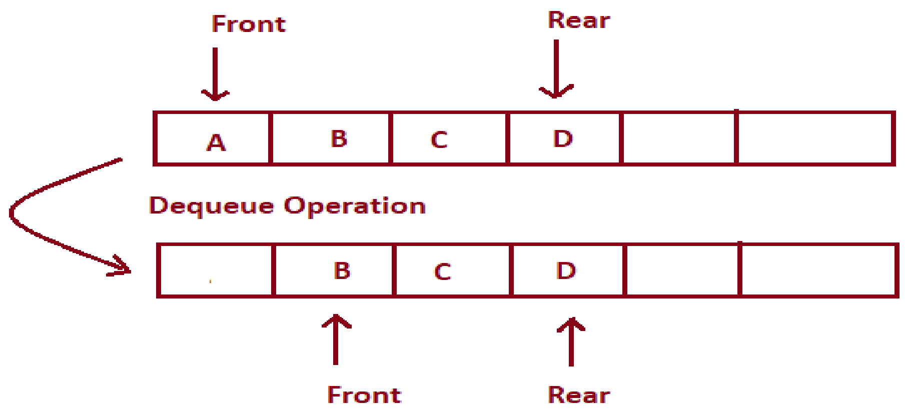
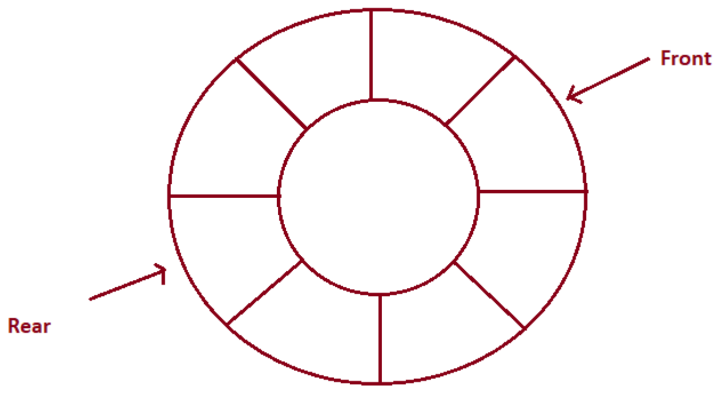
Why circular Queues?



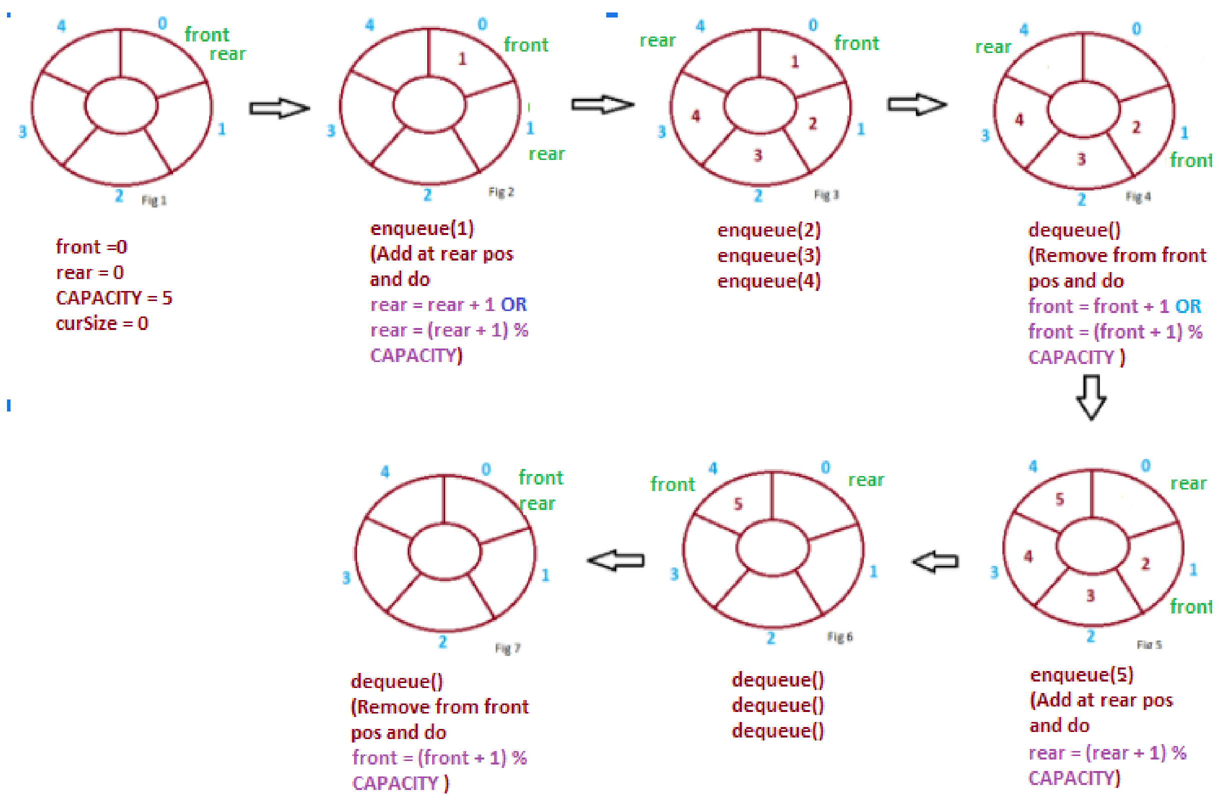
In the figure above, it can be seen that starting slots of the array are getting wasted. Therefore, simple array implementation of a queue is not memory efficient. To solve this problem, we assume arrays as circular arrays. With this representation, if we have any free slots at the beginning, the rear pointer can go to its next free slot.



Fixed size Array

How can circular queues be implemented?

Queues can be implemented using arrays and linked lists. The basic algorithm for implementing a queue remains the same.
We maintain three variables for all the operations: **front, rear,** and **curSize.**
CAPACITY of the queue is the size passed during initialization.



For the front and rear to always remain within the valid bounds of indexing, we update front as (front + 1) % CAPACITY and rear as (rear + 1) % CAPACITY. This allows front and rear to never result in an index out of bounds exception.

Notice in fig 5 we cannot do rear = rear + 1 as it will result in index out of bound exception, but there is an empty position at index 0, so we do rear = (rear + 1) % CAPACITY. Similarly, in fig 7, we cannot do front = front + 1 as it will also give an exception therefore we do front = (front + 1) % CAPACITY.

- **Enqueue Operation**

```
function enqueue(data)

        //  curSize = CAPACITY when queue is full
        if queue is full
                return "Full Queue Exception"

        curSize++
        queue[rear] = data

        //  updating rear to the next position in the circular queue
        rear = (rear+1) % CAPACITY
```

- **Dequeue Operation**

```
function  dequeue()

        //   front = rear = 0 OR curSize = 0 when queue is empty
        if queue is empty
                return "Empty Queue Exception"

         curSize--
         temp = queue[front % CAPACITY]
         front = (front + 1 ) % CAPACITY
         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)
getFront()	O(1)
boolean isEmpty()	O(1)
int size()	O(1)

Application of circular queues

- It is used in the **looped execution** of slides of a presentation.
- It is used in browsing through the open windows applications using **alt + tab** (Microsoft Windows)
- It is used for **round-robin execution** of jobs in multiprogramming OS.
- Used in the functioning of **traffic lights**.
- Used in **page replacement algorithms**: a circular list of pages is maintained and when a page needs to be replaced, the page in the front of the queue will be chosen.