Processing Algebraic Expressions

- **Infix**: each binary operator appears between its operands *a + b*
- **Prefix**: each binary operator appears before its operands *+ a b*
- **Postfix***:* each binary operator appears after its operands *a b +*

Evaluate infix expressions:

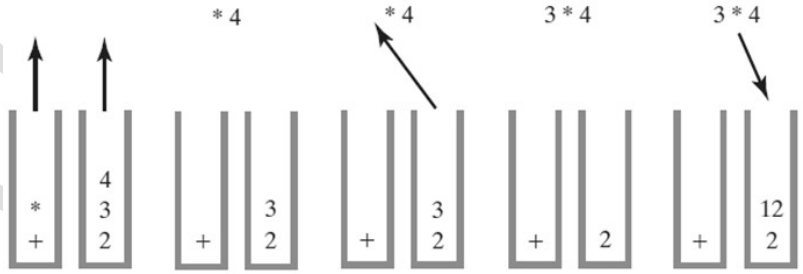
Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

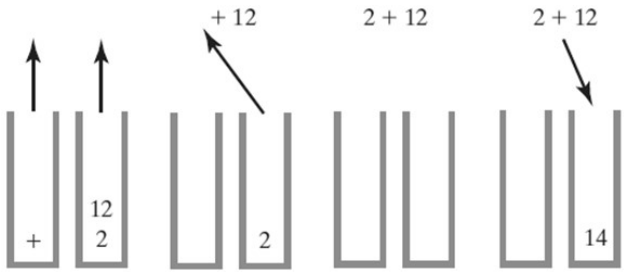
Example: evaluate a + b * c when a is 2, b is 3, and c is 4:

Step 1: Fill the two stacks until reaching the end of the expression:

Step 2: performing the multiplication:



Step 3: performing the addition:



Algorithm to evaluate infix expression:

```
Algorithm evaluateInfix(infix)
operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process) {
   nextCharacter = next nonblank character of infix
   switch (nextCharacter) {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek()) {
            // Execute operator at top of operatorStack
            topOperator = operatorStack.pop()
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
         }
         operatorStack.push(nextCharacter)
         break
      case '(' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(') {
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
            topOperator = operatorStack.pop()
         }
         break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty()) {
  topOperator = operatorStack.pop()
  operandTwo = valueStack.pop()
  operandOne = valueStack.pop()
  result = the result of the operation in topOperator and its operands
           operandOne and operandTwo
  valueStack.push(result)
}
return valueStack.peek()
```

Infix to Postfix Conversion

• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
• Open parenthesis	Push (onto the stack.
• Close parenthesis	Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

Example 1: Converting the **infix** expression **a + b * c** to **postfix** form

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Example 2: Successive Operators with Same Precedence: **a - b + c**

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	
	a b −	+
c	*a b* − *c*	+
	a b − *c* +	

Example 3: Successive Operators with Same Precedence: **a ^ b ^ c**

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Example 4: The steps in converting the infix expression **a / b * (c + (d – e))** to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
–	*a b* / *c d*	* (+ (–
e	*a b* / *c d e*	* (+ (–
)	*a b* / *c d e* –	* (+ (
	a b / *c d e* –	* (+
)	*a b* / *c d e* – +	* (
	a b / *c d e* – +	*
	a b / *c d e* – + *	

Infix-to-postfix Algorithm:

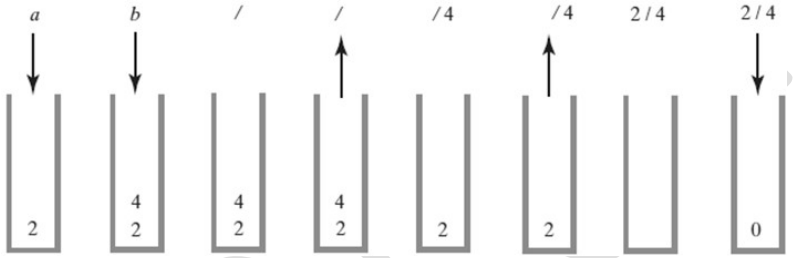
```
Algorithm convertToPostfix(infix)
operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse) {
   nextCharacter = next nonblank character of infix
   switch (nextCharacter) {

      case variable:
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek()){
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break
      case '( ' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(') {
              Append topOperator to postfix
              topOperator = operatorStack.pop()
          }
          break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty()) {
   topOperator = operatorStack.pop()
   Append topOperator to postfix
}
return postfix
```

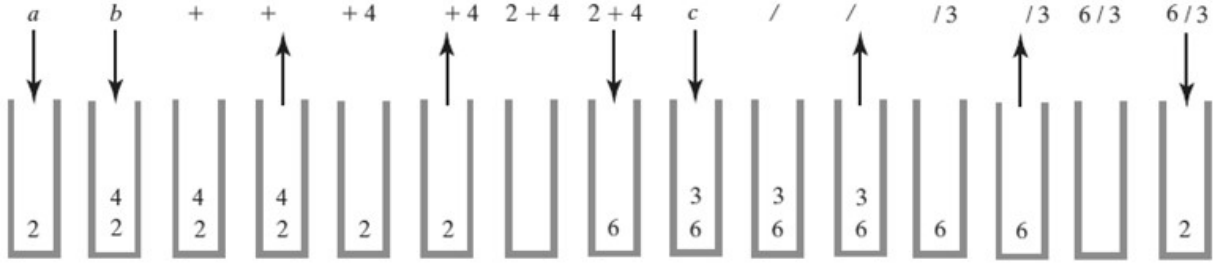
Evaluating Postfix Expressions

- When an **operand** is seen, it is **pushed** onto a stack.
- When an **operator** is seen, the appropriate numbers of **operands** are **popped** from the stack, the operator is **evaluated**, and the result is **pushed** back onto the stack.
    - Note that the **1st** item popped becomes the (**right** hand side) **rhs** parameter to the binary operator and that the **2nd** item popped is the (**left** hand side) **lhs** parameter; thus **parameters are popped in reverse order**.
    - For addition and multiplication, the order does not matter, but for subtraction and division, it does.
- When the complete postfix expression is evaluated, the result should be a single item on the stack that represents the answer.

Example 1: The stack during the evaluation of the postfix expression ***a b /*** when *a* is **2** and **b** is 4



Example 2: The stack during the evaluation of the postfix expression **a b + c /** when ***a*** is ***2***, ***b*** is ***4***, and ***c*** is ***3***



Self exercises:

- **2 3 4 + * 6 -** ➔ **8.0**
- **2 3 + 7 9 / -** ➔ **4.222**
- **10 2 8 * + 3 -** ➔ **23.0**
- **1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -** ➔ **-8.67**

Algorithm for evaluating postfix expressions.

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
    nextCharacter = next nonblank character of postfix
    switch (nextCharacter)
    {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' : case '^' :
         operandTwo = valueStack.pop()
         operandOne = valueStack.pop()
         result = the result of the operation in nextCharacter and its operands
                  operandOne and operandTwo
         valueStack.push(result)
         break
      default: break // Ignore unexpected characters
    }
}
```