Pradeep Pant

Set-2 Unit-2

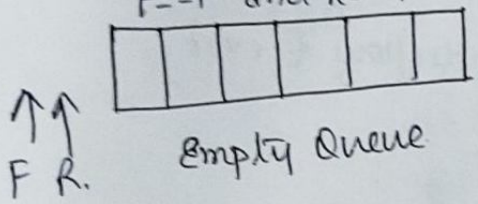
(1)

Queue → Queue is logically a first in first out types of list.

→ Queue is a non premitive linear data structure.

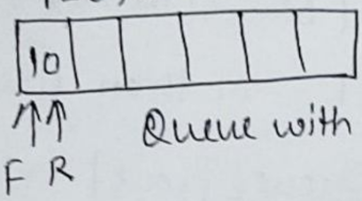
→ Queue is a homogeneous collection of elements in which new elements are added at one end called rear and existing elements are deleted from other end called the front.

F=-1 and R=-1



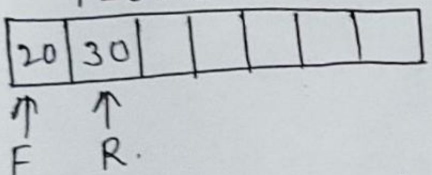
Empty Queue

F=0, R=0



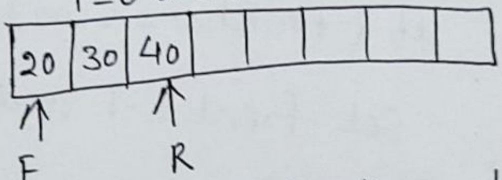
Queue with one element

F=0 and R=1



Queue with two element

F=0 and R=2



Queue with three elements

Note - if F==R then only one element is a Queue.

No. of elements in a Queue = R - F + 1

Implementation of Queue →

1) Using array
2) using linked List

1) Using array → QInsert(Queue[maxsize], ITEM, front, rear) Queue[maxsize] is the array, item is inserted or deleted in a Queue. front and Rear is used for ~~insertion~~ deletion and insertion respectively.

②

1. if rear== maxsize-1 then write overflow and exit.
2. if rear==-1 then set rear=0 and front=0
   else set rear=rear+1
3. Queue[rear]= item
4. Exit

2) <u>Deletion</u> -

QDelete(Queue[maxsize], item, front, rear)

1. if front==-1 then write underflow & exit
2. item= Queue[front]
3. if (front == rear) then
   Set front =-1 and rear=-1
   else
   set front= front+1
4. Exit

3) <u>Qtraverse</u> - Qtraverse (Queue[maxsize], front, rear)

1. if rear=-1 then write Queue is empty & Exit.
2. ~~for~~ i= front
3. repeat step 4 & 5 untill i<= rear
4. Display Queue[i]
5. i=i+1
   [END of Loop]
6. Exit

C function → Insertion

```
#define maxsize 30
int queue[maxsize], front=-1, rear=-1;

void insert( )
{
    int item;
    ~~if(front==(rear+1)%maxsize)~~
    ~~if(rear==maxsize-1)~~ {
if(rear==maxsize-1)
{   printf("Overflow");
                exit(0);
    }
    else
      {
        if(rear==-1)
        {
           front=0;
           rear=0;
        }
       else
          rear=(rear+1)~~%maxsize~~;

    printf("Enter item");
    scanf("%d", &item);

     queue[rear]=item;
    }

}
```

```
void delition ( )
{
    if ( front = -1)
        printf(" Underflow");
    else
    {
        if ( front == rear)
        {
            front = -1;
            rear = -1;
        }
        else
            front = (front + 1);
    }
}

void display ( )
{
    int i;
    if ( rear == -1)
        printf(" queue is empty");
    else
    {
        for (i = front; i <= rear; i++ i = (i + 1)
            printf(" %d", queue[i]);
    }
}

void main( )
{
    insert( );        deletion( );
    insert( );        display( );
}
```

2) Linked List Implementation

Algorithm →

Insert (Queue, front, rear, info, next, item, new)

1. new = AVAIL
2. if (new == NULL) write overflow & Exit.
3. Info[new] = item
4. LINK[new] = NULL
5. if (front == NULL) then
   front = new and rear = new
6. else
   ~~rear → next =~~
   LINK[rear] = new
   Rear = new.
7. Exit

Deletion (Queue, front, rear, PTR, next, INFO)

1. if (front == NULL)
   write ~~error~~ underflow and exit
2. PTR = front
3. Display ~~P~~INFO[PTR] is deleted
4. if (~~front→~~ LINK[front] = NULL) then
   front = NULL and rear = NULL & exit.
5. front = LINK[fron]
6. free (PTR)
7. Exit.

⑥ Traverse()

1. if (front == NULL) then write Queue is empty & exit.

2. ptr= front

3. repeat step 4 & 5 till ptr ≠ NULL.

4. Display ptr->info

5. ptr= ptr->next

6. exit.

C function→ ~~int()~~

~~void insert()~~

```
struct node
{
  int info;
  struct node *next;
} *front= NULL, *rear= NULL;

void insert()
{
  struct node *new;
  new= (struct node *) malloc(sizeof (struct node));
   if(new == null)
      printf("overflow");
  else
  {
      printf("Enter item");
      scanf("%d", &new->info);
      new->next= NULL;
```

(17)

7

```
if ( front = = NULL)
{
    front = new;
    rear = new;
}
else
{
    rear -> next = new
    rear = rew
}
}

void deletion ( )
{.
    struct node * p;
if ( front = = NULL)
    printf( " Underflow");
else
{
    P = front;
    printf( " %d is deleted ", P -> info);
    if ( P -> next = = NULL) or if (front = = rear)
        front = rear = NULL
    else
        front = front -> next
    free ( P);
}
}
```

⑧ void traverse( )
{

```
struct node *P;
if( Rear = = NULL)
    printf(" Queue is empty");

else
for (P = front; P! = NULL; P = P -> next)
    printf(" %d", P -> info);
}
```

Limitation of ~~circular~~ simple Queue -

- Overflow condition
  if (rear = = max-size - 1)

10	20	30	40	60	
↑ F				↑ R.	

Deleted ~~the~~ Two items

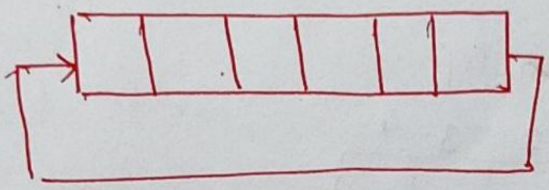
~~10~~	~~20~~	30	40	60
		↑ F		↑ R

In above R = maxsize - 1 . So no item can be inserted even there is a space in Queue for two items.

Circular Queue→

problem of simple queue can be solved by circular queue.

In a circular queue the insertion of new element is done at first location of Queue if the last location of the Queue is full.



Overflow condition (Array implementation)

if (front = (real+1) % maxsize)

printf("Overflow");

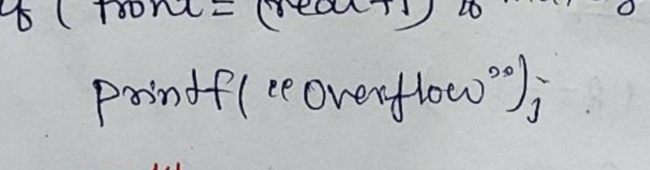
Underflow condition

front = -1 or rear = -1

operations on circular Queue→ (Array implementation)

1) Insert→

Concept→ Case 1) ~~Queue is empty~~

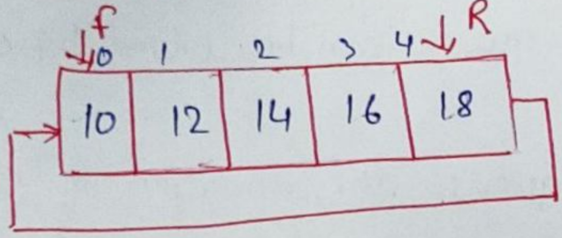


~~if (R= f+)~~

~~R= R+1~~

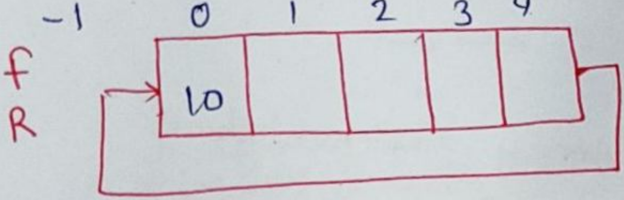
⑩ Insert → case 1:- Queue is full

N=5 (max-size)



if ((R+1) % maxsize == ~~front~~)
printf("Overflow")

Case 2:- Queue is empty



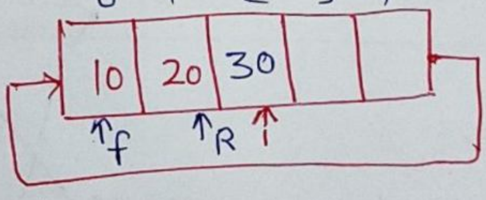
~~if (R==-1)~~ item=10

if (R == -1)
f= R=0
cq[R] = item

Case 3:- Queue is neither empty nor full

R=(R+1)% maxsize item=30

cq[R]= item



Algorithm →

Qinsert (Queue[maxsize], item, front, rear)

1. if (rear+1) % maxsize == front then
write "Overflow"

2. if (Rear ==-1) then
Set front=0, rear=0
else
Set rear= (rear+1) % maxsize

3. Queue[rear] = item
4. Exit.

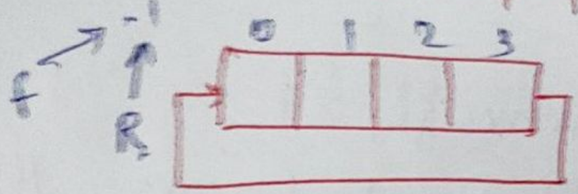
C function →

```
#define maxsize 30
int cq[maxsize], front=-1, rear=-1  //global declaration

void insert()
{
  int item;
  if ((rear+1)%maxsize == front)
    {
      printf("Overflow");
      exit(0);
    }
  if (rear==-1)
    front = rear = 0;
  else
    rear = (rear+1) % maxsize
  printf("Enter item");
  scanf("%d", &item);
  cq[rear] = item;
}
```

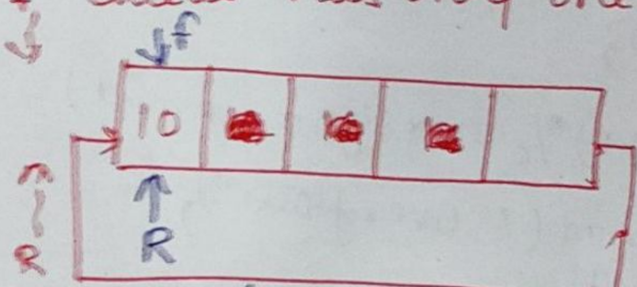
2) Deletion →

case 1:- Queue is empty



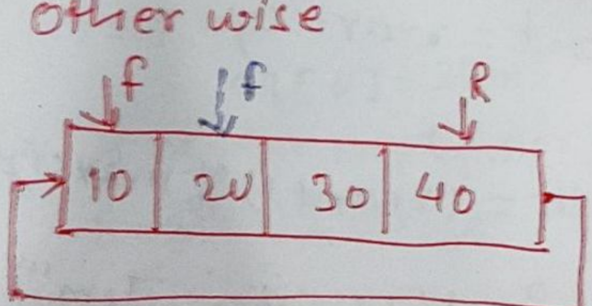

```
if (front == -1)
    printf("Underflow");
```

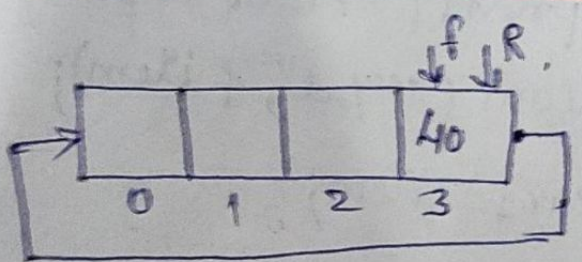
case 2: Queue has only one item




```
if(front == rear)
front = rear = -1
```

case 3: other wise





front = (front+1) % maxsize

Algorithm →

Circular_Queue_delete( )

1. if (front==-1) then
   write "Underflow" and exit.

2. item= Queue[front]

3. if (front==rear) then
   front=rear=-1
   else
   front=(front+1) % maxsize
   [End of if]

4. write item "is deleted"

5. Exit.

C function →

```
void deletion( )
{     int item;
      if (front==-1) then
      {
      printf(" underflow");
      exit(0);
      }
      item= cq[front];
      if (front==rear)
         front=rear=-1;
      else
         front= (front+1)%maxsize;
      printf("%d is deleted", item);
}
```

Traverse→

Concept→

Case 1:- Queue is empty

```
if (front == -1)
printf("Queue is empty");
```

Case 2:- Queue is not empty

```
for ( i = front; i != rear; i = (i+1) % maxsize)
    printf("%d", cq[i]);
printf("%d", cq[i]);
```

Algorithm→

Traverse( )

1. if (front == -1)
   write "Queue is empty & exit.
2. ~~for~~ i = front
3. repeat step 4 & 5 ~~while i!=rear~~ while (i != rear)
4. write Queue[i]
5. i = (i+1) mod maxsize
6. write Queue[i]
7. Exit

C function→ 15

```
void traverse( )
{

if (front==-1)
{
printf("Queue is empty");
exit(0);
}
for (i= front; i!= rear; i= (i+1)%maxsize)
printf("%d", cq[i]);
printf("%d", cq[i]);

}
```

Circular Queue (Linked List implementation) →

```
struct node
{
  int info;
 struct node *next;
} *front= NULL, *rear= NULL;
```

1) Insertion → concept

case 1 → Over flow condition
new= malloc()
if (new= = NULL)
printf("overflow");

Case2→ Queue is empty (16)

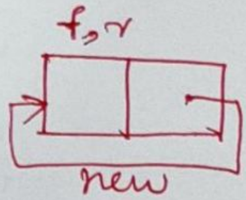
f, r = NULL

if (~~front~~ rear = = NULL)

front = new;

rear = new;

rear→next = front;

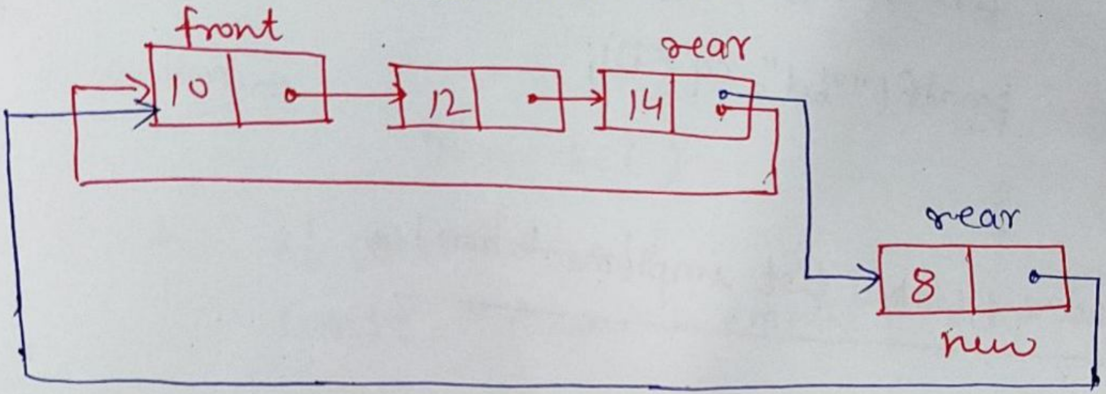


Case 3 –

rear → next = new

rear = new

rear → next = front



Algorithm→ Insert ( )

1. new = AVAIL

2. if (new = = NULL)
write "Overflow" & exit.

3. if (rear = = NULL) then
front = rear = new
~~rear→~~ LINK[rear] = front

else
LINK[rear] = new
rear = new
LINK[rear] = front.

4. Exit.

(17)

C function →

```
Void insert( )
{
  struct node *new;
new=(struct node *) malloc(sizeof(struct node));
 if(new==NULL)
 { printf("overflow");
   exit(0);
 }
 if (rear==NULL)
 {
   front=rear=new;
   rear→next=front;
  }
else
 {
  rear→next=new;
    rear=new;
  rear→next=front;
  }
}
```

2) Deletion → concept:-

f R
↓ ↓
NULL

Case 1 → Queue is empty

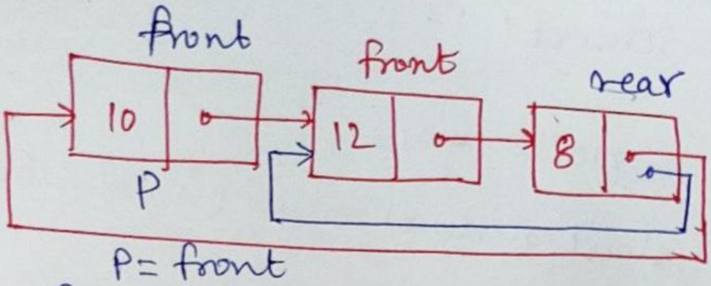
if(front==NULL)

printf("underflow");

Case 2 → Queue contains only one node

if(front==rear)
{
   p=front
front=rear=NULL
   free(p) }

case 3 → Queue contains more than one node. (18)



P = front

front = front → next

rear → next = front

free(P)

Algorithm →

deletion( )

1. if (front == NULL) then
   write " Queue is underflow " & exit.
2. P = front
3. if (front == rear) then
   front = rear = NULL
   free(P)
   else
   front = front → next;
   rear → next = front;
   free(P);
4. exit

3. if (front == rear) then
   front = rear = NULL
   free(P)
   else
   front = LINK[front]
   LINK[rear] = front
   free(P)
4. Exit.

C function →

```
void deletion( )
{
struct node *P;
if( front == NULL)
{
  printf(" Queue is underflow");
  exit(0);
}
```

```
    P= front;
  if (front== rear) then
    {
      front= rear= NULL;
      free(P);
    }
  else
    {
      front= front->next;
    rear->next= front;
      free(P);
    }
}
```

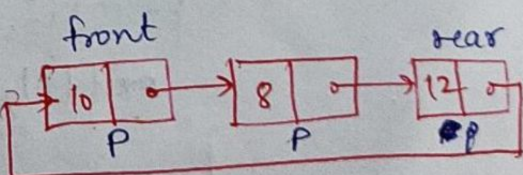
3) Traverse or display-

Concept ->

Case 1: Queue is empty

```
if (front== NULL)
  printf(" Queue is empty");
```

case 2:- Queue is not empty




```
      P= front;
  do
  {
  printf(" %d", P->info)
    P= P->next;
  } while(P!= front)
```

Algorithm

Traverse( )

1. if (front == NULL) then
write " Queue is empty" & exit.

2. P = front

3. ~~do~~ write INFO(P)

4. P = LINK[P]

5. while (P != front) repeat steps 5 & 6

6. write INFO[P]

6. P = LINK[P]

[END of while]

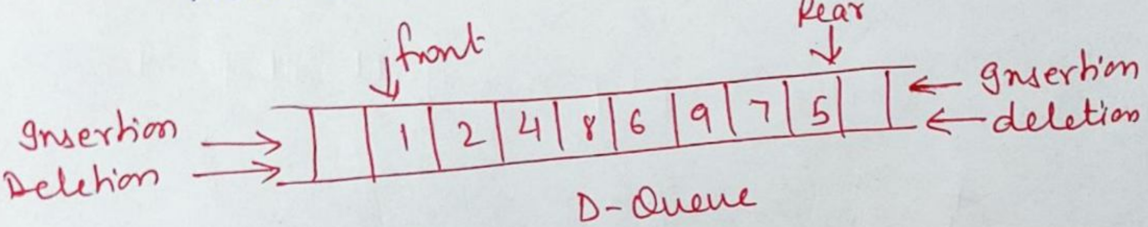
7. Exit.

C function →

```
void traverse()
{
struct node *p;
if (front == NULL)
{
  printf(" Queue is empty");
  exit(0);
}
 P = front;
  do
  {
    printf("%d", P->info);
    P = P->next;
  } while (P != front);
}
```

Double ended Queue → (D-Queue)

In a D-Queue both insertion and deletion operations are performed at either end of the queue. i.e, we performed insertion and deletion from rear as well as front end.



D-Queue

Types of D-Queue →

(i) **Input restricted D Queue :→** Element can be added at only one end but we can delete the element from both end.

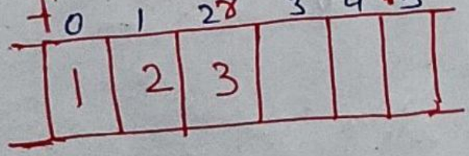
(ii) **Output restricted D Queue →** Element can be deleted only from one end but allow Insertion at both ends.

Operations on D-Queue → (Array implementation)

(1) **Insertion at front :→**

Concept →

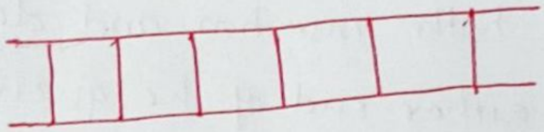
case 1 → Queue is full at front




```
if (front==0)
    printf("overflow");
```

Case2 → Queue is empty

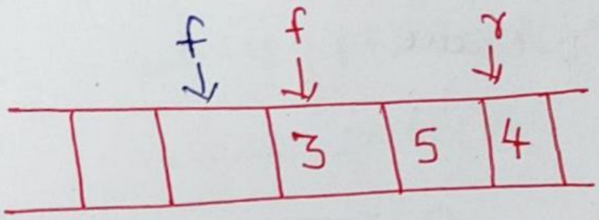
f=-1
r=-1



if (front==-1 or rear==-1)

front = rear= 0

case 3:- Queue is neither empty nor full at front



front= front-1;

DQ[front]= item

Algorithm→

Insert_at_front ( )

1. Read item

2. if (front==0) then
write "Overflow" & exit

3. if (front==-1)
front=rear=0

else
front= front-1

4. DQ[front]= item

5. Exit