

QUEUE

1. Queue
 - a. Introduction
 - b. Array implementation of queues
 - c. Function to insert an element into the queue
 - d. Function to delete an element from the queue
2. Circular queue
 - a. Circular queue with array implementation
 - b. Function to insert an element into the queue
 - c. Function for deletion from circular queue
3. Deques(Double Ended Queue)

What is a Queue?

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

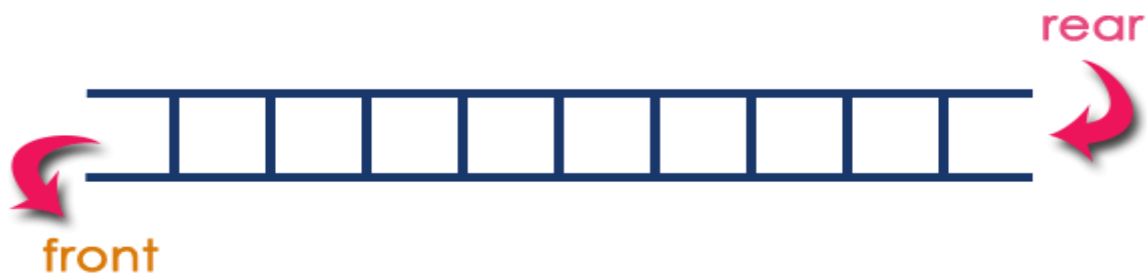
Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.

In a queue data structure, adding and removing elements are performed at two different positions.

The insertion is performed at one end and deletion is performed at another end.

In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.

In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



Queue is referred to be as First In First Out list.

For example, **people waiting in line for a rail ticket form a queue.**

In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".



Example

Queue after inserting 25, 30, 51, 60 and 85.

After Inserting five elements...



"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Operations on Queue

There are two fundamental operations performed on a Queue:

- **enqueue():** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **dequeue():** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **display() :** To display the elements of the queue
- **isEmpty():** To check if the queue is empty
- **isFull():** To check whether the queue is full or not
- **Front:** Pointer element responsible for fetching the first element from the queue
- **Rear:** Pointer element responsible for fetching the last element from the queue

Implementation of Queue

There are two ways of implementing the Queue:

- **Sequential allocation:** The sequential allocation in a Queue can be implemented using an array.
- **Linked list allocation:** The linked list allocation in a Queue can be implemented using a linked list.

Queue Datastructure Using Array

A queue data structure can be implemented using one dimensional array.

The queue implemented using array stores only fixed number of data values.

The implementation of queue data structure using array is very simple.

Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables '**front**' and '**rear**'.

Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position.

Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

➤ Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- Step 1 - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- Step 2 - Declare all the **user defined functions** which are used in queue implementation.
- Step 3 - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- Step 4 - Define two integer variables '**front**' and '**rear**' and initialize both with '-1'. (**int front = -1, rear = -1**)
- Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

➤ enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- Step 1 - Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- Step 2 - If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

➤ deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to **'-1'** (**front = rear = -1**).

➤ display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)

- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then define an integer variable 'i' and set 'i = front+1'.
- Step 4 - Display '**queue[i]**' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to **rear** (i <= rear)

Example :

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void enQueue(int);
void deQueue();
void display();

int queue[SIZE], front = -1, rear = -1;

void main()
{
    int value, choice;
    clrscr();
    while(1)
    {
        printf("\n\n***** MENU *****\n");
        printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
```

```

        enqueue(value);
        break;
case 2: dequeue();
        break;
case 3: display();
        break;
case 4: exit(0);
default: printf("\nWrong selection!!! Try again!!!");
    }
}
}
void enqueue(int value)
{
    if(rear == SIZE-1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
    else
    {
        if(front == -1)
            front = 0;
        rear++;
        queue[rear] = value;
        printf("\nInsertion success!!!");
    }
}
void dequeue()
{
    if(front > rear)
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
    else
    {
        printf("\nDeleted : %d", queue[front]);
        if(front == rear)
            front = rear = -1;
    }
}

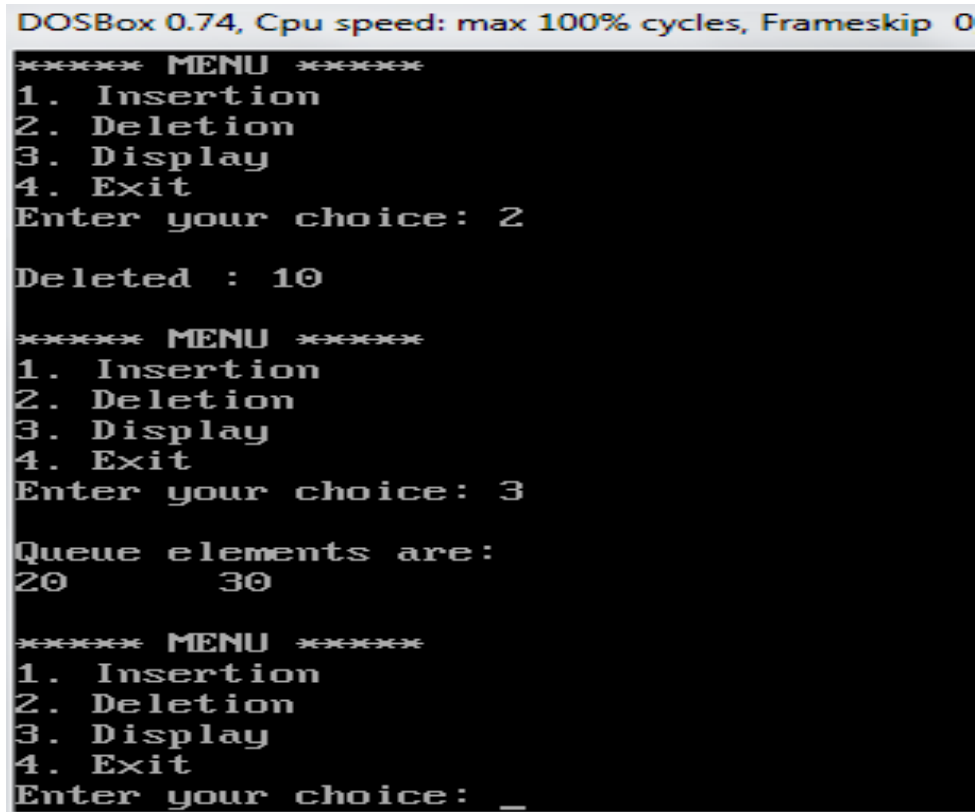
```

```

    }
    front++;
}
void display()
{
    if(rear == -1)
        printf("\nQueue is Empty!!!");
    else
    {
        int i;
        printf("\nQueue elements are:\n");
        for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
    }
}

```

Output :



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0

```

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2

Deleted : 10

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3

Queue elements are:
20      30

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: _

```


Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue. For example, consider the queue below...

The queue after inserting all the elements into it is as follows...

Queue is Full



Now consider the following situation after deleting three elements from the queue...

Queue is Full (Even three elements are deleted)



This situation also says that Queue is Full and we cannot insert the new element because 'rear' is still at last position.

In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element.

This is the major problem in a normal queue data structure.

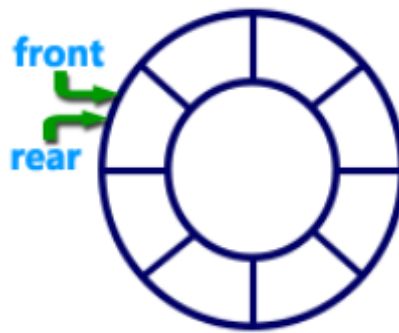
To overcome this problem we use a circular queue data structure

What is Circular Queue?

A Circular Queue can be defined as follows...

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue using array

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

- Step 1 - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- Step 2 - Declare all **user defined functions** used in circular queue implementation.
- Step 3 - Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- Step 4 - Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- Step 1 - Check whether **queue** is **FULL**. **((rear == SIZE-1 && front == 0) || (front == rear+1))**
- Step 2 - If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- Step 4 - Increment **rear** value by one (**rear++**), set **queue[rear] = value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue...

- Step 1 - Check whether **queue** is **EMPTY**. **(front == -1 && rear == -1)**
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then

check whether both **front - 1** and **rear** are equal (**front - 1 == rear**), if it **TRUE**, then set both **front** and **rear** to **-1** (**front = rear = -1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

- Step 1 - Check whether **queue** is **EMPTY**. (**front == -1**)
- Step 2 - If it is **EMPTY**, then display "**Queue is EMPTY!!!**" and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then define an integer variable 'i' and set '**i = front**'.
- Step 4 - Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment 'i' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- Step 5 - If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment 'i' value by one (**i++**). Repeat the same until '**i <= SIZE - 1**' becomes **FALSE**.
- Step 6 - Set **i** to **0**.
- Step 7 - Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Example :

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
```

```
void enQueue(int);
void deQueue();
void display();
```

```
int cQueue[SIZE], front = -1, rear = -1;
```

```

void main()
{
    int choice, value;
    clrscr();
    while(1)
    {
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: printf("\nEnter the value to be insert: ");
                    scanf("%d",&value);
                    enqueue(value);
                    break;
            case 2: dequeue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nPlease select the correct choice!!!\n");
        }
    }
}

void enqueue(int value)
{
    if((front == 0 && rear == SIZE - 1) || (front == rear+1))
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
    else
    {
        if(rear == SIZE-1 && front != 0)

```

```

        rear = -1;
        cQueue[++rear] = value;
        printf("\nInsertion Success!!!\n");
        if(front == -1)
            front = 0;
    }
}
void deQueue()
{
    if(front == -1 && rear == -1)
        printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
    else
    {
        printf("\nDeleted element : %d\n",cQueue[front++]);
        if(front == SIZE)
            front = 0;
        if(front-1 == rear)
            front = rear = -1;
    }
}
void display()
{
    if(front == -1)
        printf("\nCircular Queue is Empty!!!\n");
    else
    {
        int i = front;
        printf("\nCircular Queue Elements are : \n");
        if(front <= rear)
        {
            while(i <= rear)
                printf("%d\t",cQueue[i++]);
        }
    }
}

```

```

else
{
    while(i <= SIZE - 1)
        printf("%d\t", cQueue[i++]);
    i = 0;
    while(i <= rear)
        printf("%d\t", cQueue[i++]);
}
}
}

```

Output :

```

DOS BOX DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, P
***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Circular Queue Elements are :
10      20      30      40      50
***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted element : 10

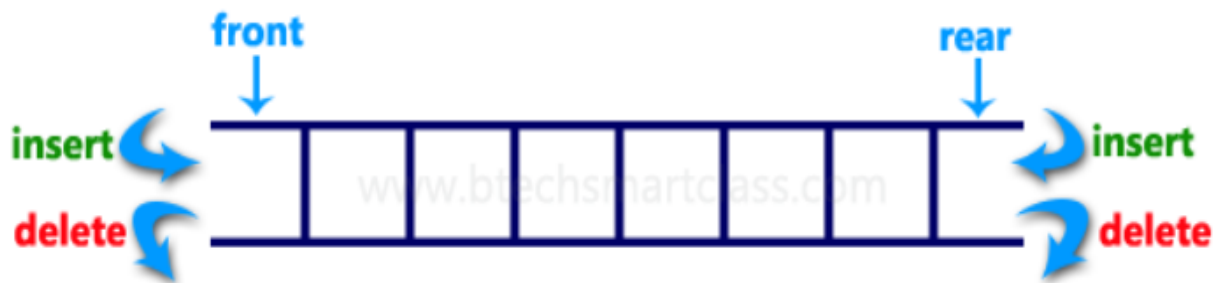
***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3_

```

Deque(Double Ended Queue Datastructure)

“deque” is the short form of double ended queue.....

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

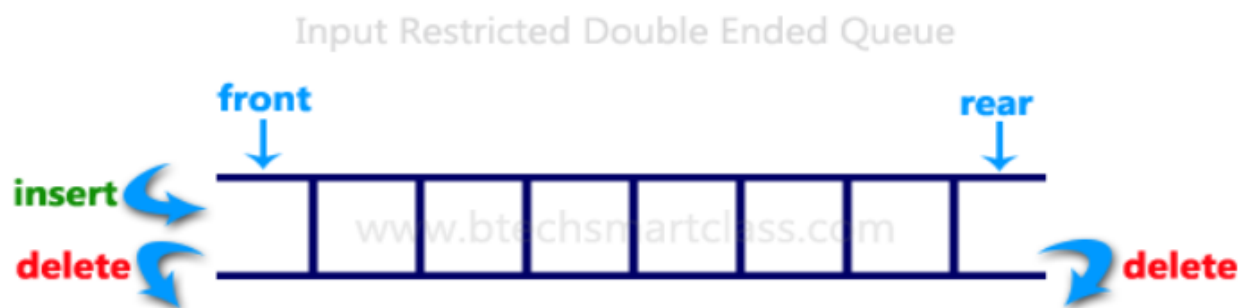


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

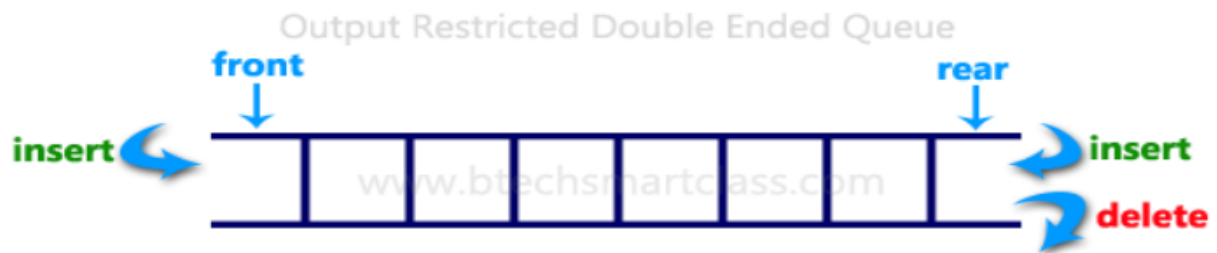
Input Restricted Double Ended Queue

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

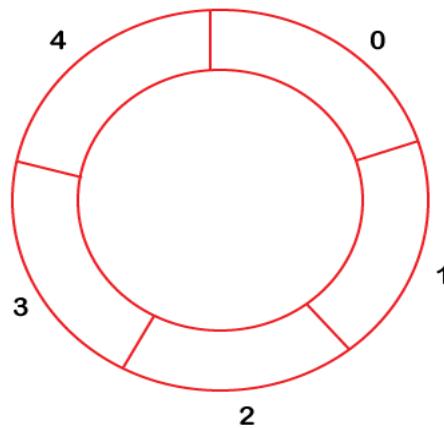
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

What is a circular array?

An array is said to be **circular** if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4, and the array is full but the first location of the array is empty. If we want to insert the array element, it will not show any overflow condition as the last element is connected to the first element. The value which we want to insert will be added in the first location of the array.



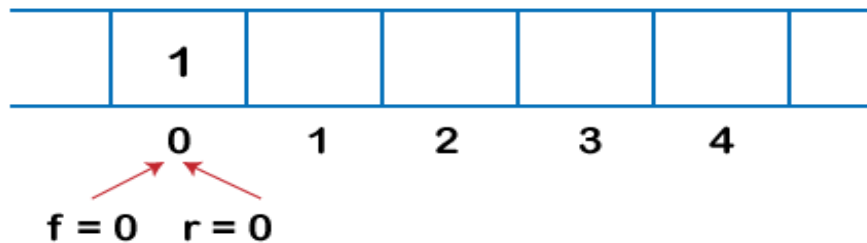
Implementation of Deque using a circular array

The following are the steps to perform the operations on the Deque:

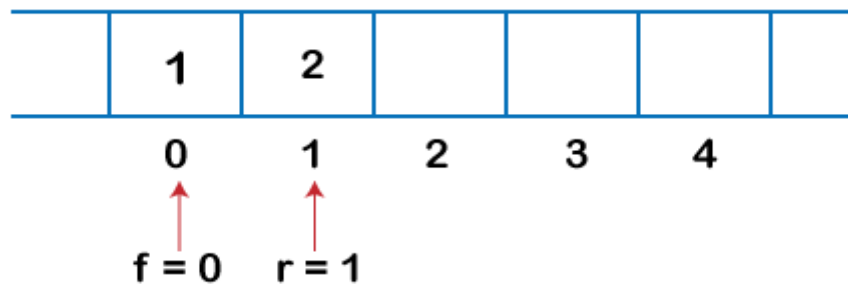
Enqueue operation

1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.

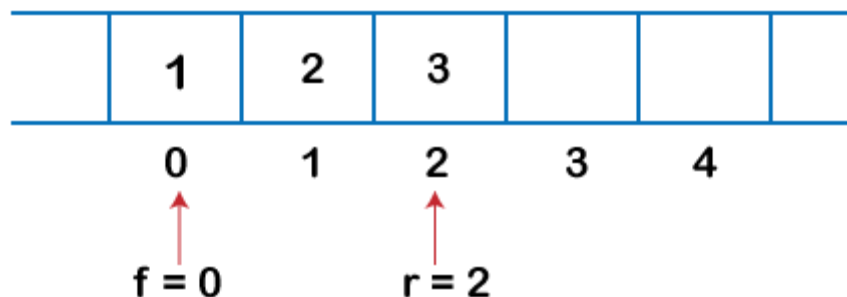
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0**, and the **rear is also equal to 0**.



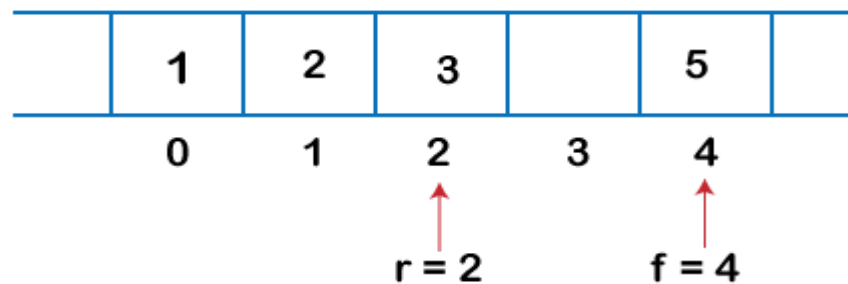
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.

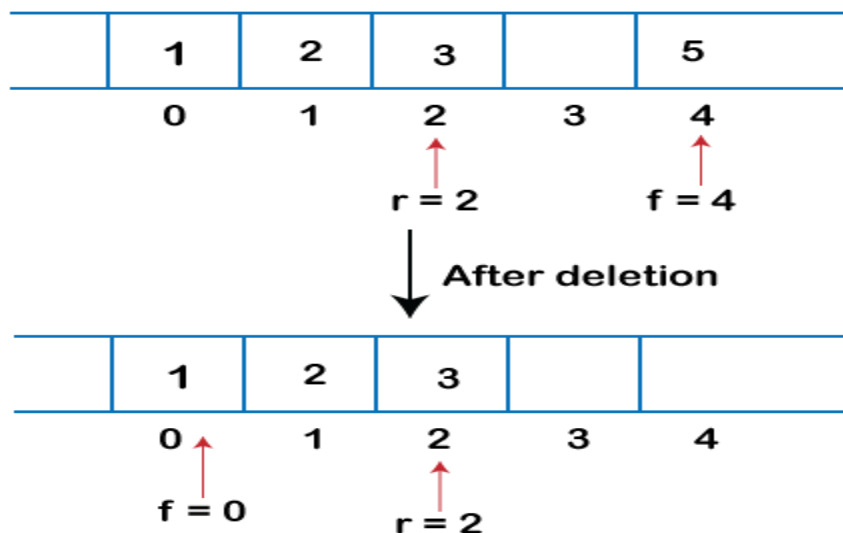


If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n - 1)**, which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



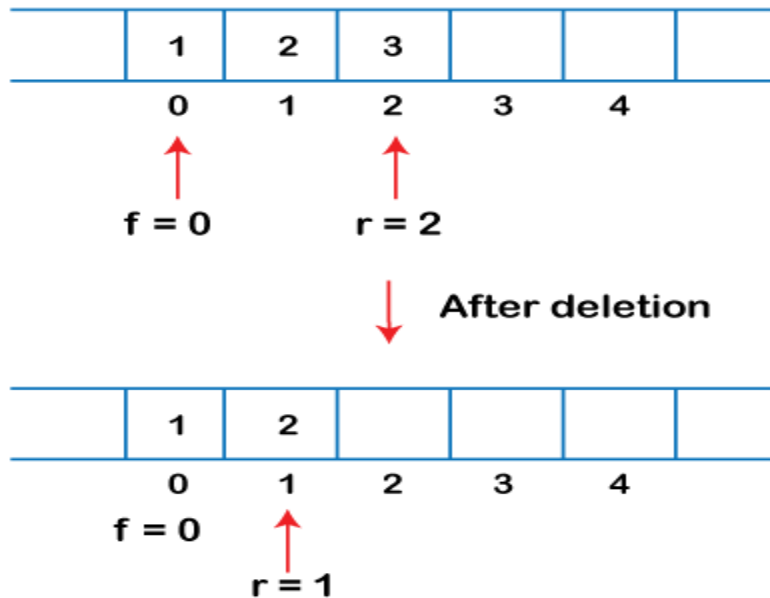
Deque Operation

1: If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index.

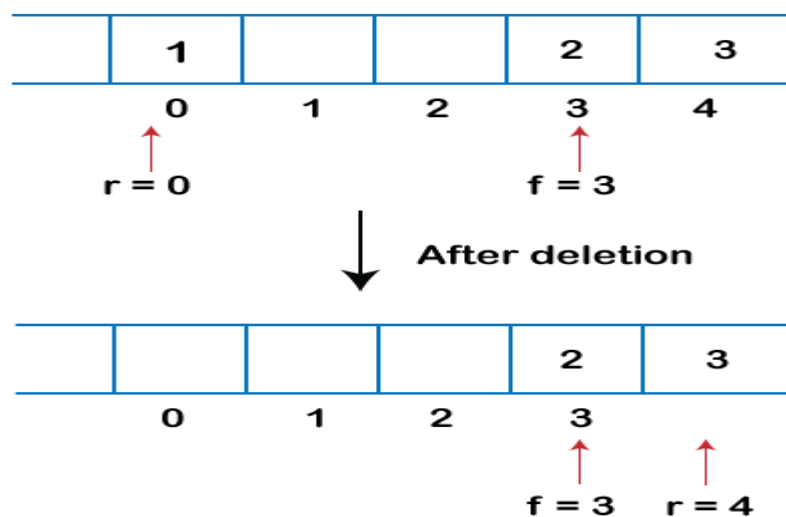


Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.

2: If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



3: If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Example :

```
#define size 5
#include <stdio.h>
int deque[size];
int f=-1, r=-1;

void enqueue_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

void enqueue_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
}
```

```

    }
    else if((f== -1) && (r== -1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
}

void display()
{
    int i=f;
    printf("\n Elements in a deque : ");

    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
    printf("%d",deque[r]);
}

void getfront()
{
    if((f== -1) && (r== -1))
    {

```

```

        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the front is: %d", deque[f]);
    }
}
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the rear is: %d", deque[r]);
    }
}
void dequeue_front()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=-1;
        r=-1;
    }
    else if(f==(size-1))
    {

```



```

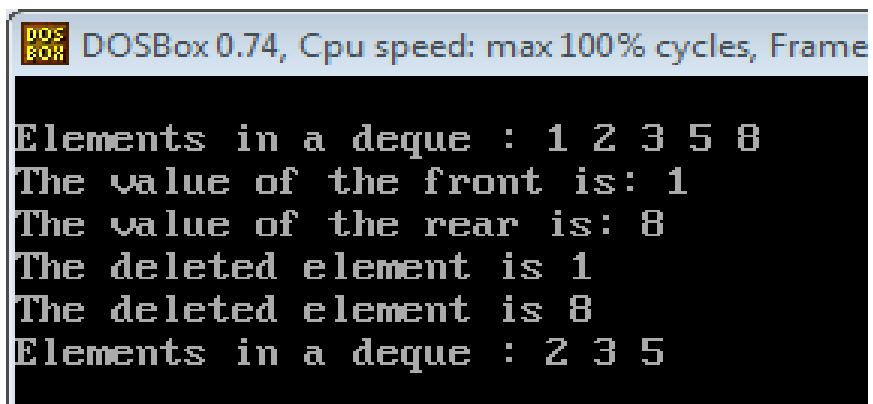
        printf("\nThe deleted element is %d", deque[f]);
        f=0;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}
void dequeue_rear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;

    }
    else if(r==0)
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=size-1;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}

```

```
int main()
{
    enqueue_front(2);
    enqueue_front(1);
    enqueue_rear(3);
    enqueue_rear(5);
    enqueue_rear(8);
    display();
    getfront();
    getrear();
    dequeue_front();
    dequeue_rear();
    display();
    return 0;
}
```

Output :



DOSBox 0.74, Cpu speed: max 100% cycles, Frame

```
Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
Elements in a deque : 2 3 5
```