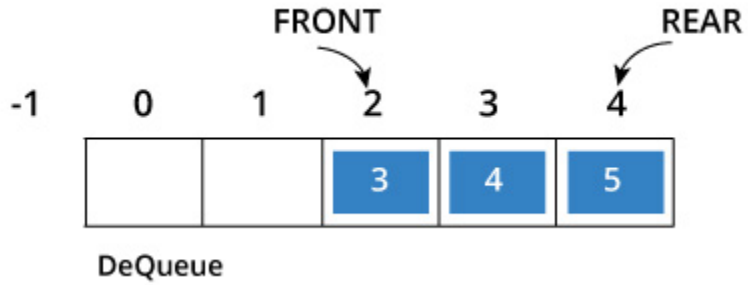
Circular queue avoids the wastage of space in a regular queue implementation using arrays.



The insertion and deletion operation in a circular queue can be performed in a manner similar to that of queue but we have to take care of two things.

If value of rear is MAX-1, then instead of incrementing rear we will make it zero and then perform insertion.

Similarly when the value of front become MAX-1,it will not be incremented but will reset to zero.

(a) Empty queue front = -1, rear = -1 [0] [1] [2] [3] [4]	(b) Insert 5 front = 0, rear = 0 5 | | | | | [0] [1] [2] [3] [4]	(c) Insert 10 front = 0, rear = 1 5 | 10 | | | | [0] [1] [2] [3] [4]
(d) Delete front = 1, rear = 1 | 10 | | | | [0] [1] [2] [3] [4]	(e) Delete front = 2, rear = 1 | | | | | [0] [1] [2] [3] [4]	(f) Insert 15, 20, 25 front = 2, rear = 4 | | 15 | 20 | 25 [0] [1] [2] [3] [4]
(g) Insert 30 front = 2, rear = 0 30 | | 15 | 20 | 25 [0] [1] [2] [3] [4]	(h) Delete front = 3, rear = 0 30 | | | 20 | 25 [0] [1] [2] [3] [4]	(i) Insert 35 front = 3, rear = 1 30 | 35 | | 20 | 25 [0] [1] [2] [3] [4]
(j) Insert 40 front = 3, rear = 2 30 | 35 | 40 | 20 | 25 [0] [1] [2] [3] [4]	(k) Delete front = 4, rear = 2 30 | 35 | 40 | | 25 [0] [1] [2] [3] [4]	(l) Delete front = 0, rear = 2 30 | 35 | 40 | | | [0] [1] [2] [3] [4]
(m) Insert 45 front = 0, rear = 3 30 | 35 | 40 | 45 | | [0] [1] [2] [3] [4]	(n) Insert 50 front = 0, rear = 4 30 | 35 | 40 | 45 | 50 [0] [1] [2] [3] [4]	(o) Delete front = 1, rear = 4 | 35 | 40 | 45 | 50 [0] [1] [2] [3] [4]

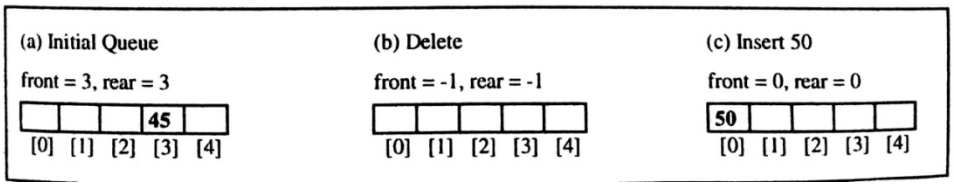
(i) As in simple queue, here also if `front` is equal to `rear` there is only one element, except initially empty queue (Cases (b) and (d)).

(ii) The circular queue will be empty in three situations, when initially `front` is equal to -1 or when `front` becomes equal to `(rear+1)`, or when `front` is equal to 0 and `rear` is equal to MAX-1 (Cases (a),(e) and the case when all elements deleted from case(o)).

(iii) The overflow condition in circular queue has changed. Here overflow will occur only when all the positions of array are occupied i.e. when the array is full. The array will be full in two situations, when `front` is equal to 0 and `rear` is equal to MAX-1 (Case (n)), or when `front` is equal to `(rear+1)` (Case (j)).

From the last two points we can see that the condition `front==(rear+1)` is true in both cases, when the queue is empty and when the queue is full. Similarly the condition (`front==0 && rear==MAX-1`) is true in both cases. We should make some change in our procedure so that we can differentiate between an empty queue and a full queue.

When the only element of the queue is deleted, `front` and `rear` are reset to -1. We can check for empty queue just by checking the value of `front`, if `front` is -1 then the queue is empty otherwise not. Let us take an example and see how this is done-




```
/*P4.6 Program of circular queue*/

#include<stdio.h>

#include<stdlib.h>

#define MAX 10

int cqueue_arr[MAX];

int front=-1;

int rear=-1;

void display( );

void insert(item);

int del();

int peek();

int isEmpty();

int isFull();
```

```
main()
{
        int choice,item;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Peek\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                case 1 :
                        printf("Input the element for insertion : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2 :
                        printf("Element deleted is : %d\n",del());
                        break;
                case 3:
                        printf("Element at the front is  : %d\n",peek());
```

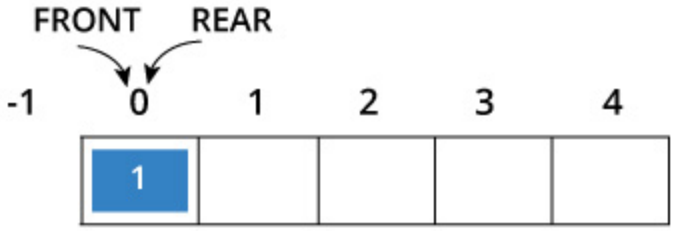
```
                    break;
            case 4:
                    display();
                    break;
            case 5:
                    exit(1);
            default:
                    printf("Wrong choice\n");
            }/*End of switch*/
    }/*End of while */
}/*End of main()*/
void insert(int item)
{
    if( isFull() )
    {
            printf("Queue Overflow\n");
            return;
    }
    if(front == -1 )
            front=0;

    if(rear==MAX-1)/*rear is at last position of queue*/
            rear=0;
    else
            rear=rear+1;
```

```
        cqueue_arr[rear]=item ;
}/*End of insert()*/
int del()
{
        int item;
        if( isEmpty() )
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        item=cqueue_arr[front];
        if(front==rear) /* queue has only one element */
        {
                front=-1;
                rear=-1;
        }
        else if(front==MAX-1)
                front=0;
        else
                front=front+1;
        return item;
}/*End of del() */
int isEmpty()
{
        if(front==-1)
```

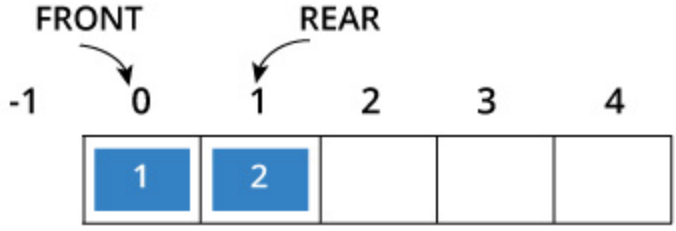
```
                return 1;
        else
                return 0;
}/*End of isEmpty()*/
int isFull()
{
        if((front==0 && rear==MAX-1) || (front==rear+1))
                return 1;
        else
                return 0;
}/*End of isFull()*/
int peek()
{
        if( isEmpty() )
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        return cqueue_arr[front];
}/*End of peek()*/
void display()
{
        int i;
        if(isEmpty())
        {
```

```
            printf("Queue is empty\n");
            return;
        }
        printf("Queue elements :\n");
        i=front;
        if( front<=rear )
        {
            while(i<=rear)
                printf("%d ",cqueue_arr[i++]);
        }
        else
        {
            while(i<=MAX-1)
                printf("%d ",cqueue_arr[i++]);
            i=0;
            while(i<=rear)
                printf("%d ",cqueue_arr[i++]);
        }
        printf("\n");
}/*End of display() */
```

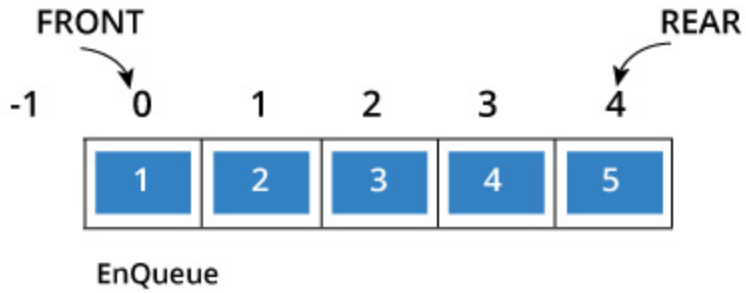
Empty Queue



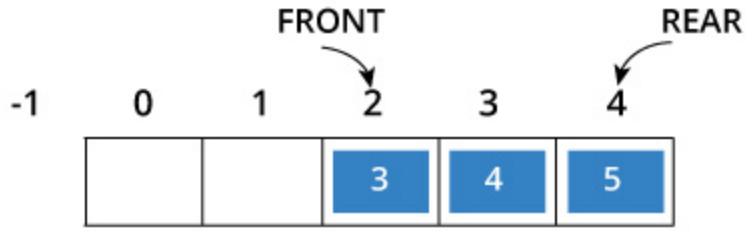
EnQueue first element



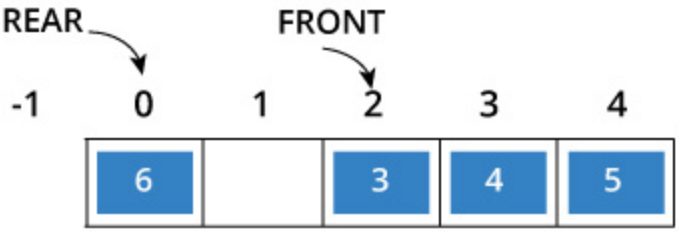
EnQueue



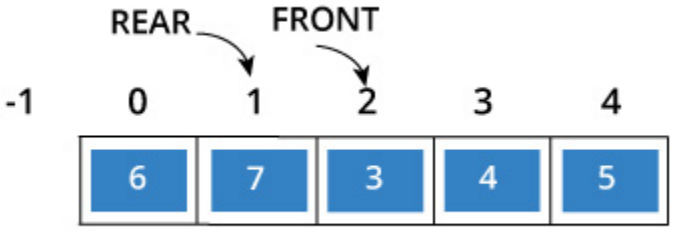
EnQueue



DeQueue



EnQueue



Queue Full