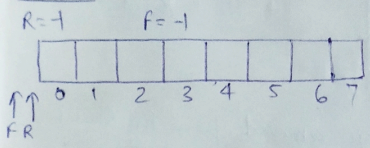
Queues

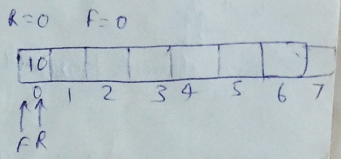
A Queue is a structure in which new elements are added at one end (Rear end) & the existing elements are deleted from other end (front end).
It is a FIFO type of list

Array Representation :-

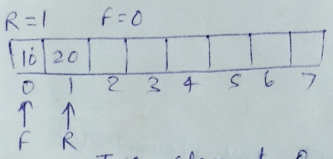
(Insertion)



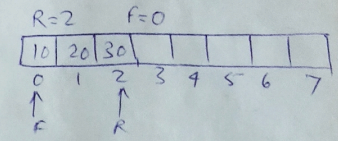
Empty Queue



One element Queue



Two element Queue

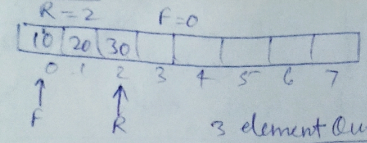


Three element Queue

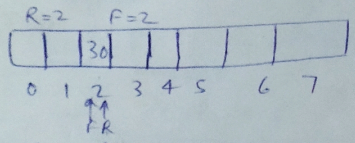
When an element is inserted in the Queue, the value of Rear is incremented by one

During the insertion of the 1st element in the front is always incremented by one. After that the front will not be changed.

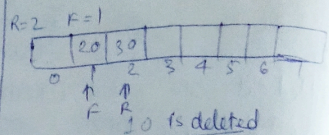
(Deletion)



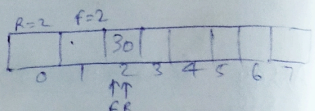
3 element Queue



Two elements (~~10~~,20) deleted from front



10 is deleted

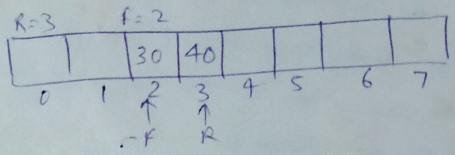


20 is deleted

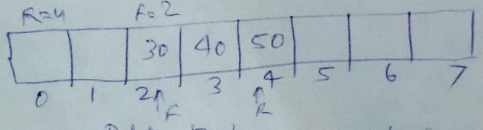
when an element is removed from the Queue the value of Front is incremented by one

Suppose that if we insert any element the above obtained Queue, then it will go the next position of the previous inserted element

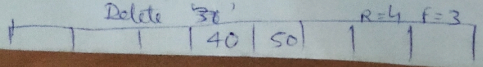
⇒ Insert '40'



⇒ Insert '50'



⇒ Delete '30'



Implementation of Queue –

It can also be implemented in 2 ways as other Lists –

1. static (using arrays)
2. Dynamic (using pointers)

static – size of array is declared at design time or before the processing starts. i.e. exact no. of elements to be stored in Queue should be known.

In this case –

(front) = Begining of the array = LB

(Rear) = Last location of array = UB

So, Total no of elements present in Queue = R-F+1

If (R<F) then there will be no element in Q (when R & F are +ve integers, Empty Queue)

→ Total = R-F+1

0 = 2-3+1 (R=2 F=3)

= 1-2+1 (R=1 F=2)

= 0-1+1 (R=0 F=1)

and if R=-1 , F=-1 , then 'Empty Queue'

Dynamic – size of Queue may be decided at run time. It is implemented using pointers

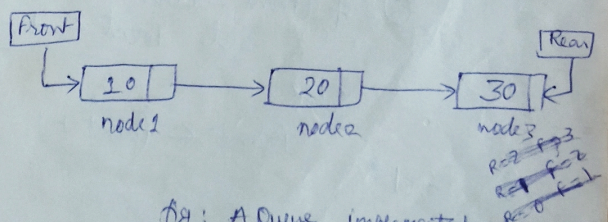


Fig: A Queue implemented using pointers

Operations on Queue :-

① Insertion.

Algo :-

@INSERT (MAXSIZE, ITEM)

① Initialize front ← 0 and Rear ← -1.

② If Rear >= MAXSIZE.
Print @ Overflow and return // Check overflow cond.
else
Set Rear ← Rear + 1 // At new val. of Rear

③ Queue[Rear] ← Item

End

② Deletion.

Algo.

@DELETE (MAXSIZE, ITEM)

① If front < 0
Print Queue is empty & return // Check underflow cond.
else
item ← Queue [Front]

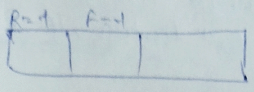
② If front = Rear // Find rear of front
set front ← -1 Rear ← -1
else
front ← front + 1

End

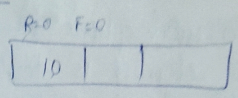
③ Create

Create@ ← It creates Q as an empty Queue
— declare Q(1 : n) ; front ← rear ← 0

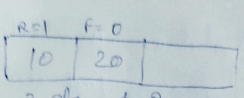
Limitation of a simple Queue



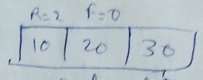
Queue Initially



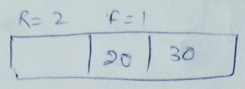
One element Queue



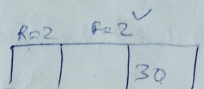
2 element Queue



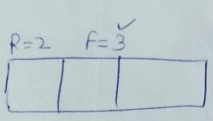
3 element Queue



On deleting an Element



On deleting an elemt



on deleting next elemt

~~Rear~~

~~Rear end = last location of Array~~

If we want to insert an element, it should be inserted to the rear end. But ~~here~~ now, element can't be added because Rear (Rear end = last location of Array)

To remove this problem CIRCULAR QUEUE will be used.

Circular Queue -

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

It is possible to insert new elements, if already if those locations are empty.

i.e. A circular Queue is a Queue in which the first element comes just after the last element. It appears as a loop of wire, whose both ends are connected together.

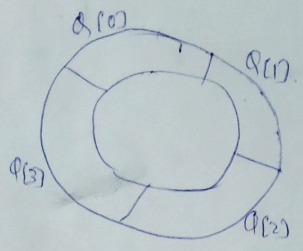


fig: A circular Queue (4 elements can be added at a time)

Algo for I/D's

QINSERT (Q[MAXSIZE], ITEM)

If (F == (R+1) % MAXSIZE)
    Write Overflow & exit
else if (F = -1)
    set F ← 0
        R ← 0
else R ← ((R+1) % MAXSIZE)
    Q[R] ← ITEM
③ Exit

QDELETE (Q[MAXSIZE], ITEM)

① If (F = -1)
    Write Underflow & exit
else ITEM ← Q[F]
② If F = R
    set F ← -1
        R ← -1
else F ← ((F+1) % MAXSIZE)
③ Exit

Algos for ...

Double Ended Queue (D-Queue/DeQue)

It is Queue in which both Insertion & Deletion are performed at the either end.

There are 2 types of D-Queue, due to the restrictions put to perform either the Insert^n or delet^n only at one end.

1. Input - restricted D-Q
2. Output - restricted D-Q

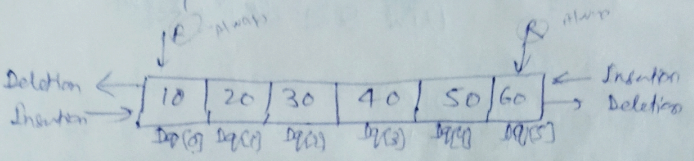


fig. A D-Que with of 6 elements

In a D-Q 4 types of operations can be performed

1. Insertion of an elemt at the Rear end
2. _____ Front
3. Deletion _____ Front ✓
4. _____ Rear

× [Input restricted D-Q : All operations are valid
Output _____ : 1 2 3 are valid]

Algo^s for above operations —

1. DQ (Q(MAX), ITEM)
   If R = (MAX+1)
   Print Q is full & exit
   else
   R ← R+1
   Q[R] ← ITEM
   End
2. DQ (Q(MAX), ITEM)
   If (F=0)
   Print Q is full & exit
   else F ← F-1
   Q(F) ← ITEM
   End

③ DQ (Q(MAX), ITEM)

```
If F<0
    Print Q is empty & exit
else
    ITEM ← Q[F]
    F ← F+1
end
```

④ DQ (Q(MAX), ITEM)

```
If R >= MAX
    Print Q is empty & exit
else
    ITEM ← Q[R]
    R ← R+1
end
```

Priority Queues –

The elements are inserted/deleted according to their assigned priority.

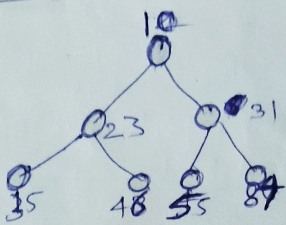
In stack or simple Queue the operations are performed according to their insertion time. The key matching But here Priority is taken into consideration.

In priority Queue, elements are sorted by priority & proceeded from Top to Bottom, maintaing a pointer. We have following basic priority Queue choices —

① Sorted array or list – Elements are sorted out according to priority. Deletion of the smallest element from the sorted list is very easy, but Insertion of a new element in a sorted list is a little bit difficult.

Thus sorted arrays are suitable where few insertions occur into the priority Queue

(7) Binary Heaps –



// heap sort

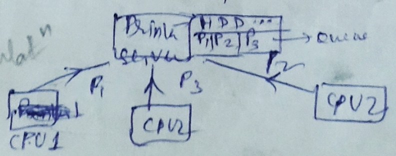
Binary Heaps form a structure in which key of the root of any subtree is less than all its descendents. Thus min key is always at the root of the tree.

If a new element is inserted, it is placed on the leaf node first. Then element is relocated upward until it gets its proper location.

Application of Queues: –

1. In CPU Scheduling priority Queue is used
2. All type of customer service center s/w's (for railway reservat^n) are designed using queues to store customer info^n
3. Printer are working on the concept of Q
4. Printer server routines are also working on this principle.

(5) Modelly & Simulat^n



// spooling/o.s.