Name of Teacher: Miss Radhika M. Patil

Class: B.Sc. Computer Science (Entire)- II **Semester : 4**

Course Title: Introduction to Data Structure using C++

<u>UNIT 2: Stack and Queue</u>

- **Stack:**

- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- A Stack is a linear data structure
- A Stack is collection of similar elements that follows the **LIFO (Last-In-First-Out)** principle.
- Stack can also be defined as A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle
- i.e. the element which is inserted last will be deleted first.
- Stack has one end (called as top of the stack) pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.
- **<u>Example</u>:** Consider the simple example of plates stacked over one another. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.
- Other Examples: stack of book, stack of coins etc.

- **The following are some common operations implemented on the stack:**

1. **isFull():** It determines whether the stack is full or not.‘
2. **isEmpty():** It determines whether the stack is empty or not.
3. **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
4. **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
5. **display():** It prints all the elements available in the stack.

- **Stack Operations:**

  Basic operations of stack are:

1. Push
2. Pop

1. PUSH operation:

- **The steps involved in the PUSH operation is given below:**

1. Before inserting an element in a stack, we check whether the stack is full.
2. If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
3. When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
4. When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
5. The elements will be inserted until we reach the ***max*** size of the stack.

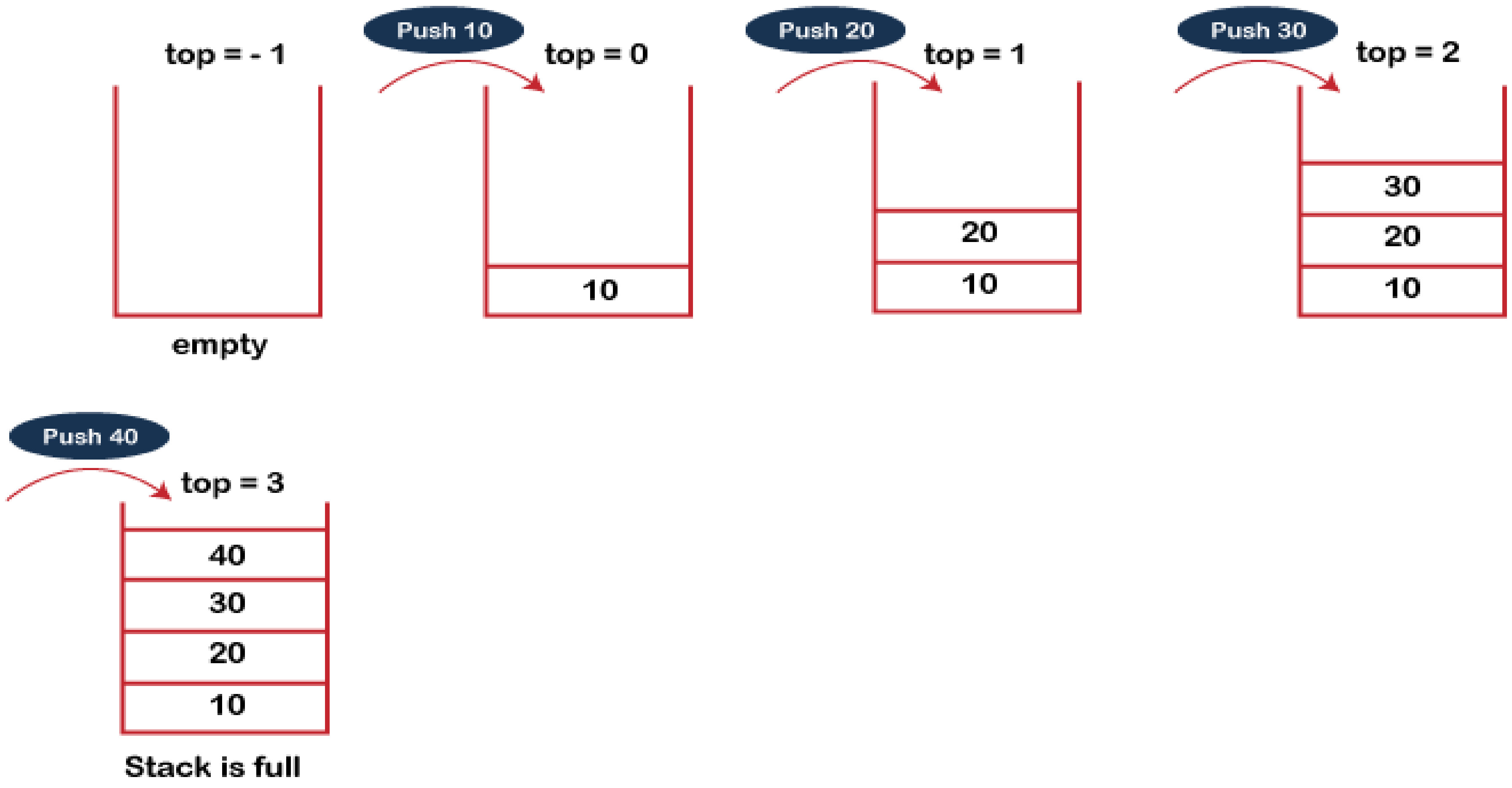


Fig. Push Operation of stack

- **Push operation**
- **<u>Algorithm:</u>**

Here 'stack' is an array of 'max' size and we want to push (insert) an element x into the stack.

```
Step 1 : Check for Overflow
            if top= = (max-1) then
                Print "Stack is full" and return
            else
             a) top=top+1              //increment top by 1
             b) stack[top]=x           // push or insert the element x at the top of the stack
            End if
Step 2 : Return.
```

2. POP operation

The steps involved in the POP operation is given below:

1. Before deleting the element from the stack, we check whether the stack is empty.
2. If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
3. If the stack is not empty, we first access the element which is pointed by the ***top***
4. Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

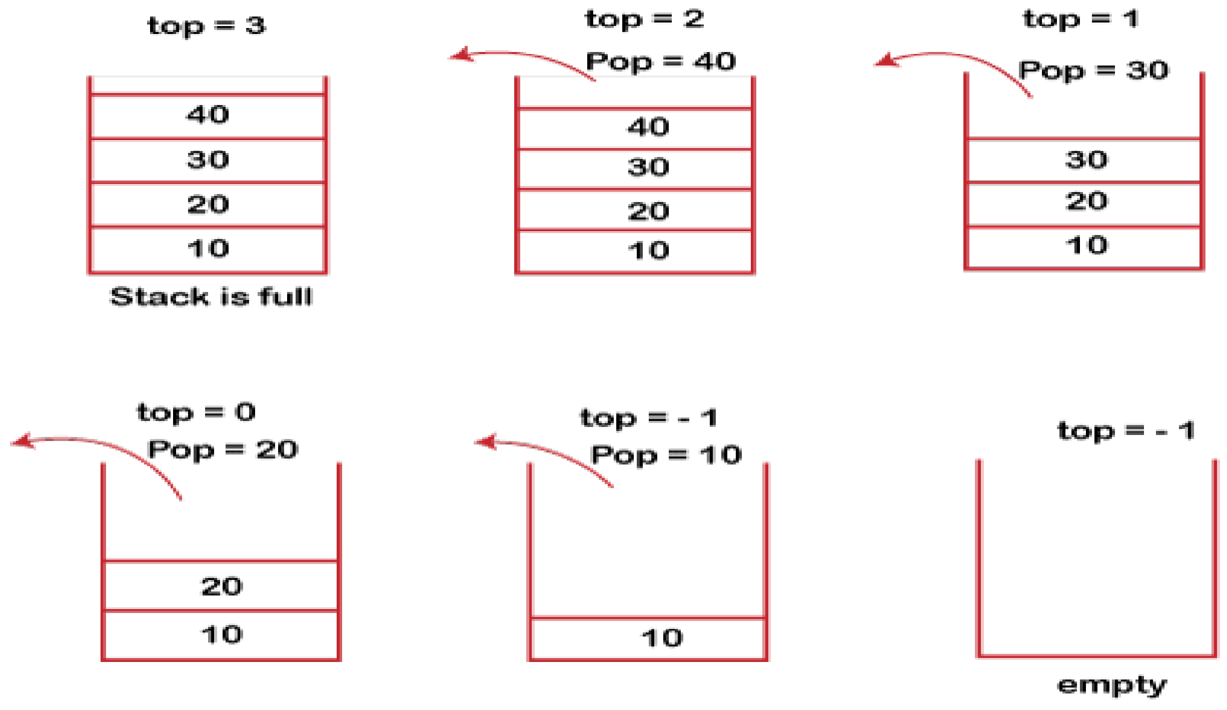


Fig. Pop Operation of stack

- **Pop operation**
- **<u>Algorithm:</u>**

Here 'stack' is an array of 'max' size and we want to pop (delete) an element from the stack.

```
Step 1 : Check for Underflow
            if top= = -1  then
               Print "Stack is Empty" and return
            else
             a) x = stack[top]              //store the element at top of the stack
             b) top = top-1                 // decrement top by 1
            End if
Step 2 : Return.
```

Program: Static implementation of stack using array.

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
int max=10;
class Stack
{
            int top;
            int s[20];
  public:
            Stack()
            {
              top=-1;
            }
            void push();
            void pop();
            void display();
};
```

```
void Stack::push()
{
            int x;
            cout<<"\n Enter number to push:";
            cin>>x;
            if(top==max-1)
            {
             cout<<"\n Stack is full";
             //exit(0);
            }
            else
            {
              top=top+1;
              s[top]=x;
              display();
            }
}
```

```
void Stack::pop()
{
            int x;
            if(top==-1)
            {
               cout<<"\n Stack is empty";
              //  exit(0);
            }
            else
            {
               x=s[top];
               cout<<"\n Popped element is:"<<x;
               top=top-1;
               display();
            }
}
```

```
void Stack::display()
{
                if(top==-1)
                {
                  cout<<"\n Stack is empty";
                 //exit(0);
                }
                else
                {
                   cout<<"\n Stack is: ";
                   for(int i=0;i<=top;i++)
                   {
                      cout<<" "<<s[i];
                   }
                }
}
```

```
void main()
{
    Stack st;
    int ch;
    clrscr();
                cout<<"\n 1. Push";
                cout<<"\n 2. Pop";
                cout<<"\n 3. Exit";
    do
    {
                cout<<"\n Enter your choice:";
                cin>>ch;
                switch(ch)
                {
                                case 1 : st.push();
                                                break;
                                case 2 : st.pop();
                                                break;
                                case 3 : cout<<"\n Exit....";
                                                break;
                                default: cout<<"\n Invalid choice";
                }
    }while(ch!=3);
    getch();
}
```

- **Applications of Stack**

  The following are the applications of the stack:

1. String reversal:

- ➢ Stack is also used for reversing a string.
- ➢ For example, we want to reverse a string, so we can achieve this with the help of a stack.
- ➢ First, we push all the characters of the string in a stack until we reach the null character.
- ➢ After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

2. UNDO/REDO:

- ➢ It can also be used for performing UNDO/REDO operations.
- ➢ For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
- ➢ So, there are three states, a, ab, and abc, which are stored in a stack.
- ➢ There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
- ➢ If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

3. Recursion:

- ➢ The recursion means that the function is calling itself again.
- ➢ To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

4. DFS(Depth First Search):

- This search is implemented on a Graph, and Graph uses the stack data structure.

5. Expression conversion:

- Stack can also be used for expression conversion.
- This is one of the most important applications of stack.
- The list of the expression conversion is given below:

Infix to prefix

Infix to postfix

Prefix to infix

Prefix to postfix

Postfix to infix

6. Memory management:

- The stack manages the memory.
- The memory is assigned in the contiguous memory blocks.
- The memory is known as stack memory as all the variables are assigned in a function call stack memory.
- The memory size assigned to the program is known to the compiler.
- When the function is created, all its variables are assigned in the stack memory.
- When the function completed its execution, all the variables assigned in the stack are released.

1. Recursion:

- One of the most important application of stack is recursion where it is used to save the parameters, local variables and return address.
- Recursion is an ability of a function or procedure or algorithm to repeatedly call itself until a certain condition is made.
- This condition is called **Base condition.**
- A function which call itself is called a recursive function.
- A recursive function is said to be well defined if the base condition is defined.
- Recursion can be classified in two types:

1. Direct
2. Indirect

Example 1:

```
sample()
{
      ---------
      ---------
      sample();
      ---------
      ---------
}
```

fig (a) Direct Recursion

Example 2:

```
sample()                    indirect()
{                           {
      ---------                   ---------
      ---------                   ---------
      indirect();                 sample();
      ---------                   ---------
      ---------                   ---------
}                           }
```

fig (b) Indirect Recursion

- Direct recursion occurs when the function call itself directly as shown in fig (a).
- Indirect recursion occurs when the function call other function which may call the original function as shown on fig (b).
- Recursion requires more memory and time to implement, therefore the function implement without recursion may work faster.

Example:

- Calculate factorial of a number using recursion.

Algorithm:

fact(n) is a function that calculates the factorial of a given number n.

```
Step 1:  if n= 0 then              // base condition
             f=1
         else
             f= n * fact(n-1)    // leading towards
         End if
Step 2: Return f
```

The factorial of any integer n is given as:

n != 1 x 2 x 3 x ……………………….(n-2) x (n-1) x n

OR

n != n x (n-1) x (n-2) x ……………………….3 x 2 x 1 x 0!

where , 0! = 1

- To find out the value of 5 factorial (5!) , we follow the sequence as:

5!= 5 * 4!

= 4 * 3!

= 3 * 2!

= 2 * 1!

= 1 * 0! (base condition)

- The value of 5! is product of 5 and 4!
- To find out the value of 5!, we first find the value of 4!
- For every positive integer
- n!= n * (n-1) !
- The fact() function calls itself with different parameter values 4,3,2,1,0.
- For value 0 the fact () does not call itself because the default for 0! is 1.
- This is the base condition.

Program: Write a C++ program to calculate factorial of a given number using recursion.

```
int fact (int);
void main()
{
        int n, f;
        cout<<“\n Enter a number:”;
        cin>>n;
        f= fact (n);
        cout<<“\n Factorial =”<<f;
        getch();
}
```

```
int fact(int n)
{
        if (n = = 0)
        return 1;
        else
         return (n * fact (n-1));
}
```

Output:

```
Enter a number: 4
Factorial= 24
```

- **Representation of Arithmetic Expressions:**

- An arithmetic expression is made up of operands and operators.
- The binary operations may have different levels of precedence, some times the precedence is changed by using parenthesis.
- An arithmetic expression can be represented in 3 different formats according to the relative position of operator with respect to two operands.

1. Infix expression
2. Prefix expression
3. Postfix expression

1. Infix Notation:

- This notation of arithmetic expression is used in most arithmetic operations.
- In this method the operator comes <u>**in between**</u> two operands
- Note the unary operators precede their operand.

Example:

1. A+B
2. A+B*C
3. X+Y*Z^M

- To determine the final value of expression the precedence of operator is determined by 'BODMAS' rule.

- We consider the operations:

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exponentiation

- We assume the following 3 levels of precedence for these 5 binary operations in such notation.

1. Highest - Exponentiation
2. Next Highest - Multiplication and Division
3. Lowest - Addition and Subtraction

- **Polish Notation:**

➢ In polish notation the operator is used before or after the operand.

➢ The fundamental property of polish notation is that the order in which the operations are performed is determined by the position of the operator and operand in the expression.

➢ Compared to infix expression the parenthesis are not required.

➢ Any infix expression is converted into polish notation the only rule for this conversion is that

1. The operators with higher precedence are converted first.
2. After a portion of expression has been converted to polish notation, it must be treated as a single operand in the rest of expression.
3. Note that the order of operands in all forms is same .

There are two types of polish notation:

1. Prefix
2. Postfix.

1. Prefix Notation (Polish Notation):

- When the operators are written before their operands then the expression is called prefix expression.

Example:

1. +AB
2. - + XY* YZ
3. - + X * YZ / MN

- **Conversion from infix to prefix expression:**

Let us take the infix expression:

A + B * C – D / E

1. We know that multiplication and division have the same precedence and they are on same level. So the conversion should be performed from left to right.

Therefore multiplication is performed first.

So B * C becomes *BC

Set T_1 = *BC

Where T_1 is the single argument.

Now expression becomes

A + T_1 – D / E

2. Now division is performed

   i.e. D / E becomes / DE

   set T_2 = / DE

   i.e. expression becomes

   $A + T_1 - T_2$

3. We know that addition and subtraction have the same precedence and they are on same level. So the conversion should be performed from left to right.

   A+ T_1 becomes + A T_1

   Set, T_3 = + A T_1

   i.e. $T_3 - T_2$

4. Now $T_3 - T_2$ becomes $-T_3T_2$

   This is the final prefix expression. Put the values of T_1 , T_2 and T_3 in final prefix expression.

5. Now the final prefix expression becomes,

   $-T_3T_2$

   - + A T_1 / DE

$$\boxed{\mathbf{- + A *BC / DE}}$$

Assignment :

Convert the following infix expressions into prefix expressions.

1. a/ b –c + d * e – a * c
2. (a - b) / c * (d + e – f / g)
3. (A + B ^ C) * D + E ^ S
4. (A – (B / C)) * ((D * E) – F)

2. Postfix Notation (Reverse Polish Notation):

➢ When the operators are written after their operands then the expression is called postfix expression.

Example:

1. AB +
2. AB + CD * -
3. ABC * + DE / -

➢ This notation is very important, because the common evaluation used by the computer is that it accepts an infix expression and produces the correct code by converting it into postfix expression and then evaluation of expression is done.

- **Conversion from infix expression to postfix expression:**

Let us take the infix expression:

A + B * C – D / E

1. We know that multiplication and division have the same precedence and they are on same level. So the conversion should be performed from left to right.

Therefore multiplication is performed first.

So B * C becomes BC *

Set T_1 = BC *

Where T_1 is the single argument.

Now expression becomes

A + T_1 – D / E

2. Now division is performed

   i.e. D / E becomes DE/

   set T_2 = DE/

   i.e. expression becomes

   $\mathbf{A + T_1 - T_2}$

3. We know that addition and subtraction have the same precedence and they are on same level. So the conversion should be performed from left to right.

   A+ T_1 becomes A T_1 +

   Set, T_3 = A T_1 +

   i.e. T_3 - T_2

4. Now T_3 - T_2 becomes T_3T_2 -

   This is the final postfix expression. Put the values of T_1 , T_2 and T_3 in final postfix expression.

5. Now the final postfix expression becomes,

   T_3T_2 -

   A T_1 + DE/ -

   $\boxed{\textbf{ABC * + DE/ -}}$

   Here prefix is not mirror image of postfix expression.

Assignment :

Convert the following infix expressions into postfix expressions.

1. A + B * C ^ D
2. A * B ^ C – D
3. X * Y ^ Z / M + N
4. A- B / (C * D ^ E)

- **Conversion of infix to postfix using Stack:**

Algorithm:

- The function postfix (I, P) converts the infix expression I into postfix expression.
- Take the empty stack, infix expression and put the special character # as a delimiter to mark the end of infix expression.
- Place # in the stack to mark empty stack.
- Here P represents the postfix expression which is initially empty.

Step 1: Read the infix expression I from left to right one character at a time.

Step 2: Repeat the step 3 until delimiter (#) is encountered.

Step 3: Read the symbol, input character from infix expression.

(a) If the scanned symbol is operand then put it in postfix string.

(b) Else if scanned symbol is an opening parenthesis then push it onto the top of the stack.

(c) Else if scanned symbol is closing parenthesis then pop from the stack until opening parenthesis is encountered (also pop the opening parenthesis); concatenate the popped elements into postfix expression in their order of popping.

(Note: Do not add opening and closing parenthesis to the postfix notation).

(d) Else if the scanned symbol is operator then

1. If the incoming operator has **higher priority** than the top of stack, insert the operator on stack.

2. If the incoming operator has **same or less precedence** than operators at the top of the stack, then pop all such operators from top of the stack and concatenate them to postfix string.

Step 4: When # is encountered while reading input then pop all elements from the stack to make the stack empty.
Example: Convert the following infix expression into postfix expression using stack.

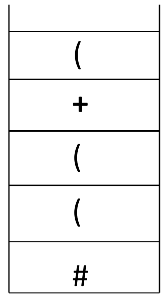
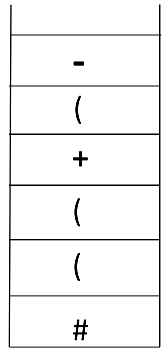
((A + (B – C)) / D)

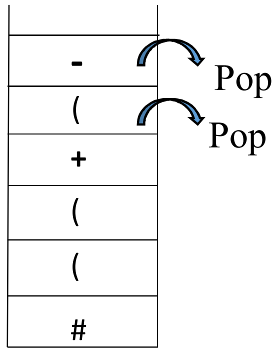
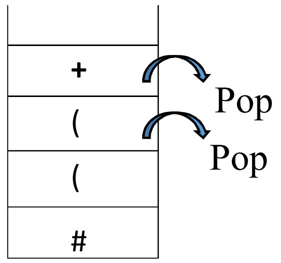
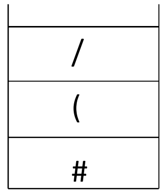
- The delimiter # is placed at the end of infix expression as

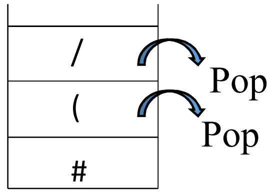
((A + (B – C)) / D) #

- The conversion from infix to postfix shown in following table.

Sr. No.	Symbol Scanned	Postfix Expression	Stack
1	(	-	(#
2	(	-	((#
3	A	A	Stack remains same
4	+	A	+ ((#

Sr. No	Scanned Symbol	Prefix Expression	Stack
5	**(**	**A**	
6	**B**	**AB**	Stack remains same
7	**-**	**AB**	
8	**C**	**ABC**	Stack remains same

Sr. No.	Scanned Symbol	Prefix Expression	Stack
9	)	**ABC -**	
10	)	**ABC - +**	
11	/	**ABC - +**	

Sr. No.	**Scanned Symbol**	**Prefix Expression**	**Stack**
12	**D**	ABC - + D	Stack remains same
13	**)**	ABC - + D /	
14	#	Stop	Stack is empty

So final postfix expression after conversion is:

A B C - + D /

- Convert following infix expressions in postfix expression using Stack.

1. A + (B * C – (D / E ^ F) * E) * H
2. (A – B) * (C / D) + E

- **Evaluation of Postfix expression using Stack:**
- **Algorithm:**

evalPostfix (P, Stack, Ans)

This function evaluates the postfix expression P using Stack and stores the result in Ans.

```
Step 1: Add a delimiter # at the end of postfix expression
Step 2: Scan the given postfix expression P from left to right reading one character at a time.
Step 3: Undertake steps 1 and 2
Step 4: Read next character from P.
Step 5: If next character is operand then push it into the stack
        Else if next character is operator (op) then
            a) Pop top element from the stack, let it be in var1.
            b) Again pop top (next to top) from the stack and let it be in var2.
            c) Ans = var2 op var1
            d) Push the result Ans into the stack.
        End if
Step 6: Until # is encountered pop the top element from the stack and let it be in var3.
Step 7: set  Ans= var3
Step 8: Stop.
```

- **Example:** Evaluate the postfix expression **5 12 3 2 2 ^ * / -**

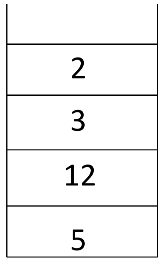
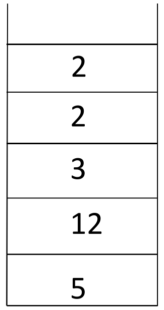
Solution:

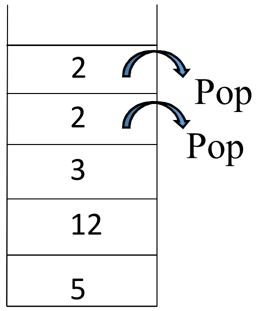
Let P= **5 12 3 2 2 ^ * / -**

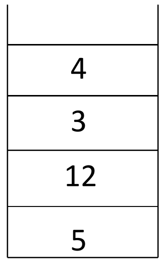
Place delimiter # at the end of P as

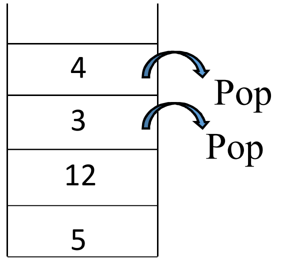
P = **5 12 3 2 ^ * / - #**

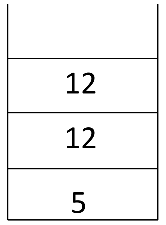
Sr. No.	**Character Read**	**Stack**	**Remark**
1	5	5	-
2	12	12 5	-
3	3	3 12 5	-

Sr. No.	Character Read	Stack	Remark
4	2		-
5	2		-

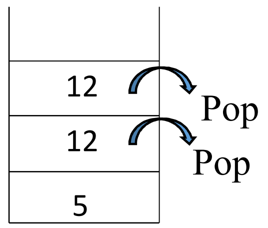
6	^		Var1 = 2
			Var2 = 2
			Ans = var2 op var1
			Ans = 2 ^ 2= 4
			Push Ans onto the stack

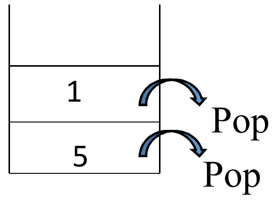
Sr. No.	Character Read	Stack	Remark
6			

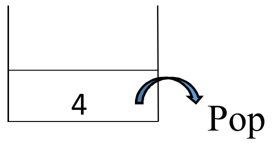
7	*		Var1 = 4
			Var2 = 3
			Ans = var2 op var1
			Ans = 3 * 4 = 12
			Push Ans (12) onto the stack

Sr. No.	Character Read	Stack	Remark

8	/		Var1 = 12
			Var2 = 12
			Ans = var2 op var1
			Ans = 12 / 12 = 1
			Push Ans (1) onto the stack

9	-		Var1 = 1
			Var2 = 5
			Ans = var2 op var1
			Ans = 5 - 1 = 4
			Push Ans (4) onto the stack

10	#		Pop the top element of the stack and assign it to Ans i.e. Ans = 4

- **Infix to prefix using Stack:**

Steps:

1. Reverse the given infix expression
2. Convert it into postfix expression
3. Again reverse the final postfix expression to get final prefix expression.

THANK YOU…