# Algorithms

Faraaz Sareshwala

http://algs4.cs.princeton.edu

## Stacks and Queues

Based on slides from Robert Sedgewick and Kevin Wayne

# Collections

A collection is a data types that store groups of items.

# Collections

A collection is a data types that store groups of items.

| data type | key operations | data structure |
|---|---|---|
| **stack** | PUSH, POP | *linked list, resizing array* |
| **queue** | ENQUEUE, DEQUEUE | *linked list, resizing array* |

# Collections

A collection is a data types that store groups of items.

| data type | key operations | data structure |
|---|---|---|
| **stack** | PUSH, POP | *linked list, resizing array* |
| **queue** | ENQUEUE, DEQUEUE | *linked list, resizing array* |
| **priority queue** | INSERT, DELETE-MAX | *binary heap* |

# Collections

A collection is a data types that store groups of items.

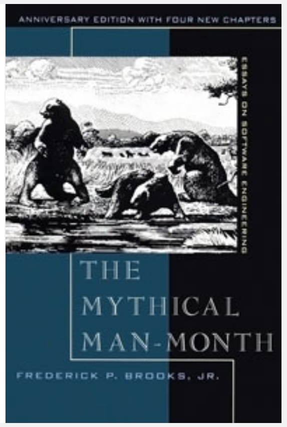
| data type | key operations | data structure |
|---|---|---|
| **stack** | PUSH, POP | *linked list, resizing array* |
| **queue** | ENQUEUE, DEQUEUE | *linked list, resizing array* |
| **priority queue** | INSERT, DELETE-MAX | *binary heap* |
| **symbol table** | PUT, GET, DELETE | *BST, hash table* |
| **set** | ADD, CONTAINS, DELETE | *BST, hash table* |

# Collections

A collection is a data types that store groups of items.

| data type | key operations | data structure |
|---|---|---|
| **stack** | PUSH, POP | *linked list, resizing array* |
| **queue** | ENQUEUE, DEQUEUE | *linked list, resizing array* |
| **priority queue** | INSERT, DELETE-MAX | *binary heap* |
| **symbol table** | PUT, GET, DELETE | *BST, hash table* |
| **set** | ADD, CONTAINS, DELETE | *BST, hash table* |

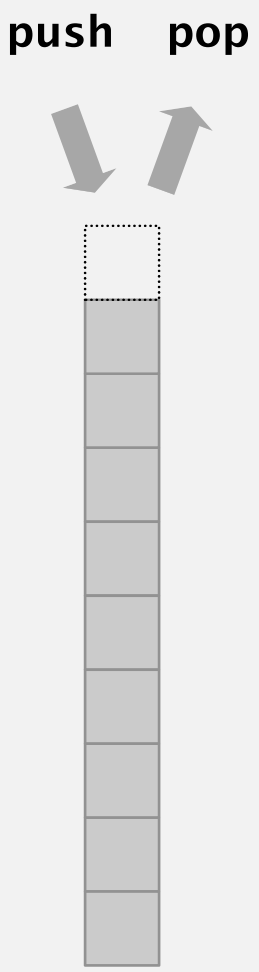
*“ Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.”* — *Fred Brooks*


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

# Stack API

Warmup API. Stack of strings data type.

| public class Stack<T> | | |
|---|---|---|
| | Stack() | *create an empty stack* |
| void | push(T item) | *insert a new item onto stack* |
| T | pop() | *remove and return the item most recently added* |
| boolean | isEmpty() | *is the stack empty?* |
| int | size() | *number of strings on the stack* |

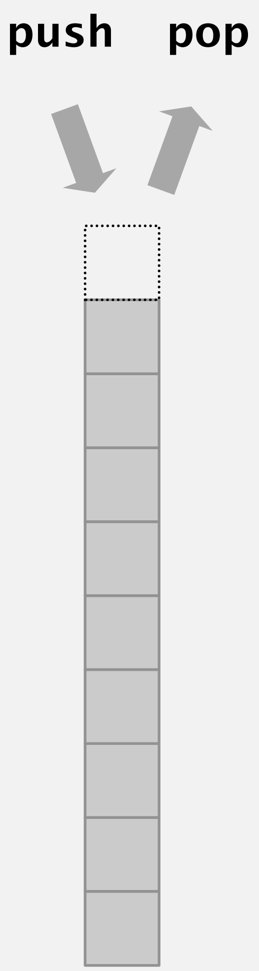


# Stack API

Warmup API. Stack of strings data type.

public class Stack<T>

| | | |
|---|---|---|
| | Stack() | *create an empty stack* |
| void | push(T item) | *insert a new item onto stack* |
| T | pop() | *remove and return the item most recently added* |
| boolean | isEmpty() | *is the stack empty?* |
| int | size() | *number of strings on the stack* |

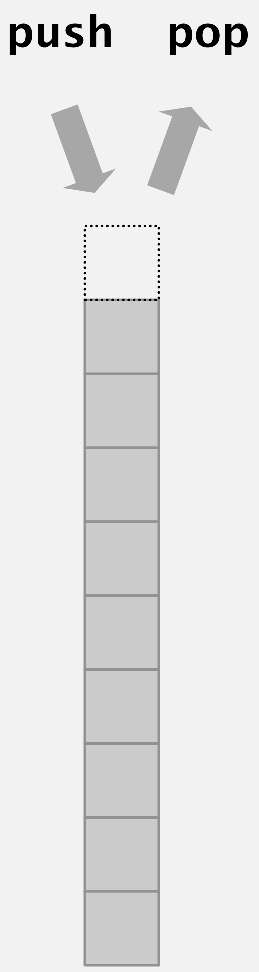


Warmup client. Reverse sequence of strings from standard input.

# Sample client

Warmup client. Reverse sequence of strings from standard input.

- Read string and push onto stack.
- Pop string and print.



```
public class ReverseStrings
{
   public static void main(String[] args)
   {
      Stack<String> stack = new Stack<>();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readString());
      while (!stack.isEmpty())
         StdOu
   }
}
```

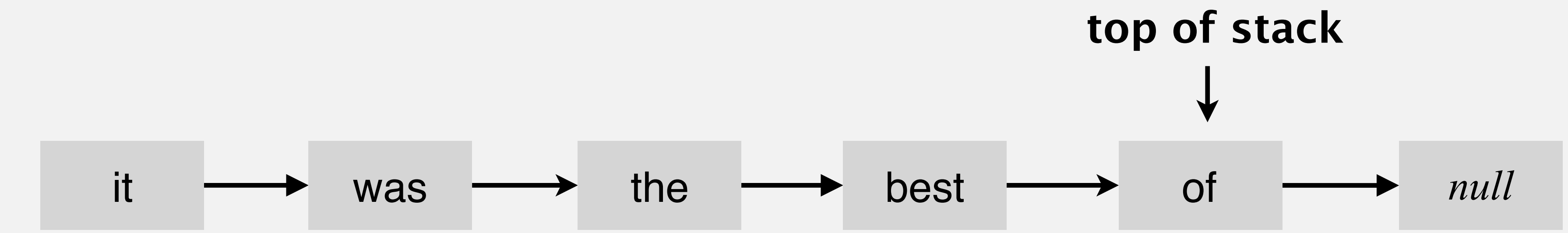
```
% more tinyTale.txt
it was the best of times ...


% java ReverseStrings < tinyTale.txt
... times of best the was it
[ignoring newlines]
```

# How to implement a stack with a linked list?

A. Can't be done efficiently with a singly-linked list.

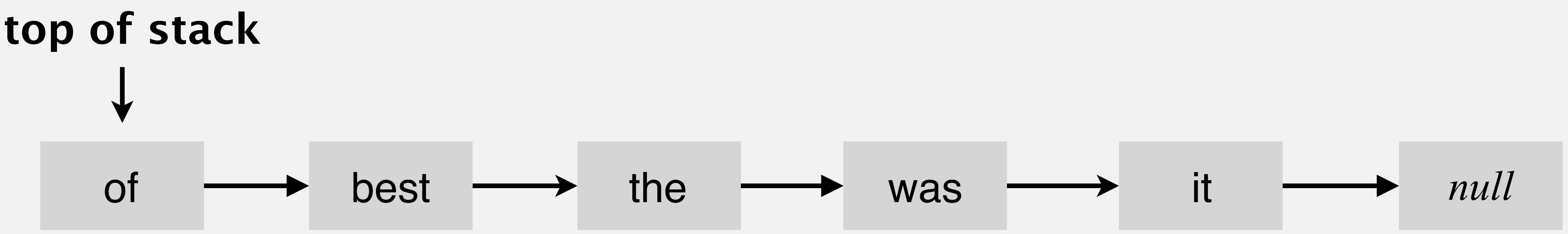
B.



top of stack

it → was → the → best → of → *null*

C.

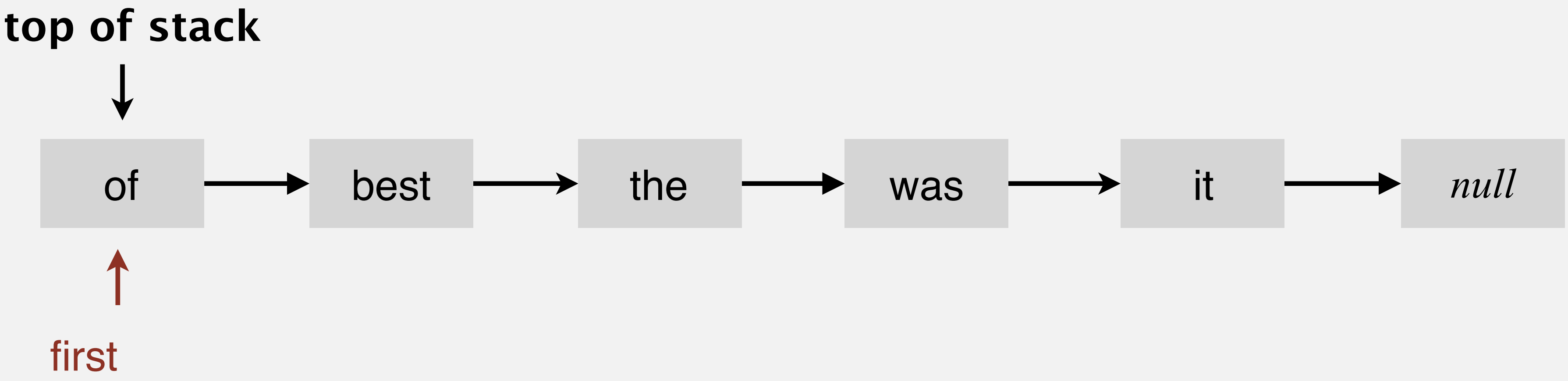


top of stack

of → best → the → was → it → *null*

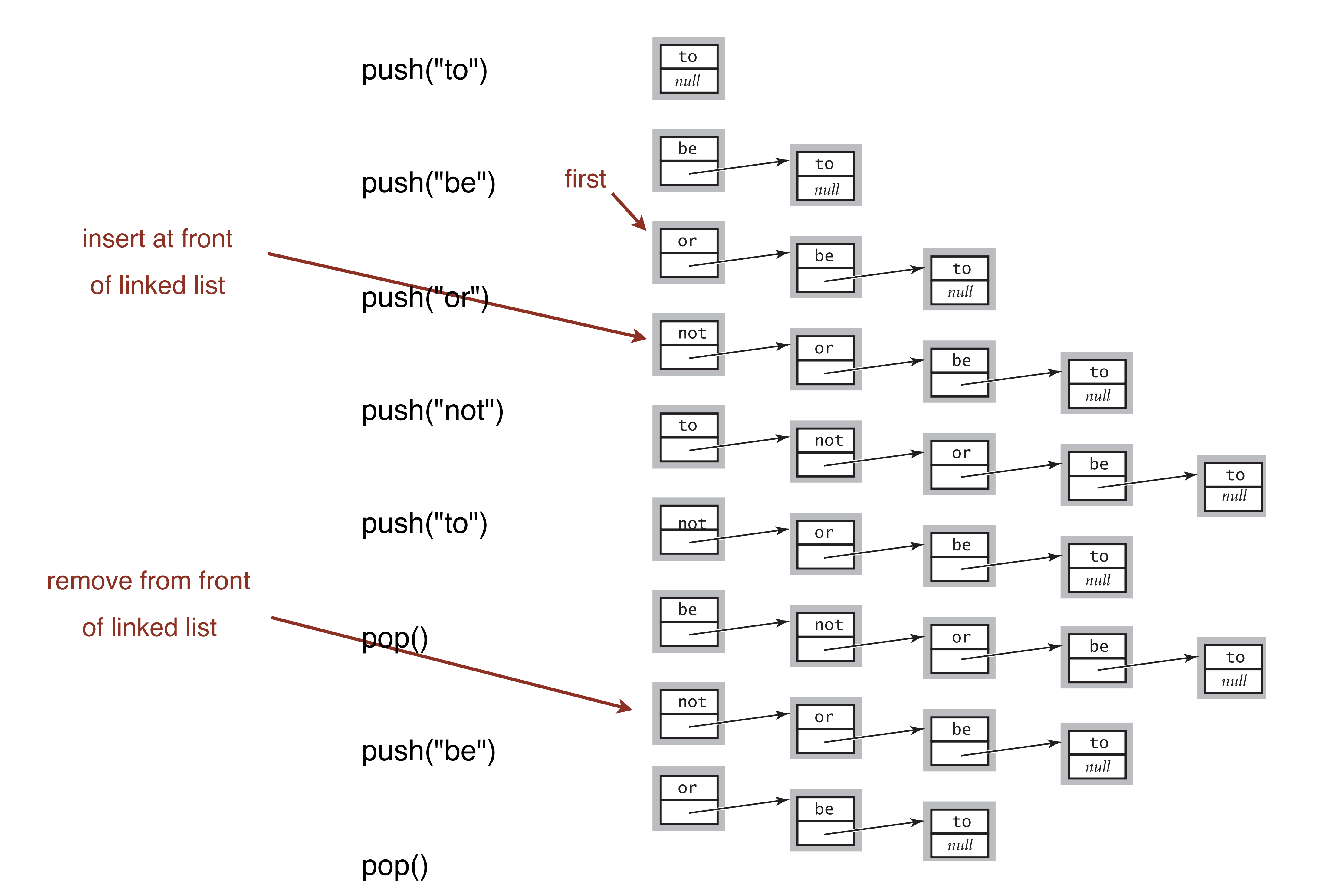
# Stack: linked-list implementation

- Maintain pointer first to first node in a singly-linked list.
- Push new item before first.
- Pop item from first.



# Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.



# Stack pop: linked-list implementation

inner class

```
private class Node
{
   String item;
   Node next;
}
```

# Stack pop: linked-list implementation

inner class

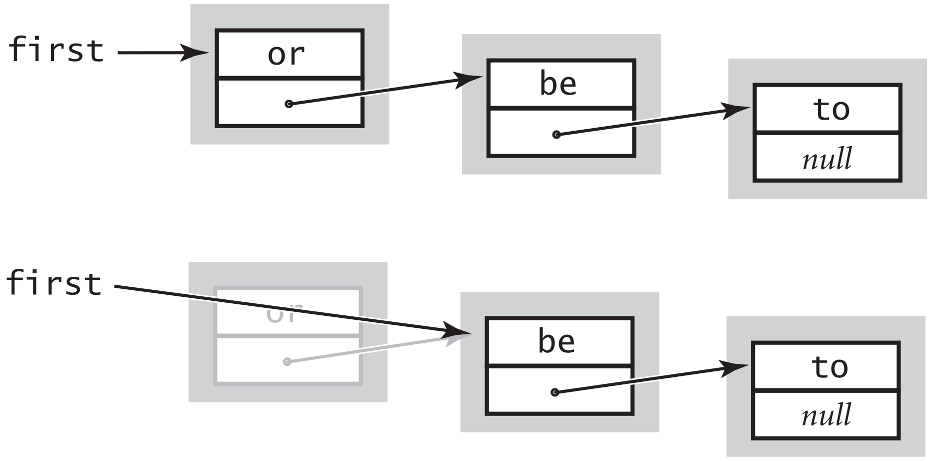
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

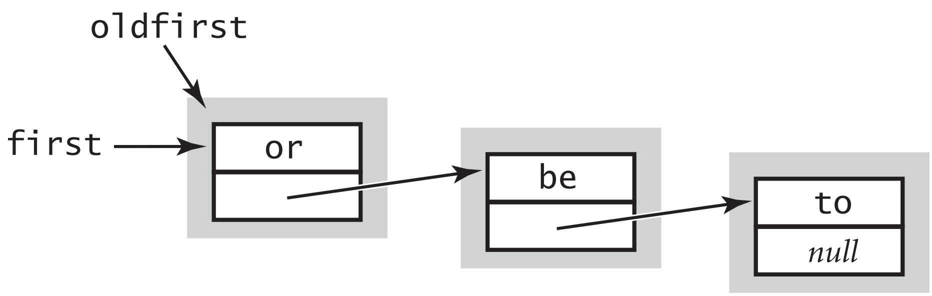
# Stack push: linked-list implementation

inner class

```
private class Node
{
   String item;
   Node next;
}
```

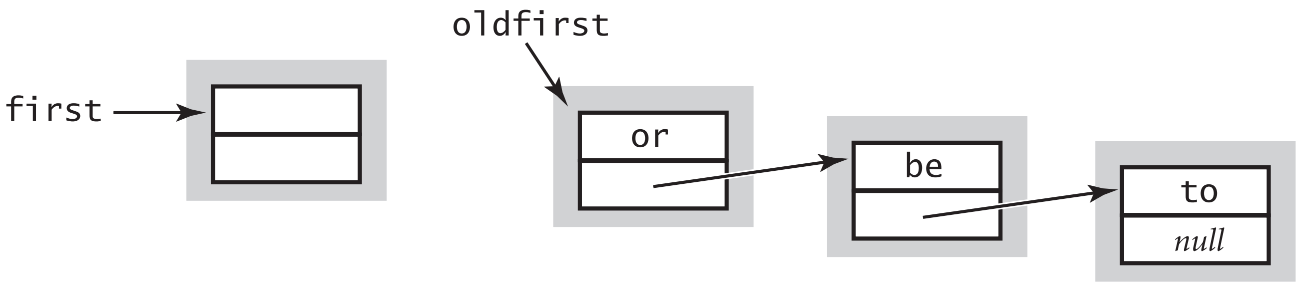
save a link to the list

```
Node oldfirst = first;
```



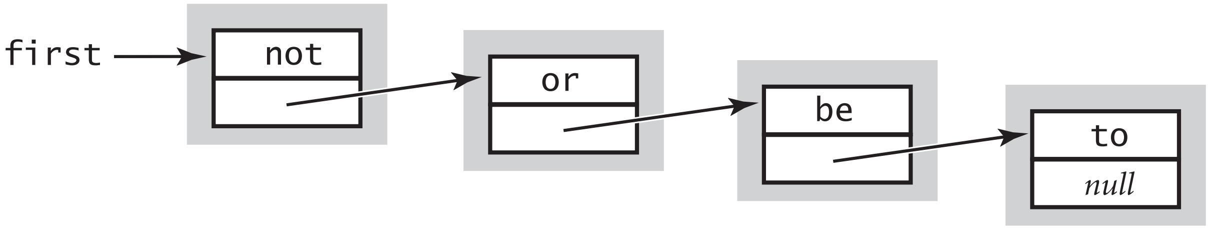
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

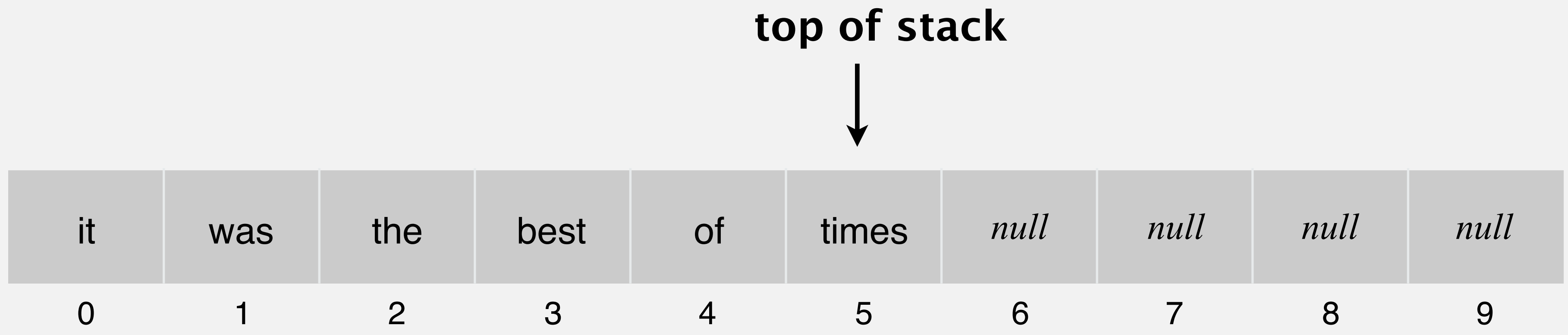
```
first.item = "not";
first.next = oldfirst;
```



# How to implement a fixed-capacity stack with an array?

A. Can't be done efficiently with an array.

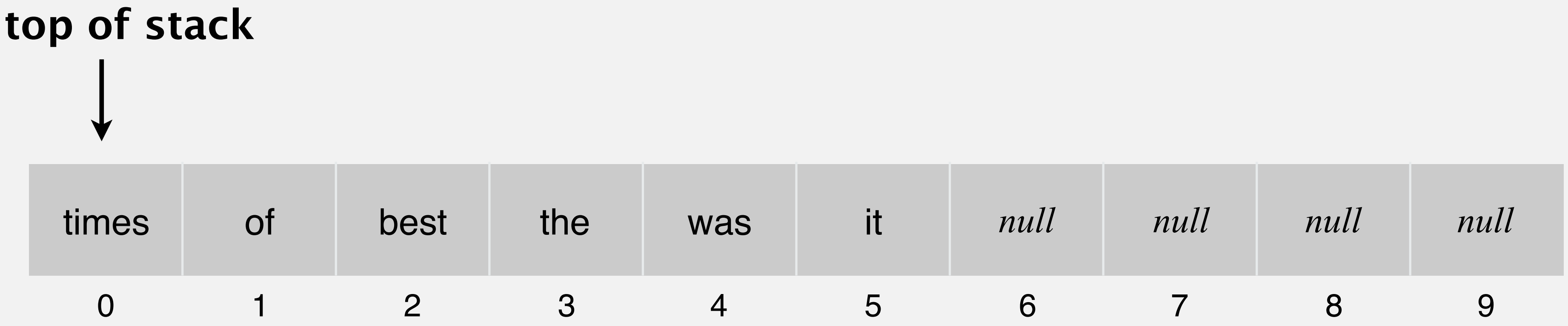
B.



top of stack

| it | was | the | best | of | times | *null* | *null* | *null* | *null* |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

C.

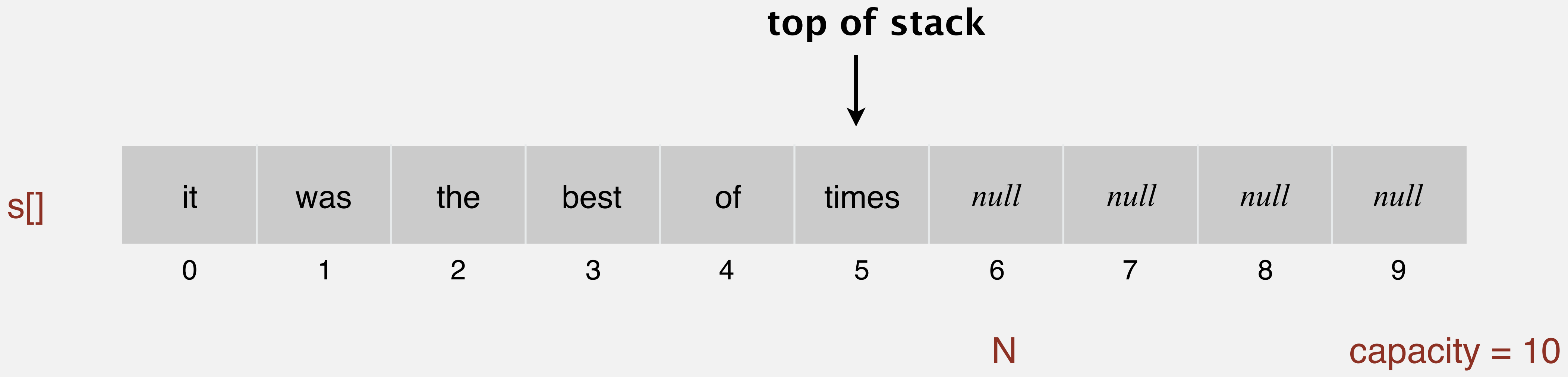


top of stack

| times | of | best | the | was | it | *null* | *null* | *null* | *null* |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Fixed-capacity stack: array implementation

- Use array s[] to store N items on stack.
- push(): add new item at s[N].
- pop(): remove item from s[N-1].



top of stack

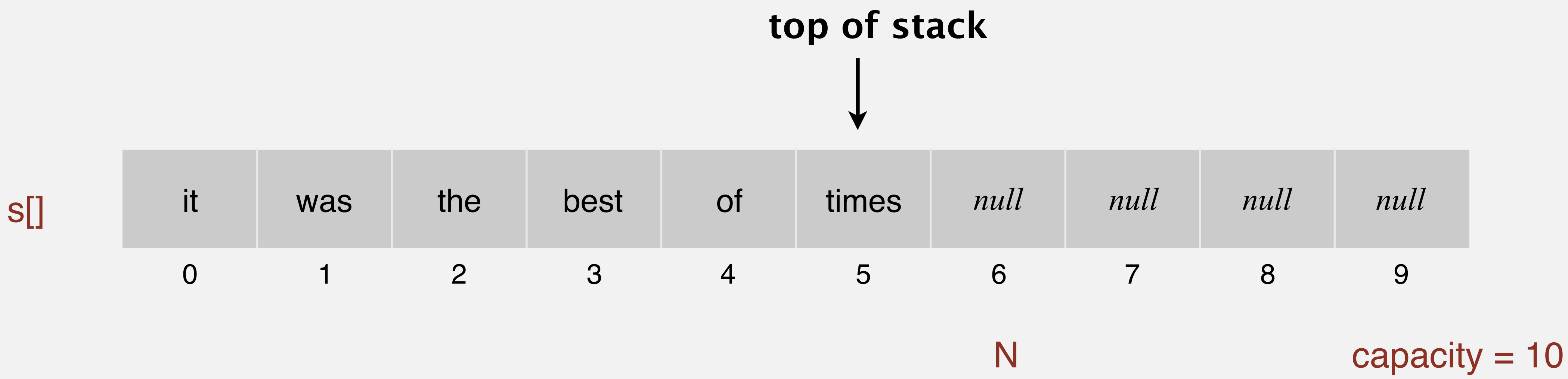
| s[] | it | was | the | best | of | times | *null* | *null* | *null* | *null* |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

N

capacity = 10

# Fixed-capacity stack: array implementation

- Use array s[] to store N items on stack.
- push(): add new item at s[N].
- pop(): remove item from s[N-1].



top of stack

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| s[] | it | was | the | best | of | times | *null* | *null* | *null* | *null* |
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

N

capacity = 10

Defect. Stack overflows when N exceeds capacity. [stay tuned]

# Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

# Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

# Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

```
public String pop()
{  return s[--N];  }
```

**loitering**

# Stack considerations

Overflow and underflow.

- Underflow:  throw exception if pop from an empty stack.
- Overflow:  use resizing array for array implementation.  [stay tuned]

Null items.  We allow null items to be inserted.

Loitering.  Holding a reference to an object when it is no longer needed.

```
public String pop()
{  return s[--N];  }
```

**loitering**

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

**this version avoids "loitering":**
**garbage collector can reclaim memory for**
**an object only if no outstanding references**

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

Too expensive.

- Need to copy all items to a new array, for each operation.

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first $N$ items = $N + \underbrace{(2 + 4 + \ldots + 2(N - 1))}_{\text{2(k–1) array accesses to expand to size k (ignoring cost to create new array)}} \sim \overset{\text{infeasible for large N}}{N^2}$.

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first $N$ items = $N + \underbrace{(2 + 4 + \ldots + 2(N-1))}_{\text{2(k–1) array accesses to expand to size k (ignoring cost to create new array)}} \sim \overset{\text{infeasible for large N}}{N^2}$.

Challenge. Ensure that array resizing happens infrequently.

# Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

Too expensive.

infeasible for large N

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first $N$ items = $N + (2 + 4 + \ldots + 2(N-1)) \sim N^2$.

2(k–1) array accesses to expand to size k
(ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Q. How to grow array?

A. If array is full, create a new array of twice the size, and copy items.

# Stack: resizing-array implementation

Q. How to shrink array?

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.

N = 5

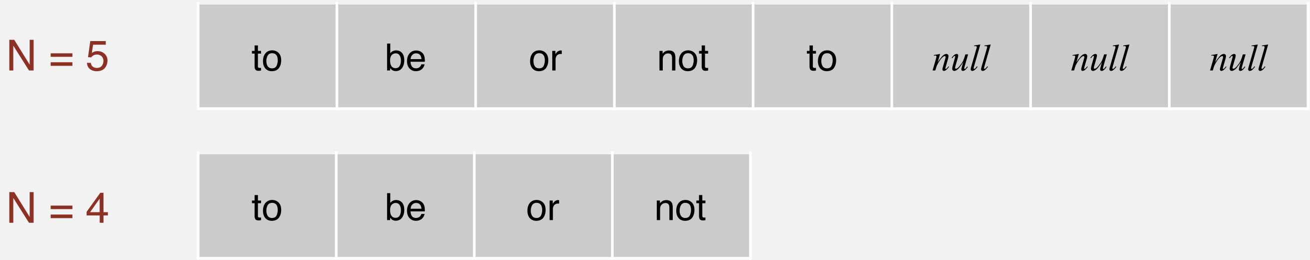
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| to | be | or | not | to | *null* | *null* | *null* |

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.



|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| N = 5 | to | be | or | not | to | *null* | *null* | *null* |
| N = 4 | to | be | or | not |  |  |  |  |

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N = 5 | to | be | or | not | to | *null* | *null* | *null* |
| N = 4 | to | be | or | not | | | | |
| N = 5 | to | be | or | not | to | *null* | *null* | *null* |

# Stack: resizing-array implementation

Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N = 5 | to | be | or | not | to | *null* | *null* | *null* |
| N = 4 | to | be | or | not | | | | |
| N = 5 | to | be | or | not | to | *null* | *null* | *null* |
| N = 4 | to | be | or | not | | | | |

# Stack: resizing-array implementation

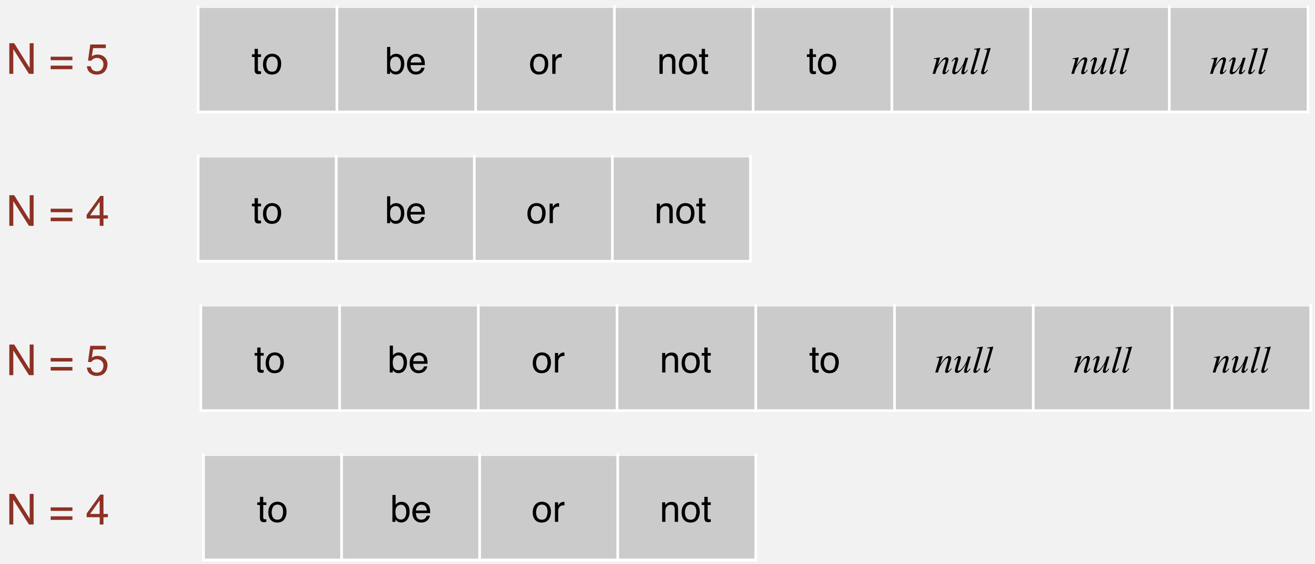
Q. How to shrink array?

First try.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-half full.

Too expensive in worst case.

- Consider push-pop-push-pop-… sequence when array is full.
- Each operation takes time proportional to $N$.



N = 5 to be or not to *null* *null* *null*

N = 4 to be or not

N = 5 to be or not to *null* *null* *null*

N = 4 to be or not

# Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-quarter full.

# Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-quarter full.

Invariant. Array is between 25% and 100% full.

# Stack: resizing-array implementation trace

| push() | pop() | N | a.length | a[] | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | | 0 | 1 | *null* | | | | | | | |
| to | | 1 | 1 | to | | | | | | | |
| be | | 2 | 2 | to | be | | | | | | |
| or | | 3 | 4 | to | be | or | *null* | | | | |
| not | | 4 | 4 | to | be | or | not | | | | |
| to | | 5 | 8 | to | be | or | not | to | *null* | *null* | *null* |
| - | to | 4 | 8 | to | be | or | not | *null* | *null* | *null* | *null* |
| be | | 5 | 8 | to | be | or | not | be | *null* | *null* | *null* |
| - | be | 4 | 8 | to | be | or | not | *null* | *null* | *null* | *null* |
| - | not | 3 | 8 | to | be | or | *null* | *null* | *null* | *null* | *null* |
| that | | 4 | 8 | to | be | or | that | *null* | *null* | *null* | *null* |
| - | that | 3 | 8 | to | be | or | *null* | *null* | *null* | *null* | *null* |
| - | or | 2 | 4 | to | be | *null* | *null* | | | | |
| - | be | 1 | 2 | to | *null* | | | | | | |
| is | | 2 | 2 | to | is | | | | | | |

Trace of array resizing during a sequence of push() and pop() operations

# Stack resizing-array implementation: performance

| | best | worst |
|---|---|---|
| construct | 1 | 1 |
| push | 1 | $N$ |
| pop | 1 | $N$ |
| size | 1 | 1 |

doubling and
halving operations

**order of growth of running time
for resizing stack with N items**

# Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.

| | best | worst |
|---|---|---|
| construct | 1 | 1 |
| push | 1 | $N$ |
| pop | 1 | $N$ |
| size | 1 | 1 |

doubling and halving operations

**order of growth of running time for resizing stack with N items**

# Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of $M$ push and pop operations takes time proportional to $M$.

| | best | worst | amortized |
|---|---|---|---|
| construct | $1$ | $1$ | $1$ |
| push | $1$ | $N$ | $1$ |
| pop | $1$ | $N$ | $1$ |
| size | $1$ | $1$ | $1$ |

doubling and halving operations

**order of growth of running time for resizing stack with N items**

# Stack implementations: resizing array vs. linked list

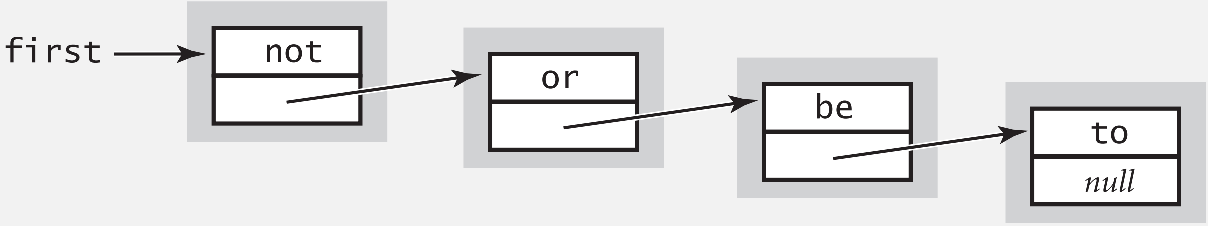
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

# Stack implementations: resizing array vs. linked list

Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.



# Stack implementations: resizing array vs. linked list

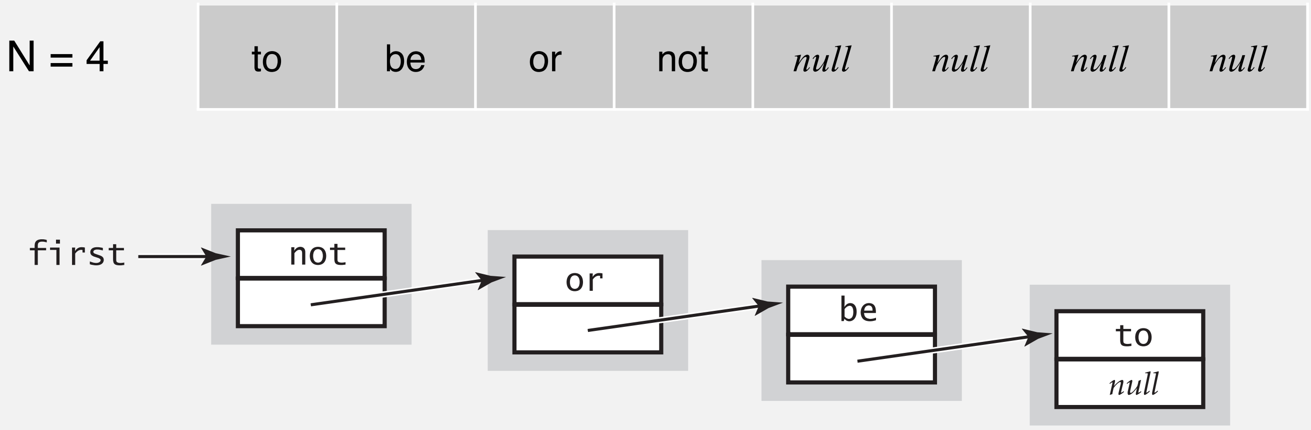
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant amortized time.
- Less wasted space.

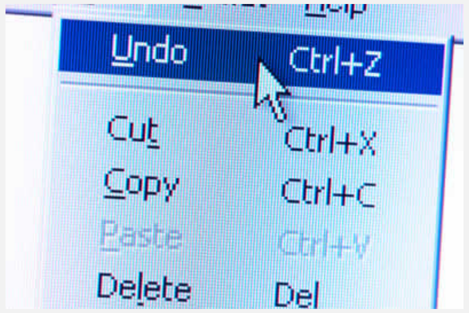


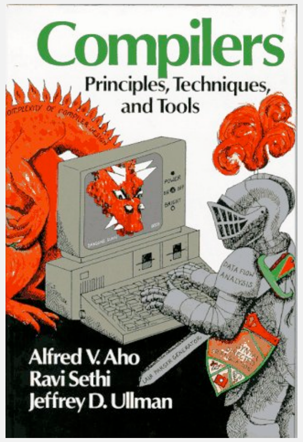
# Stack applications

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
- ...

Java

Adobe® PostScript® 3™





# Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.

# Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.

p = 216, q = 192

gcd (216, 192)

```
static int gcd(int p, int q) {
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

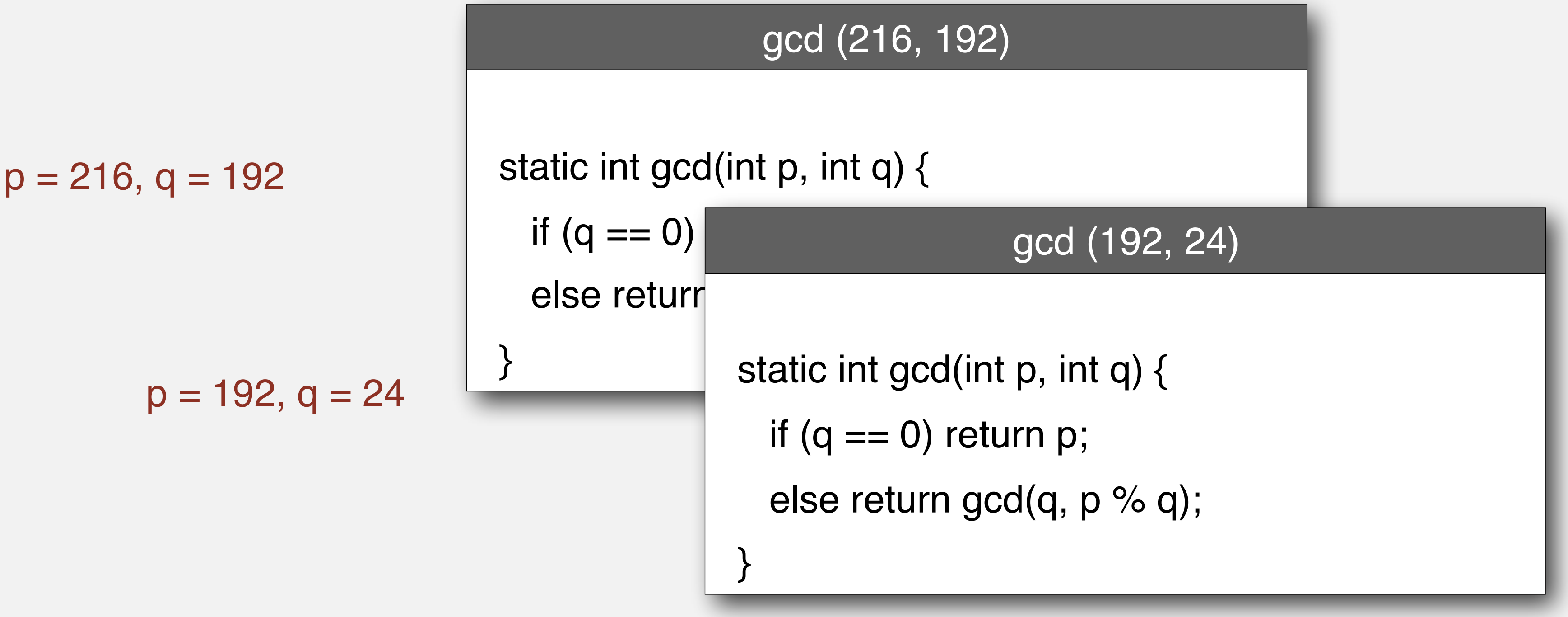
# Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



```
static int gcd(int p, int q) {
   if (q == 0)
   else returr
}
```

```
static int gcd(int p, int q) {
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

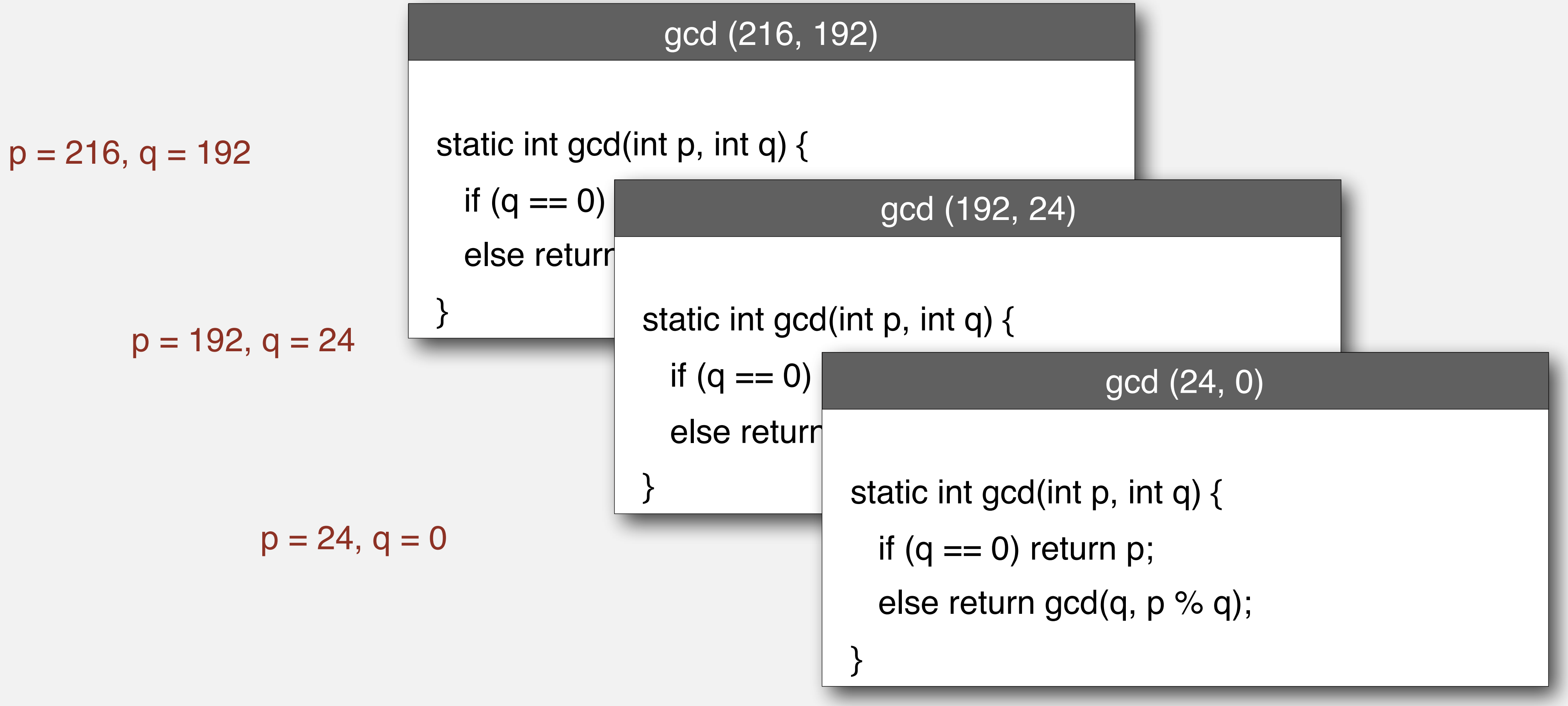
# Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



p = 216, q = 192

gcd (216, 192)

```
static int gcd(int p, int q) {
   if (q == 0)
   else return
}
```

p = 192, q = 24

gcd (192, 24)

```
static int gcd(int p, int q) {
   if (q == 0)
   else return
}
```

p = 24, q = 0

gcd (24, 0)

```
static int gcd(int p, int q) {
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

# Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

operand

operator

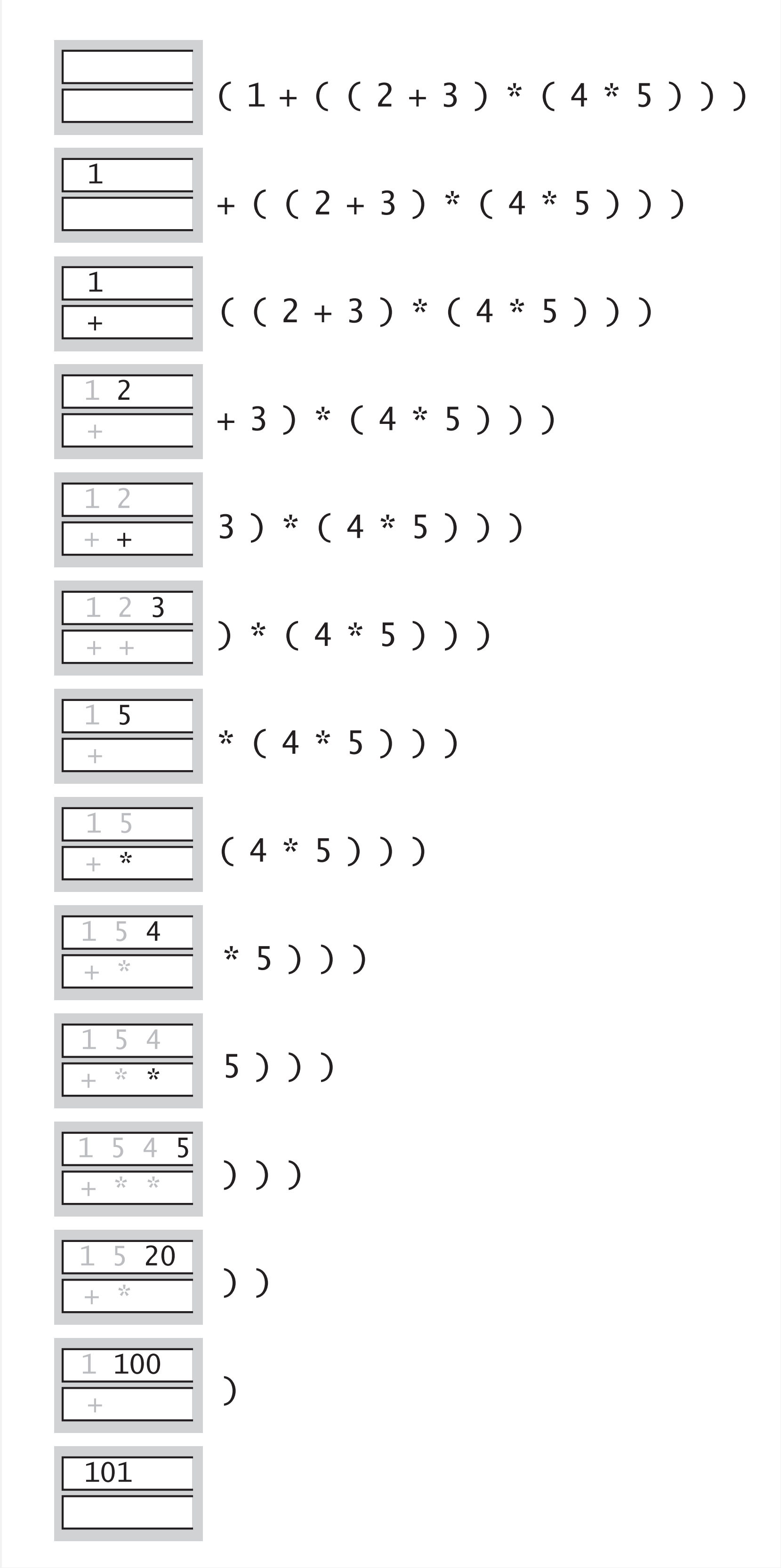
Two-stack algorithm. [E. W. Dijkstra]

- 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.

Context. An interpreter!

value stack

operator stack



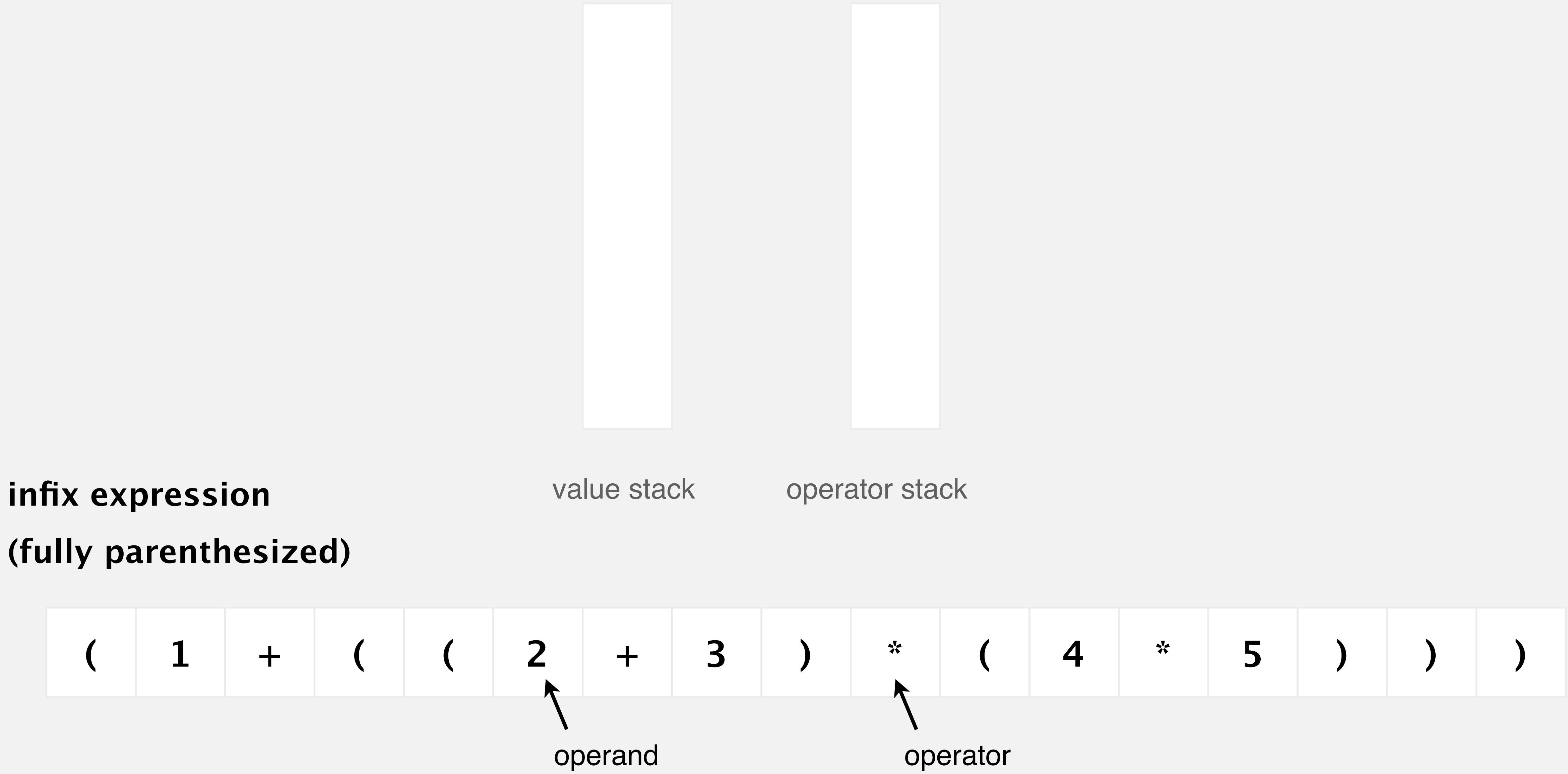
# Dijkstra's two-stack algorithm

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.



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

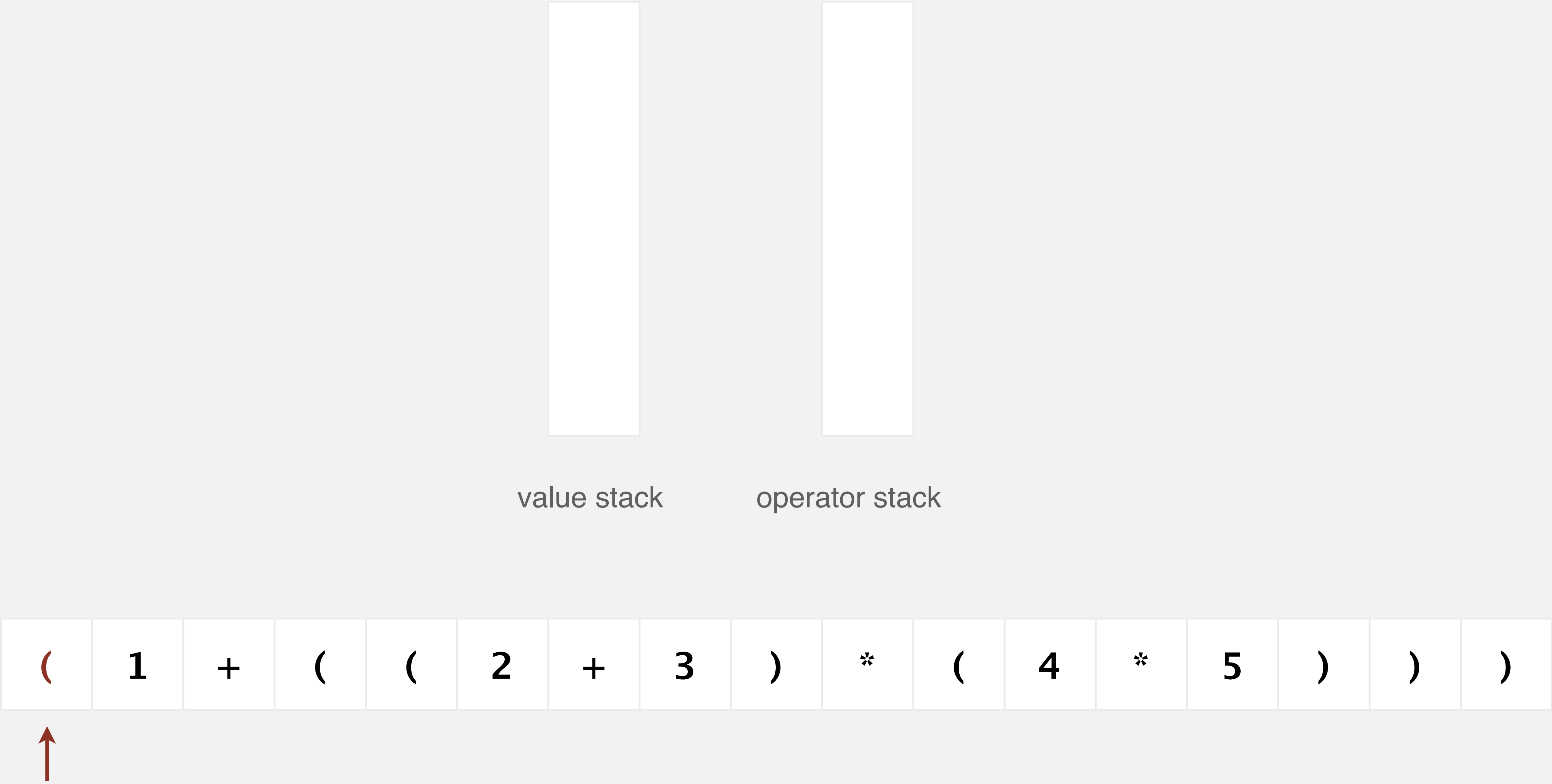
# Dijkstra's two-stack algorithm

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.



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

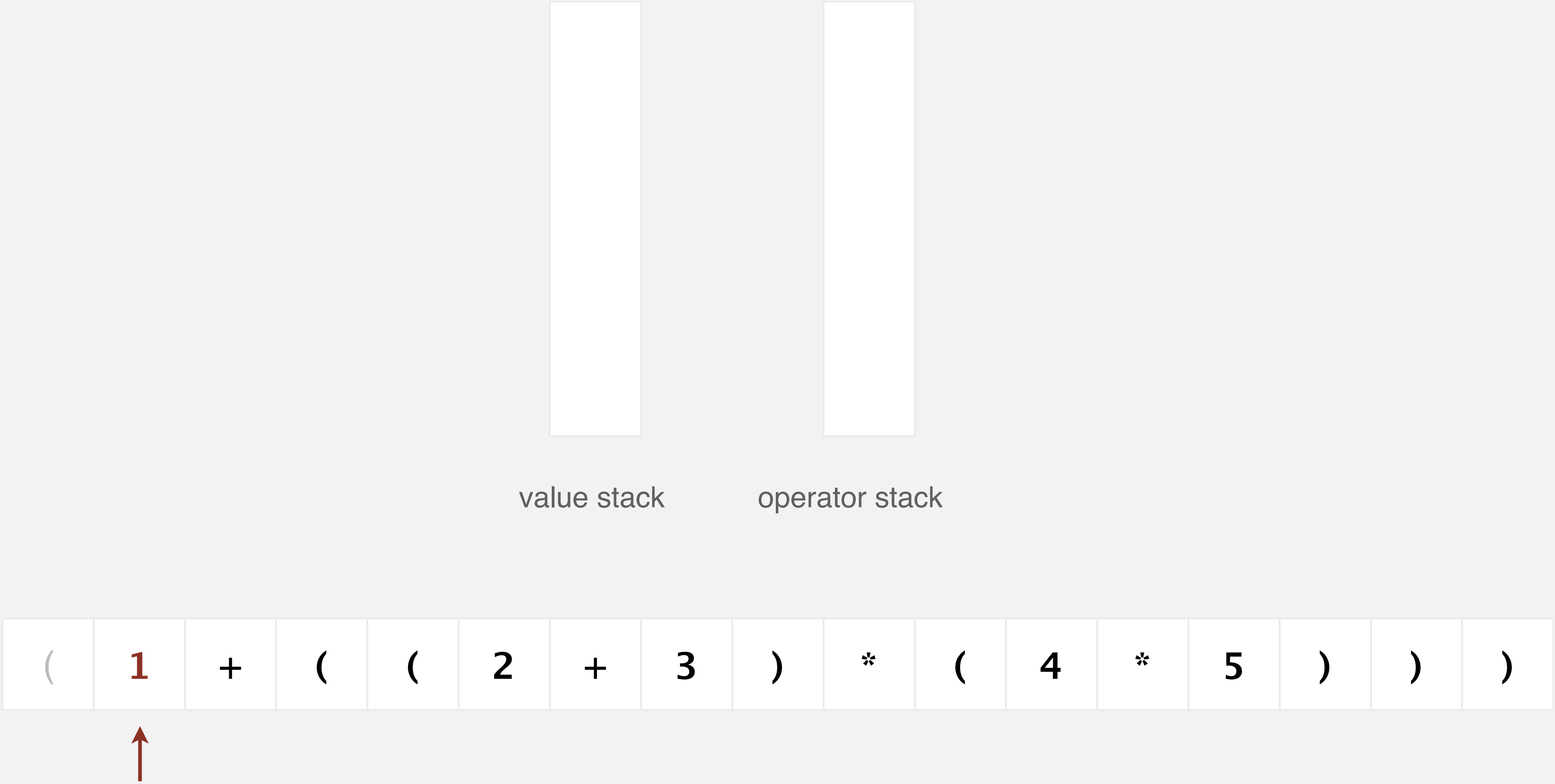
# Dijkstra's two-stack algorithm

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.



value stack

operator stack

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

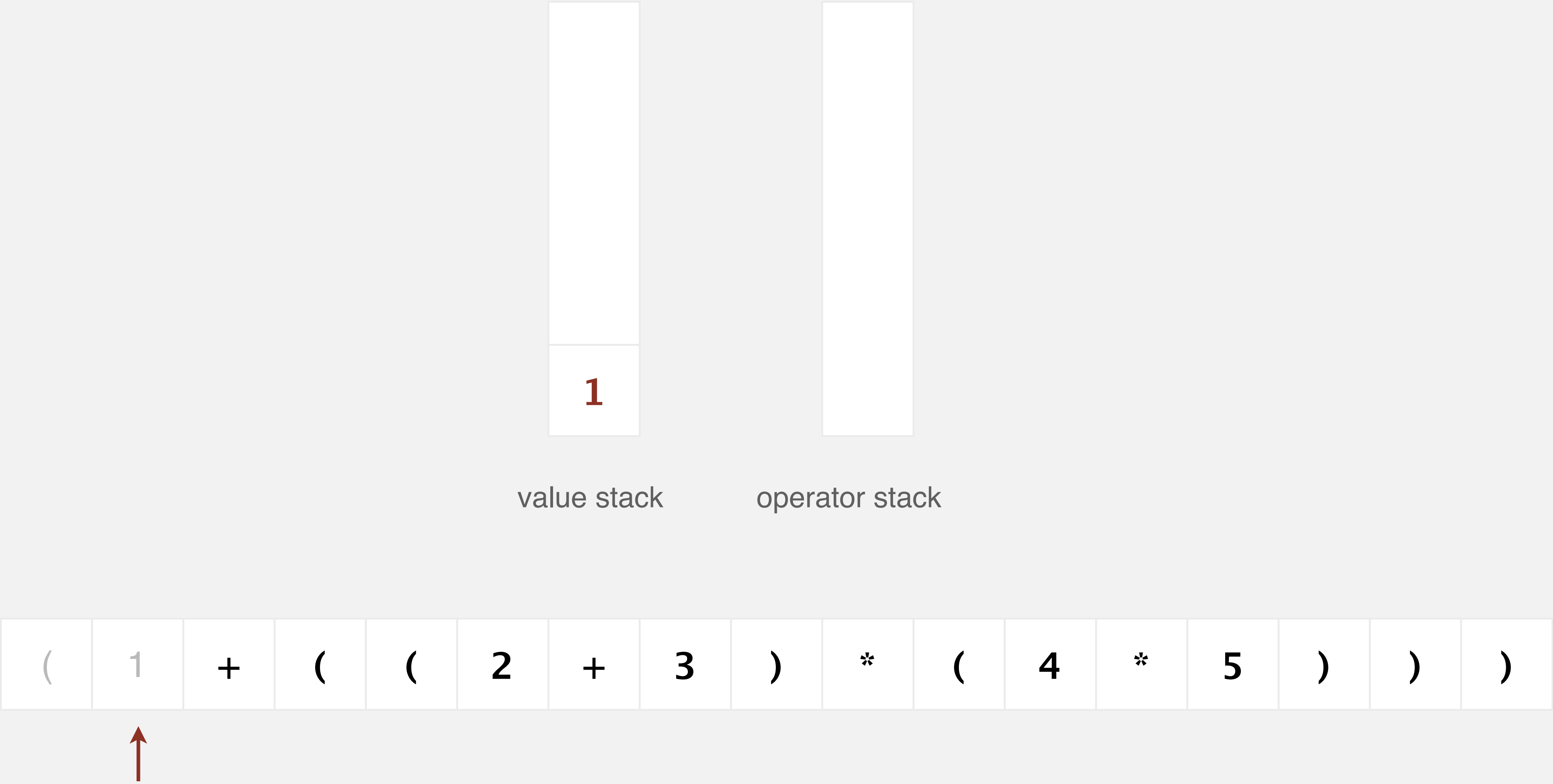
# Dijkstra's two-stack algorithm

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.



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

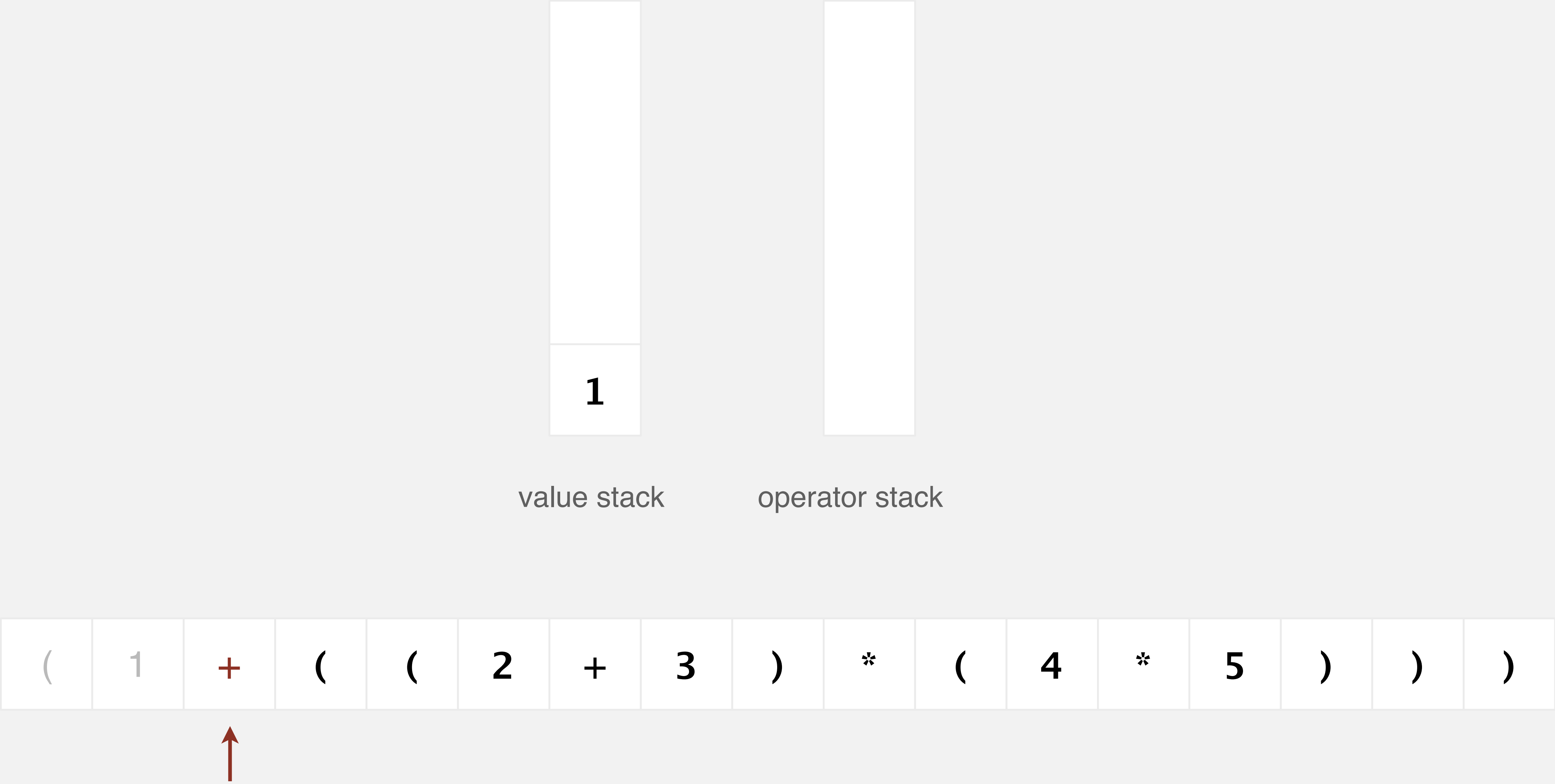
# Dijkstra's two-stack algorithm

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.



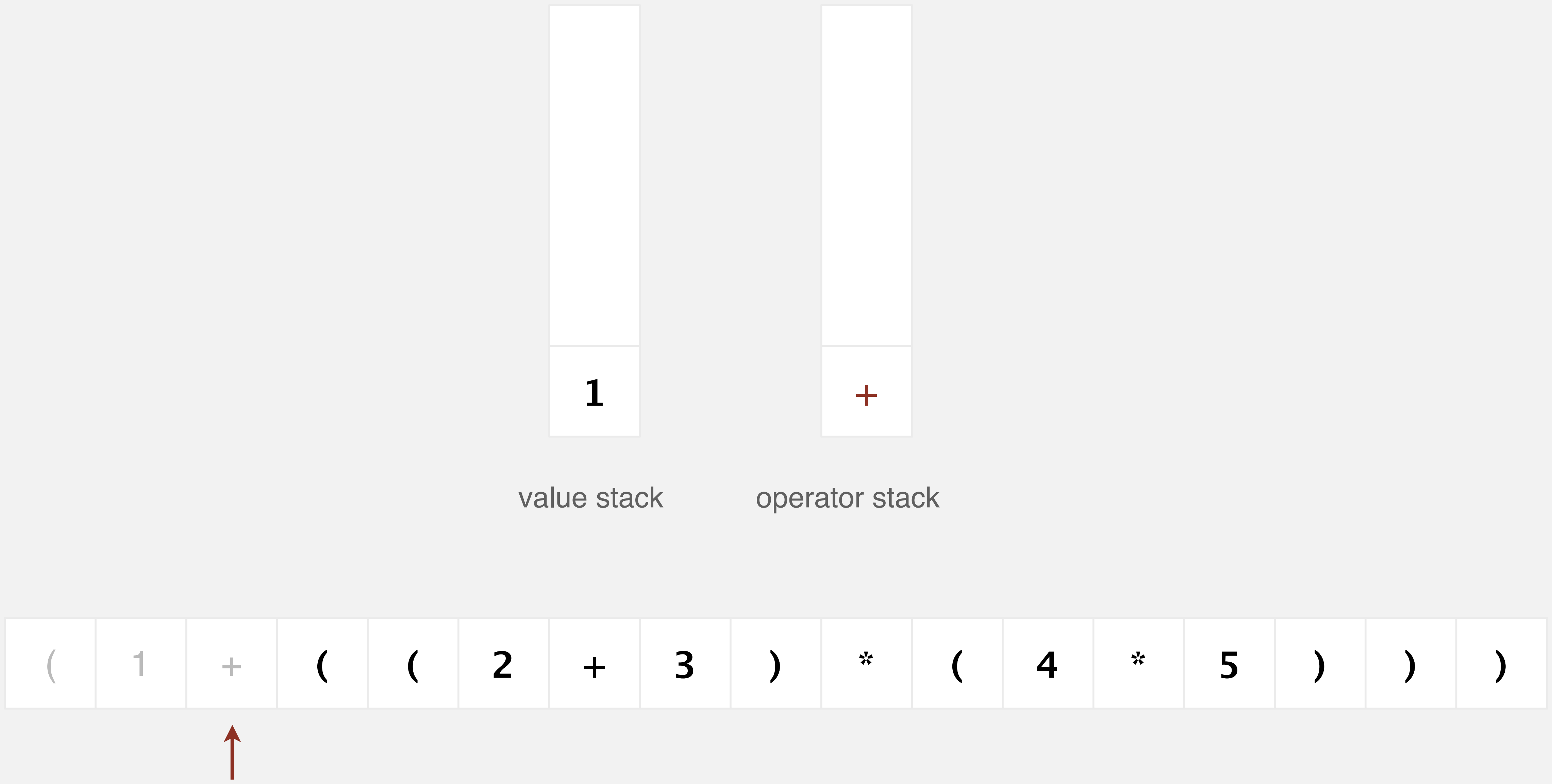
# Dijkstra's two-stack algorithm

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.



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

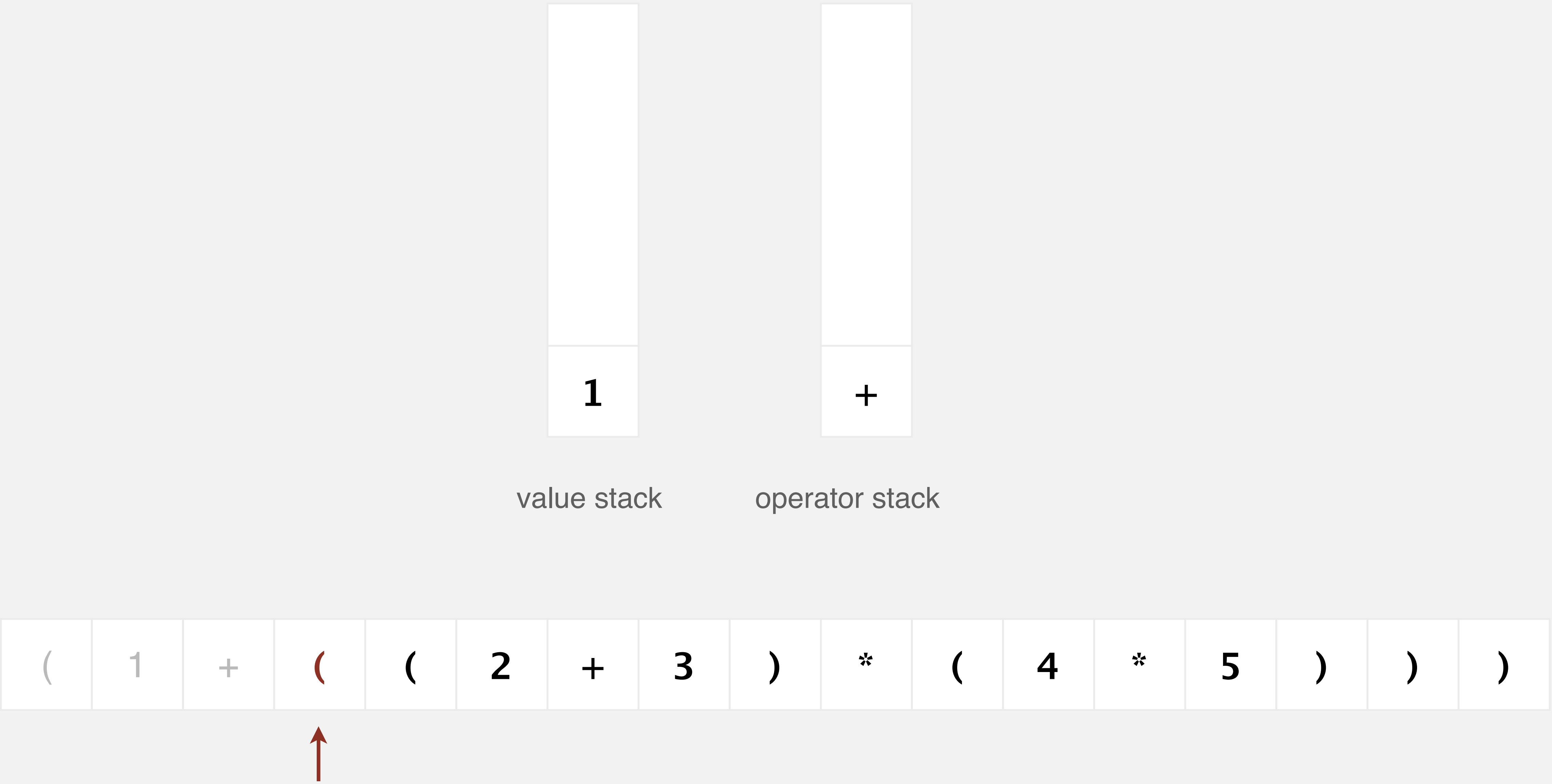
# Dijkstra's two-stack algorithm

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.



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

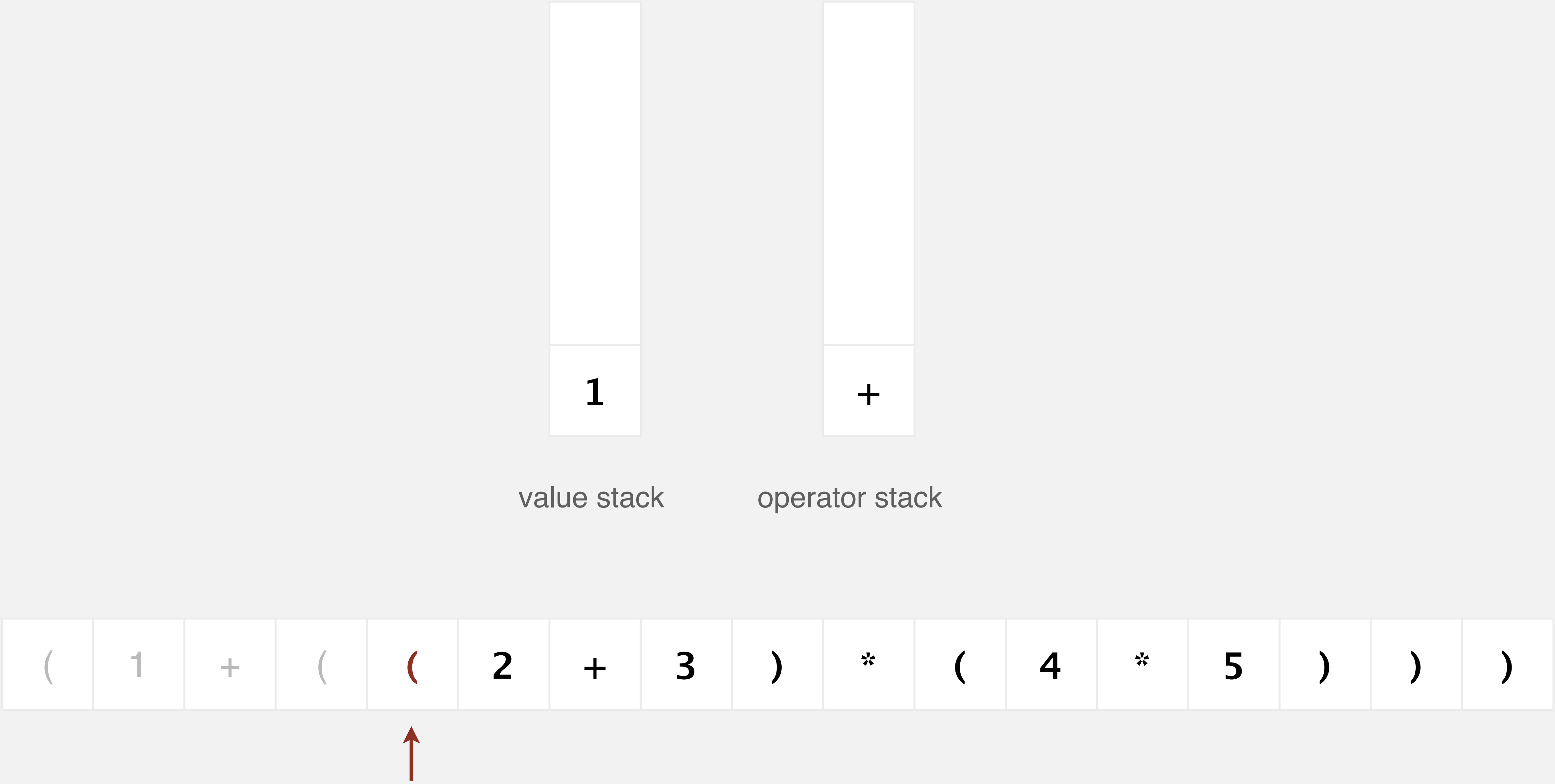
# Dijkstra's two-stack algorithm

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.



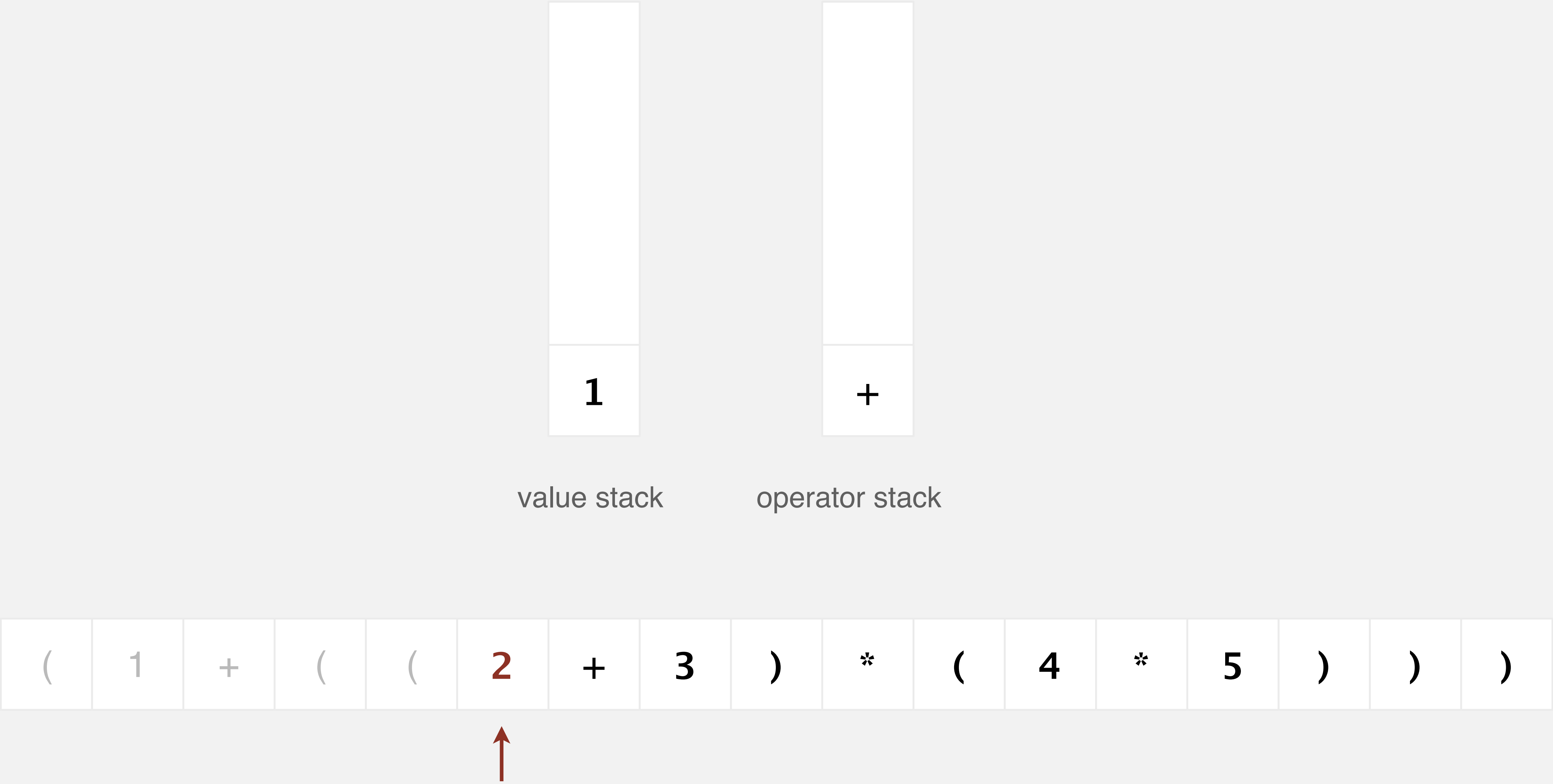
# Dijkstra's two-stack algorithm

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.



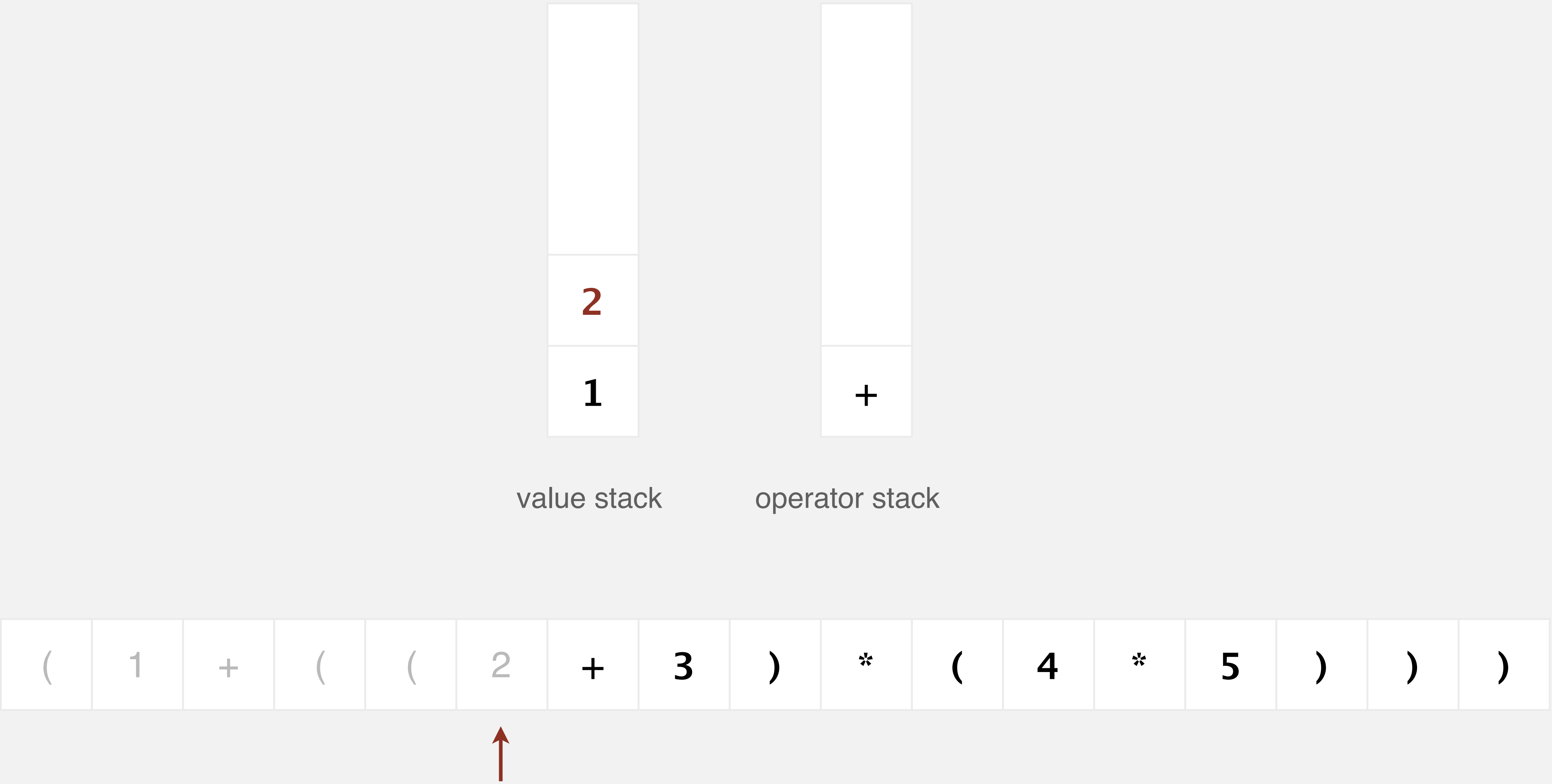
# Dijkstra's two-stack algorithm

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.



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

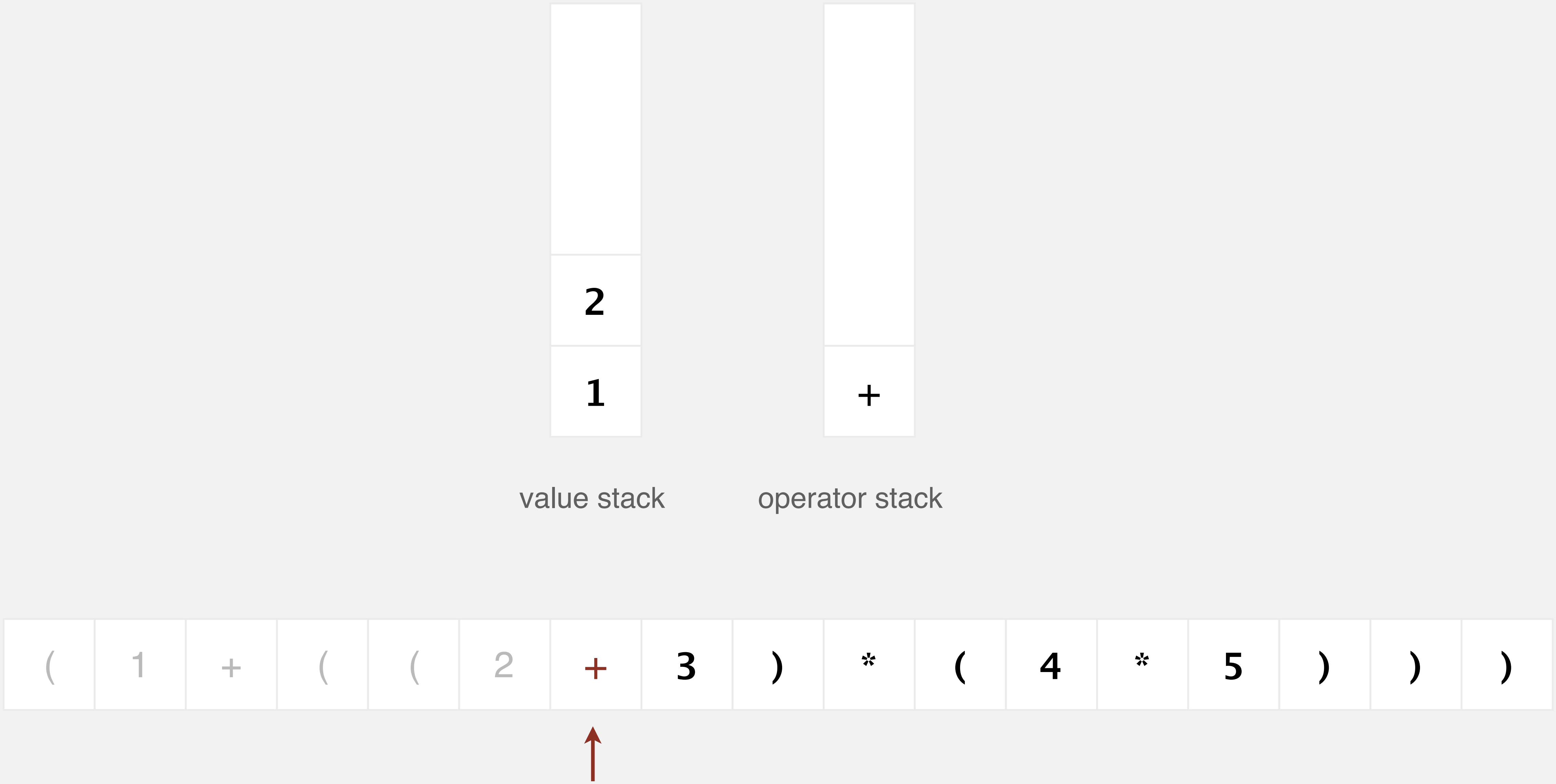
# Dijkstra's two-stack algorithm

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.



2

1

+

value stack

operator stack

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

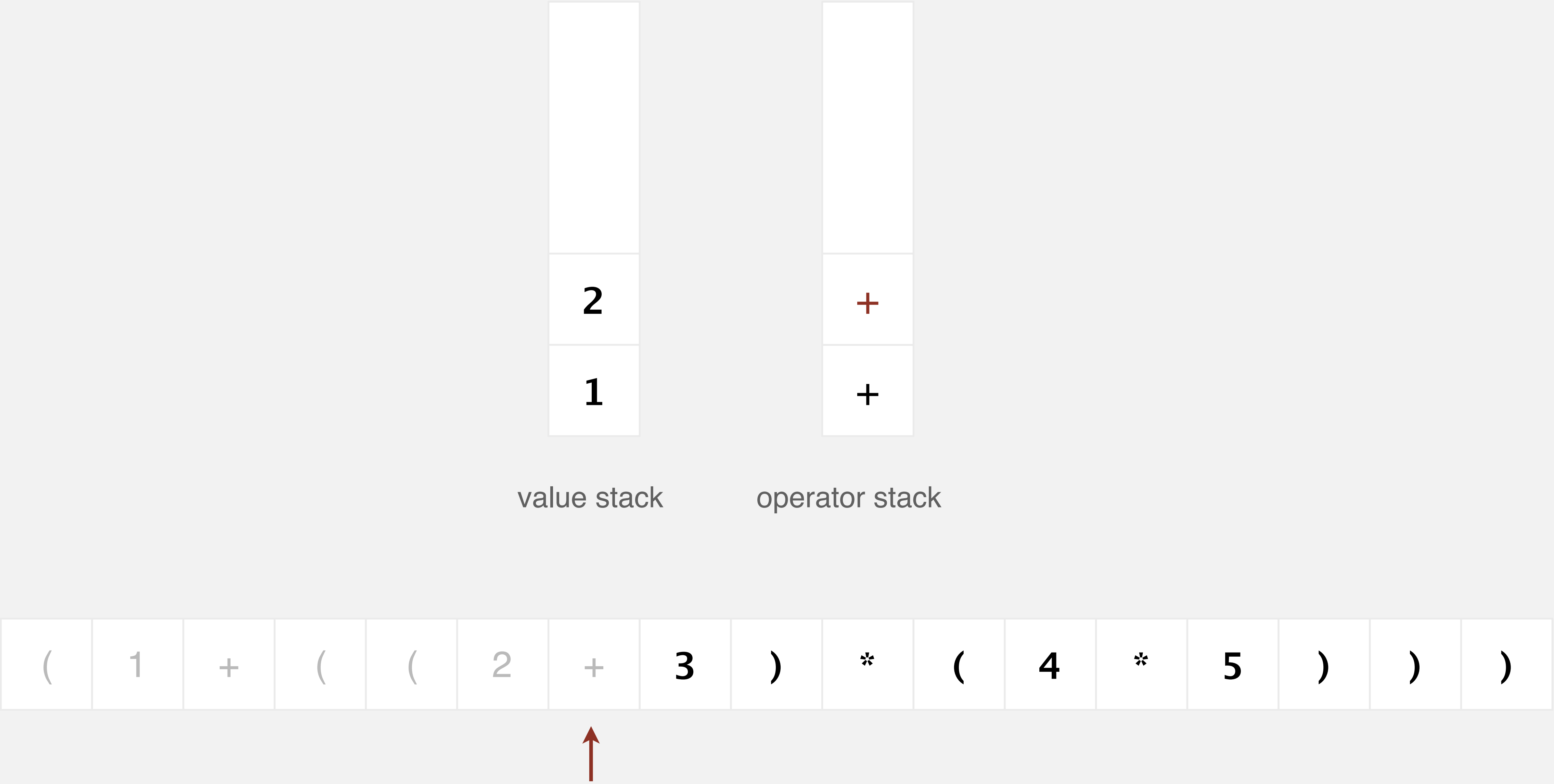
# Dijkstra's two-stack algorithm

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.



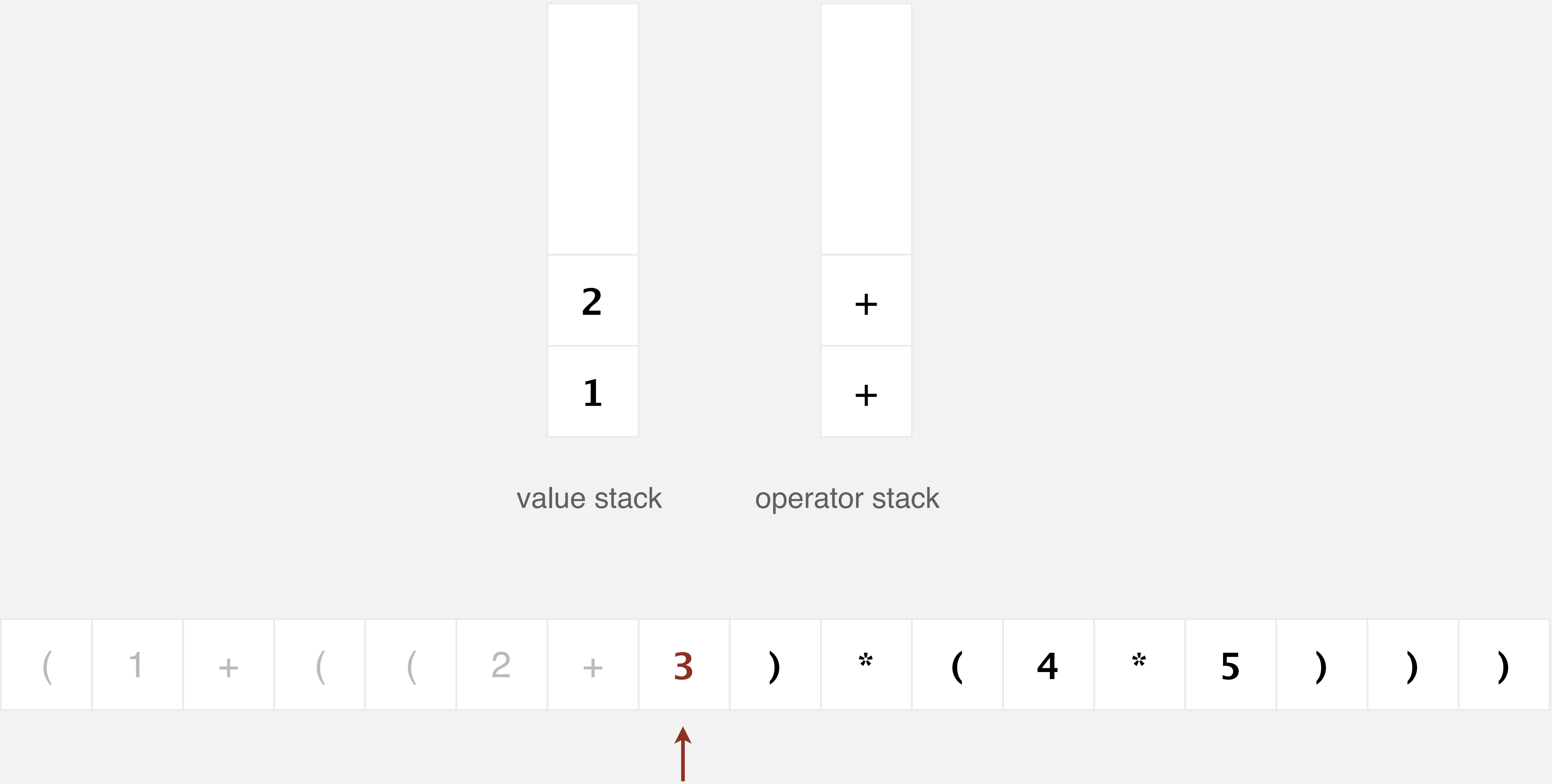
# Dijkstra's two-stack algorithm

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.



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

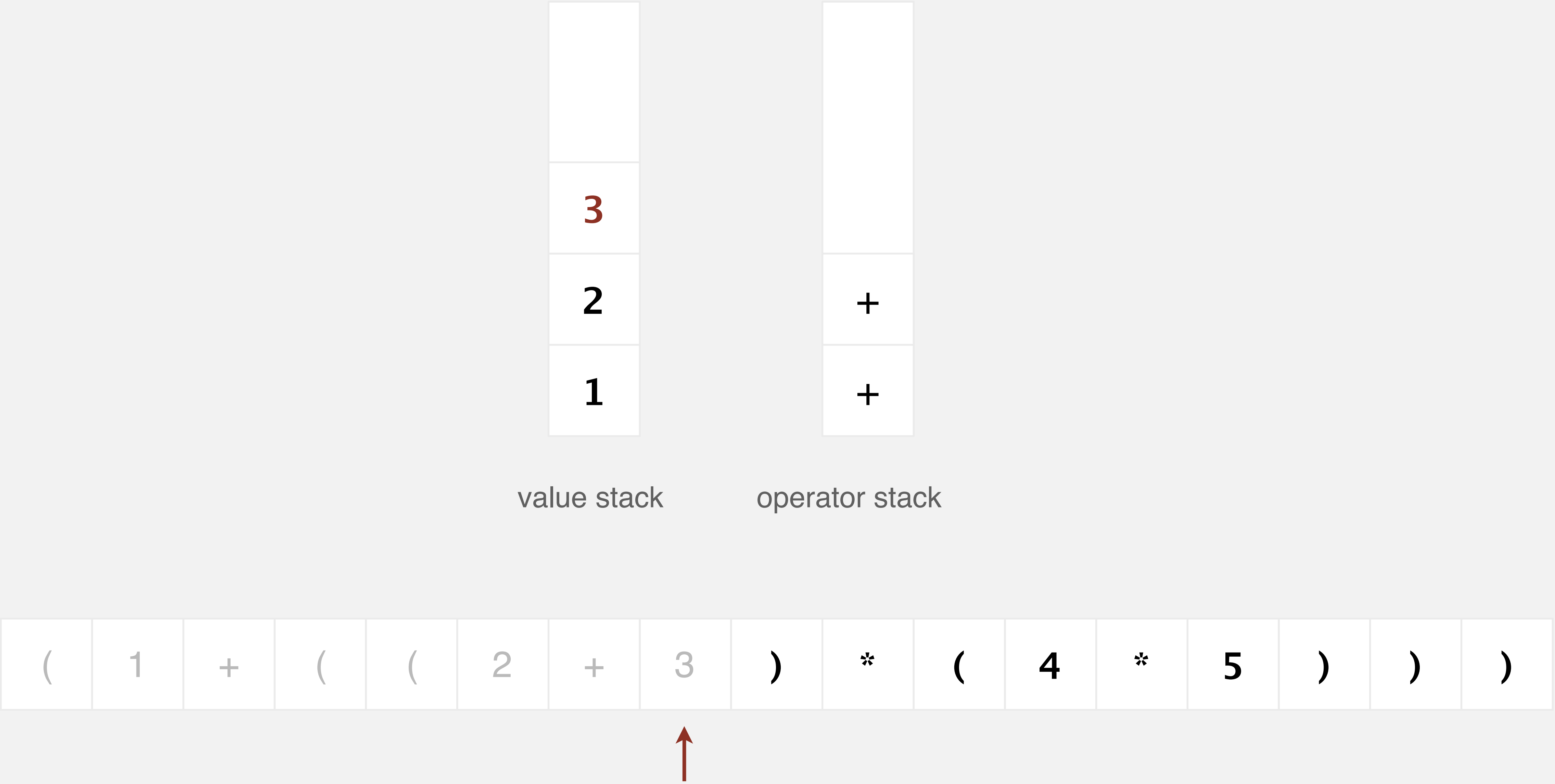
# Dijkstra's two-stack algorithm

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.



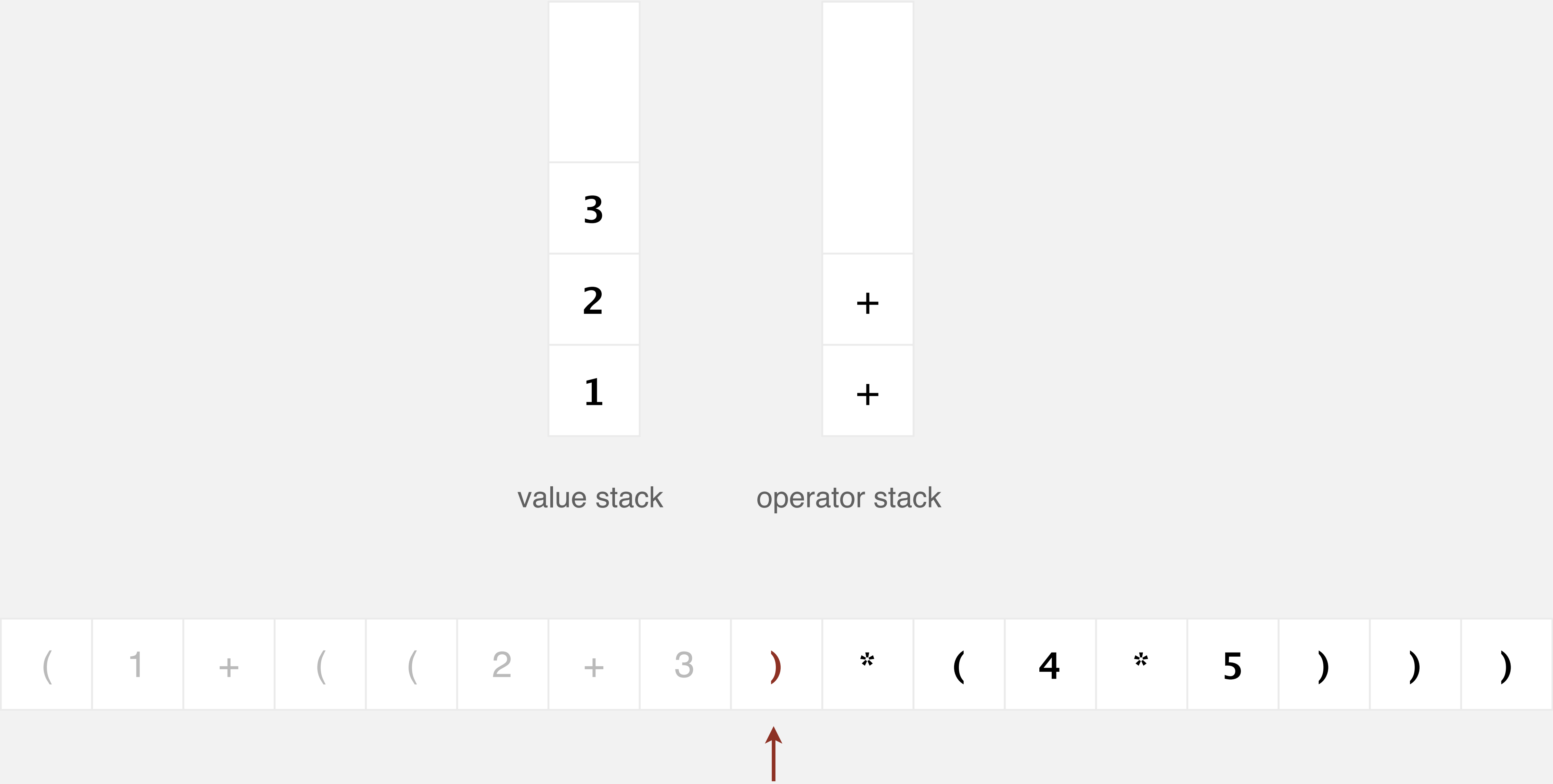
# Dijkstra's two-stack algorithm

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.



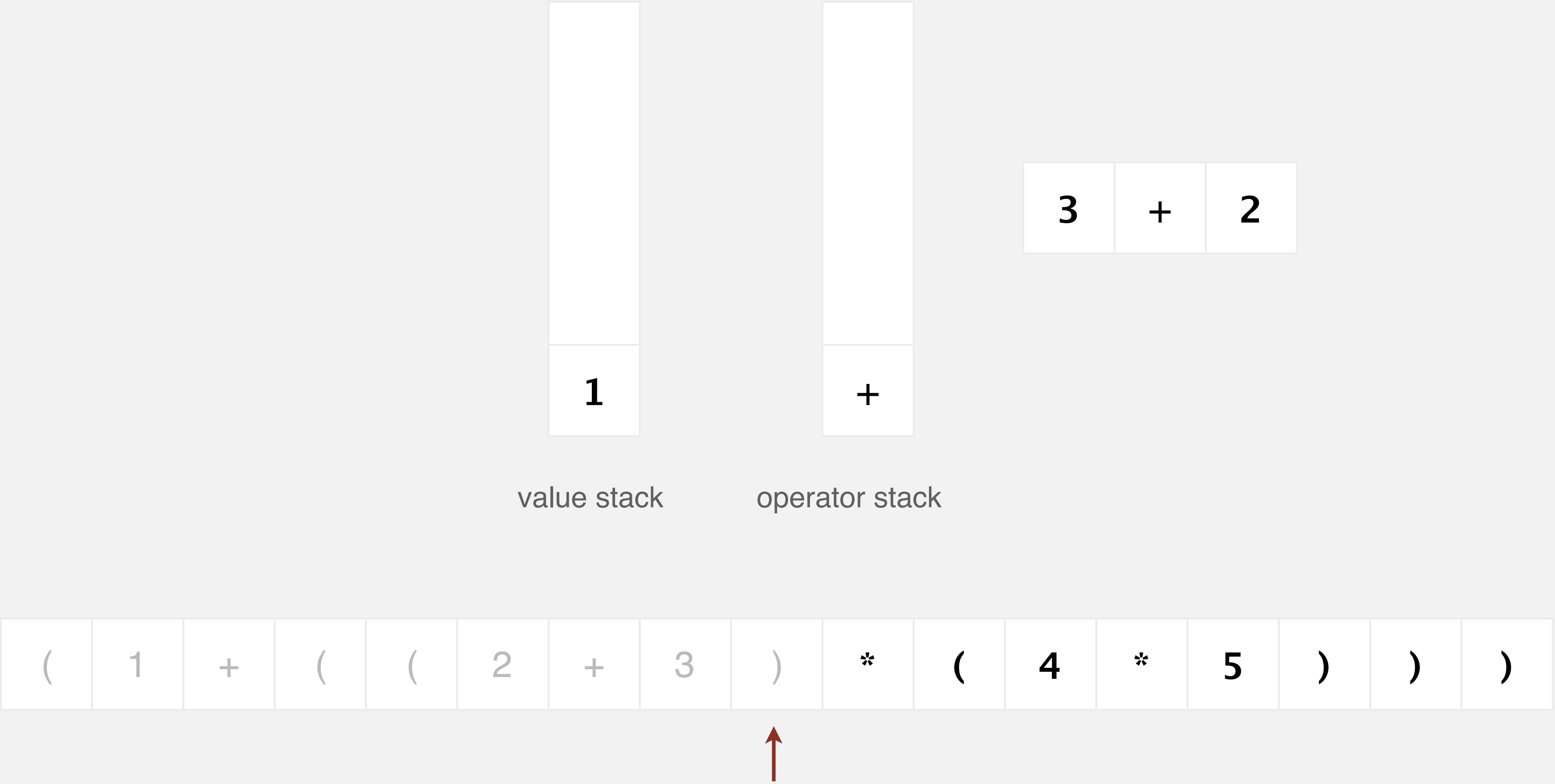
# Dijkstra's two-stack algorithm

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.



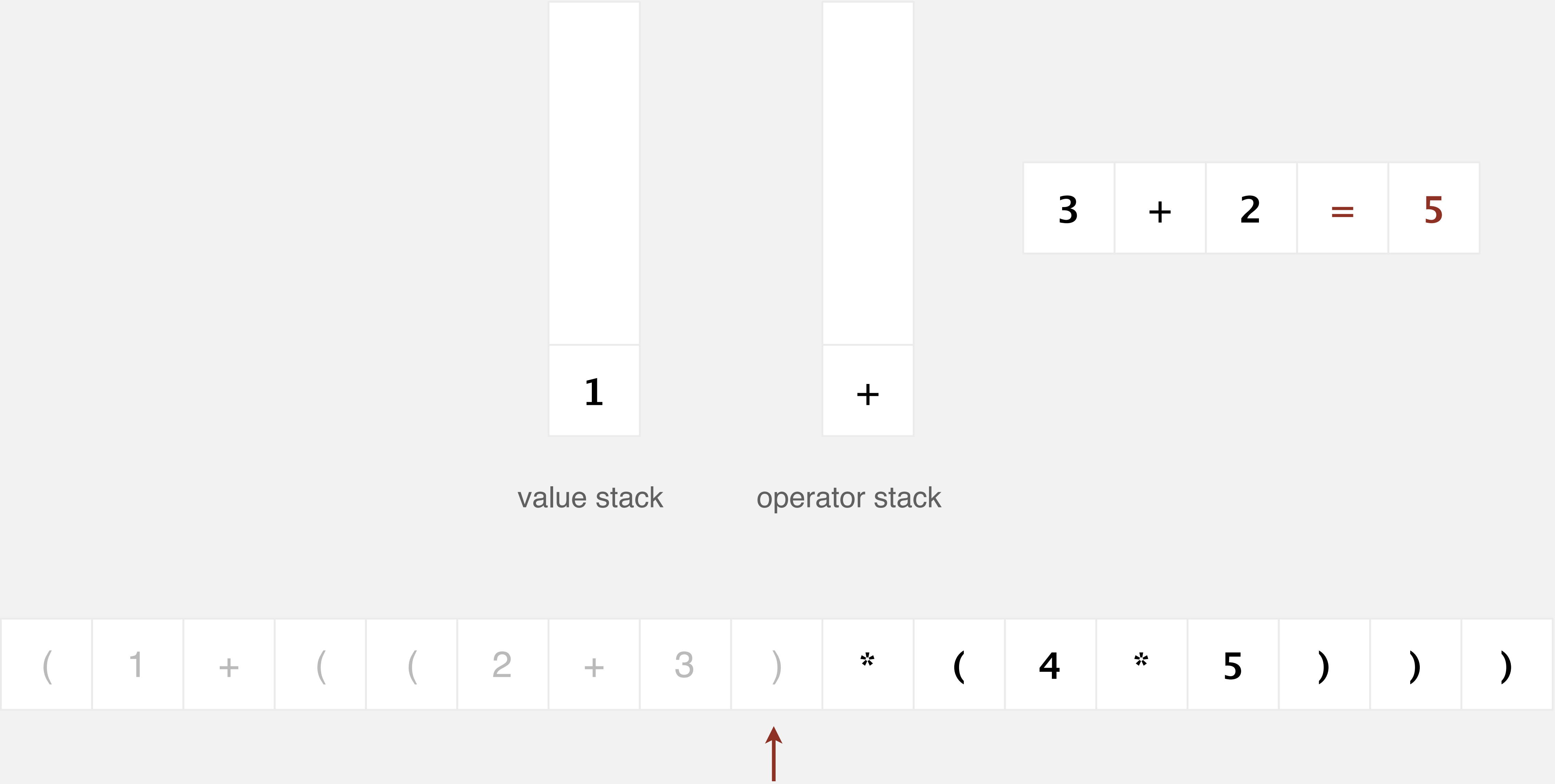
# Dijkstra's two-stack algorithm

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.



1

+

3 + 2 = 5

value stack

operator stack

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

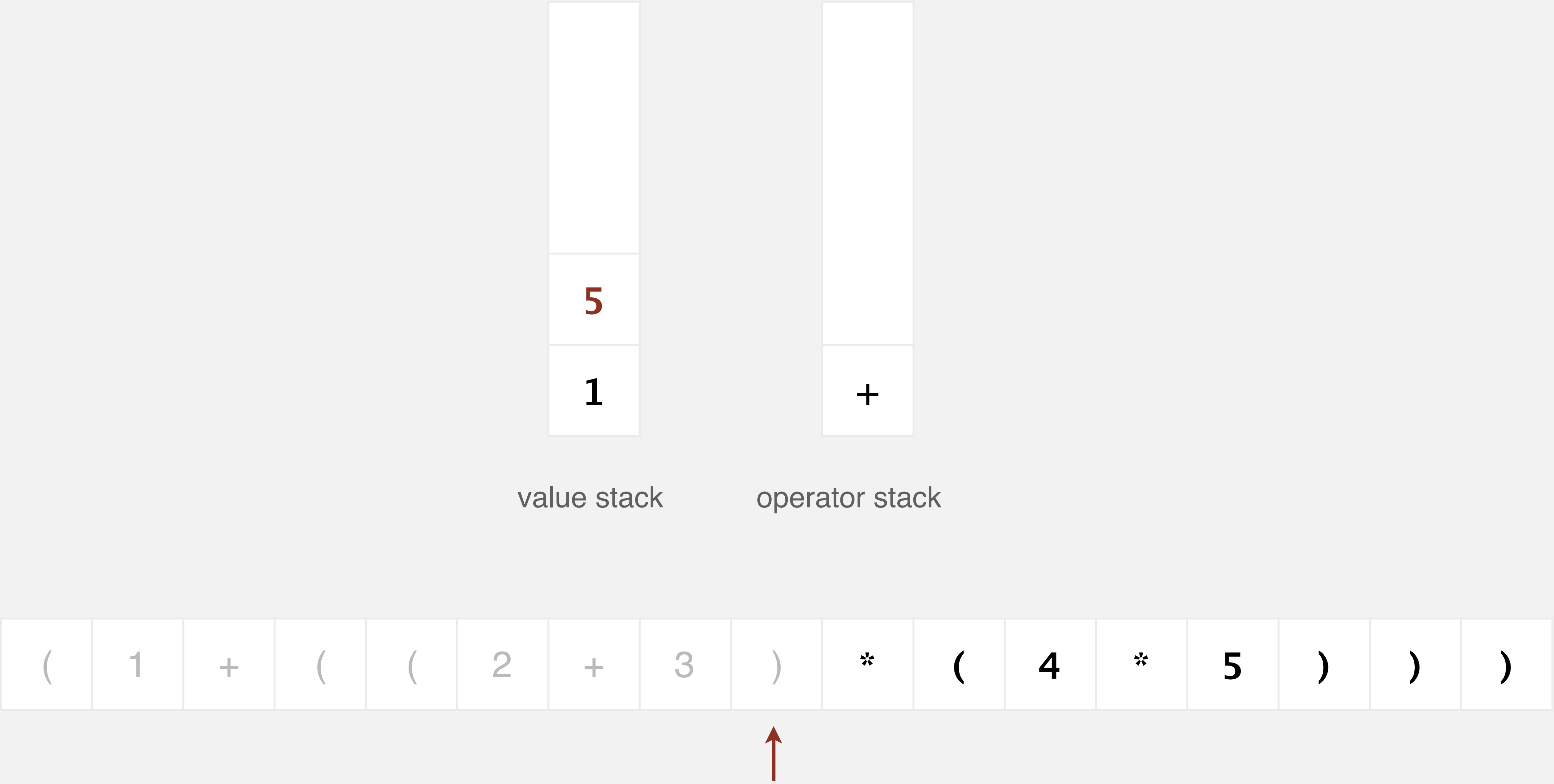
# Dijkstra's two-stack algorithm

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.



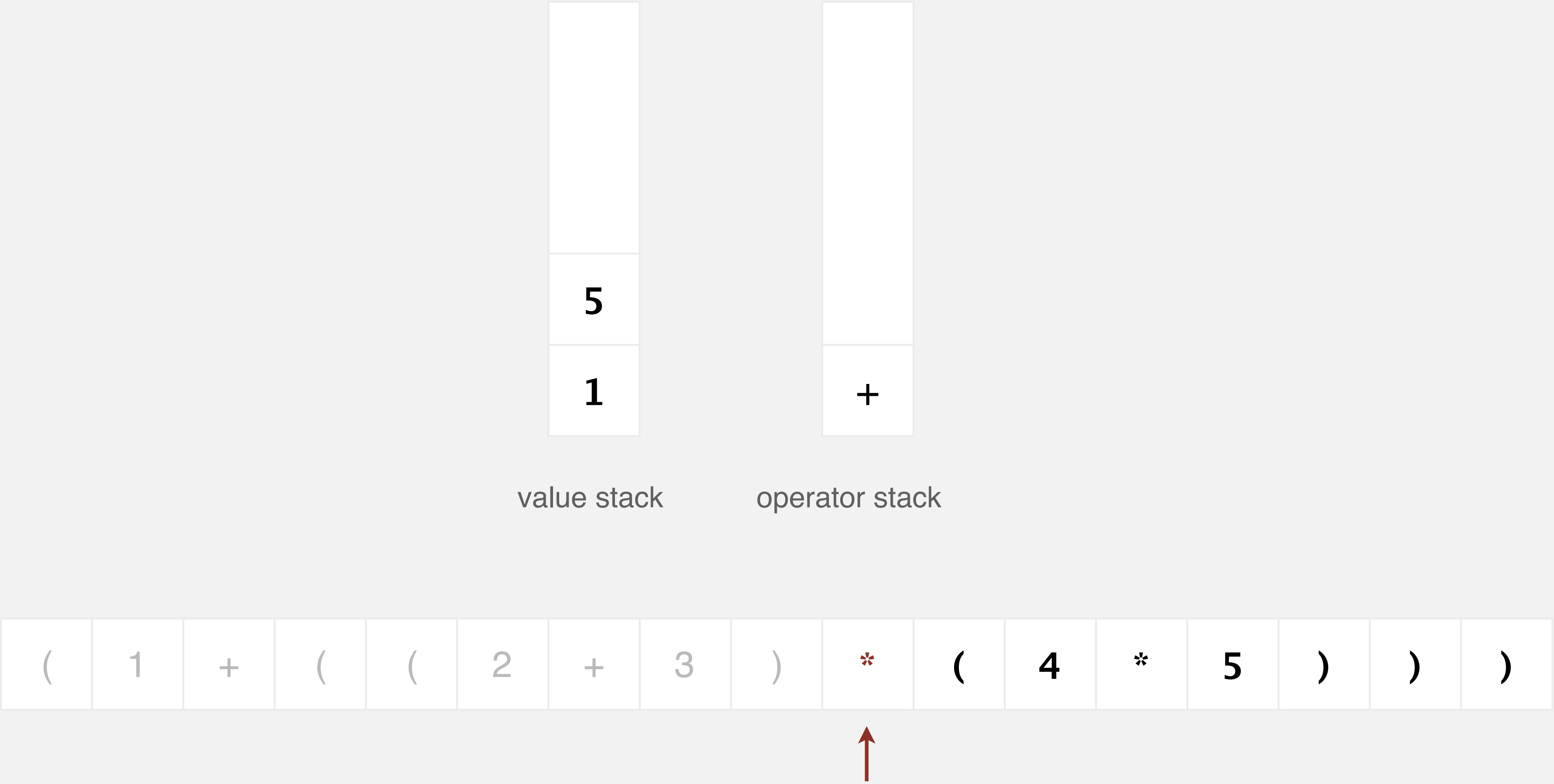
# Dijkstra's two-stack algorithm

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.



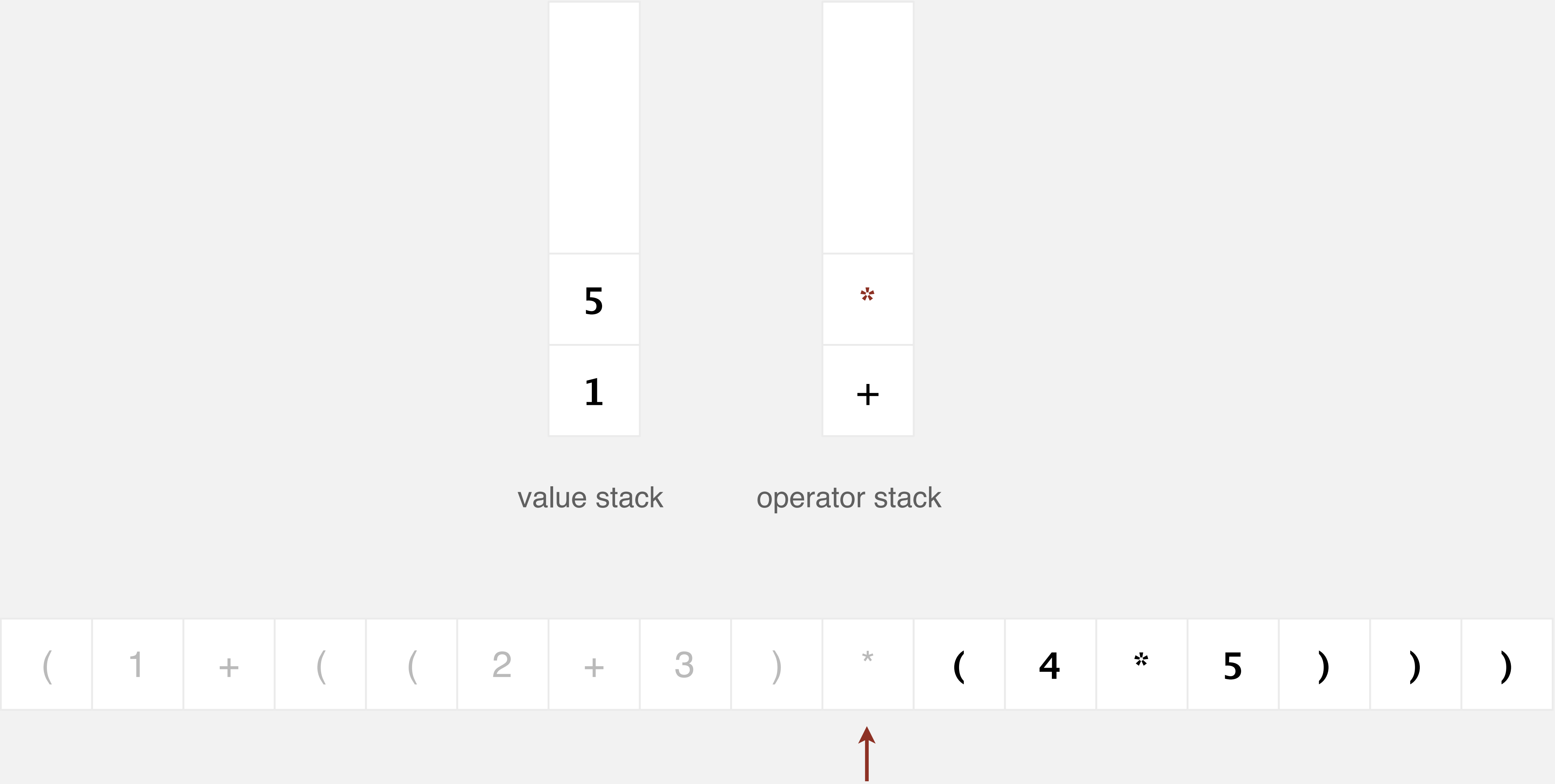
# Dijkstra's two-stack algorithm

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.



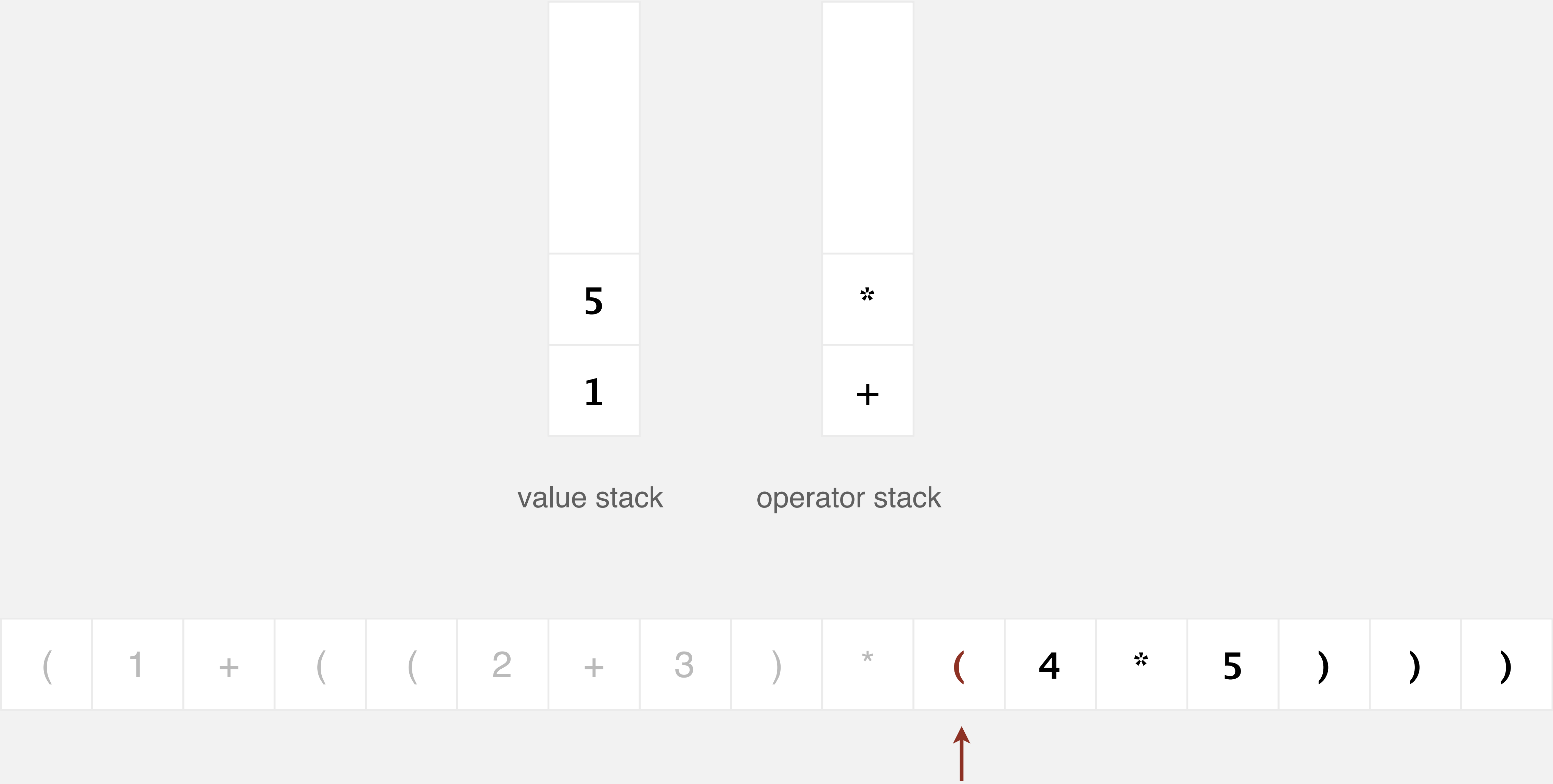
# Dijkstra's two-stack algorithm

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.



5
1

value stack

*
+

operator stack

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

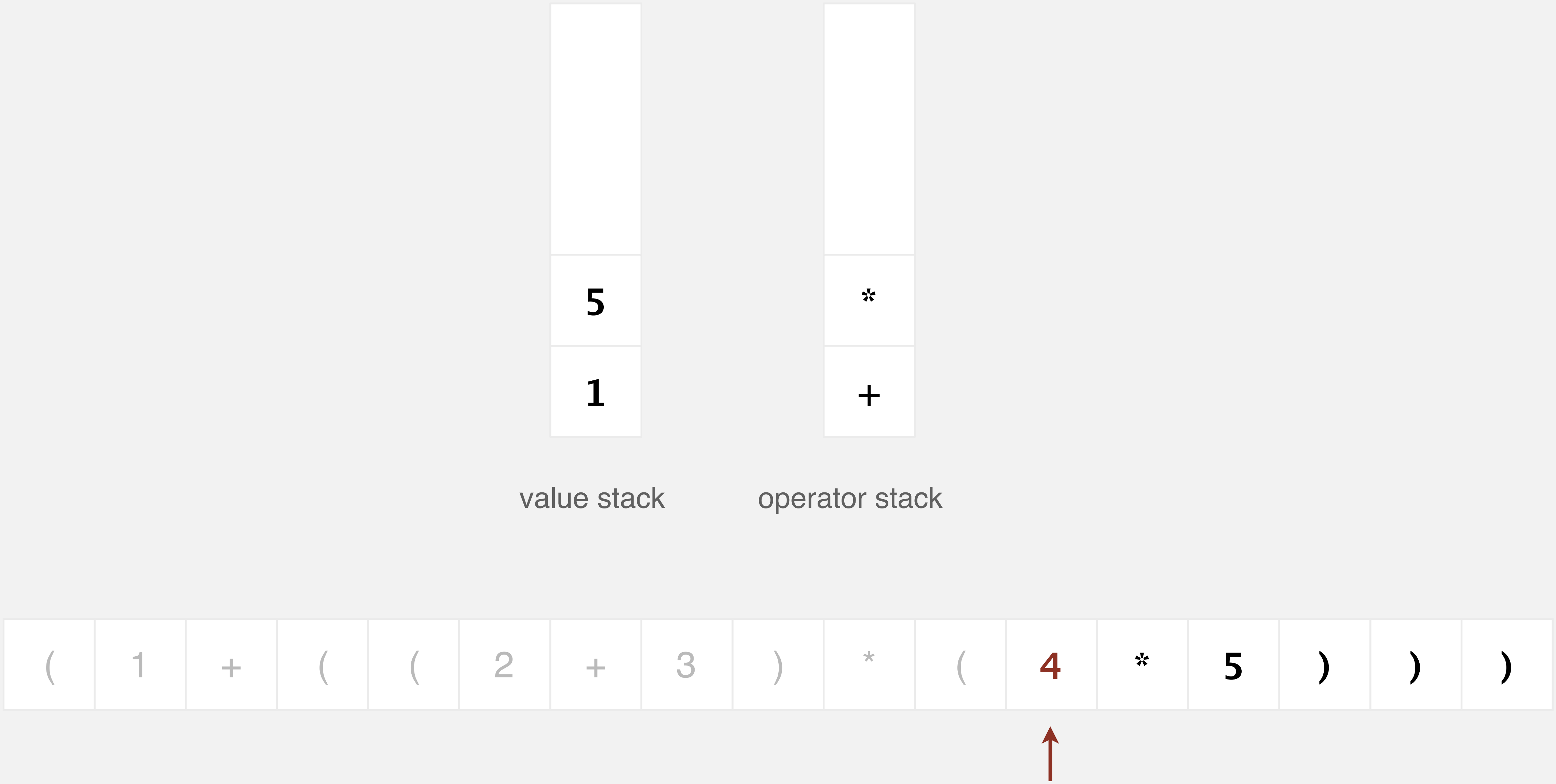
# Dijkstra's two-stack algorithm

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.



5

1

value stack

*

+

operator stack

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

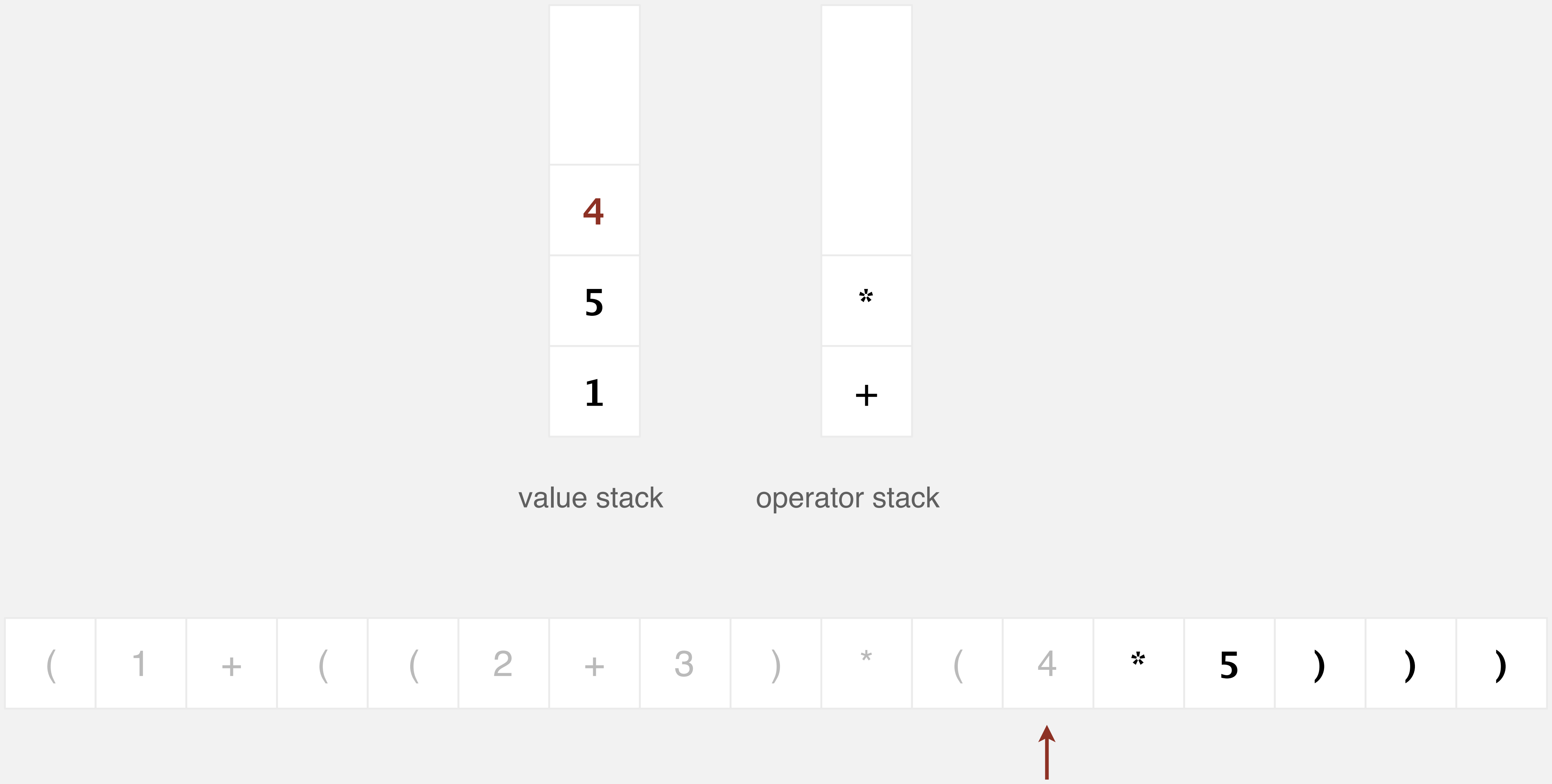
# Dijkstra's two-stack algorithm

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.



4
5
1

value stack

*
+

operator stack

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

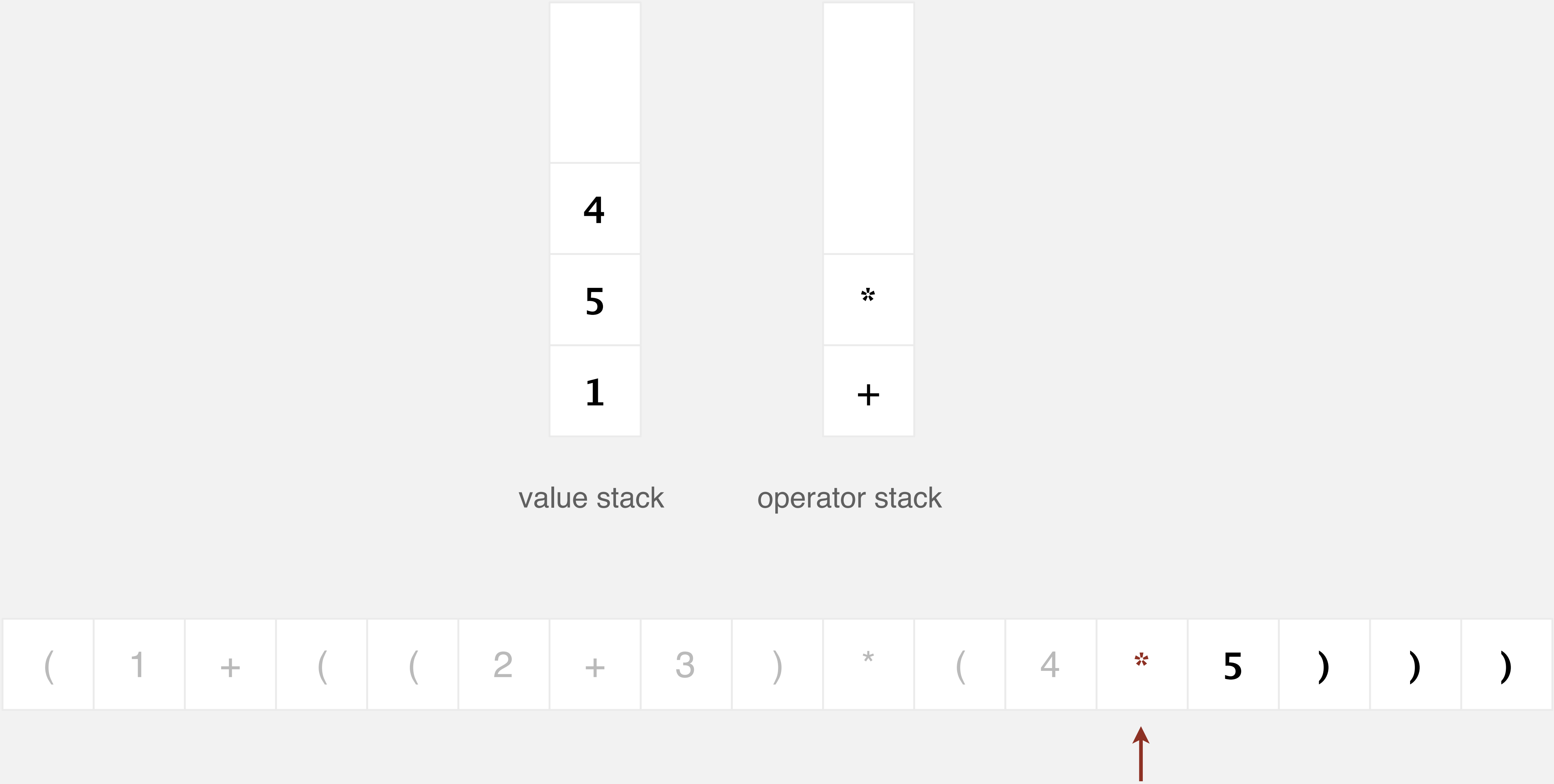
# Dijkstra's two-stack algorithm

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.



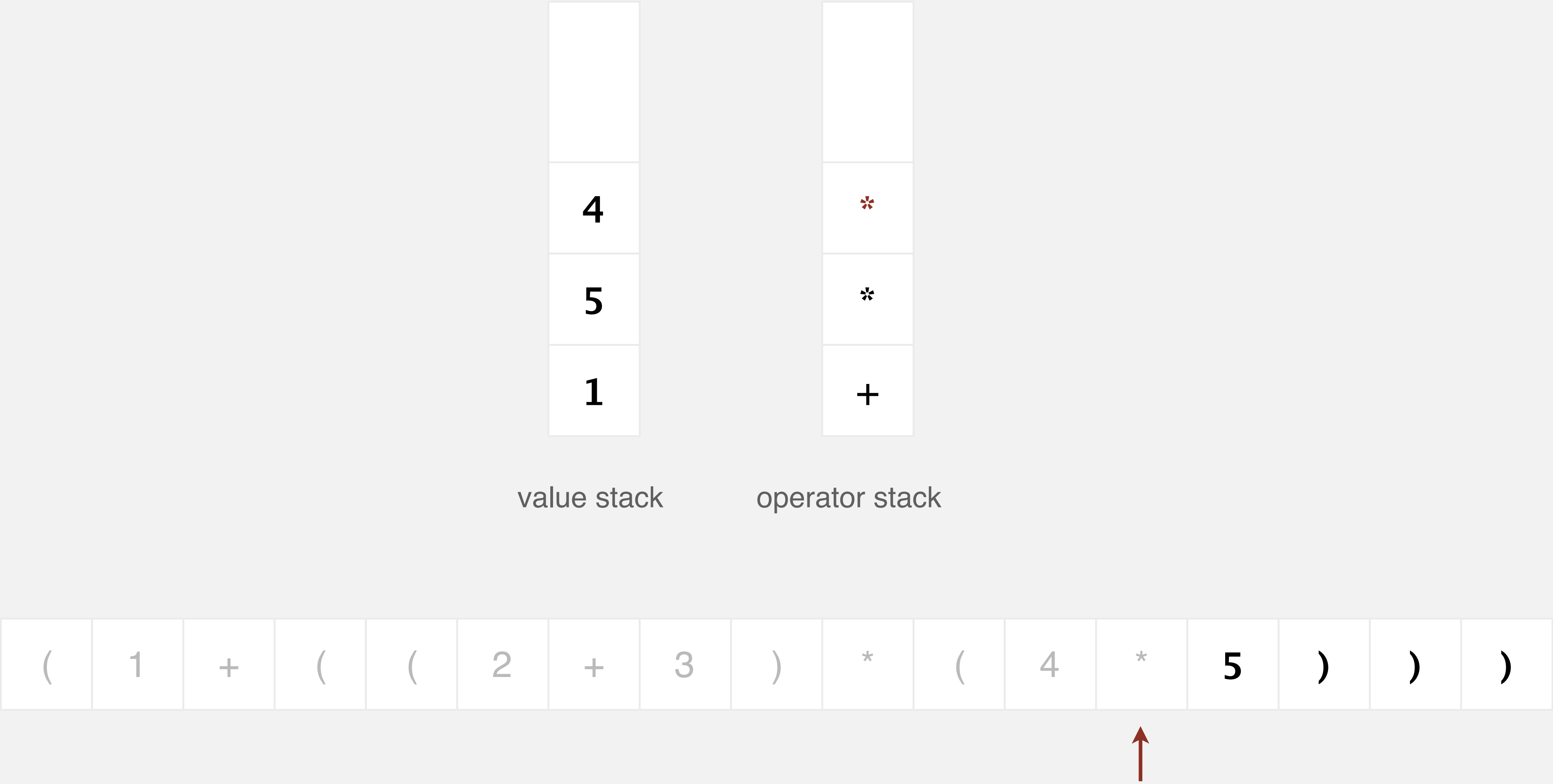
# Dijkstra's two-stack algorithm

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.



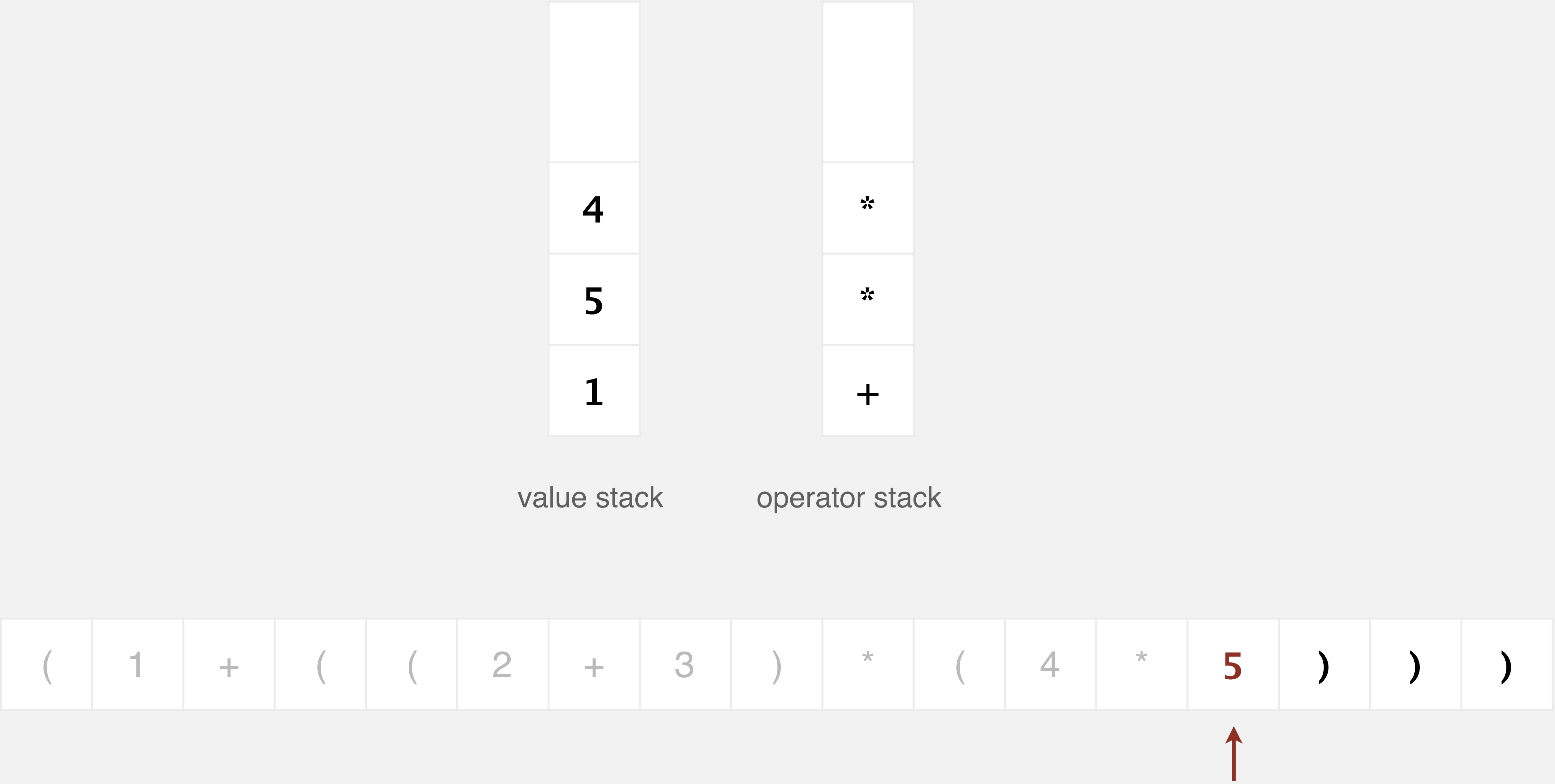
# Dijkstra's two-stack algorithm

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.



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

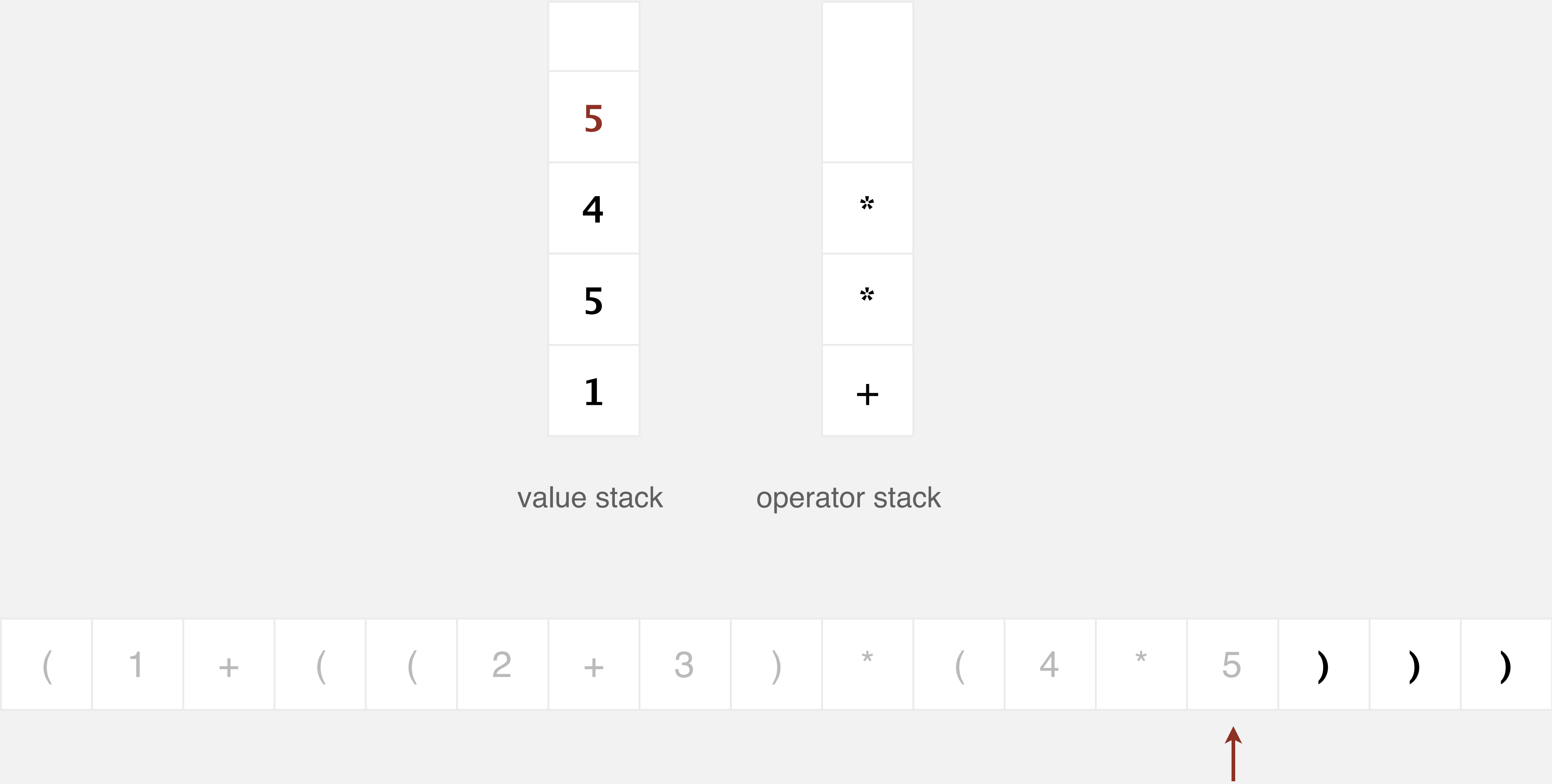
# Dijkstra's two-stack algorithm

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.



5
4
5
1

value stack

*
*
+

operator stack

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

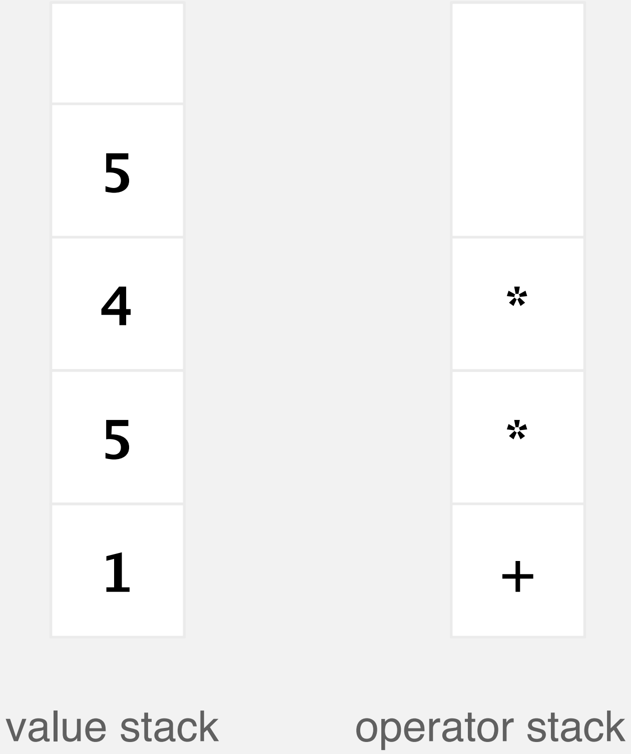
# Dijkstra's two-stack algorithm

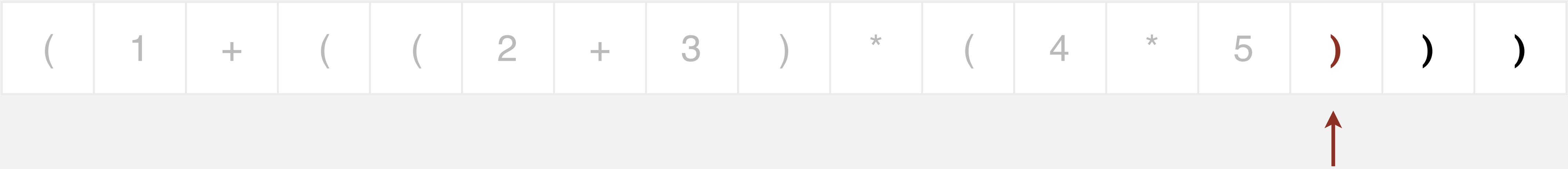
Value: push onto the value stack.

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.





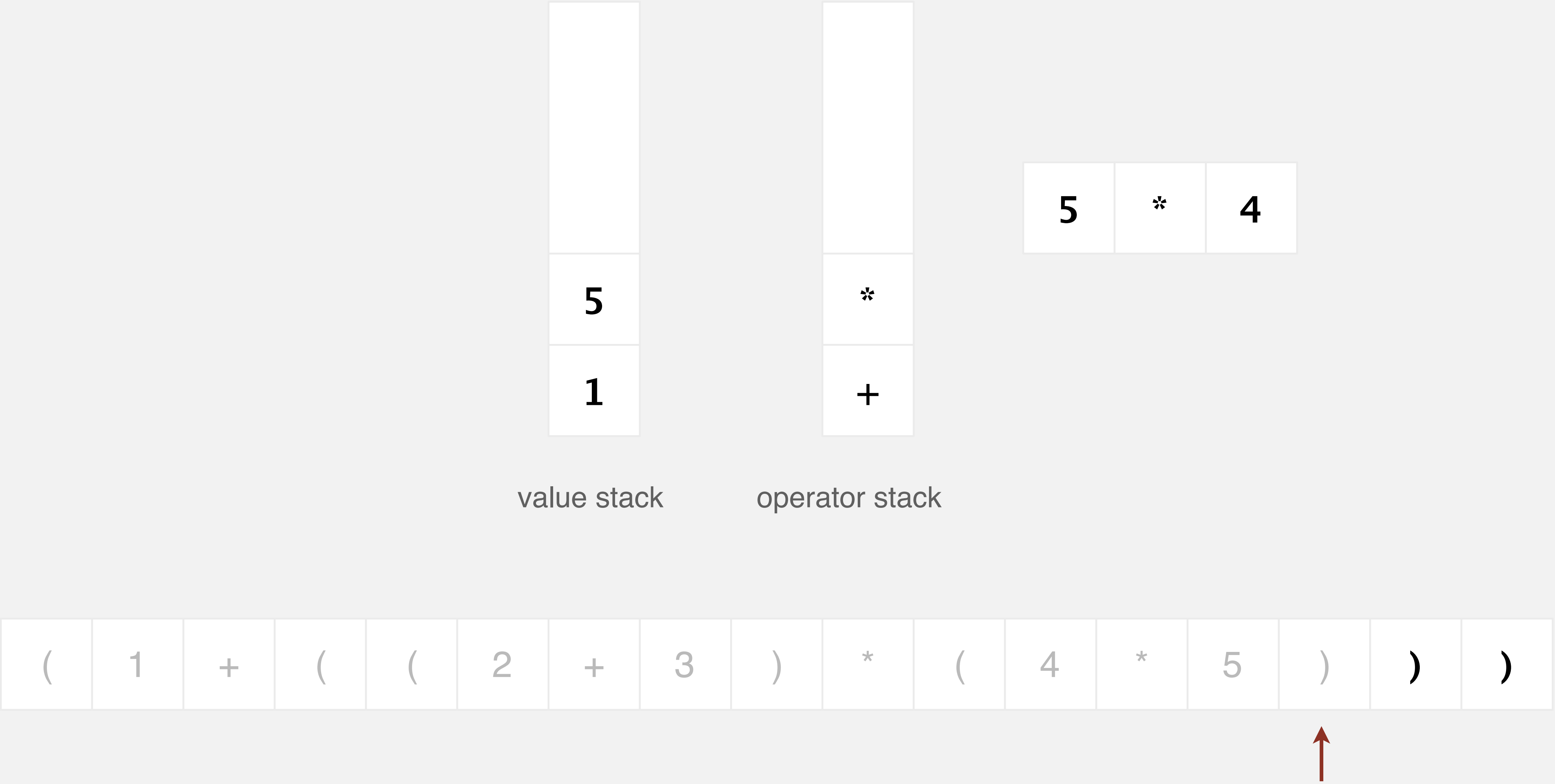
# Dijkstra's two-stack algorithm

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.



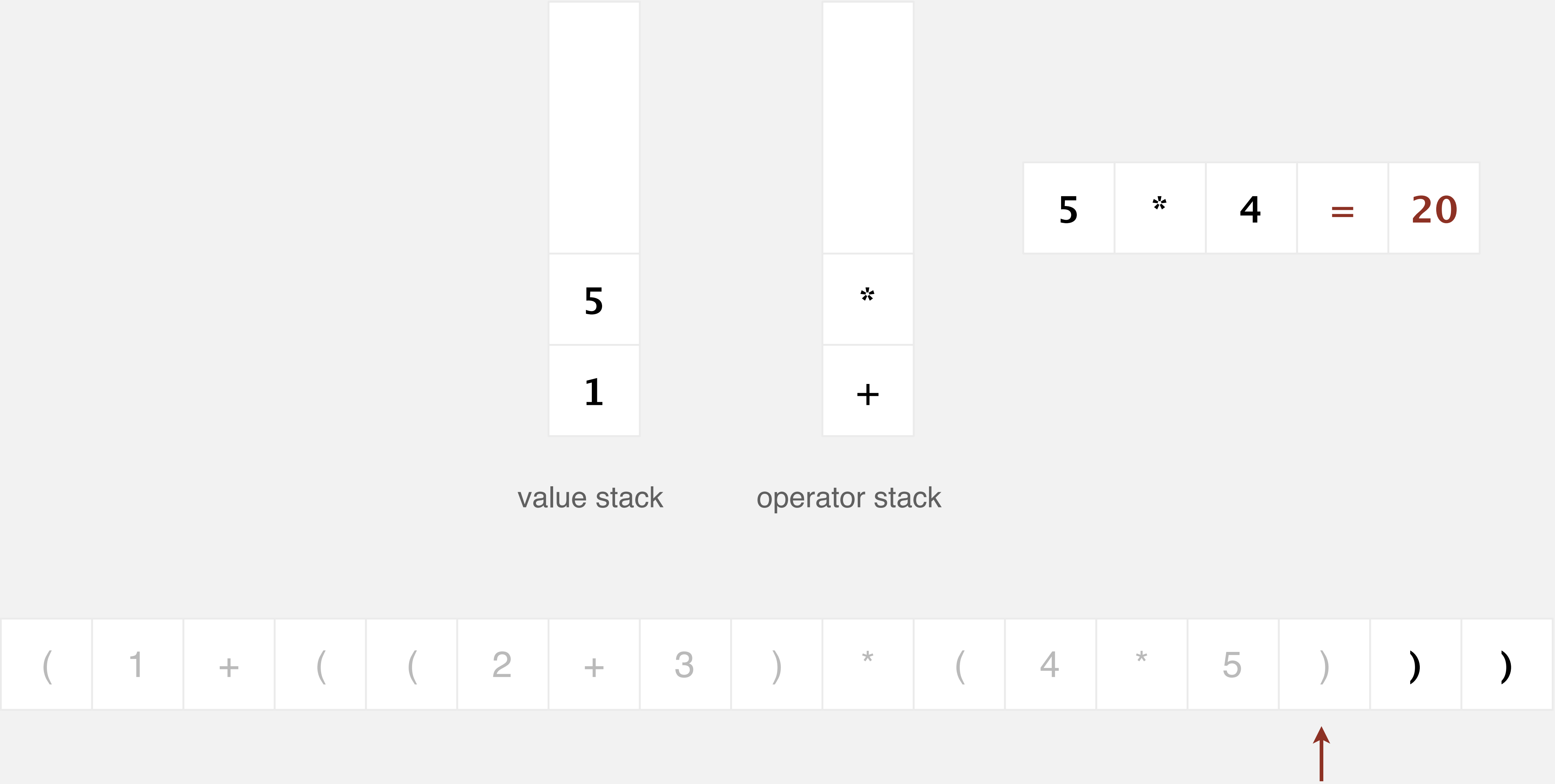
# Dijkstra's two-stack algorithm

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.



| | | | | |
|---|---|---|---|---|
| 5 | * | 4 | = | 20 |

| value stack | operator stack |
|---|---|
| 5 | * |
| 1 | + |

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

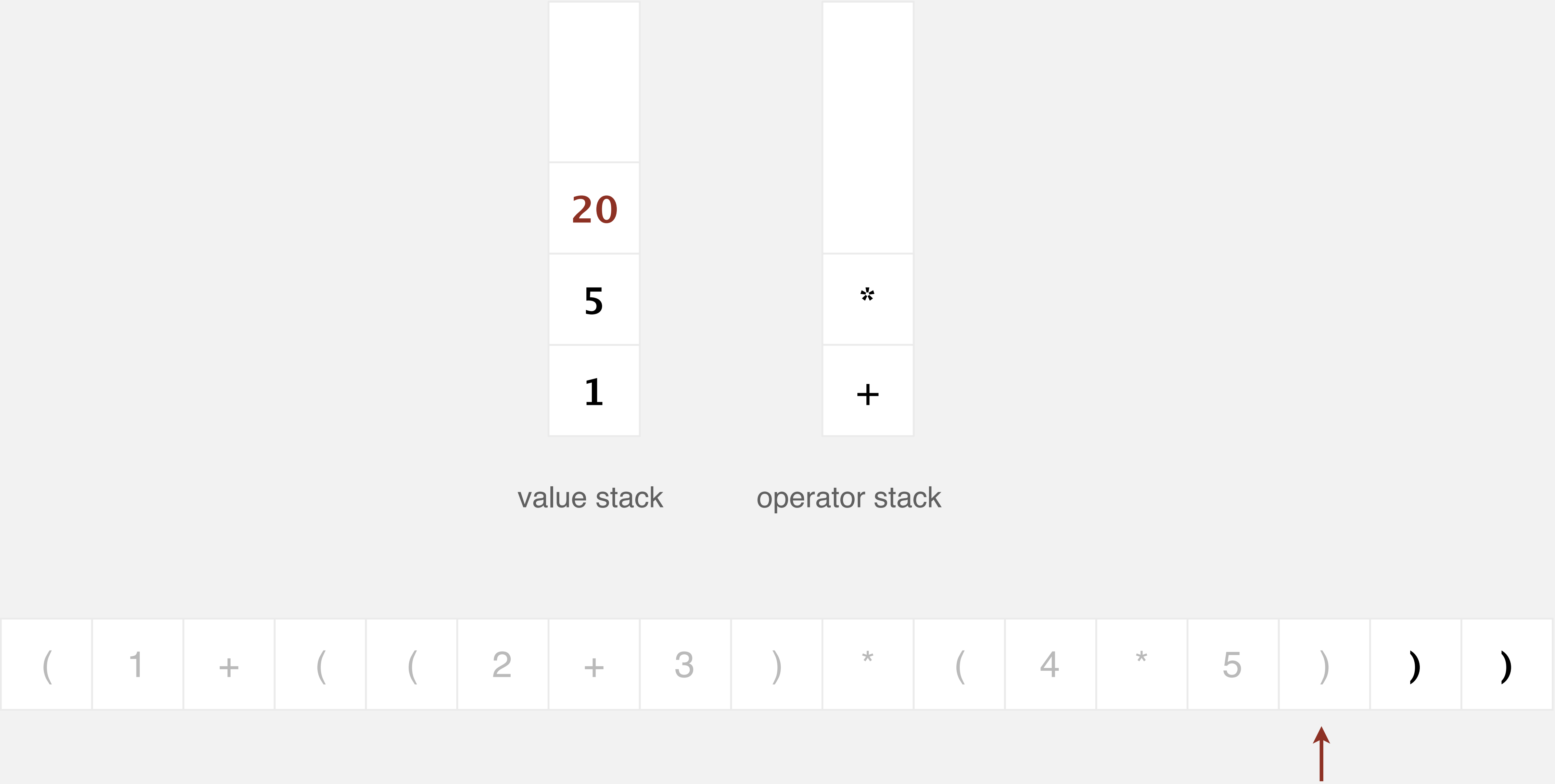
# Dijkstra's two-stack algorithm

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.



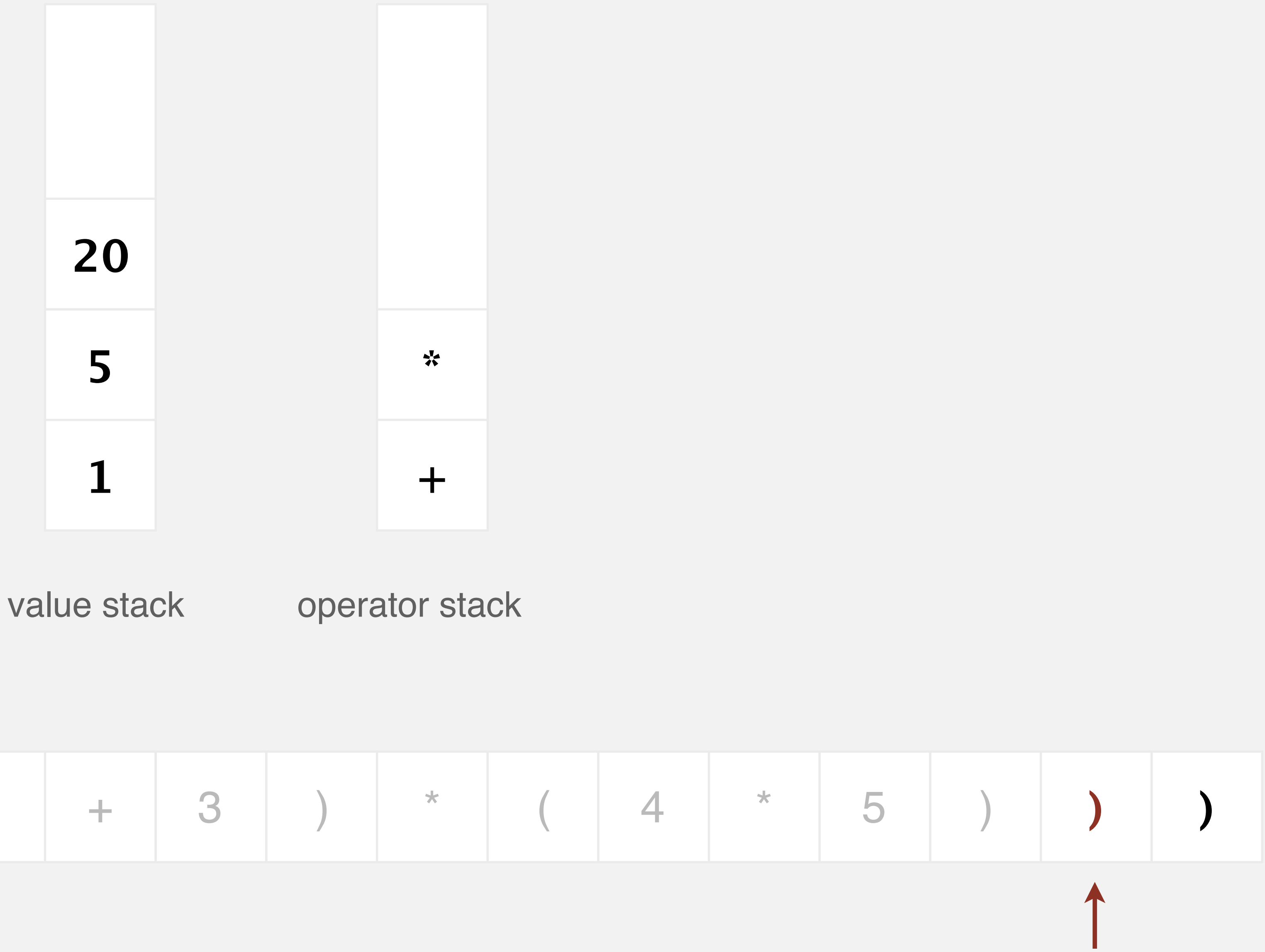
# Dijkstra's two-stack algorithm

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.



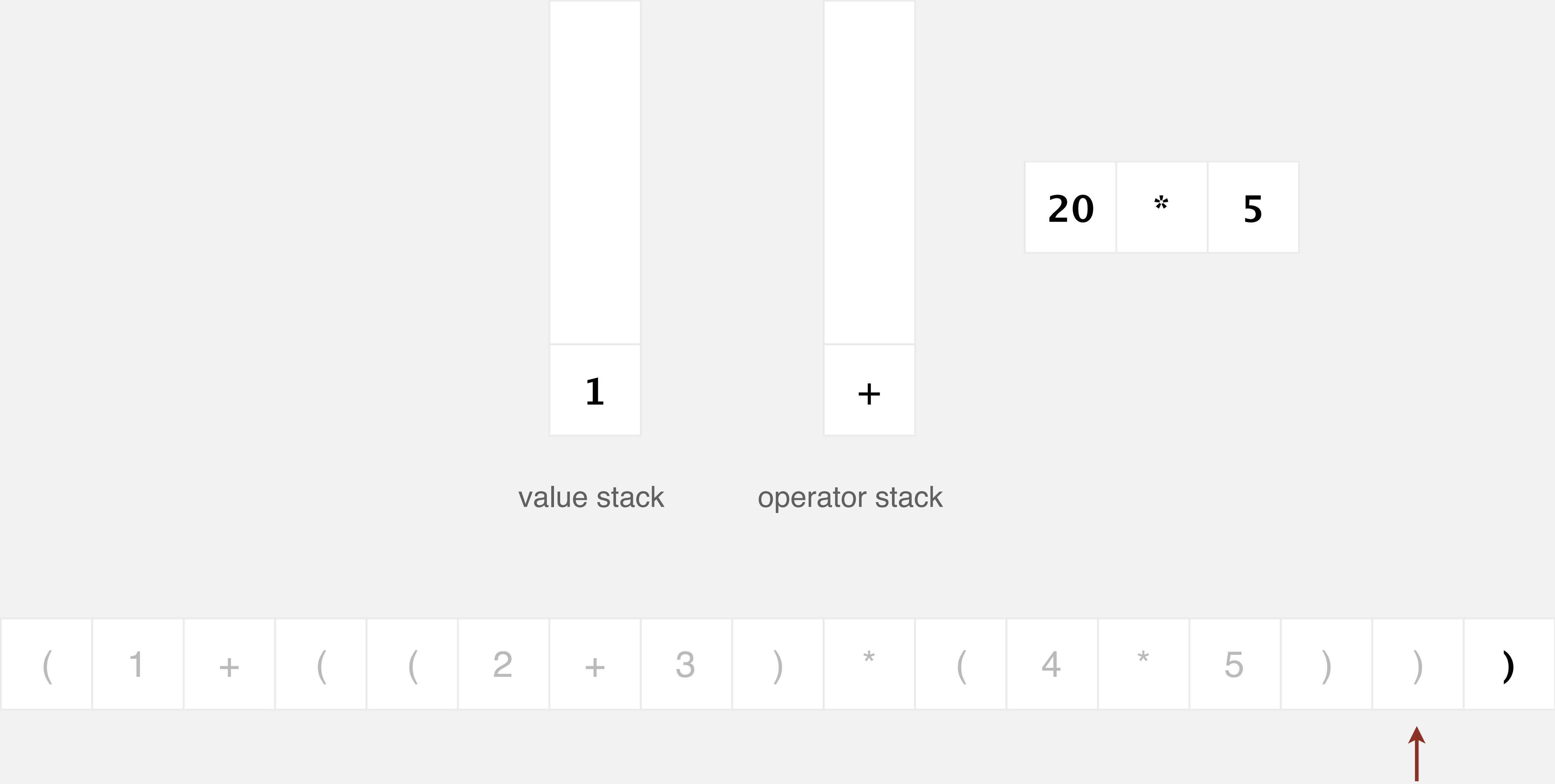
# Dijkstra's two-stack algorithm

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.



1

+

20 * 5

value stack

operator stack

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

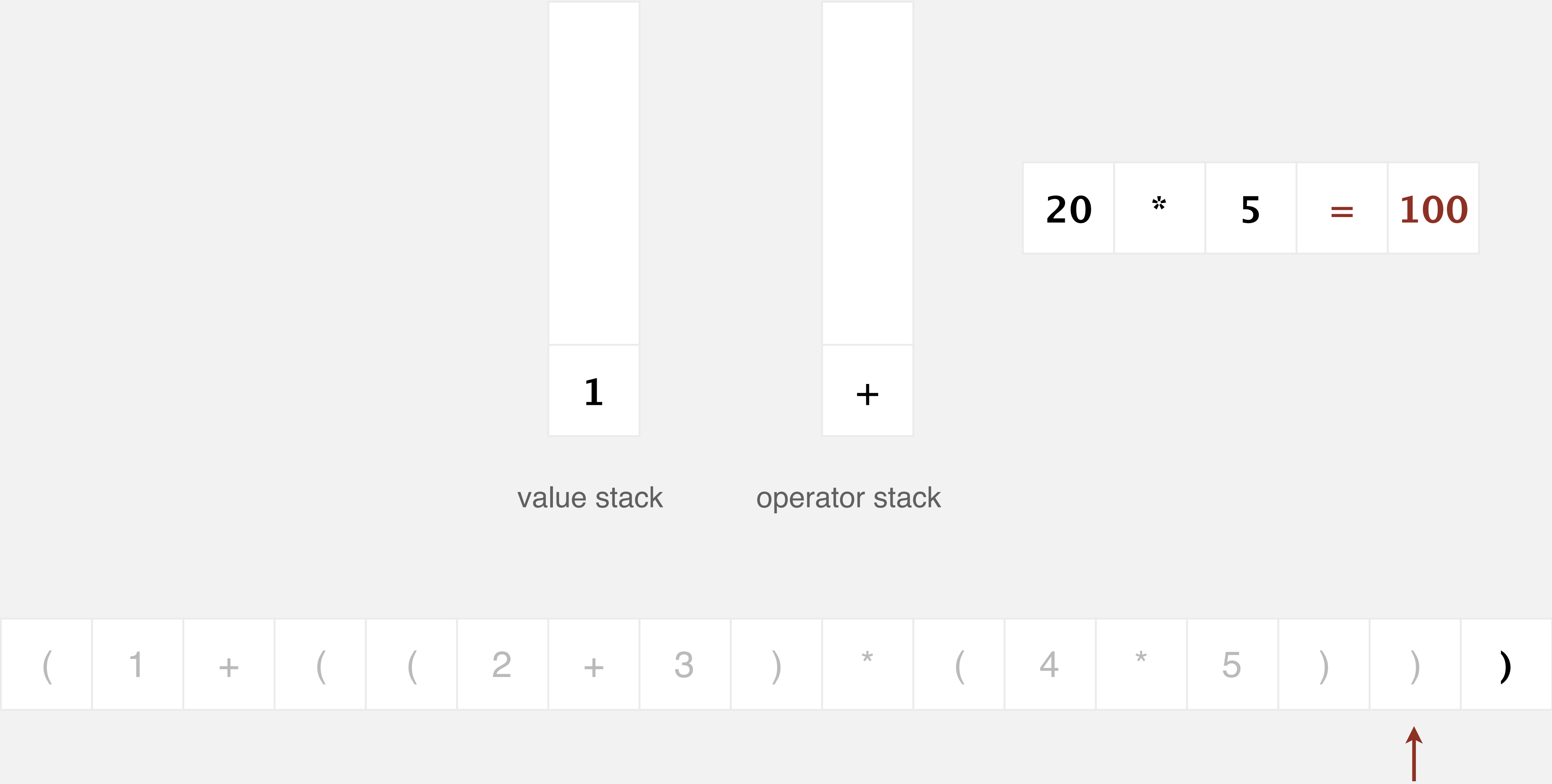
# Dijkstra's two-stack algorithm

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.



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

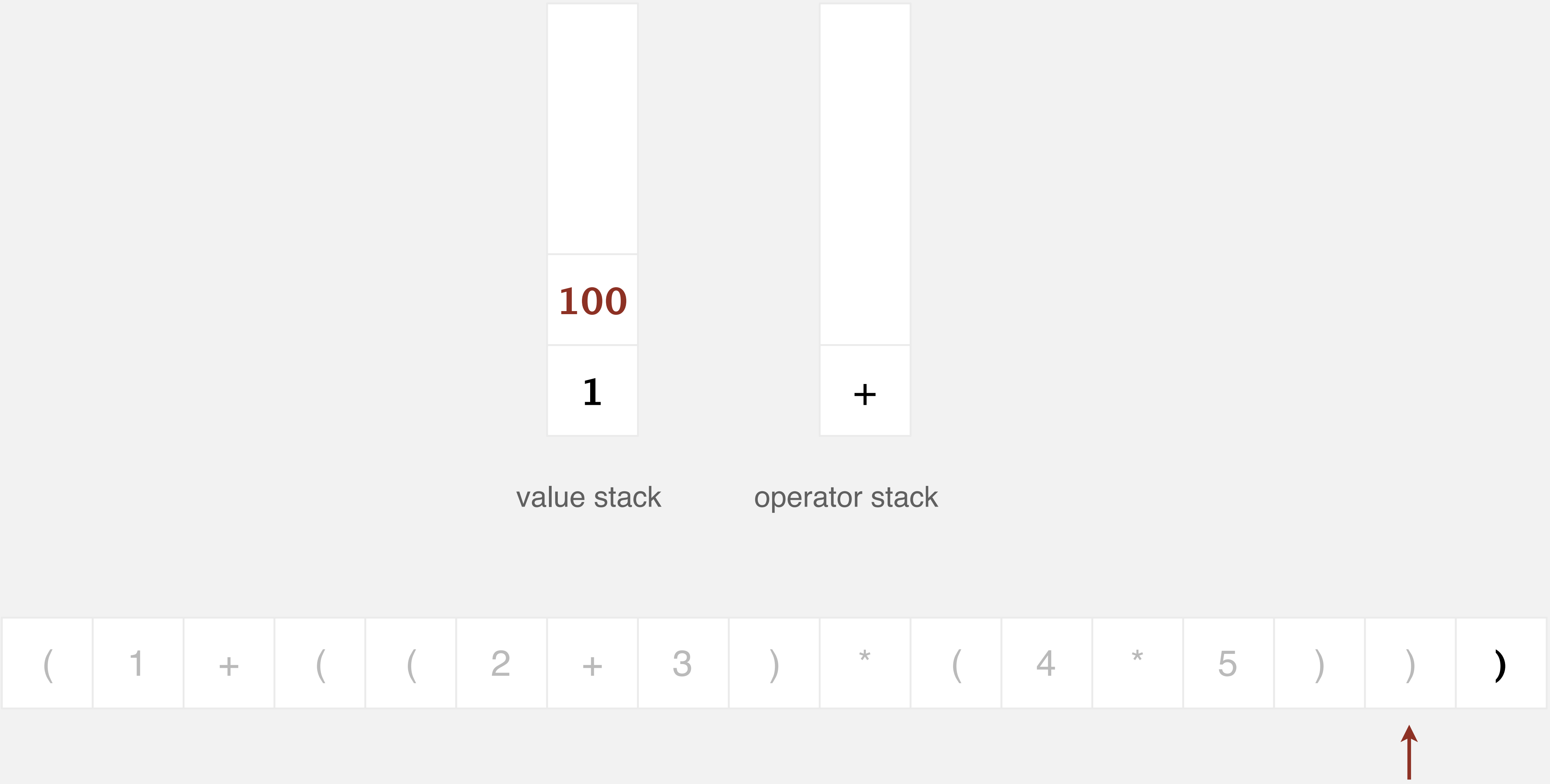
# Dijkstra's two-stack algorithm

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.



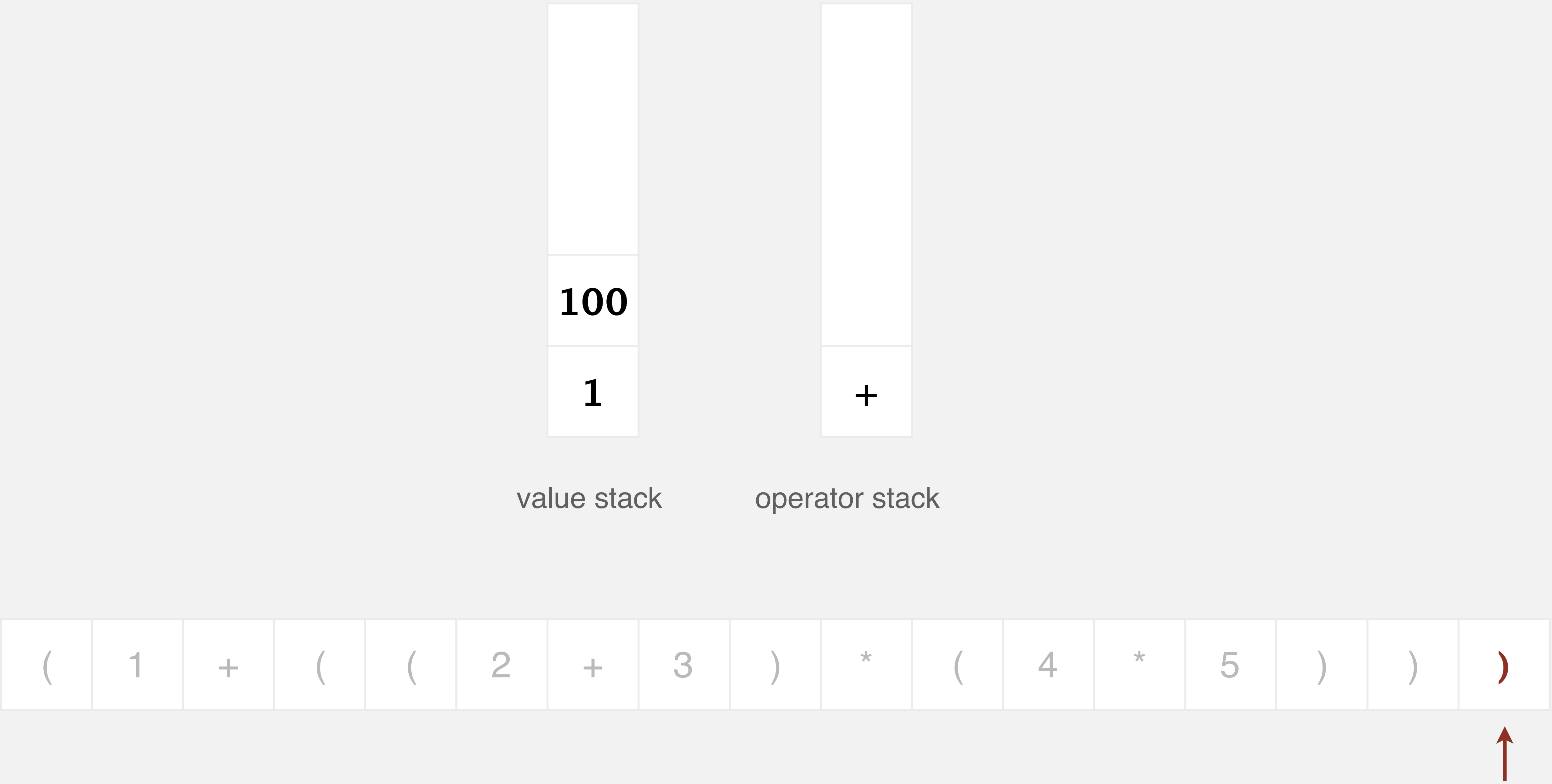
# Dijkstra's two-stack algorithm

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.



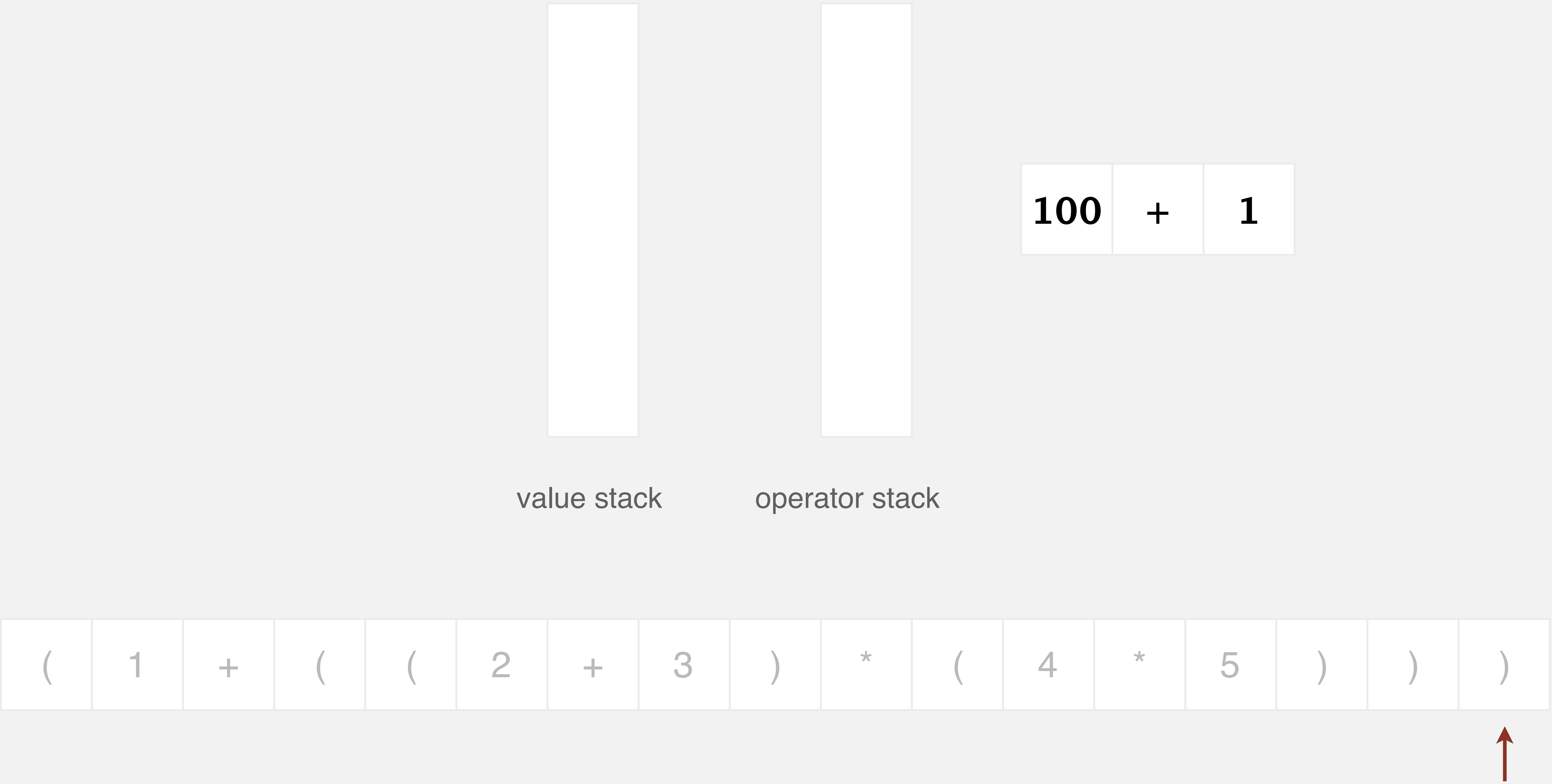
# Dijkstra's two-stack algorithm

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.



100 + 1

value stack

operator stack

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

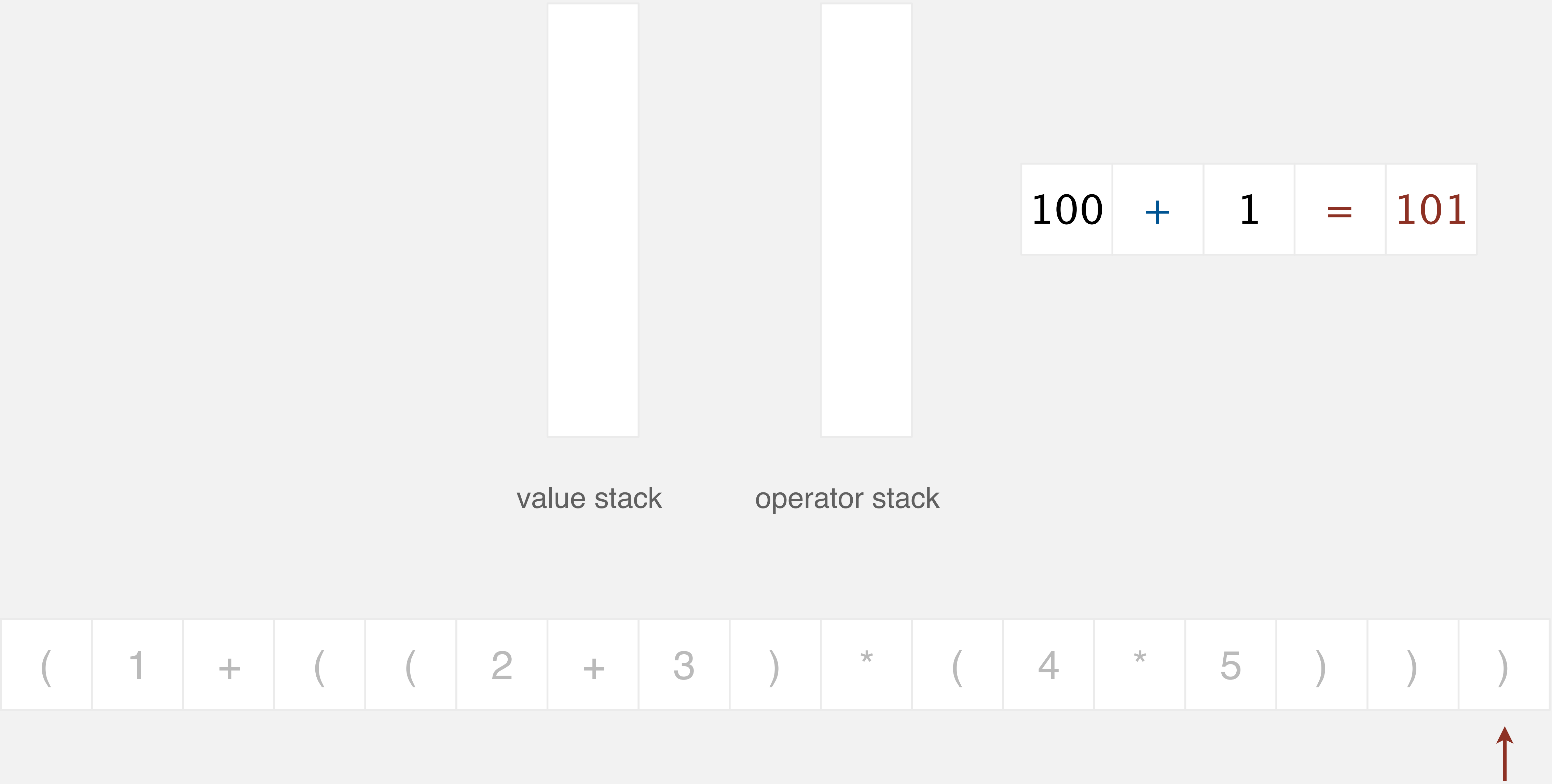
# Dijkstra's two-stack algorithm

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.



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

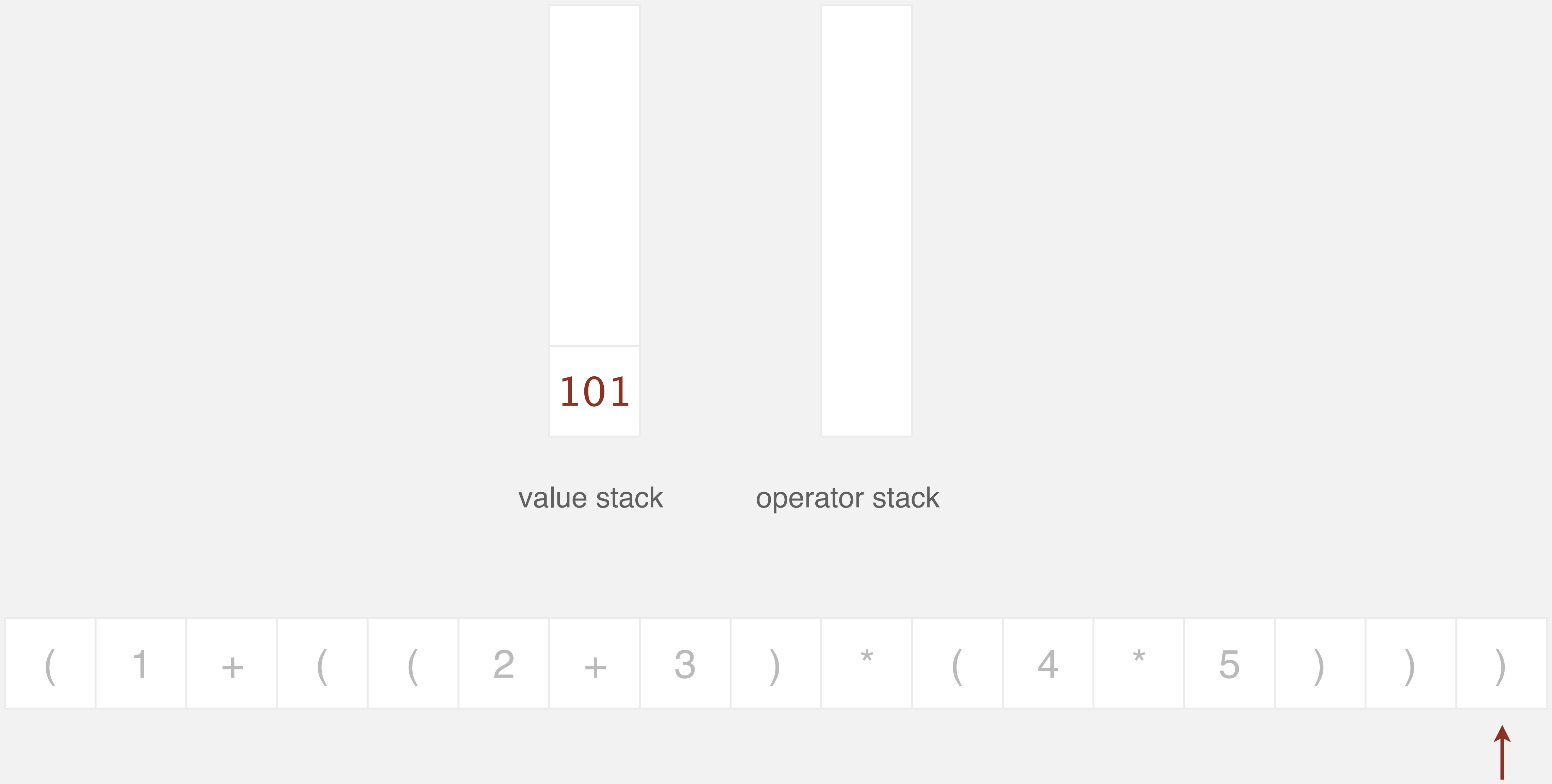
# Dijkstra's two-stack algorithm

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.



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

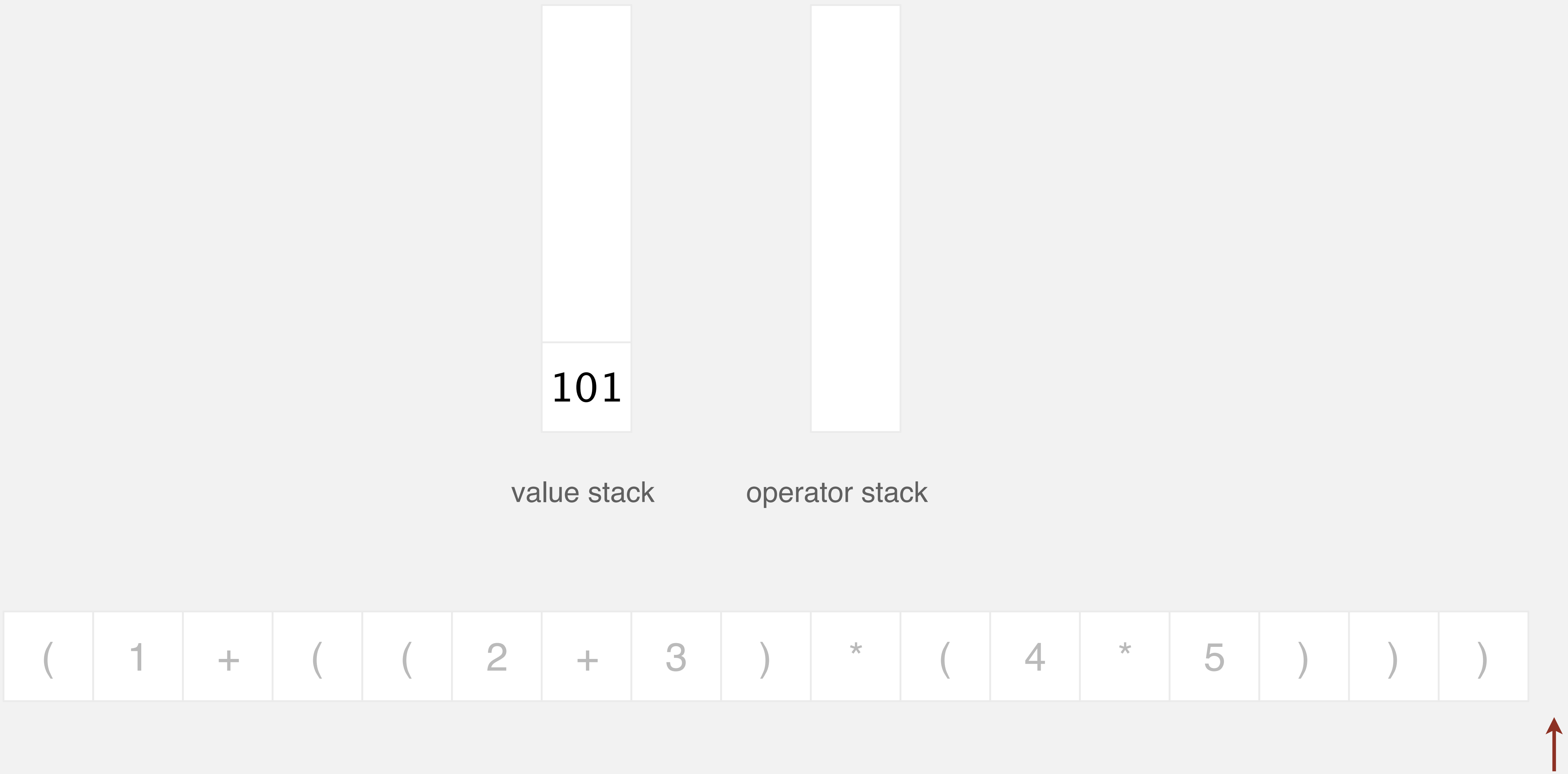
# Dijkstra's two-stack algorithm

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.



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

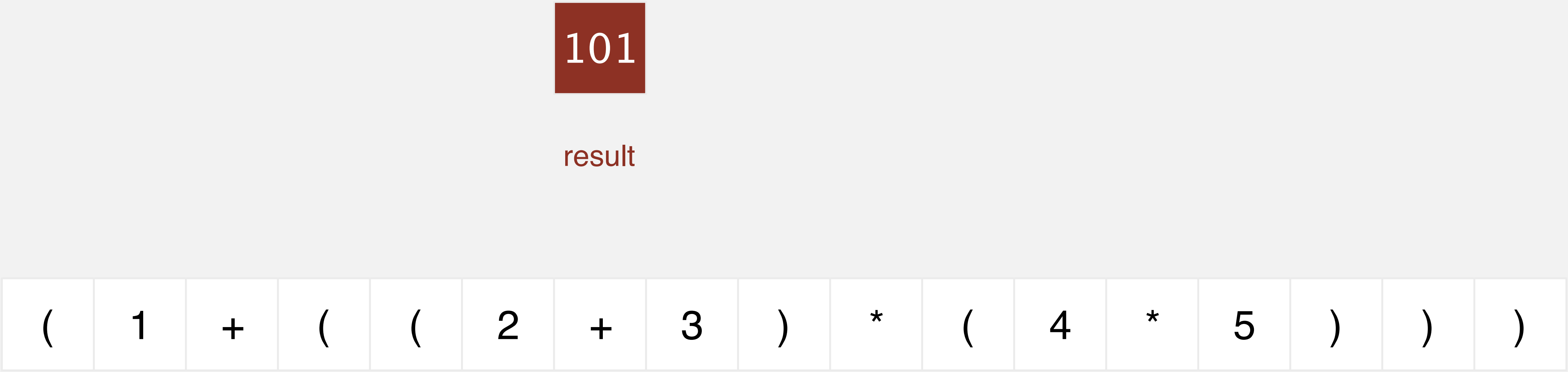
# Dijkstra's two-stack algorithm

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.



101

result

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

# Correctness

# Correctness

Q. Why correct?

# Correctness

Q. Why correct?

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 ) ) )
```

# Correctness

Q. Why correct?

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 ) ) )
```

# Correctness

Q. Why correct?

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
```

# Correctness

Q. Why correct?

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:

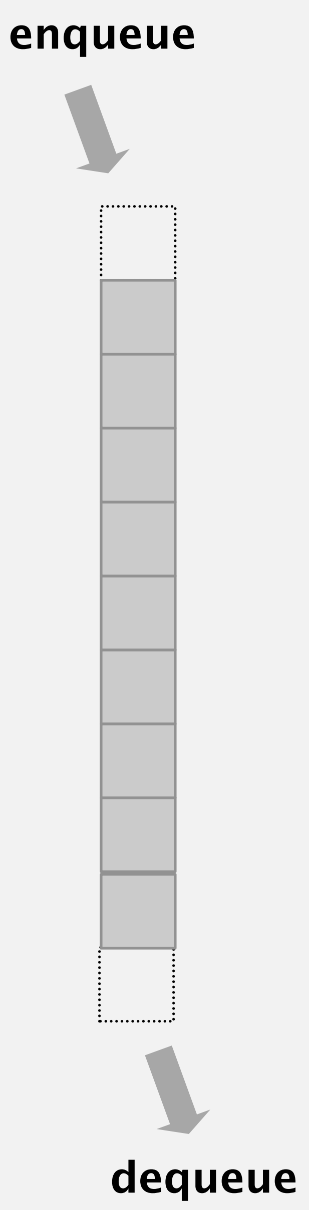
```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

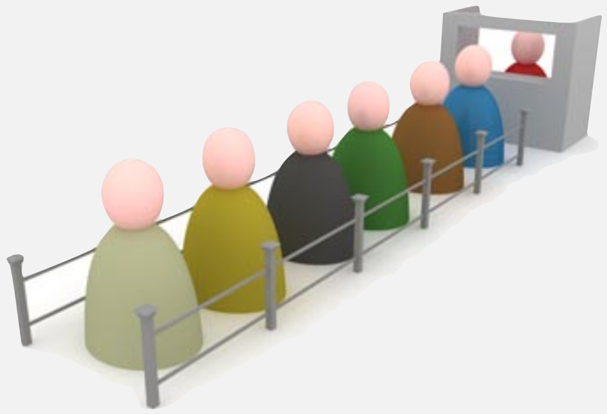
Extensions. More ops, precedence order, associativity.

# Queue API

public class Queue<T>

| | | |
|---|---|---|
| | Queue() | *create an empty queue* |
| void | enqueue(T item) | *insert a new item onto queue* |
| T | dequeue() | *remove and return the item least recently added* |
| boolean | isEmpty() | *is the queue empty?* |
| int | size() | *number of strings on the queue* |





# Queue test client

Read strings from standard input.

- If string equals "-", dequeue string and print.
- Otherwise, enqueue string.

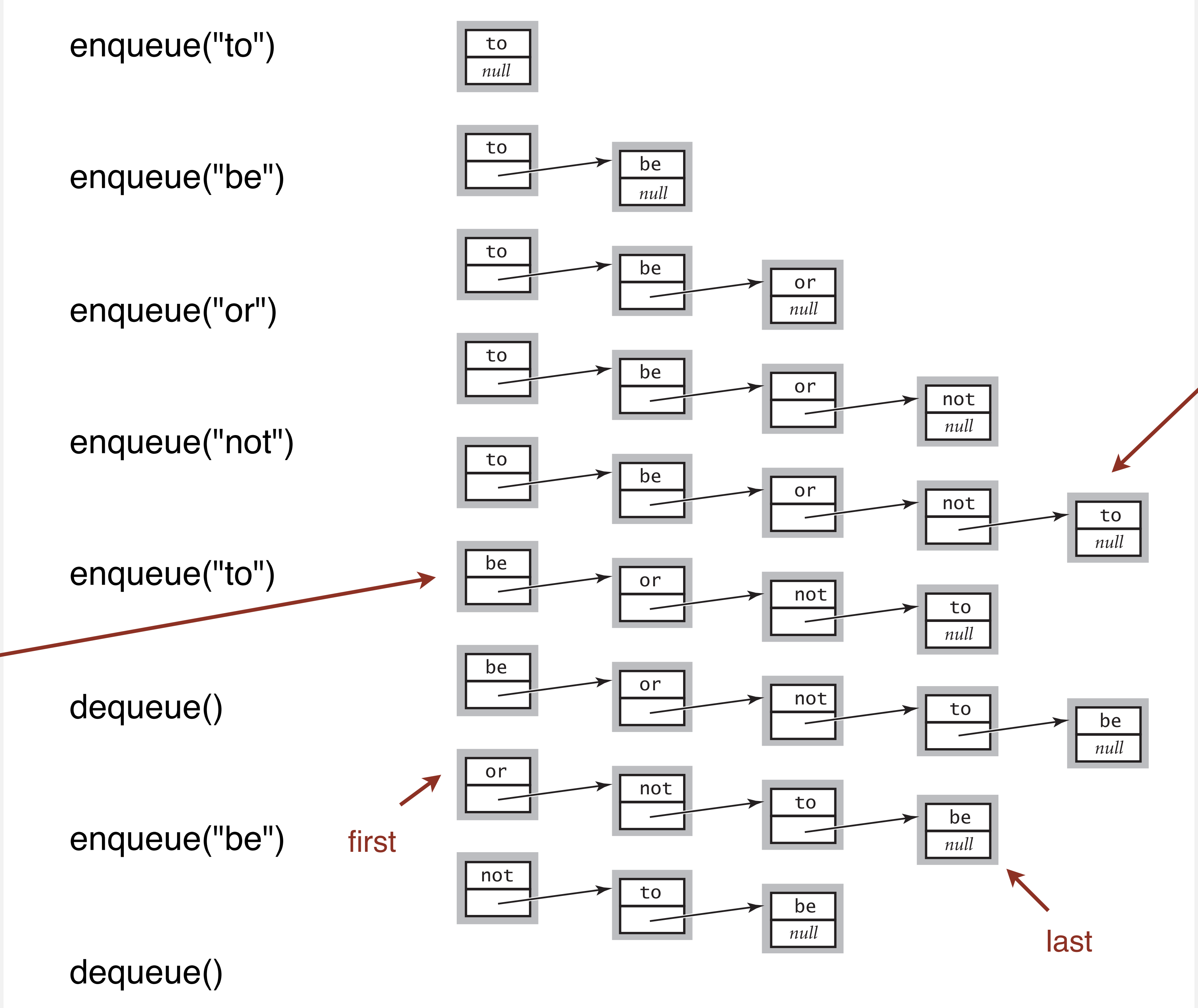
```
public static void main(String[] args)
{
   Queue<T> q = new Queue<>();
   while (!StdIn.isEmpty())
   {
      String s = StdIn.readString();
      if (s.equals("-")) StdOut.print(q.dequeue());
      else               q.enqueue(s);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java QueueOfStrings < tobe.txt
to be or not to be
```

# Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; remove from front; insert at end.



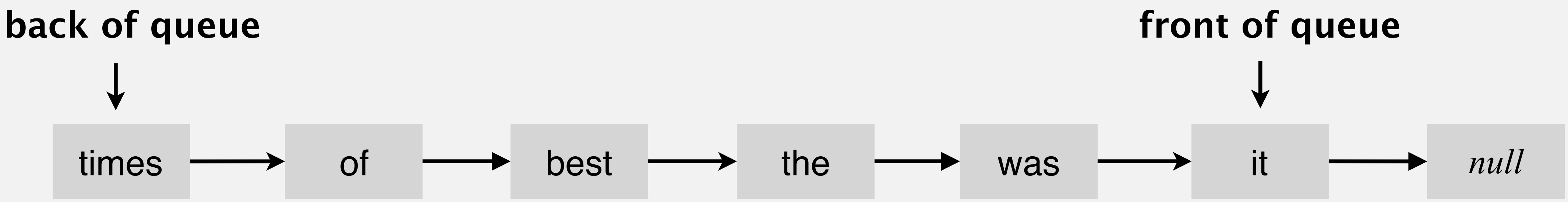
insert at end of linked list

remove from front of linked list

# How to implement a queue with a linked list?

A. Can't be done efficiently with a singly-linked list.

B.

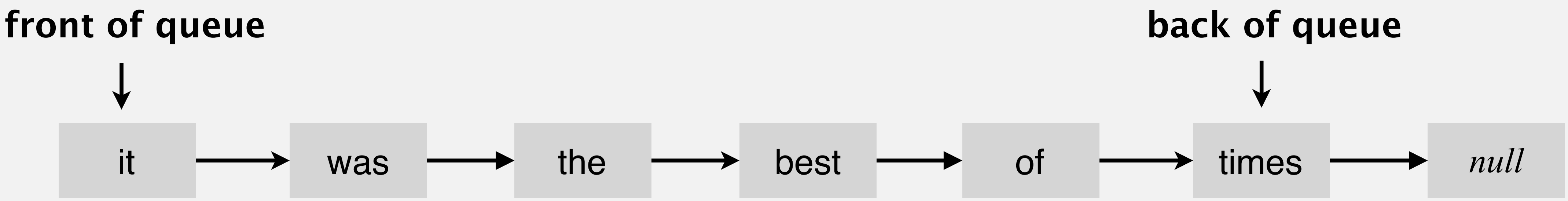


back of queue

front of queue

times → of → best → the → was → it → *null*

C.

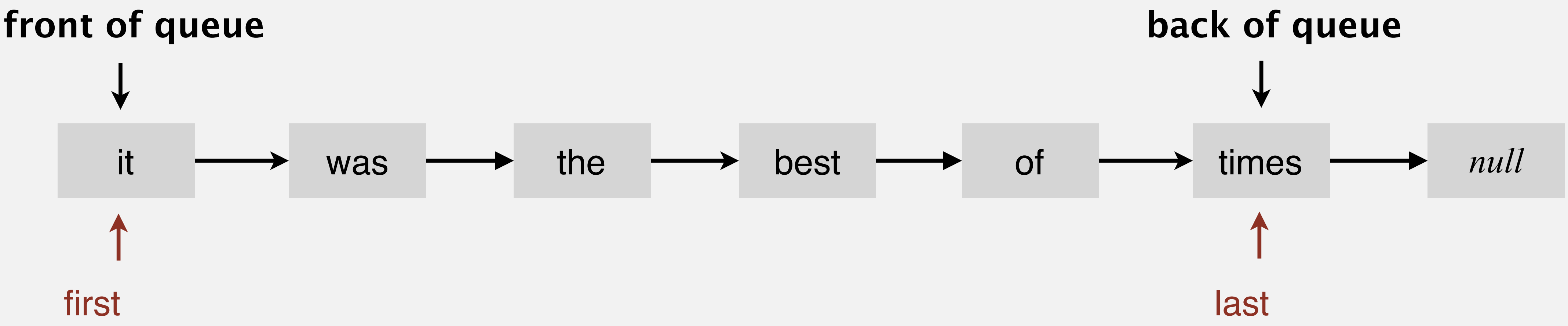


front of queue

back of queue

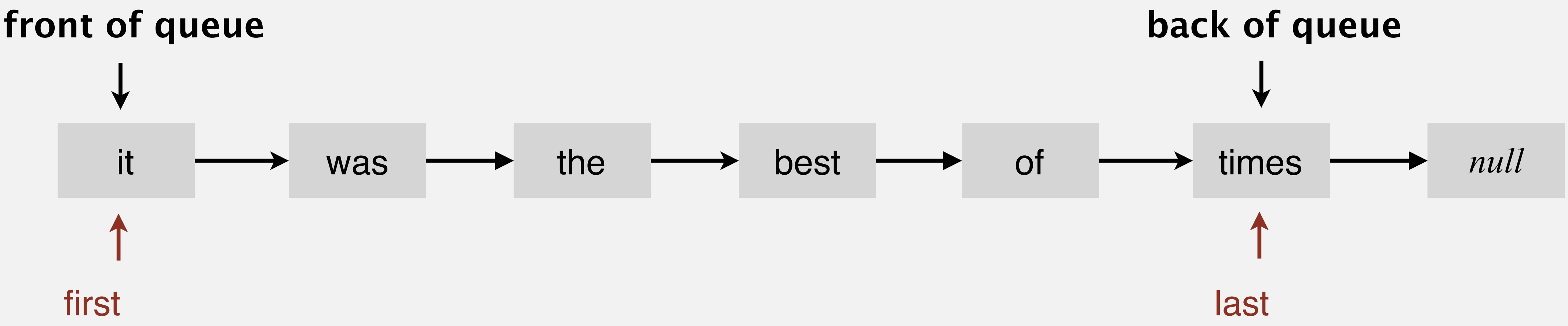
it → was → the → best → of → times → *null*

# Queue: linked-list implementation



# Queue: linked-list implementation

- Maintain one pointer first to first node in a singly-linked list.
- Maintain another pointer last to last node.
- Dequeue from first.
- Enqueue after last.



# Queue dequeue: linked-list implementation

inner class

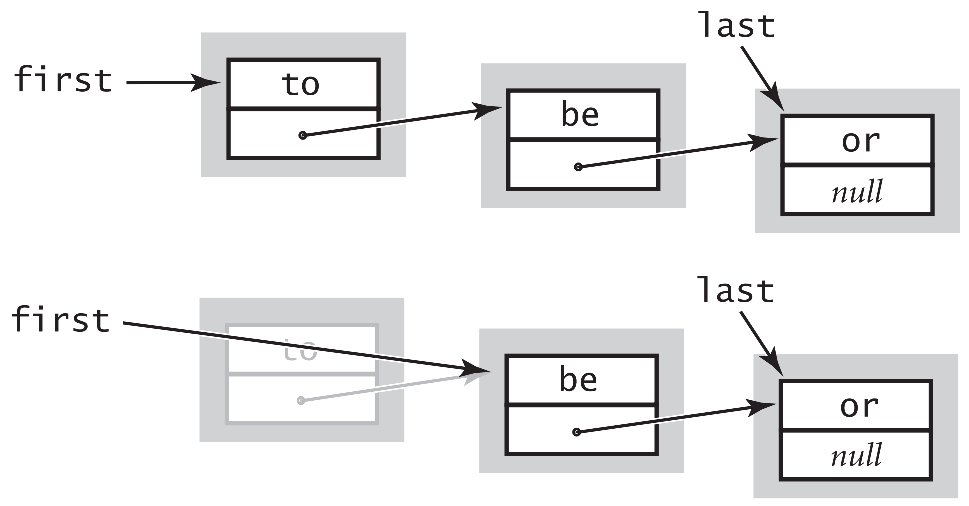
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

Remark. Identical code to linked-list stack pop().

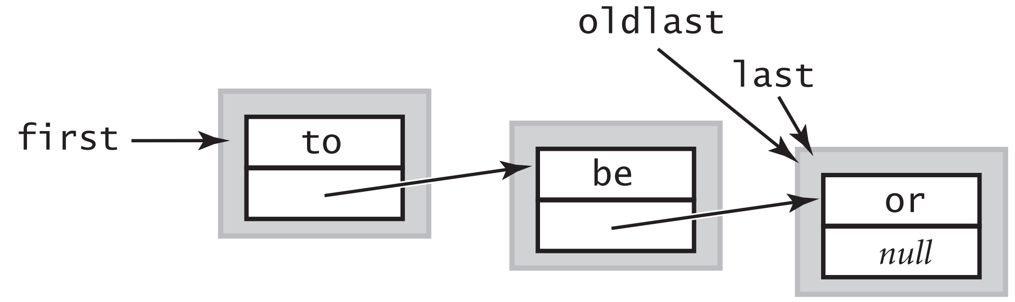
# Queue enqueue: linked-list implementation

inner class

```
private class Node
{
   String item;
   Node next;
}
```

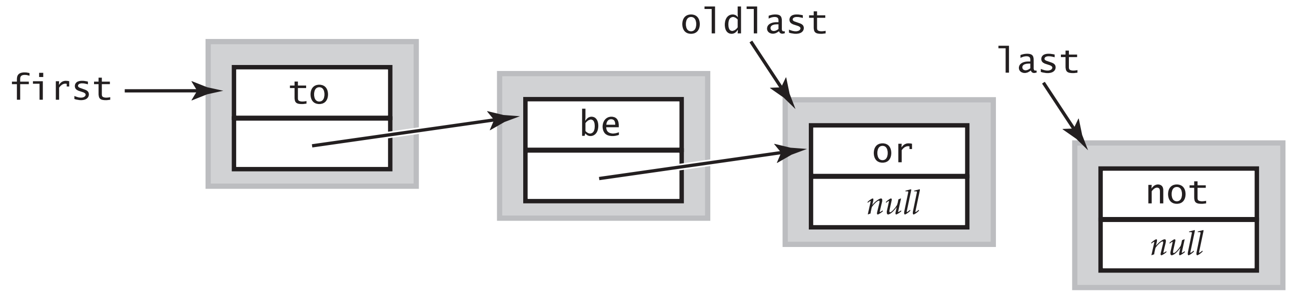
save a link to the last node

```
Node oldlast = last;
```



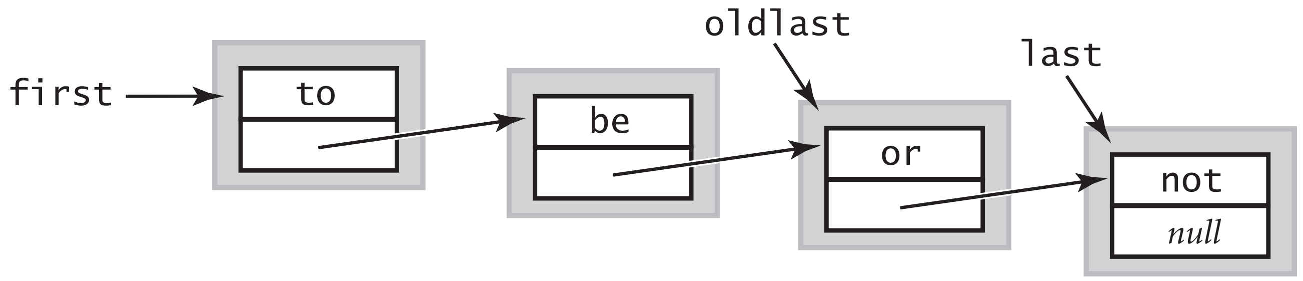
create a new node for the end

```
last = new Node();
last.item = "not";
```



link the new node to the end of the list

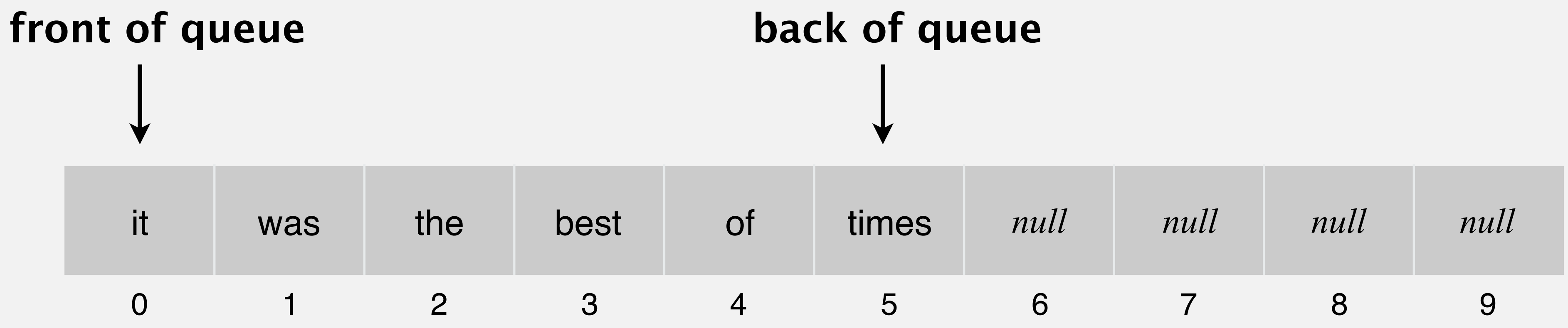
```
oldlast.next = last;
```



# How to implement a fixed-capacity queue with an array?

A. Can't be done efficiently with an array.

B.

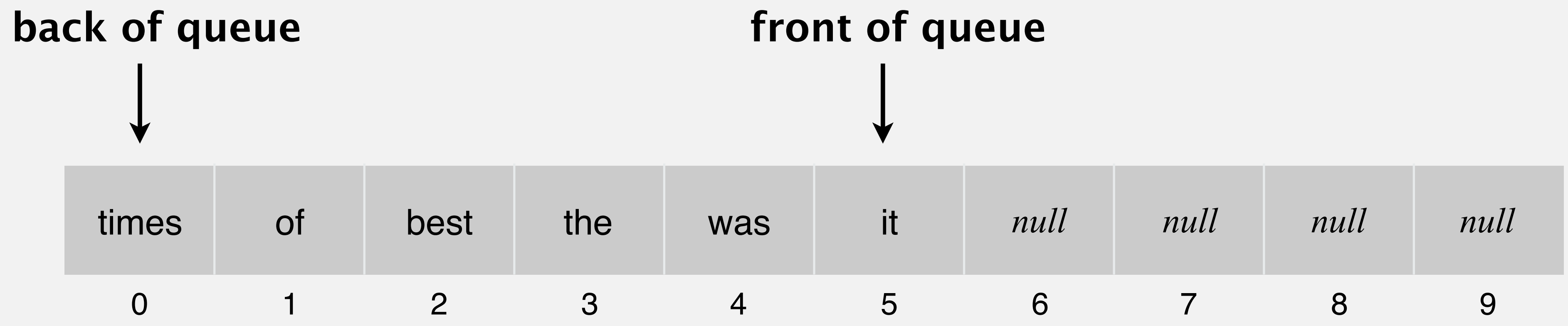


front of queue

back of queue

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| it | was | the | best | of | times | *null* | *null* | *null* | *null* |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

C.



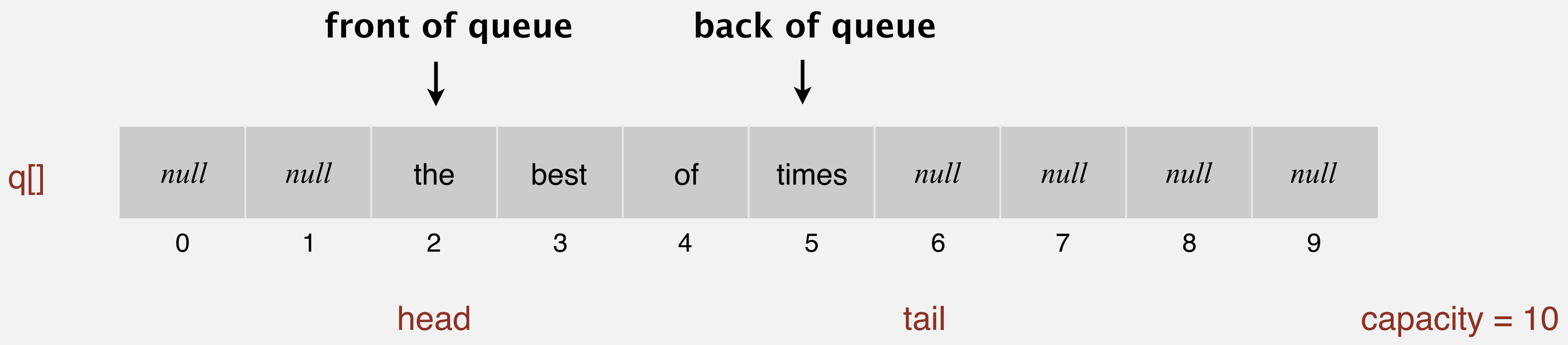
back of queue

front of queue

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| times | of | best | the | was | it | *null* | *null* | *null* | *null* |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Queue: resizing-array implementation

- Use array q[] to store items in queue.
- enqueue(): add new item at q[tail].
- dequeue(): remove item from q[head].
- Update head and tail modulo the capacity.
- Add resizing array.



front of queue

back of queue

q[]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| *null* | *null* | the | best | of | times | *null* | *null* | *null* | *null* |

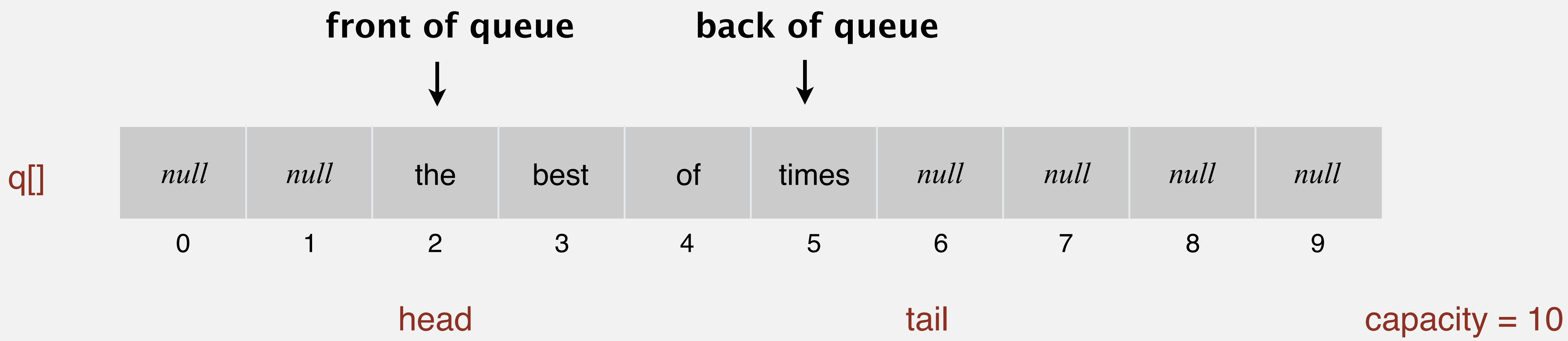
head

tail

capacity = 10

# Queue: resizing-array implementation

- Use array q[] to store items in queue.
- enqueue(): add new item at q[tail].
- dequeue(): remove item from q[head].
- Update head and tail modulo the capacity.
- Add resizing array.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| q[] | *null* | *null* | the | best | of | times | *null* | *null* | *null* | *null* |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Q. How to resize?

# Queue applications

## Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

## Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Playlists

**Lately**
Virginia Coalition

**Valerie**
Mark Ronson and Amy Winehouse

**How the Day Sounds**
Greg Laswell

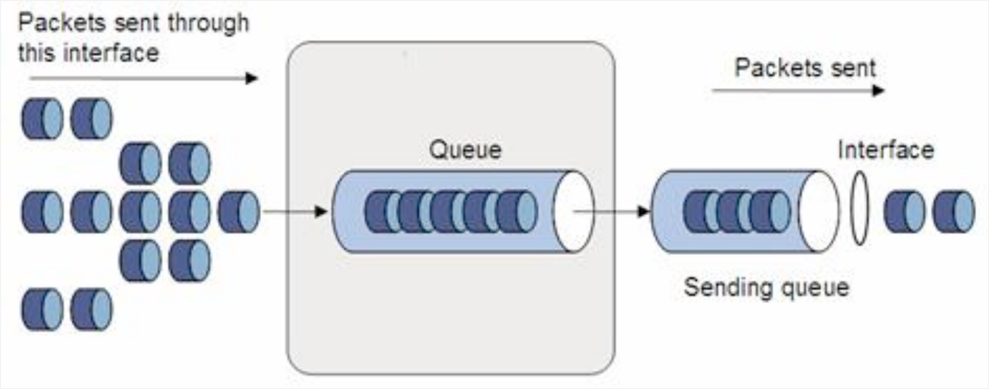
**I Will Possess Your Heart**
Death Cab for Cutie

**100 Days, 100 Nights**
Sharon Jones and the Dap-Kings

**Many the Miles**
Sara Bareilles

**Dirty City**
Steve Winwood and Eric Clapton

**Dream Catch Me**
Newton Faulkner



(CNN.COM GRPAHIC)