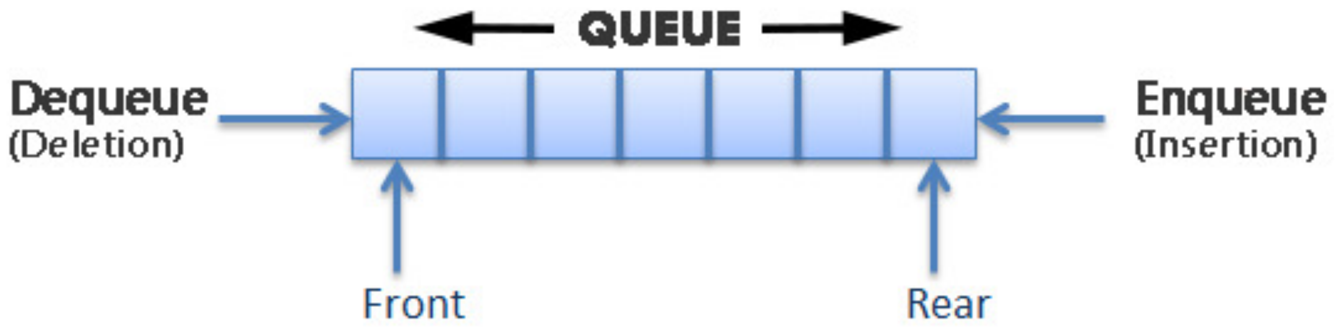
Queue

- **Queue** is a non-primitive linear data structure which stores data in **FIFO** manner where **FIFO** stands for **First-In-First-Out** .
- In queue, insertion of new value is done at **rear** and deletion of existing value is done from **front** of queue.



Queue

Implementation of Queue:

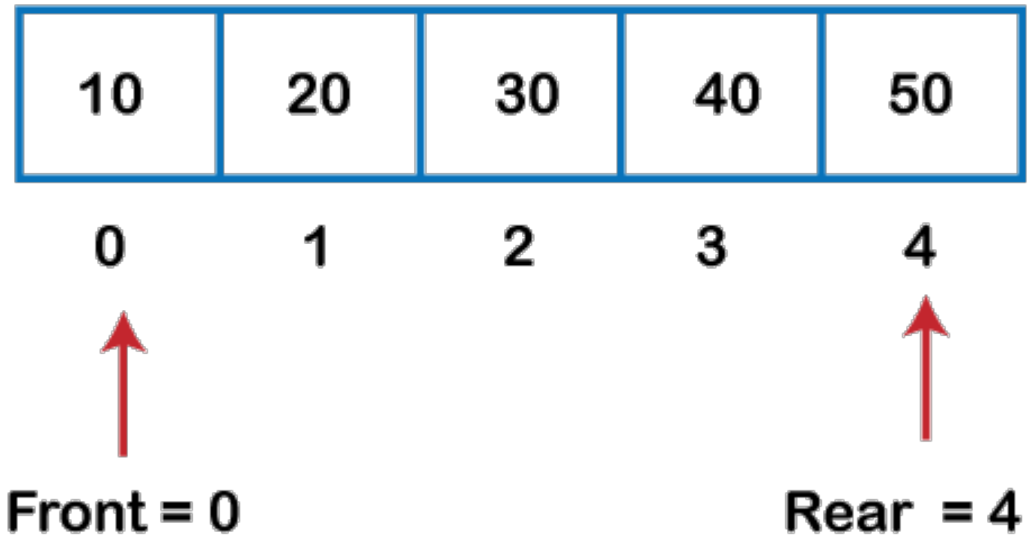
1. **Array Queue:** Queue implemented using array is called array queue.
2. **Linked Queue:** Queue implemented using linked list is called linked queue.

Operations on Queue:

1. **Enqueue:** Adding new element to the rear of queue.
2. **Dequeue:** Removing the existing element from front of queue.
3. **Traverse:** Visiting every element from front to rear of queue.

Array Queue

Queue can be implemented using array. In this implementation of queue, we will define a fixed size array and new value must be added at the end of array and existing value must be deleted from front of array. In this implementation of array value can't be added or deleted from any other position in array. This type of array implementation is called **array queue**.



Array Queue

Algo to Enqueue in Array Queue

Here **queue** is the array that will act as queue, **front** is initially having value **0** and will store the index of the first value to be enqueued into the array queue, **rear** is initially having value **0** and will store the index of the last value to be to be enqueued into the array queue, **capacity** will specify the maximum values array queue can store, and **value** will store the value to be enqueued into array queue, then the algorithm is given below:

```
1. Check, if rear == capacity, then:
    a. Print, queue is full.
2. Otherwise,
    a. Read value.
    b. Set queue[rear] = value.
    c. Set rear = rear + 1.
    d. Print, value is inserted.
3. Exit.
```

Array Queue

Algo to Traverse the Array Queue

Here **queue** is the array that is acting as queue, **front** is having the index of first value enqueued in array queue and **rear** is having the index of last value enqueued in array queue, **i** will store the index of the element to be traversed one by one in array queue, then the algorithm is given below:

```
1.  Check, if front != rear, then:
      a.  Repeat for i = front to rear
              i.    Print, queue[i].
2.  Otherwise,
      a.  Print, no value to traverse.
3.  Exit.
```

Array Queue

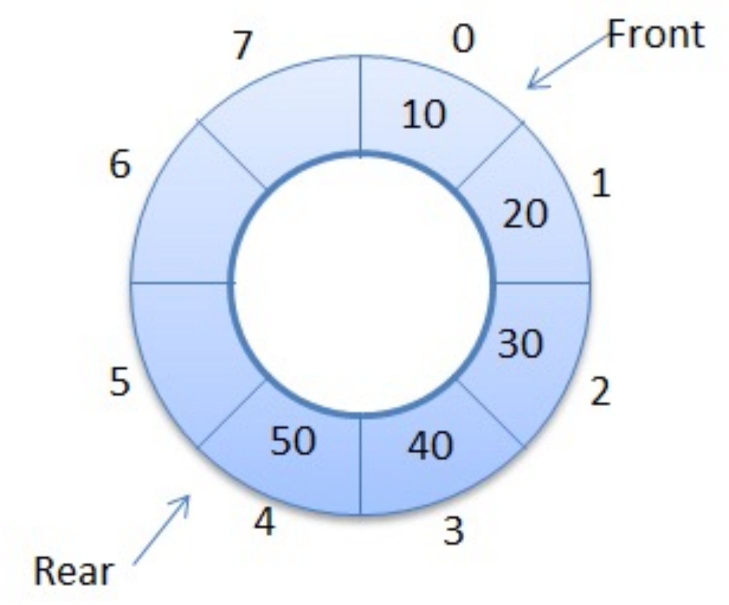
Algo to Dequeue from Array Queue

Here **queue** is the array that is acting as queue, **front** is having the index of first value enqueued in array queue and **rear** is having the index of last value enqueued in array queue, **value** will store the value to be dequeued from array queue, then the algorithm is given below:

```
1. Check, if front != rear, then:
    a. Set value = queue[front].
    b. Set front = front + 1.
    c. Print, value is deleted.
2. Otherwise,
    a. Print, queue is empty.
3. Exit.
```

Circular Queue

A circular queue is one in which enqueue is done at the very first location of the queue if the last location of the queue is full. We can say that a circular queue is one in which the first element comes just after the last element. It can be consider as a loop of wire in which the two ends of the wire are connected together. A circular queue overcomes the problem of unutilized space in simple array queue.



Circular Queue

Algo to Enqueue in Circular Queue

Here **queue** is the array that will act as circular queue, **front** is initially having value **0** and will store the index of the first value enqueued into the circular queue, **rear** is initially having value **0** and will store the index of the last value enqueued in the circular queue, **capacity** will specify the maximum values circular queue can store, and **value** will store the value to be enqueued into circular queue, then the algorithm is given below:

```
1. Check, if ( front = 0 and rear = capacity) or (front == rear  && front != 0), then:
     a. Print, circular queue is full.
2. Otherwise,
     a. Read value.
     b. Check, if rear == capacity, then:
          i. Set rear  = 0.
     c. Set queue[rear] = value.
     d. Set rear  = rear + 1
     e. Print, value is inserted.
3. Exit.
```

Circular Queue

Algo to Traverse the Circular Queue

Here **queue** is the array that is acting as circular queue, **front** is having the index of first value enqueued in circular queue and **rear** is having the index of last value enqueued in circular queue, **i** will store the index of the element to be traversed one by one in circular queue, then the algorithm is given below:

```
1. Check, if front==0 and rear==0, then:
    a. Print, no value to traverse.
2. Otherwise,
    a. Check, if front<rear, then:
        i. Repeat for i = front to rear
            • Print, queue[i].
    b. Otherwise,
        i. Repeat for i = front to capacity
            • Print, queue[i].
        ii. Repeat for i = 0 to rear
            • Print, queue[i].
3. Exit.
```

Circular Queue

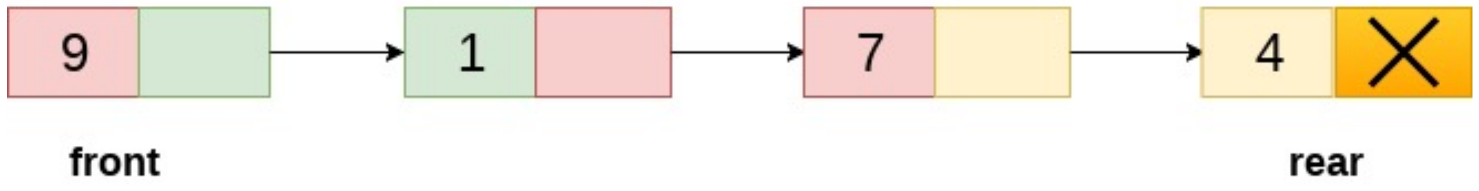
Algo to Dequeue from Circular Queue

Here **queue** is the array that is acting as circular queue, **front** is having the index of first value enqueued in circular queue and **rear** is having the index of last value enqueued in circular queue, then the algorithm is given below:

```
1. Check, if front==0 and rear==0, then:
     a. Print, no value to traverse.
2. Otherwise,
     a. if front = capacity, then:
          i. Set front = 0.
     b. Set front = front + 1
     c. Print, value is deleted.
3. Exit.
```

Linked Queue

Queue can be implemented using linked list. In this implementation of queue, new node must be added at end of linked list and existing node must be deleted from begin of linked list. In this implementation of queue using linked list, node can’t be added other than end of linked list and node can’t be deleted from other than begin of linked list. This type of queue implementation using linked list is called **linked queue**.



Linked Queue

Linked Queue

Algo to Enqueue in Linked Queue

Here **front** is node pointer initially having value **NULL** and will point to the first node, **rear** is node pointer initially having value **NULL** and will point to the last node, **newNode** is also the node pointer that will point to the newly created node and **value** will store the value to be stored in the data part of the newly created node to be enqueued in linked queue, then the algorithm is given below:

```
1. Create newNode with given value.
2. Check, if front = NULL, then:
          i.    Set front=rear=newNode.
3. Otherwise,
     a. Set rear->next = newNode .
     b. Set rear= newNode.
4. Print, value is inserted.
5. Exit.
```

Linked Queue

Algo to Traverse the Linked Queue

Here **front** is node pointer pointing to the first node, **rear** is node pointer pointing to the last node, **currentNode** is also the node pointer that will point from first node to last node one by one to be visited in linked queue, then the algorithm is given below:

```
1.  Check, if front != NULL, then:
     a.  Set currentNode = front.
     b.  Repeat while currentNode != NULL
          i.   Print, currentNode ->data.
          ii.  Set currentNode = currentNode ->next.
2.  Otherwise,
     a.  Print, no value to traverse.
3.  Exit.
```

Linked Queue

Algo to Dequeue from Linked Queue

Here **front** is node pointer pointing to the first node, **rear** is node pointer pointing to the last node, and **value** will store the value of node to be to be dequeued from linked queue, then the algorithm is given below:

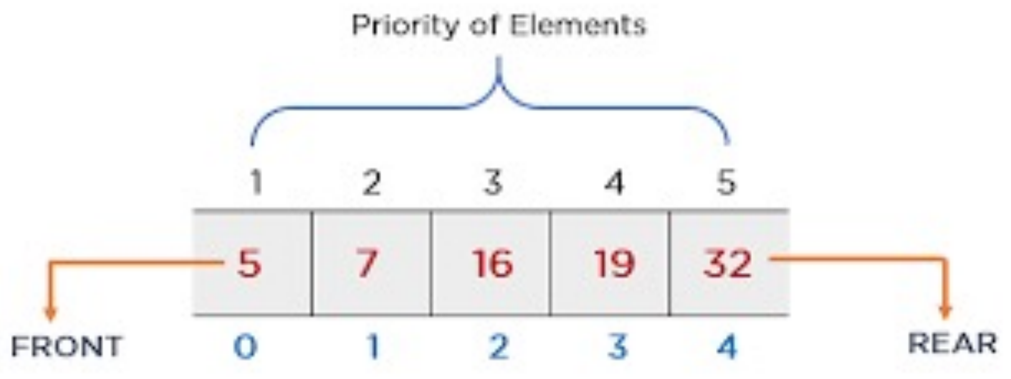
```
1. Check, if front != NULL, then:
    a. Set front = front -> next.
    b. Print, value is deleted.
2. Otherwise,
    a. Print, no value to traverse.
3. Exit.
```

Priority Queue

A priority queue is a collection of elements where the elements are stored according to their priority levels. The order in which the elements should get added or removed is decided by the priority of the element. Following rules are applied to maintain a priority queue.

(a) The element with a higher priority is processed before any element of lower priority.

(b) If there are elements with the same priority, then the element added first in the queue would get processed.


Priority of Elements
1
2
3
4
5
5
7
16
19
32
FRONT
0
1
2
3
4
REAR