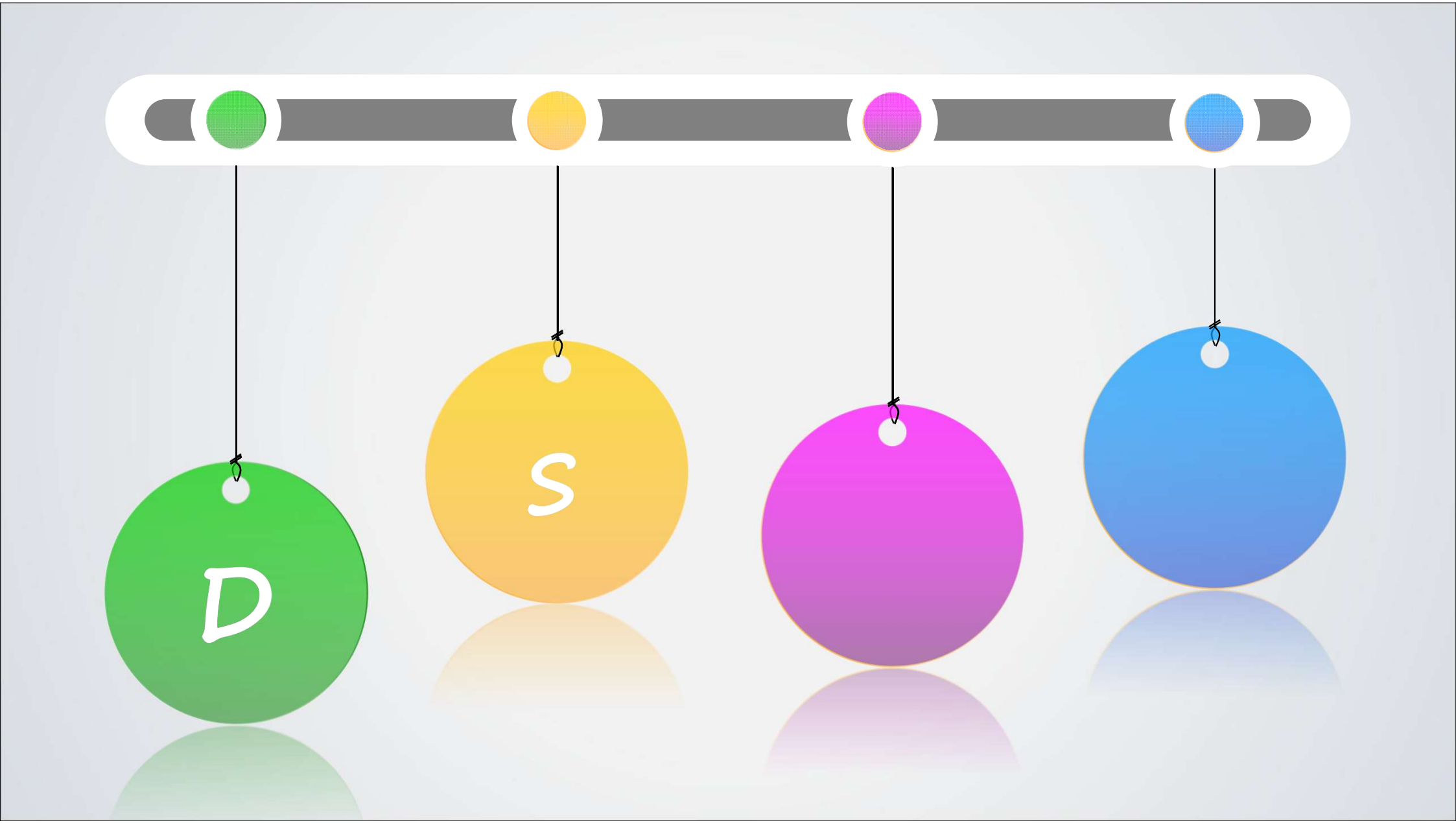

D
S

01 TOPIC

Queues: Introduction

02 TOPIC

Array and Linked List representation of Queues

03 TOPIC

Operations on Queue using Array

04 TOPIC

Operations on Queue using Linked List

05 TOPIC

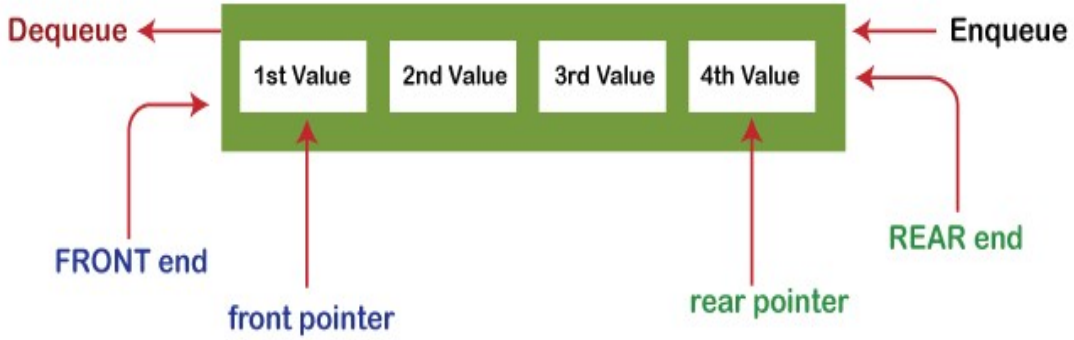
Circular Queue Using Array

06 TOPIC

Deque using array

Queue

- Queue is a linear data structure, in which the **element is inserted from** one end called the **REAR**(also called tail), and the **removal of an element takes place** from the other end called as **FRONT**(also called head).



- This makes queue as FIFO(First in First Out) data structure, which means that **element inserted first will be removed first.**
- Which is exactly **how queue system works in real world**. If you go to a **ticket counter to buy movie tickets,** and are first in the queue, then you will be the first one to get the tickets.

Queue

- There are two ways to implement a queue:

i. **Using array**

ii. **Using linked list**

Implement of Queue Using array:

Syntax:

```
#define Macro_Name  Value

Datatype  Array_Name[Macro_Name];

Int front=-1,rear=-1;
```

Ex:

```
#define Maxsize 10

int  Queue[Maxsize];

int front=-1,rear=-1;
```

Queue Using Array

- The **basic operations** of queue:

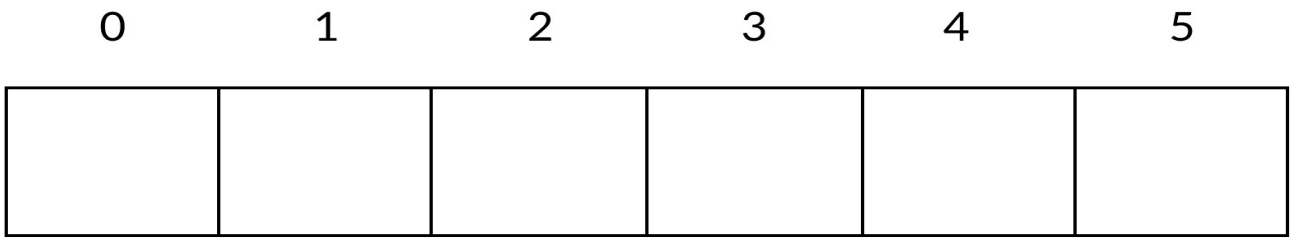
  i. ***insert:*** This operation is used ***to insert the element at the rear end*** of the queue.

  ii. ***delate:*** This operation performs the ***deletion from the front-end*** of the queue.

  iii. ***Display:*** This is the third operation that ***displays the element from front to rear.***

Queue Operations

Front = Rear = -1



Empty Queue

Insert Operation using array

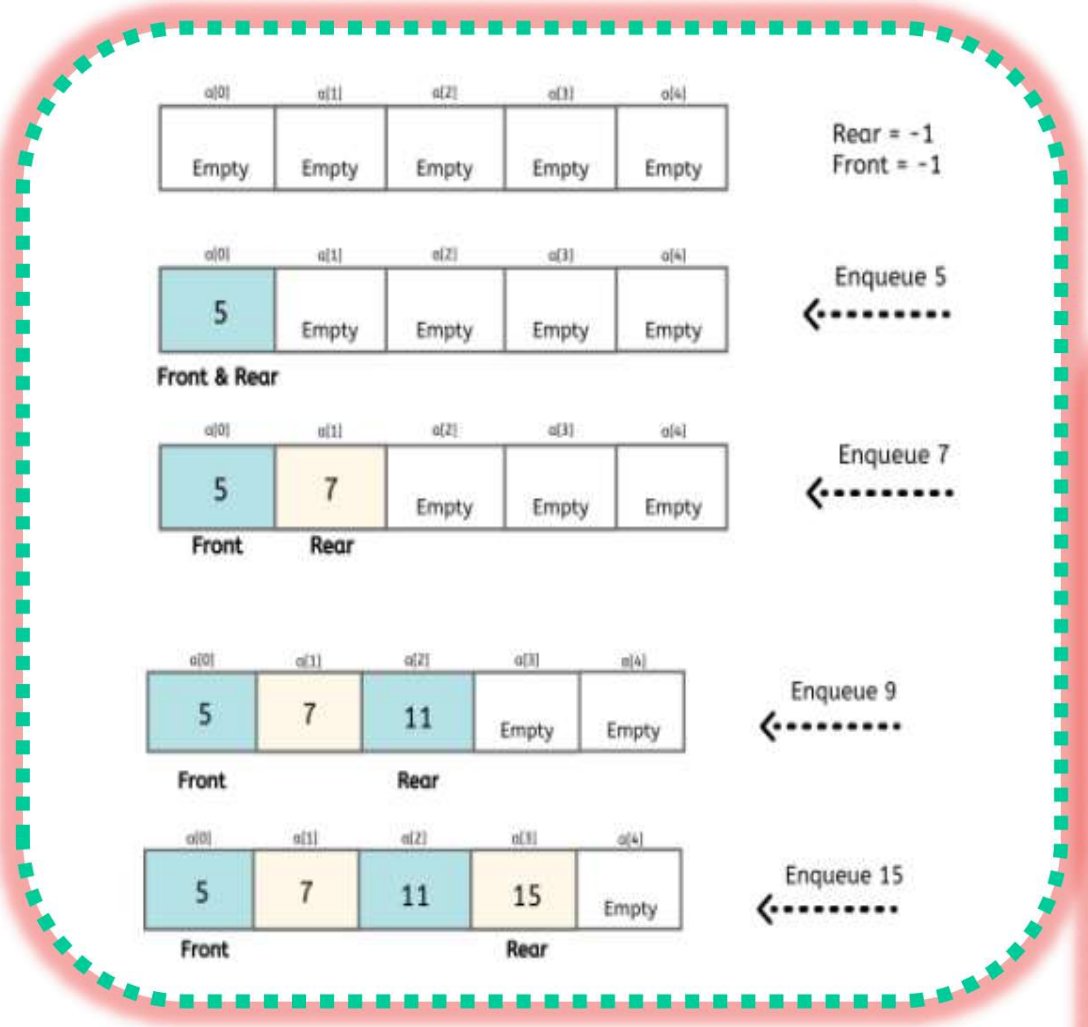
- Queues maintain **front** and **rear**.
- The following steps should be taken to insert data into a queue:

Step 1 – Check if the **queue is full.**

Step 2 – If the **queue is full**, produce **overflow error** and exit.

Step 3 – If the **queue is not full**, **increment rear** to the **next empty space.**

Step 4 – **Add data element** to the queue location, **where the rear is pointing.**




```
insert(int ele)
{
        if(rear==Maxsize-1)
        {
            printf("\n Queue is Overflow");
             return;
        }
        if(front==-1&&rear==-1)
              front=front+1;
         rear=rear+1;
         Queue[rear]=ele;
}
```

delete operation using array

- The **data where front** is **pointing** and **remove the data**

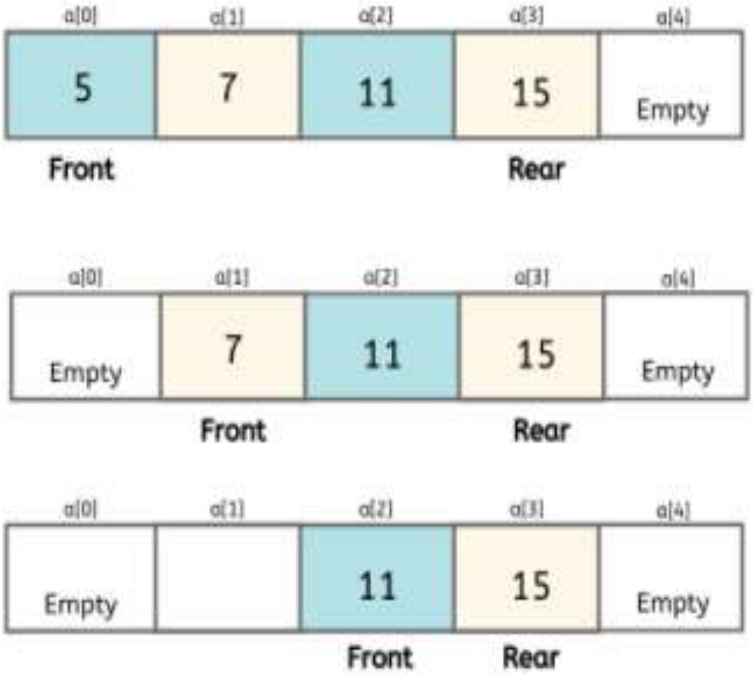
- The **following steps** are **taken** to **perform delete** operation:

  - **Step 1** – Check if the **queue is empty.**

  - **Step 2** – If the **queue is empty**, produce **underflow error** and exit.

  - **Step 3** – If the **queue is not empty**, **delete the data where front is pointing.**

  - **Step 4** – **Increment front pointer** to point to the next available data element.



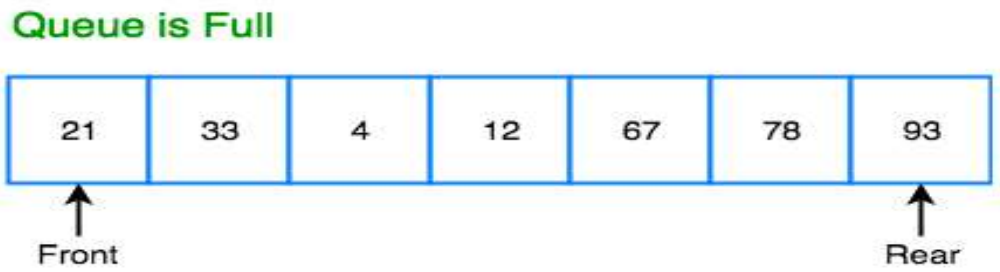

```
void delete( )
{
        if(front==-1)
        {
        printf("\Queue is underflow");
        return;
        }
        ele=Queue[front];
        if(front==rear)
             front=rear=-1;
        else
            front=front+1;
        printf(“%i is deleted”,ele);
}
```

```
void display()
{
        int i;
        if(rear==-1)
        {
                printf("\nEmpty Queue");
        }
        else
        {
                printf("\n The Queue is");
                for(i=front;i<=rear;i++)
                printf("%3i",queue[i]);
        }
}
```

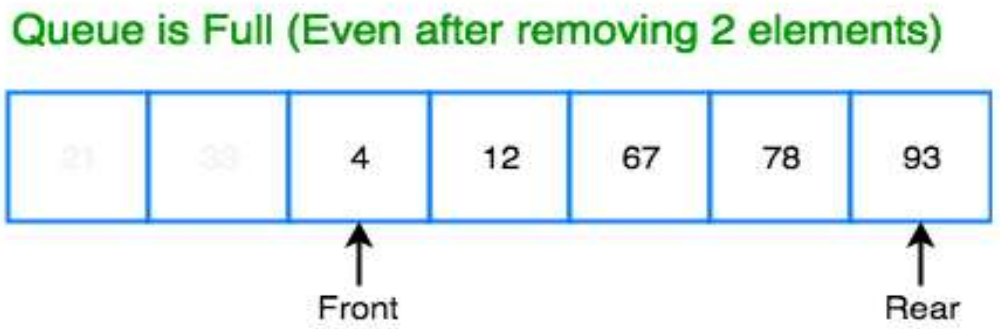
```
//program for Queue using array
#include<stdio.h>
#define maxsize 5
void insert(int);
void delete();
void display();
int queue[maxsize];
int front = -1, rear = -1,ele;
void main ()
{
   int choice;
   do
   {
      printf("\n******Main Menu******\n");
      printf("\n1.insert \n2.Delete \n3.Display");
      printf("\nEnter your choice :");
      scanf("%i",&choice);
```

```
switch(choice)
 {
    case 1:  printf("\n Enter element:");
             scanf("%i",&ele);
             insert(ele);
             break;

   case 2:    delete();
              break;

   case 3:    display();
              break;

   default: printf("\nyour choice is invalid");
      }

   } while(choice != 3) ;
}
```

Drawback of Simple Queue

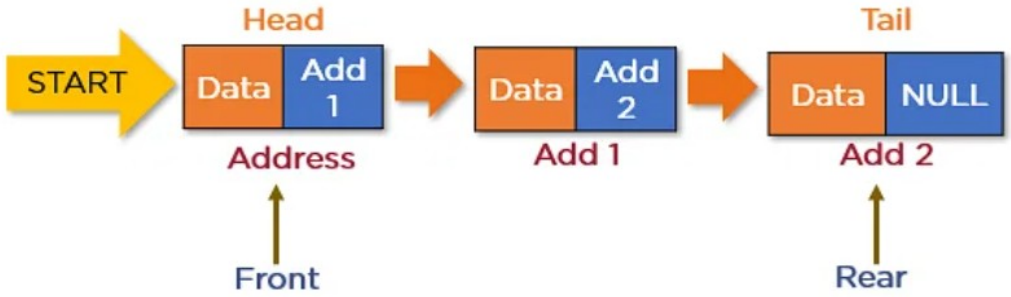


❖Even if we remove some of the elements, until the queue is reset, no new elements can be inserted.



Queue Using Linked List

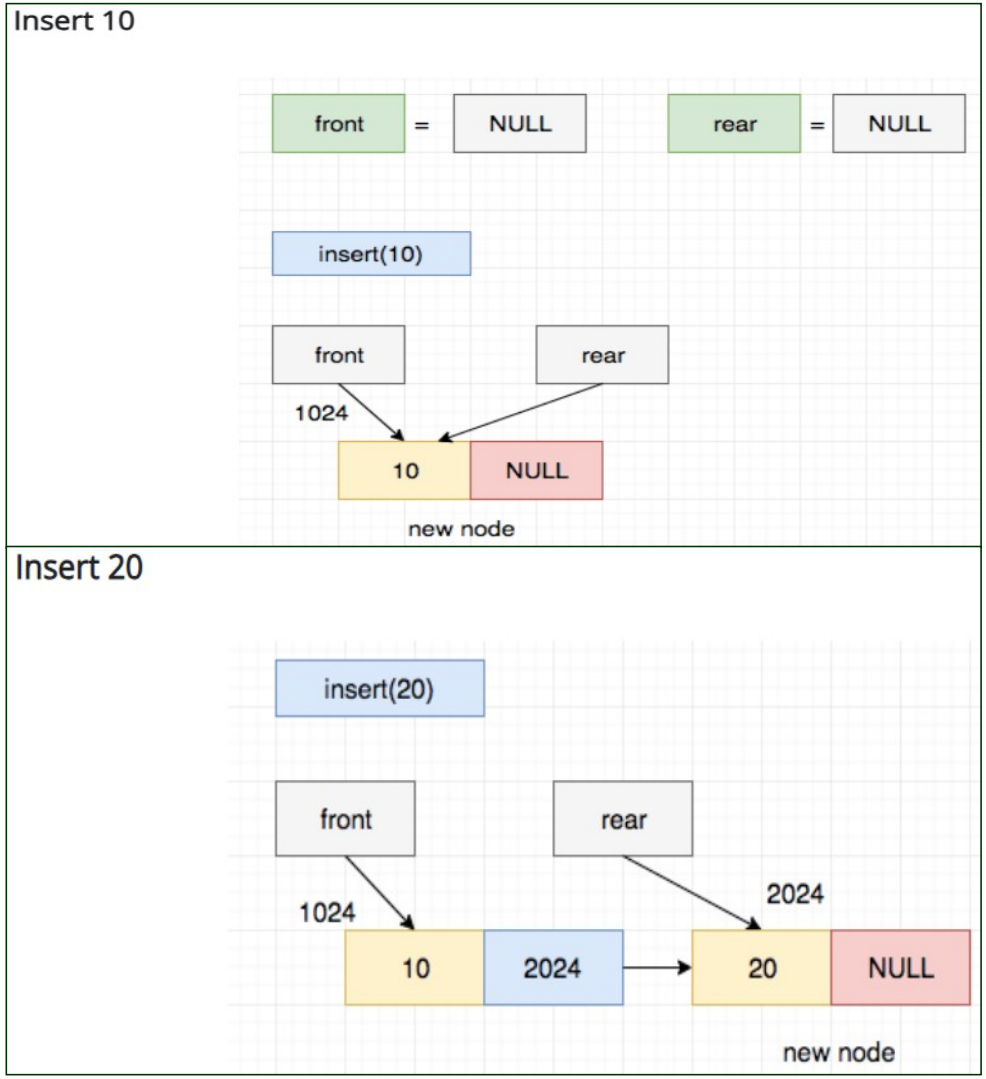
- In a **linked queue, each node of the queue** consists of **two fields**, i.e., **data** field and **address field**.
- The **front pointer stores** the location where the queue **starts**, and **rear pointer** keeps track of the **last data element** of a **queue.**

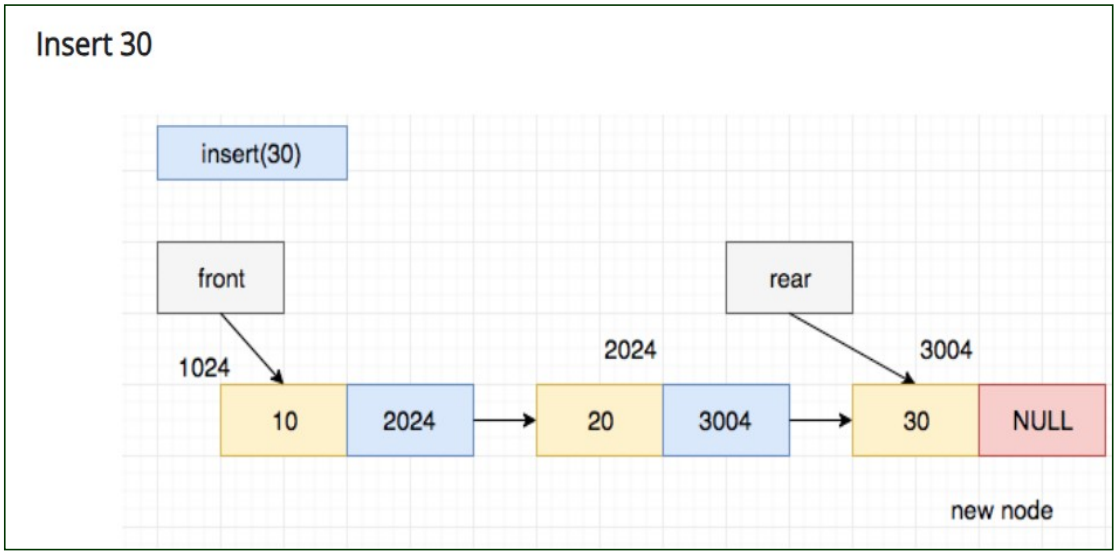



```
Struct Node
{
        int data;
        struct Node *next;
}*front=NULL,*rear=NULL.*temp=NULL;
```

- **Basic operations** of queue are:
  - i. **Insert**
  - ii. **Delete**
  - iii. **Display**

Insert Operation using linked List

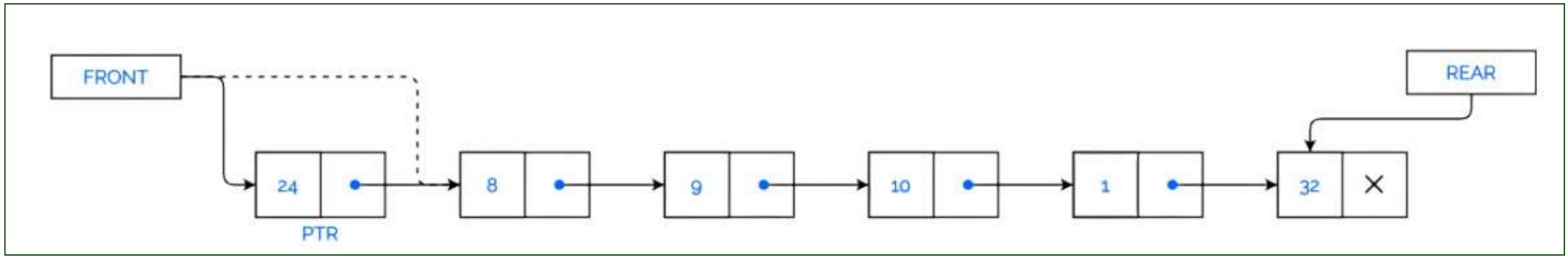


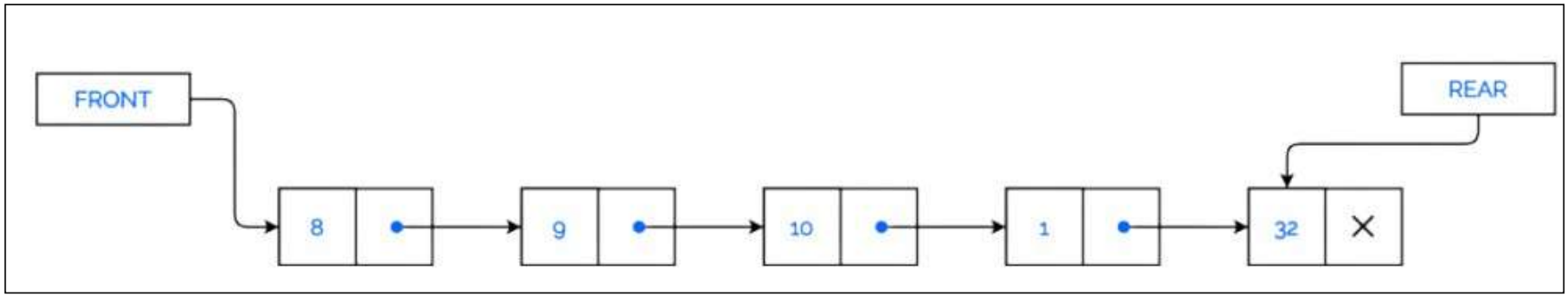


Insert Operation using linked List

```
void insert(int ele)
{
        temp=(struct Node*)malloc(sizeof(struct Node));
        temp->data=ele;
        temp->next=NULL;
        if(front==NULL)
        {
                front=rear=temp;
        }
        else
        {
                rear->next=temp;
                rear=temp;
        }
}
```

delete operation using linked list





delete operation using linked list

```
void del( )
{
        if(front==NULL)
        {
                printf("\nQueue is empty");
                return;
        }
        else
        {
                temp=front;
                front=front->next;
                ele=temp->data;
                free(temp);
                printf("\n %d is deleted",ele);
        }
}
```

Display operation using linked list

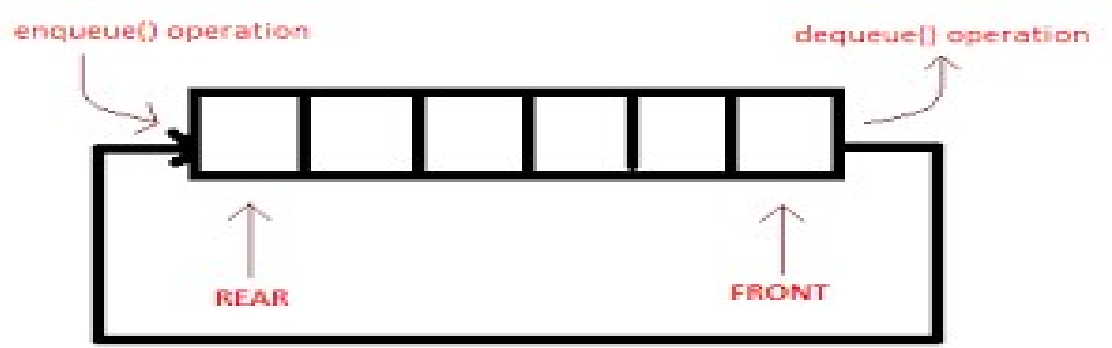
```
void display( )
{
        if(front==NULL)
                printf(“\n Queue is empty”);
        else
        {
                temp=front;
                while(temp!=NULL)
                {
                printf("%3d",temp->data);
                temp=temp->next;
                }
        }
}
```

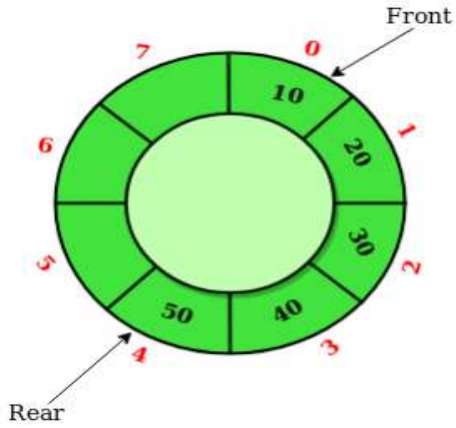
```c
//Program for Queue Using Linked List
#include<stdio.h>
Struct Node
{
        int data;
        struct Node *next;
}*front=NULL,*rear=NULL,*temp=NULL;
void insert(int);
void delete();
void display();
void main ()
{
   int choice;
   do
   {
      printf("\n******Main Menu******\n");
      printf("\n1.insert \n2.Delete \n3.Display");
      printf("\nEnter your choice :");
      scanf("%i",&choice);
```

```c
switch(choice)
 {
    case 1:  printf("\n Enter element:");
             scanf("%i",&ele);
             insert(ele);
             break;

   case 2:    delete();
              break;

   case 3:    display();
              break;

   default: printf("\nyour choice is invalid");
      }

   } while(choice != 3) ;
}
```

Circular Queue

- **Circular Queue** is also a linear data structure.
- It also follows the principle of **FIFO** rule, but **instead of ending the queue at the last position, it again starts from the first position after the last.**

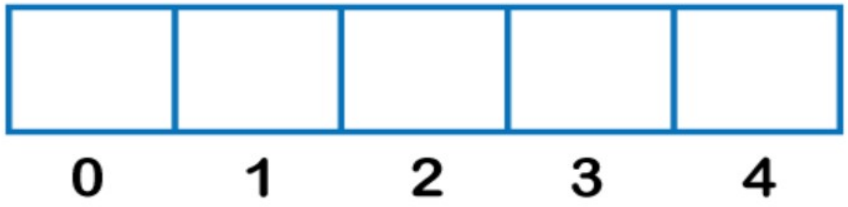




- common **real-world examples** where circular queues are used:
  - ✓Computer controlled **Traffic Signal System** uses circular queue.
  - ✓**CPU scheduling** and Memory management.

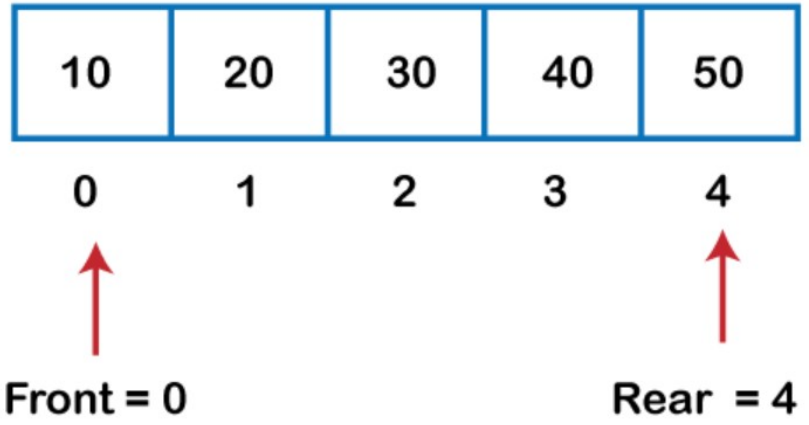
Insert Operation in Circular Queue

```
void insert(int ele)
{
   if ((front == 0 && rear == MAXSIZE-1) ||(rear==front-1))
   {
      printf("\nQueue is Full");
      return;
   }

   else if (front == -1)
   {
      front = rear = 0;
      Queue[rear] = ele;
   }
  else if (rear== MAXSIZE-1 && front != 0)
   {
      rear = 0;
      Queue[rear] = ele;
   }

   else
   {
      rear++;
      Queue[rear] = ele;
   }
}}
```



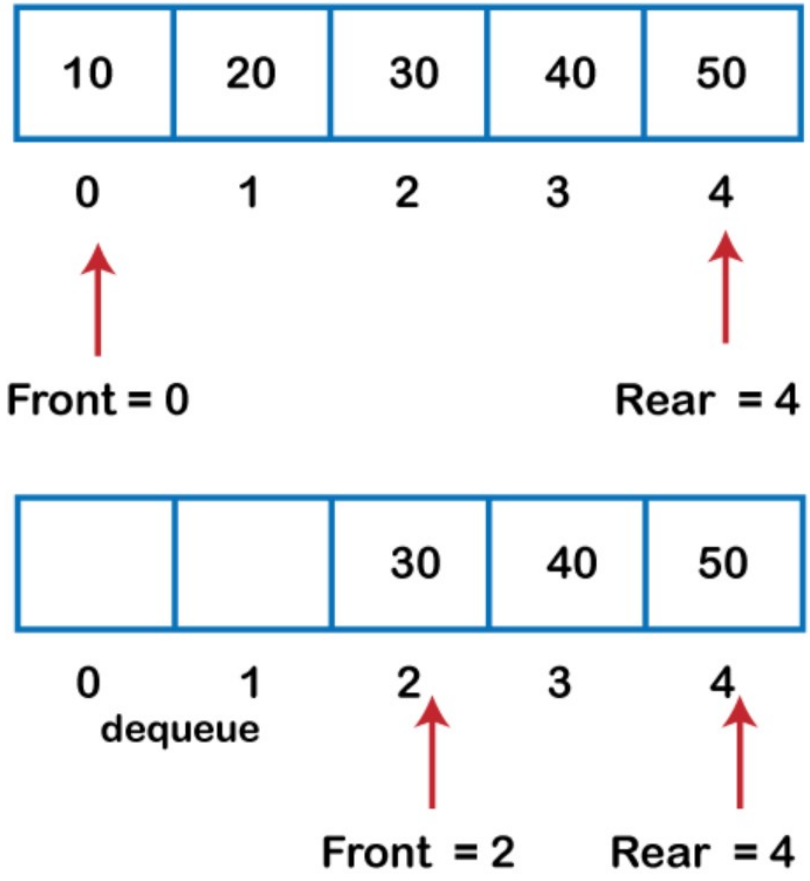
Front = -1

Rear = -1



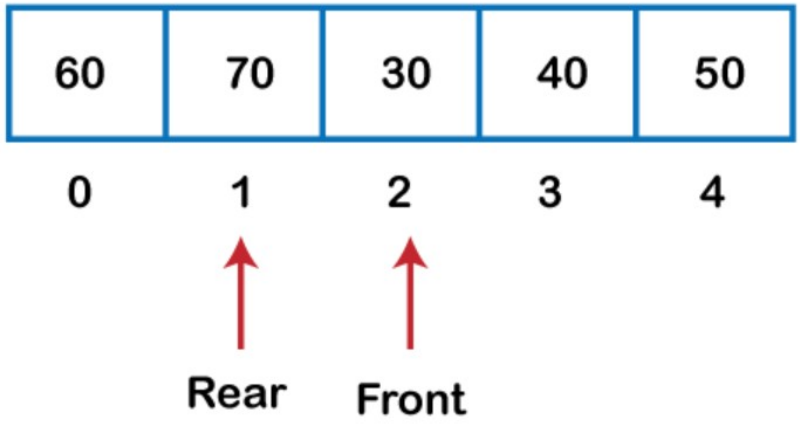
delete operation in Circular Queue

```
int del()
{
        int ele;
        if(front==–1 && rear==–1)
         {
                printf("\n UNDERFLOW");
                return 0;
        }
        ele = queue[front];
        if(front==rear)
                front=rear=–1;
        else
        {
                if(front==MAXSIZE–1)
                        front=0;
                else
                        front=front+1;
        }
        return ele;
}
```



Display operation using linked list

```
void display( )
{
        int i;
        if (front ==–1 && rear= =–1)
         printf ("\n QUEUE IS EMPTY");
        else
        {
                if(front<rear)
                 {
                        for(i=front;i<=rear;i++)
                                printf("\t %d", Queue[i]);
                }
                else
                 {
                        for(i=front;i<MAXSIZE-1;i++)
                                printf("\t %d", Queue[i]);
                        for(i=0;i<=rear;i++)
                                printf("\t %d", Queue[i]);
                }
        }
}
```

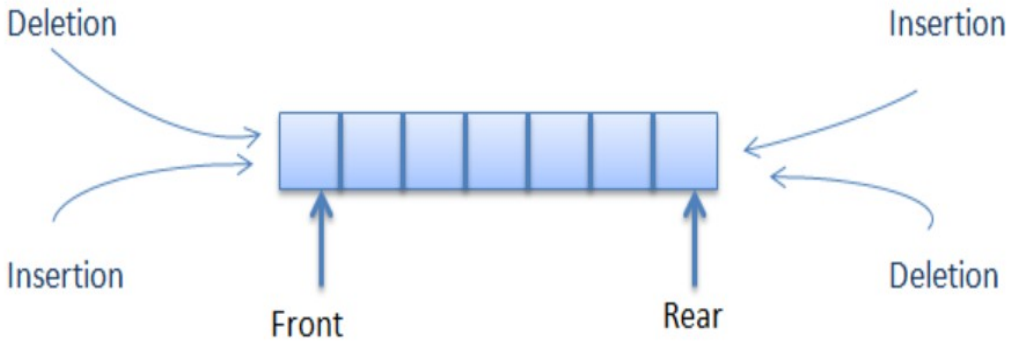


```
//Program for Circular Queue Using array
#include<stdio.h>
#define maxsize 5
void insert(int);
void delete();
void display();
int queue[maxsize];
int front = -1, rear = -1,ele;
void main ()
{
   int choice;
   do
   {
      printf("\n******Main Menu******\n");
      printf("\n1.insert \n2.Delete \n3.Display");
      printf("\nEnter your choice :");
      scanf("%i",&choice);
```

```
switch(choice)
 {
    case 1:  printf("\n Enter element:");
             scanf("%i",&ele);
             insert(ele);
             break;

   case 2:    delete();
              break;

   case 3:    display();
              break;

   default: printf("\nyour choice is invalid");
      }

   } while(choice != 3) ;
}
```

Dequeue

- The **deque** stands for **Double Ended Queue**. It is a **linear data structure** where the **insertion** and **deletion operations** are **performed** from **both ends.**
- The **representation** of a **deque** is given as follows:

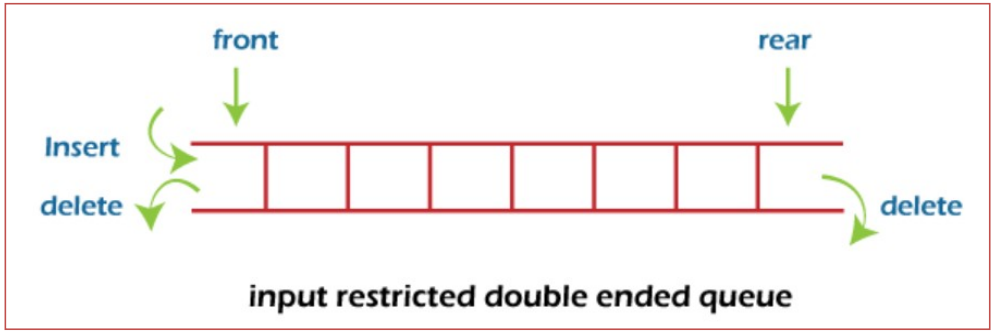


- Operations performed on deque
    i. **Insertion at front**
    ii. **Insertion at rear**
    iii. **Deletion at front**
    iv. **Deletion at rear**
- There are **two types** of **deque -**
    i. **Input restricted queue**
    ii. **Output restricted queue**

Types of Dequeue

i. Input restricted Queue:

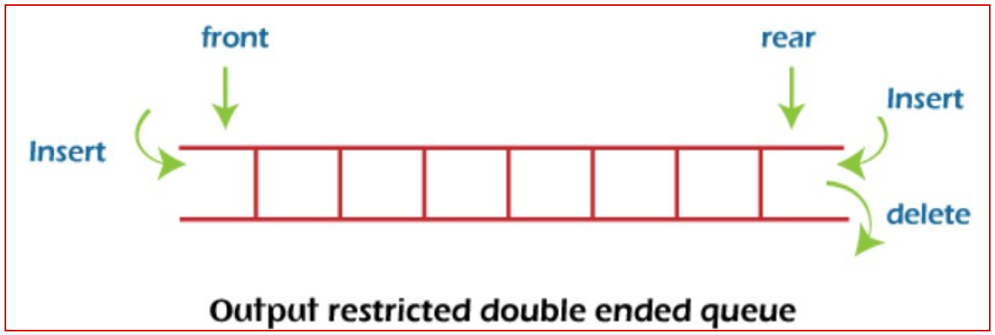
✓ In input restricted queue, **insertion operation** can be **performed at only one end**, while **deletion** can **be performed** from **both ends.**



input restricted double ended queue

ii. Output restricted Queue:

✓ In output restricted queue, **deletion operation** can be **performed** at **only one end**, while **insertion** can be performed **from both ends.**

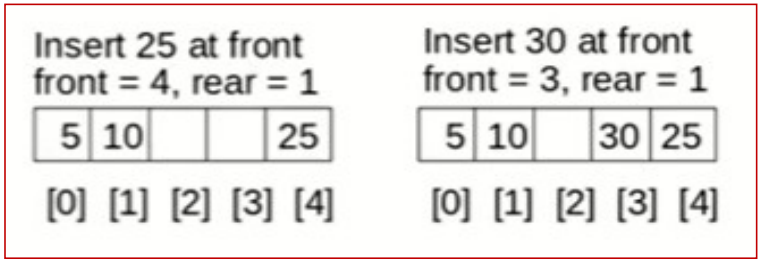


Output restricted double ended queue

Operations of Dequeue

i. Insertion at front:

- In this operation, the **element is inserted from** the **front end** of the queue.
- Before doing the operation, **we first** have to **check whether the queue is full or not.**
- If the **queue is empty**, both **rear and front** are set as 0. Now, **both** will **point** to the **first element.**
- Otherwise, check the position of the front if the **front is less than 1 (front < 1),** then reinitialize it by **front =maxsize - 1**, i.e., the **last index of the array.**

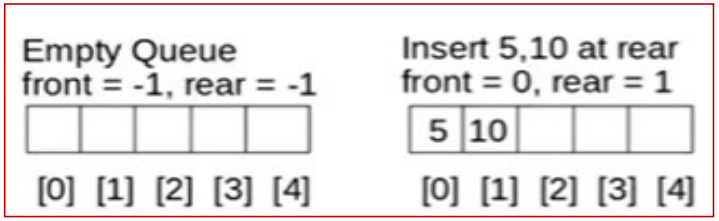



```
void insert_front(int ele)
{
        if((front==0&&rear==maxsize-1)||(front==rear+1))
          {
                printf("Overflow");
                return;
          }
          else if((front==-1) && (rear==-1))
          {
                front=rear=0;
                deque[front]=ele;
          }
          else if(front==0)
          {
                front=maxsize-1;
                deque[front]=ele;
          }
          else
          {
                front=front-1;
                deque[front]=ele;
          }
}
```

Operations of Dequeue

ii. Insertion at the rear end:

- In this operation, the **element** is **inserted from** the **rear end.**
- Before doing the operation, we **first have to check again whether** the **queue is full or not.**
- If the **queue is empty**, both **rear** and **front** are initialized **with 0**. Now, **both will point** to the **first element.**
- Otherwise, **increment the rear by 1**. If the **rear is at last index (or size - 1),** then **instead of increasing it by 1**, we have **to make it equal to 0.**




```
void insert_rear(int ele)
{
        if((front==0&&rear==maxsize-1)||(front==rear+1))
        {
                printf("Overflow");
                return;
        }
        else if((front==-1)&&(rear==-1))
        {
                rear=front=0;
                deque[rear]=ele;
        }

        else
        {
                rear=rear+1;
                deque[rear]=ele;
        }
}
```

Operations of Dequeue

iii. Deletion at the front end:

- In this operation, the **element is deleted from the front end** of the queue.
- Before doing the operation, **we first have to check whether the queue is empty or not.**
- If the **queue is empty**, we **cannot perform the deletion.**
- If the **queue is not full**, then the **element can be deleted from the front end** by **using** the **below conditions** :
- **If** the **deque has only one element**, set **rear = -1 and front = -1.**
- **Else if front is at last index, set front = 0.**
- **Else increment the front by 1.**

```
void delete_front( )
{
        if((front==-1)&&(rear==-1))
        {
                printf("Deque is empty");
                return;
        }
        else if(front==rear)
        {
           printf("\n %d is deleted from front", deque[front]);
                front=rear=-1;
        }
        else if(front==(maxsize-1))
        {
        printf("\n%d is deleted from front", deque[front]);
                front=0;
        }
        else
        {
        printf("\n %d is deleted from front", deque[front]);
                front=front+1;
        }
}
```

Operations of Dequeue

iv. Deletion at the rear end:

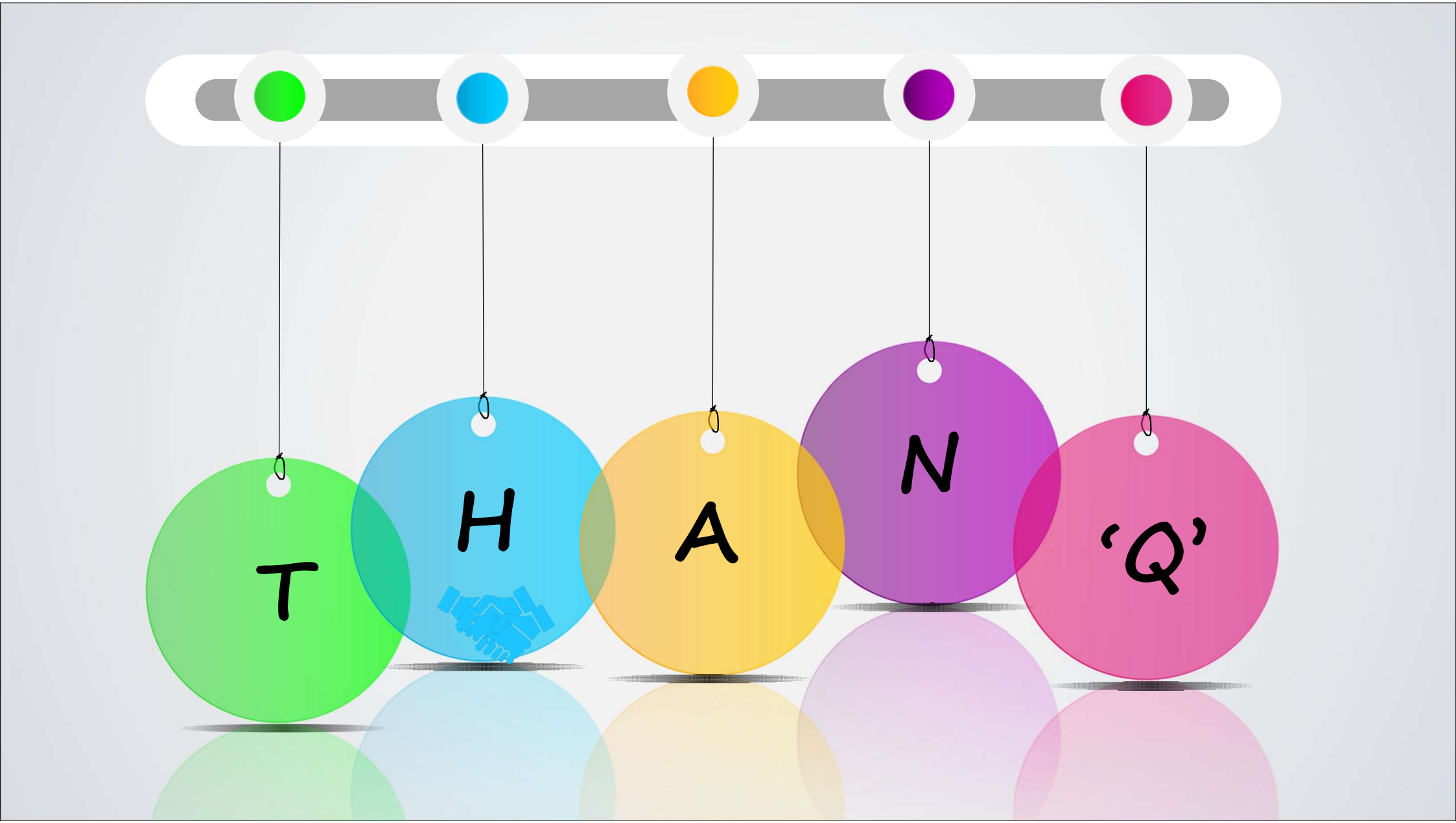
- In this operation, the **element is deleted** from the **rear end of the queue.**
- Before doing the operation, we **first have to check** whether the **queue is empty or not.**
- If the **queue is empty** ,we **cannot perform the deletion**.
- **If** the **deque has only one element**, **set rear = -1** and **front = -1.**
- **Else If** **rear = 0 , then set rear = n - 1.**
- **Else**, **decrement the rear by 1.**

```
void delete_rear()
{
        if((front==-1)&&(rear==-1))
        {
        printf("Deque is empty");
        return;
        }
        else if(front==rear)
        {
        printf("\n%d is deleted from rear",deque[rear]);
                front=rear=-1;
        }
        else if(rear==0)
        {
        printf("\n%d is deleted from rear",deque[rear]);
                rear=maxsize-1;
        }
        else
        {
        printf("\n%d is deleted from rear",deque[rear]);
                rear=rear-1;
        }
}
```

```
//Program for Dequeue Using array
#include <stdio.h>
#define maxsize 8
int deque[maxsize];
int front=-1,rear=-1,ele;
void main()
{
        int choice;
        do
        {       printf("\n******Main Menu******");
                printf("\n 1.Insert-front\t 2.Insert-rear\t 3.Delete-Front\t 4.Delete-rear\t 5.Display");
                printf("\nEnter your choice:");
                scanf("%i",&choice);
                switch(choice)
                {
                        case 1:printf("\n Enter element:");scanf("%i",&ele);insert_front(ele);break;
                        case 2:printf("\n Enter element:");scanf("%i",&ele);insert_rear(ele);break;
                        case 3:delete_front();break;
                        case 4:delete_rear();break;
                        case 5:display();break;
                        default:printf("\nyour choice is invalid");
                }
        }while(choice<6);
}
```

Operations of Dequeue

```
void display()
{
        int i=front;
        printf("\nElements in a deque are: ");
        while(i!=rear)
        {
                printf("%d ",deque[i]);
                i=(i+1)%maxsize;
        }
                printf("%d",deque[rear]);
}
```


T
H
A
N
'Q'