Let's Start with a Problem

X = ((((7 + 8) * (9 - 5)) / 2))

What is the value of X ?

A Fully Parenthesized Infix Expression

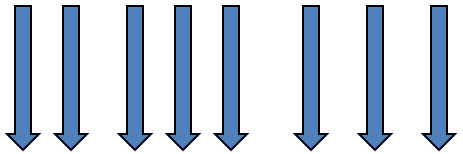
This kind of evaluation is a trivial Compiler task

What is the difficulty?

X = ((((7 + 8) * (9 - 5)) / 2))

The difficulty lies in finding proper matching brackets

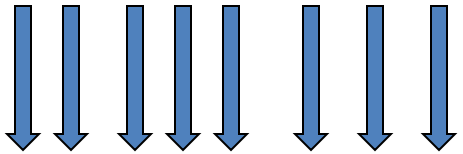
Let’s Give a First Try



X = ((((7 + 8) * (9 - 5)) / 2))

Does Not Work !!!

Let’s Try Differently



X = ((((7 + 8) * (9 - 5)) / 2))

X = (((15 * (9 - 5)) / 2))

Very Cumbersome !!!

Today's Topic

STACK

Stack is one of the most common and simple Data Structures.

Think of it like a stack of plates, where you a can insert or remove a plate only at the top.

Stack

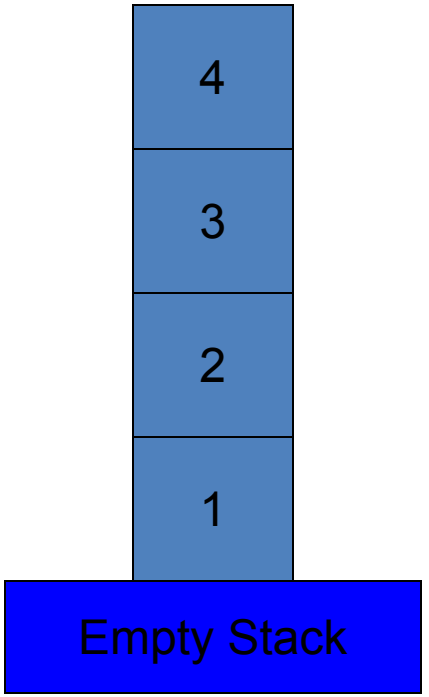
Stack has only two operation

Push: insert data on top

Pop: remove data from top

Also known as LIFO structure

Last In First Out



How does it help our problem?

We will see how Stack helps to solve our problem efficiently and elegantly

We will use two stacks:

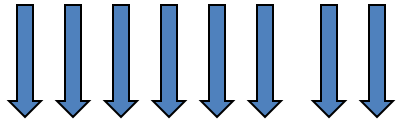
Operator Stack: + - * / ( )

Operand Stack: 0 - 9

Rules:

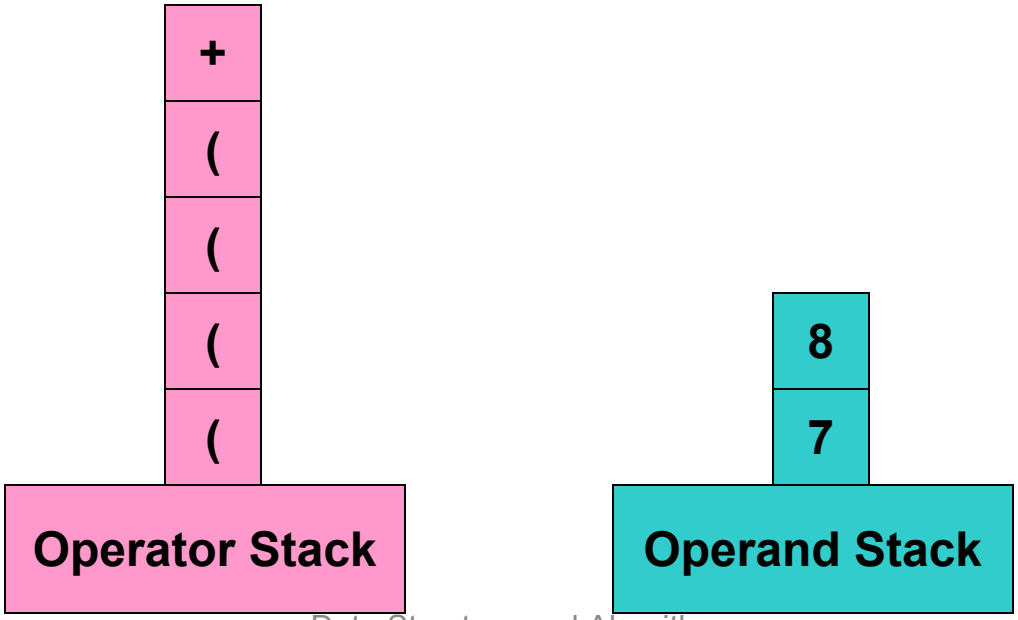
1. Until we find a ‘)’ we push
2. When we find a ‘)‘ we pop

Stack Solution

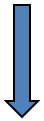


X = ((((7 + 8) * (9 - 5)) / 2))

PUSH



Stack Solution

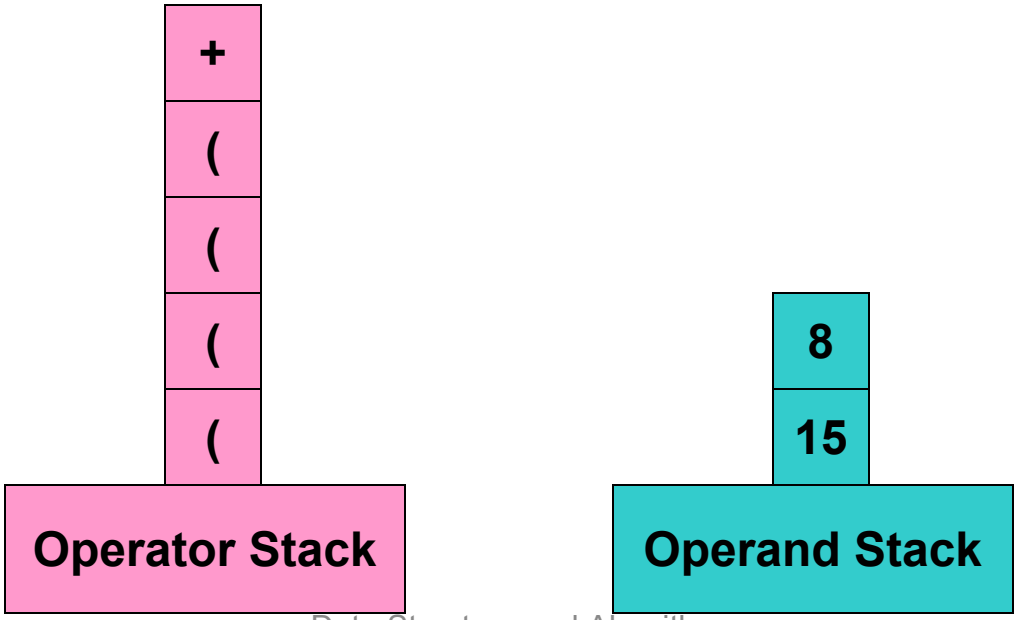


X = ((((7 + 8) * (9 - 5)) / 2))

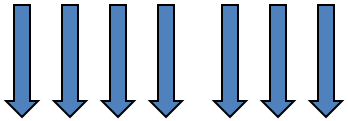
POP

PUSH

7 + 8 = 15

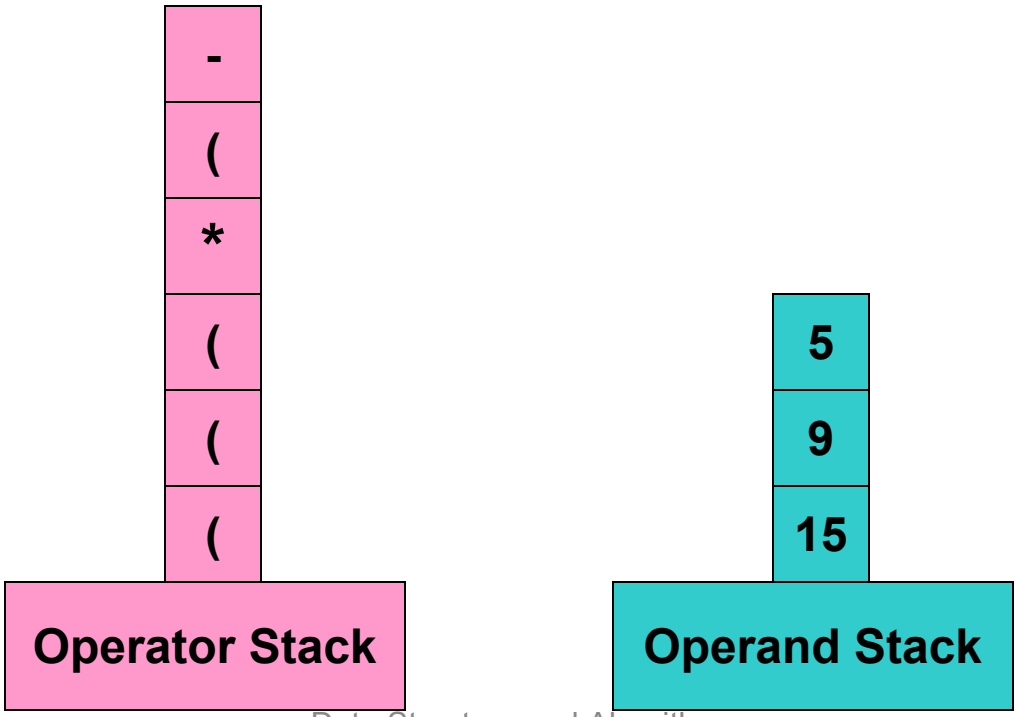


Stack Solution



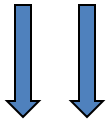
X = ((((7 + 8) * (9 - 5)) / 2))

PUSH



7 + 8 = 15

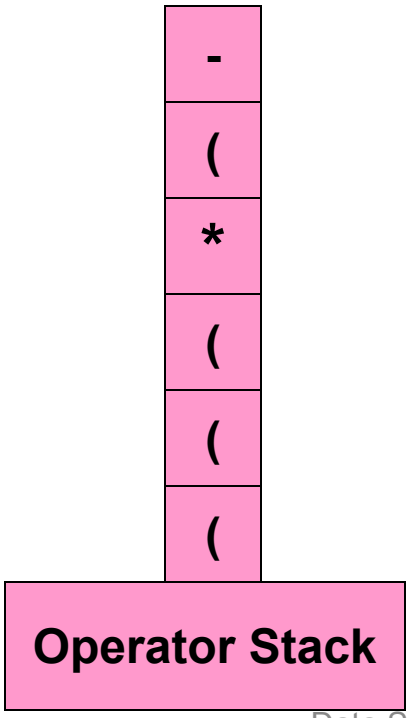
Stack Solution

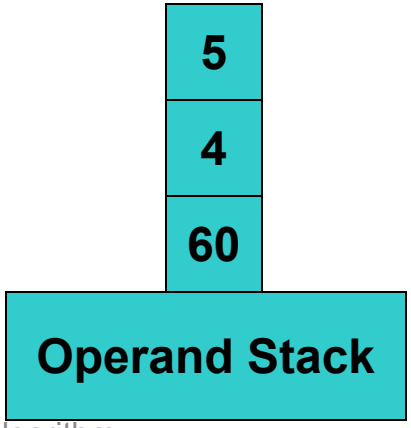


X = ((((7 + 8) * (9 - 5)) / 2))

POP

PUSH



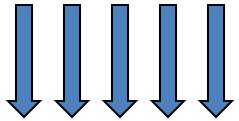


7 + 8 = 15

9 - 5 = 4

15 * 4 = 60

Stack Solution



X = ((((7 + 8) * (9 - 5)) / 2))

POP

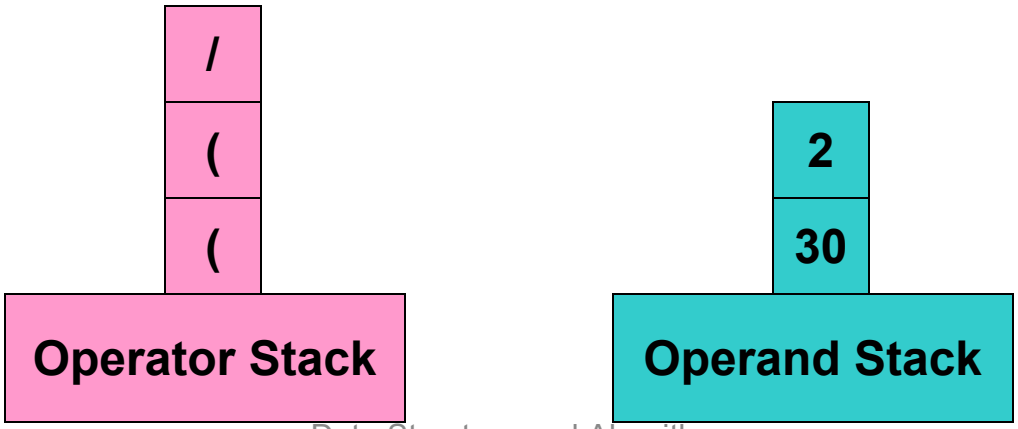
PUSH

7 + 8 = 15

9 - 5 = 4

15 * 4 = 60

60 / 2 = 30



Summary

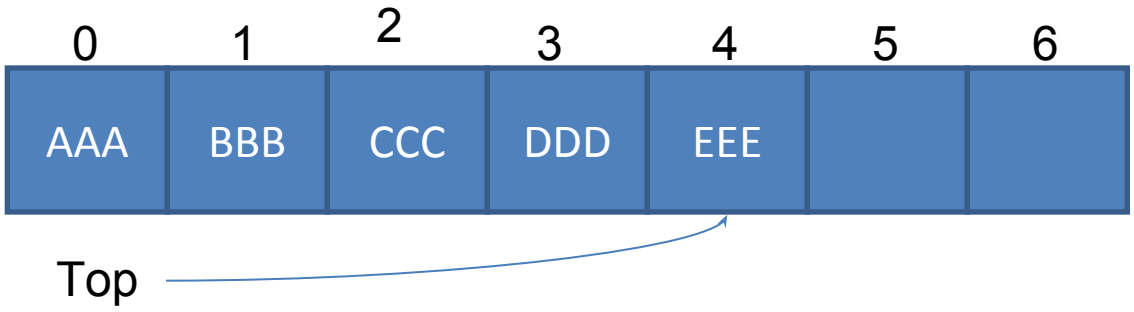
The final solution was efficient in terms of run-time and simple in terms of code complexity.

Stack can be a very handy data structure. When ever faced with a problem, try to fit it in a known data structure. If successful, it will provide us a straight-forward solution.

Stack

- It is a linear data structure consisting of list of items.
- In stack, data elements are added or removed only at one end, called ***the top of the stack***.
- Two basic operations are associated with stack:
  1. "Push" operation is used to insert an element into a stack.
  2. "Pop" operation is used to delete an element from a stack.
- Example:

  Stack: AAA, BBB, CCC, DDD, EEE



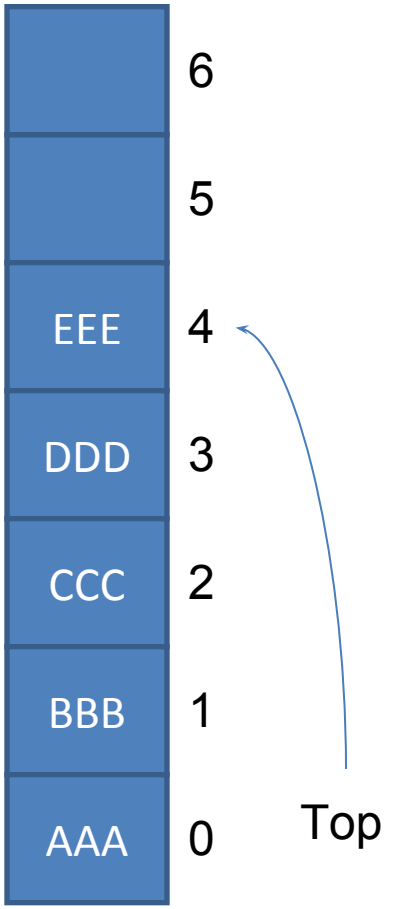


Figure: Diagrams of Stacks

Algorithms for Push and Pop Operations

For Push Operation

Push-Stack(Stack, Top, MaxSTK, Item)

Here, Stack is the place where to store data. Top represents in which location the data is to be inserted. MaxSTK is the maximum size of the stack and finally, Item is the new item to be added.

1. If Top = MaxSTK then Print: Overflow and Return. /*…Stack already filled..*/
2. Set Top := Top +1
3. Set Stack[Top] := Item
4. Return.

For Pop Operation

Pop-Stack(Stack, Top, Item)

Here, Stack is the place where data are stored. Top represents from which location the data is to be removed. Top element is assigned to Item.

1. If Top = -1 then Print: Underflow and Return. /*…Stack already Empty..*/
2. Set Item := Stack[Top]
3. Set Top := Top - 1
4. Return.

Application of Stack

- **Arithmetic Expression Evaluation**

Calculators use a stack structure to hold values for calculation.

- **Syntax Parsing**

Many compilers use a stack for parsing the syntax of expressions before translating into low level code.

- **Solving Search Problem**

Solving a search problems, regardless of whether the approach is exhaustive or optimal, needs stack space. Example of exhaustive search methods is backtracking. Example of optimal search exploring methods is branch and bound. All of these algorithms use stacks to remember the search nodes that have been noticed but not explored yet.

- **Runtime Memory Management**

Almost all computer runtime memory environments use a special stack (the “call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. They follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls.

Arithmetic Expression

- Arithmetic expression consists of operands and operators.
- Three types of arithmetic expression:

1. Infix Expression
   - Operators are placed between its two operands.
   - Example: 2 + 4, a – b / c

2. Prefix Expression(Polish Notation)
   - Operators are placed before its two operands.
   - Example: + 2 4, - a / b c

3. Postfix Expression (Reverse Polish Notation or RPN)
   - Operators are placed after its two operands.
   - Example: 2 4 +, a b c / -

Operator Precedence

- The order in which expressions are evaluated is determined by operator precedence.
- If an expression contains two or more operators with the same precedence level, the operator to the left is processed first.
- When a lower order precedence operation must be performed first, it should be surrounded by parentheses.

Operators	Precedence
( )	Highest
Unary -	Next Highest
^	Next Highest
* , / , %	Next Highest
+ , -	Lowest

Figure: Arithmetic Operator Precedence Table

Conversion of Infix Expression into Its Equivalent Prefix Expression

1. F = A + B * C /*..Infix Expression…..*/

   = A + [* B C]

   = [+ A * B C]

   = + A * B C /*..Prefix Expression…..*/

2. F = (5 + 7) * 8 - 10 /*..Infix Expression…..*/

   = [+ 5 7] * 8 - 10

   = [* + 5 7 8] - 10

   = [- * + 5 7 8 10]

   = - * + 5 7 8 10 /*..Prefix Expression…..*/

3. F = (A + B ^ D) / (E - F)+G /*..Infix Expression…..*/

   = ?

Conversion of Infix Expression into Its Equivalent Postfix Expression

1. F = A + B * C /*..Infix Expression…..*/
   = A + [B C *]
   = [A B C * +]
   = A B C * + /*..Postfix Expression…..*/

2. F = (5 + 7) * 8 - 10 /*..Infix Expression…..*/
   = [5 7 +] * 8 - 10
   = [5 7 + 8 *] - 10
   = [5 7 + 8 * 10 -]
   = 5 7 + 8 * 10 - /*..Postfix Expression…..*/

3. F = (A + B ^ D) / (E - F)+G /*..Infix Expression…..*/
   = ?

Conversion of Prefix Expression into Its Equivalent Infix Expression

1. F = + A * B C /*...Pretfix Expression…..*/

   = + A [B * C]

   = [A + [B * C]]

   = A + B * C /*..Infix Expression…..*/

2. F = - * + 5 7 8 10 /*..Prefix Expression…..*/

   = - * [5 + 7] 8 10

   = - [(5 + 7) * 8] 10

   = [(5 + 7) * 8] - 10]

   = (5 + 7) * 8 - 10 /*..Infix Expression…..*/

Conversion of Postfix Expression into Its Equivalent Infix Expression

1. F = A B C * + /*..Postfix Expression…..*/

   = A [B * C] +

   = [A + [B * C]]

   = A + B * C /*..Infix Expression…..*/

2. F = 5 7 + 8 * 10 - /*..Postfix Expression…..*/

   = [5 + 7] 8 * 10 -

   = [(5 + 7) * 8] 10 -

   = [[(5 + 7) * 8] - 10]

   = (5 + 7) * 8 - 10 /*..Infix Expression…..*/