QUEUE

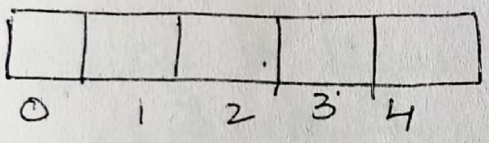
→ It is a linear data structure in which insertion take place at one end called the rear end and deletion takes place at other end called the front end.

→ It works on basis of FIFO or FCFS (First come first serve) (First in First out)

→ Example of Queues are
- Reservation lines at railway station
- Queue at bus
- Movie theatre counter.
- Print jobs submitted to a n/w printer.

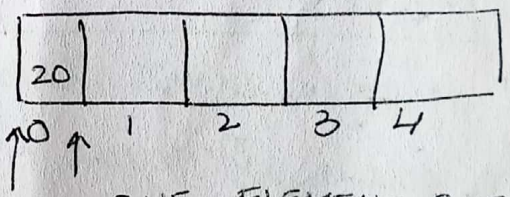
→ ~~Operation on Queue~~

R = -1 & F = -1



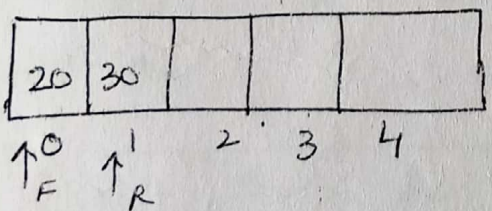
EMPTY QUEUE

R=0 & F=0



ONE ELEMENT QUEUE

R=1 & F=0



TWO ELEMENT QUEUE

Note:

1) Rear = Rear + 1 when element is added
2) During insertion of first element in the queue, we always increment the Front by one
   i.e. Front = Front + 1

During entire insertion Front value will not change

3) Whenever element is deleted or removed from queue the value of Front is incremented by one i.e.

Front = Front + 1

QUEUE IMPLEMENTATION

1) Static implementation (using array)
2) Dynamic implementation (using pointers)

→ When Queue is implemented using array of array then exact size should be known at design time or before processing starts.

→ Total no. of elements present in queue can be known by foll.

~~front - rear + 1~~ [REAR - FRONT + 1]

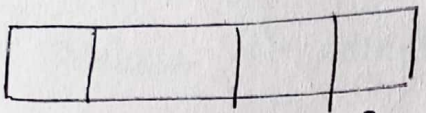
→ If rear < front then there will be no element in queue or queue is always be empty.

OPERATION ON QUEUE

1) INSERTION
2) DELETION

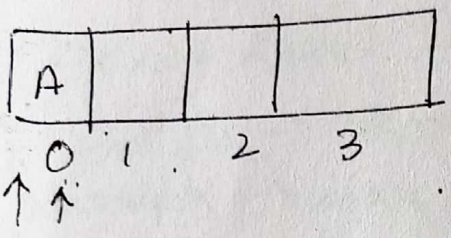
Example

Initially if Queue is empty, then Rear & Front = -1, capacity is i.e Max. size = 4



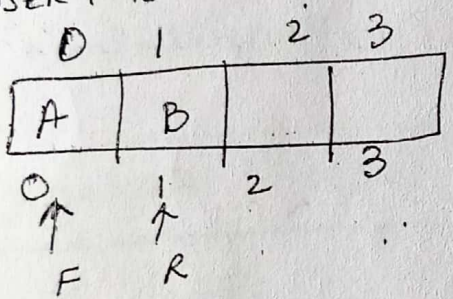
F = -1
R = -1

INSERT A



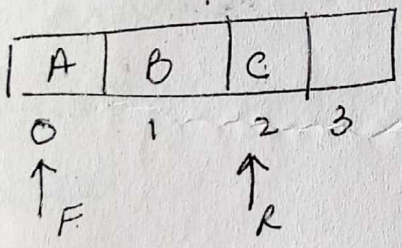
F = 0
R = 0

INSERT B



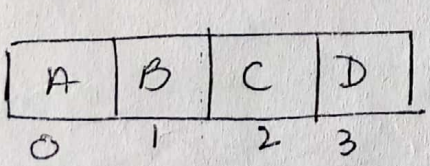
F = 0
R = 1

INSERT C



F = 0
R = 2

INSERT D



F 0
R = 3

INSERT 'E'

QUEUE IS FULL SO insertion not possible

1. INSERTION

Qinsert (maxsize, item)

1) Initialize Front = -1 and Rear = -1
2) If Rear >= Maxsize - 1
   Print Queue Overflow & return
   else:
   Set Rear = Rear + 1
3) Queue [Rear] = item
4) If (Front == -1) // SET FRONT POINTER
   front = 0
5) Exit.

Program

```
Void insert (int item)
{
 if (rear >= maxsize)
   {
   Printf ("Queue overflow");
     break;
   }
else
 {
   rear = rear + 1;
   Queue [Rear] = item;

}
```

ALGO FOR DELETION FROM A QUEUE

Delete (Maxsize, item)

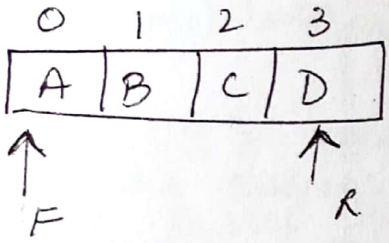
1. if front < 0 [check underflow condition]
   Print Queue is empty & return
2. else: item = Queue [Front] [remove element from front]
3. Find new value of front
   if (Front = Rear) [checking for empty Queue]
   Set Front = -1, Rear = -1 [re-initialize the pointers]
   else:
   Front = Front + 1

Program 'C code'

```
void delete()
{ if (Front < 0)
  { printf("Queue is empty");
    break;
  }
else
  { item = Queue[Front]
    Front = Front + 1;
    printf("item deleted = %d", item);
  }
}
```

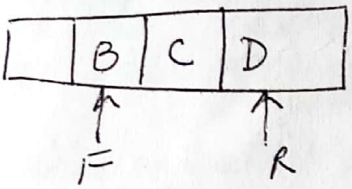
Example.

Suppose we have a Queue of 4 elements stored in it



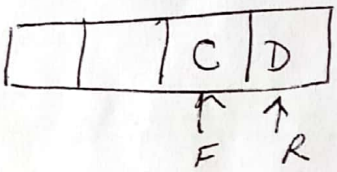
F=0 R=3

AFTER DELETING FIRST ELEMENT



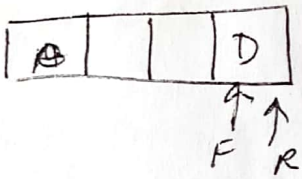
F=1
R=3

AFTER DELETING SECOND ELEMENT



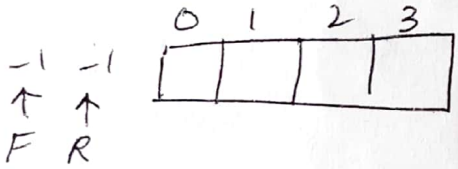
F=2
R=3

AFTER DELETING THIRD ELEMEN



F=3
R=3

AFTER DELETING FOURTH ELEMENTS



AGAIN DELETING

Queue is empty.

CIRCULAR QUEUE

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

→ In other words if we have a queue Q of say n elements, then after inserting an element last (i.e. in the n-1 th) location of the array the next element will be inserted at the very first location (i.e. location with subscript 0) of the array. & It is possible to insert new elements, if & only if those location (slots) are empty.

→ Circular queue is one in which the first element comes just after the last element.

ADVANTAGE OF CIRCULAR QUEUE

→ Circular queue overcomes the problem of unutilized space in linear queues implemented as arrays.

The following assumption are made in circular queue:

(i) FRONT will always pointing to the first element (as in the linear queue)

(ii) If FRONT = REAR the queue will be empty

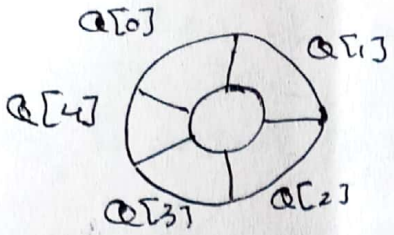
(iii) Each time a new element is inserted into the queue the Rear is incremented by one

Rear = Rear + 1

(iv) Each time an element is deleted from the queue the value of FRONT is incremented by one.

FRONT = FRONT + 1

CIRCULAR QUEUE



INSERTION

→ The position of the element to be inserted ~~would~~ will be calculated by the relation:

Rear = (Rear + 1) % MAXSIZE

QUEUE[REAR] = VALUE

DELETION

The deletion method for a circular queue also require modification as compared to linear queues.

FRONT = (~~R~~ FRONT + 1) % MAXSIZE

ALGORITHM OF INSERTION

QINSERT (Queue[MAXSIZE], item)

1. If (Front == (Rear+1) % MAXSIZE)
   Write Queue Overflow and Exit
   Else: Take the value
   → If (Front == -1)
   Set Front = Rear = 0.
   else Rear = ((Rear+1) % MAXSIZE)
   [Assign value] Queue[Rear] = Value.
   [End if]
2. Exit.

Implement Circular Queue Insertion in C

```
insert ( )
{ int num;
  if (front == (rear+1) % MAXSIZE)
    { printf ("QUEUE IS FULL");
      return;
    }
  else
  { printf ("ELEMENT TO BE INSERTED")
    scanf ("%d", &num);
      if (front == -1)
        front = rear = 0;
        else
        rear = (rear+1)% MAXSIZE
        Q[rear] = num;
  }
  return;
}
```

DELETION

ALGO OF DELETION

QDELETE (Queue [MAXSIZE], item)

1. if (FRONT == -1)
write Queue underflow and Exit
else: item = Queue [FRONT]
if (front == Rear)
set FRONT = -1
Rear = -1
else: Front = (Front + 1) % MAXSIZE
[End if structure]

2. Exit

Deletion in C

```
int Qdelete( )
{
  int num;
  if (front == -1)
  {
    Printf("QUEUE IS EMPTY");
        return;
  }
else
{ num = Q[FRONT];
  Printf ("DELETED ELEMENT IS = %d \n", Q[FRONT]);
    if (front == rear)
      front = rear = -1
    else
    front = (front+1) % MAXSIZE;
}
return(num);
}
```

PRIORITY QUEUE

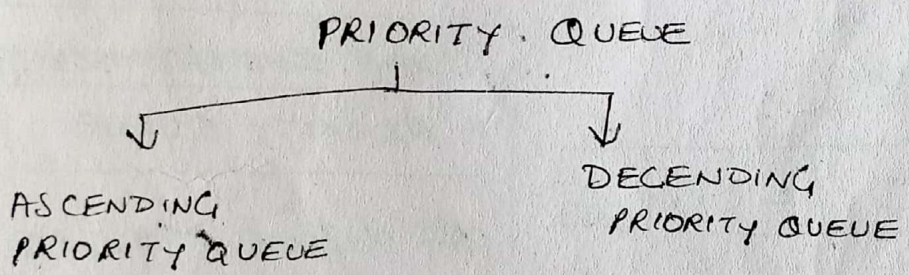
→ In priority queue, elements are arranged on the basis of associated priority number & order in which elements are processed & deleted comes from the foll. rules:

Rule 1:

The element having the higher priority is processed before any element of lower priority or vice-versa

Rule 2:

The two elements that have the same priority get processed according to the order in which they are inserted in priority queue.



Two types of Priority queue:

(1) **Ascending priority queue:**

→ In this elements are stored in ascending order priority.

→ OR,

It is a collection of items into which items can be inserted arbitarily but from which smallest item can be removed.

- SO if Pq is priority queue, operation Pq.insert(Pq, x)

insert element x into pq ~ Pq.remove(Pq) removes the minimum element from Pq & return its value.

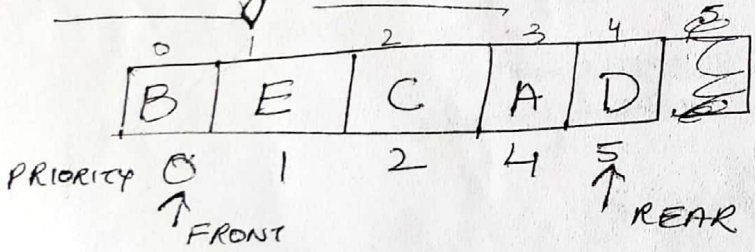
(2) Descending priority Queue:

In this queue elements are stored in descending order priority.

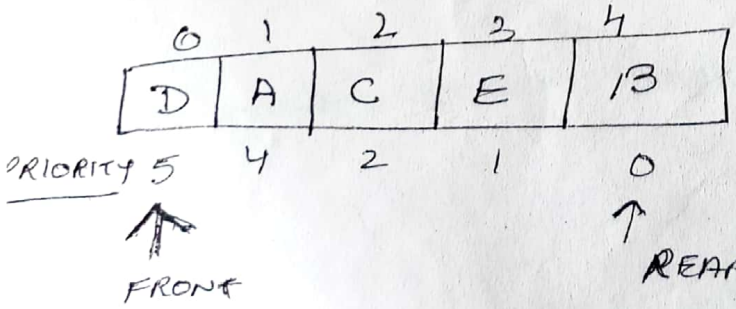
E.g.

	DATA/JOB	PRIORITY
1	A	4
	B	0
	C	2
	D	5
	E	1

Ascending Queue



Descending Queue



ARRAY IMPLEMENTATION OF A PRIORITY QUEUE

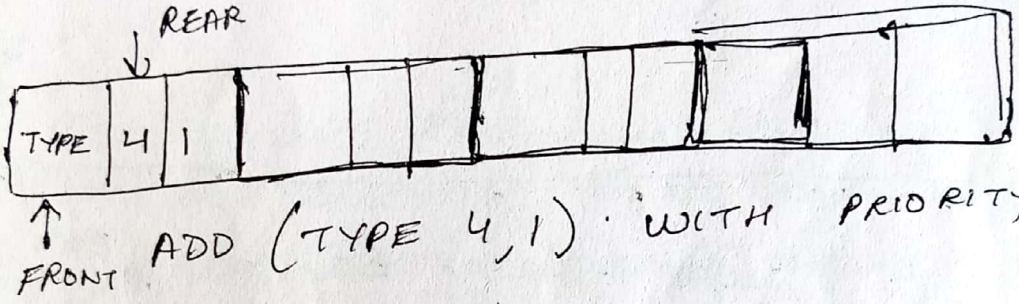
```
#define MAX_PQ 100
struct Pqueue {
    int items[MAX_PQ];
    int front, rear;
};
struct Pqueue Pq;
```

INSERTION IN PRIORITY QUEUE

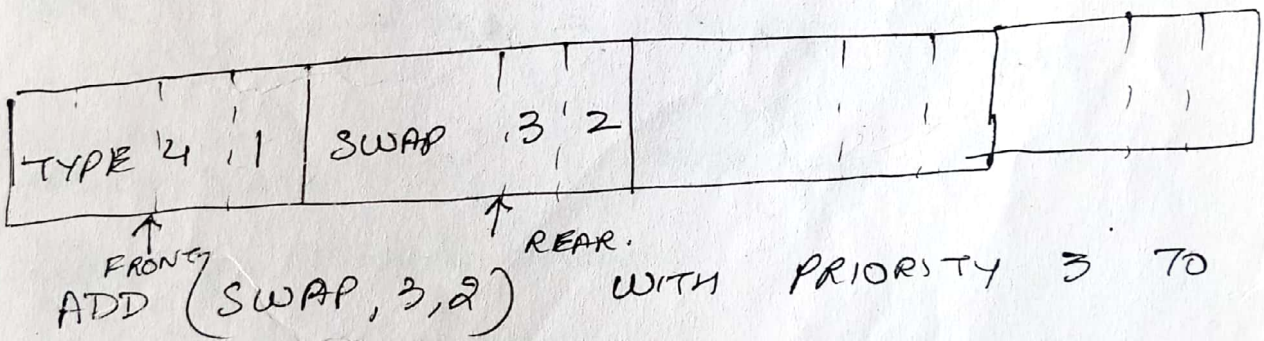
→ Rule is lower the priority no. higher is the priority.

→ Add the jobs to the queue & arrange them priority wise

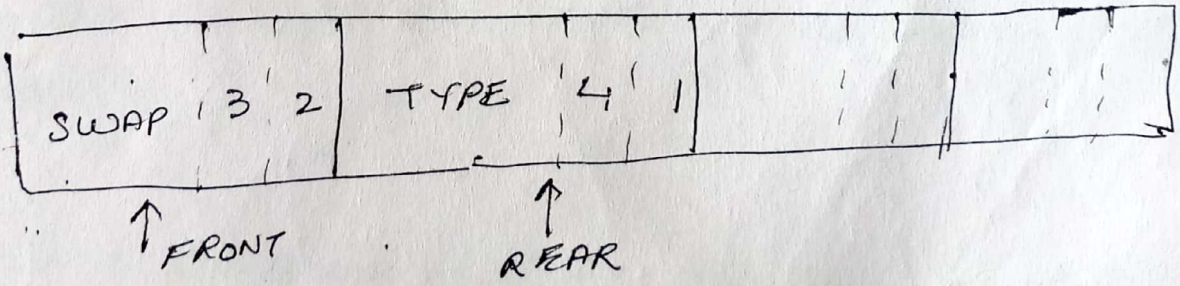
→ If the priority of two jobs is same then arrange them order wise, i.e. the order in which they are added.



ADD (TYPE 4, 1) WITH PRIORITY 4

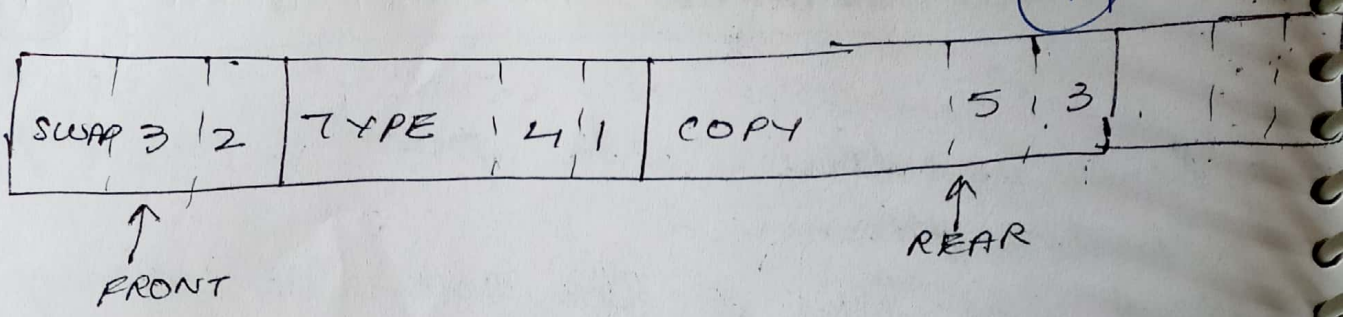


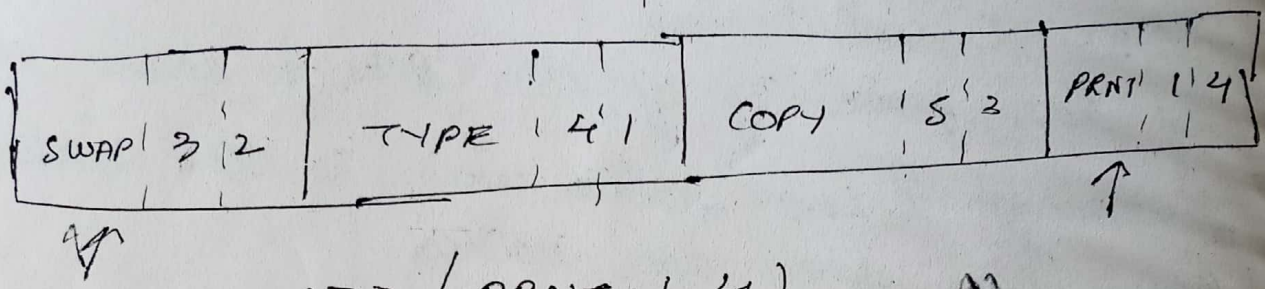
ADD (SWAP, 3, 2) WITH PRIORITY 3 TO THE QUEUE



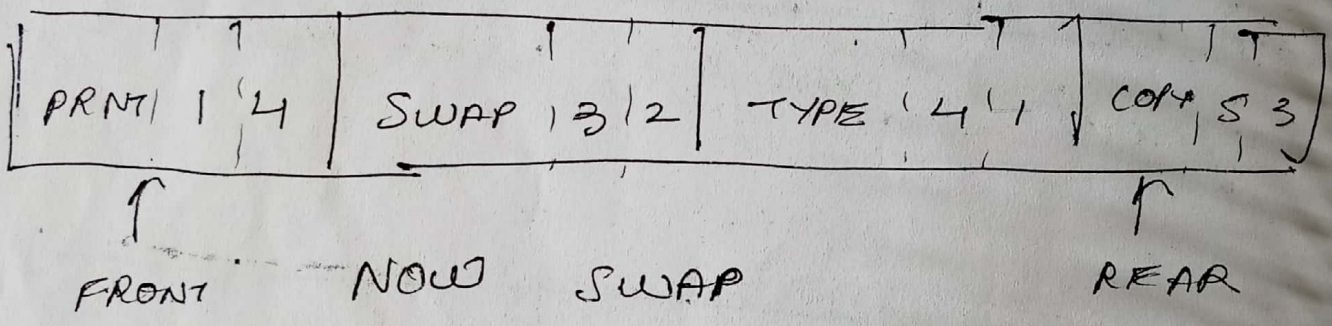
NOW SWAP ACCORDING TO PRIORITY

(14)





NOW ADD (PRNT 1, 4)



NOW SWAP

ALGO FOR CIRCULAR PRIORITY QUEUE

15

INSERTION

1) [check if Queue is full]
if (front == 0 && rear == size - 1) || (front == rear + 1)
print ("Queue is full");
return;
2) else [increment or set the value of rear]
// check whether rear points the last position of the queue
if (rear == size - 1) then
rear = 0
3) else [check for condition in which there is no element in the queue]
if (Rear == -1) then
Rear = 0
front = 0
4) else
Rear = Rear + 1
5) [store the value in the queue at rear position]
q[rear] = item.
6) [sort the queue in ascending/descending order depending on the priority queue]
sort (Front, Rear, Pq)
7) Exit

Deletion in priority queue

1) [check for empty queue]
   if (front == -1)
   print ("queue is empty")
   return

2) else [store value at front into temp]
   temp = q[front]

3) [Now set the value of front, check condition in which there is a single element]
   if (rear == front) then
   Rear = front = -1

4) [check whether front points the last element]
   else if (front == size-1)
   front = 0

5) else [increment the front]
   front = front + 1

Note → This algo is same as deletion algo of circular queue ~~by~~ but item deleted will be either max or min.

Application of queues

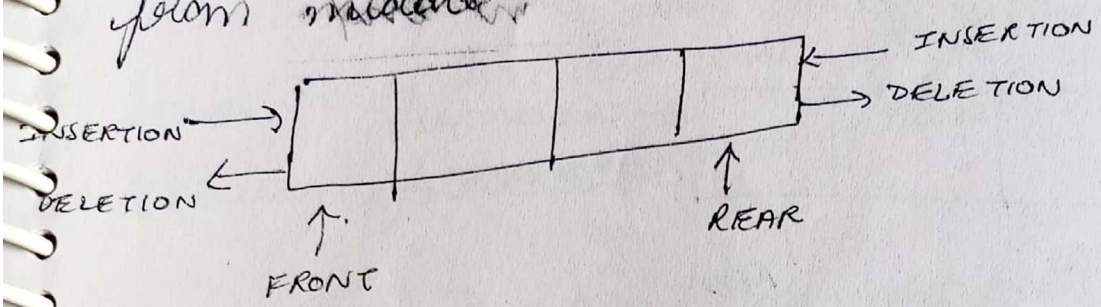
(1) Round Robin technique for processor scheduling is implemented using queues.

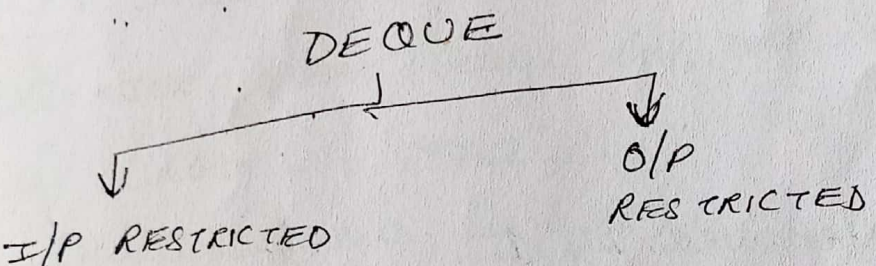
(2) All types of customer service (like railway ticket reservation) center softwares are designed using queues to store customers information.

(3) 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.

Double ended queue (DEQue)

→ Deque is also a homogenous list of element in which insertion & deletion operation can be performed from either end but not from middle.



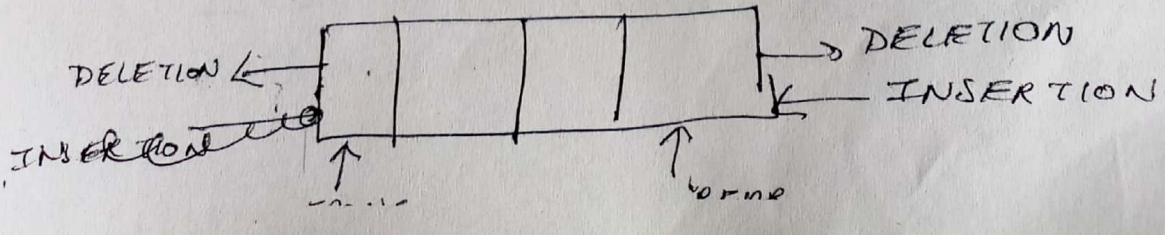


Two types of Dequeue

1) Input restricted dequeue. (Insertion one end.
2) Output restricted dequeue (Deletion one end

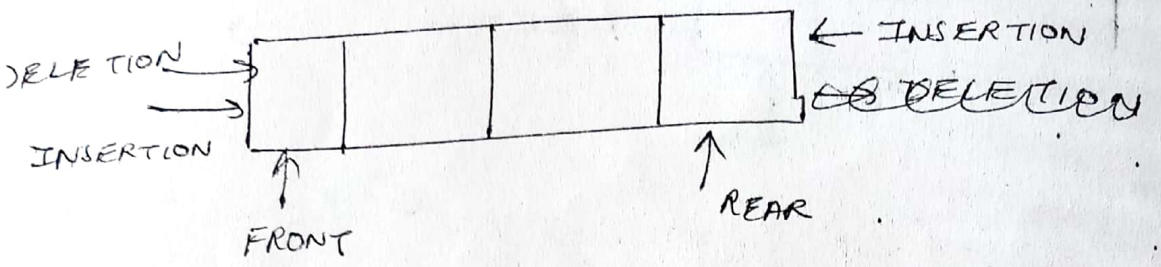
1) INPUT RESTRICTED DEQUEUE:

- It allows insertion at one end and deletion at both end.
- It can insert element from the rear end of the queue & can delete elements from both end i.e. front & rear.



(2) OUTPUT RESTRICTED DEQUE

- Allows deletion at only one end but insertion at both end.
- It can insert elements from the both end i.e. from front end & from rear end & can delete only from the front end.



Note:

In deque

→ front is decreased by 1
i.e. front = front - 1 when element inserted from front end.

→ front is incremented by 1
i.e. front = front + 1 when element deleted from front end.

→ Rear is incremented by 1
i.e. Rear = rear + 1 when element inserted from rear end

→ ~~front~~ Rear is decremented by 1
i.e. Rear = Rear - 1 when element is deleted from ~~front~~ rear end.

Since both insertion & deletion are performed from either end, it is necessary to design algo for foll. four operations:

1) Insertion of an element at the rear end
2) Deletion of an element from the front end
3) Insertion of an element at the front end
4) Deletion of an element from the rear end.

ALGO 1: Insertion of an element at REAR END

```
DQINSERT_REAR (Queue, int, ~~int~~ item)
if (rear == Maxsize-1)
  Print (Queue is full)
else
  rear = rear+1
  q[rear] = item.
```

ALGO 2: Deletion of an element from the FRONT

```
DQ DELETE_FRONT (Queue)
if (front == -1)
  Print (queue is empty)
    exit(0)
  }
else {
    item = q[front]
    Front = front+1;
    Return (item)
  }
```

ALGO-3

Insertion of an element at front end (21)

DQ INSERT_FRONT (queue, int item)

* (1) [check for space at front end]
if (front == 0 && Rear = Max-1) || (front = Rear+1)
Print ("Queue is full") // NO space for element at front side

(2) else [Insert item at front]
front = front - 1;
q[front] = item;

Algo 4: Deletion of an element at Rear END

DQ DELETE_REAR (queue q)

step 1) ~~if (front == Rear)~~ if (Rear == -1)
Print ("queue is empty")
return

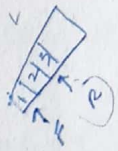
2) else
item = q[Rear]
Rear = Rear - 1

Algo 5: Algo to display the contents (status) of a queue.

DQ DISPLAY (queue q).

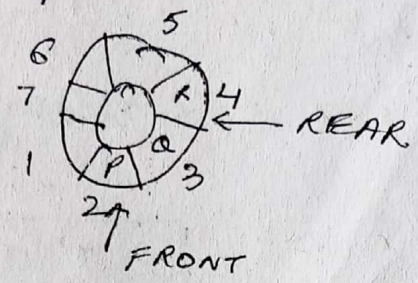
1 if (front <= Rear ~~== 0~~)
Printf ("Status of queue" "\n")
for (i = front; i <= rear; i++
printf ("%d" a[i])

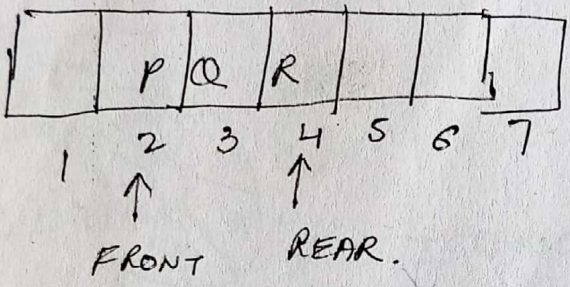
```
    }
    else
    printf ("queue is empty");
```



EXAMPLE:

Perform foll. operation over deque



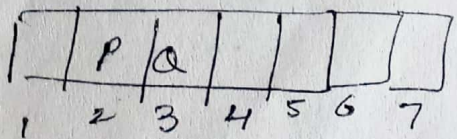


(a) Deletion of last element
(b) Insertion of element O at front end
(c) Insertion of element R, S & T at the rear end
(d) Insertion of U and V at the front-end
(e) Insertion of element I at rear end
(f) Insertion of element N at front-end.

Soln:

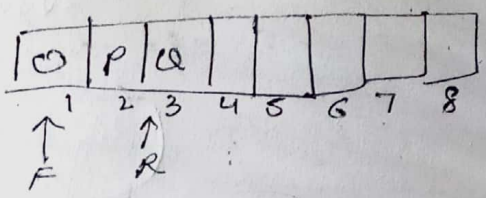
(a) Rear = Rear - 1

Rear = 3



b) Front = Front - 1

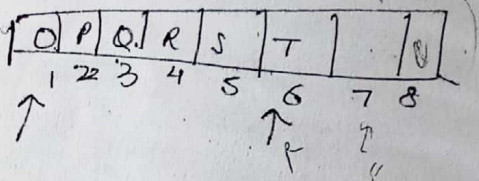
Front = 1



c) Insertion of R, S, T at rear end.

Rear = Rear + 3

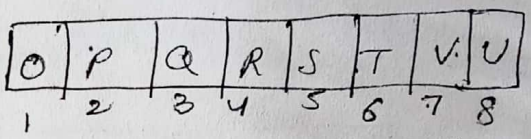
Rear = 6

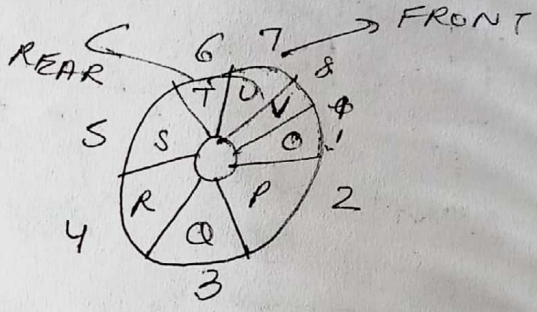


d) Insertion of element U & V at front end.

front = 7

Rear = 6





e) Insertion of element I at rear end

this operation result in an overflow.

f) Insertion of element N at front-end.

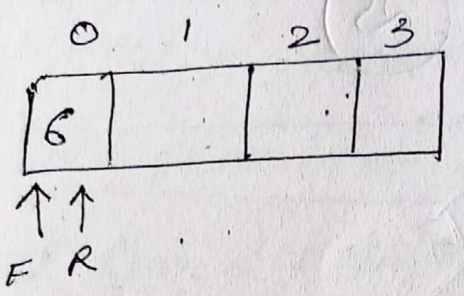
This operation result in an overflow.

INPUT RESTRICTED DEQUE

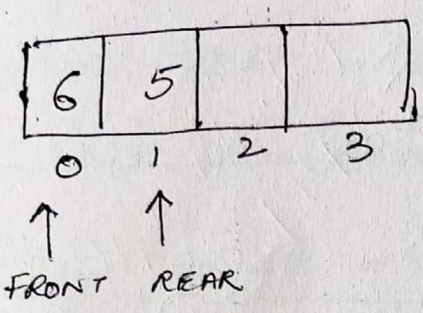
INSERTION ONLY FROM REAR & DELETION FROM FRONT & REAR

24

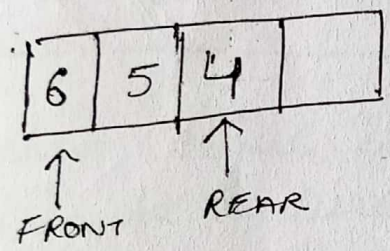
(i)



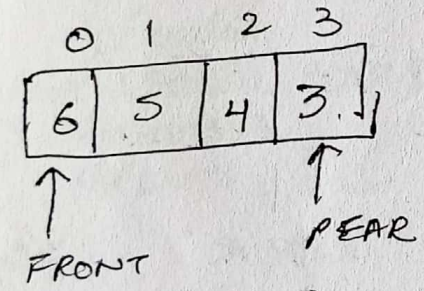
(ii)



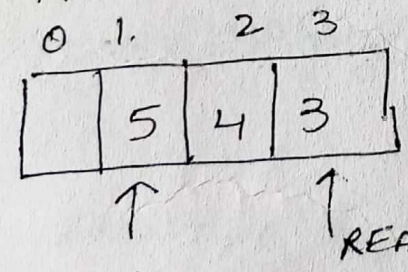
(iii)



(iv)



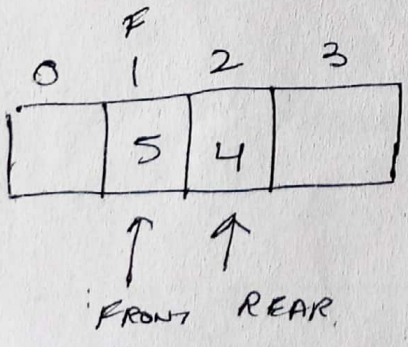
(v)



DELETION FROM FRONT

FRONT = 1

(vi)



DELETION FROM END

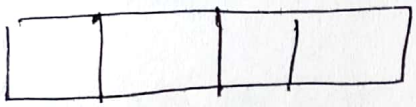
REAR = REAR - 1

OUTPUT RESTRICTED DEQUEUE

DELETION FROM FRONT BUT INSERTION FROM FRONT & REAR

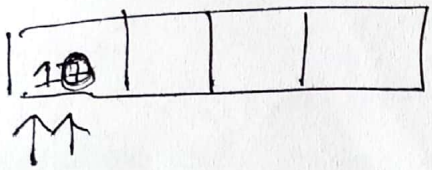
25

(i)



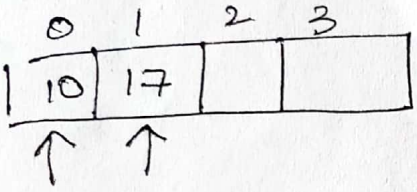
F = -1
R = -1

(ii)

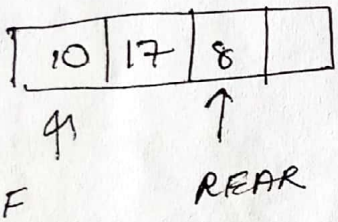


F R

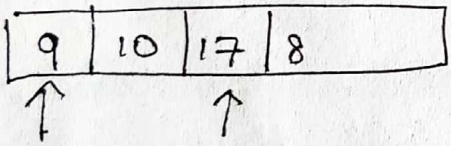
(iii)



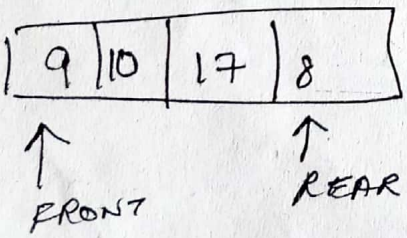
(iv)



(v) ADD 9 AT BEGINNING SO THE SHIFT ELEMENT RIGHT



NOW REAR WILL POINT TO 8



IMPORTANT POINTS

→ Number of elements in a circular queue is :

(a) if FRONT ≤ REAR

$$N = REAR - FRONT + 1$$

(b) If Rear < FRONT

then FRONT - REAR - 1 is the no. of empty cells.

∴ ~~N = N - (FRONT~~

$$N = MAXSIZE - (FRONT - REAR - 1)$$

$$N = MAXSIZE + REAR - FRONT + 1$$

(c) → when Deque is circular queue same above formula is used.

when a circular queue array is full :

(i) FRONT = 1 & Rear = N [when LB = 1]

OR

(ii) FRONT = REAR + 1

(d)

(i) FRONT = 0 & Rear = MAXSIZE - 1 [when LB = 0]

OR

(ii) FRONT = Rear + 1

Queue end →

Multiple Queue

MULTIPLE QUEUE

27

→ It is possible to maintain two queues in the same array.

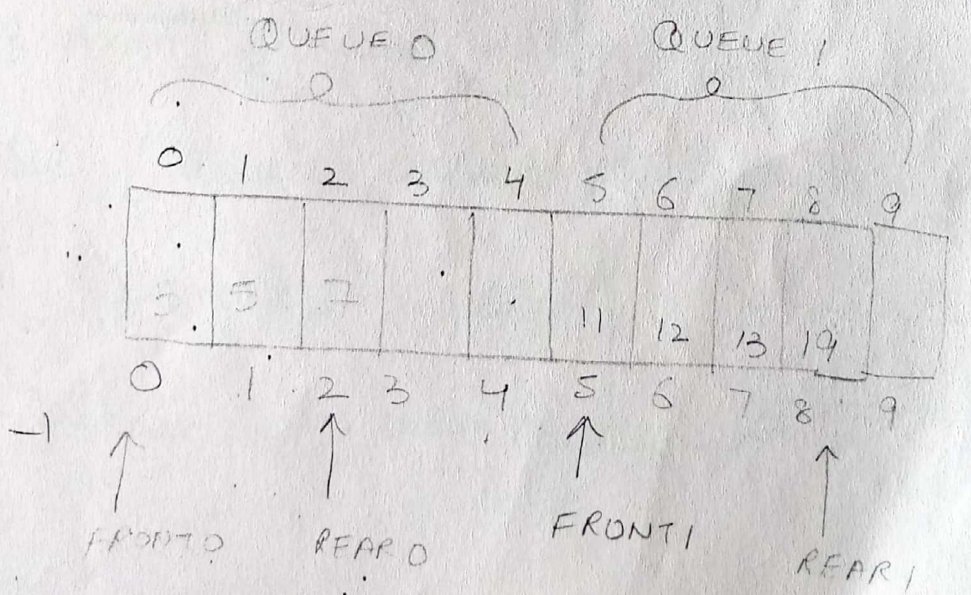
→ One can grow front position 0 of the array other can grow from the last position.

→ So to maintain two queues, there should be two front & two rear of two queue.

→ Both queues can grow upto any extent from 1st position to maximum

→ So we require one more variable to keep track of total no. of values stored & overflow condition is checked by this, if its value is equal to array size.

- if (count = array size) → overflow
- if count = 0 → underflow (it means queue is empty, there is no element)



front 0 = rear 0 = -1
front 1 = rear 1 = 4
] when both queue are empty.

Structure of multiqueue

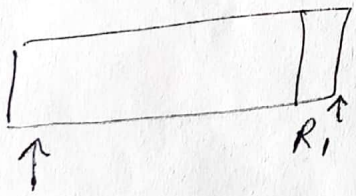
```
struct multiqueue
{
  int front0, front1;
  int rear0, rear1;
  int num[100];
  int count;
};
```

→ Initially both rear 0 & front 0 will be equal to -1 for the 1st Queue that grows from first position.

→ Front 1 = Rear 1 = size for the second queue that grows from the last position.

→ Multiple queue condition to check

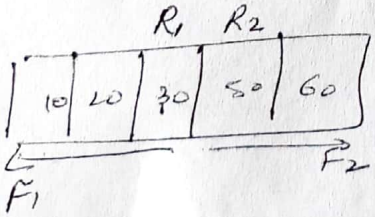
if $F_1 = 0$ & R_1 = ~~Max~~ Max size - 1



OVER FLOW

→ if $F_1 = 0$

→ if $R_2 = R_1 + 1$



Multiqueue - insertion

```
Multiqueue (queue q, int item, int queue_no)

(1) [check for overflow]
    if (count == size)
    {
      print "Overflow"
      return
    }
(2) else
  [check ~~for~~ in which queue item is to be inserted]
  if (queue_no == 0) then  // [queue starts from 1st position]
    if (q.rear 0 == -1)      // No element in the queue. New item would be the first item.
      q.rear 0 = q.front 0 = 0
  else
    q.rear = q.rear + 1
    q.num [q.rear] = item.

(3) else [queue which start from last position]
    if (q.rear 1 == size)
    {
      q.rear 1 = q.front 1 = size - 1
    }
    else
      q.rear 1 = q.rear 1 - 1;
      q.num [q.rear 1] = item;
}
```

Deletion from multiqueue

```
Delete (Queue q, int queue_no).

Step 1: [check for underflow]
    if (count == 0)          // No item in multiqueue
    Print ("Overflow")
    return
    }

Step 2: else                 // check from which
    if (queue_no == 0) then  queue deletion will
                             take place
    if (q.front0 == -1) then // Queue_no == 0 means
    {                        queue grows from
    Print ("No item in this queue")  1st position of an
    return                   array.
    }
else
{   item = q.num[q.front];
[Now check item deleted is last item in the
 queue or not]
if (q.front0 = q.rear0)      // last one in the
    q.front0 = q.rear0 = -1  queue so after
else                         the deletion of this
    q.front = q.front + 1    queue will be
                             empty.
}
```

3) else

[check for whether there is some item in this or not]

// Queue is the 2nd one which grows from last position //

```
if (q.front1 == size)
{
Print ("No item in this queue")
return
}
else
item = q.num[q.front1];
```

Now check for item is the last one in queue so after this queue will be empty)

```
if (q.front1 == q.rear1)
{
q.front1 = q.rear1 = size
else
q.front1 = q.front1 - 1
}
```

End of Queue