Stacks

Objectives

1. Understands the concepts of stacks
2. Representation of stacks as data structure
3. Different method for implementing stacks
4. Application of stacks
5. Use of stacks in recursion problem

What is a stack?

- It is an ordered group of homogeneous items of elements.

- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).

- The last element to be added is the first to be removed

(**LIFO**: Last In, First Out).

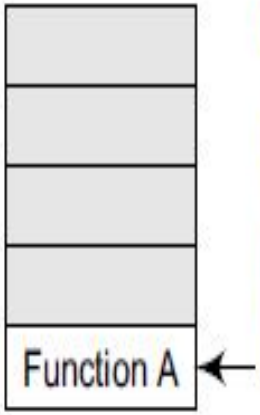
Use of stack in Computer Science

```
main()
{ A()}

A()
{ B() }

B()
{C()}

C()
{D()}

D()
{E()}
```

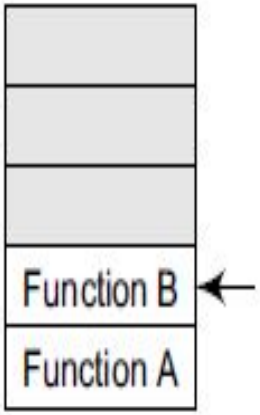
Consider an example,
where we are executing **function A**.

Use of stack in Computer Science

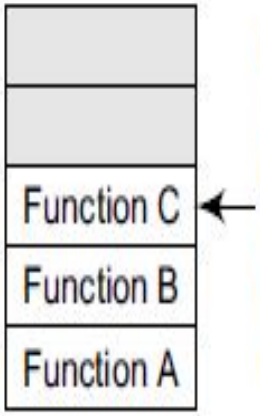
In the course of its execution, **function A** calls another **function B**. Function B in turn calls another **function C**, which calls **function D**.



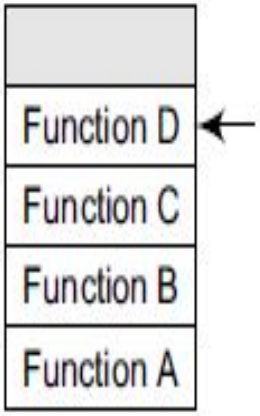
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.



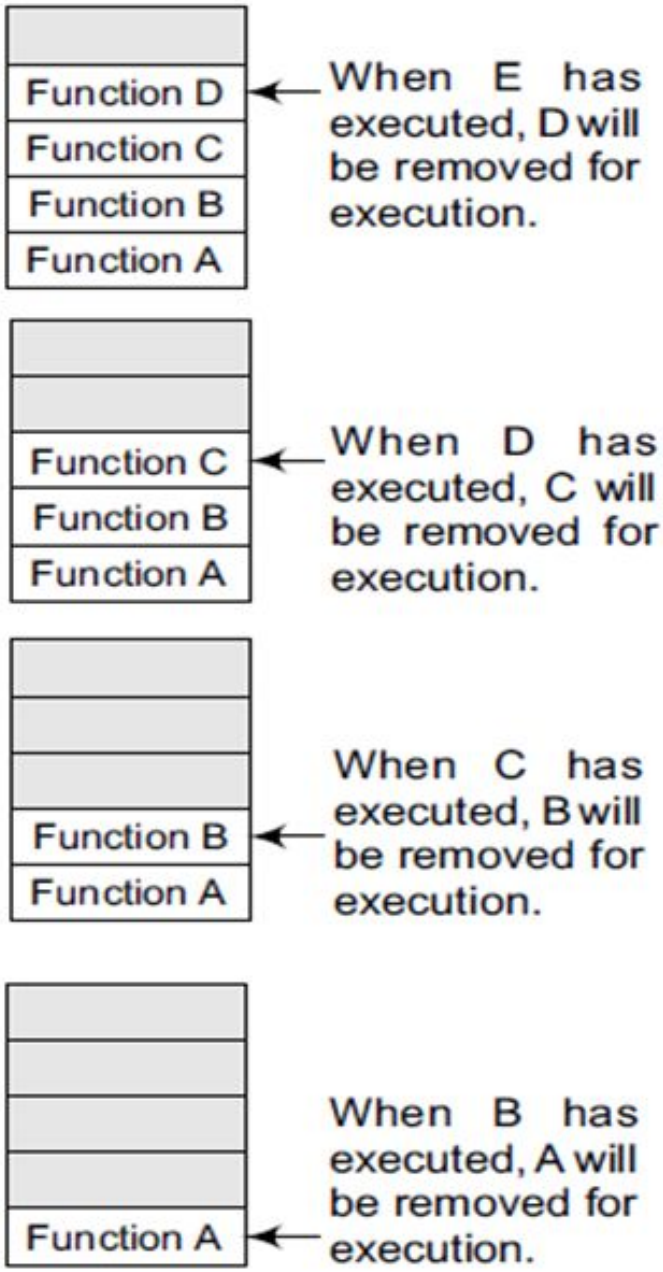
When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.



When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.



When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.



- The system stack **ensures a proper execution order of functions**.

- Therefore, stacks are frequently used in **situations where the order of processing is very important**, especially when the processing needs to be postponed until other conditions are fulfilled.

Operation on Stack

push(i) to insert the element i on the top of the stack

pop( ) to remove the top element of the stack and to return the removed element as a function value.

Top( ) to return the top element of stack(s)

empty() to check whether the stack is empty or not. It returns true if stack is empty and returns false otherwise

- Stacks can be implemented using either arrays or linked lists.

Works on the principle of **LIFO**

LIFO – Last In First Out

ARRAY REPRESENTATION OF STACKS

1. In the computer's memory, stacks can be represented as a linear array.

2. Every stack **has a variable called TOP** associated with it, which is used to store the address of the topmost element of the stack.

1. TOP is the position where the element will be added to or deleted from.

2. There is another variable called **MAX**, which is used to store the maximum number of elements that the stack can hold.

3. Underflow and Overflow

if **TOP = NULL**,(underflow), it indicates that the stack is empty and

if **TOP = MAX–1**,(overflow) then the stack is full.

Algorithm for PUSH operation

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Algorithm for POP operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Algorithm for PEEK operation

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

PUSH operation

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

POP operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

PEEK operation

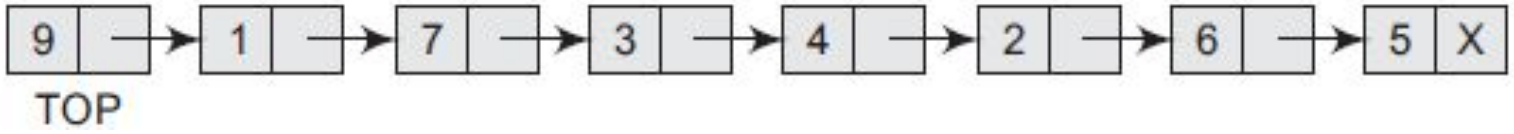
```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

LINKED REPRESENTATION OF STACK

Stack may be created using an array. This technique of creating a stack is easy, but the **drawback** is that the **array must be declared to have some fixed size**. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

1. In a linked stack, 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**.

2. All insertions and deletions are done at the node pointed by TOP.
   If **TOP = NULL**, then it indicates that the stack is empty.



The **storage requirement** of linked representation of the stack with n elements is **O(n),** and the typical **time requirement** for the operations is **O(1).**

Algorithm for PUSH operation
of Stack implemented using Linked list

```
Step 1: Allocate memory for the new
        node and 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: END
```

Algorithm for POP operation
of Stack implemented using Linked list

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Algorithm for PUSH operation

```
Step 1: Allocate memory for the new
        node and 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: END
```

Algorithm for POP operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

APPLICATIONS OF STACKS

1. Parentheses checker
2. Conversion of an infix expression into a postfix expression
3. Evaluation of a postfix expression
4. Conversion of an infix expression into a prefix expression
5. Evaluation of a prefix expression
6. Recursion
7. Tower of Hanoi

Parentheses Checker

Stacks can be used to check the validity of parentheses in any algebraic expression.

For example,

an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.

For example,

the expression **(A+B}** is invalid

an expression **{A + (B – C)}** is valid

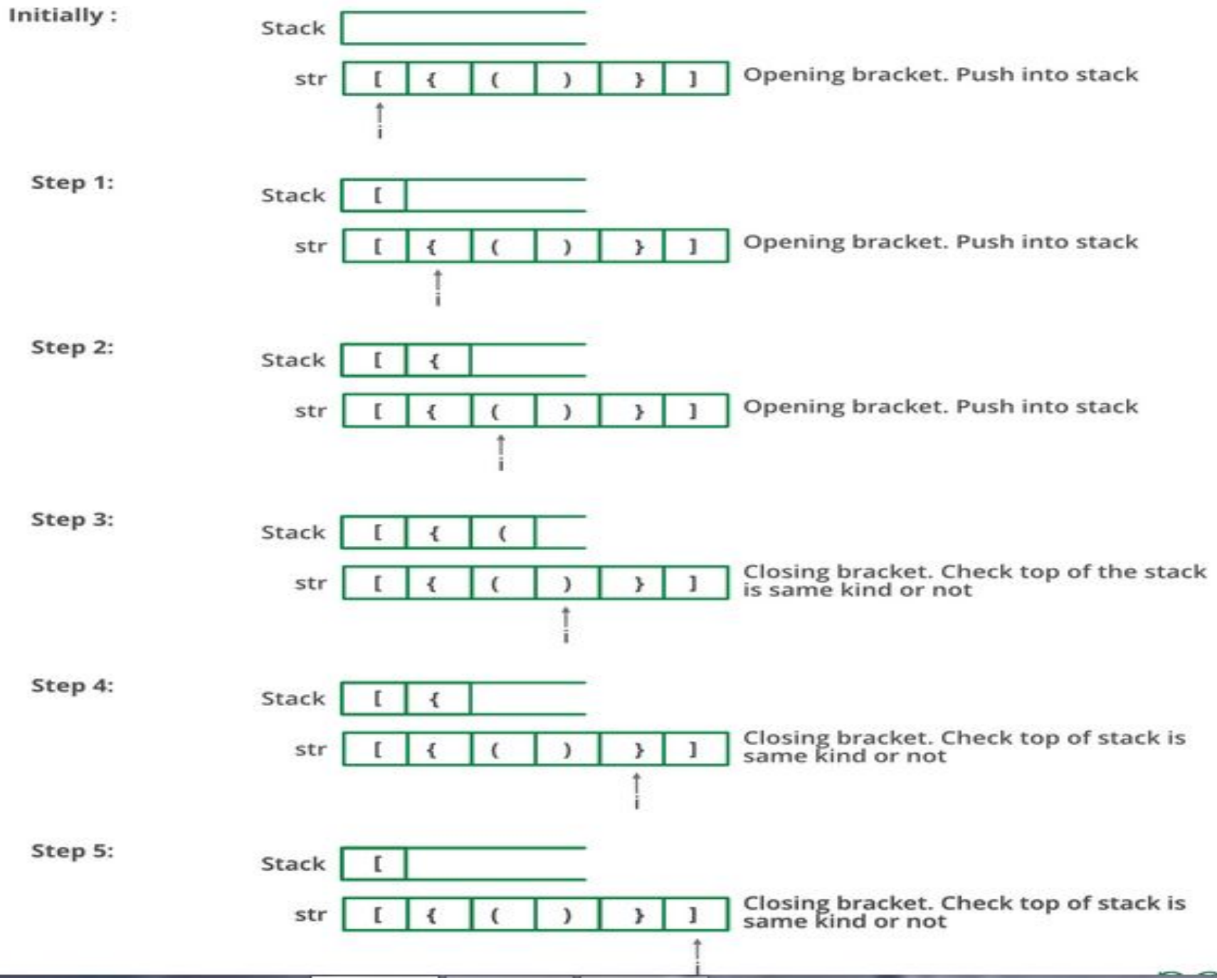
Algorithm:

- Declare a character stack S.

- Now traverse the expression string exp.
  - If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.

  - If the current character is a closing bracket (**')' or '}' or ']'**) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

- After complete traversal, if there is some starting bracket left in stack then "not balanced"


Initially :
Stack
str
[{ ( ) }]
i
Opening bracket. Push into stack
Step 1:
Stack
[
str
[{ ( ) }]
i
Opening bracket. Push into stack
Step 2:
Stack
[{
str
[{ ( ) }]
i
Opening bracket. Push into stack
Step 3:
Stack
[{ (
str
[{ ( ) }]
i
Closing bracket. Check top of the stack is same kind or not
Step 4:
Stack
[{
str
[{ ( ) }]
i
Closing bracket. Check top of stack is same kind or not
Step 5:
Stack
[
str
[{ ( ) }]
i
Closing bracket. Check top of stack is same kind or not

Pseudo code: Parenthesis Matching

```
valid = true                    /* assuming that the string is valid*/
s = empty stack;
while (not end of the string) {

                        symbol = next input string;
                        if (symbol = '(' || '[' || '{')
                              push(symbol);

                        if (symbol = ')' || ']' || '}')
                           {
                                    if (empty (s))
                                    valid = false;
                                     else {
                                       i = pop();
                                         if ( i ! =symbol)
                                            valid = false;
                                          }         //end of  else
                                 }     // end of if
          }  //end of while
    if (valid)
          cout<< "the string is valid";
    else
          cout<<( " the string is invalid";
```

Mathematical Notation Translation

prefix (polish)
postfix (reverse polish)

Mathematical Notation Translation

- Infix – prefix (polish)
- Infix – postfix (reverse polish)

The fundamental property of these notations is that the order in which the operations are to perform are completely **determined by the positions of operator and operands** in the expression. There is **no need to use parenthesis** to define any precedence

For example, if we have an expression
A + B * C, then first B * C will be done and the result will be added to A.
(A + B) * C, will evaluate A + B first and then the result will be multiplied with C

Conversion of an Infix Expression into a Postfix Expression

An algebraic expression may contain parentheses, operands, and operators.

For simplicity of the algorithm we assume only +, –, *, /, % operators.

The precedence of these operators can be given as follows:

Higher priority *, /, % Lower priority +, –

the order of evaluation of these operators can be changed by making use of parentheses.

For example, if we have an expression

A + B * C, then first B * C will be done and the result will be added to A.

(A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

Algorithm
Infix Expression to a Postfix Expression

```
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 to the
       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
```

A - (B / C + (D % E * F) / G)* H

A - (B / C + (D % E * F) / G)* H)

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 to the 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 0 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 0
b. Push the operator 0 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

CHAR	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 * -

Example

Evaluation of a Postfix Expression

Evaluation of a Postfix Expression

```
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
```

Example

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

```
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
```

Infix to Prefix Expression

Method1- Algorithm Infix to Prefix Expression

Step 1. Push “)” onto STACK, and add “(“ to start of the A.

Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty or contain only “)”

Step 3. If an **operand** is encountered add it to **B**

Step 4. If a **right parenthesis** is encountered push it onto STACK

Step 5. If an **operator** is encountered then:

a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has higher precedence than the operator.

b. Add operator to STACK

Step 6. If **left parenthesis** is encountered then

a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a right parenthesis is encountered)

b. Remove the left parenthesis

Step 7. Reverse B to get prefix form

Method2- Infix Expression to a Prefix Expression

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

For example, given an infix expression (A – B / C) * (A / K – L)

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

(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)

Therefore, [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**

Step 1. Push “)” onto STACK, and add “(“ to start of the A.

Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty or contain only “)”

Step 3. If an **operand** is encountered add it to **B**

Step 4. If a **right parenthesis** is encountered push it onto STACK

Step 5. If an **operator** is encountered then:
a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has higher precedence than the operator.
b. Add operator to STACK

Step 6. If **left parenthesis** is encountered then
a. Repeatedly pop from the STACK and add to B (each operator on of stack until a right parenthesis is encountered)
b. Remove the left parenthesis

Step 7. Reverse B to get prefix form

14 / 7 * 3 – 4 + 9 / 2

Char	Stack	Expression

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

14 / 7 * 3 – 4 + 9 / 2

Char	Stack	Expression

Evaluation of a Prefix Expression

```
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
```

Example

- + * 2 3 / 2 4 + 4 3		
Char	Stack	operation
3	3	
4	3, 4	
+	7	4+3
4	7, 4	
2	7, 4, 2	
/	7, 0	2/4
3	7, 0,3	
2	7, 0,3, 2	
*	7, 0, 6	2*3
+	7, 6	6 +0
-	-1	6-7

14 / 7 * 3 – 4 + 9 / 2

+ - * / 14 7 3 4 / 9 2

Char	Stack	Expression