3. QUEUE

QUEUE

A queue is a linear list of elements in which deletion can take place only at one end called front of stack and insert can take place only at the other end called the rear.

The term 'front' and 'rear' are used in describing a linear list only when it is explained as queue. In queue, the element which is inserted first is the element that is deleted first. So it works in FIFO manner.

Examples of queue in real life are :-

(i) People waiting for a service in a bank
(ii) Car passing through a toll-booth.
(iii) In computer, multiple processing or job waits in a queue for getting services of CPU, printer or any other devices.

Linear queue

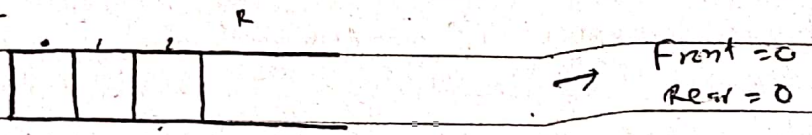


fig (a) Initialize an empty stack

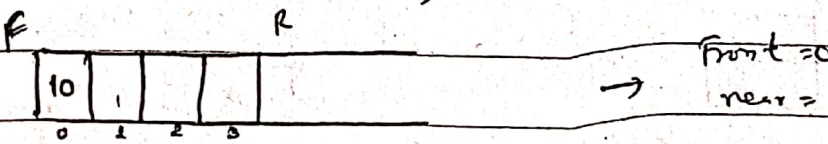


fig (b) After inserting 10

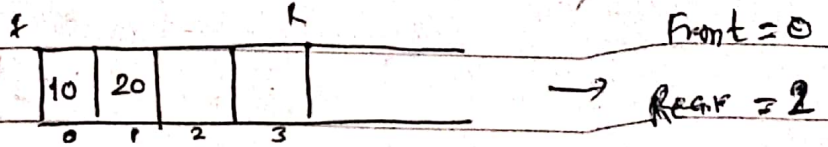


fig (c) After inserting 20

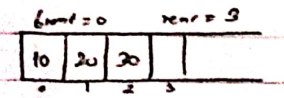


fig (d) After inserting 30

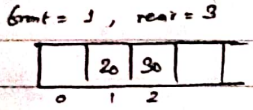
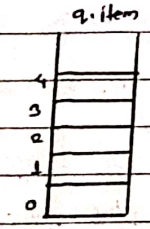


Fig (e) After deleting 10

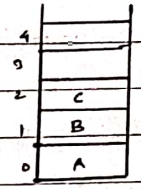
Initially q.rear = -1 and q.front is set to 0. The queue is empty whenever q.rear < q.front. The no of elements in the queue at any time is equal to the value of q.rear - q.front + 1

Figure below illustrates an array of 5 elements used to represent a queue ie. MAXQUEUE = 5



q.front = 0
q.rear = -1

fig (a)

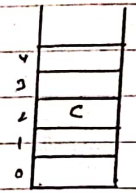


q.rear = 2.
q.front = 0

fig (b)

Items A, B, C are inserted as q.rear increased by 1. ie q.rear = 2 (ie. rear = inserted)
where as q.front = 0, no of items has been deleted.

Whenever an item is inserted in the queue, the value of rear increased by 1. ie rear = rear + 1

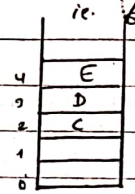


q.front = q.rear = 2

fig (c) Here, 2 items A & B has been deleted, So q.front is increased to 2 ie q.front = 2 while q.rear is same as q.rear = 2

Whenever an item is removed from the queue, the value of front is increased by 1.

ie. front = front + 1



q.rear = 4

q.front = 2

Fig (d) Here 2 item D & E are inserted, So q.rear = 4. The value of q.front = 2. So there are 4-2+1 elements in the queue ie q.rear - q.front + 1

Operation on queue or Operation as ADT

The basic operation that can be performed on queue are :-

(i) To insert an element on queue.

Insert (q, x)

means insert x item at the rear of the queue

(ii) To delete an element from the queue.

x = remove (q)

delete the front element from the queue which is x.

(iii) Empty (q) . This function returns true or false depending upon whether the queue contain any elements or not.

Algorithm for insertion and deletion in queue

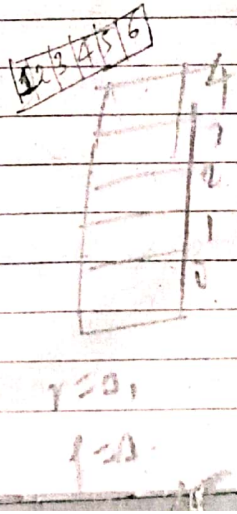
Algorithm for insertion in queue

Let queue is the array of some specified size, say maxsize then insertion algorithm will be :

```
1. If (rear > maxsize)
       display queue is full
       go to step 4 or exit

2. else
       Queue [rear] = item;

3. Rear = rear + 1
4. Exit
```



Algorithm for deletion in queue

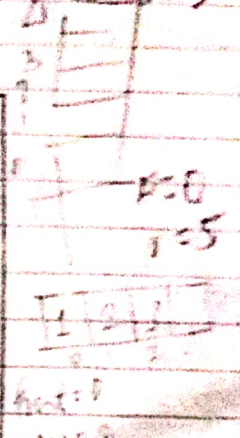
Let queue is the array of some specified size, say maxsize, then deletion algorithm is :-

```
1. If (front > rear)
       display queue is empty
       goto step 5 or exit

2. else item = queue [front]

3. if (front == rear)
       set front = 0 , rear = -1
4. else
       front = front + 1
5. exit
```

item=queue[front]
front=front+1



Array implementation of Queue

```
# include <stdlib.h>
# include <conio.h>
# include <stdio.h>
# define size 5

Void main ()
```

```
{
int cqueue [size];
int data, i, choice;
int rear = 0;      // rear of queue is initially zero
int front = 0;     // front of queue is initially zero
while (1)
  {
  printf ("\n 1. Insert Element");
  printf ("\n 2. Delete Element");
  printf ("\n 3. View Element");
  printf ("\n 4. Exit");
  printf ("Enter your choice");
  scanf ("%d", &choice);
  switch (choice)
    {
    case 1:
      if (rear < size)
      {
      printf ("Enter any data");
      scanf ("%d", &data);
      queue [rear] = data;
      rear ++;
      }
      else
      {
      printf ("Queue is full");
      }
      break;
```



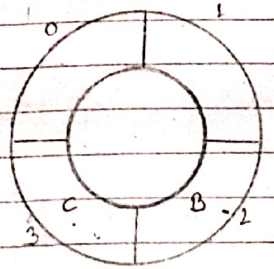
```
    case 2:
      if (front < rear)
      {
      printf ("The deleted element is %d", queue [front]);
      front ++;
      }
      else
      {
      printf ("Queue is empty");
      }
    break;

    case 3:
      if (rear > front)
      {
      printf ("\n Elements are");
      for (i = front; i< rear ; i++)
        {
        printf ("\n %d ", queue [i]);
        }
      }}
      else
      {
      printf ("Queue is empty");
      }
    break;
```

```
    case 4:
        exit(0);
    default:
        printf(" Enter correct choice 1-4 ");
    }
    }
}
```

CIRCULAR QUEUE

A circular queue is one type of queue in which insertion of new element is done at the very first location of the queue. If the last location of the queue is full (ie if we have queue of n element), after inserting an element to the last location of the array, the next element will be inserted at the first location of the array. It is possible to insert new element if and only if the first location is empty.



Rear = (3). 0

front = 2

A circular queue overcomes the problem of unutilized space in linear queue because if last element of the linear queue is not deleted, at that condition we cannot insert the element if there are available space.

Consider a circular queue of size 4

(a)

0	1	2	3
	5	10	15

front = 1
rear = 3

(b)

0	1	2	3
		10	15

front = 2, rear = 3

deletion of queue

Assumption of circular queue

1. The front will point the 1st element.
2. If (F=R) then the queue is empty.
3. Each time a new element is ~~deleted~~ inserted the rear is increased by 1.
4. Each time an element is deleted from queue, front is increased by 1, but there will be an extra logic to break of front & rear.

ALGORITHM TO INSERT ELEMENT IN CIRCULAR QUEUE

1. If (front == (rear+1) % MAXSIZE)
   Print Queue is overflow and exit.
2. else rear = (rear+1) % MAXSIZE
   queue [rear] = value
3. Exit

size = 5

i	1	2	3	4
	1	2	3	4

front = 1 rear = 4

DELETION ALGORITHM

1. If (front == rear)
   print queue is empty & exit
2. else
   item = queue [front]
   front = (front+1) % MAXSIZE
3. Exit

To display the status of the queue

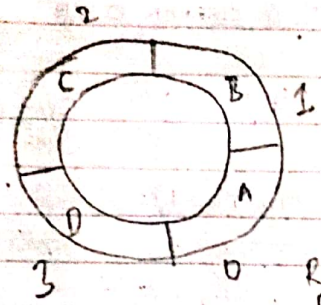
If (front == rear)
   display queue is empty
else
   for (i = front ; i < rear ; i++
   display queue [i]

ALGORITHM TO INSERT AN ELEMENT IN CIRCULAR QUEUE

1. If the queue is full:
   a) Display "Queue Overflow"
   b) Exit
2. If the queue is empty (IF FRONT = -1):
   (a) Set FRONT = 0
   (b) Set REAR = 0
   (c) Goto step 4
3. If REAR is at the last index position:
   (a) Set REAR=0 // set REAR to the first index position
   (b) Go to step 4
4. Increment REAR by 1
5. Queue [rear] = item
6. Exit

ALGORITHM TO DELETE AN ELEMENT IN CIRCULAR QUEUE

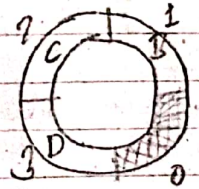
1. If the queue is empty: // front = -1
   (a) Display "Queue underflow"
   (b) Exit
2. If there is only one element in the queue: // front = rear
   (a) set FRONT = -1
   (b) set REAR = -1
   (c) Exit
3. If FRONT is at the last index position:
   (a) Set FRONT=0 // set FRONT to the first index position
   (b) Exit
4. Increment FRONT by 1
5. Item = queue [front]
6. exit



R=3
F=0
MAX=4

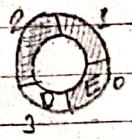
0 == 3+1 % 4

0 == 0



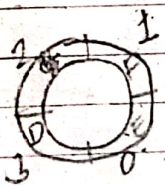
MAX=4
R=3
F=1

I=3+1%



R=1
F=2
MAX=4

R



ARRAY IMPLEMENTATION OF CIRCULAR QUEUE

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# define size 5
void main()
  {
   int queue [size];
   Int data, choice, i, j;
   int rear = 0;
   int front = 0;
   int count = 0;
   while (1)
      {
      printf ("\n 1. Insert element");
      printf ("\n 2. Delete element");
      printf ("\n 3. View element");
      printf ("\n 4. Exit");
      printf ("Enter your choice");
      scanf ("%d", & choice);

      Switch (choice)
        {
        case 1:
           if (count < size)
           {
           printf ("Enter any data");
           scanf ("%d", & data);
```

```
cqueue [rear] = data;
rear = (rear +1) % size;
count ++;
{
else
{ printf ("Queue is full");
}
break;

case 2:
if (count>0)
{
printf ("The deleted element is %d", cqueue [front]);
front = (front +1) % size;
count --;
}
else
{ printf ("Queue is empty");
}
break;

case 3:
if (count >0)
{
i = front;
j = 0;
while (j< count)
{
printf ("\n %d", cqueue [i])
```



```
i = (i +1) % size ;
j ++;
}
}
else
{
printf ("Queue is empty");
}
break;

case 4:
exit (0);

default:
printf ("\n Enter the correct choice 1-4");

}
}
}
```

ENQUEUE & DEQUEUE

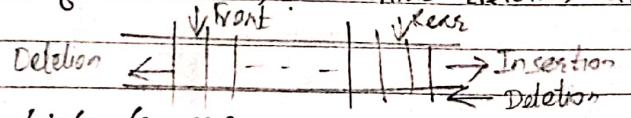
Enqueue is the process of inserting an element at the end of the queue that is at the rear of the queue.

Dequeue is the process in which elements can be added and removed at either end but not in the middle. There are two types of dequeue:-

(a) Input restrict dequeue
(b) Output restrict dequeue

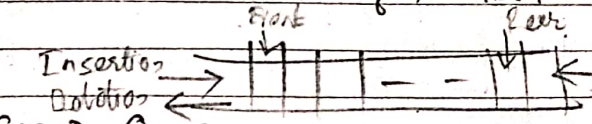
(a) Input restrict dequeue

It is a dequeue in which it allow insertion at only one end of the list but allow deletion at both end of the list.



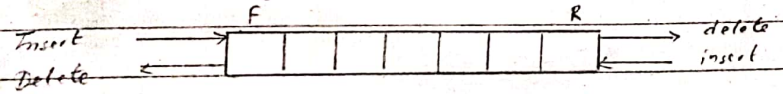
(b) Output restrict dequeue

It allow deletion at only one end of the list, but allow insertion at both end of the list.



DOUBLE ENDED QUEUE

It is the list of the element in which insertion and deletion operations are performed from both end.



Since insertion and deletion are performed from either

end, it is necessary to design algorithm to perform the following four operations:-

1. Insertion of an element at the rear of queue (Dinject)

```
(a) If (rear == maxsize)
        print queue is full and exit
(b) else
        q[rear] = item
        rear    = rear + 1
(c) exit
```

2. Deletion of element from the front end of the queue (Dpop)

```
(a) If (f == r)
        print Queue is empty and exit
(b) else
        item = q[front]
        front = front + 1
(c) exit
```

3. Insertion of an element at front end of queue [Dpush]

```
(a) if (front == 0)
        print Queue is full and exit
(b) else
        front = front - 1
        q[front] = item
(c) exit
```

4. Deletion of an element from rear end of queue (Deject)

```
(a) If (r == f)
        Queue is empty and exit
(b) else
        item = q [rear] ;
        rear = rear - 1
(c) exit
```

PRIORITY QUEUE

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

(i) An element of higher priority is processed before any element of lower priority.

(ii) Two elements with the same priority are processed according to the order in which they are added to the queue.

It can be divided into two types:

(a) Ascending Priority Queue

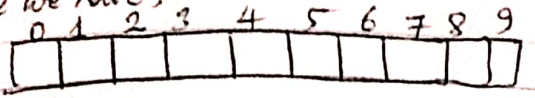
(b) Descending

In ascending priority queue, items can be inserted and removed from the queue in ascending order. Where as in descending priority queue, items can be inserted and removed in descending order from the queue.

DIFFERENCES BETN STACK & QUEQUE

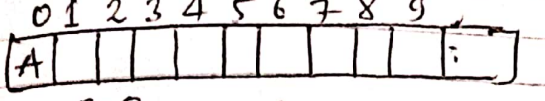
STACK	QUEUE
1. Stack is a linear data structure in which the data items are inserted and deleted ~~from/at~~ at/from one end of the list.	1. Queue is a data structure in which data is inserted at one end and deleted from the next end of the list.
2. The end where insertion and deletion is performed is called Top of Stack (TOS).	2. Insertion is done at rear end and deletion is done at the front of the list.
3. It follows LIFO manner. The last inserted element is deleted at first.	3. It follows FIFO manner. The first inserted element is deleted at first.
4. Example of stack in real life are as piling of books, etc.	4. Examples of queue in real life are as customers waiting in a line to get service in a bank, etc.
5. A stack is a recursive data structure.	5. Queue is not a recursive data structure.

Suppose we have:



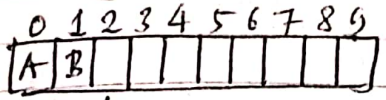
F=0
R=-1

After Inserting A



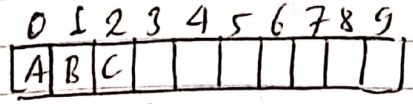
F=0; R=0

After inserting B



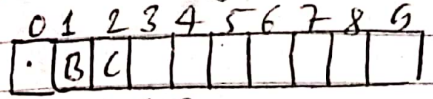
F=0, R=1

After Inserting C



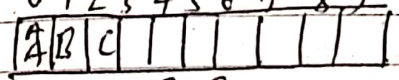
After F=0, R=2

After deleting node A,

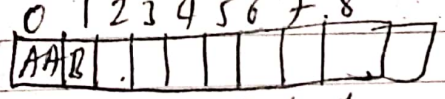


F=1; R=2

Inserting at Front End AA , F=F-1=0



F=0, R=2



F=0 R=2-1=1

4. STATIC AND DYNAMIC LIST

LIST AND LINKED LIST

List refers to linear collection of data items.

Eg. list of item, frequently used list used by person per day. etc

apple
orange
butter
...

These items can be stored in computer through array. The size of the array is defined initially. So size cannot be changed whatever declared. ie. size will be constant. So this way of storing is known as static memory allocation. We cannot add more than what it has been defined previously. In static memory allocation, memory will be wasted if there are some useless space.

The another way of storing is by using linked list which is called dynamic memory allocation. Here according to the need, memory can be allocated. If memory is not needed, then it can be freed. List can be implemented from two methods:-

(a) Array implementation (ie contiguous list)
(b) Pointer implementation (ie. linked list)