Unit-II Topic-I Queue

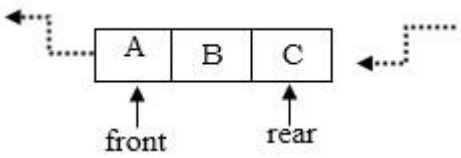
Definition:

- "Queue is a linear data structure which is an ordered collection of elements or items into which elements are inserted from one end called 'rear end' and removed from another end called 'front end'."
- The queue works in FIFO (First In First Out) manner i.e. element which inserted firstly is the first element to come out from queue because queue has two different ends for element insertion and removal.

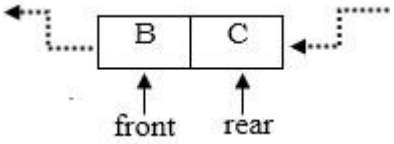
e.g. 1) Queue in front of ATM center. 2) Queue in front of Ticket window of cinema hall.

Working of Queue is shown in following figures:

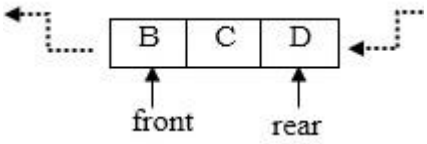
1) After inserting elements A, B and C into queue then it look like as-



2) After removing element from above queue then it look like as-



3) After inserting element D into above queue then it look like as-



Note: For implementation of Queue (By using array i.e. static implementation), we have to define one class as fallow:

```
# define    max    5
class       queue
 {
    int     item[max],front,rear;
 public:
  //declaration of all operations of queue
 };
```

Operations on Queue:

There are some following operations performed on to the queue:

1) create Operation:

This operation creates the new queue. We can say that Queue is created if we assign -1 value to front and rear.

Operation:

```
void    create(queue   *p )
{
      p->front  = -1;
      p->rear = -1;
}
```

2) isempty Operation:

This operation checks whether queue is empty or not. We can say that queue is empty if both ends i.e. front & rear is at same position. (i.e. if the value of front end and rear end are same)

Operation:

```
void    isempty( queue   *p )
{
      if (p->front = = p-> rear)
      {
            cout<<" Queue is empty ";
      }
      else
      {
            cout<<" Queue is NOT empty ";
      }
}
```

3) isfull Operation:

This operation checks whether queue is Full or not. We can say that queue is Full if the value of 'rear' end is *max – 1*.

Operation:

```
void    isfull( queue   *p )
{
      if (p-> rear = = max -1 )
      {
            cout<<" Queue is Full ";
      }
      else
      {
            cout<<" Queue is NOT Full ";
      }
}
```

4) insert or add operation:

This operation is used to add or insert new element into queue. The new element is always inserted at rear end.

Precondition- While performing insert operation, first we have to check queue is **Full or Not** and this is precondition for insert operation.

If queue is full and we are going to perform insert operation at that time element cannot insert into queue i.e. insert operation failed such condition is called "Queue Overflows." *Operation:*

```
void     insert(queue    *p, int    x )
{
      if( p-> rear = = max - 1)
      {
            cout<< " Queue Overflows..";
      }
      else
      {
            + + p-> rear;
            p ->item[p->rear] = x;
      }
}
```

5) remove or delete operation:

This operation is used to remove or delete element from queue. The element is always removed from front end.

Precondition- While performing remove operation, first we have to check queue is **Empty or Not** and this is precondition for remove operation.

If queue is empty and we are going to perform remove operation at that time element cannot removed from queue i.e. remove operation failed such condition is called "Queue Underflows." *O***peration:**

```
int   remove( queue   *p )
{
      if ( p->front = = p->rear)
      {
            cout<< " Queue Underflows..";
      }
      else
      {
            + + p->front;
            return( p-> item[p->front] );
      }
}
```

6) Display (status) operation:

This operation is used to display the elements of the queue onto console.

Operation:

```
void  display(queue   *p )
{
      int     i;
      for( i = p->front+1 ; i<=p->rear ; i++ )
      {
            cout<< "\t"<<p->item[i] ;
      }
}
```

Implementation of Queue (Linear Queue) by using array. (Static Implementation):

```
#include<conio.h>
#include<process.h>
#define  max  100
 class    queue
 {
          int     item[max], front, rear ;
public:
void   create(queue* );
void   isempty(queue* );
void   isfull(queue*);
void   insert(queue*, int);
int     remove(queue*);
void   status(queue*);
 };
void    queue::create(queue   *p )
{
p-> front  = -1;
p-> rear = -1;
cout<< "\nQueue is Created....";
}
void    queue::isempty( queue   *p )
{
   if (p->front = = p->rear)
 {
 cout<<"\n Queue is empty " ;
 }
 else
 {
         cout<<"\nQueue is NOT empty ";
 }
}
void    queue::isfull( queue   *p )
{
 if (p->rear = = max -1 )
 {
         cout<<"\nQueue is Full " ;
 }
 else
 {
     cout<< "\nQueue is NOT Full" ;
 }
}
void     queue::insert(queue    *p, int    x )
 {
 if( p->rear = = max - 1)
 {
   cout<< "\nQueue Overflows..";
```

```
void  queue::status( queue   *p )
{
          int     i;
         for( i = p->front +1 ; i<=p->rear ; i++ )
         {
                  cout<< "\t<<p->item[i];
         }
}

void   main( )
{
queue    *q , p,obj ;
q = &p;   // Initialization of pointer
int    n,r,ch;
clrscr( );
do
{
 cout<<"\nEnter Your choice:" ;
cout<<"\n1:Create\n2:Isemtpy\n3:Isfull\n4:
Insert\n5:Remove\n6:Display\n7:Exit   ";
cin>>ch;
switch(ch)
 {
 case   1:
         obj.create( q );
         break;
 case   2:
         obj.isempty( q);
         break;
case   3:
         obj.isfull( q);
         break;
case   4:
         cout<<"\n Enter any element : ";
         cin>>n;
         obj.insert( q, n);
         break;
case   5:
         r = obj.remove(q);
         cout<<"\n Removed element ="<< r;
         break;
case   6:
         obj.status( q );
         break;
 case   7:
         exit(0);
}
```

```
 }
 else
 {
  + + p->rear ;
  p->item[p->rear] = x;
 cout<< "\nElement is Inserted...";
 }
}
int     queue::remove( queue   *p )
 {
 if ( p->front = = p->rear)
 {
 cout<< " Queue Underflows..";
 }
 else
 {
         + + p-> front ;
         return( p->item[p->front] );
 }
 }
```

```
}while(ch != 7);
getch( );
}
```

Types of Queue:

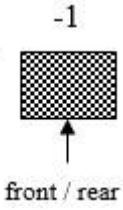
1) Linear Queue:

Definition- "Queue is linear data structure which is collection of elements or items into which elements are inserted from one end called 'rear' end and removed from another end called 'front' end."

Let's see how linear works-queue:

We know that in case of linear queue items are arranged in sequential manner such that *front* position is always less than or equal to *rear* position *(front<=rear)*

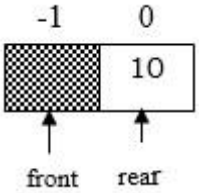
I) Creating queue:



If we want to add new element into queue at that time *rear* end is incremented, while Removing element from queue, *front* end is incremented. Thus front always follows rear end.

Consider, the linear queue having size max = 2 Here,

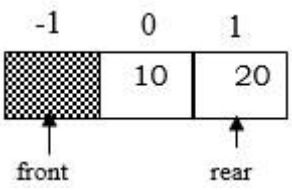
front = = rear = = -1



Here;

II) Inserting element 10 into above queue then it look like as:

front < rear



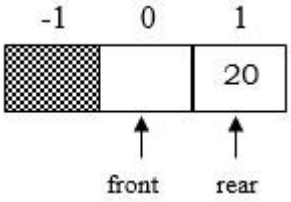
III) Inserting element 20 into above queue then it look like as:

Here;

front < rear

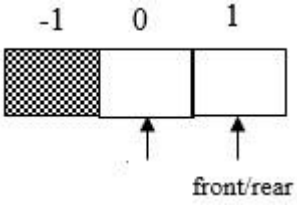
Here, Queue is Full because rear = = max - 1 ..... eq. (1)

IV) Removing element from above Queue then it look like as:



Here; front< rear

V) Removing element from above Queue then it look like as:



Here;
front = = rear

Here, Queue is Empty because front = = rear therefore we are able to insert new element in above queue but from eq. (1) it state that it is Full !

- **Drawback of linear queue:**

1. At one time it state that queue is FULL as well as queue is Empty but this is not possible.
2. It state that queue is Full still there are some empty slots.

- To overcome this drawback of linear queue the concept of circular queue is introduced.

Operations of linear queue-

(Note- all operations of linear queue are same which are discussed on page 1 to page 3.)

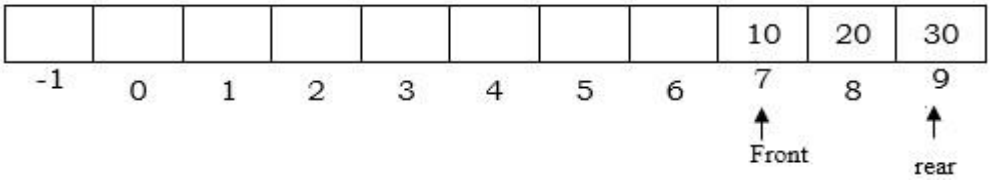
2) Circular Queue:

- **Introduction:**
- We know that, at one time linear queue state that queue is Full as well as Empty but it is not possible and this is drawback of linear queue. To overcome this drawback of linear queue, the concept of circular queue is introduced.
- **Definition-** "Circular Queue is also linear data structure which is collection of elements or items into which elements are inserted from one end called 'rear' end and removed from another end called 'front' end."
- In circular queue all elements physically stored in linear manner but logically it will appear as circle that's why it is called "Circular queue".
- **Advantage of circular queue-**
- The circular queue reported queue is full, if and only if all empty slots are fulfill with elements otherwise it state that queue is still not full.

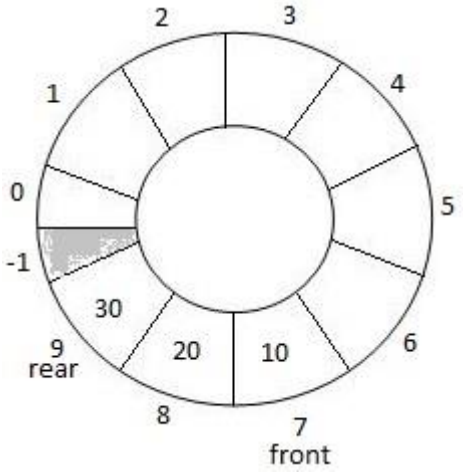
Working of circular queue:

While inserting new element into circular queue, rear end is circularly incremented whereas removing element from circular queue front end is circularly incremented. Both front end & rear end moves clockwise directions in both respective operations.

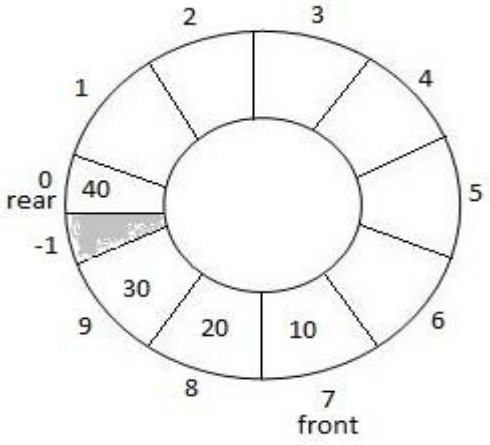
Consider the following linear queue that can stores max = 10 elements. Currently it contains three elements namely 10, 20, 30



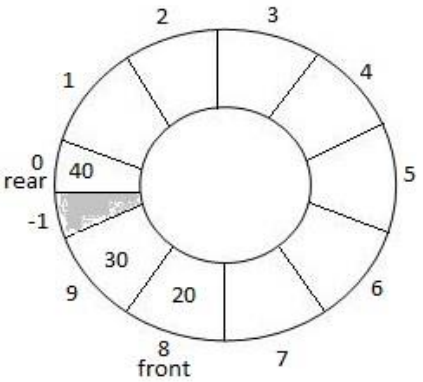
Reporting above queue as full is illegal because there are empty slots from position 0 to 6. If we fill all empty slots with particular element then we can say that queue is full. The above queue can be represented circularly as fallow:



To Insert element '40' into above queue, first 'rear' end is incremented circularly by one position (clockwise direction) and then new element is inserted at 'rear' end therefore it points to 0'th position and resultant queue is look like as follows:



To remove element from above queue, first <u>front end</u> element is <u>stored into temporary variable 'z'</u> & then front incremented circularly by one position. Therefore it points to 8'th position element and then 'z' element is removed from queue. The resultant queue is look like as follows:



Operations of Circular Queue:

For implementation of circular queue (By using array i.e. static implementation), we also have to define one class as fallow:

```
# define    max    5
class       queue
{
      int     item[max], front, rear;
      public:
      // declaration of operations
};
```

1) create operation:

This operation is used to create the circular queue. We can also say that circular queue is created if we assign -1 value to 'front' and 'rear'.

Operation-

```
void       create( queue     *p )
{
      p->front = -1;
      p->rear = -1;
 }
```

2) isempty Operation:

This operation checks whether queue is empty or not. We can say that circular queue is empty if value of front end is -1.

Operation:

```
void    isempty( queue   *p )
{
      if (p->front = = -1)
      {
            cout<<" Queue is empty ";
      }
      else
      {
            cout<<" Queue is NOT empty " ;
      }
}
```

3) isfull Operation:

This operation checks whether queue is Full or not.

We can say that circular queue is full if it satisfies one of following conditions-

- If 'front' is at 0^{th} position and at same time 'rear' is at last position i.e. at 'max-1'
- If 'front' is ahead of 'rear' by position 1.

Operation:

```
void    isfull( queue   *p )
{
       if ( (p->front==0 && p-> rear = = max -1 ) || (p->front==p->rear+1))
      {
            cout<<" Queue is Full ";
      }
      else
      {
            cout<<" Queue is NOT Full " ;
      }
}
```

4) insert or add operation:

This operation is used to insert new element into circular queue. Also new element is inserted at rear end.

While performing this operation, check queue is full or not.

If it is full then "queue overflow" occurs otherwise insert element into queue at 'rear' end.

Operation:

```
 void   insert ( queue *p, int  x)
{
  if ( (p->front = = 0 && p->rear = = max-1) || (p->front = = p->rear + 1) )
   {
            cout<< " Queue overflows…";
            return ;
   }
   else
      {
                  if ( p->front = = -1)
                  {
                         p->front = p->rear = 0;
                  }
                  else
                  {
                      p->rear =  (p->rear + 1) % max ;    //increments 'rear' circularly.
                  }

                  p->item[p->rear] = x ;
                  cout<<" Element is inserted…..";
       }
}
```

5) remove or delete operation:

This operation is used to remove the element from the queue. Also element is removed from front end.

While performing this operation, check queue is empty or not.

If it is empty then "queue underflow" occurs otherwise remove element from queue from 'front' end.

Operation:

```
int  remove ( queue    *p)
     {
          int      z;
          if ( (p->front = =  -1 )
          {
               cout<< " Queue Underflows…");
               return(0) ;
          }
          else
          {
               z = p->item[p->front];
               if ( p->front = = p->rear)
               {
                    p->front = p->rear = -1 ;
               }
               else
               {
               p->front =  (p->front + 1) % max ;  //increments 'front' circularly.
               }
               return(z) ;
          }
     }
```

6) Display (Status) operation:

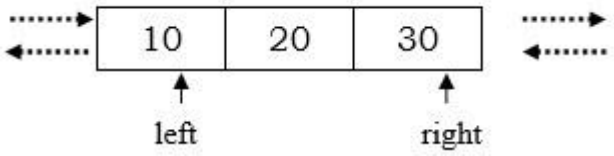
This operation is used display all the elements of circular queue.

```
      void      display( queue    *p)
     {
          int   i;
          for( i = p->front ; i != p->rear ;  i = (i+1) % max )
          {
               cout<<"\t"<< p->item[i];
          }
          cout<< "\t"<< p->item[p->rear];
     }
```

Implementation of Circular Queue: (Home work)

3) Double Ended Queue (Deque):

The queue in which, we can insert & remove elements from both (either) ends such queue is called as "Double Ended Queue (Deque)" Following figure shows Deque:



Following are some operations that can be carried out onto Deque:

1) Insert Right
2) Insert Left
3) Remove Right
4) Remove Left

Types of Deque:

Deque can be of two types viz:

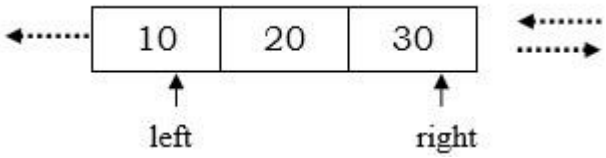
I) IRD (Input Restricted Deque)

II) ORD (Output Restricted Deque)

Let's see these types in briefly…

I) IRD (Input Restricted Deque):

The Deque in which, the element is inserted from only one end and removed from both ends, such Deque is called as "IRD" Following figure shows IRD:



Following are some basic operations that can be carried out on IRD:

1) Insert
2) Remove Left
3) Remove Right

For implementation of IRD we can define one class which is as follows:

```
#define          max 5
class      ird
{
      int   item[max], left, right;
  public:
      // declaration of operations
};
```

Operations of IRD:

1) Create operation:

This operation is used to create new IRD. We can say that IRD is created, if we assign -1 value to right and left.

Operation:

```
void  create(ird    *q)
{
      q->left = q->right = -1;
}
```

2) Insert operation:

This operation is used to insert new element into IRD. Always element is inserted at the right end.

Operation:

```
void  insert(ird   *q, int   x)
{
      if( q ->right = = max -1 )
      {
            cout<<"IRD overflows…";
      }
      else
      {
            + + q-> right;
            q->item[q->right] = x;
      }
}
```

3) Remove Left operation:

This operation is used to remove the left most element from IRD.

Operation:

```
int   rem_left(ird   *q)
 {
      if( q ->right = = q->left )
      {
            cout<<"IRD underflows…";
      }
      else
      {
            + + q-> left;
            return(q->item[q->left]);
       }
      }
```

4) Remove Right operation:

This operation is used to remove the right most elements from IRD.

Operation:

```
int   rem_right(ird   *q)
{
      if( q ->right = = q->left )
      {
            cout<<"IRD underflows…";
      }
      else
```

```
        {
            return(q->item[q->right - -]);
        }
    }
```

5) Display or Status operation:

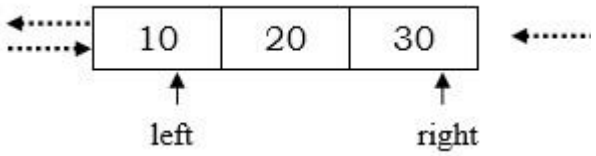
This operation is used to display the elements of IRD.

```
void  display(ird   *q)
{
    int  i;
    for( i = q -> left +1; i < = q->right ; i + + )
    {
        cout<<"\t"<<q->item[i];
    }
}
```

Implementation of IRD: ***(Home work)***

II) ORD (Output Restricted Deque):

The Deque in which, the element is removed from only one end and inserted from both ends, Such Deque is called as "ORD" Following figure shows ORD:



Following are some basic operations that can be carried out on ORD:

1) Insert Left
2) Insert Right
3) Remove

For implementation of ORD we can define one class which is as follows:

```
#define          max 5
class      ord
{
    int  item[max], left, right;
    public:
        // declaration of operations
};
```

Operations of ORD:

1) Create operation:

This operation is used to create the new ORD. We can say that ORD is created, if we assign -1 value to right and left.

Operation:

```
void create(ird    *q)
{
    q->left = q->right = -1;
}
```

2) Insert Left operation:

This operation is used to insert new element into ORD from left end.

Operation:

```
void  ins_left(ord  *q , int    x)
{
        int   i ;
        if(q->right = = max-1)
       {
                 cout<<"\n ORD Overflows....";
       }
      else
     {
           for( i = q->right+1 ; i > = q->left+2 ; i - -)
           {
             q->item[i] = q->item[i-1];    // shift all elements at right side by 1 position
            }
             q->item[q->left + 1] = x;
            + + q->right;
             cout<<"\n Element is inserted from left...";
       }
 }
```

3) Insert Right operation:

This operation is used to insert new element into ORD from right end.

Operation:

```
          void  ins_right(ord    *q , int    x)
          {
               if(q->right = = max -1 )
                {
                     cout<<"\n ORD Overflows...";
                 }
                 else
                 {
                      + + q->right;
                      q ->item[q ->right] = x;
                     cout<<"\n element is inserted from right side ...";
                 }
           }
```

4) Remove Operation:

This operation is used to remove the element from ORD. Always leftmost element is removed from ORD.

Operation:

```
int   rem_left(ord   *q)
{
          if( q ->right = = q->left )
          {
               cout<<"\nORD underflows…";
           }
```

```
            else
            {
                  + + q-> left;
                  return(q->item[q->left]);
            }
}
```

5) Display operation:

This operation is used to display the elements of ORD.

Operation:

```
void display(ord   *q)
{
      int   i;
      for( i = q ->left+1; i < = q->right ; i + + )
      {
            cout<<"\t"<<q->item[i];
      }
}
```

Implementation of ORD: (Home work)

4) Priority Queue:

- Priority queue is another type of queue in which every element of queue has some priority and based on that priority elements are processed. The element which have high priority will be processed firstly before the element which has less priority. If two or more elements have same priority at that time FIFO rule will be applied i.e. the element which come firstly will processed firstly.
- In case of **priority queue** intrinsic ordering (FIFO) of element does not determine the result of its basic operation.
- An example of priority queue can be time sharing system, in that programs of high priority are processed firstly and programs with low priority are processed lastly.

Consider the following table:

Element	Priority	Order of Insertion
A	5	3
B	9	4
C	7	1
D	5	2

In above table B element has highest priority 9 therefore it will process firstly. After B, the C element will processed because it has second highest priority 7. After C, The element A and D have same priority in such cases FIFO ordering is applicable i.e. here element D is come firstly than element A therefore D will processed firstly than element A.

Types of Priority Queue:

I) Ascending Priority Queue:

In ascending priority queue all elements are arranged in sequential manner but in ascending order therefore only smallest element among all elements will processed firstly i.e. small element have highest priority.

II) Descending Priority Queue:

In Descending priority queue all elements are arranged in sequential manner but in descending order therefore only largest or highest element among all elements will processed firstly i.e. largest or highest element have highest priority.

```
// Implementation of Priority Queue
#include<iostream.h>
#include<conio.h>
#include<process.h>
#define max 5
class queue
{
   int item[max],pri[max],front,rear;
  public:
        void create(queue*);
        void insert(queue*,int,int);
        int remove(queue*);
        void display(queue*);
};
void queue::create(queue *q)
{
  q->front=q->rear=-1;
  cout<<"\nQueue is created...";
}
void queue::insert(queue *q,int x,int p)
{
 if(q->rear==max-1)
 {
   cout<<"\n Queue Overflows...";
 }
 else
 {
    ++q->rear;
    q->item[q->rear]=x;
    q->pri[q->rear]=p;
    cout<<"\nElement is inserted...";
 }
}
int queue::remove(queue *q)
{
  int m,pos=0,i,z;
  if(q->front==q->rear)
  {
    cout<<"\nQueue underflows...";
    return(0);
  }
  else
  {
    m=q->pri[0];
    for(i=q->front+1;i<=q->rear;i++)
    {
        if(m<q->pri[i])
        {
           m=q->pri[i];
```

```
void queue::display(queue *q)
{
   int i;
   cout<<"\nPriority: ";
   for(i=q->front+1;i<=q->rear;i++)
   {
     cout<<"\t"<<q->pri[i];
   }
   cout<<"\nElements: ";
   for(i=q->front+1;i<=q->rear;i++)
   {
     cout<<"\t"<<q->item[i];
   }
}
void main()
{
 clrscr();
 int ch,n,x,pr;
 queue obj,p,*q;
 q=&p;
 do
 {

cout<<"\n1:Create\n2:Insert\n3:Remove\n4:
Display\n5:Exit :";
   cin>>ch;
   switch(ch)
   {
        case 1:
                obj.create(q);
                break;
        case 2:
                cout<<"\nEnter ele= ";
                cin>>n;
                cout<<"\nEnter Pri=";
                cin>>pr;
                obj.insert(q,n,pr);
                break;
        case 3:
                x=obj.remove(q);
                cout<<"\nRemoved Val="<<x;
                break;
        case 4:
                obj.display(q);
                break;
        case 5:
                exit(0);
   }
 }while(ch!=5);
 getch();
```

```
            pos=i;
          }
    }
    z=q->item[pos];
    for(i=pos;i<=q->rear;i++)
    {
          q->item[i]=q->item[i+1];
          q->pri[i]=q->pri[i+1];
    }
    --q->rear;
    return(z);
  }
}
```

```
}
```

Applications of Queue:

1) Queues are used in computers for scheduling of resources to applications. These resources are CPU, Printer etc.
2) In batch programming, multiple jobs are combined into a batch and the first program is executed first, the second is executed next and so on in sequential manner.
3) Multiple print jobs given to a printer are organized in FIFO manner in print queue and the first print request is printed firstly, the second print request printed nextly and so on in sequential manner.
4) The priority queue is used in time sharing operating system, where every program has some priority and those programs have high priority will processed firstly than the programs having low priority.

Difference between Stack and Queue:

Stack	Queue
1) Stack is data structure which is collection elements into which items are inserted and removed from only one end called top of stack.	1) Queue is data structure which is collection elements into which items are inserted from one end called rear end and removed from another end called front end.
2) Stack works in **LIFO** manner	2) Queue works in **FIFO** manner.
3) Stack is empty if its top is -1	3) Queue is empty if both ends i.e. front and rear is at same Position.
4) Stack has only one end called **top**	4) Queue has two ends called **rear and front**.
5) New element is added at top.	5) New element is added from rear end.
6) Element is removed from top end	6) Element is removed from front end.
7) Basic operations are push and pop	7) Basic operations are insert and remove

Theory Assignment- 3

1) What is Queue? Explain its basic operations.
2) Why Queue is called FIFO data structure? List out its applications.
3) Define “Queue”. List out its type with definition.
4) What is linear queue? Explain preconditions for insert and remove operations.
5) What is drawback of linear queue?
6) What is Circular queue? How circular queue overcomes the drawback of linear queue.
7) Explain the insert and remove operations of circular queue.
8) What is Priority Queue? Explain it with its types.
9) What is Deque? List out its types with definition.
10) What is IRD? Explain its basic operations.
11) What is ORD? Explain its basic operations.
12) Differentiate between Stack and Queue.

Practical Assignment- 3

1) Write a program to implement linear queue by using array. (Static Implementation of queue)
2) Write a program to implement Circular queue.
3) Write a program to implement IRD (Input Restricted Deque)
4) Write a program to implement ORD (Output Restricted Deque)
5) Write a program to implement Priority queue.