Queue:

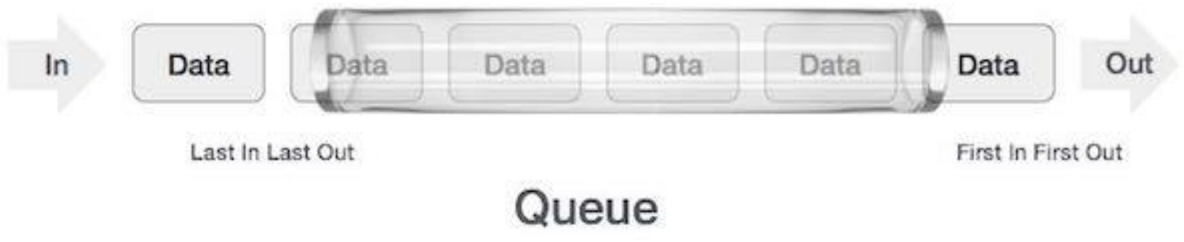
Queue is an abstract data structure, somewhat similar to Stacks.

Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

Queue follows First-In-First-Out(FIFO) methodology Or First Come First Serve(FCFS), i.e., the data item stored first will be accessed first.

the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head).

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.



There are four types of Queue:

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

Basic Operations

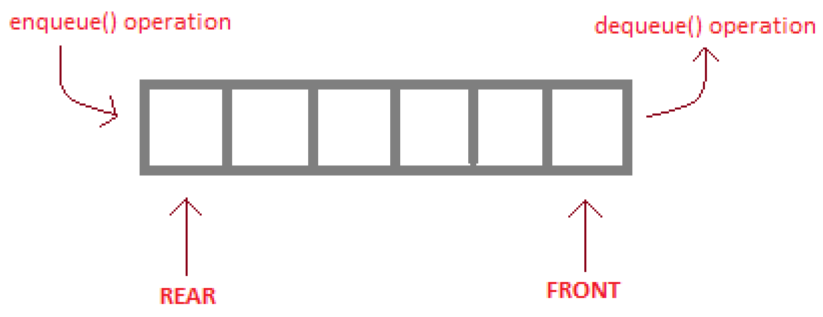
the basic operations associated with queues –

- enqueue() – add (store) an item to the queue.

- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- 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.

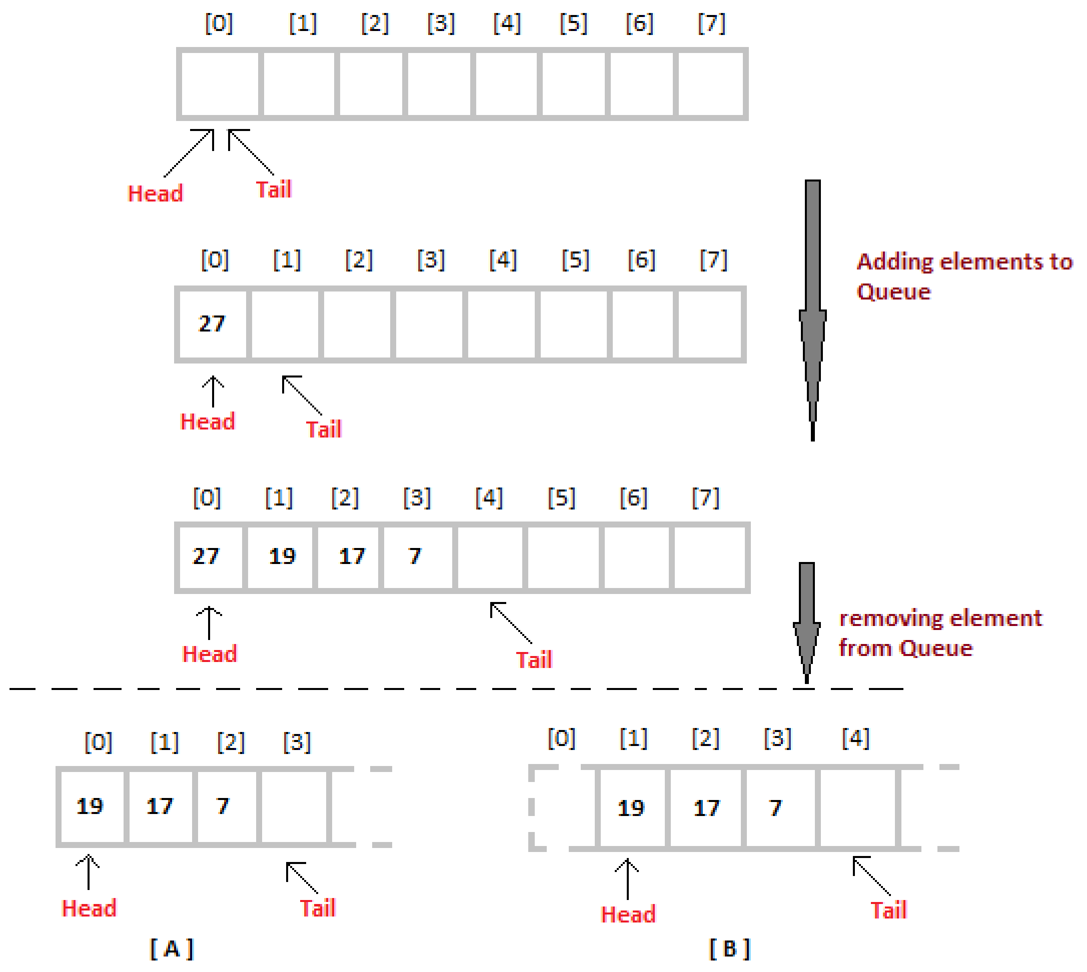


enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

<u>QUEUE DATA STRUCTURE</u>

Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



Algorithm for Simple Queue :

Algorithm for ENQUEUE operation

```
Step1: [ Check if the queue is full or not. ]
    If r=n-1
    Then write("queue overflow")
    Return
Step2: [increment the REAR pointer]
    r=r+1
step3: [ insert the element ]
    q[r]=data
step4: Is front pointer properly set
        if f=-1 then
        set f=0
step5: [finished ]
        return
```

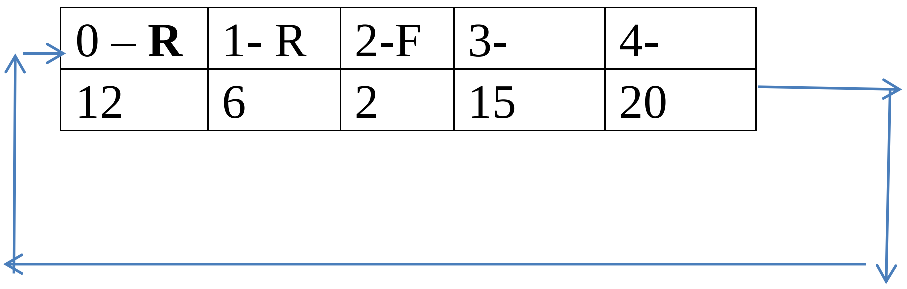
Algorithm for DEQUEUE operation

```
Step1: [ check for underflow on queue ]
    If f== -1 then
    Write ("Queue is empty")
    Return

Step2: [ assign top element of queue to data ]
    Data=q[f]

Step3: [Set Pointers]
    If F==R
    Then F = R = -1
else
    F=F+1

step 4: [return the front element of queue]
    return data
```

<u>**Circular queue Insert data**</u>

<u>**N= 5 (n-1 -> 5-1 = 4)**</u>

<u>**R=f=-1 1+1=2**</u>



0 – **R**	1- R	2-F	3-	4-
12	6	2	15	20

```
Step1: [ Check if the queue is full or not. ]
If ((r==n-1 && f==0)||(r+1==f))
    Then write("queue overflow")
  Return
```

```
Step2: [ Reset the rear element ]
    if(r==n-1)
        r=0;
    else
        r++;

step3: [insert element in queue and set the front pointer]
    q[r]=data;
    if(f==-1)
        f=0;
  step4: [finished]
    return
```

delete data

n=7   n-1 =6

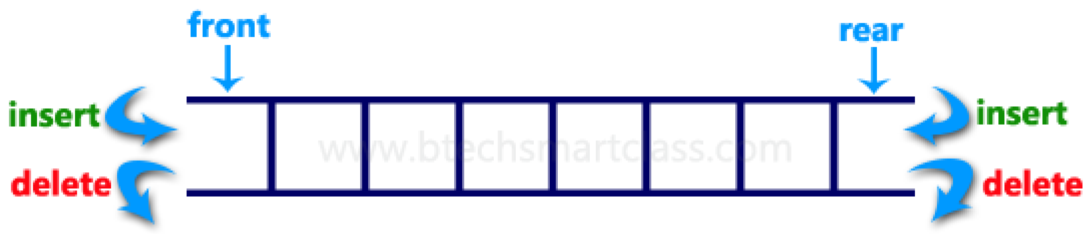
0-	1-r	2f	3	4	5	6-n-1
15	20	30	40	50	60	70

```
Step1: [ check for underflow on queue ]
    If f== -1 then
    Write ("Queue is empty")
    Return
Step2: [ assign top element of queue to data ]
    Data=q[f]

Step3: [ Increment pointer ]
    if(f==r)
        r=f=-1;
    else if(f==n-1)
        f=0;
    else
```

```
        f=f+1
step 4: [return the front element of queue]
    return data
```

Double Ended Queue (Dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

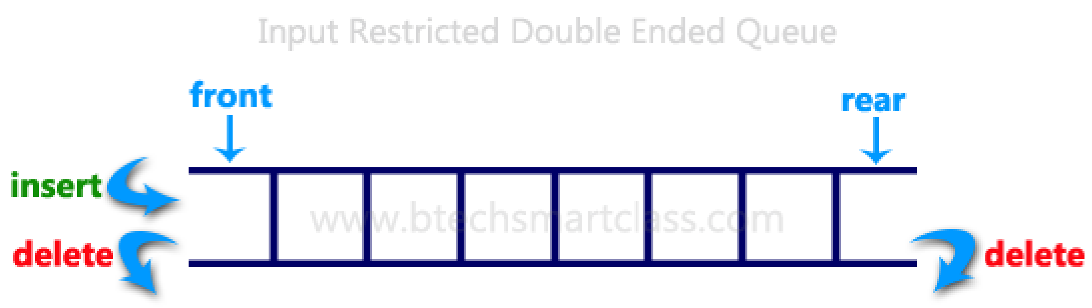
1. Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.

Algorithms :

1. Insert from Right

2. Delete from right
3. Delete from Left

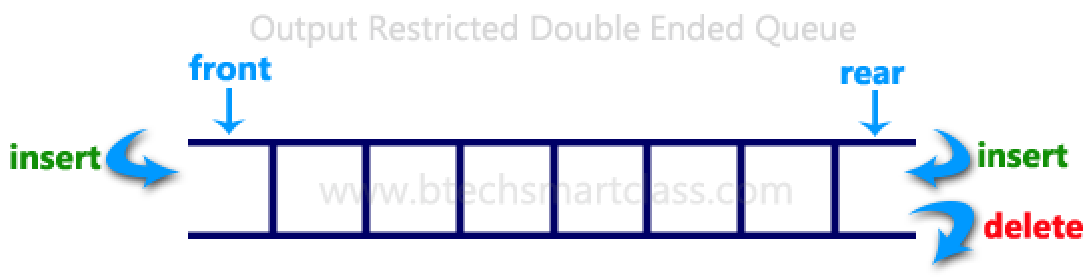


2. Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.

Algorithms:

1. Insert from left
2. Insert from Right
3. Delete from left



Algorithms:

Insert from Right(regular)

0-r,f	**1-**	**2-**	**3-**
20			

n-4 -> n-1=4-1=3   f=r=-1

```
Step1: [ Check if the queue is full or not. ]
    If r=n-1
    Then write("queue overflow")
    Return

Step2: [increment the REAR pointer]
    r=r+1

step3: [ insert the element ]
    q[r]=data

step4: if f=-1 then
        set f=0

step5: [finished ]
        return
```

Delete from Left(regular)

```
Step1: [ check for underflow on queue ]
        If f== -1 then
```

```
    Write (“Queue is empty”)
    Return
```

Step2: [assign top element of queue to data]

```
    Data=q[f]
```

Step3: [Increment FRONT pointer]

```
    f=f+1
```

step 4: [return the front element of queue]

```
    return data
```

<u>Insert from Left</u>

<u>F=n-1</u>

<u>0-f</u>	**<u>1-f</u>**	**<u>2-f</u>**	**<u>3-f</u>**	**<u>4,f,r</u>**
<u>12</u>	**<u>15</u>**	**<u>20</u>**	**<u>5</u>**	**<u>10</u>**

Step1: [Check if the queue is full or not.]

```
  If f=0
  Then write(“queue overflow”)
  Return
```

Step2: [set the Front and Rear pointer]

```
  If f==-1 then
    F=n-1
    R=n-1
Else
    F=F-1
```

step3: [insert the element]

```
  q[F]=data
```

step4: [finished]

```
    return
```

Delete from Right

R=-1

0-	1-r	2-	3-	4-

```
Step1: [ check for underflow on queue ]
    If r== -1 then
    Write ("Queue is empty")
    Return

Step2: [ assign top element of queue to data ]
    Data=q[r]

Step3: [ Increment pointer ]
    r=r-1

step 4: [return the front element of queue]
    return data
```

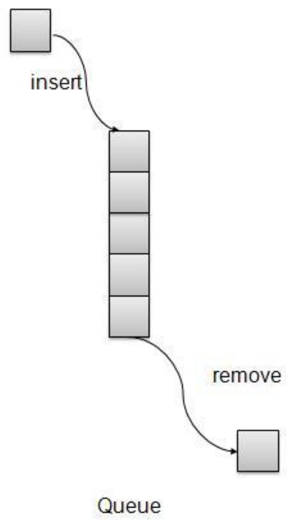
Priority Queue

- Priority Queue is more specialized data structure than Queue.
- Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.
- So we're assigned priority to item based on its key value.
- Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.
- **remove / dequeue** – remove an item from the front of the queue.

Priority Queue Representation

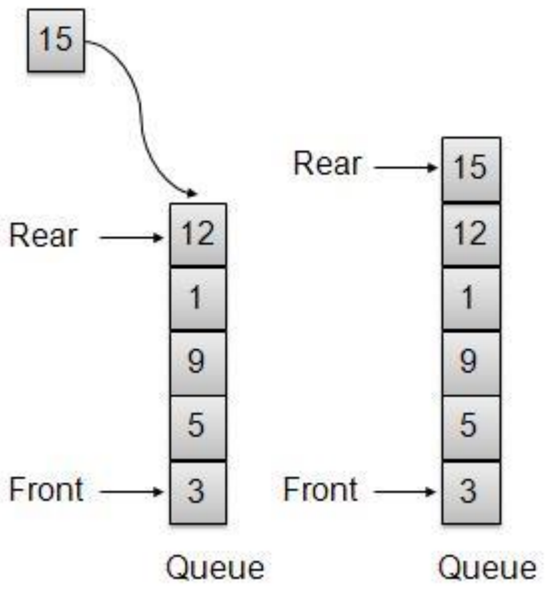


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

. Applications of Queue Data Structure

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
   2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
2) Simulation