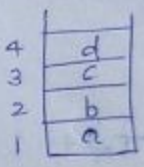
STACK

Defination :- One side is open another side closed

So Last in first out (LIFO)

Top is a variable which contain the position of the top most element in its stack

newly inserted element
or
element to be deleted



ADT of Stack :-

⇒ ADT means what operations perform on data

Push operation :-

⇒ push (stack, element) ≡ stack

pop op :-

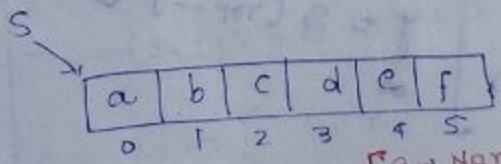
⇒ pop (stack) ≡ element

⇒ Isempty (stack) ≡ Boolean

⇒ Isfull (stack) ≡ Boolean

* Implementation of Stack :- (using Array) :->

⇒



[S → Name of array, N - Size of array
TOP → variable which contain position
x → element we want to

Push (S, N, TOP, x)

```
Push ( S, N, Top, x)
{
 if (TOP == N-1)
   {
   pf ("stack is overflow");
   exit (1);
   }
 else
   {
     Top++;
     S[TOP] = x
   }
}
```

exit (1) → Some problem than goes o of progra

exit (0) ⇒ successfull than goes o mean no pro

$\left.\begin{array}{l} \rightarrow S[++TOP] = x; \checkmark \\ \rightarrow S[TOP++] = x; \not\propto \text{ (wrong)} \end{array}\right]$

⇒ implementing stack using array will take O(1) time for inserting one element [WC, BC, AC]

```
Pop ( S, N, TOP)
{ int y;
  if ( TOP == -1)
    {pf ( "stack is underflow");
    exit (1);
    }
  else
    {
      y = S[TOP];
      TOP--
      return (y);
    }
}
```

$\left.\begin{array}{l} y = S[TOP]; \\ TOP-- \\ return(y); \end{array}\right] \quad y = S[TOP--] \checkmark$

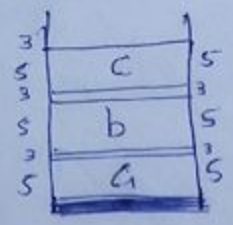
⇒ deletion will take O(1) for stack using array

(pop) $\begin{pmatrix} BC \\ WC \\ AC \end{pmatrix}$

```
Isfull ( S, N, TOP)
{
if (TOP == N-1)
{
pf ("stack is full");
return 1.
else
{
return 0;
}
```

Q. Let S be a stack of size $N \geq 1$. Starting with empty stack we push the first N natural number in the sequence and then perform N pop opⁿ. assume that push & pop opⁿ will take x sec each and y sec elapsed b/w end of one stack ~~btw~~ opⁿ and start of next opⁿ. ~~For m~~ stack life of an element m is defined as the time elapsed from the ~~end~~ end of push(m) to the start of pop opⁿ then what will be the average ~~of~~ stack life of an element in the stack?

Let $n = 3,\ x = 5,\ y = 3$



c - 3 ——— 3

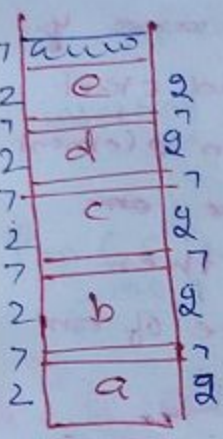
b - 3,5 3,5 3 — 19

a - 3,5 3,5 3,5 3,5 3 35

Total = 57

Avg = $\frac{57}{3} = 19$ Ans.

Let $n = 5,\ x = 2,\ y = 7$



e - 7 ——— 3

d - 2,7 2,7 7 — 19

c - 2,7 3,5 3,5 3,5 3 - 35

e - 7 ——— 7 7

d - 7,2 7,2 7 25

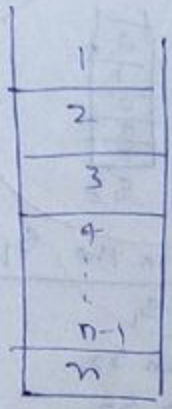
c - 7,2 7,2 7,2 7,2 7 43

b - 7,2 7,2 7,2 7,2 7,2 7,2 7 63

a - 7,2 7,2 7,2 7,2 7,2 7,2 7,2 7,2 7 215

Avg = $\frac{215}{5}$

Avg = 43



1 — y

2 — 2(x+y) + y

3 — 2x2 (x+y) + y

4 — 2x3 (x+y) + y

n — 2x(n-1) (x+y) + y

Total = $ny + 2(x+y)[1+2+3+\cdots+n-1]$

$= ny + 2(x+y)\left[\frac{(n-1+1)(n-1)}{2}\right]$

Total = $ny + (x+y)(n)(n-1)$

$Avg = \frac{ny + (x+y)\ n(n-1)}{n}$

$= y + (x+y)(n-1)$

$= y + nx + ny - x - y$

$\boxed{Avg = n(x+y) - x}$

Q. Write a C- programe to implement Queue using stack.

⇓ Enqueue dequeue

⇓ push pop.

Implementing Queue using stack :→

```
EQ (S1, x)
{
push (S1, x)
}
```

O(1)

```
DQ ( S1, S2)
{
if (S2 is empty)
  {
    if (S1 is empty)
      {
        printf (Queue is empty);
      }
    else
      {
        while ( S1 is not empty)
          {
            x = pop (S1)
            push (S2, x)
          }
        return ( pop(S2))
      }
  }
else
  return ( pop (S2))
}
```

O (n) W.C

O (1) [BC, AC]

Q implement Stack using Queue.

```
push (Q1, Q2, x)
{
if (Q1 is empty)
    EQ (Q2, x)
else
    EQ (Q1, x)
}
```

O(1)

```
pop (Q1, Q2)
{
if (Q1 is empty)
    if (Q2 is empty)
        {
        pf ("stack is underflow")
        exit (1);
        }
    else
        while (Q2 is not containing 1-el)
        {
        x = DQ (Q2)
        EQ (Q1, x)
        }
    return (DQ (Q2))
else
    while (Q1 not containing 1-ele)
        {
        x = DQ (Q1)
        EQ (Q2, x)
        }
    return (DQ (Q1))
}
```

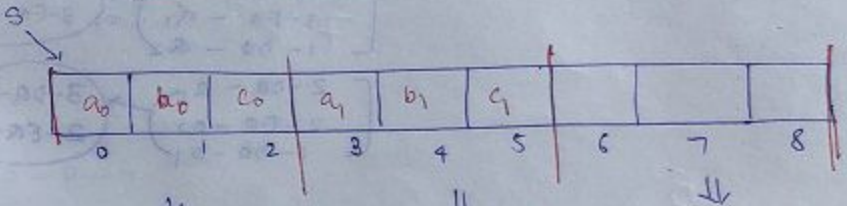
O(n)

Q. write a C- prog to implement multiple stack using single array.

(Size of array) $N = 9$

(No. of / size of stack) $M = 3$

size of every stack $= \frac{N}{M} = \frac{9}{3} = 3$



Top of stack

$T_0 \equiv -1 \not\equiv 1 \not\equiv 2$

$T_1 \equiv 2 \not\equiv 3 \not\equiv 5$

$T_2 \equiv 5$

initially top of stack

$T_0 = 0 * \left(\frac{9}{3}\right) - 1 = -1$

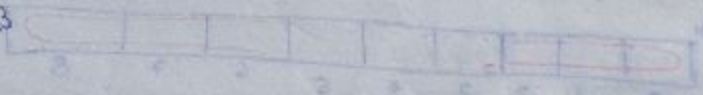
$T_1 = 1 * \left(\frac{9}{3}\right) - 1 = 2$

$T_2 = 2 * \left(\frac{9}{3}\right) - 1 = 5$

$T_{100} = (100) * \left(\frac{9}{3}\right) - 1 = 299$

AA $\Rightarrow$ $T_i = (i) * \left(\frac{N}{M}\right) - 1$

```
push (S, N, M, Ti, x)                          (i+1)
{
    if ( Ti == (i * N/M - 1) )
    {
        pf ("stack is overflow");
        exit (1);                                  ⇒ O(1)
    }
    else
    {
        Ti++;
        S[Ti] = x;
    }
}
```



```
int pop (S, N, M, Ti)
{
    if ( Ti == i * N/M - 1 )
    {
        pf ("stack is underflow");
        exit (1);                                  ⇒ O(1)
    }
    else
    {
        y = S[Ti];
        Ti--;
        return (y);
    }
}
```

Application of Stack :-

1. Recursion
   - (i) Tail - Recursion
   - (ii) Non-Tail - Recursion
   - (iii) Indirect - recursion
   - (iv) Nested - recursion
2. Infix to postfix
   prefix to postfix
   postfix Evaluation
3. Tower of Hanoi
4. fibinonci series

1 Recursion :-

⇒ Write a erecursive c-prog to print a given array of n-element.

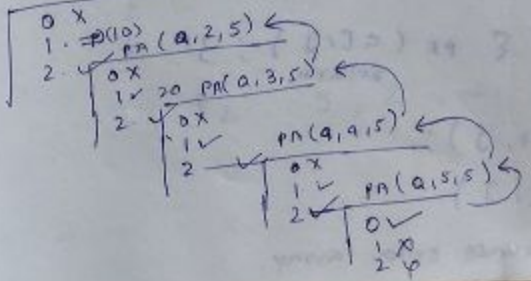
Sol

```
PA ( a, i, j )
{
0. if (i == j) { pf (a[i]) }; return
   else
   {
1.   pf (a[i]);
2.   PA ( a, i+1, j );
   }
```

Lets

10	20	30	40	50
1	2	3	4	5

PA(a, i, j)



i) Tail-recursion :- In the given recursive programme if only 1 function call is there that function call also at end of the programme than that recursive programme is called tail-recursive programme.

- The drawback with tail-recursive programme we are wasting lots of stack space unnecessarily.
- The advantage of tail-recursive programme we can write equivalent non-recursive programme very easyly with the help of loop.

[In last prog ⇒ [for (i=1 to n) pf (a[i])]]

ii) NON-Tail-Recursive Prog :→ In the given recursive programme after the function call there is some work to do then that is recursive prog called non-recursive programme.

We are not wasting stack space unnecessarily.

The drawback to non-tail RP that is very difficult to write equivalent NONRP with the help of loop.

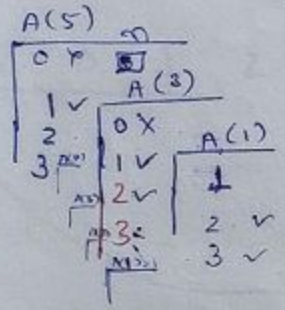
ex

```
1.  PN (a, i, j)
    {
    if (i==j) { pf (a[i]); return; }
    else
    PN (a, i+1, j);
2.  pf (a[i])
```

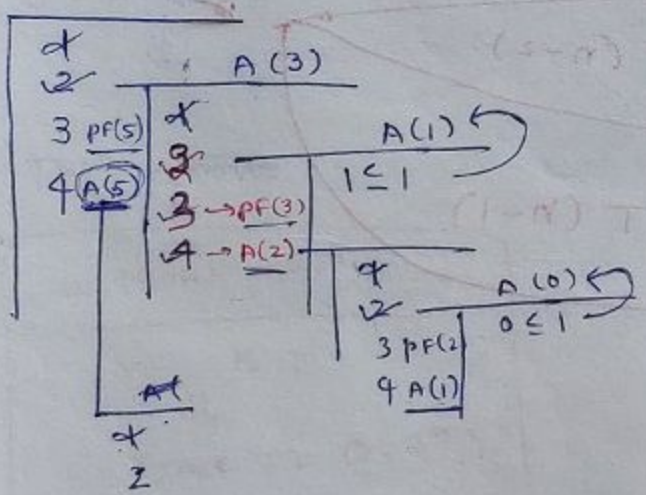
ex 2

```
A(n)
{
1 0 if (n<=1) return 1
  else
  {
2 1 A(n-2)
3 2 pf(n);
4 3   A(n-1);
  3
  3
  i/p : A(5)
  o/p : -
```

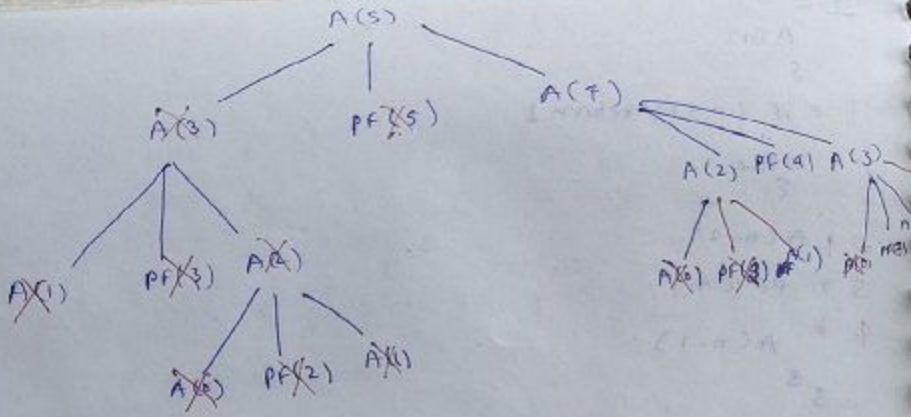
1, 3, 5



A(5)



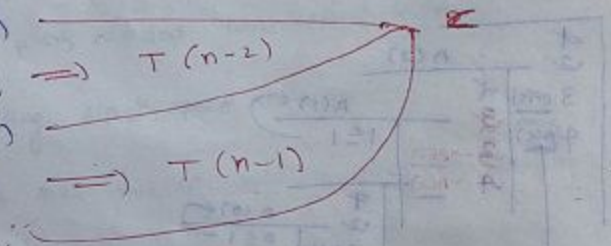
3, 2, 5, 2, 2, 4, 3, 2



3. 2 5 2 4 3 2

ex

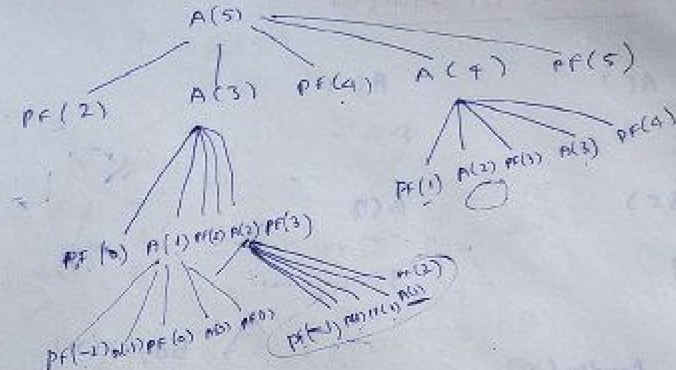
```
A(n)
{
if (n=0) return;
else
{
PF (n-3)
A(n-2);
PF (n-1)
A(n-1)
PF (n);
}
}
```



i/p :- A(5)

o/p :- 2 0 -2 0 1 2 -1 1 -2 0 1 2 3 0 1
-1 1 -2 0 1 2 3 0 -2 0 1 2 -1 1 -2 0 1 2 3 0 1

The above recursive prog will generate n-length CB
So time complexity = $O(2^n)$.



0 -2 0 1 2 -1 1 2 3 4 1 -1 1 -2 0 1 2 3

0 -2 0 1 2 -1 1 -2 0 1 2 3 4 5

The ~~above~~

without DP

n- CBT

⇓

Time ⇒ $O(3^n)$

Space = $O(n)$

with DP

⇓

Time ⇒ $O(n)$
[D-C]

Space ⇒ $O(n)$

② infix to postfix :→

Infix : a + b * c

In c - always left to right

postfix : a b c * +

prefix : + a * b c

Size of stack = 2

ex 2

Infix : a + b - c

[a bc- + → right to left associative postfix

or

postfix :- ab + c - (left to right associative)

prefix :- - + a b c (left to right associative)

~~a b c~~

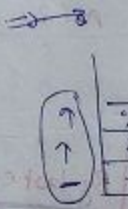
ab + c -

Size = 1

Note :- In all the above three forms operands order cannot be ~~ca~~ changed but operators order may changed.

ex 3

Infix :- a + b * c / d - e ↑ f ↑ g



postfix :- a b c * d / + e f g ↑ ↑ -

~~e f g ↑ ↑~~

Size = 3

~~Prefix :-~~

postfix :- a b c * d / + e f g ↑ ↑ -

prefix :→ - + a / * b c d ↑ e ↑ f g

Ex infix: b↑e - d↑a * c / g + f - h

postfix: be↑ da↑c * g / - f + h -

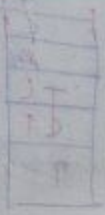
prefix: -

Ex b↑(e-d) ↑ (a * c / (g+f)) - h

postfix: b ed- ac* (gf+) / ↑ ↑ h -

prefix -↑b↑-ed /*ac +gf h

low priority → high priority	≡ push
Right → Left	≡ push (power)
High priority → low priority	≡ pop
Left → Right	≡ pop



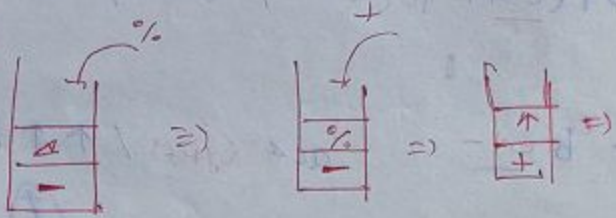
ex:- a - f * g ↑ d ↑ ↑ 4 - b c * h % + i ↑ j

Sol

e f g d ↑ ↑ 4 - b c * h % + i j ↑ +

postfix

max size of stack = 4 - operators.

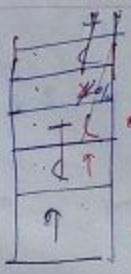


Note:→ While converting infix to postfix we are using operator stack. Only operators are pushed into the stack.

4 Infix to postfix conversion will take O(n) time.

(I scan from left to right and for every symbol print push or pop op only)

ex: b ↑ (a - d) ↑ (a * c / (g + f)) - h



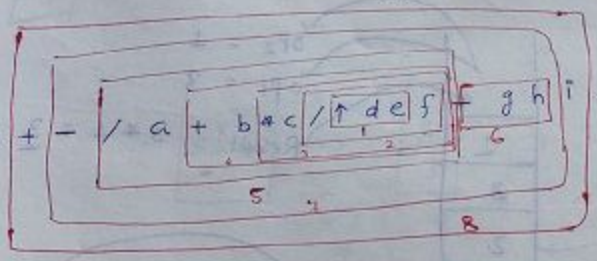
b e d - a c * g f + % ↑ ↑ h -

max size of stack = 6.

4 prefix to postfix →

prefix	postfix
1 a	a
2 +ab	ab+
3 *a+b-cd	abcd-+*
4 *+-/a+b*c/↑def-gh-ij	abcde↑f%*+%gh--i+j*

abcde↑f%*+%gh--i+j*



abcde↑f%*+%gh--i+j*

=> Lisp programming language follow prefix form.

Note:- To evaluate to the prefix & infix expression many scans are required but to evaluate the postfix notation one scans are enough. because of this reason we convert prefix to postfix or infix to postfix always.

* conversion takes O(n) & evaluation takes O(n)

so total = O(n) + O(n)

Time Com. ⇒ O(n)

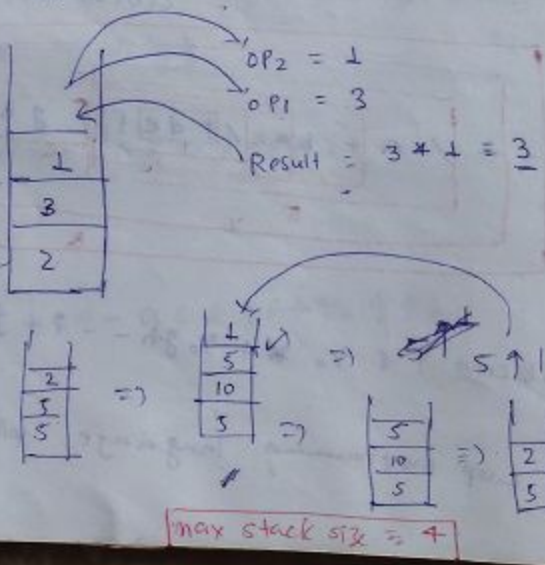
* postfix evaluation :-

infix :- (2 + (3 * 1)) + ((5 ^ 2) / 5 ↑ 1) + 5

⇓ O(n)

postfix :- 2 3 1 * + 5 2 ^ 5 1 ↑ / + 5 +

evaluation :-



max stack size = 4

Note :- To evaluate the given postfix expression we are using Operand stack. (only operand are pu into stack) -

& evaluation algo takes O(n) time (1 scan onl

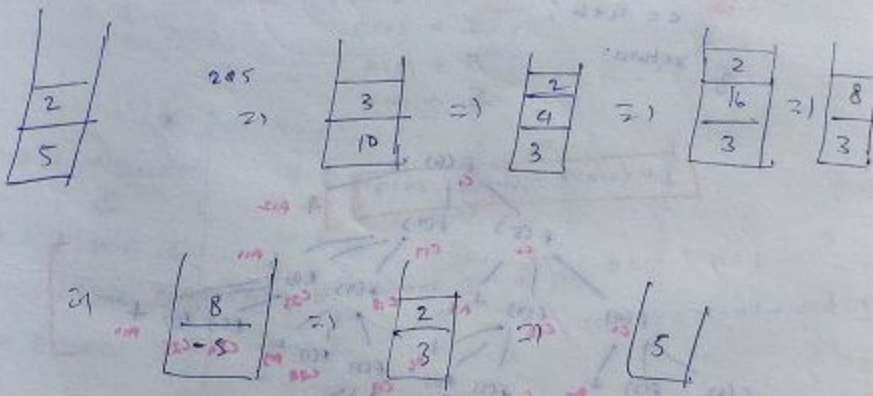
Q. infix :- $(5 \times 2 / 3) - (4 \uparrow 2) / 2 + 8 + 2$

$(10/3) - 16 \% 2 + 8 + 2$

$3 - 8 + 8 + 2 = 5$ ⇓ O(n)

postfix :- $52*3\%42\uparrow2\%-8+2+$

⇓ O(n)



max stack size = 3

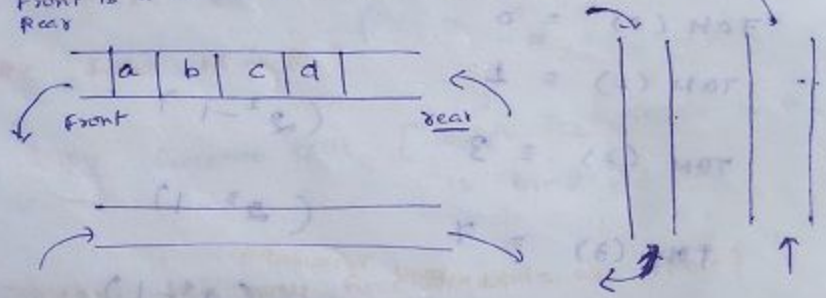
Time comp = O(n) + O(n) ⇒ O(n)

Queue :→

Def :→ One side Insertion other side deletion.

: FIFO

: Front is a variable which contain position of it element to be delete
Rear



⇒ Rear is a variable which contain position of the ne inserted element.

ADT of Queue :→

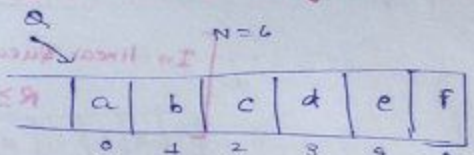
(i) Enqueue :- enqueue (Queue, element)

(ii) dequeue :- dequeue (Queue) = element

(iii) Isfull (Queue) = Boolean

(iv) IsEmpty (Queue) = Boolean

Implementing Queue using array ?



Q is empty F=-1 R=-1 ⇒ en=a F=0 R=0 ⇒ en=b F=0 R=1 ⇒ en=c F=0 R=2 ⇒ en=d F=0 R=3 ⇒ F=0 R=4 ⇒ F=0 R=5

R=N-1 Queue is full

D: F=1 R=5 ⇐ D: F=2 R=5 ⇐ D: F=3 R=5 ⇐ D: F=4 R=5 ⇐ D: F=5 R=5

F=R=-1 Q is only 1 element ⇐ D: F=-1 R=-1

Enqueue (Q, N, F, R, x)

```
{
  if (R == N-1)
  {
    pf ("Q is Overflow");
    exit (1);   Abnormal Termination
  }
  else
  {
    if (R == -1)
      F = R = 0;
    else
      R = R+1
    Q[R] = x;
  }
}
```

O(1) because ... comparison takes ... and insert also take ...

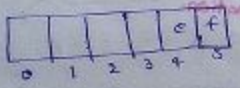
Dequeue (Q, N, F, R)

In linear Queue always R ≥ F

```
{
if ( F == -1)
{
Pf (" Queue is empty");
exit (1);
}
else
{ y = Q[F]
if ( F == R)
{
F = -1;
R = -1;
}
else
F = F + 1;
dequeue(
return (y);
}
}
```

O(1)

Circular Queue :→



F = 4

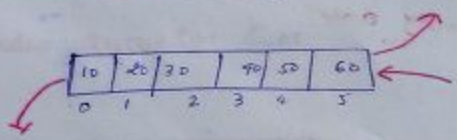
R (=5)

In the above linear Queue even though the space is available we cannot insert one more element further because R reached right most place and he can't go back. because of this reason we go circular Queue where R can go back also.

```
Circular_Queue (Q, N, F, R, x)
{
if (R+1 mod N == F)   (change)
{
if ("C-Q - Overflow)
   exit(1);
}
else
{
  if (R == -1)
     F = R = 0;
  else
  {
     R = R + 1 mod N   (change)
     Q[R] = x;
  }
}
```

```
C-Queue (Q, N, F, R)
{
  int y;
  if (F == -1)
  {
    pf ("C-Q is underflow");
    exit(1);
  }
  else
  {
    y = Q[F]
    if (F == R)
      F = R = -1
    else
      F = F + 1 mod N;
    return (y);
  }
}
```

Input Restricted Queue :→



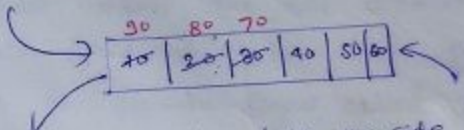
⇒ Insertion only one-side

⇒ deletion - Both the ~~side~~ side

(2-codes comes)

Not FIFO

Output Restricted Queue :→



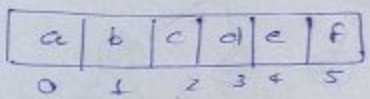
=) Insertion - both the side

deletion - One side (Normal Queue)

not FIFO

Double ended Queue :→ also called Deque

double



Note FIFO

Insertion — Both side

deletion — Both side

data structure → deque → double ended

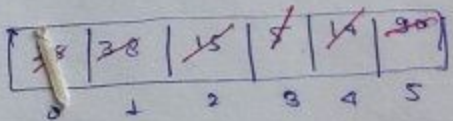
data operation → deque → deletion oper

Priority Queue :→

1. Ascending priority Queue
2. descending priority queue

1. Ascending priority Queue :→

i/p : 25, 38, 15, 5, 14, 90



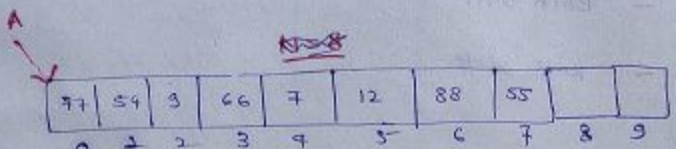
EQ → all operation done.

DQ ⇒ 5 D⇒ 14 D⇒ 15 D⇒ 25 D⇒ 38 D⇒ 90

Implementing priority Queue (ascending priority queue) :→

1 Using Unsorted array :→

i/p :- 77, 54, 9, 66, 7, 12, 88, 55



1 - EQ ⇒ O(1)

1 - DQ ⇒ Find min element place. ⇒ O(n) ⇒ delete and
Replace by last element ⇒ O(1)

total O(n) + O(1) = O(n)

2. Using Sorted Array :→

i/p :- 77, 54, 9, 66, 7, 12, 88, 55

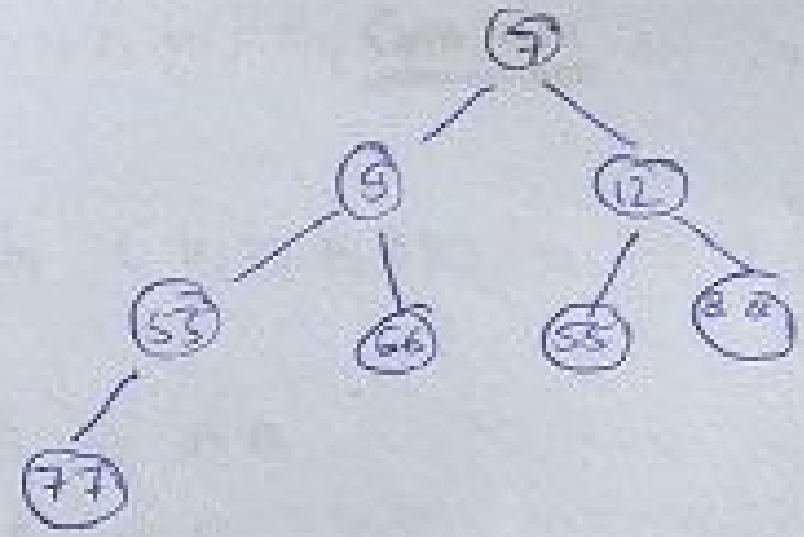
1 - EQ ⇒ O(n) [Insertion in correct place]

1 - DQ ⇒ O(1)

total O(n)

3. using Heap tree :→

i/p :- 77, 54, 9, 66, 7, 12, 88, 55



best case O(1)

I - E ⊙ ⇒ O(log n) (worst case)

I - D ⊙ ⇒ O(log n)

---

total O(log n)