

# UNIT- TWO

## THE STACK

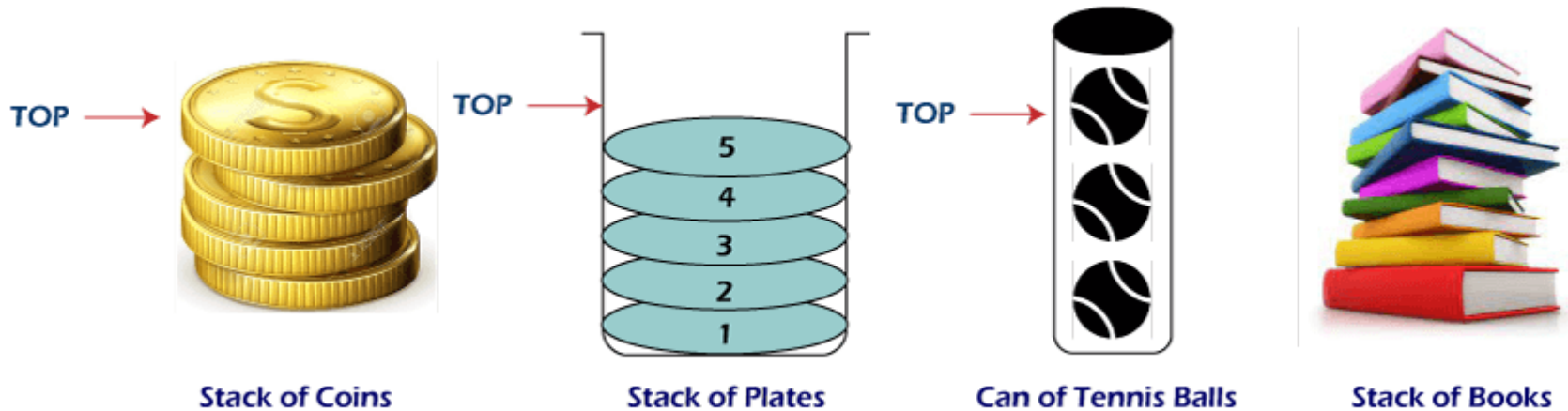
# Topics to be Covered

- Introduction
- Stack as an ADT
- POP and PUSH Operation
- Stack Application
  - Evaluation of Infix, Postfix and Prefix Expressions
  - Conversion of Expression

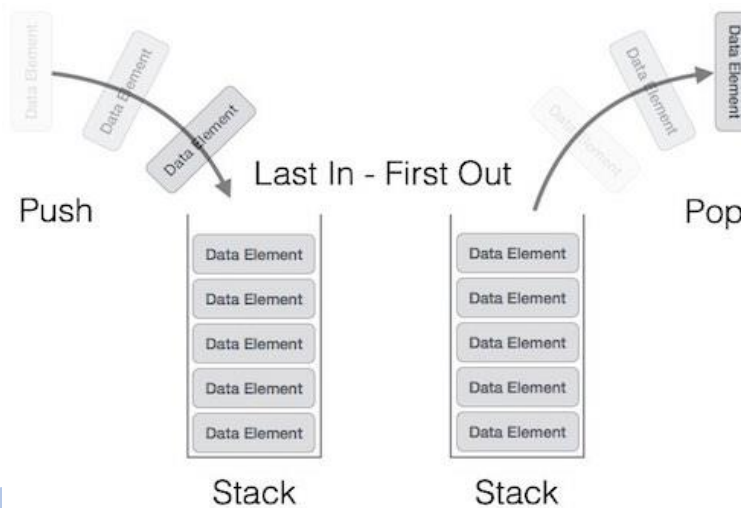
## Introduction: The Stack

- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.
- Stack is an ordered collection of homogenous data elements where insertion and deletion can be performed only at one end that is called ***top of stack (tos)***.
- Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- 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.

- Example – Stack in real world representation



## Stack Representation



# Stack as an ADT

Some operations or functions of the Stack ADT.

- `isFull()`: This is used to check whether stack is full or not
- `isEmpty()`: This is used to check whether stack is empty or not
- `push(x)`: This is used to push `x` into the stack
- `pop()`: This is used to delete one element from top of the stack
- `peek()`: This is used to get the top most element of the stack
- `size()`: this function is used to get number of elements present into the stack

## Basic Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –
- **push()** – Pushing (storing) an element on the top of the stack.
- **pop()** – Removing (accessing) an element from the top of the stack.

## Push() Operation

- The process of adding a new element to the top of the stack
- After inserting an element, max size is reached, i.e, stack is full and if more element is to be inserted, it is not possible. This situation is called stack overflow condition.
- At this point the stack is at the highest location of the stack.

*If  $TOP = MAX\ SIZE - 1$  then STACK IS OVERFLOW*

## Algorithm

1. START
2. if  $TOP = MAX - 1$  then  
write "Stack overflow" and STOP
2. READ DATA
3.  $TOP \leftarrow TOP + 1$
4.  $STACK[TOP] \leftarrow DATA$
5. STOP

## POP() Operation

- The process of deleting an element from the top of stack.
- After every pop operation the stack is decremented by 1.
- When all elements are deleted, TOP points to the bottom of the stack, after which- no any element can be deleted. This situation is called stack underflow.

*If TOP= -1 then STACK IS UNDERFLOW*

## Algorithm

1. START
2. if  $TOP < 0$  then  
    write "Stack underflow" and STOP
3.  $STACK[TOP] \leftarrow NULL$
4.  $TOP \leftarrow TOP - 1$
5. STOP



## C implementation of Push Pop operation using array

<pre>#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt; #define max 20 int stack[max],top=-1,num; void push(); void pop(); void display(); int main() {     int ch;     do     {         printf("\n 1 for push 2 for pop 3 for display             and 0 to exit :");         scanf("%d",&amp;ch);         switch(ch)         {             case 1: push();                     break;             case 2: pop();                     break;             case 3: display();                     break;             default: exit(0);         }     }while(ch);     return 0; }</pre>	<pre>void push() {     if(top==max-1)     {         printf("stack overflow");         return 0;     }     top++;     printf("\nEnter number");     scanf("%d",&amp;num);     stack[top]=num;     printf("\n Elements in stack");     display(); }</pre>	<pre>void pop() {     if(top== -1)     {         printf("Stack is underflow");         return 0;     }     else     {         top--;         printf("\nElements in stack");         display();     } }  void display() {     int i;     for(i=top;i&gt;=0;i--)     {         printf("\n %d",stack[i]);     } }</pre>
--	---	--

## Application of Stack

- **Expression Handling –**

- **Infix to Postfix or Infix to Prefix Conversion –**

- The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions.
    - These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

- **Postfix or Prefix Evaluation –**

- After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

- **Backtracking Procedure –**

- Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

# Data Structure- Expression Parsing

- 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.

# Infix Notation

- An infix notation is a notation in which an expression is written in a usual or normal format. It is a notation in which the operators lie between the operands. The examples of infix notation are  $A+B$ ,  $A*B$ ,  $A/B$ , etc.
- As we can see in the above examples, all the operators exist between the operands, so they are infix notations. Therefore, the syntax of infix notation can be written as:

**<operand> <operator> <operand>**

- 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.


## ■ Parsing Infix Notation:

- In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**. Operator precedence means the precedence of any operator over another operator. For example:

Expression:  $A + B * C \rightarrow A + (B * C)$

As the multiplication operator has a higher precedence over the addition operator so  $B * C$  expression will be evaluated first. The result of the multiplication of  $B * C$  is added to the  $A$ .

Operators	Symbols	Associativity
Parenthesis	{ }, ( ), [ ]	Left to Right
Exponential notation	$\wedge$	Right to left
Multiplication and Division	$*, /$	Left to Right
Addition and Subtraction	$+, -$	Left to Right



## Example: Illustration of Precedence and Associativity

Q1.  $2+5*3-12+3*2$

Q2.  $2*(5+3)-12+2*(6-2)$

Q3.  $2*5^2-25+2*(2*3^2)$

# Postfix Notation

- In this notation, operator is postfixed to operands, i.e. operator is written after operands. It is also known as **Reversed polish notation**. In postfix notation, an operator comes after operands. The syntax of postfix notation is given below:

**<operand> <operand> <operator>**

- **For example**, if the infix expression is  $5+1$ , then the postfix expression corresponding to this infix expression is  $51+$ .

## Example:

Q1.  $(a + b) * c$

Solution:  $\Rightarrow ab +$   
 $\Rightarrow ab + c *$

Q2.  $(a + b) * (c + d)$

Q3.  $A * B + C - (D^E) / F$



# Infix to Postfix conversion using Stack

## Algorithm:

1. Scan the infix string from left to right.
2. Initialize an empty stack.
3. If the scanned character is an operand, add it to postfix string
4. If the scanned character is an operator, PUSH the character to stack.
5. If the **top operator in stack** has equal or higher precedence than scanned operator then POP the operator present in stack and add it to postfix string else PUSH the scanned character to stack.
6. If the scanned character is a left parenthesis '(', PUSH it to stack.
7. If the scanned character is a right parenthesis ')', POP and add to postfix string from stack until an '(' is encountered. Ignore both '(' and ')'.  
8. Repeat step 3-7 till all the characters are scanned.
9. After all character are scanned POP the characters and add to postfix string from the stack until it is not empty.

- Let infix expression be:  $M * N + (O^P) * W/U/V * T + Q$

- Solution:**

Input Expression	Stack Content	Postfix Expression
M		M
*	*	M
N	*	MN
+	+	MN*
(	+(	MN*
O	+(	MN*O
^	+(^	MN*O
P	+(^	MN*OP
)	+	MN*OP^
*	+*	MN*OP^
W	+*	MN*OP^W
/	+/	MN*OP^W*
U	+/	MN*OP^W*U
/	+/	MN*OP^W*U/
V	+/	MN*OP^W*U/V
*	+*	MN*OP^W*U/V/
T	+*	MN*OP^W*U/V/T
+	+	MN*OP^W*U/V/T* MN*OP^W*U/V/T*+
Q	+	MN*OP^W*U/V/T*+Q
		MN*OP^W*U/V/T*+Q+

- Practice Session: Convert the following infix expression to Postfix Expression using stack.

1.  $2*3/(2-1*7)+5*(4-1)$

2.  $(A-B/C)*(D*E-F^G)$

3.  $X*Y+P^Q(M/N)-D$

# Prefix Notation

- In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands. It is also known as **polish notation**. In prefix notation, an operator comes before the operands. The syntax of prefix notation is given below:

**<operator> <operand> <operand>**

- Prefix notation is also known as **Polish Notation**.
- **For example**, if the infix expression is  $5+1$ , then the prefix expression corresponding to this infix expression is  $+51$ .

## Example:

Q1.  $A/B^C-D$

Solution:  $\Rightarrow A/^BC-D$

$\Rightarrow /A^BC-D$

$\Rightarrow -/A^BCD$

Q2.  $(A-B/C)*(D^E-F)$

Q3.  $A*B+C-(D^E)/F$

# Infix to Prefix conversion using Stack

## Algorithm:

1. Reverse the input string.
2. Examine the element in the input string.
3. If it is operand, then place it in the output string.
4. If input is an operator, push it into the stack.
5. If the incoming operator has equal or higher precedence than input operator, then push it to the stack .
6. If the input is a close parenthesis, push it into the stack.
7. If the input is open parenthesis, pop elements in stack one by one until we encounter close parenthesis. Discard parenthesis while writing to output string.
8. If there is more input go to step 2.

After completing all processing, reverse the data in output string and the result will be our prefix notation of the given expression.

- Let infix expression be:  $M * N + (O^P) * W/U/V * T + Q$

- **Solution:**

- If we are converting the expression from infix to prefix, we need first to reverse the expression.

- The Reverse expression would be:

$$Q + T * V/U/W * ) P^O(+ N * M$$

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input Expression	Stack Content	Prefix Expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
		QTVUWPO^*//*NM*++

Reversing again to get prefix expression: ++\*MN\*//\*^OPWUVTQ



- Practice Session: Convert the following infix expression to Prefix Expression using stack.

1.  $2*3/(2-1*7)+5*(4-1)$

1.  $(A-B/C)*(D*E-F^G)$

2.  $X*Y+P^Q(M/N)-D$

## • Postfix Expression Evaluation:

Algorithm:

1. Read the expression from left to right
2. Push the element in the stack if it is an operand.
3. If the element is an operator, Pop the two operands from the stack and then evaluate it.( remember, firstly popped operand will be 2<sup>nd</sup> operand and vice versa)
4. Push back the result of the evaluation.
5. Repeat it till the end of the expression.
6. Pop out the result from stack

- Let infix expression be:  $2*3/(2-1)+5*(4-1)$

It's equivalent postfix expression is:  $23*21-/541-*+$

Input Expression	Operation	Stack Content	Operand1	Operand2	Calculation <operand1> <operator> <operand2>
2	Push	2			
3	push	2,3			
*	Pop two elements , evaluate and push back result	6	2	3	$2*3$
2	push	6,2			
1	push	6,2,1			
-	Pop two elements , evaluate and push back result	6,1	2	1	$2-1$
/	Pop two elements , evaluate and push back result	6	6	1	$6/1$
5	push	6,5			
4	push	6,5,4			
1	push	6,5,4,1			
-	Pop two elements , evaluate and push back result	6,5,3	4	1	$4-1$
*	Pop two elements , evaluate and push back result	6,15	5	3	$5*3$
+	Pop two elements , evaluate and push back result	21	6	15	$6+15$

## Practice:

1.  $53 + 82 - *$

2.  $574 - * 84 / +$

3.  $3653 - * 31 + / 222 \wedge \wedge -$

## • Prefix Expression Evaluation:

Algorithm:

1. Reverse the prefix expression
2. Read the expression from left to right
3. Push the element in the stack if it is an operand.
4. Pop the two operands from the stack, if the element is an operator and then evaluate it.
5. Push back the result of the evaluation.
6. Repeat it till the end of the expression.
7. Pop out the result from stack

- Let infix expression be:  $2*3/(2-1)+5*(4-1)$
- Prefix expression:  $+/*23-21*5-41$
- Reverse:  $14-5*12-32*/+$

Input Expression	Operation	Stack Content	Operand1	Operand2	Calculation <operand1> <operator> <operand2>
1	Push	1			
4	push	1,4			
-	Pop two elements , evaluate and push back result	3	4	1	4-1
5	push	3,5			
*	Pop two elements , evaluate and push back result	15	5	3	5*3
1		15,1			
2	Pop two elements , evaluate and push back result	15,1,2			
-	push	15,1	2	1	2-1
3	push	15,1,3			
2	push	15,1,3,2			
*	Pop two elements , evaluate and push back result	15,1,6	3	2	3*2
/	Pop two elements , evaluate and push back result	15,6	6	1	6/1
+	Pop two elements , evaluate and push back result	21	6	15	6+15

Any Queries??