0Types of Queues:

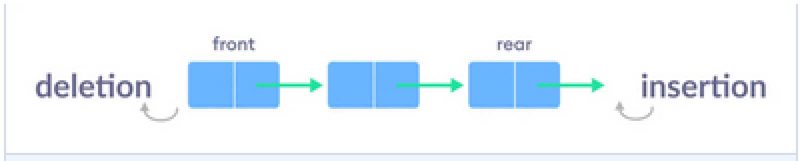
A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

Simple Queue:

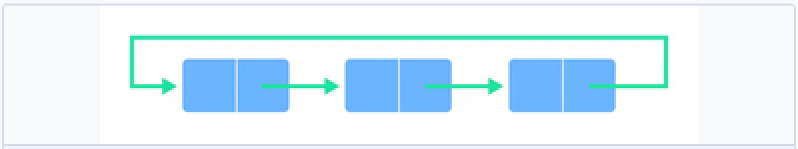
In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



Simple Queue Representation

Circular Queue:

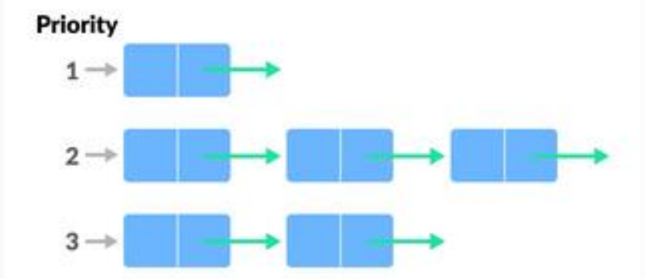
In a circular queue, the last element points to the first element making a circular link.


Circular Queue Representation

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

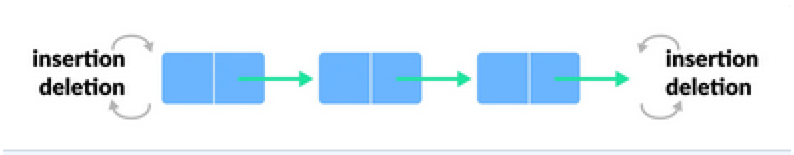


Priority Queue Representation

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Deque (Double Ended Queue):

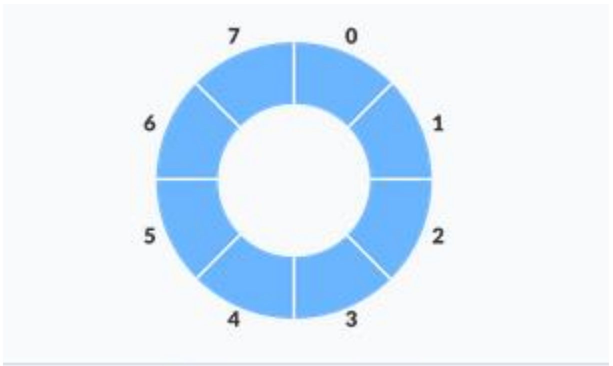
In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



Deque Representation

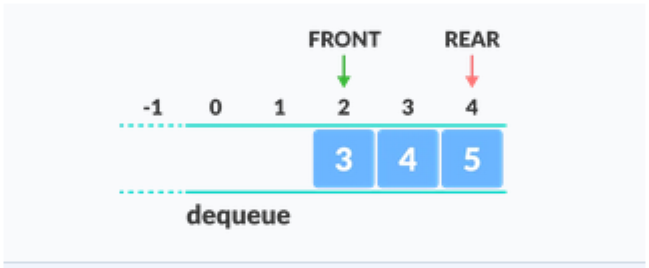
Circular Queue Data Structure:

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus, forming a circle-like structure.



Circular queue representation

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Limitation of the regular Queue

Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

<u>How Circular Queue Works:</u>

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.

<u>Circular Queue Operations:</u>

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. <u>Enqueue Operation:</u>

- 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*

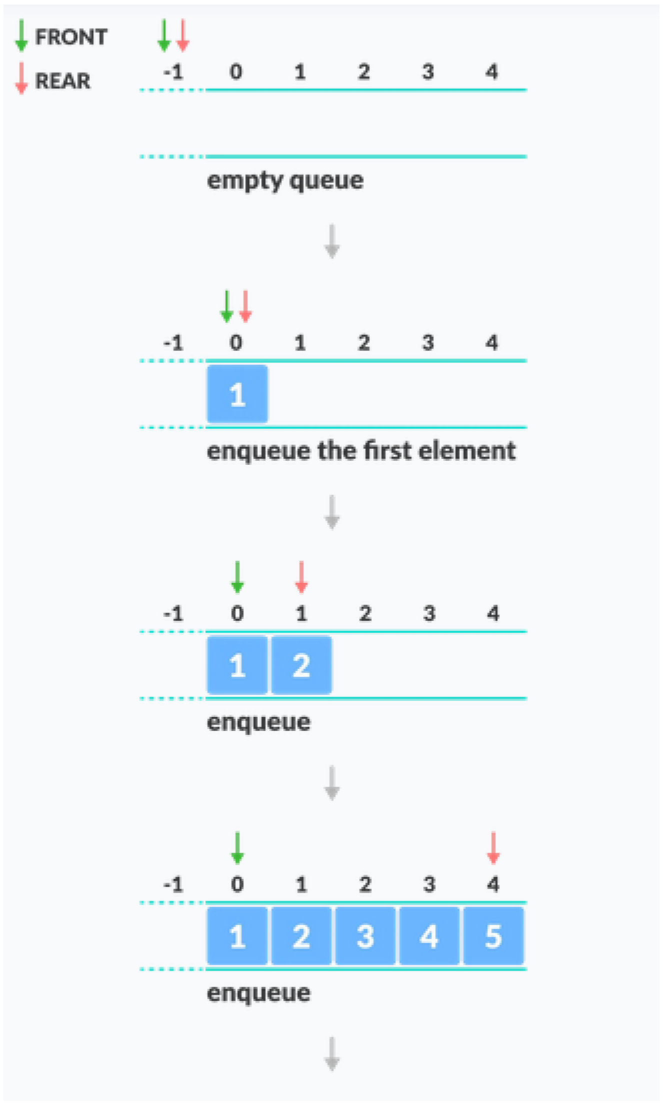
2. Dequeue Operation:

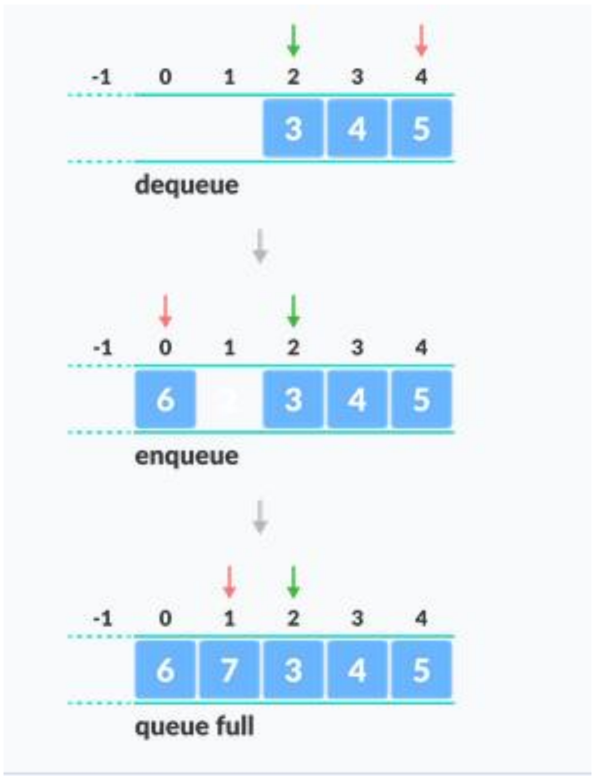
- check if the queue is empty
- return the value pointed by *FRONT*
- circularly increase the *FRONT* index by 1
- for the last element, reset the values of *FRONT* and *REAR* to -1

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

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

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.


FRONT
REAR
-1 0 1 2 3 4
empty queue
1
enqueue the first element
1 2
enqueue
1 2 3 4 5
enqueue



Enque and Deque Operations

The complexity of the enqueue and dequeue operations of a circular queue is *O(1)* for (array implementations).

Applications of Circular Queue:

- CPU scheduling
- Memory management
- Traffic Management

Priority Queue:

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

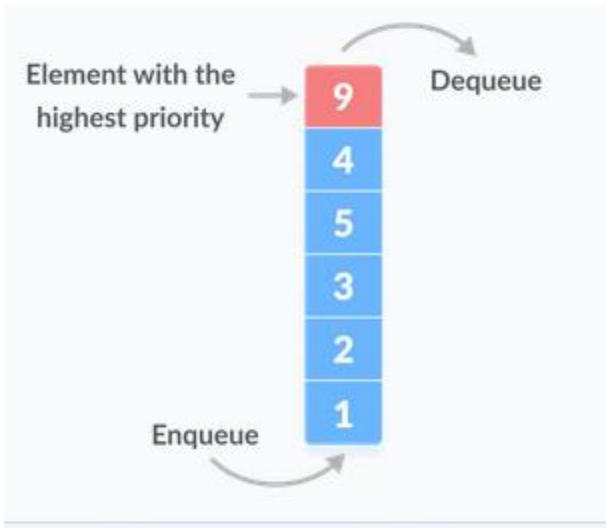
However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value:

Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



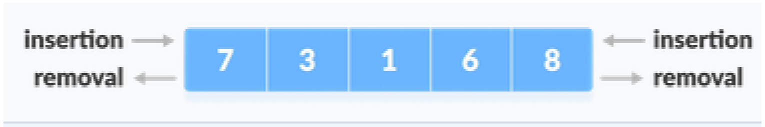
Removing Highest Priority Element

Difference between Priority Queue and Normal Queue:

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

Deque Data Structure:

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).



Representation of Deque

Types of Deque:

- **Input Restricted Deque**
  In this deque, input is restricted at a single end but allows deletion at both the ends.
- **Output Restricted Deque**
  In this deque, output is restricted at a single end but allows insertion at both the ends.

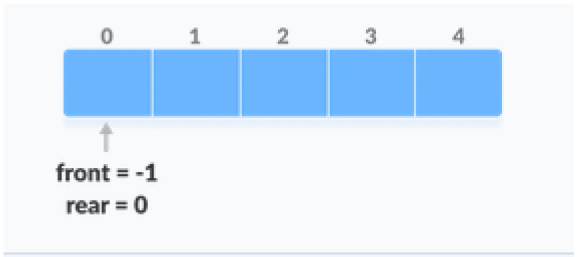
Operations on a Deque:

Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.

But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

1. Take an array (deque) of size *n*.
2. Set two pointers at the first position and set front = -1 and rear = 0.

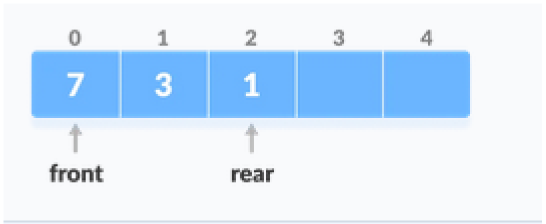


Initialize an array and pointers for deque

<u>1.Insert at the Front:</u>

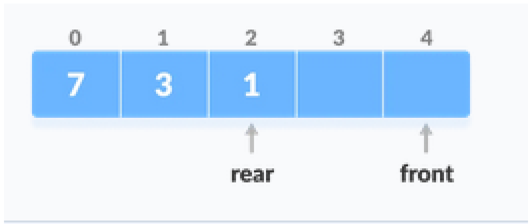
This operation adds an element at the front.

1. Check the position of front.



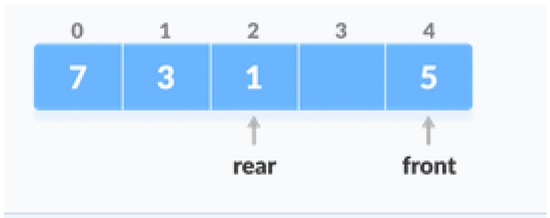
Check the position of front

2. If front < 1, reinitialize front = n-1 (last index).



Shift front to the end

3. Else, decrease *front* by 1.
4. Add the new key *5* into array[front].

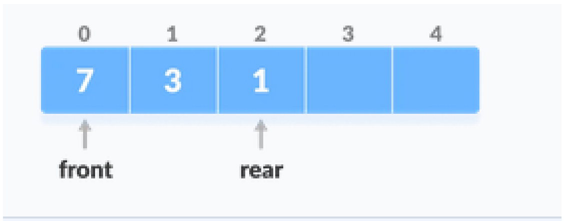


Insert the element at Front

2.Insert at the Rear:

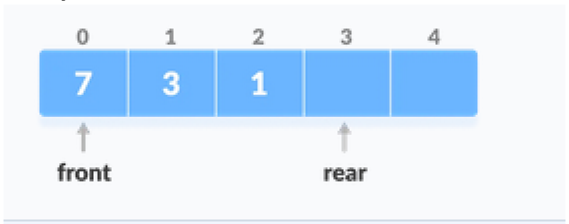
This operation adds an element to the rear.

1. Check if the array is full.



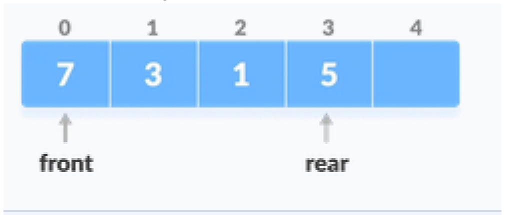
Check if deque is full

2. If the deque is full, reinitialize rear = 0.
3. Else, increase *rear* by 1.



Increase the rear

4. Add the new key *5* into array[rear].

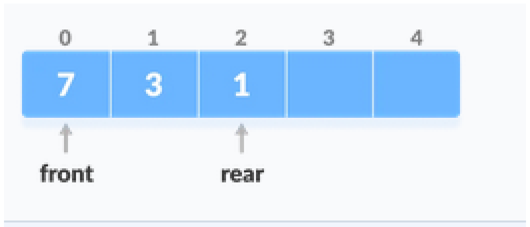


Insert the element at rear

3. Delete from the Front:

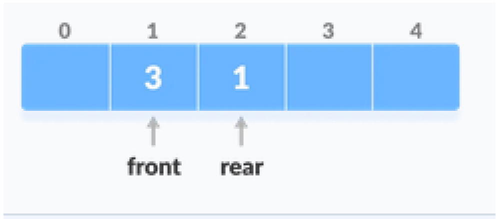
The operation deletes an element from the front.

1. Check if the deque is empty.



Check if deque is empty

2. If the deque is empty (i.e. front = -1), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1.
4. Else if *front* is at the end (i.e. front = n - 1), set go to the front front = 0.
5. Else, front = front + 1.

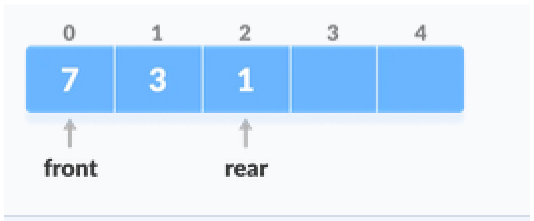


Increase the front

4. Delete from the Rear:

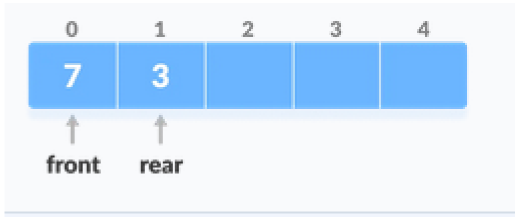
This operation deletes an element from the rear.

1. Check if the deque is empty.



Check if deque is empty

2. If the deque is empty (i.e. front = -1), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1, else follow the steps below.
4. If *rear* is at the front (i.e. rear = 0), set go to the front rear = n - 1.
5. Else, rear = rear - 1.



Decrease the rear

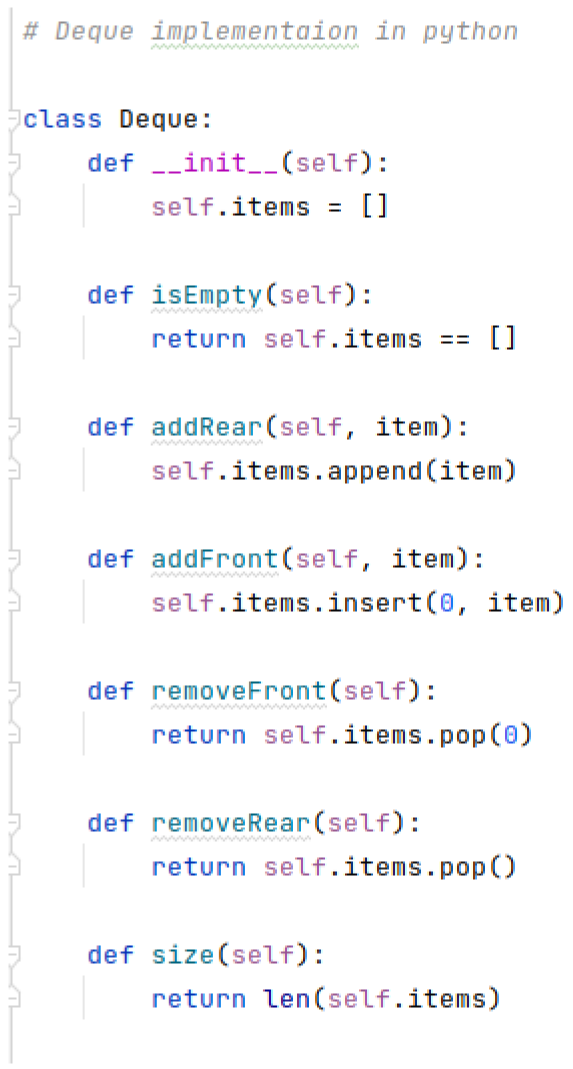
5.Check Empty:

This operation checks if the deque is empty. If front = -1, the deque is empty.

6.Check Full:

This operation checks if the deque is full. If front = 0 and rear = n - 1 OR front = rear + 1, the deque is full.

Deque Implementation:



```python
# Deque implementaion in python

class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addRear(self, item):
        self.items.append(item)

    def addFront(self, item):
        self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop(0)

    def removeRear(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

```
d = Deque()
print(d.isEmpty())
d.addRear(8)
d.addRear(5)
d.addFront(7)
d.addFront(10)
print(d.size())
print(d.isEmpty())
d.addRear(11)
print(d.removeRear())
print(d.removeFront())
d.addFront(55)
d.addRear(45)
print(d.items)
```

Output:

```
True
4
False
11
10
[55, 7, 8, 5, 45]
```

The time complexity of all the above operations is constant i.e. O(1).

Applications of Deque Data Structure:

1. In undo operations on software.
2. To store history in browsers.
3. For implementing both stacks and queues.