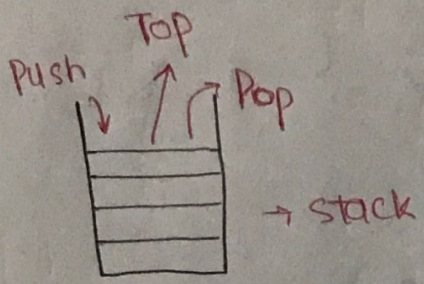
→ Stack:

- Linear data structure
- Ordered List
- It follow the rule for insertion and deletion. The rule is insertion or deletion only possible from one end.
- It follow the mechanism of LIFO (Last in First out).
- Insertion operation on stack is known as PUSH.
- Deletion operation on stack is known as POP.
- Only one end is open in stack for insertion and deletion and it known as TOP.
- The real-time example is stack of books, etc.



• one open end only

Operations:

- Push() → for insertion on stack
- Pop() → for delete the top element of stack
- Peek() | Top() → for display only the top element.
- display() → for all the element on the stack.
- Many more operations Like traversing, searching, sorting, Maximum, minimum etc.

→ There are two ways to implement stack.

1. Static Memory → using Arrays
2. Dynamic Memory → using Linked List.

OverFlow:

It is condition simply means occur when we exceed the maximum size.

Under Flow:

Occur when we access the elements but our stack is empty.

Applications:

There are many applications of stack some are:

- Reverse a string
- Undo mechanism
- Recursion / Function call
- Balance of parenthesis { → link { } } }
- Infix to Postfix/Prefix

Implementation using Arrays:

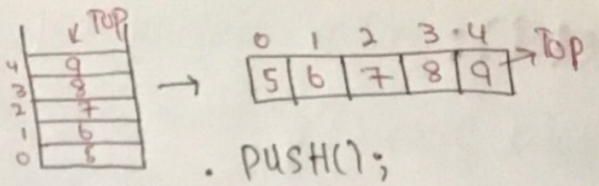
Write this code before writing the function of stack.

```
#include <iostream>
#define N 5
using namespace std;
int stack[N];
int top=-1;
```

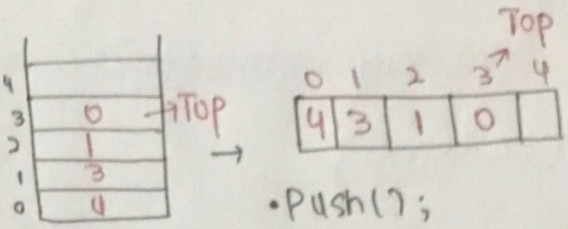
// Top == -1 because index of array start from 0.

PUSH(): → For insertion

```
void push() {
 if (top==N-1) {
  cout<< "OVERFLOW";
 }
 else {
      top ++;
 int i; // for value
  cin>>i;
stack[top] = i;
}
  }
```

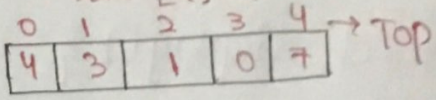


. PUSH();
top==N-1
4==5-1
4=4
OVERFLOW



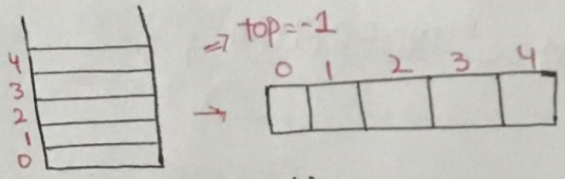
•Push();
top==N-1
3==4 X

else top++ = 3→4
cin>> 7.
stack[4]=7.

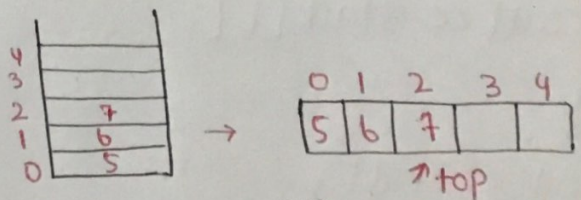


POP(): → For Deletion

```
void pop() {
  if (top==-1) {
  cout<<"UnderFlow";
}
else {
  int temp= stack[Top];
    top--;
cout<<temp;
}
  }
```

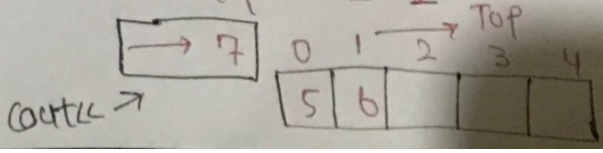


•pop();
top==-1
-1==-1
UNDERFLOW



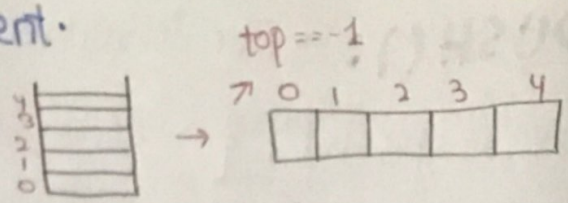
•pop();
2==-1 X

else
top-- = 1

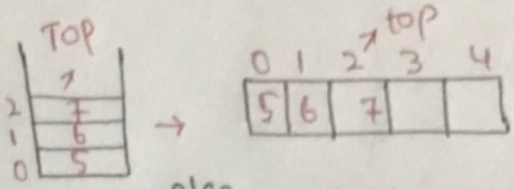


Peek(): → For display top element.

```
void peek(){
 if(top == -1){
 cout<<"UNDERFLOW";
}else{
      int temp = stack[top];
cout<<temp;
}
 }
```



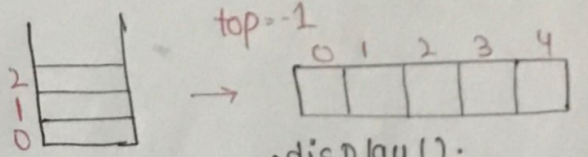
•peek();
-1 == -1
UNDERFLOW



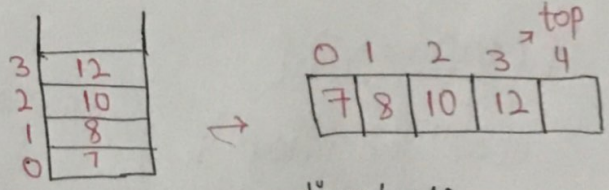
else
stack[top];
→ 7

DISPLAY(): → For display all elements of stack.

```
 void display(){
if(top==-1){
cout<<"UNDERFLOW";
}
else{
cout<<"Elements is ";
for(int i=Top;i>=0;i--){
   cout<<stack[i];
  }
  cout<<endl;
}
```



•display();
-1 == -1
UNDERFLOW



•display();

else
loop → i=top=3
3>=0 ✓
stack[3] → 12 → i--
2≥0 ✓
stack[2] → 10 → i--
1>=0
stack[1] → 8 → i--
0>=0
stack[0] → 7 → i--
→ END. -1≥0 ✗

Using Linked List:

- We know that stack follow the LIFO Mechanism.
- In Linked List we have two ways to represent the stack, one is insertion and deletion in the last and the second is insertion and deletion in the start.
- Take dynamic Memory means on run time.

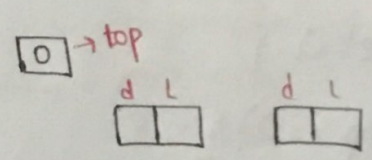
→ Write this code before write the functions.

```
#include<iostream>
using namespace std;
struct node {
    int data;
    Struct node *link;
};
struct node *top=0;
```

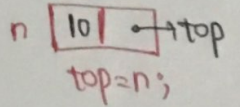
NOTE: This is the implementation of push and pop in the first bcz it has O(1) time complexity.

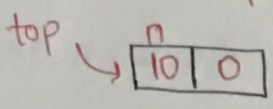
PUSH (): → For insertion

```
void push (){
struct node *n= new node();
  int x;
cout << "Enter value";
  cin>>x;
n -> data = x;
n -> link = top;
  top =n;
}
```

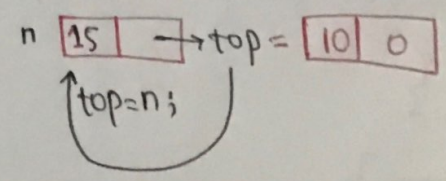


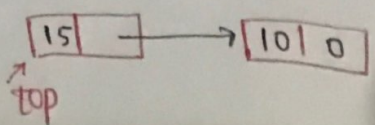
- push();
  cin>>x → 10





- push();
  cin>>x → 15

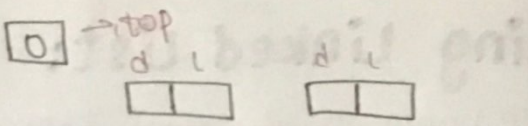




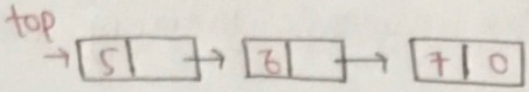
void display() {

POP(): → For Deletion

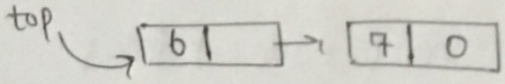
```
void pop() {
if (top == 0) {
   cout<< "UNDERFLOW";
 }
else {
  struct node *temp = new node();
     temp = top;
   top = top -> link;
  cout<< temp -> data;
  delete temp;
}
  }
```



•pop();
top == 0
0 == 0
UNDERFLOW

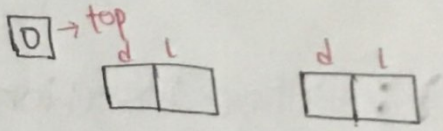


•pop();
else temp•
temp = top•
top → top → link (next)
delete temp;

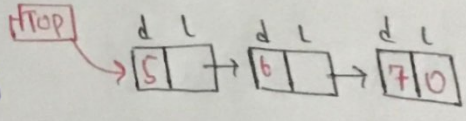


PEEK(): → For display only top element.

```
void peek() {
  if (top == 0) {
cout<< "UNDERFLOW";
 }
  else {
struct node *temp = new node();
   temp = top;
cout<< temp -> data;
 }
   }
```



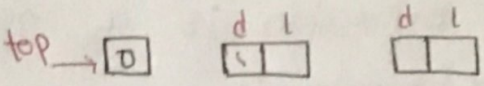
•peek();
top == 0
0 == 0
UNDERFLOW



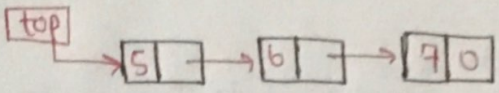
•peek();
top == 0 ✗
else temp = top
cout<< temp -> data
→ 5

Display(): → For display all elements in the stack.

```
void display () {
 if ( top == 0) {
 cout << "UNDERFLOW";
 } else {
cout << "Stack is ";
Struct node *temp = new node();
    temp = top;
 While (temp != 0) {
  cout << temp → data;
 temp = temp -> link;
 }
   }
     }
```



display();
top == 0
0 == 0
UNDERFLOW.



display ();
top == 0 X

else
temp = top.

Loop → (temp != 0)

✓ temp → data = 5
temp → link.

✓ temp → data = 6
temp → link

✓ temp → data = 7
temp → link.

X → Exit

Expression:

Expression is a container of constants, operators, operands and parenthesis.

Example:- P + q + 5 → It is it self an expression.

P → coefficient / constant

q → operands

5 → operands

+ → operators

syntax:-

<operand> <operator> <operand>

→ This is binary operator because talk about two operands.

→ This in known as Infix Expression.

Example:-

5 + 1 * 6

→ 6 * 6 = 36

→ 5 + 6 = 11

only one answer is correct. To solve the expression like this we follow the some rules and precedence of operators.

① . ( ) { } [ ]

② . ^ → Right to left

③ . * / → Left to Right

④ . + - → " "

Example:-

1 + 2 * 5 + 30 / 5

1 + 10 + 30 / 5

1 + 10 + 6

= 17

2 ^ 2 ^ 3

Right to Left

2 ^ 3 = 8

2 ^ 8

= 256

Prefix:

- It is also known as Polish notation.
- Represent as <operator> <operands> <operands>
- Prefix means operators is before to operands like.

5 + 1 → + 5 1

a * b + c → * ab + c → + * ab c → For this expression no need any rules.

Postfix:

- It is also known as Reverse Polish Notation.
- Represent as <operand> <operand> <operator>
- Postfix means operator is after to operands like

5 + 1 → 5 1 +

a * b + c → ab * + c → ab * c + → For this expression no need any rules.

→ Human can easily understand Infix expression but the computer take very time to solve the Infix expressions but the computer can easily solve the postfix and prefix expressions that's why we first convert the Infix to prefix and postfix. (we → computer.)

Infix to Postfix Conversion:

→ using stack we convert Infix to postfix.

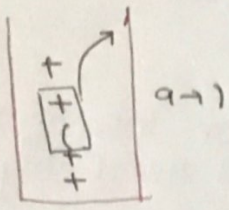
- one by one give number to each element in the expression.
- If the operands are coming move to the expression.
- If the stack is empty or left parenthesis on the top operators are push to stack until Right parenthesis.
- Left parenthesis is coming push to the stack.
- Right parenthesis is coming pop all the operators of stack to expression until left parenthesis.

Example:-

- simply means operator more to stack.

2 + 5 + (6 + 7) + 10
1 2 3 4 5 6 7 8 9 10 11

Stack:-



1→2
2→+
3→5
4→+
5→(
6→6
7→+
8→7
9→)
10→+
11→10

Expression:-
2 5 6 7 + 10 + + +

- If incoming symbol is higher precedence than top of stack so it can push to the stack.
- If incoming symbol is lower precedence that top of stack so, pop the top of stack to the expression and then again check precedence.
- If the incoming operator is same precedence than top of the stack so use **associativity Rule.**

Left to Right
→ First pop the top of the stack and then push the incoming

Right to left
→ directly push the incoming operator to stack.

- At the end pop all the operators of stack to expression.

Example:-

A - B / C * D + E
1 2 3 4 5 6 7 8 9

Stack	Expression
	A
-	A B
- /	A B C
- *	A B C /
+	A B C / D
↓	A B C / D * -
	A B C / D * - E +

Example:-

Infix to Postfix using Stack

K + L - M * N + (O ^ P) * W / U

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Incoming	Stack	Postfix Expression
K		K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M
N	-*	KL+MN
+	+	KL+MN*-
(	+(	KL+MN*-
O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^
W	+*	KL+MN*-OP^W
/	+/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U

After pop one by one stack elements to expression.

KL+MN*-OP^W*U/+

Infix to Prefix using Stack:

Prefix → before

without stack → A + B * C

= A + *BC → one operand.

= +A *BC

→ First convert the Infix Expression to reverse form and then solve after solving again reverse.

K + L - M * N + (O ^ P) * W / U / V * T + Q

↓

Q + T * V / U / W *) P ^ O (+ N * M - L + K

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Input Exp.	Stack	Prefix Expression
Q		
+	+	Q
T	+	QT
*	+*	QT

→ If incoming operator is higher precedence than top of the stack so simply push to the stack.

V	+*	QTV
/	+*/	QTV

→ If incoming operator is same precedence to the top of the stack check the associativity Rule.

Left to Right → push to the stack.

U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW

→ If incoming operator is closing parenthesis than simply push to stack.

P	++ // *)	@TVUWP
^	++ // *)^	@TVUWP
O	++ // *)^	@TVUWPO

→If incoming operator is opening parenthesis then pop all the elements of stack to expression until closing paren.

(	++ // *	@TVUWPO^
+	++	@TVUWPO^ * // *

→If incoming operator is lower precedence than the top of the stack so simply pop the top of stack to exp. and check again.

N	++	@TVUWPO^ * // * N
*	+++	@TVUWPO^ * // * N
M	+++	@TVUWPO^ * // * NM
-	++-	@TVUWPO^ * // * NM *
L	++ -	@TVUWPO^ * // * NM * L
+	++-+	@TVUWPO^ * // * NM * L
K	++-+	@TVUWPO^ * // * NM * LK

→After end pop the stack operators to expression one by one (using LIFO)

@TVUWPO^ * // * NM * LK + - + +

Now again Reverse

++ - + KL * MN * // * ^ OP WUVT @

→This is the prefix Expression.

Evaluate Prefix Expression:

- For prefix expression scan ~~Left~~ to ~~Right~~ Right to left and find first operation.
- After finding operator use the following sequence and then solve.

<operator> <operand> <operand>

$-+a*bc/d\wedge ef$ ∴ a = 2, b = 3, c = 4. d = 16, e = 2, f = 3

$-+2*34/16\wedge 23$

Right to Left.

$\wedge 23 \rightarrow 2\wedge 3 = 8$

$-+2*34/168$

Right to Left

$/168 \rightarrow 16/8 = 2$

$-+2*342$

Right to left

$*34 \rightarrow 3*4 = 12$

$-+212\ 2$

Right to Left

$+212 \rightarrow 2+12 = 14$

$-14\ 2$

Right to Left

$-14\ 2 \rightarrow 14-2 = 12$

12

Evaluation of Postfix Expression:

- For postfix expression scan from left to right and find first operator after finding operator use the following syntax to solve one by one.

<operand> <operand> <operator>

a b c * + d e f ^ / -

a=2 , b=3 , c=4
d=16 , e=2 , f=3

2 3 4 * + 16 2 3 ^ / -

Left to Right

3 4 * → 3*4 = 12

2 12 + 16 2 3 ^ / -

L to Right

2 12 + → 2+12 = 14

14 16 2 3 ^ / -

L to Right

2 3 ^ → 2^3 = 8

14 16 8 / -

L to Right

16 8 / → 16/8 = 2

14 2 -

L to Right

14 2 - → 14-2 = 12

12