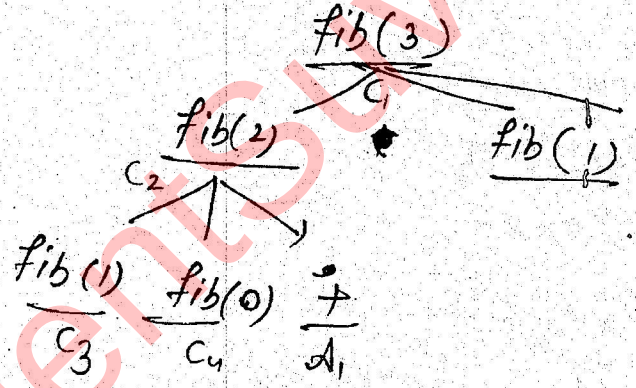
"Fibinocci Series" :-

```
f(n)
{
  if (n==0 || n==1)
     return n(n);
  else
  {
  return ( fib(n-1) + f(n-2))
  }
}
```

★ i) In fibinocci of n i.e. f(n) after How many function calls 1st addition will be taken = $\overline{n+1}$ After C_4



★ ii) In fib(n), is there any function call after last addition No

★ iii) Is there any addition after last function call Yes.

★ iv) In fib(n) How many function calls are there ?

$f(0) = 1$

$f(1) = 1$

$f(2) = 3$

$f(3) = 5$

$f(4) = \underbrace{f(3)}_{5} + \underbrace{f(2)}_{3} + 1 = 9$

$f(50) = f(49) + f(48) + 1$

$f(5) = \underbrace{f(4)}_{9} + \underbrace{f(3)}_{5} + 1 = 15$

:- $\boxed{T(n) = T(n-1) + T(n-2) + 1}$ function calls.

*v) In fib(n) How many additions ?

$f(0) = 0$

$f(1) = 0$

$f(2) = 1$

$f(3) = 2$

$f(4) = \underbrace{f(2)}_{1} + \underbrace{f(3)}_{2} + 1 = 4$

$f(5) = \underbrace{f(4)}_{4} + \underbrace{f(3)}_{2} + 1 = ?$

$\boxed{T(n) = T(n-1) + T(n-2) + 1}$ Additions

Parameter passing Techniques :-

1.) call by value
2.) call by Reference
3.) call by copy-restore
4.) call by Name
5.) call by Need
6) call by Text

1) call by value :-

```
main()
{
    int a=10, b=20;
    Pf(a,b);
    Swap(a,b);
    Pf(a,b);
}

Swap(int c, int d)
{
    int t;
    t = c;
    c = d;
    d = t;
}
```

If compiler supporting 'call by reference', No Need to put '&' for address, Address will be passed by bydefault.

★ Note :-

i) In call by value originals will no effect because modification is done on copy not in original.

ii) If you want to pass very less parameters then go for call by value otherwise call by reference

Call by Reference :-

```
main()
{ int a=10, b=20;
  Pf(a,b);
  Swap(&a,&b);
  Pf(a,b);
}
```

```
Swap(int *c, int *d)
{ int t;
  t = *c
  *c = *d;
  *d = t;
}
```

Compiler Put * itself

Compiler Put & itself

O/P :- ~~100~~ 10, 20
20, 10

(Red color : Compiles supporting call by reference) [implicit work]

*i) Originals will effect

*ii) more Parameters ~~then~~ passing use call by reference [like Array].

Que :-

```
main()
{ int a=10, b=20;
  Pf(a,b);
  f0(a,b);
  Pf(a,b);
}
f1(int e, int f)
{ *e = *e - 1
  *f = *f + *e;
  f2(f,e);
  f = f+10;
}
```

```
f(int c, int d)
{ *c = *c + *d;
  *d = *c + 10;
  f1(d,c)
  c = c+d;
}
```

```
f2(int g, int h)
{ *h = *h - 5
  g = g + h;
}
```

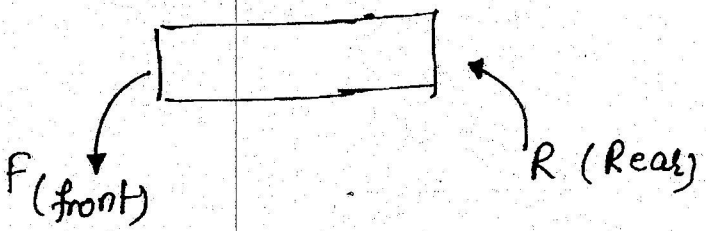
O/P :- 10, 20

Call by value = 10, 20
10, 20
] No change

Call by reference :- 147, 34

"Queue" :-

Def :- One side insertion, another side deletion

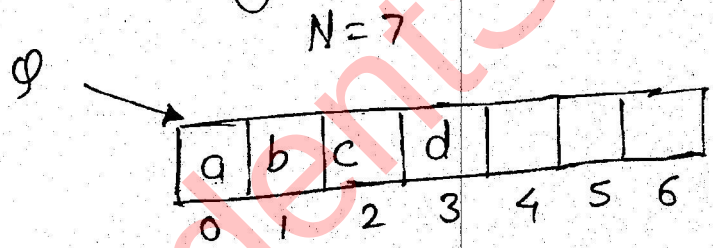


: FIFO
: Front - deletion
: Rear - Insertion

ADT of Queue :-

i) Enqueue ()
ii) Dequeue ()

Implementing Queue using Array :-



F = 0
R = 3

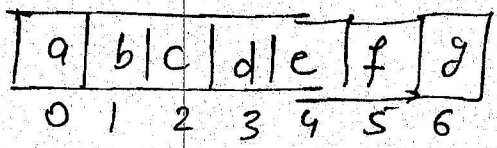
Queue Empty ⇒ F > Both -1
R

F = -1
R = -1

F = 0
R = 0
one element

F = -1
R = 3
Invalid

R 23
R = -1
Invalid



$F = -1$, $R = -1$ $\xrightarrow{a}$ $F = 0$, $R = 0$ $\xrightarrow{b}$ $F = 0$, $R = 1$ $\xrightarrow{c}$ $F = 0$, $R = 2$

1st insertion Both Front & Rear will come then after that Rear only.

$F = 0$, $R = 6$ $\xrightarrow{a}$ $F = 0$, $R = 5$ ---- !

Queue full :- $R = n-1$ or $R+1 = n$

EnQueue :-

```
EnQueue (x)
{
  If ( Rear == N-1)
  {
    Printf ("Queue is full");
  else
    Rear = Rear + 1;
    q[Rear] = x;
  }
  exit (1);
}
else
{
  If (R == -1)
    F = R = 0   ⇒ 1st insertion Both increament
  else
    R = R+1
  Q[R] = x
}
```

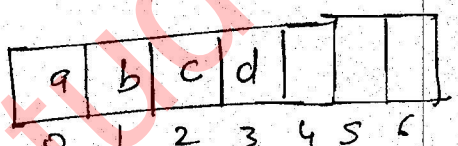
⇒ O(1) [every case]

DE-Queue :-

```
int De-Queue ()
{ int y;
  If ( F == -1)
  {
    Pf ("Queue is Empty");
                (underflow)
    exit (1);
  }
  else
  {
    If (F == R)
      return Q[F];
      F = R = -1
    else
      return Q[F];
      F = F+1;
  }
}
```

```
else
{
  If (F==R)
  { F = R = -1;
  else
  { F = F+1;
  return (y);
}
```

=> O(1)
Every case;



F = 0 Del => F = 1 Del => F = 2 Del => F = 3 Del => F = 4 Del => F = -1
R = 4 R = 4 R = 4 R = 4 R = 4 R = -1

F = 4, R = 4 : equal

Both out of Queue

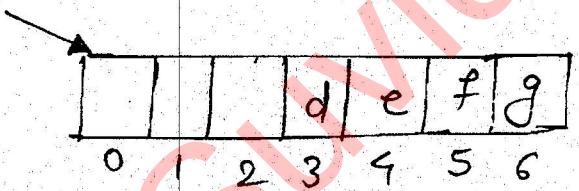
★ Using Above Program, we can implement Queue but it is not efficient.
(Linear Queue)

because bytes Real reaching Right-most place, it can not go back even though lot of space available.

★ii) To implement queue Efficiently, we are going to circular Queue.

★ Circular Queue :-

N = 7



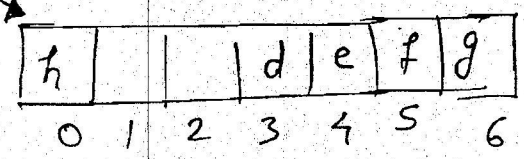
F = 3
R = 6

h → enque

Queue is full even though space available

Circular Queue :-

$$R = (R+1) \bmod N$$



F = 3
R = 6

h → enqueue

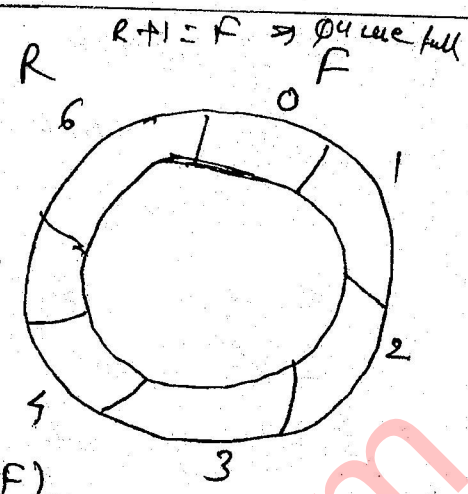
F = 3
R = 0

$R = (R+1) \bmod 7$

$\Leftarrow = 6+1 \bmod 7 = 0$

Circular Queue

full ⇒ $R+1 = F$

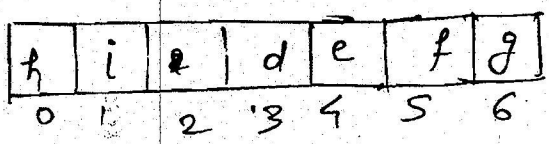


-EnQueue (x)

```
{
If ((R+1) mod N == F)
  {
    Pf ("Queue is full");
    exit (1);
  }
else
  {
    If ( R == -1)
        F = R = 0
    else
        R = (R+1) mod N
    Q[R] = x;
  }
}
```

⇒ O(1) Every Case

Circular Dequeue:-



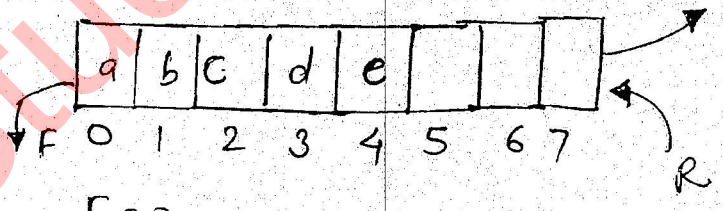
F = 3, R = 2 → Del ⇒ F = 4, R = 2 → Del ⇒ F = 5, R = 2 ⇒ F = 6, R = 2 ⇒ F = 0, R = 2

del ⇒ F = 1, R = 2 → F = 2, R = 2 (1 element) → del ⇒ F = -1, R = -1

```
int C- Dequeue ()
{ int y;
  If (F==-1)
  { Pf ("Queue is Empty");
    exit(1);
  }
  else
  { y = Q[F]
    If (F==R)
      F=R=-1
    else
      F = (F+1) mod N
    return (y);
  }
}
```

⇒ O(1) · Every case

Input- Restricted Queue :-



F=0
R=4

Insertion → one side only [Rear side]

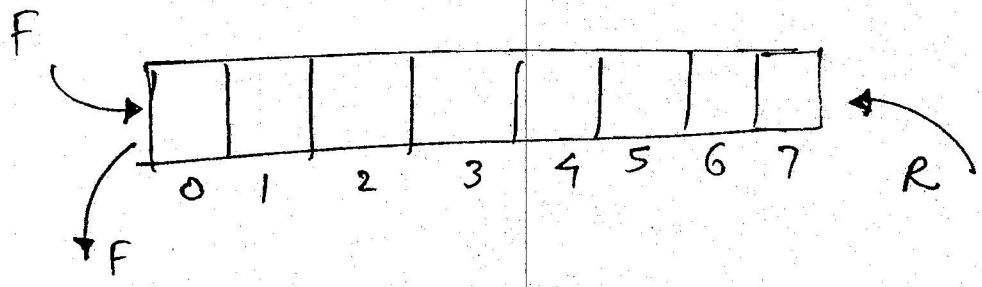
Deletion → Both side [F & R both]

Dequeue
F++ R--

EnQueue
R++

∴ FIFO Not satisfy

O/p Restricted Queue :-



O/p me side] Dequeue

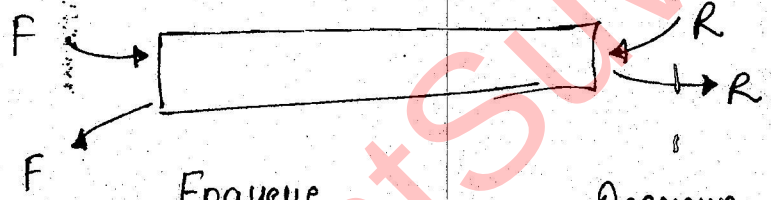
F = F+1

Enqueue [insertion Both Ends]

F-- R++

∴ FIFO (Queue) property not satisfy.

Double Ended Queue :- [Dequeue] → Data structure



Enqueue

F-- R++

Dequeue

F++ R--

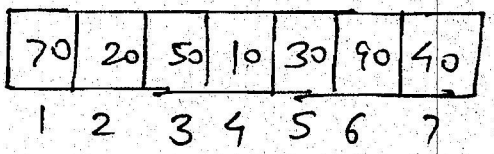
FIFO property not satisfy.

Priority Queue :-

Ascending Priority Queue

Descending Priority Queue

i/p : 70, 20, 50, 10, 30, 90, 40

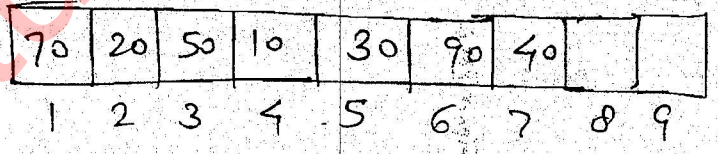


Dequeue :- 10, 20, 30 In ascending order

* Implementing Ascending Priority Queue :-

(i) "using unsorted Array" :-

i/p: 70, 20, 50, 10, 30, 90, 40
1 2 3 4 5 6 7



i=1
J = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~

LEQ ⇒ O(1)
J = J+1
Q[J] = x

Dequeue ① Find place of min element ⇒ O(n) [selection sort + one pass]
② replace by last or 1st element ⇒ O(1)

(ii) using sorted array :-

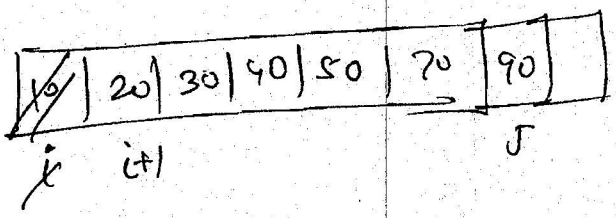
1 - Eφ ⇒ O(n)

1 - Dφ ⇒ O(1)

i = i+1

overall ⇒ O(n)

[As elements will be in ascending order. Shortest one at 1st place]



(iii) min Heap :-

1 - Eφ = O(logn)

1 - Dφ = O(logn)

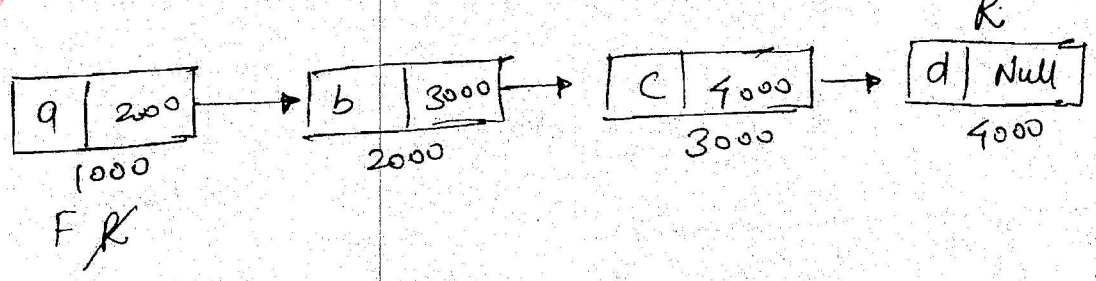
O(logn)

Best Data structure to implement Priority queue is [min Heap].

Implementing Queue using Link-List :-

:- When ~~Both~~ ~~Node~~ No Node is there Both Front & Rear are Null.

:- If one Node, then Front & Rear Both point same Node.



```
Enqueue ()
{
    If (P == Null)
    Pf ("memory full");
else
    P = malloc()
    R → next = P;
    R = P;
}
```

⇒ O(1)

Dequeue :

```
P = F
F = F → next
free (P)
P = Null
```

⇒ O(1)