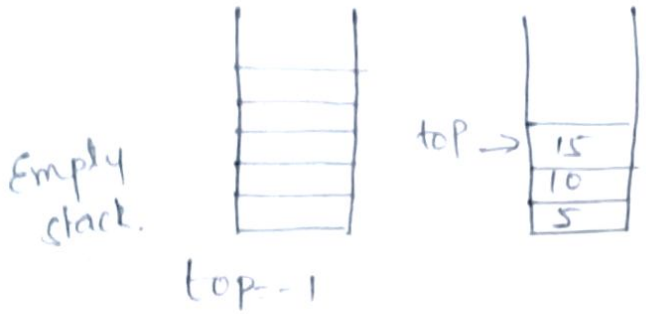
Stack→ • Stack is logically a last in First out LIFO type of list.
• Stack is a non premitive linear data structure.
• Stack is a homogeneous collection of element in which insertion and deletion is done from stack.



Representation of stack→

(i) Array representation→

The stack is maintained by a linear array STACK, a variable top which contains the index of top element.

Underflow condition

top == -1

Overflow condition

top == N-1

where N is the size of stack.

There are two operations in array

(a) push() → To insert the element in a top of the stack.

b) pop() → To delete top element from stack.

Push() ->

Algorithm

PUSH(STACK[MAXSIZE], TOP, ITEM)

1. if (TOP == MAXSIZE-1) then
   write "Overflow" and exit.
   [END of if]
2. read ITEM
3. Set TOP = TOP+1
4. Set STACK[TOP] = ITEM
5. Exit

Pop() -> Algorithm ->

POP(STACK[MAXSIZE], TOP, ITEM)

1. if (TOP == -1) then
   write "Underflow" and exit
2. ITEM = STACK[TOP]
3. TOP = TOP-1
4. write ITEM " is deleted"
5. Exit.

C function->

```
int top=-1, a[30];
void push()
{
  int item;
  if(top==29)
  {
    printf("overflow");
    exit(0);
  }
  printf("Enter item");
  scanf("%d", &item);
  a[++top]=item;
}
void pop()
{
  int item;
  if(top==-1)
  {
    printf("Underflow");
    exit(0);
  }
  item=a[top--];
  printf("%d is deleted", item);
}
void main()
{
  push();
  push();
  pop();
```

Traversing →

Display(STACK[MAXSIZE], TOP, ITEM)

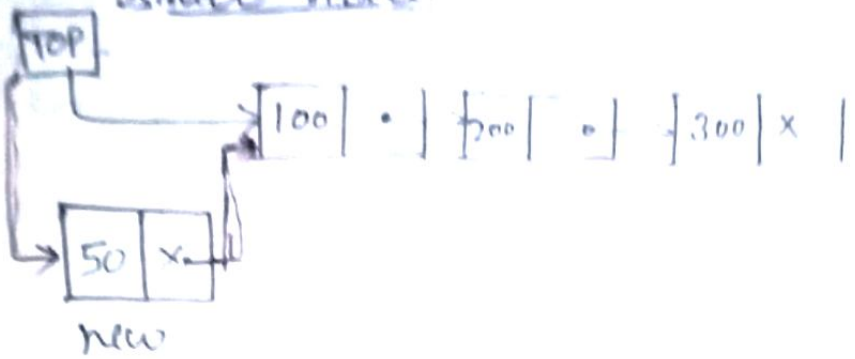
1. If TOP = = -1 then
   write "STACK is empty" & exit
2. Set I = TOP,
3. Repeat step 4 & 5 while I >= 0
4. Display STACK[I]
5. I = I - 1
6. Exit

C function →

```
void display()
{
    int i;
    if (top == -1)
    {
        printf("stack is empty");
        exit(0);
    }
    for (i = top; i >= 0; i--)
        printf("%d", a[i]);
}
```

2) Implementation of stack using linked list (5)

struct node



push → Algorithm

PUSH(STACK, TOP, INFO, NEW)

push( )

1. new = AVAIL
2. if NEW = NULL then
   write "Overflow" & exit
3. INFO[new] = item
4. LINK[new] = top
5. top = new
6. Exit

pop( ) → Algorithm

pop( )

1. if (top = NULL) then
   write "Underflow" & exit
2. ptr = top
3. Write INFO[ptr] "is deleted
4. top = LINK[top]
5. free(PTR) C. Exit.

display() →

display()

1. if (top = NULL) then
   write "stack is empty" & Exit
2. ptr = top
3. repeat step 4 & 5 till ptr != NULL
4. write INFO[ptr]
5. ptr = LINK[PTR]
6. Exit.

c function

~~int a[30], top=-1;~~

~~Void push()~~

```
{
    struct node
    {
      int info;
    struct node *next;
    } *top = NULL;

void push()
{
    struct node *new;
new = (struct node *) malloc(sizeof(struct node));
    printf("Enter node info");
    scanf("%d", &new->info);
      new->next = NULL;
      if (new == NULL)
          printf("overflow");
       else
         {
```

```
            new->next = top;
              top = new;
        }
    }

void pop()
{
    struct node *p;
    if (top == NULL)
        printf("Underflow");
    else
    {
        p = top;
        printf("%d is deleted", p->info);
        top = top->next;
        free(p);
    }
}

void display()
{
    struct node *p;
    if (top == NULL)
        printf("Empty Stack");
    else
    {
        for (p = top; p != NULL; p = p->next)
            printf("%d", p->info);
    }
}
```

Application of stack →

(1) Undo in text editor
(2) Web browser history
(3) Conversion of infix expression to post fix and prefix form
(4) Evaluation of postfix and prefix form.
(5) Checking the validity of an expression.

Notation for Arithmetic Expression –

There are basically three types of notations

(i) Infix notation
(ii) Prefix notation
(iii) Postfix notation

Infix notation → In this notation, operator is placed between two operands.

for example

A+B, C-D etc.

Prefix notation → In this notation, operator is placed before the operands.

Ex → +AB, -CD

post fix notation → In this notation, operator is placed after the operands.

AB+, CD-

operator precedence →

Exponential operator
multiplication, Division
subtraction, addition

Polish notation :- In which operator is placed after its two operands (postfix)

Reverse polish notation → In which operator is placed before its two operands. (prefix)

Convert Infix to postfix form-

1) Direct method (Inspection and hand)

Q:- Convert the expression (A+B)/(C-D) to postfix form

Ans- (A+B)/(C-D)

= AB+/(C-D)

= AB+/CD-

= AB+CD-/

Q:- Convert in the postfix form

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

Sol:- A+(B*C-(D/E^F)*G)*H

= A+(B*C-(D/EF^)*G)*H

= A+(B*C-(DEF^/)*G)*H

= A+(BC*-DEF^/*G)*H

= A+(BCBC*-DEF^/G*)*H

= A+(BC*DEF^/G*-)*H

= A+BC*DEF^/G*-H*

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

2) Using stack → PUSH (, P)

Let Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto stack and add ")" to the end of Q.
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the stack is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis (is encountered, push it onto STACK.
5. If an operator ⊗ is encountered then:
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has same precedance or higher precedance than ⊗.
   b) Add ⊗ to STACK.
6. If a right paranthesis is encounterd then:
   a) Repeatedly pop from STACK and add to P each operation (on top of the STACK) until a left paranthesis is encountered.
   b) remove the left paranthesis.
7. Exit

Q:→ convert the expression

Q: ((A+B)*D)↑(E-F)

into Postfix expression.

Q. Convert the following infix to postfix expression (11)

Q: A + (B * C - (D / E ↑ F) * G) * H)

Symbol Scanned	STACK	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	(+(-*	ABC*DEF↑/G
)	(+	ABC*DEF↑/G*-
*	(+*	ABC*DEF↑/G*-
H	(+*	ABC*DEF↑/G*-H
)	Empty	ABC*DEF↑/G*-H*+

Evaluation of postfix notation →

This Algorithm finds the value of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the ") is encountered.
3. If an operand is encountered, put it on stack.
4. If an operator is encountered then
   a) remove the top two elements of STACK, where A is the top element and B is the next to top element.
   b) Evaluate B ⊗ A
   c) place the result of (b) back on STACK.
5. Set value equal to the top element of ~~stac~~ STACK.
6. Exit.

Q:- Evaluate the following postfix expression.

P: 5,6,2,+,*,12,4,/,-)

Symbol scanned	Stack
5	5
6	5,6
2	5,6,2
+	5,8
*	40
12	40,12
4	40,12,4
/	40,3
-	37
)	

Q. Consider the postfix expression P

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +,)

Evaluate the postfix expression.

Ans.

	Symbol	STACK
	12	12
	7	12,7
	3	12,7,3
	−	12,4
	/	3
	2	3,2
	1	3,2,1
	5	3,2,1,5
	+	3,2,6
	*	3,12
	+	15
	)	15

Convert infix to prefix form –

1) Direct method →

A/B^C−D

= A/^BC−D

= /A^BC−D

= −/A^BCD

Q1:- (A−B)/((D+E)*F)

= (A−B)/+DE*F

= −AB/+DE*F

= /−AB+DE*F

= */−AB+DEF

2) By stack →

1. Reverse the input string or start from right of the infix expression.
2. Examine the next element in the input
3. If it is operand, add it to output string.
4. If it is closing paranthesis, push it on stack.
5. if it is an operator then
   a) If stack is empty, push operator on stack.
   b) if the top of the stack is), push operator on stack
   c) If it has same or higher priority than top of the stack, push operator on stack.
   d) Else pop the operator from the stack and add it to output string. repeat step 5.
6. If it is a "(", pop operators from stack and add item to output string, until a closing paranthesis is encountered. pop and discard the) (,
7. If there is more input goto step 2.
8. If there is no more input, unstack the remaining operators and add them to output string.
9. Reverse the output string.

Q:- Convert into prefix form (15)

2 * 3 / (2 - 1) + 5 * (4 - 1)

ibol inned	Stack	Prefix Expression
)	)	
1	)	1
−	)−	1
4	)−	1 4
(	empty	14−
*	*	14−
5	*	14−5
+	+	14−5*
)	+)	14−5*
1	+)	14−5*1
−	+)−	14−5*1
2	+)−	14−5*12
(	+	14−5*12−
/	+/	14−5*12−
3	+/	14−5*12−3
*	+/*	14−5*12−3
2	+/*	14−5*12−32
	empty	14−5*12−32*/+
	now reverse the output string.	
	prefix expression:	+/*23−21*5−41

Q1. consider the following postfix notation- (16)

A: 12, 6, 3, −, /, 2, 1, 5, +, *, +

a) Translate A by inspection and hand into equivalent infix expression.

b) Evaluate the infix Expression.

Soln a) A = 12, 6, 3, −, /, 2, 1, 5, +, *, +

= 12, (6−3), /, 2, (1+5), *, +

= [12/(6−3)), (2*(1+5)), +

= (12/(6−3)) + (2*(1+5))

b) A = (12/(6−3)) + (2*(1+5))

= 12/3 + 2×6

= 4 + 12

= 16

Checking validity of an expression containing nested parantheses:

Take a boolean variable valid which will be true if the expression is valid and will be false if the expression is invalid. Initially valid is true.

1) Scan the symbols of expression from left to right and repeat step 2 and 3 for each symbol.
2. If the symbol is left paranthesis then push it into a STACK.
3. if the symbol is right paranthesis then
   a) if the stack is empty then valid = false
   b) else pop an element from stack
      If Popped paranthesis doesnot match the paranthesis being scanned then valid = false
4. After scanning all the symbols if the stack is not empty then make valid = false
5. check the value of valid if true then the expression is valid else invalid
6. Exit.

Q: Check the validity of an expression

((A+B)-(C+D))-(F+G)

Scan symbol	stack	valid=true
(	(	
(	((	
A	((	
+	((	
B	((	
)	(	
-	(	
(	((	
C	((	
+	((	
D	((	
)	(	
)	empty	
-	empty	
(	(	
F	(	
+	(	
G	(	
)	empty	

valid = True

So expression is valid.

Q:- Check the validity of an expression

(a +(b + c)))

Scan symbol	STACK	valid=true
(	(	
+	(	
(	((	
b	((	
+	((	
c	((	
)	(	
)	empty	
)	✱	valid=false

So the expression is invalid.