# CS & IT ENGINEERING

## Data Structure

**Stack and Queues**
**Chapter- 4**
**Lec- 05**

By- Pankaj Sharma sir


Queue-I

# Queue

* Linear Data structure

— First In First Out

(i) Insertion : Rear

(ii) Deletion : Front

(i) Array implementation

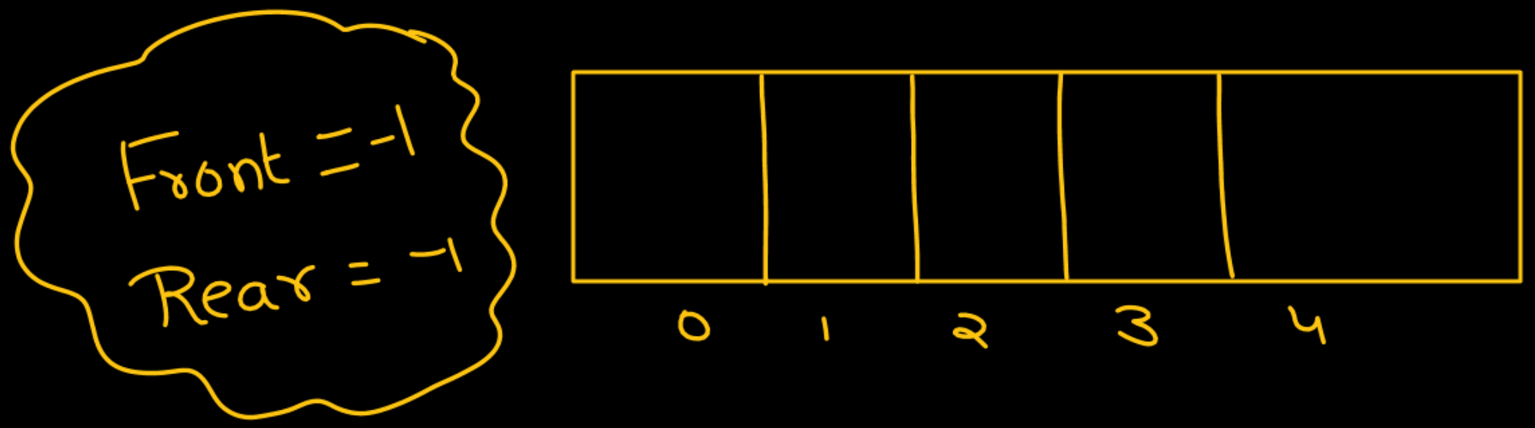
```
#define SIZE 5

int Queue[SIZE];
int Rear;
int Front;
```

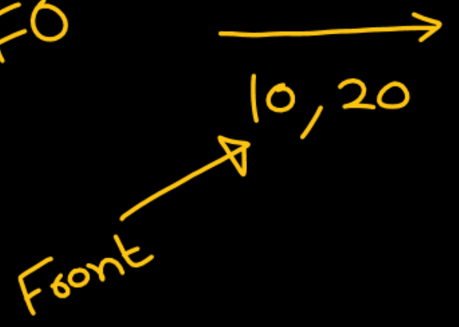
Front : index of element that can be deleted from Queue.

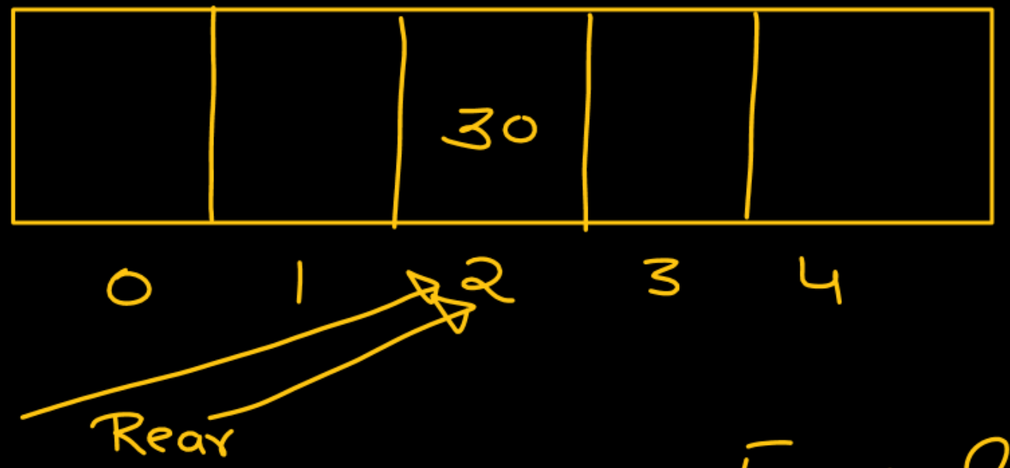
Rear : index of most recently added element.

Queue is empty



FIFO




| | Front | Rear |
|---|---|---|
| Initially | -1 | -1 |
| Insert(10) | 0 | 0 |
| Insert(20) | 0 | 1 |
| Insert(30) | 0 | 2 |
| Delete | 1 | 2 |
| Delete | 2 | 2 |

Same

Same

Same

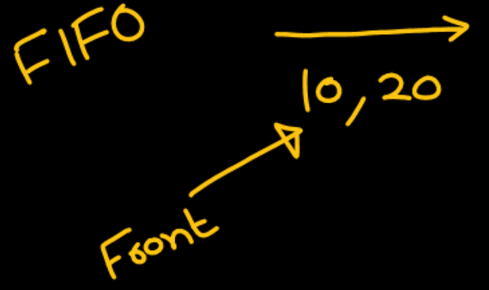
~~10, 20~~, 30

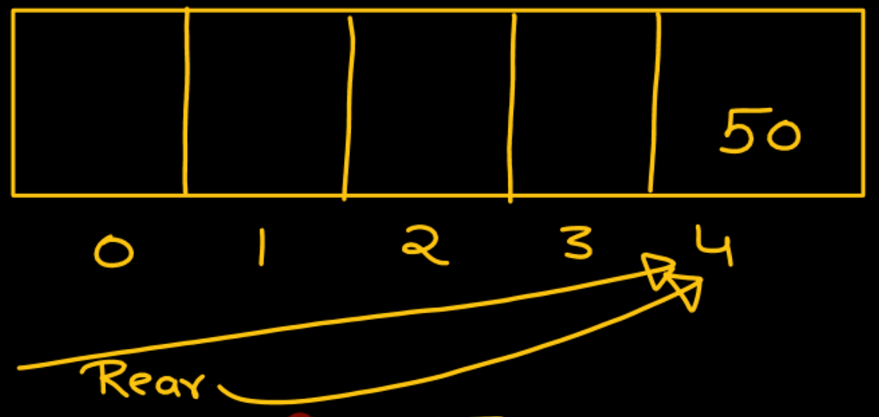
↳ For first element Front & Rear are same

Front

Case 1: When there is single ele. in queue
Front = Rear ≠ -1

Case 2: Queue is empty
Front = Rear = -1




| | Front | Rear |
|---|---|---|
| Initially | -1 | -1 |
| Insert(10) | 0 | 0 |
| Insert(20) | 0 | 1 |
| Insert(30) | 0 | 2 |
| Delete | 1 | 2 |
| Delete | 2 | 2 |
| Insert(40) | 2 | 3 |
| Insert(50) | 2 | 4 |
| Delete | 3 | 4 |
| Delete | 4 | 4 |

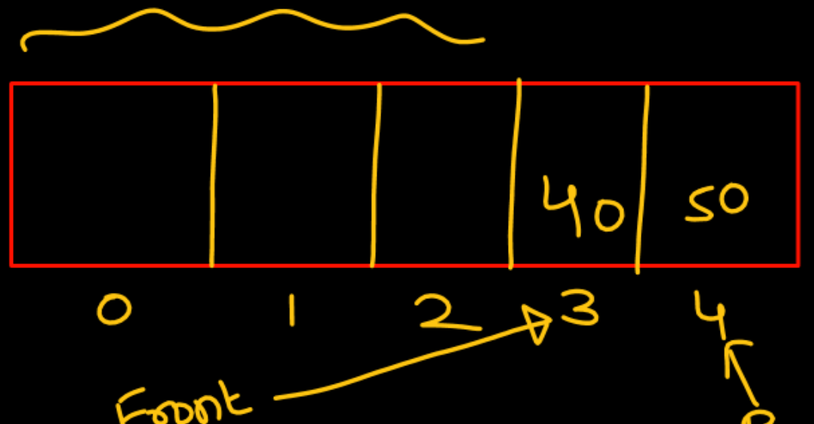
```
Rear = Rear + 1
Queue[Rear] = x ;
```

Delete

```
Front = Front + 1
```

```
Front = -1
Rear = -1
```

Queue is empty




Simple Queue

⇒ Circular Queue

| | Front | Rear |
|---|---|---|
| Initially | -1 | -1 |
| Insert(10) | 0 | 0 |
| Insert(20) | 0 | 1 |
| Insert(30) | 0 | 2 |
| Insert(40) | 0 | 3 |
| Inser(50) | 0 | 4 |

| Delete | Front | Rear |
|---|---|---|
| | 1 | 4 |
| Delete | 2 | 4 |
| Delete | 3 | 4 |

Insert(60) ⇒ bcz Rear == SIZE-1

Overflow

Insertion ⇒ Enqueue

Deletion ⇒ Dequeue

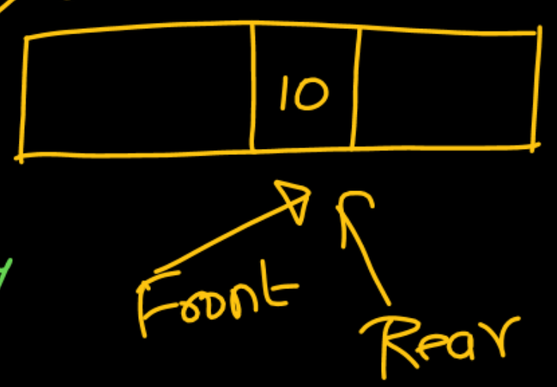
```
void Enqueue(int x){
    if(Rear == SIZE-1)
        {
        printf("overflow");
        return;
        }
    else if(Front == -1)
        Front = Rear = 0;
    else
        Rear = Rear+1;
    Queue[Rear] = x;
    }
```

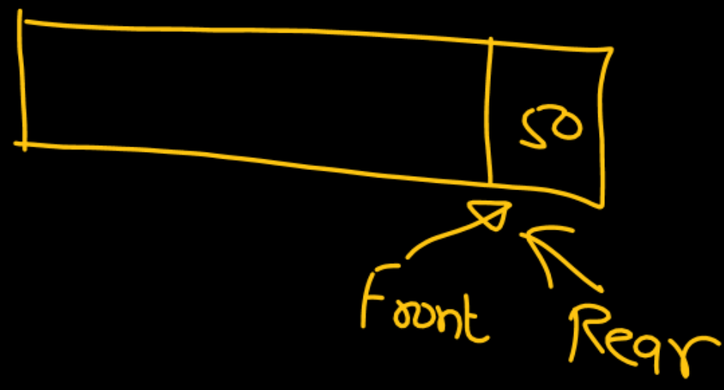
```
int Dequeue(){
    int temp;
    if( Front == -1)
    {
        return INT_MIN;
    }
    else if( Front == Rear)
    {
        temp = Queue[Front];
        Front = Rear = -1;
    }
    else
    {
        temp = Queue[Front];
        Front = Front +1;
    }
    return temp;
}
```

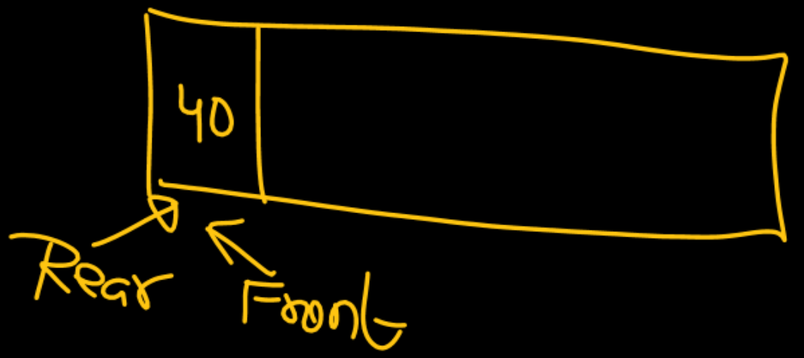
Constant time

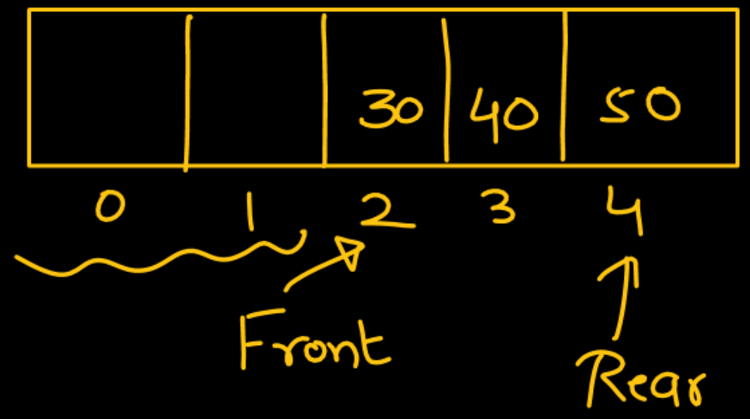
Queue is empty

When the Queue is not empty

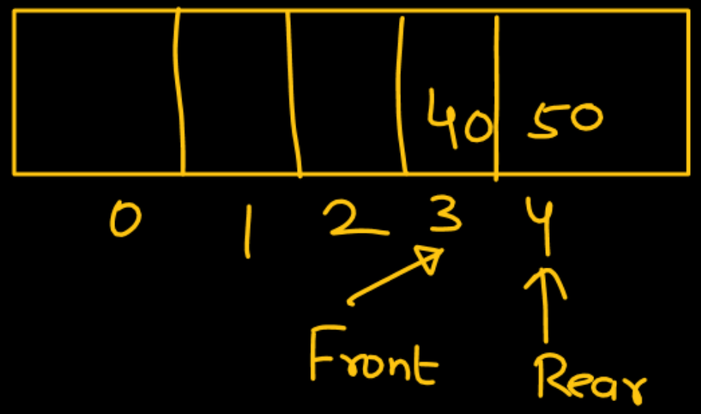




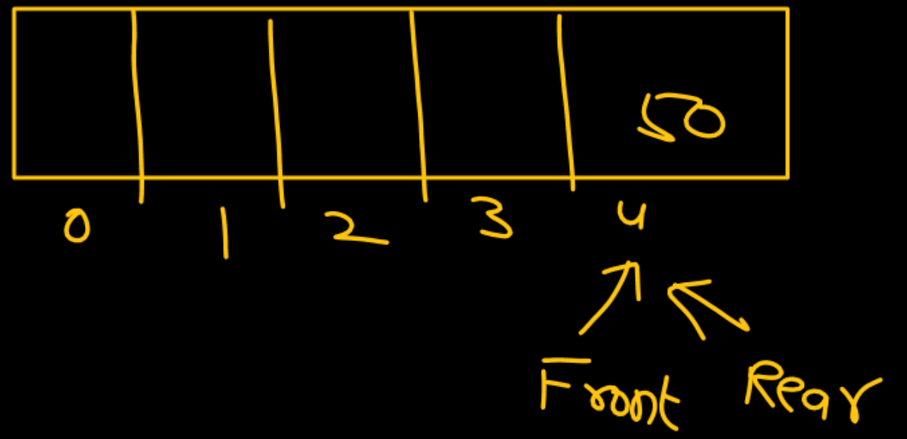

delete

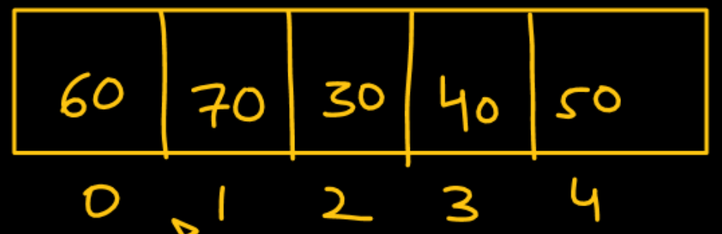



Overflow

worst case

delete

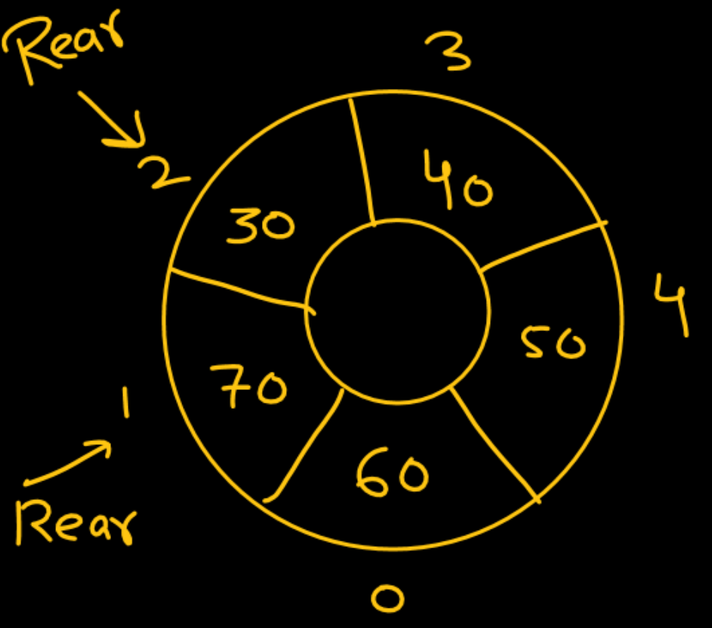





Full

Case1

Front == Rear + 1




Insert 60 $\Rightarrow$ Rear = 0

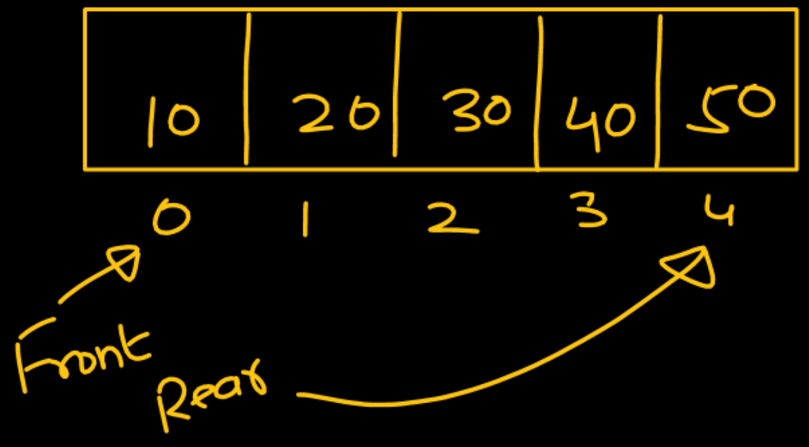
Queue[Rear] = 60

Insert 70 $\Rightarrow$ Rear++

Queue[Rear] = 70

Front = -1
Rear = -1

| | Front | Rear |
|---|---|---|
| Insert(10) | 0 | 0 |
| Insert(20) | 0 | 1 |
| Insert(30) | 0 | 2 |
| Insert(40) | 0 | 3 |
| Insert(50) | 0 | 4 |




4+1 = 5 mod 5

= 0

Full

(i) (Front = 0) && (Rear == SIZE-1)

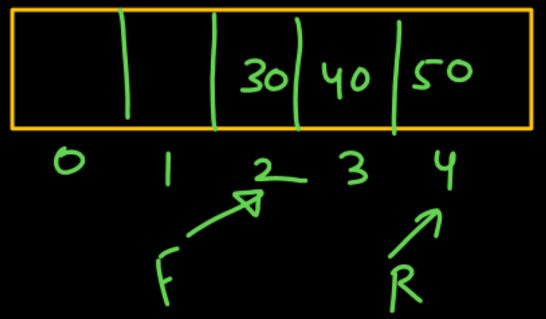
if (Front == (Rear+1) mod SIZE)

```
void CQ_Enqueue(int x)
{
    if (((Front == 0) && (Rear == size-1))
                ||
        (Front == Rear+1))
    {
        printf("overflow")
        return;
    }
}
```

```
void CQ_Enqueue(int x)
{
  if ( Front == (Rear+1) % SIZE)
                return;                    //overflow
  else if ( Rear == SIZE-1 )
                Rear = 0;
  else if ( Front == -1)
                Rear = Front = 0;
  else
                Rear++;
       Queue[Rear] = x;
}
```

CQ is not full

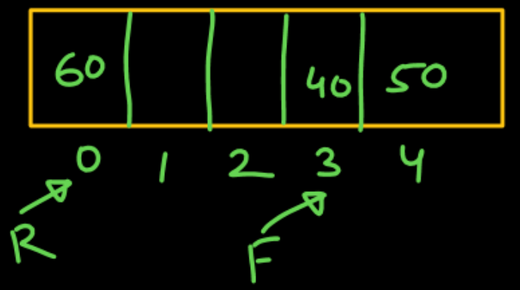
## Deletion

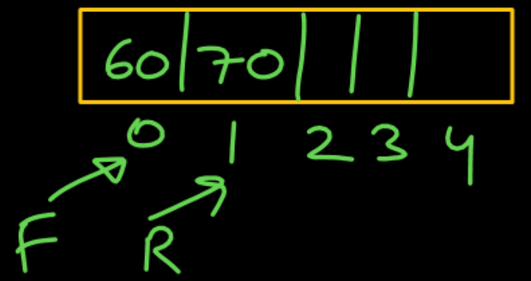



1) Insert 60


~~30~~, 40, 50, 60

2) Delete 40, 50, 60

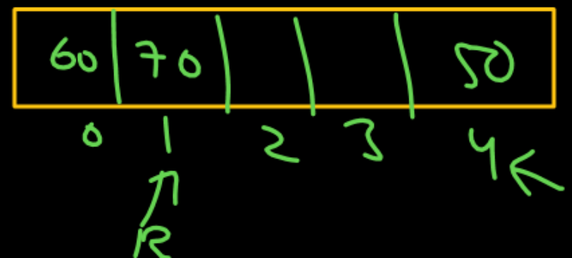



3) Insert 70 ~~40~~, 50, 60, 70


delete

4) Delete




~~60~~, 60, 70

int ~~void~~ CQ_Deletion

1) if (Front == -1)

Underflow

return

2) Special case Front == Rear

1 elem in Queue

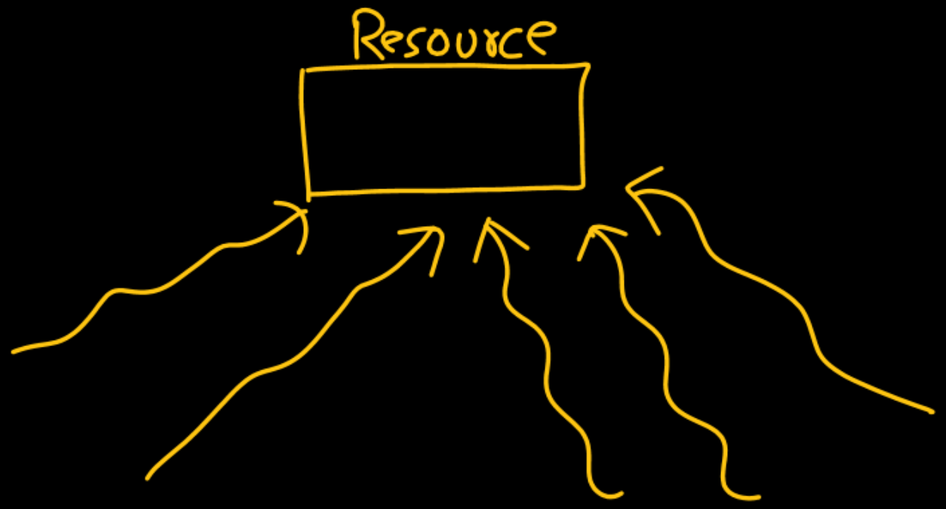
Front = Rear = -1

3) Special case Front == SIZE-1

⇒ front = 0

4) Front ++ ;

# Application




(i) CPU scheduling

(ii) Spooling

(iii) Slow & fast device ⇒ Synchronization

(iv)

- Double Ended Queue
- Priority Queue


THANK YOU GW
SOLDIERS !