Data Structures

Lecture 4 :

The Stack

Dr. Essam Halim Houssein
Lecturer, Faculty of Computers and Informatics,
Benha University
http://bu.edu.eg/staff/esamhalim14

The Stack

3.4. APPLICATIONS OF STACKS

There are a number of applications of stacks:

- ❑ Stack is internally used by compiler when we implement (or execute) any recursive function.
- ❑ Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

The Stack

3.4.1. RECURSION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. **Prologue**: Save the parameters, local variables, and return address.
2. **Body**: If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. **Epilogue**: Restore the most recently saved parameters, local variables, and return address.

The Stack

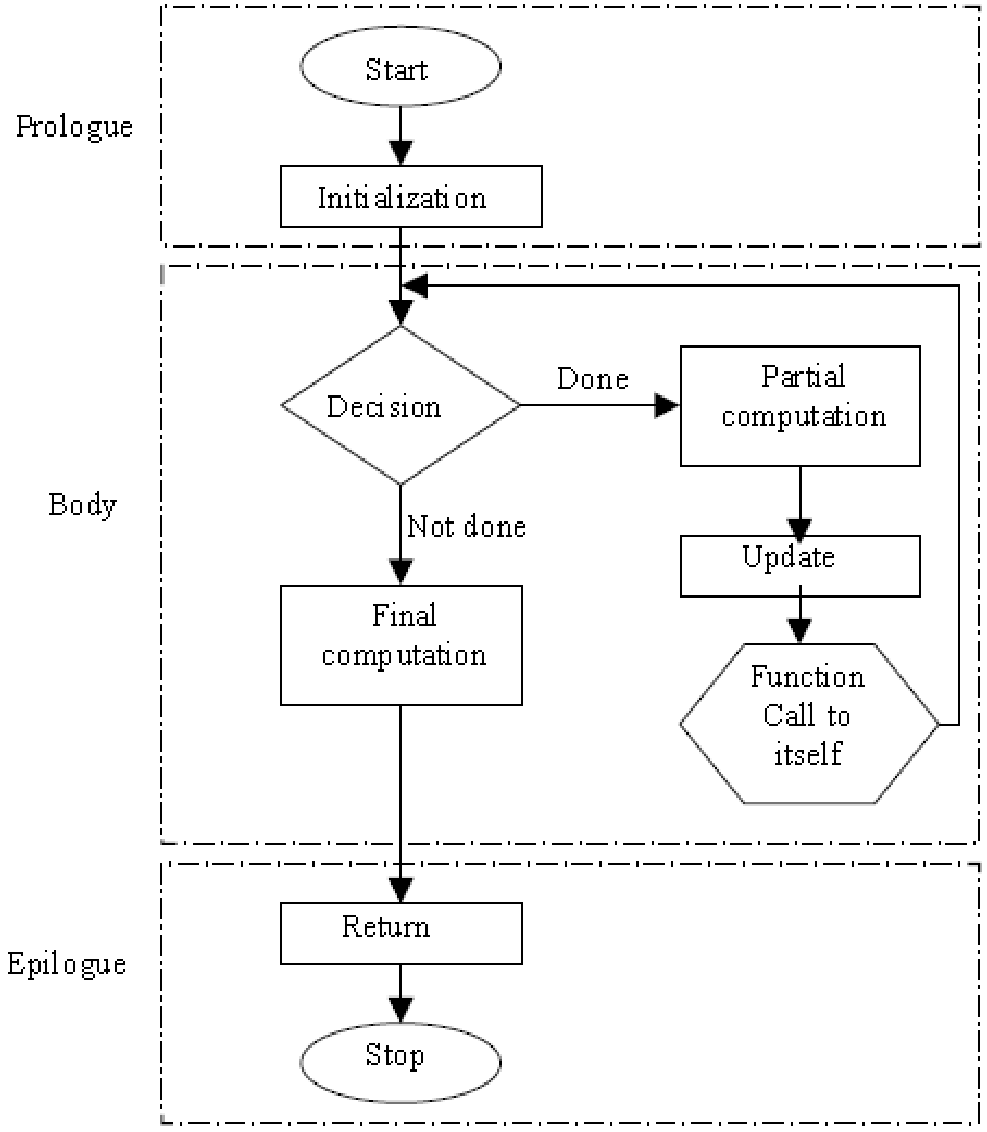


Fig. 3.2. Flowchart model for a recursive algorithm

The Stack

Programs compiled in modern high-level languages make use of a stack for the procedure or function invocation in memory. When any procedure or function is called, a number of words (such as variables, return address and other arguments and its data(s) for future use) are pushed onto the program stack. When the procedure or function returns, this frame of data is popped off the stack.

The Stack

The stack is a region of main memory within which programs temporarily store data as they execute. For example, when a program sends parameters to a function, the parameters are placed on the stack. When the function completes its execution these parameters are popped off from the stack.

The Stack

// PROGRAM TO FIND FACTORIAL OF A NUMBER, RECURSIVELY

Assignment (1) within lab

The Stack

3.4.2. RECURSION vs ITERATION

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

1. It requires stack implementation.
2. It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
3. Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

The Stack

Given below are some of the important points, which differentiate iteration from recursion.

NO.	*Iteration*	*Recursion*
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.
2	Iteration involves four clear-cut Steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

The Stack

3.4.3. EXPRESSION

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of **operands** or data items combined with several **operators**.)

1. **Infix notation**
2. **Prefix notation**
3. **Postfix notation**

The Stack

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example : A + B

In the prefix notation the operator(s) are written before the operands, like: + A B

In the postfix notation the operator(s) are written after the operands, like: A B +

The Stack

Because the **postfix notation** is most suitable for a computer to calculate any expression, and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the **postfix notation**. Moreover the postfix notation is the way the computer looks towards arithmetic expression, any expression entered into the computer is first **converted into postfix** notation, **stored in stack and then calculated.**

The Stack

Human beings are quite used to work with mathematical expressions in **infix notation**, which is rather complex.

Whenever **an infix expression** consists of more than one operator, the precedence rules (**BODMAS**) should be applied to decide which operator (and operand) is evaluated first.

But in a **postfix expression** **operands appear before the operator**, so there is no need for operator precedence and other rules.

The Stack

Notation Conversions

C++ Operator Precedence

The Stack

3.3.1 CONVERTING INFIX TO POSTFIX EXPRESSION

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. 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.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

The Stack

Algorithm

1. Push “(” onto stack, and add “)” to the end .
2. Scan from left to right and repeat Steps 3 to 6 for each element until the stack is empty.
3. If an operand is encountered, add it to Postfix_Stack (Q).
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator ⊗ is encountered, then:

   (a) Repeatedly pop from stack, if not its precedence higher precedence than ⊗.

   (b) Else Add ⊗ to stack.
6. If a right parenthesis is encountered, then:

   (a) Repeatedly pop from stack until a left parenthesis is encountered.

   (b) Remove the left parenthesis.
7. Exit.

The Stack

For, example consider the following arithmetic **Infix** to **Postfix** expression

I =(A+(B*C-(D/E^F)*G)*H)

Sr. No	Symbol Scanned	STACK	Postfix_Stack (Q)
1	A	(	A
2	+	(+	A
3	(	(+(	A
4	B	(+(	AB
5	*	(+(*	AB
6	C	(+(*	ABC
7	-	(+(-	ABC*
8	(	(+(-(	ABC*D
9	D	(+(-(	ABC*D

The Stack

Sr. No	Symbol Scanned	STACK	Expression
10	/	(+(-(/	ABC*D
11	E	(+(-(/	ABC*DE
12	^	(+(-(/^	ABC*DE
13	F	(+(-(/^	ABC*DEF
14	)	(+(-	ABC*DEF^/
15	*	(+(-*	ABC*DEF^/
16	G	(+(-*	ABC*DEF^/G
17	)	(+	ABC*DEF^/G*
18	*	(+*	ABC*DEF^/G*-
19	H	(+(*	ABC*DEF^/G*-H
20	)	-	ABC*DEF^/G*-H*+

The Stack

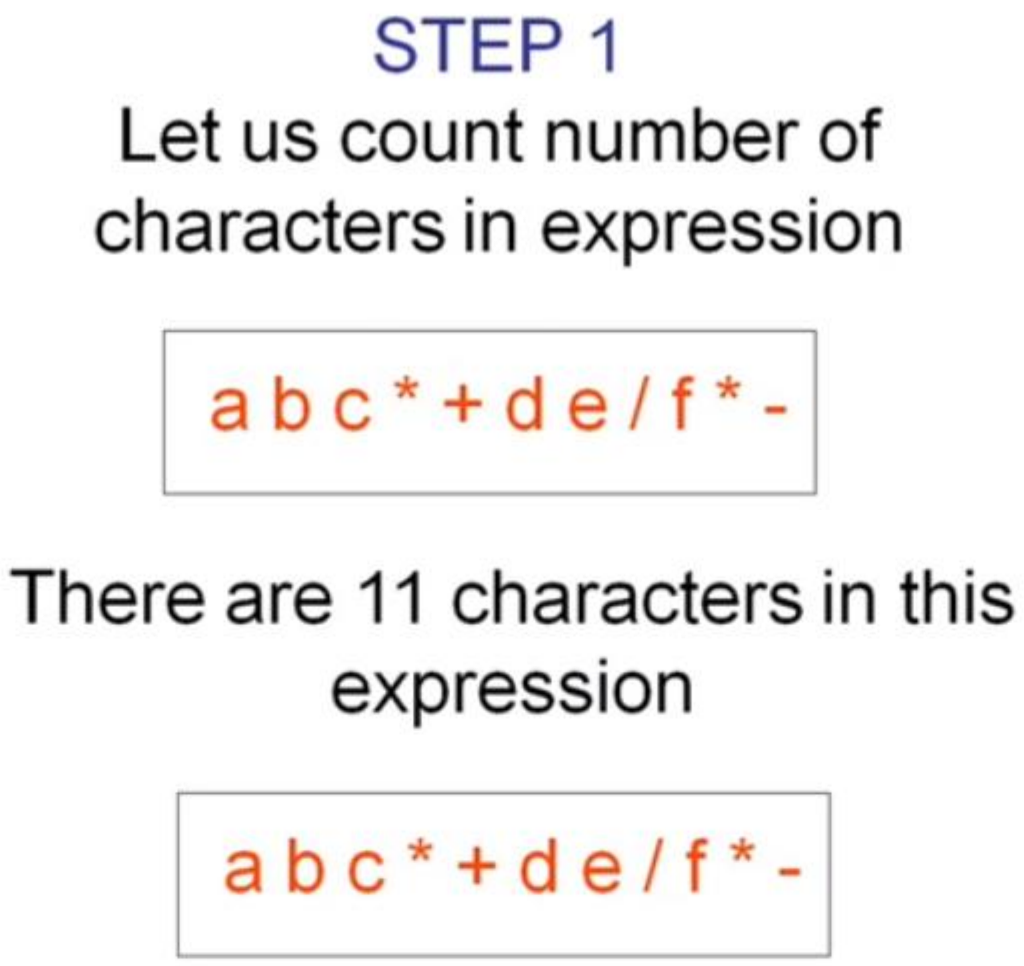
3.3.2 CONVERTING POSTFIX TO INFIX EXPRESSION

The rules to be remembered during infix to postfix conversion are:

1. Count all characters
2. Take a stack with size equal to number of characters
3. Write the Postfix expression like this Left most Char at top or Right most Char at bottom sequentially)
4. Fill up the stack
5. Apply POP on all elements one by one starting from TOP of stack

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression



STEP 1

Let us count number of characters in expression

a b c * + d e / f * -

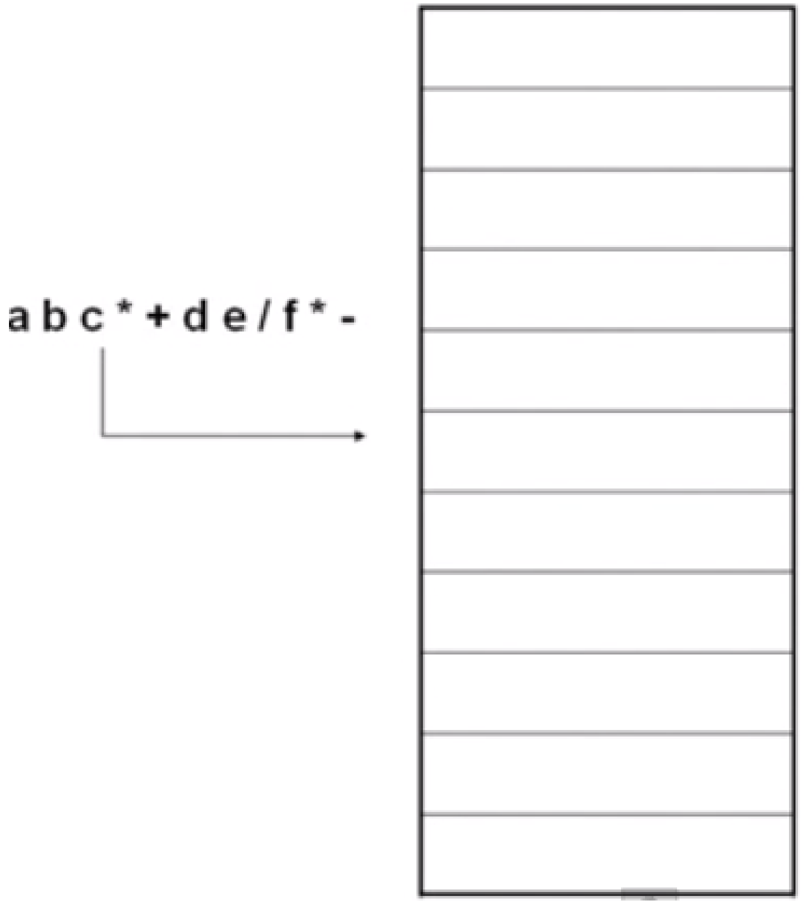
There are 11 characters in this expression

a b c * + d e / f * -

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



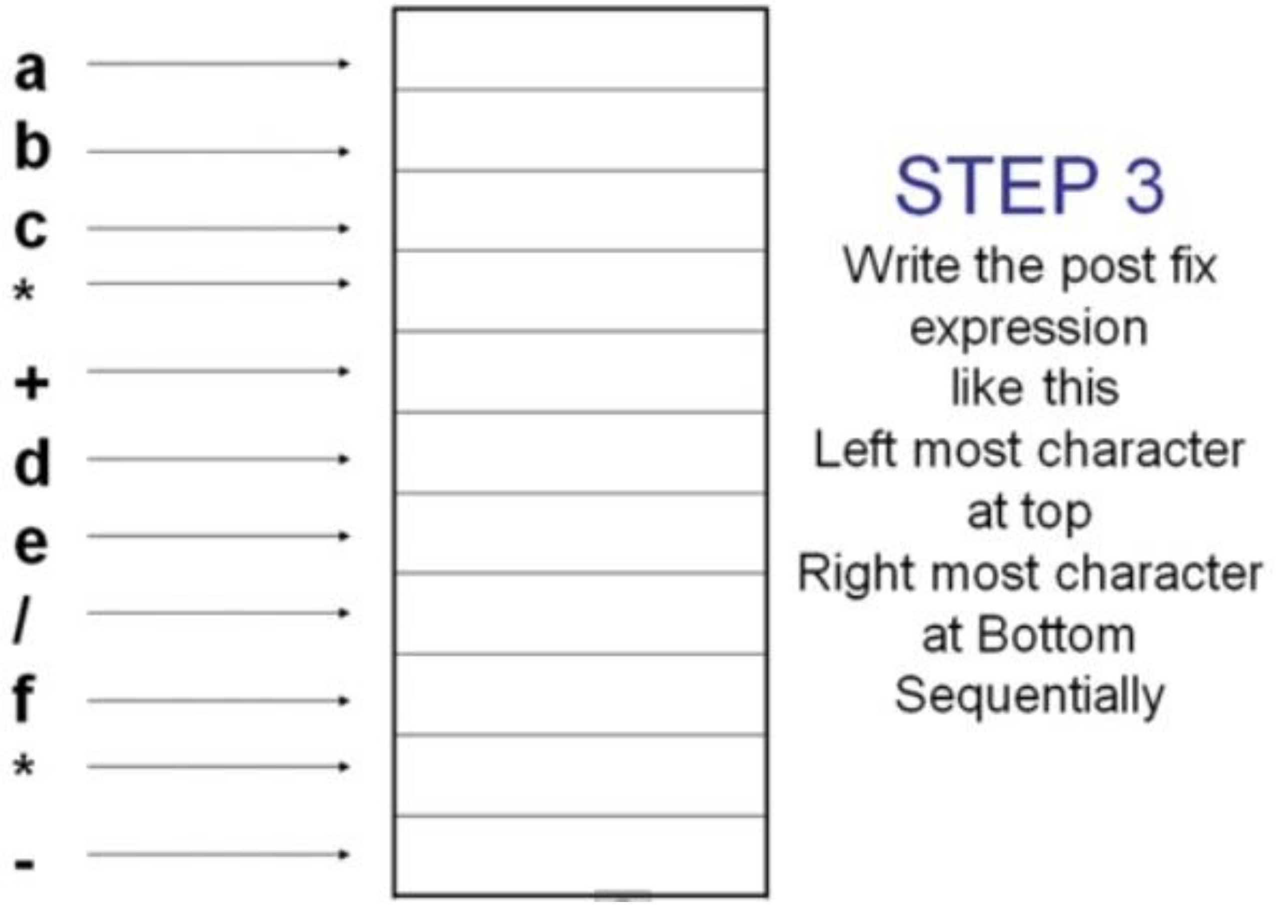
STEP 2

Take a stack with size equal to number of characters in postfix expression

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

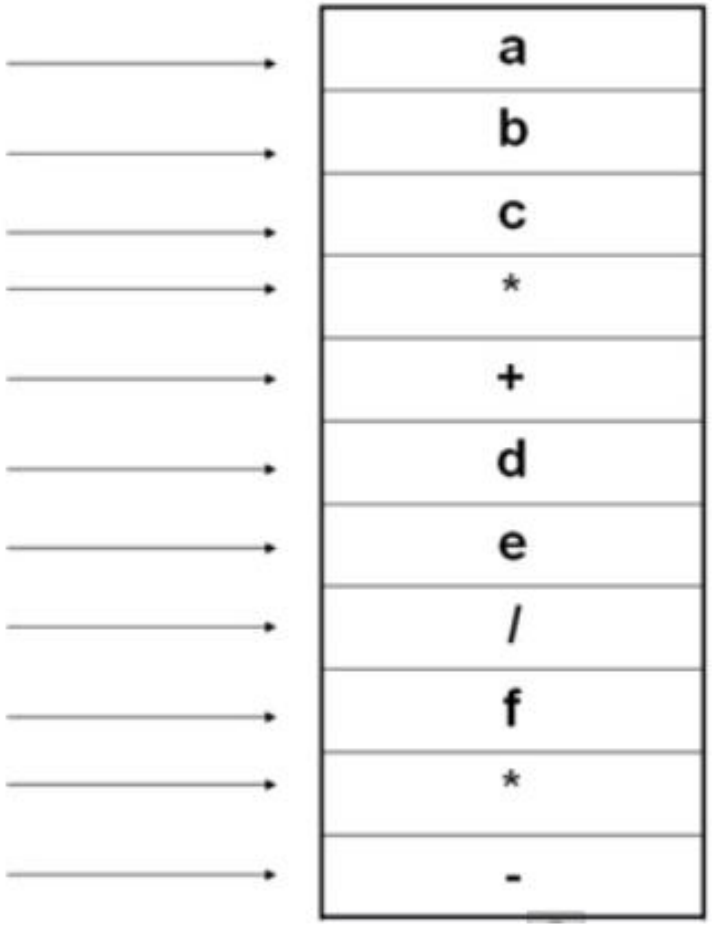
A B C * + D E / F * -


a
b
c
*
+
d
e
/
f
*
-
STEP 3
Write the post fix expression like this
Left most character at top
Right most character at Bottom
Sequentially

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



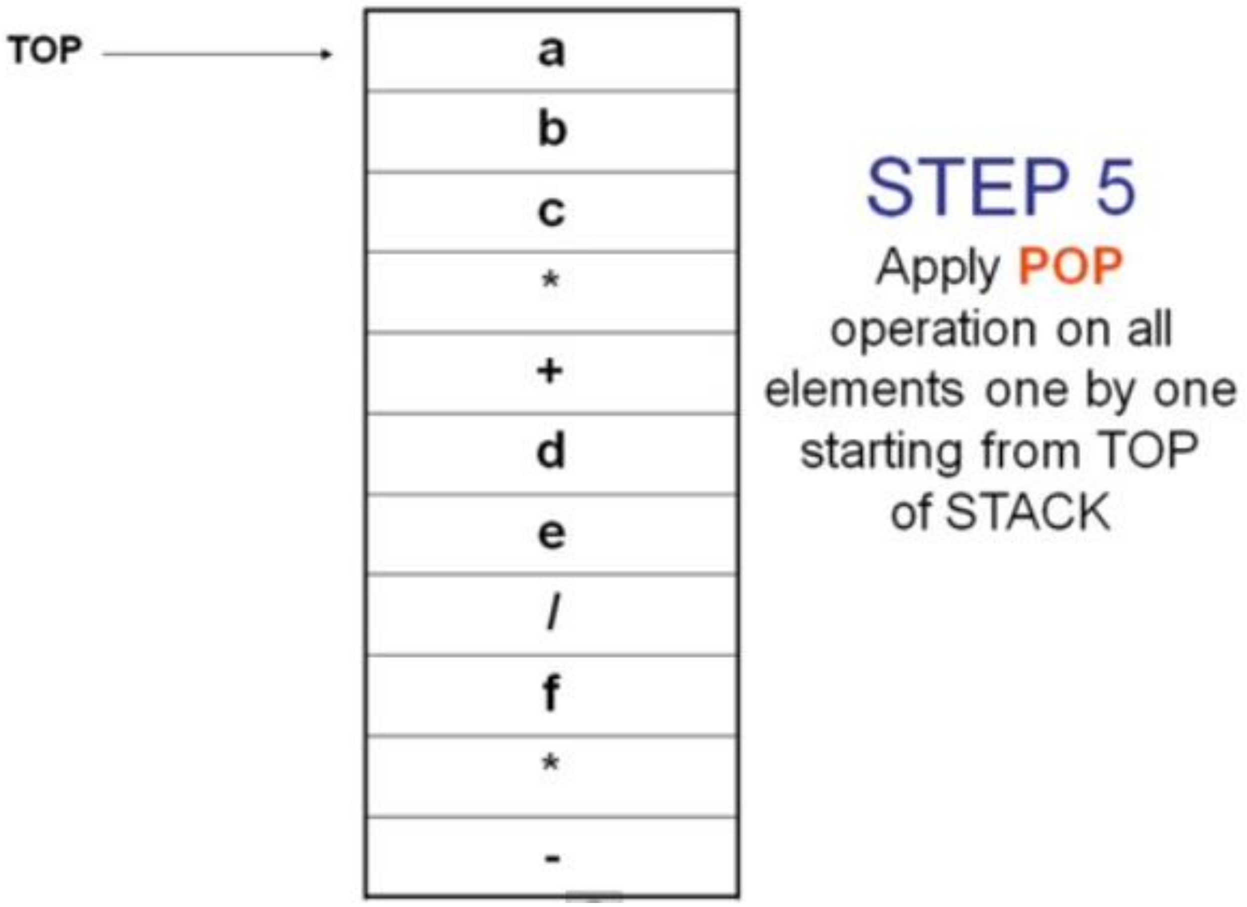
STEP 4

Fill up the Stack Sequentially

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



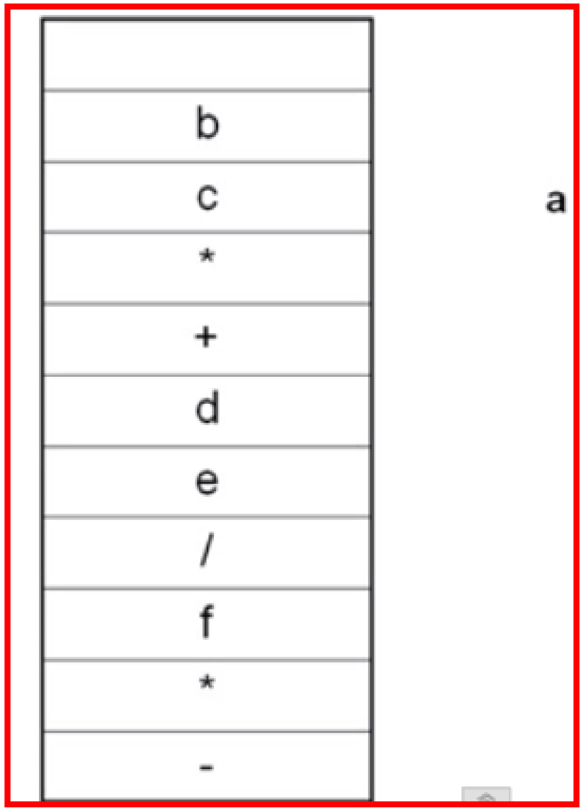
STEP 5

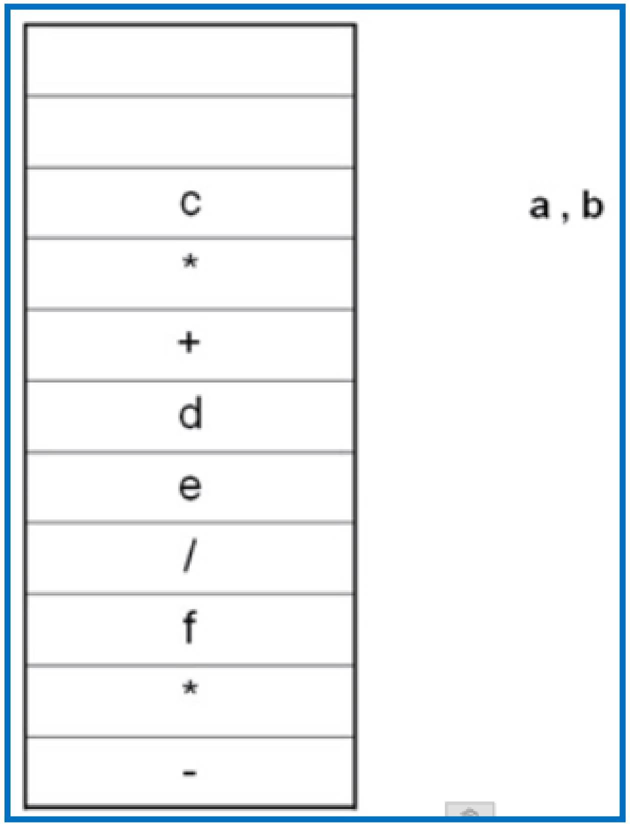
Apply POP operation on all elements one by one starting from TOP of STACK

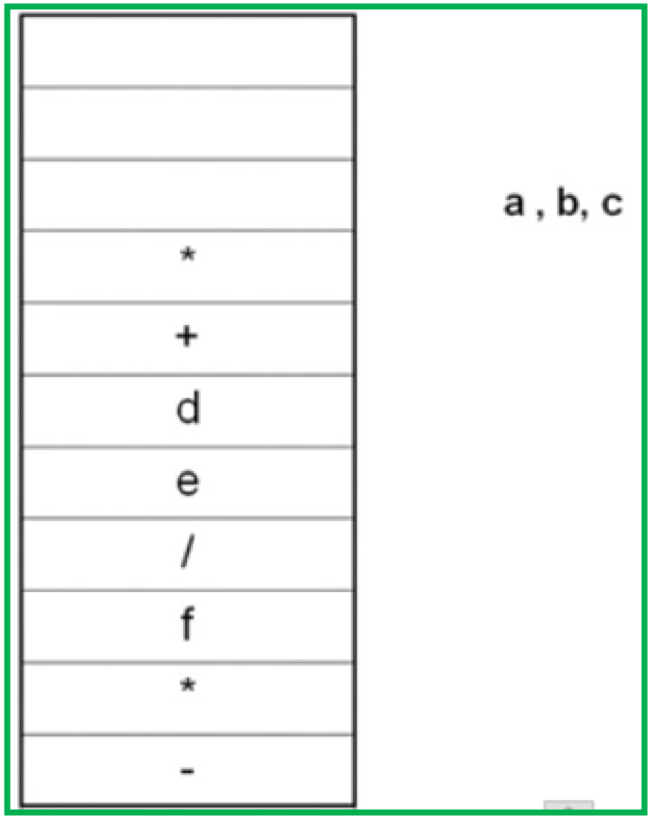
The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



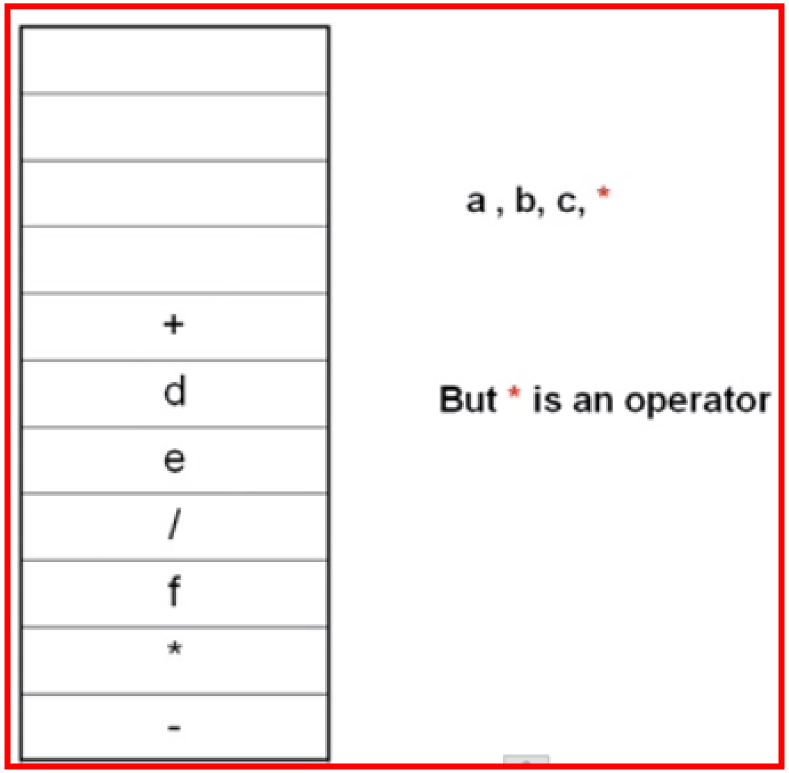


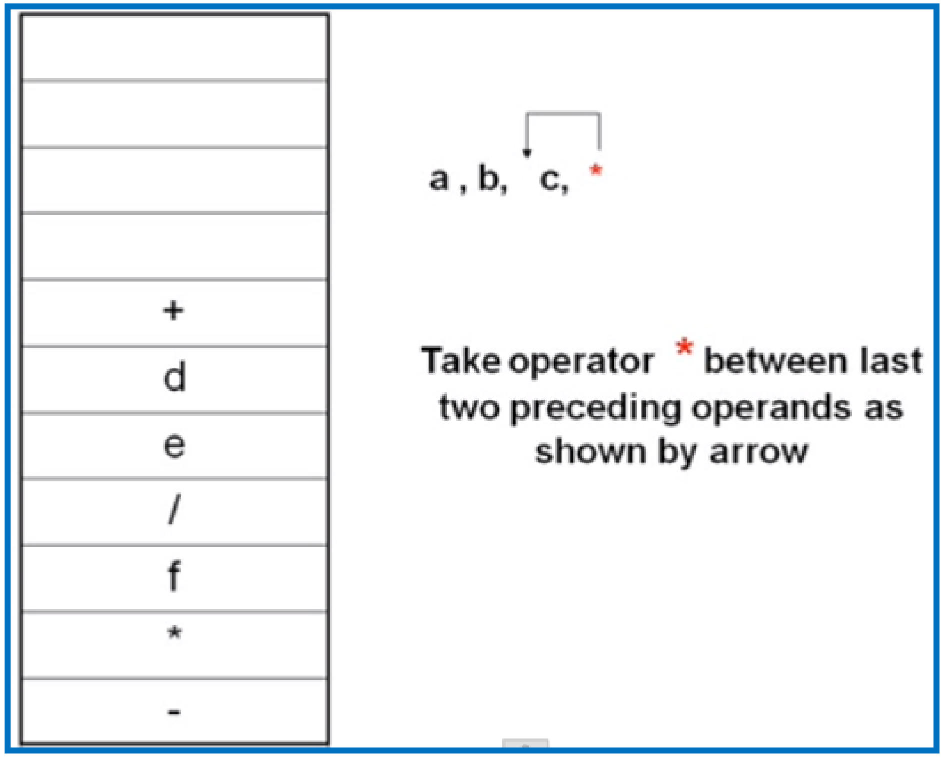


The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -

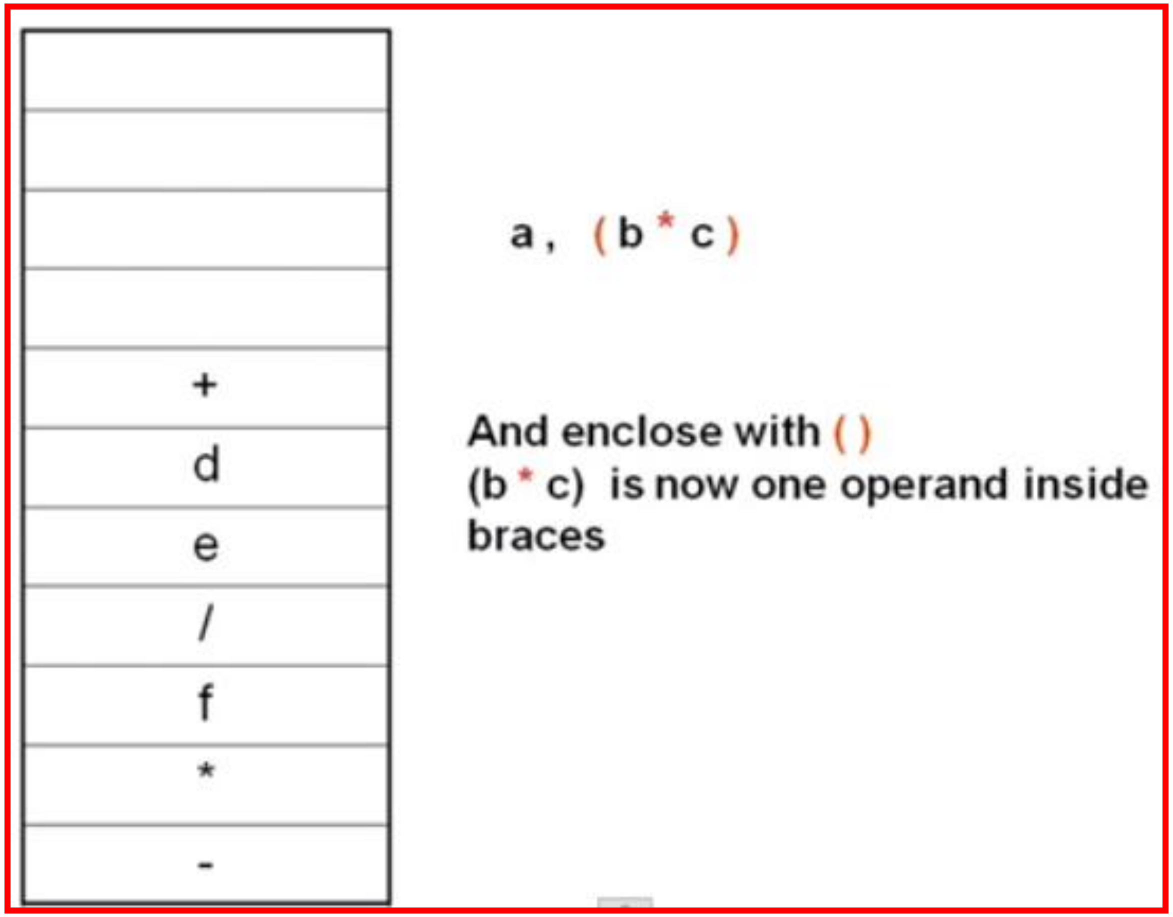



+
d
e
/
f
*
-
a , b, c, *
Take operator * between last two preceding operands as shown by arrow

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -

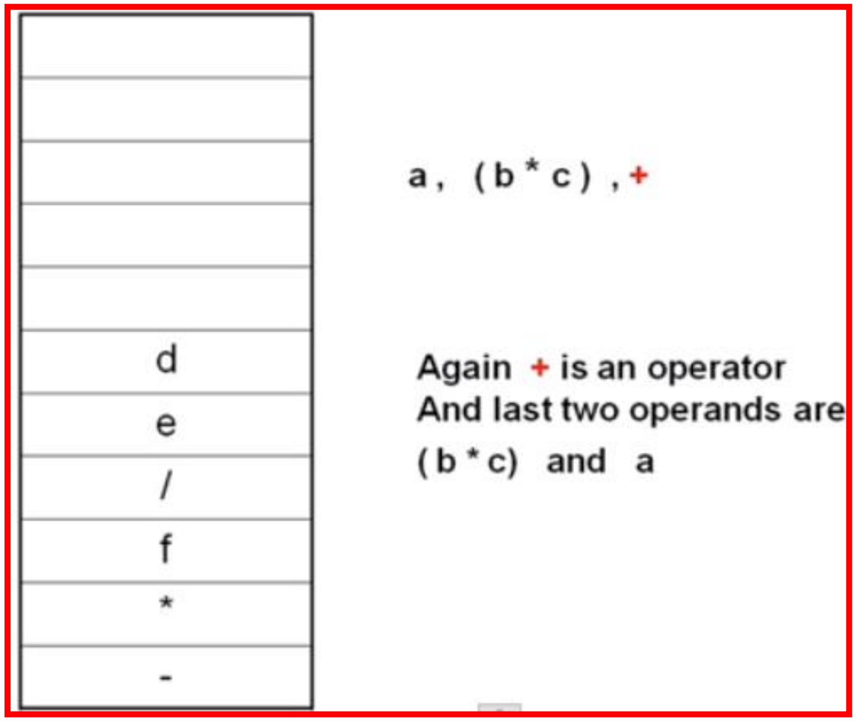


And enclose with ( )
(b * c) is now one operand inside braces

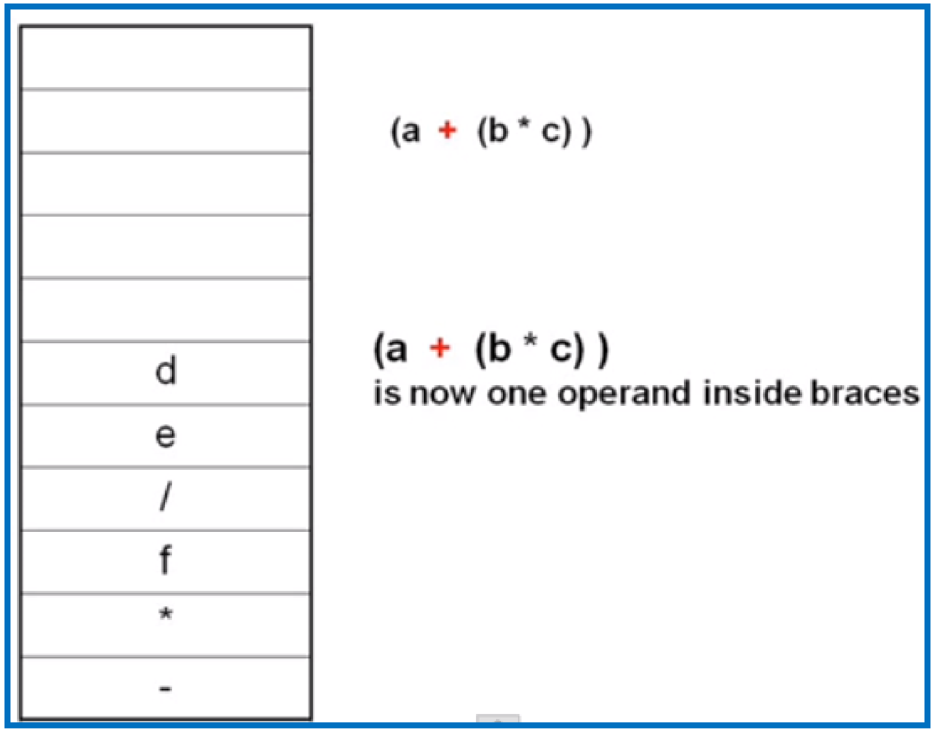
The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



Again + is an operator
And last two operands are
(b * c) and a

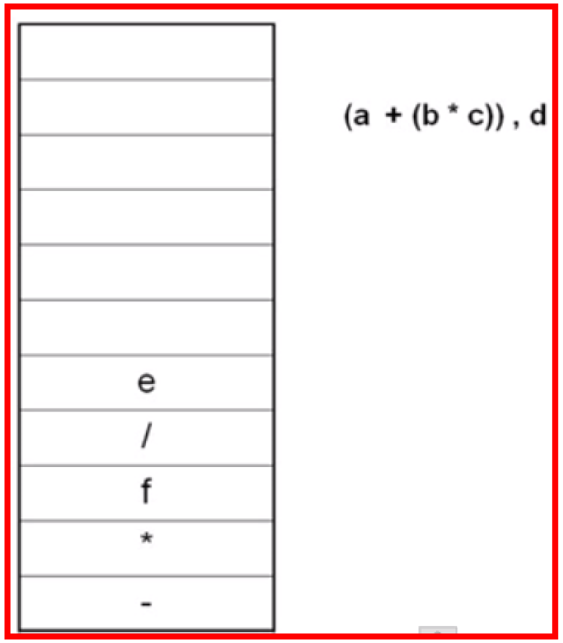


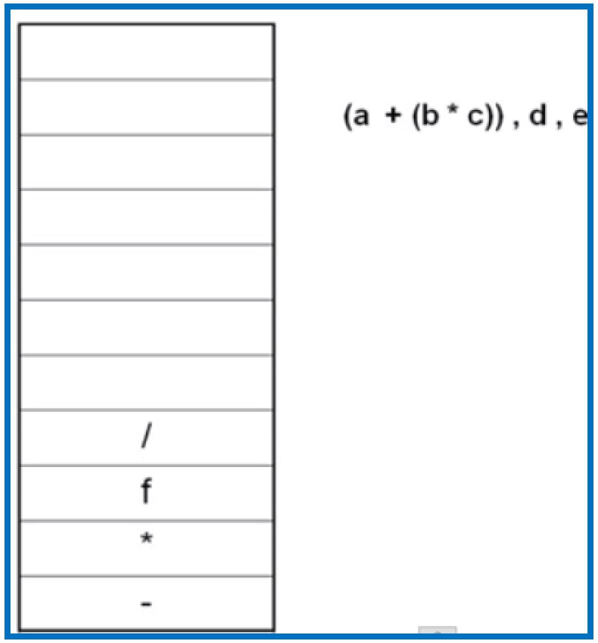
(a + (b * c))
is now one operand inside braces

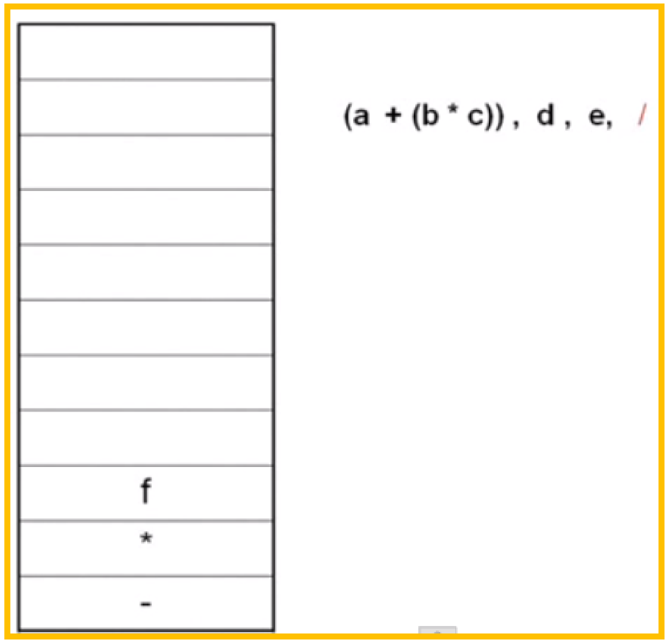
The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



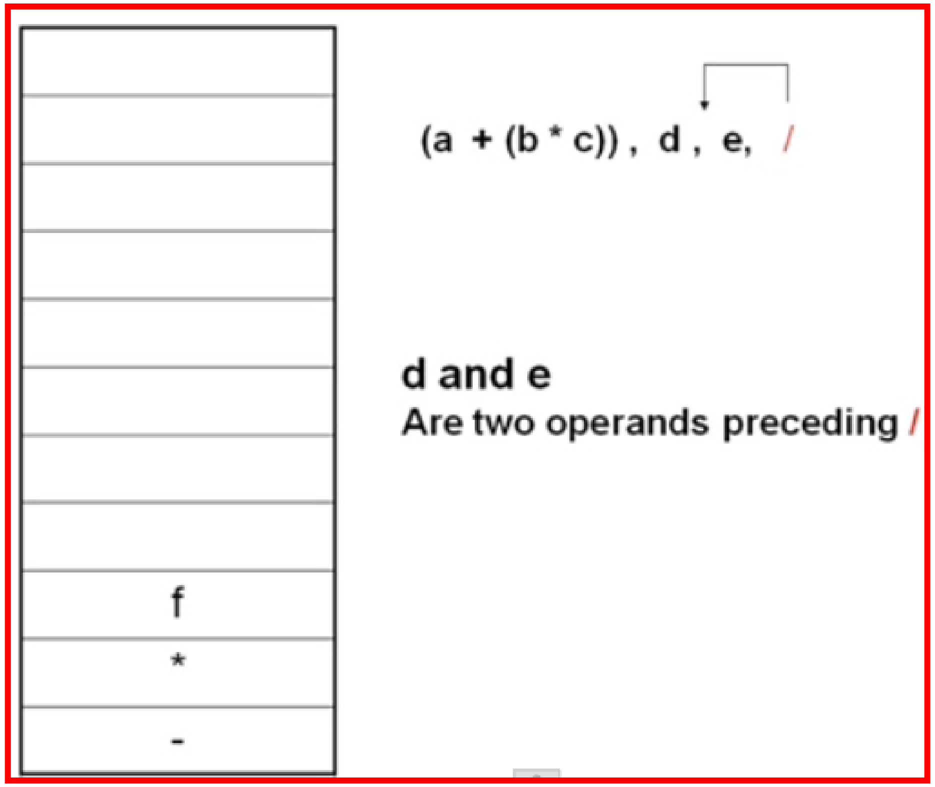




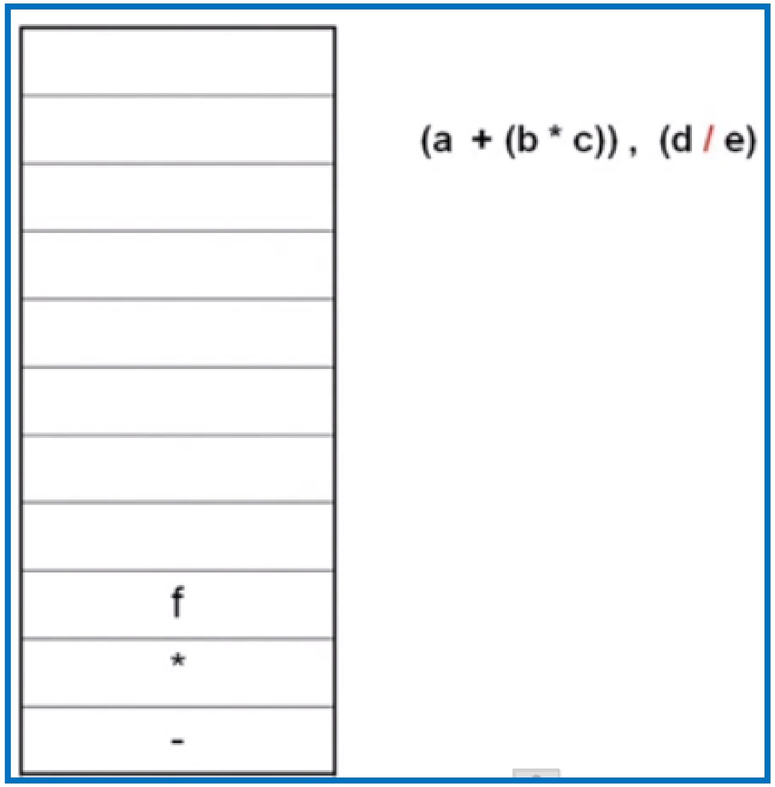
The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



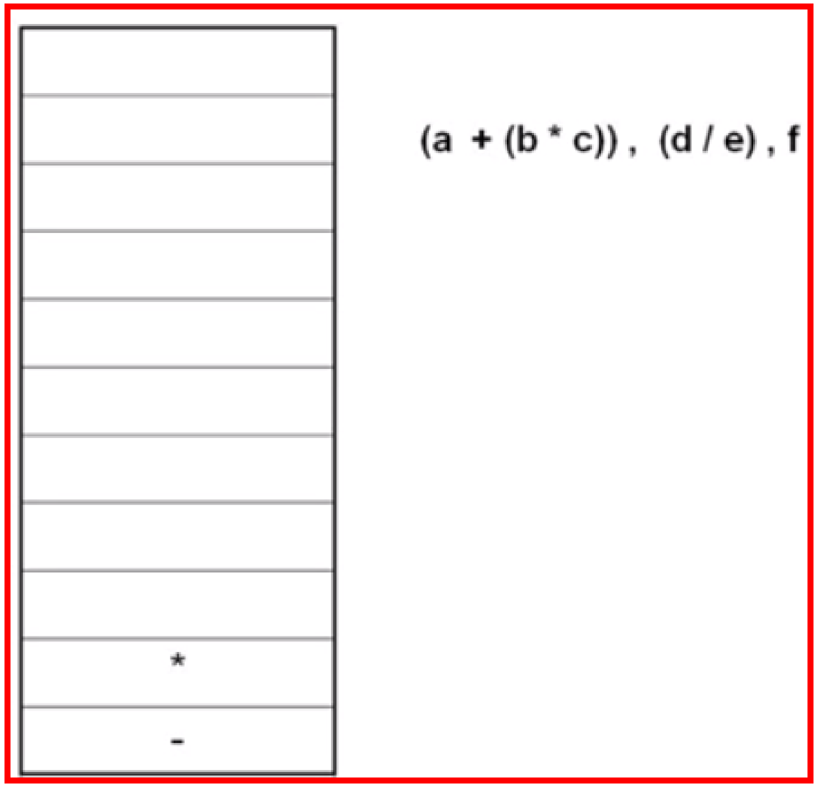
d and e
Are two operands preceding /

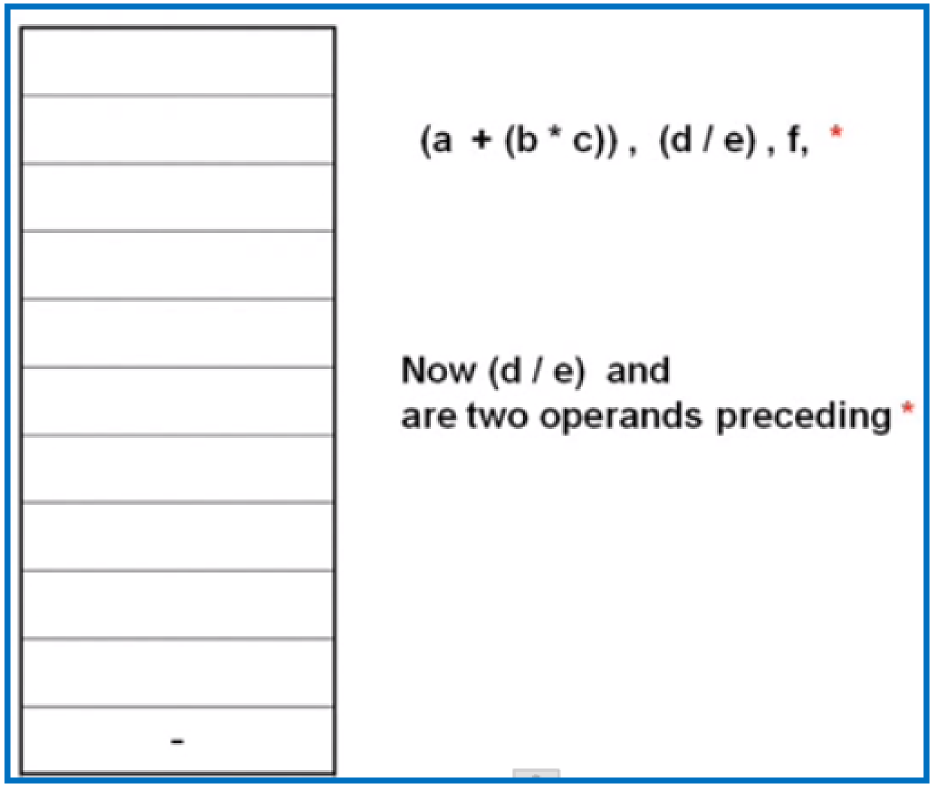


The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



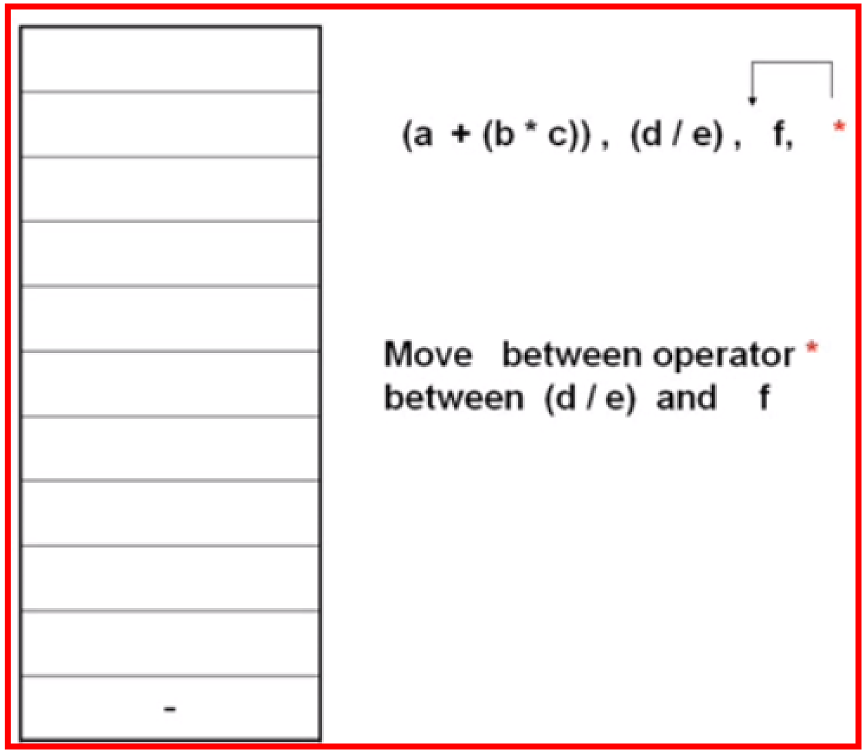


Now (d / e) and
are two operands preceding *

The Stack

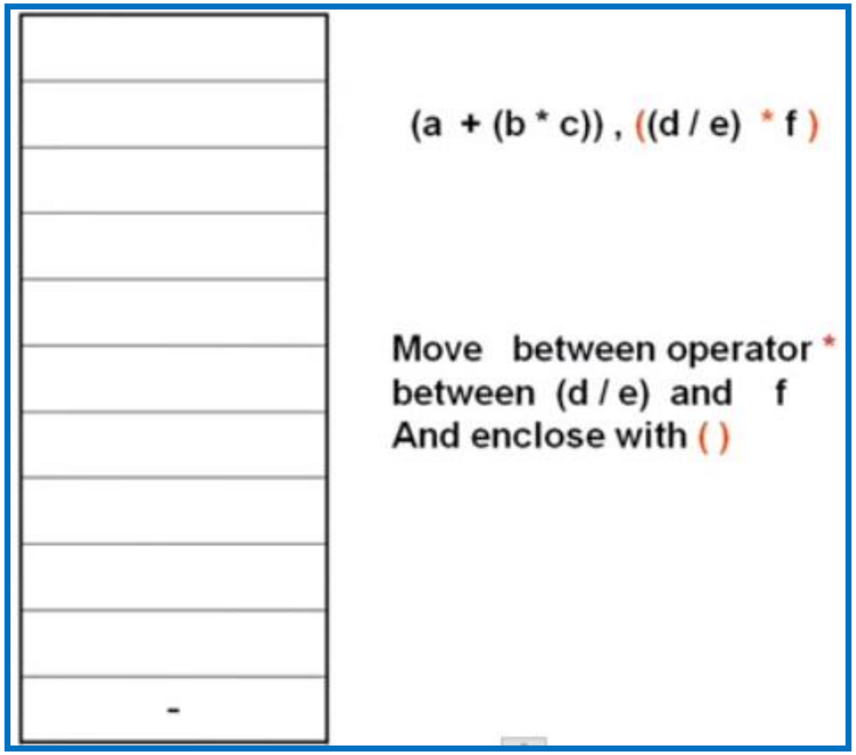
For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



(a + (b * c)) , (d / e) , f, *

Move between operator * between (d / e) and f



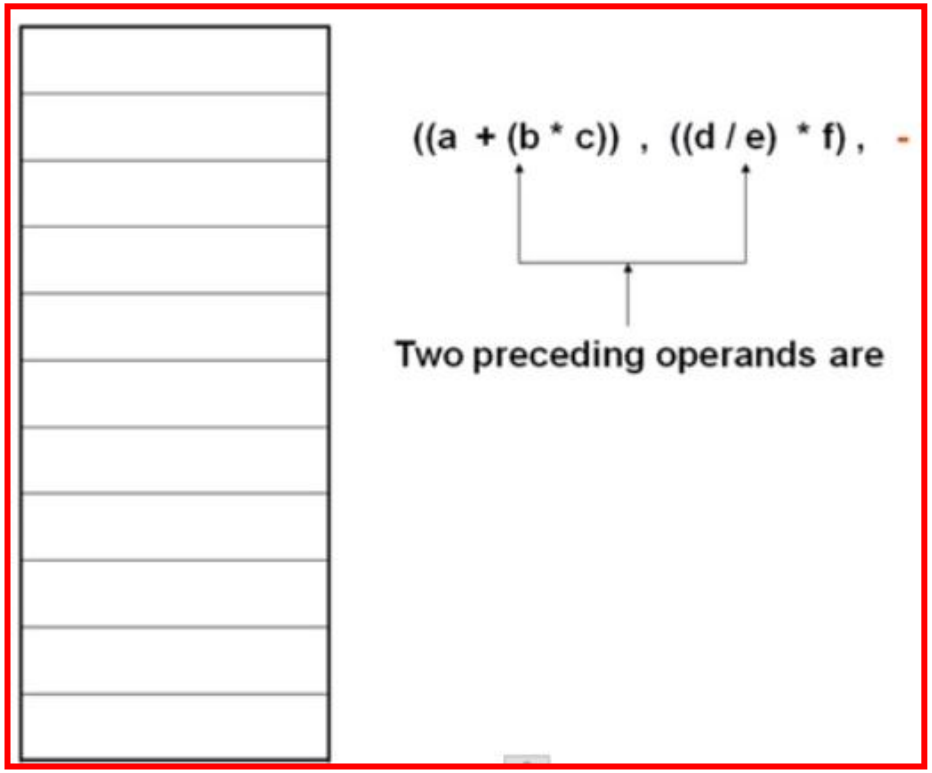
(a + (b * c)) , ((d / e) * f)

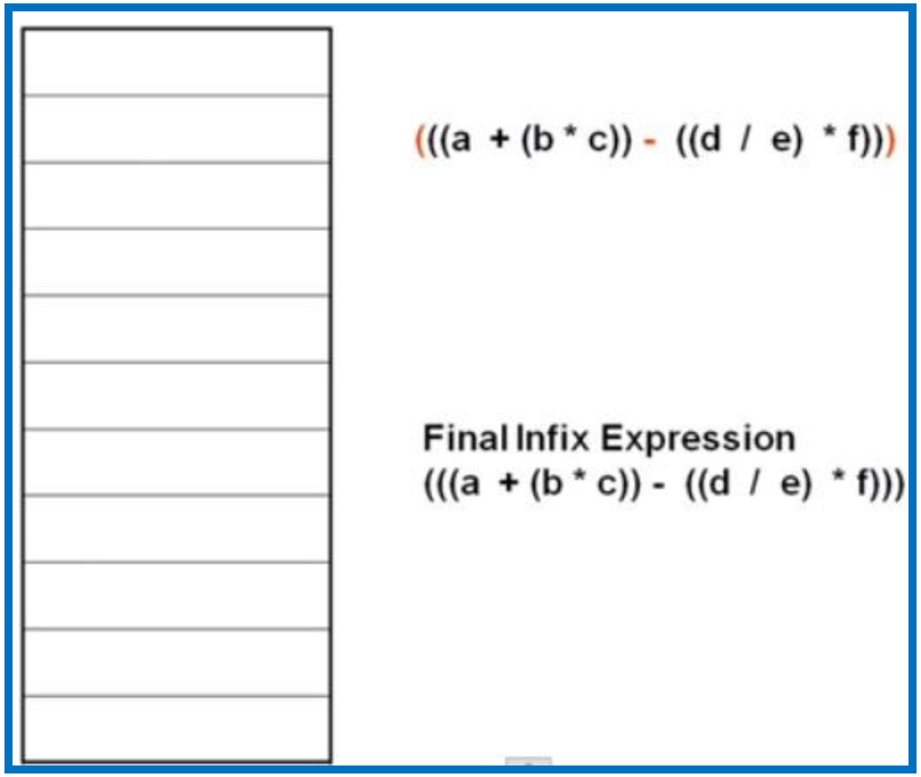
Move between operator * between (d / e) and f And enclose with ( )

The Stack

For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -


((a + (b * c)) , ((d / e) * f) , -
Two preceding operands are


(((a + (b * c)) - ((d / e) * f)))
Final Infix Expression
(((a + (b * c)) - ((d / e) * f)))

The Stack

3.3.3 CONVERTING INFIX TO PREFIX EXPRESSION

The rules to be remembered during infix to prefix conversion are:

1. **Reading Expression from “right to left” character by character.**
2. **We have Input, Prefix_Stack & Stack.**
3. **Now converting this expression to Prefix.**

The Stack

For, example consider the following arithmetic **Infix** to **Pretfix** expression

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

Input	Prefix_Stack	Stack
G	G	Empty
+	G	+
)	G	+)
)	G	+))
F	G F	+))
-	G F	+)) -
E	G F E	+)) -
(	G F E -	+)
/	G F E -	+) /

The Stack

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

Input	Prefix_Stack	Stack
)	G F E -	+) /)
D	G F E – D	+) /)
+	G F E – D	+) /) +
C	G F E – D C	+) /) +
(	G F E – D C +	+) /
*	G F E – D C +	+) / *
)	G F E – D C +	+) / *)
B	G F E – D C + B	+) / *)
+	G F E – D C + B	+) / *) +
A	G F E – D C + B A	+) / *) +
(	G F E – D C + B A +	+) / *
(	G F E – D C + B A + * /	+
Empty	G F E – D C + B A + * / +	Empty

The Stack

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

- Now pop-out Prefix_Stack to output (or simply reverse).

Prefix expression is

+ / * + A B + C D – E F G

The Stack

Assignment (2) within Lab

Write a program to do the following:

1. **Convert Infix to Postfix Expression**
2. **Convert Postfix to Infix Expression**
3. **Convert Infix to Prefix Expression**
4. **Convert Prefix to Infix Expression**

C Program to convert Prefix Expression into INFIX

The Stack

SELF REVIEW QUESTIONS

Compulsory:

Chapter three page 63:64

7, 11, 13, 14, 16, 20, 21, 23

The Stack

Any Questions?