Queue

Unlike stacks, a queue is open at both its ends.

One end is always used to insert data (enqueue)

the other is used to remove data (dequeue).

Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

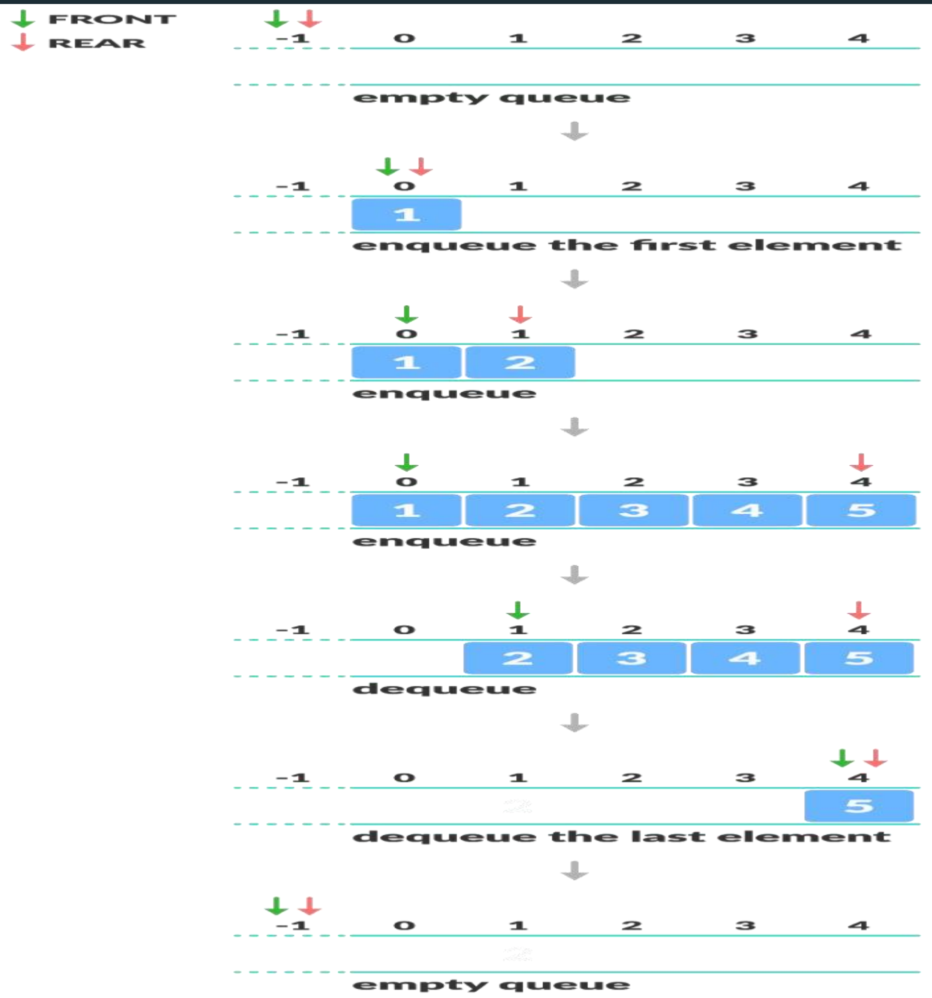
The practical examples of queues are

- The consumer who comes first to a shop will be served first.
- CPU task scheduling and disk scheduling.
- Waiting list of tickets in case of bus and train tickets.

Queue Representation

Basic Operations

- initialize()- initialize front and rear pointer
- enqueue() − add (store) an item to the queue: *enqueing (or storing) data in the queue we take help of rear pointer.*
- dequeue() − remove (access) an item from the queue: *dequeue (or access) data, pointed by front pointer*
- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.


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
-1 0 1 2 3 4
empty queue

peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows –

Algorithm	function in C programming language
begin procedure peek return queue[front] end procedure	int peek() { return queue[front]; }

isfull(): check for the rear pointer to reach at MAXSIZE to determine that the queue is full.

Algorithm	function in C programming
`begin procedure isfull` `if rear equals to MAXSIZE` `return true` `else` `return false` `endif` `end procedure`	`bool isfull() {` `if(rear == MAXSIZE - 1)` `return true;` `else` `return false;` `}`

isempty(): If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Algorithm	function in C programming
`begin procedure isempty` `   if front is less than MIN` `      return true` `   else` `      return false` `   endif` `end procedure`	`bool isempty() {` `   if(front < 0 or` `front>rear)` `      return true;` `   else` `      return false;` `}`

Enqueue Operation

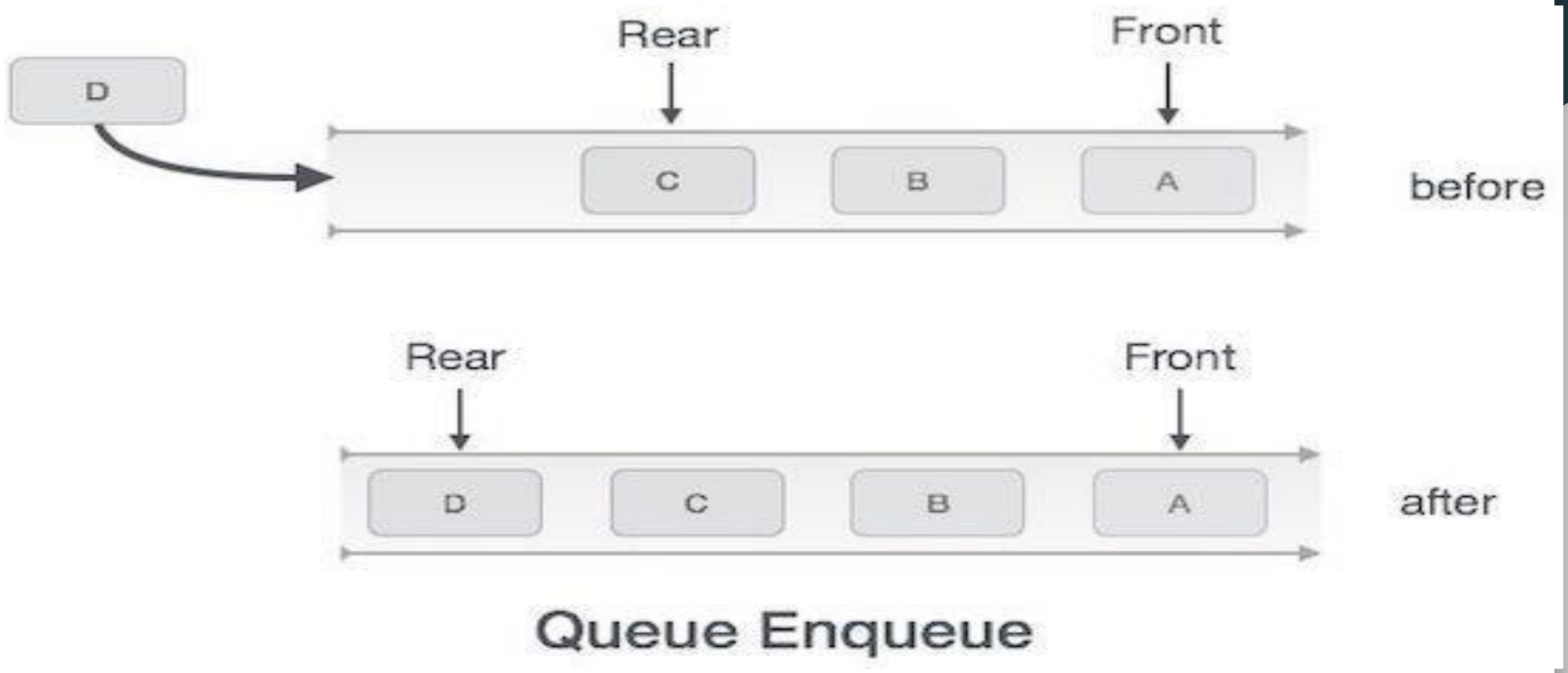
The following steps should be taken to enqueue (insert) data into a queue −

- **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 pointer to point the next empty space.**
- **Step 4 − Add data element to the queue location, where the rear is pointing.**
- **Step 5 − return success.**

Enqueue Operation

Algorithm	function in C programming
``procedure enqueue(data)``` ``` if queue is full``` ``` return overflow``` ``` endif``` ``` rear ← rear + 1``` ``` queue[rear] ← data``` ``` return true``` ```end procedure```	```int enqueue(int data)``` ``` if(isfull())``` ``` return 0;``` ``` rear = rear + 1;``` ``` queue[rear] = data;``` ``` return 1;``` ```end procedure```

Enqueue Operation



Dequeue Operation

The following steps are taken to perform dequeue operation −

- 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 − check if the dequeued element was the only element in the queue; if yes reset Front and Rear pointer to -1

  If No: Increment front pointer to point to the next available data element.
- Step 5 − Return success.

Algorithm	function in C programming

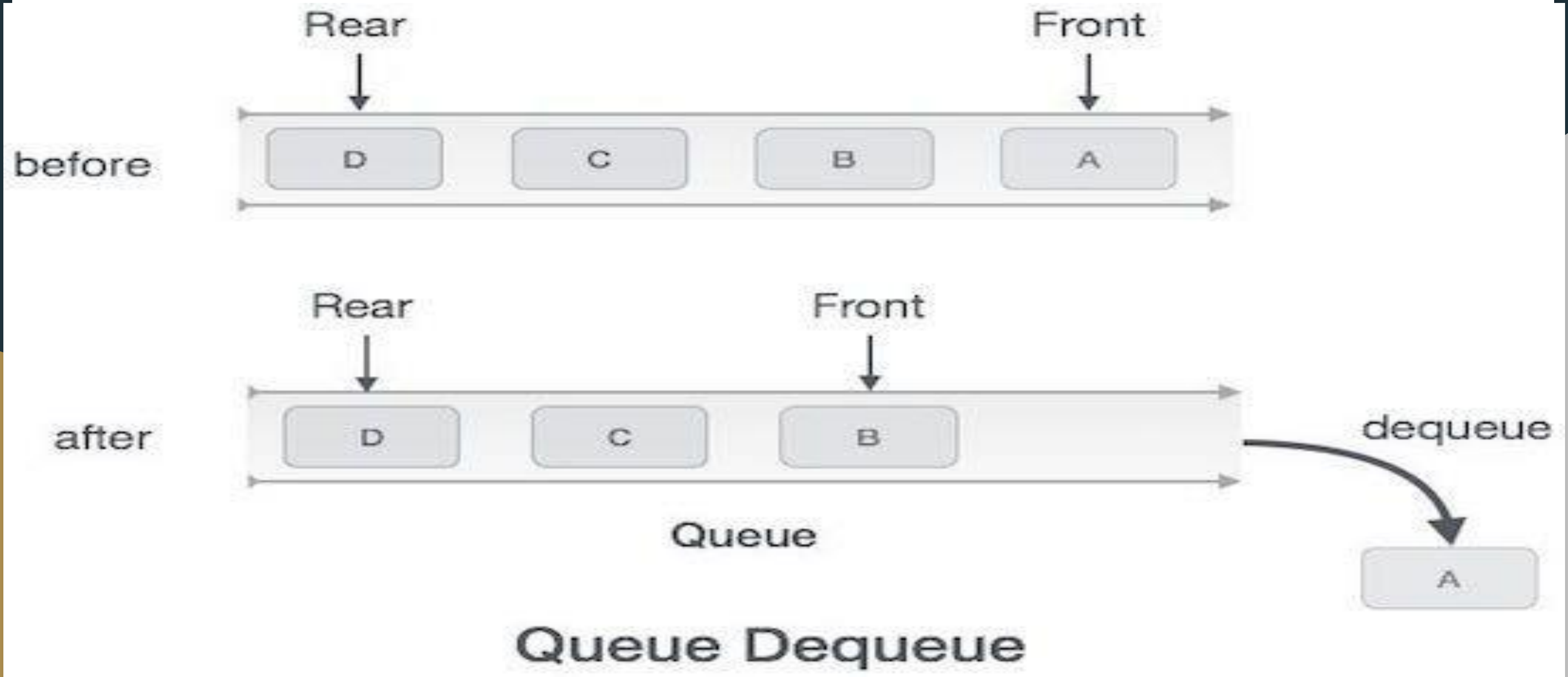
Algorithm

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   if(front==rear)
      front=rear=-1
   else
   front ← front + 1

   return true
end procedure
```

function in C programming

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   if(front==rear)
      front=rear=-1;

   else
   front = front + 1;

   return data;
}
```

Dequeue Operation



Queue Operations

Front = Rear = -1

0	1	2	3	4	5

Empty Queue

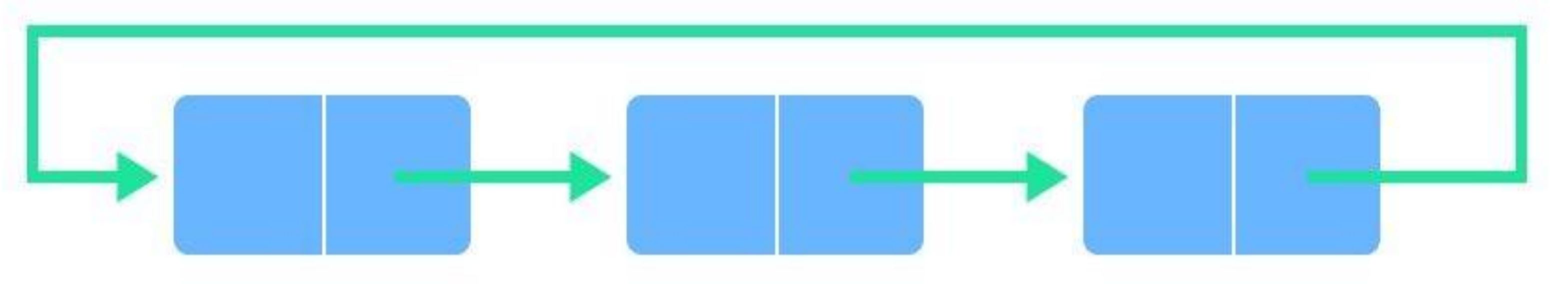
Types of Queues in Data Structure

Queue in data structure is of the following types

1. Simple/Linear Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

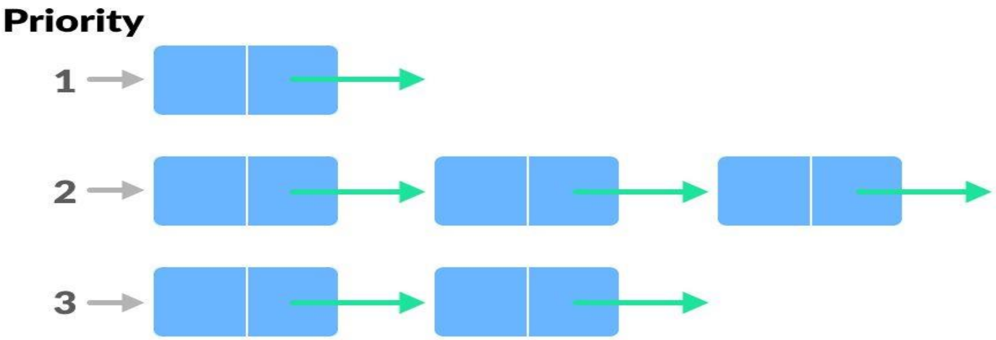
Circular Queue

- In a circular queue, the last node is connected to the first node.
- Circular queue is also called as **Ring Buffer.**
- Insertion in a circular queue happens at the **FRONT** and deletion at the **END** of the queue.



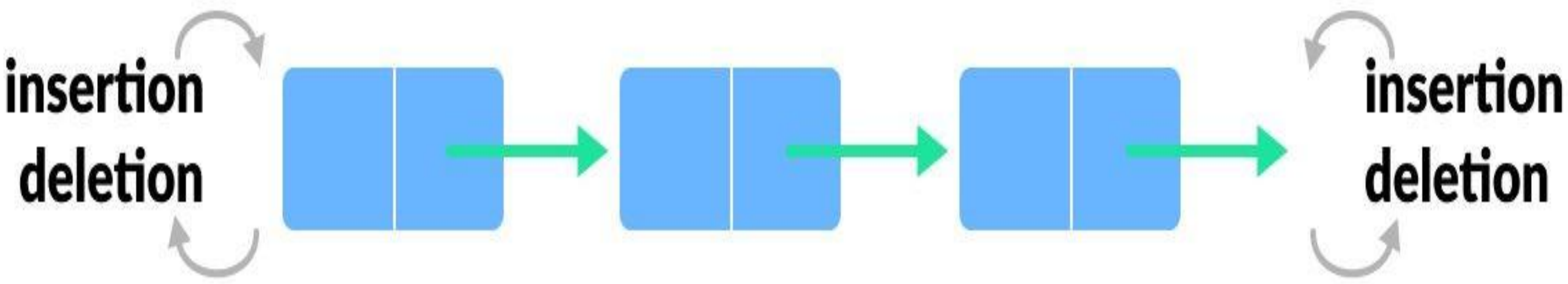
Priority Queue

- In a priority queue, the nodes will have some predefined priority.
- Insertion in a priority queue is performed in the order of arrival of the nodes.
- The node having the least priority will be the first to be removed from the priority queue.

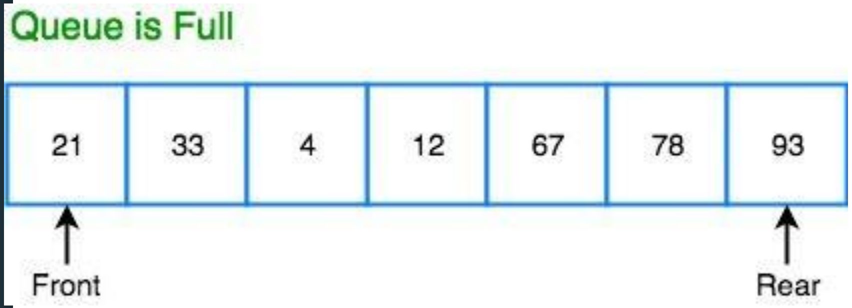


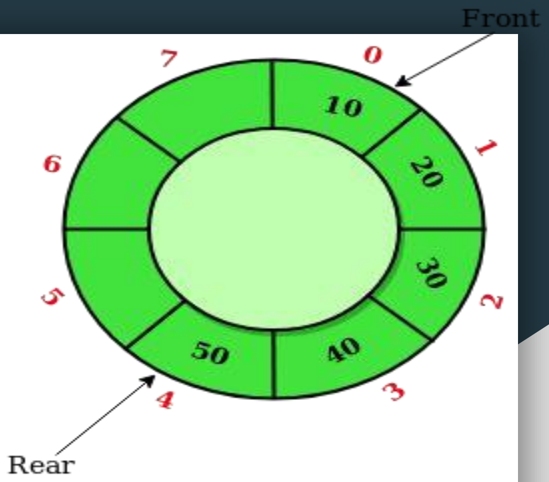
Dequeue (Doubly Ended Queue)

In a Double Ended Queue, insertion and deletion operations can be done at both **FRONT** and **END** of the queue.



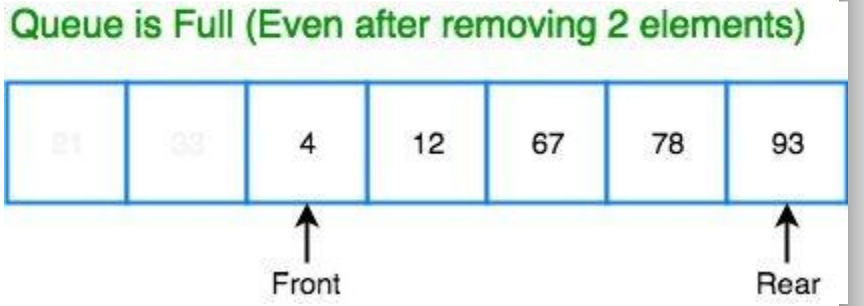
Limitations of Linear Queue



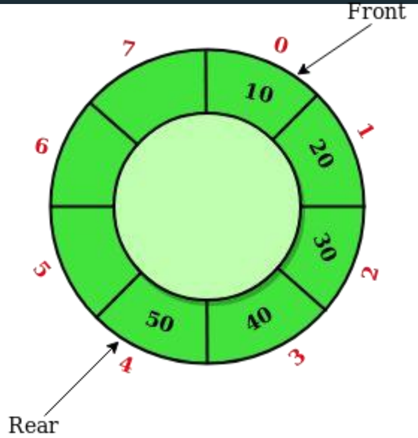


When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, thereby reducing the overall size of the queue.

And we cannot insert new elements, because the **rear** pointer is still at the end of the queue.



How Circular Queue Works



Circular Queue works by the process of circular increment

i.e. when we try to increment the pointer and

we reach the end of the queue, we start from the beginning of the queue.

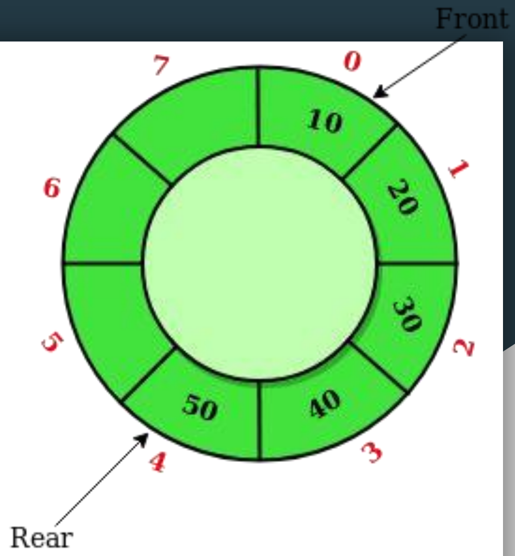
Here, the circular increment is performed by modulo division with the queue size. That is,

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0
(start of queue)
```

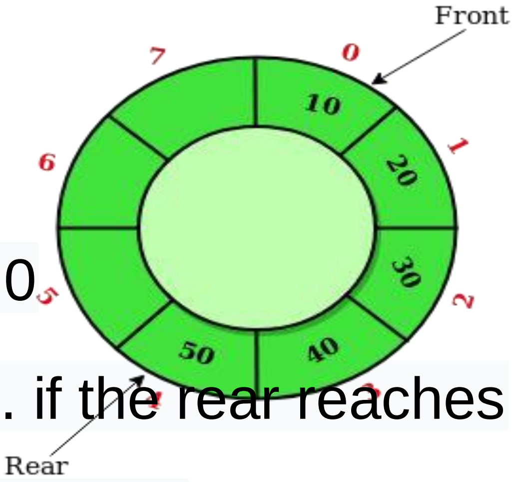
Circular Queue Operations

The circular queue work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue
- `REAR` track the last elements of the queue
- initially, set value of `FRONT` and `REAR` to -1



1. Enqueue Operation

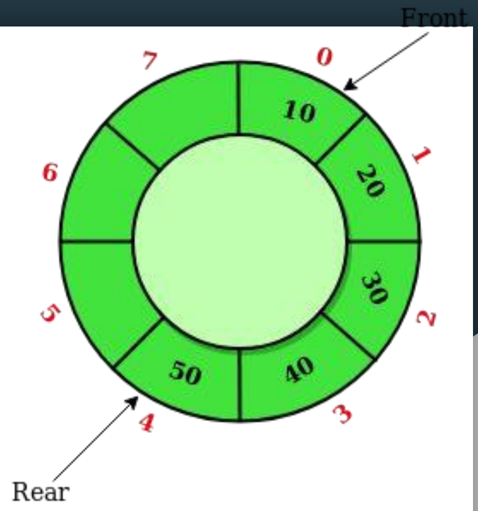


- check if the queue is full
- for the first element, set value of `FRONT` to 0
- circularly increase the `REAR` index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by `REAR`

```
enqueue(int x)
{
    if(isFull())
    {
        cout << "Queue is full";
    }
    else
    {
        if(front == -1)
        {
            front = 0;
        }
        rear = (rear + 1) % SIZE;   // going round and round concept
        // inserting the element
        a[rear] = x;
        cout << endl << "Inserted " << x << endl;
    }
}
```

2. Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- circularly increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1



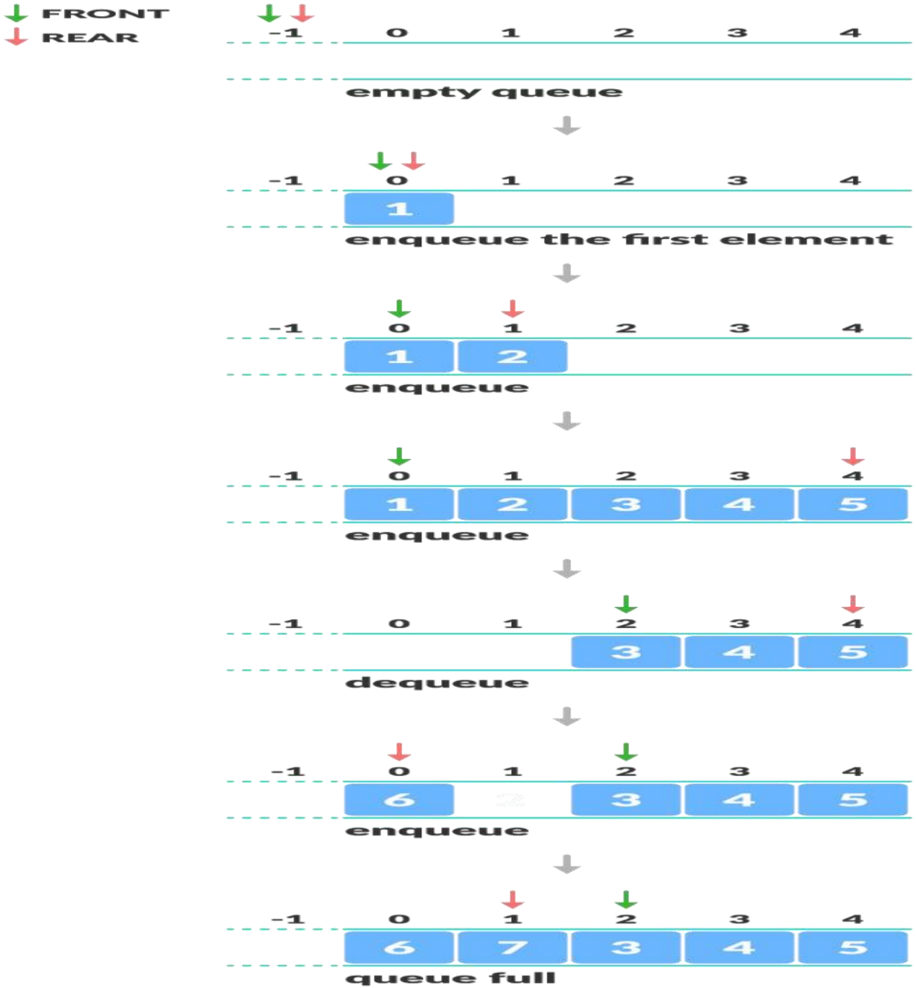
However, the check for full queue has a new additional case:

- Case 1: `FRONT` = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

```
int dequeue()
{
    int y;

    if(isEmpty())
    {
        cout << "Queue is empty" << endl;
    }
else
    {
        y = a[front];
        if(front == rear)
        {
            // only one element in queue, reset queue after removal
            front = -1;
            rear = -1;
        }
        else
        {
            front = (front+1) % SIZE;
        }
        return(y);
    }
}
```

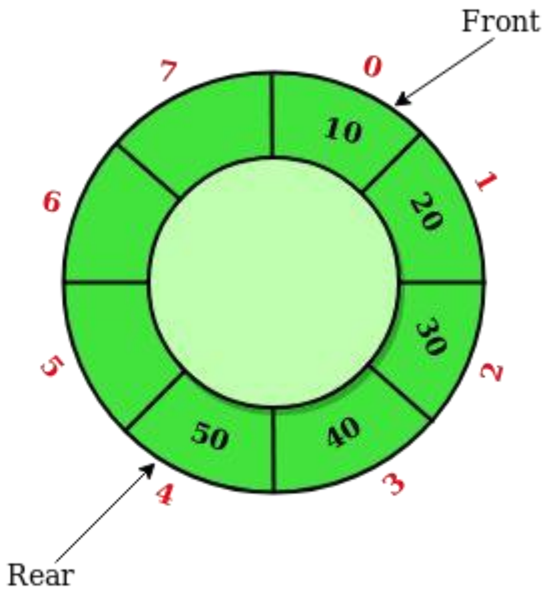
```
bool isFull()
    {
        if(front == 0 && rear == SIZE - 1)
        {
            return true;
        }
        if(front == rear + 1)
        {
            return true;
        }
        return false;

      }
```

```
bool isEmpty()
    {
        if(front == -1)
        {
            return true;
        }
        else
        {
            return false;
        }

      }
```


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

Applications of Circular Queue

- CPU scheduling
- Memory management
- Traffic Management



Josephus problem

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are skipped and kth person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

For example, if n = 5 and k = 2, then the safe position is 3. Firstly, the person at position 2 is killed, then person at position4 is killed, then person at position 1 is killed. Finally, the personat position 5 is killed. So the person at position 3 survives.

If n = 7 and k = 3, then the safe position is 4. The persons atpositions 3, 6, 2, 7, 5, 1 are killed in order, and person at position 4 survives.