# IS2020 COMP 2540: Data Structures and Algorithms

## Lecture 04: Queues

Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

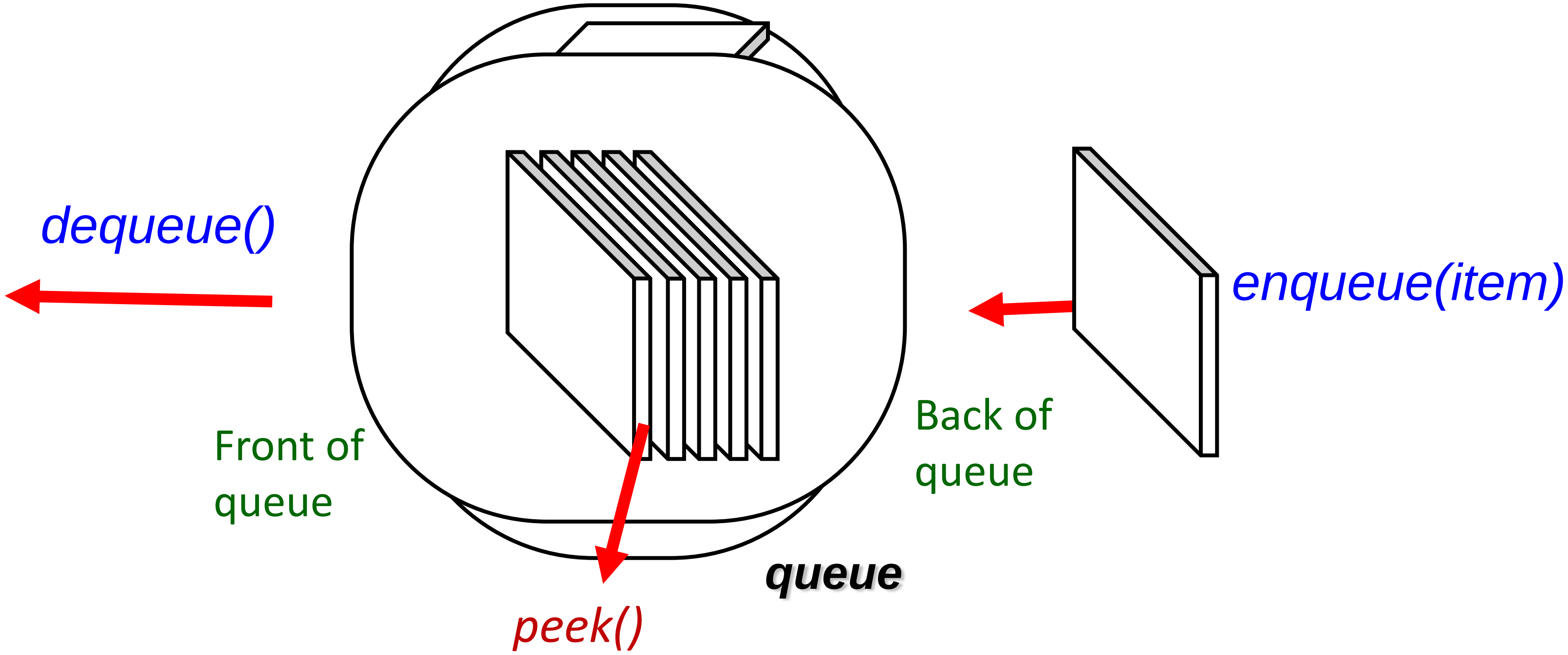
June 8, 2020

# Outline

1. Background
2. Queues Definition and Properties
3. Queue ADT
4. Queue Array based Implementation

# Background

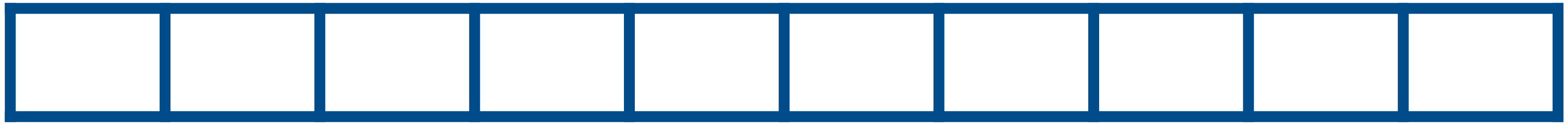
- The **Queue ADT** stores arbitrary objects and is a linear data structure.
- Insertions and deletions follow the **first-in first-out (FIFO) or Last In Last Out (LILO)** scheme.
- Insertions are at the **rear** of the queue and removals are at the **front** of the queue.
- The insertion of an element into the queue called **enqueue** operation and the
- deletion of an element from the queue called a **dequeue** operation.



# How does the Queue work?

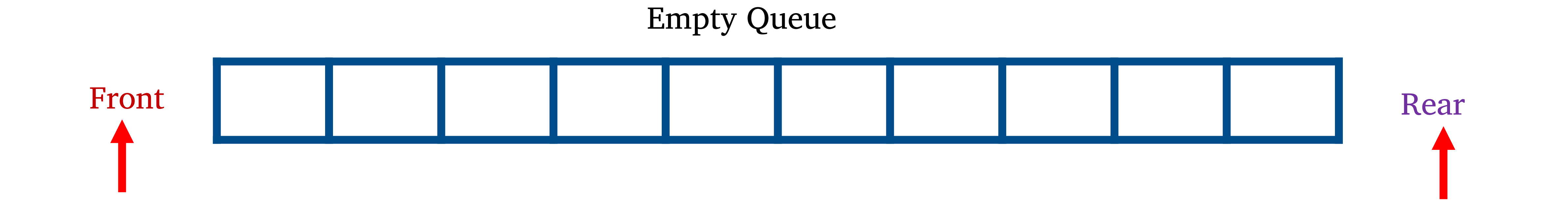
- We need two special pointers Front and Rear. We add elements to the rear of the queue and remove elements from the front of the queue.

Empty Queue



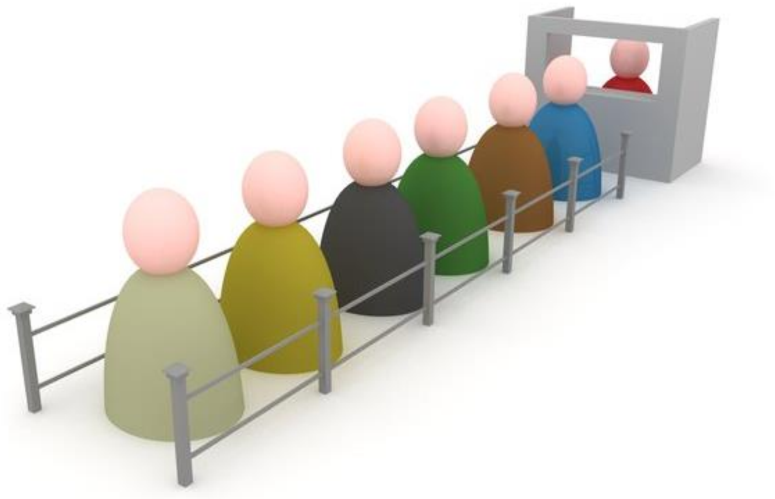
# How does the Queue work?

- We need two special pointers Front and Rear. We add elements to the rear of the queue and remove elements from the front of the queue.



*dequeue()*

*enqueue(item)*



# Queue ADT

Main queue operations:

- **enqueue(object):** inserts an element at the end of the queue
- **object dequeue():** removes and returns the element at the front of the queue

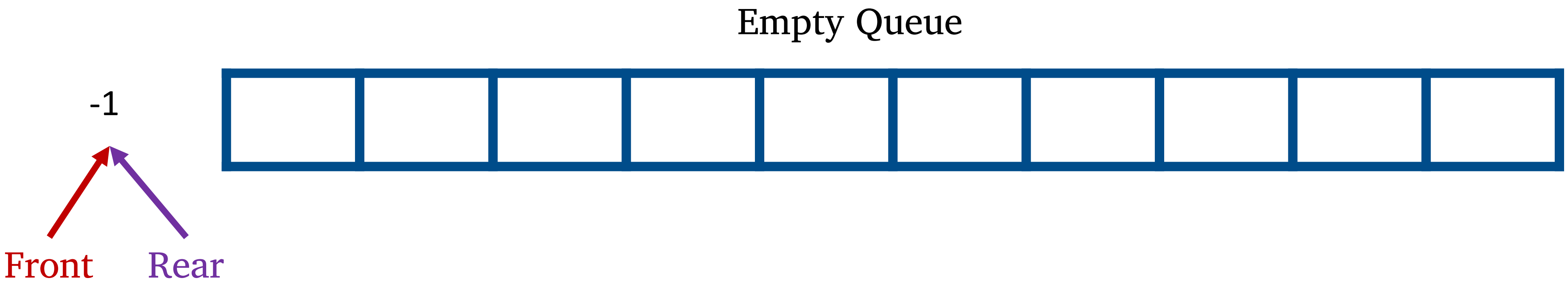
Auxiliary queue operations:

- object **peek()**: returns the element at the front without removing it
- integer **size()**: returns the number of elements stored
- boolean **isEmpty():** indicates whether no elements are stored

# Queue ADT: Applications

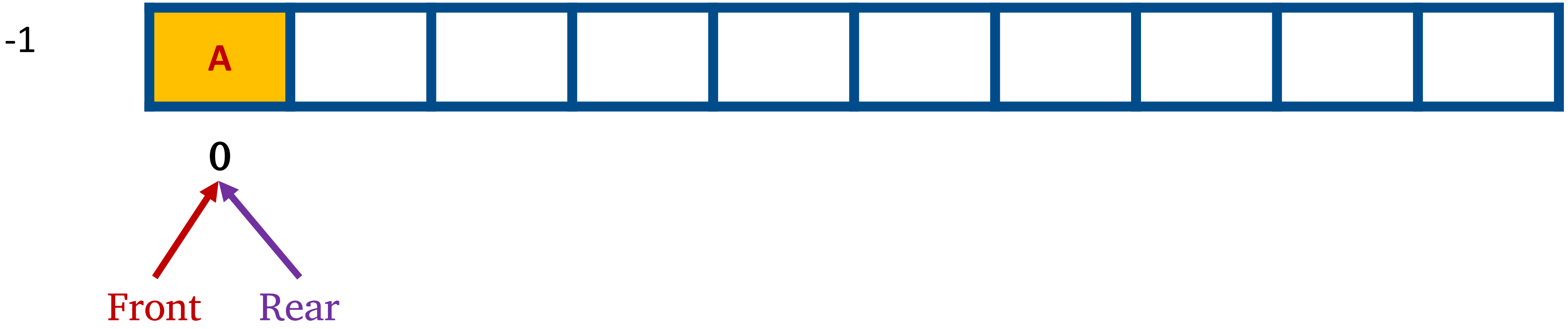
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Queue ADT: Array Based Implementation



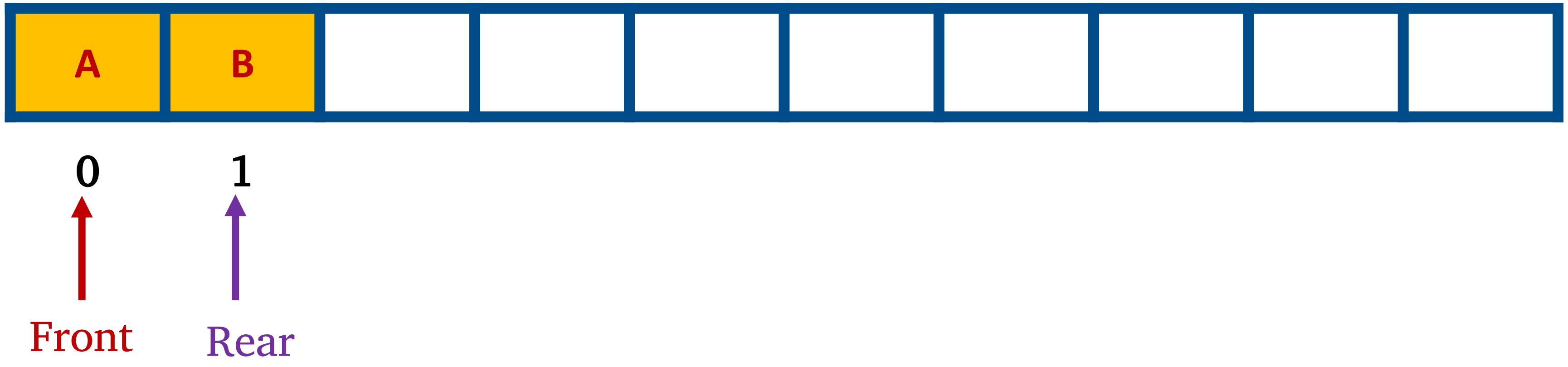
# Queue ADT: Array Based Implementation

Enqueue(A)



# Queue ADT: Array Based Implementation

Enqueue(B)



A

B

0

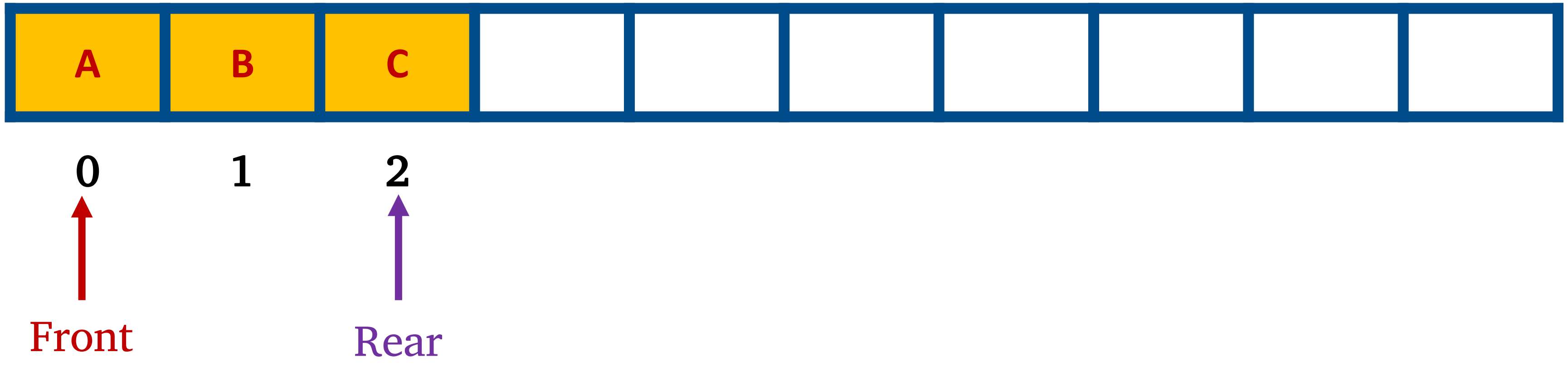
1

Front

Rear

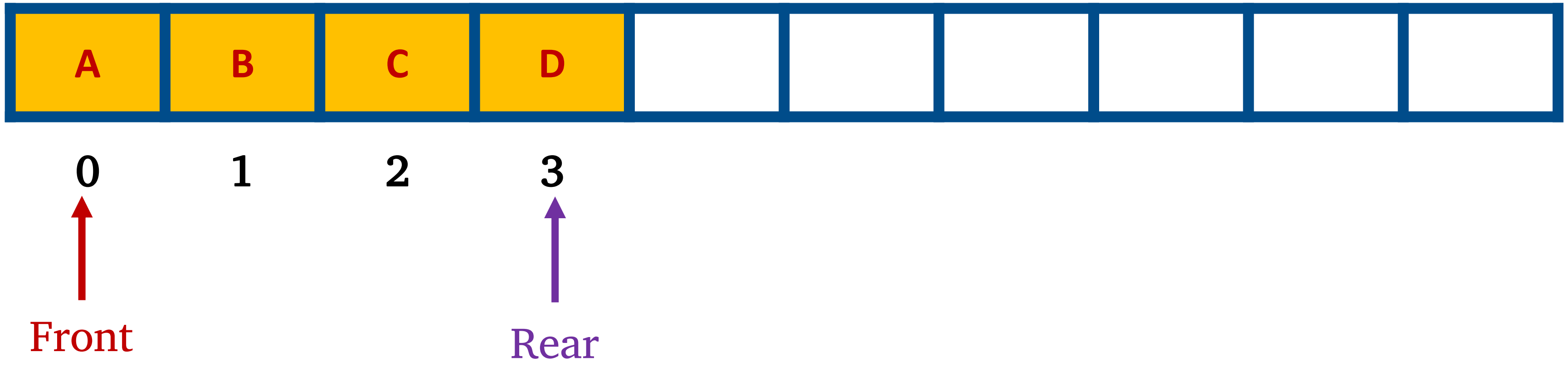
# Queue ADT: Array Based Implementation

Enqueue(C)



# Queue ADT: Array Based Implementation

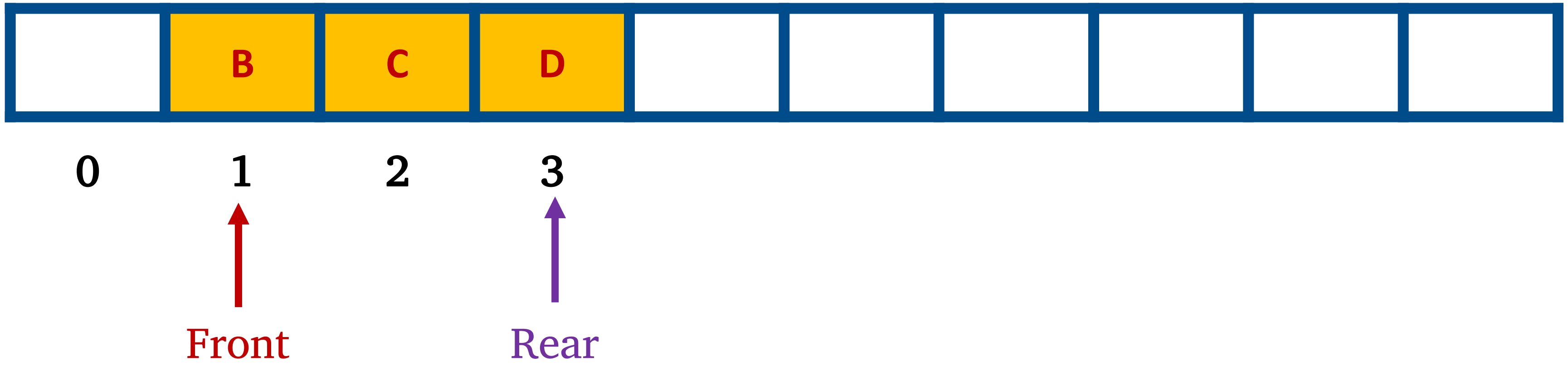
Enqueue(D)



# Queue ADT: Array Based Implementation

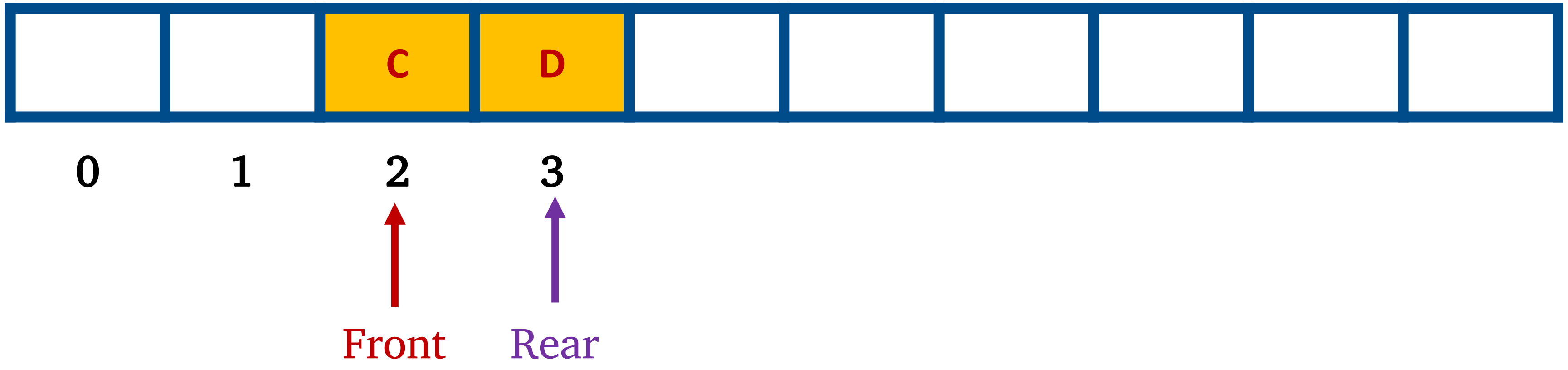
- Shifting the elements has a time complexity of O(n)
- Moving the front and end indices give a time complexity of O(1)

dequeue()



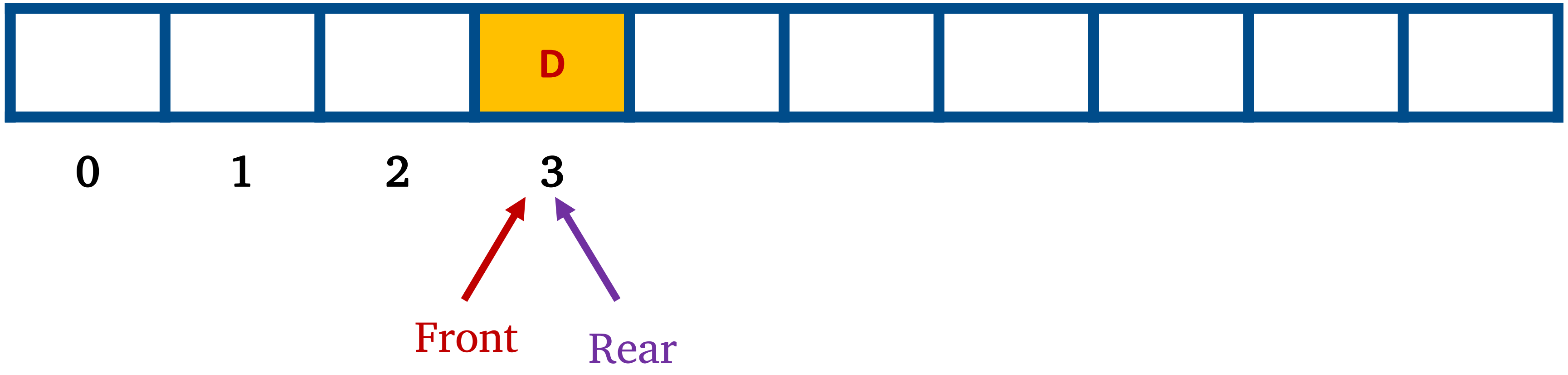
# Queue ADT: Array Based Implementation

dequeue()



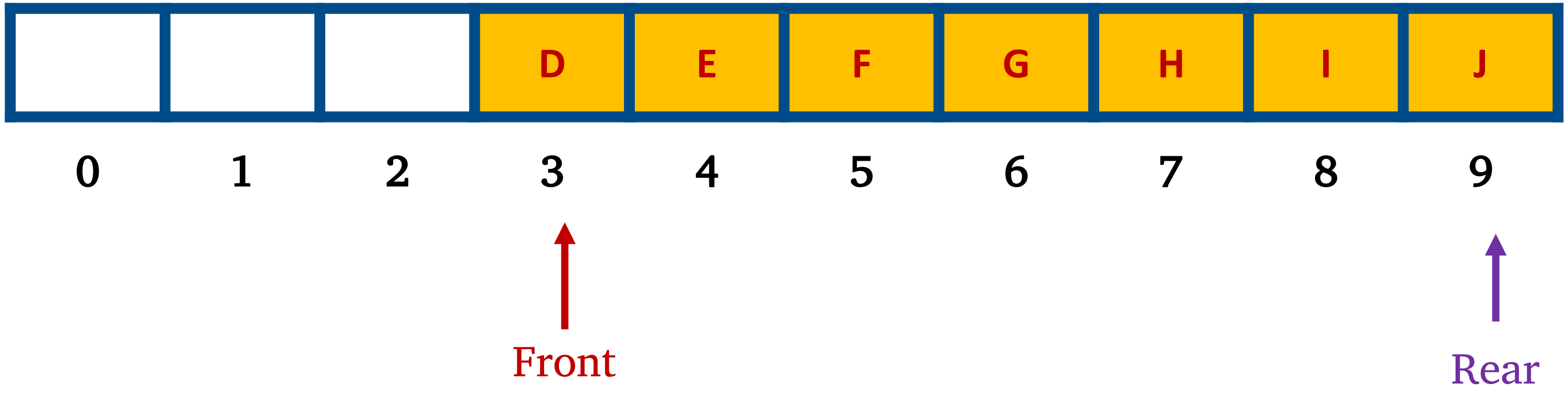
# Queue ADT: Array Based Implementation

dequeue()



# Queue ADT: Array Based Implementation

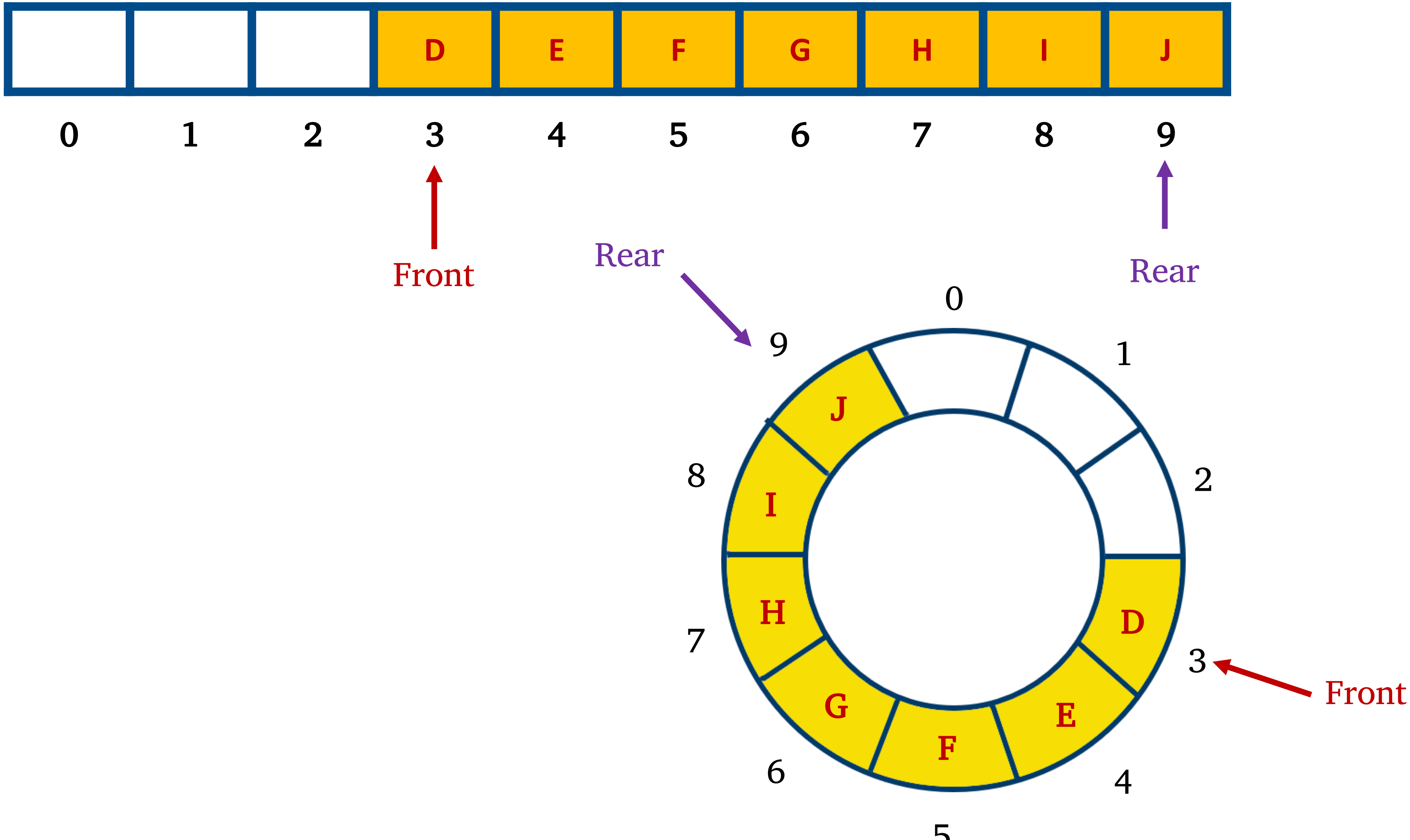
enqueue(K)



# Queue ADT: Array Based Implementation

- Using simple array to implement a queue data structure is **inefficient**.
- It is better to use a **circular array** (we move the front and the rear indices and not shifting the elements)

# Queue ADT: Array Based Implementation



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  | D | E | F | G | H | I | J |

Front

Rear

# Queue ADT: Circular Array

```
public class QueueADT {

    int front, rear, size;
    int capacity;
    int[] myArray;

    public QueueADT(int capacity)
    {
        this.capacity = capacity;
        this.front = -1;
        this.rear = -1;
        myArray = new int[capacity];
        this.size=0;
    }
```

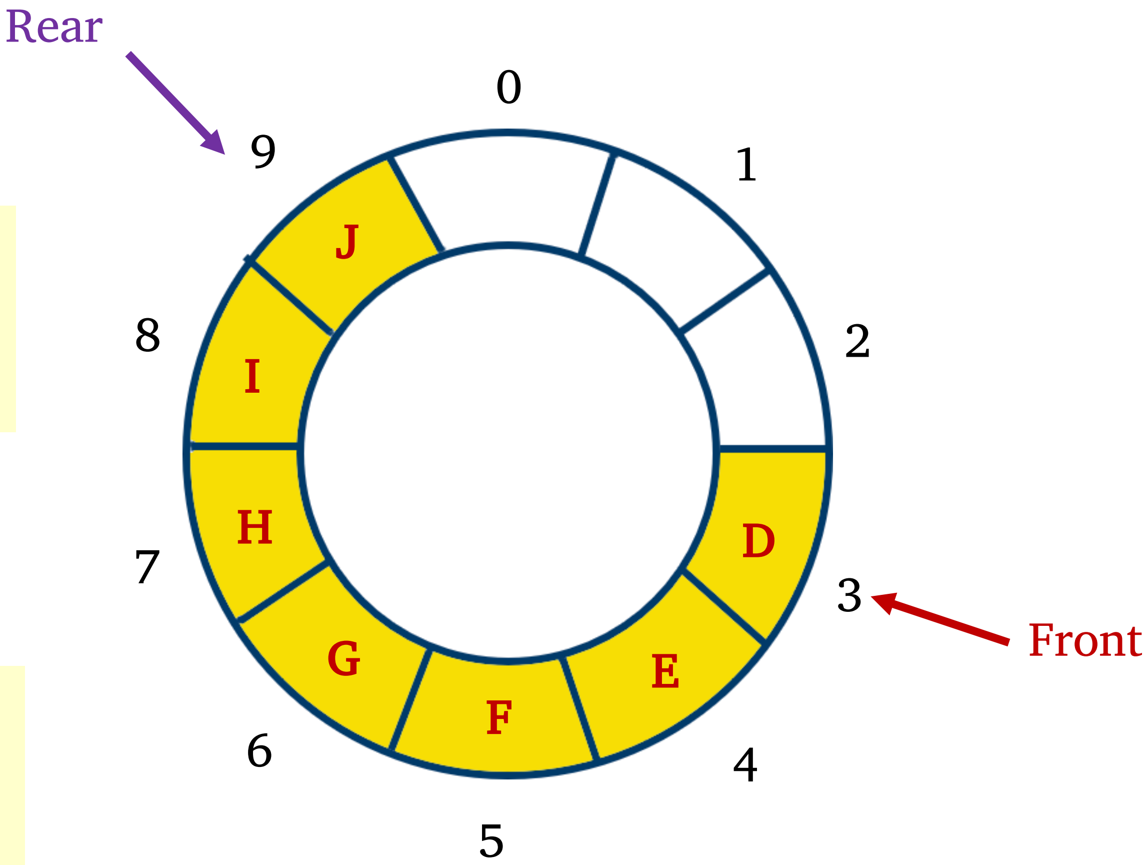
# Queue ADT: isFull()

Solution 1:

```
public boolean isFull()
{
    return (size == capacity);
}
```

Solution 2:

```
public boolean isFull()
{
    return (((rear+1) % capacity) == front);
}
```



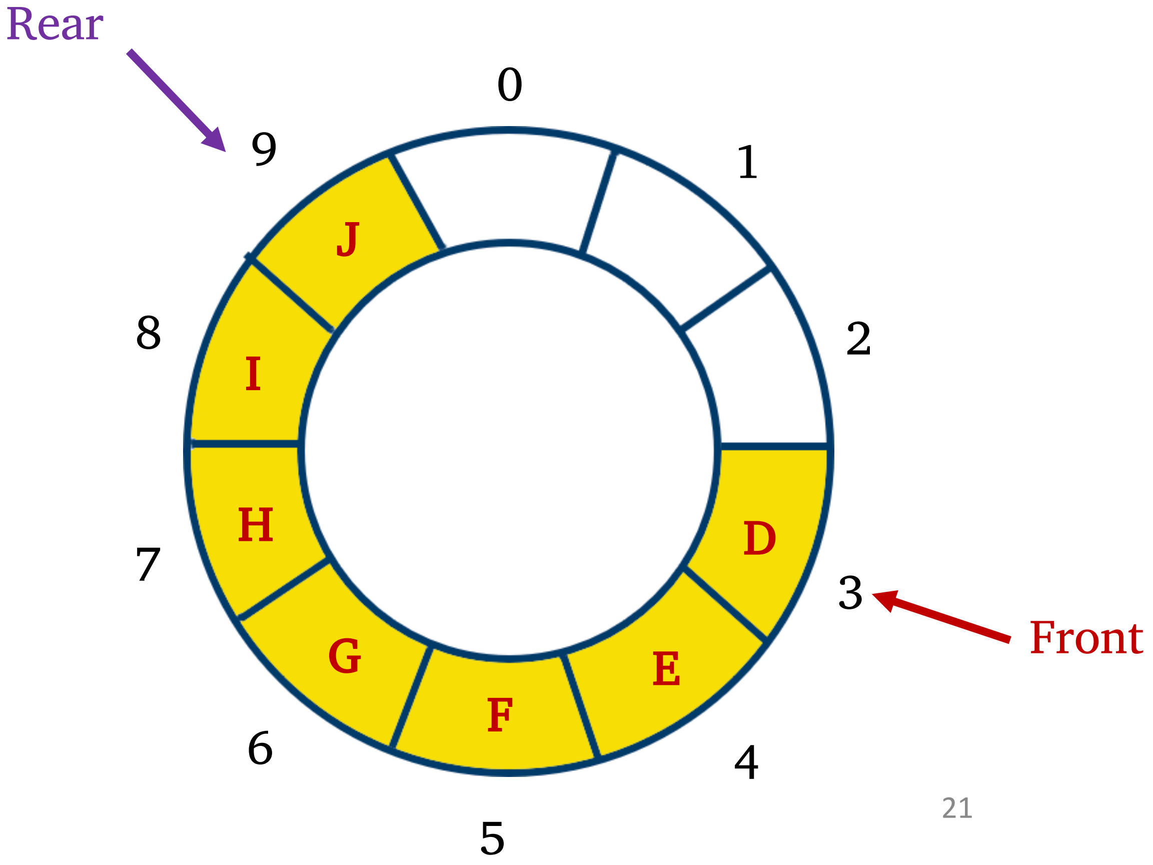
# Queue ADT: isEmpty()

Solution 1:

```
public boolean isEmpty()
{
    return (size == 0);
}
```

Solution 2:

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



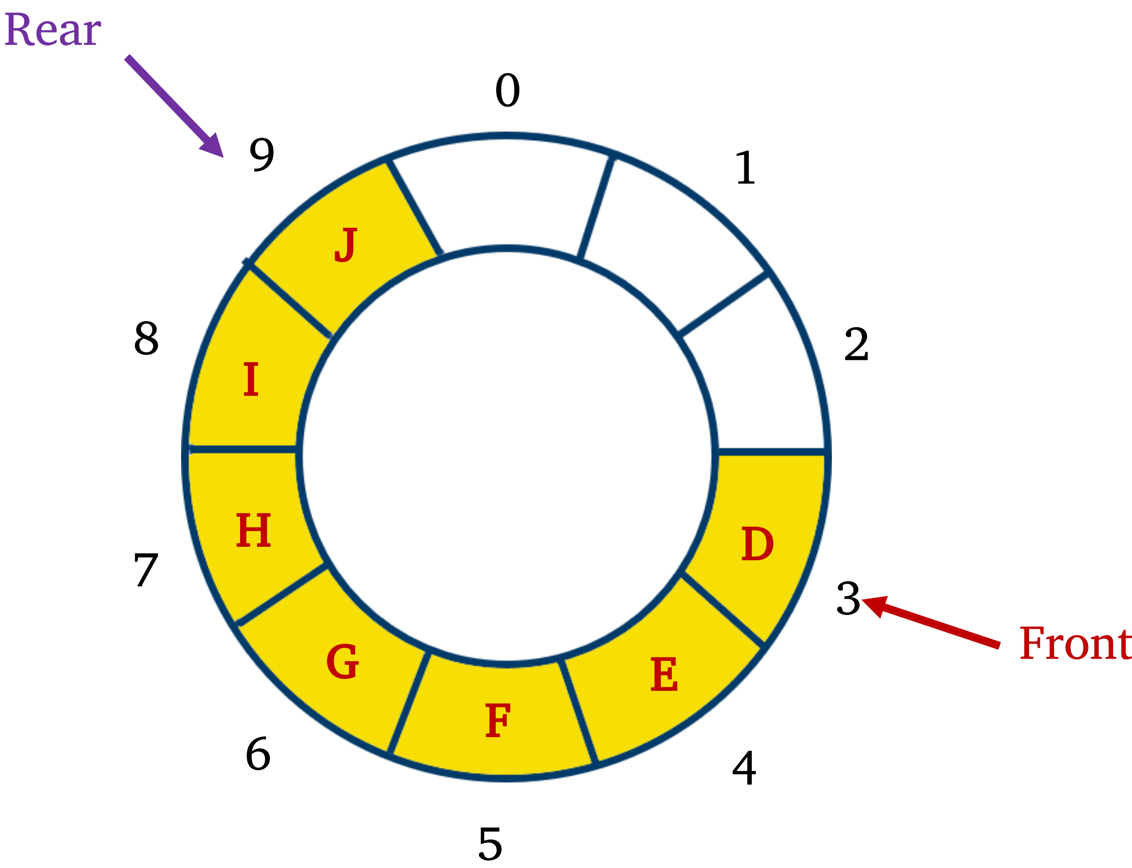
# Queue ADT: Insertion (Enqueue)

```
public void enqueue(int data)
{
    if (isFull()) {
        System.out.println("Queue is Full.");
        return;
    }
    else{
        if(isEmpty()) {
            front = 0;
        }
        rear=(rear+1)%capacity;
        myArray[rear]=data;
        size++;
    }
}
```

Circular Array

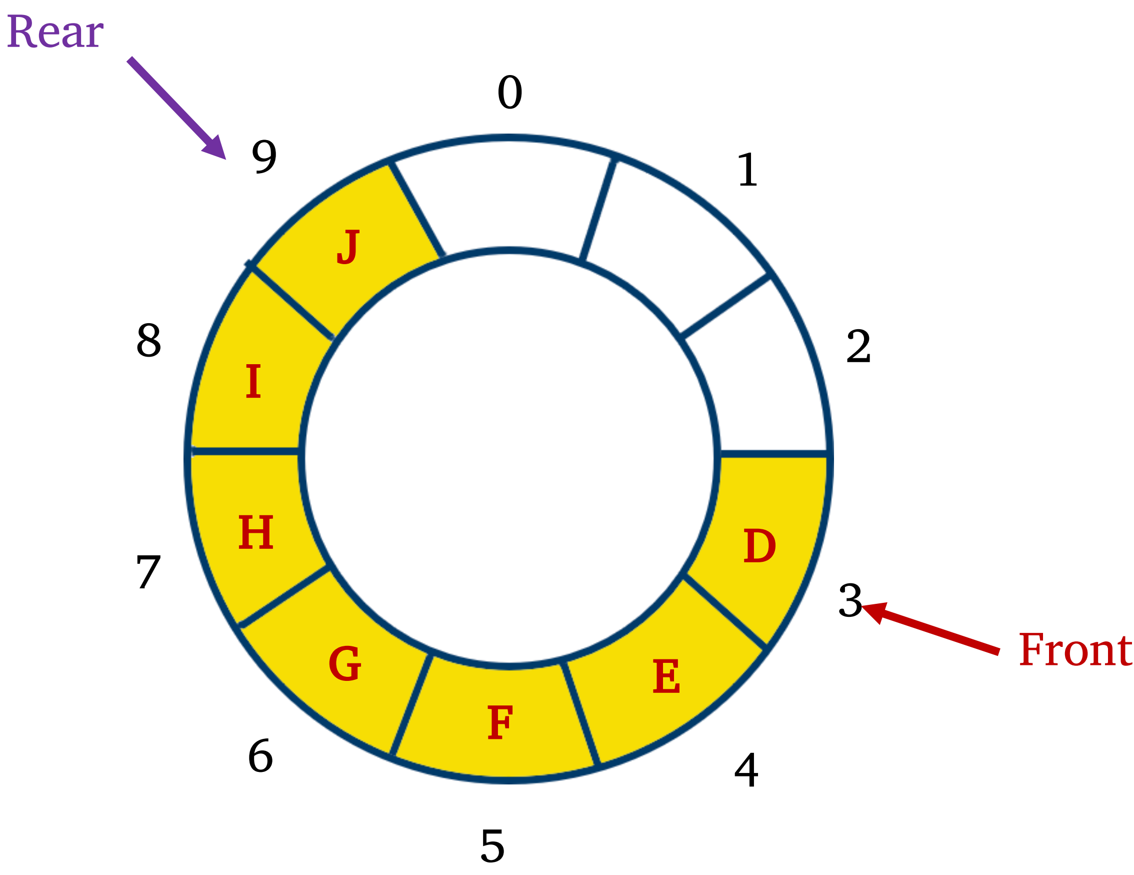
# Queue ADT: Insertion (Enqueue)

```
public void enqueue(int data)
{
    if (isFull()) {
        System.out.println("Queue is Full.");
        return;
    }
    else{
        if(isEmpty()) {
            front = 0;
        }
        rear=(rear+1)%capacity;
        myArray[rear]=data;
        size++;
    }
}
```



# Queue ADT: Deletion (Dequeue)

```
public void dequeue()
{
    if(isEmpty())
    {
        System.out.println("Queue is Empty!");
    }
    else{
        if(front==rear) {
            front = rear = -1;
        }
            front = (front + 1)%capacity;
            size--;
        }
}
```



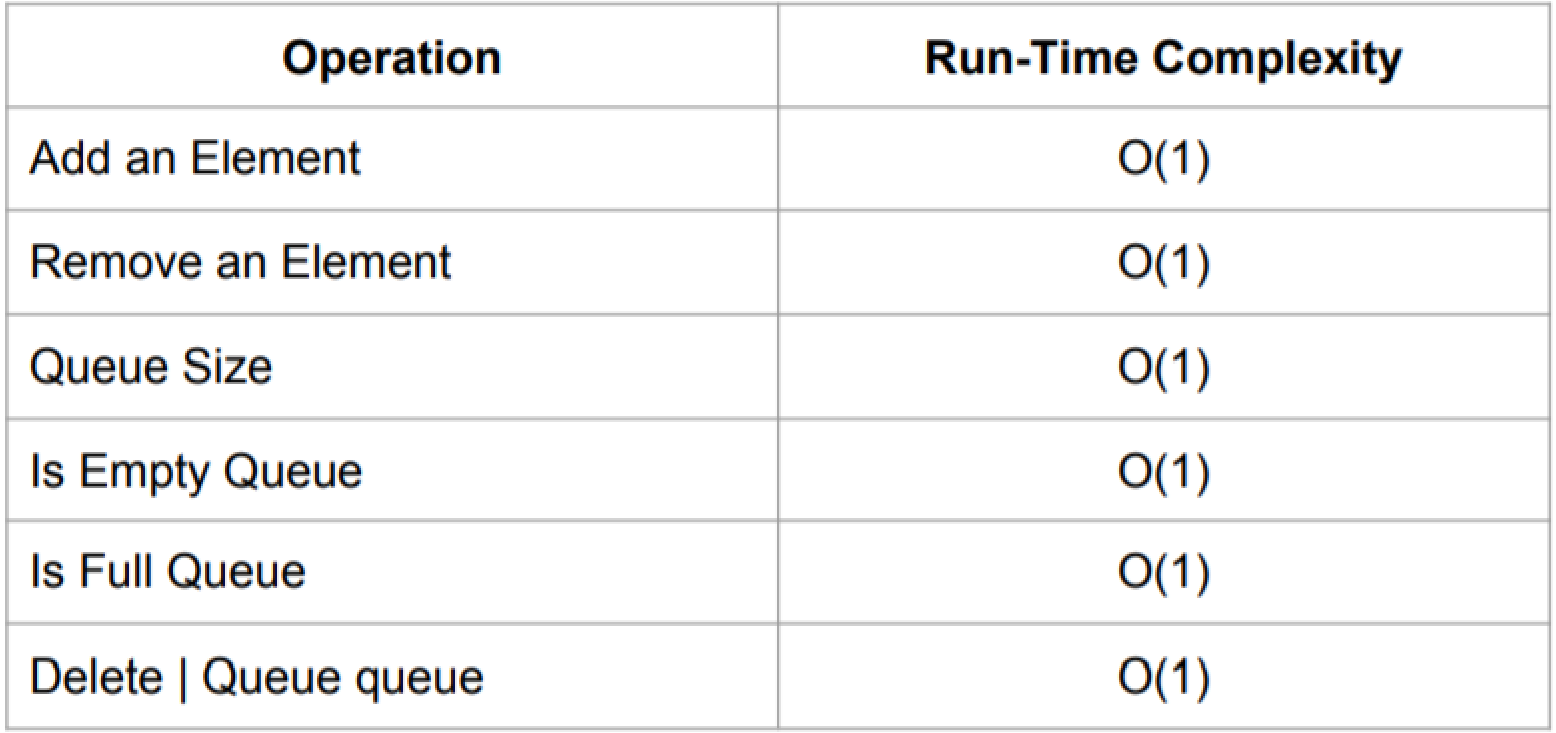
# Queue ADT: Peek()

```
public int peek()
{
    if (isEmpty())
        return -1;
    else
        return myArray[front];

}
```

# Run-Time Complexity

- The run time complexity of the Queue operations with array-based implementation are:



| Operation | Run-Time Complexity |
|---|---|
| Add an Element | O(1) |
| Remove an Element | O(1) |
| Queue Size | O(1) |
| Is Empty Queue | O(1) |
| Is Full Queue | O(1) |
| Delete \| Queue queue | O(1) |

# In class assessment

- Write an algorithm to reverse a queue of n elements using only the queue ADT methods?

# Self assessment

- Explain how could you implement a FIFO (queue) using linked list?
- Explain how could you implement a FIFO (queue) using stacks?