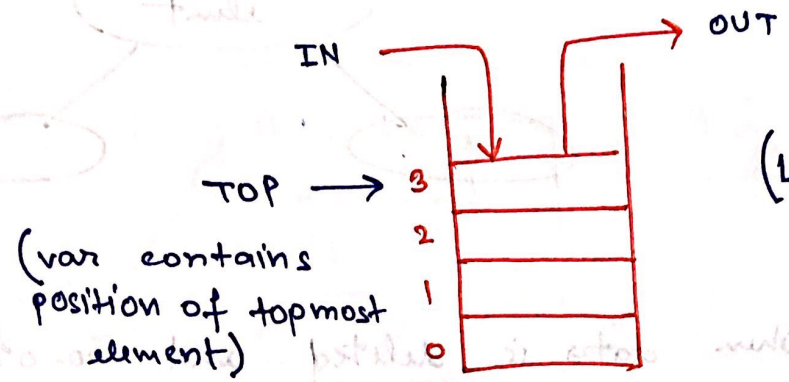
Stack :-

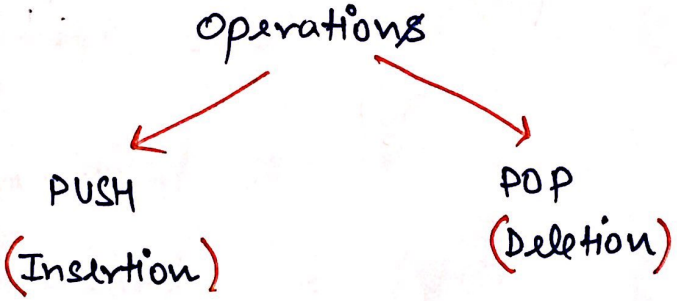
Oneside open & oneside closed :



(LIFO and FILO)

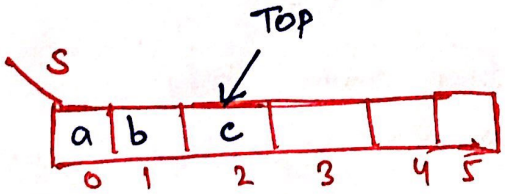
ADT of Stack :

Abstract data type of stack shows or represents operations which we can perform on STACK.



Implementation of Stack :-

(i) Using array :-



Top = -1 (Empty)

Top = N-1 (Full)

PUSH = ++Top

POP = Top--

- void PUSH(x)

```
{
    if( TOP == N-1)
    {
       printf("STACK is overflow");
       exit(1);
    }
    else
    {
       S[++TOP] = x;
    }
}
```

O(1)

- int POP( )

```
{
   if (TOP == -1)
   {
      printf("STACK is underflow");
      exit(1);
   }
   else
   {
     x=S[TOP--];
      return (s);
   }
}
```

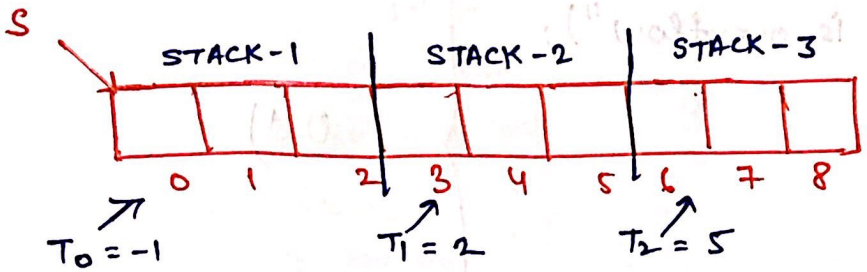
O(1)

NOTE:- Both PUSH & POP op^{n} takes constant time using array implementation.

(ii) Multiple stack using single array:-

N = 9 (size of array)

m = 3 (no. of stacks)



$N/m = 3$ (size of one STACK)

$N/m \in Z$ (uniform size)

$N/m \notin Z$ (non-uniform size)

(Top of ith stack intial value = $\left(\frac{N}{m}\right) i - 1$)

(Ti)

```
void PUSH(x, i)
{
   if (Ti == (i+1) N/m - 1):
   {
     printf("stack overflow");
     exit(1);
   }
   else
   {
      Ti++;
      S[Ti] = x;
   }
}
```

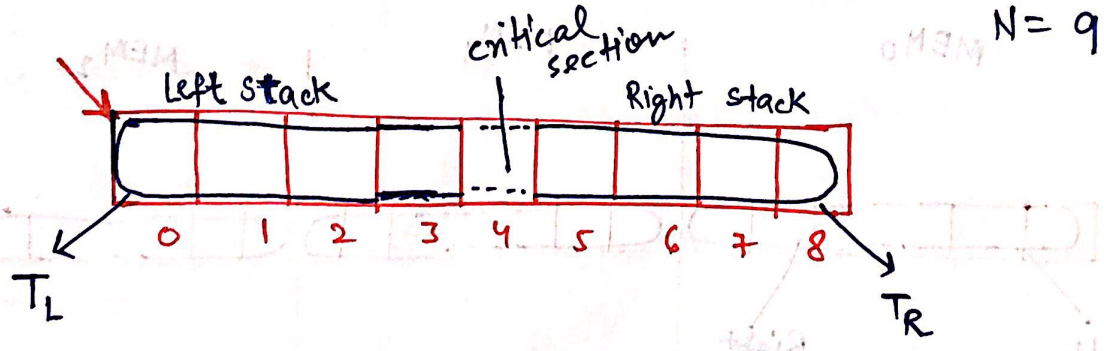
O(1)

```
int POP(i)
{
  int x;
  if( Ti == i(N/m) -1)
  {
    printf("STACK underflow");
    exit(1);
  }
  else
  {
    x = S[Ti--];
    return(x);
  }
}
```

O(1)

- Using this techniane we can implement multiple stack in single array but it is not efficient because if one stack is full and all other stacks are empty, it gives stack overflow as we can untilize space of other stacks

(iii) Multiple stack using single array efficiently :-



- Array divided into two stacks with no boundary

$STACK_L \xrightarrow{PUSH} ++T_L$

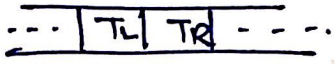
$STACK_R \xrightarrow{PUSH} --T_R$

- In critical section overwritting problem can be occurred, so we can use semaphores to prevent collision or overwritting.

Q: which of the following condition are true for Both stack full:

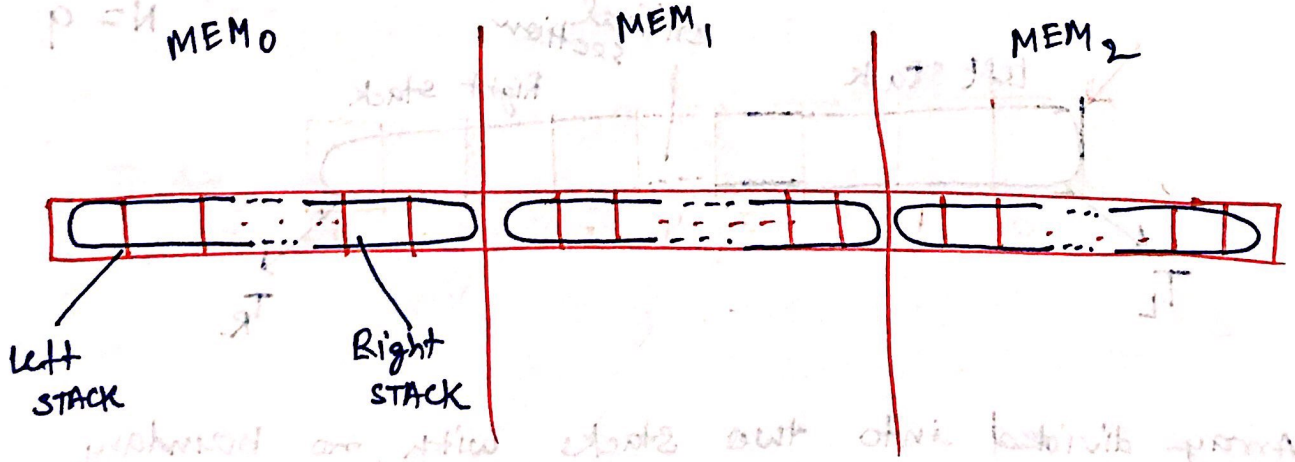
(a) $T_L == N-1$ ✗

(b) $T_R == 0$ ✗

(c) $T_L = T_R - 1$; ✓ 

(d) $T_L = T_R + 1$; ✗

- Since for better efficiency we divide memory ... of some optimum size and then each of these memory blocks contain two stacks opposite to each other.

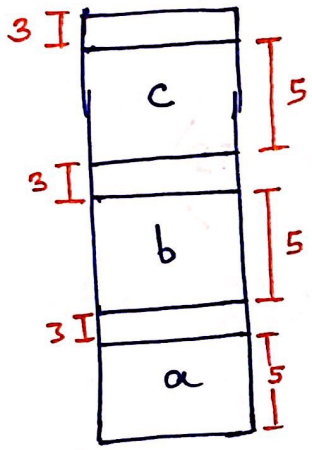


Avg. lifetime of an element in STACK:—

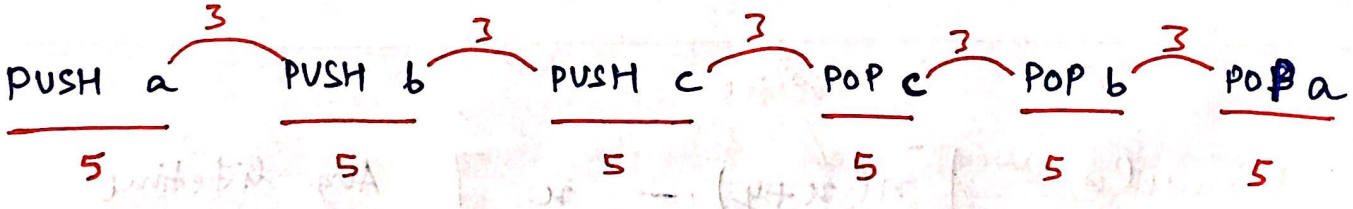
eg:- $n = 3 (a, b, c)$ ⟶ n PUSH oph continously
+
n POP oph continously

Assume each stack oph will take = x unit time
where $x = 5$.

B/w two stack oph (Elapsed time) = y unit time
$y = 3$



[Lifetime(a) = after PUSH(a) to before POP(a)]



Lifetime (c) = 3.

Lifetime (b) = 19

Lifetime (a) = 35

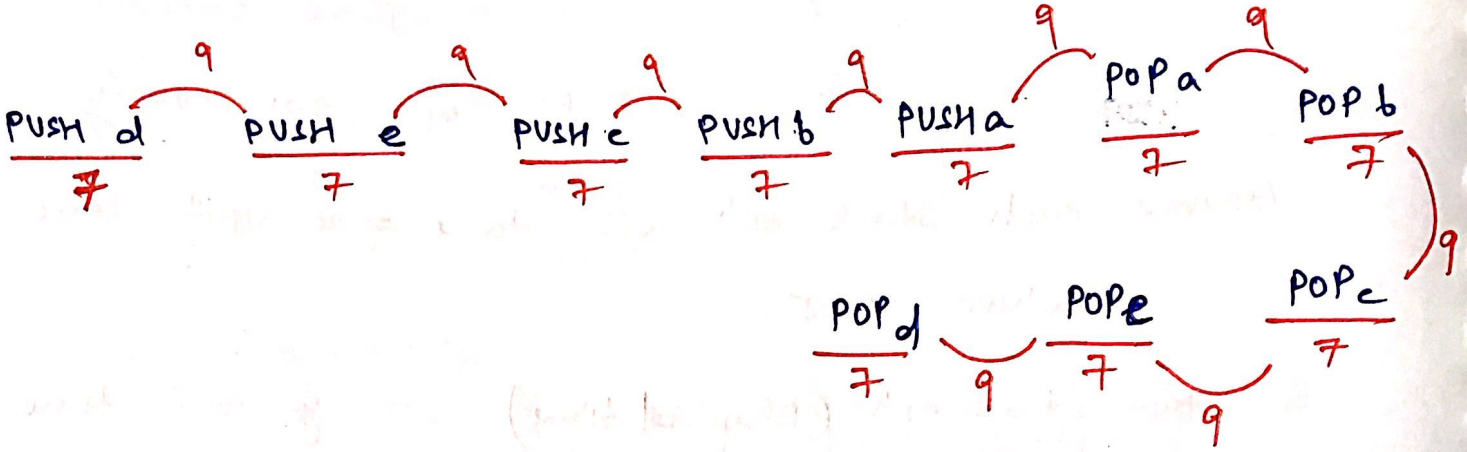
so, AVG lifetime = $\frac{3+19+35}{3}$

$= 19$

eg:- $n = 5(d, e, c, b, a)$; operation time = 7

Elapsed time = 9

Calculate Avg lifetime?



Lifetime (a) = 9

Lifetime (b) = 41

Lifetime (c) = 73

Lifetime (e) = 105

Lifetime (d) = 137

(Avg lifetime = 73)

Formula:- $\left[n(x+y) - x \right]$ Avg lifetime

x = op^h time

y = Elapsed time

x = no. of push or pop op^h

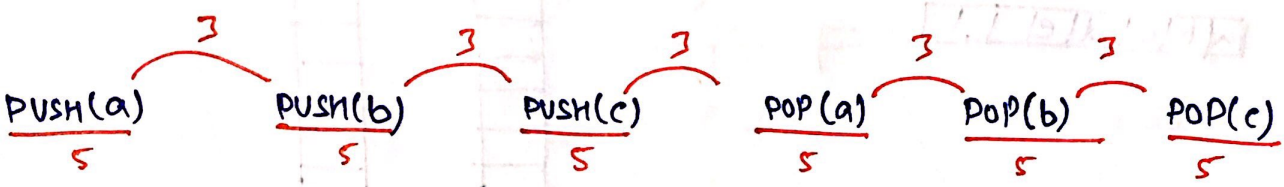
---

so; $5(7+9) - 7 = 80 - 7 = \underline{\underline{73}}$

eg: $n = 3(a,b,c)$ $n=5$ and $y=3$.

n - PUSH & n - POP oph on Queue.

Calculate avg lifetime of elments in the Queue.



Lifetime(a) = $2(5+3) + 3 = 19$

Lifetime(b) = $2(5+3) + 3 = 19$

Lifetime(c) = $2(5+3) + 3 = 19$

(Avg lifetime = 19)

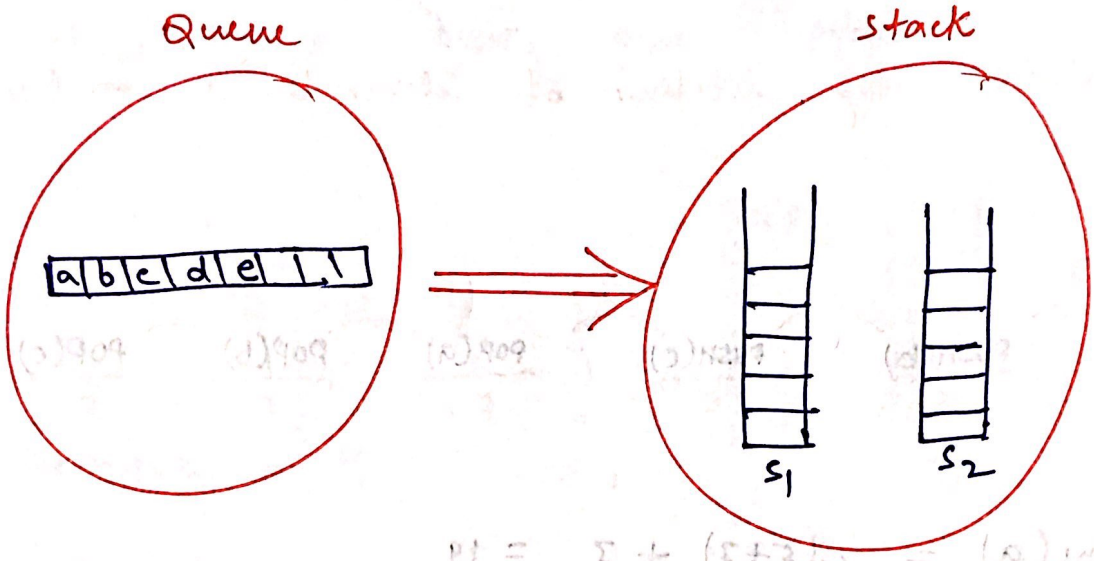
NOTE:- In Queue, each element has same lifetime, which is not in the case of stack.

Implementing Queue using Stack:-

Enqueue() ⟷ Push()

Dequeue() ⟷ Pop()

- to implement Queue using stack, we have to use only PUSH & POP op^n to perform Enqueue & Dequeue op^n.



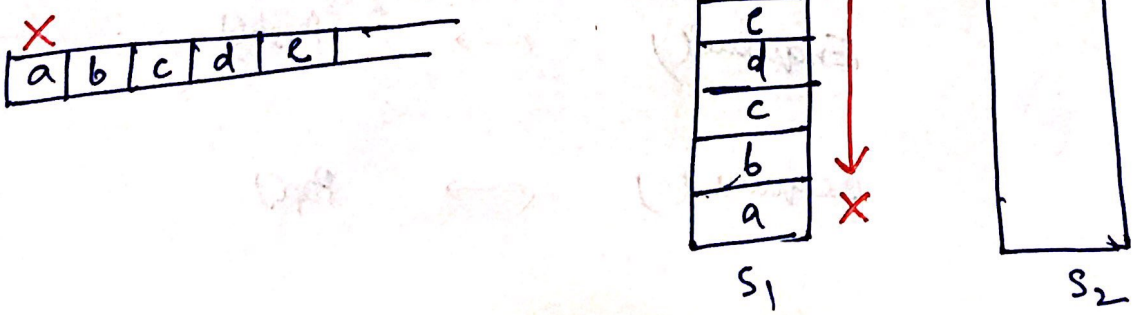
- **Enqueue op^n:**

It is implemented using simple PUSH op^n —

(↓ EQ → PUSH S_1)

- S_1 → stack used for carry elements
- S_2 → temp stack for carry elements in opposite sequence to S_1

- **Dequeue op^n:**



To delete first element from STACK to implement EnQueue, we have to first perform POP op^n on n-1 element first.

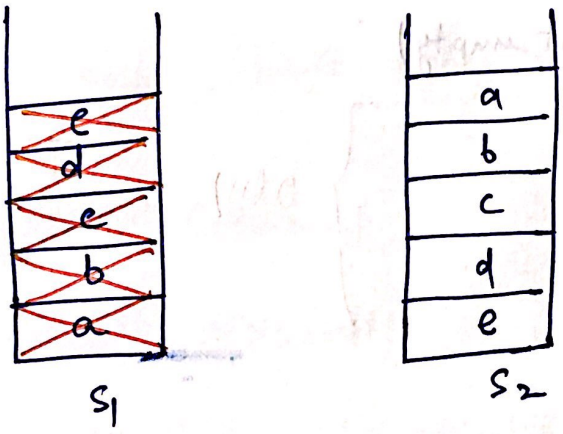
i.e $\left(1\ DQ = N\ POP - S_1\right)$

Pop element from S_1 are saved in S_2 so,

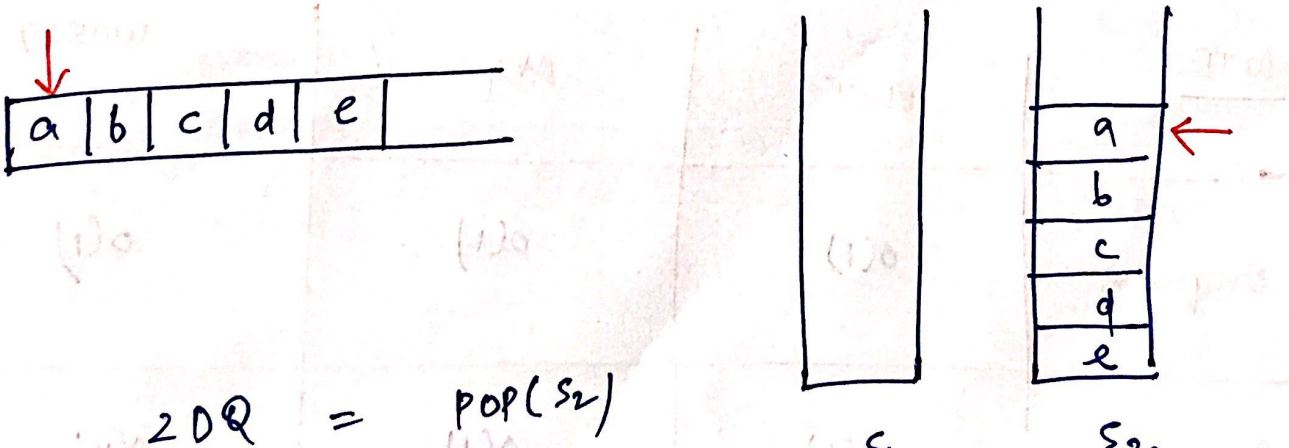
$\left(1\ DQ = N\ POP - S_1 + N\ PUSH - S_2\right)$

n - POP op^n on S_1

n - PUSH op^n on S_2



Now Perform POP from S_2 :



$2DQ = POP(S_2)$

$\vdots$

$NDQ = POP(S_2)$

```
void Enqueue (x)
{
    PUSH(x, S1);
}
```
O(1)

```
int Dequeue ()
{
    if (S2 is not empty)
    {
        return POP(S2);
    }
    else
    {
        while( S1 is not is not empty)
        {
            x = POP(S1);
            PUSH(x, S2);
        }
        return POP(S2);
    }
}
```
O(1) — After 1st Dequeue oph

O(n) — 1st Dequeue oph

NOTE:-	BEST	Avg	WORST
Enqueue	O(1)	O(1)	O(1)
Dequeue	O(1)	O(1)	O(n)

- when S_1 is empty that means Queue is empty and when S_2 is empty and S_1 not empty then we are going to perform first Dequeue oph on Queue.

[S_2 empty & S_1 empty → underflow

S_1 full → overflow]

= Implementing Queue using stack (3-oph) :-

EnQueue()

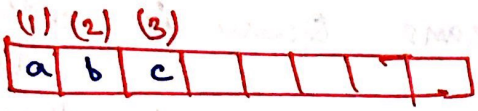
DeQueue()

PUSH()

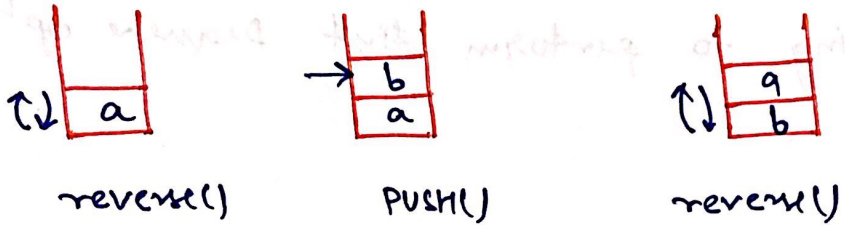
POP()

Reverse()

Enqueue()	Dequeue()
reverse() PUSH() reverse()	POP()

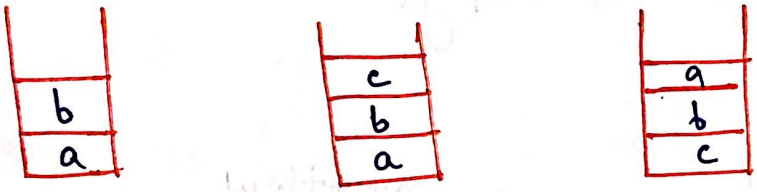


(2)



reverse() PUSH() reverse()

(3)



reverse() PUSH() reverse()

NOTE:-

- Here Dequeue op^{n} T(n) = O(1) and only one STACK is used, so it is both time and space efficient.

- Enqueue op^{n} T(n) = O(1) as we assume T(n) of reverse() as O(1).

Implementing stack using linked list:-

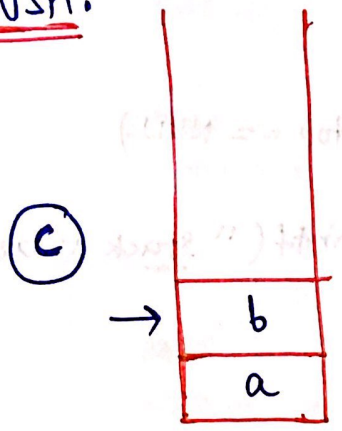
PUSH()

POP()

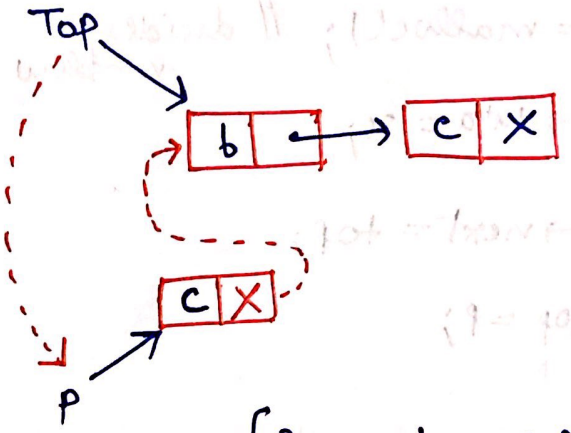
malloc()

free()

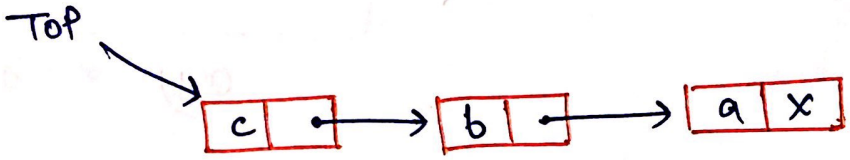
PUSH:



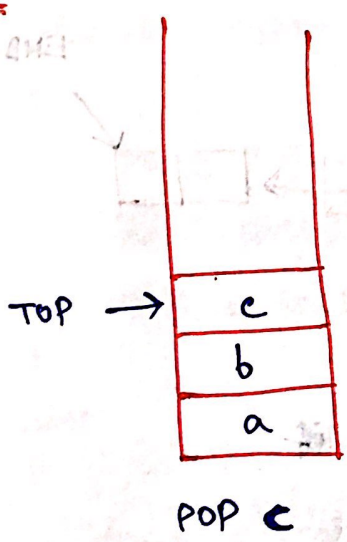
PUSH b c



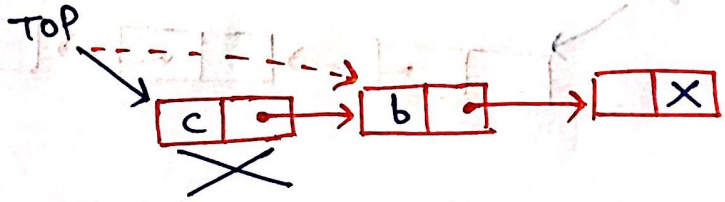

```
P->next = Top;
Top = P;
```



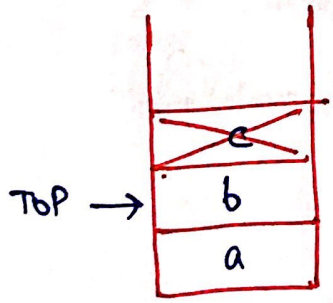
POP:-



POP c




```
temp = top->next;
free(top);
top = temp;
```



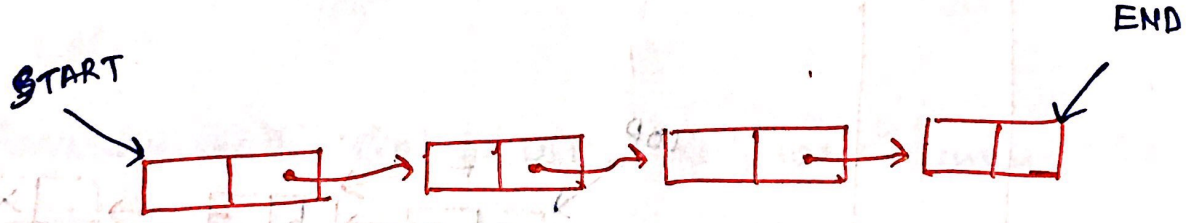
```
PUSH(x)
{
   p = malloc(); // decides overflow
   p->data = x;
   p->next = top;
   top = p;
}
```

O(1)

```
POP()
{
  if (top == NULL)
  {
    printf(" Stack underflow");
  }
  else
  {
    temp = top->next;
    free(top);
    top = temp;
  }
}
```

O(1)

= Implementing Queue using Linked List:-




```
EQ(x)
{
   p = malloc();
   if( p == NULL)
   { printf(" overflow ");
     exit(1);
   }
   else
   {  p->data = x;
```

```
    if ( END == NULL)
    {
        END = P;        // First Insertion
        START = P;
    }
    else
    {
        END -> next = P;
        END = P;
    }
}
```

```
DQ()
{
    if ( START == NULL)          // no element
    {
      printf(" Underflow");
    }
    else if( START == END)       // only one element
    {
        free (START);
        START = NULL;
        END = NULL;
    }
    else                         // more than one element
    {
        temp = START;
        START = START -> next;
        y = temp -> data;
        free(temp);
    }
}
```

Application of STACKS :-

(i) Recursion : → Tail recursion
→ Non-Tail recursion
→ Indirect recursion
→ Nested recursion

(ii) Infix to postfix conversion
Prefix to postfix "
Postfix evaluation

(iii) Towers of Hanoi

(iv) Fibonacci series.

(i) Recursion :- Func calling itself.

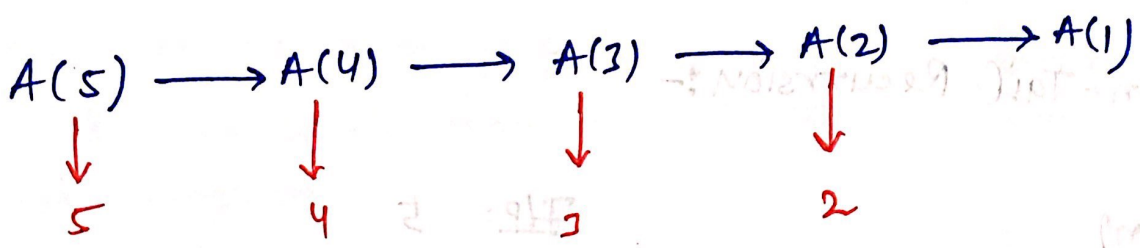
↳ Tail Recursion :

```
A(n)
{ if(n<=1)
    return
  else
  {
    Pf(n);
    A(n-1);
  }
}
```

no work to do after recursive statement.

[STACK SPACE = O(n)]

[Time T(n) = n (Print) + n (PUSH & POP) = 2n = O(n)]



- In tail recursion, func waits un-necessarily as to complete the recursive func.

as, A(2) waits to A(1) and after completion of A(1), A(2) also over so A(2) is unnecessary consuming STACK space.

Similarly A(3) waits to A(2) and so on.

Drawback: Unnecessary waste of memory (STACK space).

- We can make equivalent non-recursive program very easily using a loop.

```
for( i=n ; i>1 ; i--)
{
  printf (i);
}
```

[STACK SPACE = O(1)]

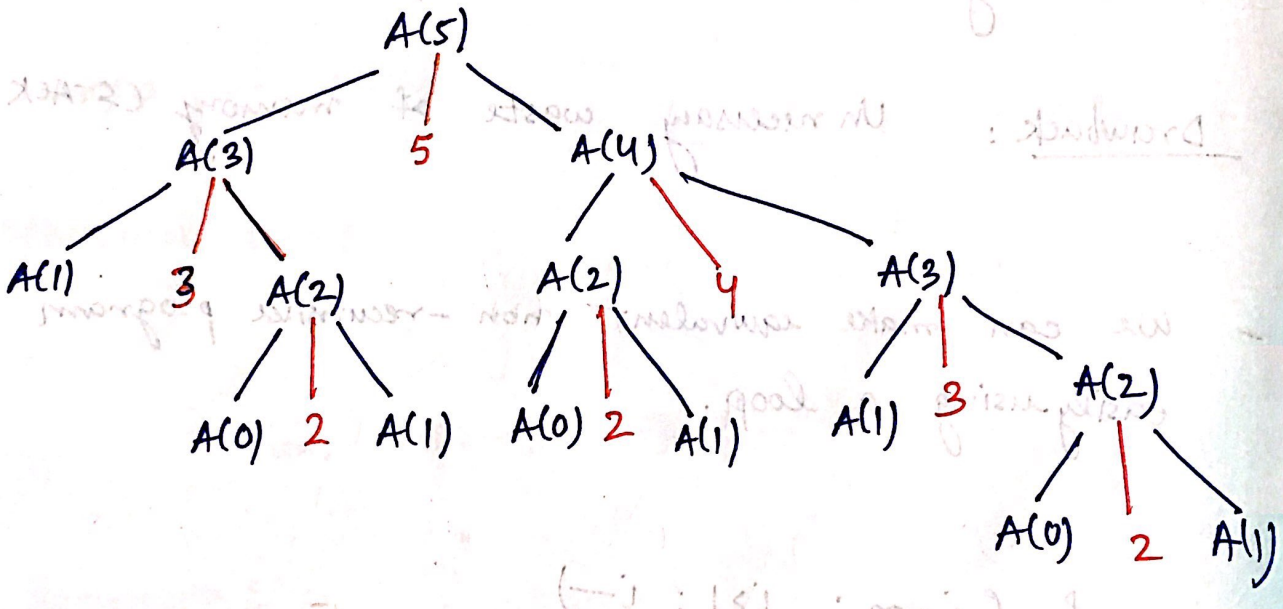
[Time T(n) = n
= O(n)]

Non-Tail Recursion:-

```
A(n)
{
  if(n<=1)
    return;
  else
  {
    A(n-2);
    Pf(n);
    A(n-1);
  }
}
```

I/P: 5

O/P:- (3, 2, 5, 2, 4, 3, 2)



Space (STACK Space) = O(n)

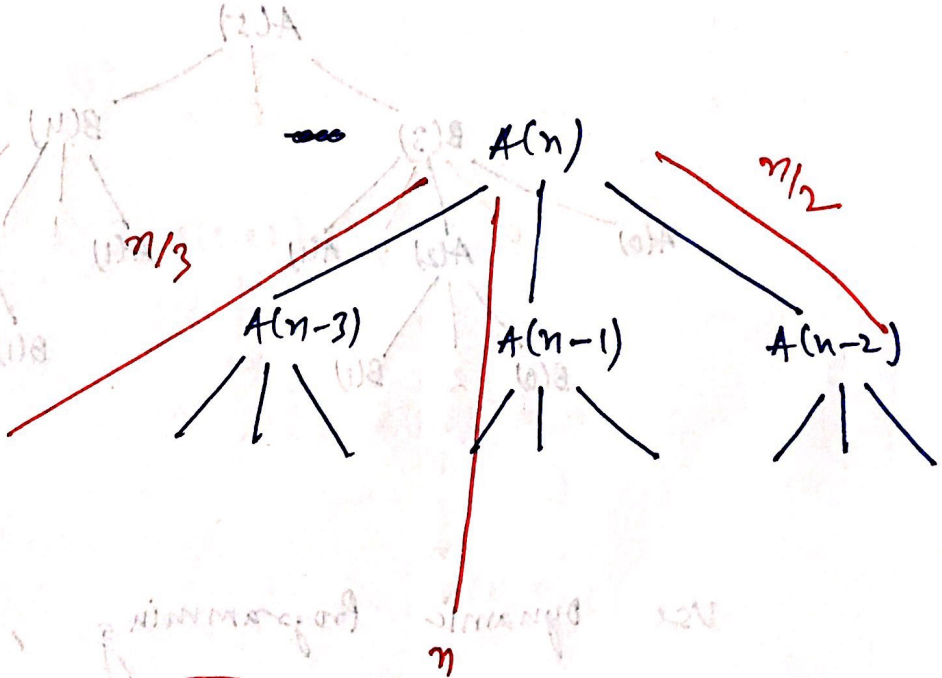
Time complexity = $O(2^n)$ without DP

with DP = O(n) — n distinct func^{h} calls

- In the given recursive program, after the func^n call there is some work than it is known as Non-tail Recursive Program.
- We are not wasting un-necessarily.
- Equivalent non-recursive program is possible but it is difficult.
- If we convert (given) recursive program into (equivalent) non-recursive program, we have to use STACK data structure.
- For the given tail recursive program if we write equivalent non-recursive program O(1) stack space is used.

eg:

```
A(n)
{ if(n≤1)
   return;
  else
  {
   Pf(n-2);
   A(n-3);
   Pf(n);
   A(n-1);
   Pf(n-3);
   A(n-2);
  }
}
```



STACK space = O(n)

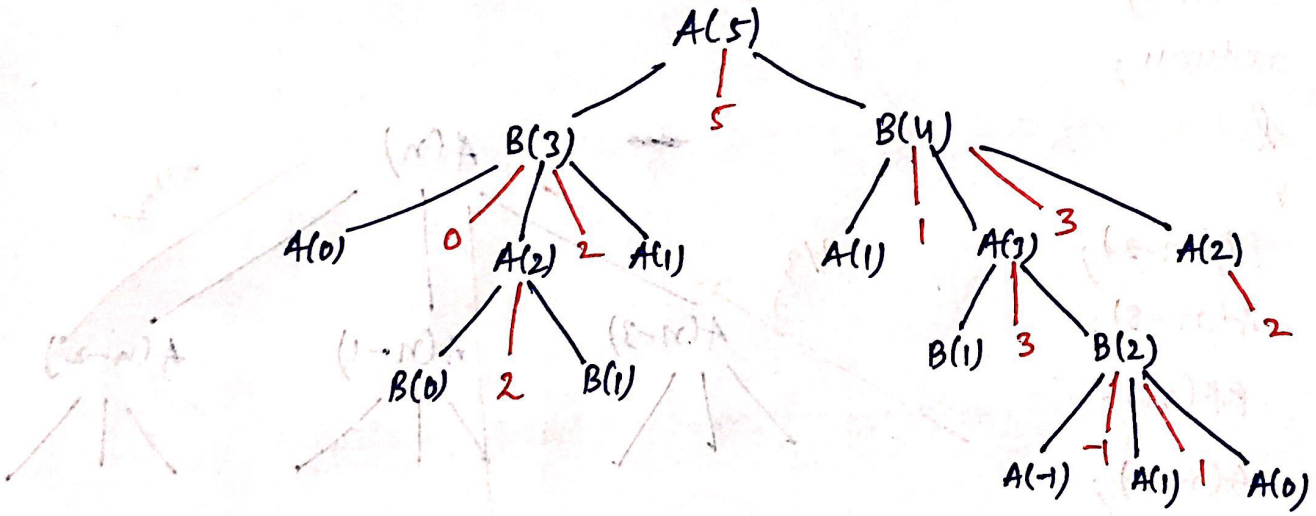
Time = $O(3^n)$ without DP

↳ Indirect Recursion :-

```
A(n)
{
  if(n<=1)
    return;
  else
  {
    B(n-2);
    Pf(n);
     B(n-1);
  }
}
```

```
B(n)
{
  if(n<=1)
    return;
  else
  { A(n-3);
    Pf(n-3);
    A(n-1);
    Pf(n-1);
    A(n-2);
  }
}
```

I/p: A(5) ; O/p:- 0, 2, 2, 5, 1, 3, -1, 1, 3, 2



Use Dynamic Programming, don't expand already visited ~~no~~ call.

without DP

$T(n) = O(3^n)$

with DP

$T(n) = 2n = O(n)$

A() B()

↳ Nested Recursion :-

Ackerman's Recursion

```
A(m,n)
{
  if(m==0)
     return(n+1);
  else
  {if(n==0)
     return (A(m-1,1));
   else
     return ( A(m-1, A(m,n-1)));
  }
}
```

I/p : A(1,5)

O/p : 7