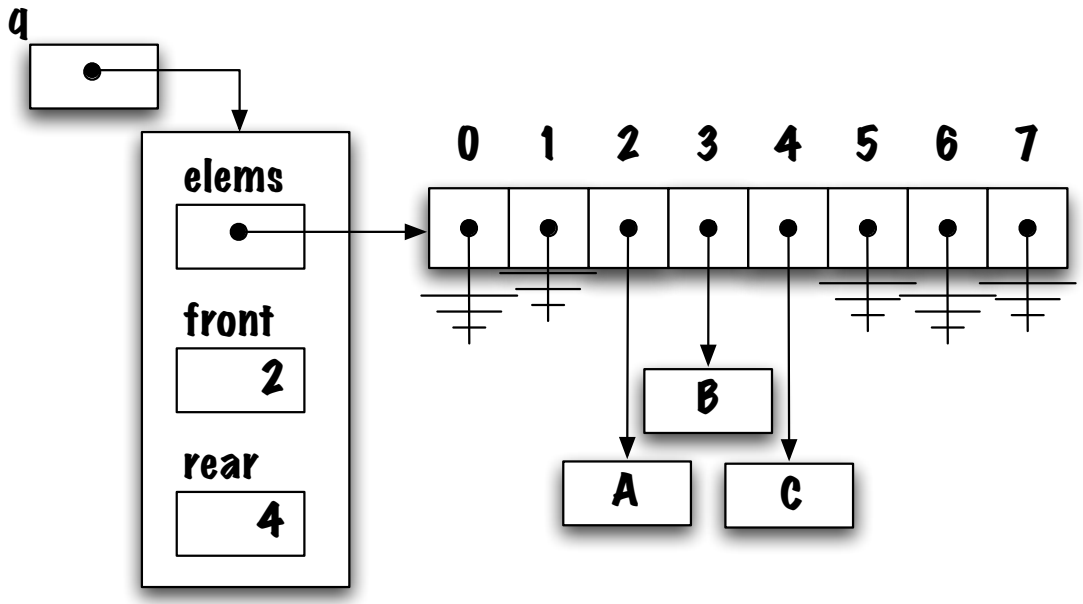
- **How** do we implement a circular array?



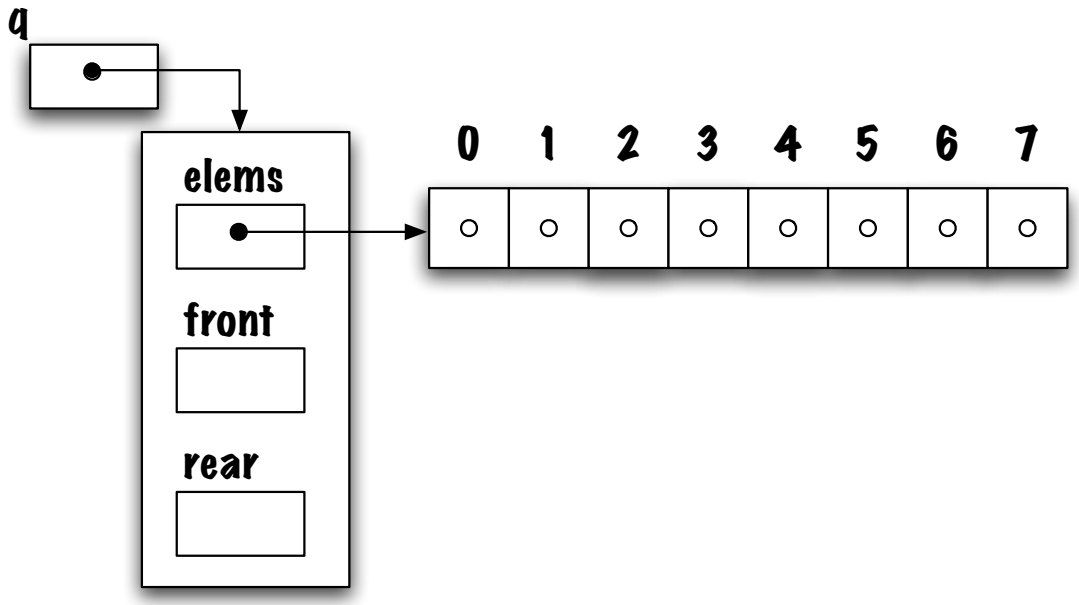
# Circular array

- The idea is the following, when the rear of the queue has **reached the right end of the array**, and there are free cells in its left part, then we start again **inserting elements at the beginning of the array**.

# Memory diagram



# Memory diagram



# Circular array

- But then again, how do you write the method **enqueue**, for instance?
- We could add a test like this:

```
rear = rear + 1;
if (rear == MAX_QUEUE_SIZE) {
    rear = 0;
}
```

# Circular array

**Alternative?**

Instead, we use **modulo arithmetic** which simplifies the writing.

```
rear = (rear + 1) % MAX_QUEUE_SIZE;
```

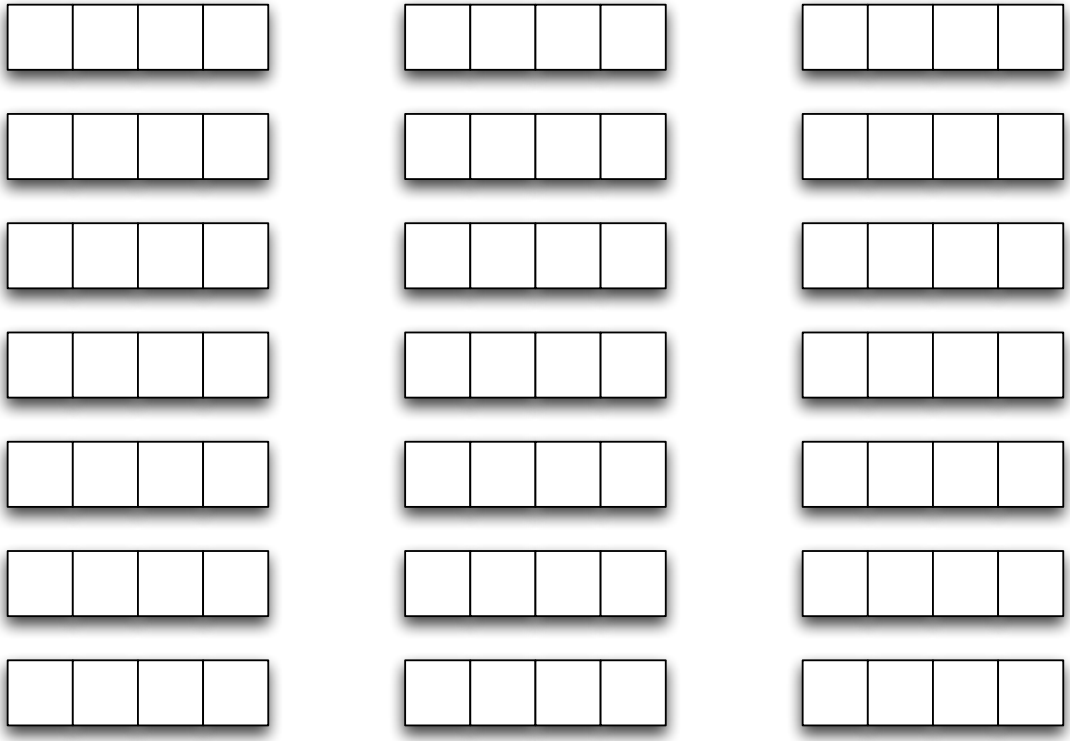
# Insertion (enqueue)

1. **rear = (rear+1) % MAX_QUEUE_SIZE**;
2. **Add** the new element at the position **rear**.

# Removal (dequeue)

1. **Save** the front value of the queue;
2. **Save** the value **null** at the `front` position of the array;
3. **front = (front+1) % MAX_QUEUE_SIZE**;
4. **Return** the saved value.

⇒ **What happens when the queue is empty?**



# Discussion

- So we can see that it's impossible to tell an **empty queue** from a **full queue** based on the variables **rear** and **front**.
- **What can we do about it?**

# How do you tell the empty queue from the full queue?

- There are several possible alternatives: **destroy the array when the queue is empty**, **use a boolean** or **sentinel** (-1) as a value for **rear** or **front**.
- Another solution is to use an instance variable in order to **count the number of elements** in the queue and to choose wisely the values of **rear** and **front**: for example 0 and 1 or **MAX_QUEUE_SIZE** and 0.

# CircularQueue

- We'll use a **sentinel value** for the rear index to signify an empty queue.

```
public class CircularQueue<E> implements Queue<E> {

    private static final int MAX_QUEUE_SIZE = 100;

    private E[] elems;
    private int front, rear;

    public CircularQueue() {
        elems = (E []) new Object[ MAX_QUEUE_SIZE ];
        front = 0; //
        rear = -1; // indicates that the queue is empty
    }

    // ...
```

# isEmpty

```
public boolean isEmpty() {
    return (rear == -1) ;
}
```

# isFull

```
public boolean isFull() {


}
```

# enqueue

- Even works for the empty queue, **why**?

```
public void enqueue(E value) {
    rear = (rear+1) % MAX_QUEUE_SIZE;
    elems[rear] = value;
}
```

# peek

```
public E peek() {
    return elems[front];
}
```

# dequeue

```
public E dequeue() {




}
```

# Insertion (enqueue)

Another way to solve this problem would have been to use an instance variable to **count elements**.

1. **rear = ( rear+1 ) % MAX_QUEUE_SIZE**,
2. **Add** the new element at the position **rear**;
3. **Increase** the value of **count**.

# Removal (dequeue)

1. **Save** the value at the front of the queue;
2. **Save** the value **null** at the `front` position of the array;
3. **front = (front+1) % MAX_QUEUE_SIZE**;
4. **Decrement** the value of **count**;
5. **Return** the saved value.

# empty()

1. `count == 0`

# isFull()

1. **count == MAX_QUEUE_SIZE**

# Prologue

# Summary

- Implementing a queue using an array requires the use of a **circular array**.

# Next module

- Abstract Data Type (ADT) : **lists**

# References I


E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.


Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

**University of Ottawa**