SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

AN AUTONOMOUS INSTITUTION

Accredited by NBA–AICTE and Accredited by NAAC–UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

COURSE NAME : 19CS307 - DATA STRUCTURES

II YEAR / III SEMESTER

Unit II

LINEAR DATA STRUCTURES

2.1 STACK ADT

Stack is a linear data structure which follows a particular order in which the operations are performed.

In linear data structures like an array and linked list a user is allowed to insert or delete any element to and from any location respectively.

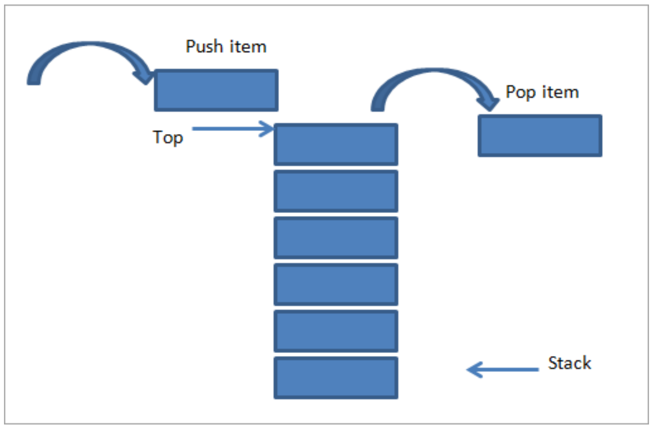
However, in a Stack, both, insertion and deletion, is permitted at one end only.

A Stack works on the LIFO (Last In – First Out) or FILO(First In Last Out)basis,

i.e, the first element that is inserted in the stack would be the last to be deleted; or the last element to be inserted in the stack would be the first to be deleted.

Real-life examples

Consider an example of plates stacked over one another in the canteen. 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 (Last In First Out)/FILO(First In Last Out) order.



2.1.2 STACK OPERATIONS

1. PUSH: PUSH operation implies the insertion of a new element into a Stack. A new element is always inserted from the topmost position of the Stack; thus, we always need to check if the top is empty or not, i.e., TOP=Max-1 if this condition goes false, it means the Stack is full, and no more elements can be inserted, and even if we try to insert the element, a Stack overflow message will be displayed.

Algorithm:

```
Step-1: If TOP = Maxsize-1
Print “Overflow”
Goto Step 4
Step-2: Set TOP= TOP + 1
Step-3: Set Stack[TOP]= ELEMENT
Step-4: END
```

POP: POP means to delete an element from the Stack. Before deleting an element, make sure to check if the Stack Top is NULL, i.e., TOP=NULL. If this condition goes true, it means the Stack is empty, and no deletion operation can be performed, and even if we try to delete, then the Stack underflow message will be generated.

Algorithm:

```
Step-1: If TOP= NULL
Print “Underflow”
Goto Step 4
Step-2: Set VAL= Stack[TOP]
```

```
Step-3: Set TOP= TOP-1
Step-4: END
```

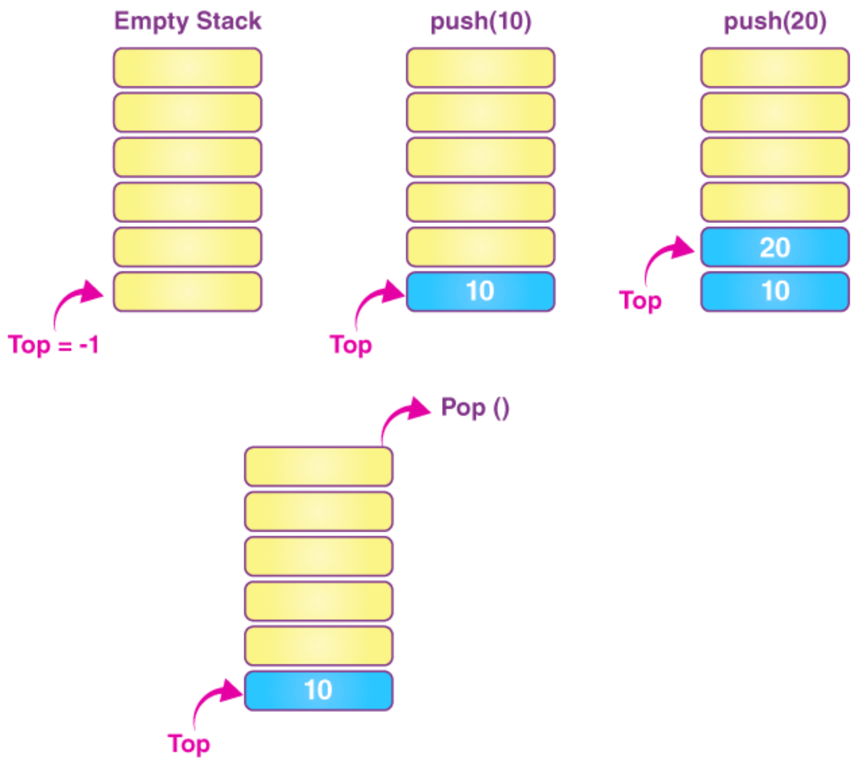
Top/PEEK: When we need to return the value of the topmost element of the Stack without deleting it from the Stack, the Peek operation is used. This operation first checks if the Stack is empty, i.e., TOP = NULL; if it is so, then an appropriate message will display, else the value will return.

The value of top changes with each push() or pop() operation.

Algorithm:

```
Step-1: If TOP = NULL
PRINT “Stack is Empty”
Goto Step 3
Step-2: Return Stack[TOP]
Step-3: END
```

IsEmpty(): When we initialize the stack, there is no element in a stack so, Top value is initialized to **Top = -1 .** The value is -1 as if use an array to implement stack if there is 1 element the location would be a[0], thus, for no element we use – 1.



Implementation

```
#include<stdio.h>

#include<stdlib.h>

#define Size 4

int Top=-1, inp_array[Size];

void Push();

void Pop();

void show();

int main()

{
```

```
int choice;
while(1)
{
printf("\nOperations performed by Stack");
printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
printf("\n\nEnter the choice:");
scanf("%d",&choice);
switch(choice)
{
case 1: Push();
break;
case 2: Pop();
break;
case 3: show();
break;
case 4: exit(0);
default: printf("\nInvalid choice!!");}}}
void Push()
{
int x;
if(Top==Size-1)
{
printf("\nOverflow!!");
}
else
```

```
{
printf("\nEnter element to be inserted to the stack:");
scanf("%d",&x);
Top=Top+1;
inp_array[Top]=x;
}
}
void Pop()
{
if(Top==-1)
{
printf("\nUnderflow!!");
}
else
{
printf("\nPopped element:  %d",inp_array[Top]);
Top=Top-1;
}
}
void show()
{
if(Top==-1)
{
printf("\nUnderflow!!");
}
```

```
else

{

printf("\nElements present in the stack: \n");

for(int i=Top;i>=0;--i)

printf("%d\n",inp_array[i]);

}

}
```

Application of the Stack

1. A Stack can be used for evaluating expressions consisting of operands and operators.
2. Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
3. It can also be used to convert one form of expression to another form.
4. It can be used for systematic Memory Management.

Advantages of Stack

1. A Stack helps to manage the data in the ‘Last in First out’ method.
2. When the variable is not used outside the function in any program, the Stack can be used.
3. It allows you to control and handle memory allocation and deallocation.
4. It helps to automatically clean up the objects.

Disadvantages of Stack

1. It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
2. It has very limited memory.
3. In Stack, random access is not possible.

1.2 Evaluating arithmetic expressions

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

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 Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations −

Sr.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 -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

a + b * c ⇨ a + (b * c)

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a + b) − c.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Infix to Postfix Conversion

Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix).

We can also convert one type of expression to another type of expression like Infix to Postfix,

Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

- Step 1 - The Operators in the given Infix Expression : **= , + , ***
- Step 2 - The Order of Operators according to their preference : *** , + , =**
- Step 3 - Now, convert the first operator ***** ----- **D = A +** B C *
- Step 4 - Convert the next operator **+** ----- **D =** A BC* +
- Step 5 - Convert the next operator **=** ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.

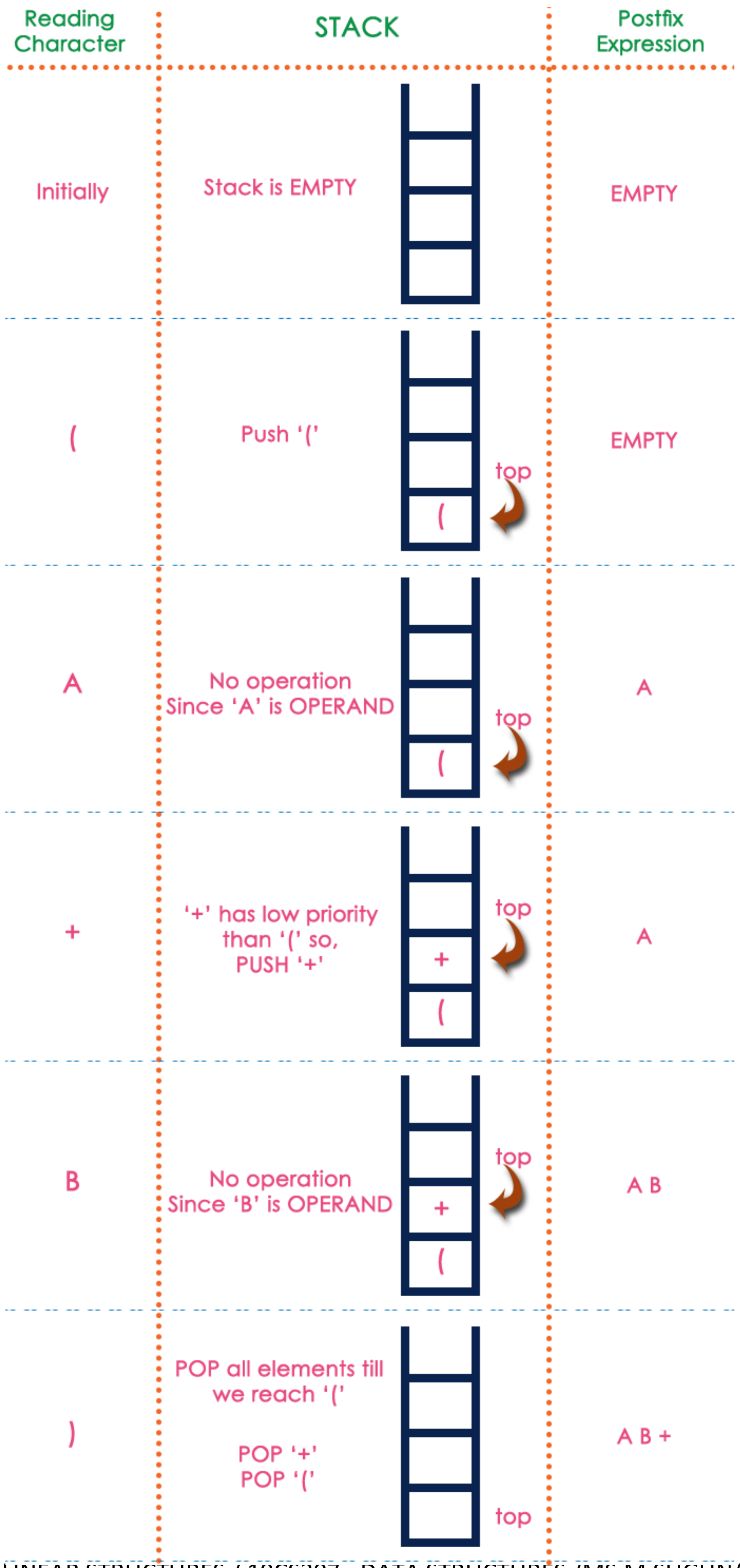
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

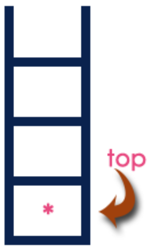
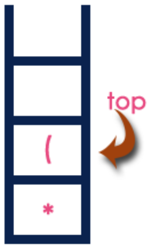
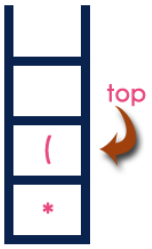
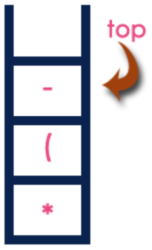
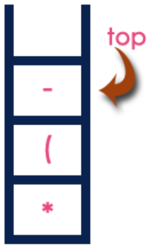
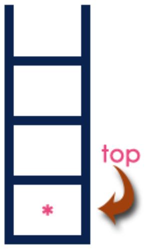
Consider the following Infix Expression...

(A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...



Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push ‘(’	top (	EMPTY
A	No operation Since ‘A’ is OPERAND	top (	A
+	‘+’ has low priority than ‘(’ so, PUSH ‘+’	top + (	A
B	No operation Since ‘B’ is OPERAND	top + (	A B
)	POP all elements till we reach ‘(’ POP ‘+’ POP ‘(’	top	A B +

*	Stack is EMPTY & '*' is Operator PUSH '*'		A B +
(	PUSH '('		A B +
C	No operation Since 'C' is OPERAND		A B + C
-	'-' has low priority than '(' so, PUSH '-'		A B + C
D	No operation Since 'D' is OPERAND		A B + C D
)	POP all elements till we reach '(' POP '-' POP '('		A B + C D -
$	POP all elements till Stack becomes Empty		A B + C D - *

Why postfix notation is more used than prefix?
With prefix, if you push an operator, then its operands, you need to have forward knowledge of when the operator has all its operands. Basically you need to keep track of when operators you've pushed have all their operands so that you can unwind the stack and evaluate.

Since a complex expression will likely end up with many operators on the stack you need to have a data structure that can handle this.

For instance, this expression: - + 10 20 + 30 40 will have one - and one + on the stack at the same time, and for each you need to know if you have the operands available.

With suffix, when you push an operator, the operands are (should) already on the stack, simply pop the operands and evaluate. You only need a stack that can handle operands, and no other data structure is necessary.

Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.
Consider the expression: **a + b * c + d**
The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
The result is then added to d after another scan.
The repeated scanning makes it very inefficient and Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix(or prefix) form before evaluation.
The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

1.2.1 Evaluating postfix representation of the expression using Stack

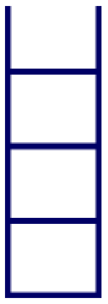
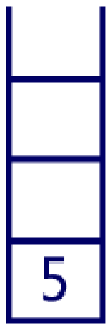
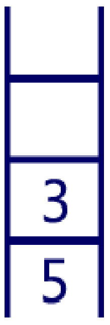
A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

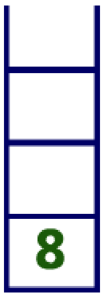
1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (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.

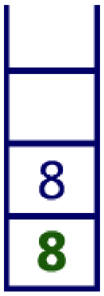
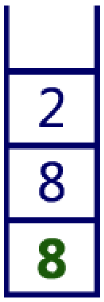
Infix Expression (5 + 3) * (8 - 2)

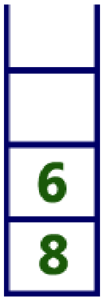
Postfix Expression 5 3 + 8 2 - *

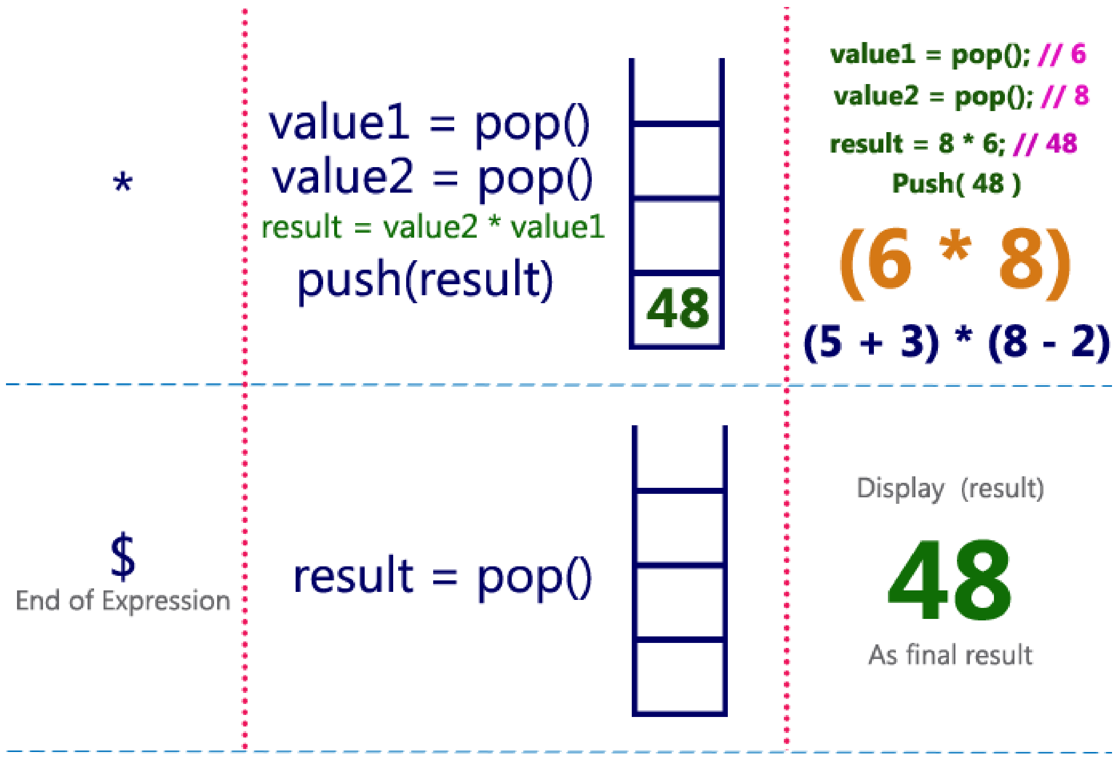
Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
			value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)		(5 + 3)
2	push(2)		(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)



Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48