The Stack

Shikha Mehrotra

• Polish Notations

- Prefix
- Postfix
- Infix
- Infix to postfix Conversion

•About Polish Notations

- Postfix, or Reverse Polish Notation (RPN) is an alternative to the way we usually write arithmetic expressions (which is called infix, or algebraic notation
  - "Postfix" refers to the fact that the operator is at the end
  - "Infix" refers to the fact that the operator is in between
  - For example, 2 2 +postfix is the same as2 + 2 Infix
  - There is also a seldom-used prefix notation, similar to postfix
- Advantages of postfix:–
  - You don't need rules of precedence
  - You don't need rules for right and left associativity
  - you don't need parentheses to override the above rules
- Advantages of infix:–
  - You grew up with it
  - It's easier to see visually what is done first

•How to evaluate postfix notation

- Going from left to right, if you see an operator, apply it to the previous two operands (numbers)
- Example:

- Equivalent infix: 2 + 10 * 4 / 5 – (9 – 3)

•Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, *A+B;* here, plus operator is placed between the two operands A and B.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

•Postfix Notation

- Postfix notation was given by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in infix notation, the same expression can be written as *AB+* in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.

•Postfix Notation

- The expression (A + B) * C is written as:

  AB+C* in the postfix notation.

- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.

- No parenthese

- For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

•Prefix Notation

- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression (A + B) * C is written as:
  *+ABC in the prefix notation

- **Arithmetic and Logical Expressions**
- repeatedly scan through the expression !
- take parentheses and priorities of operators into account
- a + b + c * d - e / g
- a + b + (c * d) - (e / g)
- a + ((b + c) * d - e) / g
- a + b <= c && a + b <= d
- (a + b <= c) || (a + b <= d)

•The Polish Notations

- Q : How can a compiler accept an expression and produce correct code ?
- A : Transforming the expression into a form called Polish notation

Infix form	Prefix form	Postfix form
a * b	* a b	a b *
a + b * c	+ a * b c	a b c * +
(a + b) * c	* + a b c	a b + c *

Reverse Polish notation

•Expression Evaluations : Stacks

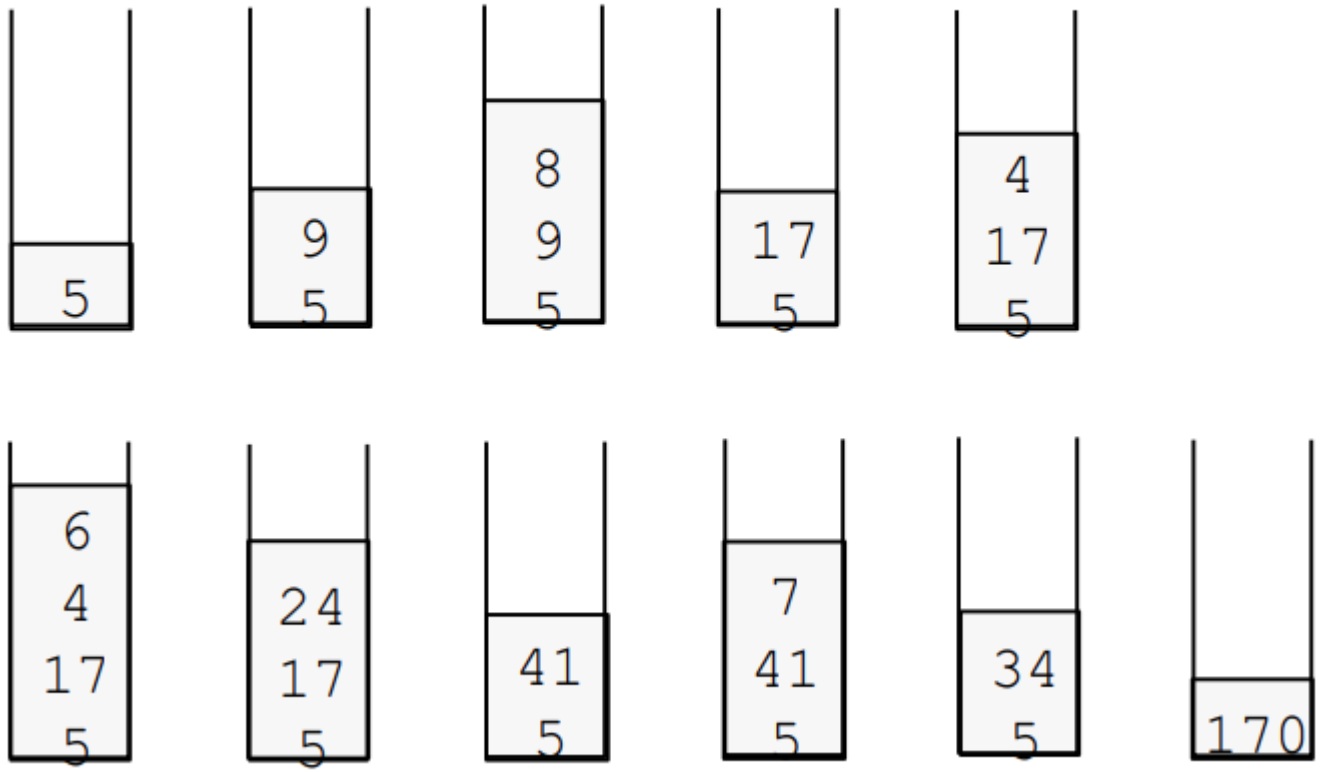
```
5 * ( ( ( 9 + 8 ) + ( 4 * 6 ) ) - 7 )
Postfix form : 5 9 8 + 4 6 * + 7 - *

Push( 5 )
Push( 9 )
Push( 8 )
Push( Pop() + Pop() )
Push( 4 )
Push( 6 )
Push( Pop() * Pop() )
Push( Pop() + Pop() )
Push( 7 )
Push( Pop() - Pop() )
Push( Pop() * Pop() )
```

•Expression Evaluations : Stacks

5 * (((9 + 8) + (4 * 6)) - 7)

Postfix form : ***5 9 8 + 4 6 * + 7 - ****

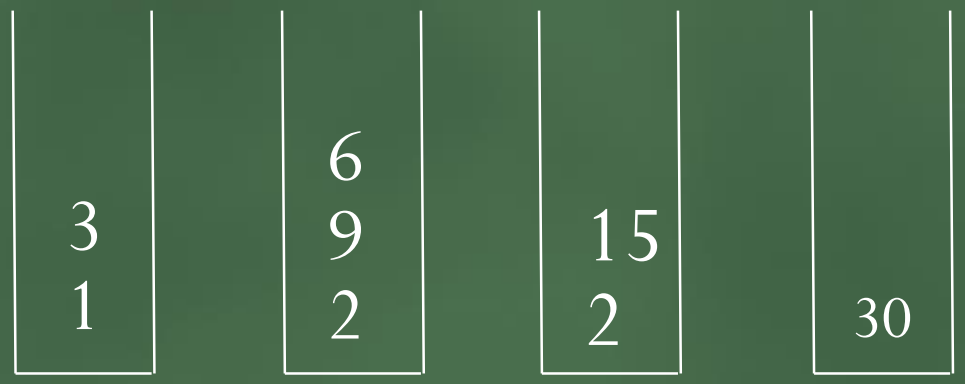


```
EvaluatePostfix(exp)
{   Create a stack S
    for i ← 0 to length(exp)-1
    {   if(exp[i] is operand)
            Push(exp[i])
        else if(exp[i] is operator)
        {   op1 ← Pop()
            op2 ← Pop()
            res ← Perform(exp[i], op1, op2)
        }   Push(res)
```

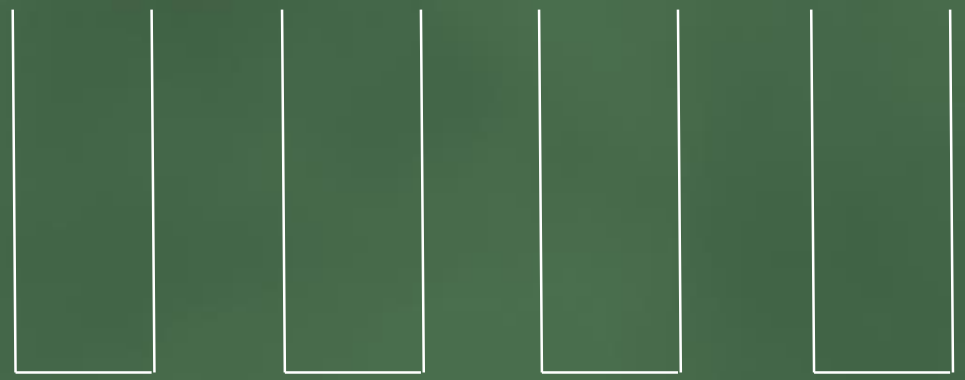
•Evaluation of Prefix expression

- **Algorithm –**
  - **Step 1:** Start from the last element of the expression.
  - **Step 2:** check the current element.
    - **Step 2.1:** if it is an operand, push it to the stack.
    - **Step 2.2:** If it is an operator, pop two operands from the stack. Perform the operation and push the elements back to the stack.
  - **Step 3:** Do this till all the elements of the expression are traversed and return the top of stack which will be the result of the operation.

*+69-31



+ * 5 8 – 7 6



Infix Form → Postfix Form

A + B * C Infix

A + (B * C)

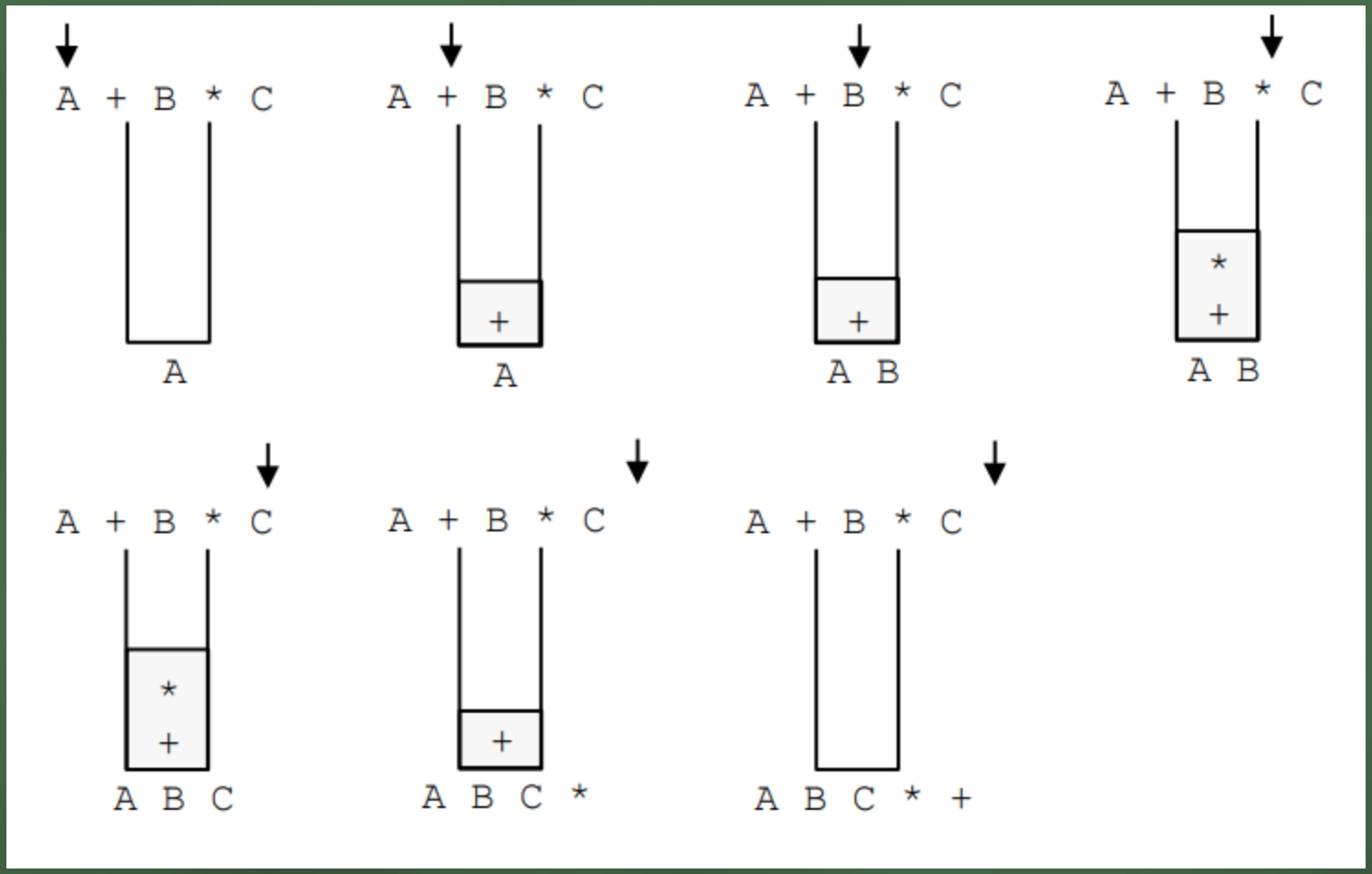
A + (B C *)

A (B C *) +

A B C * + Postfix

- Order of Operation
  1. Parenthesis {} () []
  2. Exponents (right to left)
  3. Multiplication and Division (left to right)
  4. Addition and subtraction (left to right)

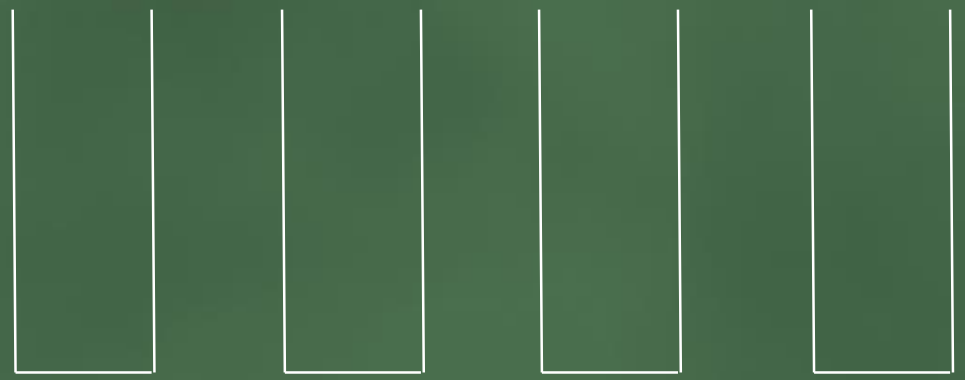
Infix Form → Postfix Form



Infix Form → Postfix Form

A + B * C - D * E

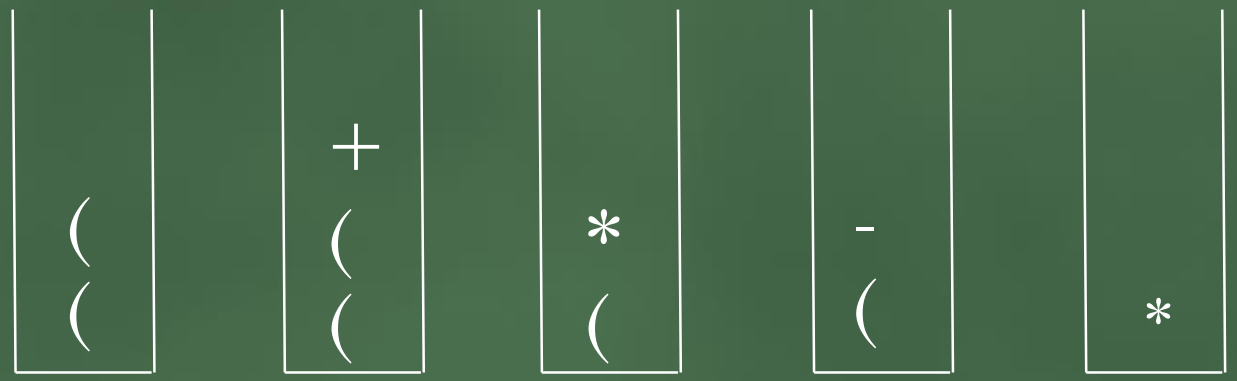
Priority (top()) < priority (waiting Operator)



A B C * + D E * -

Infix Form → Postfix Form

((A + B) * C - D) * E



A B + C * D – E *

```
InfixToPostfix(exp)
{
   Create a stack S
   res ← empty string
  for i ← 0 to length(exp)-1
  {  if exp[i] is operand
           res ← res + exp[i]
    else if exp[i] is operator
    {
      while(!S.empty() && HasHigherPrec(S.top(), exp[i]
      {
          res ← res + S.top()
          S.pop()
      }
      S.Push(exp[i])
    }
  }
  while(!S.empty())
  {
     res ← res + S.top()
     S.pop()
  }
} return res
```

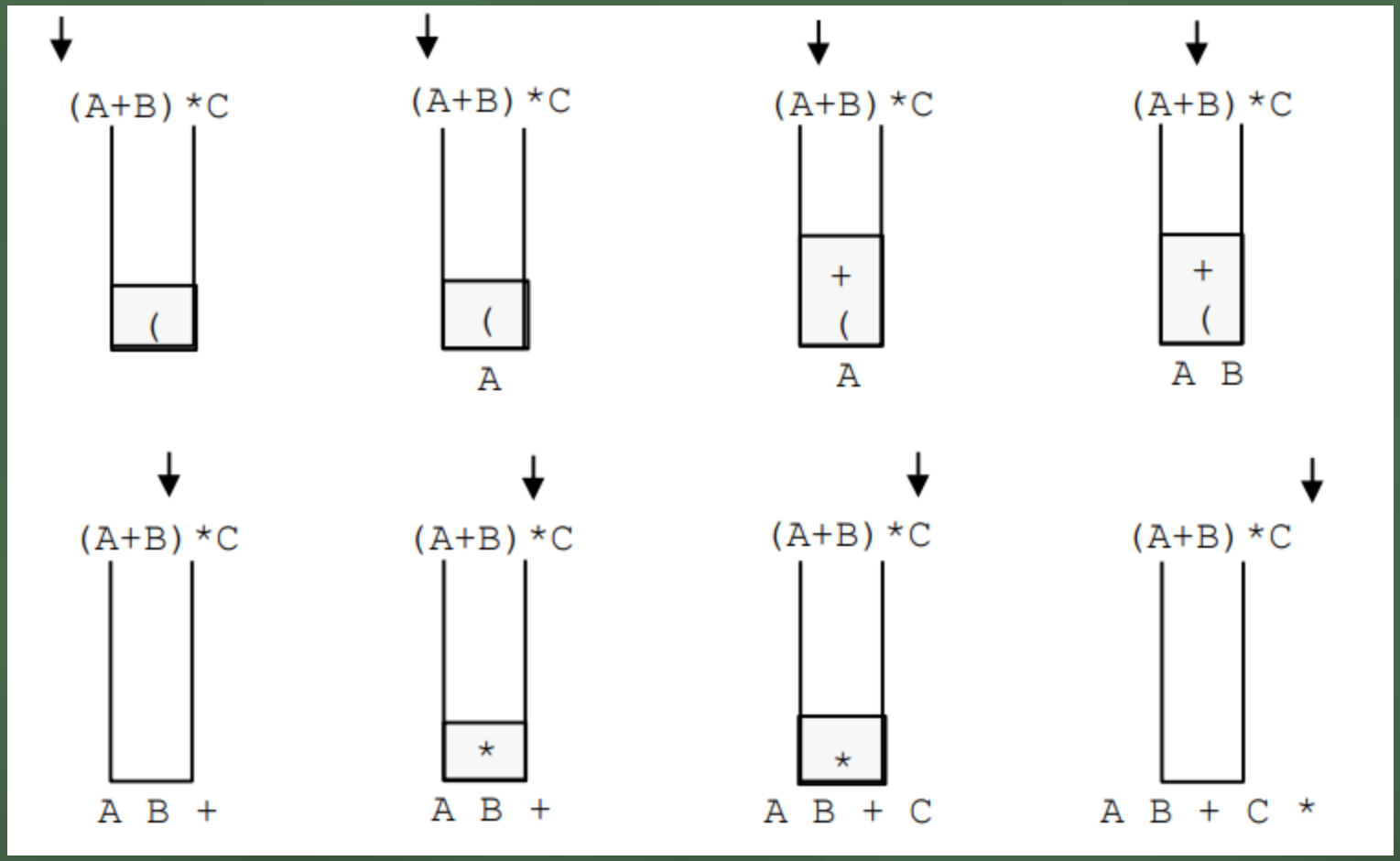
•Infix Form → Postfix Form

A / B ? C + D * E - A * C

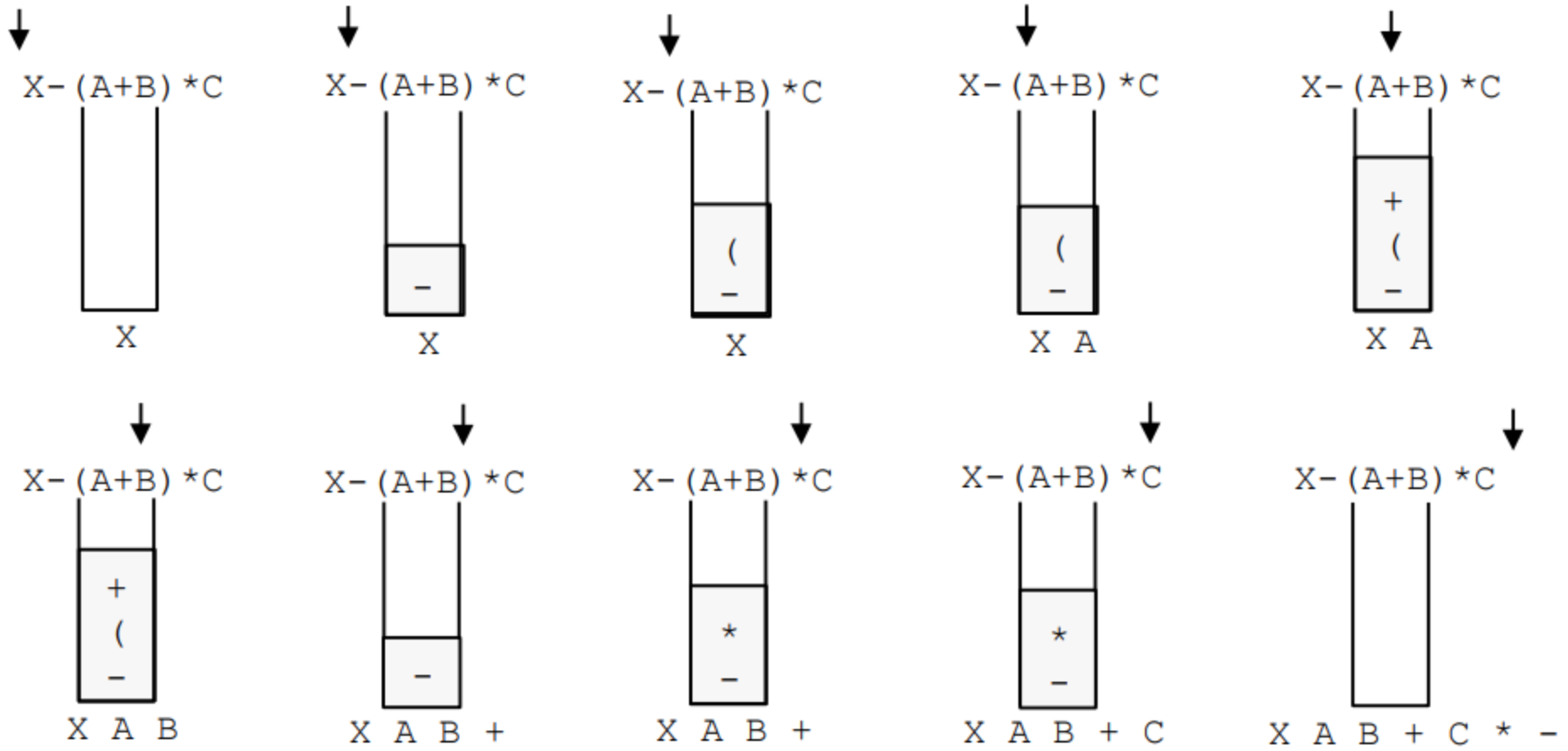
(((A / (B ? C)) + (D * E)) - (A * C))

A B C ? / D E * + A C * -

•Infix Form → Postfix Form



•Infix Form → Postfix Form



•Operator's priority

Symbol	In-Stack Priority	In-Coming Priority
)	-	-
?	3	4
*, /	2	2
+, -	1	1
(	0	4

Operators are taken out of the stack as long as the *in-stack priority* is greater than or equal to the *in-coming priority* of the new operator.


```
input : a*b?2 + 3
output: ab2
      : ab2?*
```

Algorithm

```
infixToPostfix(infix)
```

Input – Infix expression.

Output – Convert infix expression to postfix form.

```
Begin
   initially push some special character say # into the stack
   for each character ch from infix expression, do
      if ch is alphanumeric character, then
         add ch to postfix expression
      else if ch = opening parenthesis (, then
         push ( into stack
      else if ch = ^, then            //exponential operator of higher precedence
         push ^ into the stack
      else if ch = closing parenthesis ), then
         while stack is not empty and stack top ≠ (,
            do pop and add item from stack to postfix expression
         done

         pop ( also from the stack
      else
         while stack is not empty AND precedence of ch <= precedence of stack top ele
            pop and add into postfix expression
         done

         push the newly coming character.
   done

   while the stack contains some remaining characters, do
      pop and add to the postfix expression
   done
   return postfix
End
```