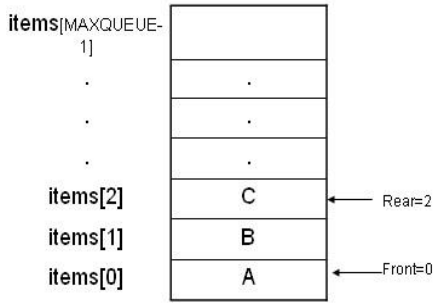
Unit 3:- Queue

- Operations in queue, Enqueue and Dequeue
- Linear and circular queue
- Priority queue

Queue

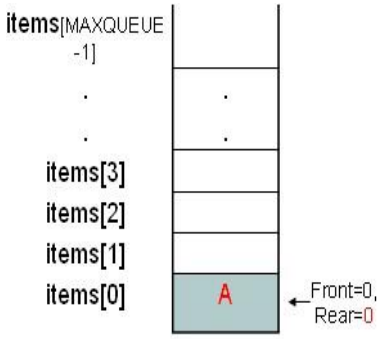
Queue is an ordered collection of item from which items may be deleted at one end (the **front** of the queue) and into which items may be inserted at the other end (the **rear** of the queue). It is also referred as **FIFO (first in first out). Example:**



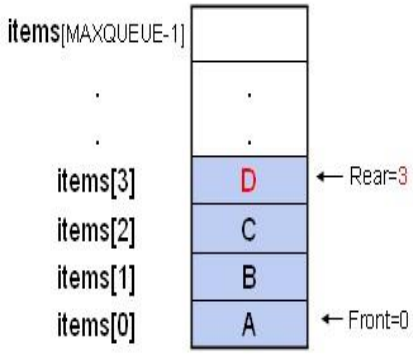
Operations on queue:

- ***MakeEmpty(q):*** To make q as an empty queue
- ***Enqueue(q, x):*** To insert an item x at the rear of the queue, this is also called by names add, insert.
- ***Dequeue(q):*** To delete an item from the front of the queue q. this is also known as Delete, Remove.
- ***IsFull(q):*** To check whether the queue q is full.
- ***IsEmpty(q):*** To check whether the queue q is empty
- ***Traverse (q):*** To read entire queue that is display the content of the queue.

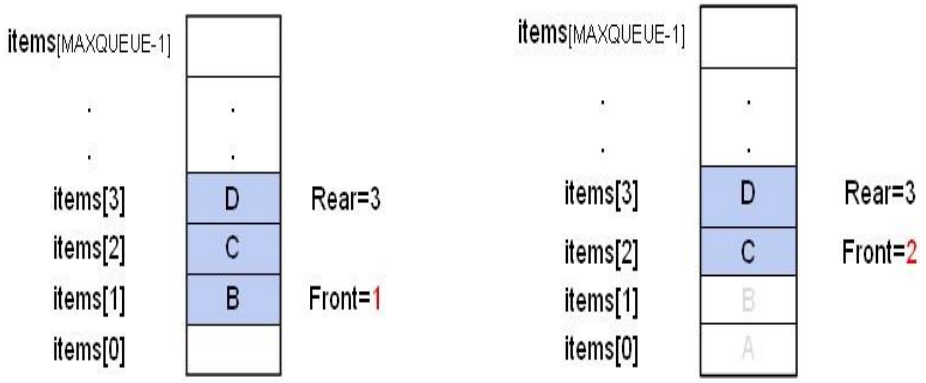
Enqueue(A):



Enqueue(B,C,D):

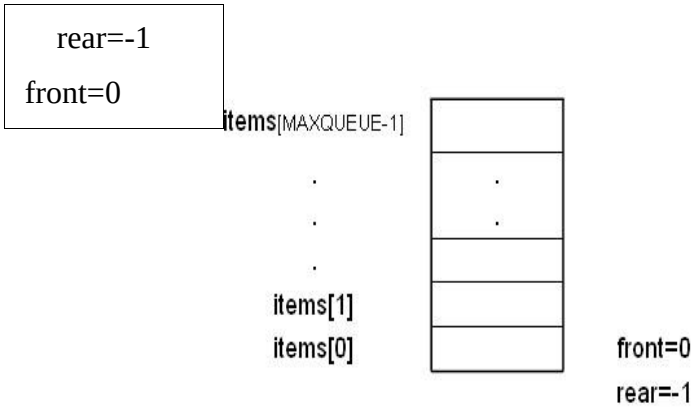


Dequeue(A): **Dequeue(B):**



Initialization of queue:

- The queue is initialized by having the *rear* set to *-1*, and *front* set to *0*. Let us assume that maximum number of the element we have in a queue is MAXQUEUE elements as shown below.



Applications of queue:

- Task waiting for the printing
- Time sharing system for use of CPU • For access to disk storage
- Task scheduling in operating system

The Queue as a ADT:

A queue q of type T is a finite sequence of elements with the operations

- ***MakeEmpty(q):*** To make q as an empty queue
- ***IsEmpty(q):*** To check whether the queue q is empty. Return true if q is empty, return false otherwise.
- ***IsFull(q):*** To check whether the queue q is full. Return true in q is full, return false otherwise.
- ***Enqueue(q, x):*** To insert an item x at the rear of the queue, if and only if q is not full.
- ***Dequeue(q):*** To delete an item from the front of the queue q. if and only if q is not empty.
- ***Traverse (q):*** To read entire queue that is display the content of the queue.

Implementation of queue:

There are two techniques for implementing the queue:

- Array implementation of queue(static memory allocation)
- Linked list implementation of queue(dynamic memory allocation)

Array implementation of queue:

In array implementation of queue, an array is used to store the data elements. Array implementation is also further classified into two types

- ✓ ***Linear array implementation:***

  A linear array with two indices always increasing that is rear and front. Linear array implementation is also called linear queue

- ✓ ***Circular array implementation:***

  This is also called circular queue.

Linear queue:

Algorithm for insertion (or Enqueue) and deletion (Dequeue) in queue:

Algorithm for insertion an item in queue:

1. ***Initialize front=0 and rear=-1***

if rear>=MAXSIZE-1

print "queue overflow" and return

else

set rear=rear+1

queue[rear]=item

2. end

Algorithm to delete an element from the queue:

1. ***if rear<front***

print "queue is empty" and return

else

item=queue[front++]

2. ***end***

Declaration of a Queue:

```
# define MAXQUEUE 50 /* size of the queue items*/
struct queue
{
        int front;
        int rear;
       int items[MAXQUEUE];
};

typedef struct queue qt;
```

Defining the operations of linear queue:

- ***The MakeEmpty function:***

```
void makeEmpty(qt *q)
{
   q->rear=-1;
   q->front=0;
}
```

- ***The IsEmpty function:***

```
int IsEmpty(qt *q)
{
 if(q->rear<q->front)
            return 1;
else
           return 0;
}
```

- ***The Isfull function:***

```
int IsFull(qt *q)
{
           if(q->rear==MAXQUEUEZIZE-1)
                      return1;
else


                      return0;
}
```

- ***The Enqueue function:***

```
void Enqueue(qt *q, int newitem)
{
if(IsFull(q))
        {   printf("queue   is   full");
                exit(1);
        }
        else
        {       q->rear++;
                q->items[q->rear]=newitem; }
}
```

- ***The Dequeue function:***

```
int Dequeue(qt *q)
{
if(IsEmpty(q))
        {
        printf("queue is Empty");
                exit(1);
        }
        else
        {
        return(q->items[q->front]);
                q->front++;
        }
}
```

Problems with Linear queue implementation:

- Both rear and front indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again. Wastage of the space is the main problem with linear queue which is illustrated by the following example.

		11	22	33	44	55
0	1	2	3	4	5	6
		f				r

front=2, rear=6

This queue is considered full, even though the space at beginning is vacant.

/*Array implementation of linear queue*/

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
  struct queue
  {
       int     item[SIZE];
       int rear;
       int front;
   };
   typedef struct queue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
    void main()
   {
       int ch;
       qu *q;
       q->rear=-1;
       q->front=0;
printf("Menu for program:\n");
printf("1:insert\n2:delete\n3:display\n4:exit\n");
do{
            printf("Enter your choice\n");
            scanf("%d",&ch);
             switch(ch)
                 {
                        case1:
                                insert(q);
                                break;
                        case2:
```

```
                    delet(q);
                    break;
                case3:
                    display(q);
                    break;
                case4:
                    exit(1);
                    break;
                default:
                    printf("Your choice is wrong\n");
            } }while(ch<5);
            getch();
    }
  /**********insert function*************/
    void insert(qu *q)
{       int d;
        printf("Enter data to be inserted\n");
        scanf("%d",&d);
        if(q->rear==SIZE-1)
             printf("Queue is full\n");
         else
         {      q->rear++;
                q->item[q->rear]=d;
         }
  }
  /***********delete      function*****************/
  void delet(qu *q)
  {
        int d;
        if(q->rear<q->front)
             printf("Queue is empty\n");
         else
        {
```

```
            d=q->item[q->front];
            q->front++;
            printf("Deleted item is:");
            printf("%d\n",d);
        }
 }
/***************display function************/
void display(qu *q)
{
        int i;
        if(q->rear<q->front)
                printf("Queue is empty\n");
        else
         {
        for(i=q->front;i<=q->rear;i++)
                {
                        printf("%d\t",q->item[i]);
                }
        }
 }
```

Circular queue:

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

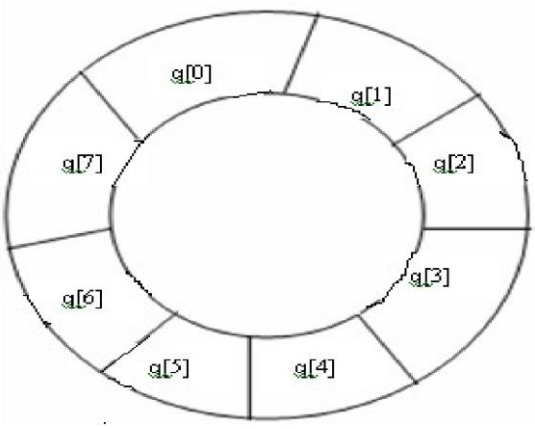


Fig:- Circular queue

- A circular queue overcomes the problem of unutilized space in linear queue implementation as array.
- In circular queue we sacrifice one element of the array thus to insert n elements in a circular queue we need an array of size n+1.(or we can insert one less than the size of the array in circular queue).

Initialization of Circular queue:

rear=front=MAXSIZE-1

Algorithms for inserting an element in a circular queue:

This algorithm is assume that rear and front are initially set to MAZSIZE-1.

1. if (front==(rear+1)%MAXSIZE)
   print Queue is full and exit
   else rear=(rear+1)%MAXSIZE;
2. cqueue[rear]=item;
3. end

Algorithms for deleting an element from a circular queue:

This algorithm is assume that rear and front are initially set to MAZSIZE-1.

1. if (rear==front) [checking empty condition]
   print Queue is empty and exit
2. front=(front+1)%MAXSIZE;
3. item=cqueue[front];
4. return item;
5. end.

Declaration of a Circular Queue:

```
# define MAXSIZE 50 /* size of the circular queue items*/
struct cqueue
{
        int front;
        int rear;
        int items[MAXSIZE];
};
typedef struct cqueue cq;
```

Operations of a circular queue:

- ***The MakeEmpty function:***

```
void makeEmpty(cq *q)
    {
        q->rear=MAXSIZE-1;
        q->front=MAXSIZE-1;
    }
```

- ***The IsEmpty function:***

```
int IsEmpty(cq *q)
    {
        if(q->rear<q->front)
            return 1;
        else
            return 0;
    }
```

- ***The Isfull function:***

```
int IsFull(cq *q)
    {
        if(q->front==(q->rear+1)%MAXDIZE)
            return 1;
        else
            return 0;
    }
```

- ***The Enqueue function:***

```
void Enqueue(cq *q, int newitem)
    {
        if(IsFull(q))
        { printf("queue is full");
          exit(1);
         }
        else
        {
            q->rear=(q->rear+1)%MAXDIZE;
            q->items[q->rear]=newitem;
        }
    }
```

- ***The Dequeue function:***

int Dequeue(cq *q)

{ if(IsEmpty(q))

{ printf("queue is Empty");

exit(1);

}

else

{

q->front=(q->front+1)%MAXSIZE;

return(q->items[q->front]);

}

}

/*implementation of circular queue with scarifying one cell */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct   cqueue
{
   int     item[SIZE];
   int rear;
   int front;
};
typedef struct cqueue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
void main()
{
  int ch;
 qu *q;
q->rear=SIZE-1;
q->front=SIZE-1;
 printf("Menu for program:\n");
 printf("1:insert\n2:delete\n3:display\n4:exit\n");
do   {
```

```
            printf("Enter your choice\n");
            scanf("%d",&ch);
            switch(ch)
            {   case 1:
                        insert(q);
                        break;
                case 2:
                        delet(q);
                        break;
                case 3:
                        display(q);
                        break;
                case 4:
                        exit(1);
                        break;
                default:
                        printf("Your choice is wrong\n");
                         break;
            }
          }while(ch<5);
          getch();
        }
/**********insert function*************/
void insert(qu *q)
        {
           int d;
           if((q->rear+1)%SIZE==q->front)
                    printf("Queue is full\n");
           else
           {
                q->rear=(q->rear+1)%SIZE;
                printf ("Enter data to be inserted\n");
                scanf("%d",&d);
                q->item[q->rear]=d;
           }
```

```
    }
    /**********delete function*****************/
    void delet(qu *q)
    {                   if(q->rear==q->front)
                printf("Queue is empty\n");
                else {
                    q->front=(q->front+1)%SIZE;
                    printf("Deleted item is:");
                    printf("%d\n",q->item[q->front]);
                }
    }
/***************display        function***********/
    void display(qu *q)
    {
      int i;
      if(q->rear==q->front)
    printf("Queue is empty\n");
      else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%SIZE;i!=q->rear;i=(i+1)%SIZE)
        {
              printf("%d\t",q->item[i]);
        }
        printf("%d\t",q->item[q->rear]);
      }
    }
```

/*implementation of circular queue without secrifying one cell by using a count variable */

```
    #include<stdio.h>
    #include<conio.h>
    #define   SIZE   20
    struct cqueue
    {
       int item[SIZE];
       int rear;
```

```
    int front;
};
int count=0;
typedef struct cqueue qu;
void insert(qu*);
void delet(qu*);
void display(qu*);
void main()
{
  int ch;
  qu *q;
q->rear=SIZE-1;
q->front=SIZE-1;
printf("Menu for program:\n");
printf("1:insert\n2:delete\n3:display\n4:exit\n");
  do{
    printf("Enter your choice\n");
     scanf("%d",&ch);
    switch(ch)
     {   case 1:
                    insert(q);
                    break;
          case 2:
                    delet(q);
                    break;
          case 3:
                    display(q);
                    break;
          case 4:
                    exit(1);
                    break;
          default:
                    printf("Your choice is wrong\n");
                    break;
     }
```

```
    }while(ch<5);
    getch();
  }
/**********insert       function*************/
  void insert(qu *q)
  {
     int d;
    if(count==SIZE)
  printf("Queue is full\n");
     else
     {
          q->rear=(q->rear+1)%SIZE;
          printf ("Enter data to be inserted\n");
          scanf("%d",&d);
          q->item[q->rear]=d;
          count++;
     }

  }
/**********delete function*****************/
void delet(qu *q)
  {
  if(count==0)
          printf("Queue is empty\n");
  else
          {
             q->front=(q->front+1)%SIZE;
          printf("Deleted item is:");
          printf("%d\n",q->item[q->front]);
           count--;
          }
  }
/***************display       function***********/
  void display(qu *q)
  {
    int i;
```

```
  if(q->rear==q->front)
     printf("Queue is empty\n");
  else
{
     printf("Items of queue are:\n");
     for(i=(q->front+1)%SIZE; i!=q->rear; i=(i +1)%SIZE)
     {
          printf("%d\t",q->item[i]);
     }
     printf("%d\t",q->item[q->rear]);
  }
}
```

Priority queue:

A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules:.

- An element of higher priority is processed before any element of lower priority.
- If two elements has same priority then they are processed according to the order in which they were added to the queue.

The best application of priority queue is observed in CPU scheduling.

- ✔ The jobs which have higher priority are processed first.
- ✔ If the priority of two jobs is same this jobs are processed according to their position in queue.
- ✔ A short job is given higher priority over the longer one.

Types of priority queues:

- ***Ascending priority queue(min priority queue):***

An ***ascending priority queue*** is a collection of items into which items can be inserted arbitrarily but from which only the smallest item can be removed.

- ***Descending priority queue(max priority queue):***

An **descending priority queue** is a collection of items into which items can be inserted arbitrarily but from which only the largest item can be removed.

Priority QUEUE Operations:

- **Insertion :**

  The insertion in Priority queues is **the same as** in non-priority queues.

- **Deletion :**

  Deletion requires a search for the element of highest priority and deletes the element with highest priority. The following methods can be used for deletion/removal from a given Priority Queue:

  - ✔ An empty indicator replaces deleted elements.
  - ✔ After each deletion elements can be moved up in the array decrementing the rear.
  - ✔ The array in the queue can be maintained as an ordered circular array

Priority Queue Declaration:

Queue data type of Priority Queue is the same as the Non-priority Queue.

```
#define MAXQUEUE 10 /* size of the queue items*/
 struct pqueue
{
        int front;
        int rear;
        int items[MAXQUEUE];
};
struct pqueue *pq;
```

The priority queue ADT:

A ascending priority queue of elements of type T is a finite sequence of elements of T together with the operations:

- **MakeEmpty(p):** Create an empty priority queue p
- **Empty(p):** Determine if the priority queue p is empty or not
- **Insert(p,x):** Add element x on the priority queue p
- **DeleteMin(p):** If the priority queue p is not empty, remove the minimum element of the quque and return it.
- **FindMin(p):** Retrieve the minimum element of the priority queue p.

Array implementation of priority queue:

- **Unordered array implementation:**
  - ✔ To insert an item, insert it at the rear end of the queue.
  - ✔ To delete an item, find the position of the minimum element and
    - ✗ Either mark it as deleted (lazy deletion) or
    - ✗ shift all elements past the deleted element by on position and then decrement rear.

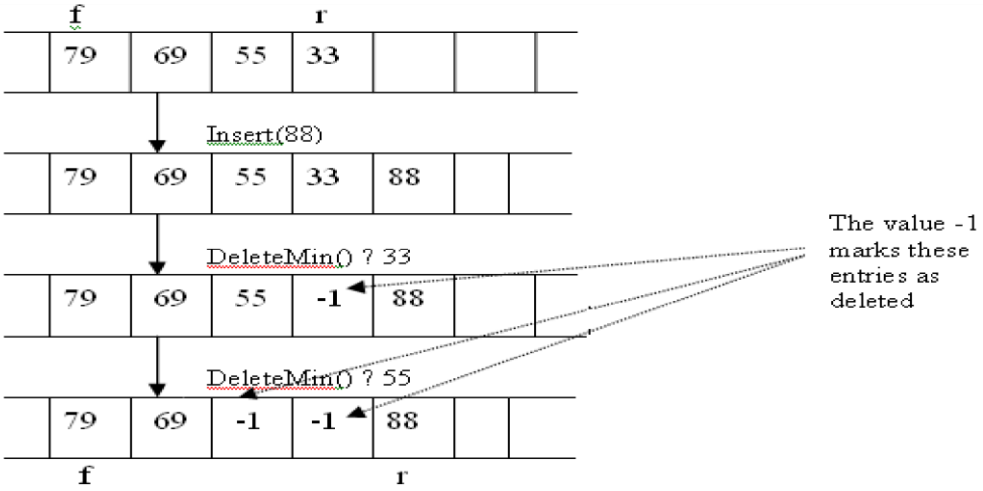


Fig Illustration of unordered array implementation

- **Ordered array implementation:**
  - ✔ Set the front as the position of the smallest element and the rear as the position of the largest element.
  - ✔ To insert an element, locate the proper position of the new element and shift preceding or succeeding elements by one position.
  - ✔ To delete the minimum element, increment the front position.

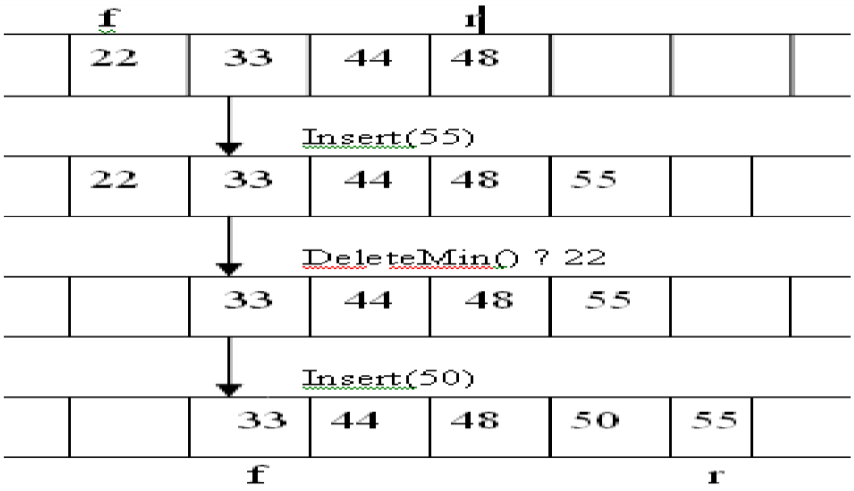


Fig Illustration of ordered array implementation

Application of Priority queue:

In a time-sharing computer system, a large number of tasks may be waiting for the CPU, some of these tasks have higher priority than others. The set of tasks waiting for the CPU forms a priority queue.

/*implementation of ascending priority queue */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct   cqueue
{
   int item[SIZE];
   int rear;
   int front;
};
typedef struct queue pq;
void insert(pq*);
void delet(pq*);
void display(pq*);
void main()
{
  int ch;
  pq *q;
 q->rear=-1;
 q->front=0;
  printf("Menu for program:\n");
  printf("1:insert\n2:delete\n3:display\n4:exit\n");
do   {
      printf("Enter youer choice\n");
      scanf("%d",&ch);
       switch(ch)
      {   case 1:
                    insert(q);
                    break;
           case 2:
                    delet(q);
```

```
                    break;
            case 3:
                    display(q);
                    break;
            case 4:
                    exit(1);
                    break;   default:
                    printf("Your choice is wrong\n");
                     break;
              }
          }while(ch<5);
          getch();
        }
/**********insert function*************/
        void insert(pq  *q)
        {
           int d;
           if(q->rear==SIZE-1)
                printf("Queue is full\n");
           else
           {
                printf ("Enter data to be inserted\n");
                scanf("%d",&d);
                q->rear++;
                q->item[q->rear]=d;
           }

        }
        /**********delete function*****************/
        void delet(pq  *q)
        {
                int i, temp=0, x;
                x=q->item[q->front];
                if(q->rear<q->front)
                {
                        printf("Queue is empty\n");
```

```
                return   0;
        }
        else
        {
        for(i=q->front+1; i<q->rear; i++)
                { if(x>q->item[i])
                        {
                          temp=i;
                          x=q->item[i];
             }
                }
                for(i=temp;i< q->rear-1;i++)
                {
                     q->item[i]=q->item[i+1];
                }
                q->rear--;
                return x;
        }
    }
  /*************display function************/
        void display(pq  *q)
        {
                int i;
                if(q->rear < q->front)
                     printf("Queue is empty\n");
                else
                {
                        printf("Items of queue are:\n");
                        for(i=(q->front i<=q->rear;i++)
                        {
                                printf("%d\t",q->item[i]);
        }
}
```

Compiled by: Er. Sandesh S Poudel