[UNIT-2]
"Queues"

▪ QUEUES

Queue is a non-primitive linear data structure that permits insertion of an element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called **front**, and the end at which insertion of a new element can take place is called **rear**. The deletion or insertion of elements can take place only at the front and rear end of the list respectively.

The first element that gets added into the queue is the first one to get removed from the list. Hence, Queue is also referred to as ***First-In-First-Out (FIFO)*** list. The name '*Queue*' comes from the everyday use of the term. Consider a railway reservation booth, at which we have to get into the reservation queue. New customers got into the queue from the rear end, whereas the customers who get their seats reserved leave the queue from the front end. It means the customers are serviced in the order in which they arrive the service center (i.e. first come first serve type of service). The same characteristics apply to our *Queue*. Fig. 1. shows the pictorial representation of a Queue.

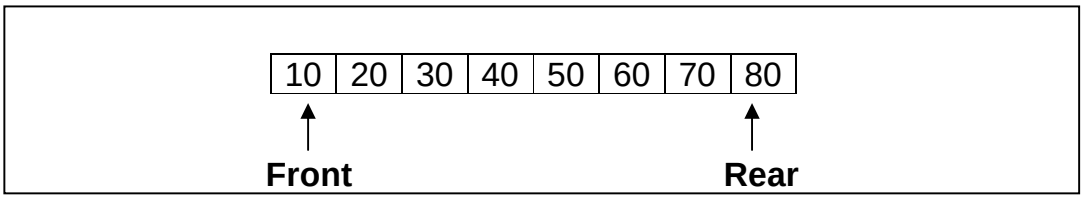

Fig. (1) : Pictorial representation of a Queue

In fig (1), **10** is the first element and **80** is the last element added to the Queue. Similarly, **10** would be the first element to get removed and **80** would be the last element to get removed.

Figures 2(a) to 2(d) shows queue graphically during insertion operation :

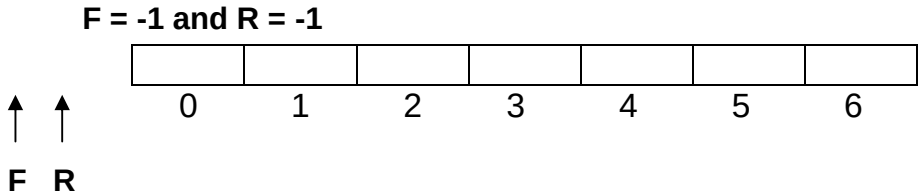

Fig. 2(a) Empty Queue

F = 0 and R = 0

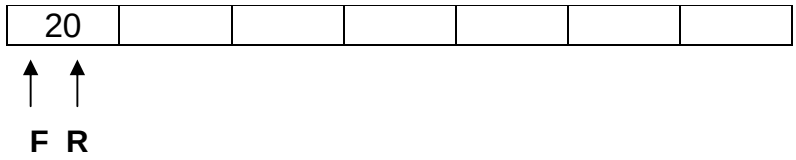

Fig. 2(b) One Element Queue

F = 0 and R = 1

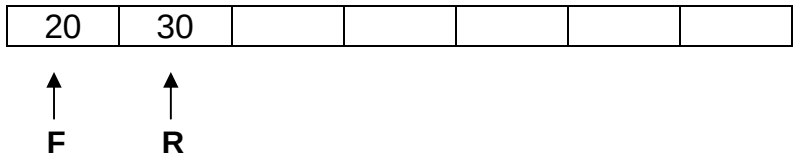

Fig. 2(c) Two Element Queue

F = 0 and R = 2

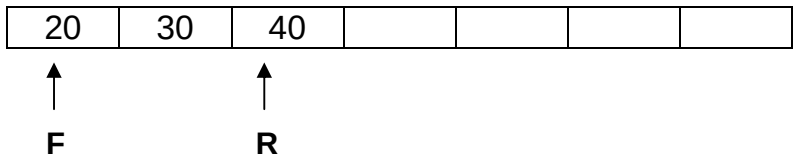

Fig. 2(d) Three Element Queue

It is clear from the above figures that whenever we insert an element in the queue, the value of Rear is incremented by one i.e.

Rear = Rear + 1

Also, during the insertion of the first element in the queue we always incremented the Front by one i.e.

Front = Front + 1

Afterwards the Front will not be changed during the entire operation. The following figures show Queue graphically during deletion operation :

F = 1 and R = 2

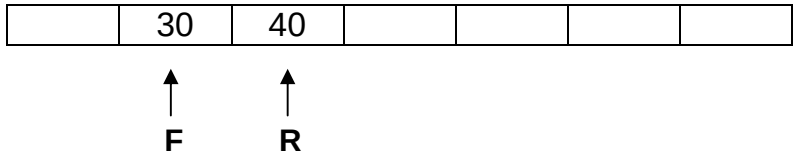

Fig. 2(e) One Element (20) Deleted from Front

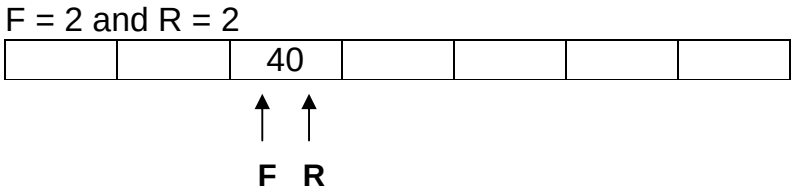

Fig. 2(f) Second Element (30) Deleted from Front

This is clear from Fig. 2(e) and 2(f), that whenever an element is removed from the queue, the value of Front is incremented by one i.e.,

Front = Front + 1

Now, if we insert any element in the queue, the queue will look like :

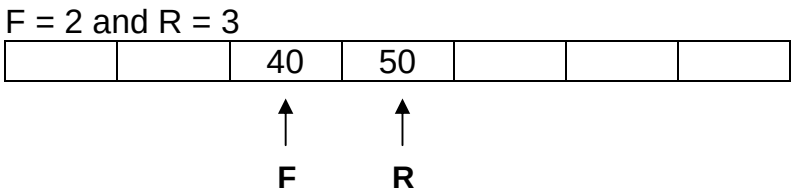

Fig. 2(g) Insertion after Deletion

Sequential implementation of Linear queues

Queues can be implemented in two ways :

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

Static implementation :

Static implementation of Queue is represented by arrays. If Queue is implemented using arrays, 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. In this case, the beginning of the array will become the *front* for the queue and the last location of the array will act as *rear* for the queue. Fig. (3) shows the representation of a queue as an array.

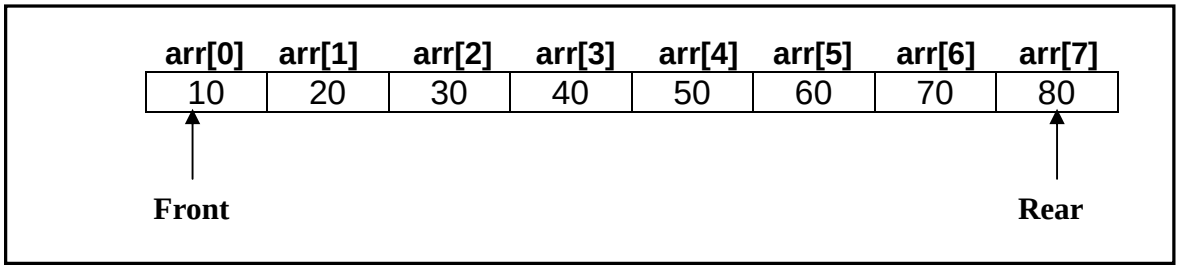

Fig. (3) Representation of a Queue as an array

The following relation gives the total number of elements present in the queue, when implemented using arrays :

rear – front + 1

Also note that if *front* > *rear*, then there will be no element in the queue or queue is empty.

OPERATIONS ON A QUEUE

The basic operations that can be performed on queue are :

1. To **Insert** an element in a Queue
2. To **Delete** an element from a Queue.
3. To **Traverse** all elements of a Queue.

ALGORITHMS & FUNCTIONS FOR INSERTION AND DELETION IN A LINEAR QUEUE (USING ARRAYS)

(1) Algorithm for Insertion in a Linear Queue

Let QUEUE[MAXSIZE] is an array for implementing the Linear Queue & NUM is the element to be inserted in linear queue, FRONT represents the index number of the element at the beginning of the queue and REAR represents the index number of the element at the end of the Queue.

```
Step 1 :If REAR  = (MAXSIZE –1)  : then
            Write : “Queue Overflow” and return
           [End of If structure]
Step 2 : Read NUM to be inserted in Linear Queue.
Step 3 : Set REAR := REAR + 1
Step 4 : Set QUEUE[REAR] := NUM
Step 5 : If FRONT = –1 : then
             Set FRONT=0.
          [End of If structure]
Step 6 : Exit
```

Function for insertion in a linear queue (using arrays)

```
void lqinsert()
{
int num;
 if(rear==MAXSIZE-1)
 {
   printf("\nQueue is full (Queue overflow)");
   return;
 }
 printf("\nEnter the element to be inserted : ");
 scanf("%d",&num);
 rear++;
 queue[rear]=num;
 if(front==-1)
  front=0;
}
```

(2) Algorithm for Deletion from a Linear Queue

Let QUEUE[MAXSIZE] is an array for implementing the Linear Queue & NUM is the element to be deleted from linear queue, FRONT represents the index number of the element at the beginning of the queue and REAR represents the index number of the element at the end of the Queue.

Step 1 : If FRONT = -1 : then
Write : “Queue Underflow” and return
[End of If structure]
Step 2 : Set NUM := QUEUE[FRONT]
Step 3 : Write “Deleted item is : ”, NUM
Step 4 : Set FRONT := FRONT + 1.
Step 5 : If FRONT=REAR : then
Set FRONT := REAR := -1.
[End of If structure]
Step 6 : Exit

Function(Procedure) for Deletion from a Linear Queue

```
void lqdelete()
{
 if(front == -1)
 {
  printf("\nQueue is empty (Queue underflow)");
  return;
 }
 int num;
 num=queue[front];
 printf("\nDeleted element is : %d",num);
 front++;
 if(front==rear)
  front=rear=-1;
}
```

Program 1 : Static implementation of Linear Queues using arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5
void initialize();
void lqinsert();
void lqdelete();
void lqtraverse();
int queue[MAXSIZE];
int front,rear;

void main()
{
clrscr();
initialize();
int choice;
 while(1)
 {
  clrscr();
  printf("\nSTATIC IMPLEMENTATION OF LINEAR QUEUE");
  printf("\n-----------------------------------");
  printf("\n1. Insert");
  printf("\n2. Delete");
  printf("\n3. Traverse");
  printf("\n4. Exit");
  printf("\n-----------------------------------");
  printf("\n\nEnter your choice [1/2/3/4] : ");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1 : lqinsert();
             break;
  case 2 : lqdelete();
             break;
  case 3 : lqtraverse();
             break;
  case 4 : exit(0);
  default : printf("\nInvalid choice");
  }
 getch();
 }
}
```

```
// Function to initialize queue

void initialize()
{
 front=rear=-1;
}

// Function to insert an element into queue

void lqinsert()
{
int num;
 if(rear==MAXSIZE-1)
 {
   printf("\nQueue is full (Queue overflow)");
   return;
 }
 printf("\nEnter the element to be inserted : ");
 scanf("%d",&num);
 rear++;
 queue[rear]=num;
 if(front==-1)
  front=0;
}

// Function for Delete an element from queue

void lqdelete()
{
 if(front==-1)
 {
  printf("\nQueue is empty (Queue underflow)");
  return;
 }
 int num;
 num=queue[front];
 printf("\nDeleted element is : %d",num);
 front++;
 if(front==rear)
  front=rear=-1;
}
```

1. // Function to display Queue

```
void lqtraverse()
{
 if(front==-1)
 {
  printf("\nQueue is empty (Queue underflow)");
  return;
 }
 else
 {
  printf("\nQueue elements are : \n");
  for(int i=front;i<=rear;i++)
   printf("%d\t",queue[i]);
 }
}
```

DYNAMIC IMPLEMENTATION OF LINEAR QUEUE

ALGORITHM FOR INSERTION AND DELETION IN A LINEAR QUEUE (USING POINTERS)

Let queue be a structure whose declarations looks like follows :

```
struct queue
{
int info;
struct queue *link;
}*start=NULL;
```

ALGORITHMS FOR INSERTION & DELETION IN A LINEAR QUEUE FOR DYNAMIC IMPLEMENTATION USING LINKED LIST

(1) Algorithm for inserting an element in a Linear Queue :

Let PTR is the structure pointer which allocates memory for the new node & NUM is the element to be inserted into linear queue, INFO represents the information part of the node and LINK represents the link or next pointer pointing to the address of next node. FRONT represents the address of first node, REAR represents the address of the last node. Initially, Before inserting first element in the queue, FRONT=REAR=NULL.

Step 1 : Allocate memory for the new node using PTR.
Step 2 : Read NUM to be inserted into linear queue.
Step 3 : Set PTR->INFO = NUM
Step 4 : Set PTR->LINK= NULL
Step 5 : If FRONT = NULL : then
Set FRONT=REAR=PTR
Else
Set REAR->LINK=PTR;
Set REAR=PTR;
[End of If Else Structure]
Step 6 : Exit

Function(Procedure) for Inserting an element in a Linear Queue :

```
void lqinsert()
{
 struct queue *ptr;
 int num;
 ptr=(struct queue*)malloc(sizeof(struct queue));
 printf("\nEnter element to be inserted in queue : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=NULL;
 if(front==NULL)
 {
  front=ptr;
  rear=ptr;
 }
 else
 {
  rear->link=ptr;
  rear=ptr;
 }
}
```

(2) Algorithm for Deleting a node from a Linear Queue :

Let PTR is the structure pointer which deallocates memory of the first node in the linear queue & NUM is the element to be deleted from queue, INFO represents the information part of the deleted node and LINK represents the link or next pointer of the deleted node pointing to the address of next node. FRONT represents the address of first node, REAR represents the address of the last node.

Step 1 : If FRONT = NULL : then
Write 'Queue is Empty(Queue Underflow)' and return.
[End of If structure]
Step 2 : Set PTR = FRONT
Step 3 : Set NUM = PTR->INFO
Step 4 : Write 'Deleted element from linear queue is : ',NUM.
Step 5 : Set FRONT = FRONT->LINK
Step 6 : If FRONT = NULL : then
Set REAR = NULL.
[End of If Structure].
Step 7 : Deallocate memory of the node at the beginning of queue using PTR.
Step 8 : Exit.

Function(Procedure) for deleting a node from a Linear Queue

```
void lqdelete()
{
if(front==NULL)
 {
  printf("\nQueue is empty (Queue underflow)");
  return;
 }
  struct queue *ptr;
  int num;
  ptr=front;
  num=ptr->info;
  printf("\nThe deleted element is : %d",num);;
  front=front->link;
  if(front==NULL)
   rear=NULL;
  free(ptr);
 }
}
```

Program 2 : Dynamic implementation of linear queue using pointers

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct queue
{
int info;
struct queue *link;
}*front,*rear;

void initialize();
void lqinsert();
void lqdelete();
void lqtraverse();

void main()
{
int choice;
initialize();
 while(1)
 {
  clrscr();
  printf("\nDYNAMIC IMPLEMENTATION OF LINEAR QUEUE");
  printf("\n1. Insert");
  printf("\n2. Delete");
  printf("\n3. Traverse");
  printf("\n4. Exit");
  printf("\n\nEnter your choice [1/2/3/4] : ");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1: lqinsert();
            break;
  case 2: lqdelete();
            break;
  case 3: lqtraverse();
            break;
  case 4: exit(0);;
  default : printf("\nInvalid choice");
  }
 getch();
}
}
```

```
// Function for initialize linear Queue

void initialize()
{
 front=rear=NULL;
}


// Function to insert element in Linear queue

void lqinsert()
{
 struct queue *ptr;
 int num;
 ptr=(struct queue*)malloc(sizeof(struct queue));
 printf("\nEnter element to be inserted in queue : ");
 scanf("%d",&num);
 ptr->info=num;
 ptr->link=NULL;
 if(front==NULL)
 {
  front=ptr;
  rear=ptr;
 }
 else
 {
  rear->link=ptr;
  rear=ptr;
 }
}


// Function to delete element from Linear queue

void lqdelete()
{
  if(front==NULL)
  {
   printf("\nQueue is empty (Queue underflow)");
   return;
  }
```

```
  struct queue *ptr;
  int num;
  ptr=front;
  num=ptr->info;
  printf("\nThe deleted element is : %d",num);;
  front=front->link;
  if(front==NULL)
   rear=NULL;
  free(ptr);
}

// Function to display Linear Queue
void lqtraverse()
{
struct queue *ptr;
if(front==NULL)
{
 printf("\nQueue is empty (Queue underflow)");
 return;
}
else
{
ptr=front;
printf("\n\nQueue elements are : \n");
printf("\nROOT");
 while(ptr!=NULL)
 {
  printf(" -> %d",ptr->info);
  ptr=ptr->link;
 }
 printf(" -> NULL");
}
}
```

CIRCULAR QUEUES

The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full (since rear has reached the end of the array), even though in actuality there might be empty slots at the beginning of the queue. To overcome this limitation we can implement the queue as a circular queue. Here as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot the array (provided it is free). Suppose an array **arr** of **n** elements is used to implement a circular queue we may reach **arr[n-1]**. We

cannot add any more elements to the queue since we have reached at the end of the array. Instead of reporting the queue as full, if some elements in the queue have been deleted then there might be empty slots at the beginning of the queue. In such a case these slots would be filled by new elements being added to the queue. In short just because we have reached the end of the array, the queue would not be reported as full. The queue would be reported as full only when all the slots in the array stand occupied. Figure (4) shows the pictorial representation of a circular queue.

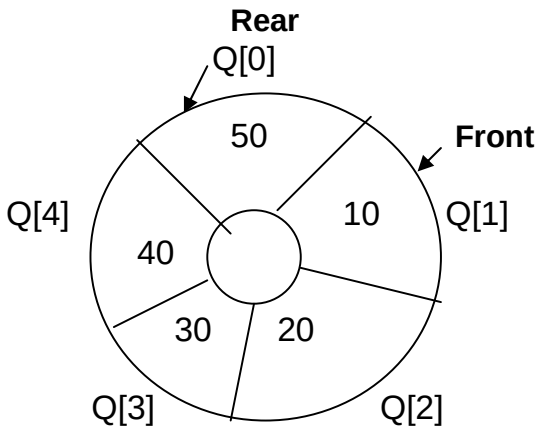

Fig. (4) : Pictorial representation of a circular queue

ALGORITHM FOR INSERTION AND DELETION IN A CIRCULAR QUEUE (USING ARRAYS)

(1) Algorithm for Insertion in a Circular Queue

Let CQUEUE[MAXSIZE] is an array for implementing the Circular Queue, where MAXSIZE represents the max. size of array. NUM is the element to be inserted in circular queue, FRONT represents the index number of the element at the beginning of the queue and REAR represents the index number of the element at the end of the Queue.

```
Step 1  : If FRONT  = (REAR + 1) % MAXSIZE  : then
            Write : "Queue Overflow" and return.
          [End of If structure]
Step 2  : Read NUM to be inserted in Circular Queue.
Step 3 : If FRONT= -1 : then
            Set FRONT = REAR =0.
          Else
            Set REAR=(REAR + 1) % MAXSIZE.
          [End of If Else structure]
Step 4 : Set CQUEUE[REAR]=NUM;
Step 5  : Exit
```

Function(Procedure) for Insertion in a Circular Queue using arrays:

```
void cqinsert()
{
 int num;
 if(front==(rear+1)%MAXSIZE)
 {
  printf("\nQueue is Full(Queue overflow)");
  return;
 }
 printf("\nEnter the element to be inserted in circular queue : ");
 scanf("%d",&num);
 if(front==-1)
    front=rear=0;
 else
    rear=(rear+1) % MAXSIZE;
 cqueue[rear]=num;
}
```

(2) Algorithm for Deletion from a Linear Queue :

Let CQUEUE[MAXSIZE] is an array for implementing the Circular Queue, where MAXSIZE represents the max. size of array. NUM is the element to be deleted from circular queue, FRONT represents the index number of the first element inserted in the Circular Queue and REAR represents the index number of the last element inserted in the Circular Queue.

```
Step 1  : If FRONT  = - 1 : then
              Write : “Queue Underflow” and return.
           [End of If Structure]
Step 2 : Set NUM = CQUEUE[FRONT].
Step 3 : Write ‘Deleted element from circular queue is : ",NUM.
Step 4 : If FRONT = REAR : then
             Set FRONT = REAR = -1;
           Else
             Set FRONT = (FRONT + 1) % MAXSIZE.
Step 5 : Exit
```

Function(Procedure) to Delete an element from a Queue

```
void cqdelete()
{
 int num;
 if(front==-1)
 {
  printf("\nQueue is Empty (Queue underflow)");
  return;
 }
  num=cqueue[front];
  printf("\nDeleted element from circular queue is : %d",num);

  if(front==rear)
   front=rear=-1;
  else
   front=(front+1)%MAXSIZE;
}
```

Program 3 : Static implementation of Circular queue using arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5
void cqinsert();
void cqdelete();
void cqdisplay();

int cqueue[MAXSIZE];
int front=-1,rear=-1;
void main()
{
int choice;
while(1)
{
 clrscr();
 printf("\nSTATIC IMPLEMENTATION OF CIRCULAR QUEUE");
 printf("\n-------------------------------------");
 printf("\n1. Insert");
 printf("\n2. Delete");
 printf("\n3. Traverse");
 printf("\n4. Exit");
 printf("\n-------------------------------------");
```

```
 printf("\n\nEnter your choice [1/2/3/4] : ");
 scanf("%d",&choice);
 switch(choice)
 {
  case 1 : cqinsert();
           break;
  case 2 : cqdelete();
           break;
  case 3 : cqdisplay();
           break;
  case 4 : exit(0);
  default : printf("\nInvalid choice");
 }
 getch();
 }
}
```

// Function to insert element in the Circular Queue

```
void cqinsert()
{
 int num;
 if(front==(rear+1)%MAXSIZE)
 {
  printf("\nQueue is Full(Queue overflow)");
  return;
 }
 printf("\nEnter the element to be inserted in circular queue : ");
 scanf("%d",&num);
 if(front==-1)
    front=rear=0;
 else
    rear=(rear+1) % MAXSIZE;
 cqueue[rear]=num;
}
```

// Function to delete element from the circular queue

```
void cqdelete()
{
 int num;
 if(front==-1)
 {
  printf("\nQueue is Empty (Queue underflow)");
  return;
 }
```

```
  num=cqueue[front];
  printf("\nDeleted element from circular queue is : %d",num);

  if(front==rear)
   front=rear=-1;
  else
   front=(front+1)%MAXSIZE;
}
```

// Function to display circular queue

```
void cqdisplay()
{
 int i;
 if(front==-1)
 {
  printf("\nQueue is Empty (Queue underflow)");
  return;
 }
 printf("\n\nCircular Queue elements are : \n");
 for(i=front;i<=rear;i++)
   printf("\ncqueue[%d] : %d",i,cqueue[i]);
 if(front>rear)
 {
  for(i=0;i<=rear;i++)
   printf("cqueue[%d] : %d\n",i,cqueue[i]);
  for(i=front;i<MAXSIZE;i++)
   printf("cqueue[%d] : %d\n",i,cqueue[i]);
 }
}
```

Advantages of Circular queue over linear queue :

In a linear queue with max. size 5, after inserting element at the last location (4) of array, the elements can't be inserted, because in a queue the new elements are always inserted from the rear end, and rear here indicates to last location of the array (location with subscript 4) even if the starting locations before front are free. But in a circular queue, if there is element at the last location of queue, then we can insert a new element at the beginning of the array.

PRIORITY QUEUE

A **priority queue** is a collection of elements where the elements are stored according to their priority levels. The order in which the elements get added or removed is decided by the priority of the element.

Following **rules** are applied to maintain a priority queue :

(1) The element with a higher priority is processed before any element of lower priority.
(2) If there are elements with the same priority, then the element added first in the queue would get processed.

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first. Another application of Priority queues is simulation systems where priority corresponds to event times.

There are mainly two ways of maintaining a priority queue in memory. One uses a one-way list, and the other uses multiple queues. The ease or difficultly in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-way List Representation of a Priority Queue :

One way to maintain a priority queue in memory is by means of a one-way list, as follows :

(a) Each node in the list will contain three items of information; an information field INFO, a priority number PRN and a link number LINK.
(b) A node X precedes a node Y in the list
  (I) When X has higher priority then Y and
  (II) When both have the same priority but X is added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Priority queues will operate in the usual way : the lower the priority number, the higher the priority.

Array representation of a Priority Queue :

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.

Out of these two ways of representing a Priority Queue, the array representation of a priority queue is more time-efficient than the one way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation overflow occurs when the number of elements in any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the total capacity. Another alternative is to use a linked list for each priority level.

APPLICATIONS OF QUEUES :

1. Round Robin technique for processor scheduling is implemented using queues.
2. All types of customer service (like railway ticket reservation) center software's are designed using queues to store customers information.

Printer server routines are designed using queues. A number of users share a printer using printer server (a dedicated computer to which a printer is connected), the printer server then spools all the jobs from all the users, to the server's hard disk in a queue. From here jobs are printed one-by-one according to their number in the queue.

Insertion in priority Queue:

INSERT_PRIORITY(HEAD, DATA, PRIORITY):

Step 1: Create new node with DATA and PRIORITY
Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.
Step 3: NEW -> NEXT = HEAD
Step 4: HEAD = NEW
Step 5: Set TEMP to head of the list
Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY
Step 7: TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: NEW -> NEXT = TEMP -> NEXT
Step 9: TEMP -> NEXT = NEW
Step 10: End

DELETE_PRIORITY(HEAD):

Step 1: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.
Step 2: Free the node at the head of the list
Step 3: End