Data Structure-Stack

Kaustuv Bhattacharjee
University of Engineering & Management, Kolkata

Stack-Introduction

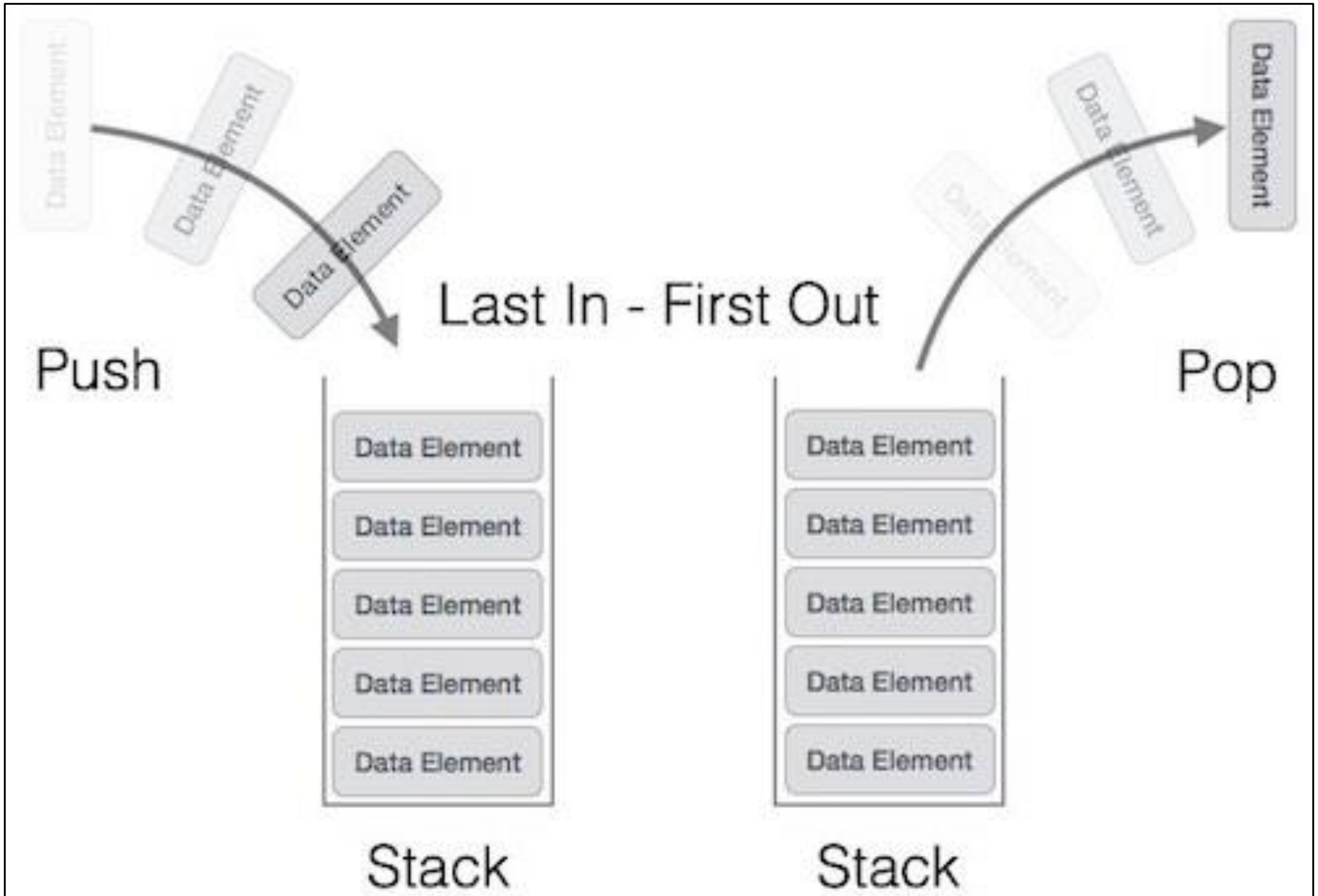
- Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called *top*.
- Stack is a recursive data structure having pointer to its top element.
- Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Stack-Introduction

- Stacks can be implemented using either arrays or linked lists

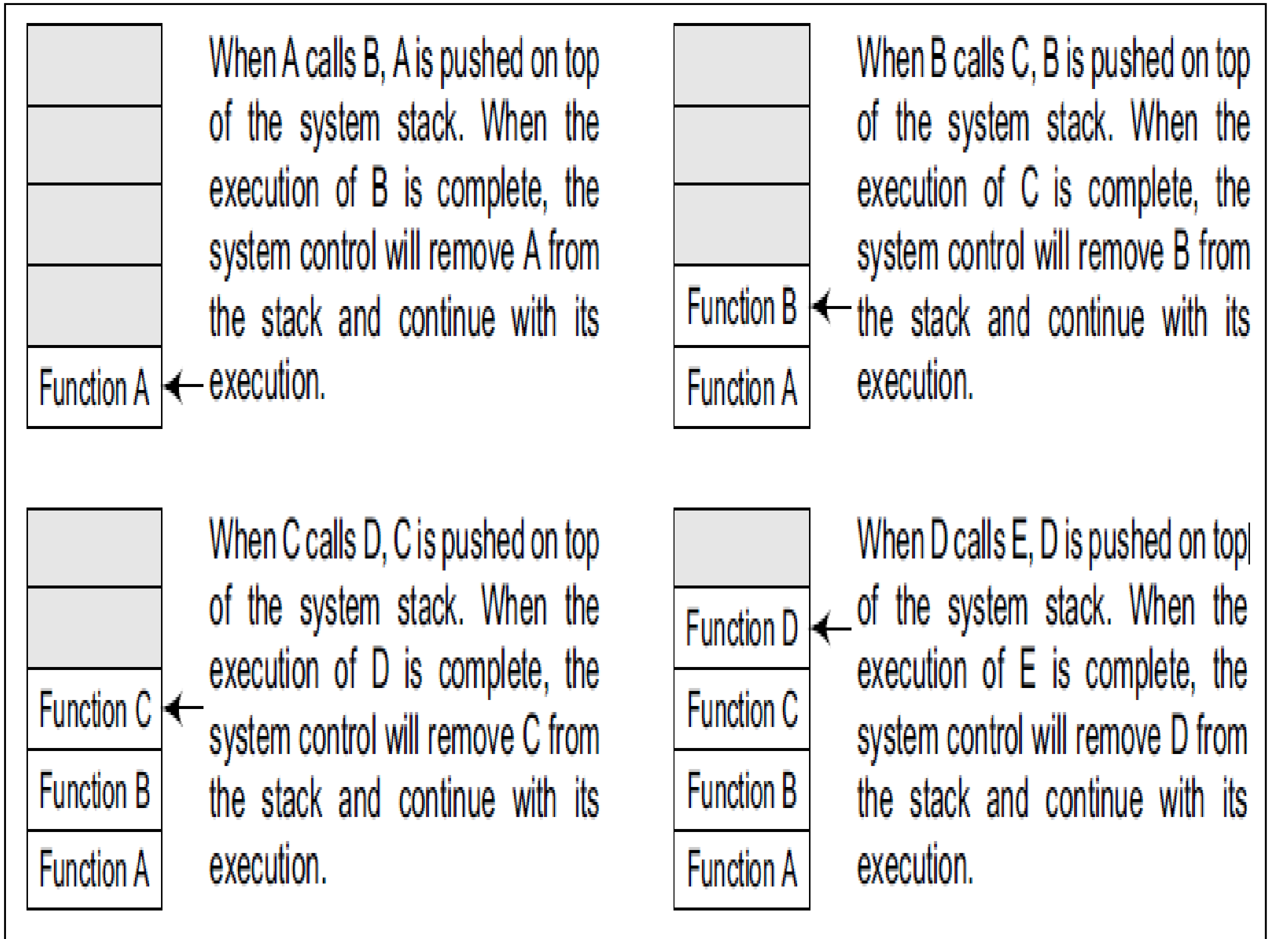
- There are three operations which can be performed on stack:
  - **Push:** Adding an element onto the stack (on the top)
  - **POP:** Removing an element from the stack (from the top)
  - **Peek:** Returns the value of the topmost element of the stack without removing it from the stack

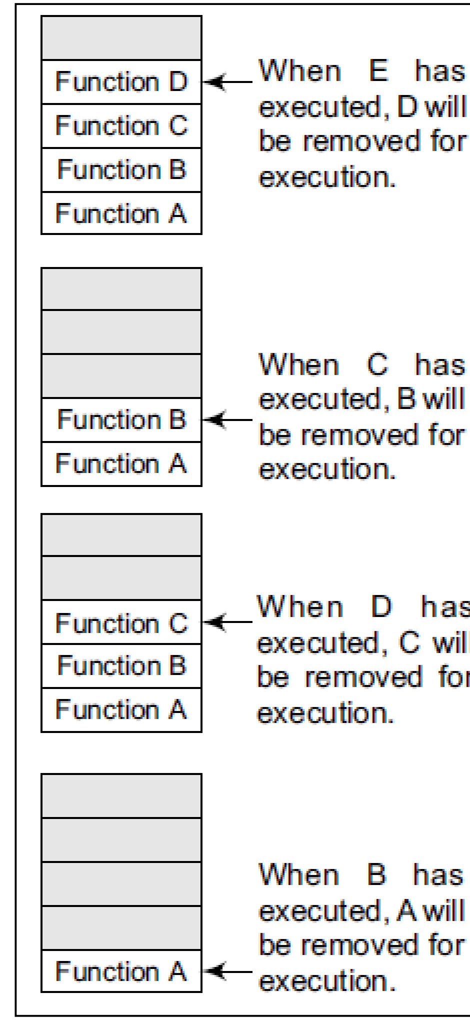
Stack-Pictorial Representation



Stack-Why in Computer Science

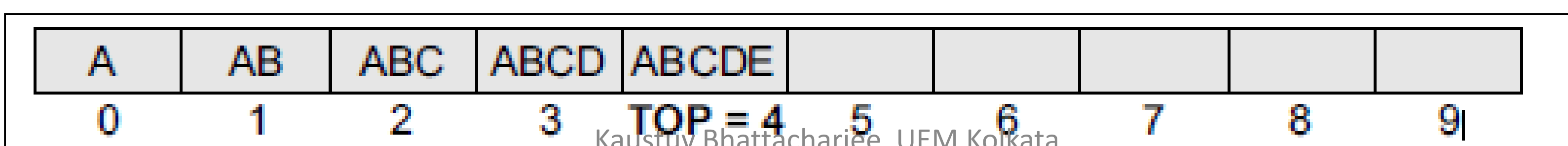
- During Function Call





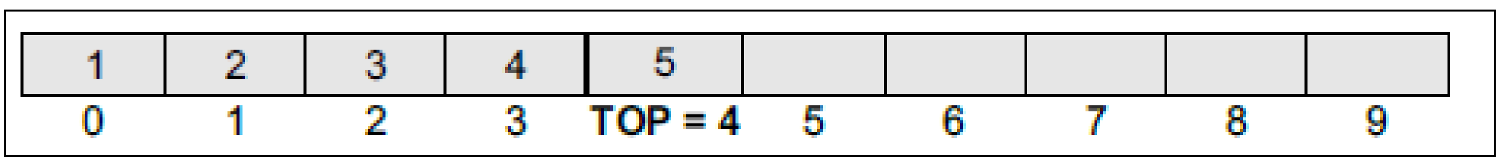
Array Representation of Stack

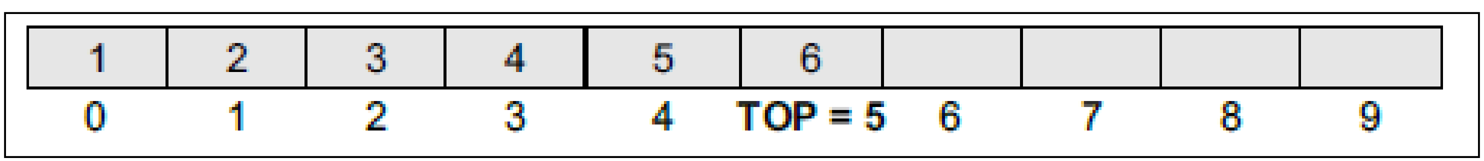
- Stack can be represented as a linear array
- Every stack has a variable called *TOP* associated with it, which is used to store the address of the topmost element of the stack
- Element will be added to or deleted from *TOP*
- Variable *MAX* is used to store the maximum number of elements that the stack can hold
- *TOP=NULL* indicates stack is empty
- *TOP=MAX-1* indicates stack is full



Operations on Stack-Push

- **PUSH** operation
  - Used to insert an element into the stack
  - The new element is added at the topmost position of the stack
  - Before inserting the value, first check if TOP=MAX–1, indicating stack OVERFLOW (stack is full)





Operations on Stack-Push Contd…

- Algorithm/Steps for PUSH operation (using Array)

Step 1: IF TOP = MAX-1

PRINT OVERFLOW

[END OF IF]

Step 2: SET TOP = TOP + 1

Step 3: SET STACK[TOP] = VALUE

Step 4: END

Operations on Stack-POP

- **POP** operation
  - Used to delete the topmost element from the stack
  - Before deleting the value, first check if TOP=NULL, indicating stack UNDERFLOW (stack is empty)

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

1	2	3	4						
0	1	2	TOP = 3	4	5	6	7	8	9

Operations on Stack-POP Contd…

- <u>Algorithm/Steps for POP operation (using Array)</u>

Step 1: IF TOP = NULL

    PRINT UNDERFLOW

    GO TO Step 4

    [END OF IF]

Step 2: SET VAL = STACK[TOP]

Step 3: SET TOP = TOP - 1

Step 4: END

Operations on Stack-Peep

- **Peep** operation
  - Used to return the value of the topmost element of the stack without deleting it from the stack
  - Before returning the value, first check if TOP=NULL, indicating stack is empty

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

Operations on Stack-Peep Contd…

- <u>Algorithm/Steps for Peep operation (using Array)</u>

Step 1: IF TOP = NULL

PRINT “STACK IS EMPTY”

GO TO STEP 3

[END OF IF]

Step 2: RETURN STACK[TOP]

Step 3: END

Stack using Array: Assignments

Assignments:

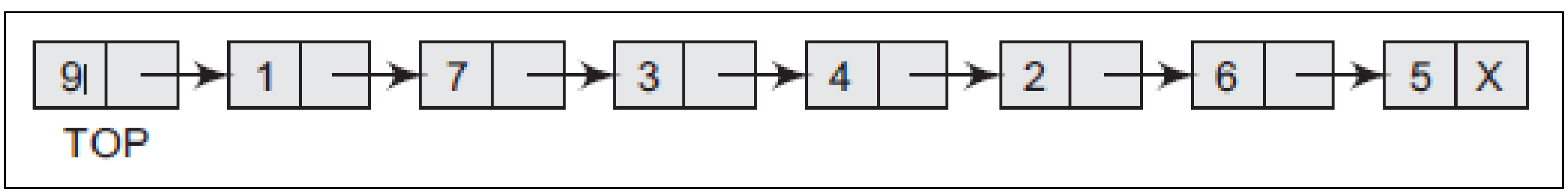
1. Write a program to insert an element into the stack using an array (Push Operation).
2. Write a program to delete an element from the stack using an array (Pop Operation).
3. Write a program to return the value of the topmost element of the stack (without deleting it from the stack) using an array (Peep operation).
4. Write a program to display the elements of a stack using an array.

Stack using Array: Limitations

- Array must be declared to have some fixed size
- If stack is very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation
- It is difficult to implement stack using array if the array size can’t be determined in advance

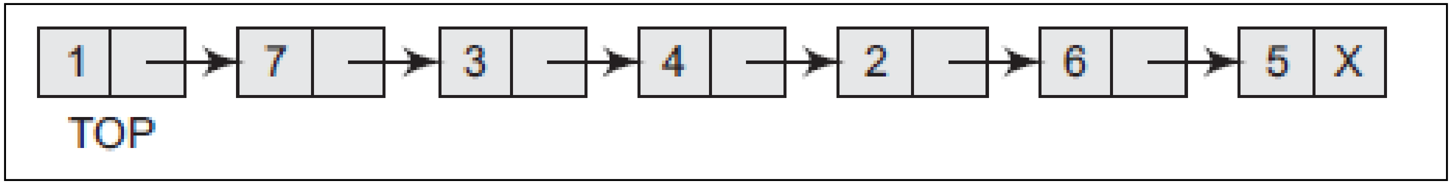
Stack using Linked List

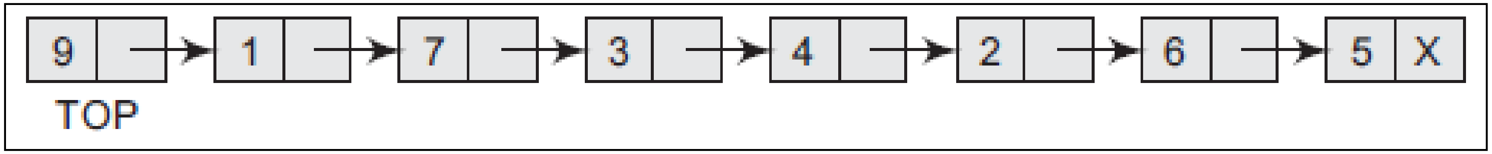
- Stack can be represented using linked list
- Every node has two parts—one that stores data and another that stores the address of the next node
- The *START* pointer of the linked list is used as *TOP*
- All insertions and deletions are done at the node pointed by *TOP*
- *TOP=NULL* indicates stack is empty



Operations on Linked Stack-Push

- **PUSH** operation
  - Used to insert an element into the stack
  - The new element is added at the topmost position of the stack





Operations on Linked Stack-Push Contd…

- **<u>Algorithm/Steps for PUSH operation (using Linked List)</u>**

Step 1: Allocate memory for new node , name it as NEW_NODE

Step 2: SET NEW_NODE ->DATA = VAL

Step 3: IF TOP = NULL

SET NEW_NODE->NEXT=NULL

SET TOP=NEW_NODE

ELSE

SET NEW_NODE->NEXT=TOP

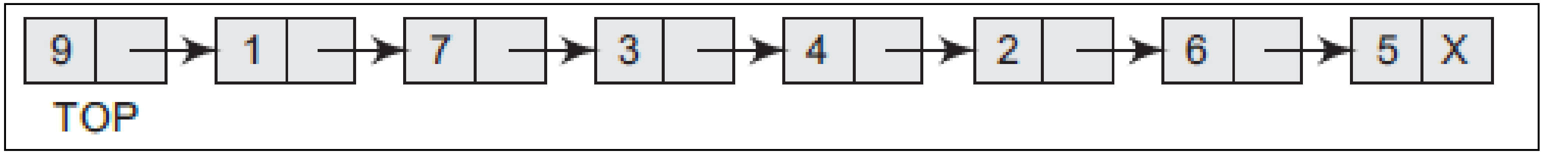
SET TOP=NEW_NODE

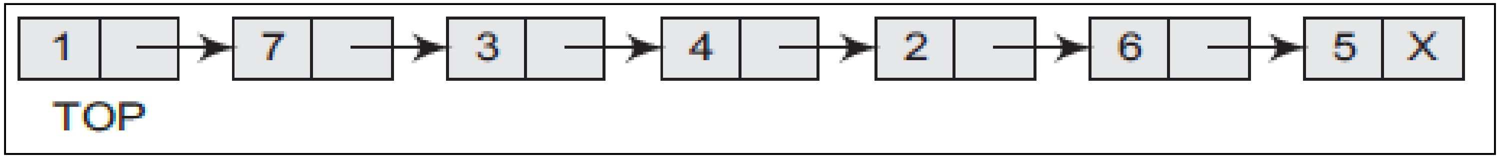
[END OF IF]

Step 4: Stop

Operations on Linked Stack-POP

- **POP** operation
  - Used to delete the topmost element from the stack
  - Before deleting the value, first check if TOP=NULL, indicating stack UNDERFLOW (stack is empty)





Operations on Linked Stack-POP Contd…

- <u>Algorithm/Steps for POP operation (using Linked List)</u>

Step 1: IF TOP = NULL

        PRINT UNDERFLOW

        GO TO STEP 5

    [END OF IF]

Step 2: SET PTR=TOP

Step 3: SET TOP=TOP->NEXT

Step 4: FREE PTR

Step 5: END

Stack using Linked List: Assignments

<u>Assignments:</u>

1. Write a program to insert an element into the stack using linked list (Push Operation).
2. Write a program to delete an element from the stack using linked list (Pop Operation).
3. Write a program to return the value of the topmost element of the stack (without deleting it from the stack) using linked list (Peep operation).
4. Write a program to display the elements of a stack using linked list .

Application of Stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Evaluation of Arithmetic Expressions

- **Polish Notations**
  - **Infix**, **postfix**, and **prefix** notations are three different but equivalent notations of writing algebraic expressions
  - While writing an algebraic expression using **infix** notation, the operator is placed in between the operands
  - Ex: A+B*C: Here operator is placed between the two operands
  - Humans can easily evaluate algebraic expression using infix notation
  - Computers find it difficult to parse infix notations (because of operator precedence, associative rules, brackets etc)
  - Computers work more efficiently with expressions written using **prefix** and **postfix** notations

Evaluation of Arithmetic Expressions … Contd.

- **Postfix Notations**
  - A parenthesis free notation, also known as Reverse Polish Notation or RPN
  - The operator is placed after the operands
  - Order of evaluation of a postfix expression is always from left to right
  - Operators are applied to the operands that are immediately left to them
  - Does not even follow the rules of operator precedence, even brackets cannot alter the order of evaluation

Evaluation of Arithmetic Expressions … Contd.

- **Infix to Postfix: Example**

1. (A + B) * C
   [AB+]*C
   AB+C*
2. (A + B) / (C + D) – (D * E)
   [AB+] / [CD+] – [DE*]
   [AB+CD+/] – [DE*]
   AB+CD+/DE*–
3. (A–B) * (C+D)
   [AB–] * [CD+]
   AB–CD+*

Evaluation of Arithmetic Expressions … Contd.

- **Prefix Notations**
  - Also known as Polish Notation
  - The operator is placed before the operands
  - Order of evaluation of a prefix expression is always from left to right
  - Operators are applied to the operands that are present immediately on the right of the operator
  - Does not even follow the rules of operator precedence, even brackets cannot alter the order of evaluation

Evaluation of Arithmetic Expressions … Contd.

- **<u>Infix to Prefix: Example</u>**

1. (A + B) * C

   [+AB]*C

   *+ABC

2. (A + B) / (C + D) – (D * E)

   [+AB] / [+CD] – [*DE]

   [/+AB+CD] – [*DE]

   -/+AB+CD*DE

3. (A–B) * (C+D)

   [-AB] * [+CD]

   *-AB+CD

Infix to Postfix

- **Infix to Postfix: Steps/Algorithm**

Step 1: Add) to the end of the infix expression

Step 2: Push (on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a (is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it postfix expression.

IF a) is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until a (is encountered.

b. Discard the (. That is, remove the (from stack and do not add it to the postfix expression

IF an operator O is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than O

b. Push the operator O to the stack

[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Infix to Postfix-Example

- Convert the following infix expression into postfix expression using the algorithm given earlier:A – (B / C + (D % E * F) / G)* H

<u>Soln:</u> A – (B / C + (D % E * F) / G)* H

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Evaluate a Postfix Expression

- **Evaluate a Postfix Expression: Steps/Algorithm**

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Evaluate a Postfix Expression-Example

- Consider the infix expression 9 – ((3 * 4) + 8) / 4. The equivalent postfix expression is: 9 3 4 * 8 + 4 / –
- After Step 1, the expression appears like: 9 3 4 * 8 + 4 / –)

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

Evaluate a Prefix Expression

- **Evaluate a Prefix Expression: Steps/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
```

Evaluate a Prefix Expression-Example

- Consider the prefix expression is: + – 2 7 * 8 / 4 12

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

Infix to Prefix

- **Infix to Prefix: Steps/Algorithm**
- Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
- Step 2: Obtain the Postfix expression of the infix expression obtained in step 1.
- Step 3: Reverse the postfix expression to get the prefix expression.

Infix to Prefix-Example

- Convert the following infix expression into prefix expression using the algorithm given earlier: (A – B / C) * (A / K – L)

<u>Soln:</u>

Step 1: Reverse the infix string. Note that while reversing the string we must interchange left and right parentheses. So we get

(L – K / A) * (C / B – A)

Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1. The expression is:

(L – K / A) * (C / B – A)

= [L – (K A /)] * [(C B /) – A]

= [LKA/–] * [CB/A–]

= L K A / – C B / A – *

Step 3: Reverse the postfix expression to get the prefix expression

Therefore, the prefix expression is

* – A / B C – /A K L

Arithmetic Expressions: Assignments

<u>Assignments (to be implemented using stack):</u>

1. Write a program to reverse a list of given numbers.
2. Write a program to convert an infix expression into its equivalent postfix notation.
3. Write a program to evaluate a postfix expression.
4. Write a program to convert an infix expression to a prefix expression.
5. Write a program to evaluate a prefix expression.
6. Convert the following infix expression into postfix expression using appropriate algorithm showing all steps: A – (B / C + (D % E * F) / G)* H
7. Consider the infix expression: 9 – ((3 * 4) + 8) / 4. Convert it to the equivalent postfix expression first. Then evaluate the postfix expression using appropriate algorithm showing all steps.
8. Consider the infix expression: 9 – ((3 * 4) + 8) / 4. Convert it to the equivalent prefix expression first. Then evaluate the prefix expression using appropriate algorithm showing all steps.
9. Convert the following infix expression into prefix expression using appropriate algorithm showing all steps: (A – B / C) * (A / K – L).

Recursion: Introduction

- A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases:
  - *Base case,* in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - *Recursive case,* in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Recursion: Example

```
return 5 * factorial(4) = 120
  └── return 4 * factorial(3) = 24
        └── return 3 * factorial(2) = 6
              └── return 2 * factorial(1) = 2
                    └── return 1 * factorial(0) = 1
```

- **<u>Factorial of a number:</u>**

For a number n, $n! = n \times (n-1)!$

Which can have a following solution approach (for n=5):

As every recursive function must have a base case and a recursive case, for factorial function, we have the following:

- <u>Base case</u> is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
- <u>Recursive case</u> of the factorial function will call itself but with a smaller value of n, this case can be given as:

factorial(n) = n × factorial (n–1)

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Recursion: Assignments

<u>Assignments (using recursion):</u>

1. Write a program to calculate the factorial of a given number.
2. Write a program to calculate the GCD of two numbers using recursive functions.
3. Write a program to calculate exp(x,y) using recursive functions.
4. Write a program to print the Fibonacci series using recursion.
5. Write a program to solve the tower of Hanoi problem using recursion.