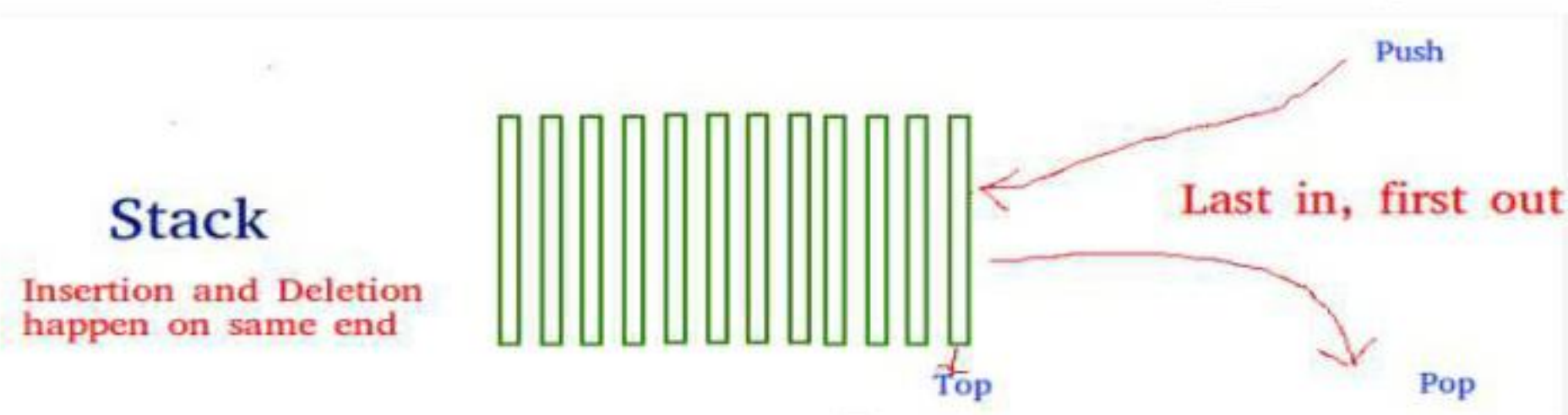


## 2 The Stack

- A stack is a non-primitive linear data structure.
- It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end, known as **top of stack (TOS)**.
- The most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
- A stack is said to be empty or underflow, if the stack contains no elements.
- At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.
- Unlike arrays, access of elements in a stack is restricted.
- It has two main function **PUSH** and **POP**.
- Insertion in a stack is done using **push** function and removal from a stack is done using **pop** function.
- Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack.
- It is therefore, also called **Last-In-First-Out (LIFO)**.



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added (pushed), each new plate becomes the top of the stack, and hides each plate below. As plates are removed (popped) from the stack, they can be used, and second plate becomes the top of the stack. Two important principles are illustrated by this metaphor, the Last-In-First-Out principle is one. The second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed.

### 2.1 Stack Operations

**PUSH operation:** The push operation is used to add (or push or insert) elements in a stack

- When we add an item to a stack, we say that we push it onto the stack
- The last item put into the stack is at the top

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
	6
	5
23	4
15	3
22	2
41	1
34	0

*Before PUSH*  
(top=4, count=5)

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
15	5
23	4
15	3
22	2
41	1
34	0

*After PUSH*  
(top=5, count= 6)



**POP operation:** The pop operation is used to remove or delete the top element from the stack.

- When we remove an item, we say that we pop it from the stack
- When an item popped, it is always the top item which is removed

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
	6
	5
23	4
15	3
22	2
41	1
34	0

*Before POP*  
(top=4, count=5)

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
	5
	4
15	3
22	2
41	1
34	0

*After POP*  
(top=3 count=4)

The **PUSH** and the **POP** operations are the **basic or primitive** operations on a stack. Some others operations are:

- **CreateEmptyStack operation:** This operation is used to create an empty stack.
- **IsFull operation:** The isfull operation is used to check whether the stack is full or not ( i.e. stack overflow)
- **IsEmpty operation:** The isempty operation is used to check whether the stack is empty or not. (i. e. stack underflow)
- **Top operations:** This operation returns the current item at the top of the stack, it doesn't remove it

### ✚ The Stack ADT:

A stack of elements of type T is a finite sequence of elements of T together with the operations

**CreateEmptyStack(S):** Create or make stack S be an empty stack

**Push(S, x):** Insert x at one end of the stack, called its top

**Top(S):** If stack S is not empty; then retrieve the element at its top

**Pop(S):** If stack S is not empty; then delete the element at its top

**IsFull(S):** Determine if S is full or not. Return true if S is full stack; return false otherwise

**IsEmpty(S):** Determine if S is empty or not. Return true if S is an empty stack; return false otherwise.

#### Stack ADT

**Definition** A list of data items that can only be accessed at one end, called the *top* of the stack.

**Operations**

- stack:** Creates an empty stack.
- push:** Inserts an element at the top.
- pop:** Deletes the top element.
- empty:** Checks the status of the stack.

## 2.2 Stack Implementation

Stack can be implemented in two ways:

- Array Implementation of stack (or static implementation)
- Linked list implementation of stack (or dynamic implementation)



**2.2.1 Array (static) implementation of a stack:**

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1 (top=-1).

```
#define maxsize 10
struct stack
{
    int items[maxsize]; //Declaring an array to store items
    int top;             //Top of a stack
};
typedef struct stack STACK;
```

**Creating Empty stack :**

The value of top=-1 indicates the empty stack in C implementation.

```
/*Function to create an empty stack*/
void initStack(STACK *s)
{
    s->top=-1;
}
```

**Stack Empty or Underflow:**

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int isempty(STACK *s)
{
    if(s->top==-1)
        return 1;
    else
        return 0;
}
```

**Stack Full or Overflow:**

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (maxsize-1) of the stack. The following function returns true (1) if stack is full false (0) otherwise.

```
int isfull(STACK *s)
{
    if(s->top==maxsize-1)
        return 1;
    else
        return 0;
}
```

### ✚ Algorithm for PUSH and POP operations on Stack

Let `items[maxsize]` be an array to implement the stack. The variable `top` denotes the top of the stack

#### Algorithm for PUSH (inserting an item into the stack) operation:

This algorithm adds or inserts an item at the top of the stack

Step 1: [Check for stack overflow?]

if `top==maxsize-1` then

print "Stack Overflow" and Exit

else

Set `top=top+1` [Increase top by 1]

Set `items[top] = data` [Inserts data in new top position]

Step 2: Exit

#### Algorithm for POP (removing an item from the stack) operation

This algorithm deletes the top element of the stack and assign it to a variable item

Step 1: [Check for the stack Underflow]

If `top<0` then

Print "Stack Underflow" and Exit

else

[Remove the top element]

Set `data = items [top]`

[Decrement top by 1]

Set `top=top-1`

Return the deleted item from the stack

Step 2: Exit

### ✚ Representing Stack in C

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define maxsize 50
struct stack
{
    int top;
    int items[maxsize];
};
typedef struct stack STACK;

void initStack(STACK *s)
{
    s->top = -1;
}

int isEmpty(STACK *s)
{
    if(s->top == -1)
```



```
        return 1;
    else
        return 0;
}
int isFull(STACK *s)
{
    if(s->top==maxsize-1)
        return 1;
    else
        return 0;
}
void push(STACK *s, int data)
{
    if(isFull(s) == 1)
    {
        printf("Data overflow.");
        exit(1);
    }
    else
    {
        (s->top)++;
        s->items[s->top] = data;
    }
}
int pop(STACK *s)
{
    int v;
    if(isEmpty(s)==1)
    {
        printf("Data Underflow");
        exit(1);
    }
    else
    {
        v = s->items[s->top];
        (s->top)--;
        return v;
    }
}
int stackTop(STACK *s)
{
    if(isEmpty(s)==1)
    {
        printf("Underflow");
        exit(1);
    }
    else
    {
        int v = s->items[s->top];
        return v;
    }
}
void printStack(STACK *s)
{

```

```

    int i;
    for(i=0;i<=s->top;i++)
    {
        printf("\n%d",s->items[i]);
    }
}

int main()
{
    int element,choice;
    int flag=1;
    STACK st;
    STACK *s;
    s=&st;
    initStack(s);    /* s->top=-1; indicates empty stack */
    do
    {
        printf("\n\n Enter your choice");
        printf(" \n\n\t 1:Push the elements");
        printf(" \n\n\t 2: To display the elements");
        printf(" \n\n\t 3: Pop the element");
        printf(" \n\n\t 4: To check top element");
        printf(" \n\n\t 5: Exit");
        printf("\n\n\n Enter of your choice:\t");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("\n Enter the number:");
                scanf("%d", &element);          /*Read an element from keyboard*/
                push(s,element);
                break;
            case 2:
                printf("Data Elements in Stack are:")
                printStack(s);
                break;
            case 3:
                element = pop(s);
                printf("\nPoped Element is %d",element);
                break;
            case 4:
                element = stackTop(s);
                printf("\nTop Element is %d",element);
                break;
            case 5:
                flag=0;
                break;
            default:
                printf("\n Invalid Choice");
        }
    }while(flag);
    getch();
    return 0;
}

```



## 2.3 Applications of Stack

Stack is used directly and indirectly in the following fields:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Conversion of an infix expression into a prefix expression
- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

### 2.3.1 Expression

An expression is defined as the number of operands or data items combined with several operators. One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- **Infix notation**
- **Prefix notation**
- **Postfix notation**

**Infix expression :** It is an ordinary mathematical notation of expression where operator is written in between the operands. Example:  $A+B$ . Here '+' is an operator and A and B are called operands

**Prefix notation:** In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. Example:  $+AB$

**Postfix notation :** In postfix notation the operators are written after the operands so it is called the postfix notation (post means after). In this notation the operator follows the two operands. Example:  $AB+$

Examples:

$A + B$  (Infix)  
 $+ AB$  (Prefix)  
 $AB +$  (Postfix)

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:

**add(A, B)**

Note that the operator add (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

In a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

### Operator Precedence

Exponential operator	$^$	Highest precedence (Note : $^ = \$$ )
Multiplication/Division	$*, /$	Next precedence
Addition/Subtraction	$+, -$	Least precedence



- You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated
- Both prefix and postfix are parenthesis free expressions. For example

$(A + B) * C$  Infix form

$* + A B C$  Prefix form

$A B + C *$  Postfix form

Infix	Postfix	Prefix
$A \mid B$	$AB \mid$	$\mid AB$
$A \mid B - C$	$AB \mid C -$	$- \mid ABC$
$(A \mid B) * (C - D)$	$AB \mid CD - *$	$* \mid AB - CD$

### 2.3.1.1 Converting an Infix Expression to Postfix

- First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.
- The method of converting infix expression  $A + B * C$  to postfix form is:

$A + B * C$  Infix Form

$A + (B * C)$  Parenthesized expression

$A + (B C *)$  Convert the multiplication

$A (B C *) +$  Convert the addition

$A B C * +$  Postfix form

- The rules to be remembered during infix to postfix conversion are:
  - Parenthesize the expression starting from left to right.
  - During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression  $B * C$  is parenthesized first before  $A + B$ .
  - The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
  - Once the expression is converted to postfix form, remove the parenthesis.

**Example:**

$(A + B) * ((C - D) + E) / F$  Infix form

$(AB+) * ((C - D) + E) / F$

$(AB+) * ((CD-) + E) / F$

$(AB+) * (CD-E+) / F$

$(AB+CD-E+*) / F$

$AB+CD-E+*F/$  Postfix form

**Example:** (Convert the infix expression listed in the table into postfix notation and verify yourself.)

<i>Infix</i>	<i>Postfix</i>
$A + B$	$AB +$
$A + B - C$	$AB + C -$
$(A + B) * (C - D)$	$AB + CD - *$
$A \$ B * C - D - E / F / (G + H)$	$AB \$ C * D - EF / GH + / +$
$((A + B) * C - (D - E)) \$ (F + G)$	$AB + C * DE - - FG + \$$
$A - B / (C * D \$ E)$	$ABCDE \$ * / -$



### 2.3.1.2 Converting an Infix expression to Prefix expression

The precedence rule for converting from an expression from infix to prefix are identical. Only changes from postfix conversion is that the operator is placed before the operands rather than after them. The prefix of

$$\begin{aligned} &A+B-C \text{ (infix)} \\ &= (+AB)-C \\ &= -+ABC \text{ (prefix)} \end{aligned}$$

Example Consider an example:

$$\begin{aligned} &A \$ B * C - D + E / F / (G + H) \\ &= A \$ B * C - D + E / F / (+GH) \\ &= \$AB * C - D + E / F / (+GH) \\ &= *\$ABC - D + E / F / (+GH) \\ &= *\$ABC - D + (/EF) / (+GH) \\ &= *\$ABC - D + //EF + GH \\ &= (- *\$ABCD) + (/EF + GH) \\ &= + - *\$ABCD //EF + GH \end{aligned}$$

infix form

which is in prefix form.

### 2.3.1.3 Algorithm to convert Infix to Postfix notation:

#### POSTFIX( Q, P)

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**. Besides operands and operators; **Q** (infix notation) may also contain left and right parentheses. We assume that the operators in **Q** consists of only exponential (  $^$  ), multiplication (  $*$  ), division (  $/$  ), addition (  $+$  ) and subtraction (  $-$  ). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression **P** will be constructed from left to right using the operands from **Q** and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of **Q**. The algorithm is completed when the stack is empty.

1. Push "(" onto stack, and add ")" to the end of **Q**.
2. Scan **Q** from left to right and repeat Steps 3 to 6 for each element of **Q** until the stack is empty.
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator  $\otimes$  is encountered, then:
  - a Repeatedly pop from stack and add **P** each operator (on the top of stack), which has the same precedence as, or higher precedence than  $\otimes$ .
  - b Add  $\otimes$  to stack. [End of If structure]
6. If a right parenthesis is encountered, then:
  - a Repeatedly pop from stack and add to **P** (on the top of stack until a left parenthesis is encountered.
  - b Remove the left parenthesis. [Do not add the left parenthesis to **P**.]

[End of If structure]

[End of step 2 loop]
7. Exit

**Note:** Special character  $\otimes$  is used to symbolize any operator in **P**.



**Example:** The following tracing of the algorithm illustrates the algorithm. Consider an infix expression

$$((A-(B+C))*D)\$(E+F)$$

Scan character	Stack	Postfix String
(	(	.....
(	((	.....
A	((	A
-	(( -	A
(	(( -(	A
B	(( -(	AB
+	(( -( +	AB
C	(( -( +	ABC
)	(( -	ABC+
)	(	ABC+-
*	(*	ABC+-
D	(*	ABC+-D
)	.....	ABC+-D*
\$	\$	ABC+-D*
(	\$(	ABC+-D*
E	\$(	ABC+-D*E
+	\$(+	ABC+-D*E
F	\$(+	ABC+-D*EF
)	\$	ABC+-D*EF+
.....	.....	ABC+-D*EF+\$ (postfix)

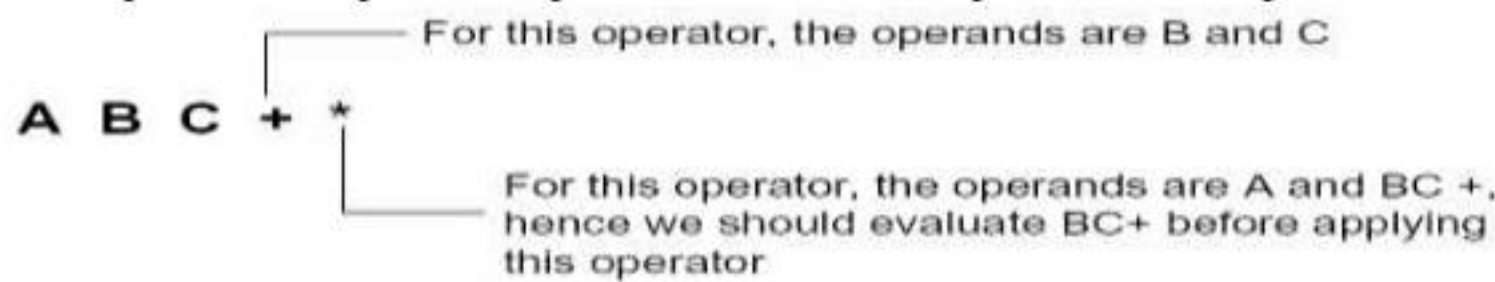
**Example:** Consider the Following Infix String:  $a+b*c+(d*e+f)*g$

Character scanned	Operator Stack	Postfix String
a		a
+	+	a
b	+	ab
*	+ *	ab
c	+ *	abc
+	+	abc * +
(	+ (	abc * +
d	+ (	abc * + d
*	+ ( *	abc * + d
e	+ ( *	abc * + de
+	+ ( +	abc * + de *
f	+ ( +	abc * + de * f
)	+	abc * + de * f +
*	+ *	abc * + de * f +
g	+ *	abc * + de * f + g
		abc * + de * f + g * +



### 2.3.1.4 Evaluating the Postfix expression

- Each operator in a postfix expression refers to the previous two operands in the expression.



- To evaluate the postfix expression we use the following procedure:  
Each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.
- Consider an example  

$$3\ 4\ 5\ *\ +$$

$$= 3\ 20\ +$$

$$= 23\ (\text{answer})$$

### 2.3.1.5 Algorithm to Evaluate a Postfix expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

#### Algorithm

1. Add a right parenthesis “)” at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel “)” is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator  $\otimes$  is encountered, then:
  - a Remove the two top elements of STACK, where A is the top element and B is the next to-top element.
  - b Evaluate  $B \otimes A$ .
  - c Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

#### Trace of Evaluation:

Consider an example to evaluate the postfix expression tracing the algorithm (A = 1, B = 2, C = 3)

ABC+\*CBA-+\*

123+\*321-+\*

Scanned character	value	Op2	Op1	Result	Stack
A	1	.....	.....	.....	1
B	2	.....	.....	.....	1 2
C	3	.....	.....	.....	1 2 3
+	.....	3	2	5	1 5
*	.....	5	1	5	5
C	3	.....	.....	.....	5 3
B	2	...	.....	.....	5 3 2
A	1	.....	.....	.....	5 3 2 1
-	.....	1	2	1	5 3 1
+	.....	1	3	4	5 4
*	.....	4	5	20	20



### 2.3.1.6 Evaluating the Prefix Expression

To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms. One stack is used to store operators and another is used to store the operands. Consider an example for this

+ 5 \* 3 2 prefix expression  
 = + 5 6  
 = 11

Illustration: Evaluate the given prefix expression

/ + 5 3 - 4 2 prefix equivalent to (5+3)/(4-2) infix notation  
 = / 8 - 4 2  
 = / 8 2  
 = 4

**Algorithm:**

```

Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
    (a) Scan the prefix expression
        from right, one character at a
        time.
    (b) If the scanned character is an
        operand, push it on the
        operand stack.
    (c) If the scanned character is an
        operator, then
        (i) Pop two values from the
            operand stack
        (ii) Apply the operator on
            the popped operands
        (iii) Push the result on the
            operand stack
Step 3: END
  
```

- For example, consider the prefix expression + - 9 2 7 \* 8 / 4 12. Let us now apply the algorithm to evaluate this expression

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29