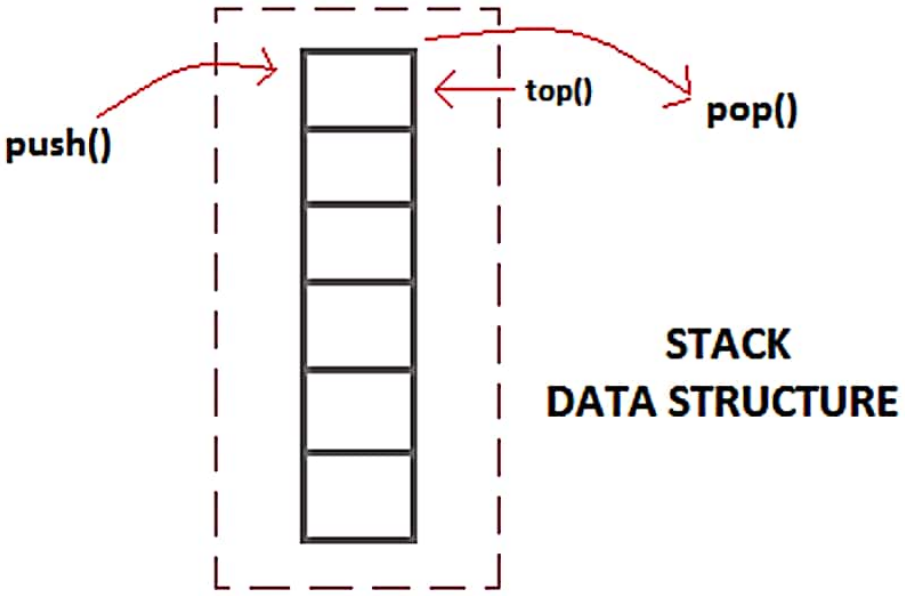
MODULE- II

What is Stack Data Structure?

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

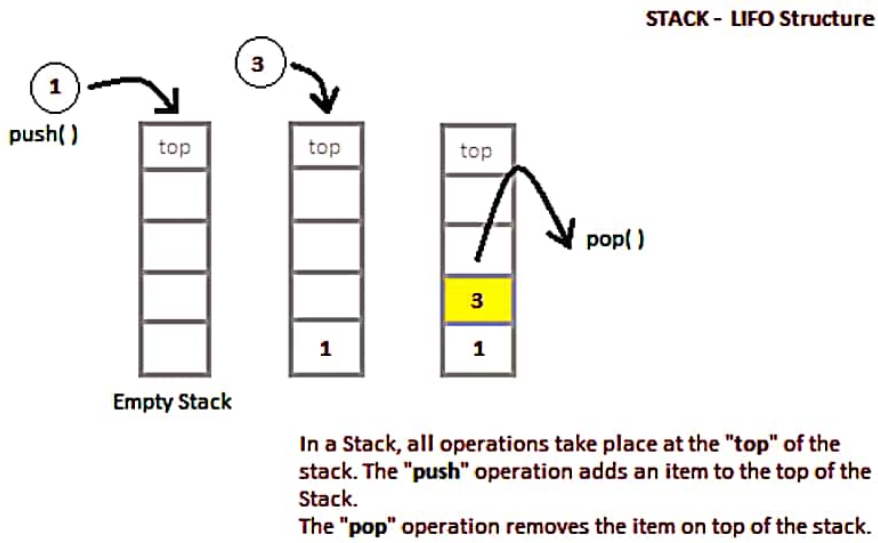
The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

1. Parsing
2. 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. 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.



Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

Conversion & Evaluation of Arithmetic Expressions

Arithmetic expression notations

In any arithmetic expression, each operator is placed in between two operands of it (ie mathematical representation) This way of representing an arithmetic expression is called as infix expression

eg. A+B.

Apart from usual mathematical representation of an arithmetic expression, an expression can also be represented in the following two ways:

1. Polish or Prefix Notation

2. Reverse Polish or Postfix Notation

Polish or Prefix Notation

The notation in which the operator symbol is placed before its operands, is referred as polish or prefix notation.

For Example: +AB. Reverse

Reverse or Postfix Notation

The notation, in which the operator symbol is placed after its operands, is referred as reverse polish or postfix notation.

For Example: AB+

Advantage of Prefix & Postfix over Infix notation is that there no need of parenthesis in the expression

Mathematical Procedure for conversion:

The possible conversions are:

1. **Infix to Prefix**
2. **Prefix to Infix**
3. **Infix to Postfix**
4. **Postfix to Infix**
5. **Prefix to Postfix**
6. **Prefix to Postfix**

Standard Arithmetic Operators and Precedence Levels :

^ (exponential)	Higher Level
***, /, %**	Middle Level
+, -	Lower Level

Infix to Prefix:

> Identify the inner most brackets
> Identify the operator according to the priority of evaluation
> Represent the operator and corresponding operator in prefix notation
> continue this process until the equivalent prefix expression is achieved.

Example : (A + B * (C – D ^ E) / F)

= (A + B * (C – [^ DE]) / F)

= (A + B * [– C ^ DE] / F)

= (A + [* B – C ^ DE] / F)

= A + [/ * B – C ^ DEF]

= + A / * B – C ^ DEF

Prefix to Infix:

> Identify the operator from right to left order
> The two operands which immediately follows the operator are for evaluation
> Represent the operator and operands in infix notation
> Continue this process until the equivalent infix expression is achieved.
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : + A / * B – C ^ DEF

= + A / * B – C [D ^ E] F

= + A / * B [C – D ^ E] F

= + A / [B * (C – D ^ E)] F

= + A [B * (C – D ^ E) / F]

= A + B * (C – D ^ E) / F

Infix to Postfix:

> Identify the inner most brackets
> Identify the operator according to the priority of evaluation
> Represent the operator and corresponding operator in postfix notation
> continue this process until the equivalent postfix expression is achieved.

Example : (A + B * (C – D ^ E) / F)

= (A + B * (C – [DE ^]) / F)

= (A + B * [CDE ^ –] / F)

= (A + [BCDE ^ – *] / F)

= A + [BCDE ^ – *F/]

= ABCDE ^ – *F/ +

Postfix to Infix:

> Identify the operator from left to right order
> The two operands which immediately preceeds the operator are for evaluation
> Represent the operator and operands in infix notation
> Continue this process until the equivalent infix expression is achieved
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : ABCDE ^ – *F/ +

= ABC[D ^ E] – *F/+

= AB [C – D ^ E] * F / +

= A [B *(C – D ^ E)] F / +

= A [B * (C – D ^ E) / F] +

= A + B * (C – D ^ E) / F

Prefix to Postfix:

> Identify the operator from right to left order
> The two operands which immediately follows the operator are for evaluation
> Represent the operator and operands in Postfix notation
> Continue this process until the equivalent Postfix expression is achieved.
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : + A / * B – C ^ DEF

= +A / * B – C [DE ^] F

= + A / * B [CDE ^ –] F

= + A / [BCDE ^ – *] F

= + A [BCDE ^ – * F /]

= ABCDE ^ – * F / +

Postfix to Prefix:

> Identify the operator from left to right order
> The two operands which immediately preceeds the operator are for evaluation
> Represent the operator and operands in prefix notation
> Continue this process until the equivalent prefix expression is achieved
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : ABCDE ^ – *F/ +
= ABC[^ DE] – *F/+
= AB [– C ^ DE] * F / +
= A [* B – C ^ DE] F / +
= A [/ * B – C ^ DE F] +
= + A / * B – C ^ DE F

4.7.3.3 Importance of Postfix Expression

Although human beings are quite used to work with mathematical expression i.e. INFIX notation, which is rather complex as using this notation one has to remember a set of rules. The rules include BODMAS and ASSOCIATIVITY.

In case of POSTFIX notation, the expression, which is easier to work or evaluate the expression as compared to the INFIX expression. In a POSTFIX expression, operands appear before the operators, there is no need to follow the operator precedence and any other rules.

Actually, the processor represents the mathematical expression in postfix expression and uses it for evaluation. To do this it implements the concept of stack.

4.7.3.4 Conversion of an INFIX expression into POSTFIX expression using Stack

Algorithm for converting from INFIX to POSTFIX

[Assume Q is an Infix expression and P is the corresponding Postfix notation.]

Step 1: PUSH '(' onto STACK and Add ')' at the end of Q

Step 2: Repeatedly scan from Q Until STACK is Empty

Step 2.1: If Q[I] is an operand, Then Add it to P[J]
Step 2.2: Else if Q[I] is '(', Then PUSH Q[I] into STACK
Step 2.3: Else if Q[I] is an operator , Then

Step 2.3.1: While STACK[TOP] is an operator and has higher or equal priority compared to Q[I]
Pop operators from STACK and add to P[J]
[End Of While]

Step 2.3.2: Push operator Q [I] onto STACK

Step 2.4: Else if Q[I] is an ')', Then

Step 2.4.1: Repeatedly pop operators from STACK and add to P[J] until '(' is encountered

Step 2.4.2: Remove '(' from STACK

[End Of IF]

[End Of Loop Step – 2]

Step 3: Exit

Procedure of Conversion

e.g. Q = (B * C - (D / E ^ F))

Symbol Scanned from Q	Stack	Postfix Expression P
(	(	
B	(	B
*	(*	B
C	(*	BC
-	(-	BC*
(	(- (	BC*
D	(- (	BC*D
/	(- (/	BC*D
E	(- (/	BC * DE
^	(- (/ ^	BC * DE
F	(- (/ ^	BC * DEF
)	(- (/ ^	BC * DEF ^/
)	(-	BC * DEF ^ / -

- Infix to postfix conversion using stack

Q:- (B*C-(D/E^F))

Q[i]	Stack	P[j]
(	(	
B	(	B
*	(*	B
C	(*	BC
-	(-	BC*
(	(-(	BC*
D	(-(	BC*D
/	(-(/	BC*D
E	(-(/	BC*DE
^	(-(/^	BC*DE
F	(-(/^	BC*DEF
)	(-	BC*DEF^/
)	empty	BC*DEF^/-

Scanned with CamScanner

4.7.3.8 Conversion of infix expresion to prefix expression using stack

[Assume Q is an infix expression. Consider there are two stacks S1 and S2 exist. The following algorithm converts the infix expression Q into its equivalent Prefix notation.]

Algorithm :

Step 1: Add left parenthesis '(' at the beginning of the expression Q

Step 2: PUSH ')' onto Stack S1

Step 3: Repeatedly Scan Q in right to left order, Until Stack S1 is Empty

Step 3.1: If Q[I] is an operand, Then PUSH it onto Stack S2

Step 3.2: Else if Q [I] is ')', Then PUSH it onto Stack S1

Step 3.3: Else if Q [I] is an operator (OP) , Then

Step 3.3.1: Set X := POP (S1)

Step 3.3.2: Repeat while X is an Operator AND (Precedence(X) > Precedence(OP))

PUSH (X) onto Stack S2

Set X := POP (S1)

[End of While – Step 3.3.2]

Step 3.3.3: PUSH (X) onto Stack S1

Step 3.3.4: PUSH (OP) onto Stack S1

Step 3.4: Else if Q [I] is '(' , Then

Step 3.4.1: Set X := POP(S1)

Step 3.4.2: Repeat, While(X != ')') [Until right parenthesis found]

Step 3.4.2.1: PUSH (X) onto Stack S2

Step 3.4.2.2: Set X := POP(S1)

[End of While – Step 3.4.2]

[End of IF – Step 3.1]

[End of Loop – Step 3]

Step 4: Repeat, While Stack S2 is not Empty

Step 4.1: Set X := POP (S2)

Step 4.2: Display X

[End of While]

Step 5: Exit

Procedure of Conversion

e.g. Q = (A + B * C * (M * N ^ P + T) – G + H

Symbol Scanned from Q	Stack S1	Stack S2
H	)	H
+	)+	H
G	)+	HG
-	)+-	HG
)	)+-)	HG
T	)+-)	HGT
+	)+-)+	HGT
P	)+-)+	HGTP
^	)+-)+^	HGTP
N	)+-)+^	HGTPN
*	)+-)+*	HGTPN^
M	)+-)+*	HGTPN^M
(	)+-	HGTPN^M*+
*	)+-*	HGTPN^M*+
C	)+-*	HGTPN^M*+C
*	)+-**	HGTPN^M*+C
B	)+-**	HGTPN^M*+CB
+	)+-+	HGTPN^M*+CB**
A	)+-+	HGTPN^M*+CB**A
(		HGTPN^M*+CB**A+-+

So the Prefix Notation We can get by popping all the symbols from Stack S2.
i.e. +-+A**BC+*M^NPTGH

Infix to prefix using stack

Q = K+L-M*N+(O^P)* W/U/V* T+Q ←

Input (Q)	Stack S_1	Prefix on stack S_2
Q		Q
+	+	
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+**//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWP
(	+*//*	QTVUWP^
+	+*//+	QTVUWP^*//*
N	++	QTVUWP^*//*N
*	++*	QTVUWP^*//*N
M	++*	QTVUWP^*//*NM
-	++-	QTVUWP^*//*NM*
L	++-	QTVUWP^*//*NM*L
+	++-+	QTVUWP^*//*NM*L
K	++-+	QTVUWP^*//*NM*L+-++

← ++-+KL*MN*//*^OPWUV+Q

Q. ((A + B) * C - (D - E) ^ (++G)) .

Q[i]	Stack S_1	Stack S_2
)	)	
)	))	
G	))	G
++	)) ++	G
(	)	G++
^	) ^	G++
)	) ^)	G++
E	) ^)	G++E
-	) ^) -	G++E
D	) ^) -	G++ED
(	) ^	G++ED-
-	) -	G++ED-^
C	) -	G++ED-^C
*	) - *	G++ED-^C
)	) - *)	G++ED-^C
B	) - *)	G++ED-^CB
+	) - *) +	G++ED-^CB
A	) - *) +	G++ED-^CBA
(	) - *	G++ED-^CBA+
(	empty	G++ED-^CBA+*-

- * + A B C ^ - D E ++G

Scanned with CamScanner

4.7.3.6 Evaluation of post fix expression using stack

[Assume P is a post fix expression]

Step 1: Add ')' at the end of P

Step 2: Repeatedly scan P from left to right until ')' encountered

Step 2.1: if P[I] is an operand, then

Push the operand onto STACK

Step 2.2: else If P[I] is an operator ⊗ then

Step 2.2.1: Pop two elements from STACK

(1st element is A and 2nd element is B)

Step 2.2.2: Evaluate B ⊗ A

Step 2.2.3: Push result back to STACK

[End of IF]

[End of loop – Step 2]

Step 3: VALUE: = STACK [TOP]

Step 4: Display VALUE

Step 5: Exit.

Q. Evaluate the following postfix expression:

4 6 2 + * 12 3 / -

P[I]	Stack	A	B	B (op) A
4	[4]			
6	[4 | 6]			
2	[4 | 6 | 2]			
+	[4 | 8]	2	6	6+2 = 8
*	[32]	8	4	4*8 = 32
12	[32 | 12]			
3	[32 | 12 | 3]			
/	[32 | 4]	3	12	12 / 3 = 4
-	[28]	4	32	32-4 = 28

Q

P[I]	Stack	A	B	B(op) A
5	[5]			
6	[5 | 6]			
*	[30]	6	5	6 * 5 = 30
24	[30 | 24]			
2	[30 | 24 | 2]			
3	[30 | 24 | 2 | 3]			
^	[30 | 24 | 8]	3	2	2^3
/	[30 | 3]	8	24	24 / 8 = 3
-	[18]	12	30	30 - 12

18

Scanned with CamScanner

4.7.3.10 Evaluation of prefix expression

[Assume P is a Prefix Expression]

Step 1: Add '(' at the beginning of the prefix expression
Step 2: Repeatedly scan from P in right to left order until '(' encountered
    Step 2.1: If P [I] is an operand, then
        PUSH the operand onto STACK
    Step 2.2: Else if P [I] is operator (OP), then
        Step 2.2.1: Pop two elements from STACK
        (1st element is A and 2nd element is B)
        Step 2.2.2: Evaluate A (OP) B
        Step 2.2.3: Push result back to STACK
        [End of if]
    [End of loop – Step 2]
Step 3: VALUE: = STACK [TOP]
Step 4: Display VALUE
Step 5: Exit.

For Example :

Evaluate the following Prefix Expression using stack :

P = (- , * , 3 , + , 16 , 2 , / , 12 , 6

Symbol Scanned from Prefix Expression in Right to Left Order	Stack	A	B	A (OP) B
6	6			
12	6 , 12			
/	2	12	6	12 / 6 = 2
2	2 , 2			
16	2 , 2 , 16			
+	2 , 18	16	2	16 + 2 = 18
3	2 , 18 , 3			
*	2 , 54	3	18	3 * 18 = 54
-	52	54	2	54 – 2 = 52

Finally the value in the Stack is 52.