# ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 10, 2014

**Abstract**

- Queues
  - ArrayQueue

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

# Definitions

A **queue** is a linear abstract data type such that insertions are made at one end, called the **rear**, and removals are made at the other end, called the **front**.

Queues are sometimes called FIFOs: *first-in first-out*.

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

The two basic operations are:

**enqueue:** adds an element to the rear of the queue;

**dequeue:** removes and returns the element at the front of the queue.

$\Rightarrow$ Software queues are similar to physical ones: queuing at the supermarket, at the bank, at cinemas, etc.

# ADT

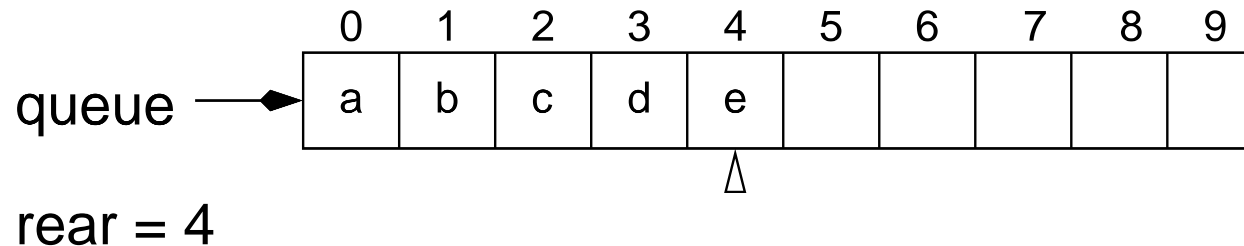
```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E o );
    public abstract E dequeue();
}
```

## Using an array

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

    private E[] elems;

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

}
```

# Array implementation of a queue

**Implementation 1:** Let's set the front of the queue to a fix location, say 0, and use an instance variable to indicate the location of the rear element.




⇒ Contrary to the array-based implementation of a stack, using an array to implement a queue will create some problems, which we will circumvent, of course!

# Array implementation of a queue

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

    private E[] elems;
    private int rear;

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

}
```

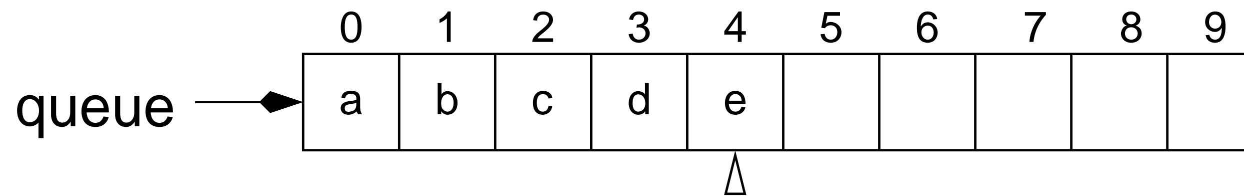
## Remarks

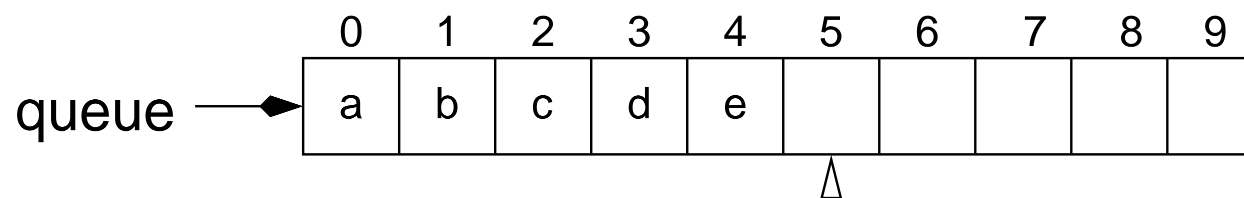
- Just like stacks,
  - *rear* can be made to designate the first free location (instead of the element itself);
  - a queue could be implemented in the upper part of the array.

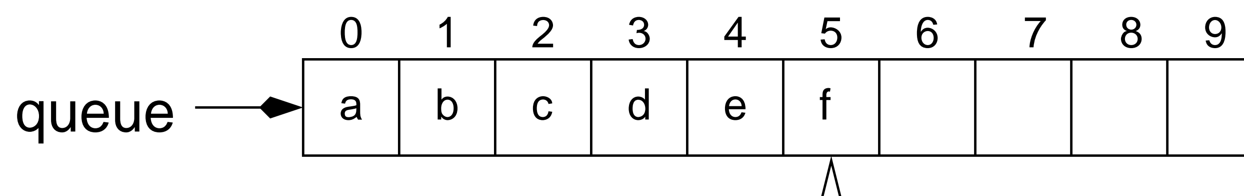
$\Rightarrow$ Without effects on the efficiency.

# Insertion (enqueue)

Inserting an element into a queue, when the front location is fixed, is similar to adding an element onto a stack.




$\Rightarrow$ *(i)* the variable `rear` is incremented by 1, then *(ii)* the new element is stored at the position designated by `rear`.

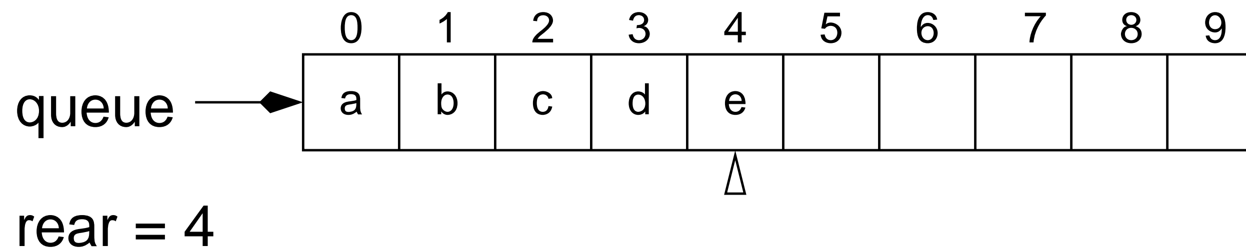
# Removal (dequeue)

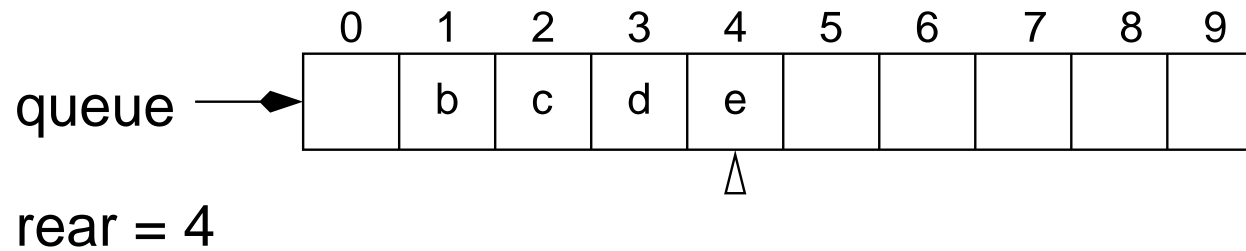
What about removing an element?

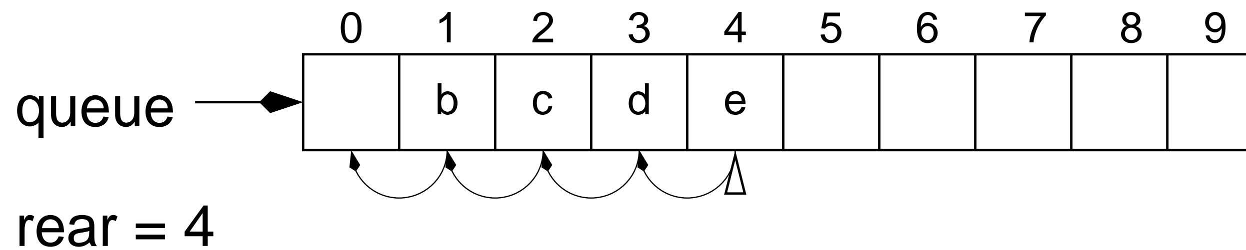
All the elements must be moved one position toward the front of the queue so that the front element remains at a fixed location, here 0.

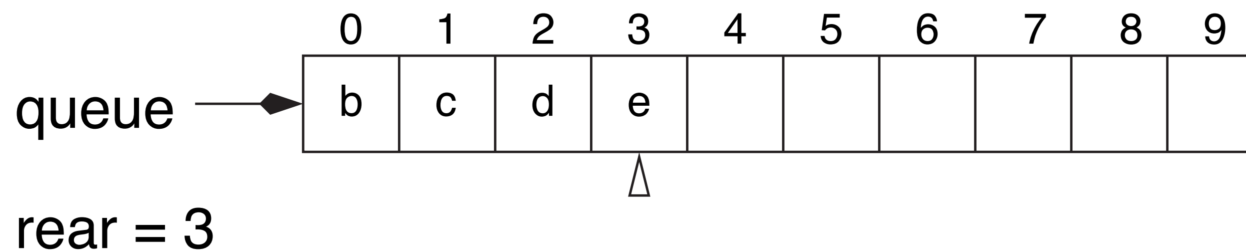
1. Save the value to be returned;
2. For $i$ from 1 to (*rear* - 1) move the element at position $i$ to $i - 1$;
3. Set *rear* cell to *null*;
4. Decrement *rear* by 1;
5. Return the saved value.

# Removal (dequeue)



# Removal (dequeue)

For each element that's removed, $n - 1$ elements must be moved!

The more elements there are in the queue, the more costly the removal operation becomes.

Doubling the number of elements in the queue means doubling the cost of the removal operations.

On the other hand, for a fixed-size array, the cost of inserting an element remains constant.

$\Rightarrow$ What improvements can be made?

# Array-based implementation

**Implementation 2:** the *front* and the *rear* of the queue are now moving, this will require two instance variables, one that points at the front element and one that points at the rear.

## Removal (dequeue)

**General case:**
To remove an element, *(i)* save the value located at the position designated by **front** to a (local) temporary variable, *(ii)* set the cell designated by **front** to *null*, *(iii)* increment **front** and *(iv)* return the saved value.

Removal is now done in constant time, always the same four steps above.

Mission accomplished?

Any particular problem with this implementation?

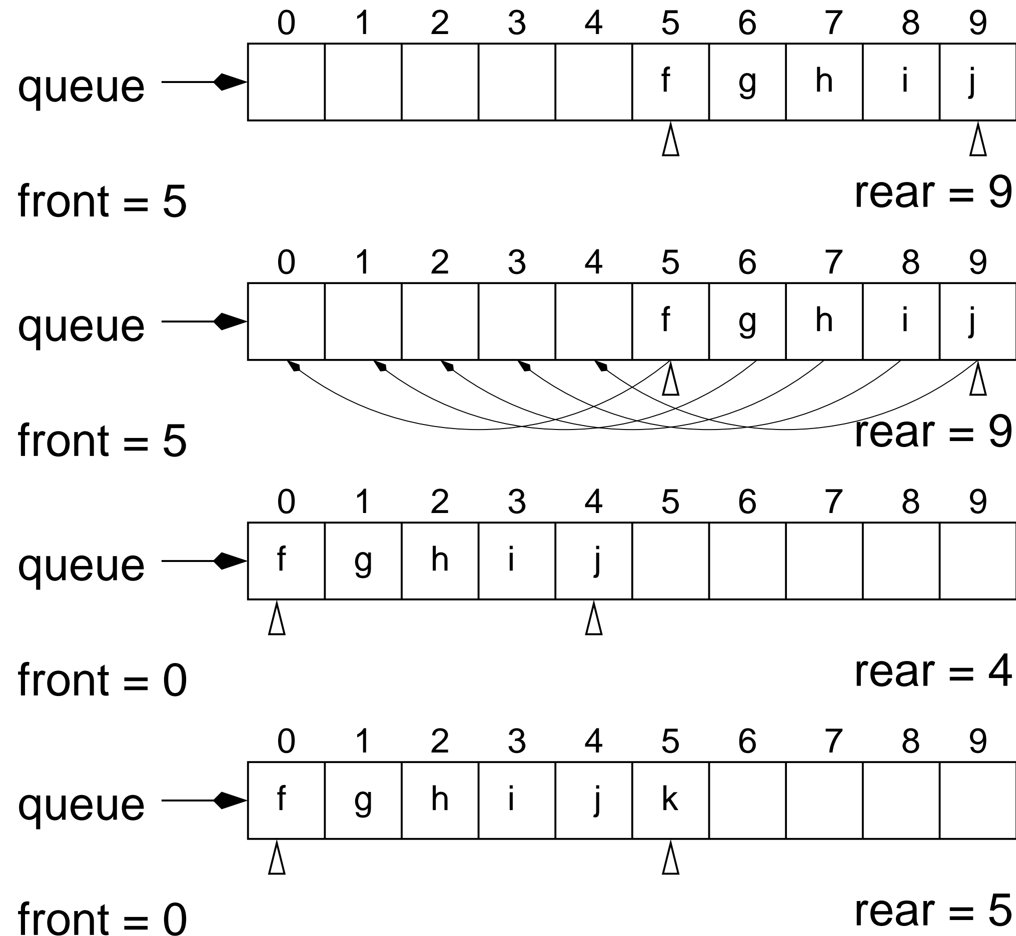
# Inserting (enqueue)

What about insertions?

Just like before:

1. Increment by one the variable **rear**,
2. Insert the new value at the position designated by **rear**.

This implementation has a special case, when the queue has reached the upper part of the array, and there are free cells in its lower part, all the elements must be moved.



This new implementation allows to remove elements efficiently, i.e. does not depend on the size of the queue.

The price to pay is that whenever the queue reaches the limit of the array and **front** is not 0, i.e. the queue is not full, then the queue must be re-positioned to the lower part of the array, i.e. all the elements are moved.

The larger the queue the more costly this operation becomes.

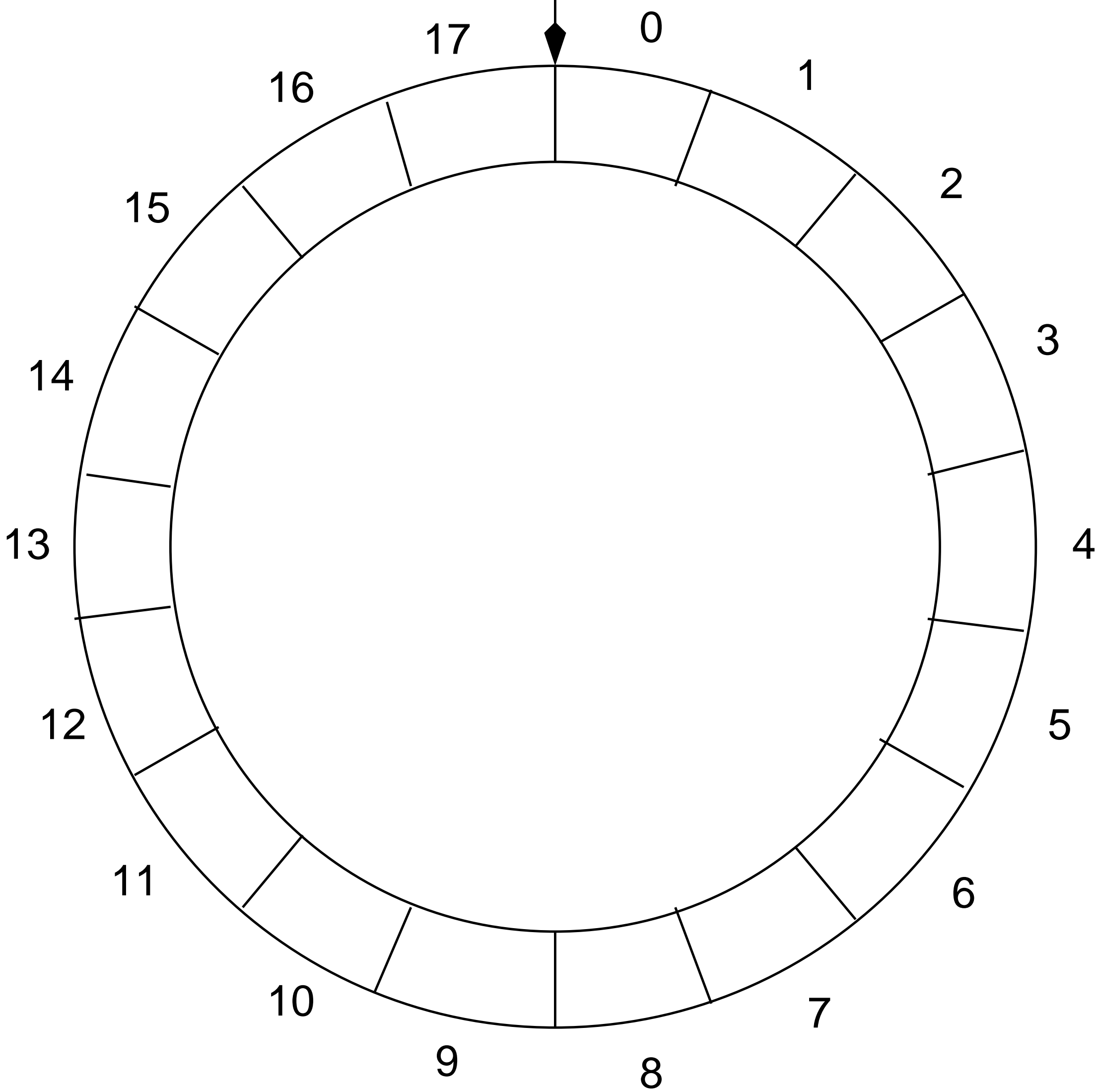
For implementation 1, adding an element is done efficiently but removal is slow.

For implementation 2, it is the contrary, adding an element now becomes the slow operation (because it needs to re-position the queue from time to time) but removal is always fast.

$\Rightarrow$ Can both operations be implemented efficiently?

# Array-based implementation

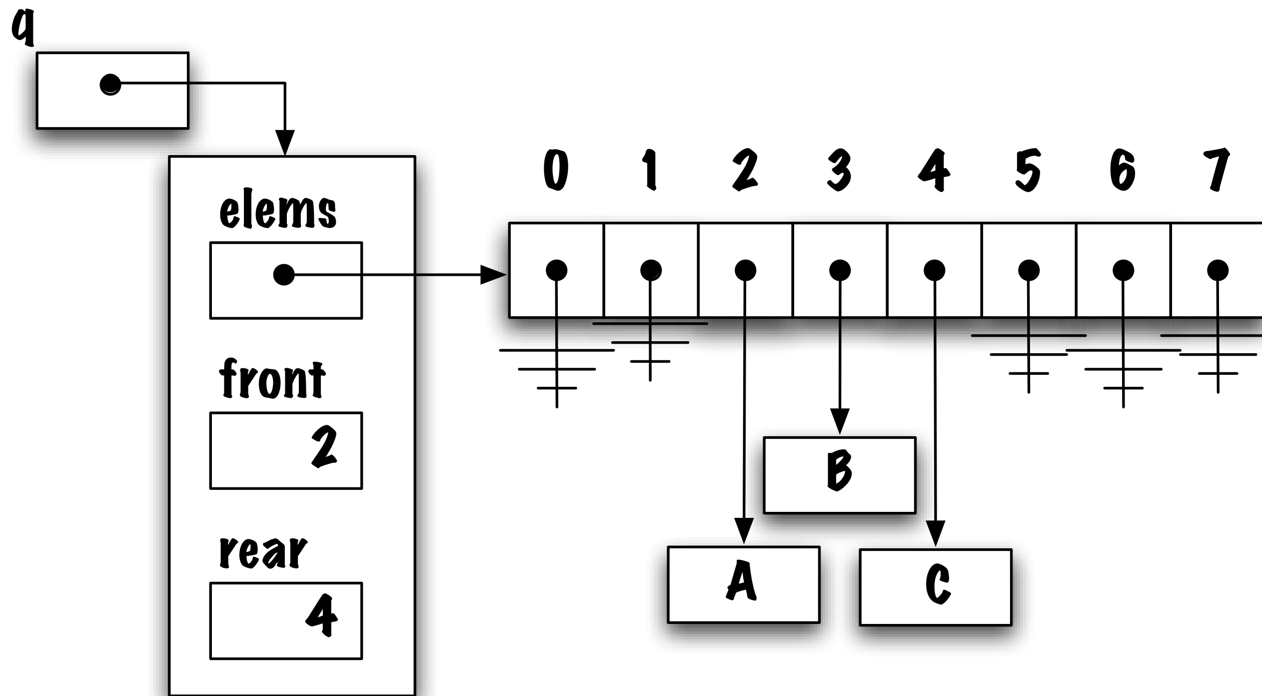
**Implementation 3:** to make both operations fast this implementation uses a **circular array**. Two instance variables will be needed, **front** and **rear**.


front =
queue
rear =
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

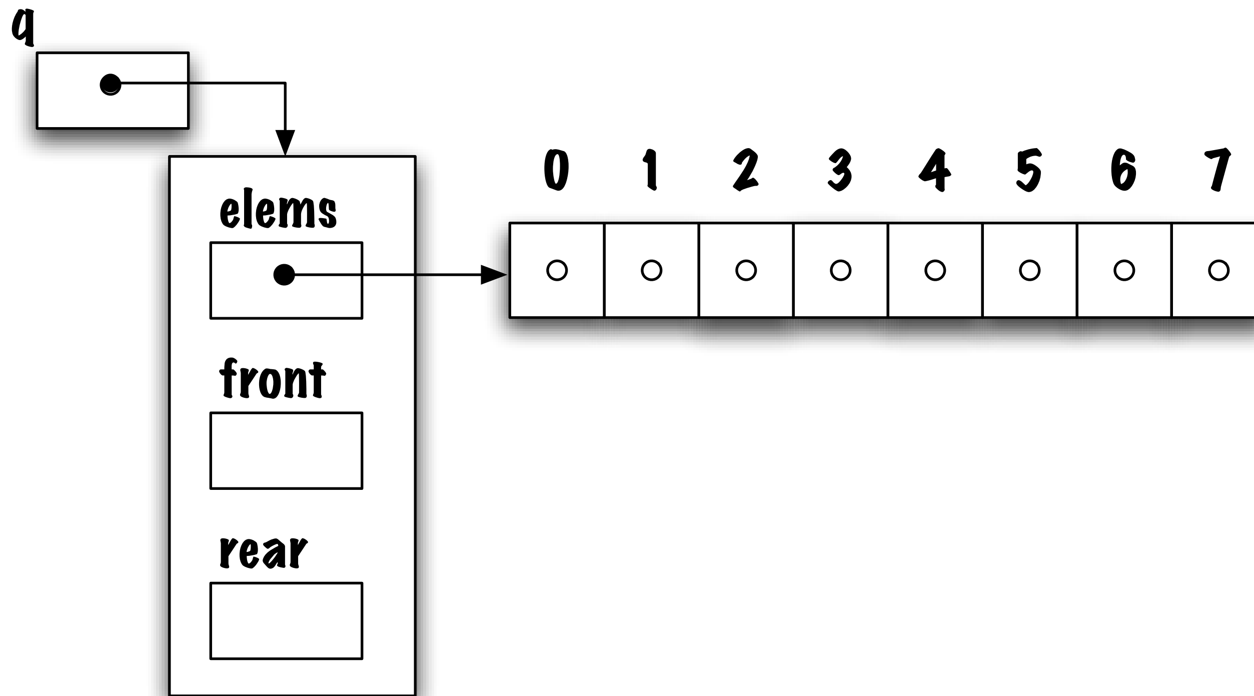
Insertion: *(i)* increment the value of *rear*, *(ii)* insert the element at the position designated by *rear*.

Removal: save the element found at position *front*, set the position to *null*, increment *front* and return the saved value.

# Memory diagram



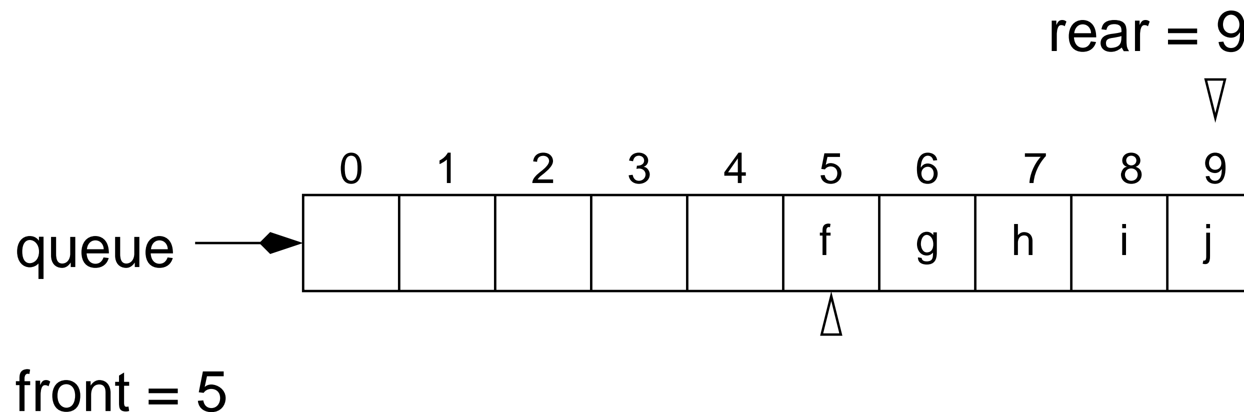
# Memory diagram

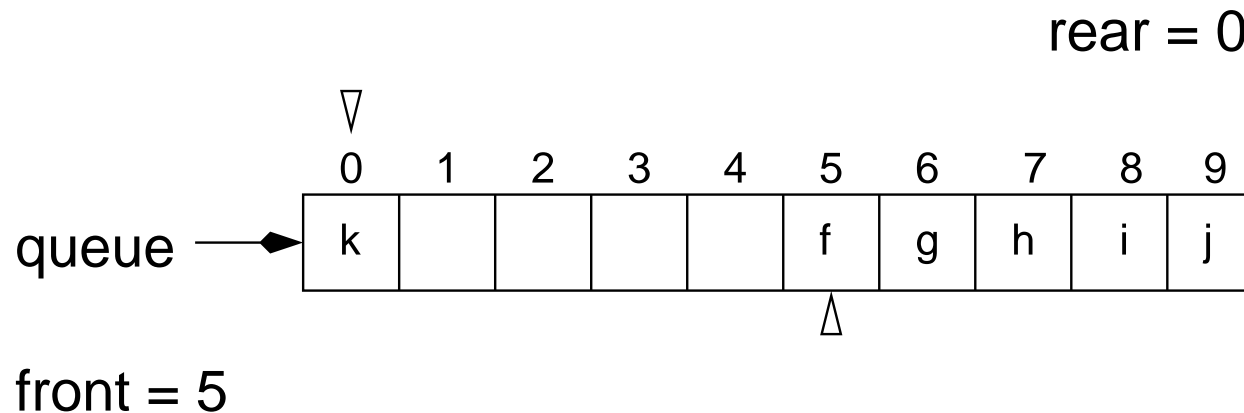


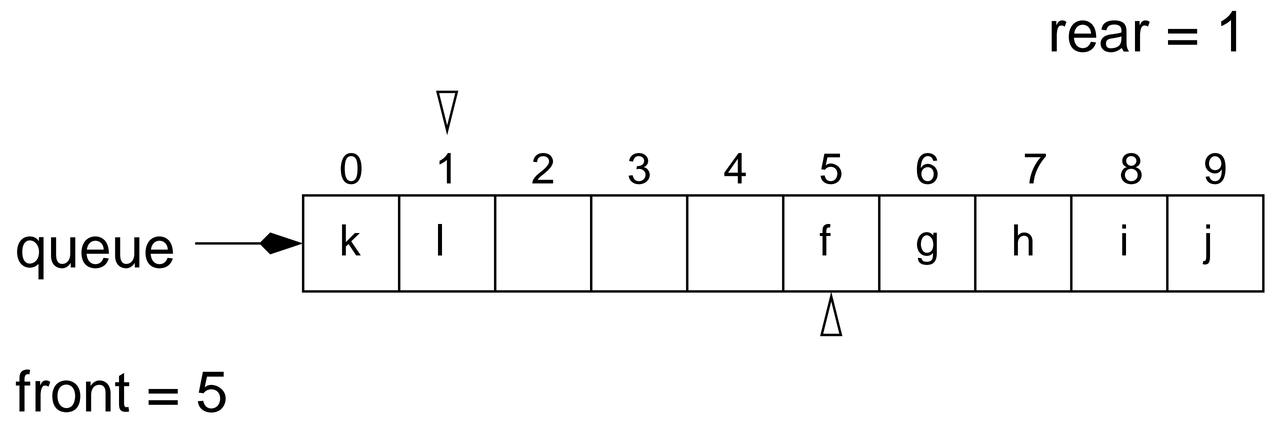
# Circular array

How to implement a circular array?

The idea is as follows, once the queue reaches the end of the array, the free cells at the start of the array are re-used (if they are free, of course).





rear = 1
0
1
2
3
4
5
6
7
8
9
queue
k
l
f
g
h
i
j
front = 5

Enqueue could be implemented as follows:

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

What other technique can be used to implement a wraparound?

We can use modulo arithmetic [1]:

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

[1] Just like increasing the number of seconds in the class Time.

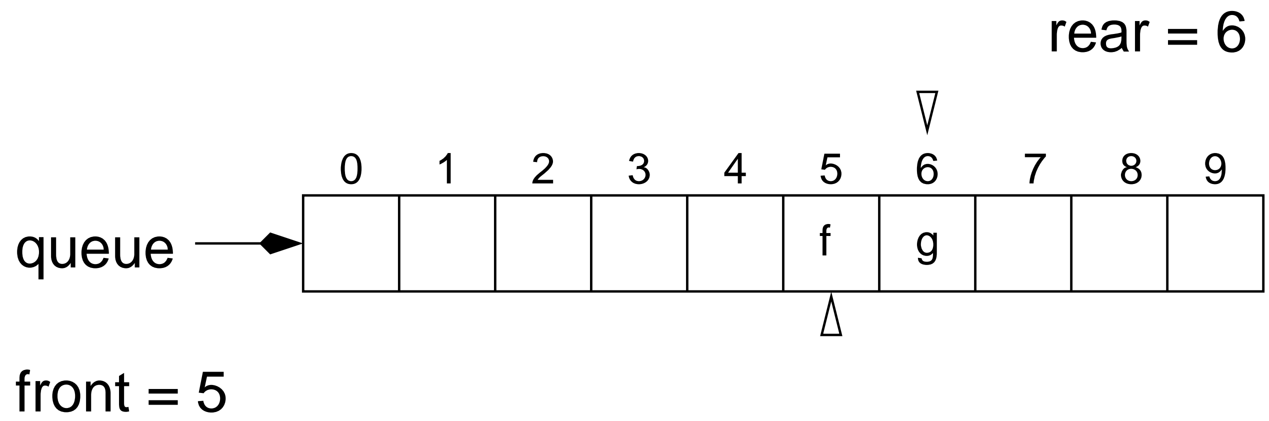
# Insertion (enqueue)

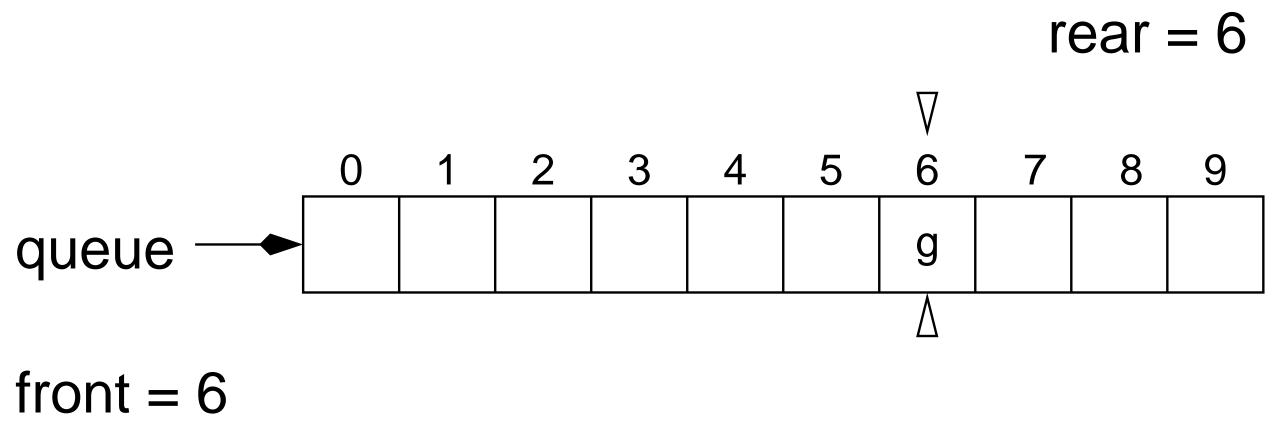
1. `rear = ( rear+1 ) % MAX_QUEUE_SIZE`,
2. add the element at the position designated by `rear`.

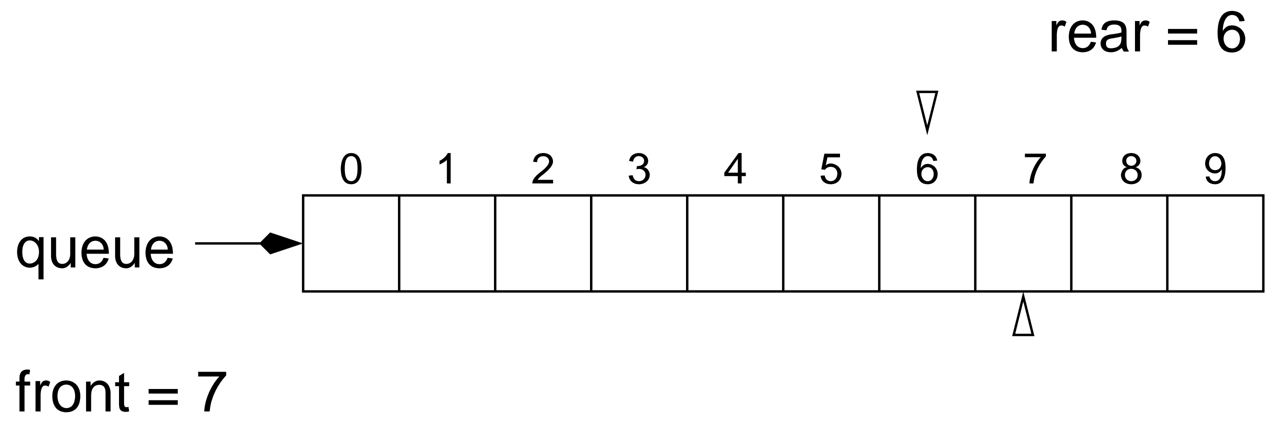
## Removal (dequeue)

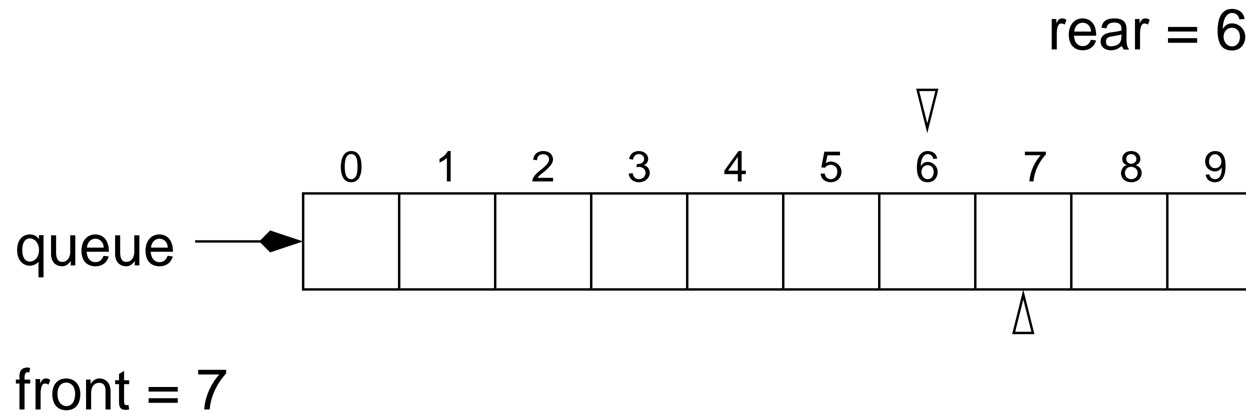
1. save the value found at the position designated by front,
2. set that location to null,
3. `front = ( front+1 ) % MAX_QUEUE_SIZE`,
4. return the saved value.

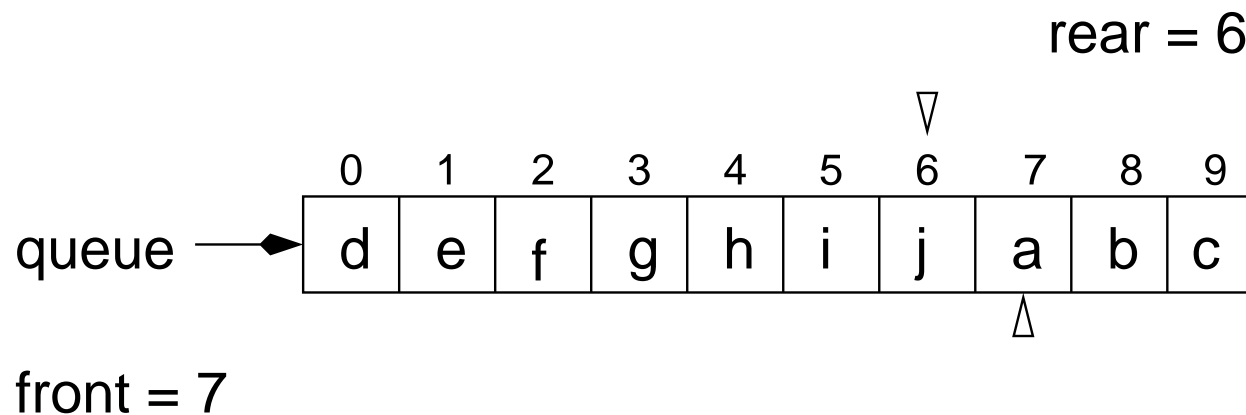
$\Rightarrow$ What occurs when the queue is empty?


rear = 6
0 1 2 3 4 5 6 7 8 9
queue
f g
front = 5


rear = 6
0 1 2 3 4 5 6 7 8 9
queue
g
front = 6


rear = 6
0
1
2
3
4
5
6
7
8
9
queue
front = 7




We can see that it is no longer possible to distinguish the empty queue and the one which is full on the basis of the values of `rear` and `front` alone.

Solutions?

There are many implementation techniques:

- use a boolean instance variable (isEmpty);
- use a sentinel value (-1) for rear and/or front;
- count the number of elements (size)

For our implementation, we will use a sentinel for the value of rear, here -1.

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

    private static final int MAX_QUEUE_SIZE = 100;

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

    public CircularQueue() {
        q = (E[]) new Object[ MAX_QUEUE_SIZE ];
        front = 0;
        rear  = -1; // represents the empty queue
    }

    // ...
```

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

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

public boolean isFull() {


}
```

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


public E dequeue() {




}
```

## Insertion (enqueue)

An alternative implementation would be to use an instance variable to count the number of elements.

1. `rear = ( rear+1 ) % MAX_QUEUE_SIZE`;
2. add the new element at the position designated by `rear`;
3. **increment** `count`.

# Removal (dequeue)

1. save the front value;
2. set that location to null;
3. `front = ( front+1 ) % MAX_QUEUE_SIZE`;
4. **decrement** `count`;
5. return the saved value.

# empty()

1. `count == 0`

# isFull()

1. `count == MAX_QUEUE_SIZE`