ITP20001/ECE 20010 Data Structures

Data Structures

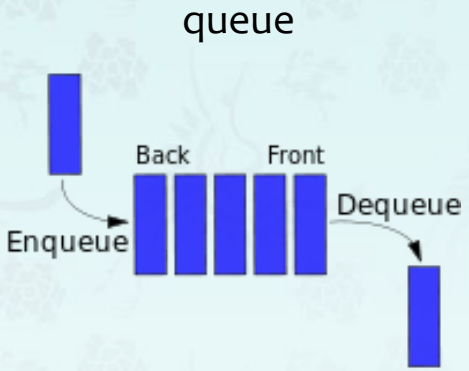
Chapter 3

- *stacks & queues applications*
  - ***infix to postfix***

Chapter 3 – Stacks and queues

3.3 Queues

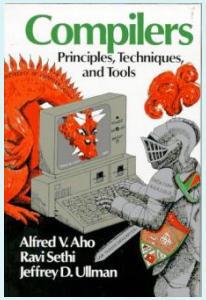
Queue: An ordered list in which **enqueues** (insertion or add) at the **rear** and **dequeues** (deletion or remove) take place at different end or **front**.
It is also known as a Fist-in-first-out(FIFO) list.

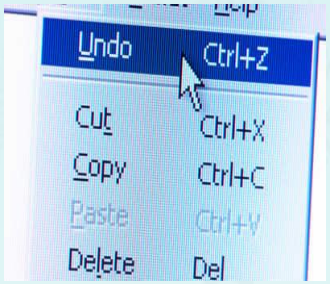


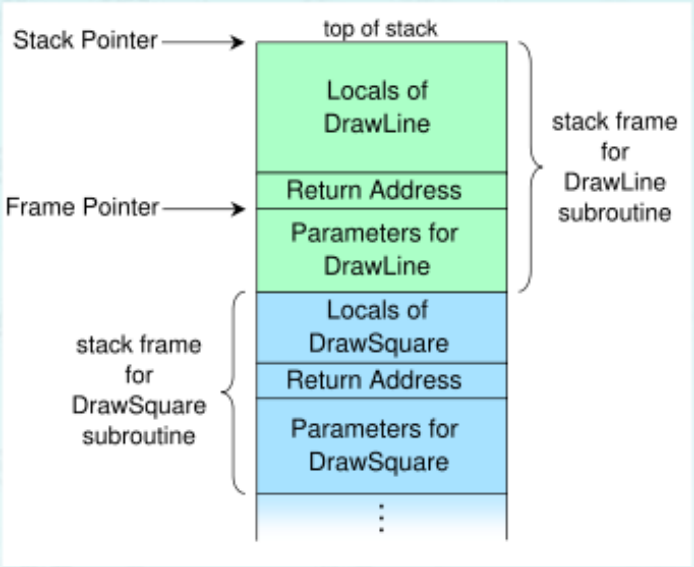
- Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

Chapter 3 – Stacks and queues

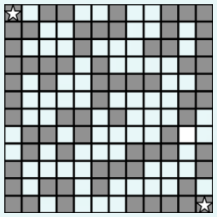
3.4 Stack and queue applications







- Parsing in a compiler. (p.127)
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Backtracking as in a maze (p.121)
- Implementing function calls in a compiler. (p.108)
- ...



3.4 Stack and 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
- need a file from the disk system
- send an email
- job scheduling

3.6 Arithmetic expression evaluation

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

(1 + 2) * 3

operand operator

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 *

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

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

postfix

3.6 Arithmetic expression evaluation

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

(1 + 2) * 3

operand operator

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 *

- Operands are output immediately
- Stack operators until right parens
- Unstack until left parens
  Delete left parens
- In general, higher precedence operator must be output before lower one.)

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

postfix 1 2 3 + 4 5 * * +

3.6 Arithmetic expression evaluation

Goal: 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 value stack.

3.6 Dijkstra's two-stack algorithm

- **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	)	)	)

3.6 Dijkstra's two-stack algorithm

- **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	)	)	)

3.6 Dijkstra's two-stack algorithm

- **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)

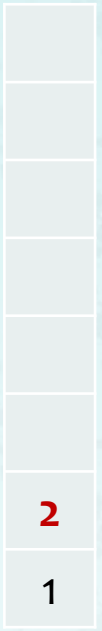
operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

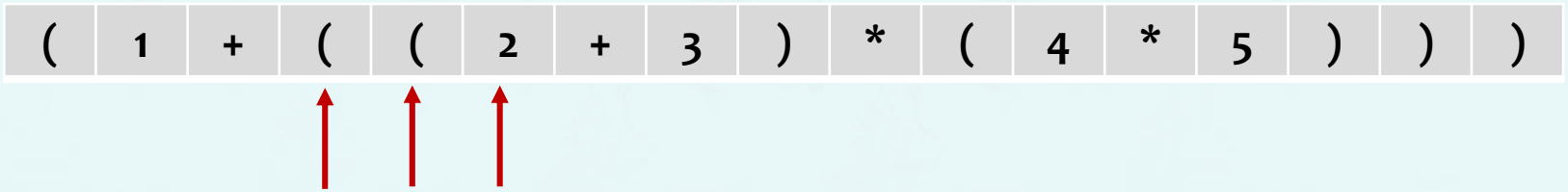
- **Left parenthesis:** ignore.
- **Right parenthesis:**
  - pop operator and two values;
  - push the result of applying that operator to those values onto the value stack.



value stack
(operand stack)

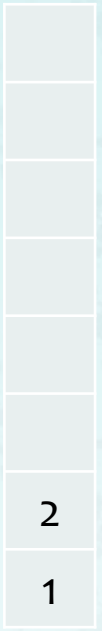
operator stack

infix expression
(fully parenthesized)



3.6 Dijkstra's two-stack algorithm

- **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)

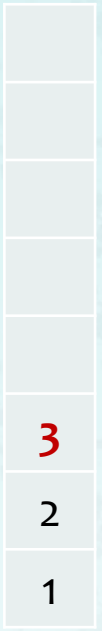
operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

- **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)

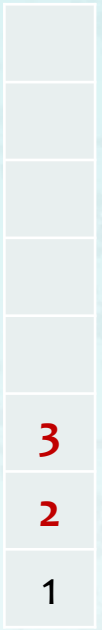
operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

- **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 + 3

value stack
(operand stack)

operator stack

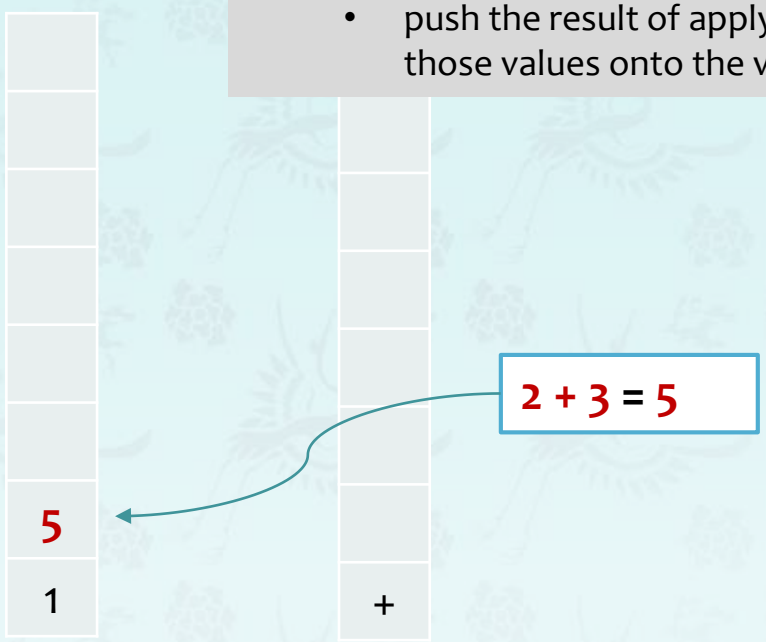
infix expression
(fully parenthesized)

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

- pop operator & two values
- push the result

3.6 Dijkstra's two-stack algorithm

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



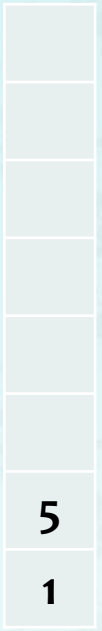
infix expression
(fully parenthesized)

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

- pop operator & two values
- push the result

3.6 Dijkstra's two-stack algorithm

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



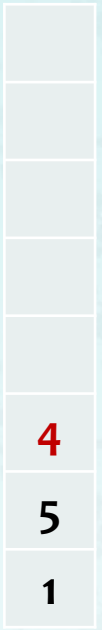
value stack
(operand stack)

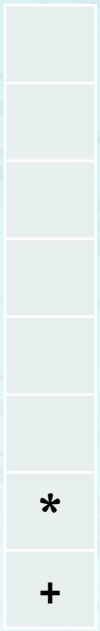
operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

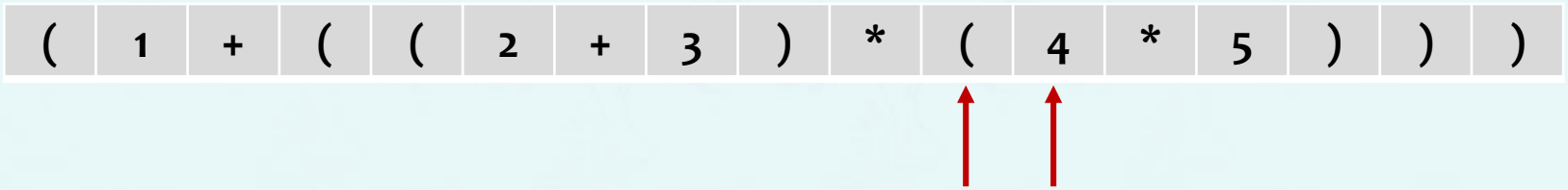




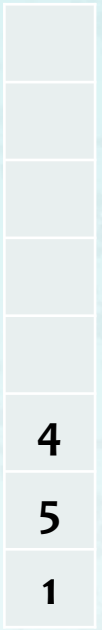
value stack
(operand stack)

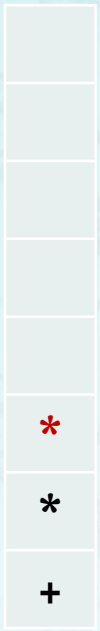
operator stack

infix expression
(fully parenthesized)



3.6 Dijkstra's two-stack algorithm





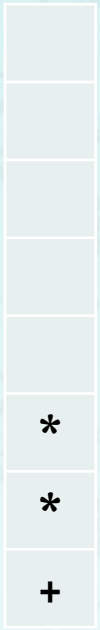
value stack
(operand stack)

operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm



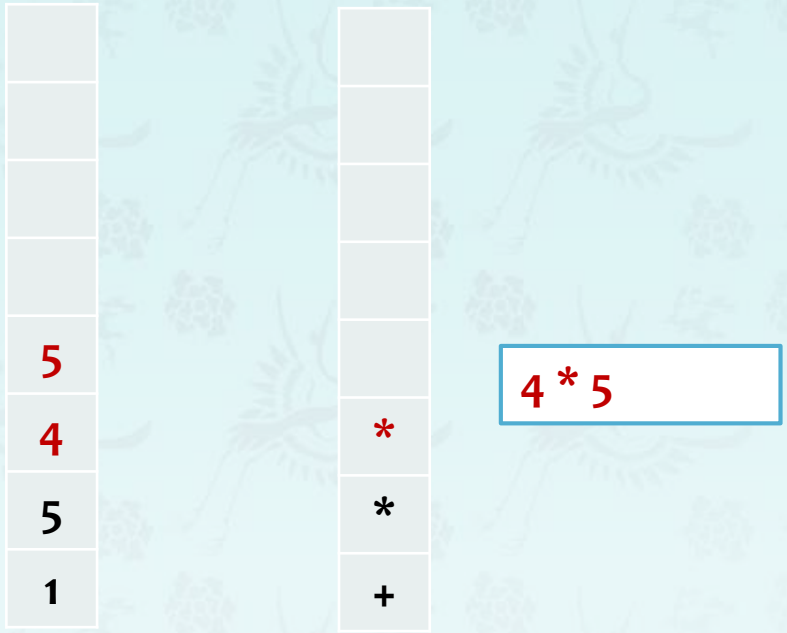
value stack
(operand stack)

operator stack

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

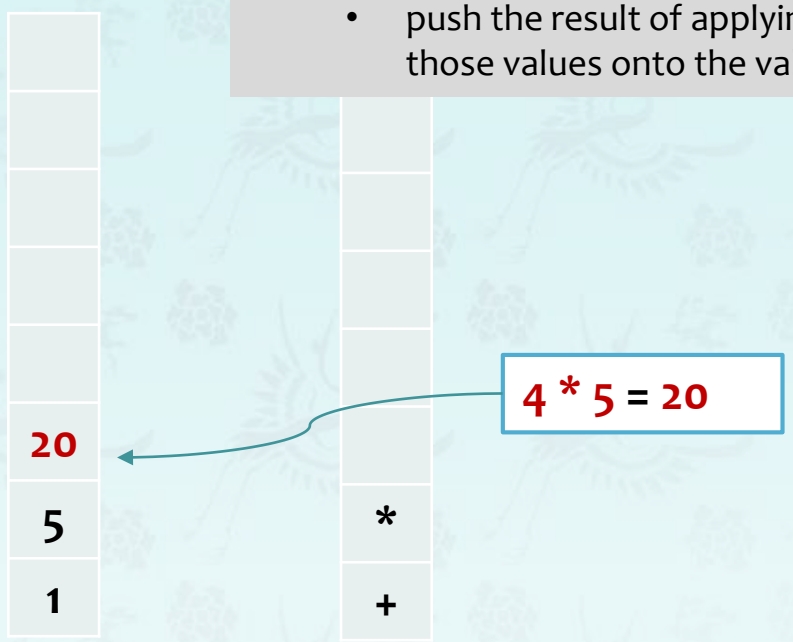


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

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

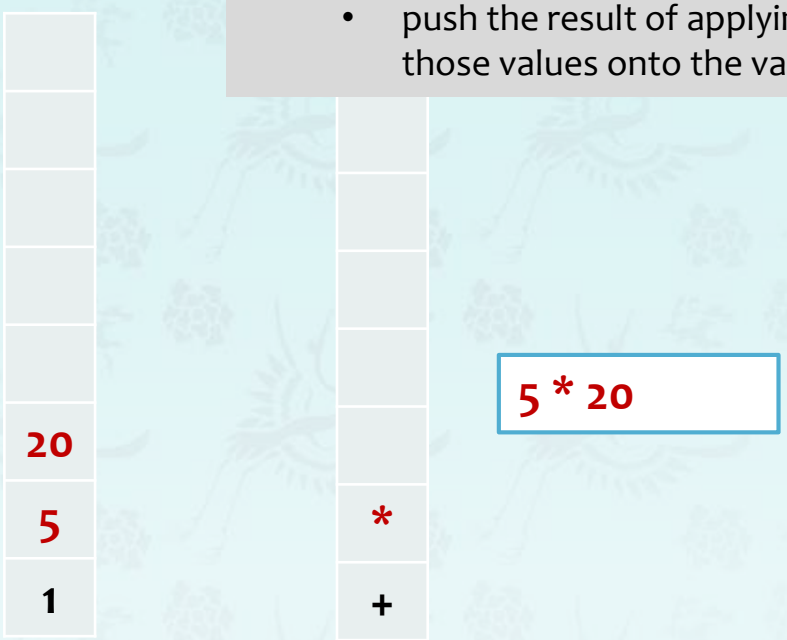


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

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

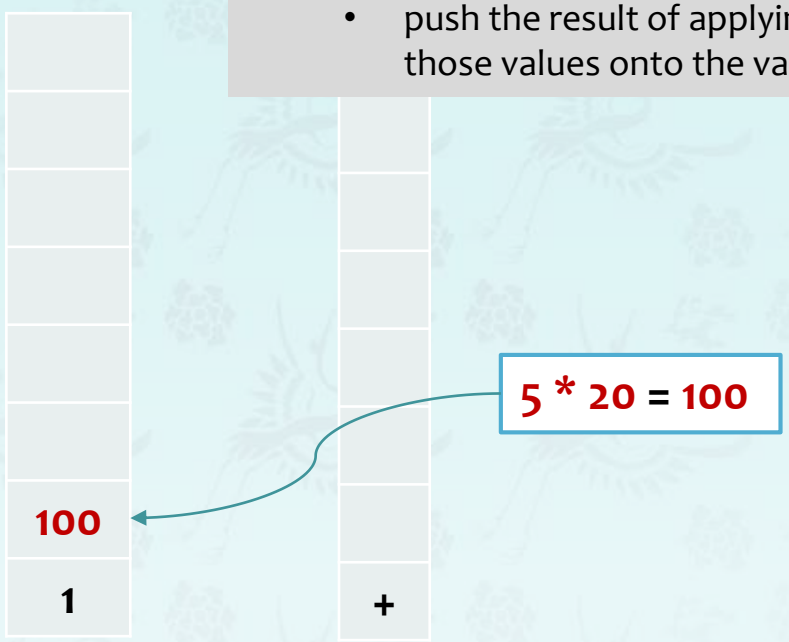


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

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

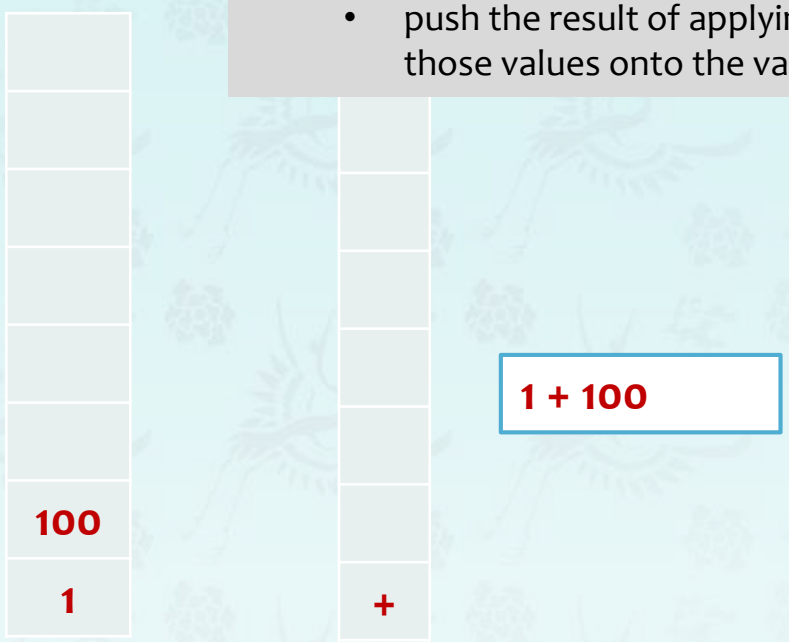


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

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

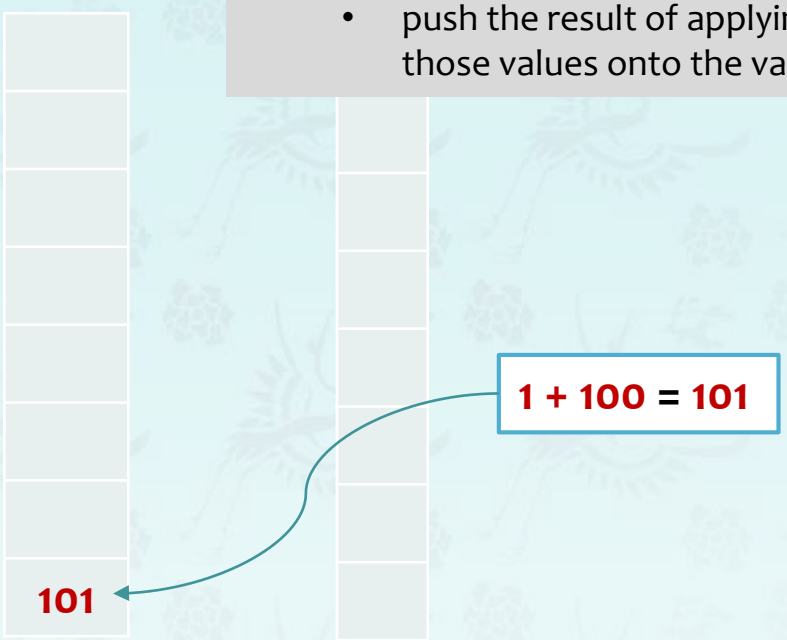


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

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

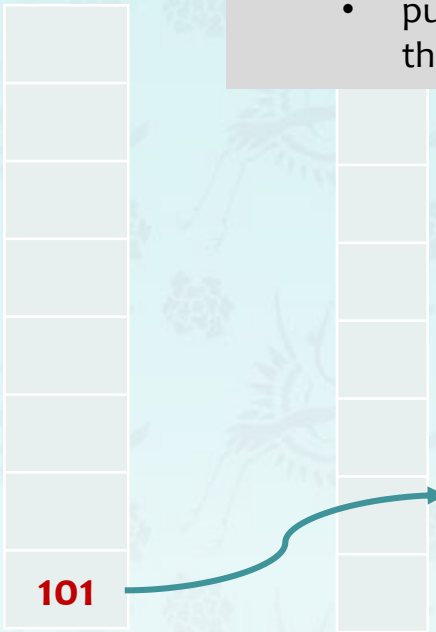


infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

- **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

Empty the stack for the result at the end of expression

infix expression
(fully parenthesized)

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

3.6 Dijkstra's two-stack algorithm

```
public class ArithmeticExpression {
  public static void main(String[] args) {
    Stack<Character> ops = new Stack<Character>();
    Stack<Double> vals = new Stack<Double>();
    String e = JOptionPane.showInputDialog(null,
      "Enter an expression", "Stack application", JOptionPane.QUESTION_MESSAGE);
    if (e == null) return;           // Check "Cancel"

    for (int i = 0; i < e.length(); i++) {
      Character c = e.charAt(i);
      if (c.equals(' ') || c.equals('(')) ;
      else if (c == '+') ops.push(c);
      else if (c == '*') ops.push(c);
      else if (c == ')') {
         Character op = ops.pop();
         if     (op.equals('+')) vals.push(vals.pop() + vals.pop());
         else if(op.equals('*')) vals.push(vals.pop() * vals.pop());
      }
      else {
         String s = "" + c;
         vals.push(Double.parseDouble(s));
      }
    }
    JOptionPane.showMessageDialog(null, e + " = " + vals.pop());
  }
}
```

3.6 Arithmetic expression evaluation

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.

3.6 Arithmetic expression evaluation

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, ….

3.6 Arithmetic expression evaluation

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

Figure 3.13 Infix and postfix notation

3.6 Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

a b c – d + / e a – * c *

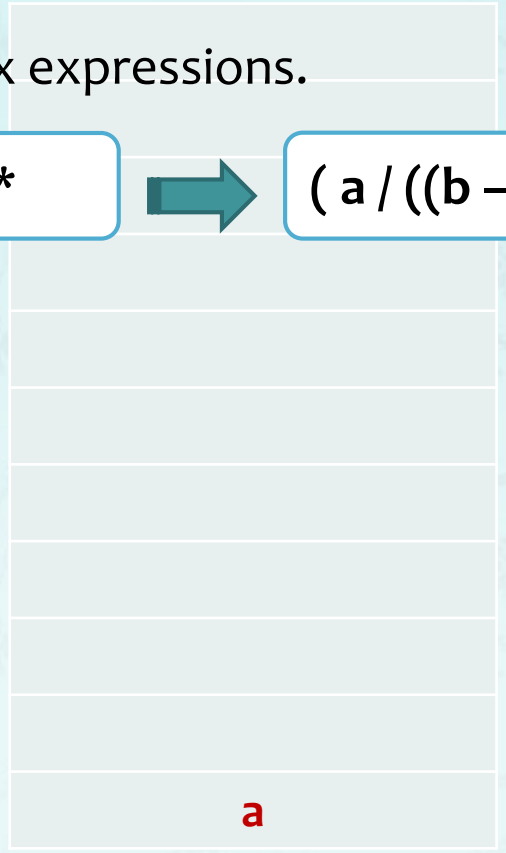
(a / ((b – c) + d)) * (e – a) * c

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

3.6 Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

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



value stack
(operand stack)

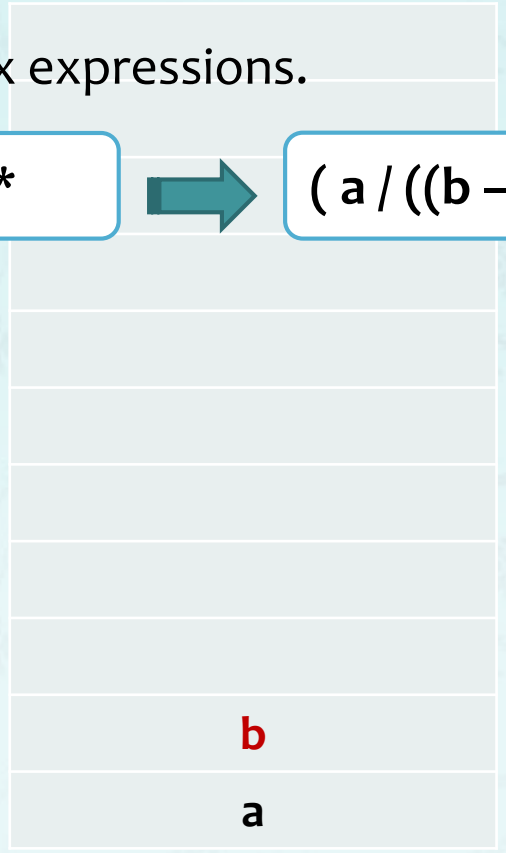
a	b	c	-	d	+	/	e	a	-	*	c	*

push the operands
until an operator comes up.

3.6 Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

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



value stack
(operand stack)

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

3.6 Arithmetic expression evaluation

a b c – d + / e a – * c *

(a / ((b – c) + d)) * (e – a) * c

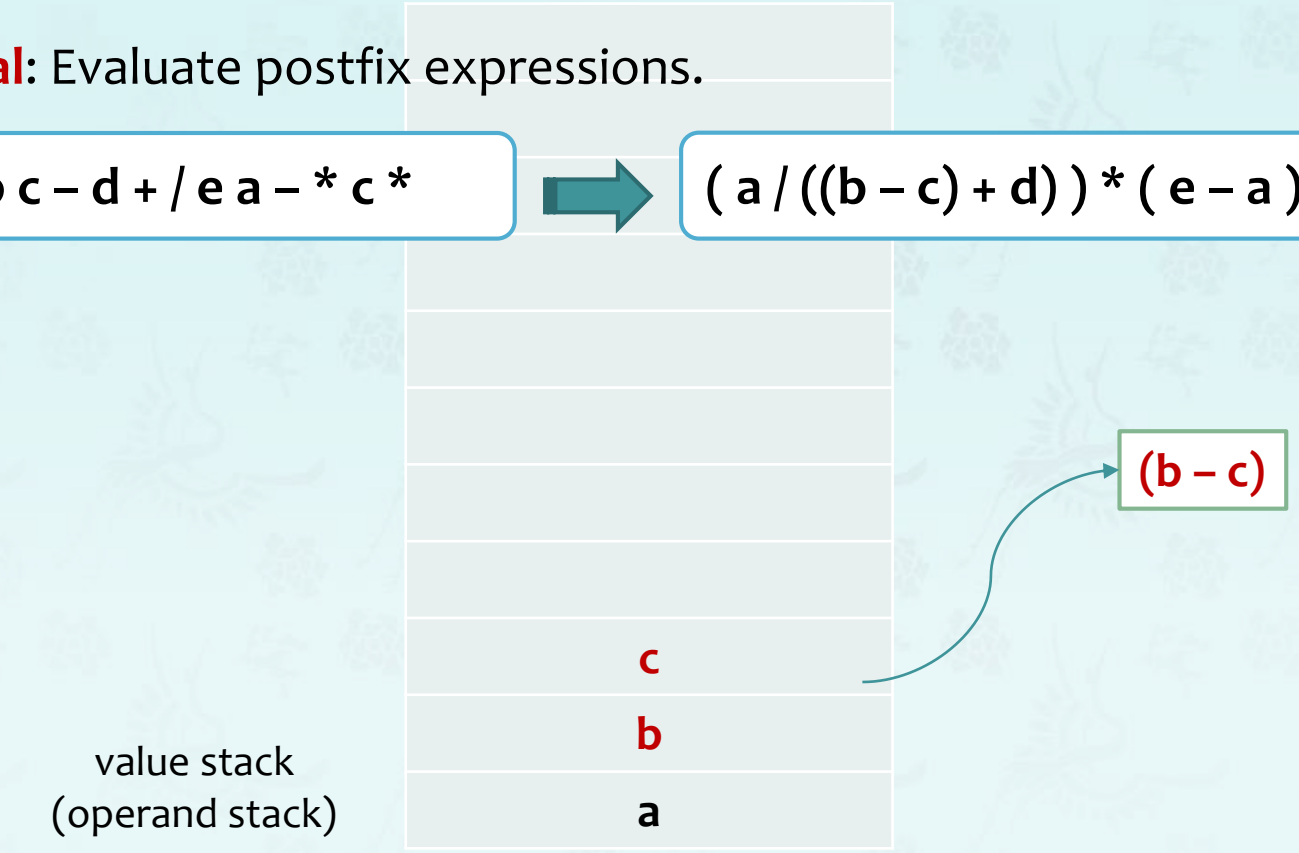
value stack
(operand stack)

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

3.6 Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

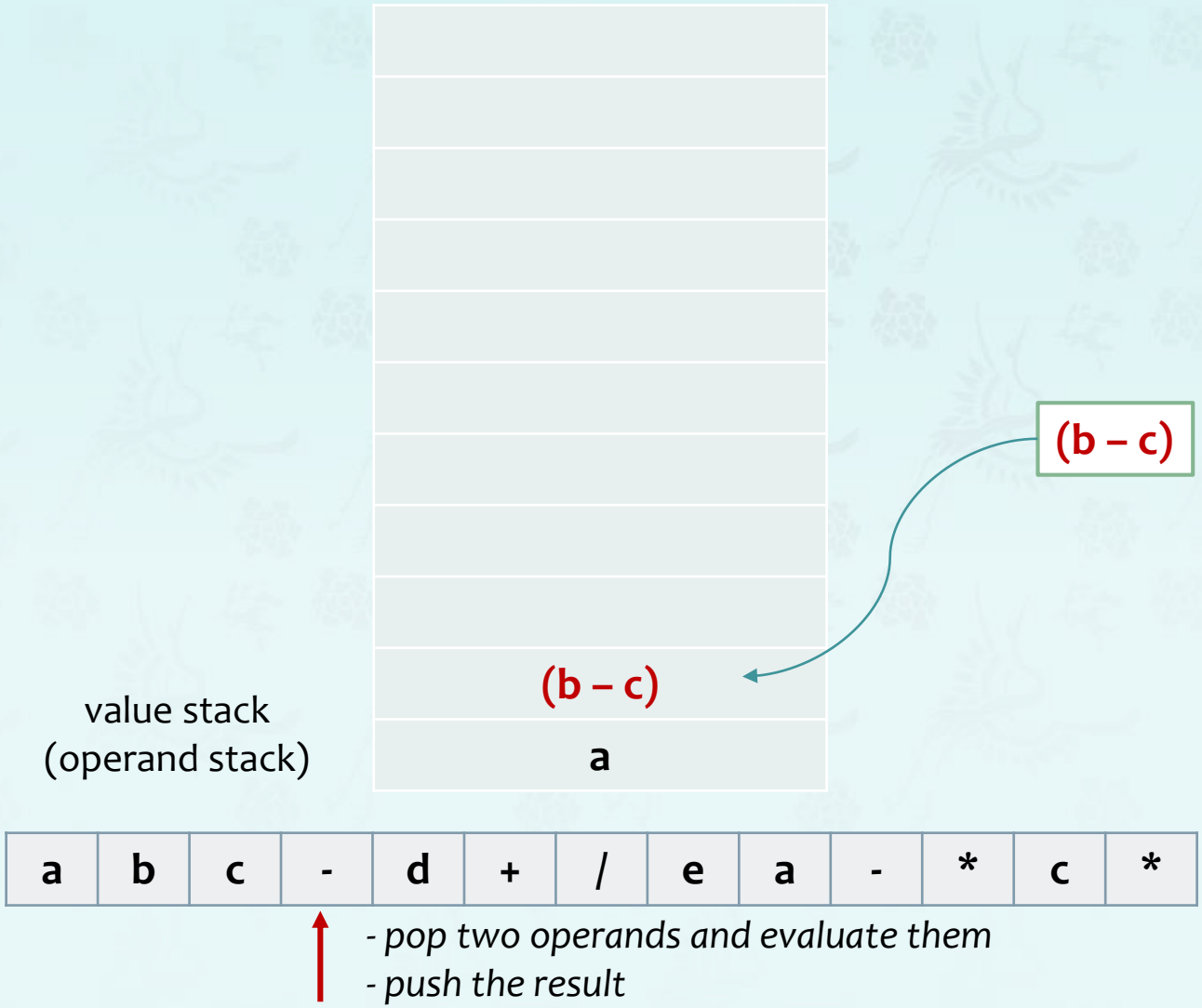
a b c – d + / e a – * c * ⇒ (a / ((b – c) + d)) * (e – a) * c



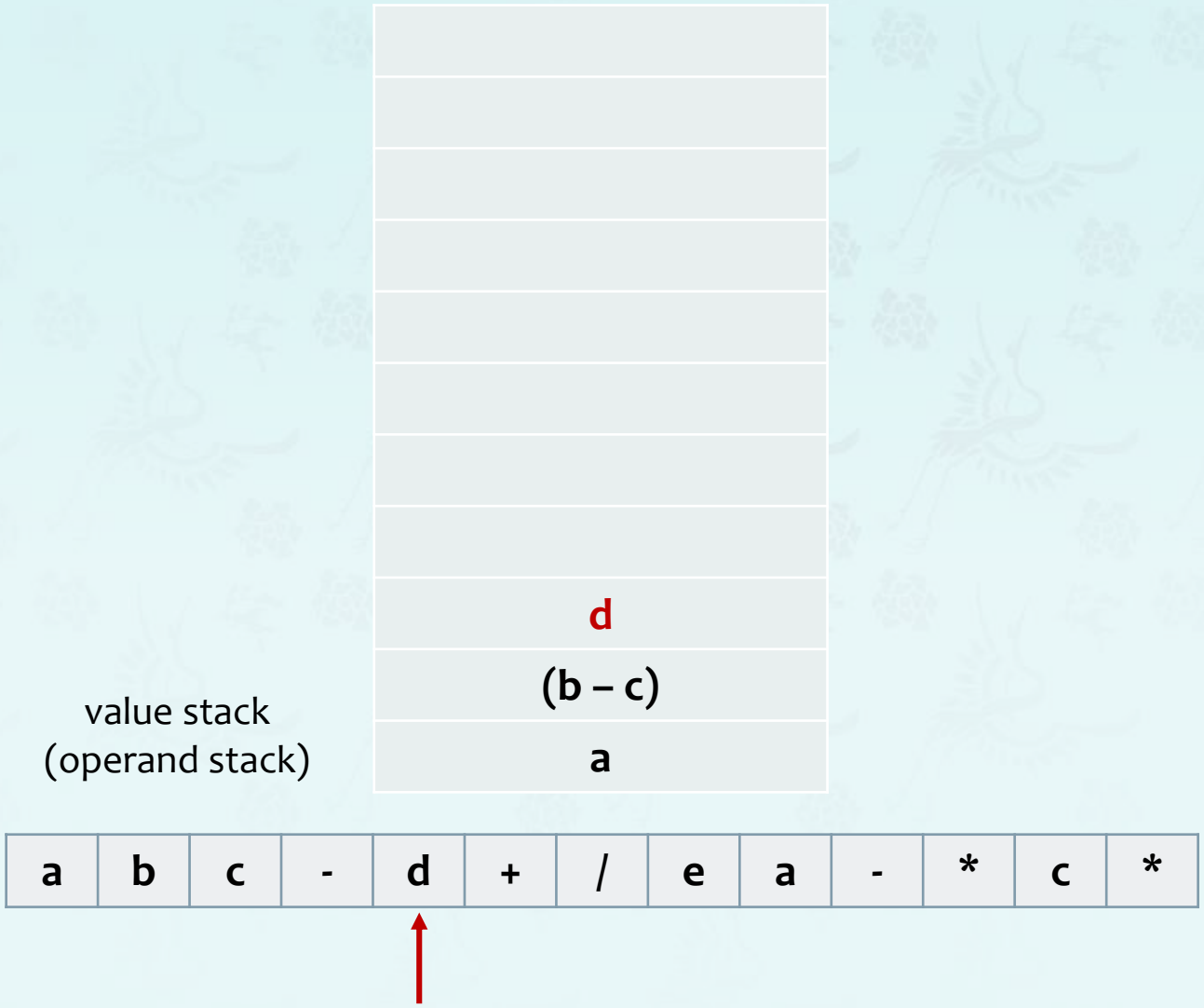
a	b	c	-	d	+	/	e	a	-	*	c	*

- pop two operands and evaluate them
- push the result

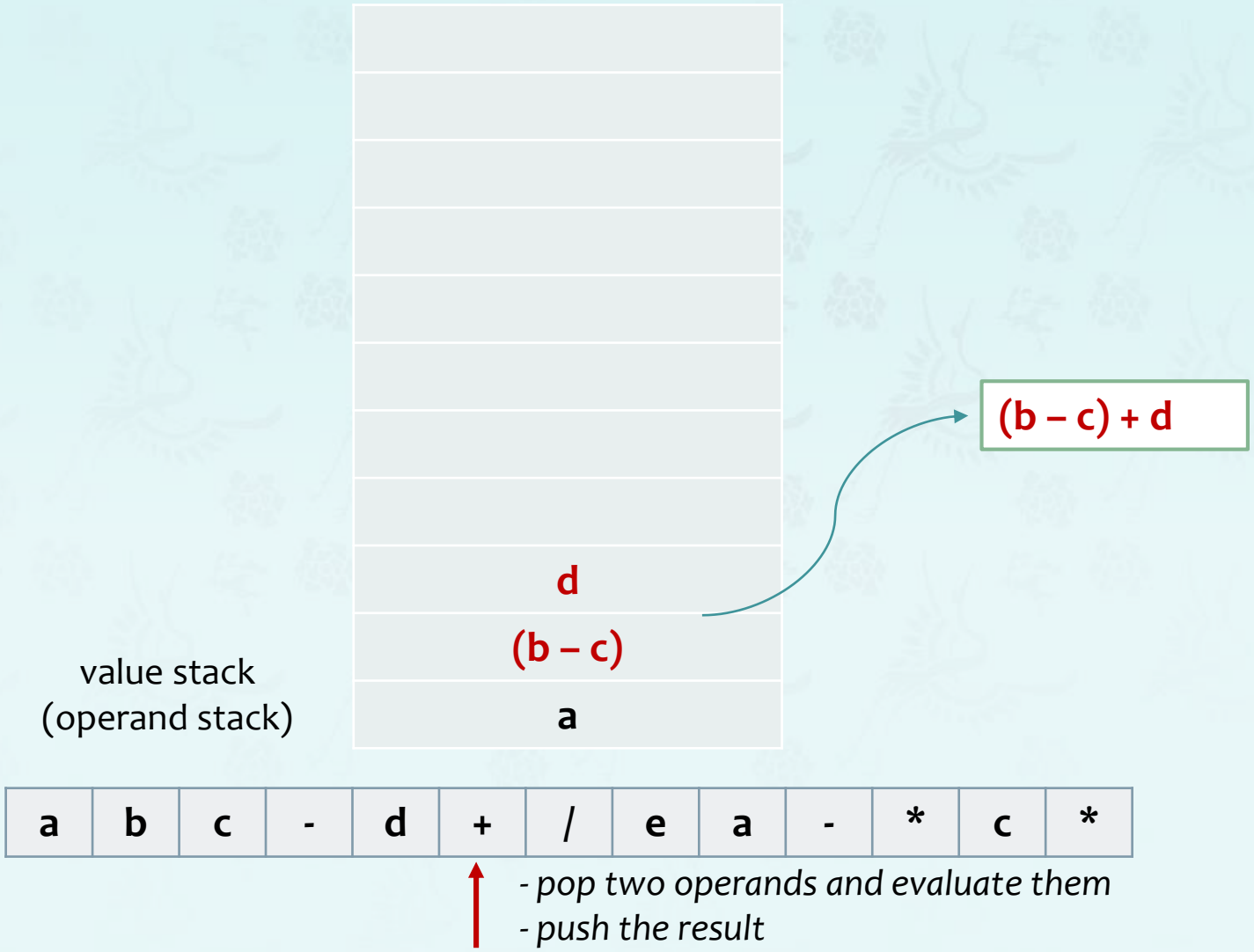
3.6 Arithmetic expression evaluation



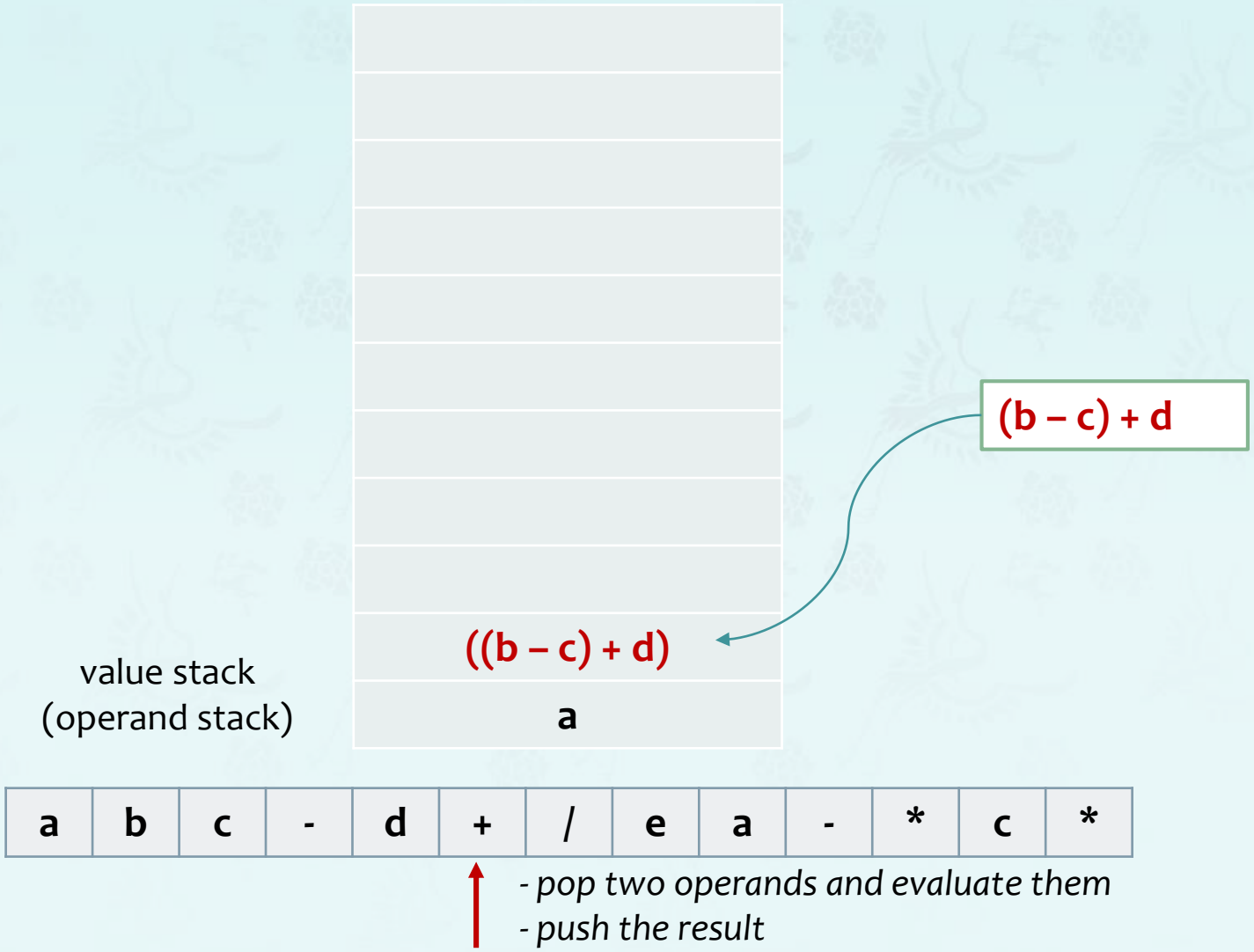
3.6 Arithmetic expression evaluation



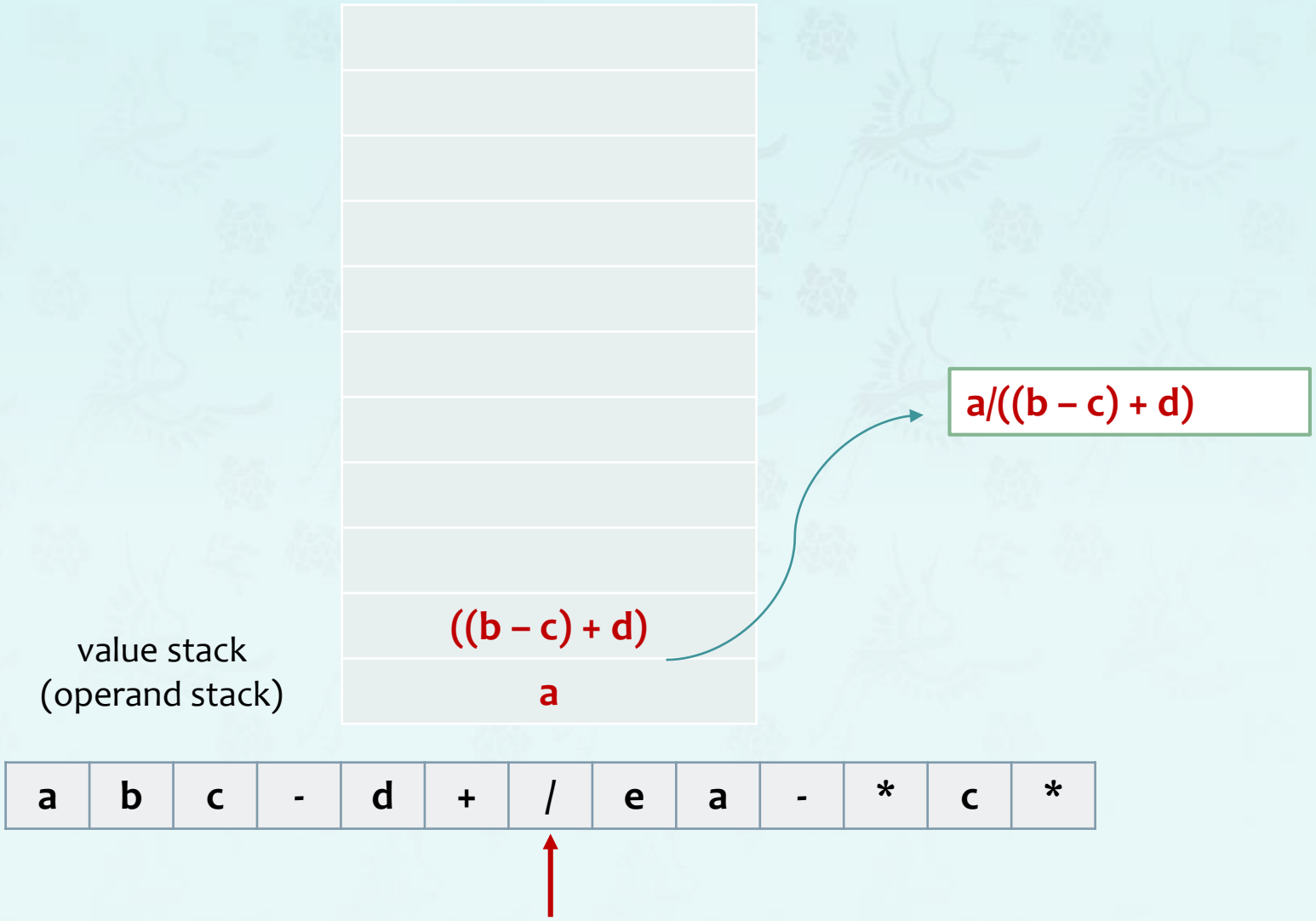
3.6 Arithmetic expression evaluation



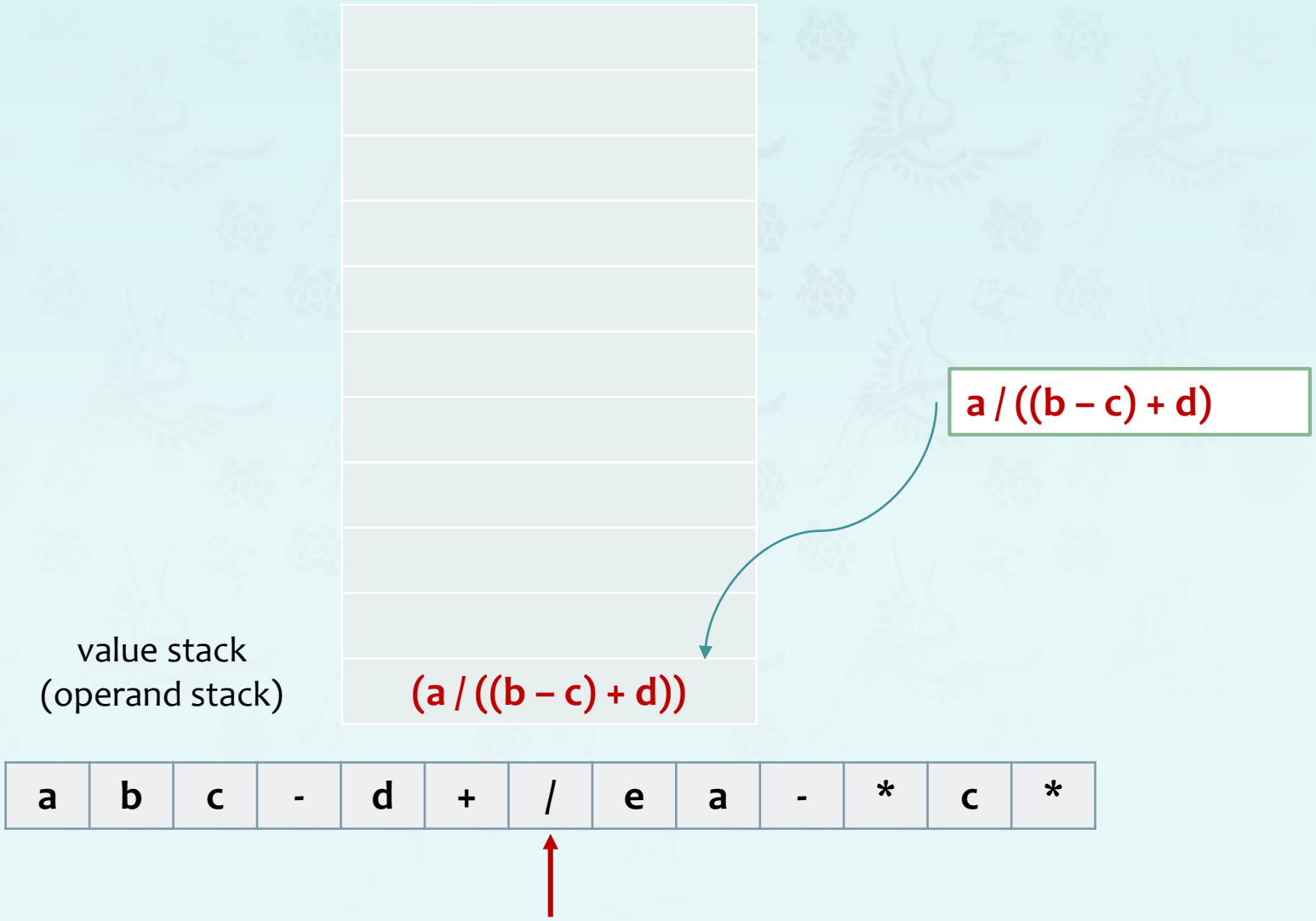
3.6 Arithmetic expression evaluation



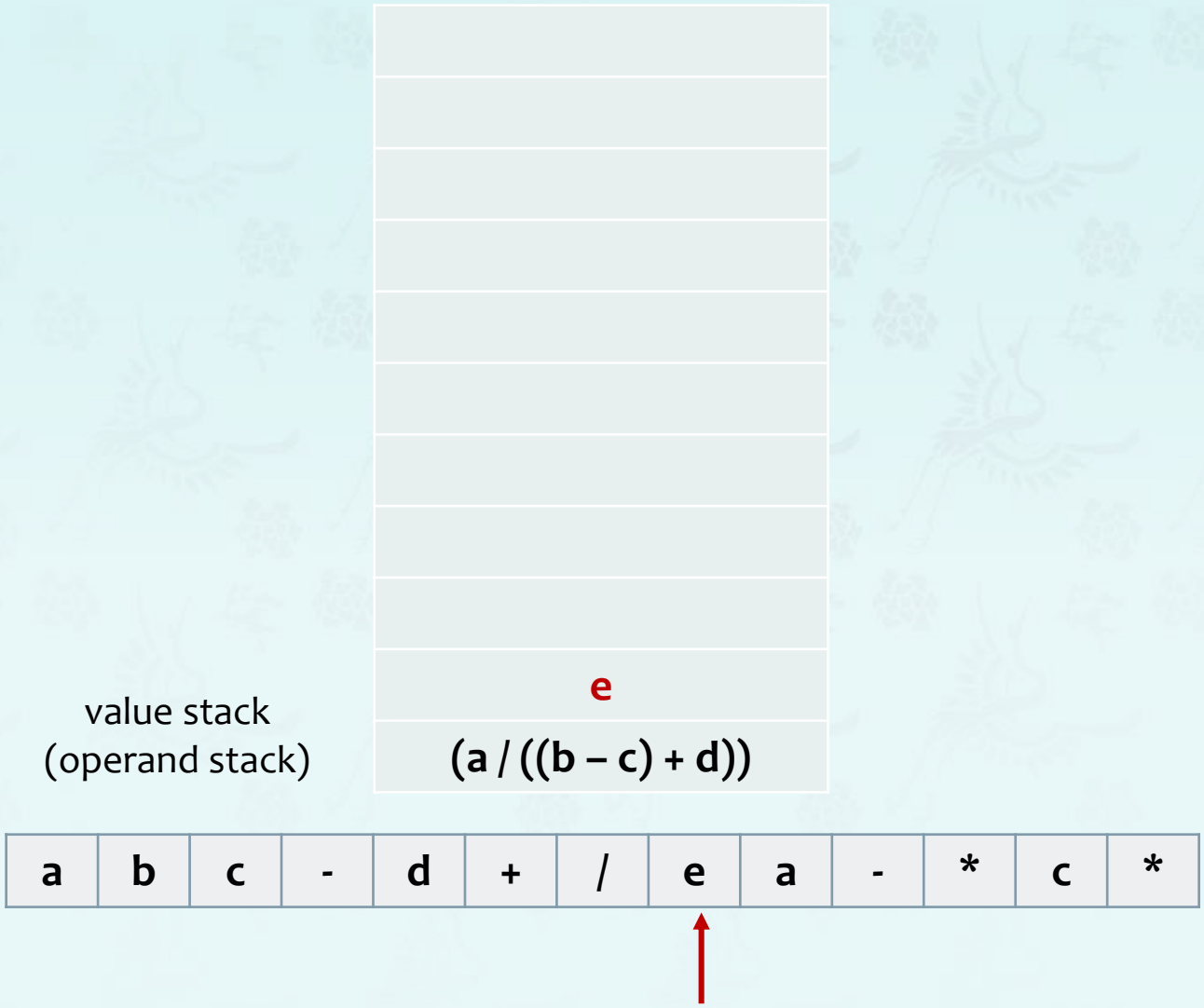
3.6 Arithmetic expression evaluation



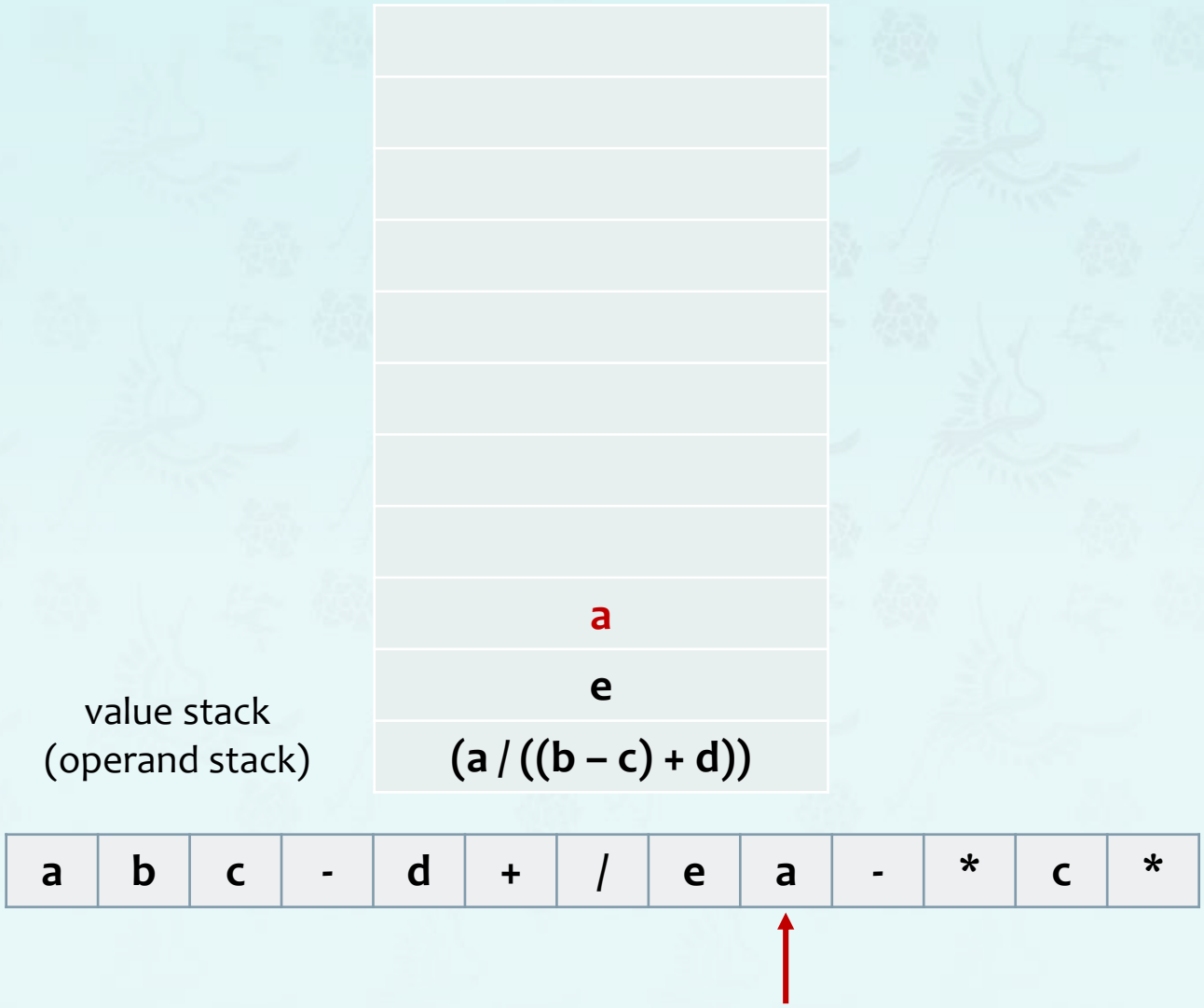
3.6 Arithmetic expression evaluation



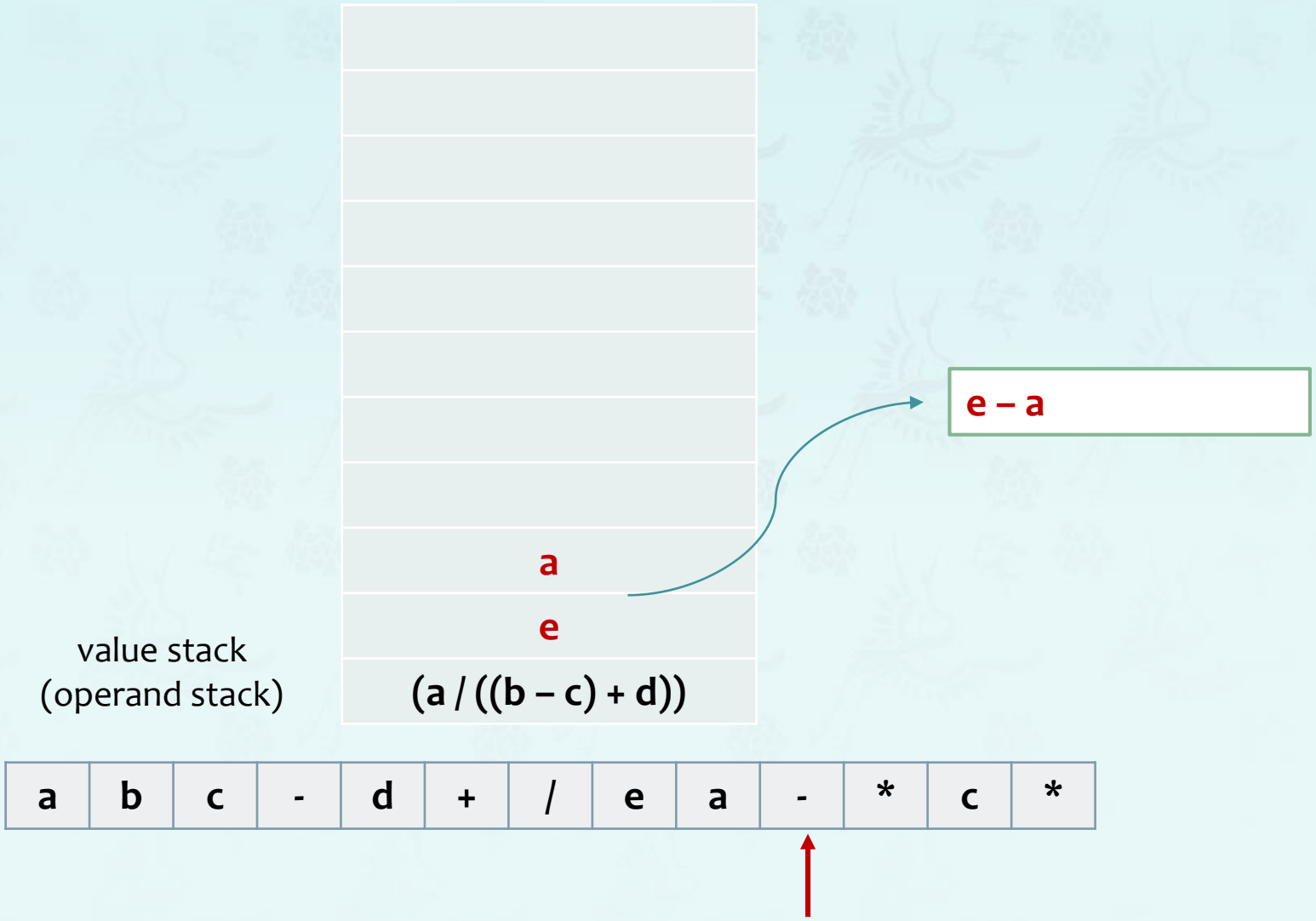
3.6 Arithmetic expression evaluation



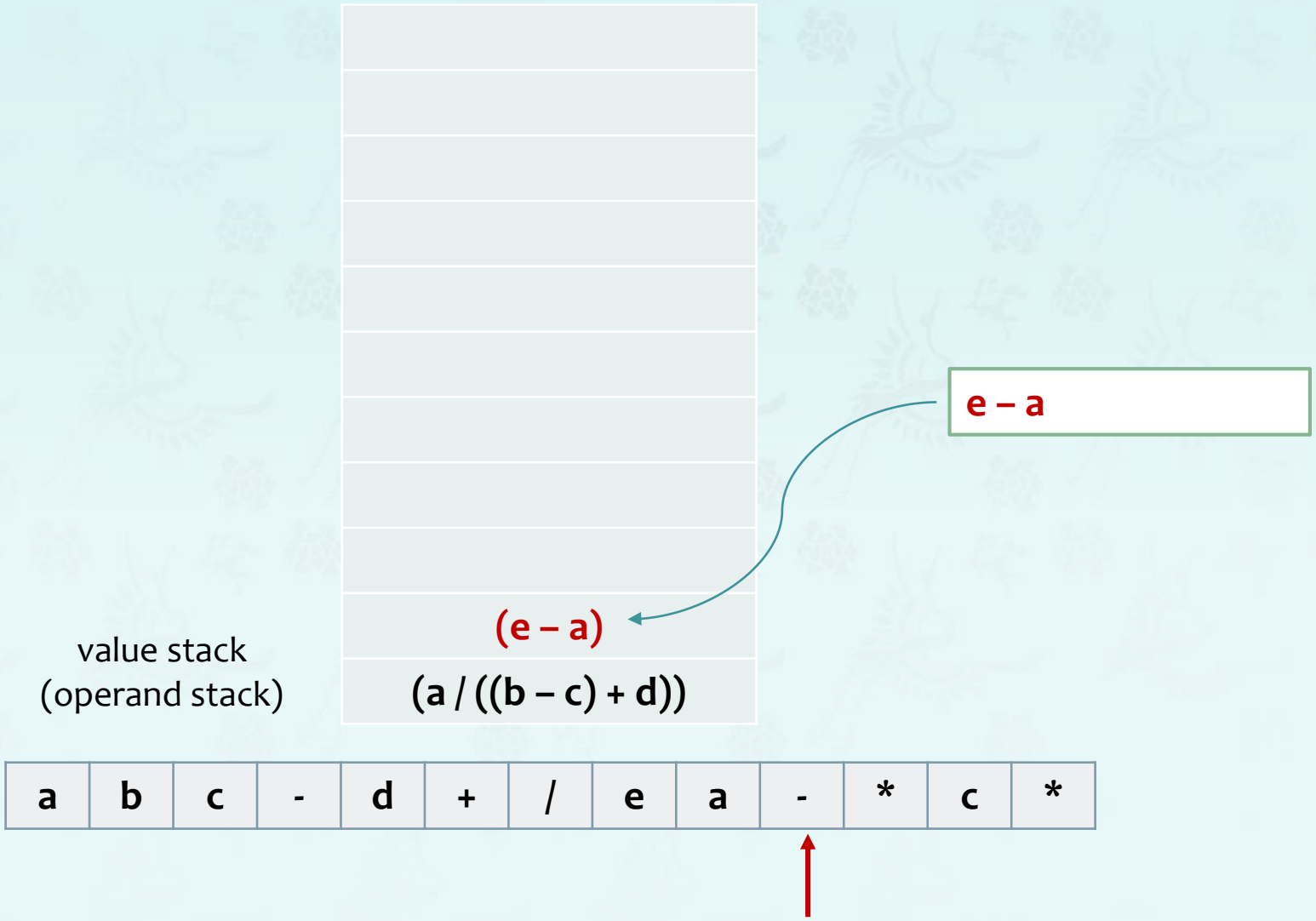
3.6 Arithmetic expression evaluation



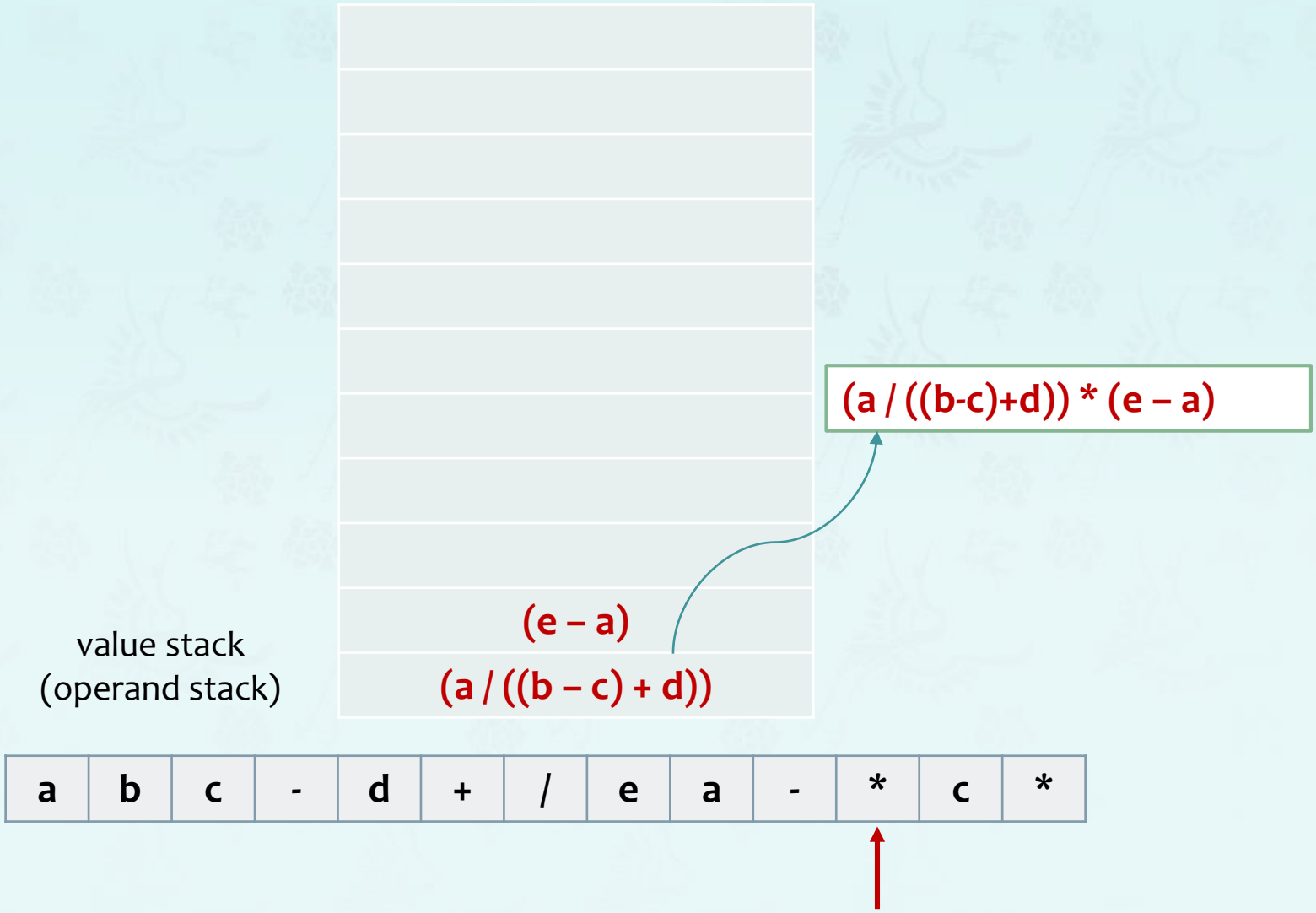
3.6 Arithmetic expression evaluation



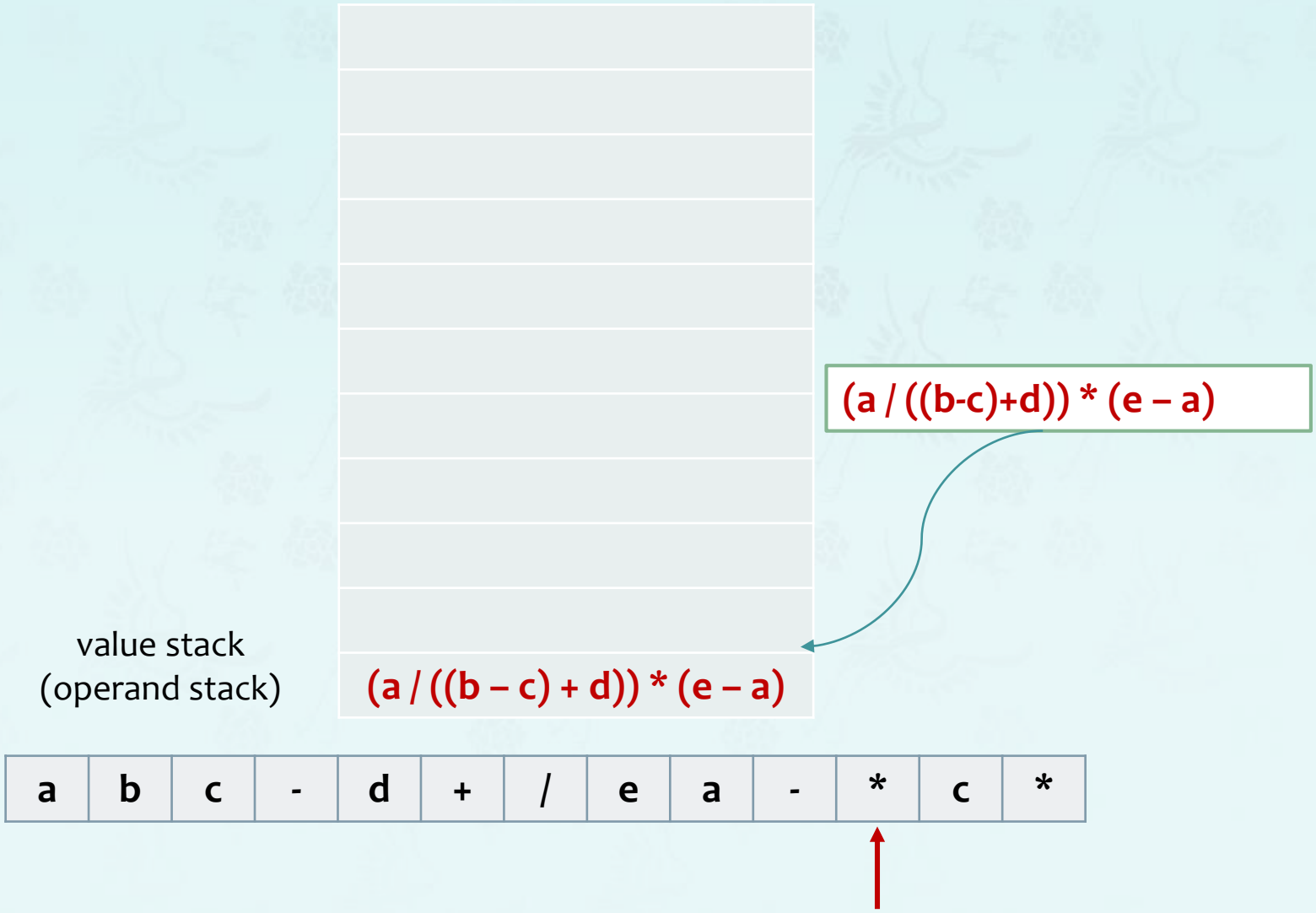
3.6 Arithmetic expression evaluation



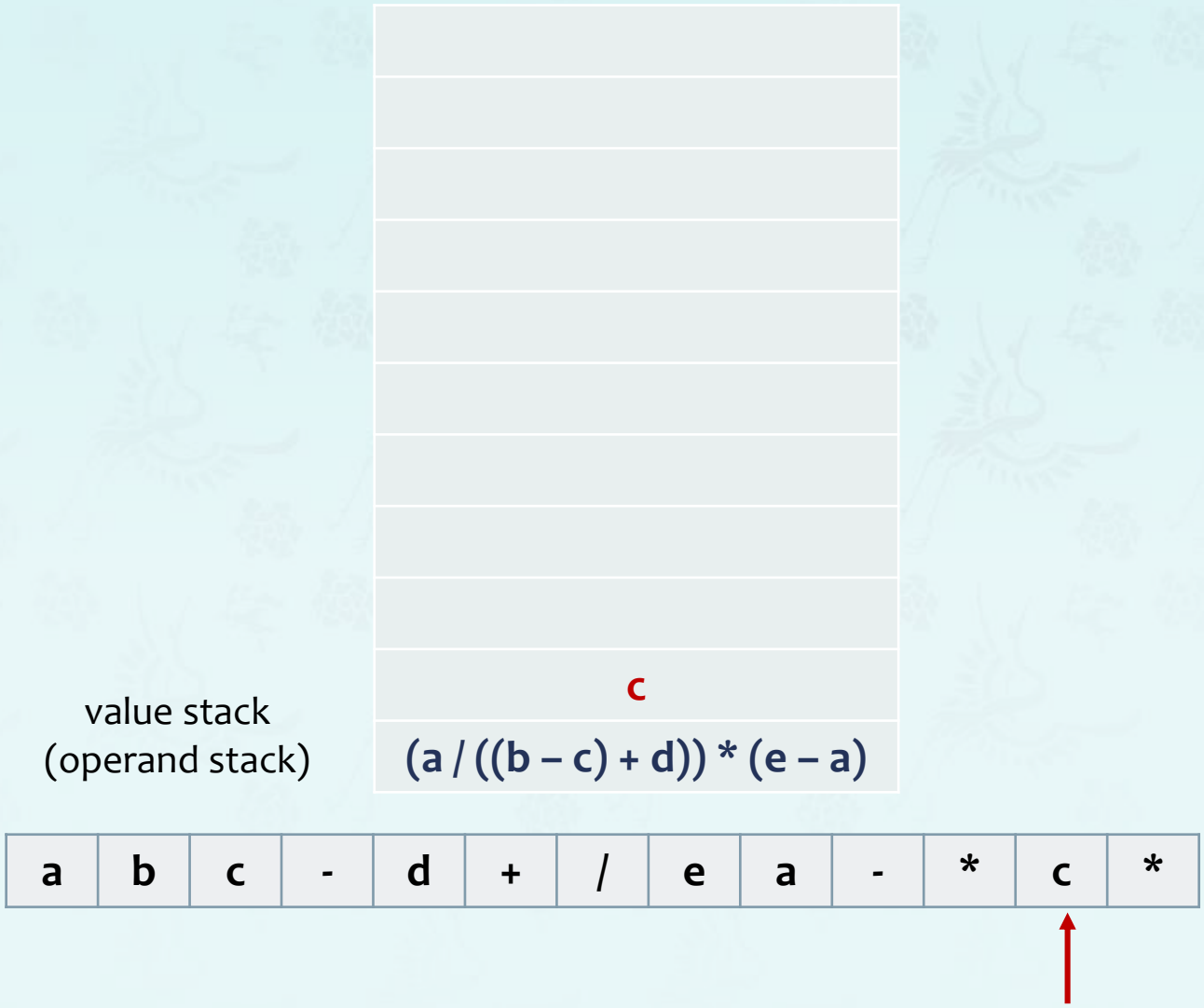
3.6 Arithmetic expression evaluation



3.6 Arithmetic expression evaluation



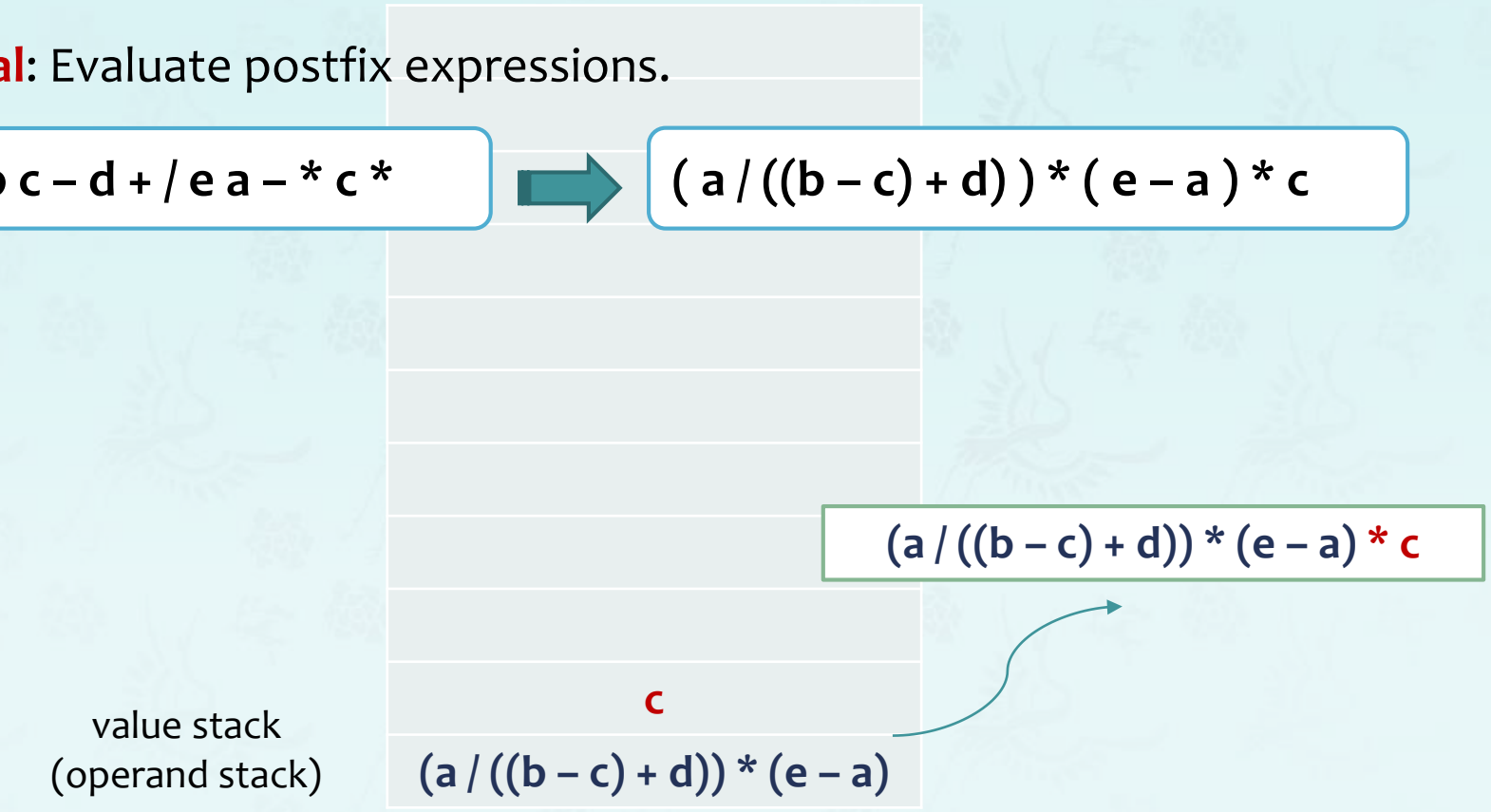
3.6 Arithmetic expression evaluation



3.6 Arithmetic expression evaluation

Goal: Evaluate postfix expressions.

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



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

Chapter 3 – Stacks and queues

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

We use a stack.

1. When an **operand** is read, output it.
2. When an **operator** is read,
   - **Pop** until the top of the stack has an element of **lower** precedence.
   - Then **push** it.
3. When) is found, pop until we find the matching (.
4. (has the lowest precedence when in the stack
   but has the highest precedence when in the input.
5. When we reach the end of input, pop until the stack is empty.

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

Example 1:

infix: 3+4*5/6

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

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

Example 1:

infix: 3+4*5/6

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

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

Example 2:

infix: (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 - *
(	/ (	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 + /

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

Example 3:

infix: a – (b + c * d) / e

in	stack(bottom to top)	postfix
a		
-		
(		
b		
+		
c		
*		
d		
)		
/		
e		

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

Example 3:

infix: a – (b + c * d) / e

in	stack(bottom to top)	postfix
a		a
-	-	
(	- (	
b		ab
+	- (+	
c		abc
*	- (+ *	
d		abcd
)	-	abcd*+
/	- /	
e		abcd*+e
		abcd*+e/-

Chapter 3 – Stacks and queues

3.6 Arithmetic expression evaluation - Infix to Postfix Conversion

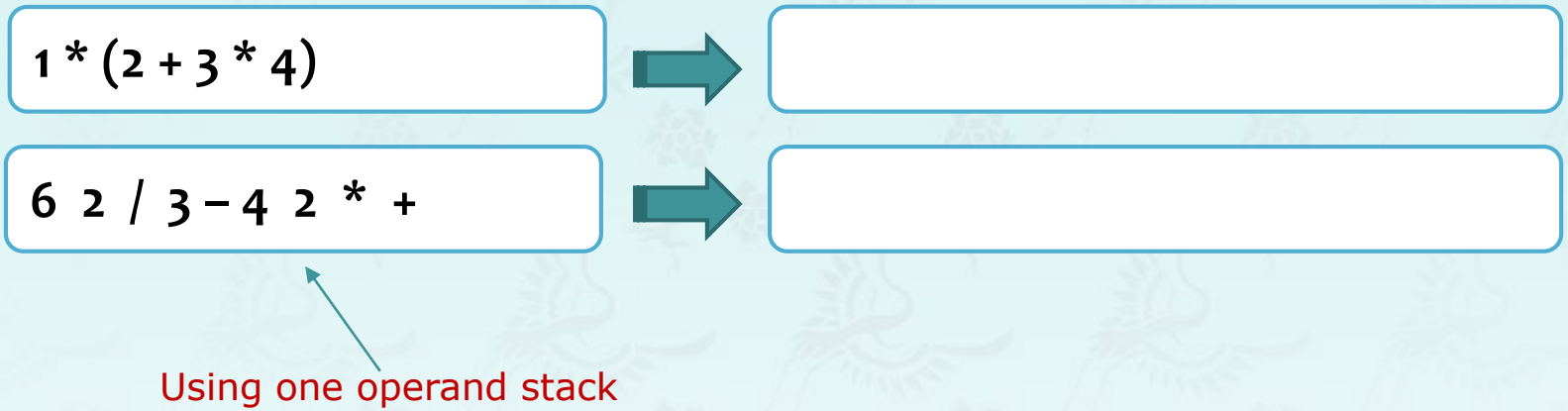
Example 3:

infix: A * (B + C * D) + E

	in	stack(bottom to top)	postfix
1	A		
2	*		
3	(		
4	B		
5	+		
6	C		
7	*		
8	D		
9	)		
10	+		
11	E		
12			

3.6 Arithmetic expression evaluation

Exercise: Infix to postfix, postfix to infix parsing



ECE 20010 Data Structures

Data Structures

Chapter 3

- *abstract data types - review*
- *stacks & queues*
  using dynamic arrays
- ***some applications***

Youngsup Kim, idebtor@handong.edu, Handong Global University