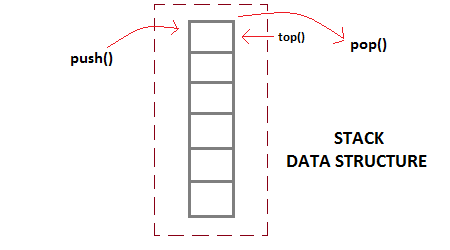
**Stacks**

**Stack** is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



## Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

**Applications of Stack**

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

## There are other uses also like: Parsing, Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

## Implementation of Stack Data Structure

## Stack can be easily implemented using an Array or a [Linked List](https://www.studytonight.com/data-structures/introduction-to-linked-list). Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

## Implementation of Stack

## Algorithm for PUSH operation

## Check if the stack is full or not.

## If the stack is full, then print error of overflow and exit the program.

## If the stack is not full, then increment the top and add the element.

## Algorithm for POP operation

## Check if the stack is empty or not.

## If the stack is empty, then print error of underflow and exit the program.

## If the stack is not empty, then print the element at the top and decrement the top.

|  |  |
| --- | --- |
| Position of Top | Status of Stack |
| -1 | Stack is Empty |
| 0 | Only one element in Stack |
| N-1 | Stack is Full |
| N | Overflow state of Stack |

## Analysis of Stack Operations

## Push Operation : O(1)

## Pop Operation : O(1)

## Top Operation : O(1)

## Search Operation : O(n)

## Expression Conversion and Evaluation

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C\*D,D/A** etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)\*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

## Infix Expression

It follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., **A+B**

**Prefix Notation**

In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**

## Postfix Expression

It follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>. E.g., **AB+.** Postfix notation is also known as **Reverse Polish Notation**

### Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
   1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
   2. Add operator to Stack.  
      [End of If]
6. If a right parenthesis is encountered ,then:
   1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
   2. Remove the left Parenthesis.  
      [End of If]  
      [End of If]
7. END.

**Example**

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression No** | **Infix Notation** | **Prefix Notation** | **Postfix Notation** |
| 1 | a + b | + a b | a b + |
| 2 | (a + b) \* c | \* + a b c | a b + c \* |
| 3 | a \* (b + c) | \* a + b c | a b c + \* |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) \* (c + d) | \* + a b + c d | a b + c d + \* |
| 6 | ((a + b) \* c) - d | - \* + a b c d | a b + c \* d - |

**Postfix Expression Evaluation using Stack Data Structure**

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. **Read all the symbols one by one from left to right in the given Postfix Expression**
2. **If the reading symbol is operand, then push it on to the Stack.**
3. **If the reading symbol is operator (+ , - , \* , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.**
4. **Finally! perform a pop operation and display the popped value as final result.**
5. **Example**
6. Consider the following Expression...
7. 