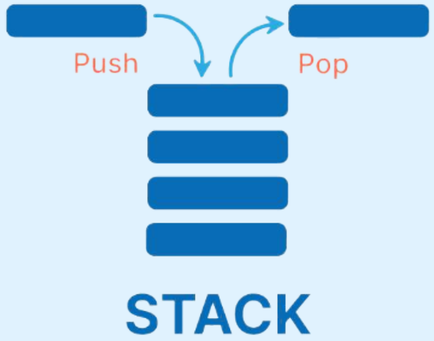
fit@hcmus

Reverse Polish Notation

Course: Data Structures and Algorithms



Presented by Group 19 - 24C06

24127454 - Đặng Hoàng Nam

24127163 - Nguyễn Sơn Hải

24127171 - Huỳnh Thái Hoàng

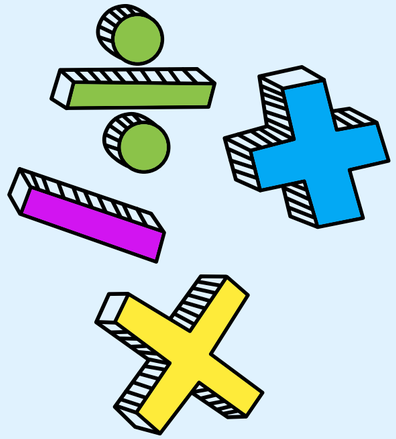
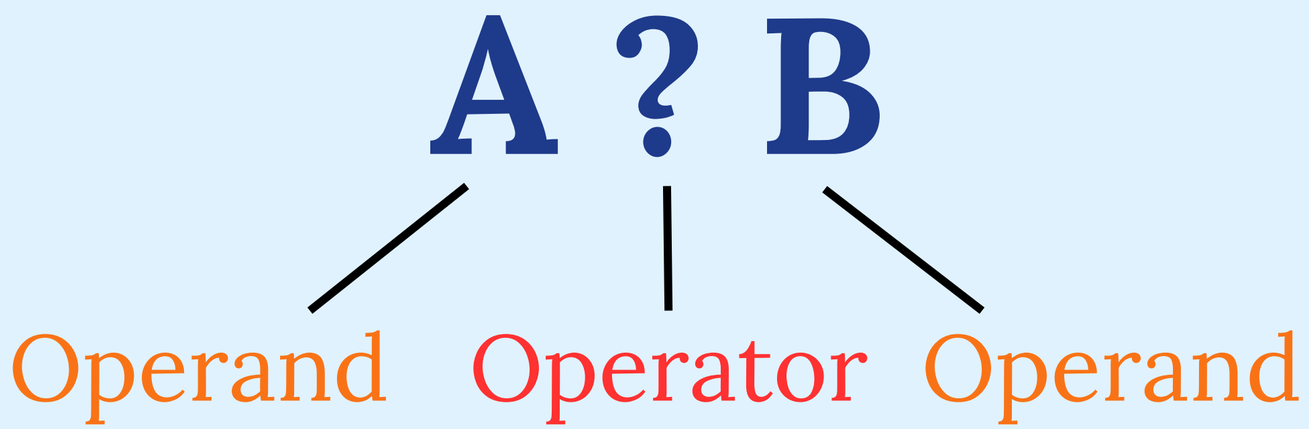


Table of Content

I. An Overview Of Notation

1. Infix Notation.

- The traditional and most common way of representing mathematical expressions.
- Operators are placed **between** operands.
- Usually use in mathematics, daily life,..



I. An Overview Of Notation

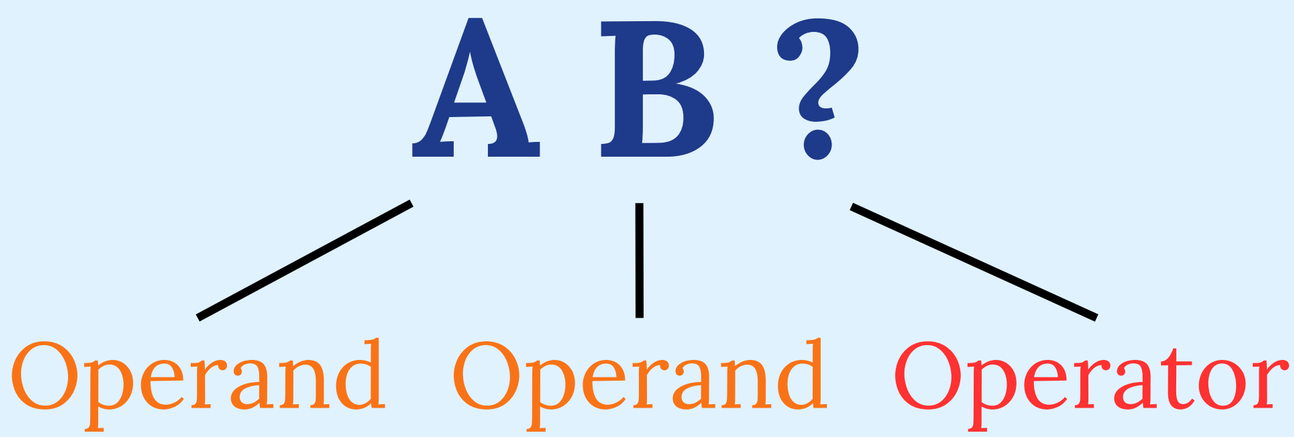
2. Prefix Notation (Polish Notation).

- Introduced by Jan Łukasiewicz (Polish logician) in the 1920.
- Operators are placed **before** operands.
- Efficiency in stack-based computation, and compiler design.

I. An Overview Of Notation

3. Postfix Notation (Reverse Polish Notation).

- A variation of Polish Notation, developed by Charles Hamblin in the 1950s.
- Operators are placed **after** operands (Examples: “3 4 +” = 3 + 4, “2 6 3 / *” = 2 * (6 / 3).).
- More common than PN, RPN plays an important role in computer science and engineering STEM (stack-based efficiency, hardware simplicity, ..).



II. Comparing Infix, Prefix, and Postfix Notations

Notation	Infix	Prefix	Postfix
Definition	Operator between operands	Operator before operands	Operator after operands
Parentheses	Required for precedence	Not needed	Not needed
Evaluation	Complex, needs precedence & parentheses	Recursive, right-to-left	Recursive, left-to-right
Readability	Easy, natural for humans	Moderate	Difficult
Machine Efficiency	Conversion requirement	Effective	Very effective

III. Converting Between Notations

1. Infix To Postfix Notation (RPN).

a. Algorithm.

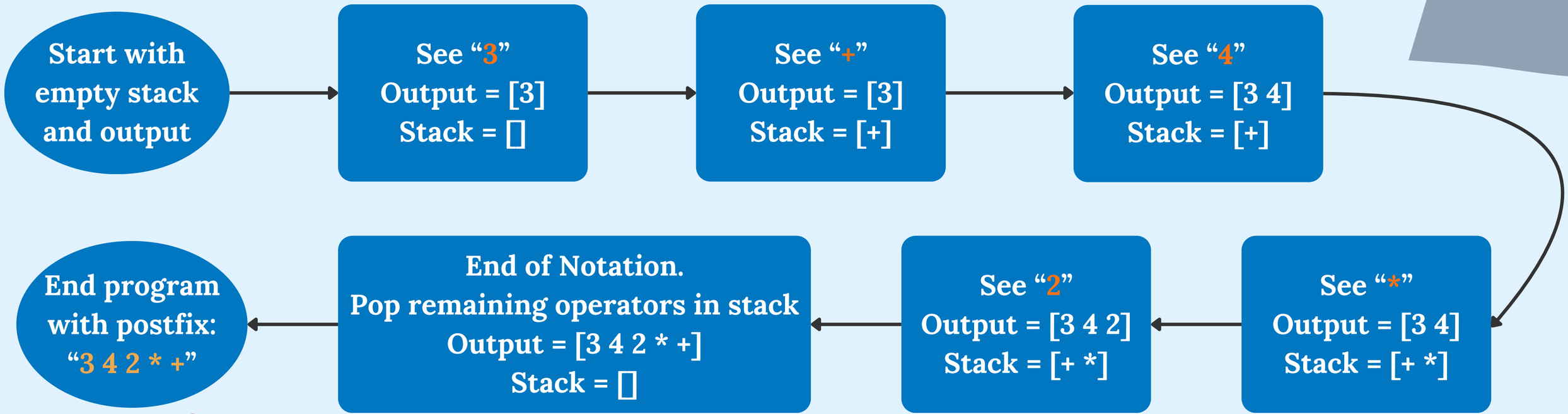
1. Read the character from **left to right** :
   - If the character is an **operand** : Add it directly to the output.
   - If the character is an **operator (+, -, *, /, ^, etc.)** :
     - While there's an operator at the **top of the stack** with **greater or equal precedence**, pop it to the output
     - Push the current operator onto the stack.
   - If the character is a **left parenthesis '('** : Push it onto the stack.
   - Else if it is a **right parenthesis ')'** : popped to output until matching '(' is encountered.
2. After processing all tokens, the **output** is the postfix notation.

III. Converting Between Notations

1. Infix To Postfix Notation (RPN).

b. Example.

Convert Infix notation : 3 + 4 * 2 to Postfix



III. Converting Between Notations

2. Infix To Prefix Notation

a. Algorithm.

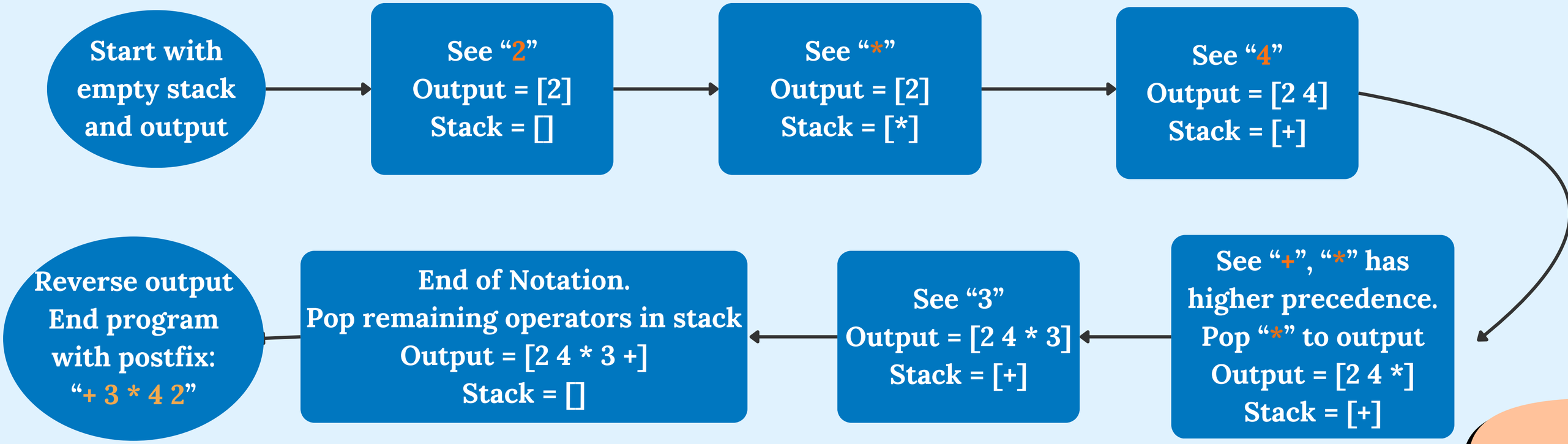
1. Initialize an empty stack, reverse parentheses.
2. Read the character from **right to left** :
   - If the character is an **operand** : Add it directly to the output.
   - If the character is an **operator (+, -, *, /, ^, etc.)** :
     - While there's an operator at the **top of the stack** with **greater or equal precedence**, pop it to the output.
     - Push the current operator onto the stack.
   - If the character is a **left parenthesis '('** : Push it onto the stack.
   - Else if it is a **right parenthesis ')'** : popped to output until matching '(' is encountered.
3. After processing all tokens, pop **remaining operators** from the stack to the output.

III. Converting Between Notations

2. Infix To Prefix Notation

b. Example.

Convert Infix notation : 3 + 4 * 2 to Prefix



III. Converting Between Notations

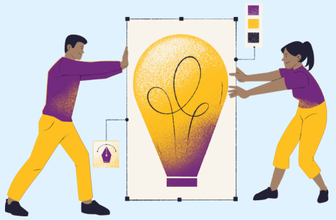
3. Prefix To Infix Notation

a. Algorithm.

1. Initialize an empty stack.
2. Process from **right to left** :
   - If the character is an **operand** : Push it onto stack.
   - If the character is an **operator (+, -, *, /, ^, etc.)** :
     - Pop the **top two values** of stack (op1, op2).
     - Form an infix expression as “**(op1 operator op2)**”.
     - Push infix expression onto stack
3. After processing all tokens, the **remaining value** in stack is infix expression.

III. Converting Between Notations

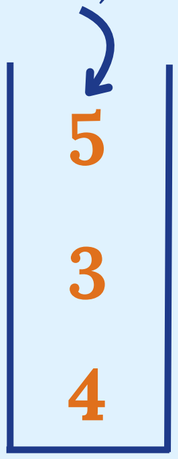
3. Prefix To Infix Notation



b. Example.

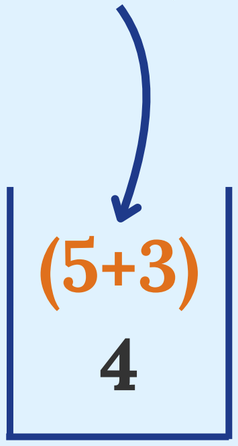
Convert Prefix notation : / * 2 + 5 3 4 to Infix

Push “4”, “3” and “5” (Operands) to stack



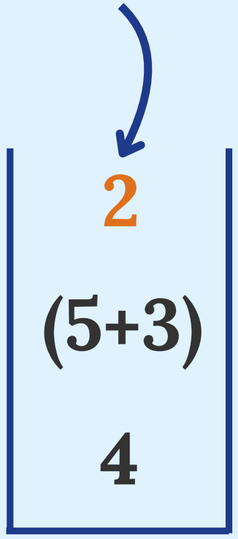
Step 1

See “+”. Pop “5” and “3”. Push “(5 + 3)”



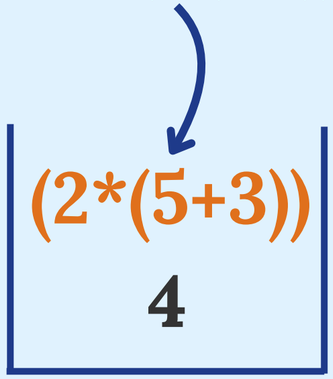
Step 2

Push “2” to stack.



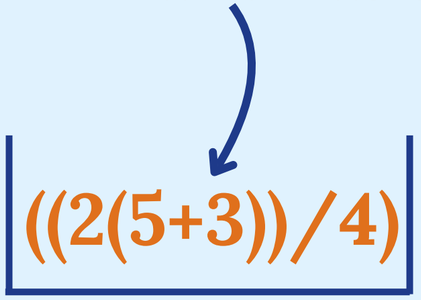
Step 3

See “*”. Pop “2” and “(5+3)”. Push “(2*(5+3))”



Step 4

See “/”. Pop “(2*(5+3))” and “4”. Push “((2*(5+3))/4)”



Step 5

III. Converting Between Notations

4. Prefix To Postfix Notation

a. Algorithm.

1. Initialize an empty stack.
2. Process from **right to left** :
   - If the character is an **operand** : Push it onto stack.
   - If the character is an **operator (+, -, *, /, ^, etc.)** :
     - Pop the **top two values** of stack (op1, op2).
     - Form a postfix expression as "**op1 op2 operator**".
     - Push postfix expression onto stack
3. After processing all tokens, the remaining value in stack is postfix expression.

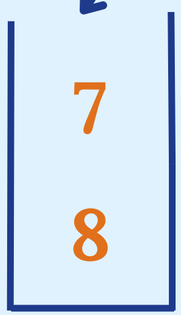
III. Converting Between Notations

4. Prefix To Postfix Notation

b. Example.

Convert Prefix notation : - * + 5 6 7 8 to Postfix

Push "8" and "7" (Operands) to stack

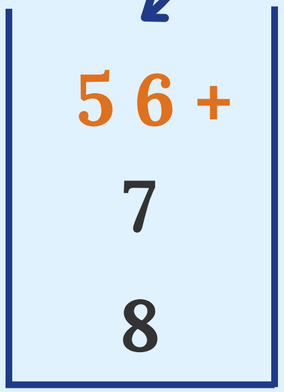


Step 1

Push "6" and "5" (Operands) to stack

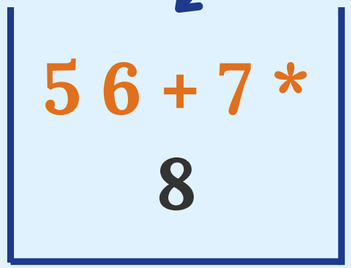
Step 2

See "+". Pop "6" and "5" out of stack. Push "5 6 +"



Step 3

See "*". Pop "5 6 +" and "7" out of stack. Push "5 6 + 7 *"



Step 4

See "-". Pop "5 6 + 7 *" and "8" out of stack. Push "5 6 + 7 * 8 -"

Step 5

III. Converting Between Notations

5. Postfix To Infix Notation

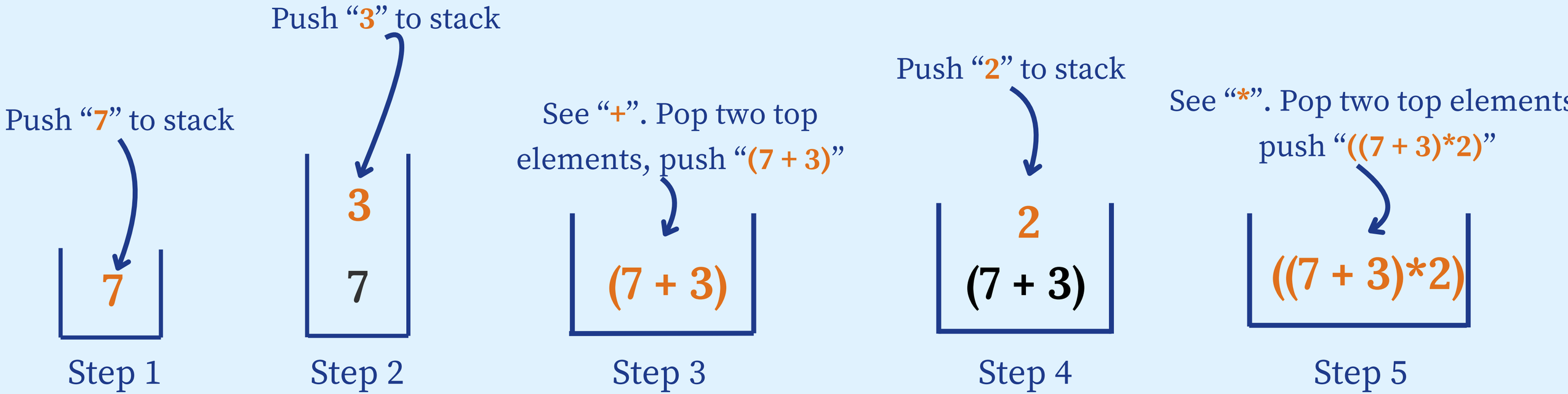
a.Algorithm.

1. Initialize an empty stack.
2. Process from left to right :
   - If the character is an **operand** : Push it onto stack.
   - If the character is an **operator** (+, -, *, /, ^, etc.) :
     - Pop the **top two values** of stack (op1, op2).
     - Form an infix expression as “**(op2 operator op1)**”.
     - Push infix expression onto stack
3. After processing all tokens, the remaining value in stack is infix expression.

III. Converting Between Notations

5. Postfix To Infix Notation

b. Example. **Convert Postfix notation : 7 3 + 2 * to Infix**



III. Converting Between Notations

6. Postfix To Prefix Notation

a.Algorithm.

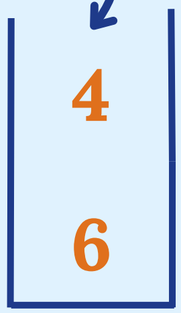
1. Initialize an empty stack.
2. Process from **left to right** :
   - If the character is an **operand** : Push it onto stack.
   - If the character is an **operator (+, -, *, /, ^, etc.)** :
     - Pop the **top two values** of stack **(op1, op2)**.
     - Form an prefix expression as “**operator op2 op1** ”.
     - Push prefix expression onto stack
3. After processing all tokens, the remaining value in stack is prefix expression.

III. Converting Between Notations

6. Postfix To Prefix Notation

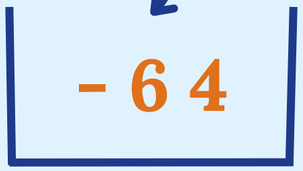
b. Example. **Convert Postfix notation : 6 4 - 4 7 + * to Prefix**

Push "6" and "4" (Operands) to stack



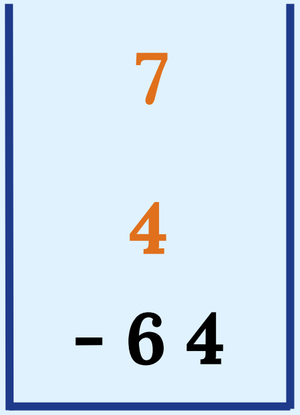
Step 1

See "-". Pop two top elements, push "- 6 4"



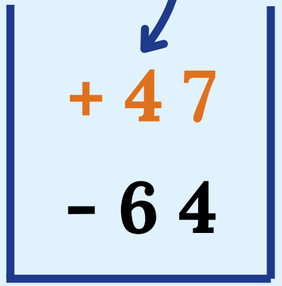
Step 2

Push "4" and "7" (Operands) to stack



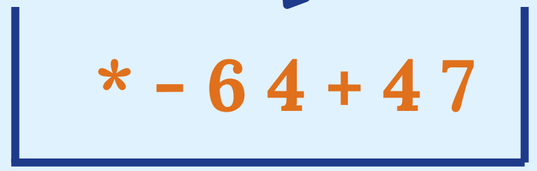
Step 3

See "+". Pop two top elements, push "+ 4 7"



Step 4

See "*". Pop two top elements. Push "* - 6 4 + 4 7"



Step 5

IV. Expression Evaluation

1. Infix Evaluation

a.Algorithm.

1. Use **two stacks**: one for **operands** and one for **operators**.
2. We start from **left to right**, if we encounter:
   - **Operand**: Push to operand stack.
   - **Operator (+, -, *, /, ^, (,), etc.)**:
     - While operator stack is not empty and the **top operator** has **higher or equal precedence** than the current one, pop and apply operator.
     - Then push current value to the operator stack.
   - '(': Push to operator stack.
   - ')': Pop and apply operators until '(', discard '('.
3. After scanning, process **remaining** operators.
4. The final result is the only element left in the **operand stack**.

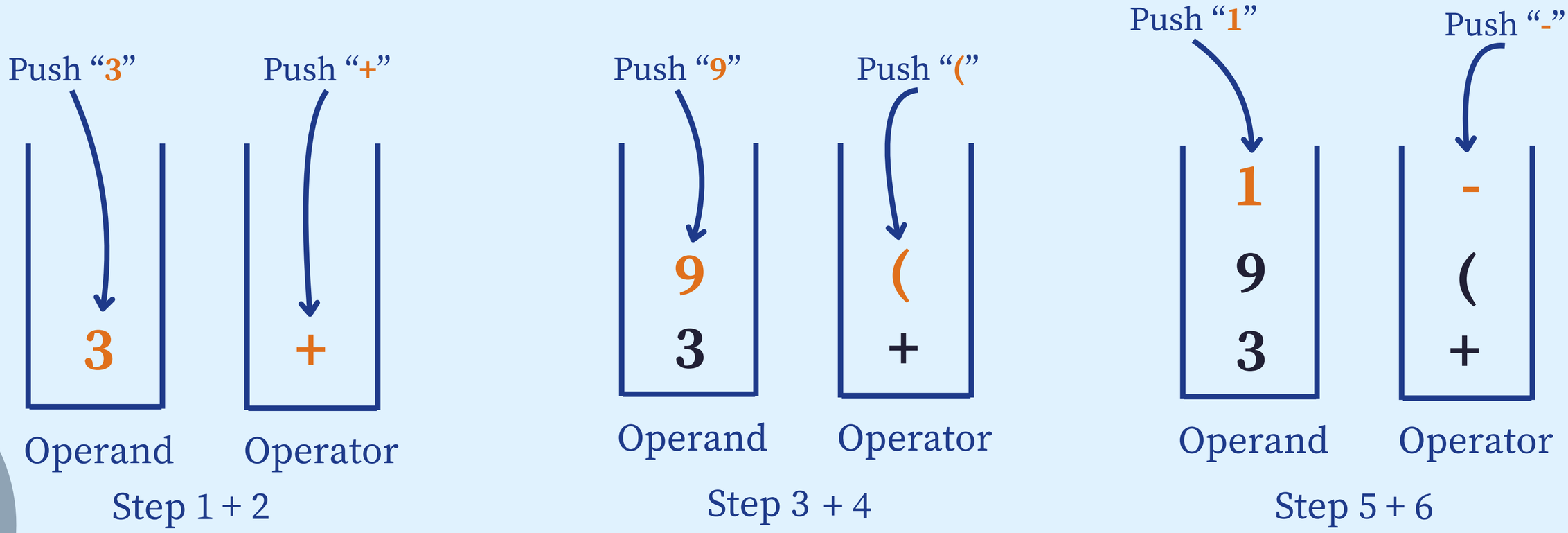
IV. Expression Evaluation

1. Infix Evaluation

b. Example.

Evaluating Infix Expression:

3 + (9 - 1) / 2



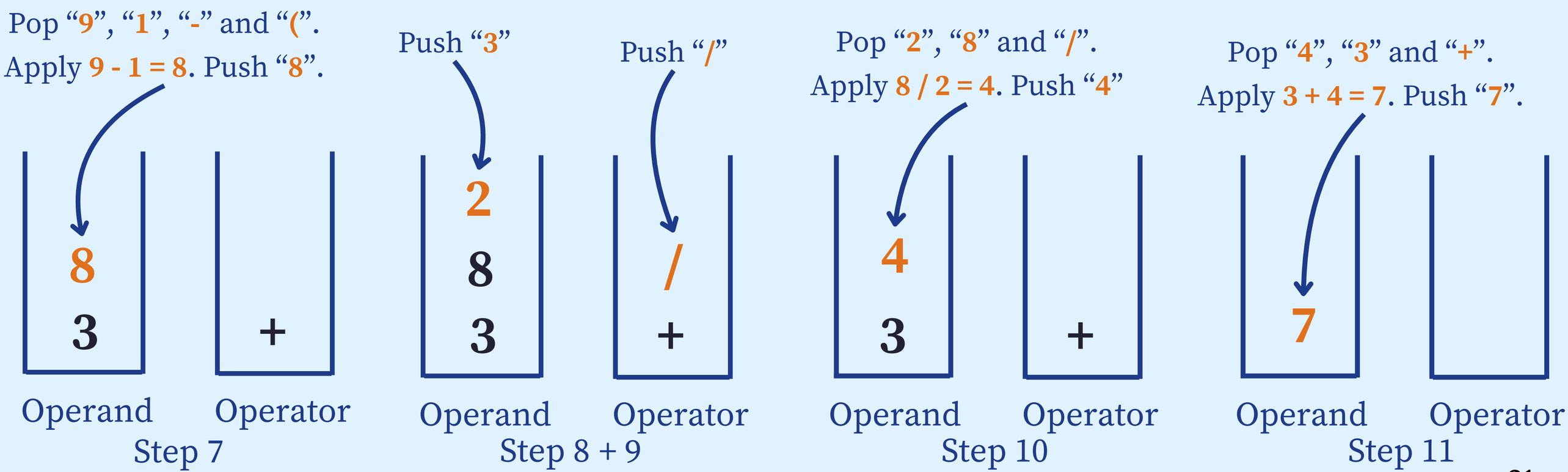
IV. Expression Evaluation

1. Infix Evaluation

b. Example.

Evaluating Infix Expression:

3 + (9 - 1) / 2



IV. Expression Evaluation

2. Prefix Evaluation

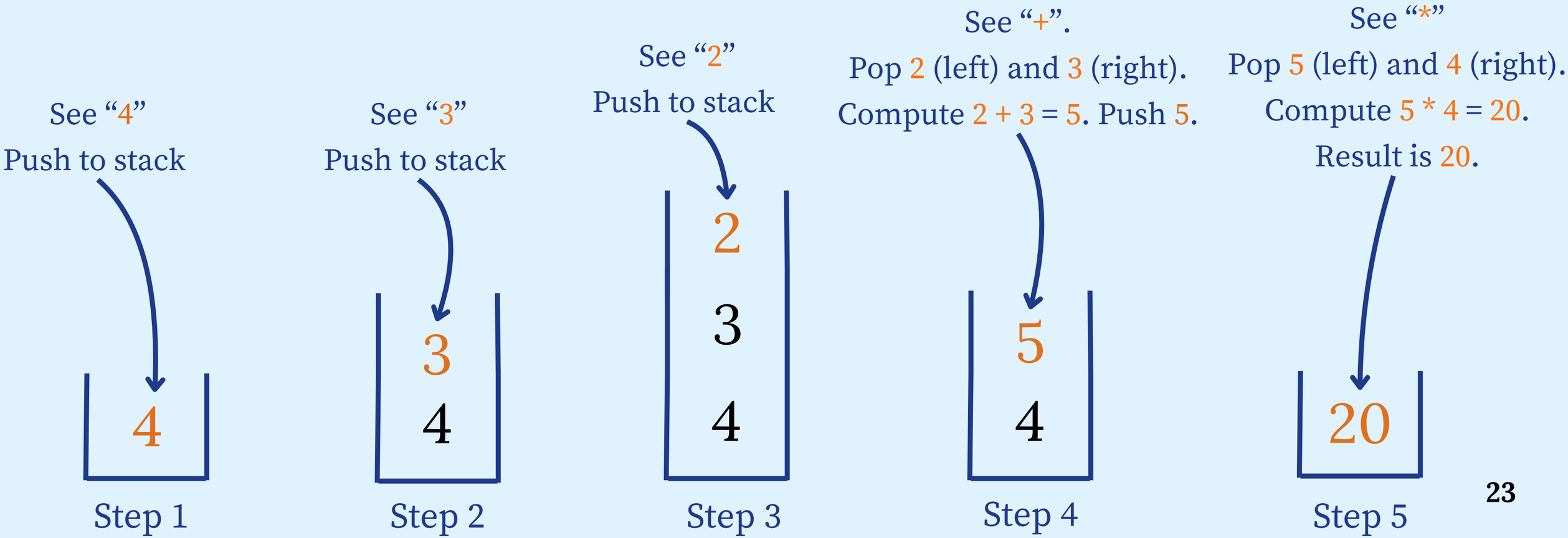
a.Algorithm.

1. Initialize a stack.
2. We start from **right to left**:
   - **Operand**: Push to operand stack.
   - **Operator (+, -, *, /, ^, (,), etc.)**:
     - Pop **two operands** from the stack (**first** popped is the **left operand**, **second** is the **right**)
     - Apply the operator as **left operator right**, and push the result back to the stack.
3. After scanning, the result is the only element left in the stack.

IV. Expression Evaluation

2. Prefix Evaluation

b. Example. **Evaluating Prefix Expression: * + 2 3 4.**



IV. Expression Evaluation

3. Postfix Evaluation with Stack

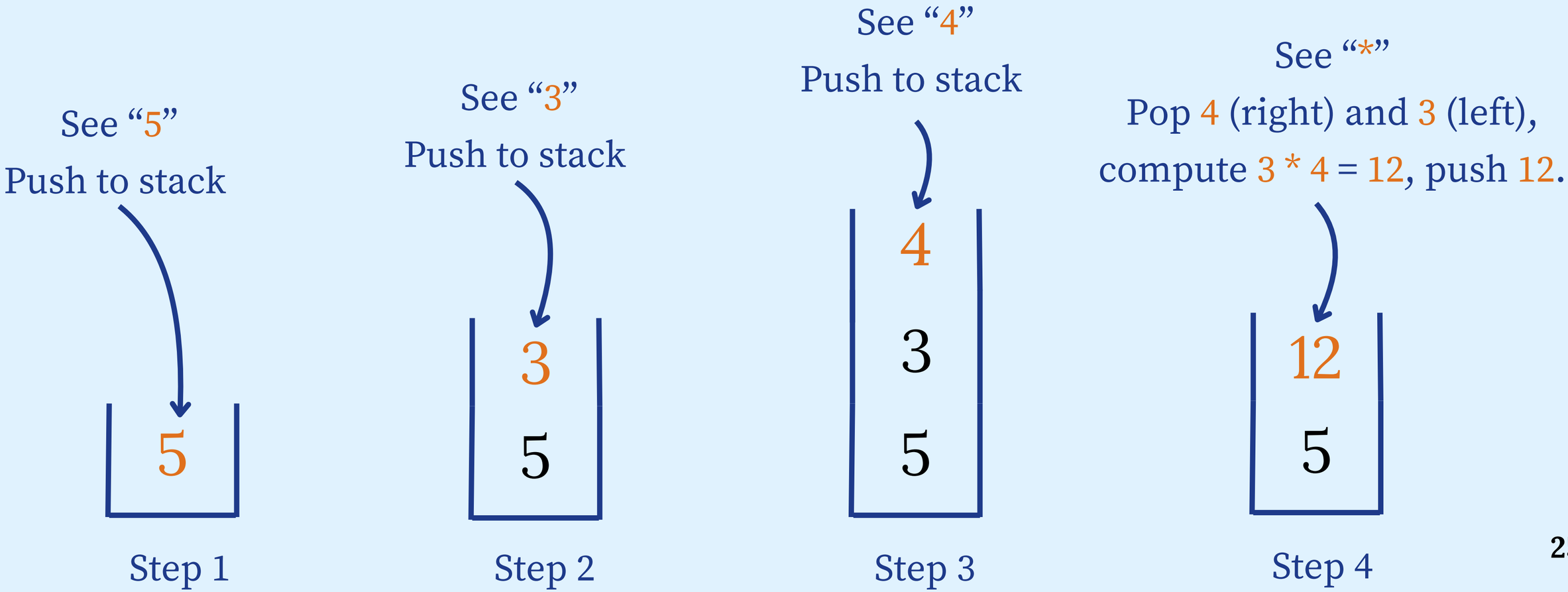
a. Algorithm.

1. Initialize a stack.
2. We start from **left to right**:
   - **Operand**: Push to operand stack.
   - **Operator (+, -, *, /, ^, (,), etc.)**:
     - Pop two operands from the stack (**first** popped is the **left operand**, **second** is the **right**)
     - Apply the operator as **left operator right** , and push the result back to the stack.
3. After scanning, the result is the only element left in the stack.

IV. Expression Evaluation

3. Postfix Evaluation with Stack

b. Example. **Evaluating Postfix Expression: 5 3 4 * + 2 -.**



IV. Expression Evaluation

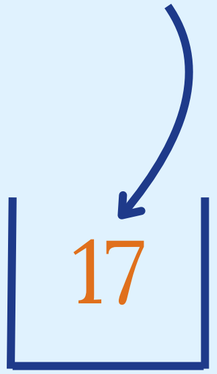
3. Postfix Evaluation with Stack

b. Example. **Evaluating Postfix Expression: 5 3 4 * + 2 -.**

See “+”

Pop 12 (right) and 5 (left),

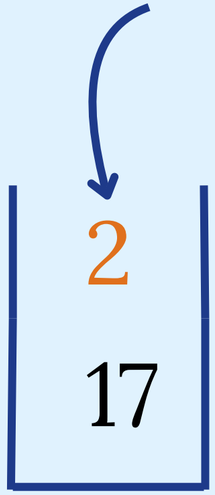
compute 5 + 12 = 17, push 17.



Step 5

See “2”

Push to stack

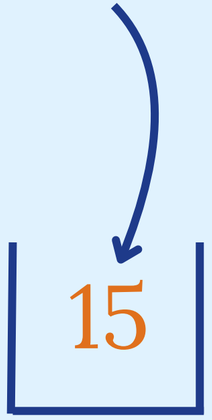


Step 6

See “-”

Pop 2 (right) and 17 (left),

compute 17 - 2 = 15, push 15.

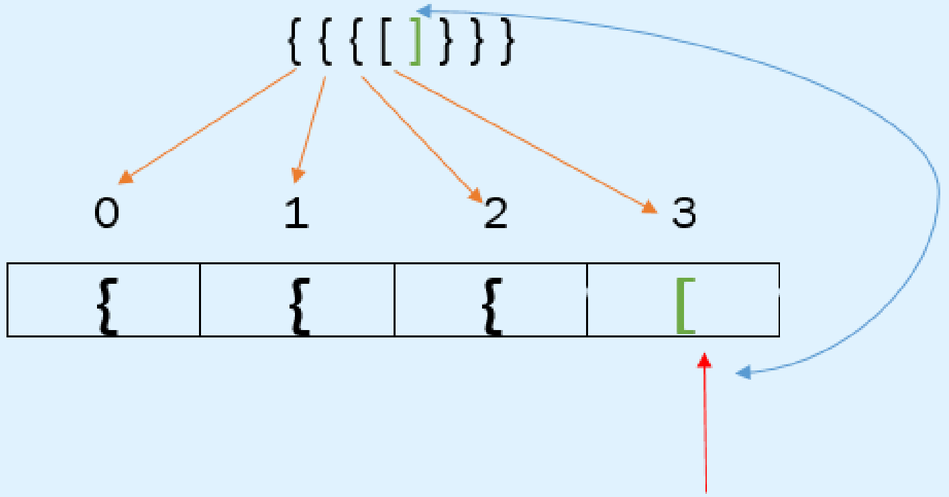


Step 7

V. Stack Application

1.Parentheses Matching

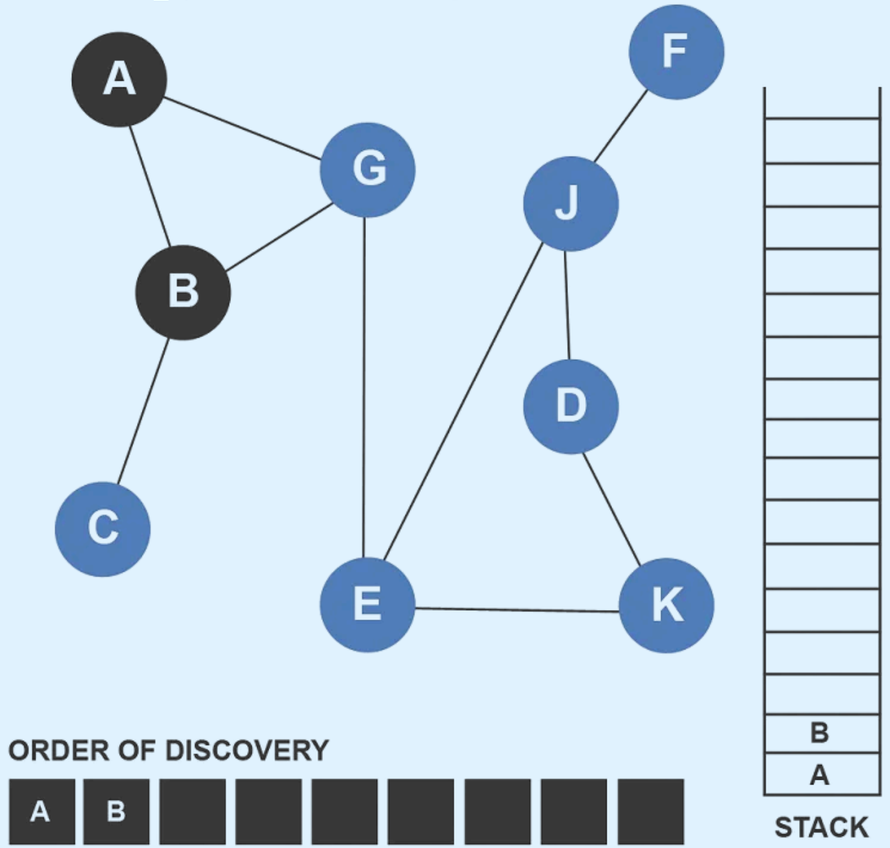
- In some programming problems, we have to check if an expression has **balanced parentheses.**
- Or we need to check the **correctness of parentheses** (), {}, and [] in mathematical expressions or programming code.
- Example: Compilers use stacks to verify correct syntax in source code.



V. Stack Application

2. Depth-First Search (Graph Traversals)

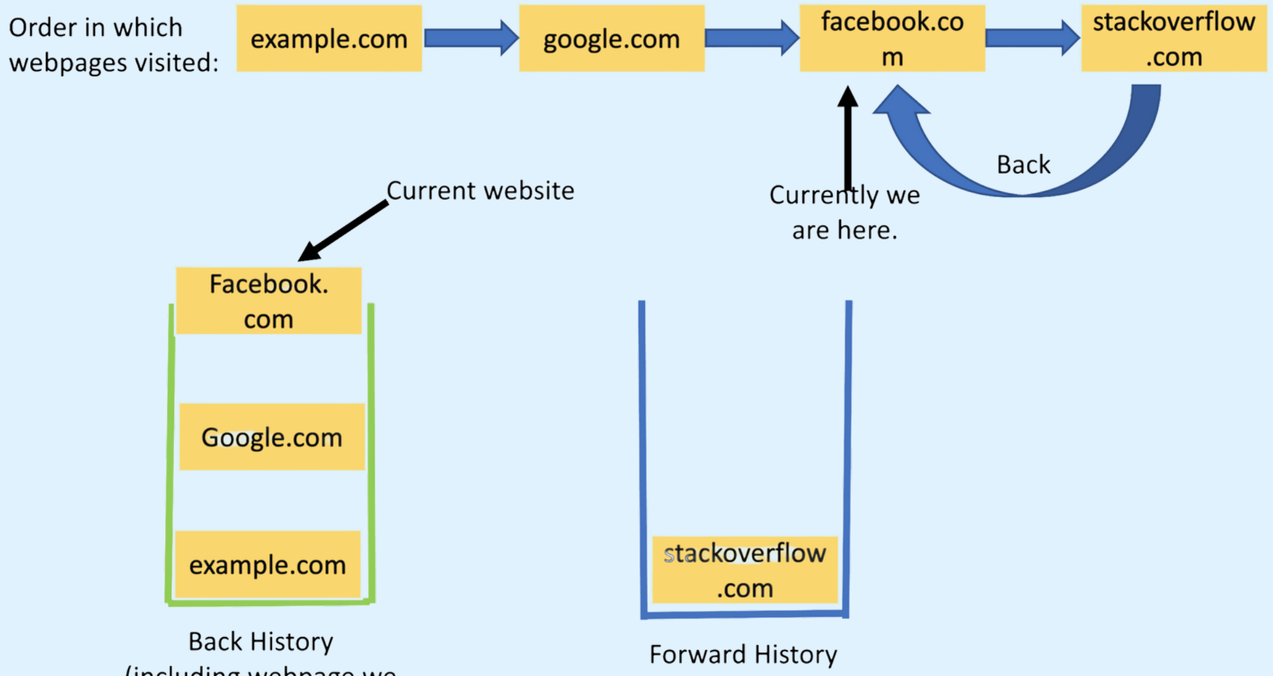
- Stacks are used in **DFS algorithms** to traverse trees and graphs.
- Example: Pathfinding in map, search algorithms.



V. Stack Application

3. The Web browser History

- Web browsers use two stacks **(backward & forward stacks)** to efficiently handle navigation using the **Last-In-First-Out (LIFO)** principle.
- Stack Operations in Browsing: New pages are pushed onto the **history stack**, the **Back button** pops the last page, and the **Forward button** restores it from the **forward stack.**



fit@hcmus

This is the end of the presentation

Thanks for listening

We appreciate your time! Let us know if you have any questions.

Presented by Group 19 - 24C06