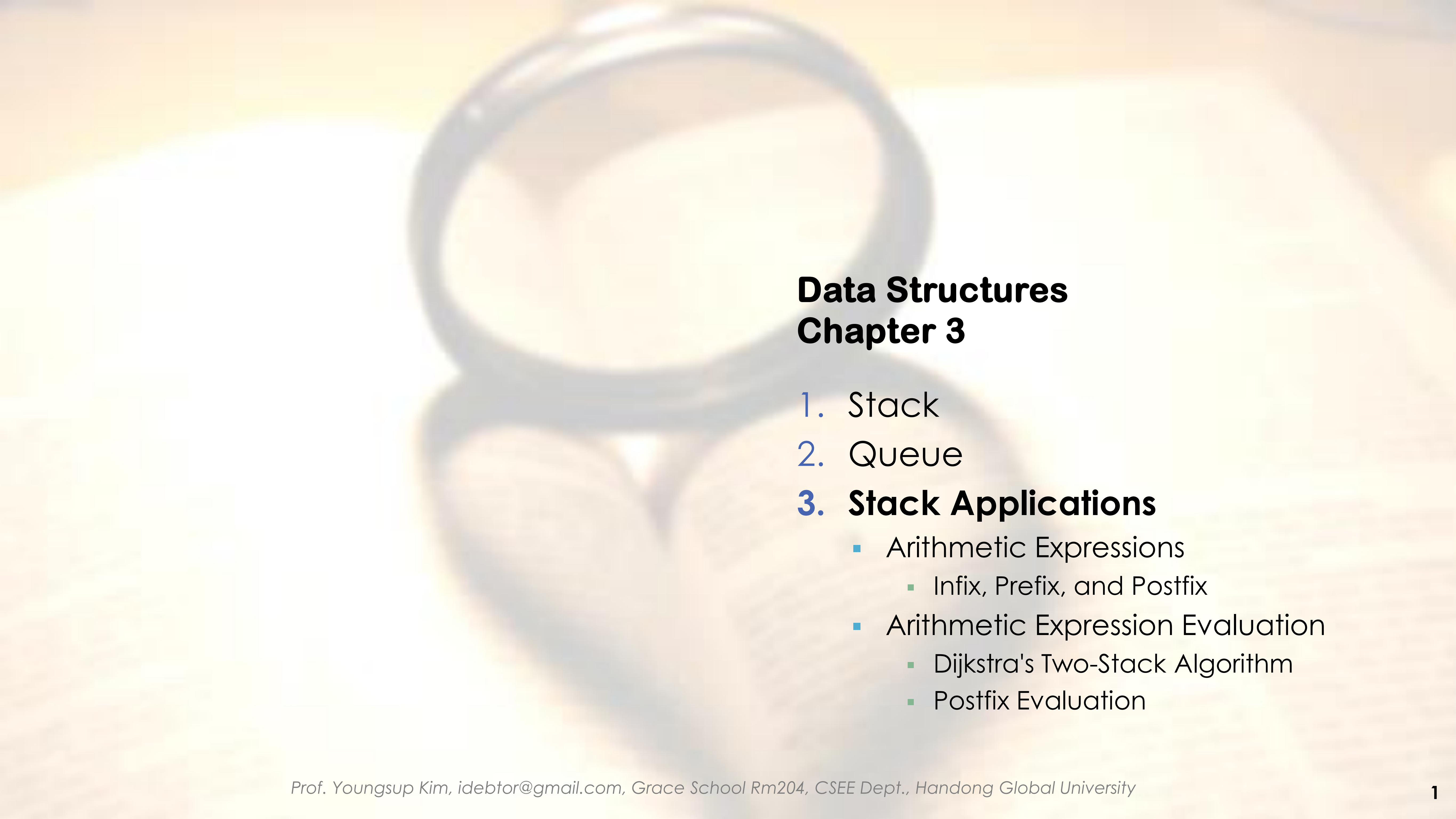
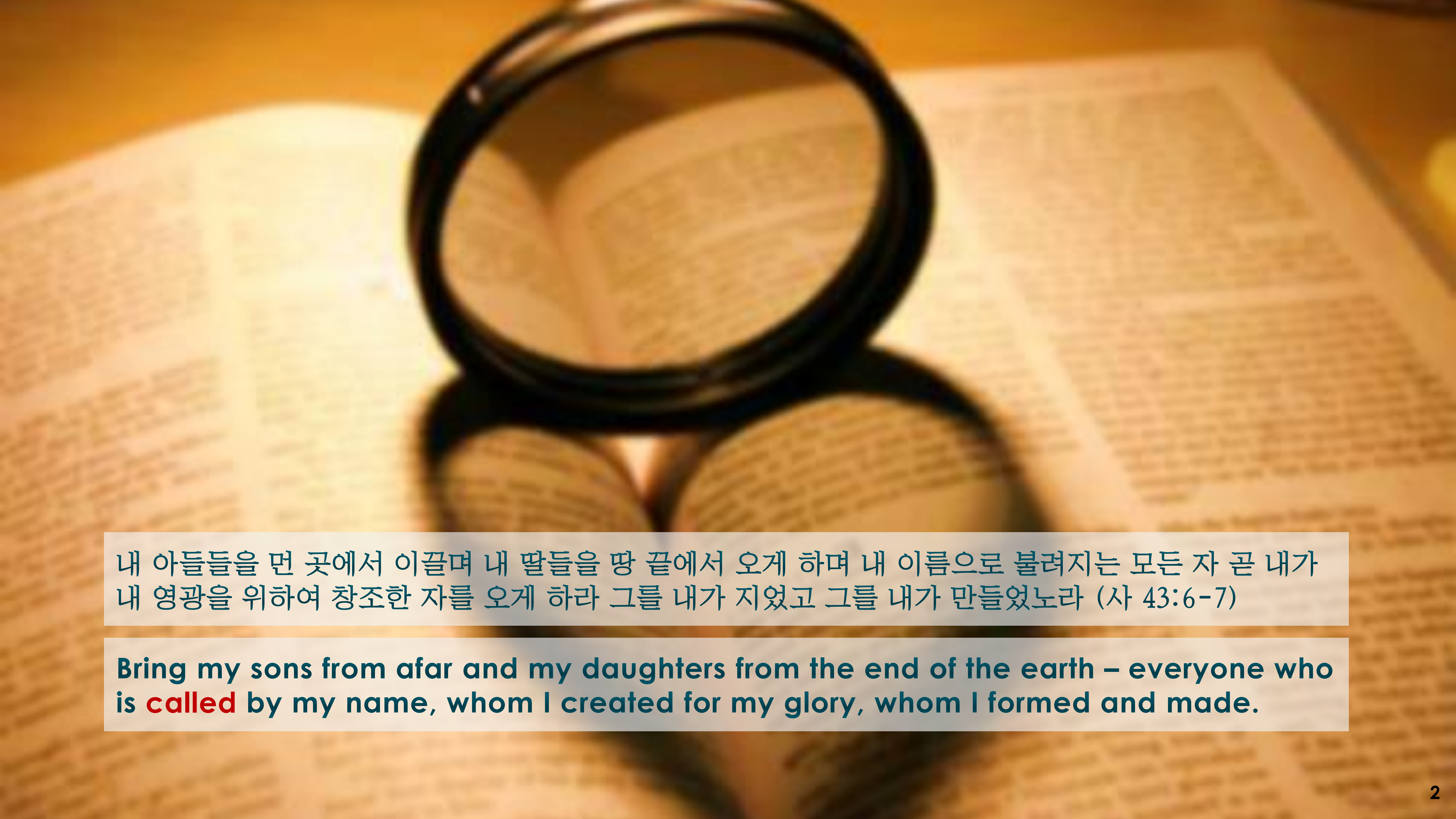


Data Structures
Chapter 3

1. Stack
2. Queue
3. **Stack Applications**
   - Arithmetic Expressions
     - Infix, Prefix, and Postfix
   - Arithmetic Expression Evaluation
     - Dijkstra's Two-Stack Algorithm
     - Postfix Evaluation

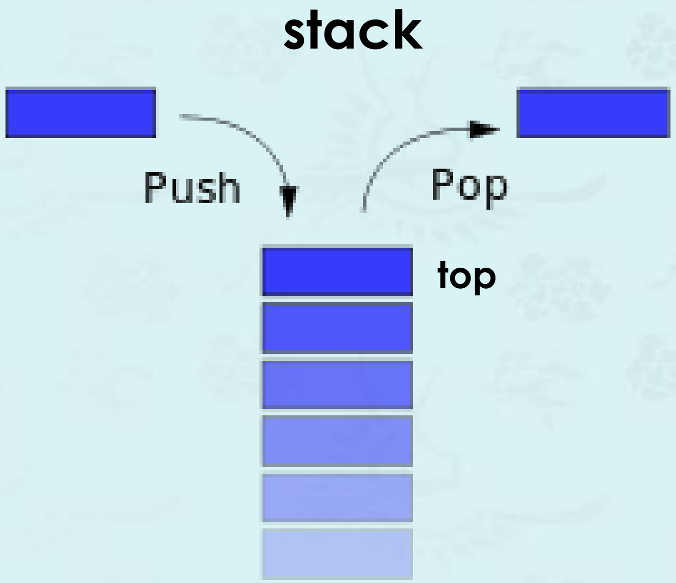


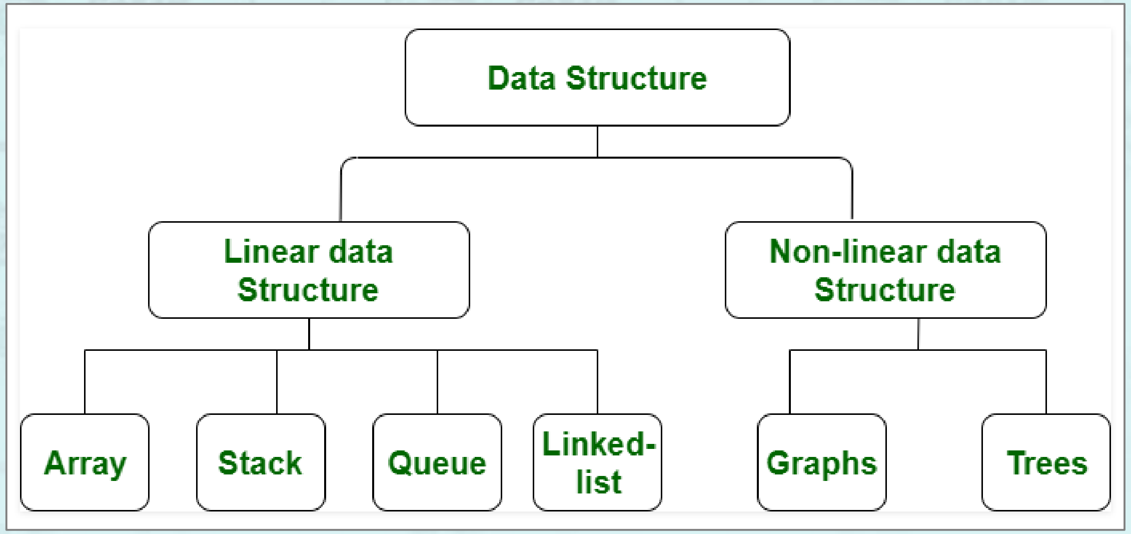
내 아들들을 먼 곳에서 이끌며 내 딸들을 땅 끝에서 오게 하며 내 이름으로 불려지는 모든 자 곧 내가
내 영광을 위하여 창조한 자를 오게 하라 그를 내가 지었고 그를 내가 만들었노라 (사 43:6-7)

Bring my sons from afar and my daughters from the end of the earth – everyone who is called by my name, whom I created for my glory, whom I formed and made.

Stack and Queue

- **Stack** is a linear data structure handled by a particular order of the operation is called **LIFO(Last In First Out).** It removes the item **most** recently added.

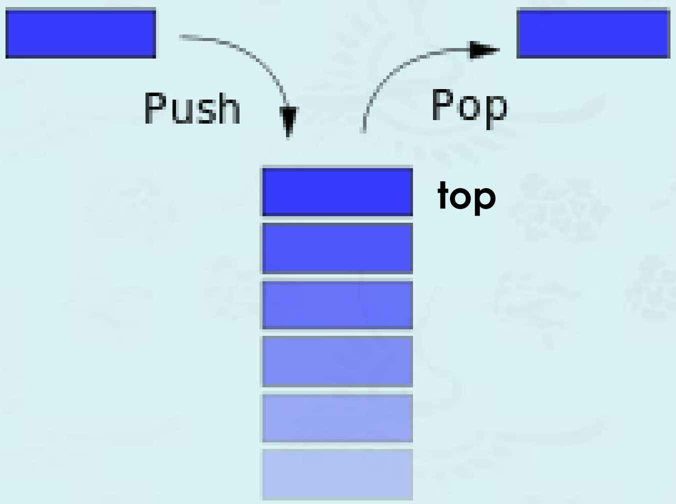




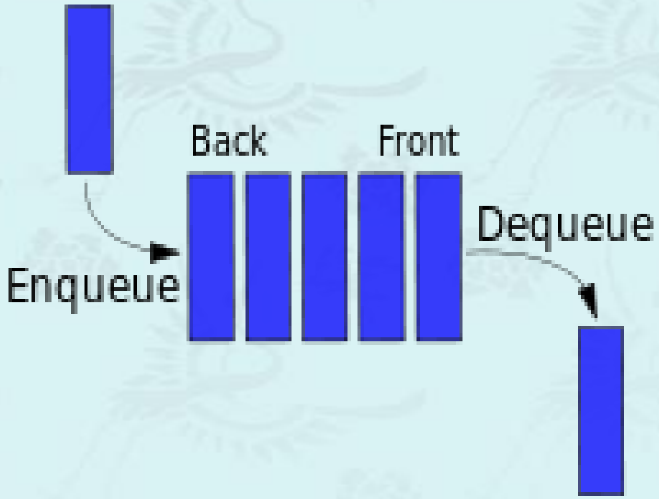
Stack and Queue

- **Stack** is a linear data structure handled by a particular order of the operation is called **LIFO(Last In First Out).** It removes the item **most** recently added.
- **Queue** is known as a **Fist-in-first-out(FIFO)** list since it removes the item **least** recently added.

stack

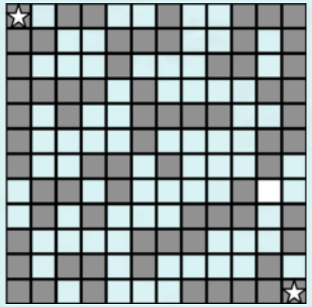


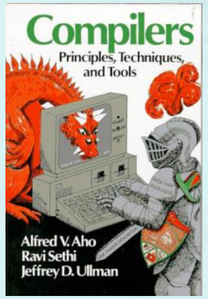
queue

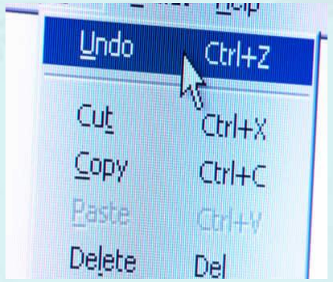


Stack Applications

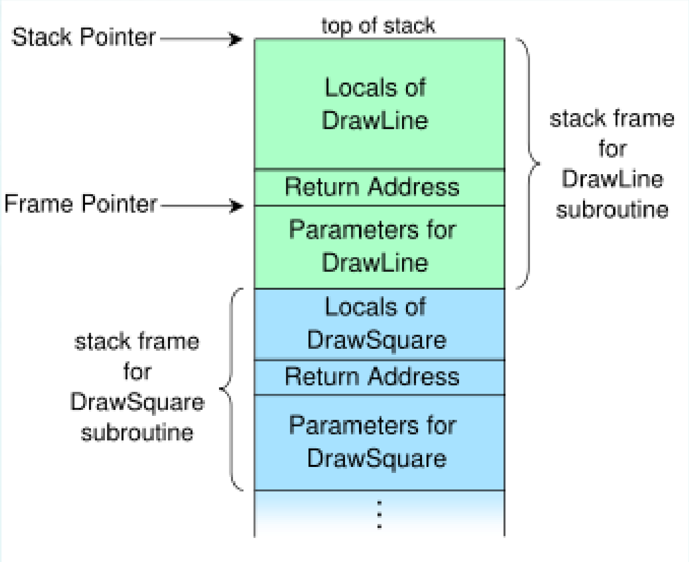
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Backtracking as in a maze
- Implementing function calls in a compiler.
- …





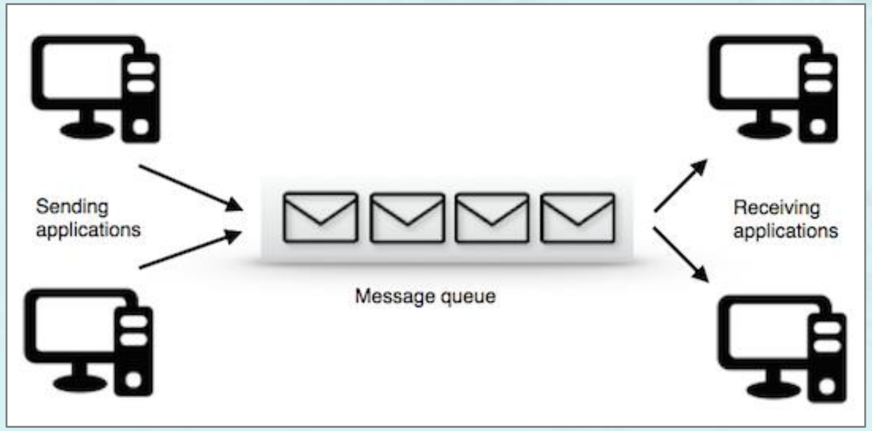


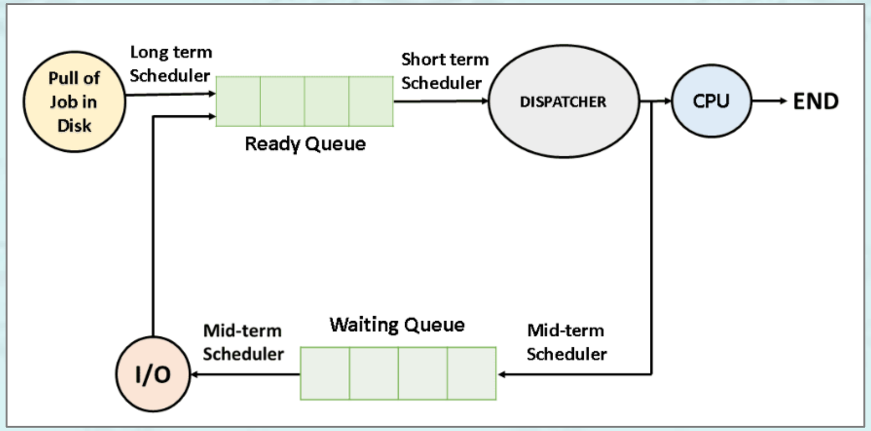
Adobe® PostScript® 3™



Queue Applications

- In a computer OS, requests for services come in unpredictable order and timing, sometimes faster than they can be serviced .
  - print a file
  - send an email
  - CPU scheduling
  - job scheduling in general





Infix, postfix and prefix expressions

Stacks can be used to implement algorithms involving Infix, postfix and prefix expressions.

- **Infix:**
  - An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another infix string.
  - A, A + B, (A + B) + (C – D)
- **Prefix:**
  - A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.
  - A, + A B, + + A B – C D
- **Postfix:**
  - A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.
  - A, A B +, A B + C D – +

Infix, postfix and prefix expressions

- **Prefix** and **postfix** notations are methods of writing mathematical expressions **without parenthesis.**
- Why: Time to evaluate a postfix and prefix expression is **O(n)**, where n is the number of elements in the array.

Infix	Prefix	Postfix
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 –

Infix, postfix and prefix expressions

- **Prefix** and **postfix** notations are methods of writing mathematical expressions **without parenthesis**.
- Why: Time to evaluate a postfix and prefix expression is O(n), where n is the number of elements in the array.

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
(a / (b – c + d)) * (e – a) * c	a b c – d + / e a – * c *
a / b – c + d * e – a * c	a b / c – d e * + a c * –

Infix to Postfix Conversion

- **Goal**: Convert an **infix** expression to a **postfix** expression using a **stack**.

operand

operator

(1 + 2) * 3

```
Stack: (
Output:
```

```
Stack: (
Output: 1
```

```
Stack: ( +
Output: 1
```

```
Stack: ( +
Output: 1 2
```

```
Stack:
Output: 1 2 +
```

```
Stack: *
Output: 1 2 +
```

```
Stack: *
Output: 1 2 + 3
```

```
Stack:
Output: 1 2 + 3 *
```

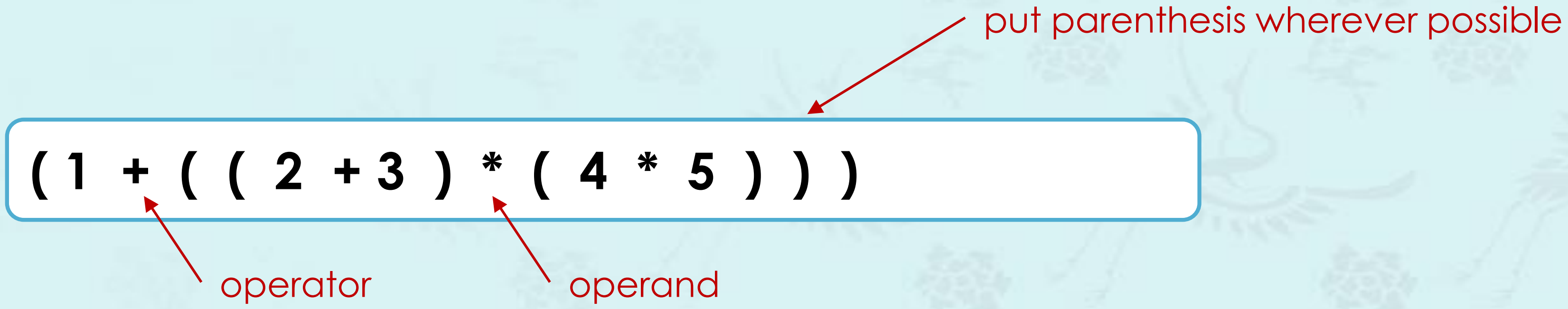
1. Operands are output immediately
2. Push "(" always and operators in general.
3. For ")", pop until "(". Discard "(" and ")".
4. For higher precedence operator, push it.
5. For lower or equal precedence operator, pop them until "(" and push it.

infix (1 + ((2 + 3) * (4 * 5)))

postfix 1 2 3 + 4 5 * * +

Infix Expression Evaluation

- **Goal**: Evaluate infix expressions.



put parenthesis wherever possible

(1 + ((2 + 3) * (4 * 5)))

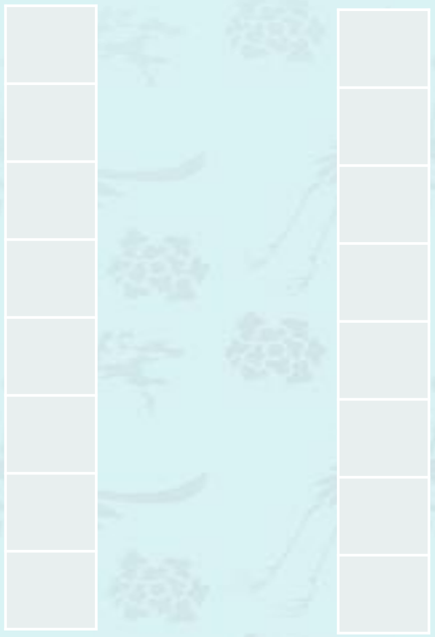
operator

operand

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 value stack.

Dijkstra’s two-stack algorithm - Evaluate infix expressions.



value stack
(operand stack)

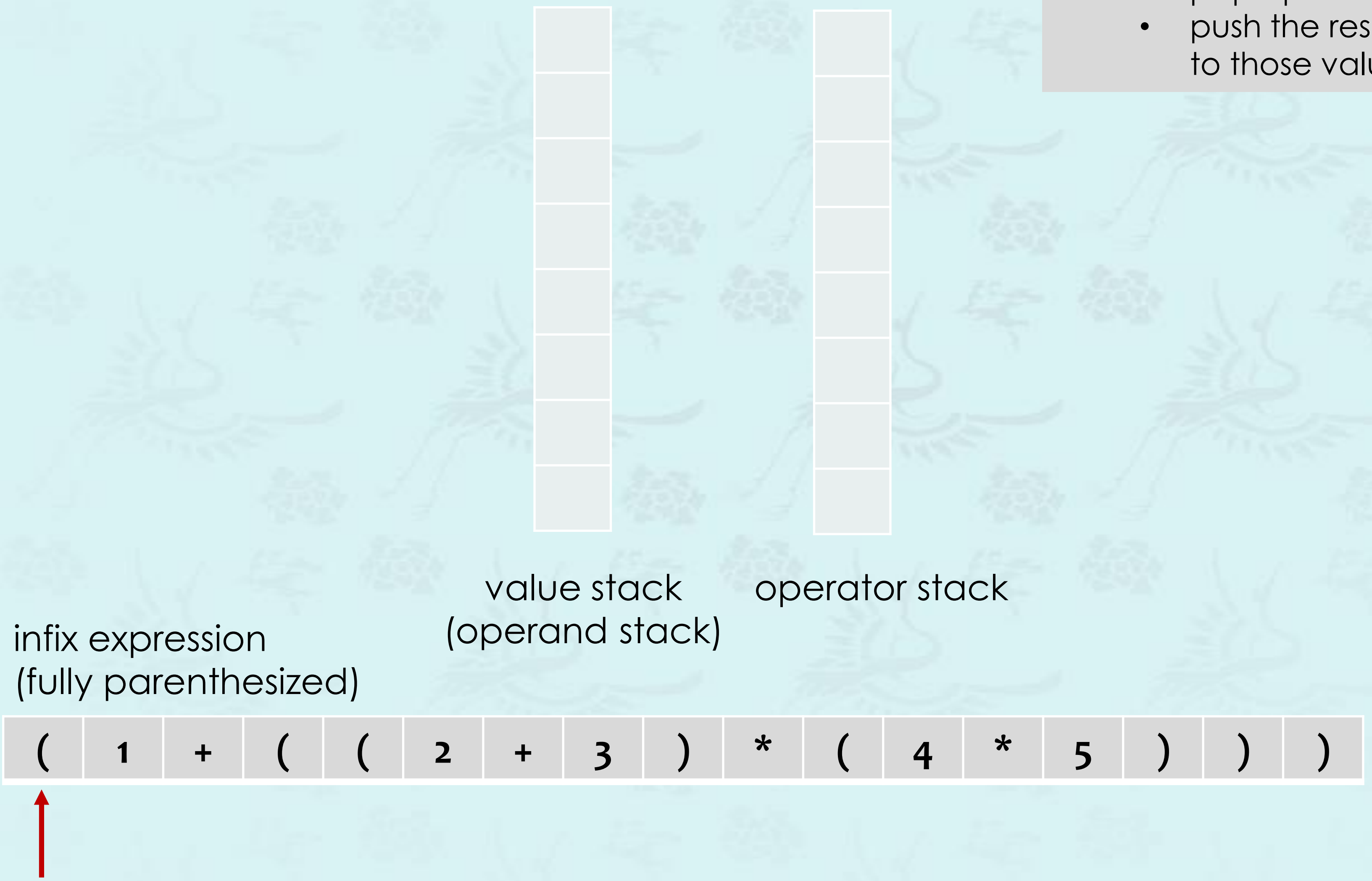
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



value stack
(operand stack)

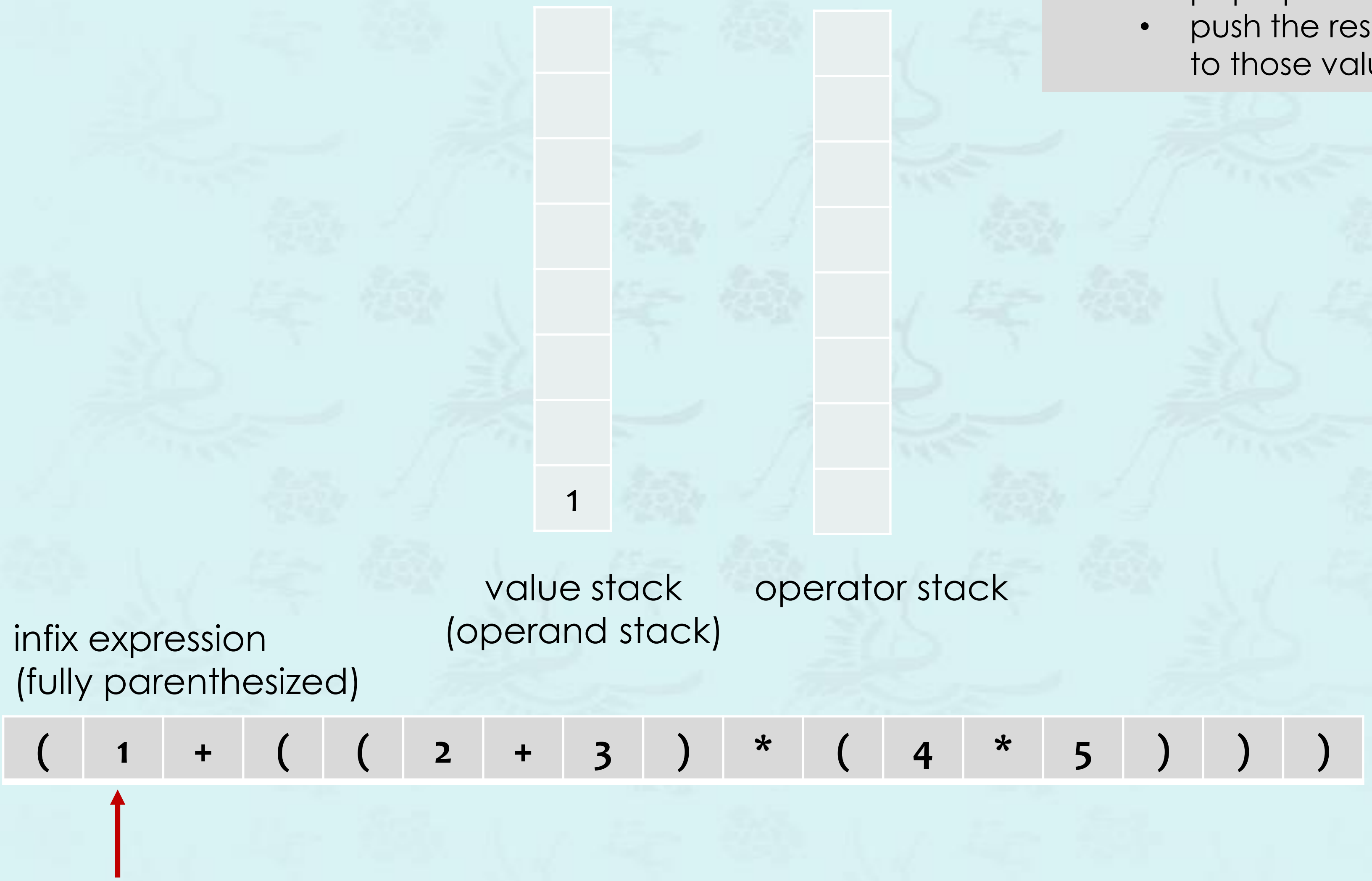
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm - Evaluate infix expressions.

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



value stack
(operand stack)

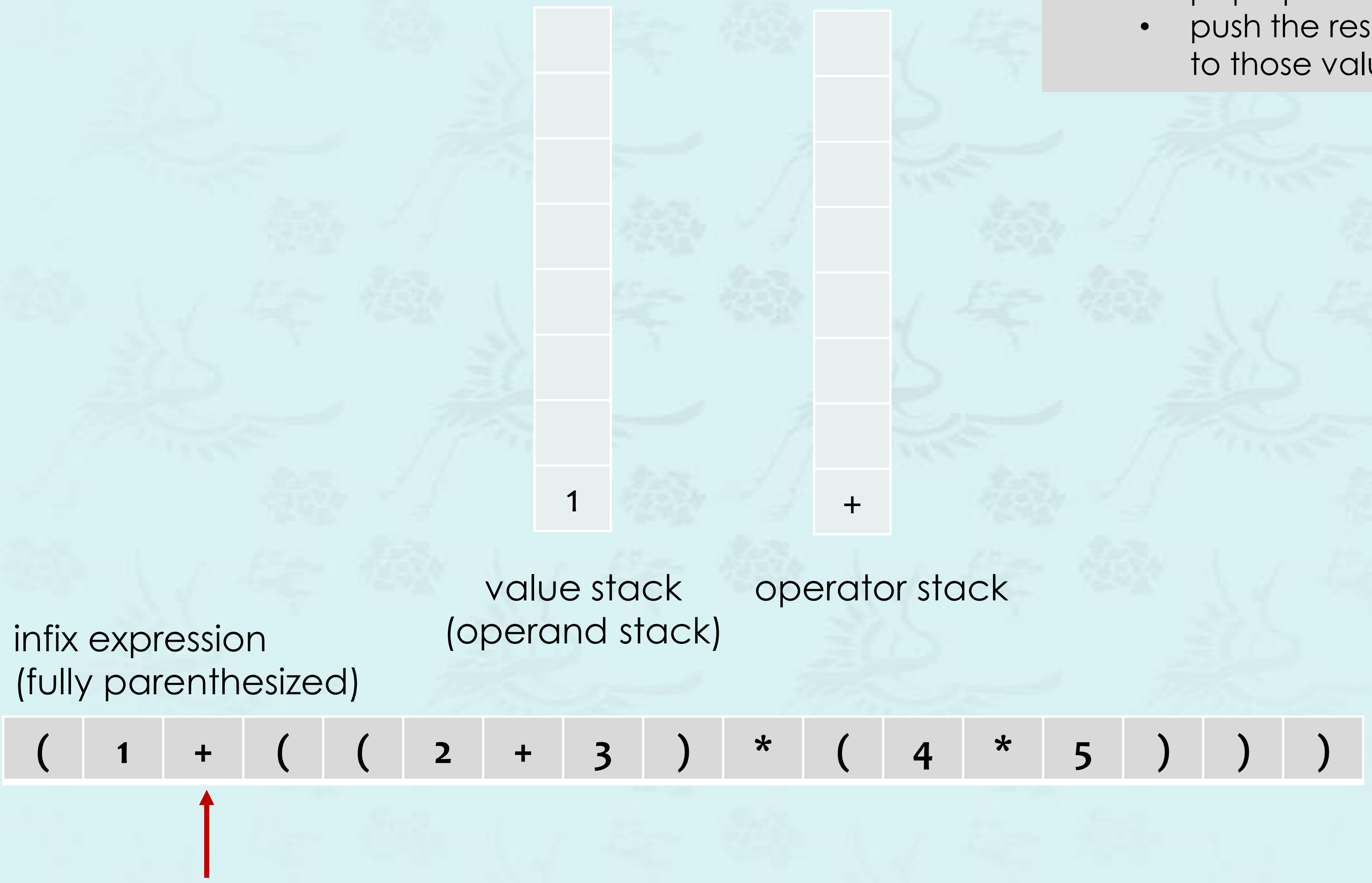
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm - Evaluate infix expressions.

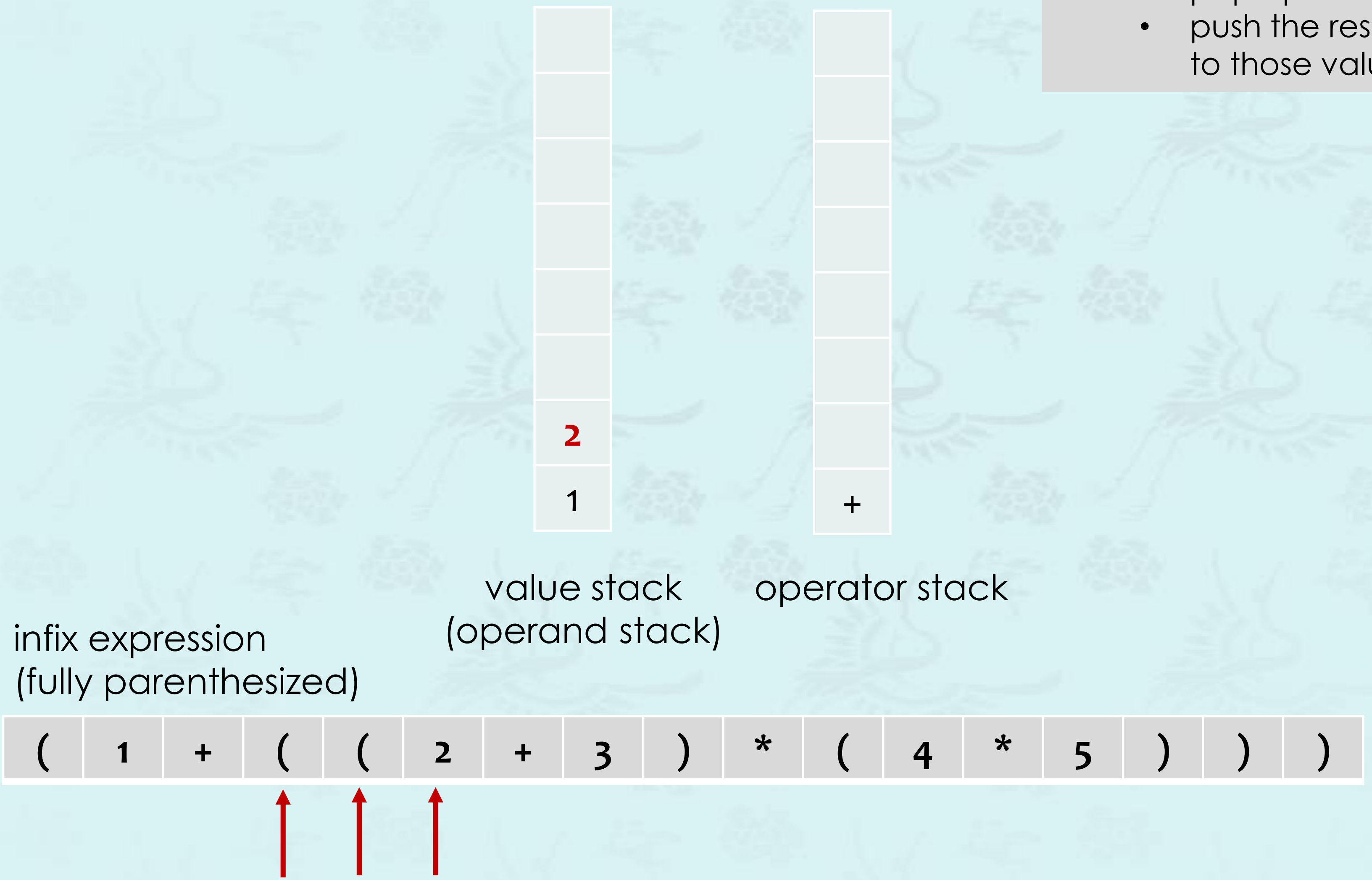
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



2
1

+

value stack
(operand stack)

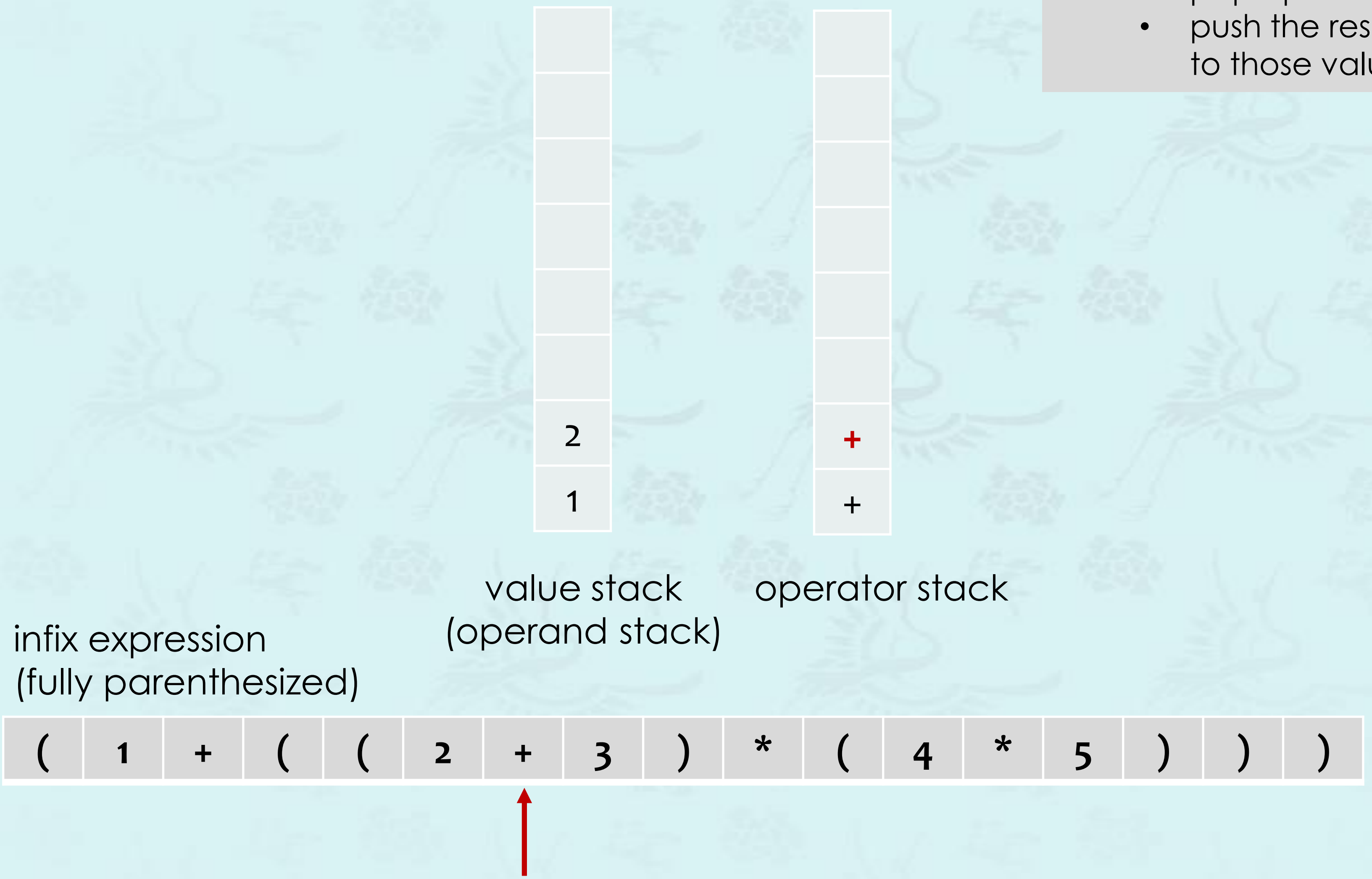
operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



value stack (operand stack)

2

1

operator stack

+

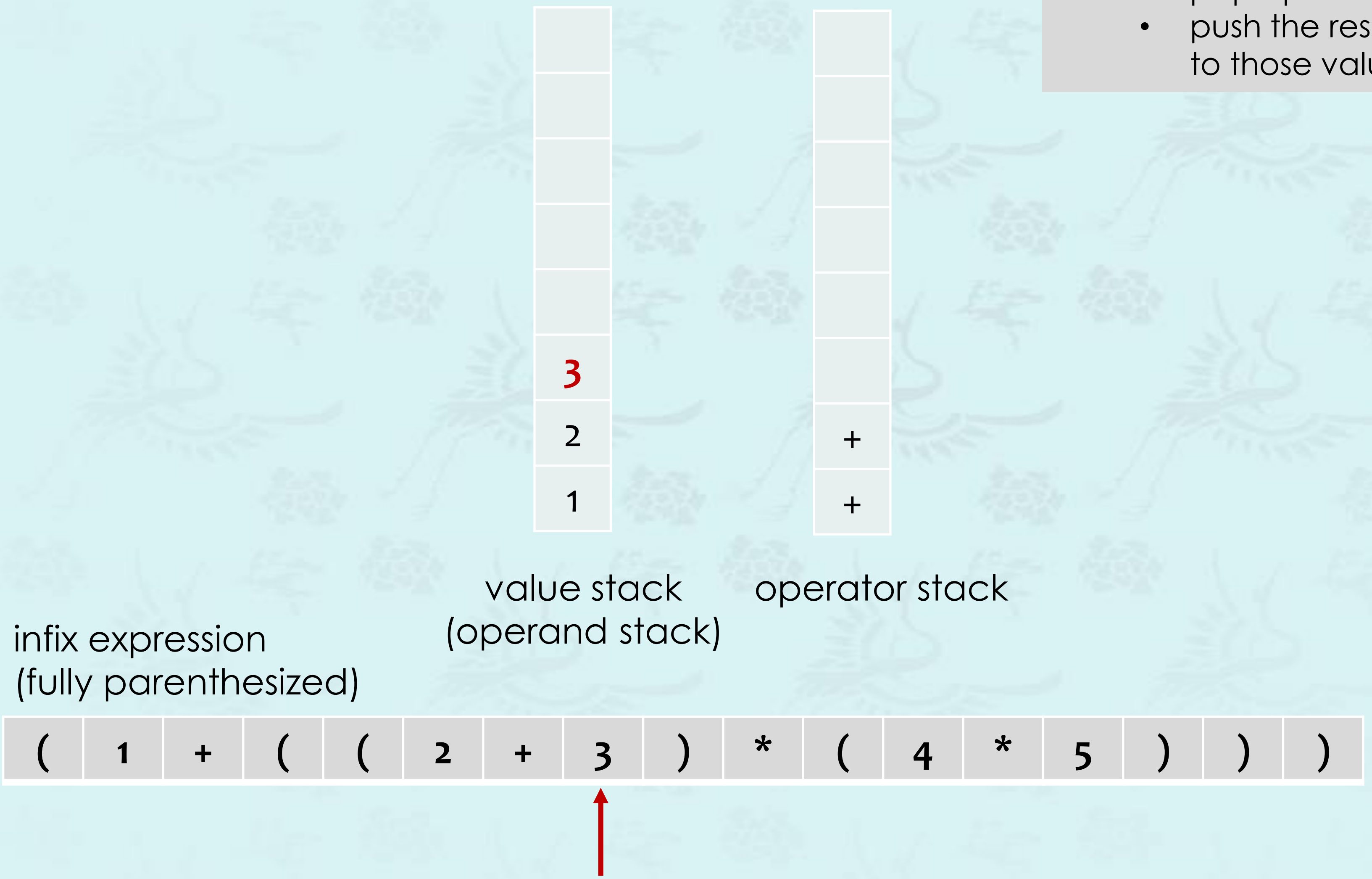
+

infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



value stack (operand stack)
3
2
1

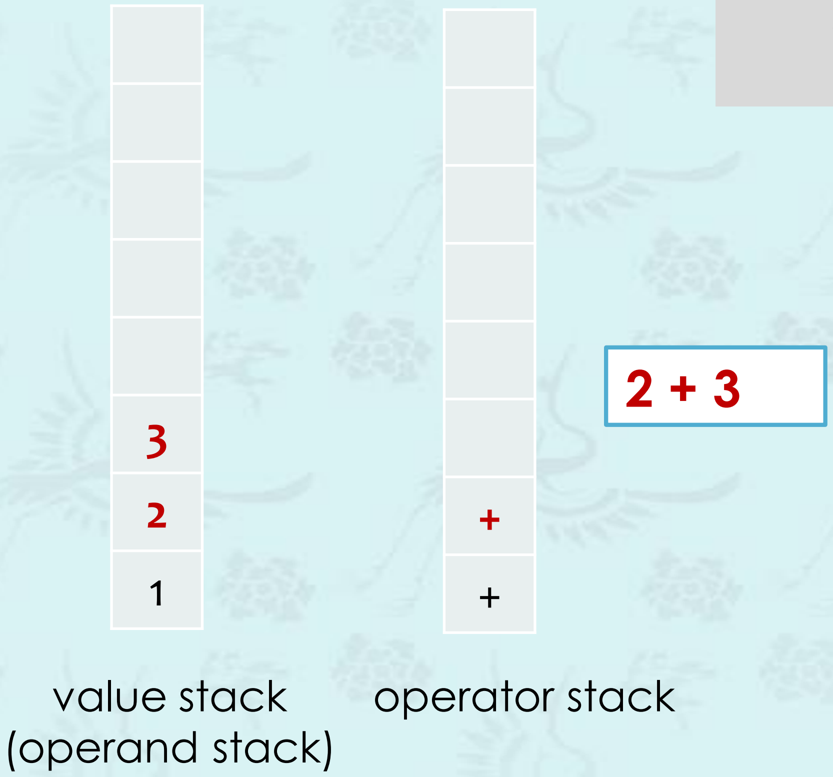
operator stack
+
+

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



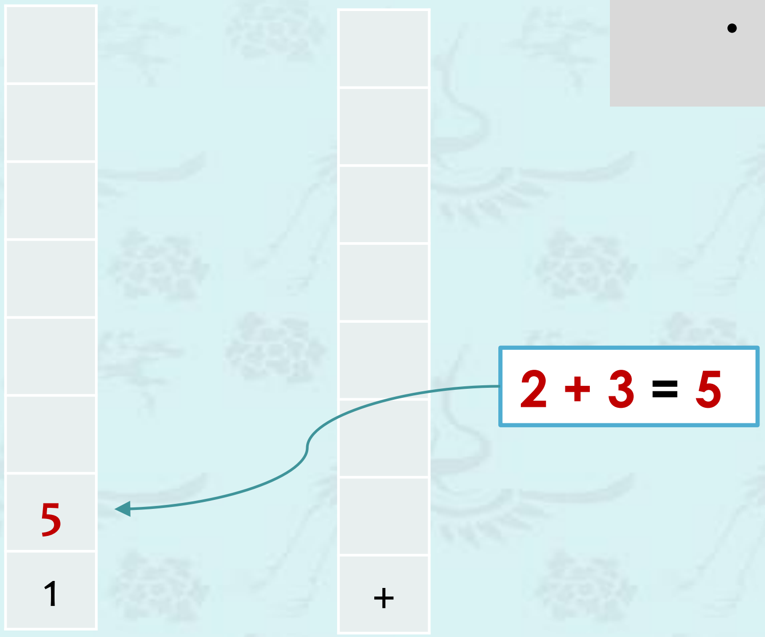
infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

- pop operator & two values
- push the result

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



value stack (operand stack)

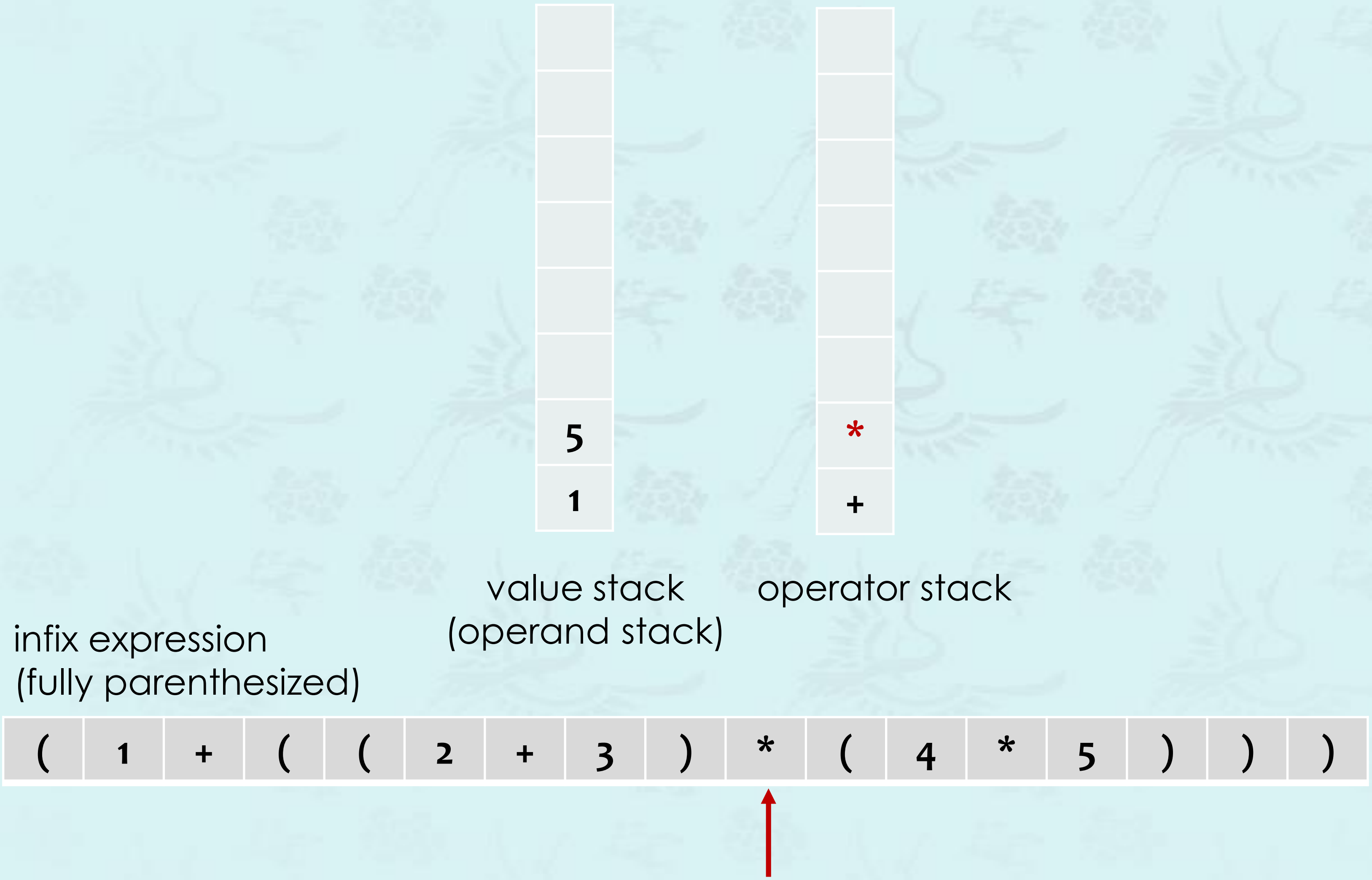
operator stack

infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

- pop operator & two values
- push the result

Dijkstra’s two-stack algorithm - Evaluate infix expressions.



5
1

value stack
(operand stack)

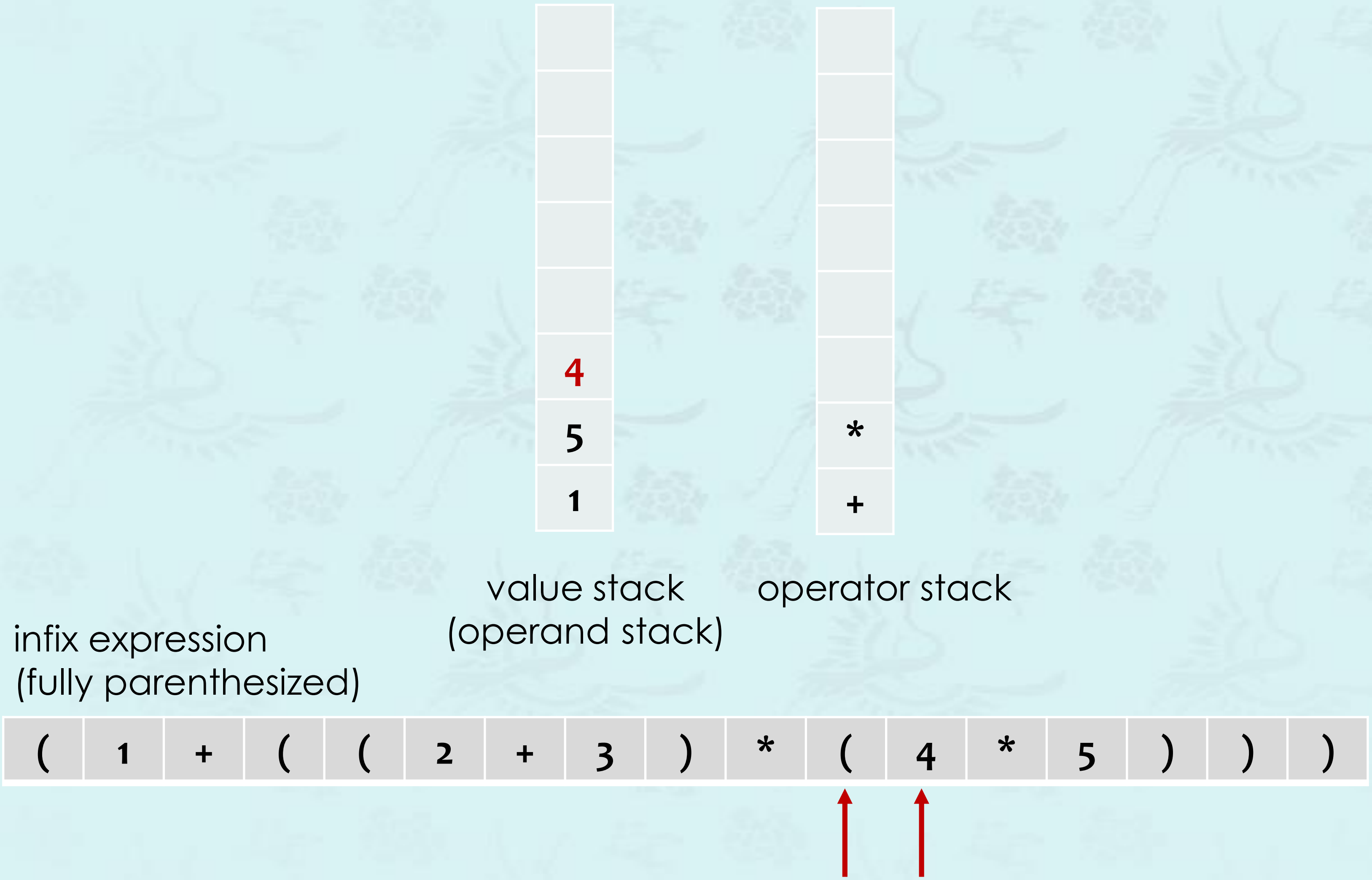
*
+

operator stack

infix expression
(fully parenthesized)

(1 + ((2 + 3) * (4 * 5)))

Dijkstra’s two-stack algorithm - Evaluate infix expressions.



4

5

1

value stack
(operand stack)

*

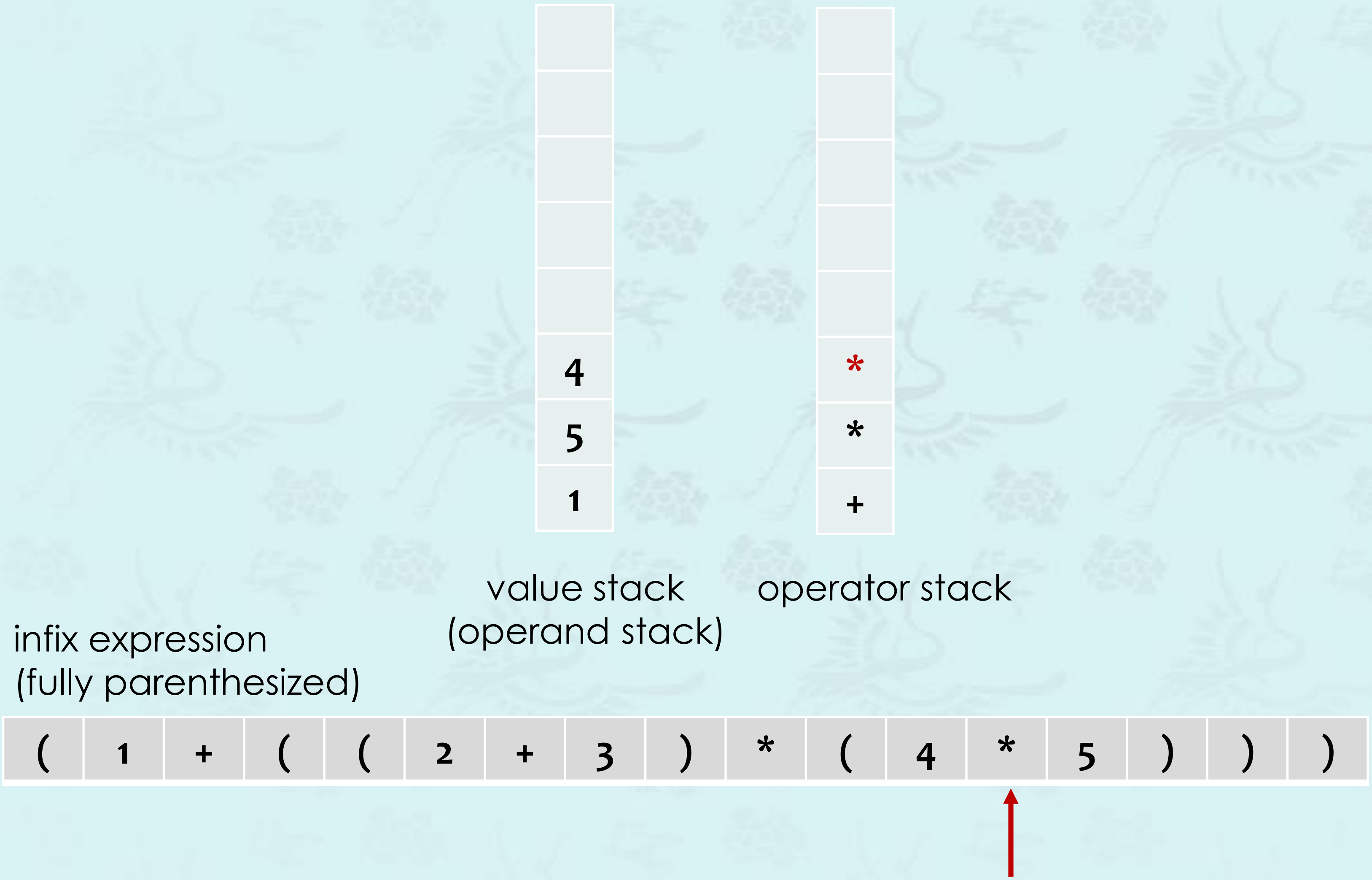
+

operator stack

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.



value stack (operand stack):

4
5
1

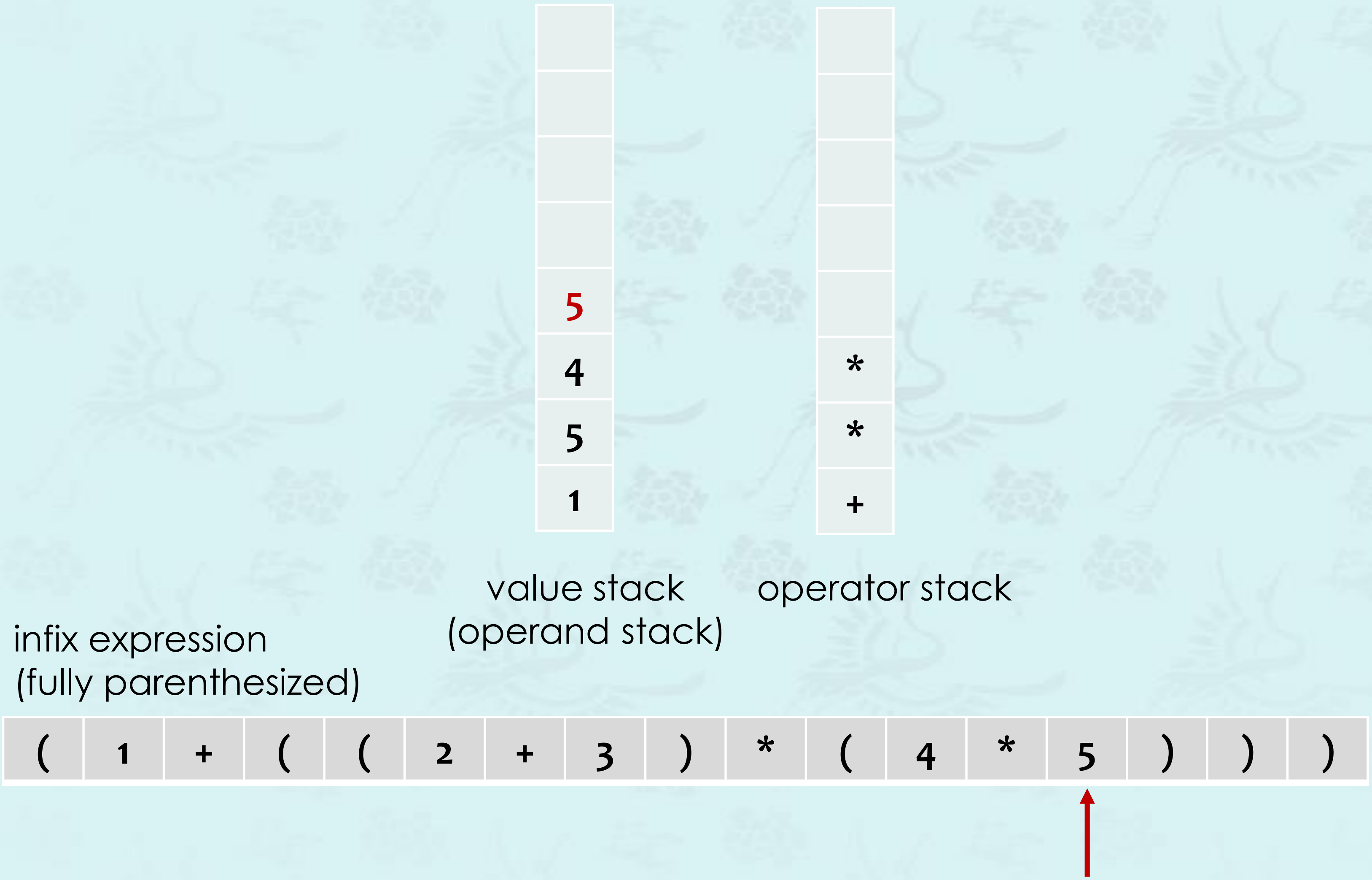
operator stack:

*
*
+

infix expression
(fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.



5
4
5
1

value stack
(operand stack)

*
*
+

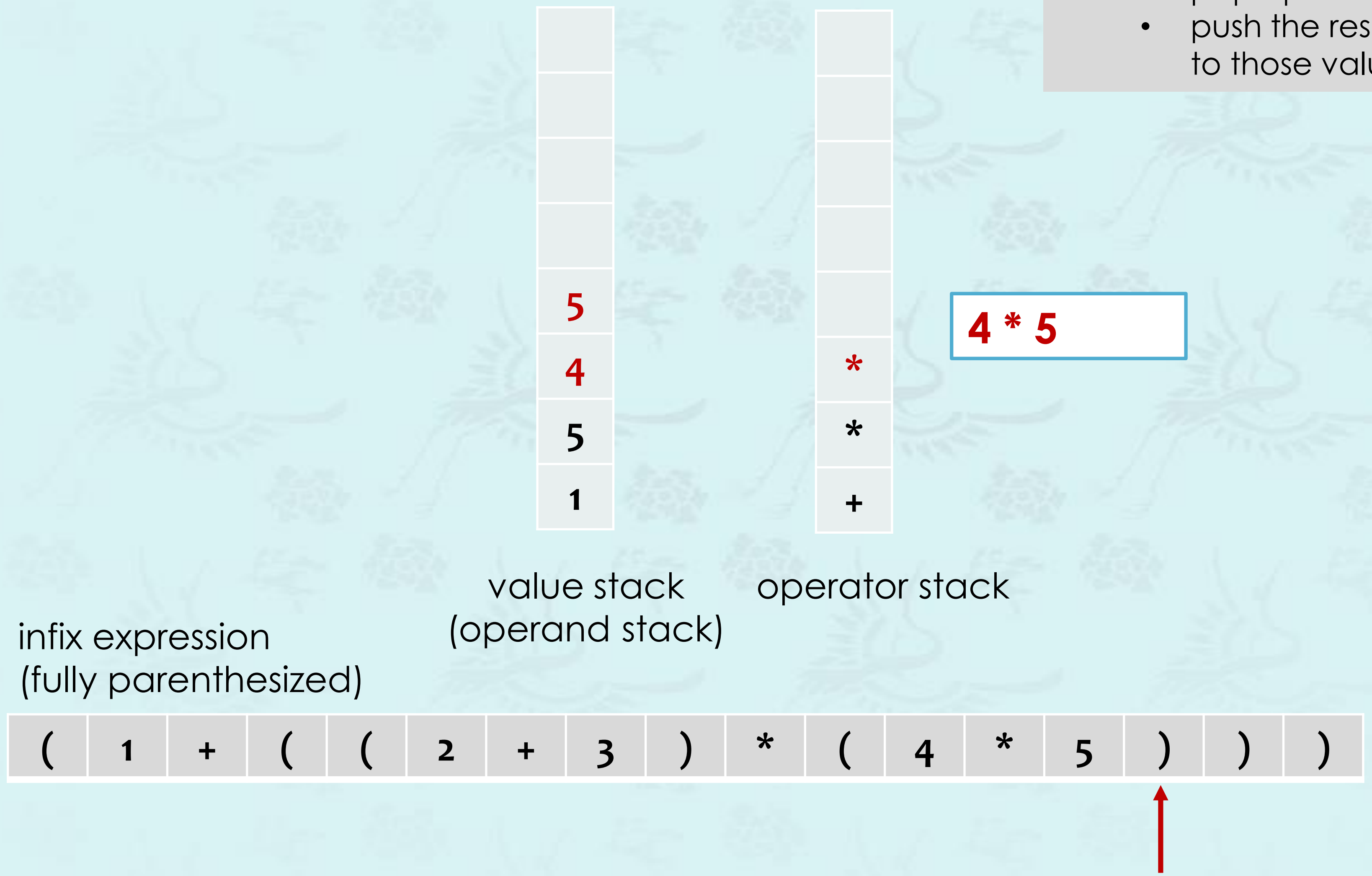
operator stack

infix expression
(fully parenthesized)

(1 + ((2 + 3) * (4 * 5)))

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

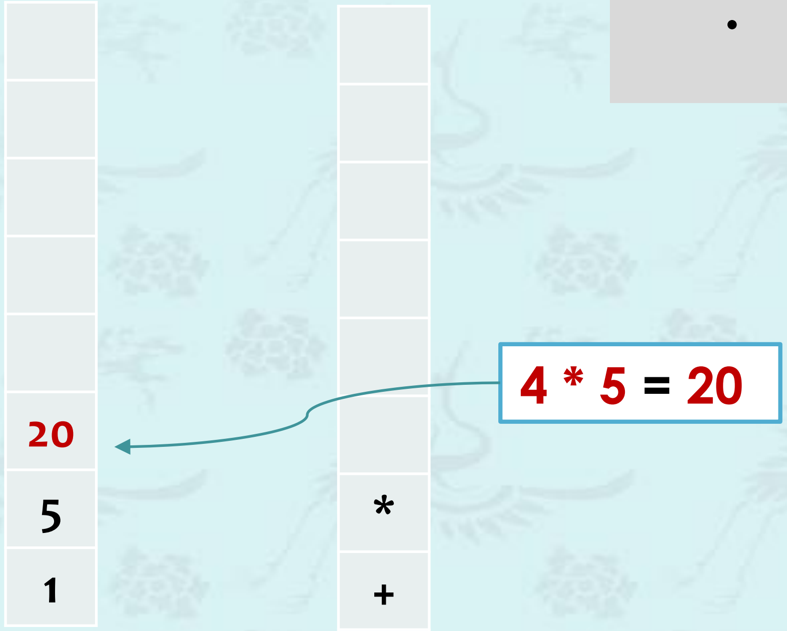
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



value stack (operand stack)

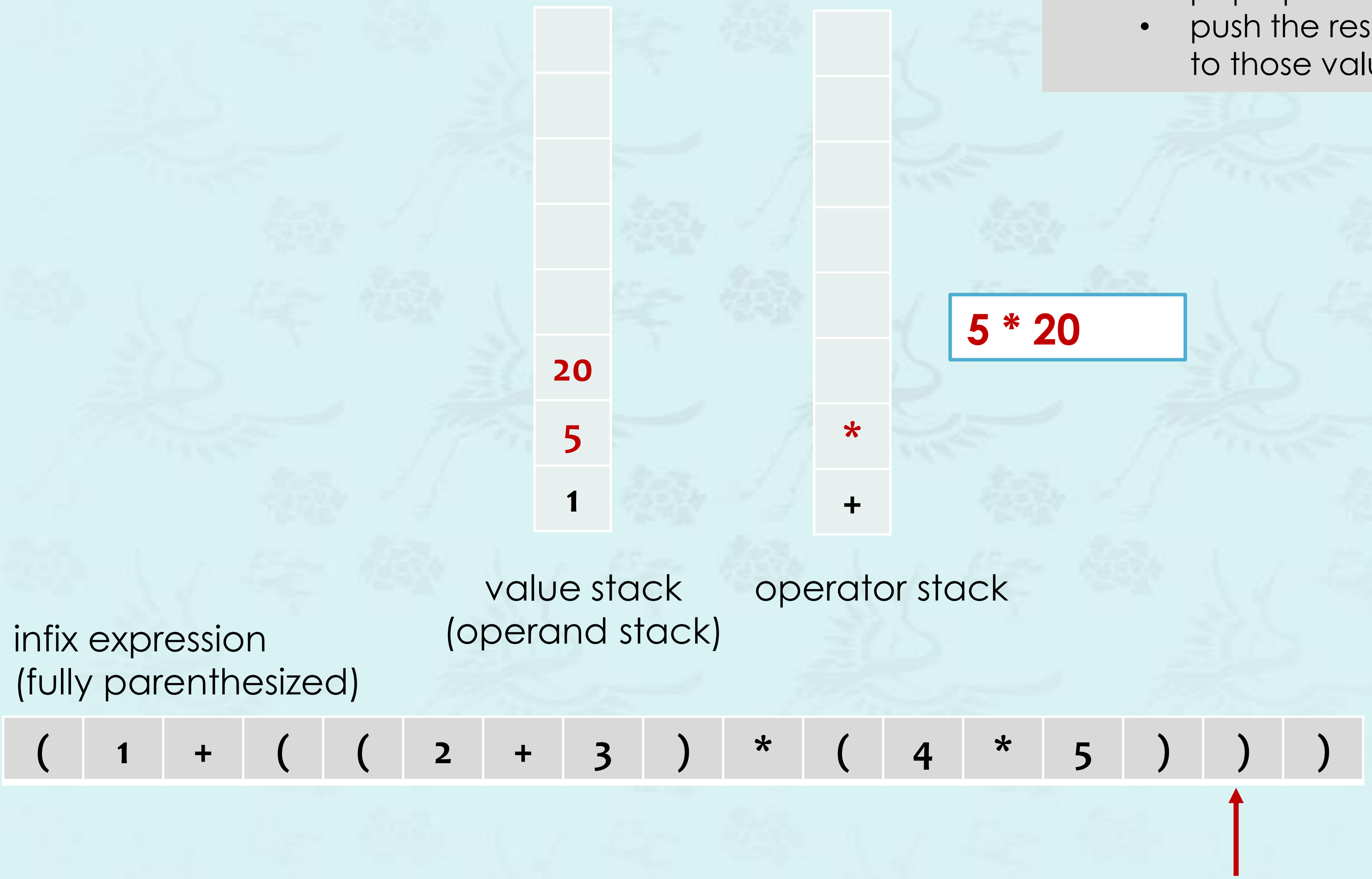
operator stack

infix expression (fully parenthesized)

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



20
5
1

value stack
(operand stack)

*
+

operator stack

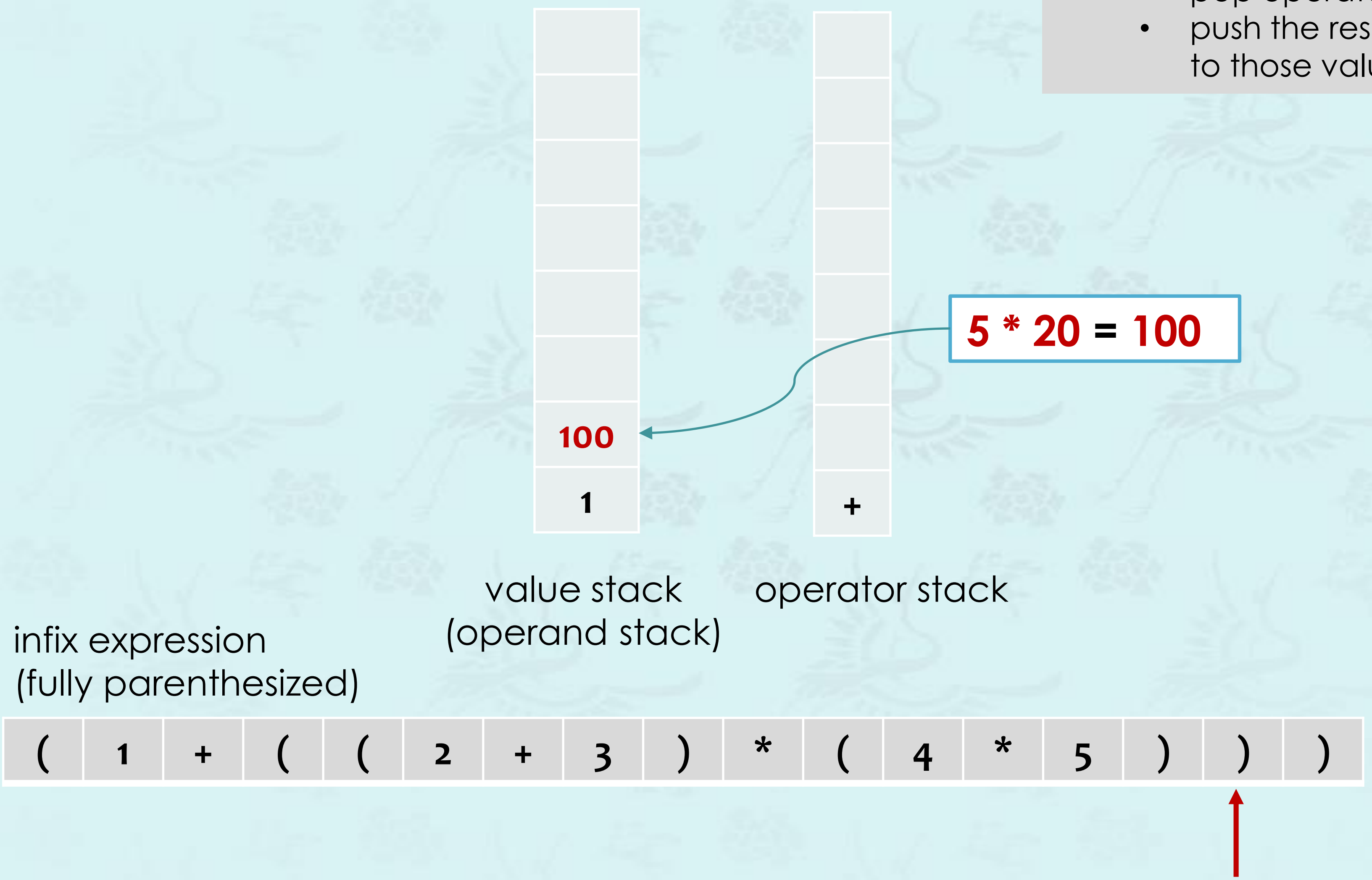
5 * 20

infix expression
(fully parenthesized)

(1 + ((2 + 3) * (4 * 5)))

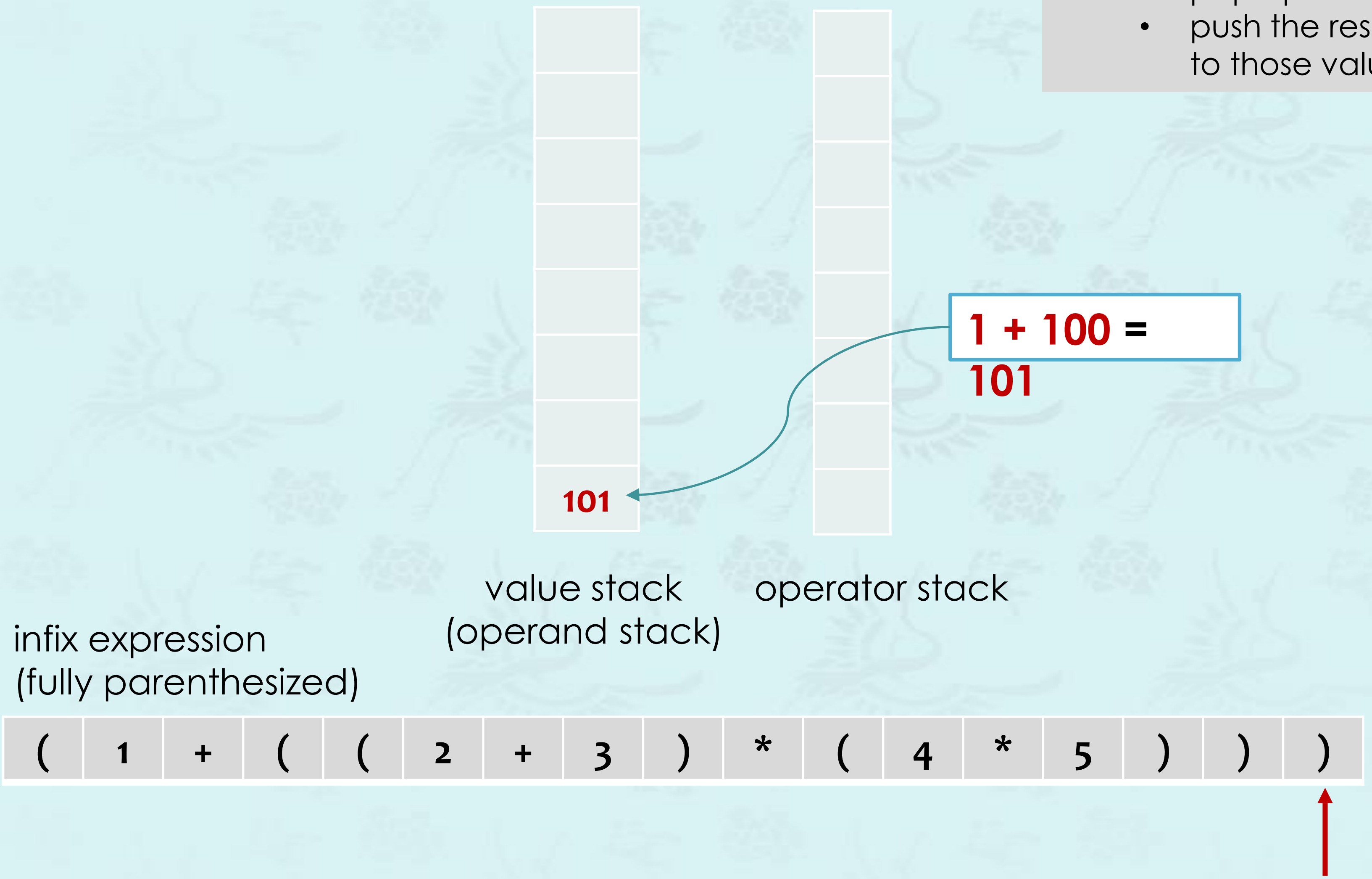
Dijkstra's two-stack algorithm - Evaluate infix expressions.

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



Dijkstra’s two-stack algorithm - Evaluate infix expressions.

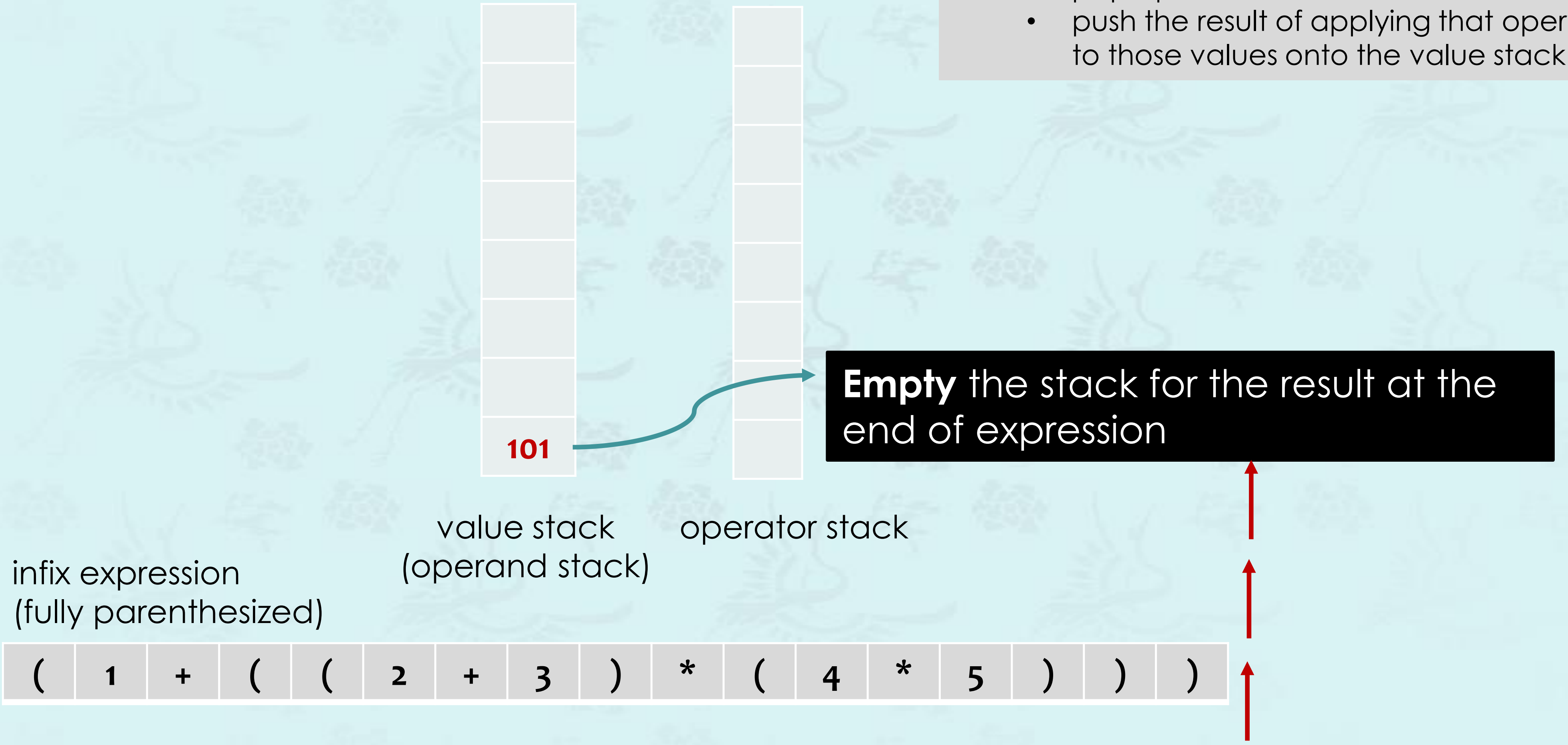
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

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



Empty the stack for the result at the end of expression

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

Dijkstra's two-stack algorithm - Evaluate infix expressions.

- Q: How does it work
- A: When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

- Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

- Extensions: More ops, precedence order, associativity.

Dijkstra’s two-stack algorithm - Evaluate infix expressions.

- **Observation 1.** Dijkstra’s two-stack algorithm computes the same value if the operator occurs **after** the two values.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

- **Observation 2.** All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

- Bottom line: Postfix or “reverse Polish” notation.
- Applications: Postscript, calculators, JVM, ….

Dijkstra's two-stack algorithm – version.1 (fully parenthesized)

```
1     While there are still tokens to be read in,
      1.1    Get the next token.
      1.2    If the token is:
             1.2.1 A space: ignore it
             1.2.2 A left brace: ignore it
             1.2.3 A number:
                   1.2.3.1 read the number (it could be a multiple digit.)
                   1.2.3.2 push it onto the value stack
             1.2.4 A right parenthesis:
                   1.2.4.1 Pop the operator from the operator stack.
                   1.2.4.2 Pop the value stack twice, getting two operands.
                   1.2.4.3 Apply the operator to the operands, in the correct order.
                   1.2.4.4 Push the result onto the value stack.
             1.2.5 An operator
                   1.2.5.1 Push the operator to the operator stack
2     (The whole expression has been parsed at this point.
      Apply remaining operators in the op stack to remaining values in the value stack)
      While the operator stack is not empty,
      2.1    Pop the operator from the operator stack.
      2.2    Pop the value stack twice, getting two operands.
      2.3    Apply the operator to the operands, in the correct order.
      2.4    Push the result onto the value stack.
3     (At this point the operator stack should be empty, and the value stack should have only
      one value in it, which is the result.)
      Return the top item in the value stack.
```

Dijkstra’s two-stack algorithm – version.1 (fully parenthesized)

```
int evaluate(string tokens) {
    stack<int>  va_stack;                 // stack to store operands or values
    stack<char> op_stack;                 // stack to store operators.

    for (int i = 0; i < tokens.length(); i++) {
        if (tokens[i] == ' ') continue;  // skip if token is whitespace or (
        if (tokens[i] == '(') continue;
        if (isdigit(tokens[i])) {
            // if token is a value, push it to va_stack
            // add the code for multi-digits value(operand)
        }
        else if (tokens[i] == ')') {  // closing brace encountered
              // compute it and push the result to the value stack.
        }
        else {  // current token is an operator;
                // push it to the operator stack.
        }
    } // Parsing is over

    while (!op_stack.empty()) {
         // apply remaining op_stack to remaining va_stack.
    }
    return // va_stack top contains the result, return it.
}
```

Dijkstra’s two-stack algorithm – version.1 (fully parenthesized)

```
int evaluate(string tokens) {
    stack<int>  va_stack;                 // stack to store operands or values
    stack<char> op_stack;                 // stack to store operators.

    for (int i = 0; i < tokens.length(); i++) {
        if (tokens[i] == ' ') continue;  // skip if token is whitespace or (
        if (tokens[i] == '(') continue;
        if (isdigit(tokens[i])) {
            // if token is a value, push it to va_stack
            // add the code for multi-digits value(operand)
        }
        else if (tokens[i] == ')') {  // closing brace encountered
              // compute it and push the result to the value stack.
        }
        else {  // current token is an operator;
                // push it to the operator stac
        }
    } // Parsing is over

    while (!op_stack.empty()) {
          // apply remaining op_stack to remainir
    }
    return // va_stack top contains the result, return it.
}
```

```
int compute(stack<int>& va_stack, stack<char>& op_stack) {
  int right = va_stack.top(); va_stack.pop();
  int left  = va_stack.top(); va_stack.pop();
  char op = op_stack.top(); op_stack.pop();
  int answer = apply_op(left, right, op);
  return answer;
}
```

Dijkstra’s two-stack algorithm – version.2 (parenthesis are not required)

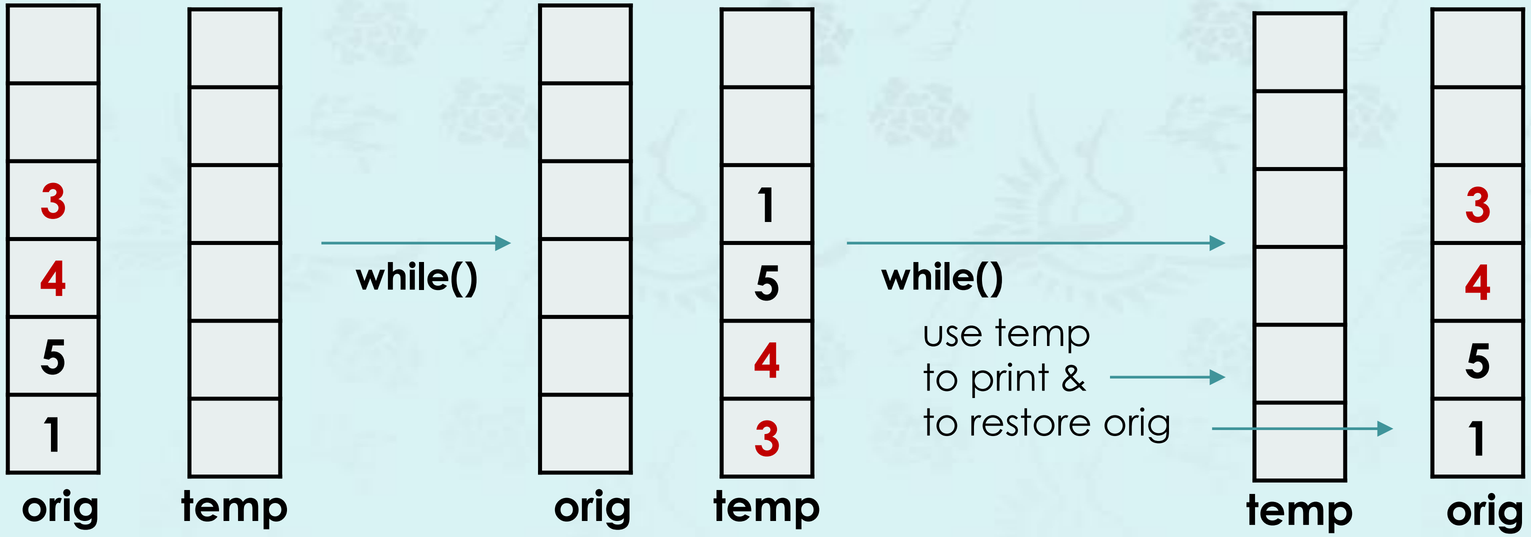
```
1     While there are still tokens to be read in,
      1.1    Get the next token.
      1.2    If the token is:
          1.2.1  A space: ignore it
          1.2.2  A left brace: push it onto the operator stack.
          1.2.3  A number:
              1.2.3.1 read the number (it could be a multiple digit.)
              1.2.3.2 push it onto the value stack
          1.2.4  A right parenthesis:
              1.2.4.1 While the item on top of the operator stack is not a left brace,
                  1.2.4.1.1       Pop the operator from the operator stack.
                  1.2.4.1.2       Pop the value stack twice, getting two operands.
                  1.2.4.1.3       Apply the operator to the operands, in the correct order.
                  1.2.4.1.4       Push the result onto the value stack.
              1.2.4.2  Pop the left brace from the operator stack and discard it.
          1.2.5  An operator (let’s call it thisOp)
              1.2.5.1 While the operator stack is not empty, and the top item on the operator
                      stack has the same or greater precedence as thisOp,
                  1.2.5.1.1       Pop the operator from the operator stack
                  1.2.5.1.2       Pop the value stack twice, getting two values
                  1.2.5.1.3       Apply the operator to two values in the correct order
                  1.2.5.1.4       Push the result on the value stack
              1.2.5.2  Push the operator (thisOp) onto the operator stack
2     (The whole expression has been parsed at this point.
      Apply remaining operators in the op stack to remaining values in the value stack)
      While the operator stack is not empty,
      2.1    Pop the operator from the operator stack.
      2.2    Pop the value stack twice, getting two values.
      2.3    Apply the operator to two values, in the correct order.
      2.4    Push the result onto the value stack.
3     (At this point the operator stack should be empty, and the value stack should have only
      one value in it, which is the result.)
      Return the top item in the value stack.
```

Dijkstra's two-stack algorithm – Implementation (iteration & template version)

- printStack() prints the contents of a stack from the bottom to top.
  - The stack contents should be the same as before after printing.

Algorithm:
- Given a stack called **orig**.
- Create an empty stack called **temp.**
- While **orig** is not empty,
  - Top/Pop push an item from **orig** to **temp**.
- While **temp** is not empty,
  - Top/Pop an item from **temp**, print it and push it **orig**.

- Use both iteration and recursion version of stack in PSet.
- Use **`stack<double>`** to handle division operation in PSet.



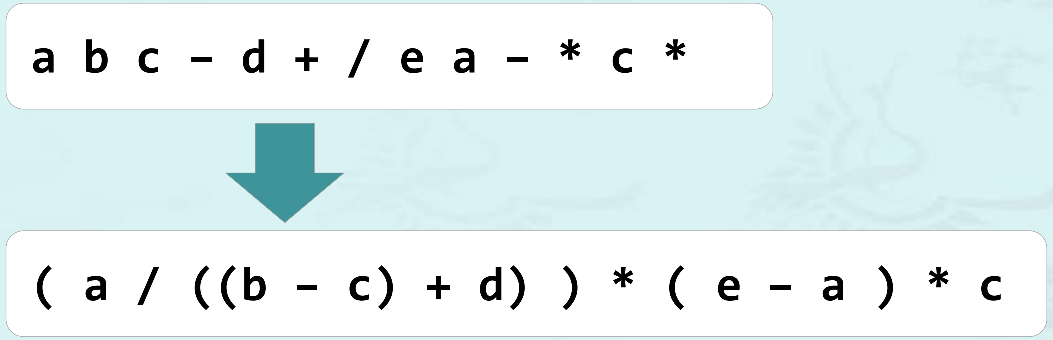
Postfix Evaluation

- **Goal**: Evaluate postfix expressions.

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
(a / (b – c + d)) * (e – a) * c	a b c – d + / e a – * c *
a / b – c + d * e – a * c	a b / c – d e * + a c * –

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



```
a b c – d + / e a – * c *
```

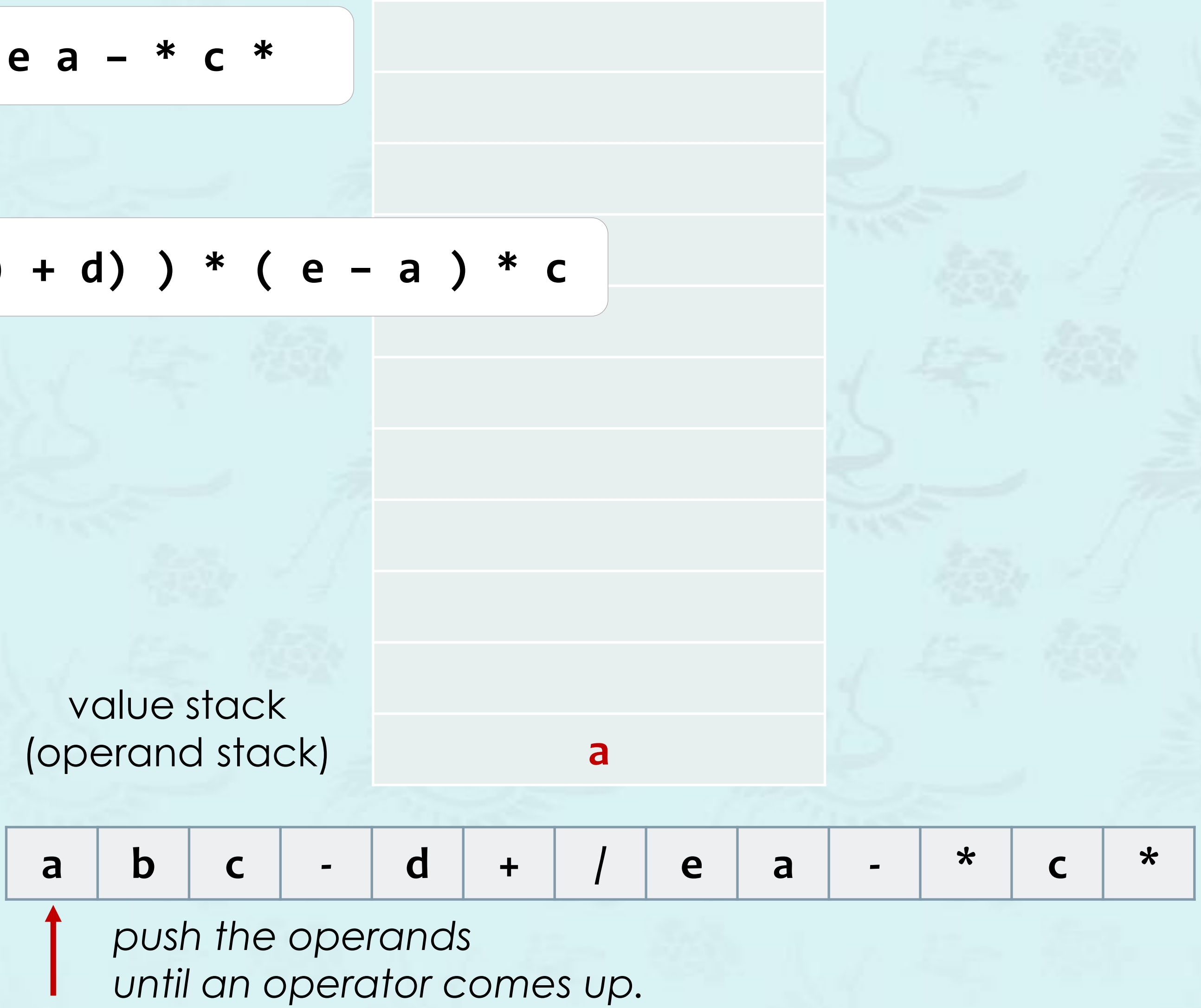
```
( a / ((b – c) + d) ) * ( e – a ) * c
```

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.

a b c – d + / e a – * c *

(a / ((b – c) + d)) * (e – a) * c



value stack
(operand stack)

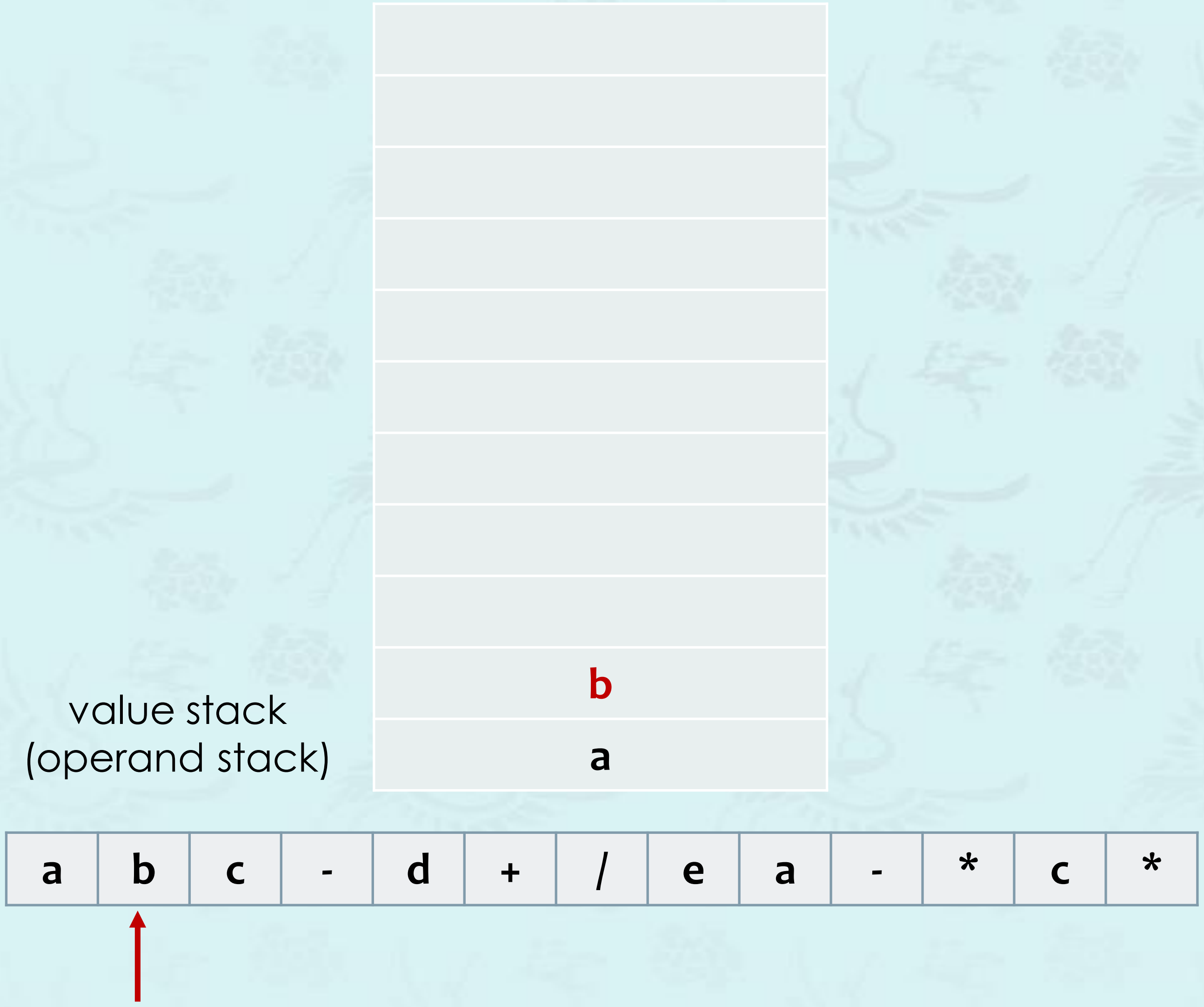
a

a	b	c	-	d	+	/	e	a	-	*	c	*

push the operands
until an operator comes up.

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

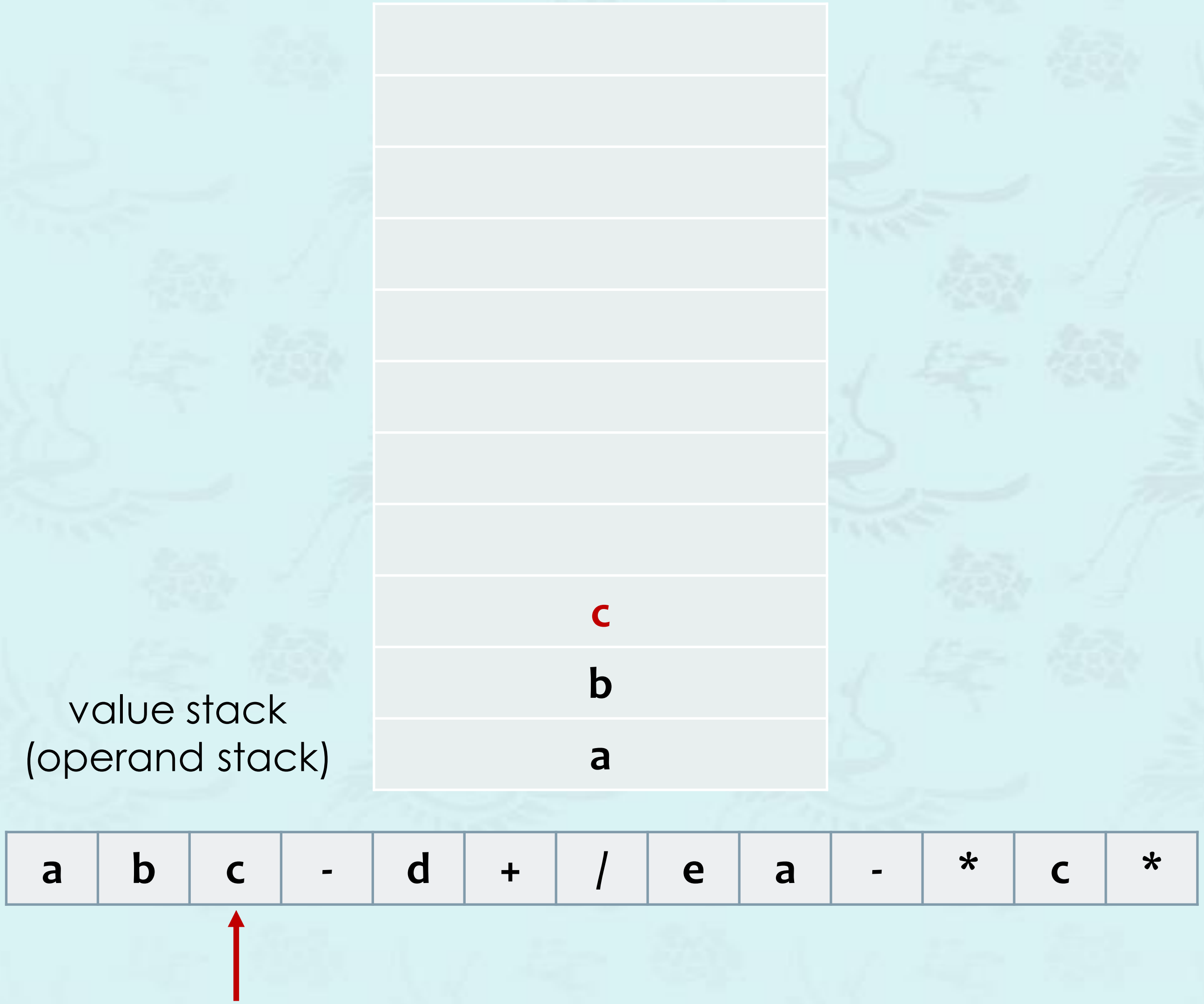
b

a

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.

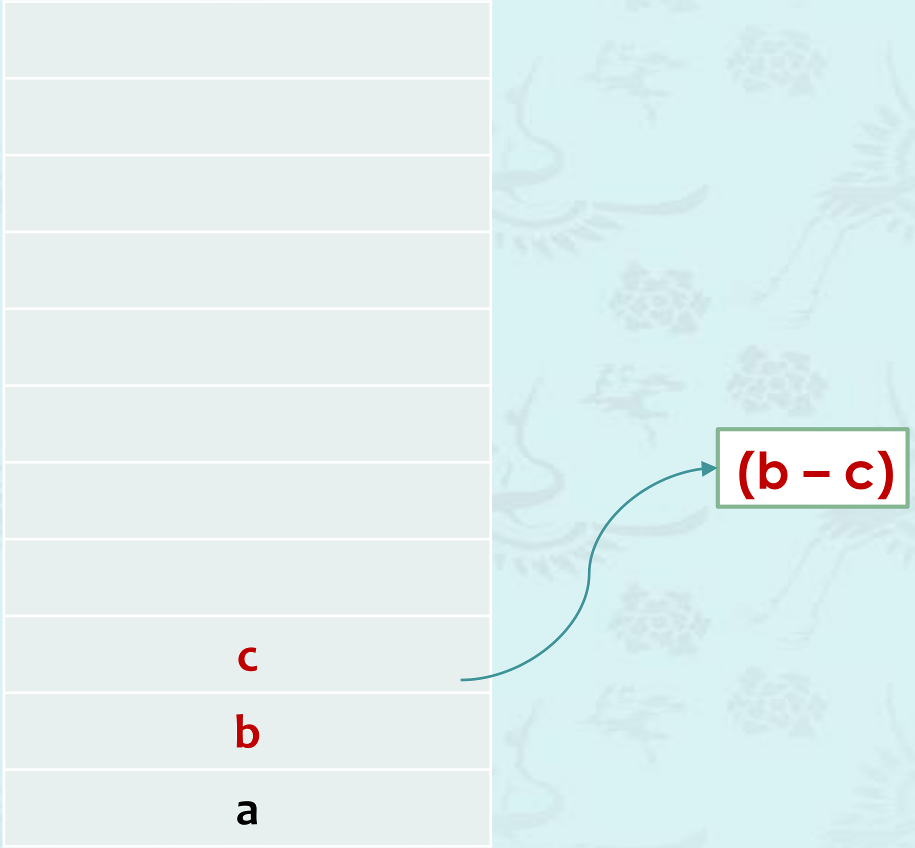


a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.

value stack
(operand stack)

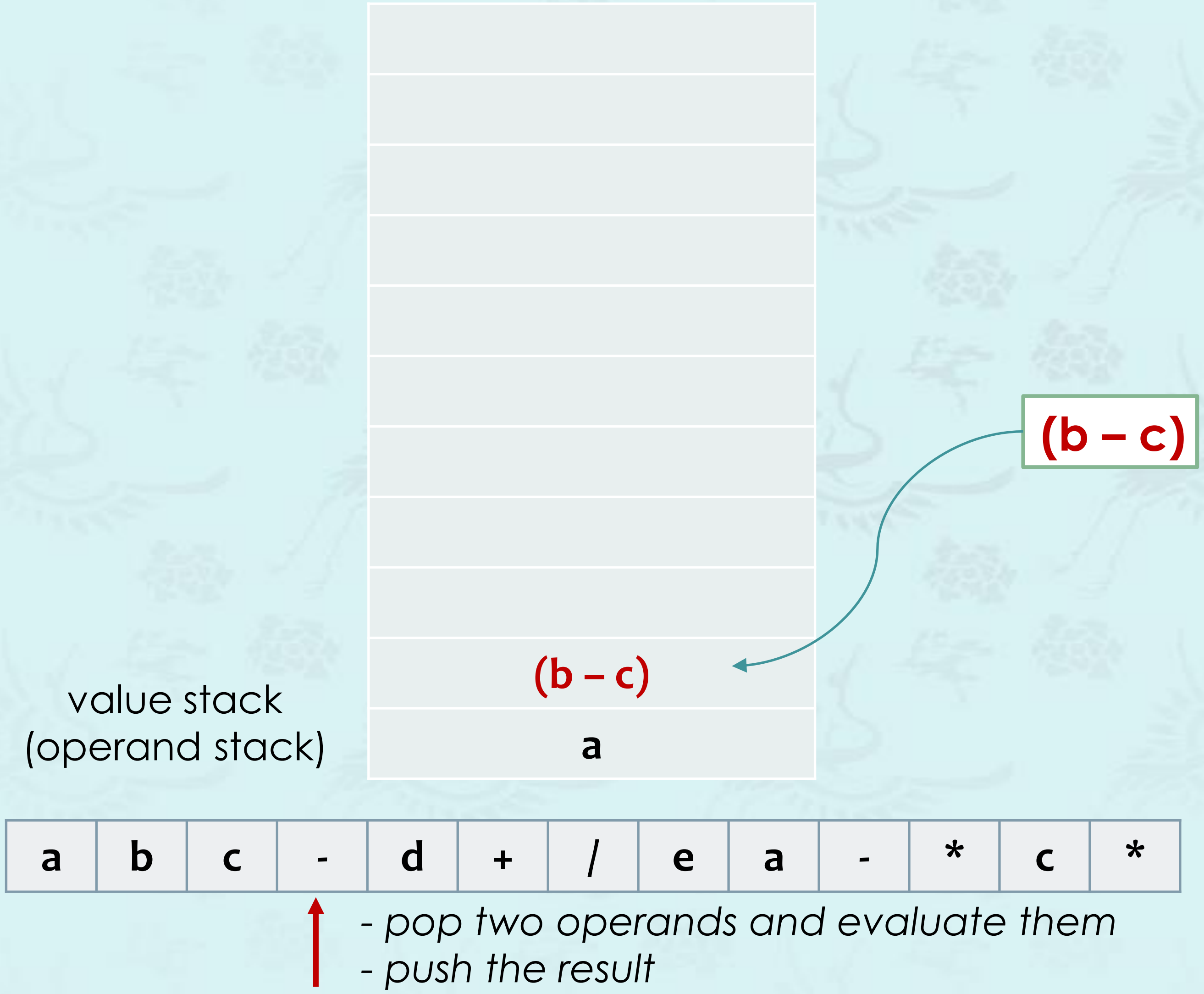


a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

(b – c)

a

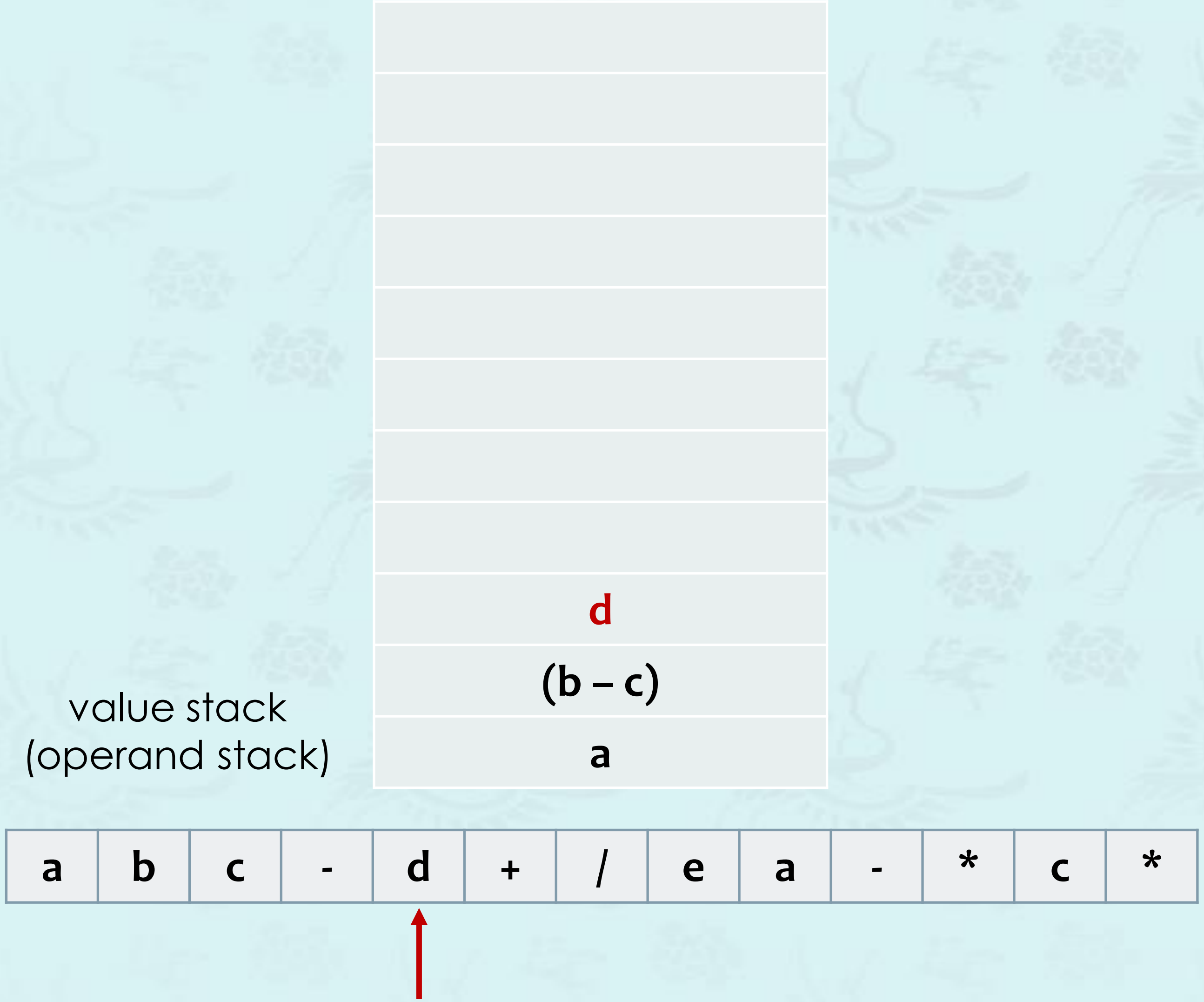
(b – c)

a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



d

(b – c)

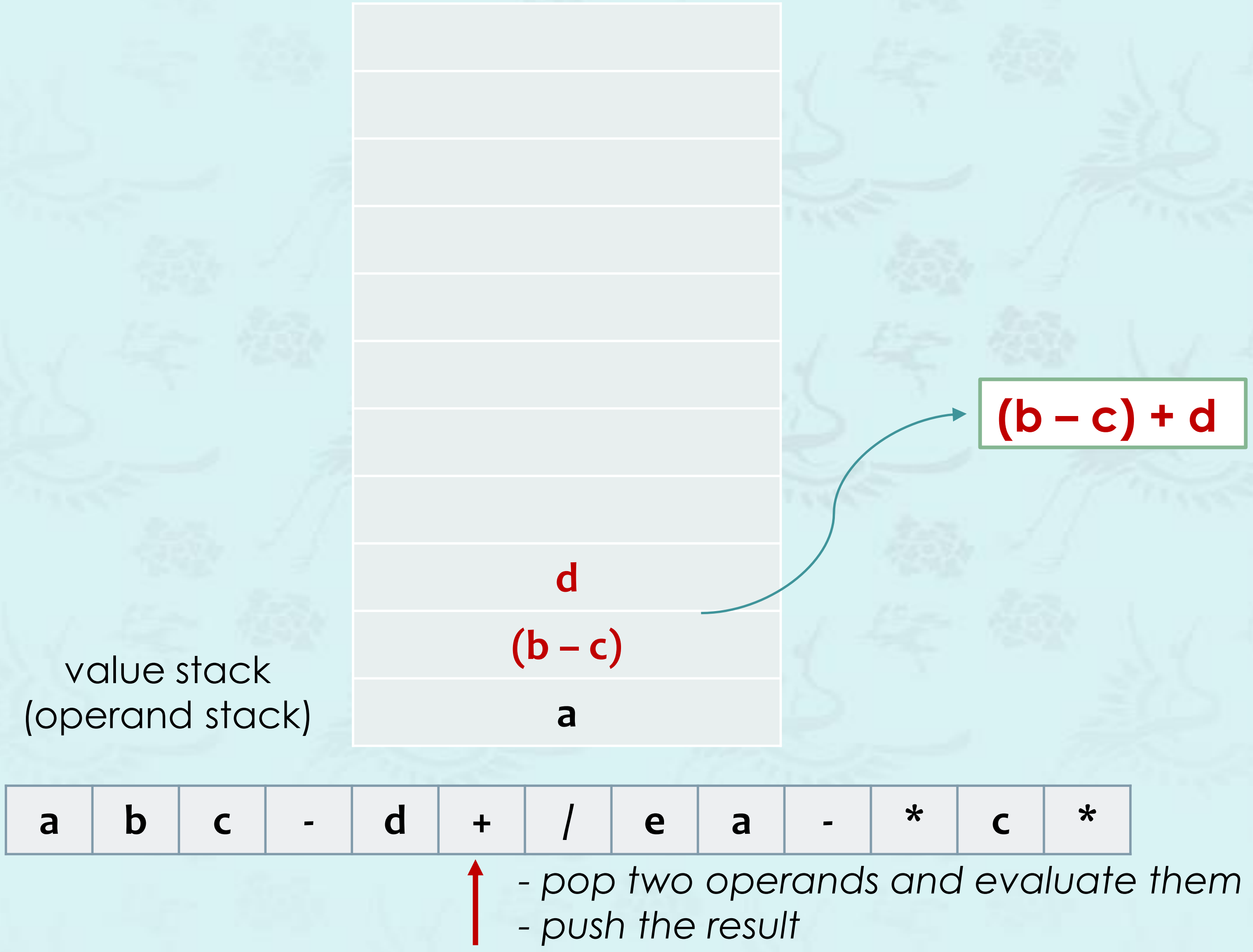
a

value stack
(operand stack)

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.

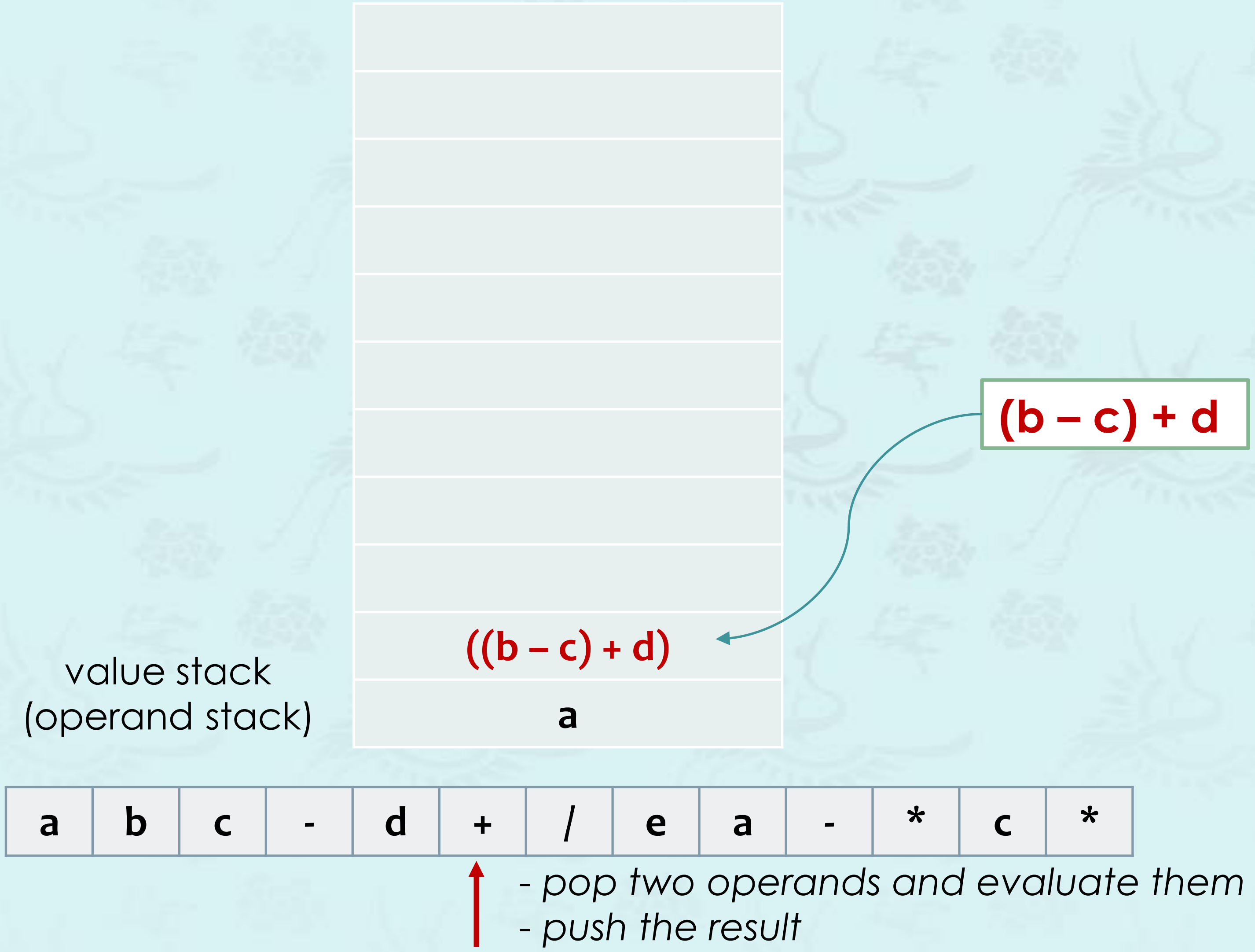


a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



(b – c) + d

((b – c) + d)

a

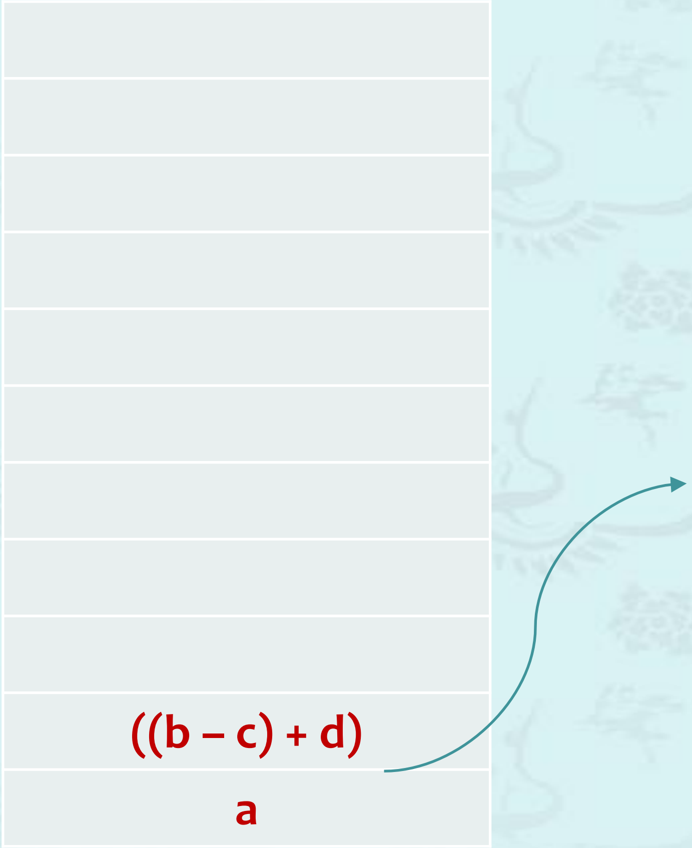
value stack
(operand stack)

a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

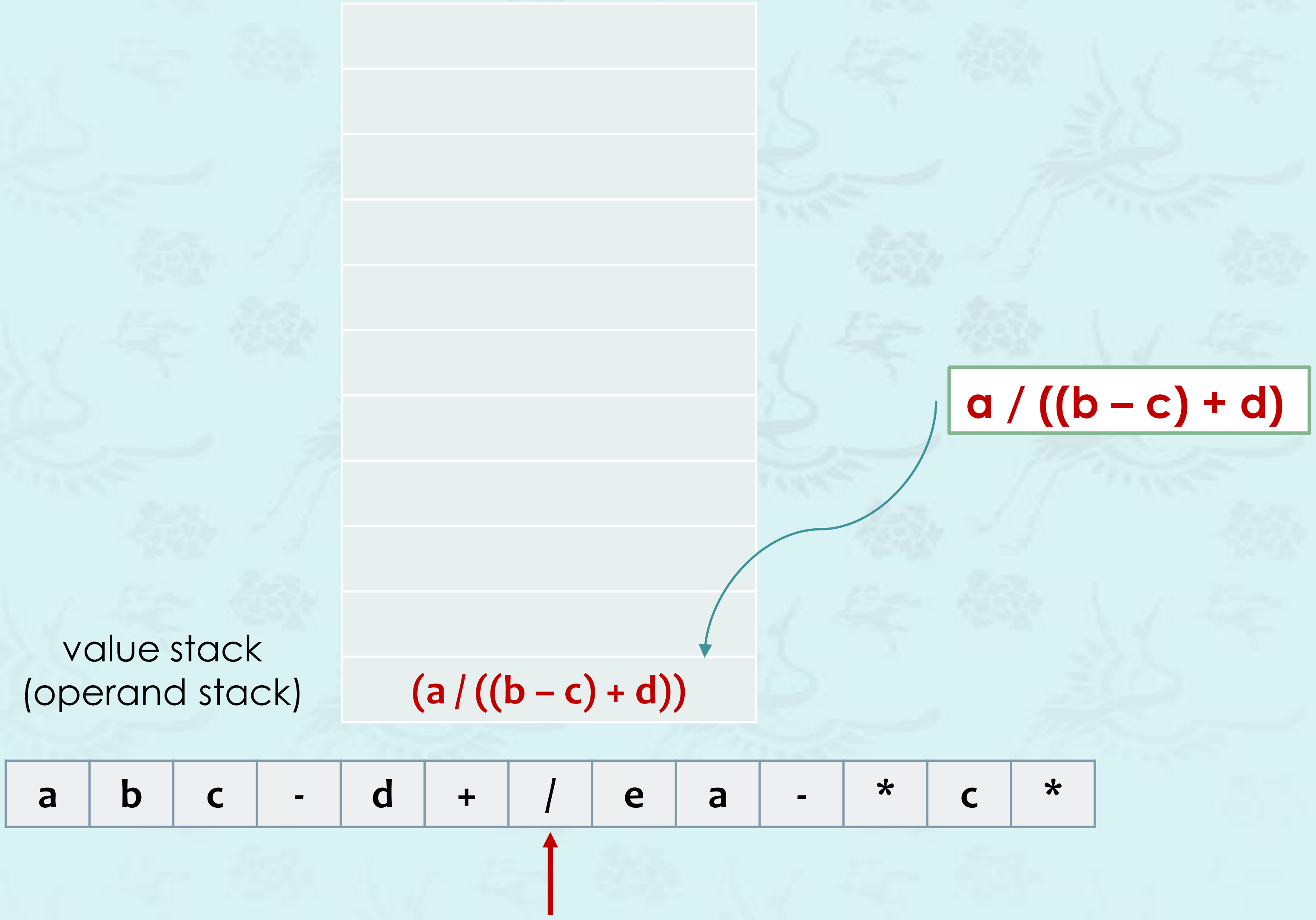
((b – c) + d)

a

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



a / ((b – c) + d)

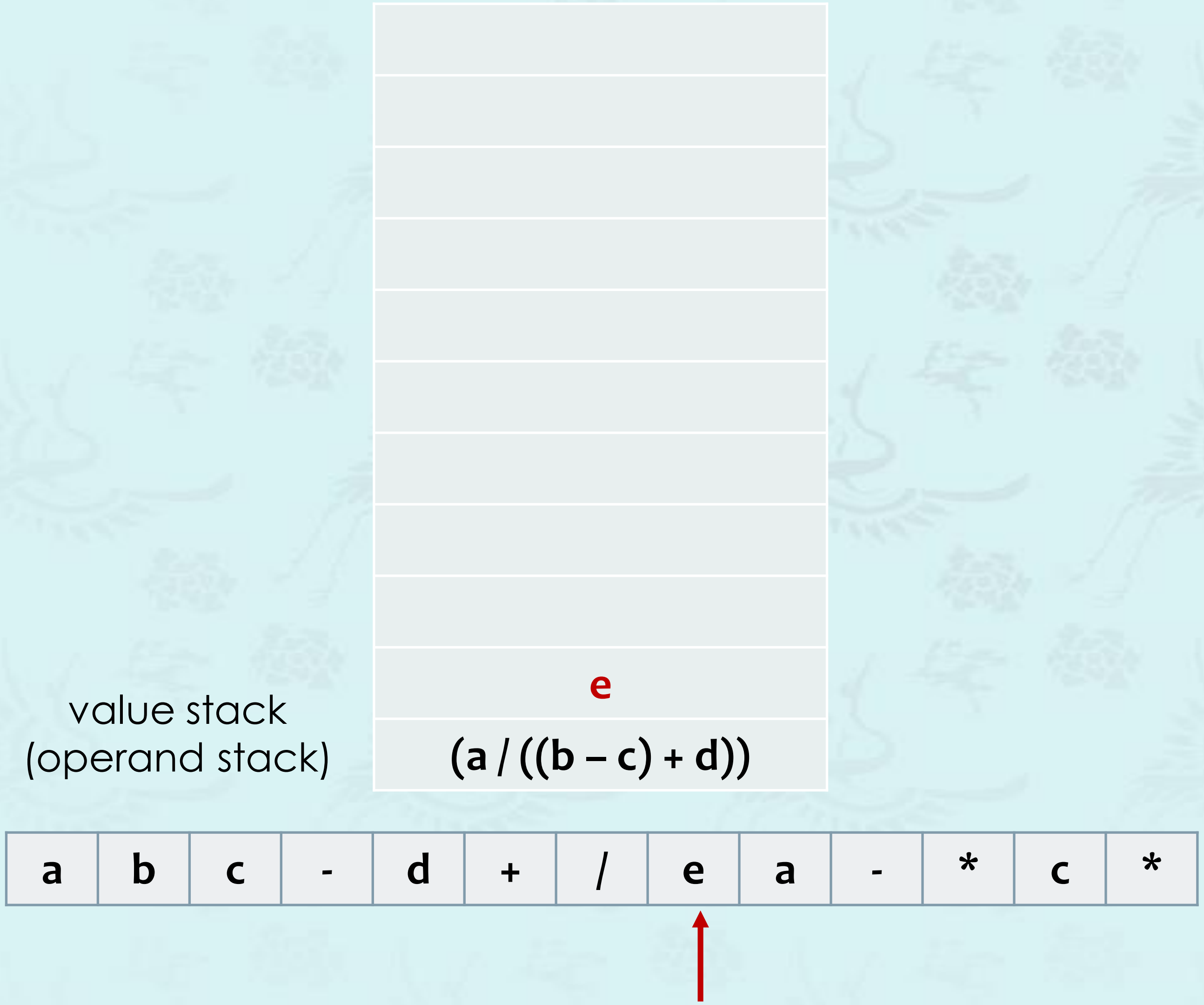
value stack
(operand stack)

(a / ((b – c) + d))

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



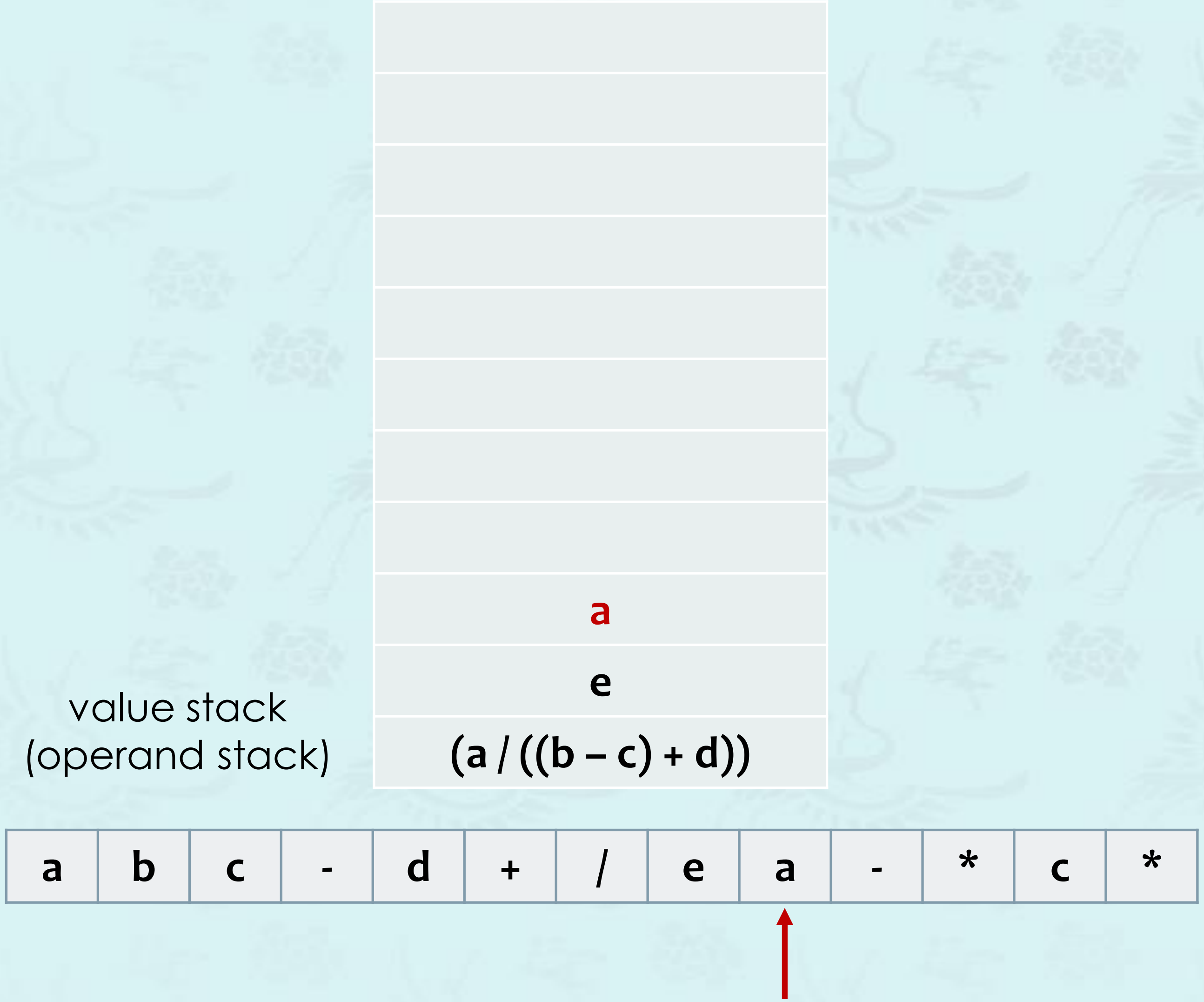
value stack
(operand stack)

e
(a / ((b – c) + d))

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

a

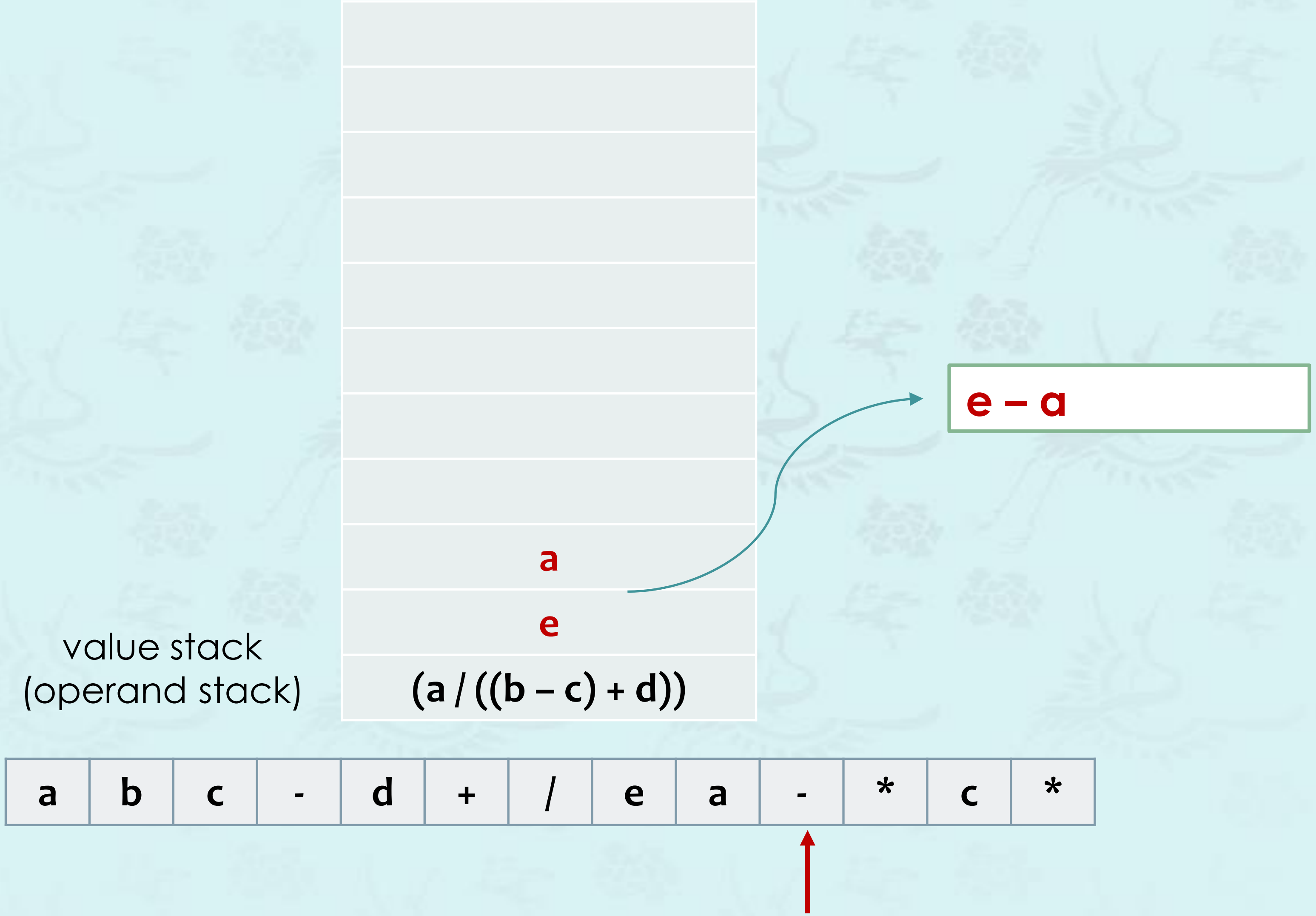
e

(a / ((b – c) + d))

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



a

e

(a / ((b – c) + d))

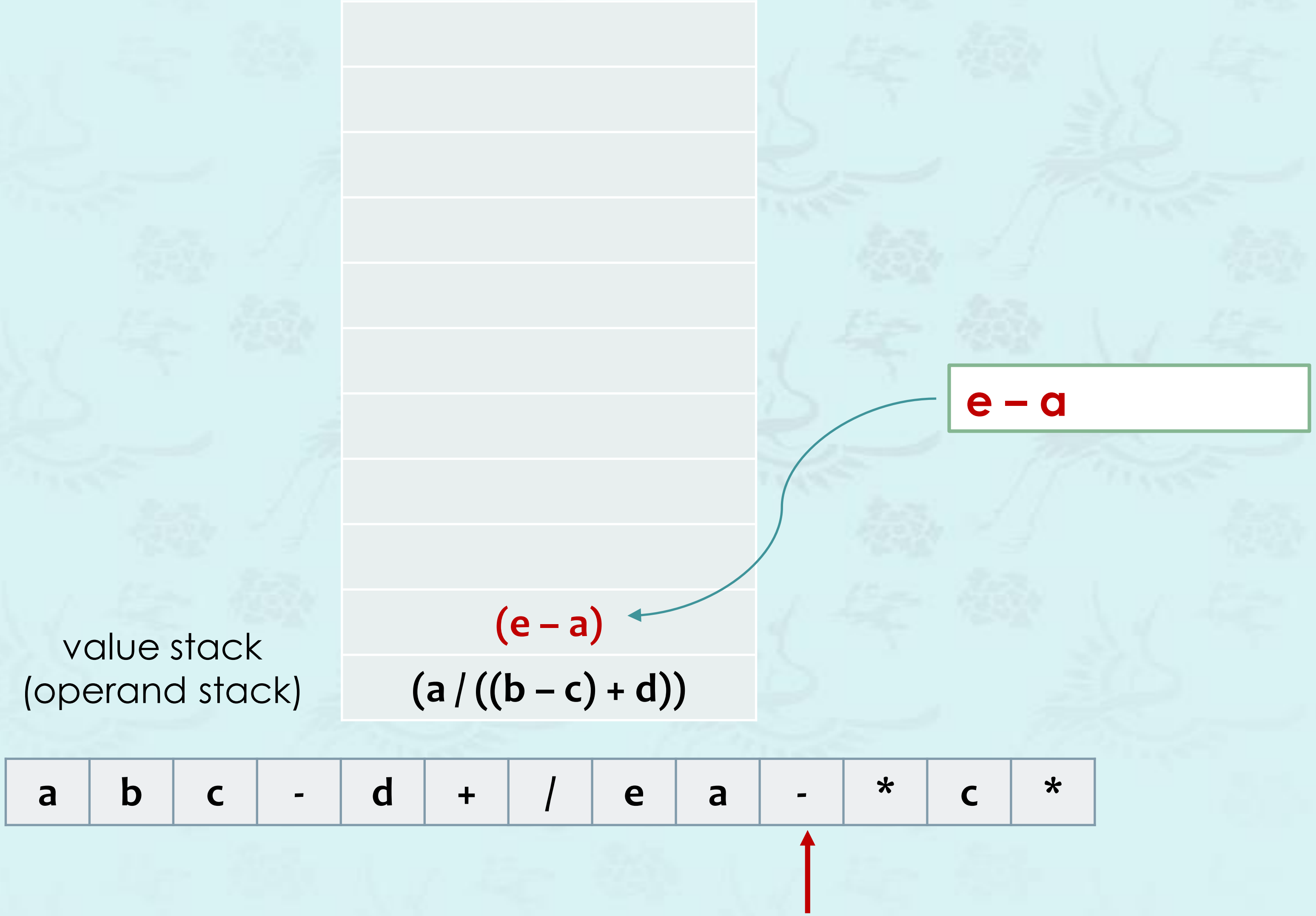
value stack
(operand stack)

e – a

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



e – a

(e – a)

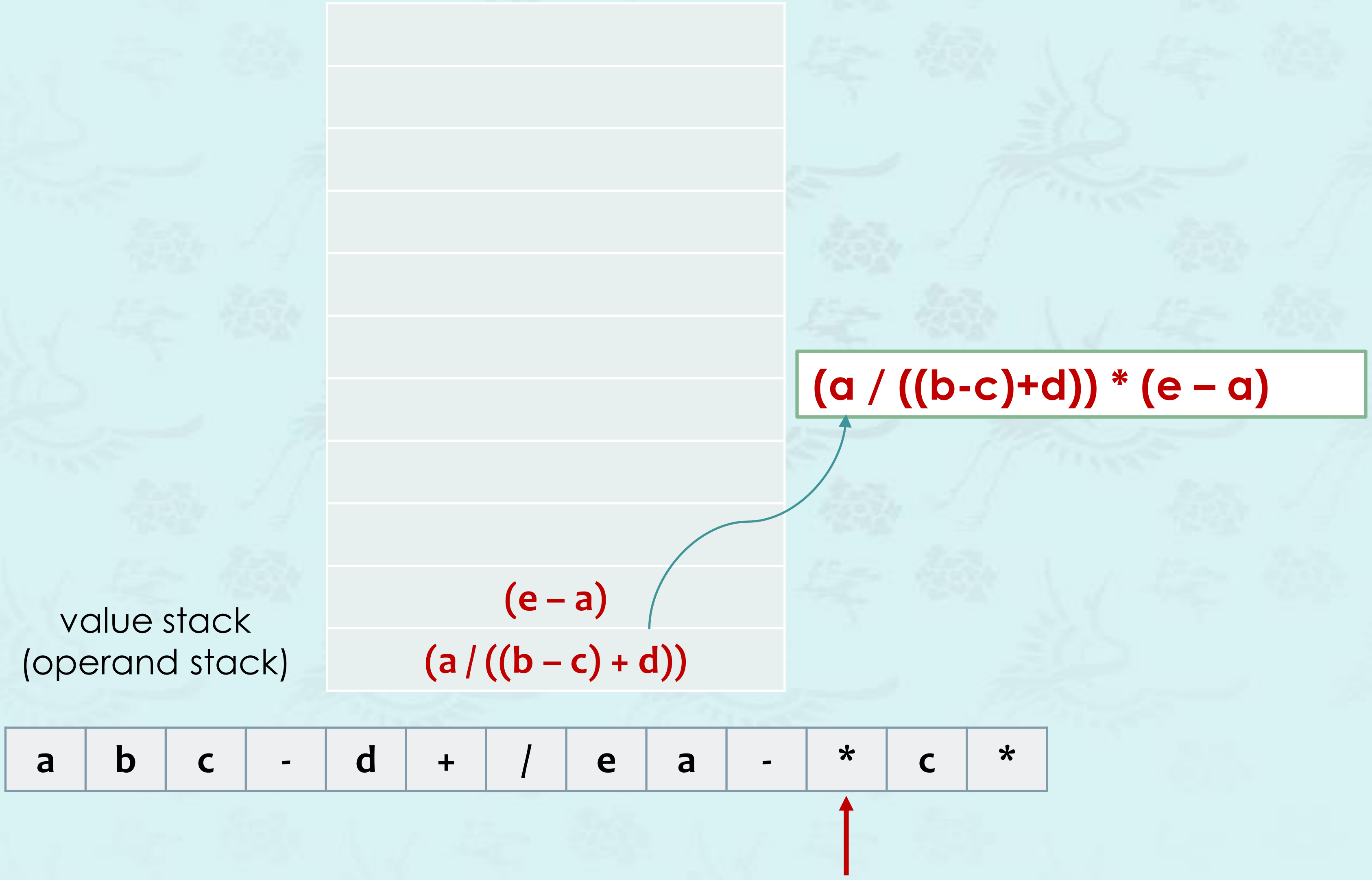
(a / ((b – c) + d))

value stack
(operand stack)

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



(a / ((b-c)+d)) * (e – a)

value stack
(operand stack)

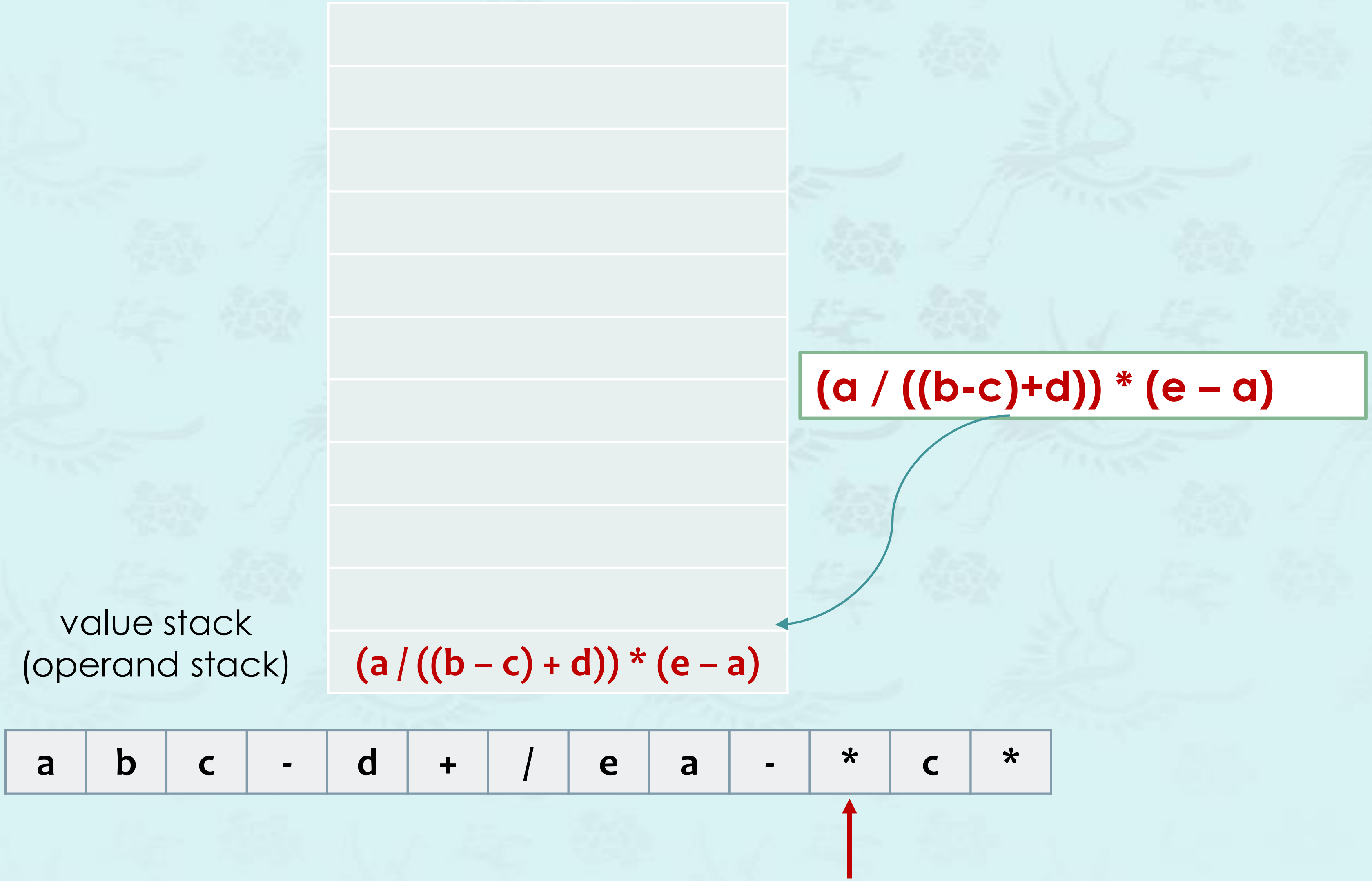
(e – a)

(a / ((b – c) + d))

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

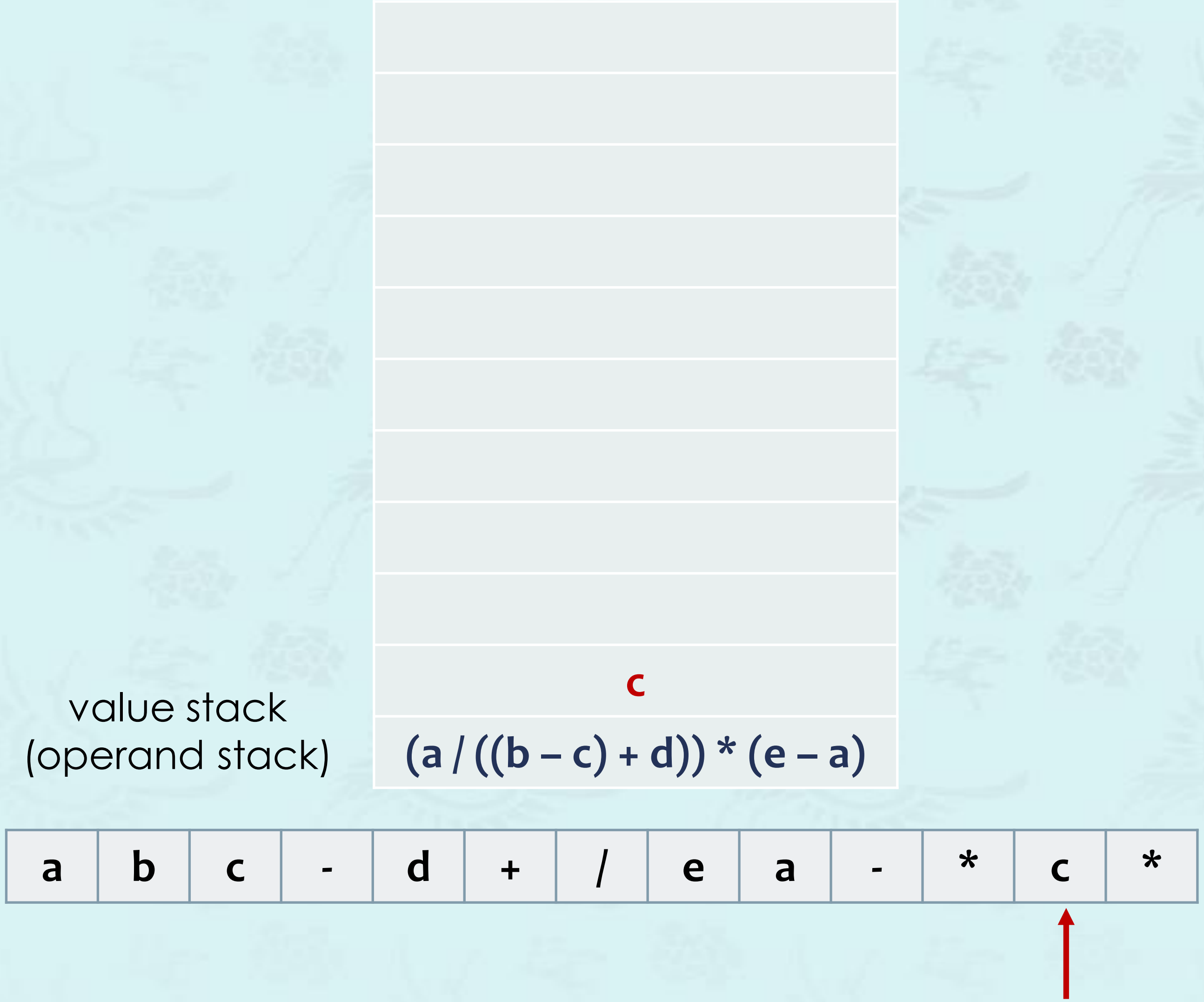
(a / ((b – c) + d)) * (e – a)

(a / ((b-c)+d)) * (e – a)

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



value stack
(operand stack)

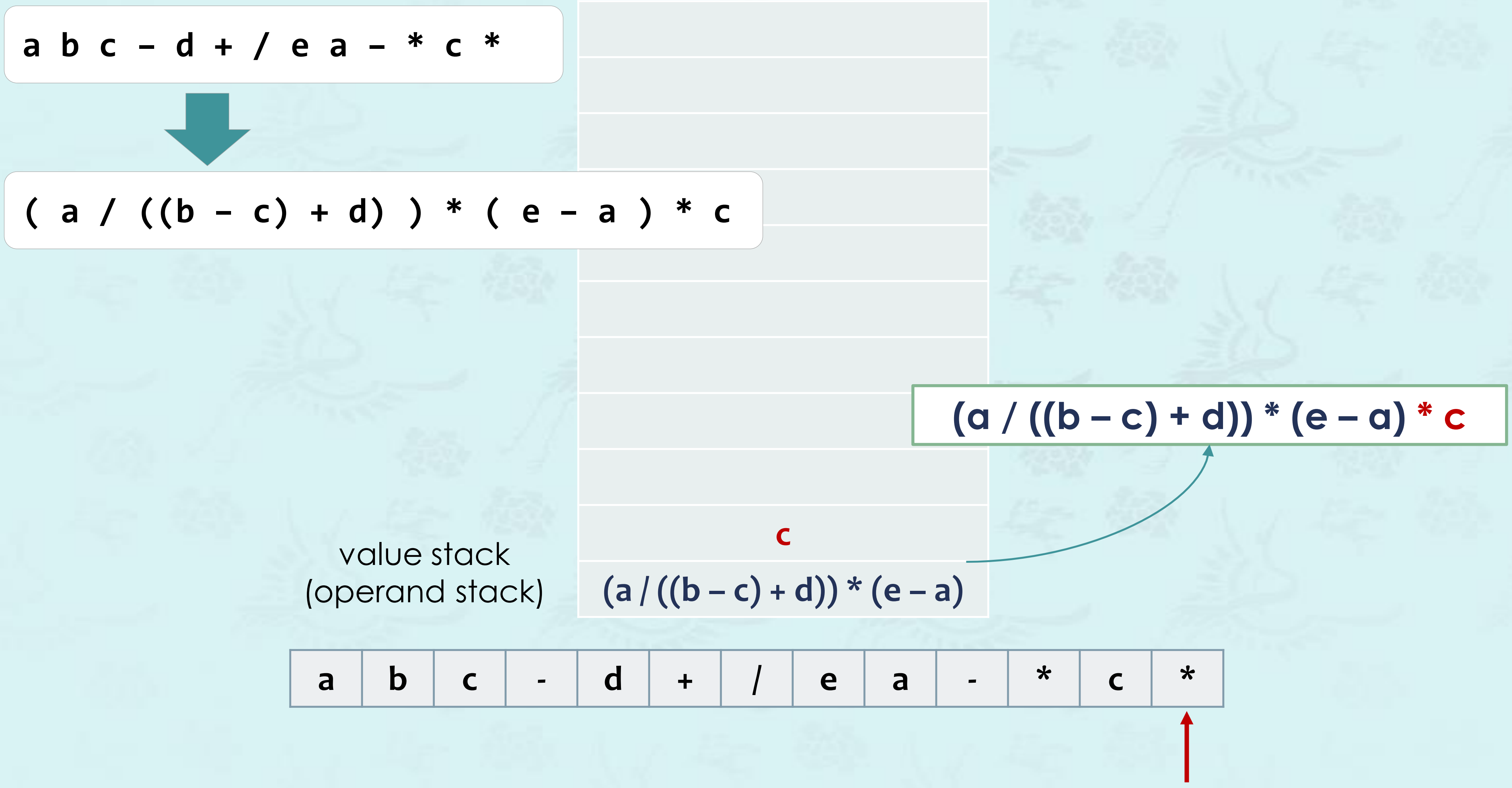
c

(a / ((b – c) + d)) * (e – a)

a	b	c	-	d	+	/	e	a	-	*	c	*

Postfix Evaluation

- **Goal**: Evaluate postfix expressions.



a b c – d + / e a – * c *

(a / ((b – c) + d)) * (e – a) * c

value stack
(operand stack)

c

(a / ((b – c) + d)) * (e – a)

(a / ((b – c) + d)) * (e – a) * c

a	b	c	-	d	+	/	e	a	-	*	c	*

Extra Examples – Infix to Postfix Conversion

- **Example 1:** 3+4*5/6

in	stack(bottom to top)	postfix
3		3
+	+	
4		3 4
*	+ *	
5		3 4 5
/	+ /	3 4 5 *
6		3 4 5 * 6
		3 4 5 * 6 / +

1. Operands are output immediately
2. Push "(" always and operators in general.
3. For ")", pop until "(". Discard "(" and ")".
4. For higher precedence operator, push it.
5. For lower or equal precedence operator, pop them until "(" and push it.

Extra Examples – Infix to Postfix Conversion

- **Example 2:** (1+3)*(4-2)/(5+7)

in	stack (bottom to top)	postfix
(	(	
1		1
+	(+	
3		1 3
)		1 3 +
*	*	
(	* (	
4		1 3 + 4
-	* (-	
2		1 3 + 4 2
)	*	1 3 + 4 2 -
/	/	1 3 + 4 2 - *

in	stack	postfix
(	/ (	1 3 + 4 2 - *
5		1 3 + 4 2 - * 5
+	/ (+	
7		1 3 + 4 2 - * 5 7
)		1 3 + 4 2 - * 5 7 +
		1 3 + 4 2 - * 5 7 + /

1. Operands are output immediately
2. Push "(" always and operators in general.
3. For ")", pop until "(". Discard "(" and ")".
4. For higher precedence operator, push it.
5. For lower or equal precedence operator, pop them until "(" and push it.

Data Structures
Chapter 3

1. Stack
2. Queue
3. **Stack Applications**
   - Arithmetic Expressions
     - Infix, Prefix, and Postfix
   - Arithmetic Expression Evaluation
     - Dijkstra's Two-Stack Algorithm
     - Postfix Evaluation

Summary &

quaestio quaestio qo ? ? ? ?