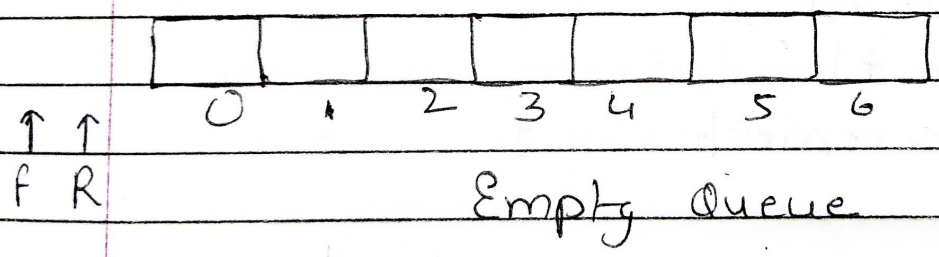
CHAPTER - 5
Queues

Queue is a non-primitive linear data-structure. It is an homogenous collection of elements in which new elements are added at one end called the Rear end and the existing elements are deleted form other end called front end

Follows concept of First In First Out (FIFO)

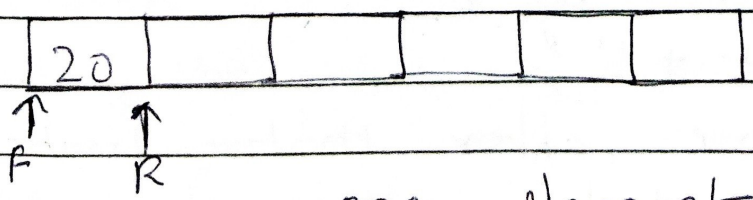
★ **Representation :-**

1) R = -1 & F = -1 // because queue is empty.



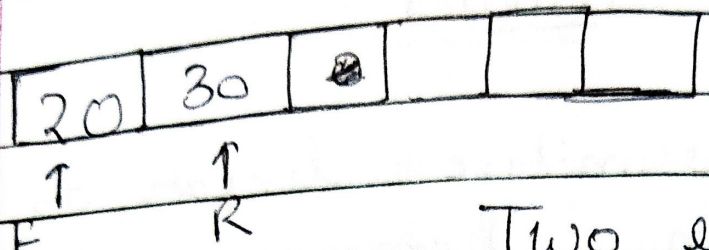
Empty Queue

2) R = 0 & F = 0



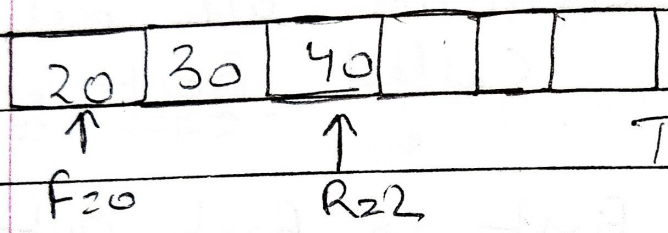
one element Queue

3) R = 1 and F = 0



Two element Queue

4) R = 2 and F = 0

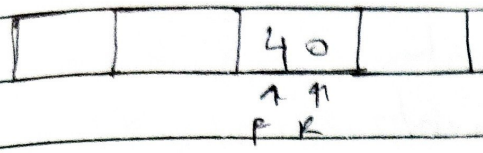


Three element Queue

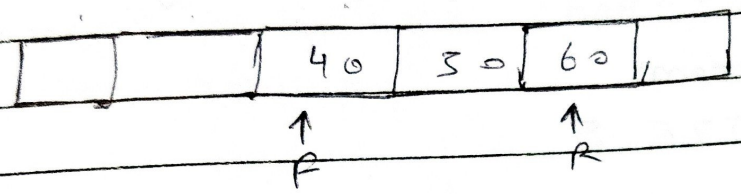
→ It is clear from the above steps that whatever we insert an element in the queue :
Rear = Rear + 1 ;
Rear is increment by 1.

→ NOTE that during insertion of first element in the queue we always increment front by 1
front = front + 1 ;
But only once, after that front is not incremented ;

OTER CASES;



F = 2
R = 2



F = 2
R = 4

★ Queue Implementation :-

1) Static implementation (using arrays)
2) Dynamic implementation (using pointers)

→ Static Implementation :- We must be sure about the exact number of elements, we want to store in the Queue, because we have to declare the size of the array at design time or before the processing starts.

A. OPERATIONS ON A QUEUE :-

1) To insert an element
2) Deletion of an element.

Implementation of queue using arrays :-

```
#include <iostream>
#define MAXSIZE 5

Class menu {
    int front, rear;
    int q[10];
    Public:

    void qinsert ();
    int qdelete ();
    menuqueue () { front = -1; rear = -1; }
};

Void main ()
{
    int choice;
    menuqueue menu;
```

```
do
{ cout << endl << "1 -> insert" << endl;
  cout << "2 -> delete" << endl;
  cout << "3 -> display" << endl;
  cout << "4 -> exit" << endl;

  cout << "enter choice" << endl;
  cin >> choice;

  switch (choice)
  {
  case 1 : menu.qinsert();
           break;
  case 2 : menu.qdelete();
           break;
  case 3 : menu.qdisplay();
           break;
  case 4 : Return;
  }
}
while (choice != 4);
}
```

```
void menuqueue :: qinsert ()
{ int num;
  if (rear == (MaxSize-1))
  {  cout << "queue is full";
     return;
  }
  else {
     cout << "enter num" << endl;
     cin >> num;
     rear = rear + 1;
     q[rear] = num;

  if (front == -1) {
        front ++;
     } }
  return;
  }
```

```
int menuqueue :: qdelete ()
{ int num;
  if (front == -1) {
     cout << " Empty" << endl;
        return 0;
     }
  else
  {
```

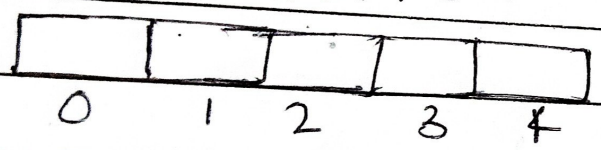
```
if (front == rear){
    front = rear = -1; } //last element
else
{ num = q[front];
  cout << "deleted items = " << q[front];
  front ++;
 }
}
return (num);
}

void menuqueue :: qdisplay()
{ int i;
 if (fronnt == -1){
      cout << " queue empty ";
      return;
    }
else {
cout << " The elements in queue " << endl;
For (i = front ; i <= rear ; i++)
  {
    cout << q[i] << endl;
  }
 }
cout << endl;
}
```

A. Limitation of Simple Queue :-

If we implement queue using arrays, so we will face problem after the element cannot be enter when the queue is full.

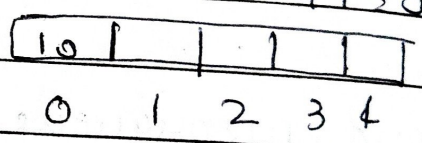
→ Example of Simple Queue {Q[5]}

R = -1 and F = -1



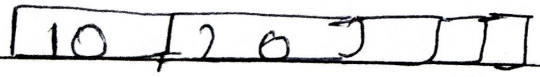
(a) Empty

R = 0 & F = 0



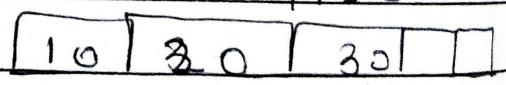
(b) One elem

R = 1 & F = 0



(c)
Two element

R = 2 & F = 0



(d)
Three element

NOW,

R = 2 4 F = 1

	20	30		
0	1	2	3	4

(e) deleting an element

R = 4 4 F = 1

	20	30	40	50

(f) on adding two elements

R = 4 4 F = 2

		30	40	50

(g) delete an element

R = 4 4 F = 3

			40	50

(h) delete

R = 4 , F = 4

				50
0	1	2	3	4

(i) delete

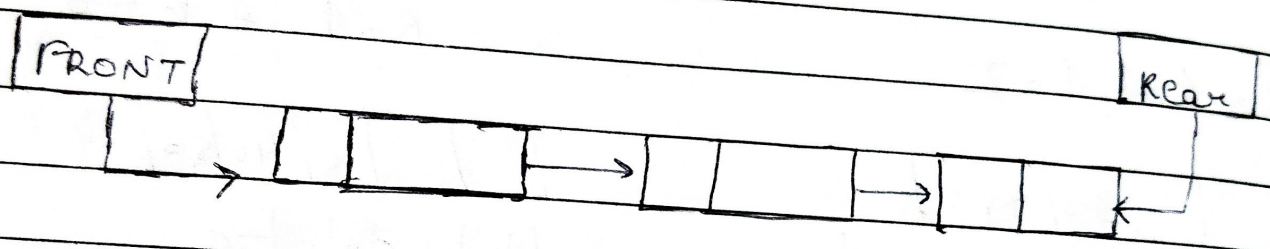
R = 4 4 F = 5

(j)

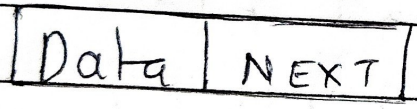
* Now, if we attempt to add more elements, the elements can't be inserted, because in a queue the elements are always inserted from the rear end, and now rear indicates to last location of the array (R = 4).

One solution can be when the elements are deleted, shift all afterwards elements to the left by one position, which is not possible if we have 200 or more elements.

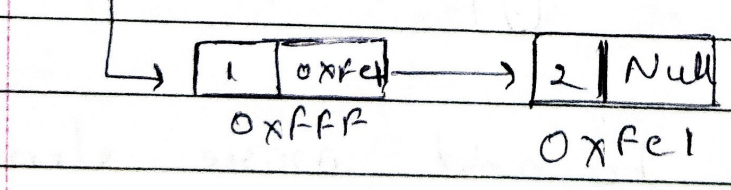
Algo for Insertion and deletion using pointers:-



(b) A Queue



(a) A Node

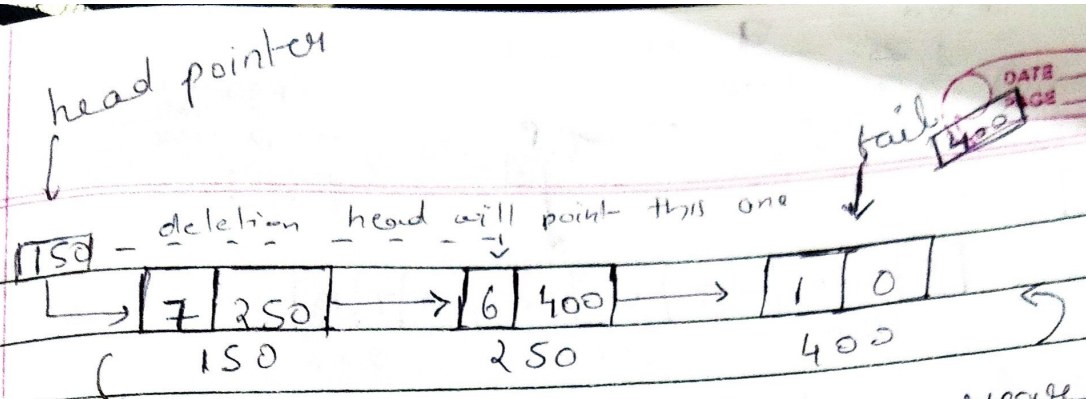


→ Requirement :-

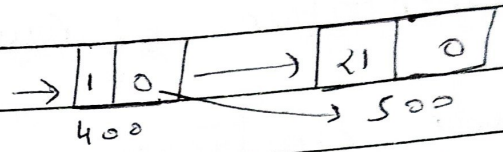
1) Node

2) front, rear, link pointers.

3) First In First Out O(1)



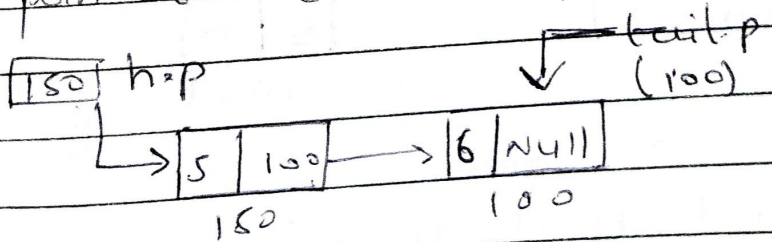
→ Now, to add element, we must have to use pointer.



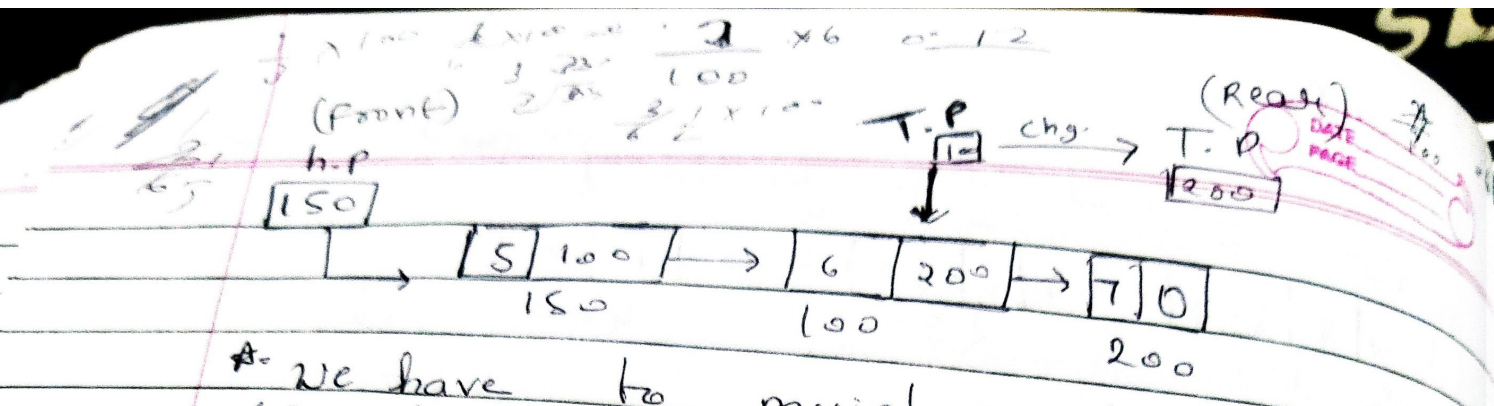
But we have only one pointer i.e "head pointer".

→ So, we will create another pointer *temp, which will traverse through queue and then "temp→link = newnode".

★ Queue must have space complexity of O(1) not O(n), so we maintain two pointer (1) head pointer (2) tail pointer.



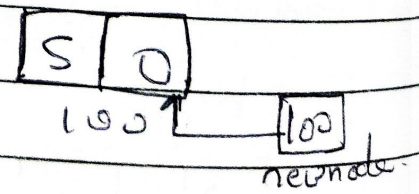
If we add new node, we dont have traverse the whole queue. We, simply add using (tail pointer) it will provide S.C O(1).



* We have to maintain two pointers (FIFO)

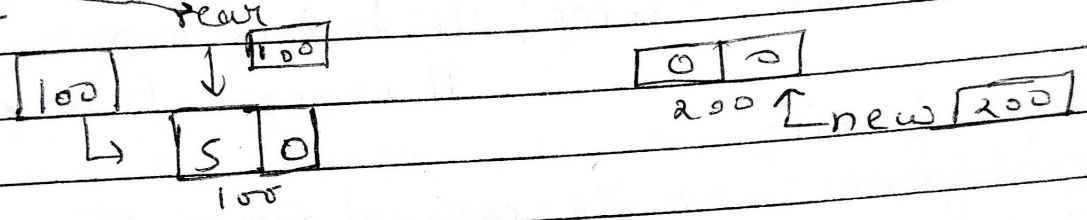
Implementation of Queue using pointers :-

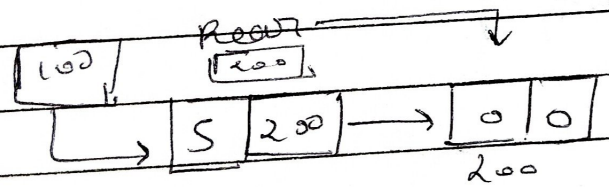
```
# Include <iostream>
using namespace std;

struct node {
        int data;
        struct node * next;
}
struct *Node *front = 0;
struct Node *rear = 0;

void enqueue (int x)
{
Struct node *Newnode;
newnode = new (struct node);   //pointer to node
newnode -> data = x;
newnode -> next = 0;
```



```
if (front==0 && rear==0)
{
  front = rear = newnode;
}

else {
  rear = rear -> next;
  rear = newnode;
}//end
```

/*





newnode = 200

From rear we can only insert

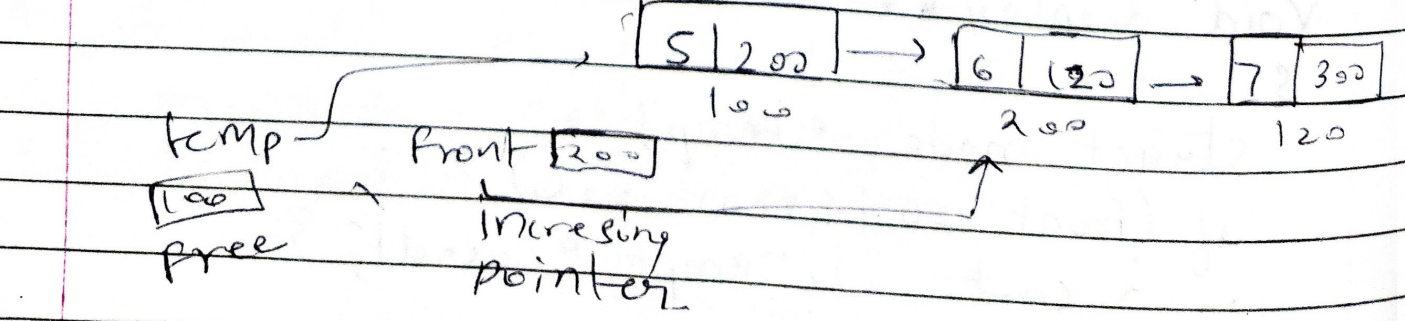
*/

```
void display ()
{
  struct node *temp;
  if (front==0 && rear==0)
    { cout << "empty" << endl; }

  temp = front;
```

```
while (temp != 0)
{
    cout << "data is" << temp->data;
    temp = temp->Next;
}
} //end
```

temp pointing using address

```
void dequeue()    //front data will be deleted
{
struct node *temp; // to free memory after
                   // changes front pointer
temp = front;
if (front == 0 && rear == 0) return 0;
    cout << "empty";

else
{
 front = front->next;
 delete (temp);
}
} //end
```



18/PPR/23

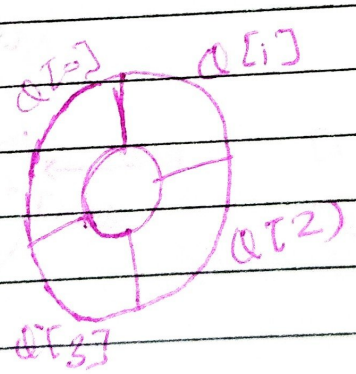
Variations in Queue :-

There major variations made in a simple queue are :-

1) Circular Queue
2) Double ended queue
3) Priority Queue.

1) Circular Queue :-

A circular queue is one in which the insertion of a new element is done at the very first location of the queue is full.



* A Circular queue overcomes the problem of unutilized space in linear queues implemented as arrays.
A Circular queue also have a front and rear to keep the track of the elements to be deleted and inserted.

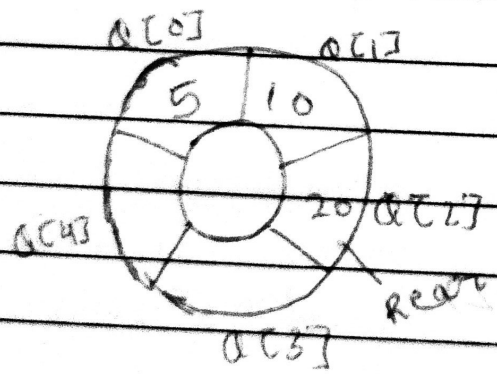
The following assumptions are made:-

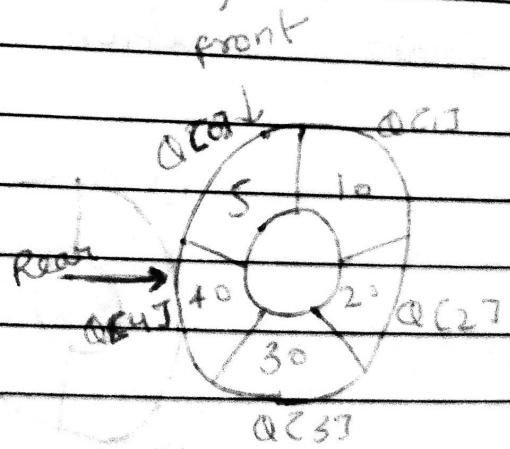
1) Front will always be pointing to the first element (as in linear queue).

2) If (front = rear) the queue is empty

3) new element is inserted
Rear = Rear + 1;

4) Element deleted, front is incremented
front = front + 1;





Insertion :-

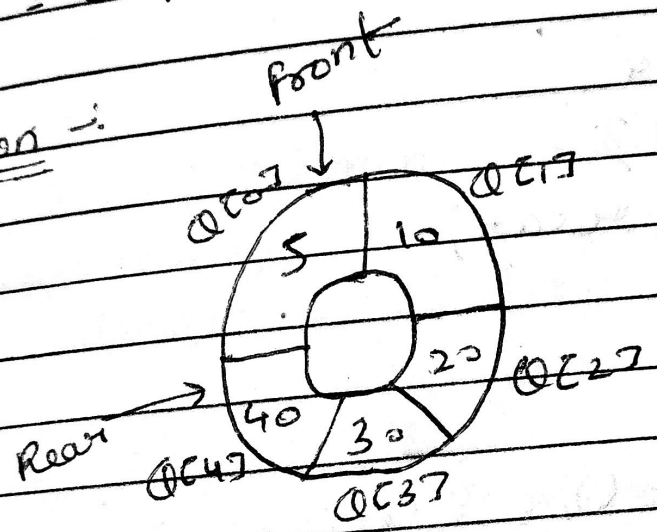
→ The position of the element to be inserted will be calculated by the relation :-

Rear = (Rear + 1) % Max size;
Queue [Rear] = value;

For the current Queue, the value of Rear is 4 and Maxsize is 5, so

→ Rear = (Rear+1) % Maxsize
Rear = (4+1) % 5
= 0 ;

Deletion :-



→ Now, if we delete an element from Queue, front will be incremented by relation :-

{ front = (front +1) % Maxsize; }

* "%" operation is used so, we can create a circular structure.

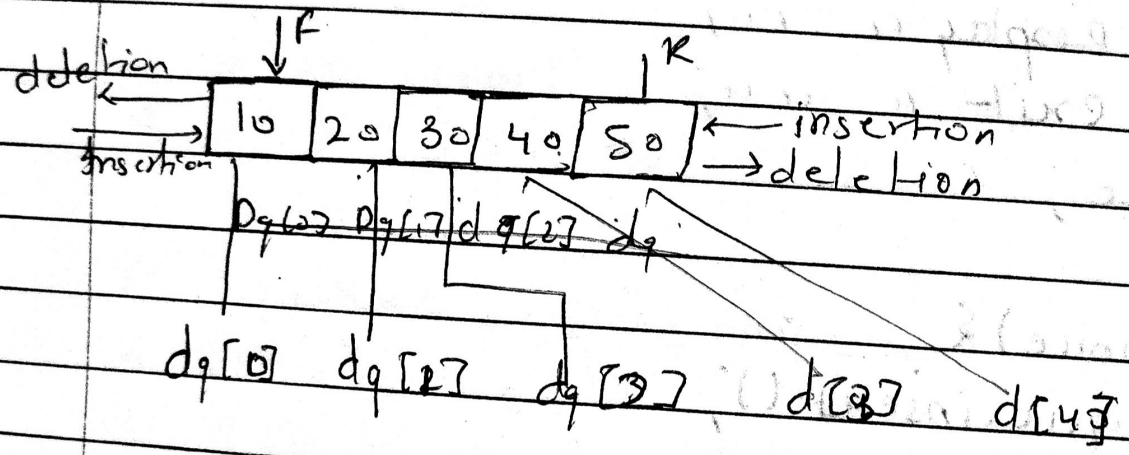
Implementation of Circular Queue using arrays:-

```
#define Maxsize 5
#include <iostream>
using namespace std;

~~int main()~~
class queue
{ int cq[10];
  int front, rear;
  char ch;

Public :
void cqinsert ();
int cqdelete ();
void cqdisplay ();
queue()
{
    front = -1, rear = 0;
}
};
```

```
void queue :: qinsert()
{
int num;
if (front == (rear+1) % Maxsize)
{
  cout << "Queue is full" << endl; return;
}
else {
  cout << " element to be inserted ";
  (in >> num;
  if (front == -1) front = rear = 0;

else {
  rear = (rear +1) % maxsize ; }
  q[rear] = num;
}
return ;
}

//Deletion
int void queue :: qdelete ()
{ int num;
  if (front == -1)
  { cout << " Que is empty ";
  return 0;
}
```

```
else
{ num = cq[front];
  cout << "Deleted element" << cq[front];
  if (front == rear) {
      front = rear = -1;
    }
  else {
      front = (front+1) % Maxsize;
  }
  return (num);
}


void queuer :: cqdisplay ()
{ int i;
  if (front == -1) cout << "Queue empty"; exit();

else
{ for (i=front ; i<=rear ; i++)
    { cout << cq[i] << endl;
    }
}
if (front > rear) {
  for (i=front ; i<maxsize ; i++)
  { cout << cq[i] << endl;
  }
  for (i=0 ; i<=rear ; i++) {
      cout << cq[i] << endl; } } }      //end
```

```
//Driver Code
int main()
{ int choice;
queueq que;

do{
cout<<" 1. insert "<<endl;
cout<<" 2. delete "<<endl;
" " 3. Display " " ;
" " 4. exit " " ";
cin>>choice;

Switch(choice){
case 1: que.insert();
        break;
case 2: que.cqdelete();
        break;
case 3: que.cqdisplay();
        break;
case 4: return 0;
}
}
while (choice != 4);
return 0;
}.
```

★ **Double Ended Queues** —: We can perform insertion and deletion from both the ends. It is also referred as "deque".

There are two types of deques:-
1) Input restricted deques
2) Output restricted deques.