Data Structures

Lecture 3 :

The Stack

Dr. Essam Halim Houssein
Lecturer, Faculty of Computers and Informatics,
Benha University

The Stack

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out **(LIFO).**

The Stack

The operation of the stack can be illustrated as in Fig. 3.1.

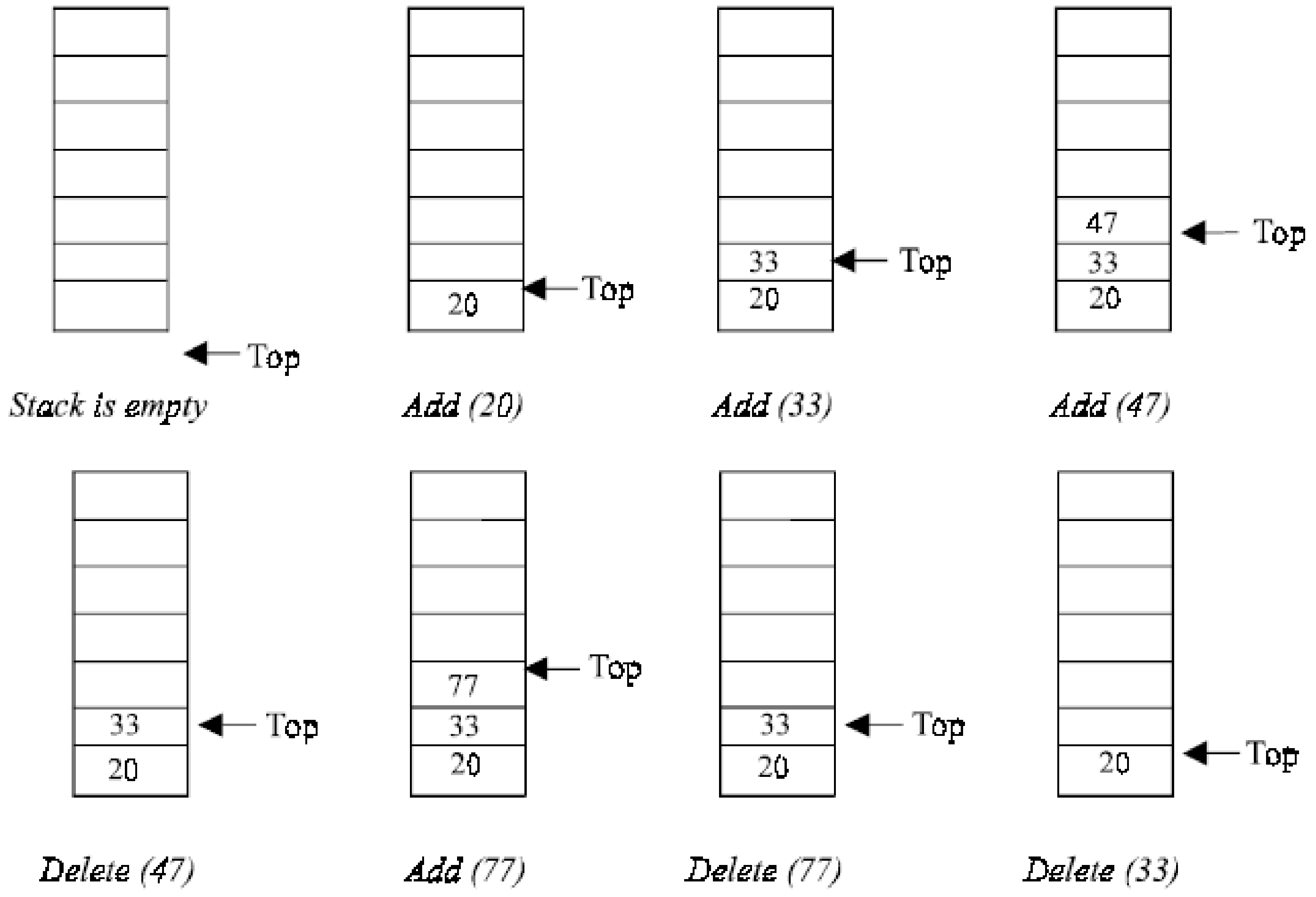


Fig. 3.1. Stack operation.

The Stack

The insertion (or addition) operation is referred to as **push**, and the deletion (or remove) operation as **pop**. 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.

The Stack

3.1. OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

The Stack

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the top is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

The Stack

3.2. STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (using pointers)

The Stack

Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied. **Moreover static implementation is not an efficient method** when resource optimization is concerned (i.e., memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be **wasted**, on the other hand if there are more number of elements to be stored in the stack (say 60) then we **cannot change the size array to increase its capacity.**

The Stack

3.3. STACK USING ARRAYS

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. **TOP** is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

The Stack

```
1. If TOP = SIZE – 1, then:
        (a) Display “The stack is in overflow condition”
        (b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit
```

The Stack

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. **TOP** is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

The Stack

```
1. If TOP < 0, then
        (a) Display "The Stack is empty"
        (b) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit
```

The Stack

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS
//PERFORMED ON STACK & IS IMPLEMENTATION USING ARRAYS
//CODED AND COMPILED IN TURBO C++
#include<iostream.h>
#include<conio.h>
//Defining the maximum size of the stack
#define MAXSIZE 100
//A class initialized with public and private variables and functions
class STACK_ARRAY
{
        int stack[MAXSIZE];
        int Top;
        public:
        //constructor is called and Top pointer is initialized to –1
        //when an object is created for the class
                STACK_ARRAY()
                {
                        Top=–1;
                }
                void push();
                void pop();
                void traverse();
};
```

The Stack

```
//This function will add/insert an element to Top of the stack
void STACK_ARRAY::push()
{
I               int item;
                //if the top pointer already reached the maximum allowed size then
                //we can say that the stack is full or overflow
                if (Top == MAXSIZE–1)
                {
                                cout<<“\nThe Stack Is Full”;
                                getch();
                }
                //Otherwise an element can be added or inserted by
                //incrementing the stack pointer Top as follows
                else
                {
                                cout<<“\nEnter The Element To Be Inserted = ”;
                                cin>>item;
                                stack[++Top]=item;
                }
}
```

The Stack

```
//This function will delete an element from the Top of the stack
void STACK_ARRAY::pop()
{
        int item;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == –1)
                cout<<“\nThe Stack Is Empty”;
        //Otherwise the top most element in the stack is poped or
        //deleted by decrementing the Top pointer
        else
        {
                item=stack[Top--];
                cout<<“\nThe Deleted Element Is = ”<<item;
        }
}
```

The Stack

```
//This function to print all the existing elements in the stack
void STACK_ARRAY::traverse()
{
        int i;
        //If the Top pointer points to NULL, then the stack is empty
        //That is NO element is there to delete or pop
        if (Top == -1)
        cout<<“\nThe Stack is Empty”;
        //Otherwise all the elements in the stack is printed
        else
        {
                cout<<“\n\nThe Element(s) In The Stack(s) is/are...”;
                for(i=Top; i>=0; i--)
                cout<<“\n ”<<stack[i];
        }
}
```

The Stack

```
void main()
{
        int choice;
        char ch;
        //Declaring an object to the class
        STACK_ARRAY ps;
        do
        {
                clrscr();
                //A menu for the stack operations
                cout<<“\n 1. PUSH”;
                cout<<“\n 2. POP”;
                cout<<“\n 3. TRAVERSE”;
                cout<<“\n Enter Your Choice = ”;
                cin>>choice;
```

The Stack

```
            switch(choice)
            {
                    case 1://Calling push() function by class object
                    ps.push();
                    break;
                    case 2://calling pop() function
                    ps.pop();
                    break;
                    case 3://calling traverse() function
                    ps.traverse();
                    break;
                    default:
                    cout<<“\nYou Entered Wrong Choice” ;
            }
            cout<<“\n\nPress (Y/y) To Continue = ”;
            cin>>ch;
    } // end do
    while(ch == ‘Y’ || ch == ‘y’);
    }
```

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

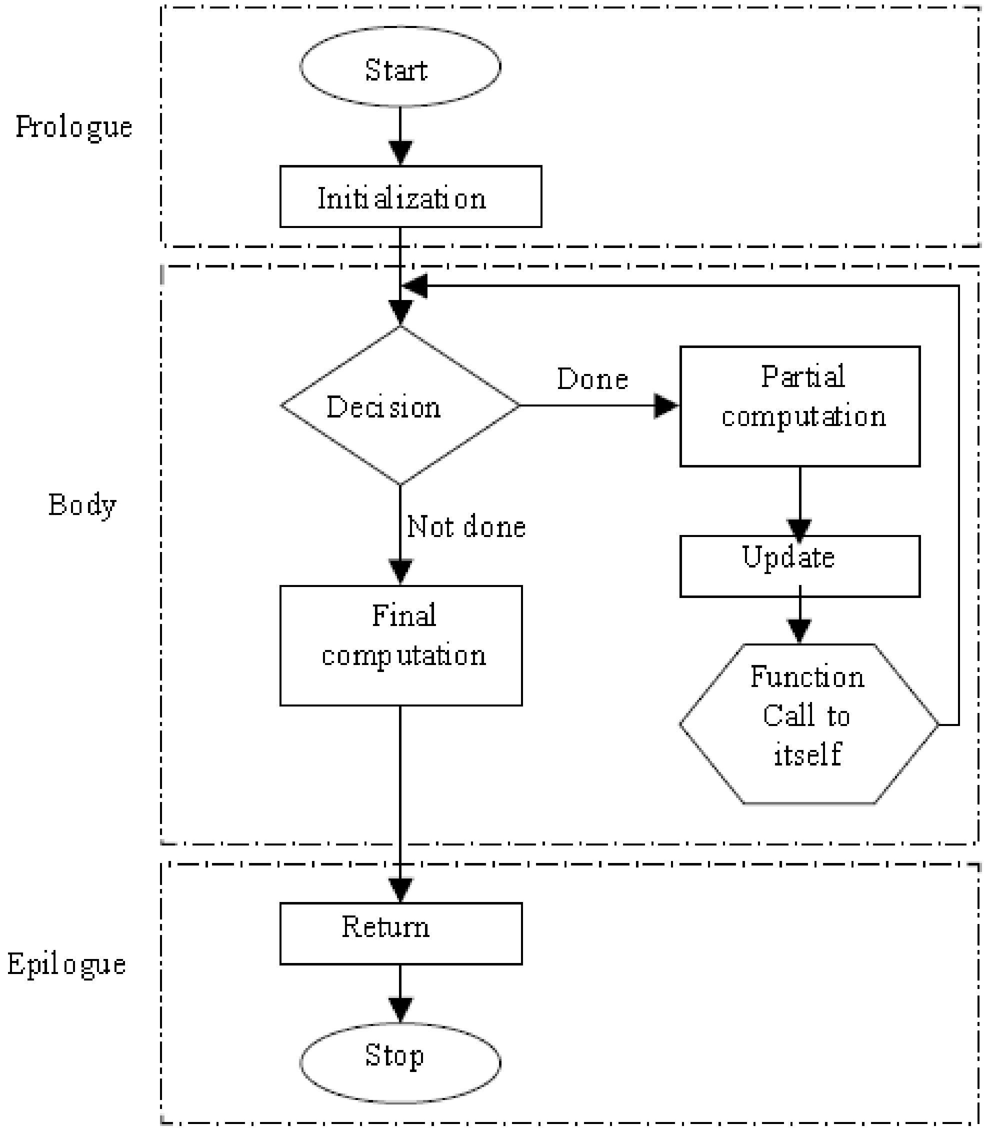


Fig. 3.2. Flowchart model for a recursive algorithm

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

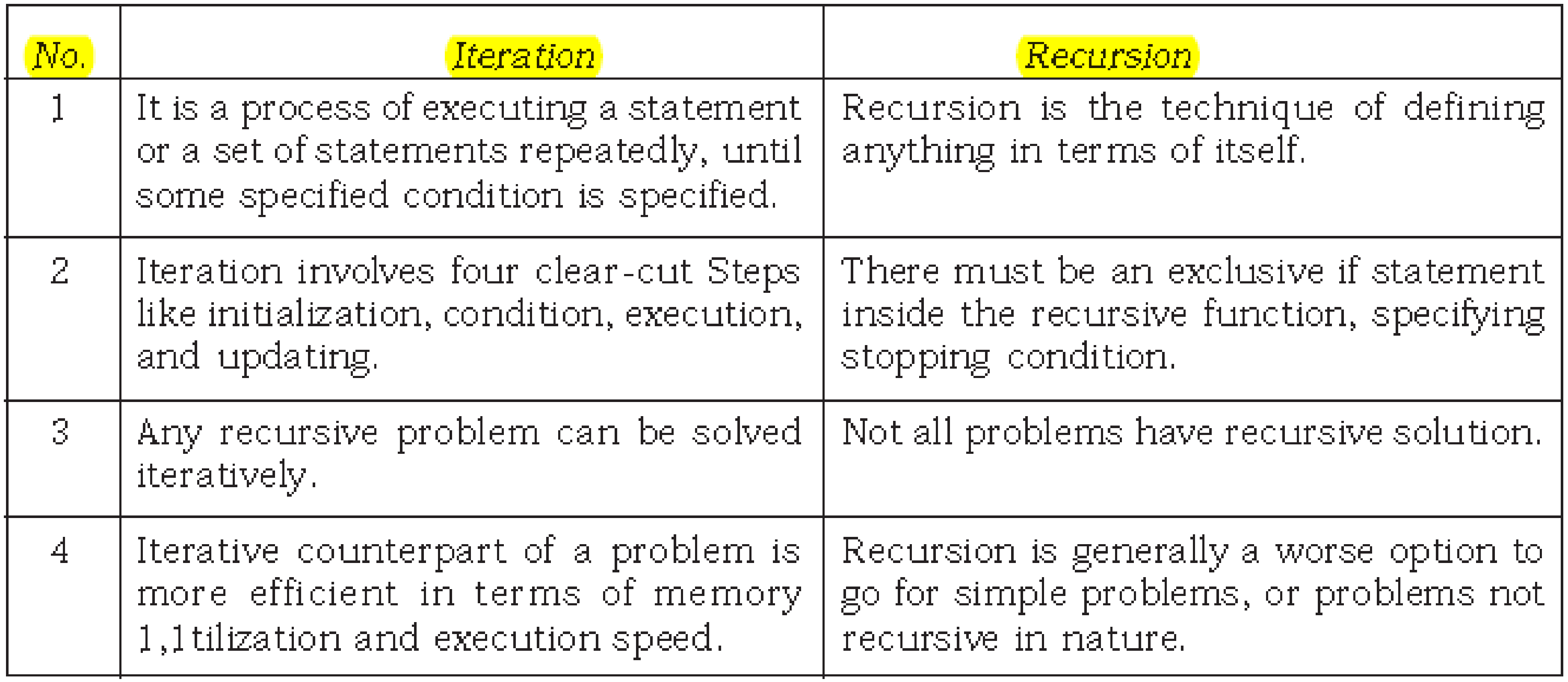
3.4.2. RECURSION vs ITERATION

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 1,1tilization 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. DISADVANTAGES OF RECURSION

1. It consumes more storage space.
2. The computer may run out of memory.
3. It is not more efficient in terms of speed and execution time.
4. Recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration.

The Stack

3.4.4. 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 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, 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 computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. ***In the preceding sections we will study the conversion of the expression from one notation to other.***

The Stack

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

Using **infix notation**, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) 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

3.5.1 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 Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator $\otimes$ is encountered, then:

   (a) Repeatedly pop from stack, if not its precedence higher precedence than $\otimes$.

   (b) Add $\otimes$ 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

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

Sr. No	Symbol Scanned	STACK	Expression (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

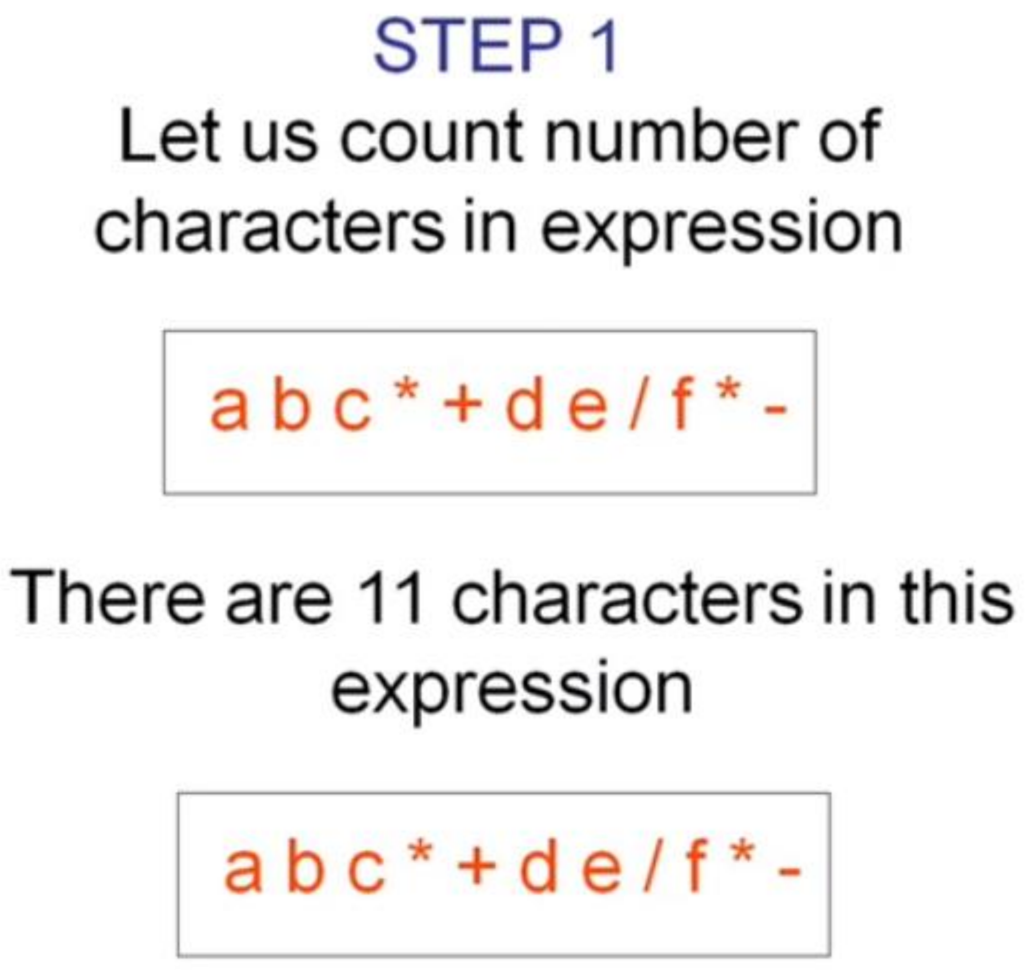
3.5.2 CONVERTING POSTFIX TO INFIX EXPRESSION

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 * -

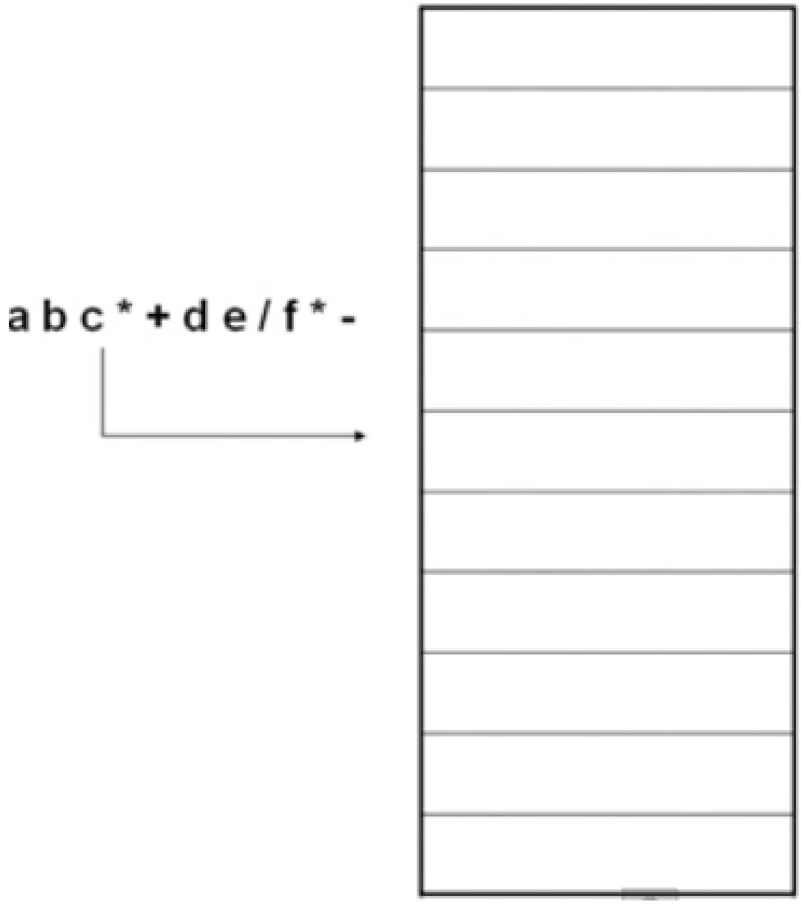
There are 11 characters in this expression

a b c * + d e / f * -

The Stack

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

A B C * + D E / F * -



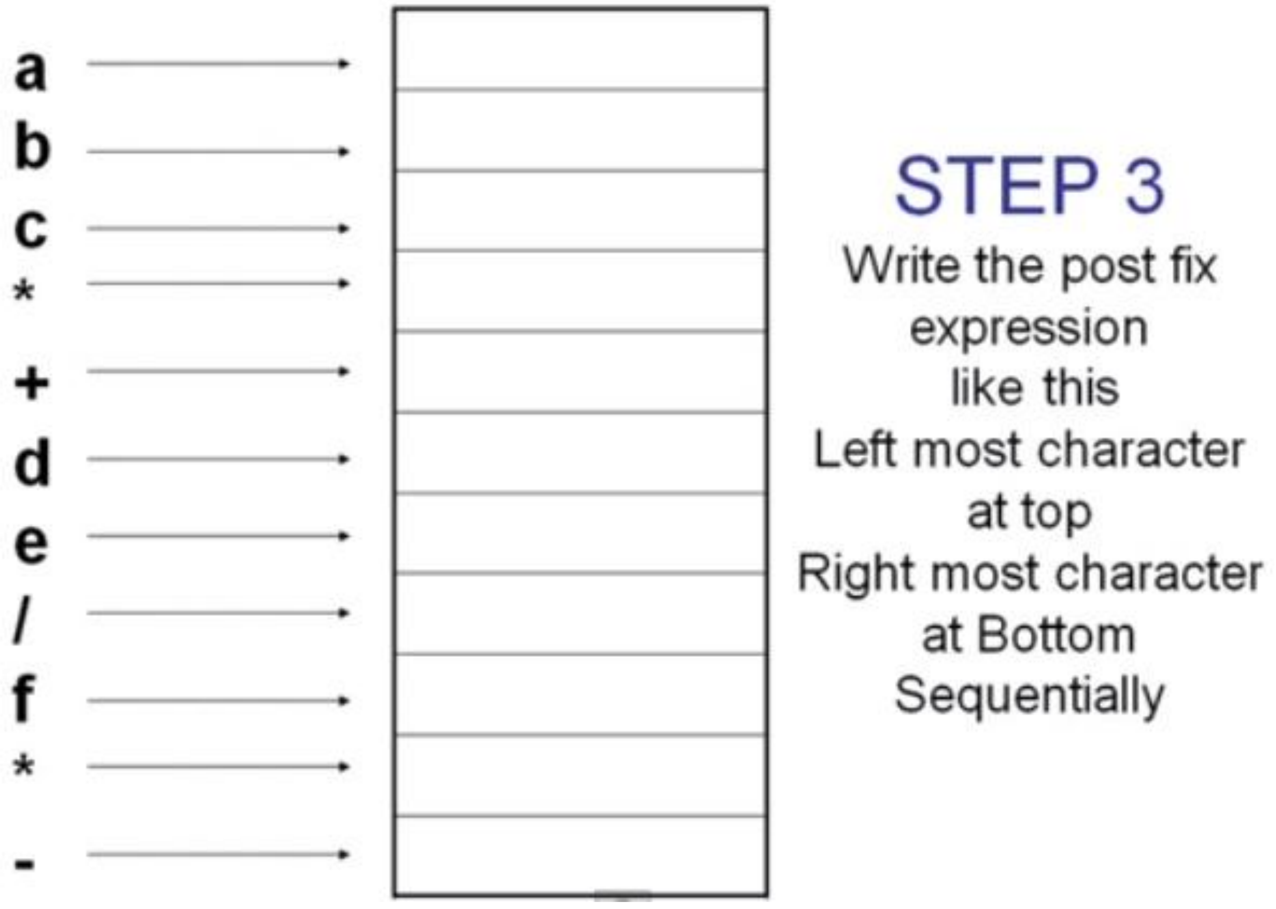
STEP 2

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

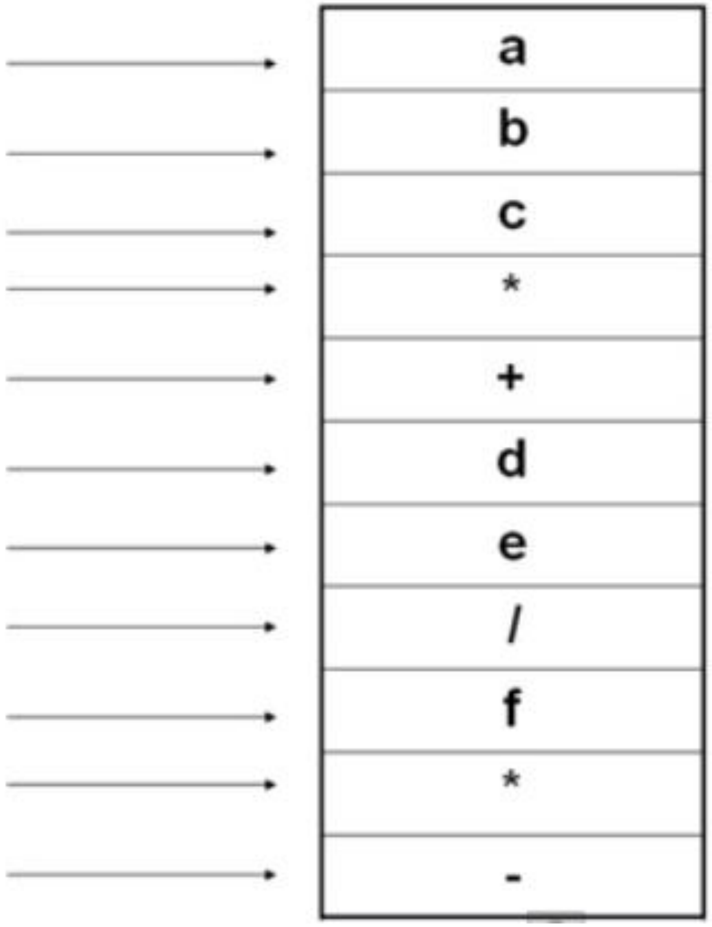
A B C * + D E / F * -


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 * -



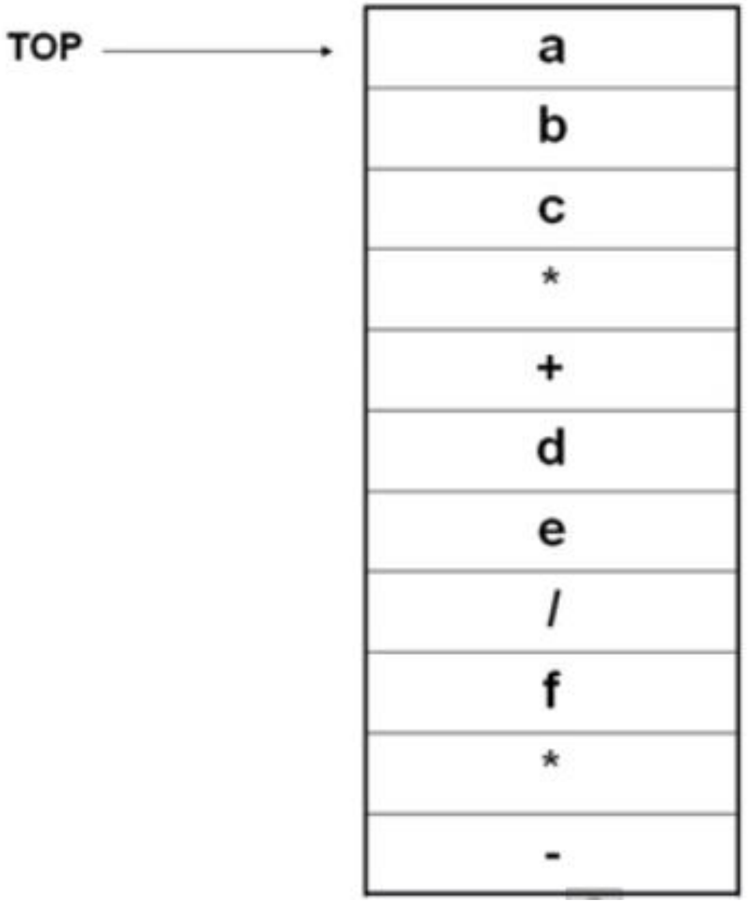
STEP 4

Fill up the Stack Sequentially

The Stack

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

A B C * + D E / F * -



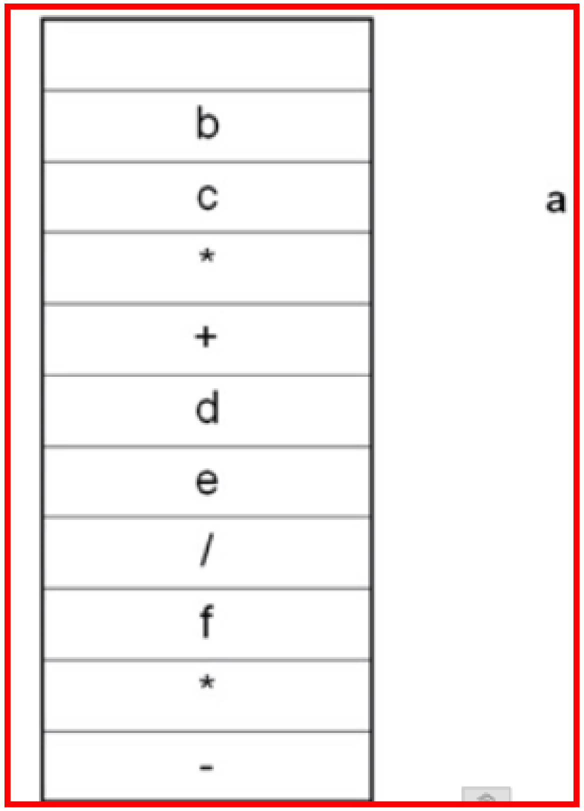
STEP 5

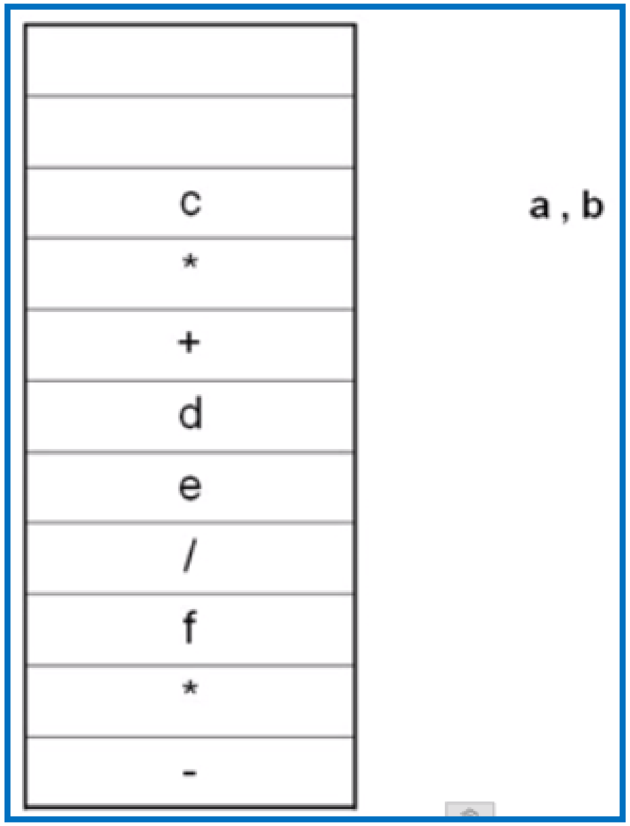
Apply POP operation on all elements one by one starting from TOP of STACK

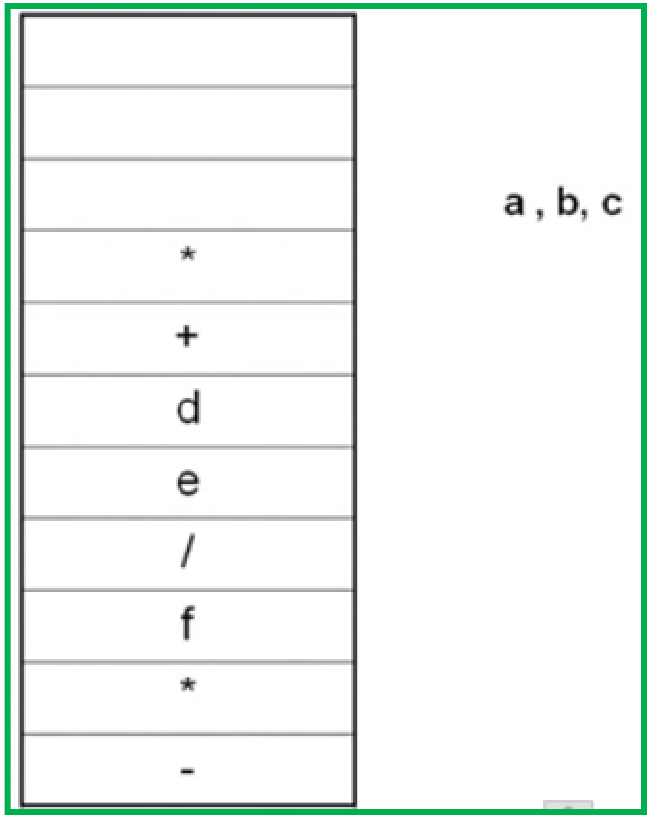
The Stack

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

A B C * + D E / F * -



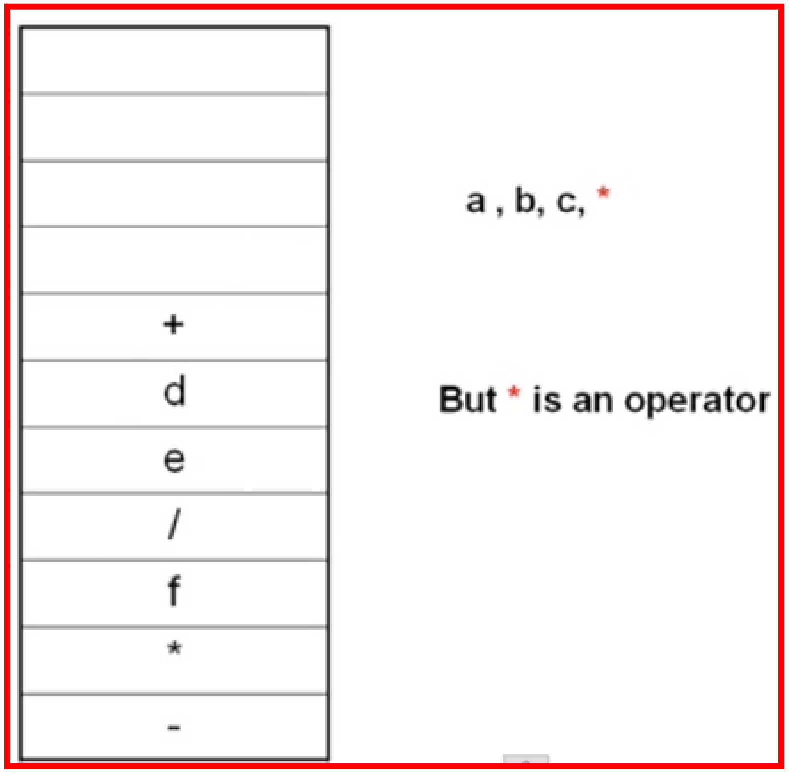


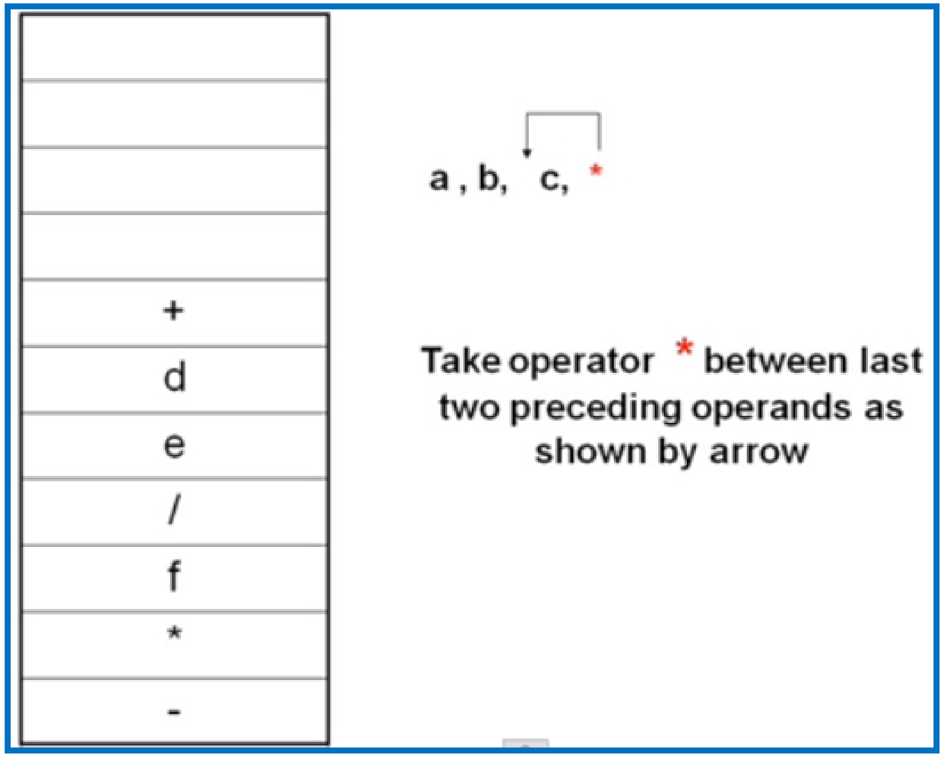


The Stack

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

A B C * + D E / F * -

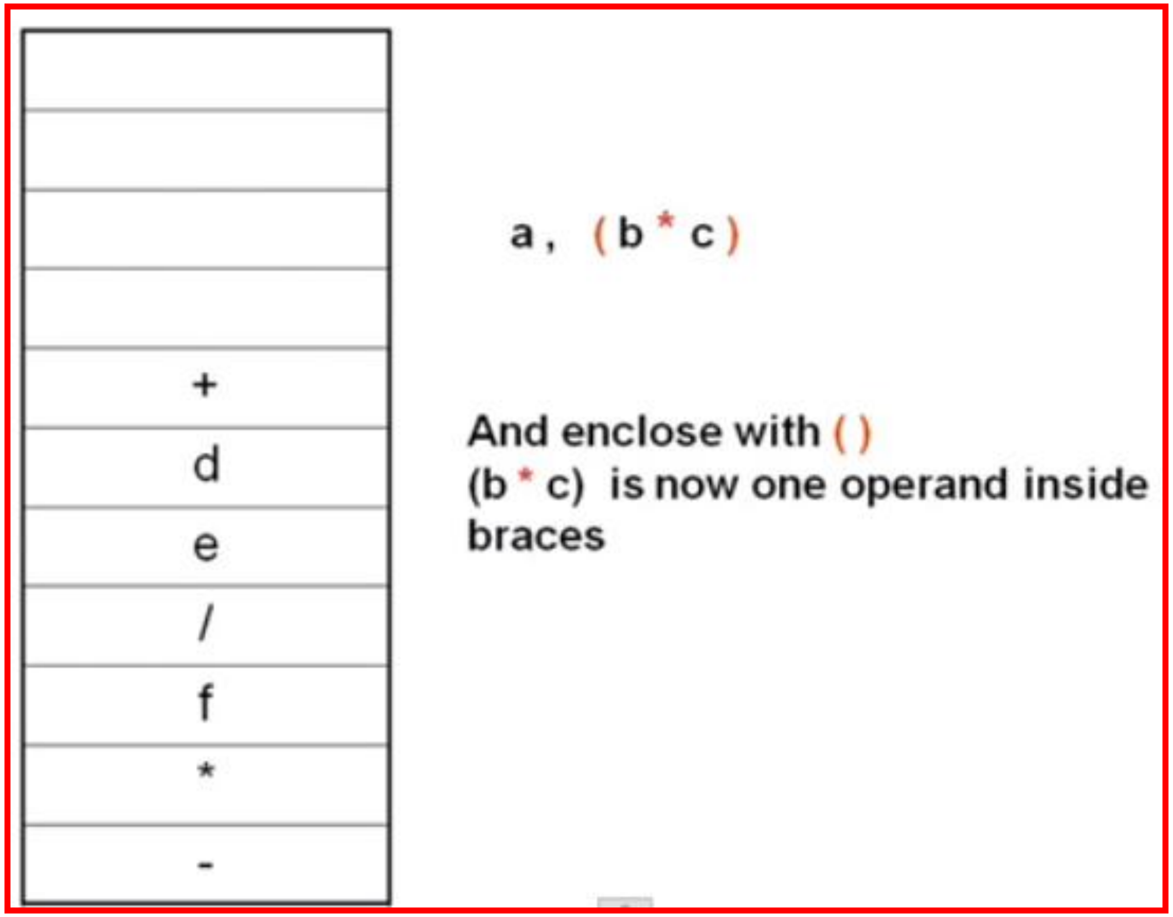



+
d
e
/
f
*
-
a , b, c, *
Take operator * between last two preceding operands as shown by arrow

The Stack

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

A B C * + D E / F * -

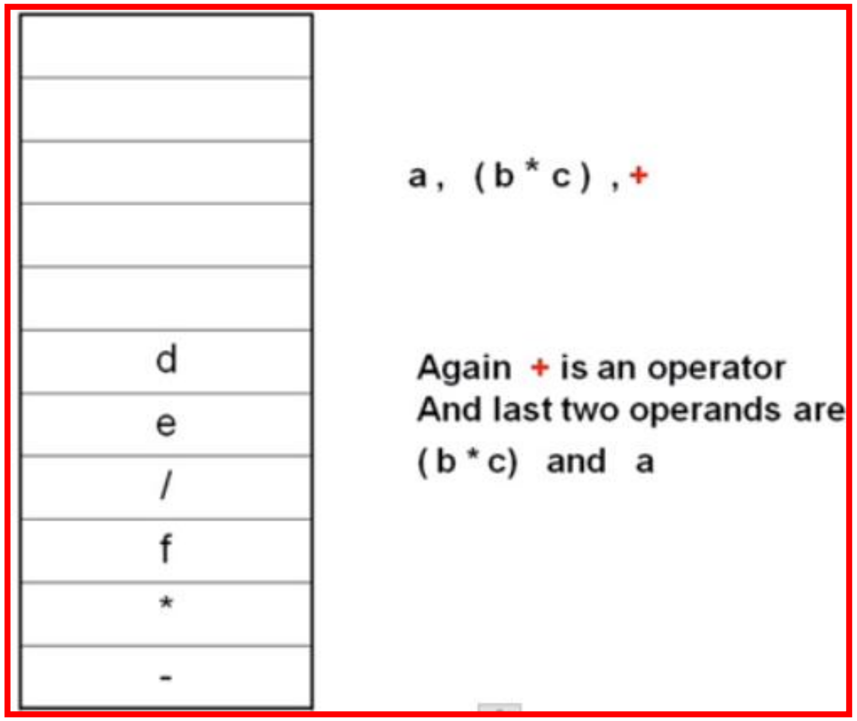


And enclose with ( )
(b * c) is now one operand inside braces

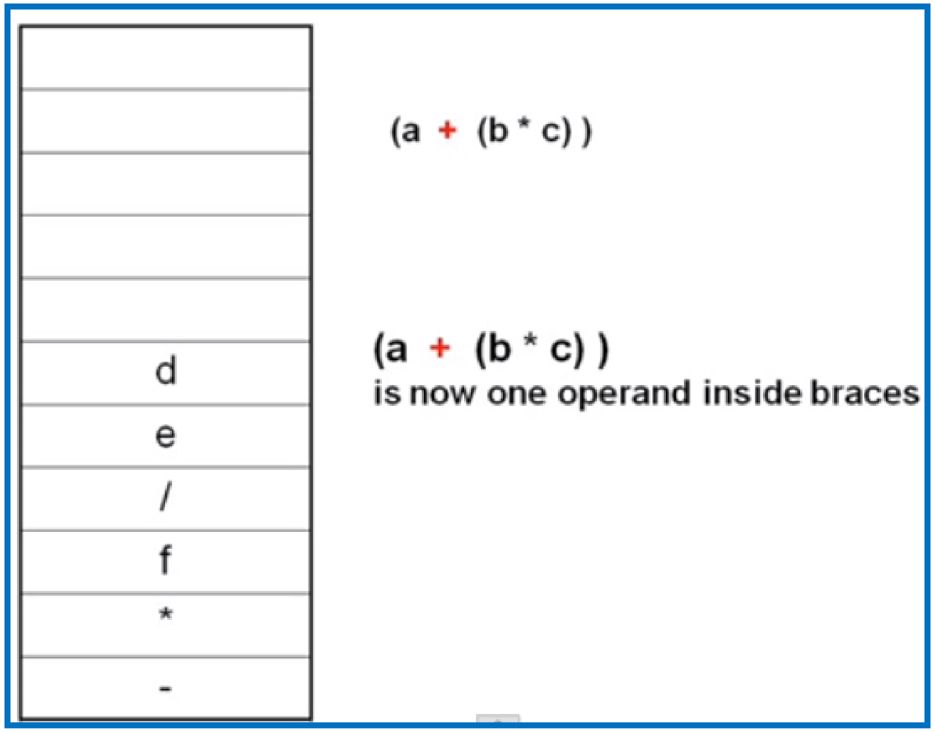
The Stack

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

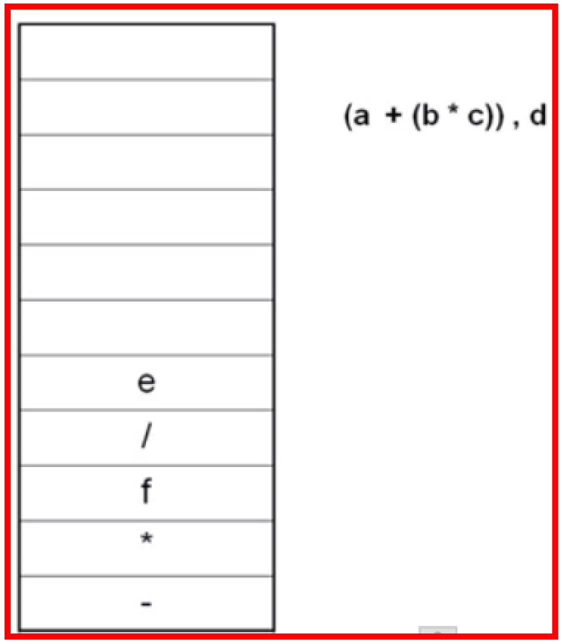


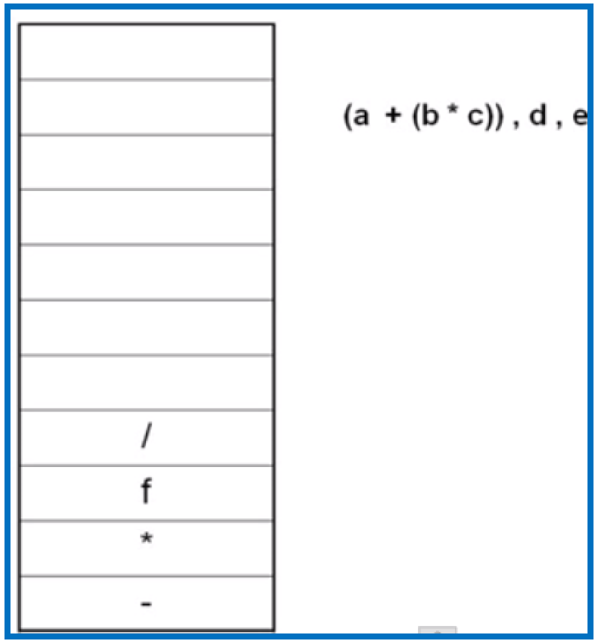
(a + (b * c))
is now one operand inside braces

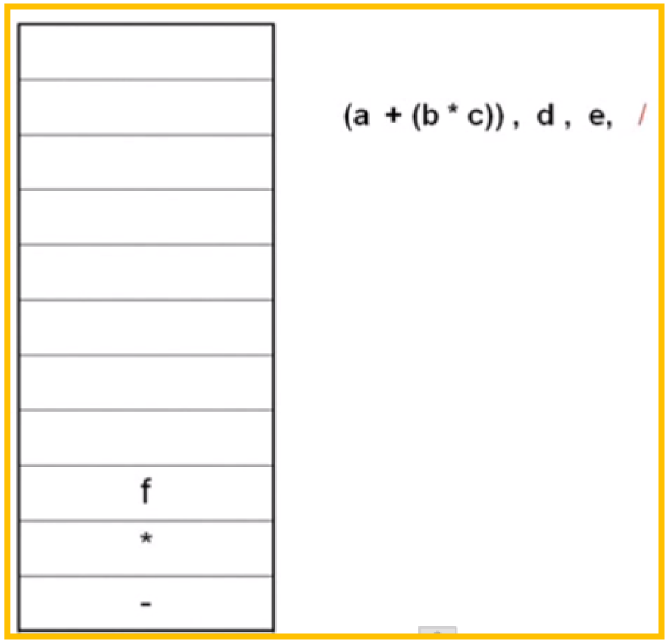
The Stack

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

A B C * + D E / F * -



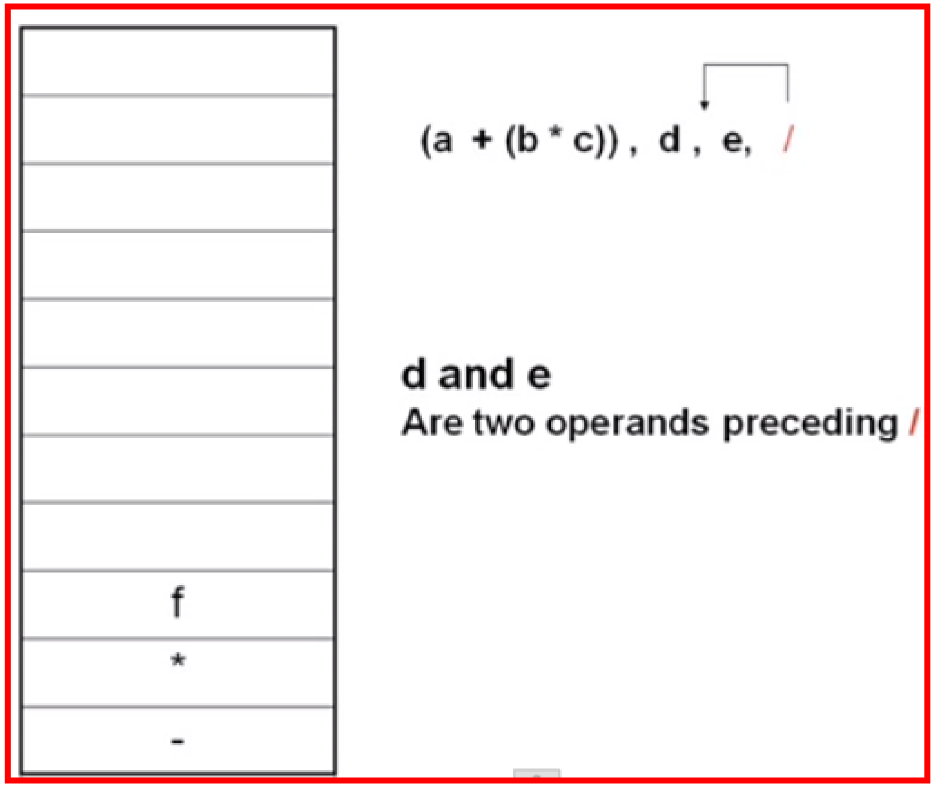


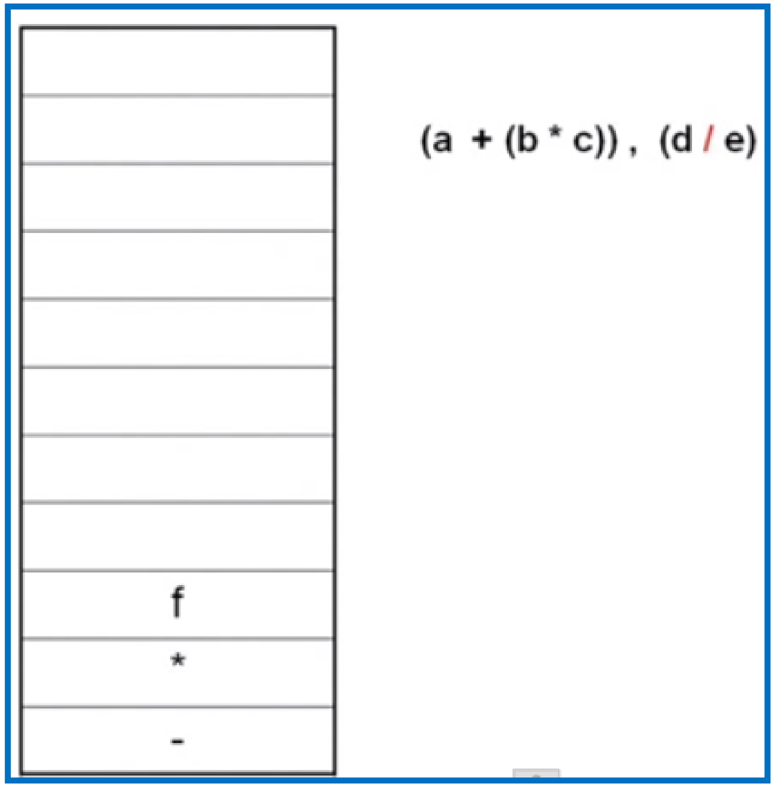


The Stack

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

A B C * + D E / F * -

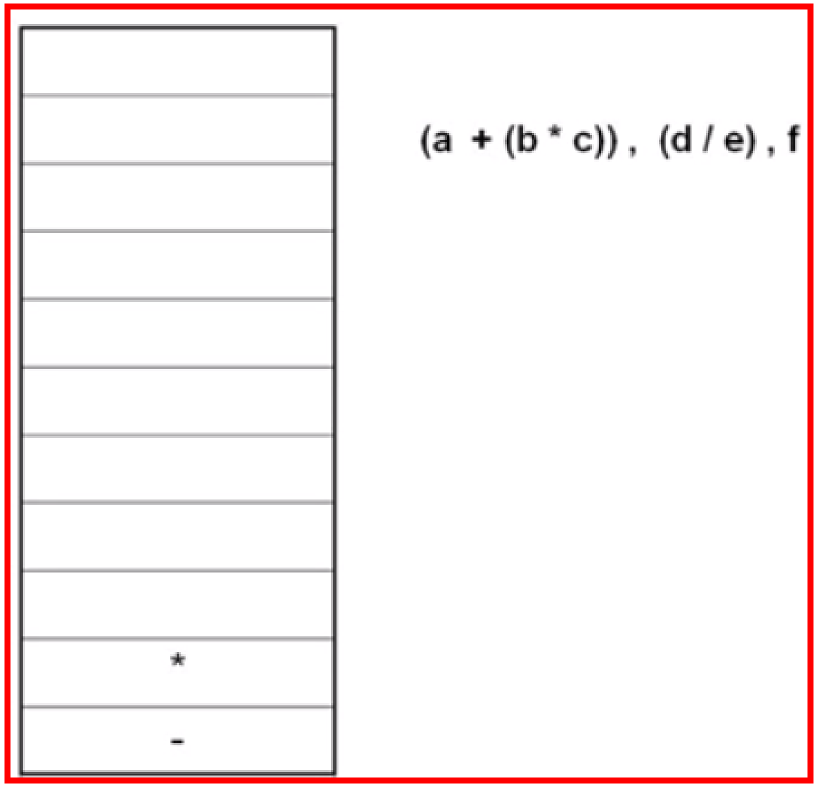


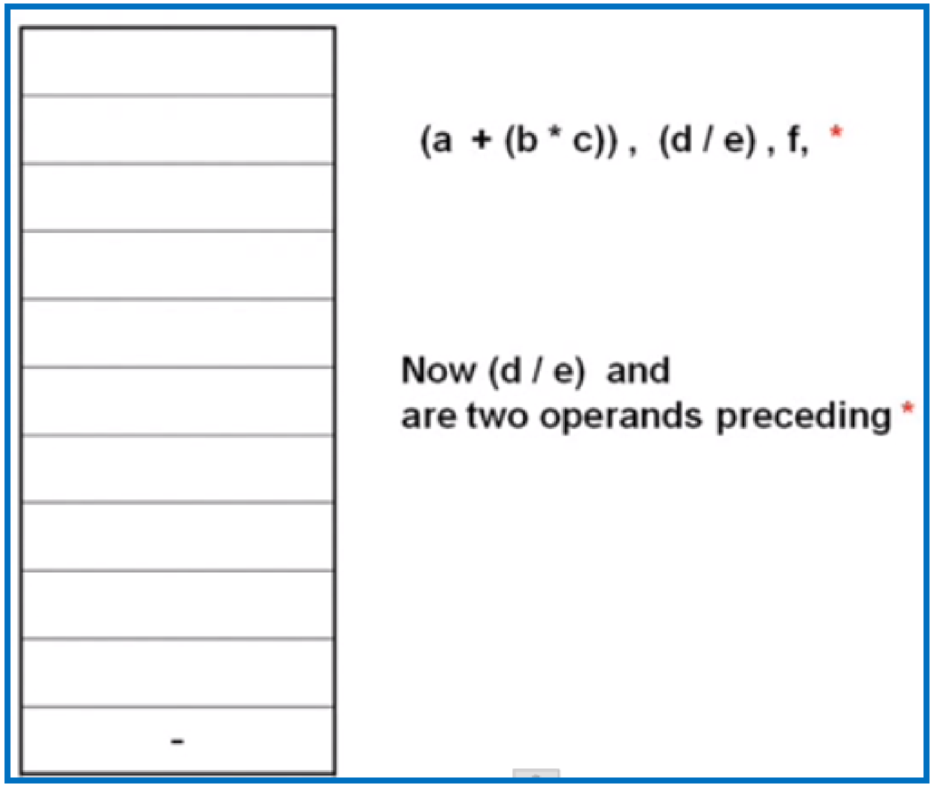


The Stack

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

A B C * + D E / F * -

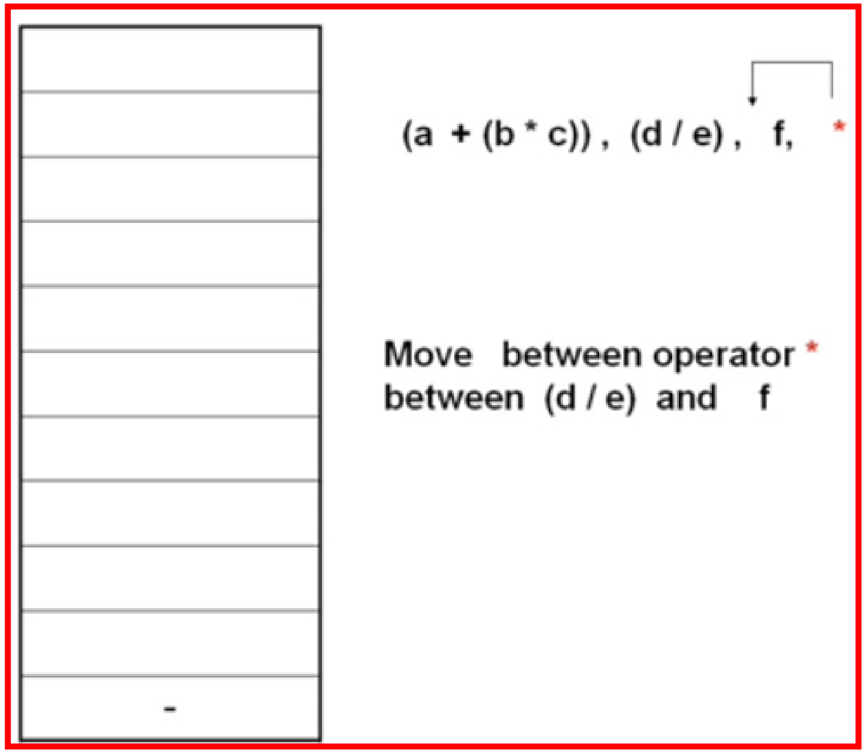




The Stack

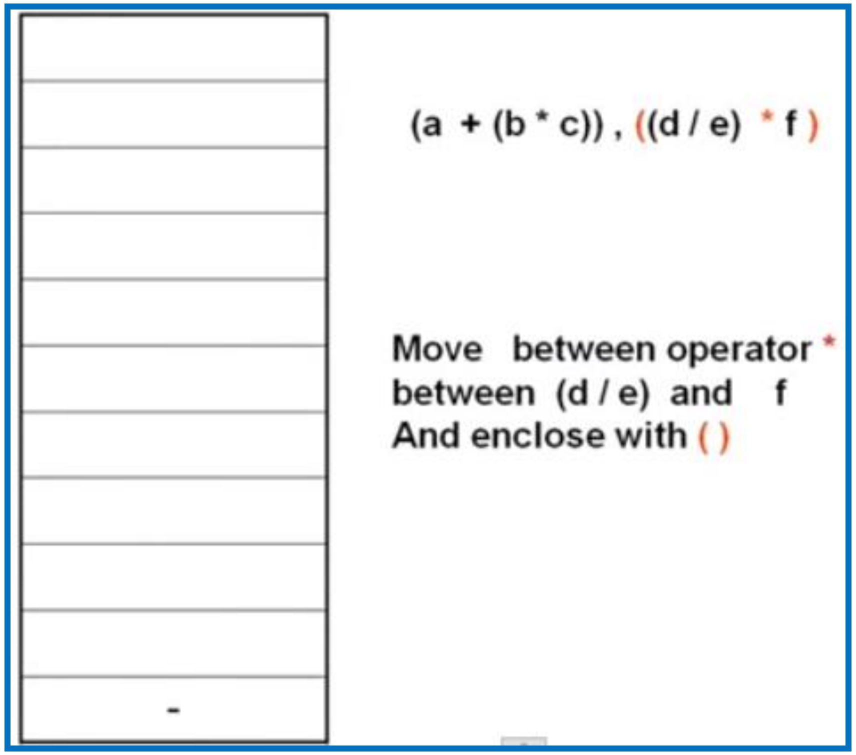
For, example consider the following arithmetic **Postfix** to **Infix** expression

A B C * + D E / F * -



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

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



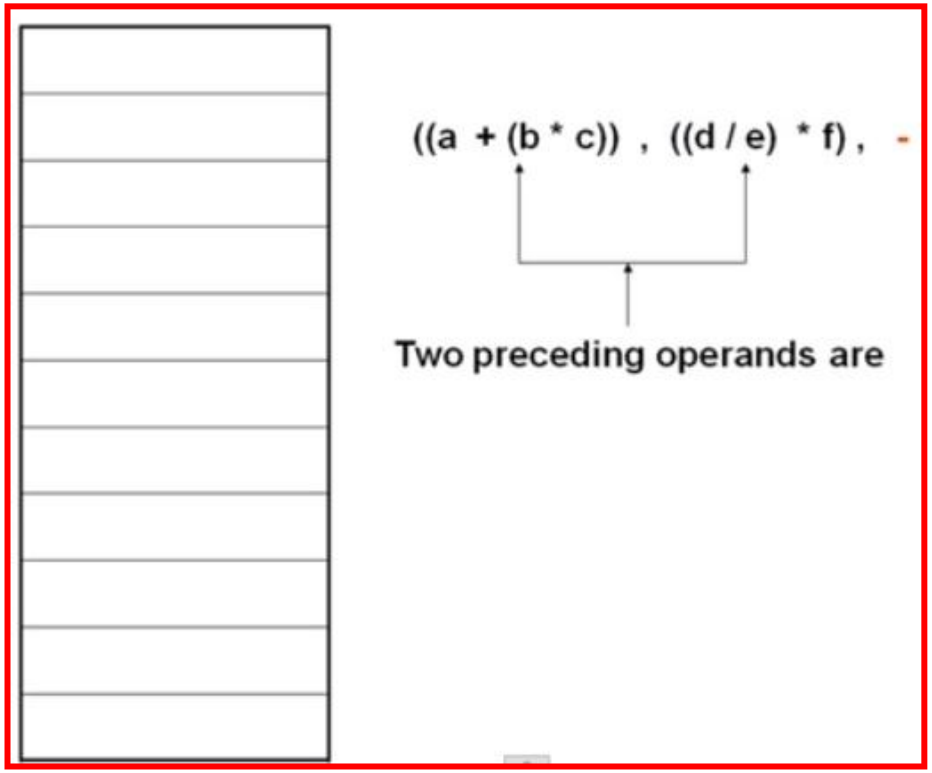
(a + (b * c)) , ((d / e) * f)

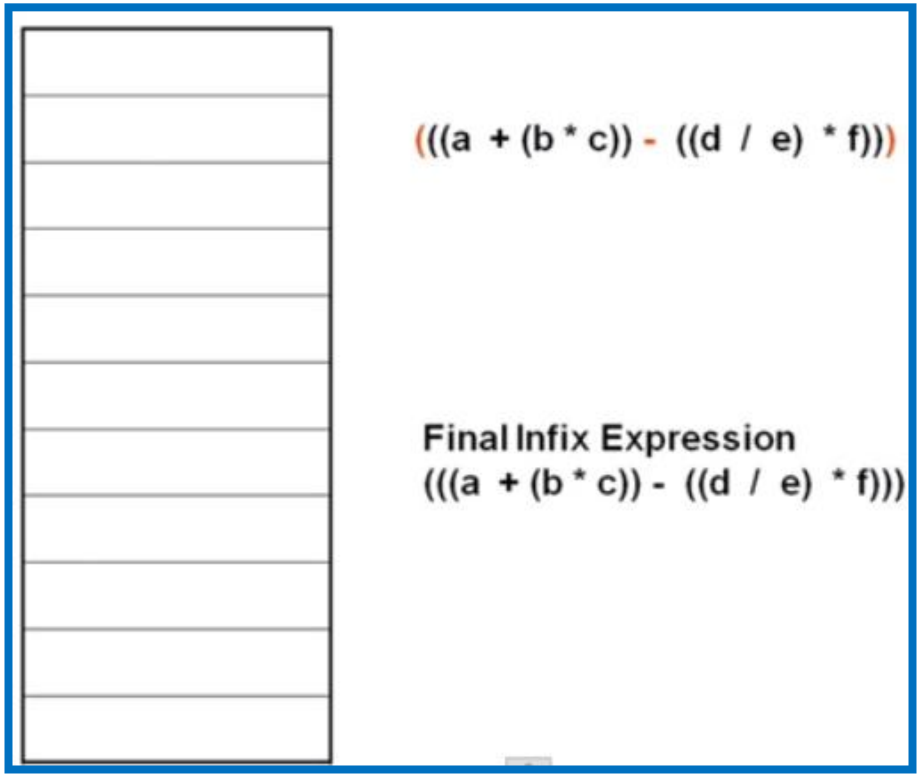
Move between operator *
between (d / e) and f
And enclose with ( )

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

3.5.3 CONVERTING INFIXTO PRETFIX EXPRESSION

The rules to be remembered during infix to postfix 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

Any Questions?