Data Structure

Problem Solving with Stack
and its Applications

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

Problem Solving (Cont.…)

- To solve this problem, think of each left parenthesis as opening a scope, right parenthesis as closing a scope
- Nesting depth at a particular point in an expression is the number of scopes that have been opened but not yet closed
- Let "parenthesis count" be a variable containing number of left parenthesis minus number of right parenthesis, in scanning the expression from left to right

Problem Solving (Cont.…)

- For an expression to be of a correct form following conditions apply
  - Parenthesis count at the end of an expression must be 0
  - Parenthesis count should always be non-negative while scanning an expression
- Example :
  - Expr: 7 – (A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 1 1 1 0 0 1 2 2 2 2 1 1 1 0
  - Expr: 7 – ((A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 2 2 2 2 1 1 23 3 3 3 2 2 21

Problem Solving (Cont.…)

- Evaluating the correctness of simple expressions like this one can easily be done with the help of a few variables like “Parenthesis count”
- Things start getting difficult to handle by your program when the requirements get complicated e.g.
- Let us change the problem by introducing three different types of scope de-limiters i.e. (parenthesis), {braces} and [brackets].
- In such a situation we must keep track of not only the number of scope delimiters but also their types
- When a scope ender is encountered while scanning an expression, we must know the scope delimiter type with which the scope was opened
- We can use a stack ADT to solve this problem

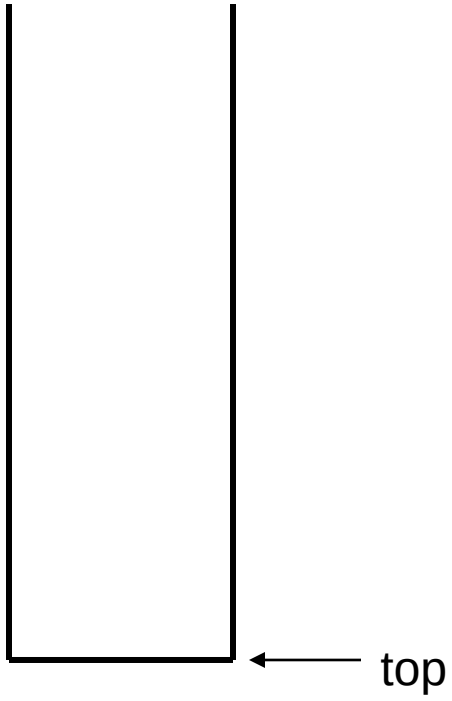
Problem Solving with Stack

- A stack ADT can be used to keep track of the scope delimiters encountered while scanning the expression
- Whenever a scope "opener" is encountered, it can be "pushed" onto a stack
- Whenever a scope "ender" is encountered, the stack is examined:
  - If the stack is "empty", there is no matching scope "opener" and the expression is invalid.
  - If the stack is not empty, we pop the stack and check if the "popped" item corresponds to the scope ender
  - If match occurs, we continue scanning the expression
- When end of the expression string is reached, the stack must be empty, otherwise one or more opened scopes have not been closed and the expression is invalid

Why the need for a Stack

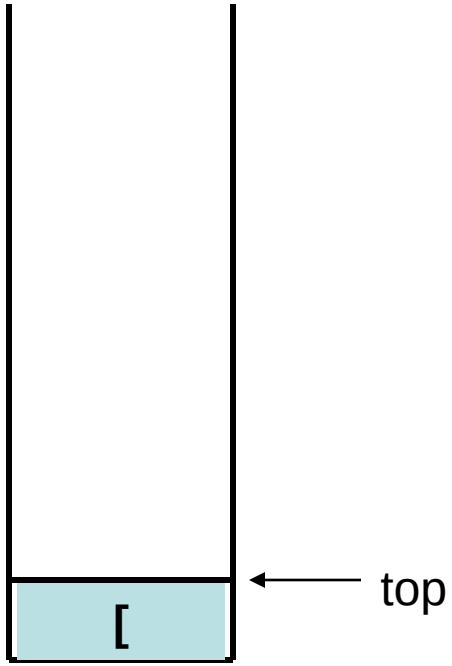
- Last scope to be opened must be the first one to be closed.
- This scenario is simulated by a stack in which the last element arriving must be the first one to leave
- Each item on the stack represents a scope that has been opened but has yet not been closed
- Pushing an item on to the stack corresponds to opening a scope
- Popping an item from the stack corresponds to closing a scope, leaving one less scope open

Stack in Action ….



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

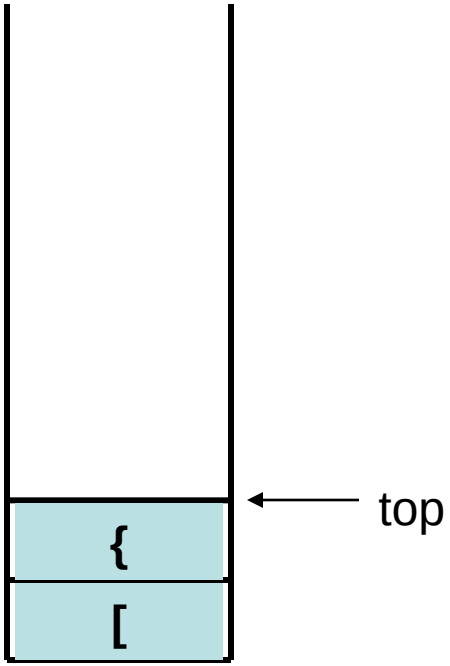
Stack in Action ….



Push('[');

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

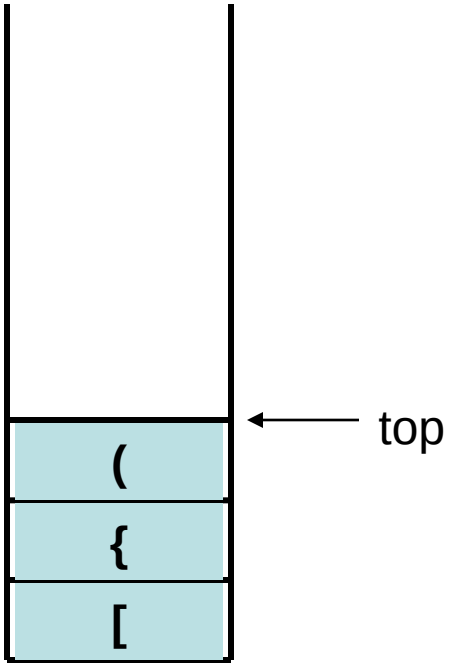
Stack in Action ….



Push(‘{‘);

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

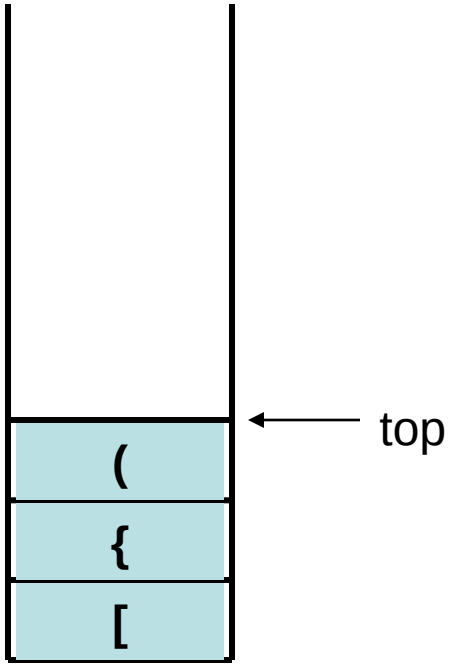
Stack in Action ….



Push(‘(‘);

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

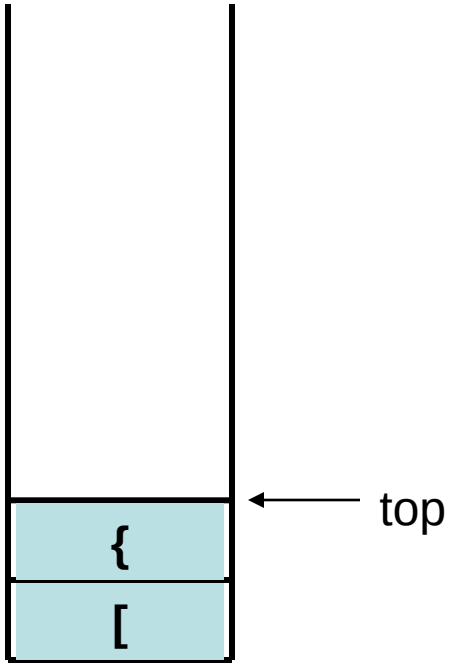
Stack in Action ….



Pop( );

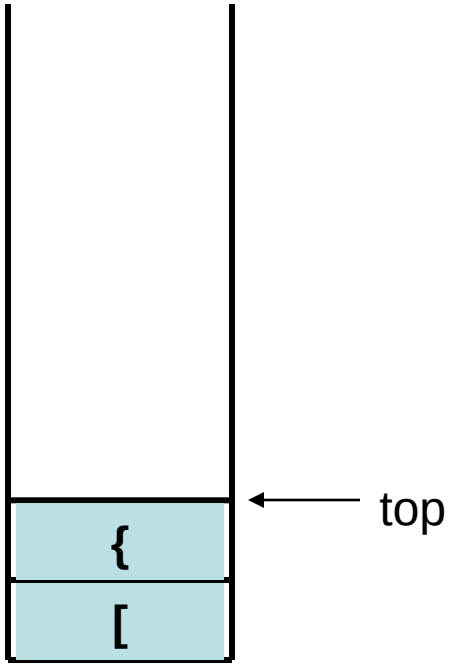
[A + { B - C + (D + E) }]

Stack in Action ….



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

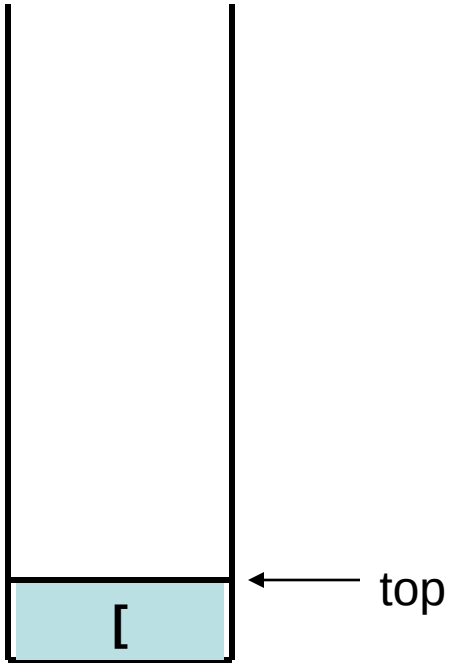
Stack in Action ….



Pop( );

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

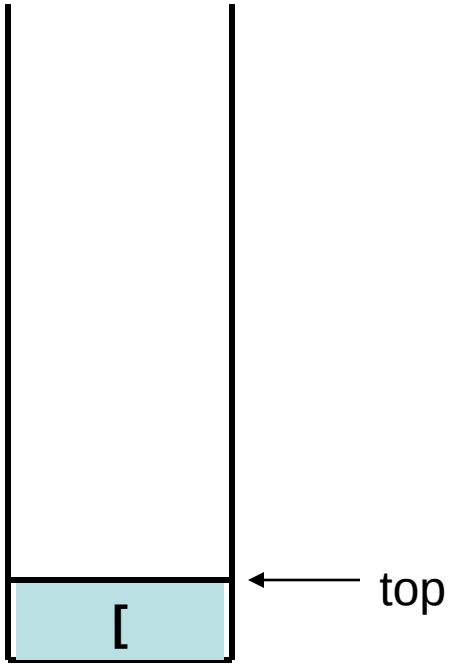
Stack in Action ….



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

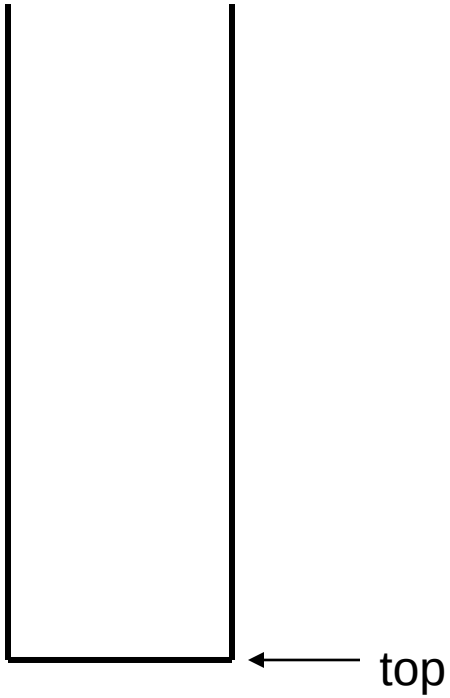
Stack in Action ….

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 (as in C)

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

- The Algorithm
  - 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

Try it yourself

- Show a trace of algorithm that converts the infix expression
  - (X + Y) * (P – Q / L)
  - L – M / (N * O ^ P)

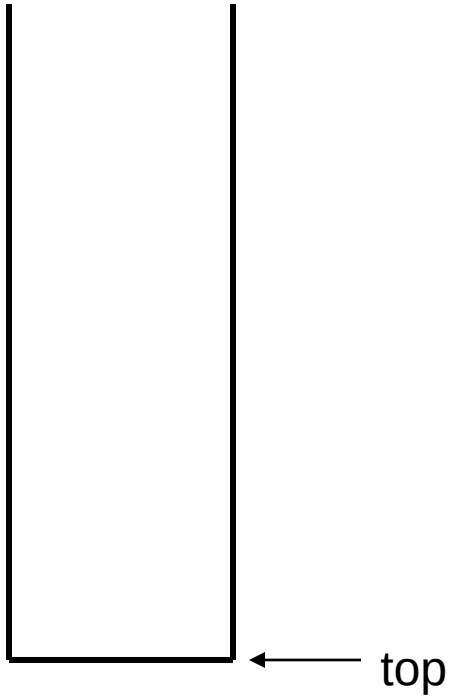
Evaluation of Postfix Expression

- 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

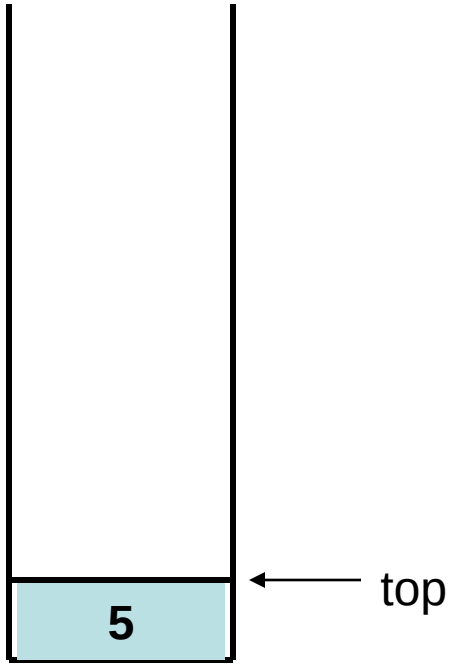
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

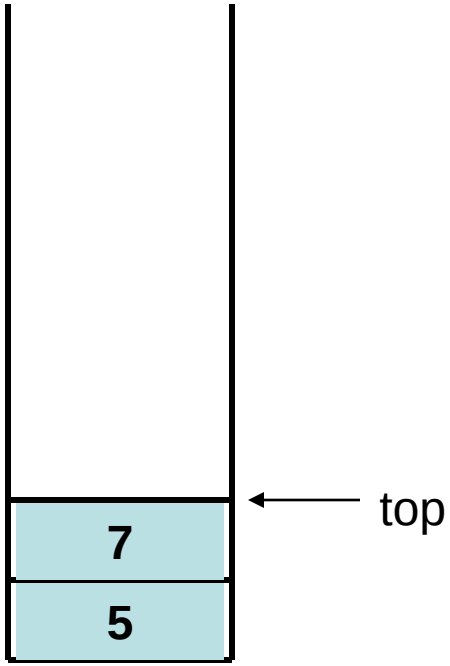
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

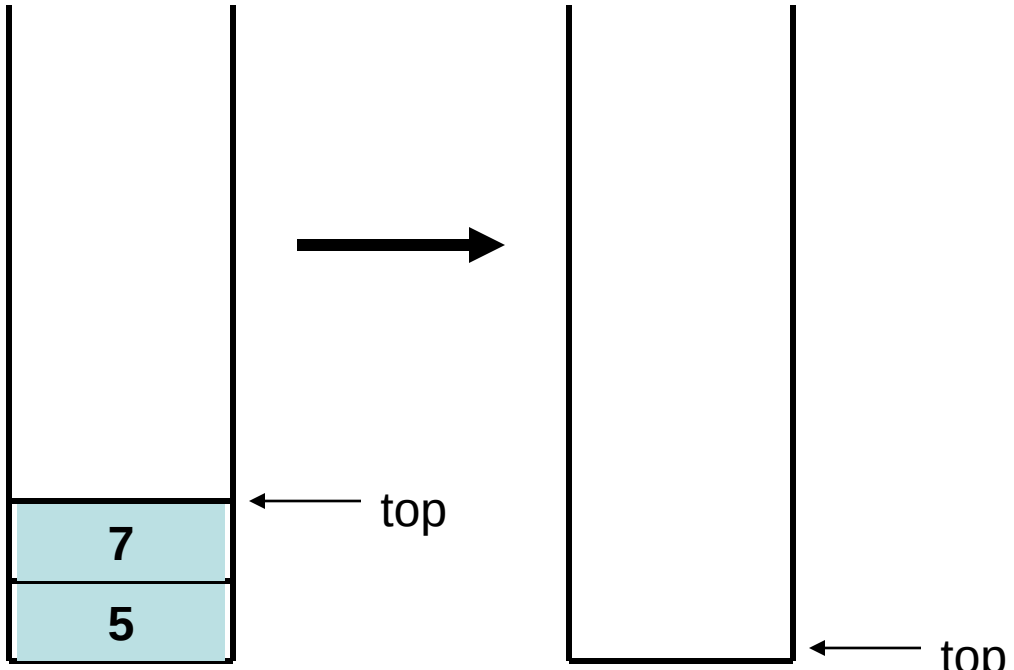
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

5 7 + 6 2 - *



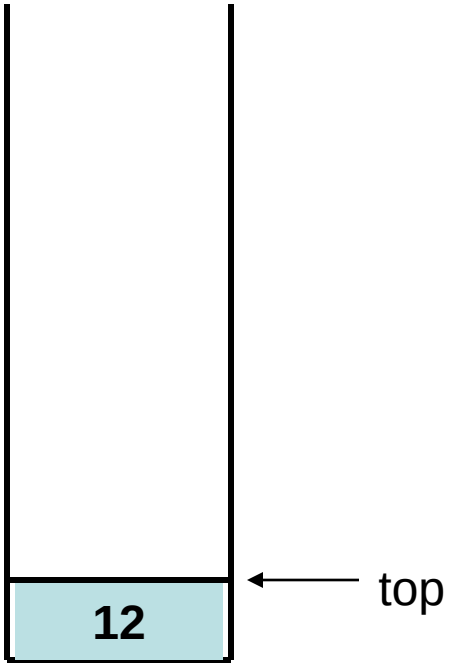
Result = Pop( ) “+” Pop( )

Push (Result)

Stack in Action ….

Postfix Expression

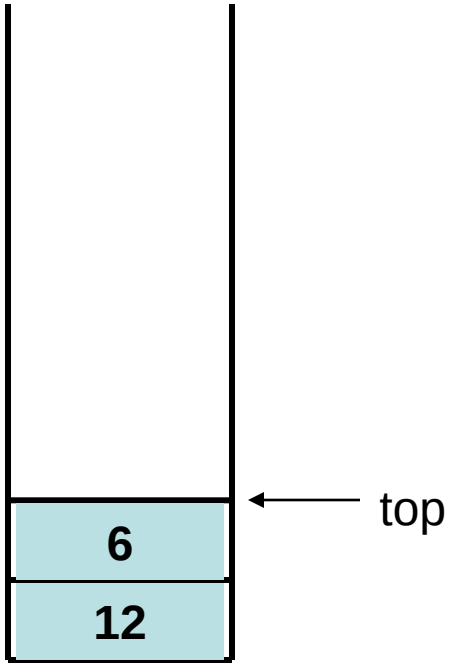
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

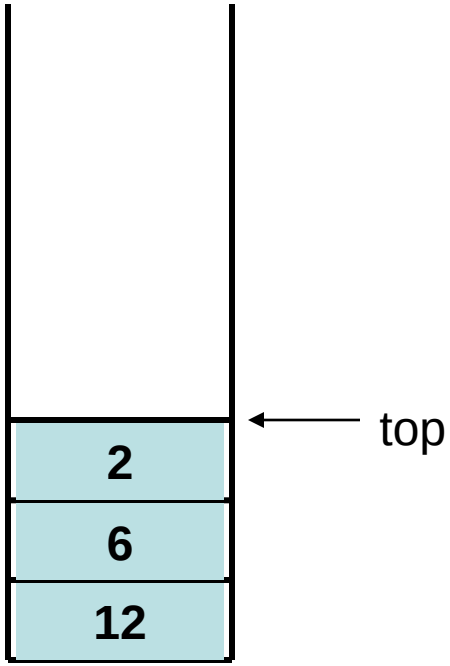
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

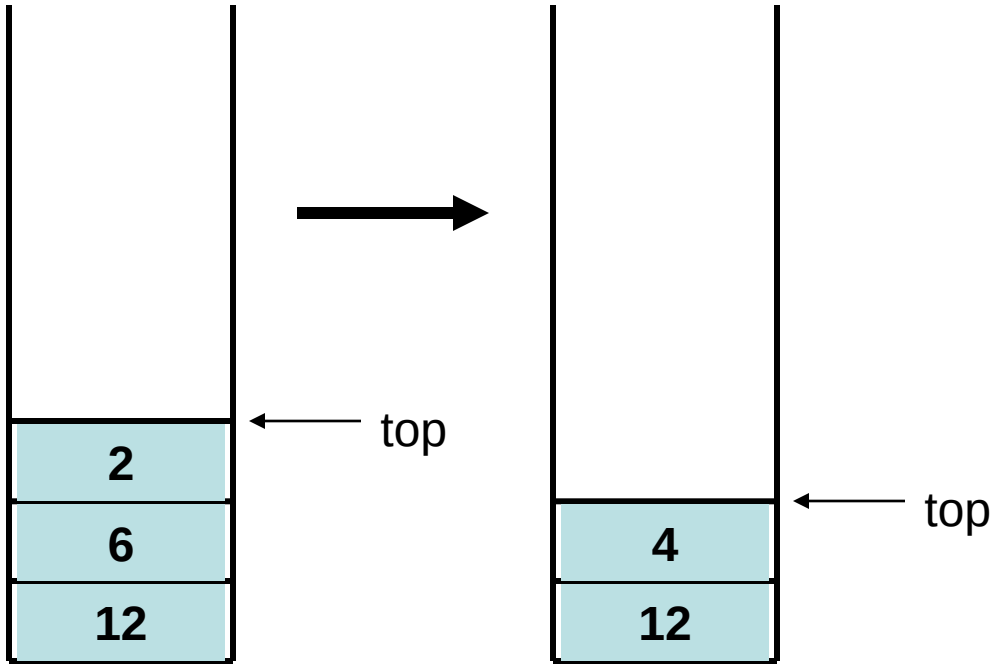
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

5 7 + 6 2 - *

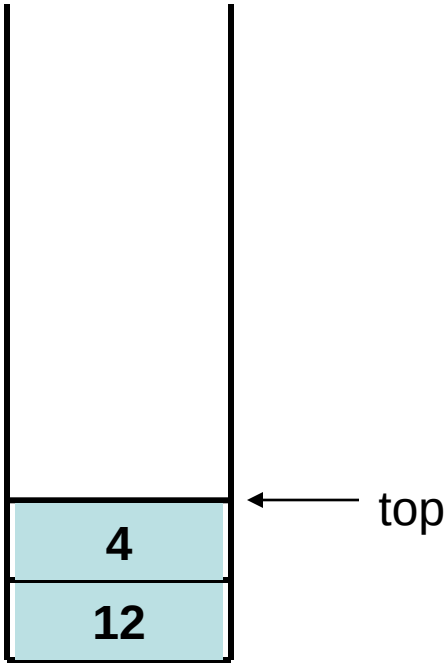


Result = Pop( ) “-” Pop( ) Push (Result)

Stack in Action ….

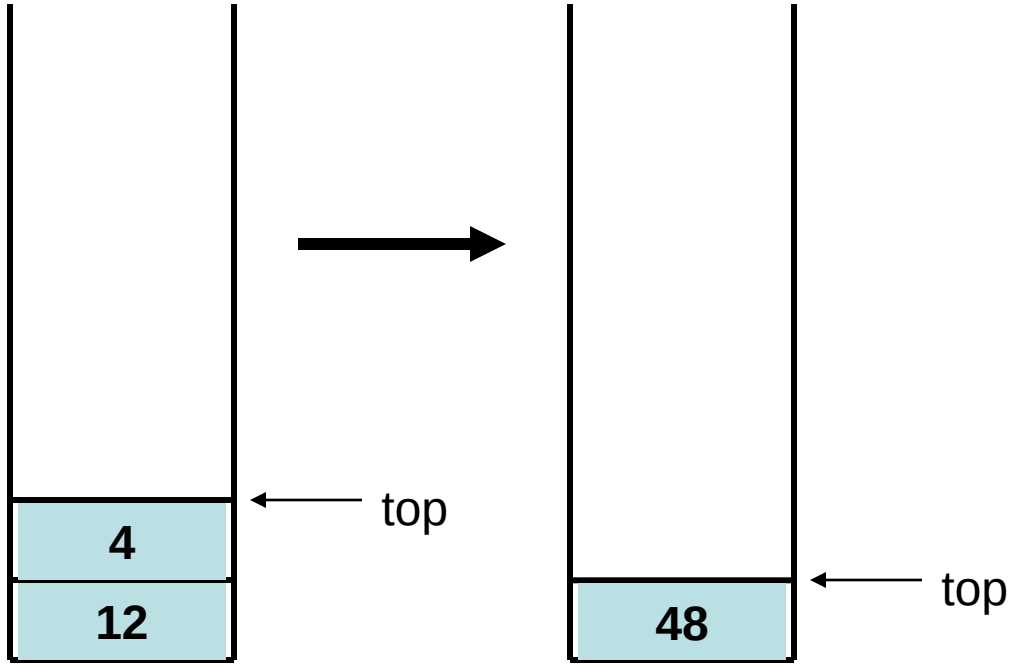
Postfix Expression

5 7 + 6 2 - *

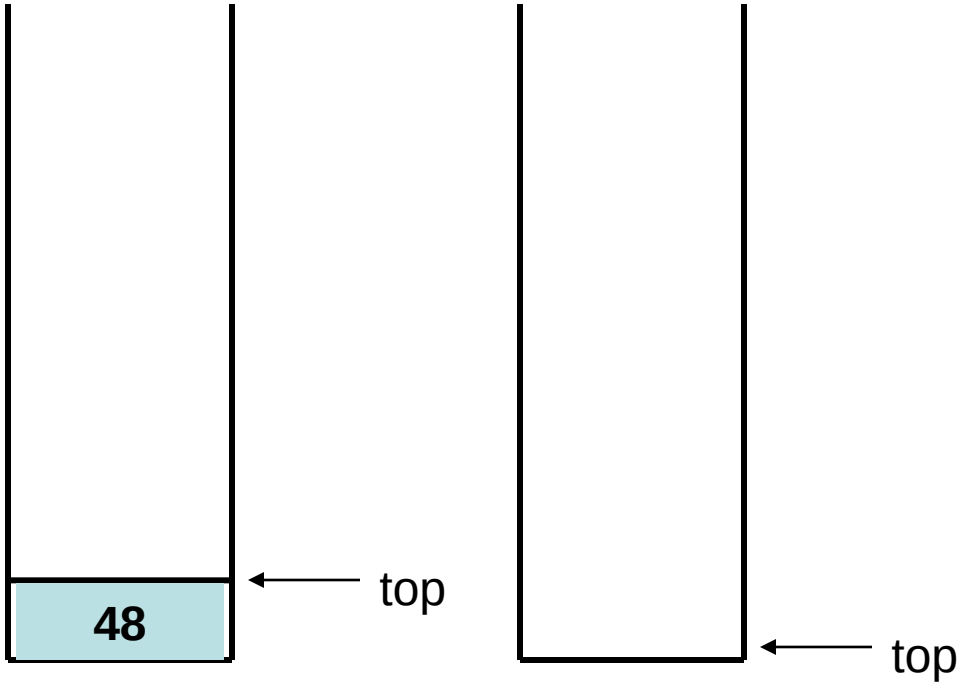


Postfix Expression

5 7 + 6 2 - *



Result = Pop( ) “ * ” Pop( ) Push (Result)



Postfix Expression

5 7 + 6 2 - *

Result = Pop( )

Result = 48

Evaluation of Postfix Expression

- Evaluation of infix Expression is difficult because :
  - Rules governing the precedence of operators are to be catered for
  - Many possibilities for incoming characters
  - To cater for parentheses
  - To cater for error conditions / checks
- Evaluation of postfix expression is very simple to implement because operators appear in precisely the order in which they are to be executed

Motivation for the conversion

- Motivation for this conversion is the need to have the operators in the precise order for execution
- While using paper and pencil to do the conversion we can "foresee" the expression string and the depth of all the scopes (if the expressions are not very long and complicated)
- When a program is required to evaluate an expression, it must be accurate
- At any time during scanning of an expression we cannot be sure that we have reached the inner most scope
- Encountering an operator or parentheses may require frequent "backtracking"
- Rather than backtracking, we use the stack to **"remember"** the operators encountered previously

Assignment # 1

- Write a program that gets an Infix arithmetic expression and converts it into postfix notation
- The program should then evaluate the postfix expression and output the result
- Your program should define the input and output format, enforce the format and handle Exceptions (exceptional conditions).
- Use appropriate comments at every stage of programming