Queue

STACK

→ Stack is a primitive, linear data structure in which insertion of one element and deletion of the existing element can be done at one end called as top of the stack.

→ It works on LIFO principle that means last_In_first_out principle ie Last element to be inserted to the stack will be the first element to be removed at last.

→ In a stack, the most accessible element is only the top element of the stack.

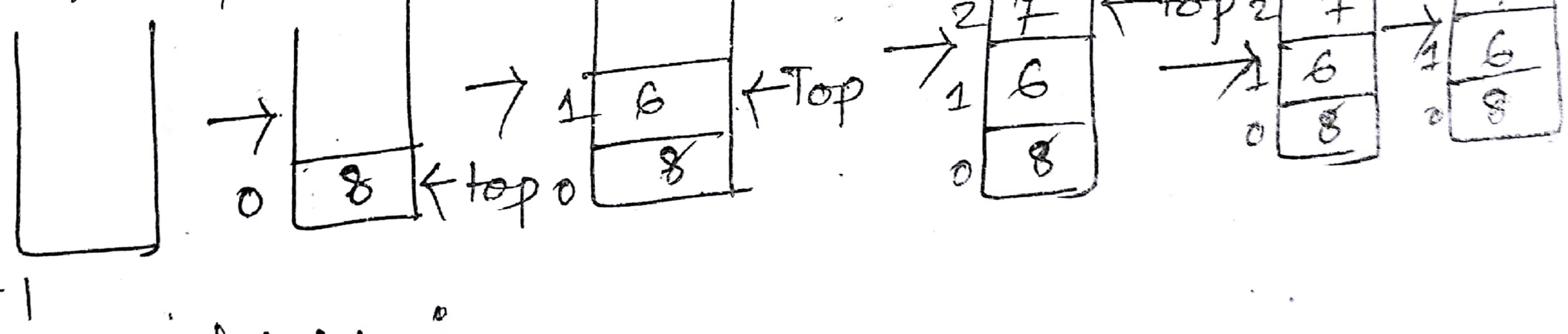
Ex :- stack of dishes.

Basic operation :

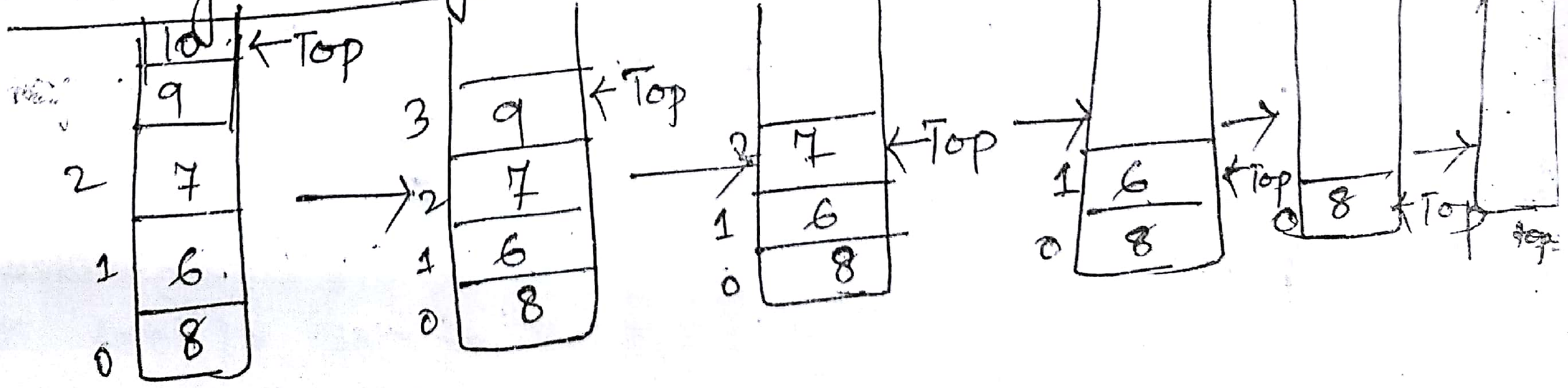
1) Insertion operation termed as push()
2) deletion operation termed as pop()

Ex :- insertion operation into a stack termed as push()
ie Push operation :-

i/p → 8, 6, 7, 9, 10.



Similarly deleting :



Memory Representation of stack :- It of 2 type

(1) Static Representation :-

> In case of linear array rep^n of stack, a index variable top contains the location of top element of the stack and a variable MAXSTK which gives the maximum number of elements that can be held by the stack.

(2) Dynamic Representation :- (by using pointer & dynamic memory allocation ie linked list)

List of operation on stack :-

1) Create :- To create stack 's' as an empty stack.
2) Pop (s, i) :- To access and remove the top element of the stack 's'.
3) Push (s, i) :- To push element i on to stack's'.
4) Peep (s) :- To access the top element of the stack 's' without removing it from the stack
5) Isfull(s) :- to check wheather the stack is full
6) Isempty(s) :- to check wheather the stack s is empty

Condition for push & pop operation :-

i) for push operation :-

> for adding a element to stack (push operation) we have to check wheather there is space (free) is present in stack or not, if not then overflow condition arises. ie top = MAXSTK - 1

ii) For Pop operation :-

> For deleting a element from stack (pop operation) we have to check wheather there is an element in the stack to be deleted or not, if not then underflow condition arises.

ie top = -1

Program

For Push operation :-

'C' code

```
void PUSH(int element)
{
  if(top==MAXSTK-1)
    {
      Pf("overflow");
        getch();
        exit(0);
    }
else
    {
      top=top+1;
      S[top] = element;
    }
}
```

#define MAXSTK 10
int S[MAXSTK], top=-1;

Algorithm:

Push (S[MAXSTK], element)

1) [check the stack is filled ?]
   if top = MAXSTK-1
2) Print overflow
3) else
   set top = top+1
4) set S[top] = element
   ends of if structure
5) exit.

For Pop operation :-

'C' code

```
void pop(int element)
{
  if(top==-1)
    {
      Pf("underflow");
        getch();
        exit(0);
    }
else
    { element = S[top];
      top = top - 1;
      Pf("%d element is deleted", element);
    }
}
```

Algorithm

S[MAXSTK]
Pop (element)

1) [Is stack empty ?]
   if top = -1
2) Write/Print underflow
3) else
   set element = S[top]
4) ~~Print~~ set top = top-1
5) Write/Print element
6) Exit

Menu driven program for Stack

```
#include<stdio.h>
#include<conio.h>
#include<Process.h>
#define MAXSTK 10
int S[MAXSTK], top=-1;
void push(); void display();
void pop(); void main()
void main()
{ int choice;

clrscr();
while(1)
{
clrscr();                                    3: Display \n
pf("1. PUSH \n 2. POP \n 4. EXIT \n others");
pf("\nEnter choice");
sf("%d", &choice);
switch(choice)
{
case 1: push();
        break;
case 2: pop();
        break;
case 3: display(); break;
case 4: exit(0);
        break;
default: pf("default choice");
}
}
getch();
}
```

```
void push()
{
  int element;
  ~~pf("Enter element to be inserted");~~
  ~~sf("%d",&element);~~
  if (top == MAXSTK-1)
    {
      pf("overflow");
      getch();
      exit(0);
    }
else
    {
      pf("Enter element to be inserted");
      sf("%d",&element);
        top = top+1;
          S[top] = element;
    }
  }
void pop()
  {
    int element;
    if(top == -1)
      {
        pf("underflow");
          getch();
          exit(0);
      }
else
    {
      element = S[top];
        top = top-1;
      pf("%d element is deleted", element);
    }
}
```

```
void display()
{
  int i;
  Pf("Elements in stack");
  for(i=top; i>=0; i--)
    Pf("%d\n", s[i]);
}
```

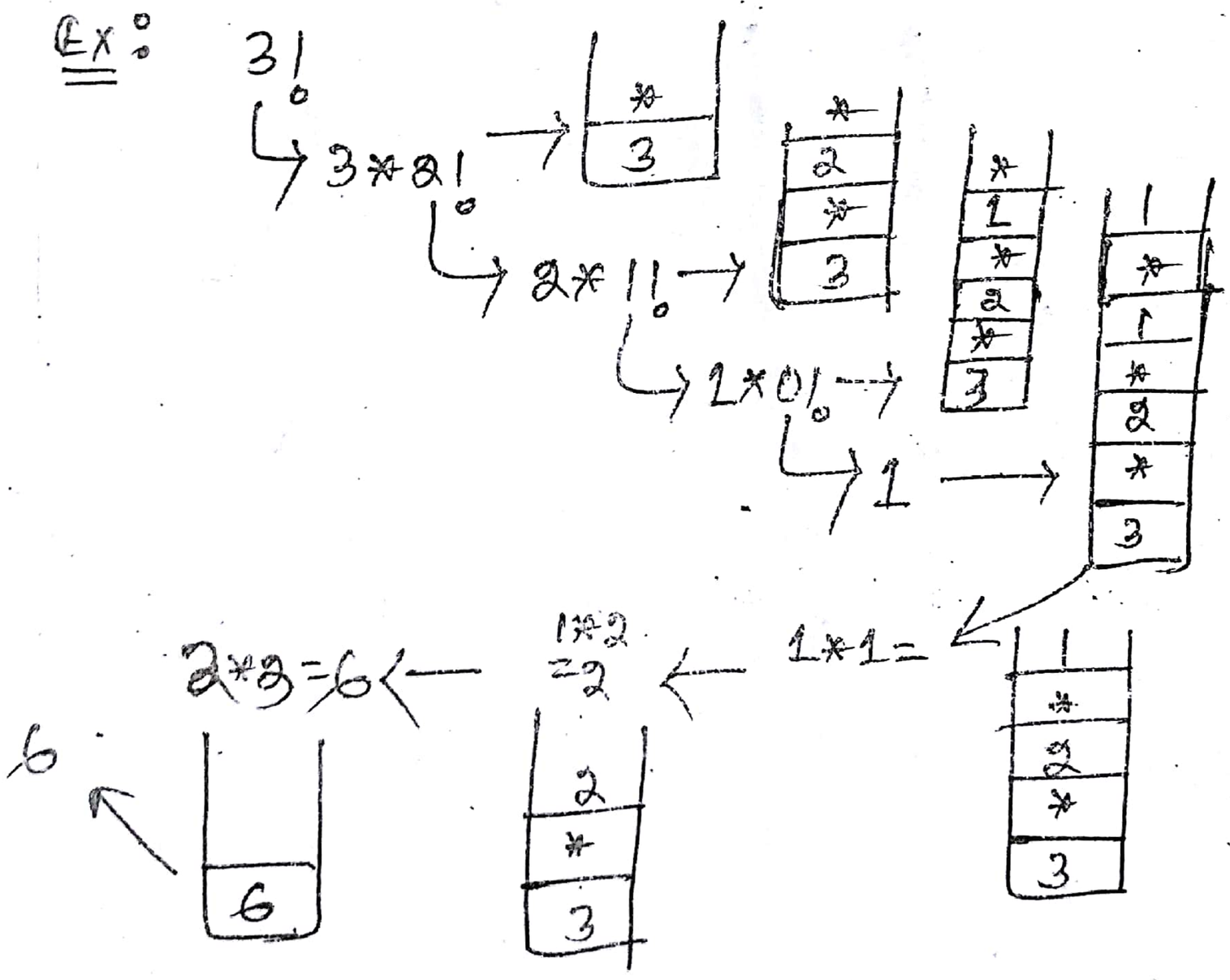
Application of stack:-

A stack uses LIFO principle, it is an appropriate DS for application in which information must be saved and later retrieved in reverse order.

a) Implementation of recursion
b) Evaluation of arithmetic expression.

a) Implementation of recursion :-

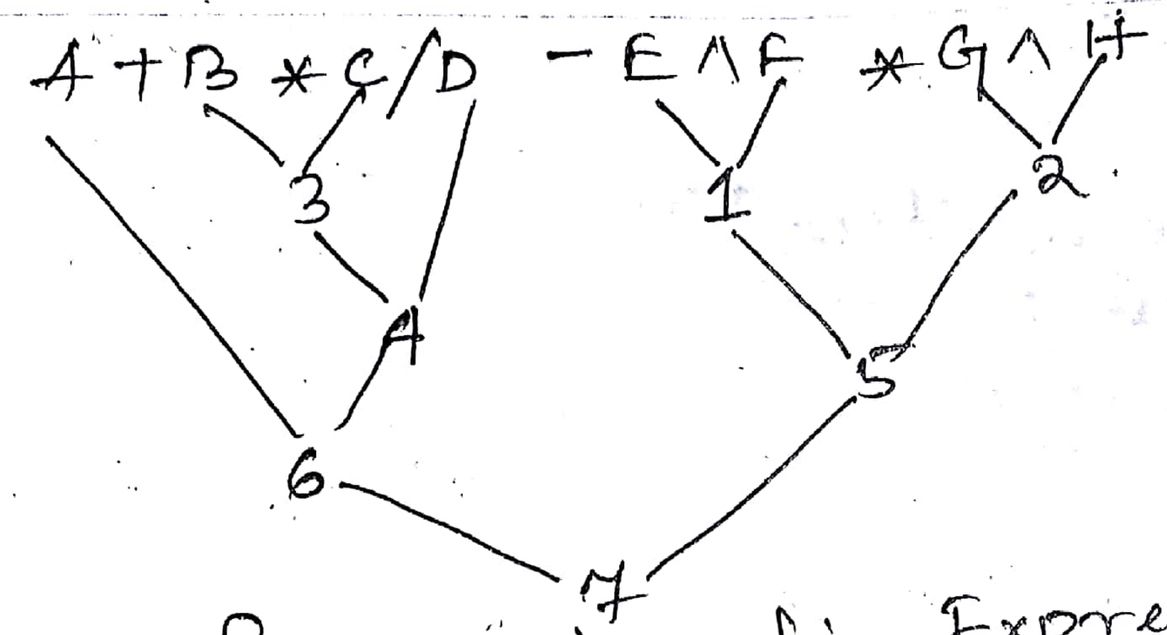
Ex:-



b) Evaluation of arithmetic expression :-

- Arithmetic exp^n consists of operands & operators
- operator may be unary operator, binary operators, relational operator, logical operators

Ex-: A+B*C/D - E^F *G^H



Notation for arithmetic Expression -:

1) Infix -: <operand> <operater> <operand>
Ex-: A+B , C-D , E*F

2) Prefix -: <operator> <operand> <operand>
Ex-: +AB , -CD , *EF

3) postfix -: or reverse polish notation <operand> <operand> <operater>
Ex-: AB+ , CD- , EF*

Converting infix exp^n to postfix exp^n -: (by manually)

A+[(B+C) + (D+E)*F)/G

= A+((BC+) + (D+E)*F)/G

= A+ ((BC+) + (DE+)*F)/G

= A+ ((BC+) + (DE+F*))/G

= A+ (BC+DE+F*+)/G

= A+ (BC+DE+F*+G/)

= ABC+DE+F*+G/+

Algorithm for converting infix to postfix by using Stack.

Algo - Postfix(Q,P)

Q is an arithmetic exp^n written in infix notation. This Algorithm finds the equivalent postfix exp^n P.

1) Push "(" on to stack, and add ")" to exp^n.
2) Scan Q from left to right and repeat step 3 to 6 for each element of Q till the stack is empty.

3) If an operand is encountered, add it to expn P.

4) If a left parenthesis is encountered, push it on to stack

5) If an operator (*) is encountered, then

a) Add (*) to stack

b) Repeatedly pop from stack and add each operator (on the top of stack) to P which has same precedence as or heigher precedence than (*)

6) If a right parenthesis is encountered then Repeatedly pop from stack and add to P each operator on the top of the stack till the first left parenthesis from right to left. (to P)
(don't add the left parenthesis to P)

[end of if structure]

[end of step 2 loop]

7) Exit.

Ex:-

Q = A + (B * C - (D / E ^ F) * G) * H

Soln

Q = A + (B * C - (D / E ^ F) * G) * H)

Scanned Symbol	Stack	Expn (P)
	(	
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^/G
G	(+(—*	ABC*DEF^/G*
)	(+	ABC*DEF^/G**-
*	(+*	ABC*DEF^/G*-H
H	(+*	ABC*DEF^/G*-H*
)		ABC*DEF^/G*-H*+

P = ABC*DEF^/G*-H*+

Ex 2 :

Q = A*B+C/(D+E-F)/G^H-I

Sol^n :

Q = A*B+C/(D+E-F)/G^H-I)

Scanned Symbol	Stack	Exp^n (P)
(	(	
A	(	A
*	(*	A
B	(*	AB
+	(+	AB*
C	(+	AB*C

/	(+/	AB*C
(	(+/(	AB*C
D	C+/(	AB*CD
+	(+/C+	AB*CD
E	C+/C+	AB*CDE
−	C+/C−	AB*CDE+
F	(+/(−	AB*CDE+F
)	(+/	AB*CDE+F−
/	(+/	AB*CDE+F−/
G	C+/	AB*CDE+F−/G
^	(+/^	AB*CDE+F−/G
H	(+/^	AB*CDE+F−/GH
^	(−	AB*CDE+F−/GH^/+
I	(−	AB*CDE+F−/GH^/+I
)		AB*CDE+F−/GH^/+I

Q) Q = A*B+C−D/E+(F+G−H/I)+J

Ans P = AB*C+DE/−FG+HI/−+J+

Q) A+((B+C)+(D+E)*F])/G

Scanned Symbol	Stack	Exp?
	(	
A	(	A

+	(+	A
(	(+(	A
(	(+((	A
B	(+((	AB
+	(+((+	AB
C	(+((+	ABC
)	(+(·	ABC+
+	(+(+	ABC+
(	(+(+(	ABC+
D	(+(+(	ABC+D
+	(+(+(+	ABC+D
E	(+(+(+	ABC+DE
)	(+(·+	ABC+DE+
*	(+(+*	ABC+DE+
F	(+(+*	ABC+DE+F
)	(+	ABC+DE+F*+
/	(+/	ABC+DE+F*+
G	(+/	ABC+DE+F*+G
)	-	ABC+DE+F*+G/+

Ex Q = A+B*C OR D *E+F ANDG

Scanned Symbol	Stack	exp^n
	(	
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
C	(+*	ABC
OR	(OR	ABC*+
D	(OR	ABC*+D

*	(OR *	ABC*+D
E	(OR *	ABC*+DE
+	(OR +	ABC*+DE*
F	(OR +	ABC*+DE*F
AND	(OR AND	ABC*+DE*F+
G	(OR AND	ABC*+DE*F+G
)		ABC*+DE*F+G AND OR

Convertion of INFIX exp^n to PREFIX exp^n :-

A+B*C = A+(*BC)

= +A*BC

Algorithm:

1) Reverse the i/p string and add ")" to the end of Q
2) push "(" on to stack
3) Scan from left to right repeat step 4 to 7
4) if it is operand, add it to the o/p string.
5) if it is opening parentheses "(", Push it on the stack
6) If it is an operator then

   i) if it has same or heigher priority than the operator at top of stack then push operation it Done.

   ii) else pop the operator from the stack and add it exp^n.

7) If it is a closing parenthesis / Pop operator from stack and add them to exp^n till 1st opening parenthesis is encountered (from right to left) & discards the parenthesis.
8) Reverse the o/p string.
9) Exit.

$Q = A + (B * C - (D / E \wedge F) * G) * H$

A) Reverse the Q

$Q = H * (G * (F \wedge E / D) - C * B) + A$

Scanned Symbol	Stack	Exp^n
H	(	
*	(	H
G	(*	H
*	(* (	H G
(	(* ((	H G
F	(* (* (	H G F
^	(* (* (^	H G F
E	(* (* (^	H G F E
/	(* (* (/	H G F E ^
D	(* (* (/	H G F E ^ D
)	(* (*	H G F E ^ D /
-	(* (-	H G F E ^ D / *
C	(* (-	H G F E ^ D / * C
*	(* (- *	H G F E ^ D / * C
B	(* (- *	H G F E ^ D / * C B
)	(*	H G F E ^ D / * C B * -
+	(+	H G F E ^ D / * C B * - *
A	(+	H G F E ^ D / * C B * - * A
)		H G F E ^ D / * C B * - * A +

Reverse the exp^n.

+ A * - * B C * / D ^ E F G H

Q) Q=A*B+C/(D+E-F)/G^H-I convert infix to prefix & postfix.

Post :- AB*CDE+F-/GH^/+I-

Pre :- -+*AB/C-+DEF^GHI

Q) WAP to reverse the string by using stack

```
#define MAXSTK 10
int top = -1;
char stack[MAXSTK];
char pop();
void push(char);
void main
{ char str[20];
int i;
pf("enter string");
gets(str);
for(i=0; i<strlen(str); i++)
    push(str[i]);
for(i=0; i<strlen(str); i++)
  { str[i] = pop(); }
Pf("%s", str);
getch();
}
void push(char ele)
{ if (top == MAXSTK-1)
  { pf("overflow");
    getch();
    exit(0);
  }
else { top = top+1;
stack[top] = ele;
  }
}
char pop()
{ char e;
if (top == -1)
  { pf("underflow");
    getch();
    exit(0);
  }
else
  { e = stack[top];
    top = top-1;
  }
return(e);
}
```

Raise

Postfix Evaluation

Algorithm to do the postfix evaluation :-

This algorithm finds the value of an arithmetic expn written in postfix notation. The following algorithm which uses a stack to hold operands & evaluates P.

1) Add a ")" at the end of P.
2) scan P from left to right and repeat step 3 & 4 for each element of P until ")" is encountered.
3) If an operand is encountered, push it onto stack
4) If an operator (*) is encountered, then
   a) Remove the two top elements of the stack if A is the top element & B is the next to top element
   b) evaluate B (*) A
   c) Place / Push the result back on stack.

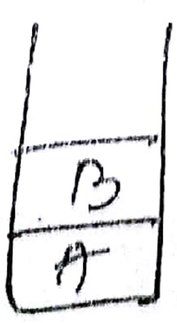
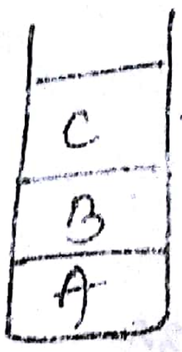
   [end of if structure]

   [end of step2 loop]
5) Set value equal to top element on stack.
6) Exit.

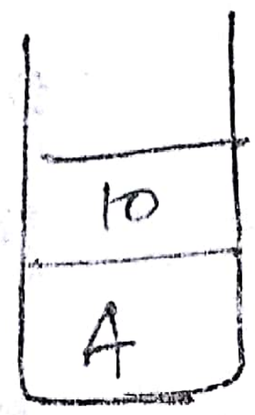
Q) Q = A+(B*C-(D/E^F)*G)*H

P = ABC*DEF^/G*-H*+

A=4, B=2, C=5, D=2, E=2, F=1, G=5, H=3

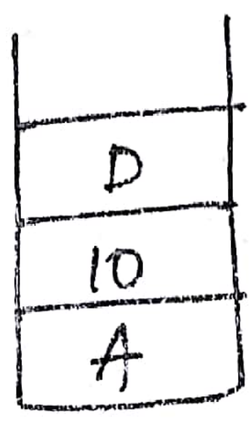
Scanned symbol	Stack	operation
A		Push(A)
B		Push(B)
C		Push(C)

*



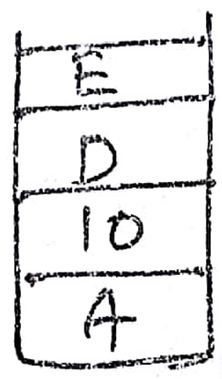
Pop ⇒ C
Pop ⇒ B
B * C = 2 * 5 = 10
Push 10

D



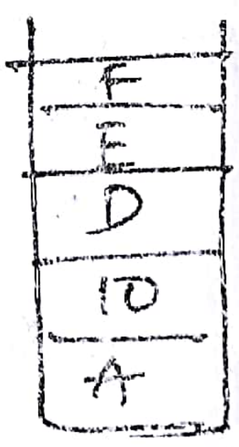
Push(D)

E



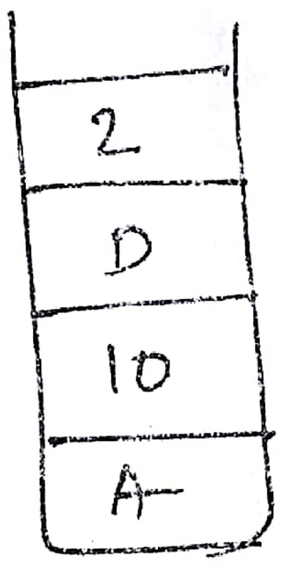
Push(E)

F



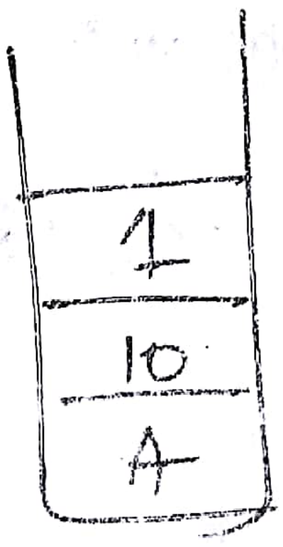
Push(F)

^



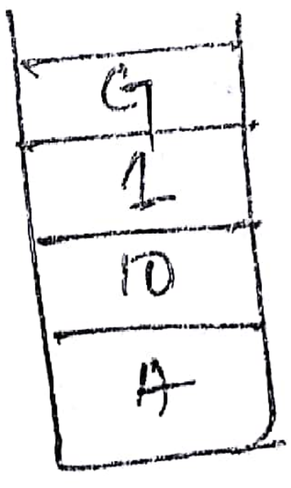
Pop → F
Pop → E
E ^ F = 2 ^ 1 = 2
Push(2)

/



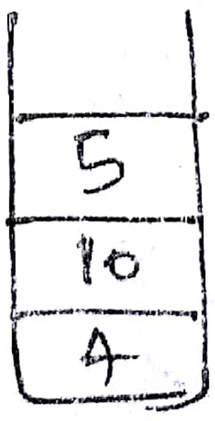
Pop → 2
Pop → D
D / 2 = 2 / 2 = 1
Push(1)

G



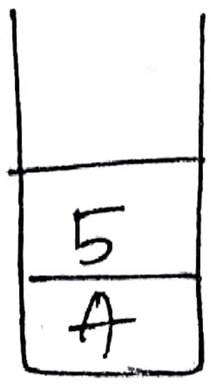
push(G)

*



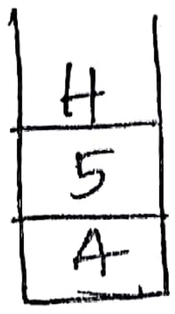
Pop → G
Pop → 1
1 * G = 1 * 5 = 5

—



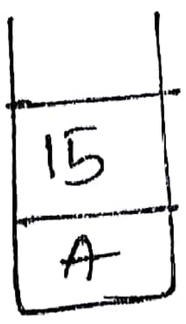
pop → 5
pop → 10
10-5=5
-push(5)

H



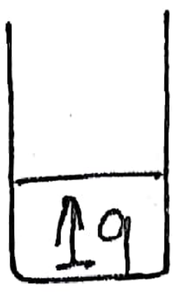
push(H)

*



pop → H
pop → 5
5*H = 5*3 = 15

†



pop → 15
pop → A
A+15 = 4+15 = 19
push 19

)

operation completed.

Prefix evaluation:

1→ Add a left parenthesis
2→ Scan' right to left and repeat step 3 & 4
3→ If an operand is encountered, push it onto stack.
4→ If an operator (*) is encountered, then
   a) Remove the two top elements of the stack if A is the top element & B is the next to top element
   b) evaluate A (*) B
   c) push the result to stack.
   [end of if structure]
[end of step 2 loop.]

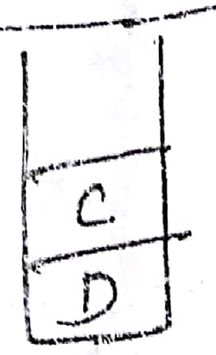
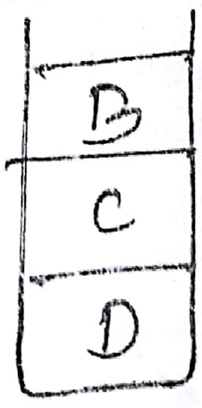
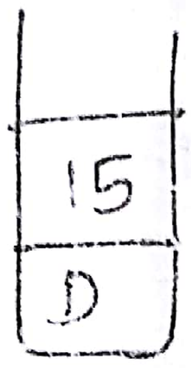
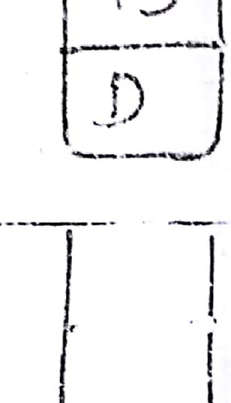
5) Set the value equal to top element

6) Exit.

Q) Q = A + B * C / D

$A = 5$
$B = 3$
$C = 5, D = 5$

Prefix = +A/*BCD

Scanned Symbol	Stack	operation
D		Push(D)
C		Push(C)
B		Push(B)
*		Pop → B Pop → C B * C = 3 * 5 = 15
/		Pop → 15 Pop → D 15/D = 15/5 = 3
A		~~Push(B)~~ Push(A)
+		Pop ~~⇒ A~~ Pop ⇒ 3 A + 3 = 5 + 3 = 8 Push(8)
(		Operation completed.