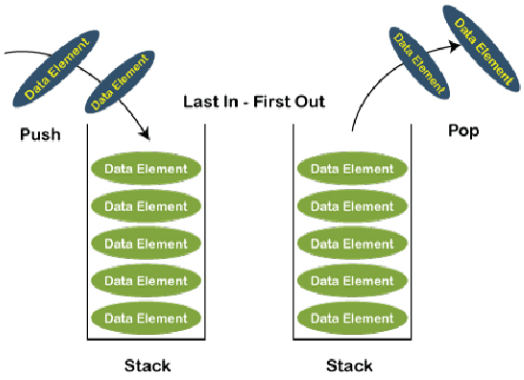
What is a Stack?

A Stack is a linear data structure. In case of an array, random access is possible, i.e., any element of an array can be accessed at any time, whereas in a stack, the sequential access is only possible. It is a container that follows the insertion and deletion rule. It follows the principle **LIFO (Last In First Out)** in which the insertion and deletion take place from one side known as a **top**. In stack, we can insert the elements of a similar data type, i.e., the different data type elements cannot be inserted in the same stack. The two operations are performed in LIFO, i.e., **push** and **pop** operation.



The following are the operations that can be performed on the stack:

- **push(x):** It is an operation in which the elements are inserted at the top of the stack. In the **push** function, we need to pass an element which we want to insert in a stack.
- **pop():** It is an operation in which the elements are deleted from the top of the stack. In the **pop()** function, we do not have to pass any argument.
- **peek()/top():** This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element but does not remove that element from the stack.
- **isEmpty():** If the stack is empty, then this function will return a true value or else it will return a false value.
- **isFull():** If the stack is full, then this function will return a true value or else it will return a false value.

n stack, the **top** is a pointer which is used to keep track of the last inserted element. To implement the stack, we should know the size of the stack. We need to allocate the memory to get the size of the stack. There are two ways to implement the stack:

- **Static:** The static implementation of the stack can be done with the help of arrays.
- **Dynamic:** The dynamic implementation of the stack can be done with the help of a linked list.

The following are the differences between the stack and queue:

Basis for comparison	Stack	Queue
Principle	It follows the principle LIFO (Last In- First Out), which implies that the element which is inserted last would be the first one to be deleted.	It follows the principle FIFO (First In -First Out), which implies that the element which is added first would be the first element to be removed from the list.
Structure	It has only one end from	It has two ends, i.e., front and

	which both the insertion and deletion take place, and that end is known as a top.	rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

1. Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: A + (B - C)

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new notation.

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

Example: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

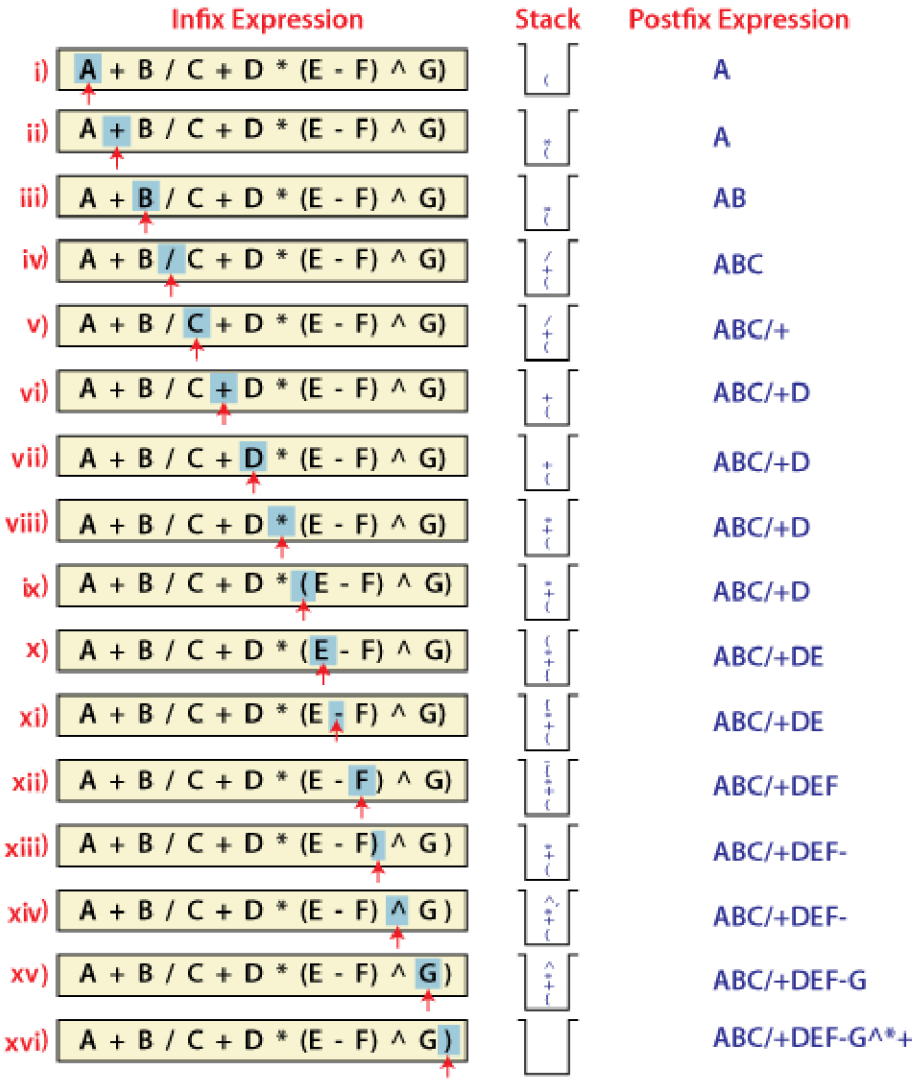
Example: AB +, CD+, etc.

All these expressions are in postfix notation because the operator comes after the operands.

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Let's take the example of Converting an infix expression into a postfix expression.



In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression:

Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.

Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

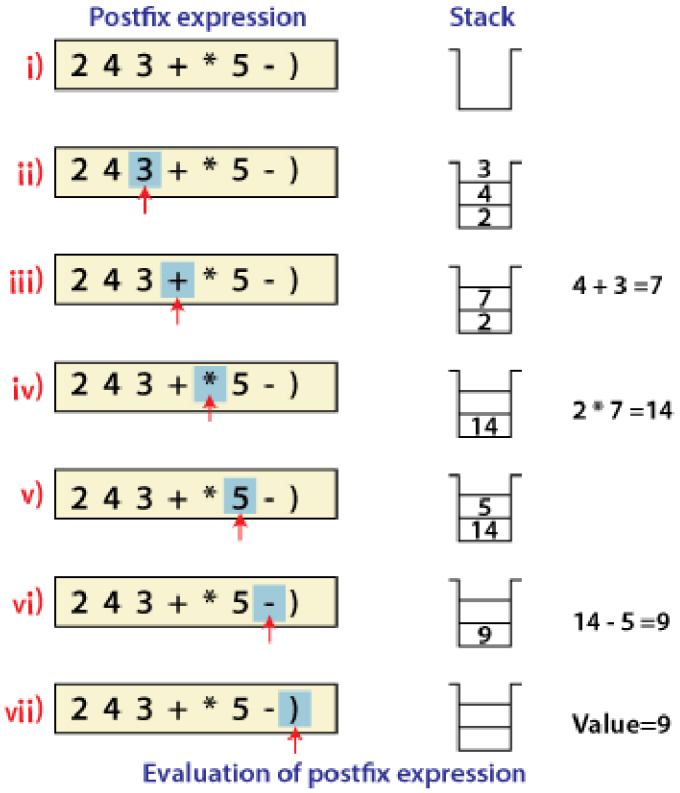
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example:

Now let us consider the following infix expression 2 * (4+3) - 5.

Its equivalent postfix expression is 2 4 3 + * 5.

The following step illustrates how this postfix expression is evaluated.



. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

Valid Delimiter	Invalid Delimiter
While (i > 0)	While (i >
/* Data Structure */	/* Data Structure
{ (a + b) - c }	{ (a + b) - c

To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

On matching, the following cases may arise.

- If the delimiters are of the same type, then the match is considered successful, and the process continues.
- If the delimiters are not of the same type, then the syntax error is reported.

When the end of the program is reached, and the Stack is empty, then the processing of the source program stops.

Example: To explain this concept, let's consider the following expression.

[{a -b) * (c -d)}/f]

Input left	Characters Read	Stack Contents
{{(a-b) * (c-d)}/f]	[	[
{(a-b) * (c-d)}/f]	{	[{
(a-b) * (c-d)}/f]	(	[{ (
a-b) * (c-d)}/f]	a	[{ (
-b) * (c-d)}/f]	-	[{ (
b) * (c-d)}/f]	b	[{ (
) * (c-d)}/f]	)	[{
* (c-d)}/f]	*	[{
(c-d)}/f]	(	[{ (
c-d)}/f]	g	[{ (
-d)}/f]	-	[{ (
d)}/f]	d	[{ (
)}/f]	)	[{
}/f]	}	[
/f]	/	[
f]	f	[
]		

4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

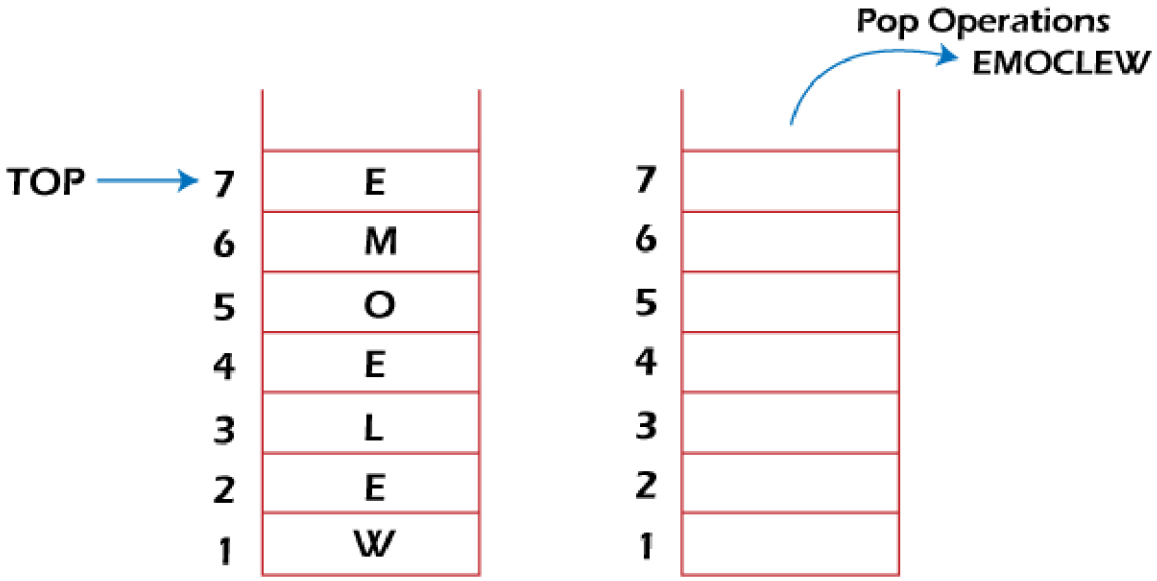
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Implementation of stack

Insertion: push()

push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

1 − Checks if the stack is full.
2 − If the stack is full, produces an error and exit.
3 − If the stack is not full, increments top to point next empty space.
4 − Adds data element to the stack location, where top is pointing.
5 − Returns success.

Deletion: pop()

pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

Algorithm

1 − Checks if the stack is empty.
2 − If the stack is empty, produces an error and exit.
3 − If the stack is not empty, accesses the data element at which top is pointing.
4 − Decreases the value of top by 1.
5 − Returns success.

Stack implementation in python

```
# Creating a stack
def create_stack():
        stack = []

    return stack


# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0


# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)


# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"

    return stack.pop()


stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).