SPRING 2023

Information Technology Research

Yi Han

Department of Information Management
National Sun Yat-sen University

Lecture slides are based on the supplemental materials of the textbook: https://algs4.cs.princeton.edu

http://algs4.cs.princeton.edu

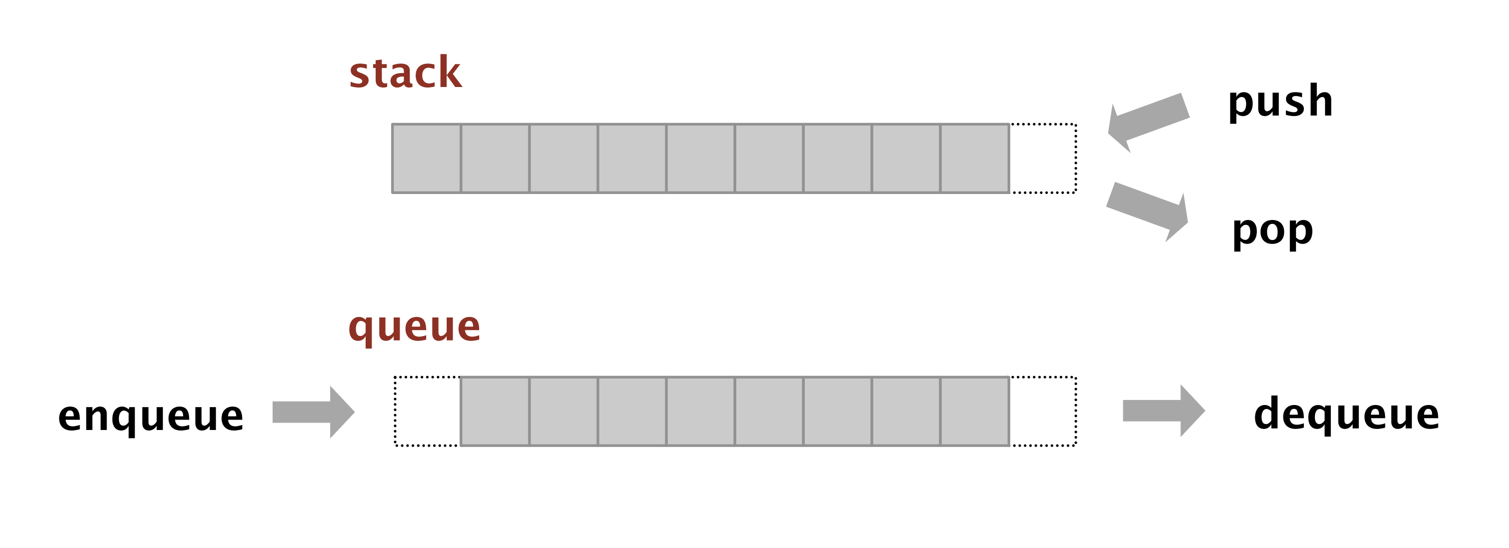
1.3 STACKS AND QUEUES

- ‣ *stacks*
- ‣ *resizing arrays*
- ‣ *queues*
- ‣ *iterators*
- ‣ *applications*

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Different methods and names for stacks and queues.



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

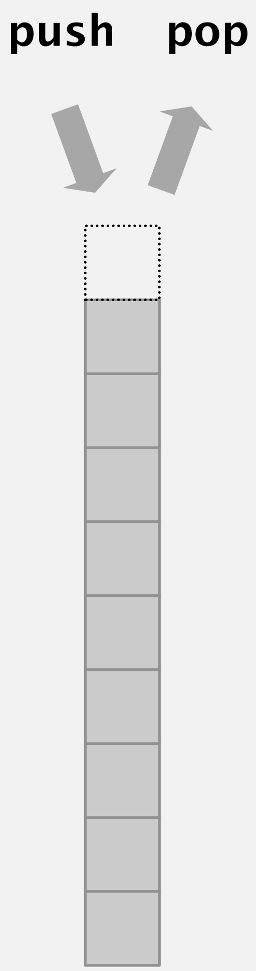
1.3 STACKS AND QUEUES

- ▸ *stacks*
- ▸ *resizing arrays*
- ▸ *queues*
- ▸ *iterators*
- ▸ *applications*

Stack API

Stack of strings data type.

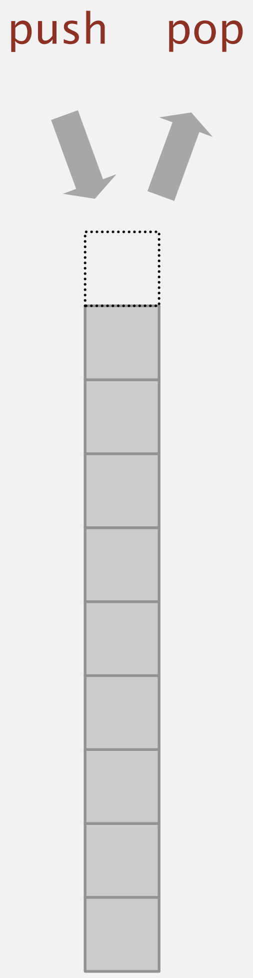
public class	StackOfStrings	
	`StackOfStrings()`	*create an empty stack*
`void`	`push(String item)`	*insert a new string onto stack*
`String`	`pop()`	*remove and return the string most recently added*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of strings on the stack*



Stack test client

Read strings from standard input.

- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.



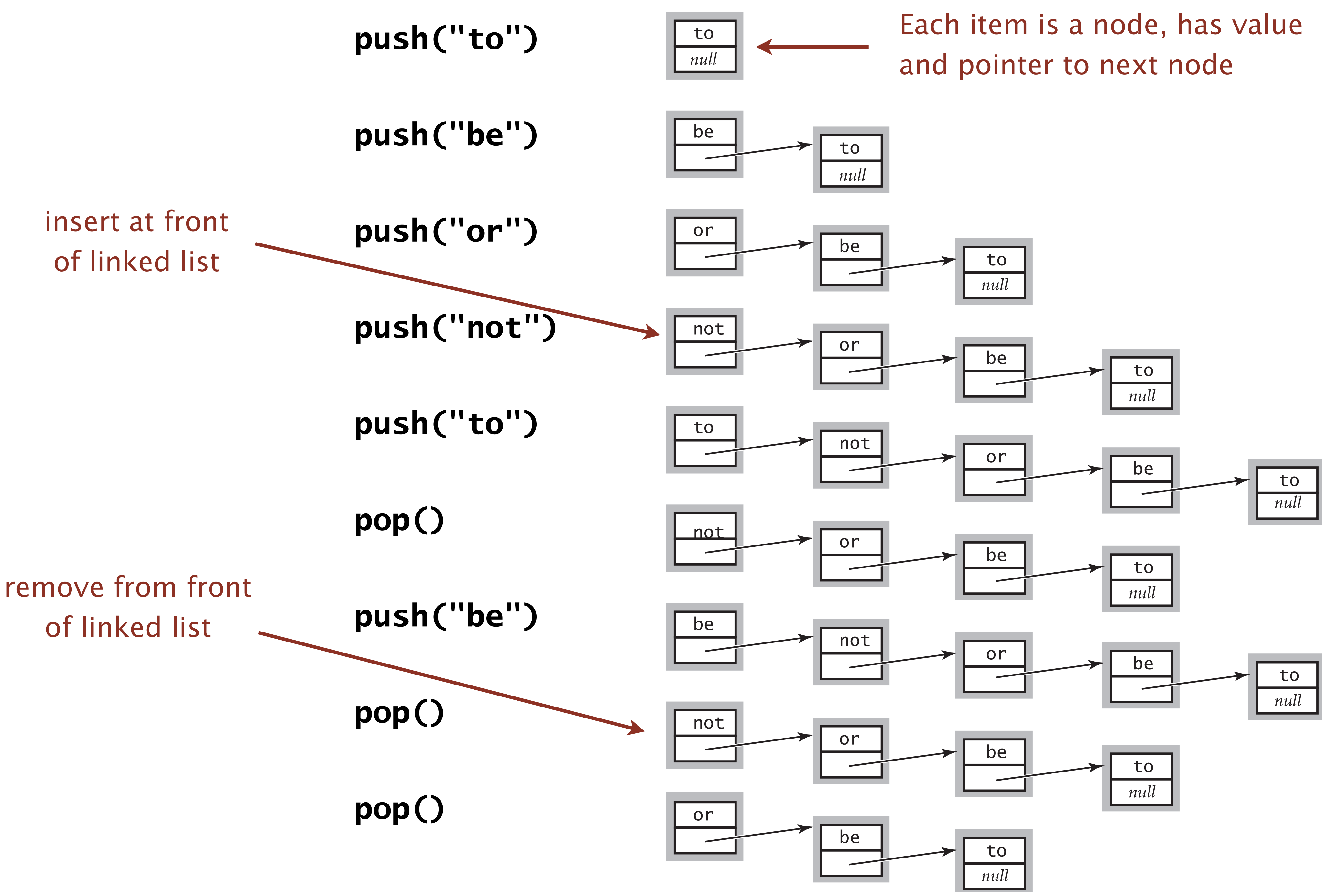
```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String s = StdIn.readString();
      if (s.equals("-")) StdOut.print(stack.pop());
      else               stack.push(s);
   }
}
```

```
% more tobe.txt
to be or not to - be - -
% java StackOfStrings < tobe.txt
```

```
to be not
```

Stack: linked-list representation

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



push("to")

push("be")

push("or")

push("not")

push("to")

pop()

push("be")

pop()

pop()

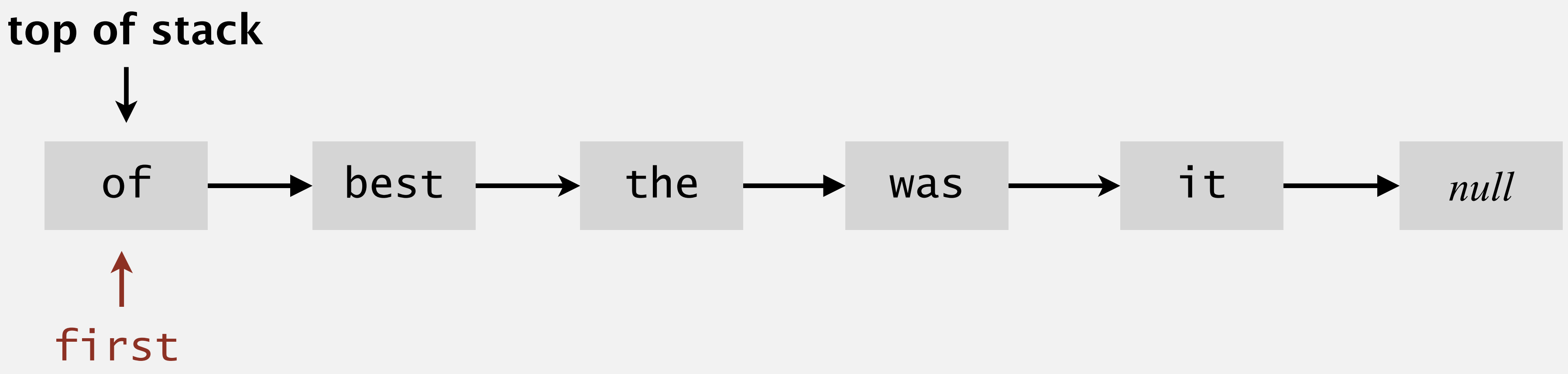
Each item is a node, has value and pointer to next node

insert at front of linked list

remove from front of linked list

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 pop: linked-list implementation

inner class

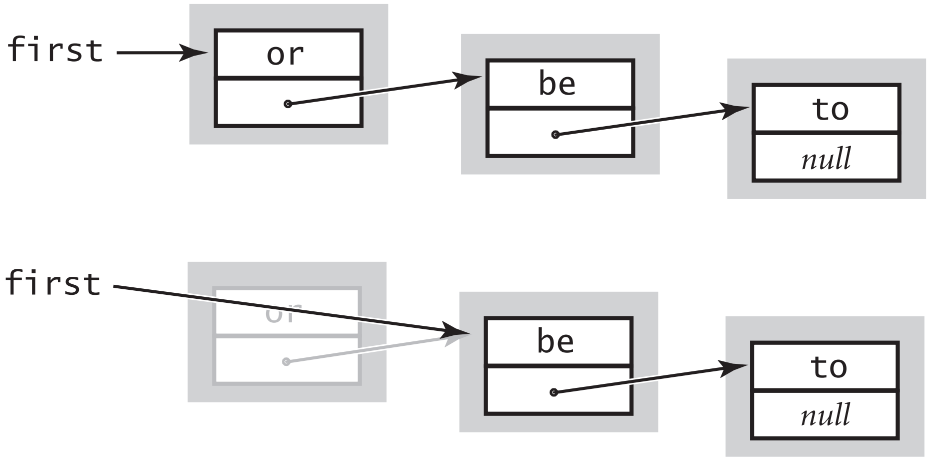
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {

      What to write here? 4 mins.
      Use new Node() to create new node.

   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

private inner class
(access modifiers for instance
variables don't matter)

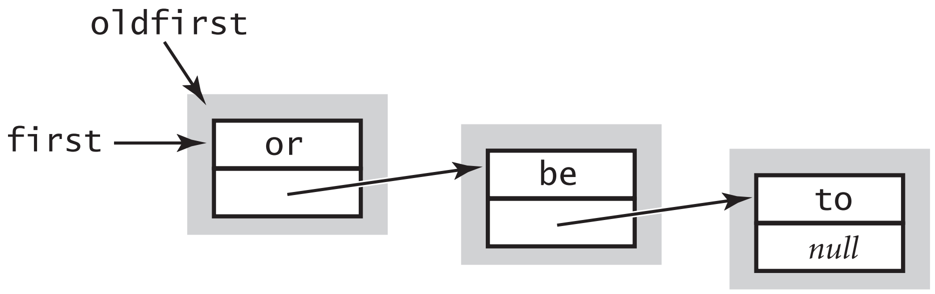
Stack push: linked-list implementation

inner class

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

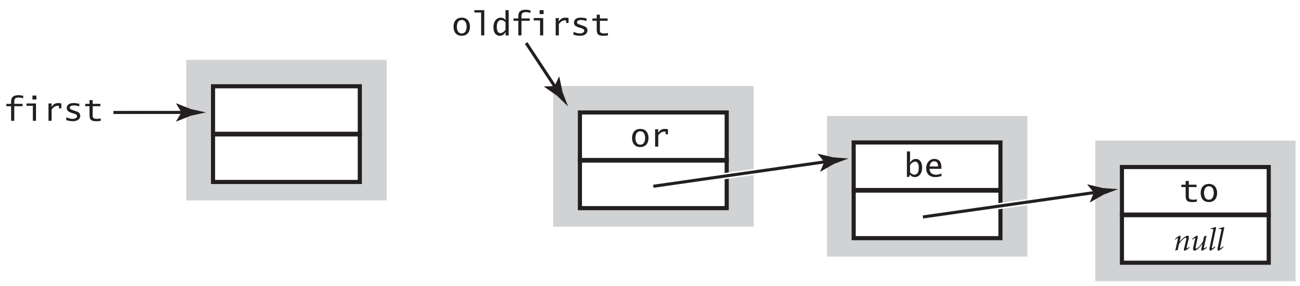
save a link to the list

```
Node oldfirst = first;
```



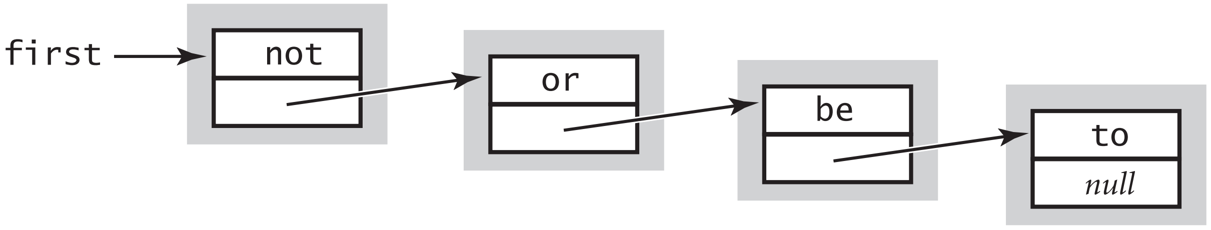
create a new node for the beginning

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



set the instance variables in the new node

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



Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

private inner class
(access modifiers for instance
variables don't matter)

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

Running Time of Stack with Linked List implementation?

- *Push?*
- *Pop?*

Constant
Linear
Logarithmic
Quadratic

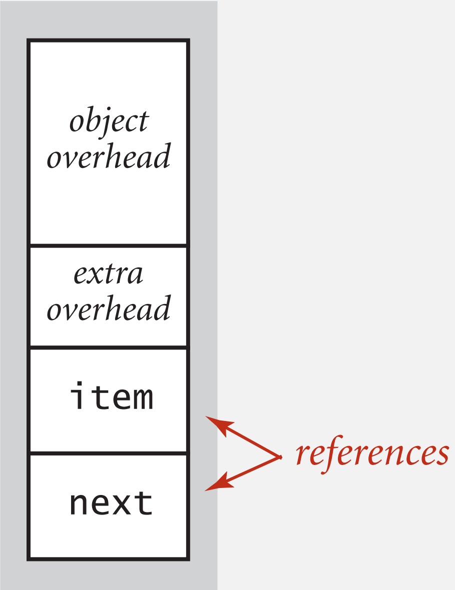
Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with N items uses $\sim 40\,N$ bytes.

inner class

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



16 bytes (object overhead)
8 bytes (inner class extra overhead)
8 bytes (reference to String)
8 bytes (reference to Node)
40 bytes per stack node

Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).

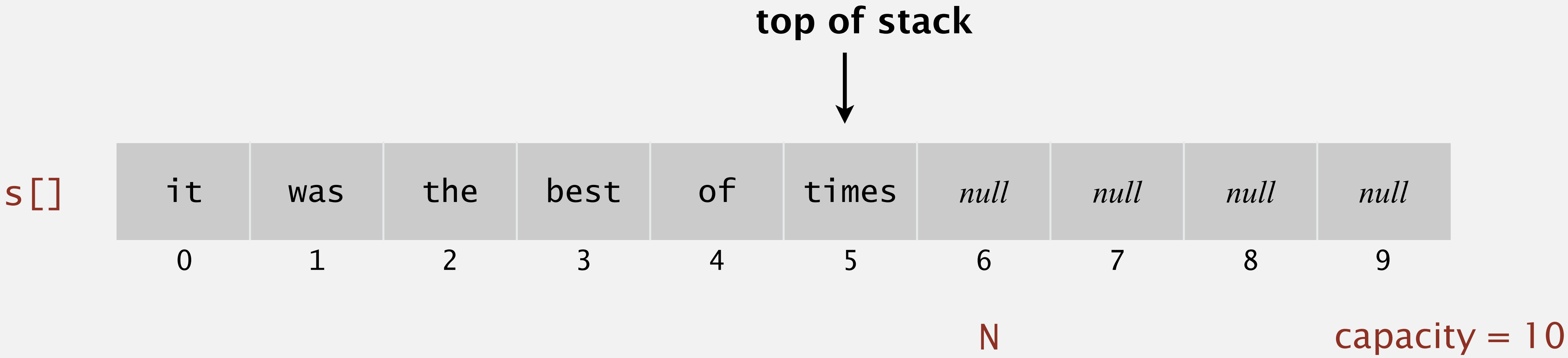
http://algs4.cs.princeton.edu

How about an array implementation?

- *Will it be better?*

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]`.



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

Fixed-capacity stack: array implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

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

not ideal
(stay tuned)

Any Problems?

use to index into array;
then increment N

decrement N;
then use to index into array

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**

http://algs4.cs.princeton.edu

1.3 Stacks and Queues

- ‣ *stacks*
- ‣ *resizing arrays*
- ‣ *queues*
- ‣ *iterators*
- ‣ *applications*

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.
- Read through original array, push into new array.
- Array accesses to insert first N items $\sim N^2$.

↑ infeasible for large N

Challenge. Ensure that array resizing happens infrequently. Any ideas?

Stack: resizing-array implementation

Q. How to grow array?

"repeated doubling"

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

```
public ResizingArrayStackOfStrings()
{   s = new String[1];   }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}
```

Array accesses to insert first N = 2^i items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

1 array access per push

k array accesses to double to size k (ignoring cost to create new array)

Stack: resizing-array implementation

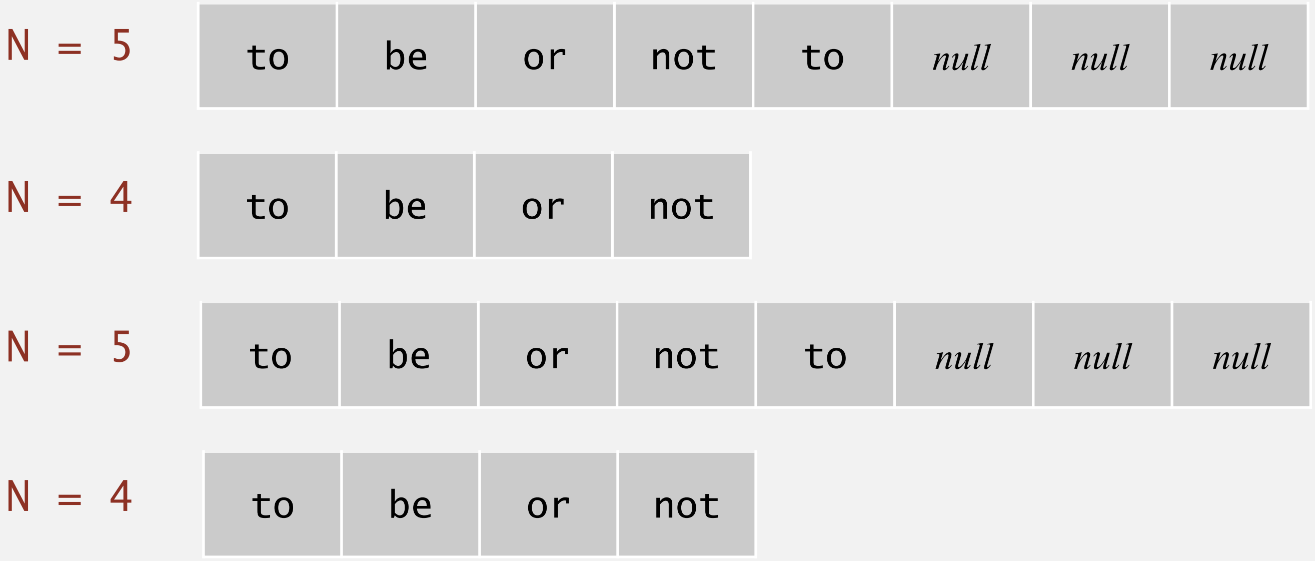
Q. How to shrink array?

First try.

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

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

Any Ideas?

- *To solve this problem*

http://algs4.cs.princeton.edu

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.

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   What to write here? 2 mins.
   return item;
}
```

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.

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length/2);
   return item;
}
```

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

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 resizing-array implementation: memory usage

Proposition. Uses between ~ $8N$ and ~ $32N$ bytes to represent a stack with N items.

- ~ $8N$ when full.
- ~ $32N$ when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
   private String[] s;
   private int N = 0;
   …
}
```

← 8 bytes × array size

Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).

Stack implementations: resizing array vs. linked list

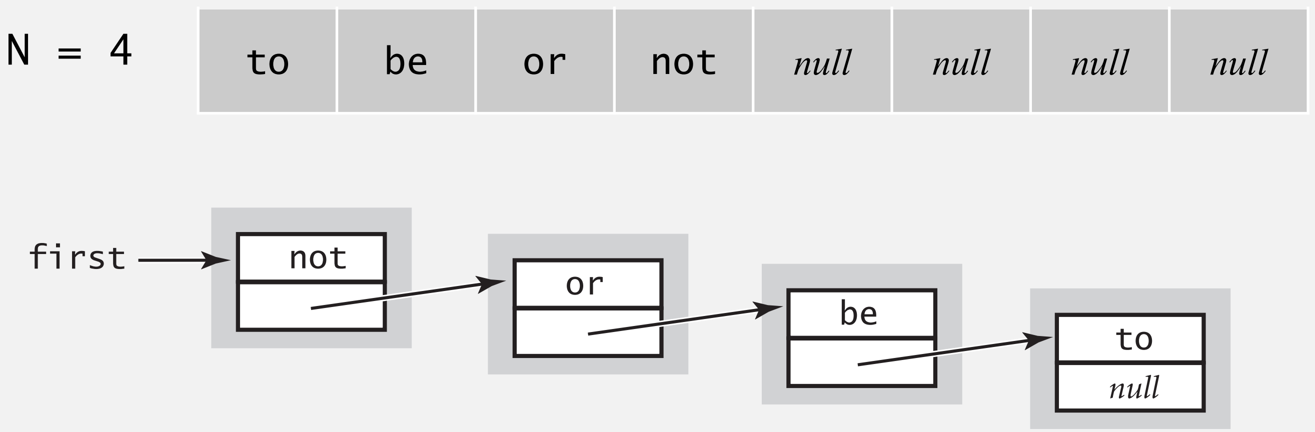
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

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.



Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

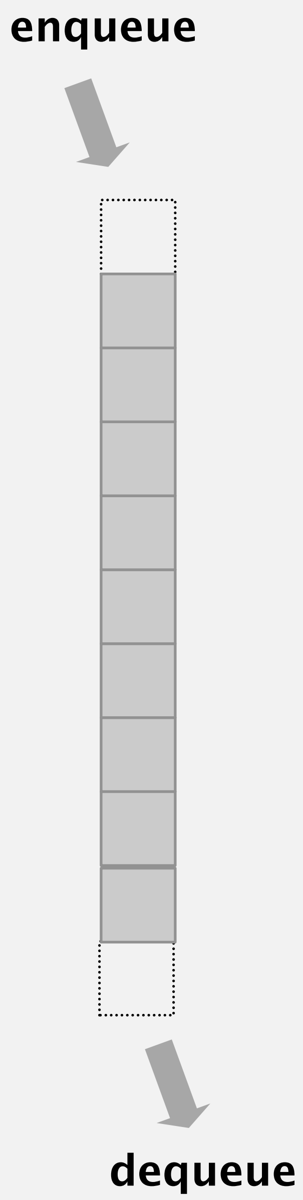
1.3 Stacks and Queues

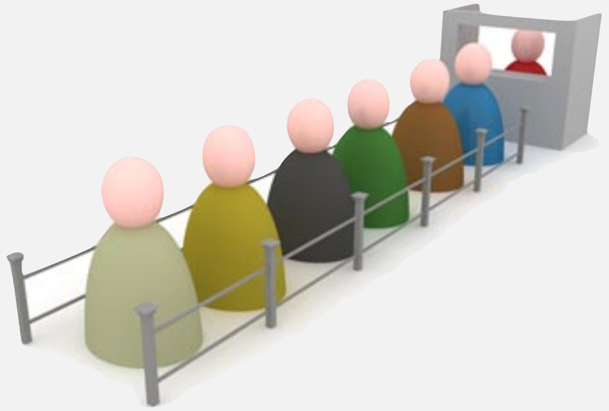
- *stacks*
- *resizing arrays*
- ***queues***
- *iterators*
- *applications*

Queue API

`public class QueueOfStrings`

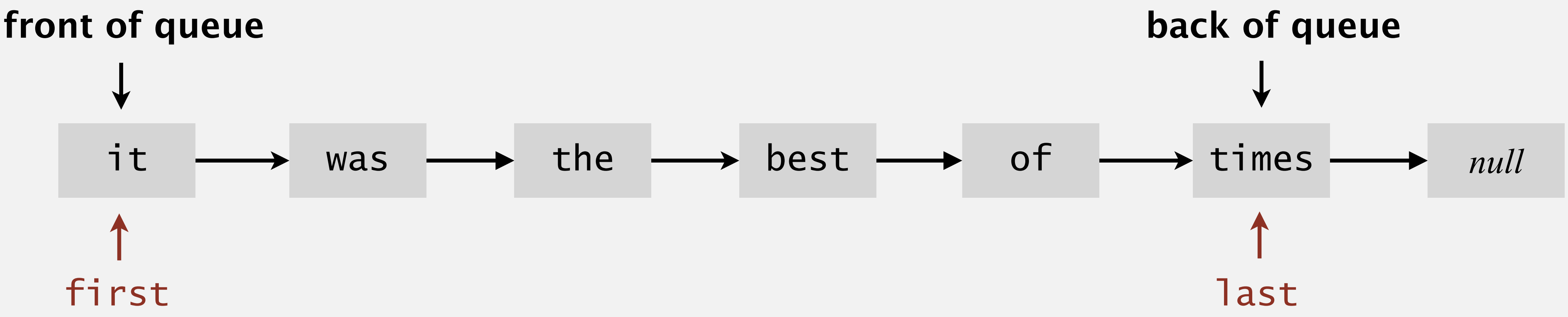
	`QueueOfStrings()`	*create an empty queue*
`void`	`enqueue(String item)`	*insert a new string onto queue*
`String`	`dequeue()`	*remove and return the string least recently added*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of strings on the queue*





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`.



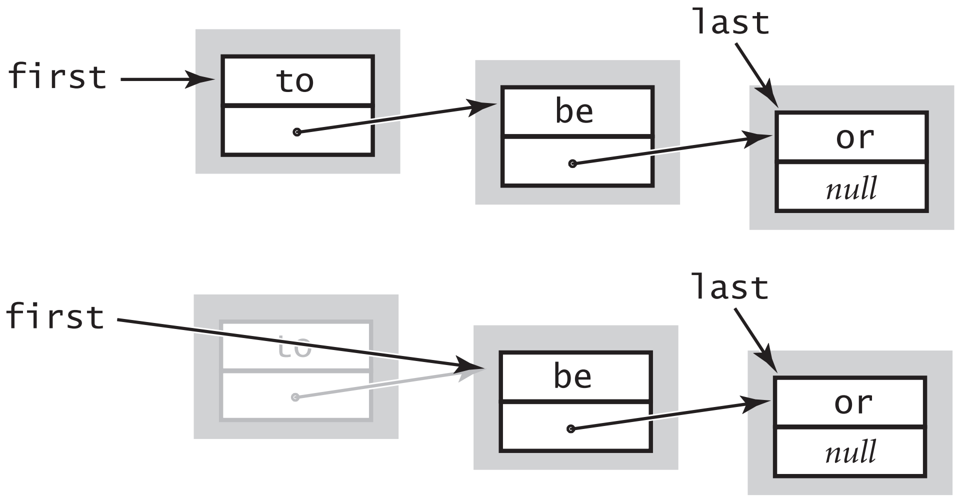
Queue dequeue: linked-list implementation

save item to return

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

delete first node

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



return saved item

```
return item;
```

inner class

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

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

Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in LinkedStackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {


     What to write here? 3 mins.
     Remember to handle empty queue.


   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

special cases for empty queue

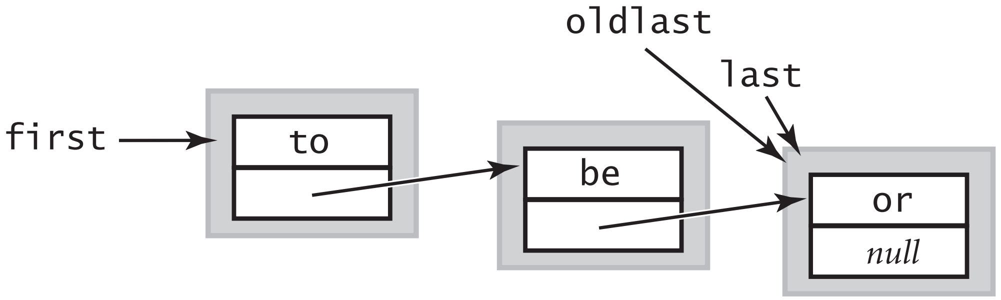
Queue enqueue: linked-list implementation

inner class

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

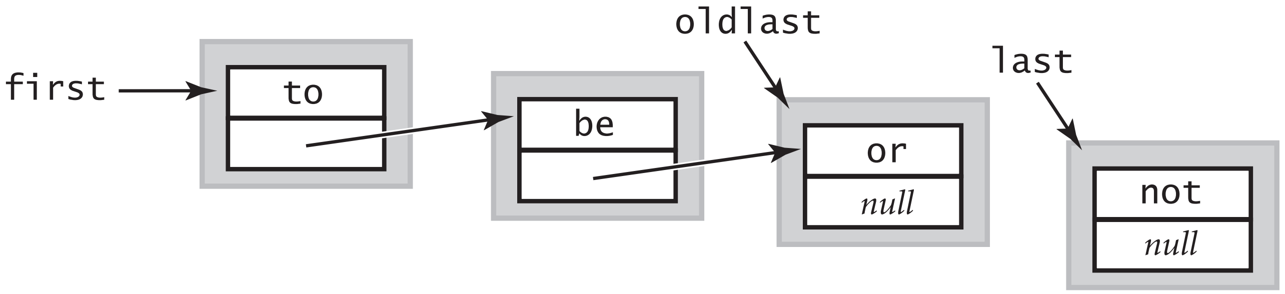
save a link to the last node

```
Node oldlast = last;
```



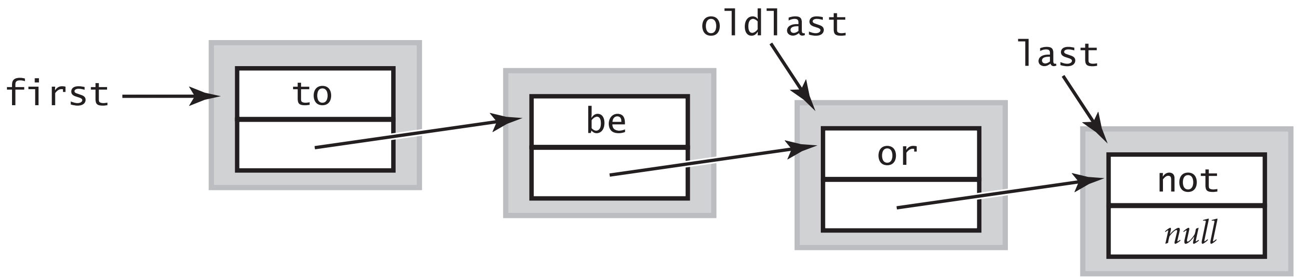
create a new node for the end

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



link the new node to the end of the list

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



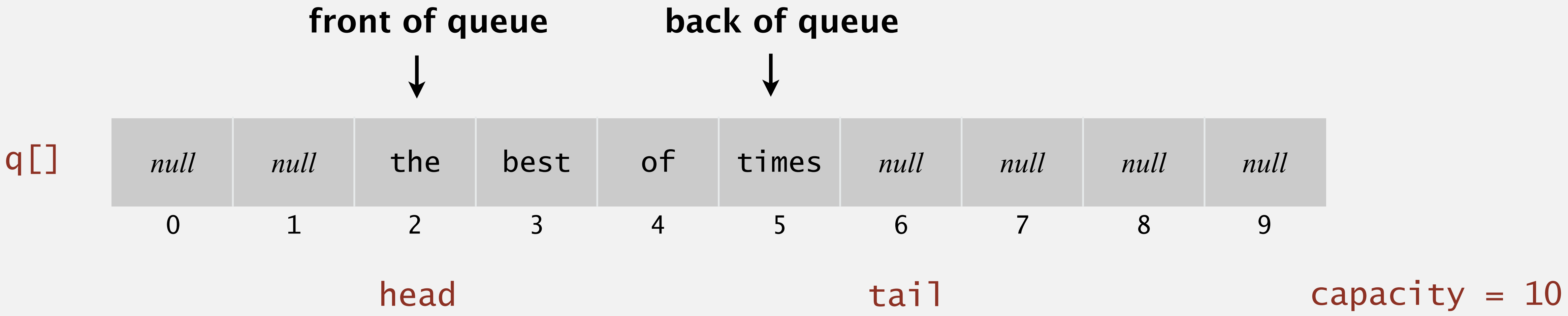
Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in LinkedStackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

special cases for empty queue

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]`.
- Add resizing array.



front of queue

back of queue

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

head

tail

capacity = 10

Q. How to resize?

Algorithms

Robert Sedgewick | Kevin Wayne

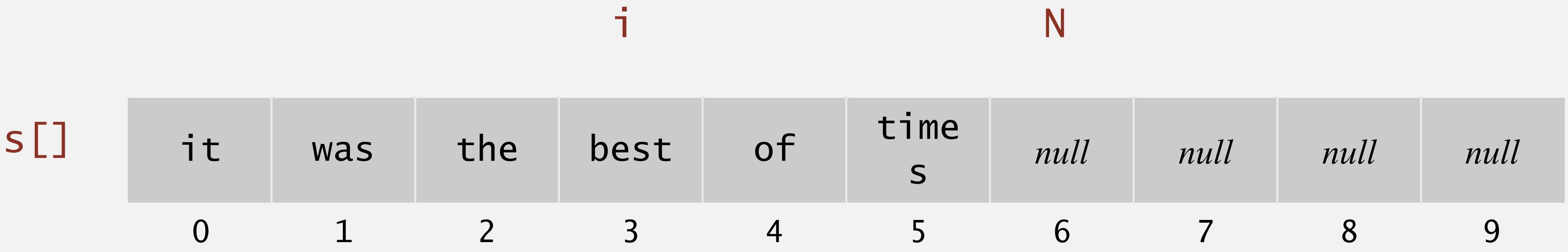
http://algs4.cs.princeton.edu

1.3 Stacks and Queues

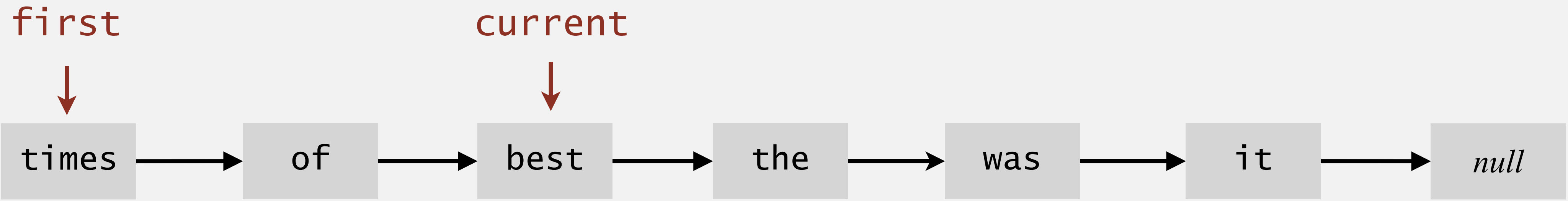
- ▸ *stacks*
- ▸ *resizing arrays*
- ▸ *queues*
- ▸ *iterators*
- ▸ *applications*

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



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



Java solution. Make stack implement the `java.lang.Iterable` interface.

Iterators

Q. What is an `Iterable` ?

A. Has a method that returns an `Iterator`.

java.lang.Iterable interface

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?

A. Has methods `hasNext()` and `next()`.

java.util.Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
}
```

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement (shorthand)

```
for (String s : stack)
   StdOut.println(s);
```

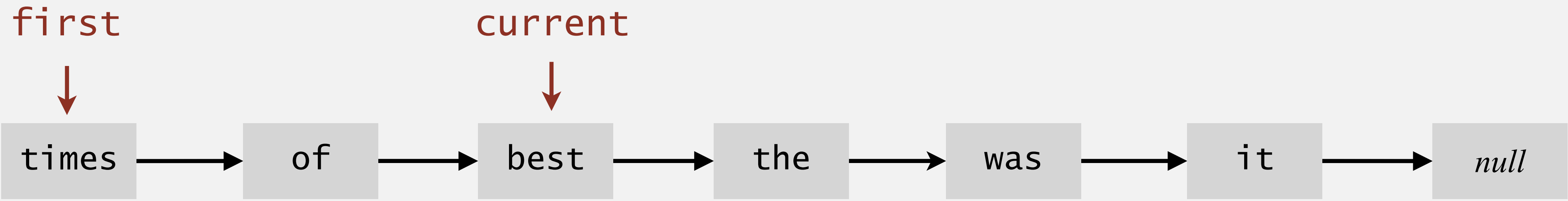
equivalent code (longhand)

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

Stack iterator: linked-list implementation

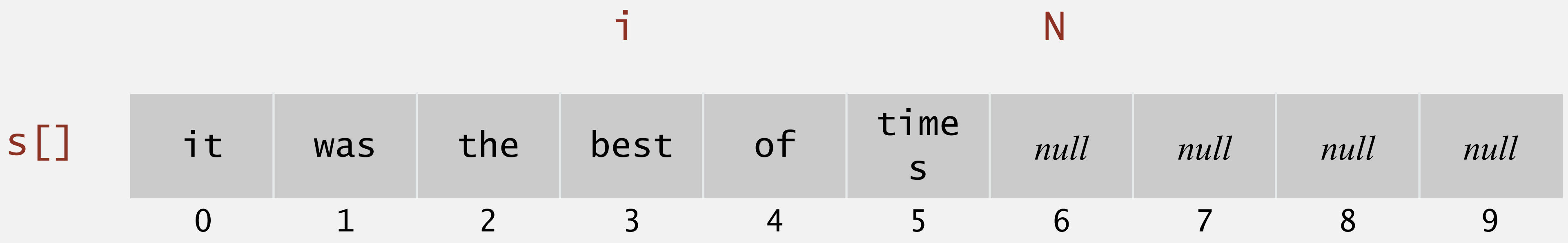
```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }

        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```

throw NoSuchElementException
if no more items in iteration



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    …

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;        }

        public Item next()       {  return s[--i];       }
    }
}
```



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

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

1.3 Stacks and Queues

- ‣ *stacks*
- ‣ *resizing arrays*
- ‣ *queues*
- ‣ *iterators*
- ‣ *applications*

Java collections library

List interface. `java.util.List` is API for an sequence of items.

public interface List<Item> implements Iterable<Item>		
	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*append item to the end*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
`Iterator<Item>`	`iterator()`	*iterator over all items in the list*
`...`		

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.
- Bloated and poorly-designed API (why?)

Java 1.3 bug report (June 27, 2001)

> The iterator method on java.util.Stack iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

status (closed, will not fix)

> It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.
- Bloated and poorly-designed API (why?)

`java.util.Queue`. An interface, not an implementation of a queue.

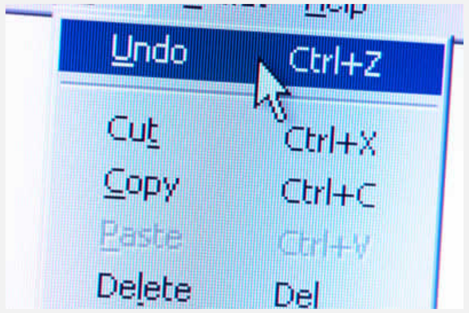
Best practices. Use our implementations of `Stack` and `Queue`.

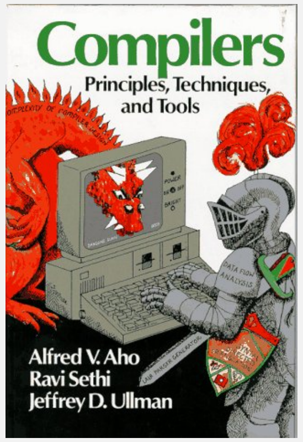
Stack applications

- 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

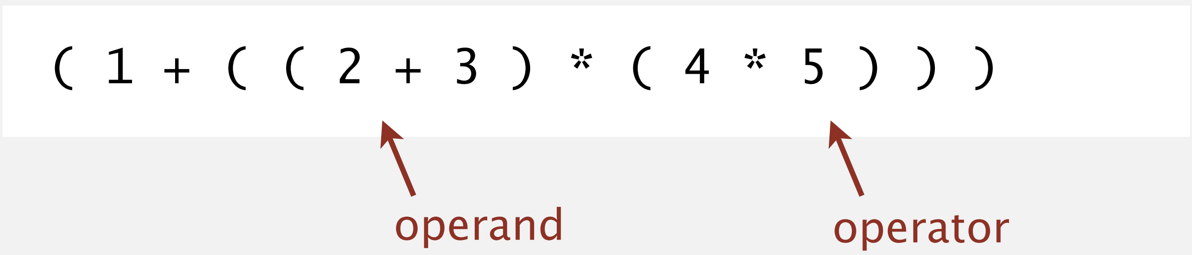
Adobe® PostScript® 3™





Arithmetic expression evaluation

Goal. Evaluate infix expressions.

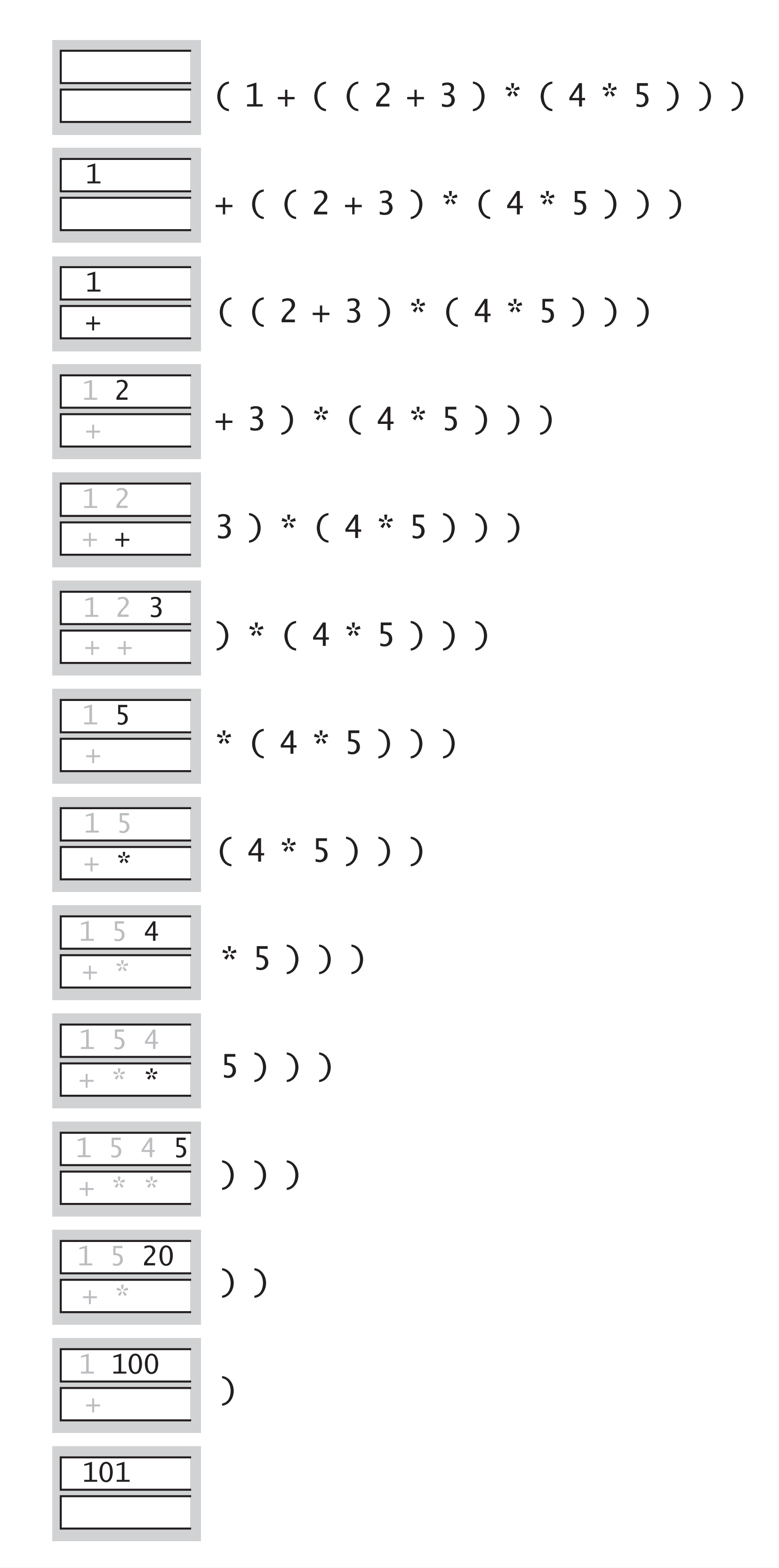


(1 + ((2 + 3) * (4 * 5)))

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	
		(1 + ((2 + 3) * (4 * 5)))
1		+ ((2 + 3) * (4 * 5)))
1	+	((2 + 3) * (4 * 5)))
1 2	+	+ 3) * (4 * 5)))
1 2	+ +	3) * (4 * 5)))
1 2 3	+ +	) * (4 * 5)))
1 5	+	* (4 * 5)))
1 5	+ *	(4 * 5)))
1 5 4	+ *	* 5)))
1 5 4	+ * *	5)))
1 5 4 5	+ * *	)))
1 5 20	+ *	))
1 100	+	)
101		

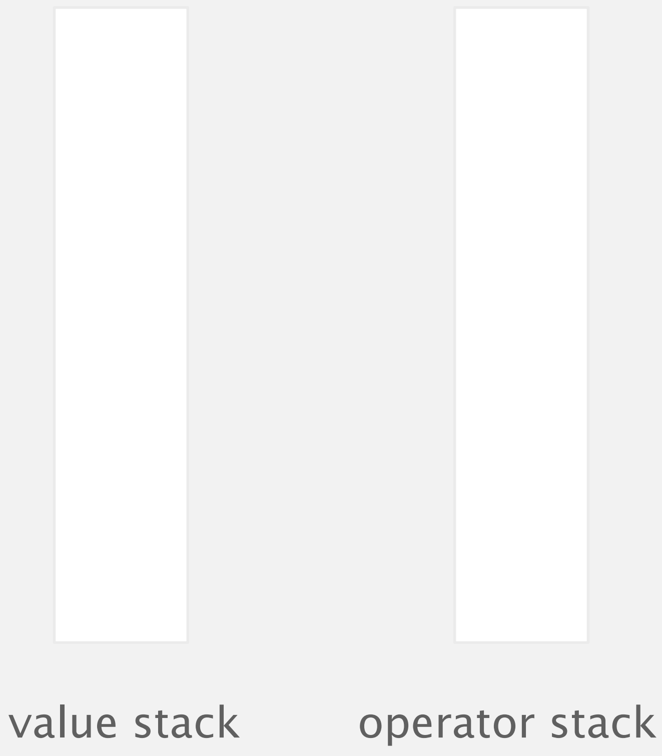
Dijkstra's two-stack algorithm

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.



infix expression
(fully parenthesized)

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

operand

operator

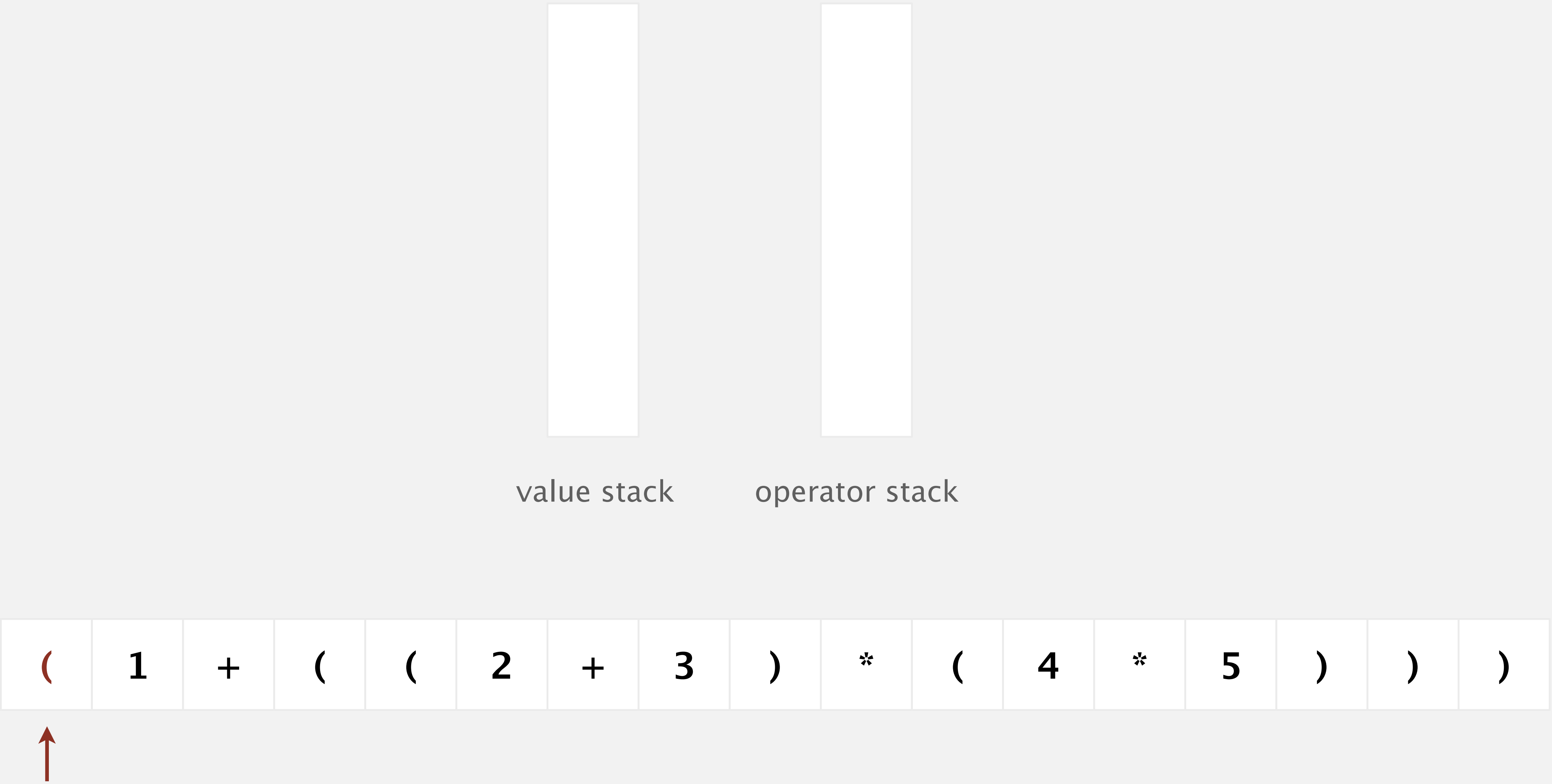
Dijkstra's two-stack algorithm

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)))

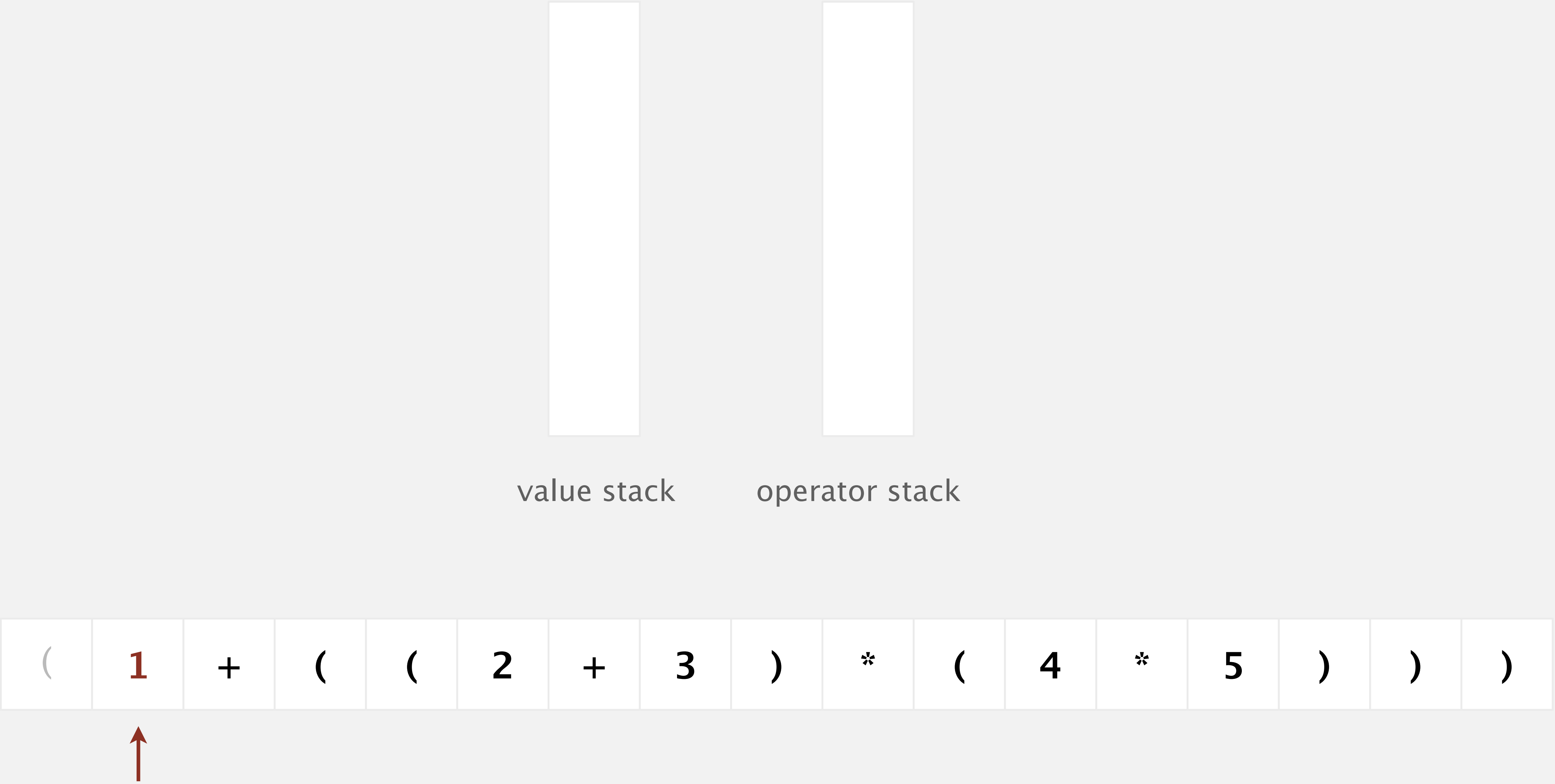
Dijkstra's two-stack algorithm

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.



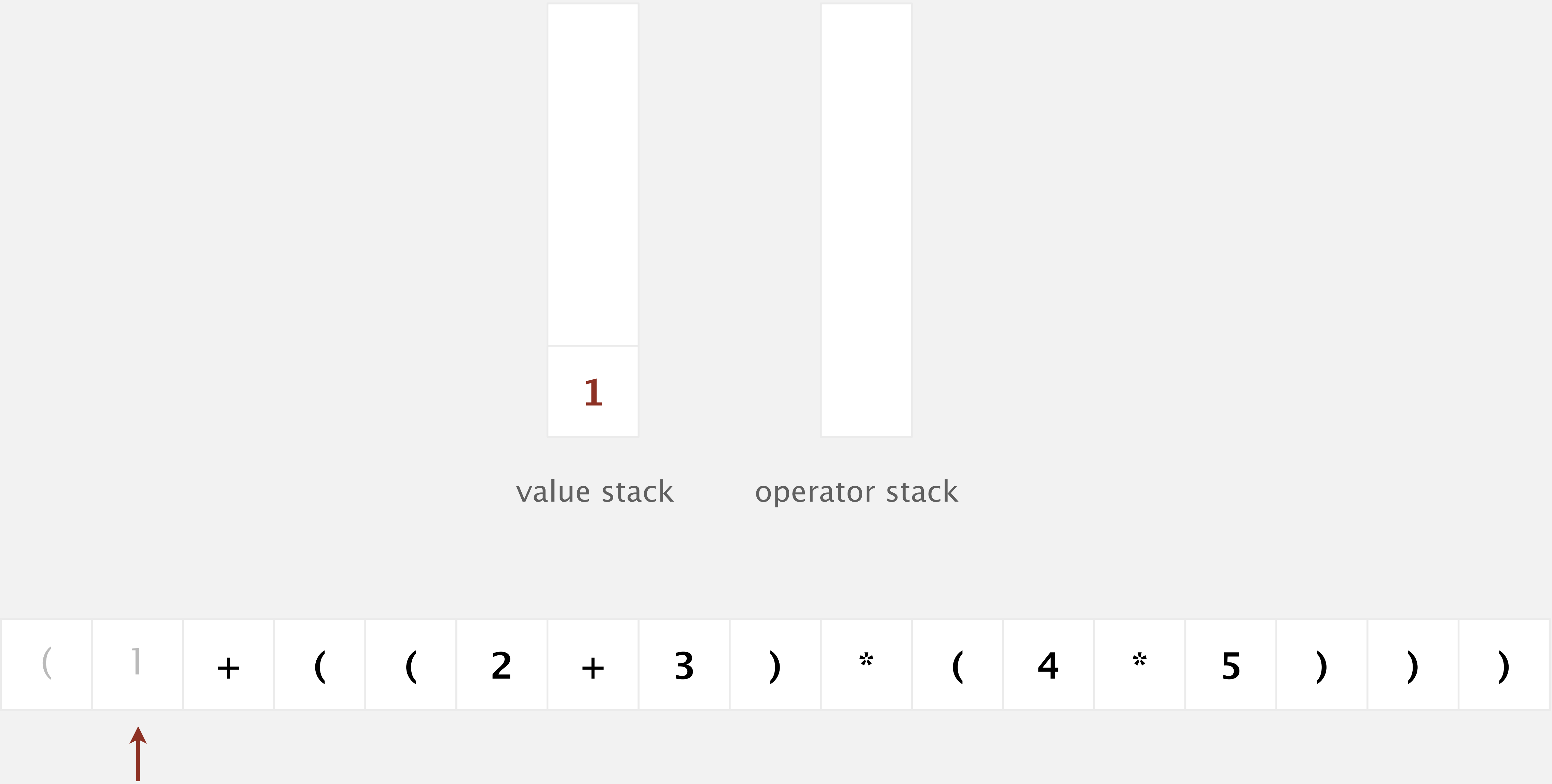
Dijkstra's two-stack algorithm

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)))

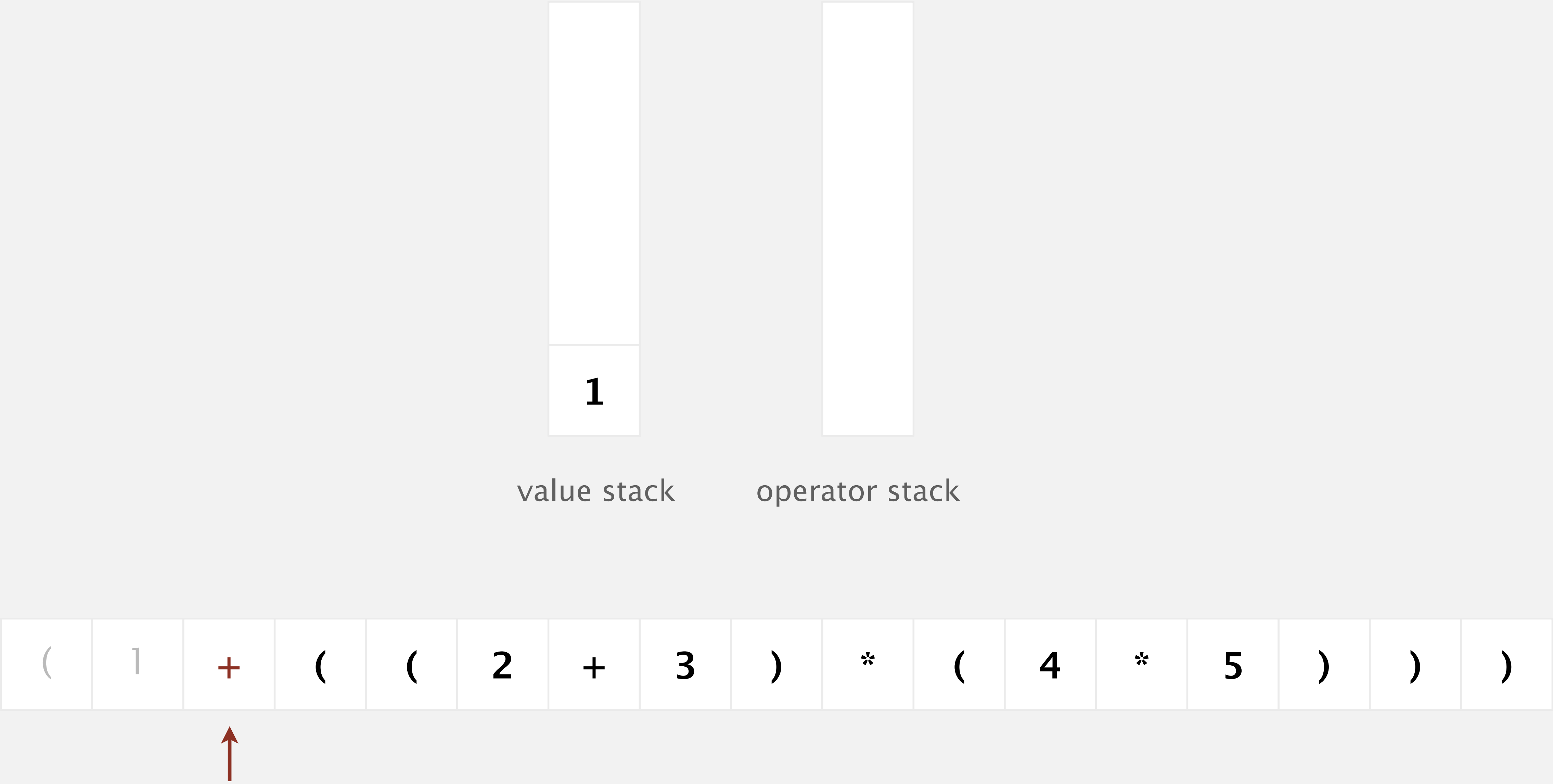
Dijkstra's two-stack algorithm

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

value stack

operator stack

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

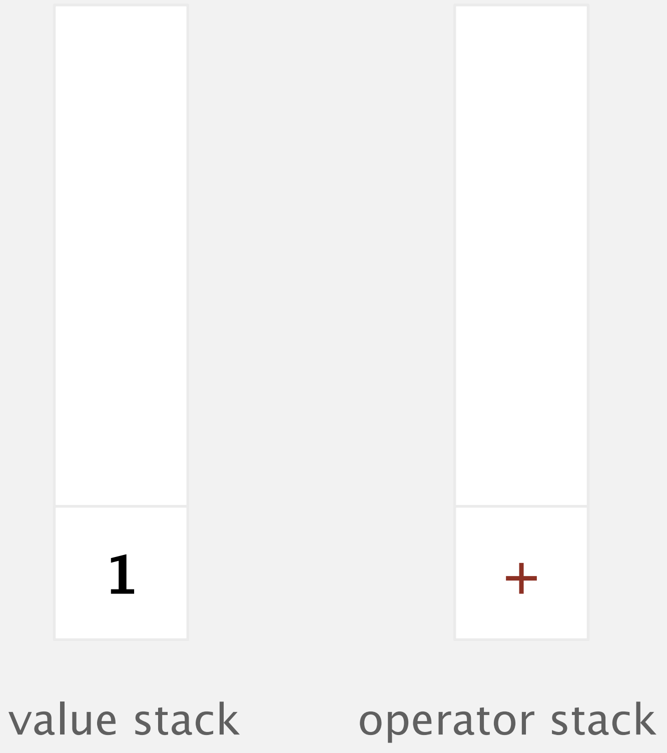
Dijkstra's two-stack algorithm

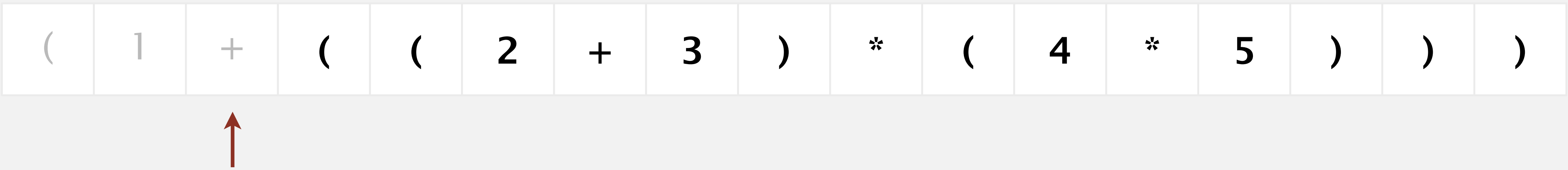
Value: push onto the value stack.

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.





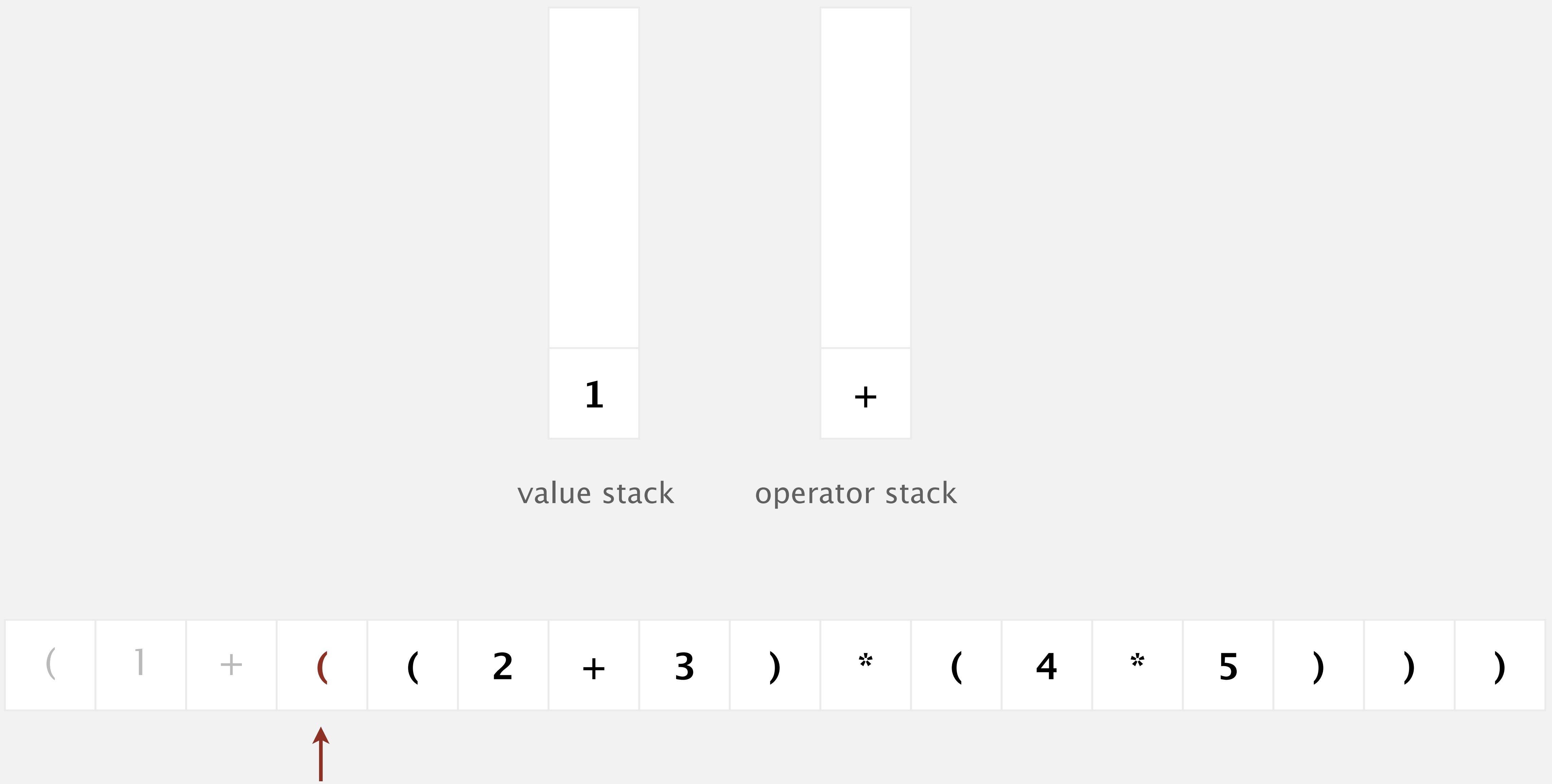
Dijkstra's two-stack algorithm

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

+

value stack

operator stack

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

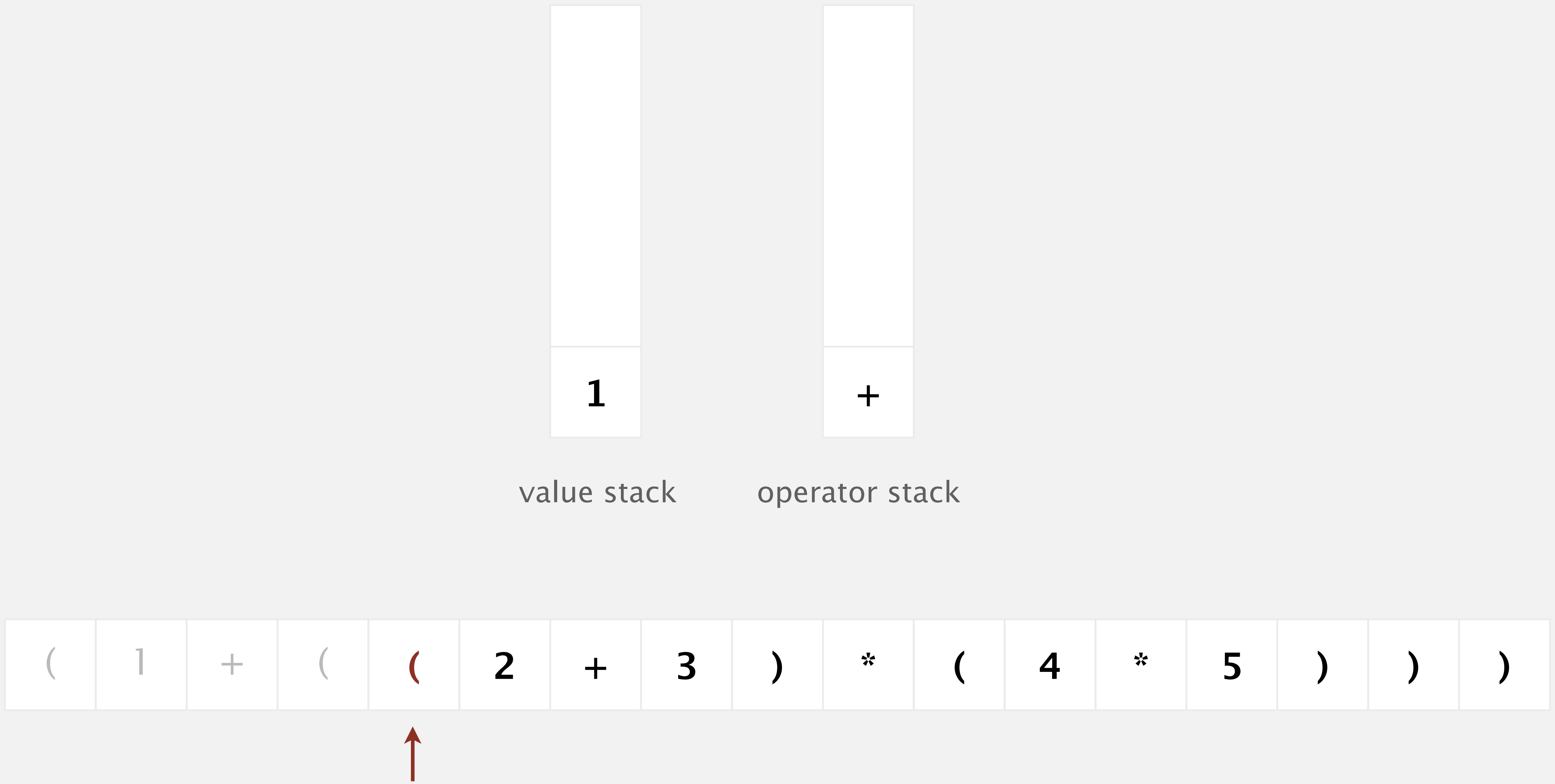
Dijkstra's two-stack algorithm

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

+

value stack

operator stack

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

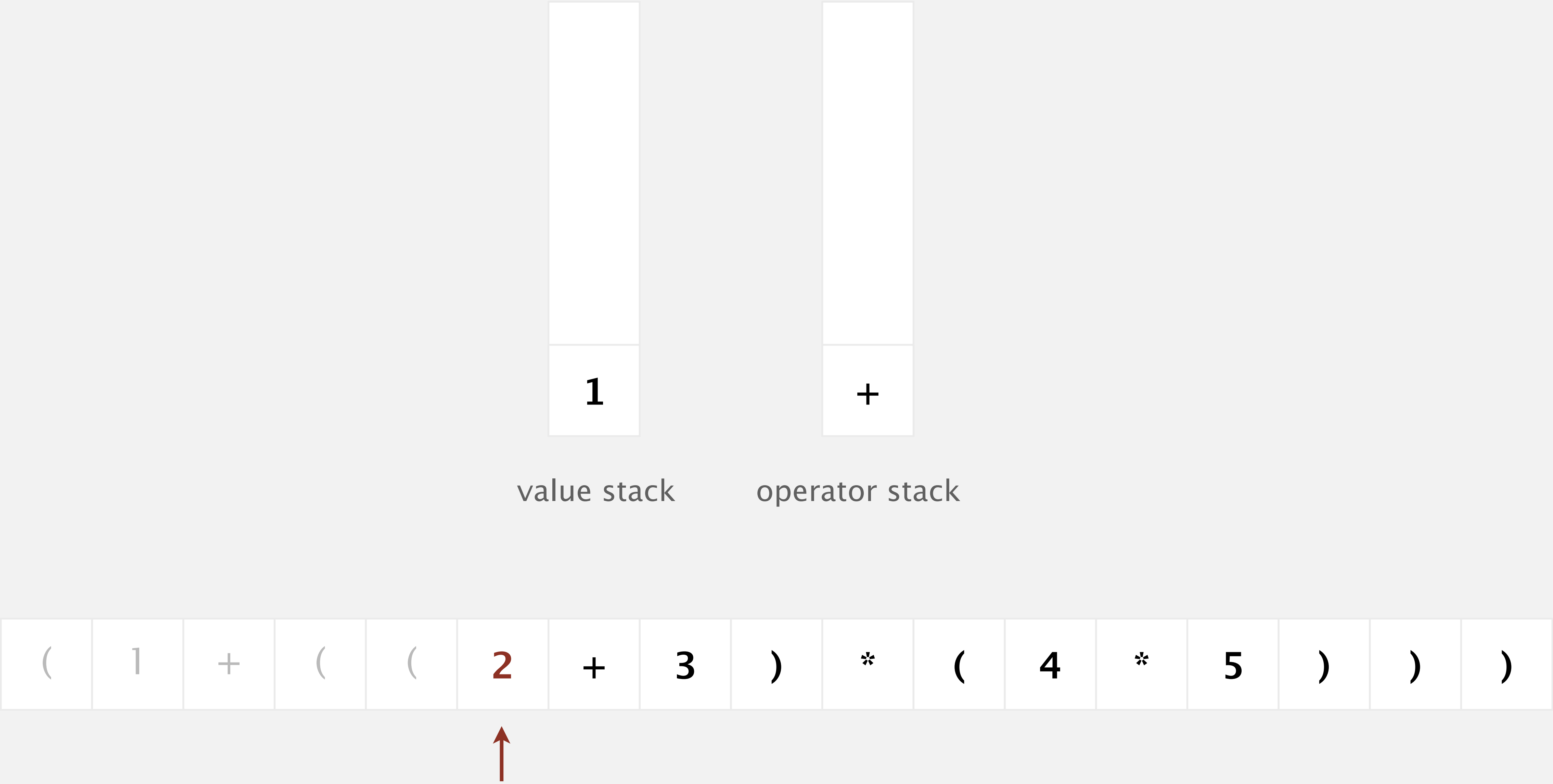
Dijkstra's two-stack algorithm

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.



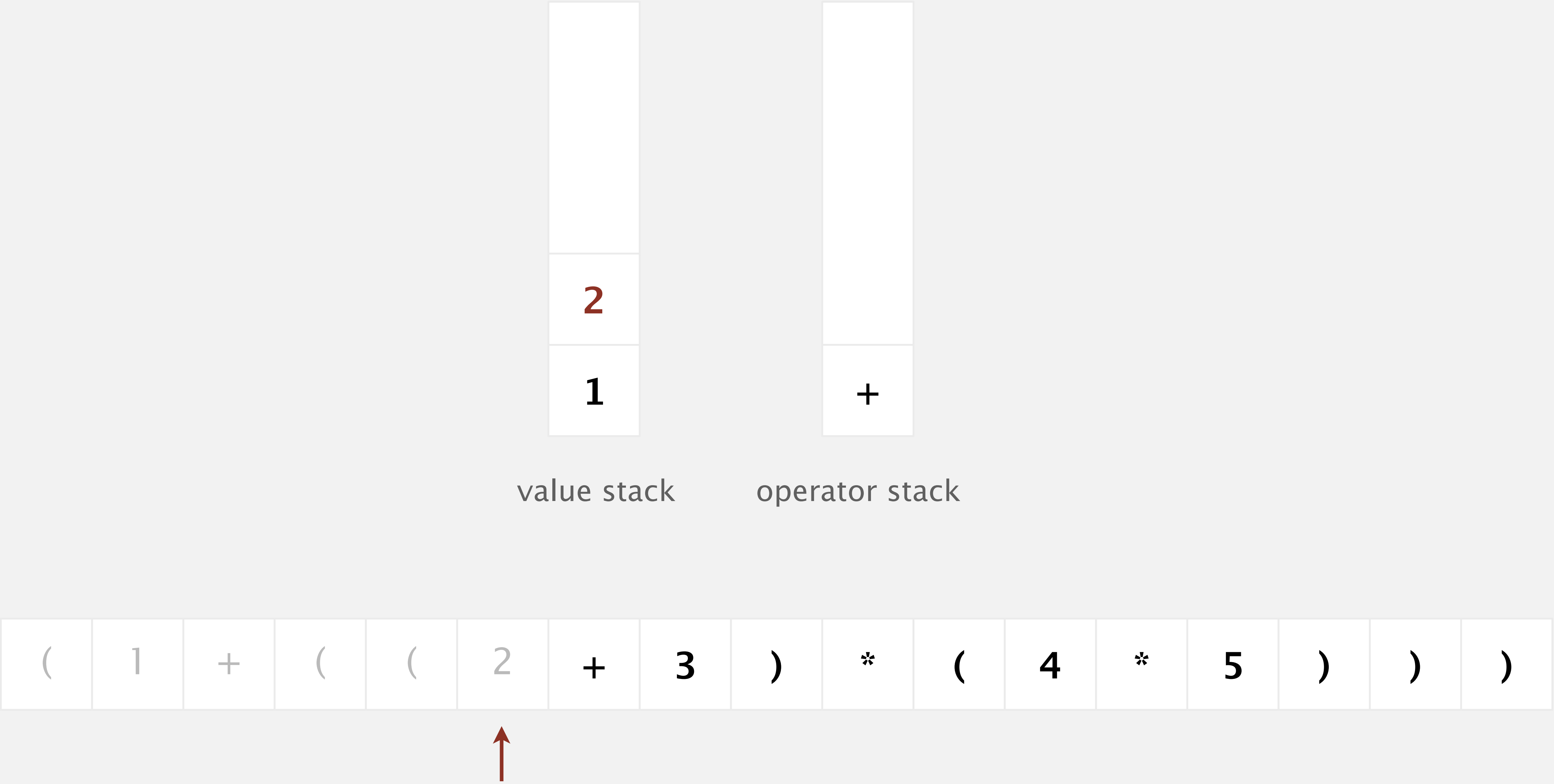
Dijkstra's two-stack algorithm

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.



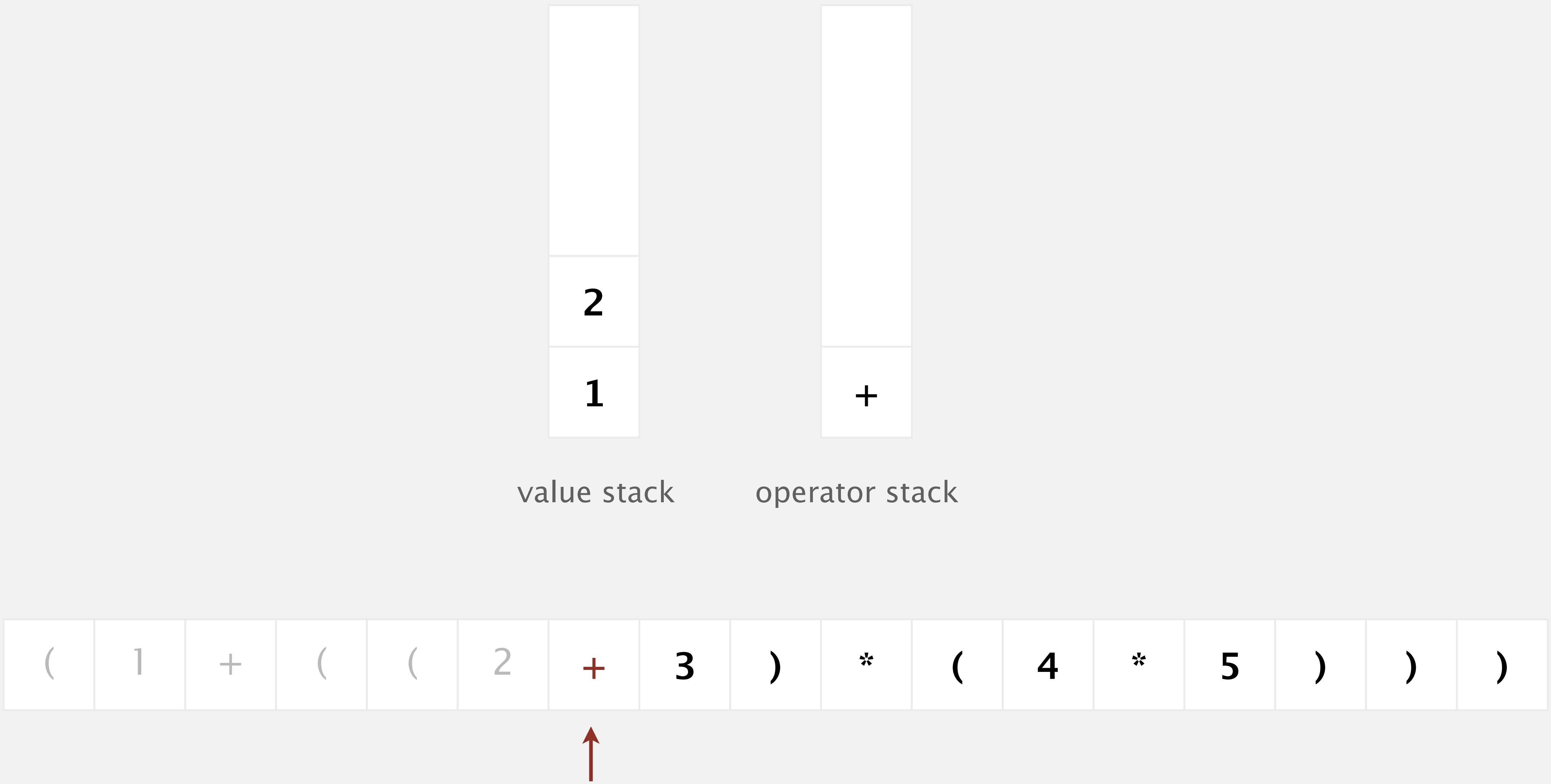
Dijkstra's two-stack algorithm

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.



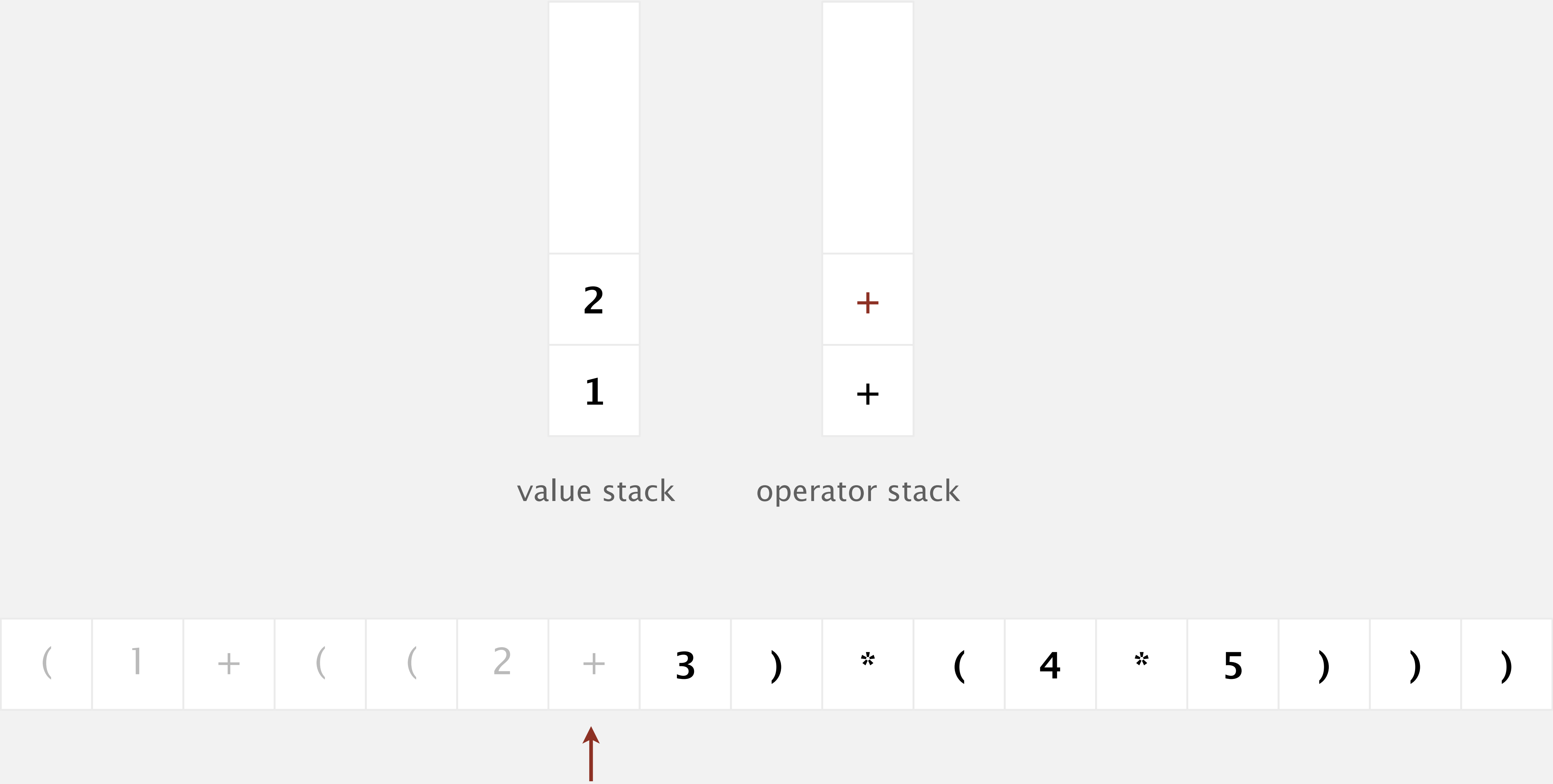
Dijkstra's two-stack algorithm

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.



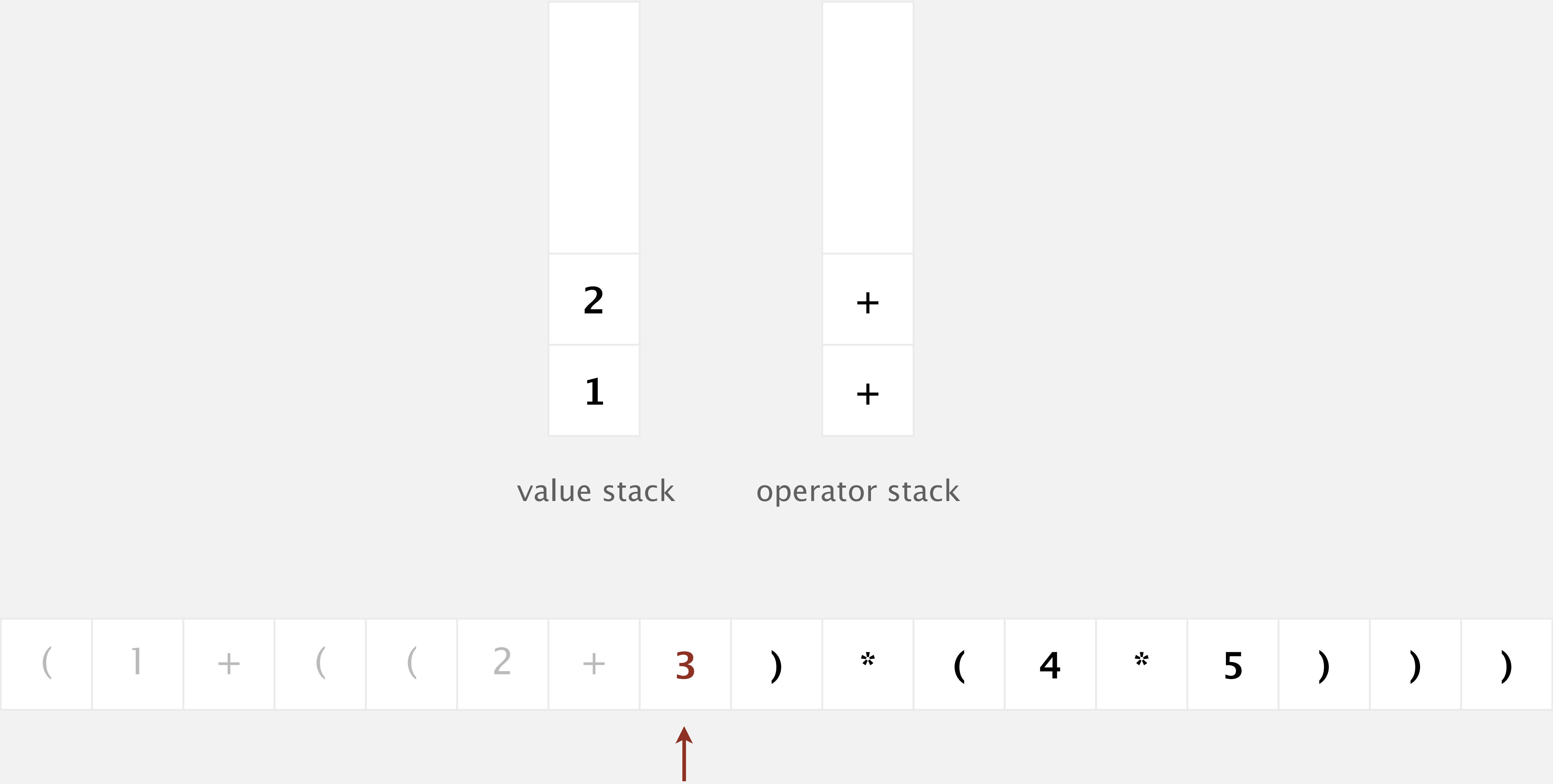
Dijkstra's two-stack algorithm

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
2	+
1	+

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

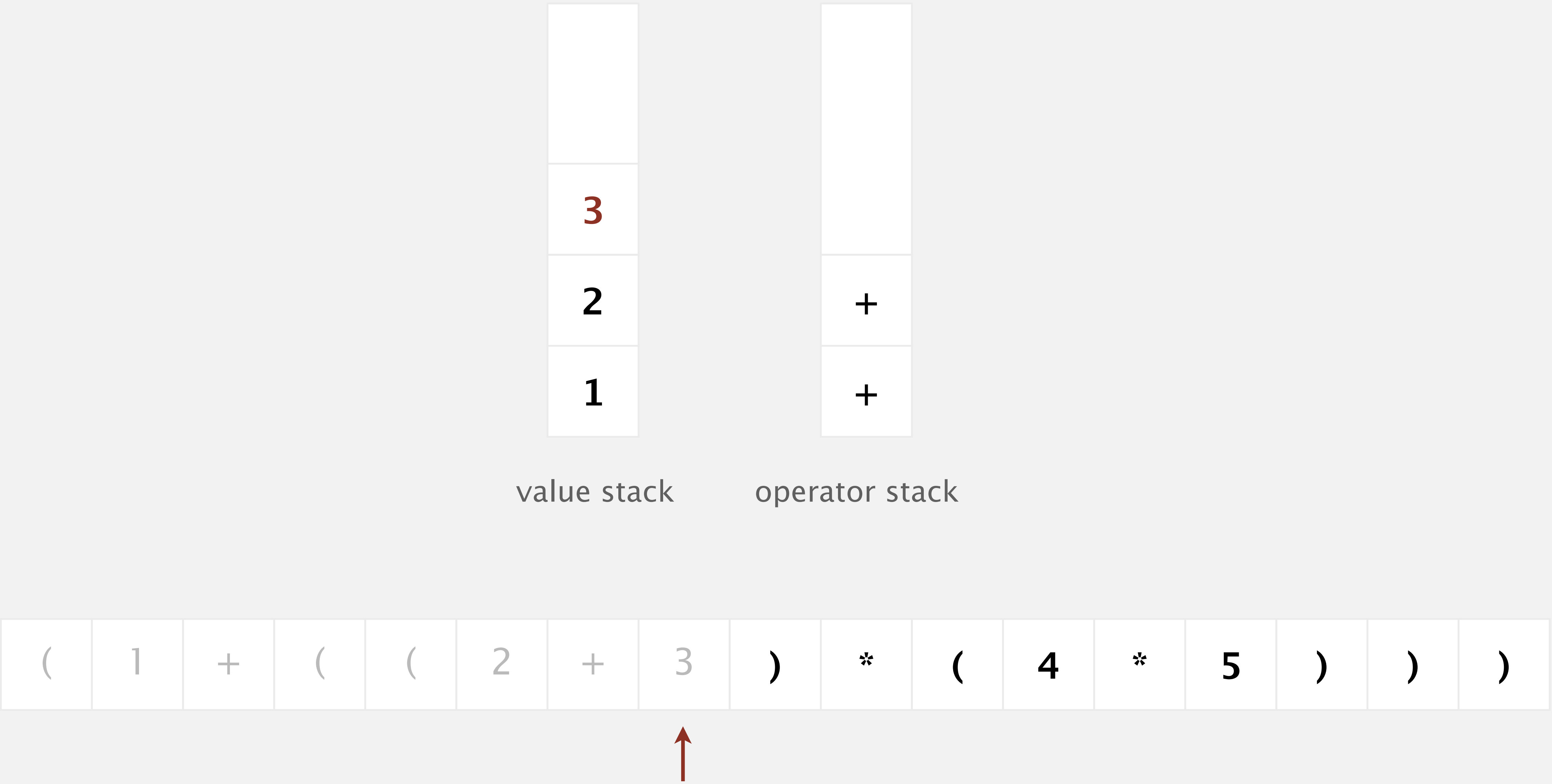
Dijkstra's two-stack algorithm

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.



3
2
1

value stack

+
+

operator stack

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

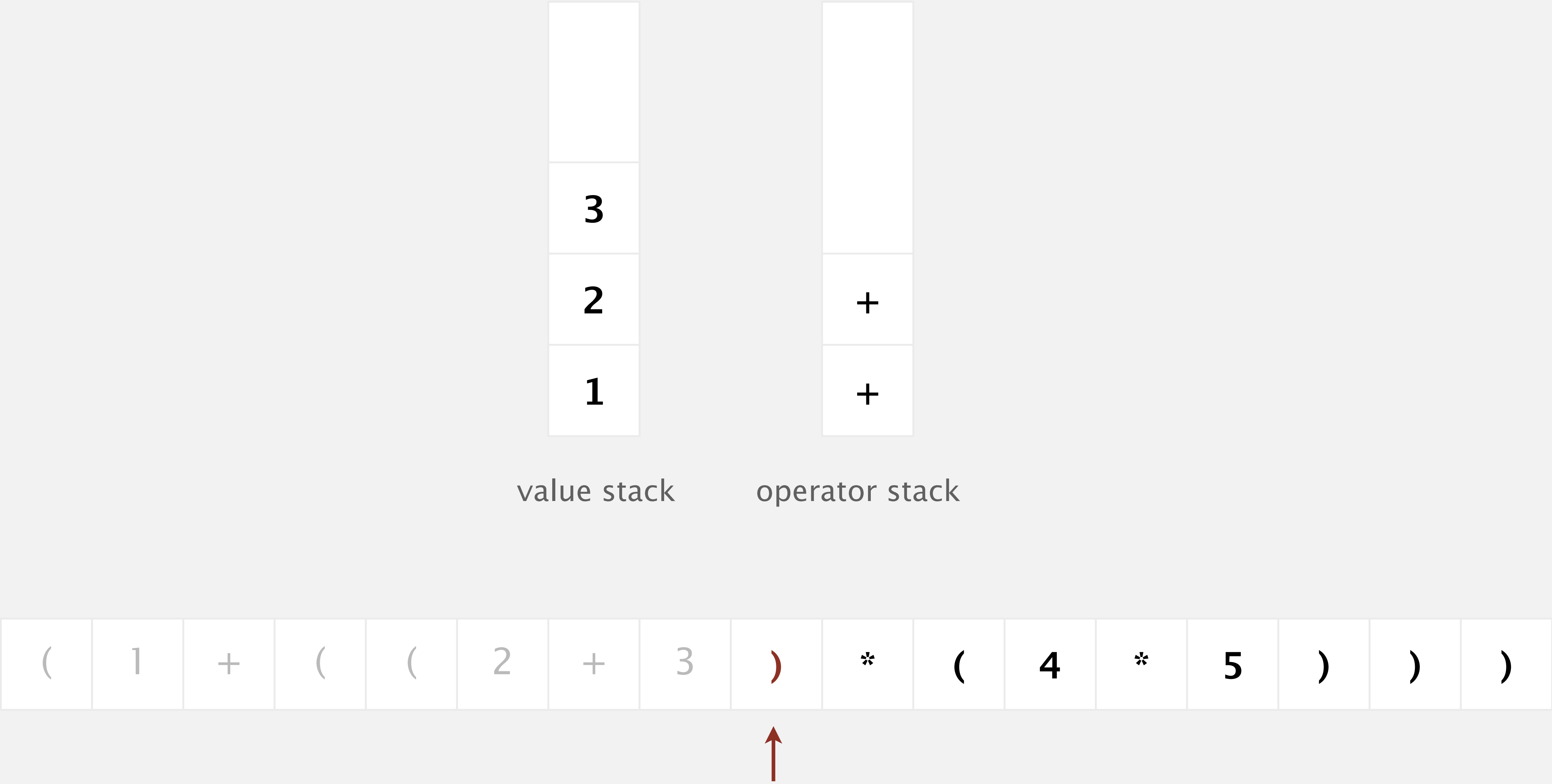
Dijkstra's two-stack algorithm

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.



3
2
1

value stack

+
+

operator stack

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

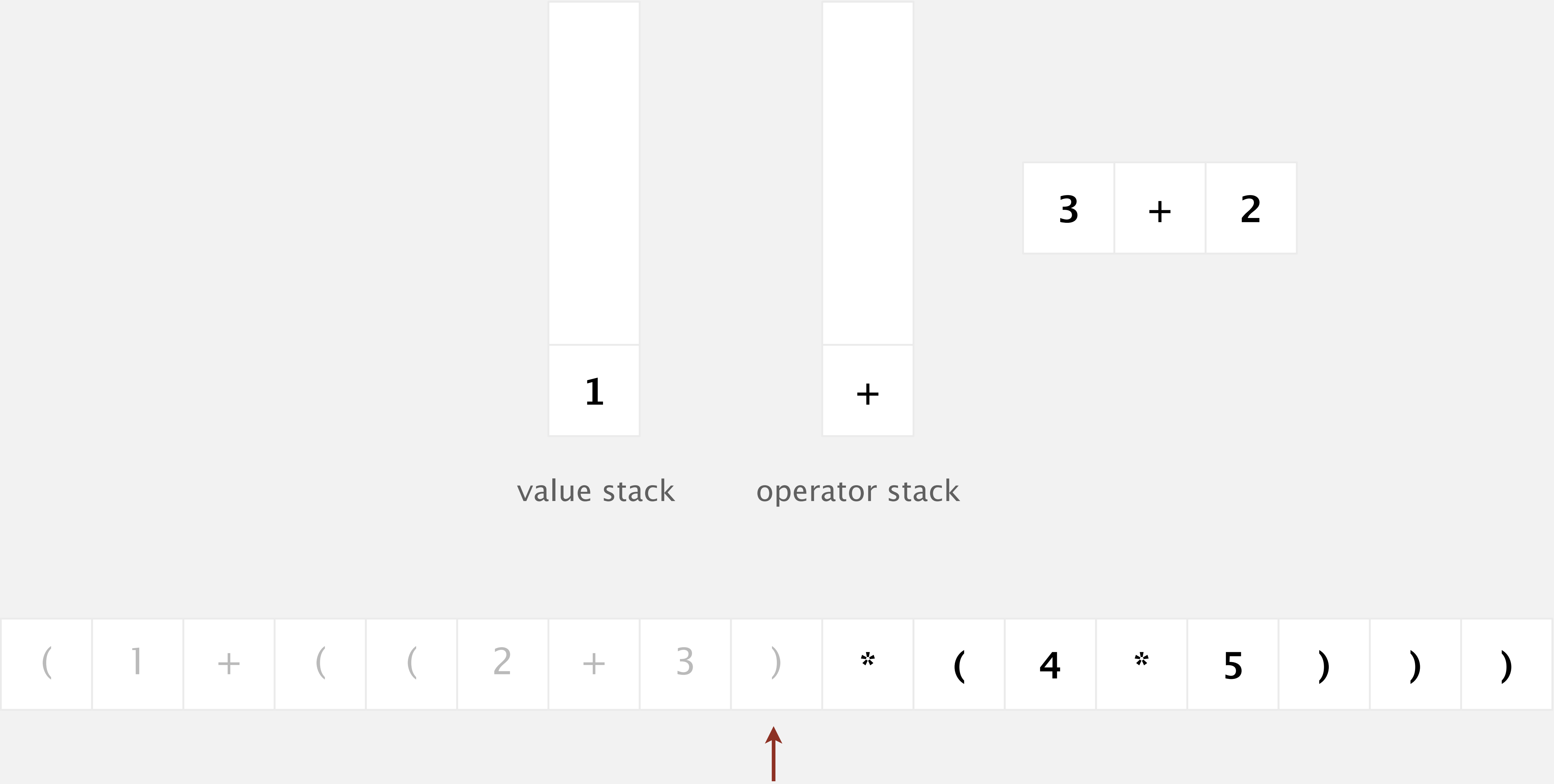
Dijkstra's two-stack algorithm

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.



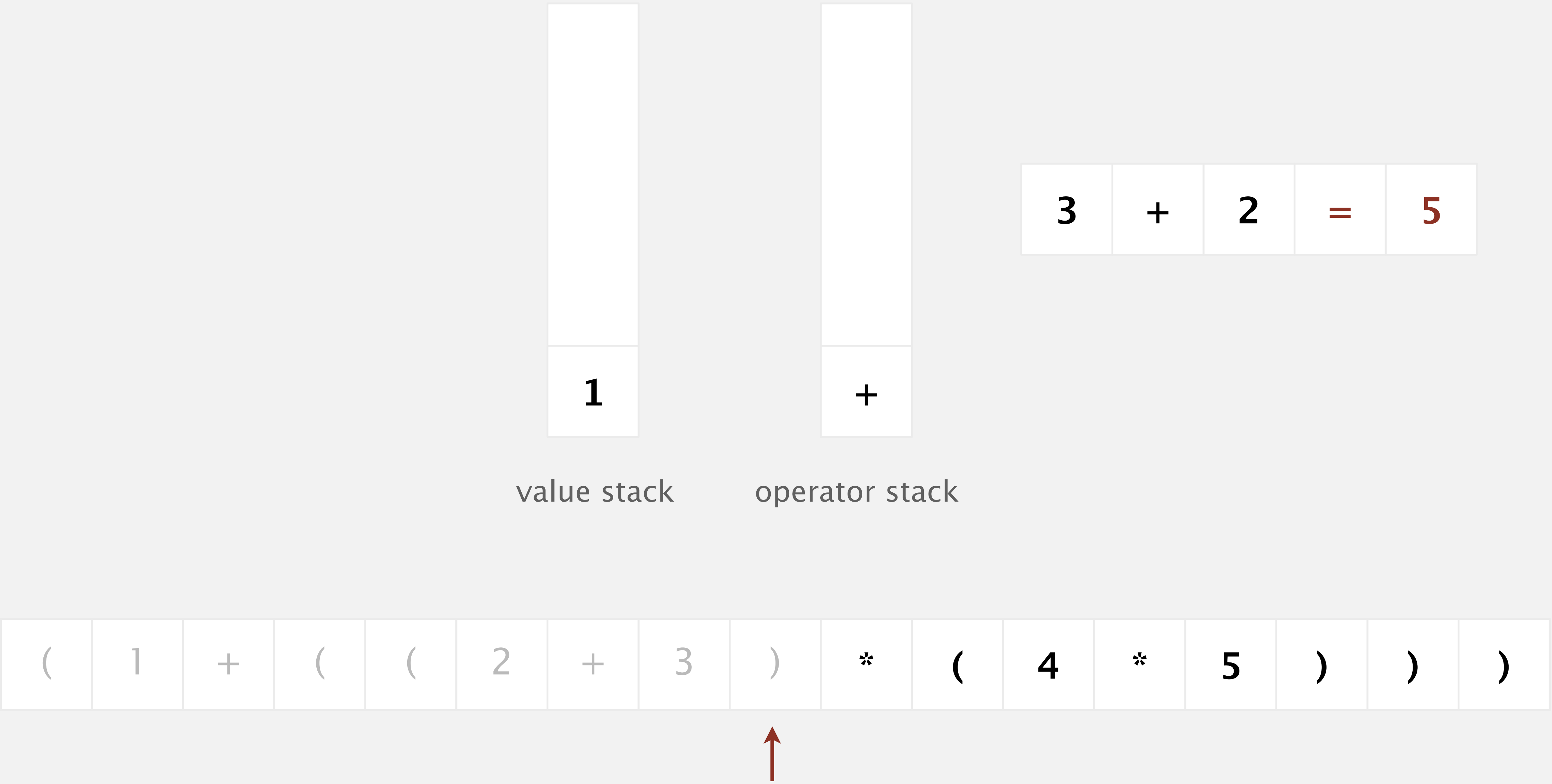
Dijkstra's two-stack algorithm

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.



3 + 2 = 5

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

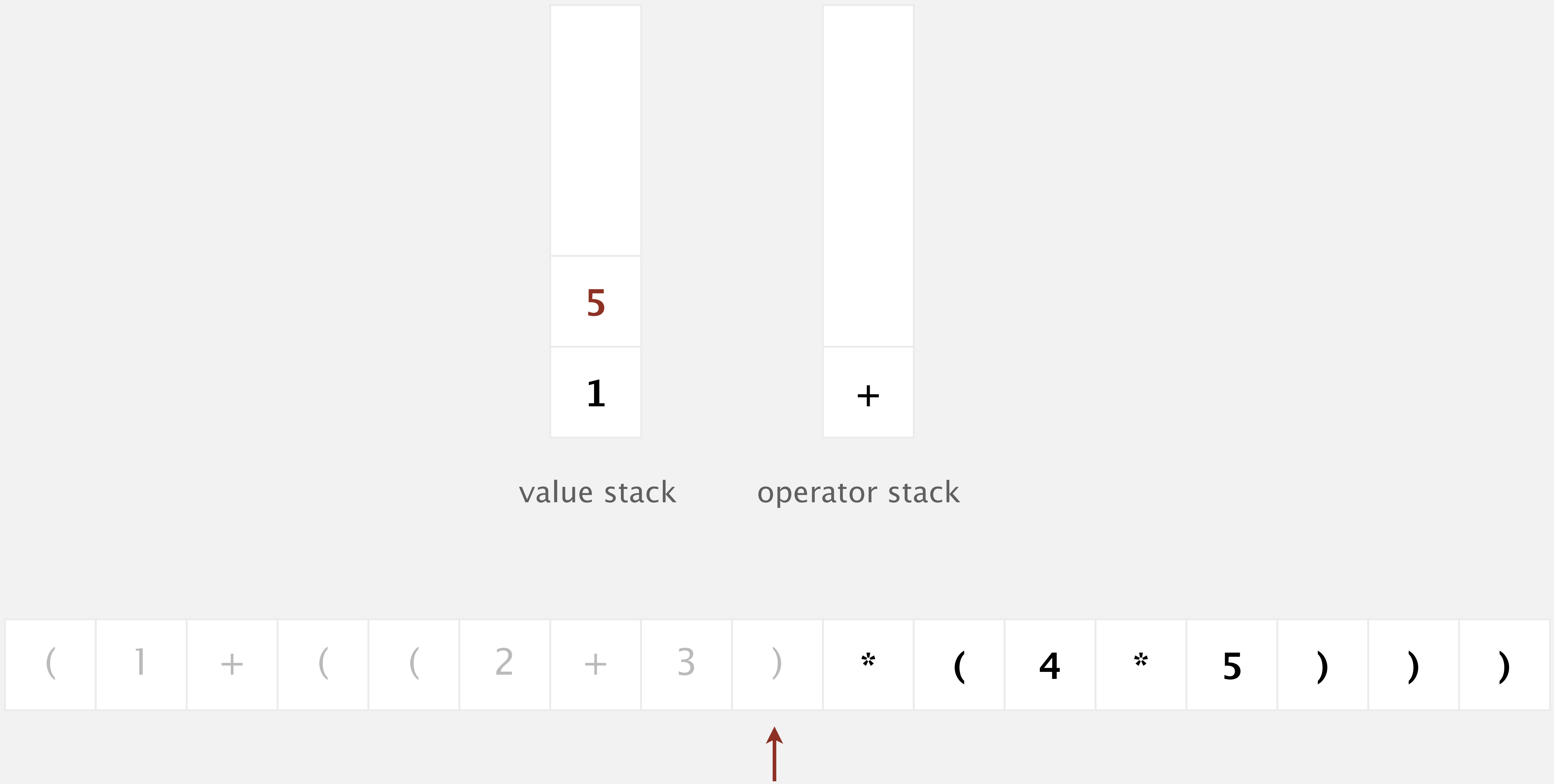
Dijkstra's two-stack algorithm

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	)	)	)

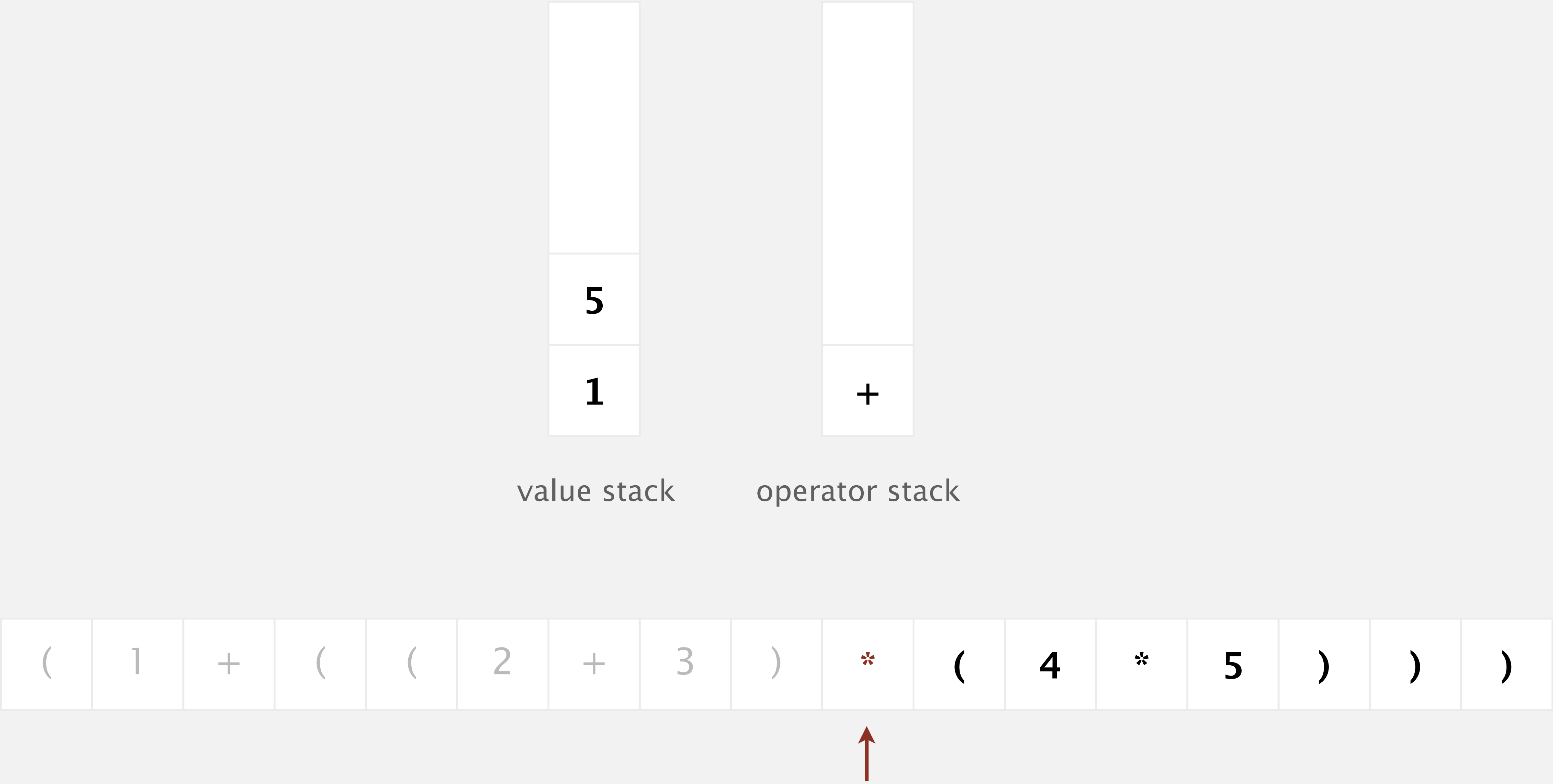
Dijkstra's two-stack algorithm

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.



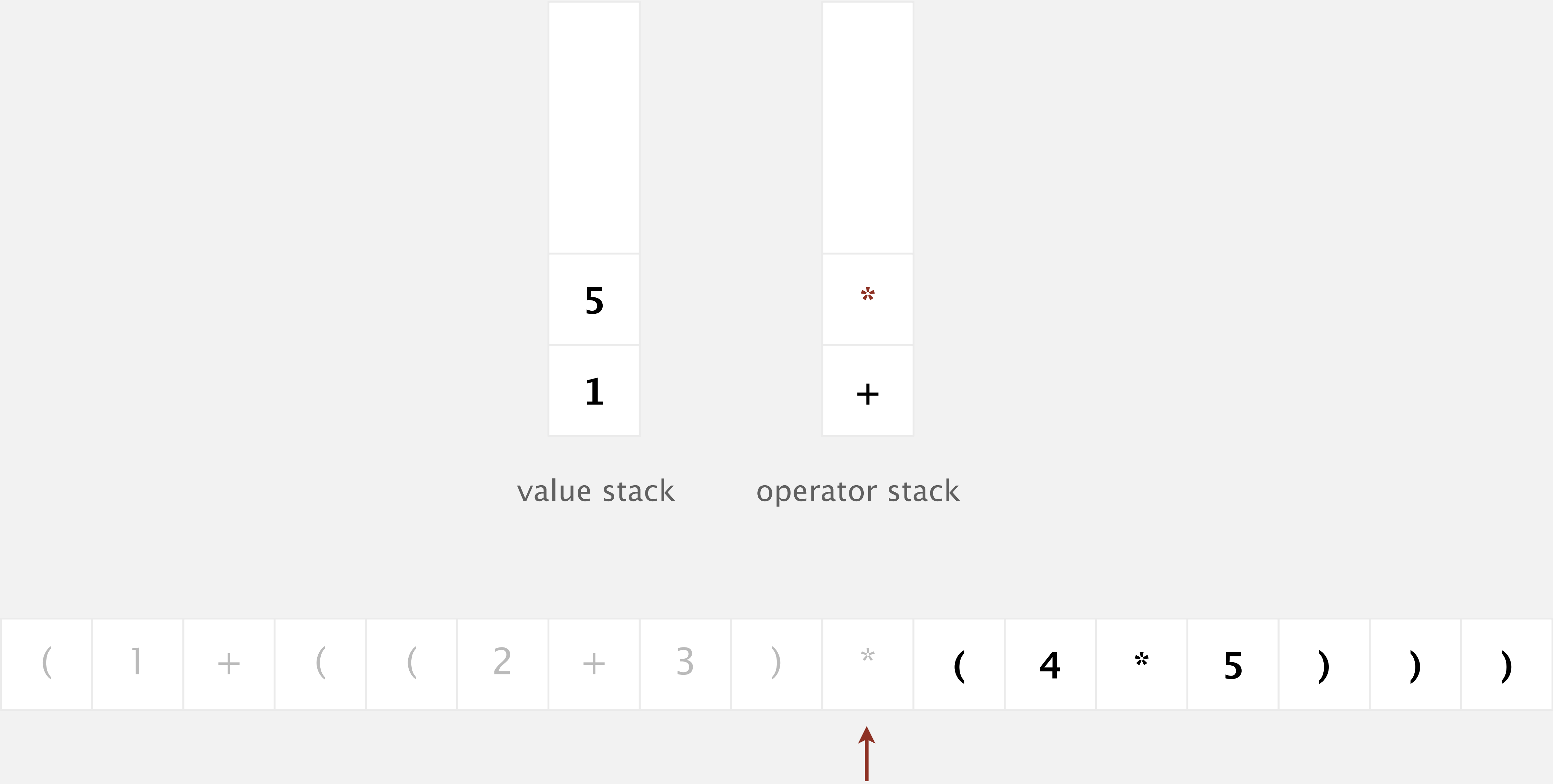
Dijkstra's two-stack algorithm

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.



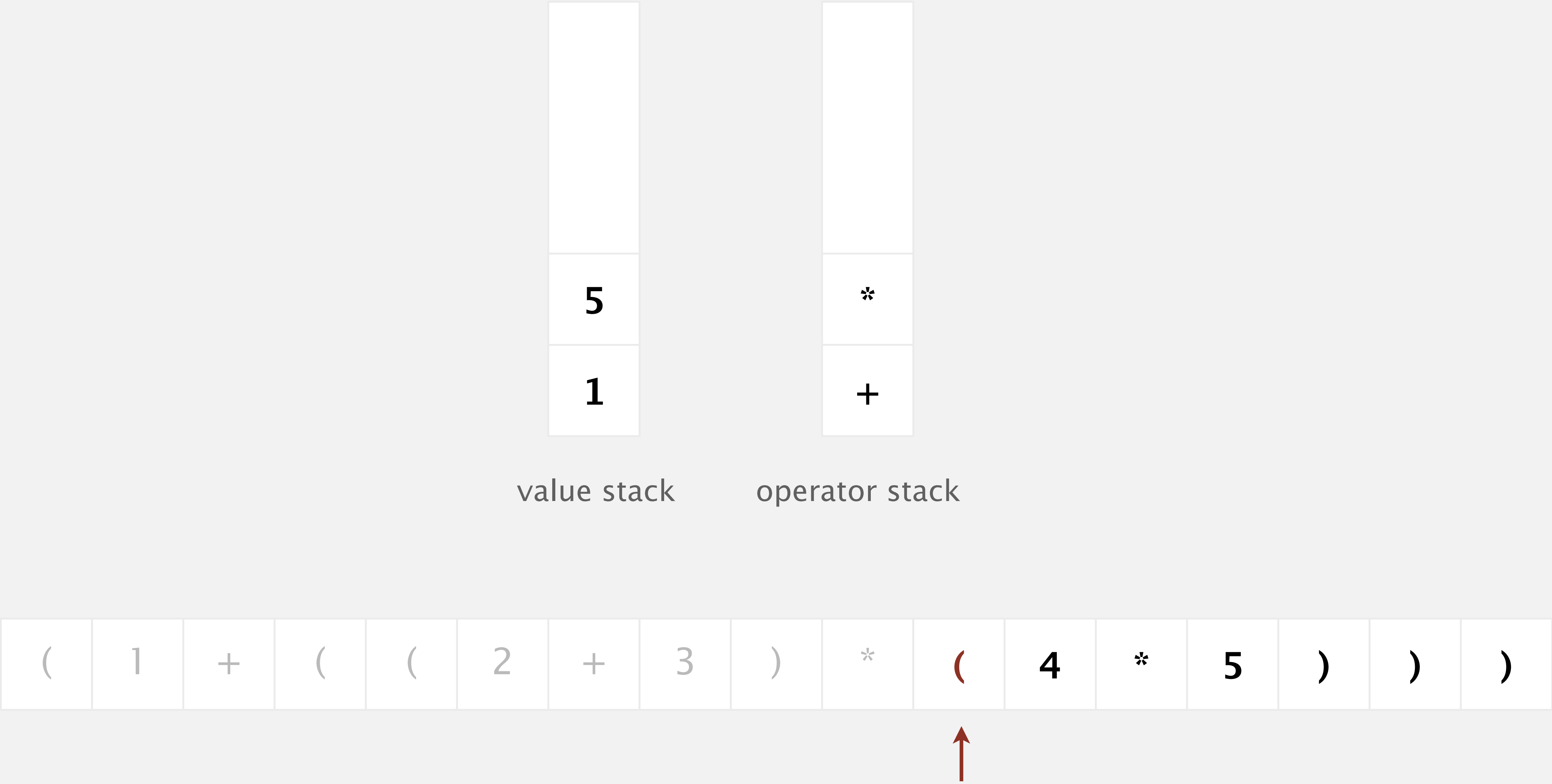
Dijkstra's two-stack algorithm

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.



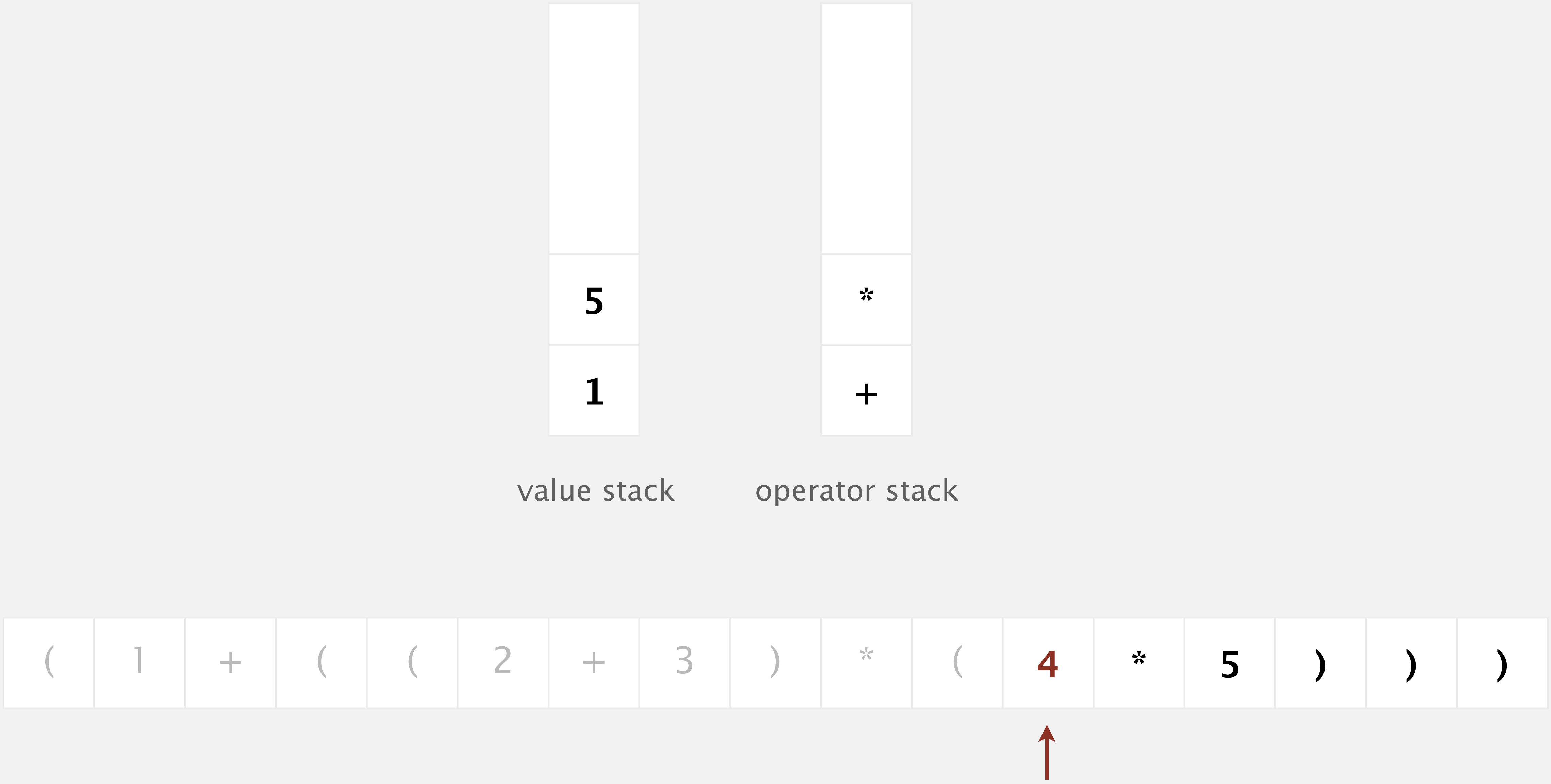
Dijkstra's two-stack algorithm

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.



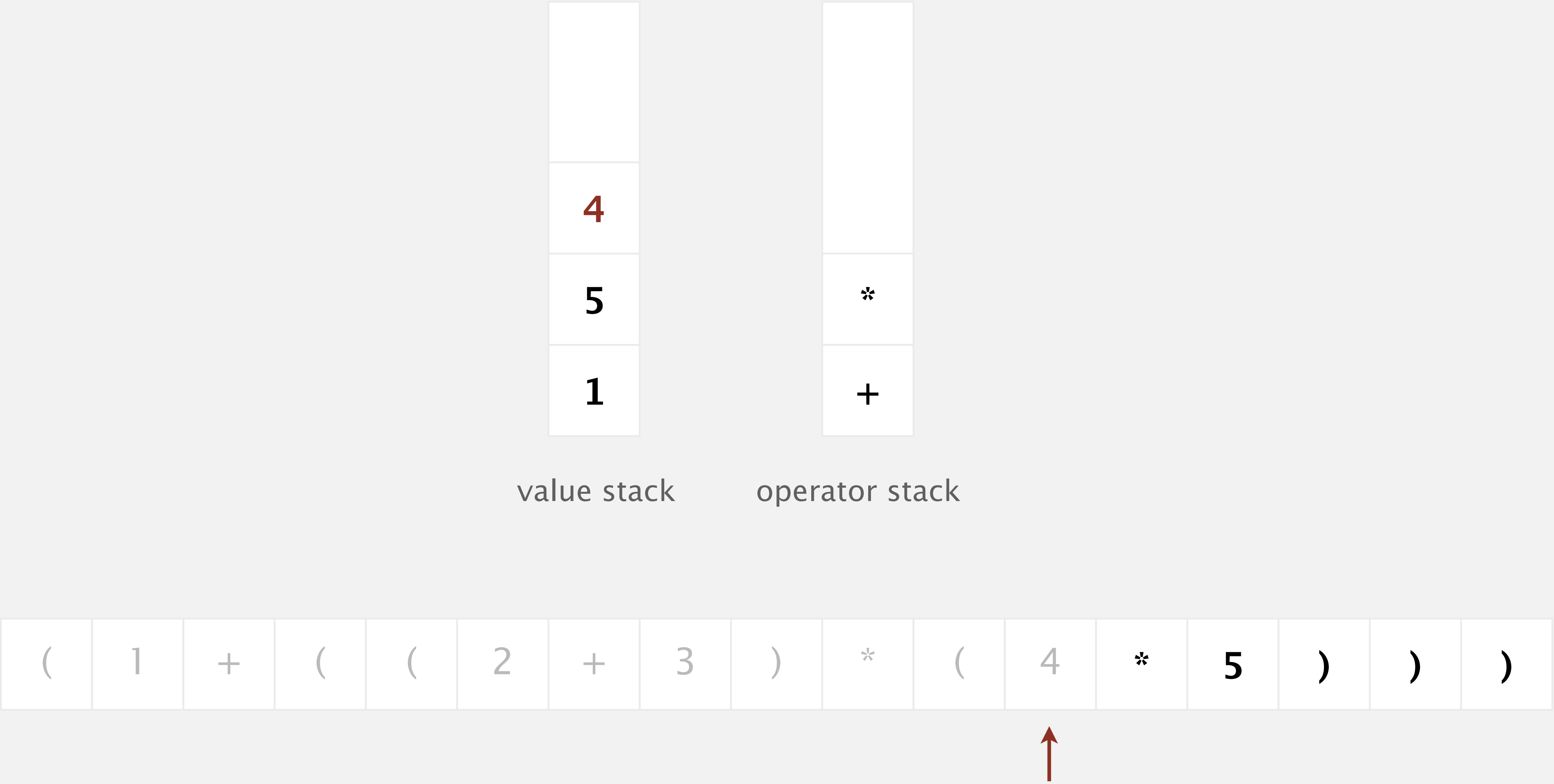
Dijkstra's two-stack algorithm

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.



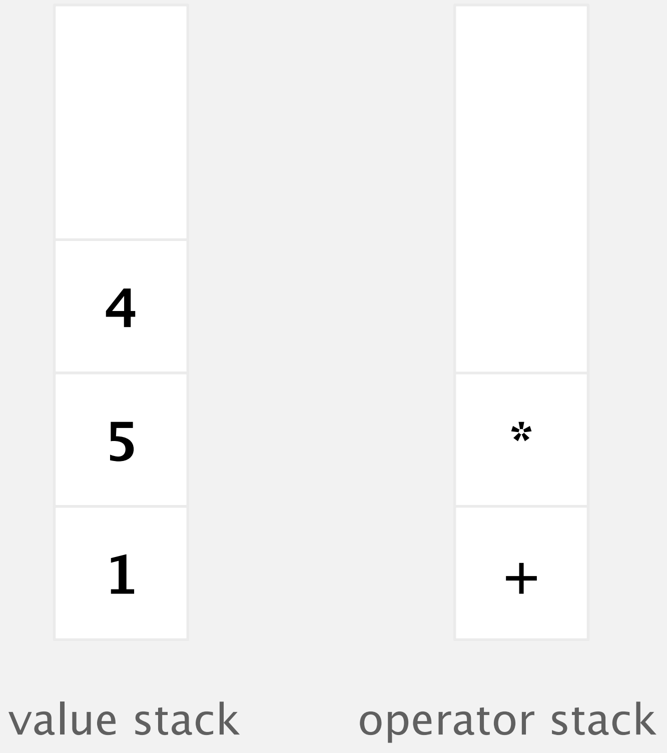
Dijkstra's two-stack algorithm

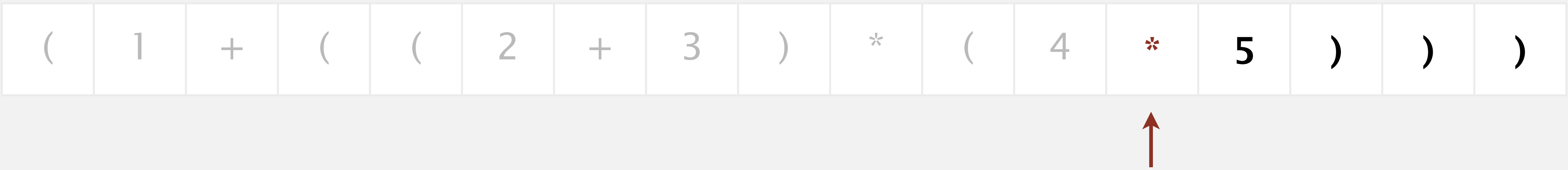
Value: push onto the value stack.

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.





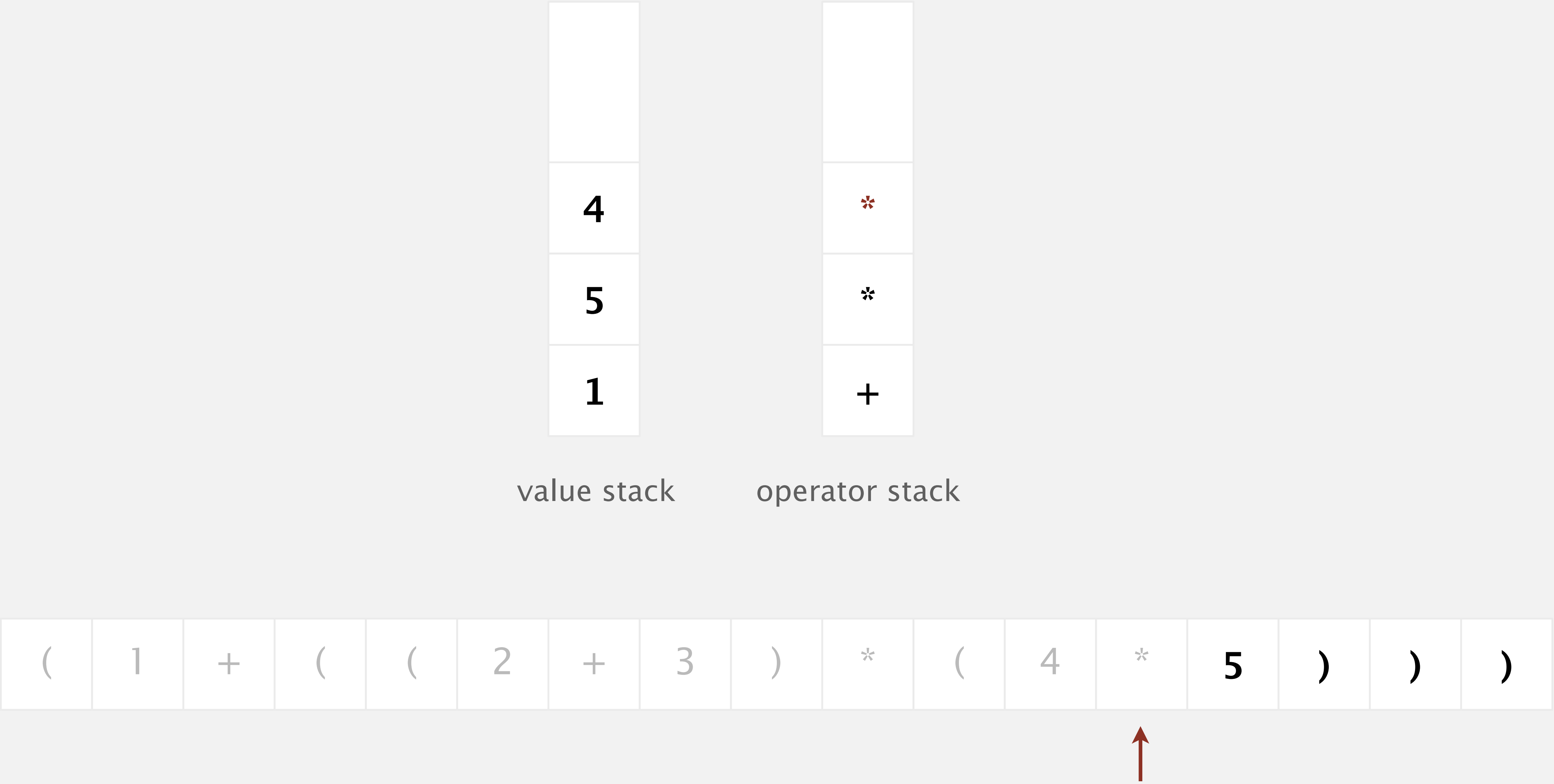
Dijkstra's two-stack algorithm

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.



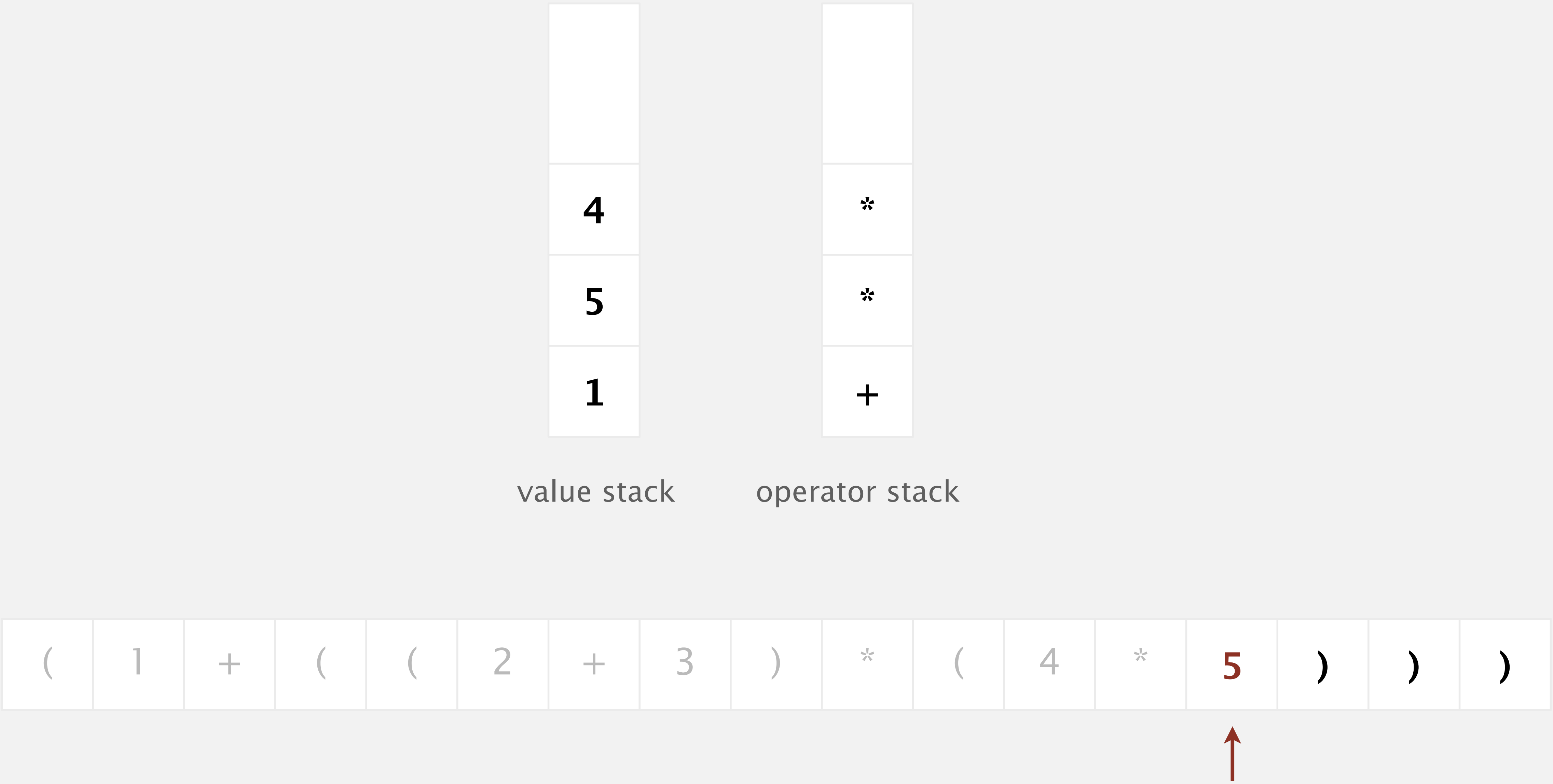
Dijkstra's two-stack algorithm

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.



value stack	operator stack
4	*
5	*
1	+

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

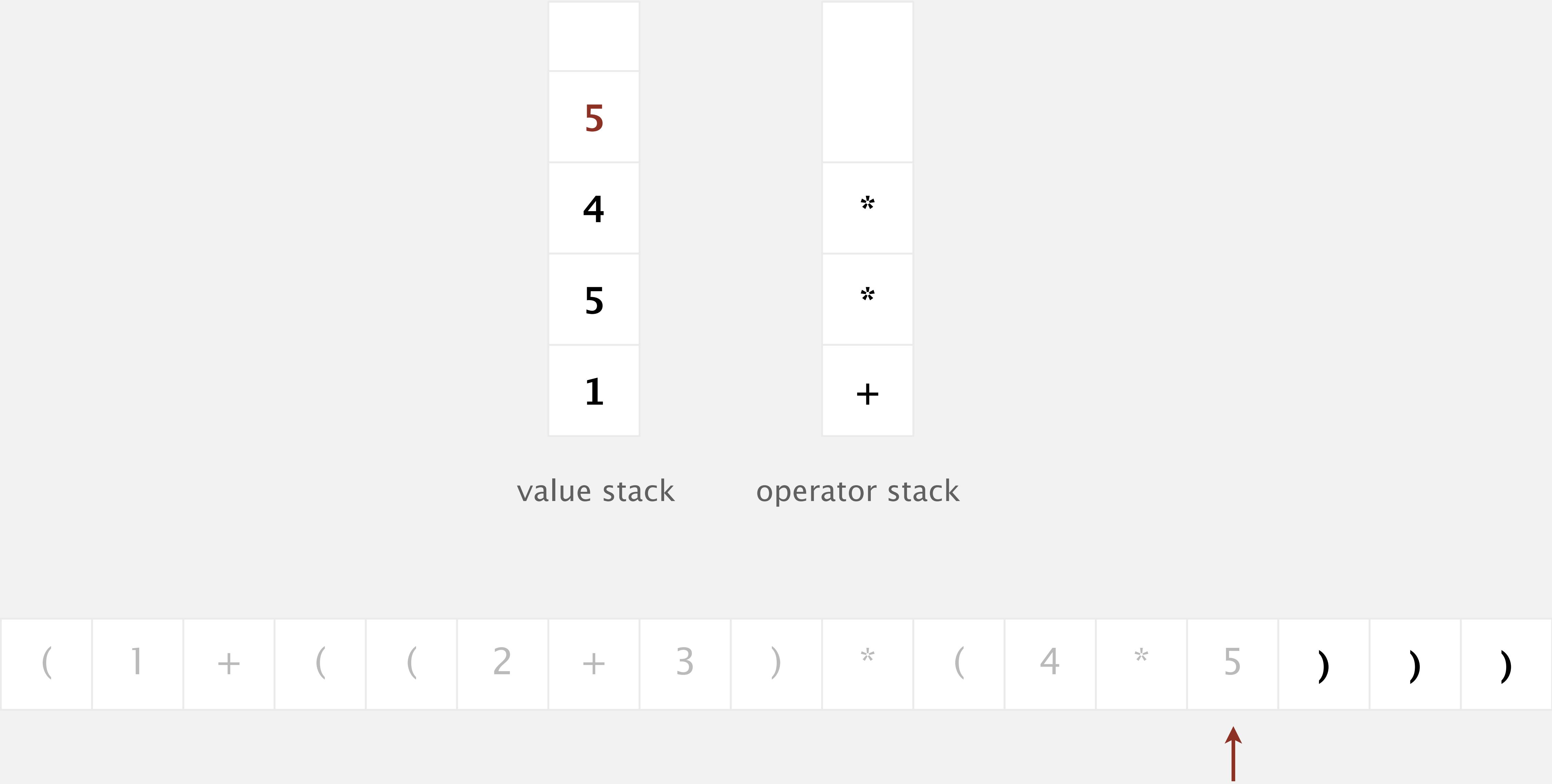
Dijkstra's two-stack algorithm

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.



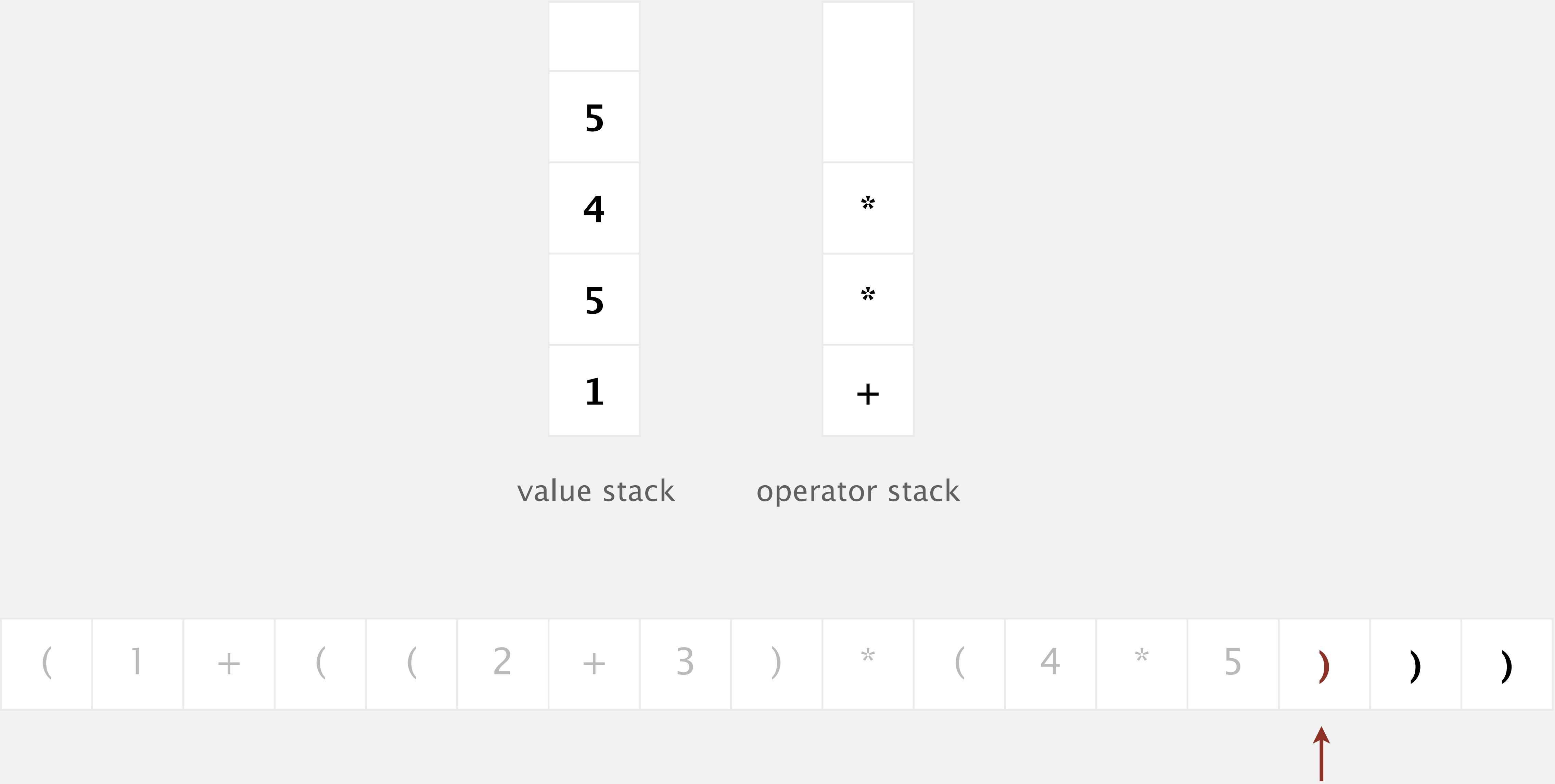
Dijkstra's two-stack algorithm

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.



5
4
5
1

value stack

*
*
+

operator stack

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

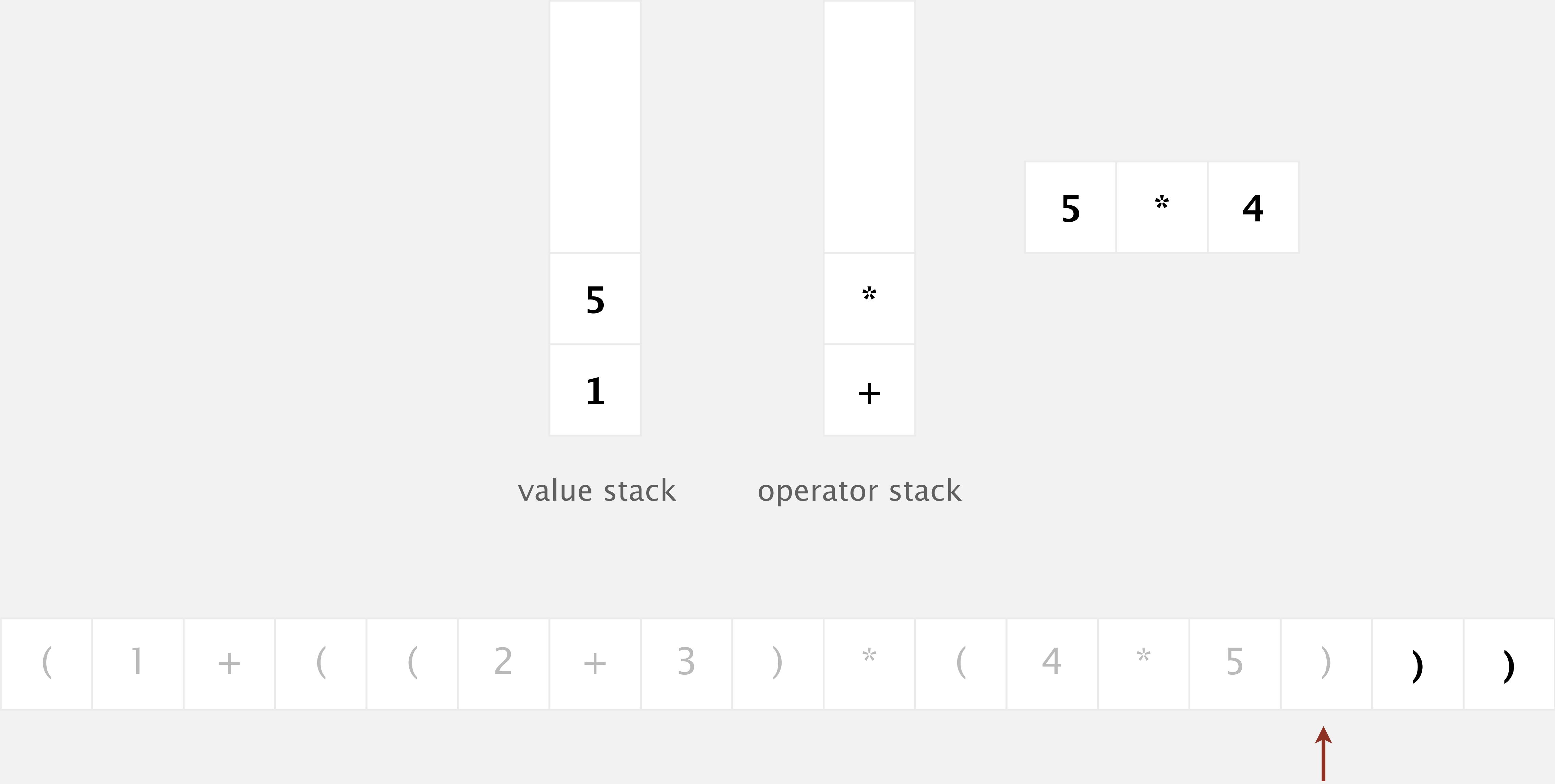
Dijkstra's two-stack algorithm

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.



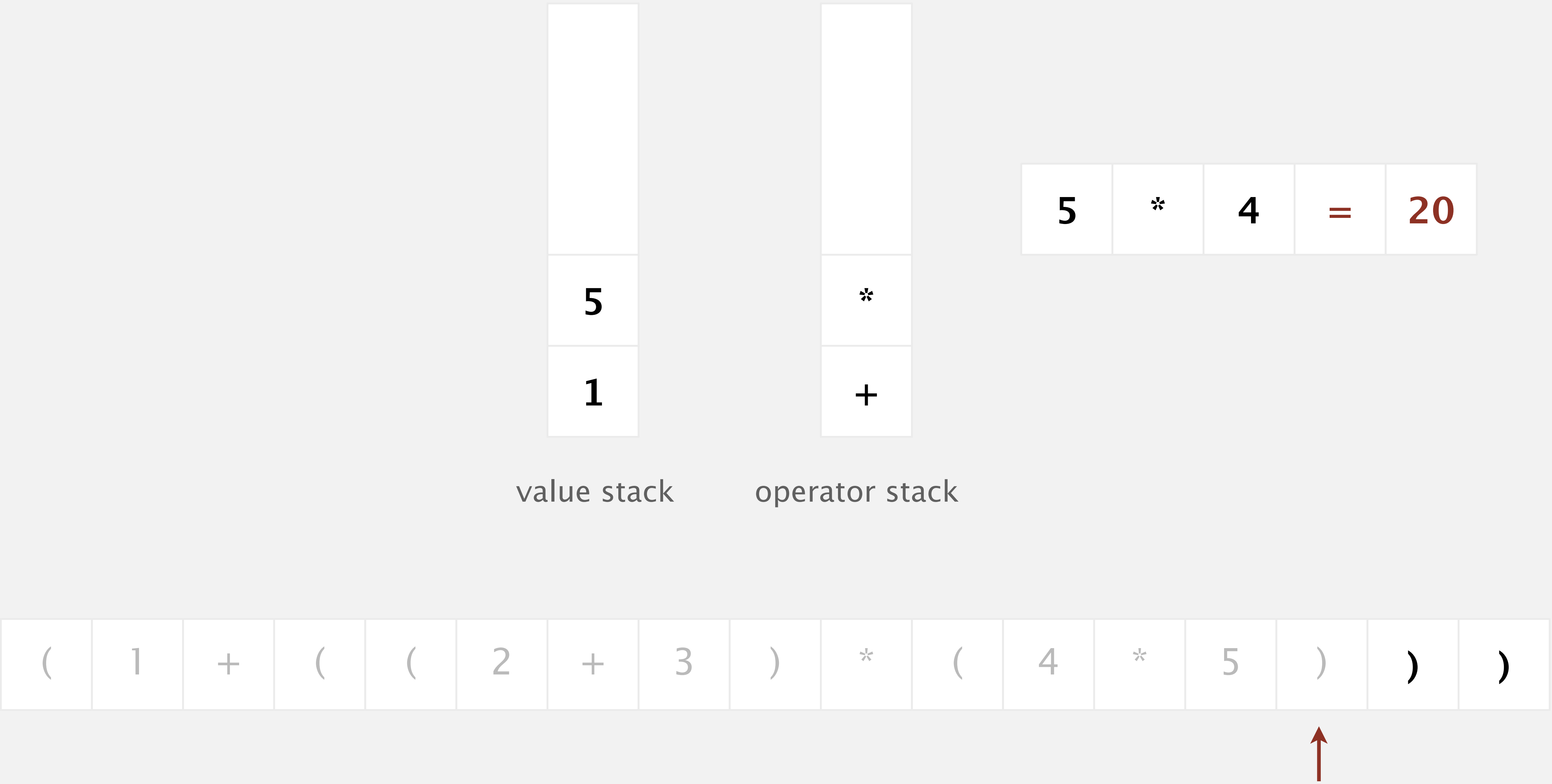
Dijkstra's two-stack algorithm

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.



5

1

value stack

*

+

operator stack

5 * 4 = 20

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

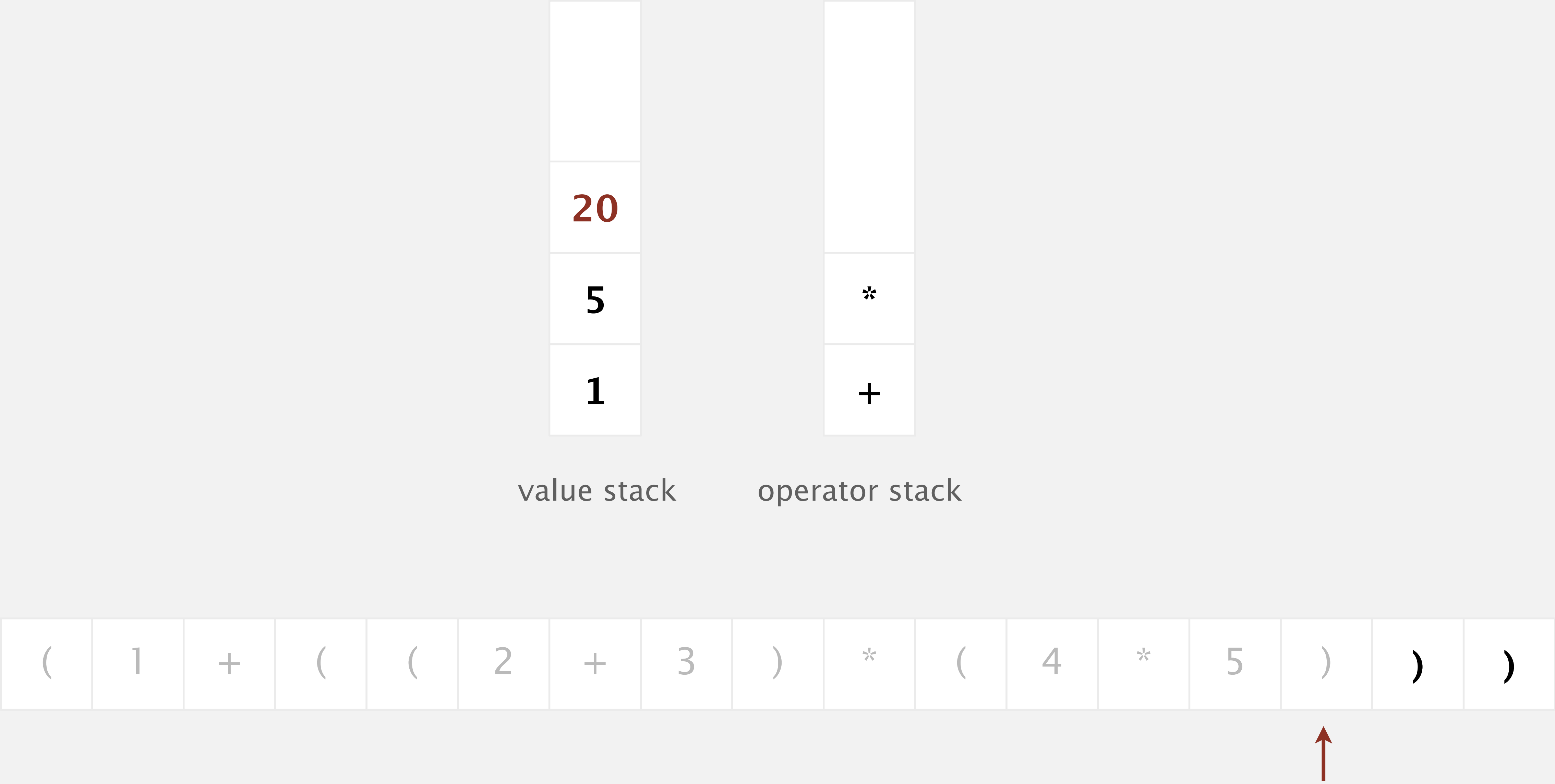
Dijkstra's two-stack algorithm

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.



20
5
1

value stack

*
+

operator stack

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

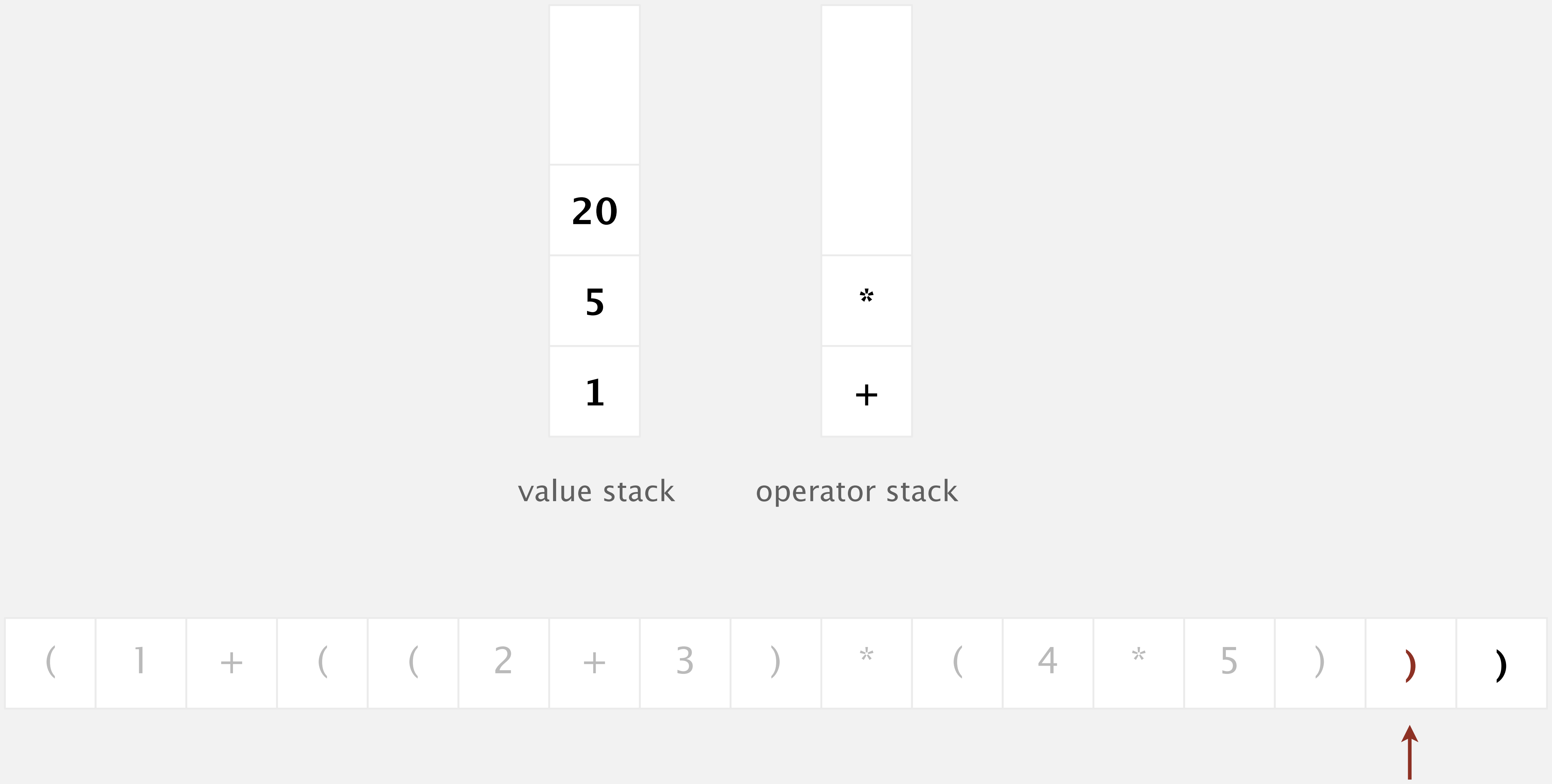
Dijkstra's two-stack algorithm

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.



20

5

1

value stack

*

+

operator stack

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

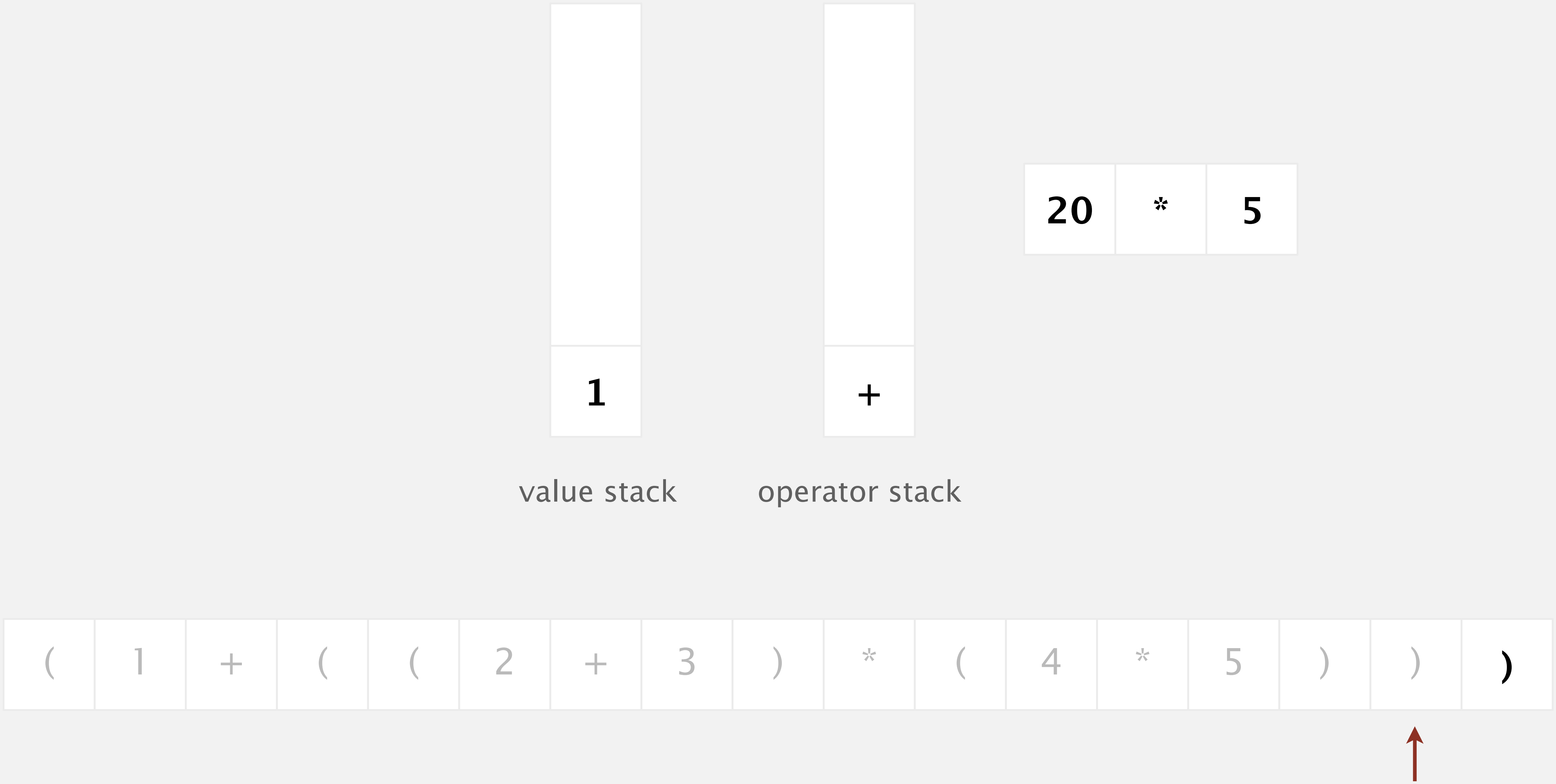
Dijkstra's two-stack algorithm

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.



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

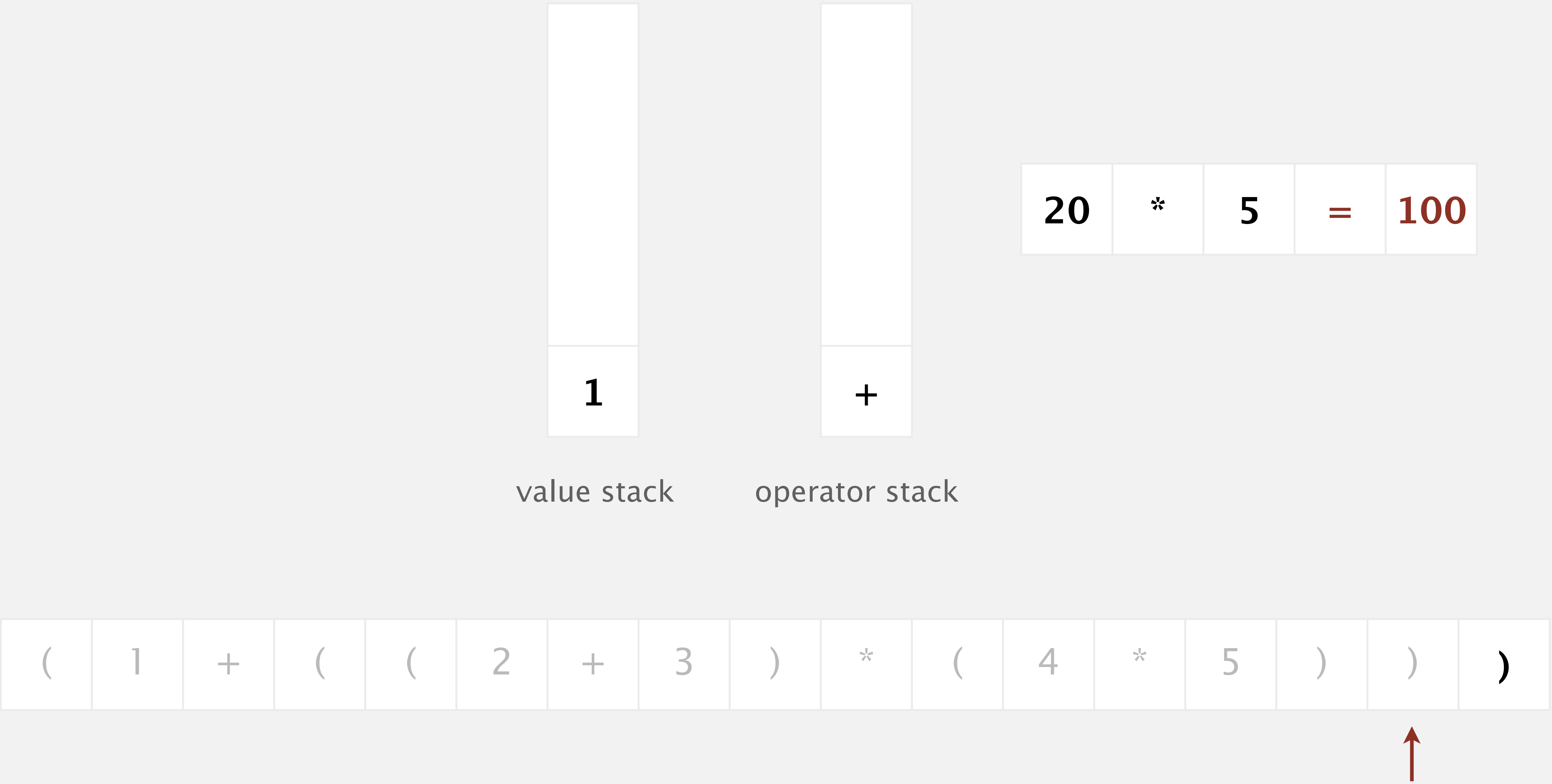
Dijkstra's two-stack algorithm

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.



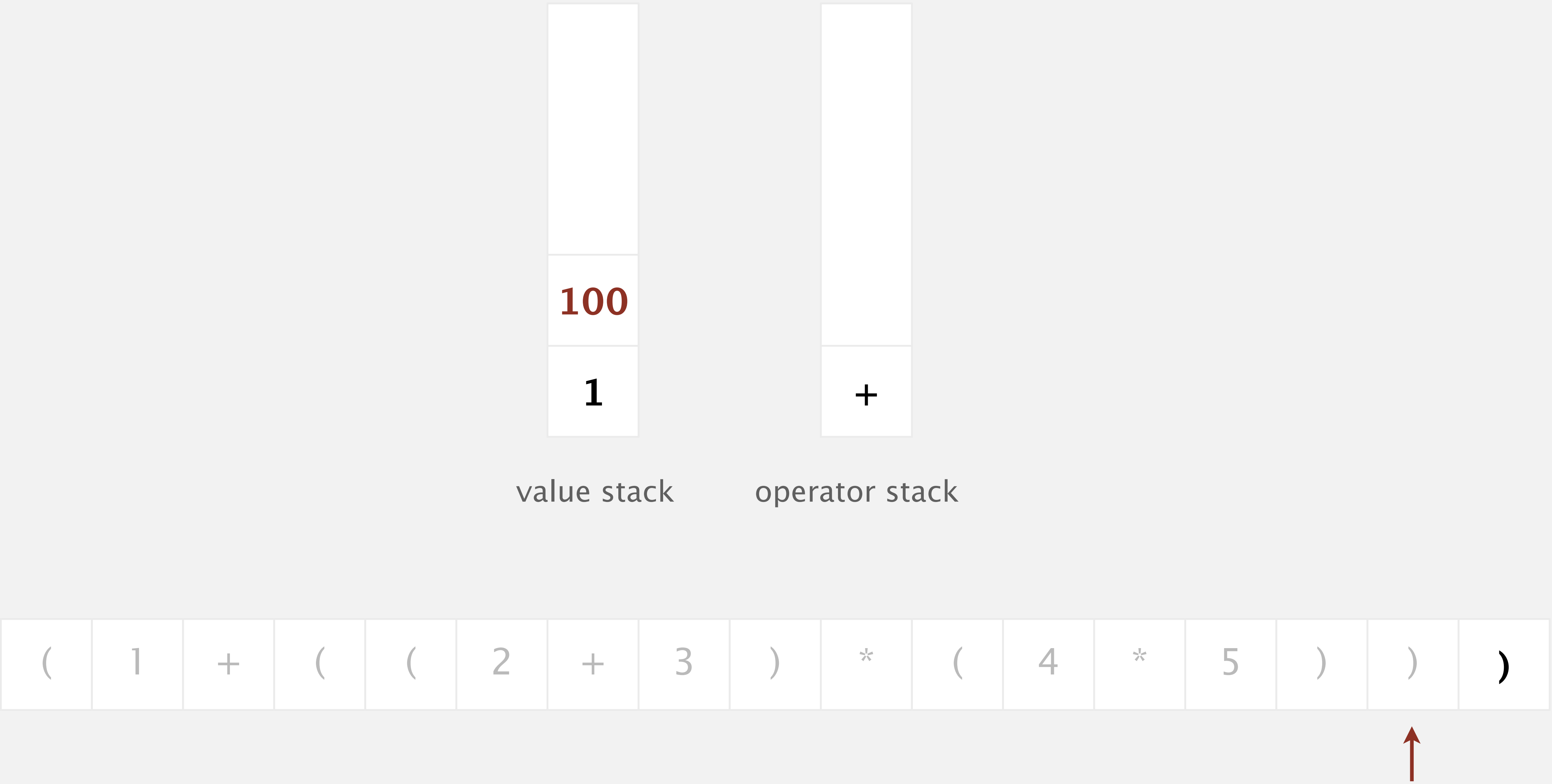
Dijkstra's two-stack algorithm

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.



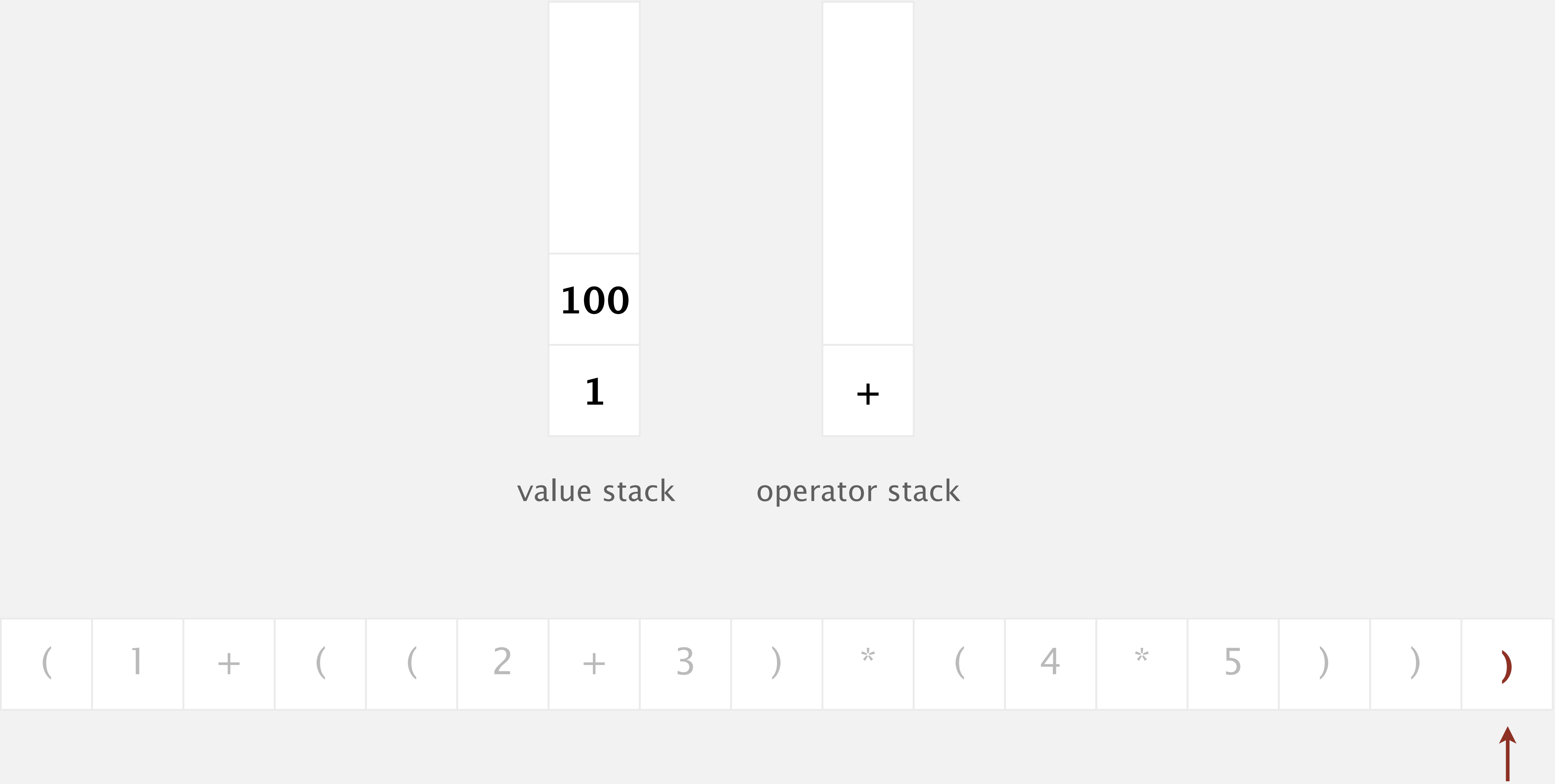
Dijkstra's two-stack algorithm

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.



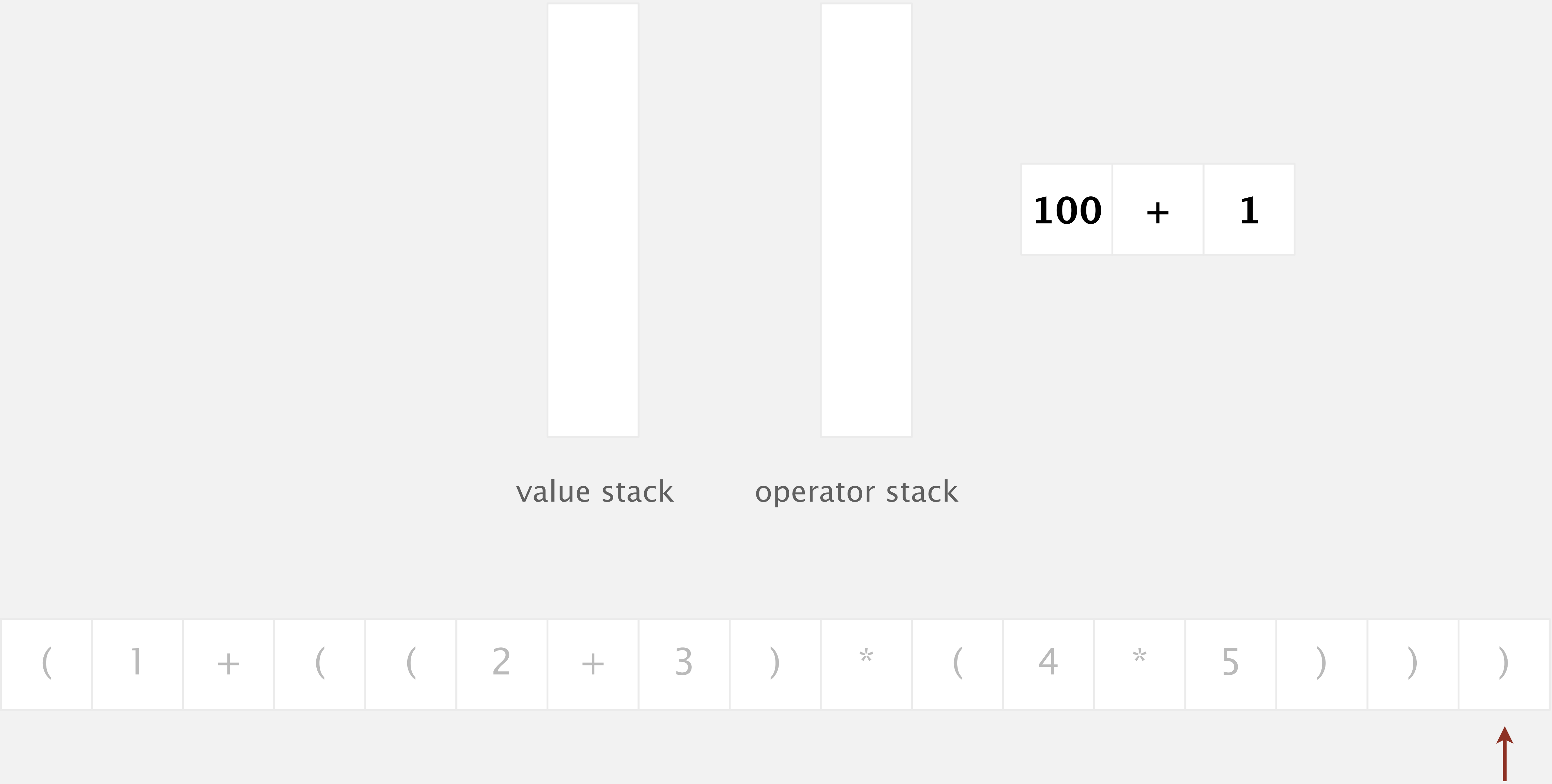
Dijkstra's two-stack algorithm

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.



100 + 1

value stack

operator stack

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

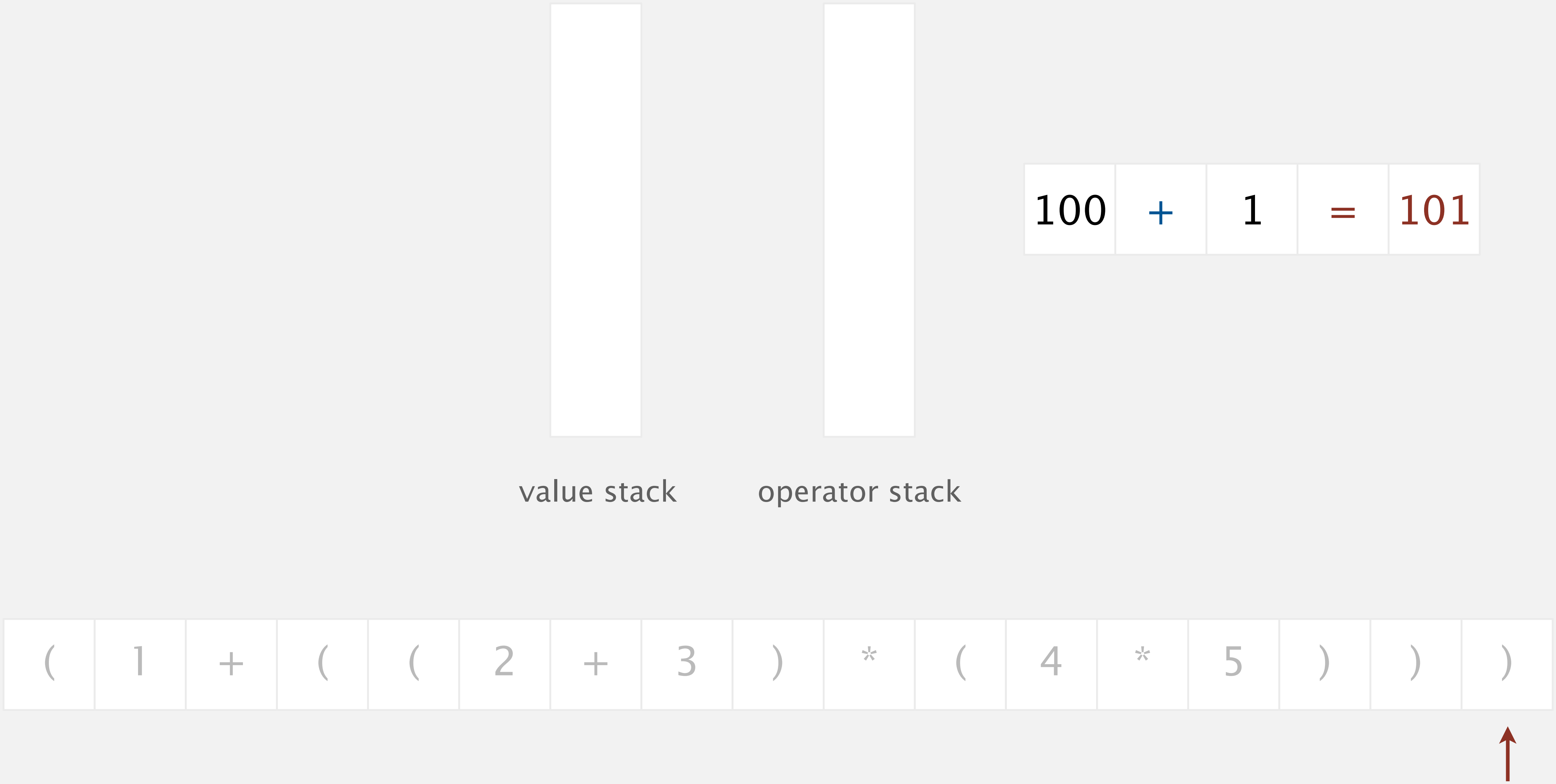
Dijkstra's two-stack algorithm

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.



100	+	1	=	101

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

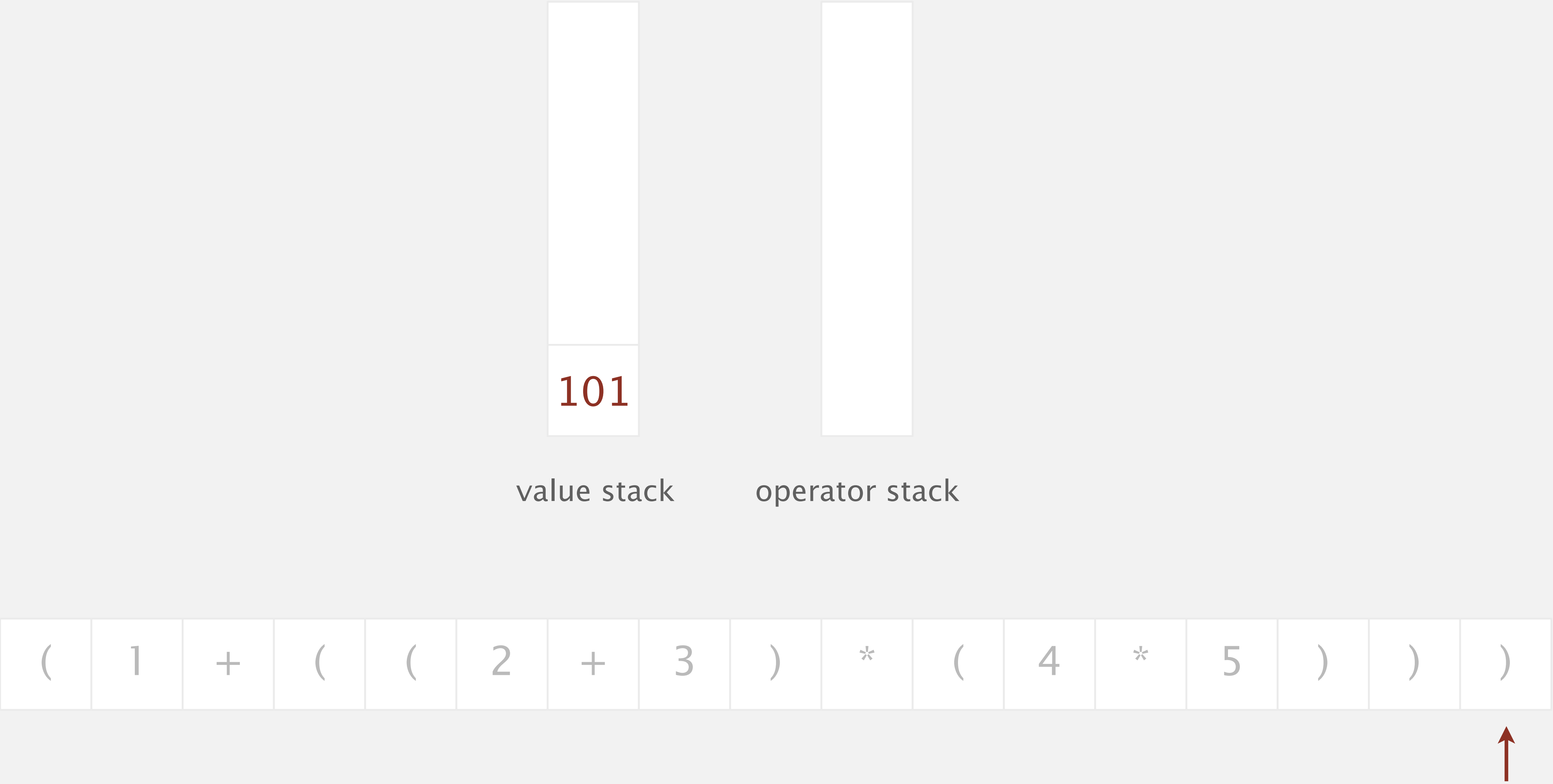
Dijkstra's two-stack algorithm

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.



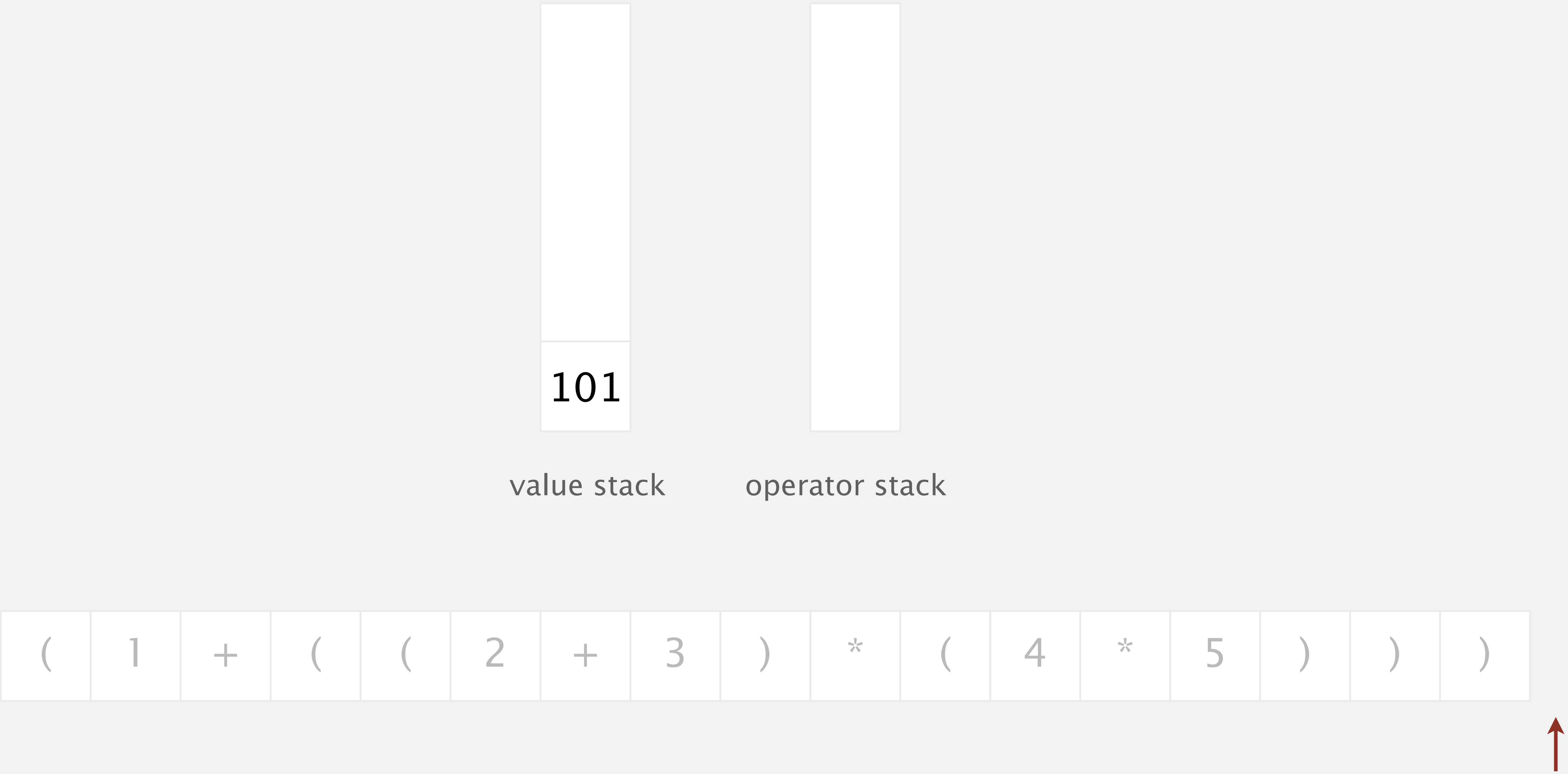
Dijkstra's two-stack algorithm

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.



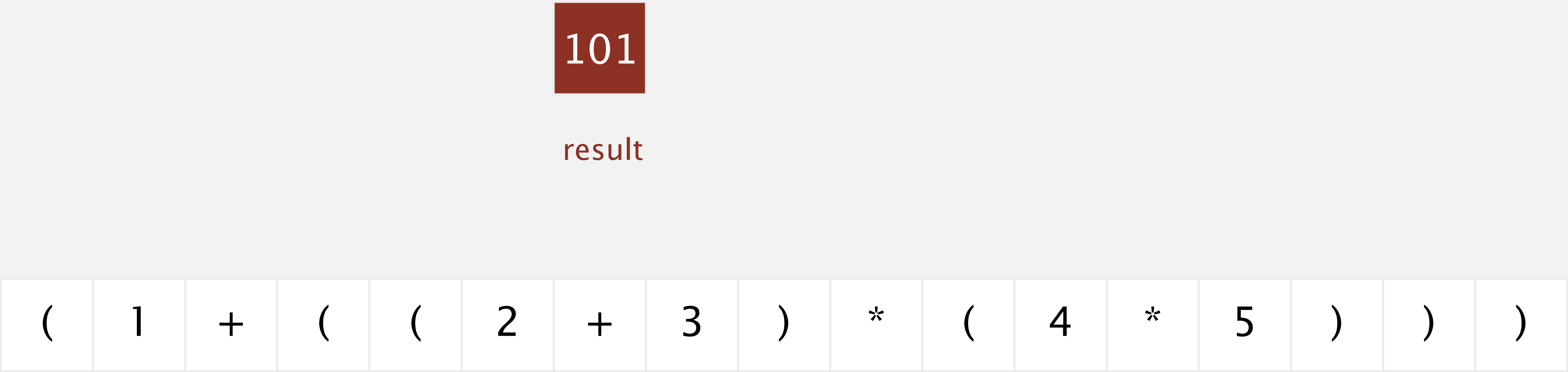
Dijkstra's two-stack algorithm

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	)	)	)

Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {

         }
         else
      }
      StdOut.println(vals.pop());
   }
}
```

What to write here? 4 mins.
Hint: use push(value) and pop() methods.

And here.

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

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...

Tips for Writing Algorithm Assignments

Understand the question.

- Input, expected output, performance requirement.

Understand the given code.

- Classes, variables, and methods.

Brainstorm solution ideas.

- Use pen and paper when necessary.

Breakdown solution into multiple steps.

- Make sure you know the expected output for each step.

Write code for one step at a time, verify output before proceeding.

- Use Java debugger/print to check variable values and expected output.
- Write pseudo code on paper first if necessary.

Paired programming

Method

- Find a partner in class that day, register your team.
- Take turns in paired programming.
- One student is responsible for uploading the assignment.
- Both students receive the same grade.

Students will take turns in two roles

- *Driver*: on the keyboard typing code.
  talk about what is being done at the moment.
- *Navigator*: observe, direct, and criticize code.
  review code on-the-go, gives directions, and share thoughts.

https://martinfowler.com/articles/on-pair-programming.html

In-class Assignment

Write the missing methods in the UnionFind2023.java file posted in the online assignment.

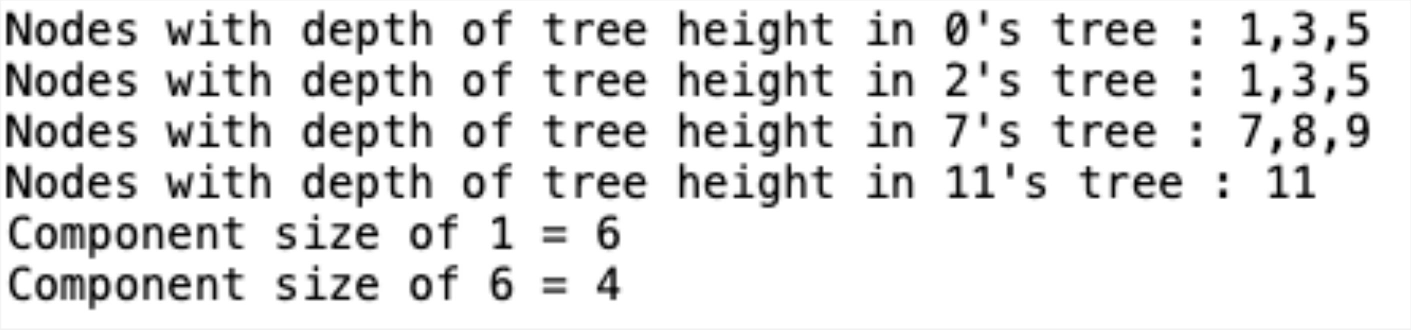
Mandatory

- public int[] nodesWithDepthOfTreeHeight(int p)
  - complete the method below to return a list of nodes in the same tree as node p, that have the depth of the tree's height.

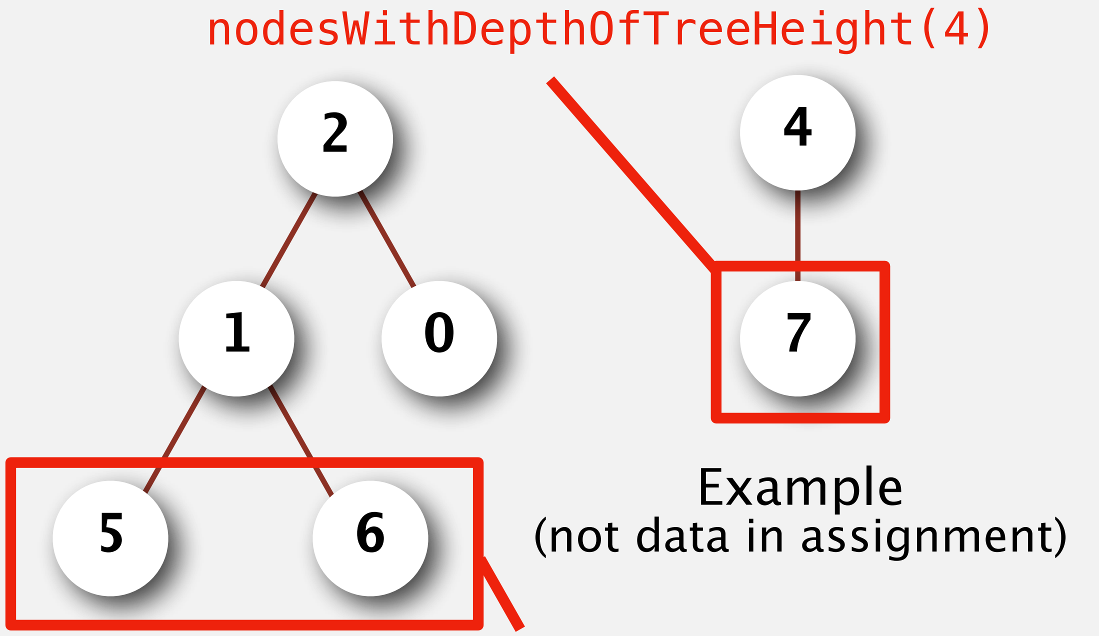
Optional

- public int componentSize(int p)
  - complete the method to return the size of the connected component with item p.

The expected output is as follows:



```
Nodes with depth of tree height in 0's tree : 1,3,5
Nodes with depth of tree height in 2's tree : 1,3,5
Nodes with depth of tree height in 7's tree : 7,8,9
Nodes with depth of tree height in 11's tree : 11
Component size of 1 = 6
Component size of 6 = 4
```



Example
(not data in assignment)

DO NOT EDIT other functions NOR add global variables.

In-class Assignment

Deliverables

- Register your team with the TA.
- Include both your names at the top of the .java file as a comment.
- Write comment for each key operation.
- Cite referenced websites and others that helped you in the comments.
- Upload your .java file to the course website by the end of the class.
  - Only one of you in your team should upload the .java file.
  - The other person needs to make sure that the upload is successful.