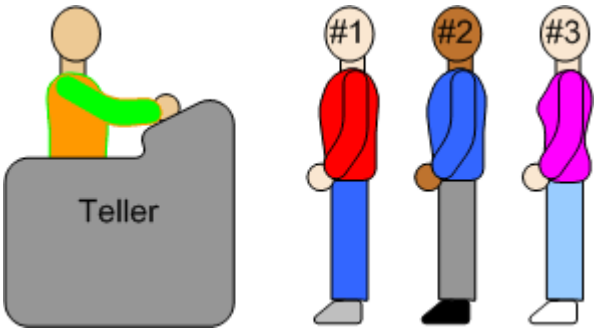
Introduction to queues

- Queues are simple data structures in which insertions are done at one end and deletions are done at the other end.
- It is a linear data structure as arrays.
- It is an abstract data type **(ADT)**.
- The first element to be inserted is the first element to be deleted. It follows either **FIFO (First in First Out)** or **LILO (Last In Last Out)**.
- Consider the queue as the line of people.

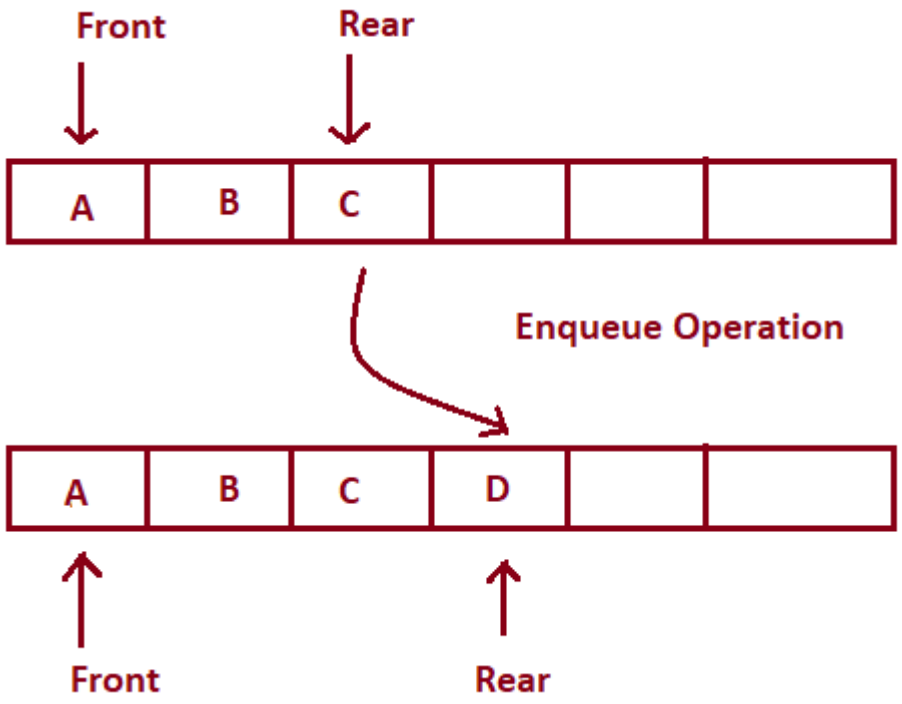


The above picture shows that when we enter the queue/line, we stand at the end of the line and the person who is at the front of the line is the one who will be served first.

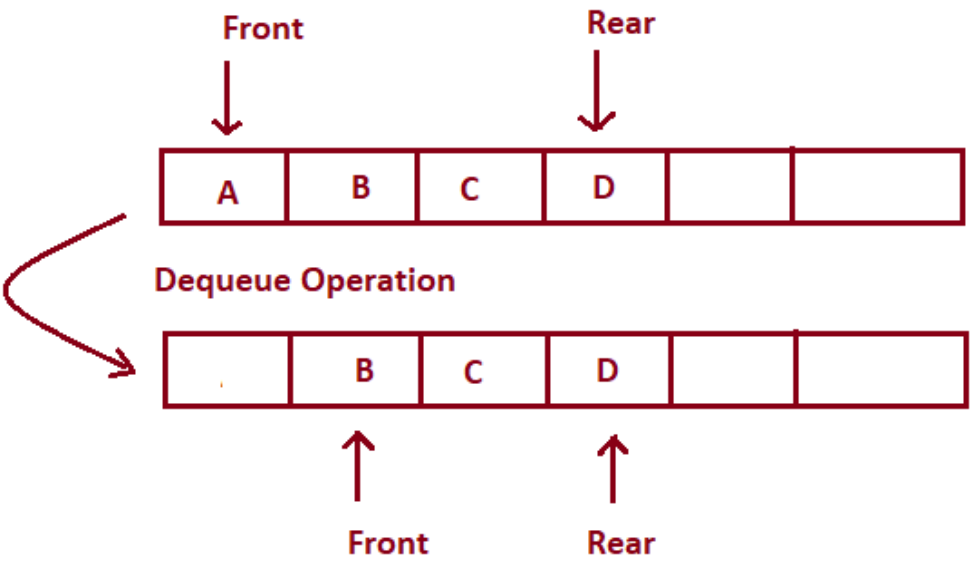
After the first person is served, he will exit, and as a result, the next person will come at the head of the line. Similarly, the head of the queue will keep exiting the line as we move towards the front/head of the line. Finally, we reach the front. we are served and we exit. This behavior is useful when we need to **maintain the order of arrival**.

Various operations on queues

In queues, insertion is done at one end (rear) and deletion is done at other end (front) :



Enqueue Operation



Dequeue Operation

- **Insertion**: Adding elements at the rear side. The concept of inserting an element into the queue is called Enqueue. Enqueueing an element, when the queue is full, is called **overflow**.

- **Deletion**: Deleting elements at the front side. The concept of deleting an element from the queue is called Dequeue. Deleting an element from the queue when the queue is empty is called **underflow**.

Main Queue Operations:

- enqueue(data) : Insert data in the queue (at rear).
- dequeue() : Deletes and returns the first element of the queue (at front).

Auxiliary Queue Operations:

- front() : returns the first element of the queue.
- int size() : returns number of elements in the queue.
- boolean isEmpty() : returns whether the queue is empty or not.

How can queues be implemented?

Queues can be implemented using arrays, linked lists or stacks. The basic algorithm for implementing a queue remains the same.

We will use array-based implementation here. We maintain two variables for all the operations: **front, rear**.

- **Enqueue Operation**

```
function enqueue(data)
        if queue is full
        return “Full Queue Exception”

   queue[rear] = data
   rear++
```

- **Dequeue Operation**

```
function  dequeue()
        if queue is empty
        return “Empty Queue Exception”

   temp = queue[front]
   front++
   return temp
```

- **getFront() Operation**

```
function getFront()
```

```
        if queue is empty
        return “Empty Queue Exception”
    temp = queue[front]
    return temp
```

Time Complexity of various operations

Let ‘n’ be the number of elements in the queue. The complexities of queue operations with this representation can be given as:

Operations	Time Complexity
enqueue(data)	O(1)
dequeue	O(1)
int getFront()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)

Applications of queues

Real-Life Applications

- Scheduling of jobs in the order of arrival by the operating system
- Multiprogramming
- Waiting time of customers at call centers
- Asynchronous data transfer

Application in solving DSA problems

- Queues are useful when we need **dynamic addition and deletion of elements in our data structure**. Since queues require O(1) time complexity for all the operations, we can achieve our tasks very efficiently.
- It is useful in many **Graph Algorithms** like breadth-first search, Dijkstra’s algorithm, and prim’s algorithm.