DATA STRUCTURE & ALGORITHMS

PROBLEM SOLVING WITH STACKS

PROBLEM SOLVING WITH STACKS

Many mathematical statements contain nested parenthesis like :-

- (A+(B*C)) + (C – (D + F))

We have to ensure that the parenthesis are nested correctly, i.e. :-

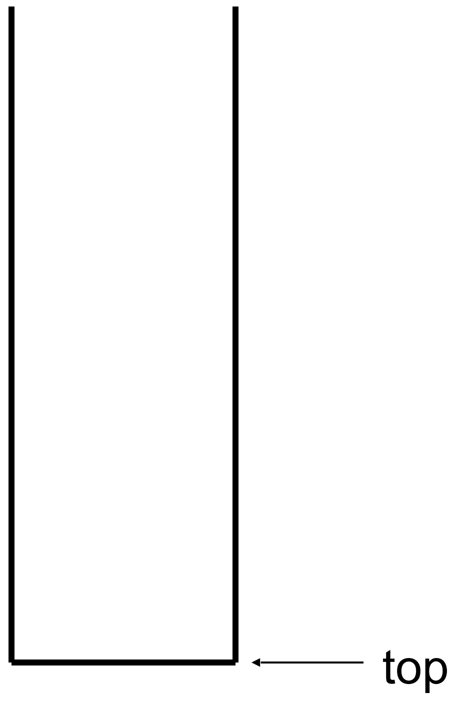
1. There is an equal number of left and right parenthesis
2. Every right parenthesis is preceded by a left parenthesis

Expressions such as ((A + B) violate condition 1

And expressions like) A + B (- C violate condition 2

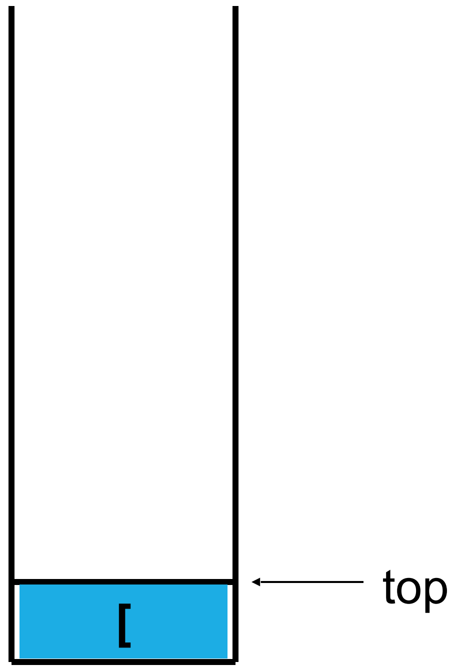
What if there are many braces () [] {} ?

Stack in Action ....



[A + { B - C + (D + E) }]

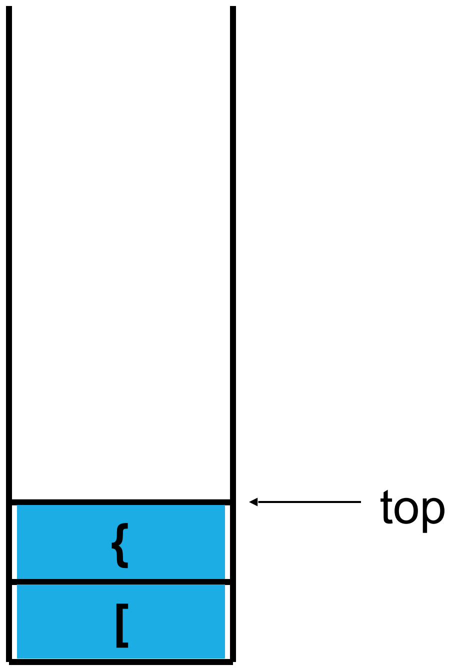
Stack in Action ….



Push(‘[‘);

[A + { B - C + (D + E) }]

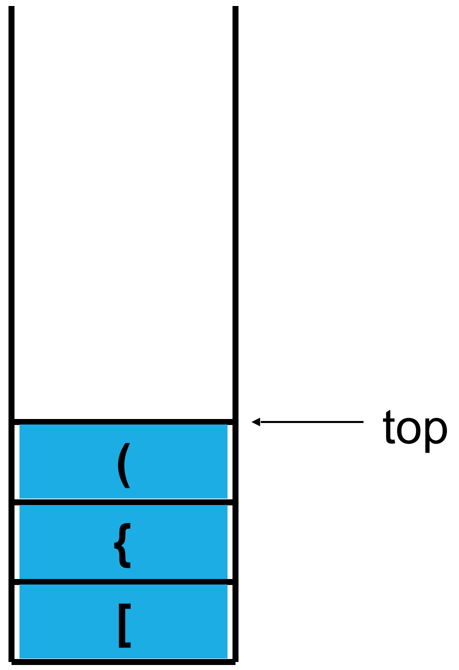
Stack in Action ….



Push(‘{‘);

[A + { B - C + (D + E) }]

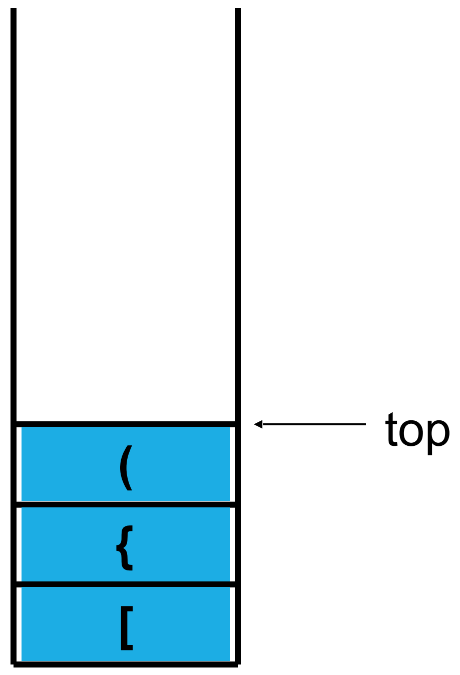
Stack in Action ….



Push(‘(‘);

[A + { B - C + (D + E) }]

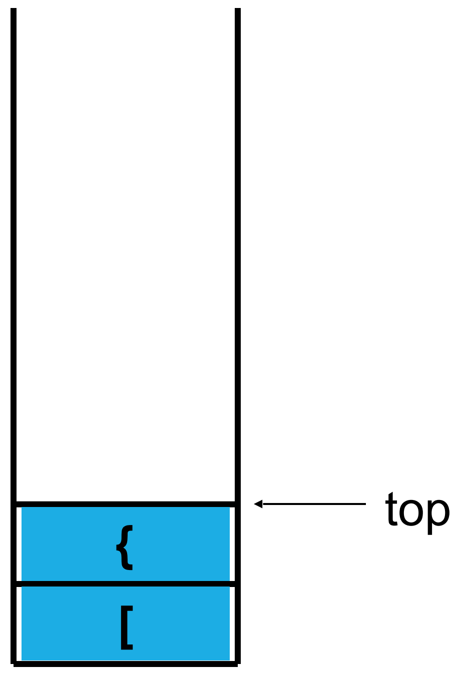
Stack in Action ….



Pop( );

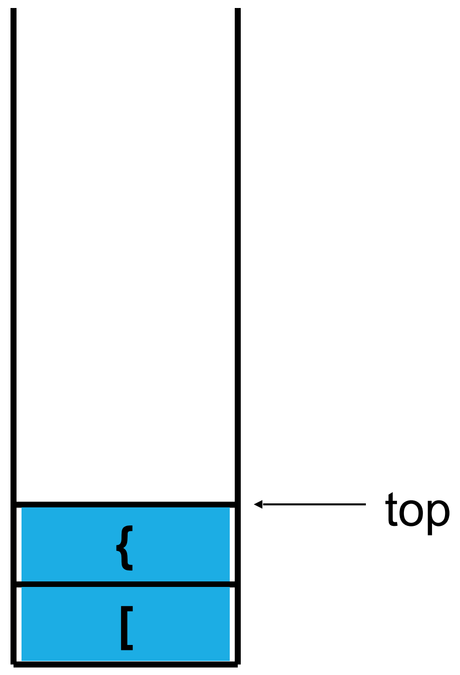
[A + { B - C + (D + E) }]

Stack in Action ….



[A + { B - C + (D + E) }]

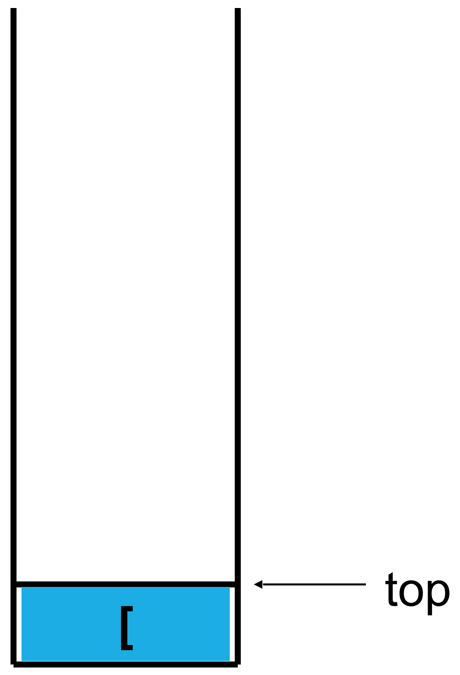
Stack in Action ….



Pop( );

[A + { B - C + (D + E) }]

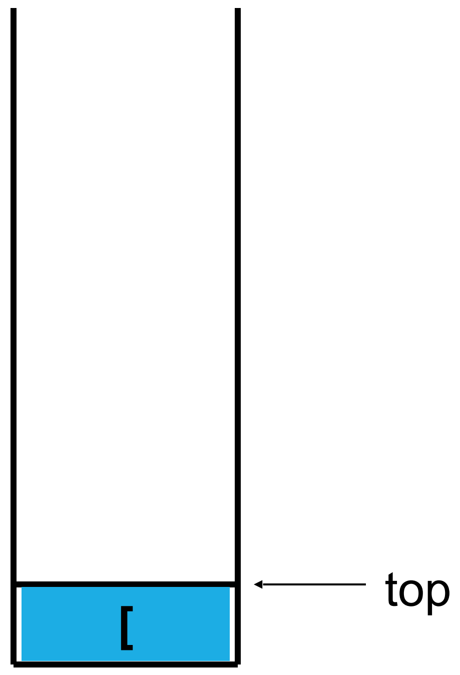
Stack in Action ….



[A + { B - C + (D + E) }]

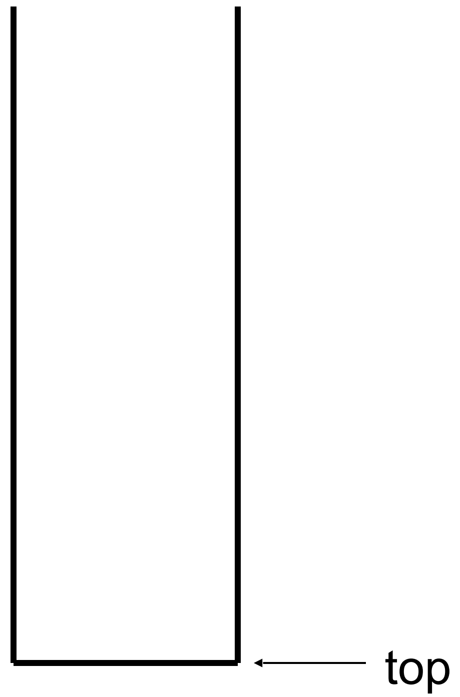
Stack in Action ….

Pop( );



[A + { B - C + (D + E) }]

Stack in Action ….



[A + { B - C + (D + E) }]

Result = A valid expression

INFIX, PREFIX AND POSTFIX NOTATIONS

INFIX, POSTFIX AND PREFIX NOTATIONS

The usual way of expressing the sum of two numbers A and B is :

A+B

The operator '+' is placed between the two operands A and B

This is called the "***Infix Notation***"

Consider a bit more complex example:

(13 – 5) / (3 + 1)

When the parentheses are removed the situation becomes ambiguous

13 – 5 / 3 + 1

is it (13 – 5) / (3 + 1)

or 13 – (5 / 3) + 1

To cater for such ambiguity, you must have operator precedence rules to follow.

INFIX, POSTFIX AND PREFIX NOTATIONS

In the absence of parentheses

13 – 5 / 3 + 1

Will be evaluated as 13 – (5 / 3) + 1

Operator precedence is **by-passed** with the help of parentheses as in (13 – 5) / (3 + 1)

The infix notation is therefore cumbersome due to

- Operator Precedence rules and
- Evaluation of Parentheses

POSTFIX NOTATION

It is a notation for writing arithmetic expressions in which operands appear before the operator

E.g. A + B is written as A B + in postfix notation

There are no precedence rules to be learnt in it.

Parentheses are never needed

Due to its simplicity, some calculators use postfix notation

This is also called the “Reverse Polish Notation or RPN”

POSTFIX NOTATION – SOME EXAMPLES

Infix Expressions	Corresponding Postfix
5 + 3 + 4 + 1	5 3 + 4 + 1 +
(5 + 3) * 10	5 3 + 10 *
(5 + 3) * (10 – 4)	5 3 + 10 4 - *
5 * 3 / (7 – 8)	5 3 * 7 8 - /
(b * b – 4 * a * c) / (2 * a)	b b * 4 a * c * - 2 a * /

CONVERSION FROM INFIX TO POSTFIX NOTATION

We have to accommodate the presence of operator precedence rules and Parentheses while converting from infix to postfix

Data objects required for the conversion are

- An operator / parentheses stack
- A Postfix expression string to store the resultant
- An infix expression string read one item at a time

CONVERSION FROM INFIX TO POSTFIX

<u>The Algorithm</u>

- What are possible items in an input Infix expression
- Read an item from input infix expression
- If item is an operand append it to postfix string
- If item is “(“ push it on the stack
- If the item is an operator
  - If the operator has higher precedence than the one already on top of the stack then push it onto the operator stack
  - If the operator has lower precedence than the one already on top of the stack then
    - pop the operator on top of the operator stack and append it to postfix string, and
    - push lower precedence operator onto the stack
- If item is “)” pop all operators from top of the stack one-by-one, until a “(“ is encountered on stack and removed
- If end of infix string pop the stack one-by-one and append to postfix string

Converting Infix Expressions to Equivalent Postfix Expressions

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	**postfixExp** until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to **postfixExp**

A trace of the algorithm that converts the infix expression ***a* - (*b* + *c* * *d*)/*e*** to postfix form

EVALUATION OF POSTFIX EXPRESSION

After an infix expression is converted to postfix, its evaluation is a simple affair

Stack comes in handy, AGAIN

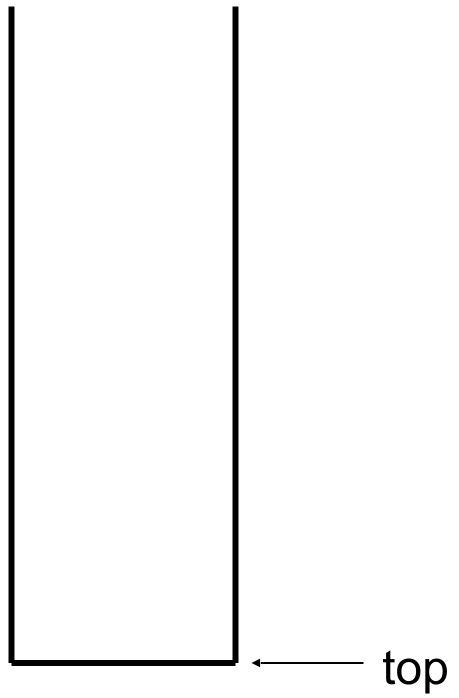
The Algorithm

- Read the postfix expression one item at-a-time
- If item is an operand push it on to the stack
- If item is an operator pop the top two operands from stack and apply the operator
- Push the result back on top of the stack, which will become an operand for next operation
- Final result will be the only item left on top of the stack

Stack in Action ….

Postfix Expression

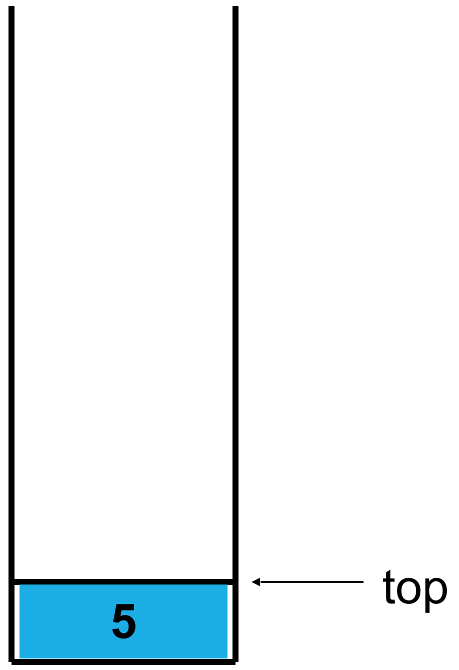
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

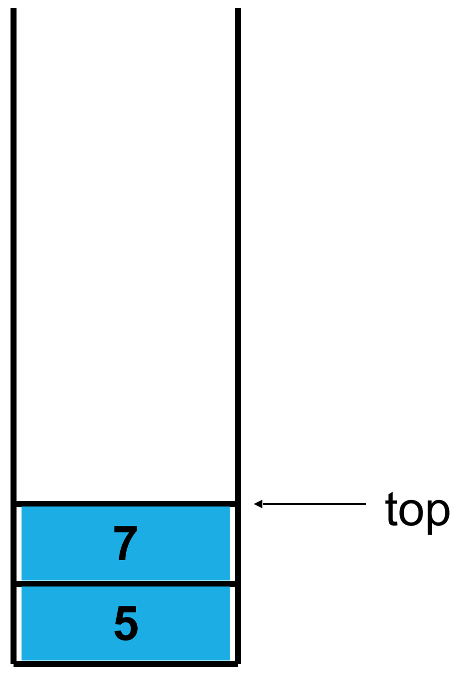
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

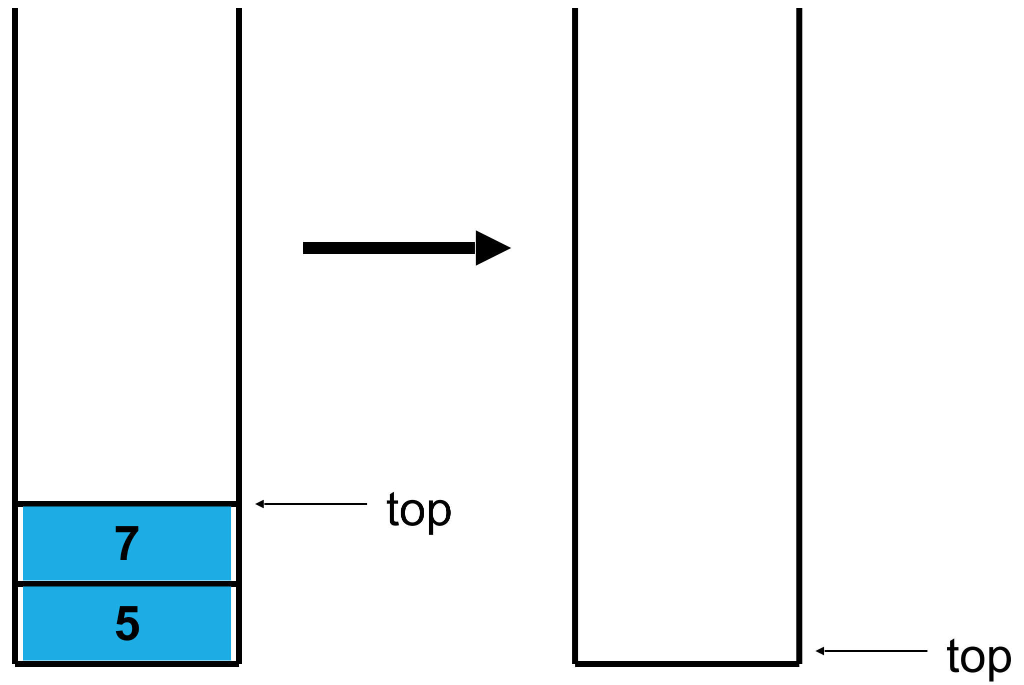
5 7 + 6 2 - *



Stack in Action ….

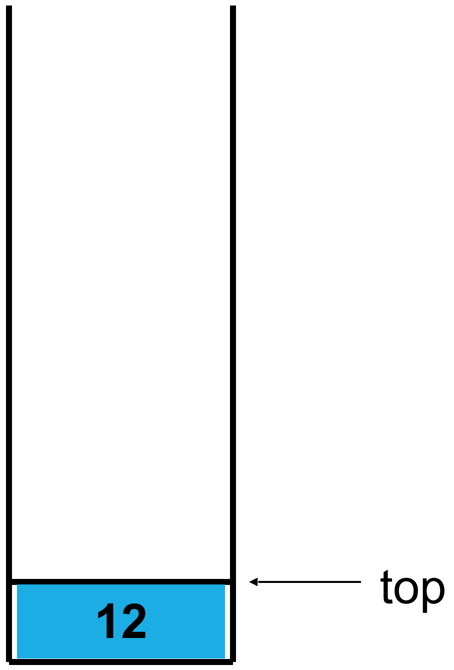
Postfix Expression

5 7 + 6 2 - *



Result = Pop( ) "+" Pop( ) Push (Result)

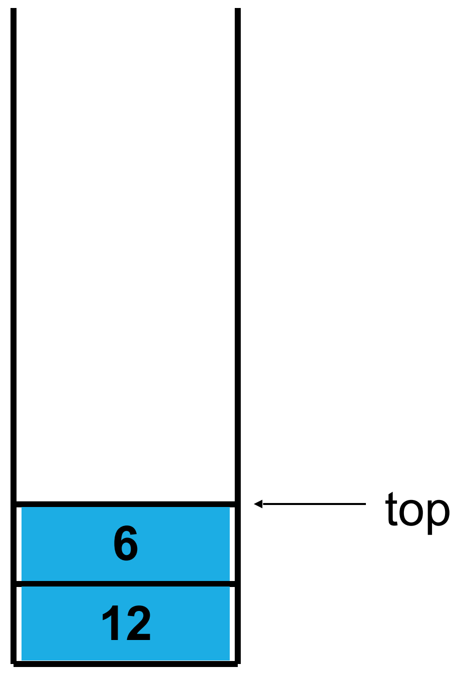
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

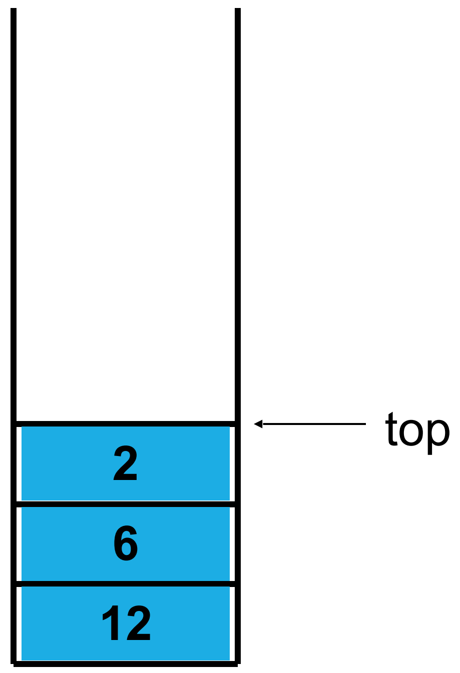
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

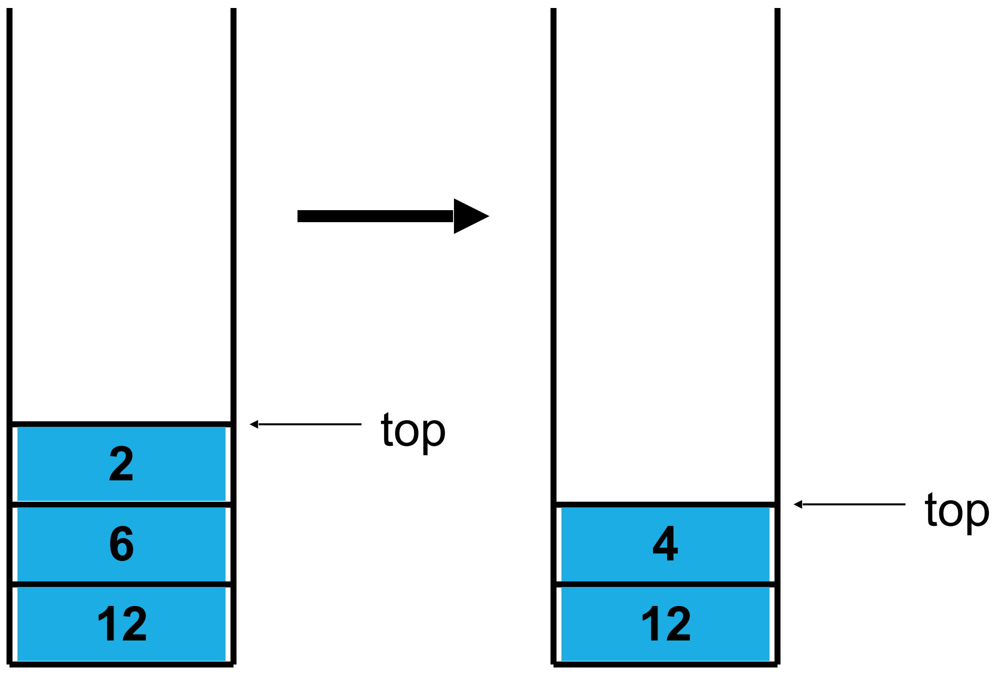
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

Stack in Action ….



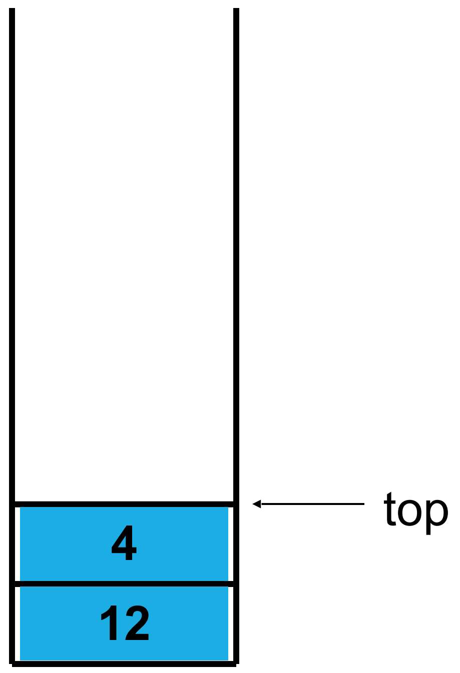
Postfix Expression

5 7 + 6 2 - *

Result = Pop( ) “-” Pop( )

Push (Result)

Stack in Action ….



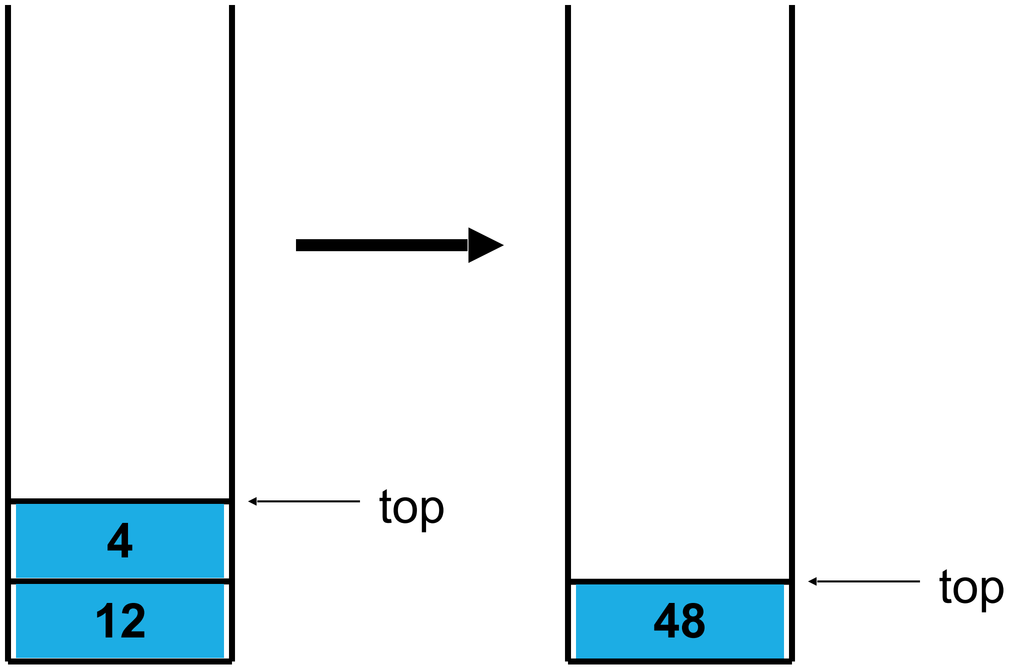
Postfix Expression

5 7 + 6 2 - *

Stack in Action ….

Postfix Expression

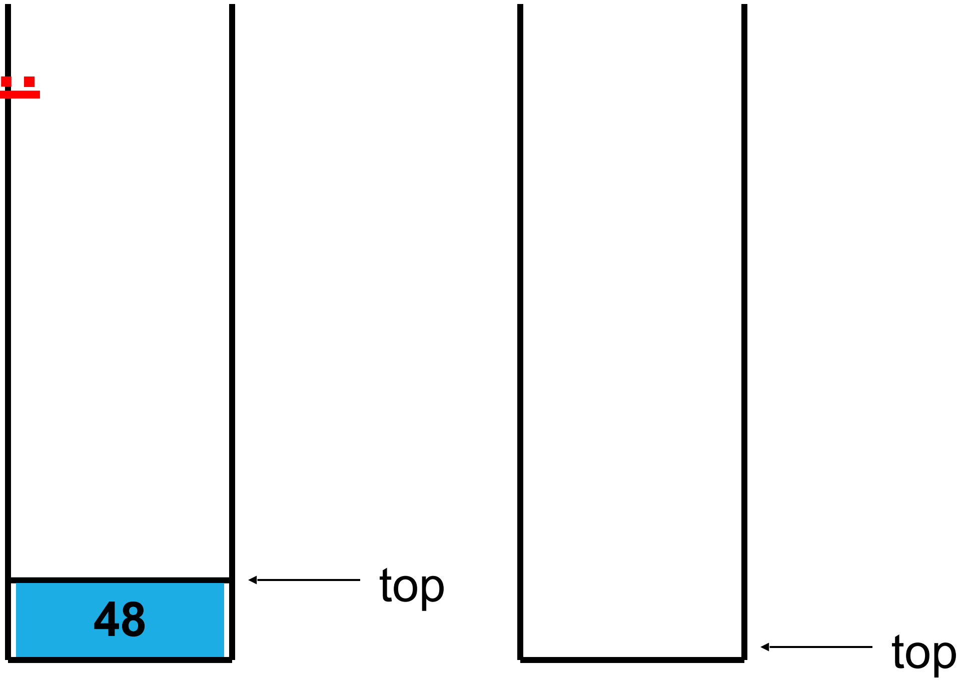
5 7 + 6 2 - *



Result = Pop( ) “ * ” Pop( )

Push (Result)

Stack in Action ….



Postfix Expression

Result = Pop( )

Result = 48

EXERCISE

Show a trace of algorithm that converts the infix expression

- (X + Y) * (P – Q / L)
- L – M / ((N * O) ^ P)