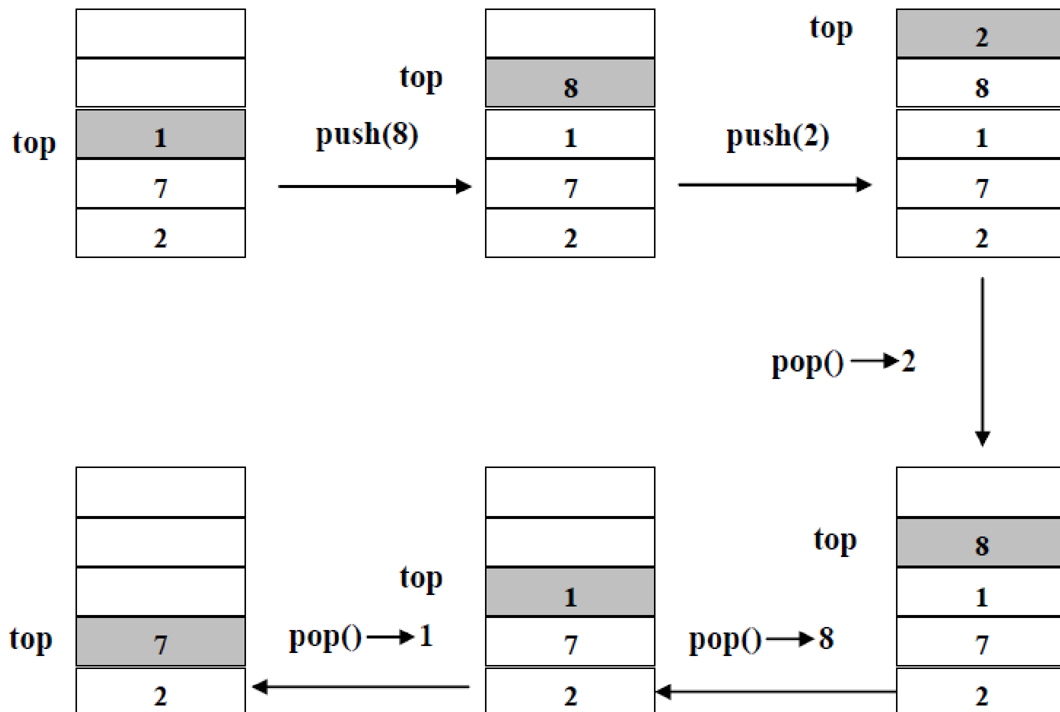
Stack المكدس

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).
Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Top**: Returns top element of stack.
- **Is Empty:** Returns true if stack is empty, else false.



Applications of stack:

1. Infix to Postfix /Prefix Conversion using Stack
2. Check for balanced parentheses in an expression
3. Next Greater Element
4. Expression Evaluation
5. Call of subprogram
6. Use in recursion

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Analysis of Stacks:

There are various operations that can be performed on the Stack data structure:

- **Push Operation**.
- **Pop Operation.**
- **Top Operation**.
- **Search Operation**.

The following program explain **push** and **pop** operations:

```
#include <iostream>

using namespace std;

const int size=10;
int   stack[size];
int   top=-1;

void push(int n)
{
 int item;
 if (top==size-1)
  cout<<"Stack is Overflow:\n";
 else

      for(int j=0;j<n;j++)
      {
       cout<<"Enter Item: \n";
       cin>>item;
       top++;
       stack[top]=item;
      }
}
/*--------------------------------*/

void pop(int n)
{
 int d,j;
 if (top<0)
  cout<<" Stack is UnderFlow: \n";
```

```
 else
    for(j=n;j>0;j--)
     {
      d=stack[top];
      top--;
     }

}
/*------------------------------------*/
void display()
{
 if (top<0)
   cout<<"Stack is Empty:\n";
 else
   for(int i=top;i>=0;i--)
   {
    cout<<"stack["<<i<<"] --> ";
    cout<<stack[i]<<endl;
   }
}
/*-----------------------------------*/
int main()
{
 int item,n;

 cout<<"Enter n: \n";
 cin>>n;

  push(n);
  display();
  cout<<"delete items from stack:\n";
  pop(n);
  display();
  return 0;
}
```

EX1:Write program to find the average of **M** lessons for **N** student.

```
#include<iostream>

 const int n=100, m=10;
 int stack[n][m];

void avarage(int stn, int lesn)
{
 int i,j;
 float sum=0,av=0;
```

```
for(i=0;i<stn;i++)
 {
  for(j=0;j<lesn;j++)
  {
   cout<<"Enter  Mark:\n";
   cin>>stack[i][j];
   sum+=stack[i][j];
  }
  av=sum/lesn;
  cout<<"Avarage = "<< av<<endl;
  }
}
/*--------------------------------*/
int main()
{
 int sn,ln;

 cout<<"Enter no of student:\n";
 cin>> sn;
 cout<<"Enter no of lessons:\n";
 cin>> ln;
 avarage(sn,ln);
 return 0;
 }
```

EX2:Write program to enter elements in the stack and reverse it.

```
#include<iostream>


const int size=10;
int   stack1[size];
int   stack2[size];
int   top1=-1,top2=-1;

void push(int n)
{
 int item;
 if (top==size-1)
  cout<<"Stack is Overflow:\n";
 else
      for(int j=0;j<n;j++)
      {
       cout<<"Enter Item: \n";
       cin>>item;
       top1++;
       stack1[top1]=item;
      }
}
```

```
void reverse(int n)
{
 int d,j;

 for(j=n;j>0;j--)
 {
  if (top1<0)
   cout<<" Stack is UnderFlow: \n";
  else
  {
   top2++;
   stack2[top2]=stack1[top1];
   top1--;
  }
 }
}

void display(int top,int st[size])
{
 if (top<0)
   cout<<"Stack is Empty \n";
 else
   for(int i=top;i>=0;i--)
   {
    cout<<"stack["<<i<<"] --> ";
    cout<<st[i]<<endl;
   }
}

int main()
{
 int n;

 cout<<"Enter n: \n";
 cin>>n;
 push(n);

 display(top1,stack1);

 reverse(n);
 display(top2,stack2);
 return 0;
}
```

H.W: Write program to find **min** and **max** element in the stack.

Infix , Postfix and Prefix:

Consider a simple expression : A + B This notation is called Infix Notation , where Postfix notation is A B + , and the prefix notation is + A B .

As you might have noticed, the operator is placed inside the operands is called infix notation, when the operator is placed after the operands (Hence the name 'post'). Postfix is also known as Reverse Polish Notation.

Likewise the equivalent expression in prefix would be + A B, where the operator precedes the operands.

So if **X1** and **X2** are two operands and **OP** is a given operator then

Infix		postfix		prefix
X1 OP X2	=	X1 X2 OP	=	OP X1 X2

Infix to Postfix:

• Generally postfix expressions are free from operator precedence that why they are preferred in computer system. Computer system uses postfix to represent expression.
• This process can be done by using one stack and output list. The following table show the Order Of Precedence For Operators

1 ^, ( ) ,Not
2 *, /, AND, DIV, MOD
3 +, -, OR
4 =, <, >, !=, <=, >=

Infix To Postfix: Algorithm

- A trace of the algorithm that converts the infix A/B^C-D expression to postfix form

Expression	Current Symbol	Stack	List	Comment
A/B^C-D	Initial State	NULL	–	Initially Stack is Empty
/B^C-D	A	NULL	A	Print Operand
B^C-D	/	/	A	Push Operator Onto Stack
^C-D	B	/	AB	Print Operand
C-D	^	/^	AB	Push Operator Onto Stack because Priority of ^ is greater than Current Topmost Symbol of Stack i.e '/'
-D	C	/^	ABC	Print Operand
D	–	/	ABC^	**Step 1 :** Now '^' Has Higher Priority than Incoming Operator So We have to Pop Topmost Element . **Step 2 :** Remove Topmost Operator From Stack and Print it
D	–	NULL	ABC^/	**Step 1 :** Now '/' is topmost Element of Stack Has Higher Priority than Incoming Operator So We have to Pop Topmost Element again. **Step 2 :** Remove Topmost Operator From Stack and Print it
D	–	–	ABC^/	**Step 1 :** Now Stack Becomes Empty and We can Push Operand Onto Stack
NULL	D	–	ABC^/D	Print Operand
NULL	NULL	–	ABC^/D-	Expression Scanning Ends but we have still one more element in stack so pop it and display it

Infix To Postfix: Algorithm

(A + (B * C)) – (D / E)

Infix	Stack(bot->top)	Out Put Postfix
a) (A + (B * C)) – (D / E)		
b) + (B * C))- (D / E)	(	A
c) (B * C))- D / E	(+	A
d) * C))- (D / E)	(+	A B
e) C))- (D / E)	(+(*	A B
f) - (D / E)	(+ (*	A B C
g) (D / E)	-	A B C *+
h) / E)	- (	A B C *+ D
i) E)	- (/	A B C *+ D
j)	- (/	A B C *+ D E
k)		A B C * +D E / -

EX: Give the postfix string to the following expressions:
1- A*B^C+D
2-A*(B+(C*D))+E
3-(B^2- 4*A*C)^(1/2)
4-A/B^C+D*E-A*C

Evaluating a Postfix Expression: Algorithm

Reading the expression takes place from left to right

1. Read the next element **/* first element for first time */**
2. If element is operand than
    1. Push the element in the stack
3. If element is operator then
    1. Pop two operands from the stack **/* POP one operand in case of NOT operator*/**
    2. Evaluate the expression formed by the two operands and the operator
    3. Push the results of the expression in the stack end.
4. If no more elements then
    1. POP the result else go to step 1
5. Exit

Evaluating a Postfix Expression: Example

1 2 3 + *

Postfix		Stack(bot -> top)
a)	**1 2 3 + ***	
b)	**2 3 + ***	**1**
c)	**3 + ***	**1 , 2**
d)	**+ ***	**1 , 2, 3**
e)	*****	**1, 5 // 5 from 2 + 3**
f)		**5 // 5 from 1 * 5**

EX: Find the evaluation for the following postfix strings:

1- 67+3*
2- 84/62/+
3- 84+73+*
4- 234+5**