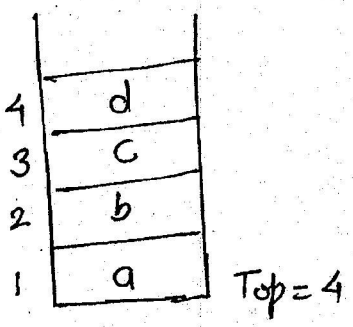
Date:- 13/10/2017

"Stack" :-

Def: One side open, Another side close.



Top of stack :- Position of top-most element

* LIFO (or) FILO

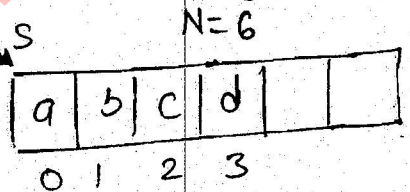
- ADT of Stack :- (Abstract of Data Structure)

ADT :- what operation, we can perform on that Data structure.

ADT of stack :- Push()
Pop()

"Implementing Stack" :-

implementing stack using Array :-




```
Push (x)
{
    If (Top == N-1)
        Pf ("overflow");
    else
    { Top ++;
      S[Top] = x;
    }
}
```

⇒ `S[Top++] = x` X
`S[++Top] = x` ✓

```
Void Push (int x)          /* Array & Top are
{                             global variables,
                              accessed by both
  If (Top == N-1)             Push & Pop */
  {
    Printf ("Stack is overflow");
    exit (1); /* exit(1) abnormal termination
  }                  exit(0) normal termination */
else
  {
    Top++;
    S[Top] = x;
  }
}
```

Time :- O(1) [Every case].

Pop() :-

```
If (Top == -1)
  Pf ("underflow");
else
{ y = S[Top]; /* first copy the data
Top--;          to a variable as
return (y);     we have to return it */
}
```

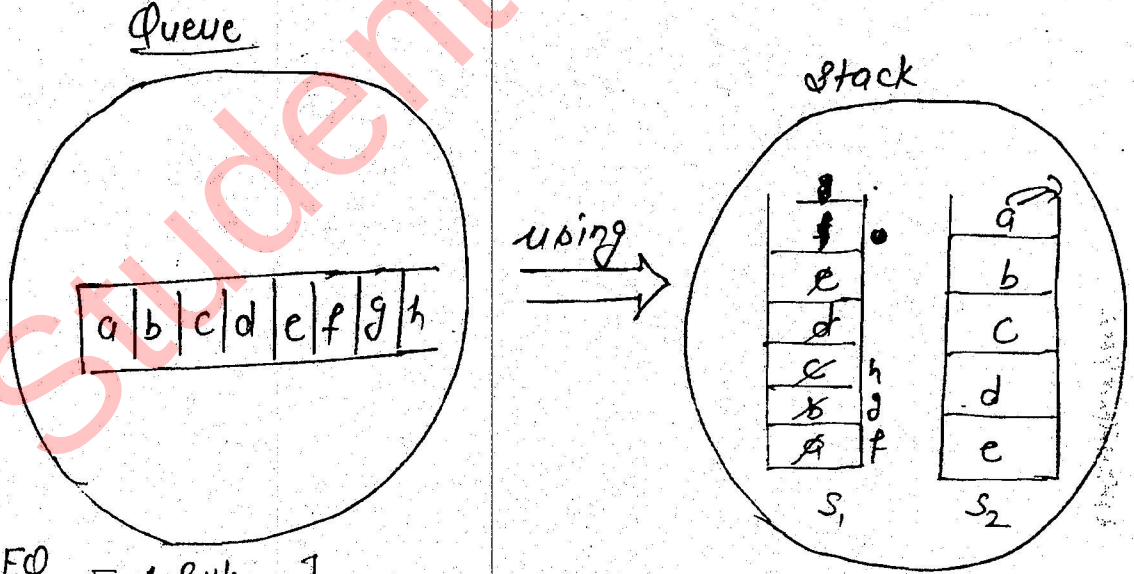
```
int Pop( )                    /* always pop top element
                                 so no parameter */
{
  If ( Top == -1)
  {
   Printf(" Stack is underflow");
    exit(1);
  }
else
  { int y;
    y = S[Top];
    Top--;
    return (y);
  }
}
```

Time: O(1) [every case]

* Implementing Queue using Stack :-

Queue ⇩ : Enqueue, Dequeue

Stack ⇩ : Push, Pop



1 - EQ = 1 Push
3 - EQ = 3 Push
1 - EQ = 1 Push

a, b, c, d, e Pushed (enqueued)

1 - DQ = 5 - Pop - S_1
= 5 - Push S_2 e, d, c, b, a
= 1 - Pop - S_2 'a'

1 - Dφ = 1 - Pop - S_2 'b'

Stack S_1 having the reverse values, so for Pop operation we need another Stack.

★ [2 Stacks required for implementing Queue using Stack].

So, Pop operation always performed on Stack S_2.. and Push always performed on Stack S_1.

1 - Dφ = 1 - Pop - S_2 'c'

1 - Eφ = 1 - Push - S_1 'f'

2 - Eφ = 2 - Push - S_1 'g, h'

1 - Dφ = 1 - Pop - S_2 'd'

1 - Dφ = 1 - Pop - S_2 'e' [S_2 is empty now] so check S_1 and reverse again by Pushing S_1 elements to S_2].

1 - Dφ = 3 - Pop - S_1 f, g, h

= 3 - push - S_2 h, g, f

= 1 - Pop - S_2 f'

Eφ (x) [Enqueue operation]

```
{
    Push (S1, x);
}
```

```
int DQ (S1, S2)
{
    If (S2 not empty)
        return (Pop(S2));
    else
    {
        while (S1 not empty)
        {
            x = Pop(S1);
            Push (S2, x);
        }
        return (Pop(S2));
    }
}
```

⇒ O(n) worst case

Time :-

Best case = O(1)

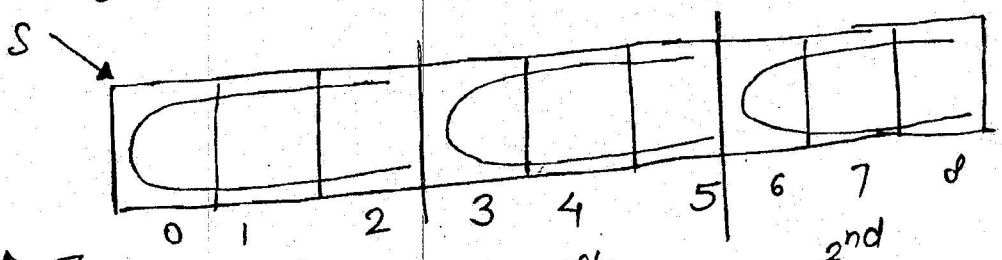
Avg case = O(1)

worst case = O(n)

Que :- implement Stack using Queue :-

Que :- Implementing multiple stacks in a single Array :-

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



Top of stack's → $T_0 = -1$ 0th $T_1 = 2$ 1st $T_2 = 5$ 2nd initially.

Push

0th stack	1st stack
T_0++	T_1++
$S[T_0] = x$	$S[T_1] = x$

Size of Every Stack $= \frac{N}{m}$ [elements in Array] / [elements in stack]

0th Stack :- Initial Top of stack

~~0 * 3~~ $0 * 3 - 1 = -1$

1st stack :- initial Top of stack

$1 * 3 - 1 = 2$

2nd stack :- initial Top of stack

$2 * 3 - 1 = 5$

100th stack - initial Top of stack

$100 \times 3 - 1 = 299$

ith stack :- initial Top of stack

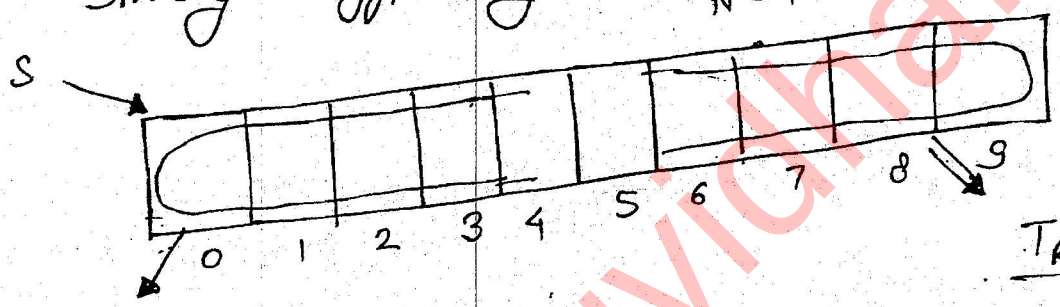
* $i \times \frac{N}{m} - 1 \Rightarrow T_i$

```
void Push (x, i)                    ← Next stack initial Top of stack
{
  if ( Ti == ((i+1) N/m - 1) )
  {
    Pf (" Stack overflow");
    exit (1);
  }
  else
  {
    Ti ++;
    S[Ti] = x;
  }
}

int Pop (i)          /* Pop operation in ith stack so pass i (#stack No) as parameter */
{ int y;
  if (Ti == (i* N/m - 1)) {
  {
    Pf (" Stack is underflow");
    exit (1);
  }
  else
  {
    y = S[Ti];
    Ti --;
    return (y);
  }
}
```

⇒ Using above Program, we can impliment multiple Stack in a single array But It is not efficient because If one Stack is full and All other Stacks are empty, we are getting error message even though lot of space available.

* Implementing multiple stacks in a single array Efficiently :-



$T_L = -1$

Push

$T_L ++;$

$S[T_L] = x$

$T_R = 10$

Push

$T_R --;$

$S[T_R] = x;$

Conditions for full (Stack) T/F :

a) $T_L = T_R - 1$ ✓ ∵ T_L always be < T_R

b) $T_L == T_R = N/2$ ✗ [both may not grow half]

c) $T_L = T_{N-1}$ ✗ [T_L may not grow always]

d) $T_L == T_R + 1$ ✗ [can't be $T_R < T_L$].

:- Avg Life Time of an Element in to Stack :-

Ex:-① :- $n = 3$ (a, b, c) → 3 element

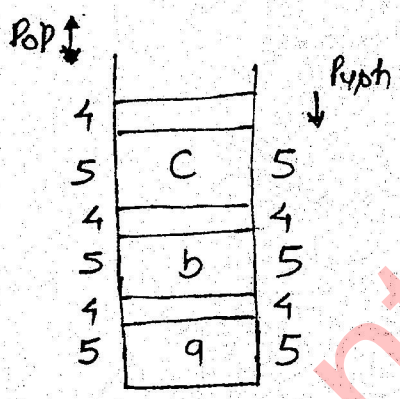
⇒ continuously push followed by continuous Pop.

Each Stack operation = x sec (5) ← given

Elapsed Time = n sec. (4)

Life time of an Element in Stack = After Push ○ Before Pop

in b/w



Life time of

c: = 4 ← {4}

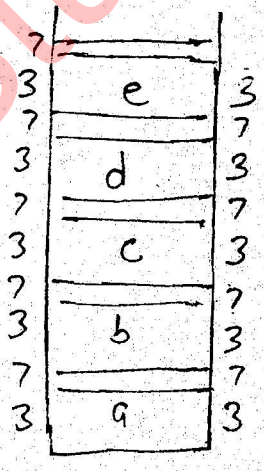
b: = 22 ← {4, 5, 4, 5, 4}

a: = 40 ← {4, 5, 4, 5, 4, 5, 4, 5, 4}

Total 66

Avg = $\frac{66}{3}$ = 22

Ex:-② :- $n = 5$ (a, b, c, d, e) $x = 3$, $y = 7$ ← elapsed time



Life time:

e = 7 {7}

d = {7, 3, 7, 3, 7} = 27

c = {7, 3, 7, 3, 7, 3, 7, 3, 7} = 47

b = {7, 3, 7, 3, 7, 3, 7, 3, 7, 3, 7} = 67

a = {7, 3, 7, 3, 7, 3, 7, 3, 7, 3, 7, 3, 7, 3, 7} = 87

235

Avg = $\frac{235}{5}$ = 47

~~$\frac{n(x+y)-x}{5x}$~~

formula :- $\boxed{n(x+y)-x}$ for Avg life time.

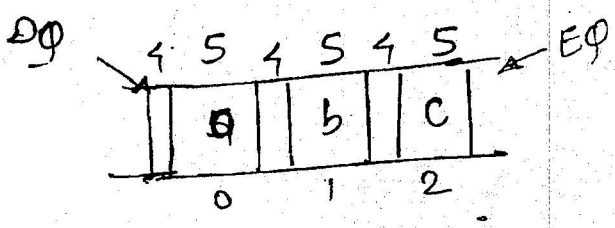
$5(7+3)-x$

$= 47$

- Avg Life time of an Element in "Queue" :-

Ex: ③ :- $n = 3(a,b,c)$ 3 element

$x = 5$, $y = 4$ ⇒ Continuous 3 Enqueue followed by Dequeue



$x = 5$ [Enqueue/Dequeue time]

$y = 5$ [Elapsed time]

Life time :-

EQ b EQ c DQ a

$a = \{4, 5, 4, 5, 4\} = 22$

EQ c DQ a

$b = \{5, 4, 4, 5, 4\} = 22$

$c = \{5, 4, 5, 4, 4, 5, 4, 5, 4\} = 40$

$= \{4, 5, 4, 5, 4\} = 24$

DQ a DQ b

* "Applications of Stack" :-

① Recursion
   - i) Tail Recursion
   - ii) Non Tail Recursion
   - iii) Nested Tail Recursion
   - iv) Indirect Tail Recursion.

② Infix to Postfix
   Prefix to Postfix.
   Postfix Evaluation

3. Towers of Hanoi

4. Fibinocci series.

"Recursion":-

Ex:- write a 'C' Program to Print Array of 'n' numbers.

A {10, 20, 30, 40, 50}
1 2 3 4 5

```
main ()
{
    Print_Array (a[], i, J);
}
Print_Array (a[], i, J)
{
    If (i==J)
        Pf (a[i]);
    else
    {
        Printf (a[i])
        Print_Array (a[], i+1, J);
    }
}
```

⇒ Tail Recursion :-

* i) In the Recursive program after the function call no work. [In the Recursive Program only one function call & after that No work] then it is called Tail Recursive Program.

Ex: Above Program.