**INFIX to POSTFIX**

OPERATORS = set(['+', '-', '\*', '/', '(', ')', '^'])  # set of operators

PRIORITY = {'+':1, '-':1, '\*':2, '/':2, '^':3} # dictionary having priorities

def infix\_to\_postfix(expression): #input expression

    stack = [] # initially stack empty

    output = '' # initially output empty

    for ch in expression:

        if ch not in OPERATORS:  # if an operand then put it directly in postfix expression

            output+= ch

        elif ch=='(':  # else operators should be put in stack

            stack.append('(')

        elif ch==')':

            while stack and stack[-1]!= '(':

                output+=stack.pop()

            stack.pop()

        else:

            # lesser priority can't be on top on higher or equal priority

             # so pop and put in output

            while stack and stack[-1]!='(' and PRIORITY[ch]<=PRIORITY[stack[-1]]:

                output+=stack.pop()

            stack.append(ch)

    while stack:

        output+=stack.pop()

    return output

expression = input('Enter infix expression')

print('infix expression: ',expression)

print('postfix expression: ',infix\_to\_postfix(expression))

**OUTPUT:**

**infix expression: a\*b\*c**

**postfix expression: ab\*c\***

**INFIX to PREFIX**

def isOperator(c):

    return (not (c >= 'a' and c <= 'z') and not(c >= '0' and c <= '9') and not(c >= 'A' and c <= 'Z'))

def getPriority(C):

    if (C == '-' or C == '+'):

        return 1

    elif (C == '\*' or C == '/'):

        return 2

    elif (C == '^'):

        return 3

    return 0

def infixToPrefix(infix):

    operators = []

    operands = []

    for i in range(len(infix)):

        if (infix[i] == '('):

            operators.append(infix[i])

        elif (infix[i] == ')'):

            while (len(operators)!=0 and operators[-1] != '('):

                # operand 1

                op1 = operands[-1]

                operands.pop()

                # operand 2

                op2 = operands[-1]

                operands.pop()

                # operator

                op = operators[-1]

                operators.pop()

                tmp = op + op2 + op1

                operands.append(tmp)

            operators.pop()

        elif (not isOperator(infix[i])):

            operands.append(infix[i] + "")

        else:

            while (len(operators)!=0 and getPriority(infix[i]) <= getPriority(operators[-1])):

                op1 = operands[-1]

                operands.pop()

                op2 = operands[-1]

                operands.pop()

                op = operators[-1]

                operators.pop()

                tmp = op + op2 + op1

                operands.append(tmp)

            operators.append(infix[i])

    while (len(operators)!=0):

        op1 = operands[-1]

        operands.pop()

        op2 = operands[-1]

        operands.pop()

        op = operators[-1]

        operators.pop()

        tmp = op + op2 + op1

        operands.append(tmp)

    return operands[-1]

s = "(A-B/C)\*(A/K-L)"

print( infixToPrefix(s))

**Output:**

**\*-A/BC-/AKL**

**Theory to study for stack:**

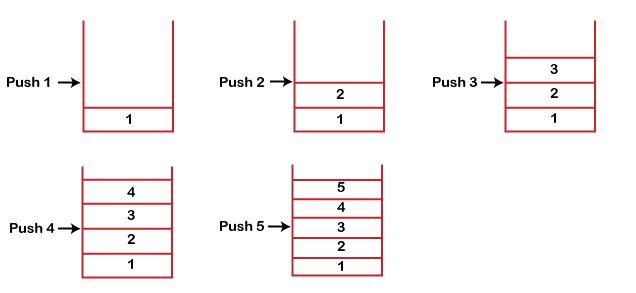
1. **What is a Stack?**

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** 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.**

1. **Working of Stack**

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

1. **Standard Stack Operations**

**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.

**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.

**isEmpty():** It determines whether the stack is empty or not.

**isFull():** It determines whether the stack is full or not.'

**peek():** It returns the element at the given position.

**count():** It returns the total number of elements available in a stack.

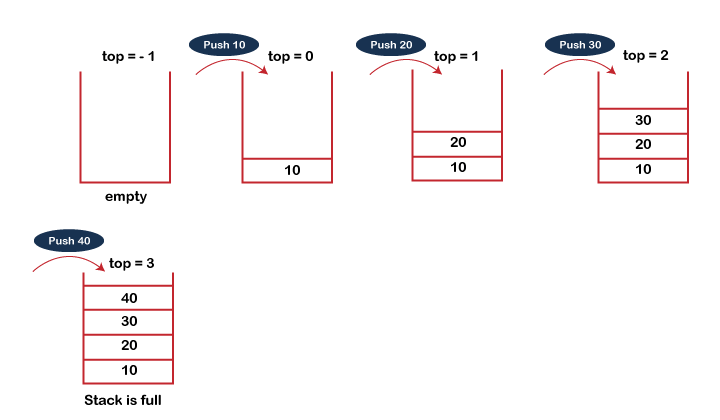
**change():** It changes the element at the given position.

**display():** It prints all the elements available in the stack.

### PUSH operation

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

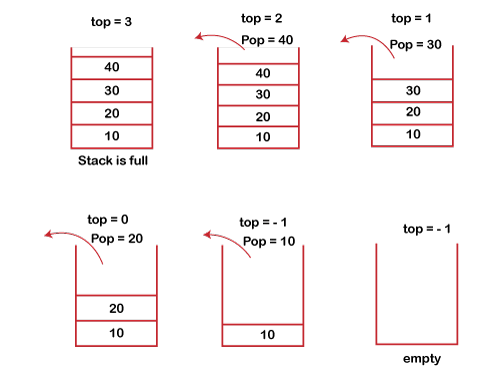
* Before inserting an element in a stack, we check whether the stack is full.
* If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
* When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
* When the new element 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**.
* The elements will be inserted until we reach the **max** size of the stack.



### POP operation

**The steps involved in the POP operation is given below:**

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



### 4. **Applications of Stack**

**The following are the applications of the stack:**

* **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

int main()

{

   cout<<"Hello";

   cout<<"javaTpoint";

}

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

* **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" 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.
* **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.
* **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.
* **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
* **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
* **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

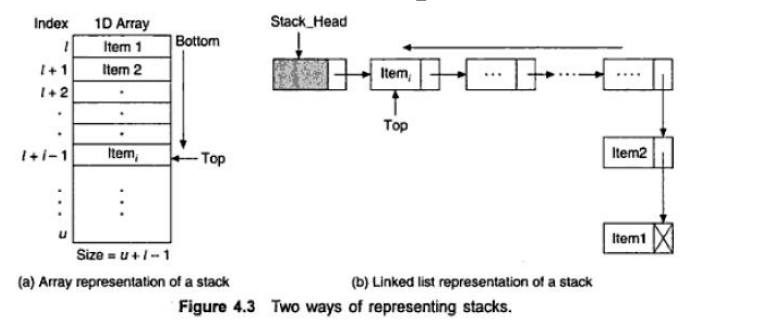
* **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is 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.

## 5. **Disadvantages of Array in Data Structure**

1. **Fixed Size** – One major disadvantage of arrays is that they have a fixed size. Once an array is created, its size cannot be changed. This means that if you need to add or remove elements from the array, you will need to create a new array with a different size and copy the elements from the old array to the new one. This can be time-consuming and inefficient.
2. **Wastage of Memory** – Another disadvantage of arrays is that they can waste memory. If you create an array with a larger size than you need, you may end up wasting memory because some of the elements in the array will be empty or unused. This can be a problem if you are working with large amounts of data or if memory is limited.
3. **Difficulty in Inserting and Deleting Elements** – Difficulty in Inserting and Deleting Elements
4. **Inefficient for Search** – Searching for a specific element in an array can also be inefficient. If the array is unsorted, you will need to check each element in the array until you find the one you are looking for. If the array is sorted, you can use a binary search algorithm to find the element more quickly, but this requires that the array be sorted in advance.
5. **Limited Flexibility** – Finally, arrays can be limited in terms of flexibility. While they can store a variety of data types, they cannot easily handle complex data structures or non-uniform data. For example, if you need to store a list of names and addresses, you may need to create an array of objects that contain multiple fields, such as name, address, city, and state. This can be more complex than using a different data structure, such as a linked list or a hash table.
6. **REPRESENTATION OF A STACK**

A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list. Array Representation of Stacks: First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion. In Figure, Item i denotes the ith item in the stack; l and u denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack. With this representation, the following two ways can be stated:

EMPTY: TOP < l FULL: TOP ≥ u



Linked List Representation of Stacks: Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications, the size of the stack may vary during program execution. An obvious solution to this problem is to represent a stack using a linked list. A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usual, to point to the next' item. Above Figure b depicts such a stack using a single linked list. 2 In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item. Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list.

**\*\*\* Study sums on conversion from infix to prefix and infix to postfix\*\*\***