Chapter 3
Data Structures
Static Queue

By: Dr. Aryaf A. Al-adwan

Faculty of Engineering Technology

Computer and Networks Engineering Dept.

Data Structures Course

Outline

- Queue Theory and Example
- Linear Queue Data Members and Operations
- Linear Queue Implementation in C++
- Queue Applications
- Circular Queue Data Members and Operations
- Circular Queue Implementation in C++

Static Linear Queue

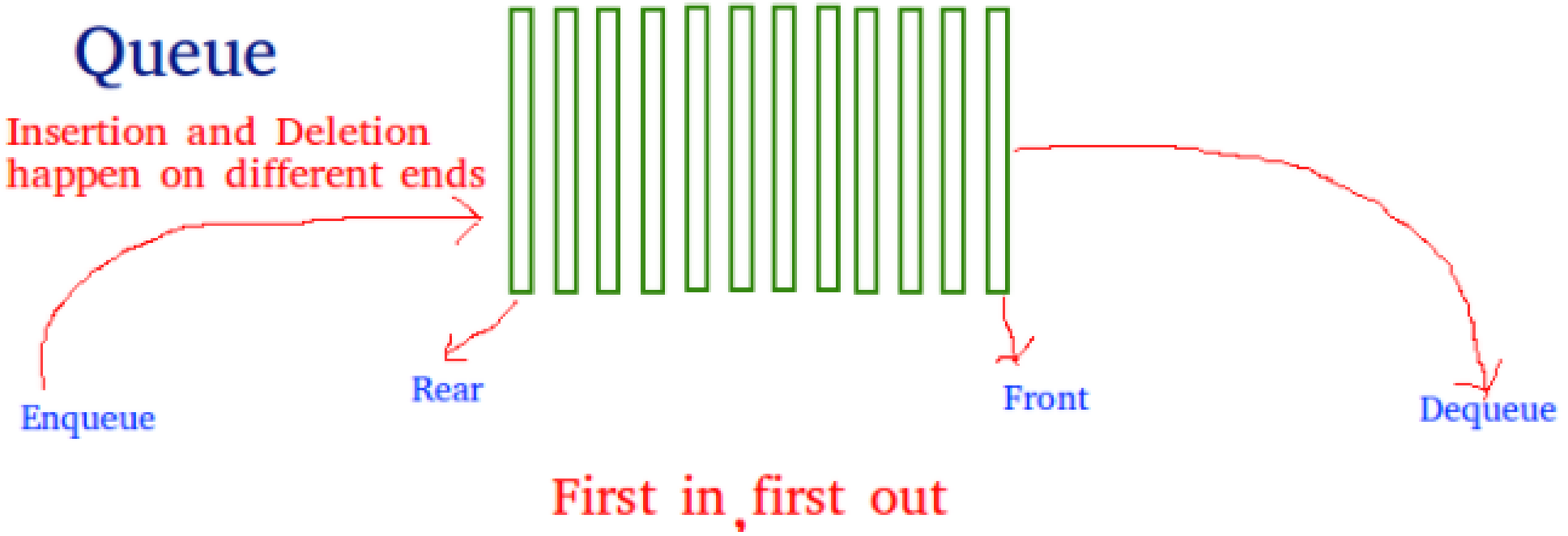
Queue

- A queue is an Abstract Data Type (ADT)
- It is queue is an ordered list of elements where an element is inserted at the end of the queue and is removed from the front of the queue.
- Unlike a stack, which works based on the last-in, first-out (LIFO) principle, a queue works based on the first-in, first-out (**FIFO**) principle.
- The name *queue* comes from the analogy to a queue of customers at a bank. The customer who comes first will be served first, and the one who comes later is queued at the end of the queue and will be served later.

Cont.

- Queue can be implemented by means of **Array** and **Linked List**. Here, we are going to implement stack using arrays, which makes it a fixed size queue implementation.
- A queue has two main operations involving inserting a new element and removing an existing element.
- The insertion operation is called ***enqueue***, and the removal operation is called ***dequeue***.
- The enqueue operation inserts an element at the end of the queue
- The dequeue operation removes an element from the front of a queue.

Cont.



we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Data Members

1. A template array.
2. **Front** index for **enqueing (also called head)**
3. **Rear** index for **dequeueing (also called tail)**

Basic Operations

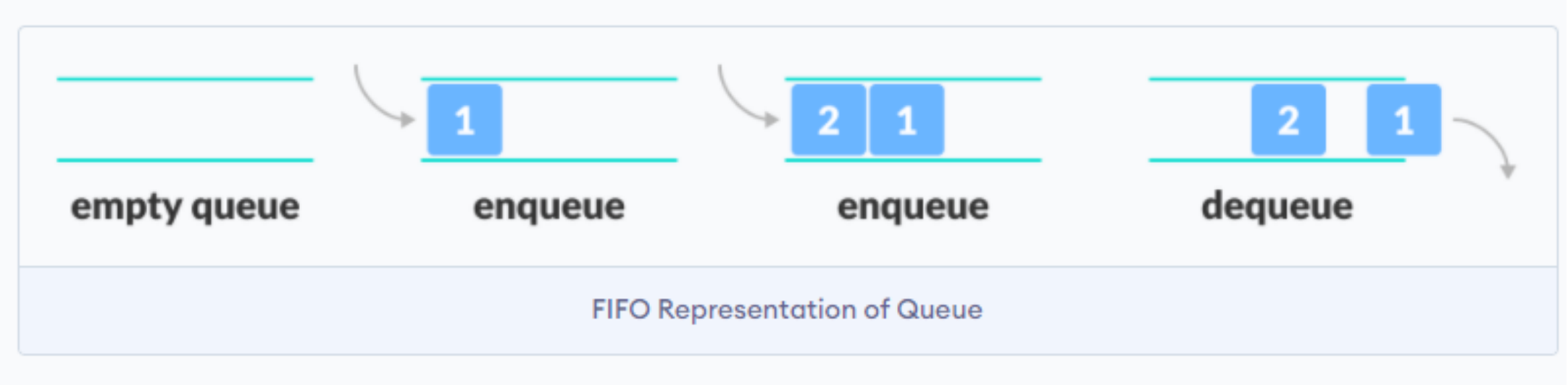
1. **enqueue()** – add (store) an item to the queue.
2. **dequeue()** – remove (access) an item from the queue.
3. **peek()** – Gets the element at the front of the queue without removing it.
4. **isfull()** – Checks if the queue is full.
5. **isempty()** – Checks if the queue is empty.

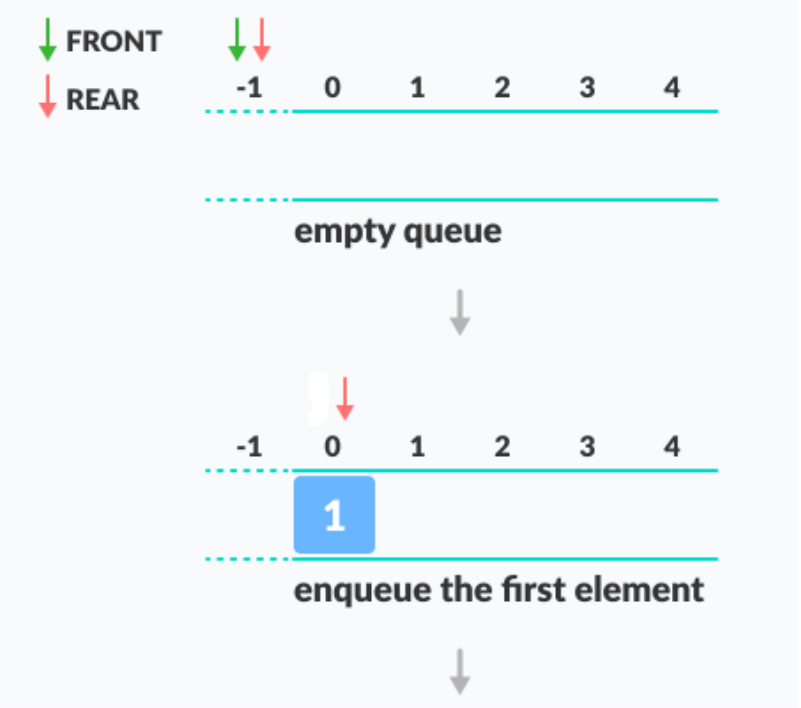
Cont.

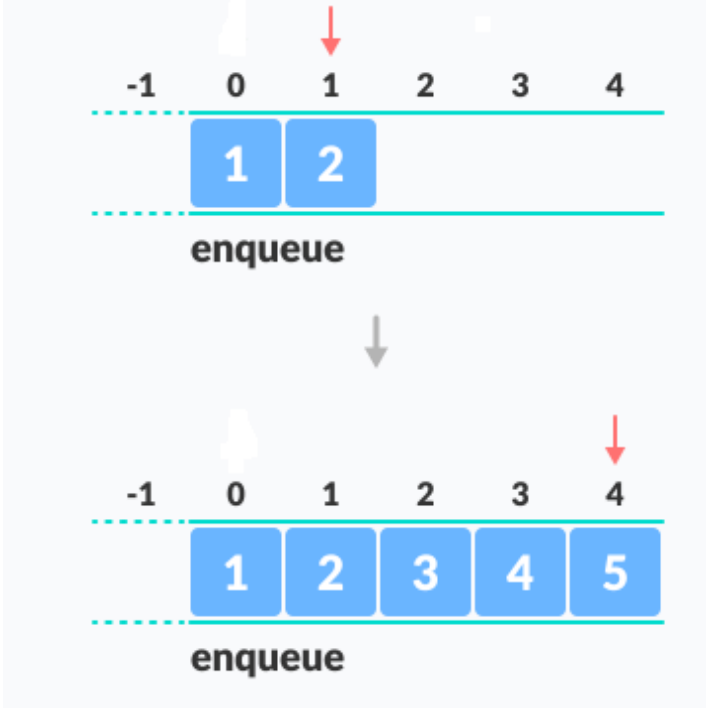
enqueue() Operation	dequeue() Operation	isFull () Operation	isEmpty () Operation
Step 1 – Check if the queue is full. **Step 2** – If the queue is full, produce overflow error and exit. **Step 3** – If the queue is not full, increment rear index to point the next empty space. **Step 4** – Add data element to the queue location, where the rear is pointing. **Step 5** – return success.	**Step 1** – Check if the queue is empty. **Step 2** – If the queue is empty, produce underflow error and exit. **Step 3** – If the queue is not empty, access the data where **front** is pointing. **Step 4** – Increment **front** index to point to the next available data element. **Step 5** – Return success.	**Step 1** – Checks if the **rear** equals the size -1 **Step 2** – Return True if yes and return False if no.	**Step 1** – Checks if the **front** equals **rear**. **Step 2** – Return True if yes and return False if no.

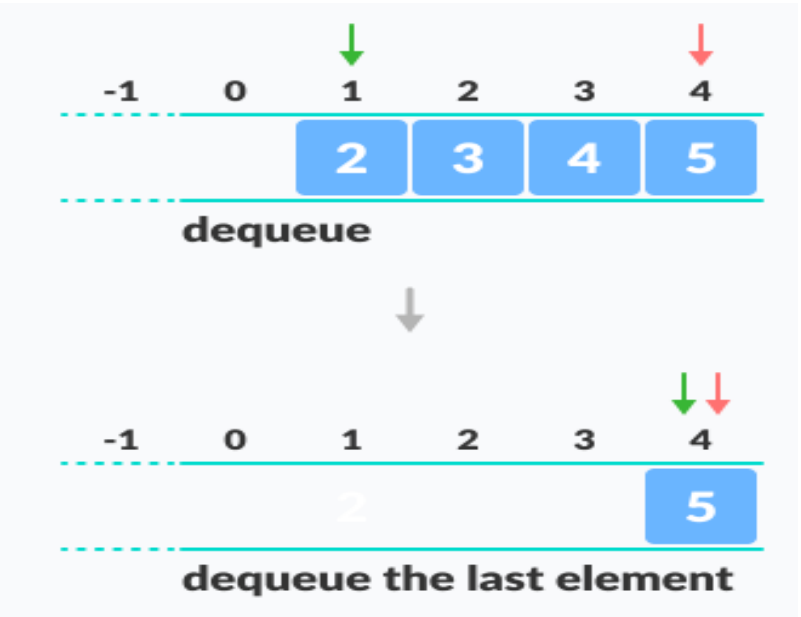
Basic Operations

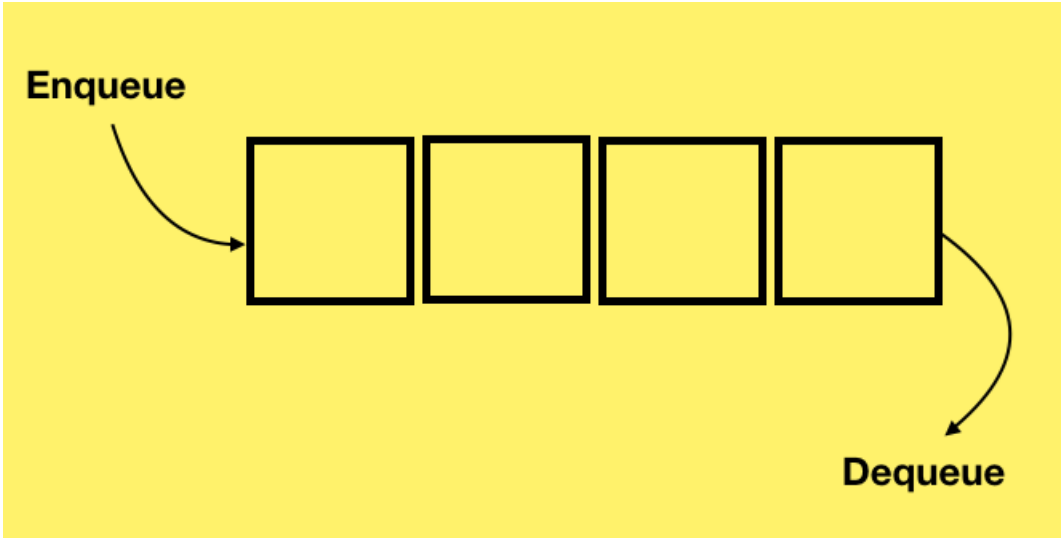
- https://www.youtube.com/watch?v=PjQdvpWfCmE




FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element


-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue


-1 0 1 2 3 4
2 3 4 5
dequeue
-1 0 1 2 3 4
5
dequeue the last element



Queue Size=4

Front=-1

Rear=-1

Enequeue(10)➔ rear++ ➔ insert(10)

Enequeue(5)➔ rear++ ➔ insert(5)

Enequeue(6)➔ rear++ ➔ insert(6)

Enequeue(9)➔ rear++ ➔ insert(9)

Enequeue(30)➔ Full ➔ rear=size-1

Queue Size=4

Front=-1

Rear=3

Dequeue ()➔ front++ ➔ return 10

Dequeue ()➔ front++ ➔ return 5

Dequeue ()➔ front++ ➔ return 6

Dequeue ()➔ front++ ➔ return 9

Dequeue ()➔ Empty ➔ rear=front

Applications

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served
- Queue of packets data in networks
- Process scheduling in operating systems

```
// this program for implementing a static linear queue using templates
// programmed by Dr.Aryaf Al-adwan
#include<iostream>
Using namespace std;
const int size=8;
template <class T>
class linearqueue
{
private:
int front;
int rear;
int count;
T arr[size];
public:
linearqueue()
{
front=-1;
rear=-1;
count=0;
}
```

```
 bool isEmpty()
{
if(rear== front)
return true;
else
return false;
}

bool isFull()
{
if((rear==size-1))
return true;
else
return false;
}
```

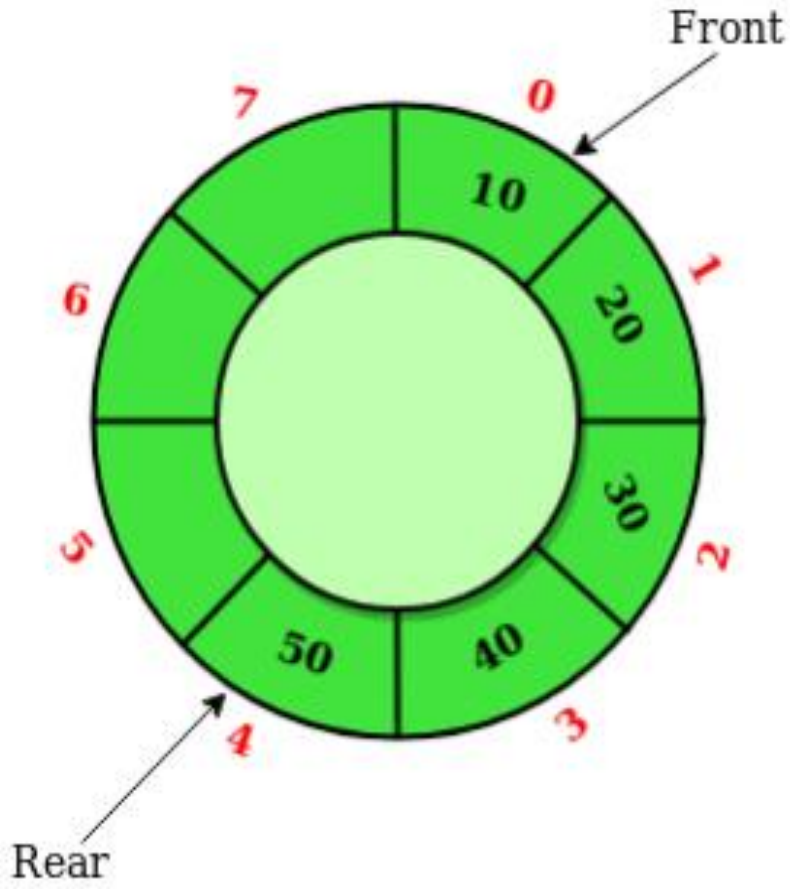
```
void enqueue (T item)
{
if(isFull())
{
cout<<"Queue is Full\n";
}
else
{
rear++;
arr[rear]=item;
count++;
}
}

T dequeue ()
{
T dequeueitem;
if(isEmpty())
{
 cout<<"Queue is Empty\n";
return 0;
}
```

```
else
{
front++;
dequeueitem=arr[front];
count--;
if(count<0)
count=0;
return dequeueitem;
}
}
};

void main()
{
linearqueue <int> q1;
q1.dequeue();
q1.enqueue(1);
 q1.enqueue(2);
q1.enqueue(3);
for(int i=0;i<size;i++)
 cout << q1.dequeue() << endl;
}
```

Static Circular Queue

Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is **connected back to the first position (Wrapping the index)** to make a circle. It is also called 'Ring Buffer'.
- To apply this we use the **modulus operator (%).**
- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Queue Size = 6

front = 0

rear = Size - 1 = 5

count = 0

Enequeue(10)➔ rear=(rear+1)%Size ➔ rear = 0 ➔ insert(10) , count++ ➔ count=1

Enequeue(13)➔ rear=(rear+1)%Size ➔ rear = 1 ➔ insert(10) , count =2

Enequeue(15)➔ rear=(rear+1)%Size ➔ rear = 2 ➔ insert(10) , count =3

Enequeue (9)➔ rear=(rear+1) %Size ➔ rear = 3 ➔ insert(10) , count =4

Enequeue(77)➔ rear=(rear+1)%Size ➔ rear = 4 ➔ insert(10) , count =5

Enequeue(56)➔ rear=(rear+1)%Size ➔ rear = 5 ➔ insert(10) , count =6

Enequeue(25)➔ Full Queue ➔ count= Size = 6

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=1 ➔ count =5

Enequeue(40)➔ rear=(rear+1)%Size ➔ rear = 0 ➔ insert(40) , count =6 ➔ **Overwrite**

front = 1

rear = 0

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=1 ➔ count- -➔ count =5

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=2 ➔ count =4

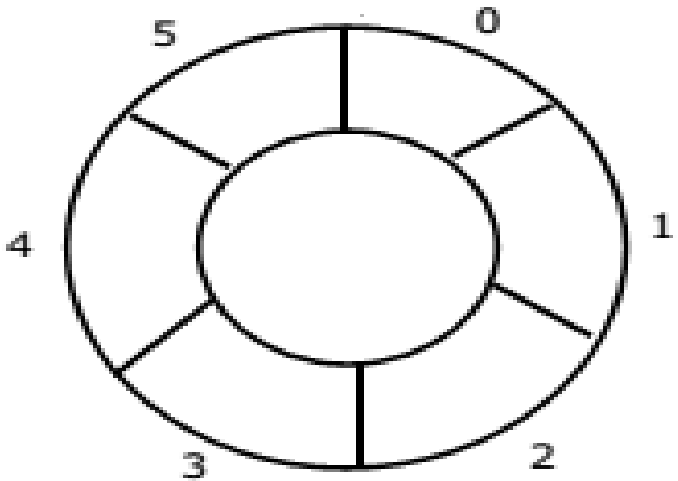
Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=3 ➔ count =3

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=4 ➔ count =2

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=5 ➔ count =1

Dequeue( ) ➔ return 10 ➔ front =(front+1)%Size ➔ front=6➔ count =0

Dequeue( ) ➔ Empty Queue➔ count = 0



Circular Queue

```
// this program for implementing a static
circular queue using templates
// programmed by Dr.Aryaf Al-adwan
#include<iostream>
Using namespace std;
const int size=8;
template <class T>
class circularqueue
{
private:
int front;
int rear;
int count;
T arr[size];
public:
circularqueue()
{
front= 0;
rear= size-1;
count=0;
}
```

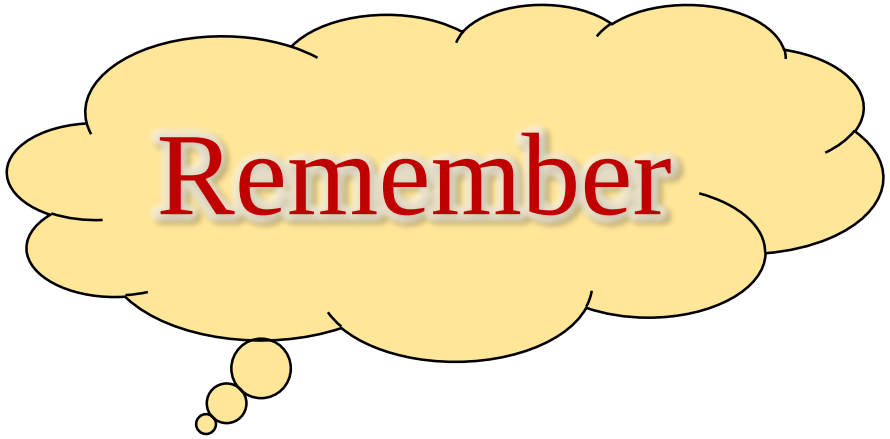
```
 bool isEmpty()
{
if( count == 0)
return true;
else
return false;
}

bool isFull()
{
if (count==size)
return true;
else
return false;
}
```

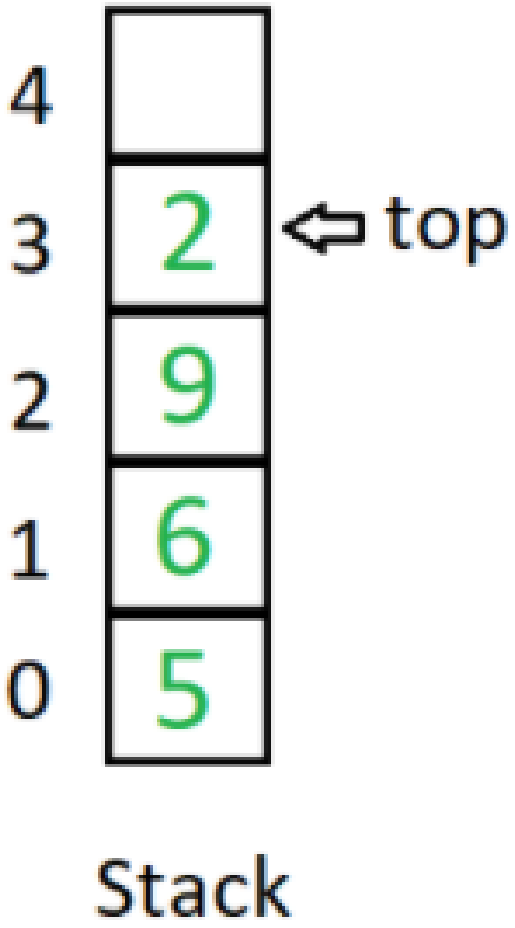
```
void enqueue (T item)
{
if(isFull())
{
cout<<"Queue is Full\n";
}
else
{
rear = (rear+1) % size;
arr[rear]=item;
count++;
}
}

T dequeue ()
{
T dequeueitem;
if(isEmpty())
{
 cout<<"Queue is Empty\n";
return 0;
}
```

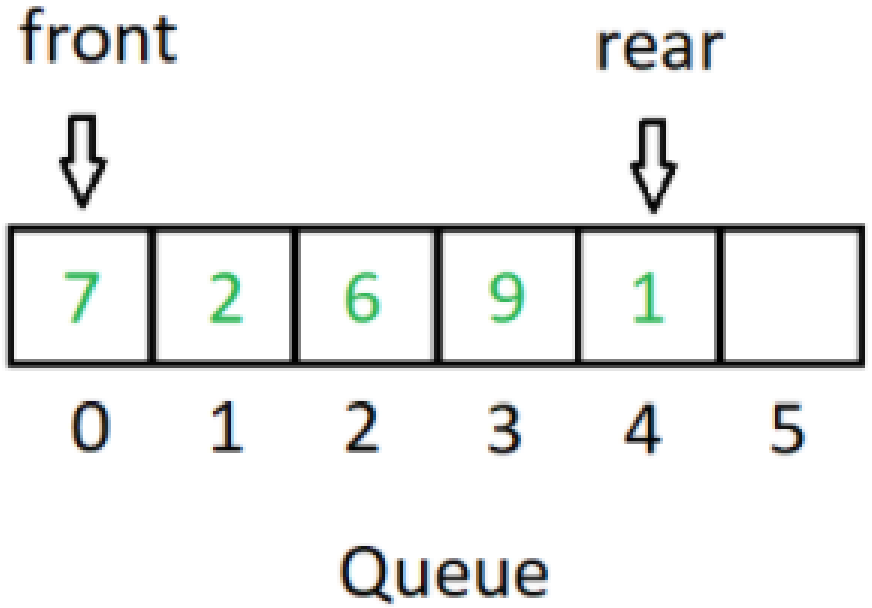
```
else
{
dequeueitem=arr[front];
front= (front+1) % size;
count--;
return dequeueitem;
}
}
};

void main()
{
circularqueue <int> q1;
q1.dequeue();
q1.enqueue(1);
 q1.enqueue(2);
q1.enqueue(3);
for(int i=0;i<size;i++)
 cout << q1.dequeue() << endl;
}
```

Question?

- How to determine if the queue is full or is empty using the rear and front not the count variable?


Remember


4
3
2
1
0
2
9
6
5
top
Stack


front
rear
7
2
6
9
1
0
1
2
3
4
5
Queue

End