STACKS

A stack is a linear data structure in which insertions & deletions are allowed only at the end, called the top of stacks.

Operations

→ push(data) → Inserts data

→ pop() → deletes last inserted element

→ top() → returns the last inserted element without ~~use~~ removing it

→ size() → returns no. of elements in the stack

→ isEmpty() → returns True if stack is Empty else False

→ ~~if~~ isFull() → returns True if stack Full else false

Array Implementation

As it's an array insertions & deletions can be performed anywhere but as it's because we want to perform it like a stack, insertions & deletions can only be performed at the ~~one~~ ends.

We keep a variable top which keeps track of last inserted element (topmost) element in an array.

To indicate stack is empty we put -1 in top.

Note: top = 0 indicates that topmost element is at index 0 hence there is one element in the stack.

For PUSH

→ top incremented by 1
→ New element pushed at position top.
→ When a stack is full it's called overflow state & new element can't be pushed

FOR POP

→ The element at top is deleted
→ top decremented by 1

Implementation
almost every func^n req. the stack_arr[] array, hence instead of passing it to every function, it is a better option declare it globally.

Note: We cannot remove the array element, still, we can give the user an illusion that element is deleted by decrem.. the top variable

Code:

```
#include<stdio.h>
#--"------<stdlib.h>
#define MAX 4

int stack_arr[MAX]; } Global declaration
int top=-1;

void push(int data){

if(top==MAX-1){
printf("stack Overflow\n");
return; // indicates end of func^n
}
```

```
    top = top + 1;
    stack_arr[top] = data;

}

int pop()
{
    int value;
    if (top == -1)
    {
        pf("Stack Underflow \n");
        exit(1); // abnormal termination of
    }                // program
    value = stack_arr[top];
        top--;
    return value;
}

void print()
{
    int i;
    if (top == -1)
    {
        pf("Stack underflow \n");
        return;
    }
    for (i = top; i
    pf("Elements of stack\n");
    for(i=top; i>=0; i--)
        pf("%d\n", stack_arr[i]);

}
```

```
int main(){

  int data;
push(1);
--11--(2);
--11--(3);
--11--(4);
--11--(5);

data = pop();
--11--------;
pf("After pushing & popping");
print();

data = pop();
--11--------;
--11--------;

return 0;

}
```

o/p:

Stack Overflow

After pushing & poping
Elements of stack
2
1

Stack Underflow

Introduction to infix, prefix & postfix

Infix Expression:

⟨operand⟩ ⟨⟨operator⟩⟩ ⟨operand⟩

For: 2 + 3 , 4 * 5 + 6

Precedence:
1) (), {}, []
2) ^ → R→L
3) *, /, L→R
4) +, -, L→R

Prefix Expression: (Polish notation)

⟨⟨operator⟩⟩ ⟨operand⟩ ⟨operand⟩

Shortcut: (Infix to Prefix)

→ Put brackets everywhere
→ Start from innermost bracket
→ Put the operators outside resp. bracket before the rg.

for e-g.

① 4 * 2 + 3 → Infix

⇒ ~~((4 * 2) + 3)~~

⇒ ((4 * 2) + 3)

⇒ +(*(4 * 2) + 3)

= + * 4 2 3 → Preffix

⇒ To conv. to Infix

+ * 4 2 3

(2) $5 - 6/3$ → infix

$\Rightarrow (5-(6/3))$

$\Rightarrow (5-(6/3))$

$\Rightarrow -5/63$ → prefix

To decode this to infix
Read from R → L & Take most close sign
L to R

$-5/63$

$5-6/3$

Postfix expression: (Reverse polish)

<operand><operand><<operator>>

To conv. to postfix
1) 1st & 2nd step same
2) operator after resp. brackets after ng.
for e.g.

(1) $4*2+3$

→ $((4*2)+3)$

→ $((4*2)+3)$

⇒ ~~42*3+~~ $42*3+$

⇒ (2) 5 - 6/3

⇒ (5 - (6/3))

563/-

To Decode: entities (groups)
Take two ~~ng-s~~ at a ~~time~~ time from R to L & insert the closest sign bet" them until you don't finish with the operators

PREFIX EXPRESSION EVALUATION

→ ~~Start~~ Go from R to L.
→ Operate when an operand arrives

-+7*45+20

(1)

	+			*			+				
Left	2	↗ 2+0 = 2	L	4	↗ 4*5 = 20	L	7	↗ -7+20			
Right	0		R	5		R	20				
				2			2				

	-	↗ 27-2 = 25	
L	27		
R	2		25

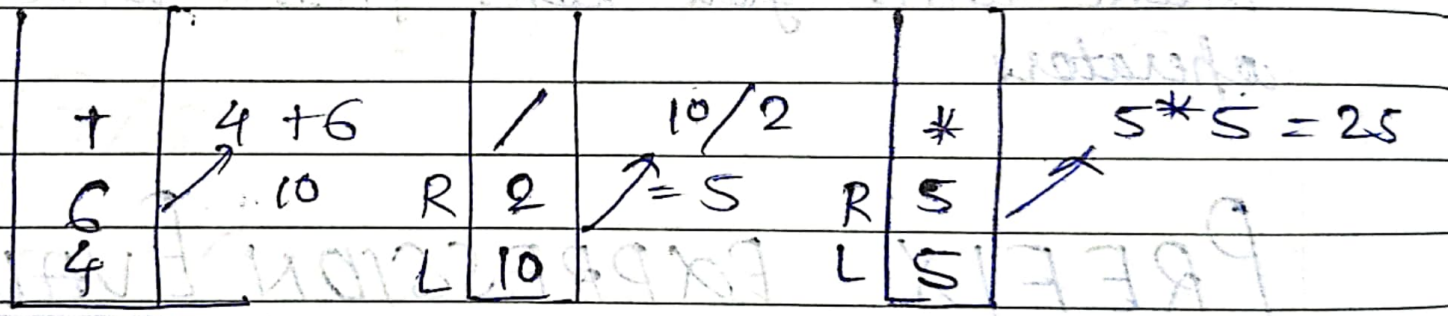
∴ 25 is our final ans.

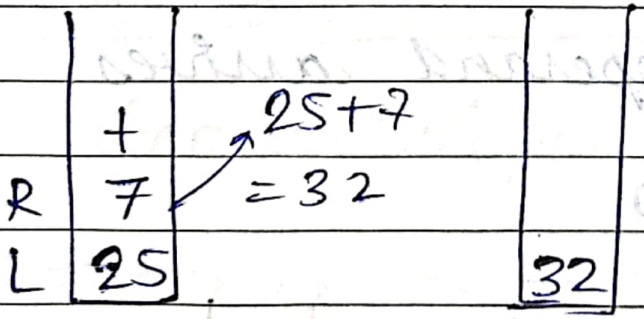
POSTFIX EVALUATION

→ Go from L to R
→ Operate when an operand arrives

E-g.

46 + 2 / 5 * 7 +





32 is the final ans.

INFIX TO POSTFIX

→ Go from L to R

→ If operand print

→ If 'C' push

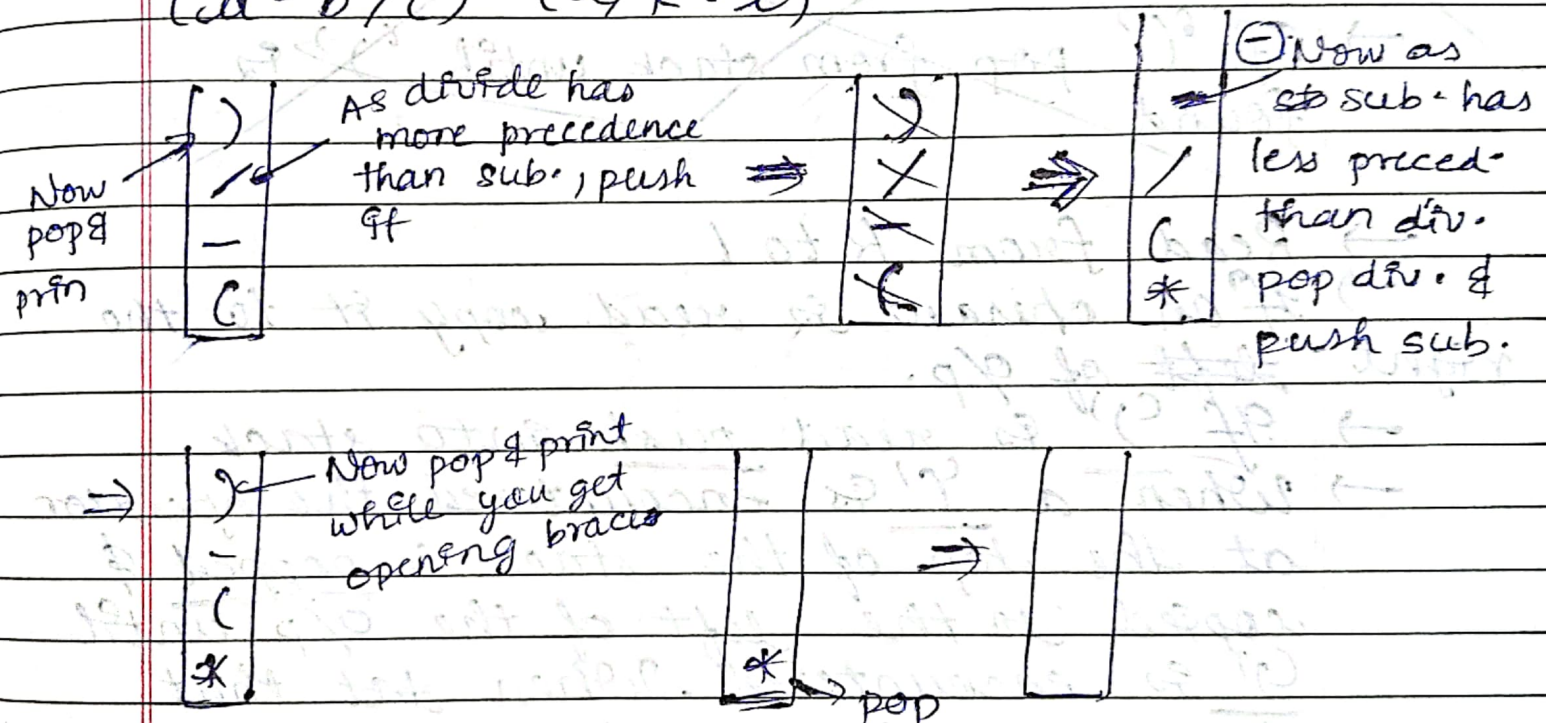
→ If '(' push to stack

→ If ')' pop from stack until '(' is found

→ For an operator pop from stack & print until an operator with less precedence is found.
When an operator with less precedence is push into the stack

For eeg.

(a - b / c) * (a / k - l)

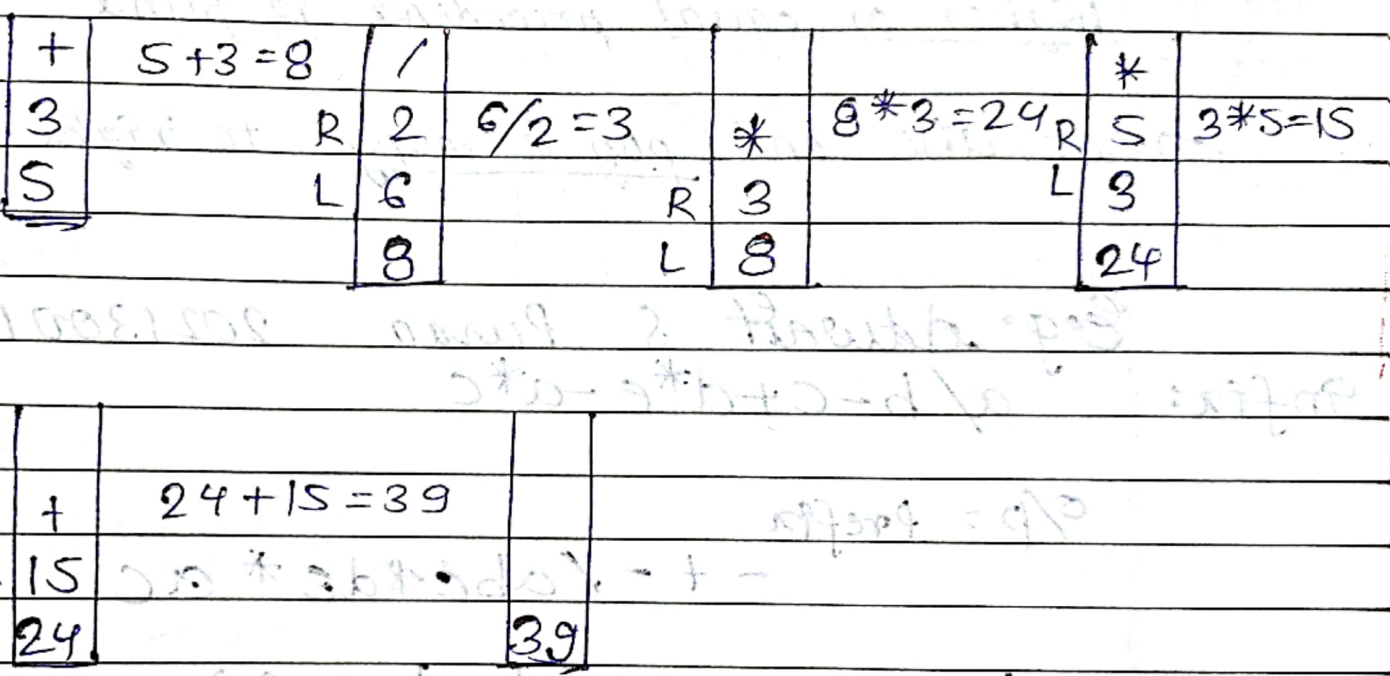


abc/-ak/l-*

Postfix Postfix Evaluation:

1) Exp^{n}: 53+62/*35*+
Res: 39

Go L-R



Inf. to Prefix

→ If operand print

→ If ')'

→ Go from R–L

→ If ')' push

→ If operand copy to right of o/p

→ If '(' pop & print to right of o/p until ')'

→ If operator has higher or equal precedence than top push
else for lower prec. than top pop & copy to the right of o/p until higher or equal precedence is found

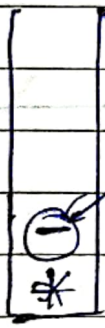
→ At the end pop & copy to right of o/p

Eg: Adwait S Purao 2021300101

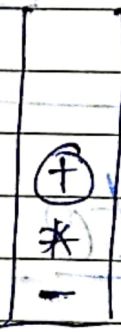
Infix: a/b-c+d*e-a*c

o/p: Prefix

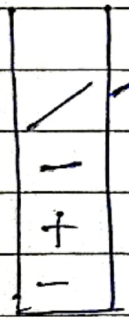
- + - / a b c * d e * a c



As '-' has less precedence than *, pop * & print



As '+' has less prec. than *, ∴ pop & print *

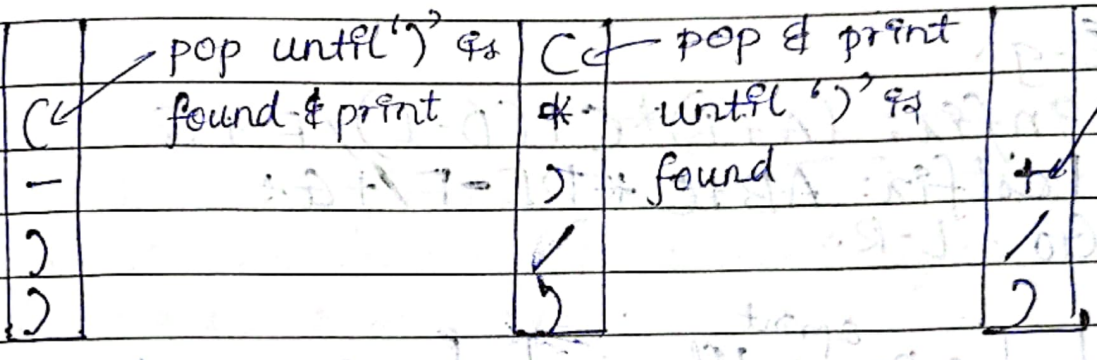


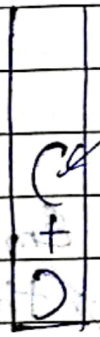
Now as the expression is over pop & print the stack until its not empty

E.g:

Infix: (P+(Q*R)/(S-T))

O/P: Prefix: +P/*QR-ST





Inf. to Postfix

→ Go L-R

→ If '(' then push into stack

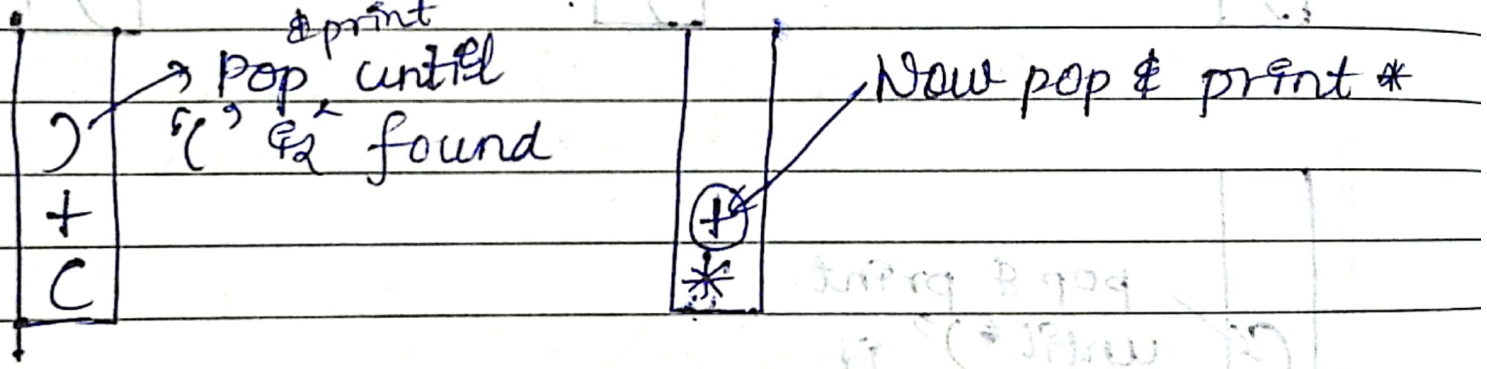
→ If ')' then pop & print until '('

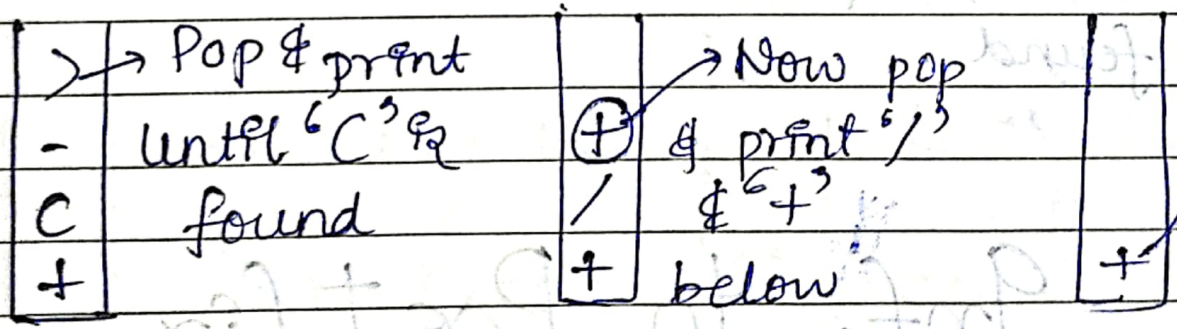
→ If operand print to the O/P

→ If operator has higher prec. than top push
else lower than or equal prec. then pop & print

→ At the end pop entire stack.

E.g.
Infix: (A+B)*C+(D-E)/F+G
Postfix: AB+C*DE-F/+G+
Go L-R.

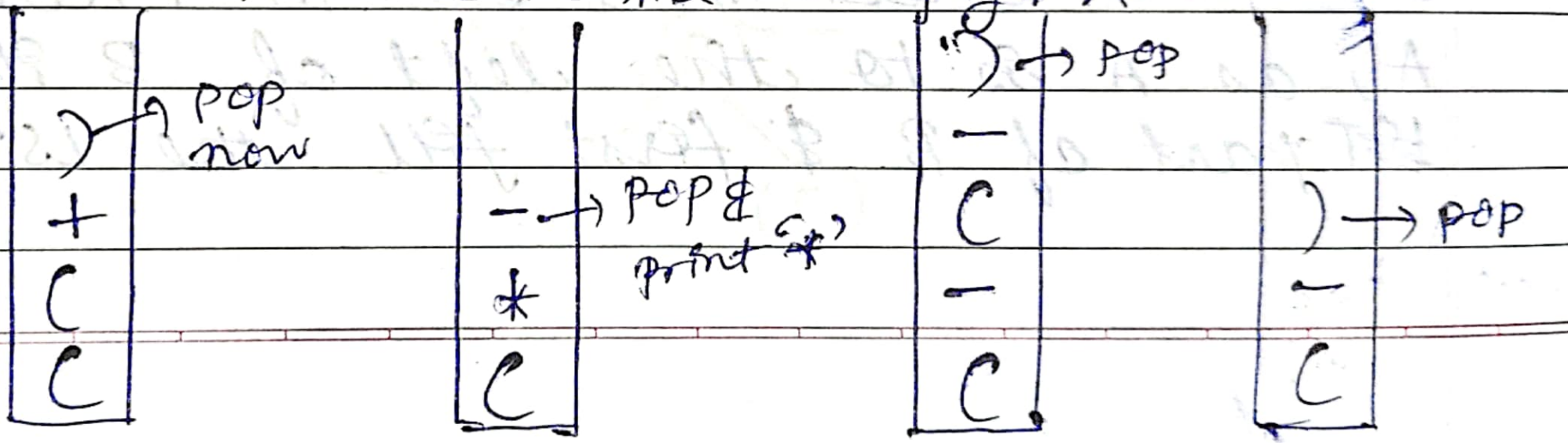


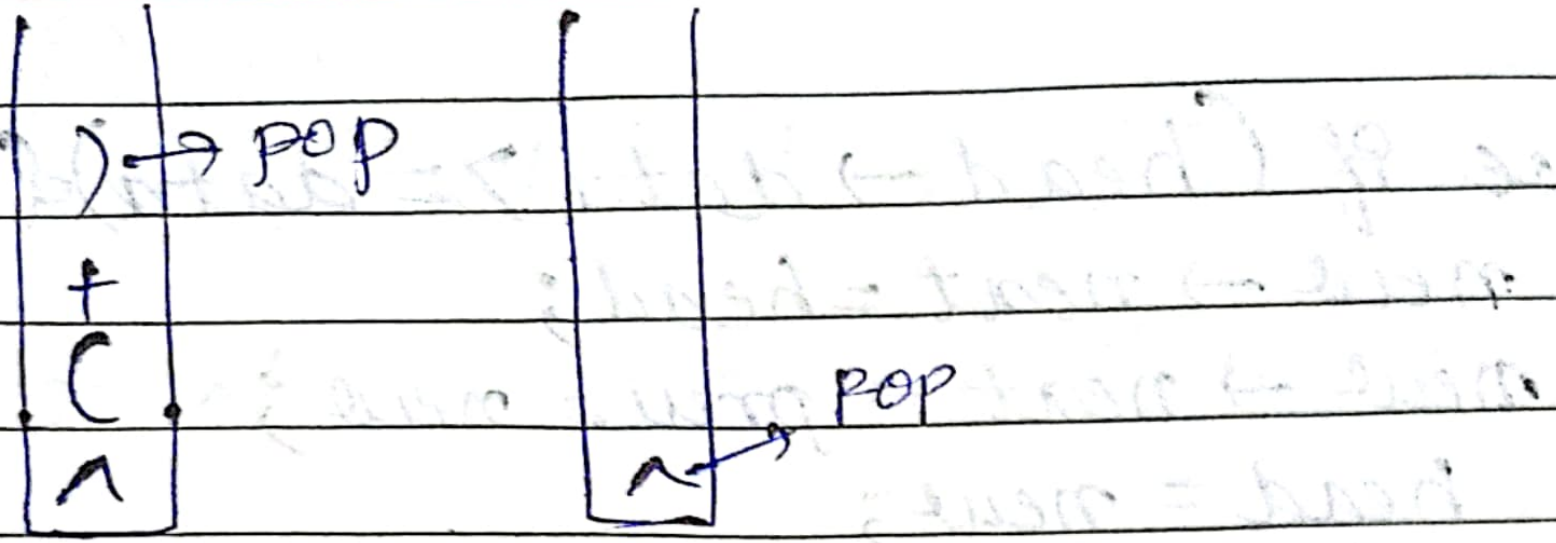


Infix to Postfix:

(i) $(a+b)*c-(d-e))\wedge(f+g)$

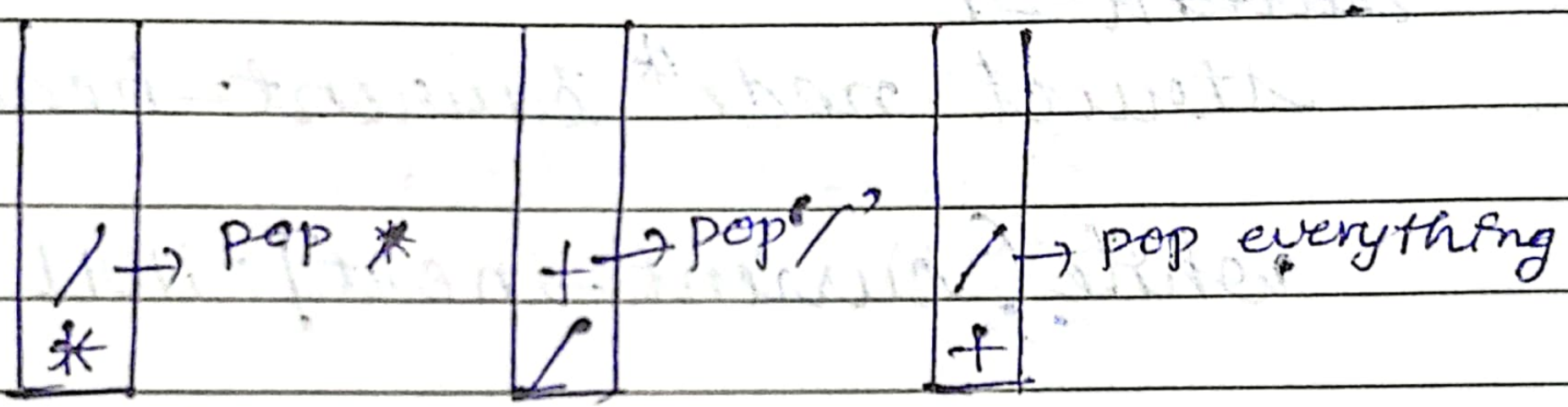
o/p : $ab+c*de--fg+\wedge$





(2) a * b / C + d / e

o/p: ab*c/de/+



PREFIX EVALUATION:

1) Exp^n : + 9 * 26
Res: 21

Go from R-L

S-I		*				+	
	L	2	2*6=12		L	9	9+12=21
	R	6			R	12	

2) - + 7 * 45 + 20

		+				*			
	L	2	2+0=2		L	4	4*5=20		
	R	0			R	5		20	
						2		2	

2) / + 33 - + 47 * + 123

		+						+			-		
	L	1	1+2=3		*		3*3=9	L	4	4+7=11	L	11	11-9=2
	R	2		L	3			R	7		R	9	
		3		R	3				9				

	+						
L	3	3+3=6		/	6/2 = 3		
R	3		L	6		3	
	2		R	2			