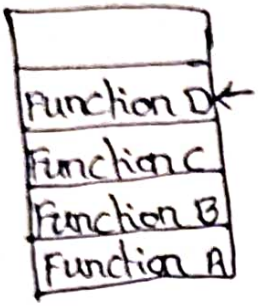
UNIT - 3

STACKS

- A stack is a Linear datastructure.
- Example for stack is, you must seen a pile of plates where one plate is placed on top of another.
- when you want to remove a plate, you remove the topmost plate first.
- when you want to insert a plate, you insert at the topmost position only.
- Hence, you can add and remove an element only from one position which is the topmost position, means only from one end, which is called TOP.
- Hence a stack is called a LIFO (Last-In-First-Out), data structure, as the element that was inserted last is the first one to be taken out.
- Now the question is where do we need stacks in computer science?
- the answer is in function calls.

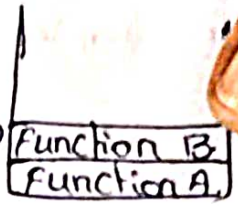
Ex:- · we are executing function A, in the course of its execution, function A calls another function B, Fuction B in turn calls another function C, which calls function D.

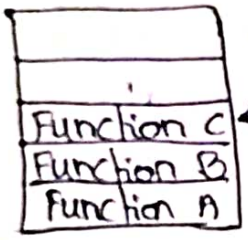
System stack when a called function returns to the calling function :-



← when E has executed, D will be removed for execution.

- when C has executed, B will be removed for execution.





← when D has executed, C will be removed for execution.

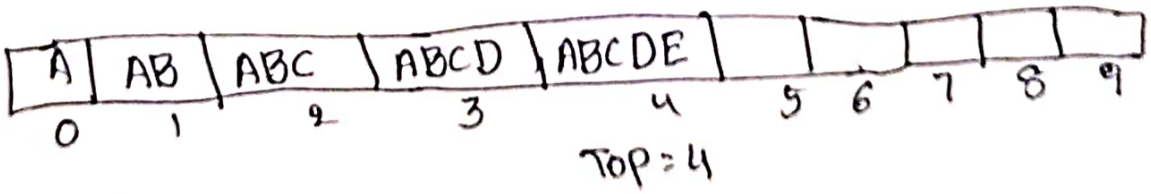
- when B has executed, A will be removed for execution.

- Now when function E is executed, function D will be removed from the top of the stack and executed.
- once function D gets completely executed, function C will be removed from the stack for execution.
- The whole procedure will be repeated until all the functions get executed.

- The system stack ensures a proper execution order of functions.
- Therefore, stacks are frequently used in situations where the order of processing is very important.
- Stacks can be implemented using either arrays or linked lists.

Array Representation of Stacks :-

- In computer's memory, stacks can be represented as a linear array.
- Every stack has a variable called stack pointer (TOP), which is used to store the address of the topmost element of the stack.
- It is the position, where the element will be added or deleted.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If SP or TOP = NULL, then it indicates that the stack is empty and if SP (or) TOP = MAX - 1, then the stack is FULL.

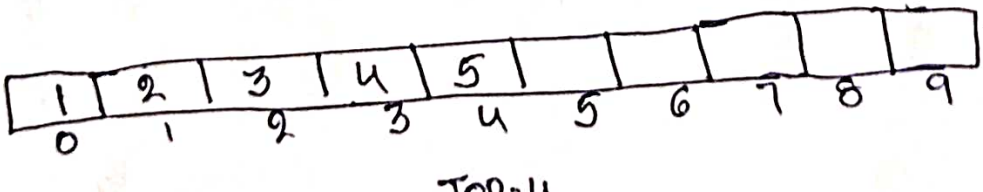


Operations on STACK :-

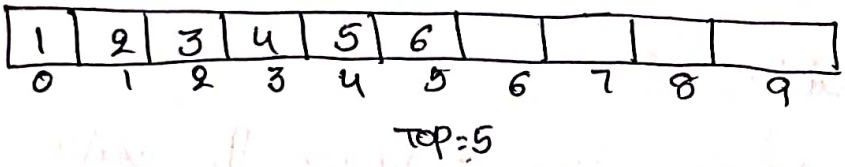
- A stack supports 3 basic operations.
- Push : the push operation adds an element to the top of the stack.
- Pop : The pop operation removes the element from the top of the stack.
- Peek : The peek operation returns the value of the topmost element of the stack.

Push operation :-

- The push operation is used to insert an element into the stack.
- the new element is added at the topmost position of the stack.
- However, before inserting the value, we must first check if TOP : MAX-1, because in this case, the stack is full and no more insertions can be done.
- If we want to insert a value, an OVERFLOW message is pointed.



- To insert an element with value 6, we first check if TOP = MAX-1.
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].



Algorithm:-

Step 1: IF TOP = MAX-1
PRINT "OVERFLOW"
Goto step 4
[END OF IF]

Step 2: SET TOP = TOP+1

Step 3: SET STACK[TOP] = VALUE

Step 4: END.

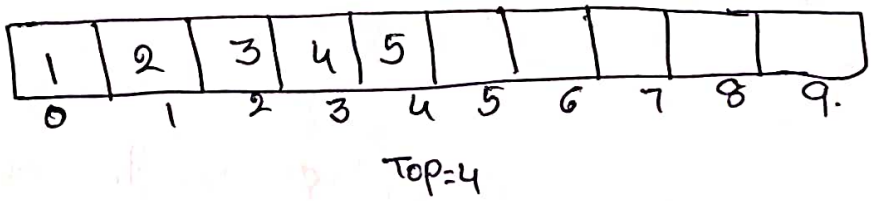
Step 1: we first check for the OVERFLOW condition

Step 2: TOP is incremented so that it points to the next location in the array.

Step 3: The value is stored in the stack at the location pointed by TOP.

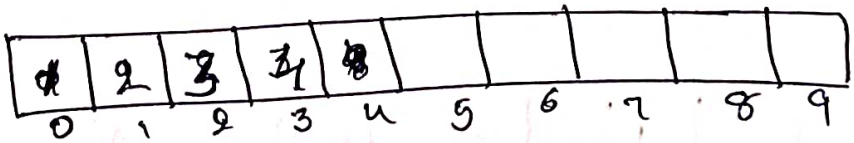
Pop operation :-

- The pop operation is used to delete the topmost element from the stack.
- However before deleting the value, we must first check if TOP = NULL because, it means the stack is empty and no more deletions can be done.
- In this condition, we want to delete the element an UNDERFLOW message is pointed.



TOP=4

- Now we delete the element at topmost position, first check TOP = NULL.
- If the condition is false, then we decrement the value pointed by TOP.



TOP=3

Algorithm :-

Step 1: If TOP = NULL
PRINT "UNDERFLOW"
Goto Step 4
[END OF IF]

Step 2: SET VAL = STACK[TOP]

Step 3: SET TOP = TOP-1

Step 4: END.

- In step 1, we first check for the UNDERFLOW condition
- In step 2, the value of the location in the stack pointed by TOP is stored in VAL.
- In step 3, TOP is decremented.

Peek operation:

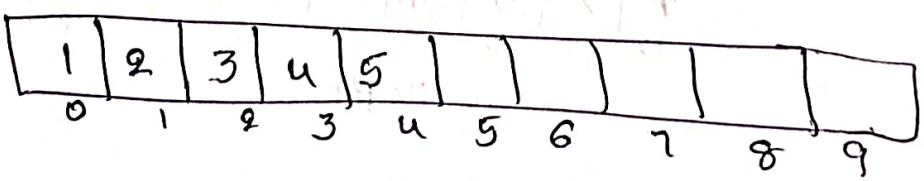
Algorithm:

Step 1: IF TOP = NULL
PRINT "STACK is EMPTY"
Goto step 3

Step 2: RETURN STACK[TOP]

Step 3: END.

- peek is the operation that returns the value of the topmost element of the stack without deleting it from the stack.



TOP: 4

Here, the peek operation will return 5, as it is the topmost element of the stack.

Linked list Representation of STACKS :-

- In a linked stack, every node has two parts.
  (i) one that stores data
  (ii) Another that stores the address of the next node.
- the START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the node pointed by TOP.
- If TOP = NULL, then it indicates that the stack is empty.

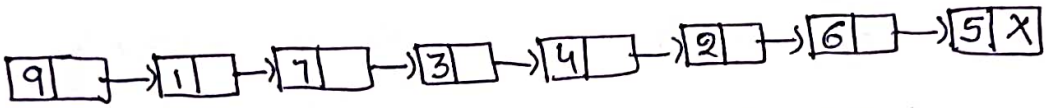


Fig: linked representation of STACK.

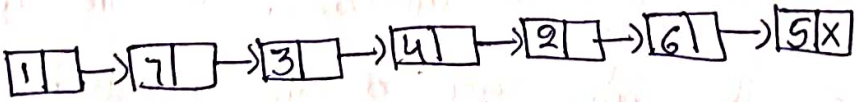
operations on a linked list :-

3 operations push, pop, peek.

1. push operation :-

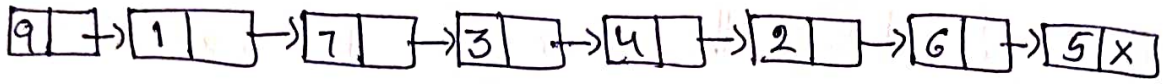
- The push operation is used to insert an element into the stack.
- the new element is added at the topmost position of the stack.

Ex :-



- To insert an element with value 9,
  (i) First we check TOP = NULL, in this case, we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part.
- the new node will then be called TOP.

- However, if TOP != NULL, then we insert the new node at the beginning of the linked STACK and name this new node as TOP.



Algorithm :-

Step 1: Allocate memory for the new node and name it as NEW_NODE.

Step 2: SET NEW_NODE → DATA = VAL

Step 3: IF TOP = NULL

SET NEW_NODE → NEXT = NULL

SET TOP = NEW_NODE

ELSE

SET NEW_NODE → NEXT = TOP.

SET TOP = NEW_NODE

[END OF IF]

Step 4: END.

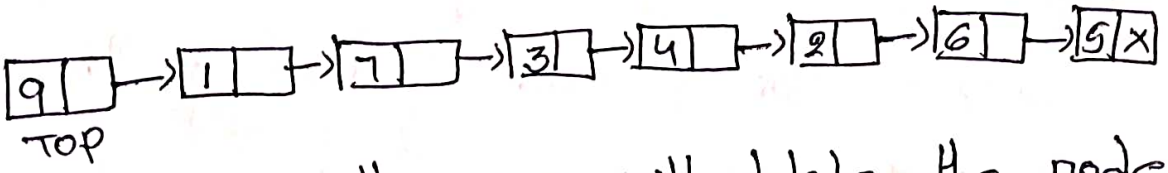
- In step 1, memory is allocated for the new node.
- In step 2, the data part of the new node is initialized with the value to be stored in the node.
- We check if the new node is the first node of the linked list.
- This is done by checking if TOP = NULL

- In case IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP.
- However, if the new node is not the first node in the list, then it is added before the first node of the list and termed as TOP.

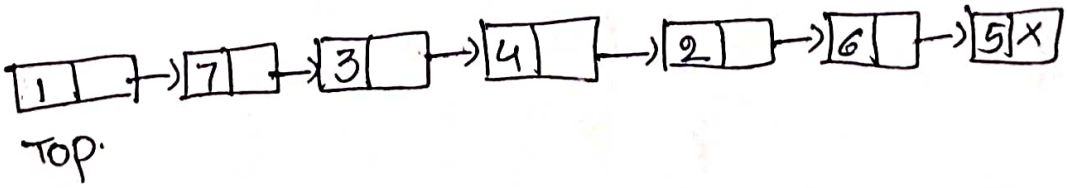
pop operation :-

- The pop operation is used to delete the topmost element from a stack.
- However, before deleting the value, we must first check if TOP = NULL, means that the stack is empty and no more deletions can be done.
- so stack is empty means, it prints UNDERFLOW msg

Ex :-



Here TOP != NULL, then we will delete the node pointed by TOP. and make TOP point to the second element of the linked STACK.



Step 5: EXIT.

Algorithm :-

Step 1: IF TOP = NULL
Print "UNDERFLOW"
Goto Step 5
[END OF IF].

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP → NEXT

Step 4: FREE PTR

Step 5: END.

- In step 1, we first check for the UNDERFLOW condition.
- In step 2, we use a pointer PTR that points to Top.
- In step 3, Top is made to point to the next node in sequence.
- In step 4, the memory occupied by PTR is given back to the free.

Step 5: EXIT.

Applications of STACKS :-

(i) Reversing a list

(ii) parentheses checker

(iii) conversion of an infix expression into post fix

(iv) Evaluation of a postfix expression

(v) conversion of an infix expression into a prefix exp

(vi) Evaluation of a prefix expression.

(vii) Recursion

(viii) Towers of Hanoi

1. Reversing a list :-

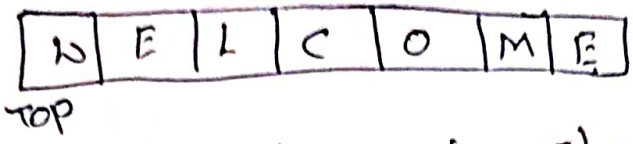
- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack.
- once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Algorithm :-

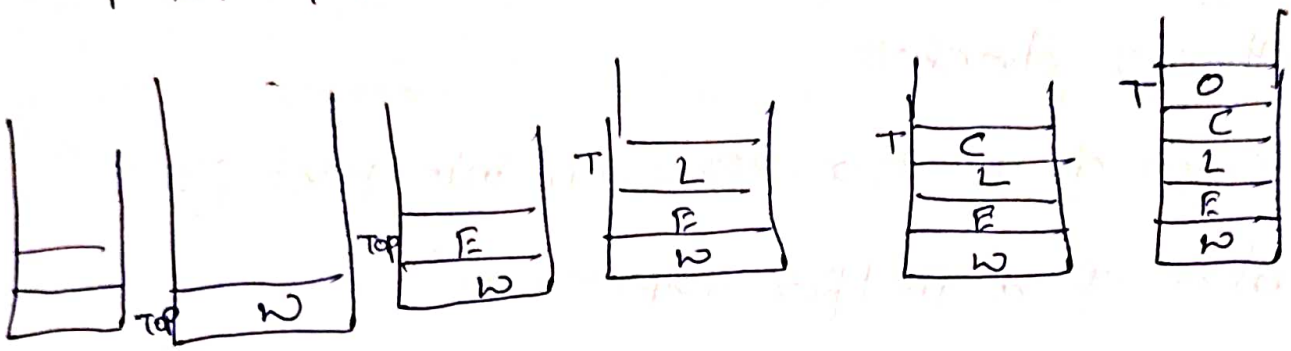
```
Void reverse ()
{
  // input : array A[], size N
  // top = -1, stack S
  //output: Reversing array A[]
  for i: 0 to N-1
      TOP = TOP+1
      S[TOP] = A[i]
  for i: 0 N-1
      A[i] = S[TOP]
  TOP=TOP-
}
```

STOP & EXIT

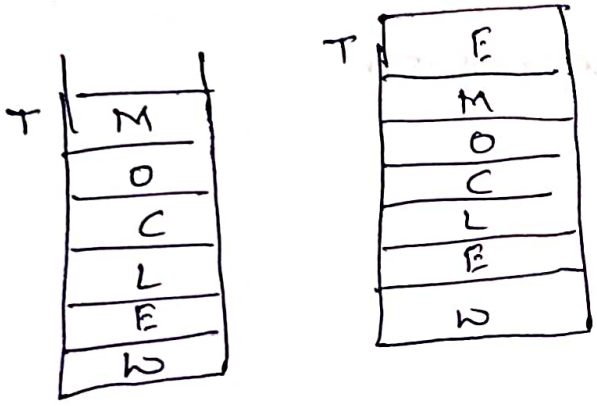
Ex:-



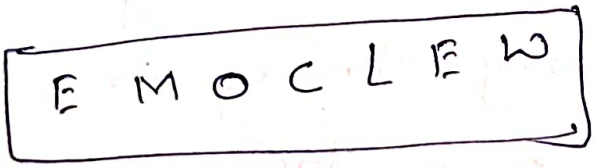
First push elements into stack.

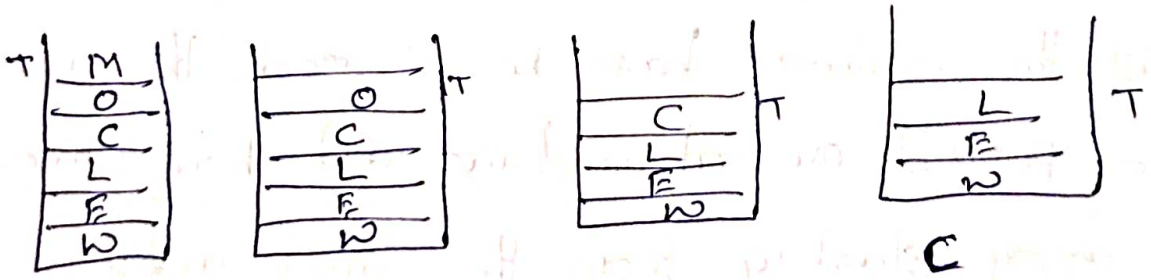


TOP:-1

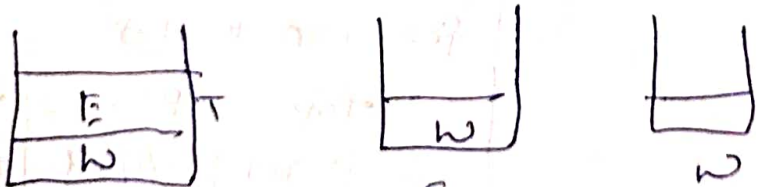


Now Pop element from stack.





E M O



L E W

Finally E M O C L E W.

Evalution of Arithmatic expression :-

3 - (Notations) Expressions

(i) Infix Expression :- The operator is placed in between the operands

Ex: A+B.

A stack is a very effective data structure for evaluating arithmatic expressions, in programming languages.

- An arithmetic expression consists of operands and operators.
- In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Ex:- A + (B-C).

- To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression
- The precedence rules are

operators	Associativity

- Associativity means, if an expression having two operators with same priority, then which one is first evaluated.

Step 5: EXIT.

- Evaluation of arithmetic expression requires two steps.
  (i) First convert the given expression into special notation
  (ii) Evaluate the expression in this new notation.

Notations for Arithmetic Expression:.

- There are three notations to represent an arithmetic expression.
  (i) Infix Notation (ii) prefix Notation (iii) postfix notation.

Infix Notation :-

- the infix notation is a simple, in which each operator is placed between the operands.
- It can be Parenthesized or unparenthesized depending upon the problem requirement.

Ex:- A+B , (C-D) etc.

Prefix Notation :- (also called polish notation)

- In this the operator places before the operands.

Ex:- +AB , -CD etc.

Postfix Notation :-

- The notation places the operator after the operands.
- This notation is just reverse the reverse of polish notation, so it is known as Reverse polish notation. Ex:- AB+ , CD+ ..

Reverse Polish Notation :

operators	precedence & Associativity. (Associ.)
( ), [ ]	left to right
*, /, %.	left to right
+, -	left to right

Ex: for Associativity.

A * B / D.

- Here * & / having same priority, so
- these operators associativity is left to right
- so first multiplication then division.

∴ AB* D/.

Step 5: EXIT.

Conversion of Arithmetic Expression into various notations:.

Convert the following infix expressions into postfix expression

Ex:. Infix exp $(A-B) * (C+D)$

$[AB-] * [CD+]$

$AB-CD+*$ post fix.

Prefix:.

$(-AB) *(+CD)$

$* -AB+CD.$

Ex:-2 Infix exp = $(A+B) / (C+D) - (D*E)$

Prefix

$[+AB] / [+CD] -[*DE]$

$[/+AB+CD] - [* DE]$

$-/+AB+CD * DE$

Postfix

$(AB+] / (CD+) -(DE*)$

$[AB+ CD+/] -[DE*].$

$AB+CD+/ DE* -$

Ex:. 3

Infix	Prefix	postfix.
$A * B$	$*AB$	$AB*$
$(A+B)/C$	$/+ABC$	$AB+C/$
$(A*B)+(D-C)$	$+*AB-DC$	$AB*DC-+$

Algorithm to convert infix to postfix notation:-

Step : 1 Add ")" to the end of the infix expression

Step 2: push "(" on to the stack.

Step 3: Repeat until each character in the infix notation is scanned.

IF a "(" is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it to the postfix notation.

IF a ")" is encountered, then

a. Repeatedly pop from the stack and add it to the postfix expression until a "(" is encountered.

b. Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression.

IF an operator o is encountered, then

(a). Repeatedly pop from the stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 'o'.

(b) push the operator in to stack.

Step 4: Repeatedly pop from stack and add it to the Postfix expression until the stack is empty.

Step 5: EXIT.

Ex:- Infix exp A - (B/C + (D % E * F) / G) * H.

1. (A - (B/C + (D % E * F) / G) * H)

Infix	Stack	Postfix
	(	
A	(	A
-	(-	A
(	(-(	A
B	(-(	AB
/	(-(/	AB
C	(-(/	ABC
+	(-(/+	
	(-(+	ABC/
(	(-(+(	ABC/
D	(-(+(	ABC/D
%	(-(+(%	ABC/D
E	(-(+(%	ABC/DE
*	(-(+(%*	ABC/DE
F	(-(+(%*	ABC/DEF
)	(-(+(%	ABC/DEF*
	(-(+	ABC/DEF*%
/	(-(+/	ABC/DEF*%
G	(-(+/	ABC/DEF*%G
)	(-(+	ABC/DEF*%G/
	(-	ABC/DEF*%G/+

*	(- *	ABC / DEF * % G / +
H	(- *	ABC / DEF * % G / + H
)		ABC / DEF * % G / + H *
		ABC / DEF * % G / + H * -

A = (B / C + (D % E * F) / G) * H

A + [B / C + [D % E * F] / G] * H

~~ABC/+~~ ~~BC/DE%F*G/+~~

Ex: 2 A * B + C * D => AB* CD* +

Ex: 3 (A+B) * (C * (D+E) + F)

=> AB+ CDE+ * F+ *

Evaluation of Arithmetic Expression :-

(i) push the operands into the stack until an operator is reached.

(ii) pop the two operands from the stack, compute the result and also push the result back into the stack

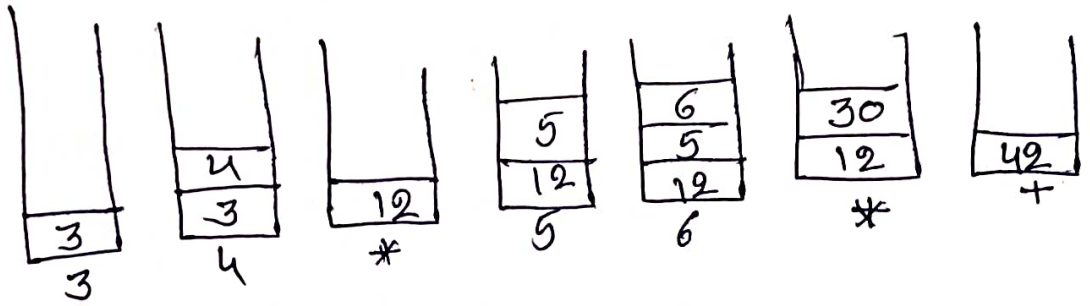
(iii) continue this process until there are no more operators in the RPN and the final result is in the stack.

Ex:- $(3*4)+(5*6)$

in reverse polish notation, it is expressed as

$34*56*+$

stack operation to evaluate this one is

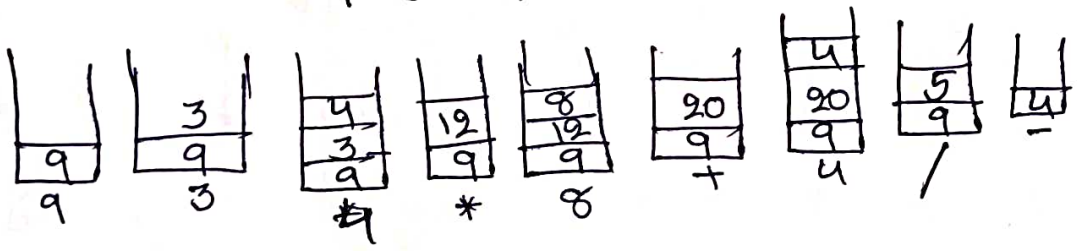


Ex:- $9-((3*4)+8)/4$.

$9-[34*8+]/4 \Rightarrow 9-[34*8+4/] \Rightarrow 934*8+4/-$

$9-20/4$

$9-5=4$.



Conversion of an infix exp into a postfix EXP:

1. Scan each character in the infix expression. For this repeat steps 2-8 until the end of the infix exp.
2. push the operators into the stack, operand into the postfix, and ignore all the left parenthesis until a right parenthesis is encountered.

(or)

Rules:

1. Reverse the Infix expression.
2. Scan the exp from left to right.
3. Whenever the operands arrive, print them.
4. If the operators arrives, push into stack.
5. If the incoming operators has the same precedence with a than the Top of the stack, then push the incoming operators into the stack.
6. If the incoming operators has lower precedence than the Top of the stack, pop and print the top of the stack.
   - Test the incoming operators against the top of the stack again and pop the operators from the stack till it finds the operators of a lower precedence or same precedence.
7. If the incoming operators has the same precedence with the top of the stack and the incoming operators is ^, then pop the top of the stack till the condition is true.
   - If the condition is not true, push the ^ operators.

8. When we reach the end of the expression, Pop and Print all the operators from the top of the stack.

9. If the operator is ")" then push into the stack.

10. If the operator is "(" then pop all the operators from the stack till it finds ")".

11. At the end, reverse the output.

Eg: Exp is K+L-M*N+(O^P) * W/U/V * T+Q.

+ + - +KL *MN*//*^OPWUV

1. ^OP

1. *MN *//*^OPWUVT

+KL

- +KL *MN *//*^OPWUVT +Q

+ + - +KL* MN + _____

^OP *W

*^OPW /U

/*^OPWU /V.

//*^OPWUV *T

+KL - *M - . .

Infix exp : K+L-M*N+(OVP)*W/U/V*T+Q. 14 2

1. Reverse Exp : Q+T*V/U/W*)PVO(+N*M-L+K.

Infix	Stack	Prefix
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTUV
W	+*//	QTUVW
*	+*//*	QTUVW
)	+*//*)	QTUVW
P	+*//*)	QTUVWP
V	+*//*)V	QTUVWP
O	+*//*)V	QTUVWPO
(	+*//*	QTUVWPOV
+	+*+	QTUVWPOV*//*
N	++	QTUVWPOV*//*N
*	+**	QTUVWPOV*//*N
M	+**	QTUVWPOV*//*NM
-	+*-	QTUVWPOV*//*NM*
L	+-	QTUVWPOV*//*NM*L
+	+-+	QTUVWPOV*//NM*L
K	+-+	QTUVWPOV*//NM*LK.
		QTUVWPOV*//NM*LK+-++

The final exp is

ATUVWPO^**//*NM*LK+-++

Now reverse this exp

∴ ++-+KL*MN*//*^OPWVUTA.

Ex:- Infix Exp is (A-B/C)*(A/K-L).

Reverse the infix exp

)L-K/A(*)C/B-A(

Infix	STACK	Postfix
)	)	
L	)	L
-	)-	L
K	)-	LK
/	)-/	LK
A	)-/	LKA
(		LKA/-
*	*	LKA/-
)	*)	LKA/-
C	*)	LKA/-C
/	*)/	LKA/-C
B	*)/	LKA/-CB
-	*)-	LKA/-CB/
A	*)-	LKA/-CB/A
)	*	LKA/-CB/A-
		LKA/-CB/A-*

Recursion :

- A recursive function is defined as a function that calls itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases.

(i) Base case :- in which the problem is simple enough to be solved directly with out making any further calls to the same function.

(ii) Recursive case :- (i) The problem is devided into simpler sub parts. (ii) The function calls itself but with sub-parts of the problem. (iii) The result is obtained by combining the solutions of simpler sub-parts.

Ex: calculating factorial of a number.

$n! = n \times (n-1)!$ [To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$

This can be written as

$5! = 5 \times (5-1)! = 5 \times 4!$

$4 \times (4-1)! = 4 \times 3!$

$3 \times (3-1)!$

$3 \times 2!$

$2 \times (2-1)!$

$2 \times 1!$ [$\because 1! = 1$]

$= 5 \times 4 \times 3 \times 2 \times 1$

$= 120.$

- Every recursive function must have a base case and a recursive case.

Base case: is when n=1, because if n=1, the result will be 1 as 1! = 1.

- Recursive case:- of factorial function will call itself.

```
#include<stdio.h>
int fact(int);
int main()
{
    int num, val;
    printf("enter the number");
    scanf("%d", &num);
    val = fact(num);
```

```
#include<stdio.h>
int main()
{
    int i, fact=1, number;
    printf("enter a number");
    scanf("%d", &number);
    for(i=1; i<=number; i++)
    {
        fact = fact * i;
    }
    printf("factorial is %d", fact);
    return 0;
}
```

Types of Recursion :-

- Recursion is a technique that breaks a problem into one or more subproblems that are similar to the original problem.
- Any recursive function can be characterized based on
  (i) whether the function calls itself directly or indirectly (direct or indirect recursion).
  (ii) whether any operation is pending at each recursive call (tail recursive or not),
  (iii) the structure of the calling pattern (linear or tree recursive)

Direct Recursion :-

- A function is said to be directly recursive if explicitly calls itself.

Ex:-
```
int Func(int n)
{
    if (n == 0)
        return n;
    else
        return (Func(n-1));
}
```

- The function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

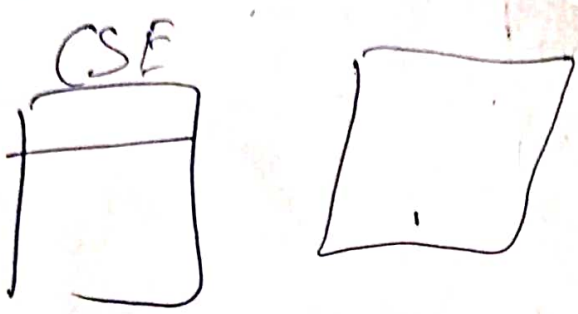
- Direct recursion occurs when a function calls itself within its body.
- this type of recursion doesn't involve any intermediary functions.

Using Recursion :-

```
int fact (int n)
{
   if (n == 1)
      return 1
   else
   return (n* fact(n-1));
}
void main()
{
  int number;
  int fact()
  Printf("enter a number");
  scanf("%d", &number);
   fact = factorial(number)
   Printf(" Factorial is %d", fact);
   return 0;
}.
```

Fibonacci

```
#include<stdio.h>
int Fibonacci(int);
int main()
{
  int n, i=0, res;
  Printf(" enter the no. of terms");
  scanf("%d", &n);
  Printf(" Fibonacci series");
  for(i=0; i<n; i++)
  {
     res = Fibonacci(i);
     Printf("%d", res);
  }
  return 0;
}
int Fibonacci(int n)
  {
      if (n==0)
         return 0;
      else if (n==1)
         return 1;
      else
        return(Fibonacci(n-1)
                + Fibonacci(n-2));
  }.
```



converting Recursive Functions to tail Recursive :-

- A non tail recursive function can be converted into a tail recursive function by using an auxiliary parameter.
- the auxilary parameter is used to form the result.
- when we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation.

Recursion versus iteration :-

- Recursion is a top-down approach to problem solving in which the original problem is divided into smaller sub problems.
- Iteration follows bottom up approach that begins with what is known and the constructing the solution step by step.
- Recursion is an excellent way of solving complex problems.
- The main application of Recursion is Towers of Hanoi.

Indirect Recursion :-

- It occurs when two or more functions call each other in a circular manner.

Ex:-

```
int func1 (int n)
{
    if (n == 0)
        return n;
    else
        return func2(n);
}
int func2 (int x)
{
    return func1(x-1);
}
```

Tail Recursion :-

- It occurs when a function calls itself as its last action, meaning there are no pending operations to perform after the recursive call.

Ex:

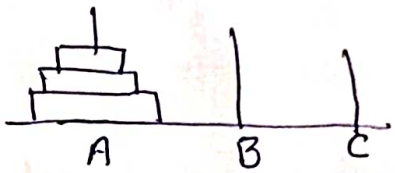
```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return(n * fact(n-1));
}
```

Non-tail Recursion

```
int Fact(n)
{
    return Fact1(n,1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
        return Fact1(n-1, n*res);
}
```

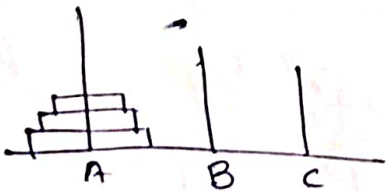
Towers of Hanoi :-

- The problem consists of 3 rods (poles) and 3 rings mounted on a pole A.
- The problem is to move all these rings from pole A to pole C, while maintaining the same order.
- We will do this using a spare pole.
- Here A is the source pole, C is the destination pole and B is the spare pole.

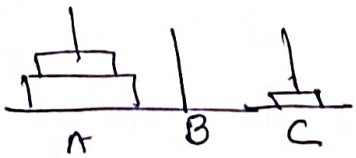


- only one disk can be moved at a time.
- Each move consists of taking the top disk from one stack and placing it onto another stack.
- No disk may be placed on top of a smaller disk.

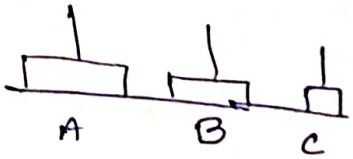
Step 1:



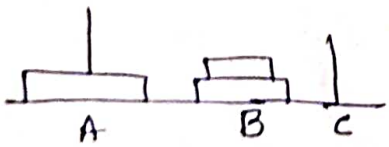
Step 2



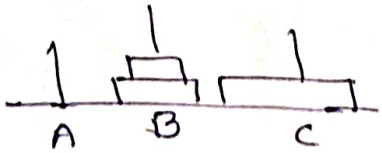
Step 3



Step 4



Step 5



Step 6