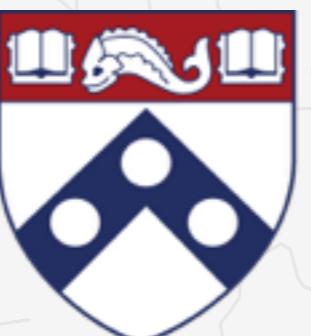


CIS 121, FALL 2016

DATA STRUCTURES AND ALGORITHMS

AKA PROGRAMMING LANGUAGES & TECHNIQUES II

CHRIS CALLISON-BURCH



Penn
UNIVERSITY of PENNSYLVANIA

CIS 121 course overview

What is CIS 121?

- Third course in the intro sequence CIS 120, 160, 121
- Programming and problem solving, with applications.
- **Algorithm:** method for solving a problem.
- **Data structure:** method to store information.

topic	data structures and algorithms
data types	stack, queue, bag, union-find, priority queue
sorting	quicksort, mergesort, heapsort, radix sorts
searching	BST, red-black BST, hash table
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	KMP, regular expressions, tries, data compression
advanced	B-tree, k-d tree, suffix array, maxflow

Why study algorithms?

Their impact is broad and far-reaching.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

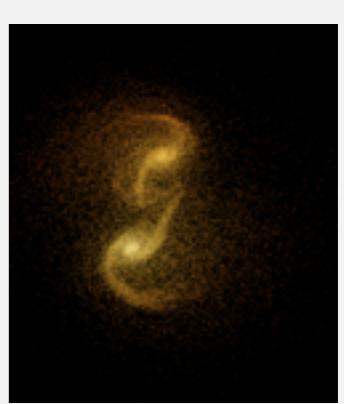
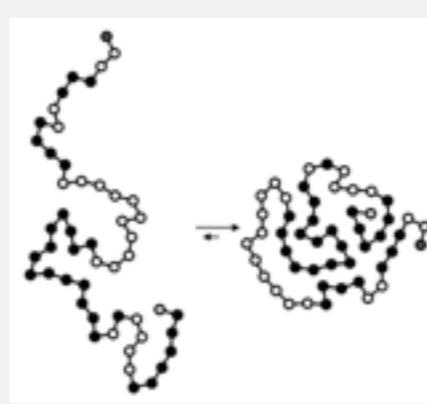
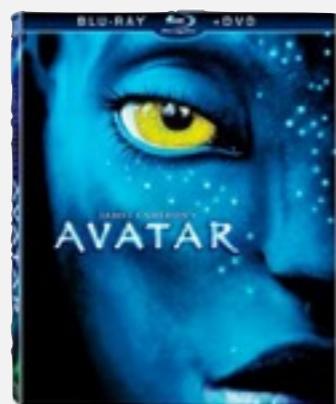
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

Social networks. Recommendations, news feeds, advertisements, ...

Physics. N-body simulation, particle collision simulation, ...

:

Google
YAHOO!
bing



Why study algorithms?

For intellectual stimulation.

“For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.” — Francis Sullivan

FROM THE EDITORS

THE JOY OF ALGORITHMS

Francis Sullivan, Associate Editor-in-Chief



THE THEME OF THIS FIRST-OF-THE-CENTURY ISSUE OF COMPUTING IN SCIENCE & ENGINEERING IS ALGORITHMS. IN FACT, WE WERE BOLD ENOUGH—AND PERHAPS FOOLISH ENOUGH—to call the 10 examples we've selected “THE TOP 10 ALGORITHMS OF THE CENTURY.”

Computational algorithms are probably as old as civilization. Some of the most ancient written records, including the Sumerian cuneiform, are computational. In fact, the first algorithm was named after a Sumerian, the Babylonian mathematician and astronomer, Beroth, who lived around 2300 BC. He was referring to the 15 minutes during which he'd sketched out a fundamental operation of the calendar. The word “algorithm” is derived from the name of the Persian mathematician, Abu Jafar Muhammad ibn Musa al-Khwarizmi, who lived around 800 AD. And I suppose we could claim that the Droid algorithm for estimating the start of summer is embodied in Stonehenge. (That's right, it's a real algorithm.)

Like so many other things that technology affects, algorithms have advanced in startling and unexpected ways in the last few decades. In fact, the 10 algorithms we chose for this issue have been essential for progress in communications, health care, manufacturing, economics, weather prediction, and many other fields.

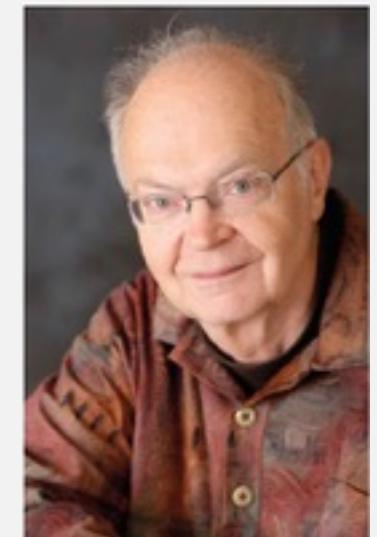
Curiously, progress in these areas has stimulated the search for even more algorithms. Recall one late-night session on the Maryland shore where Bill Gates and Steve Jobs asked, “What are we going to do about the disk?” After all, they don't look very appealing. After the usual quibbling over what to do with them, the two geniuses agreed what must be the right answer—namely, “A very hungry person first ate a crab.”

The point here is that the number of “computations” to invention creates its own necessity. Our need for powerful machines always exceeds their availability. Each significant computation is a challenge to invent a better way to do it. As the gap between the demand for cycles and the available data is still almost untouched, there are still very big challenges coming from more “traditional” fields of computation. For example, how do you get the results of a large floating-point calculation is likely to be correct. Think of the usual question: “How many digits of accuracy does the cost give?” After all, the added confidence in the answer is large. It is there an analog for things such as huge, multidisciplinary projects? The answer is “Yes.”

Computers have cracked many hard problems since January 1990, but we are posing some even harder ones on the next century. In spite of a lot of good work, there are still many problems left to solve. One of the most important is how to bridge the gap between the demand for more computation and the understanding of the basic notions of computational theory. Questions already arising from quantum computing and problems associated with the generation and distribution of quantum states are likely to bring together theories of computing, logic, and the nature of the physical world.

The next century is not going to be very fruitful for us, but it is not going to be dull either. ■

“An algorithm must be seen to be believed.” — Donald Knuth

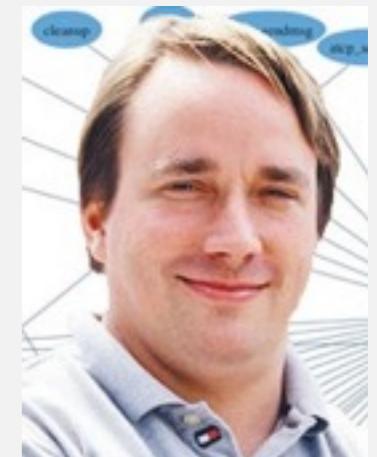


Why study algorithms?

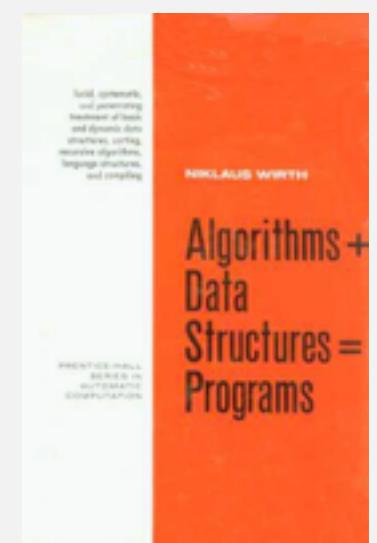
To become a proficient programmer.

“I will, in fact, claim that the difference between a bad programmer and a good one is whether whether the programmer considers code or data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

— Linus Torvalds (creator of Linux)

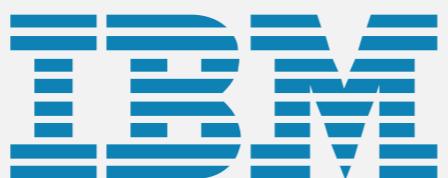


“Algorithms + Data Structures = Programs.” — Niklaus Wirth



Why study algorithms?

For the interview.



Why study algorithms?

- Their impact is broad and far-reaching.
- For intellectual stimulation.
- To become a proficient programmer.
- To pass your job interviews.
- They may unlock the secrets of life and of the universe.

Why study anything else?



Coursework and grading

Weekly assignments and term project. 55%

- Mix of programming and written assignments
- Collaboration/lateness policies: see web.

Exams. 10% + 15% + 15%

- Midterm 1 (in class on Thursday, September 29).
- Midterm 2 (in class on Thursday, November 3)
- Final (finals week, Thursday, December 15 at 9am).

Attendance of Lecture and Recitations. 5%

- Attendance of both is mandatory



Textbook

Required reading. Algorithms 4th edition by R. Sedgewick and K. Wayne.
This course closely follows the textbook and uses their lectures.



Available in hardcover and Kindle.

- Online: \$60/\$55 to buy, \$18 to rent

Where to get help?

Piazza discussion forum.

- Low latency, low bandwidth.
- Mark solution-revealing questions as private.

piazza

<http://piazza.com/upenn/fall2016/cis121>

Office hours.

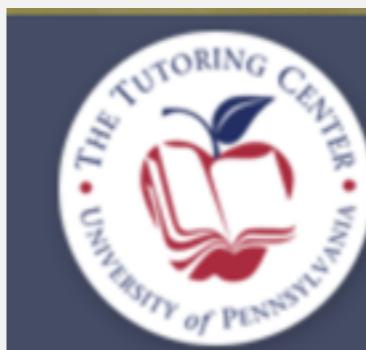
- High bandwidth, high latency.
- See web for schedule.



<http://www.seas.upenn.edu/~cis121/>

Tutoring.

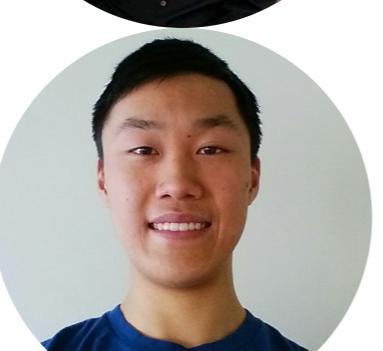
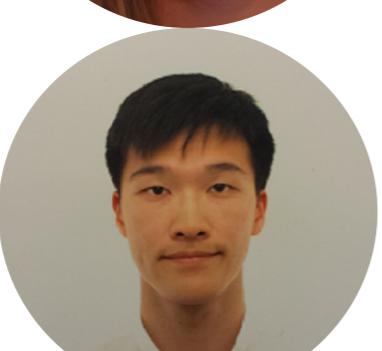
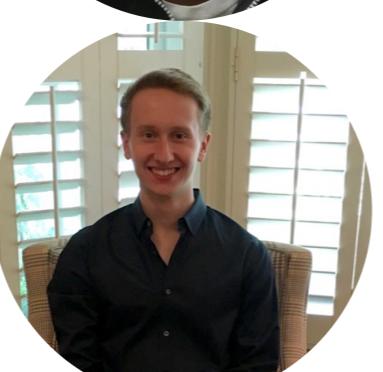
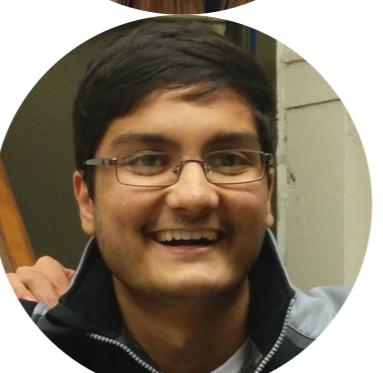
- The Tutoring Center



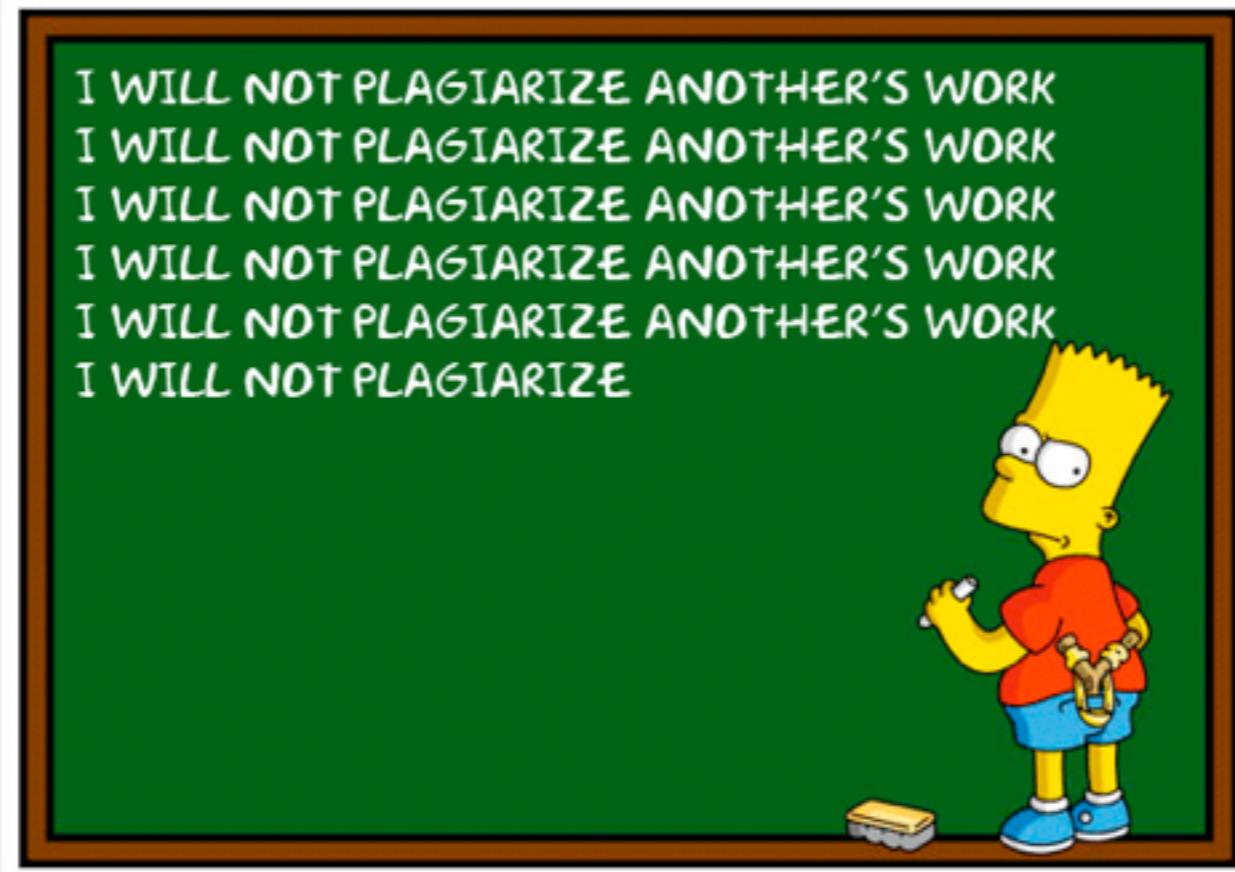
The Tutoring Center

<http://www.vpul.upenn.edu/tutoring/>

Course Staff



Where not to get help?



<http://world.edu/academic-plagiarism>

http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html

Cheating policy

- All homework submissions are checked by plagiarism detection software
- Any assignments that are flagged will be automatically referred to the Office of Student Conduct, which will adjudicate whether the course collaboration policy was violated
- First violation = 0 on the assignment. Second violation = F in the class

How to succeed in CIS 121

Make the class part of your routine

- Come to lectures
- Engage in recitations
- Read the assigned chapters immediately before or after lecture
- Start homework assignments early
- Do not wait until the last day

Make use of your TAs and instructors

- Attend recitations
- Go to office hours (fine to come even if you don't have a specific question)

Do not cheat

Quiz on course policies, due this Thursday. Go to the 121 web page.

<http://www.seas.upenn.edu/~cis121/current/>

CIS 121 Lectures Labs Homework ▾ Exams Staff Calendar Resources

Course:	CIS 121 - Fall 2016
Instructor:	Chris Callison-Burch
Teaching Assistants:	Course staff
Discussion Forum:	Piazza
Lectures:	Tuesdays and Thursdays at 10:30 AM in Chem 102
Prerequisites:	CIS 120 and CIS 160
Textbook:	Algorithms (4th Edition) by Robert Sedgewick and Kevin Wayne. The lectures and the required reading will draw from the textbook. It is available from the Penn Bookstore, as well as various online retailers.
Materials:	We require everyone in the course to have their own clicker. You can re-use the one that you have from CIS 110 or 120, or you can purchase one from the bookstore if you don't have one. Please register your clicker on the Turning Technologies web site .

There is a quiz on the course policies which is due by 11:59pm on Thursday September 1. You can complete the quiz by filling out this Google form.

Grading:

The grading for the course will consist of:

- 55% for homework assignments and the final project
- 10% for midterm 1

Professor Callison-Burch (not Professor Burch)

Bachelors from Stanford

PhD from University of Edinburgh

6 years at Johns Hopkins University

4th year here at Penn

Research interests:

Natural language processing

Crowdsourcing

Come talk to me about research!

See more:

<http://www.cis.upenn.edu/~ccb/>

<http://paraphrase.org/>

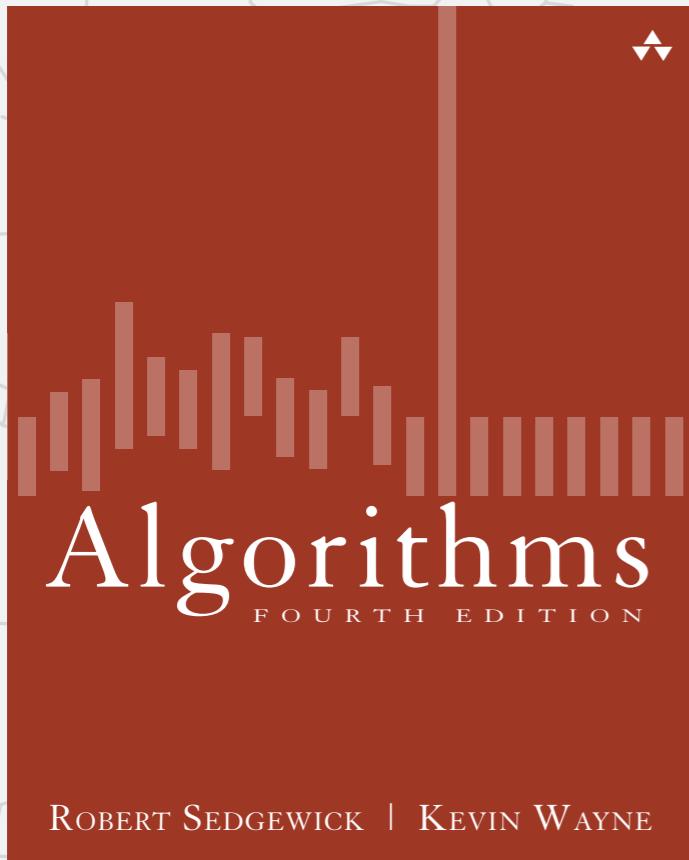
<http://gun-violence.org/>

Favorite data structure: Bloom Filter



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

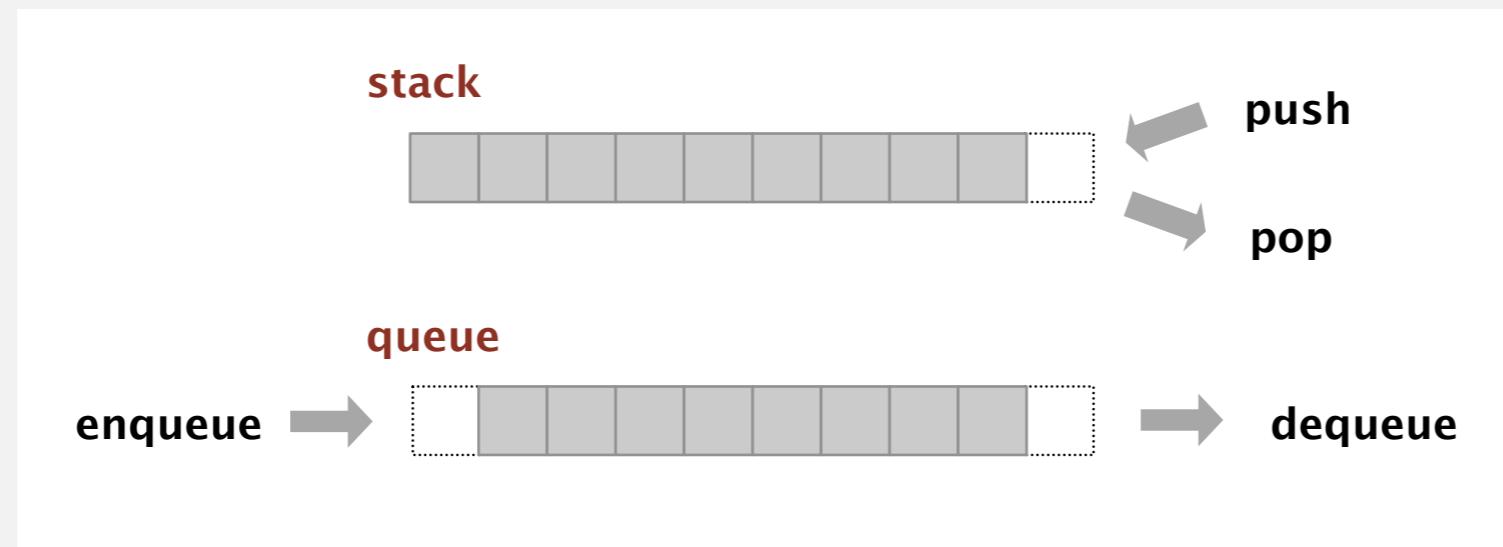
1.3 STACKS AND QUEUES

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *applications*

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: **add**, **remove**, **iterate**, **test if empty**.
- Intent is clear when we add.
- Which item do we remove?

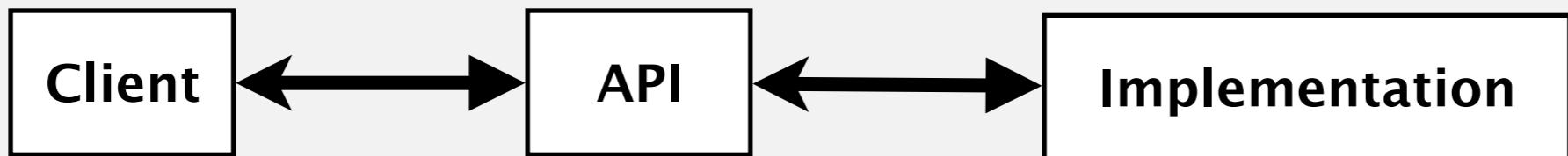


Stack. Examine the item most recently added. ← LIFO = "last in first out"

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

Client, implementation, API

Separate client and implementation via API.



API: description of data type, basic operations.

Client: program using operations defined in API.

Implementation: actual code implementing operations.

Benefits.

- **Design:** creates modular, reusable libraries.
- **Performance:** substitute optimized implementation when it matters.

Ex. Stack, queue, bag, priority queue, symbol table, union-find,

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

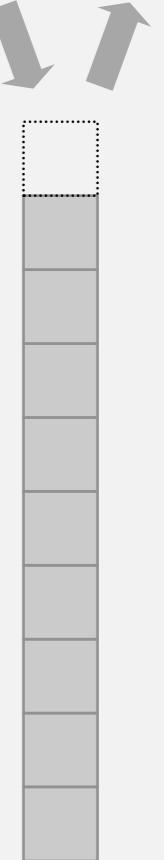
1.3 BAGS, QUEUES, AND STACKS

- ▶ ***stacks***
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *applications*

Stack API

Warmup API. Stack of strings data type.

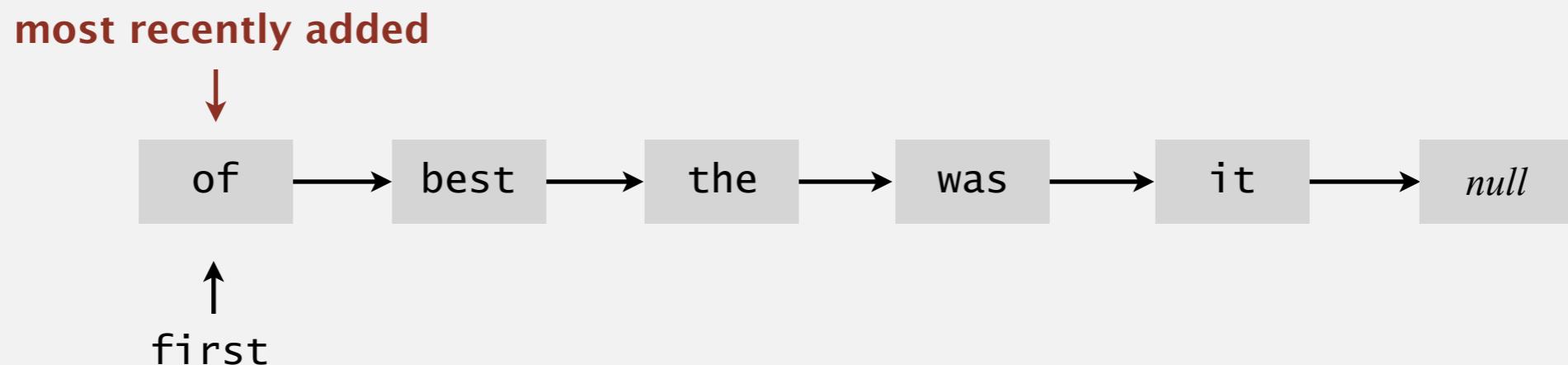
		push pop
<code>public class StackOfStrings</code>		
	<code>StackOfStrings()</code>	<i>create an empty stack</i>
	<code>void push(String item)</code>	<i>add a new string to stack</i>
	<code>String pop()</code>	<i>remove and return the string most recently added</i>
	<code>boolean isEmpty()</code>	<i>is the stack empty?</i>
	<code>int size()</code>	<i>number of strings on the stack</i>



Potential client use case. Reverse sequence of strings from standard input.

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

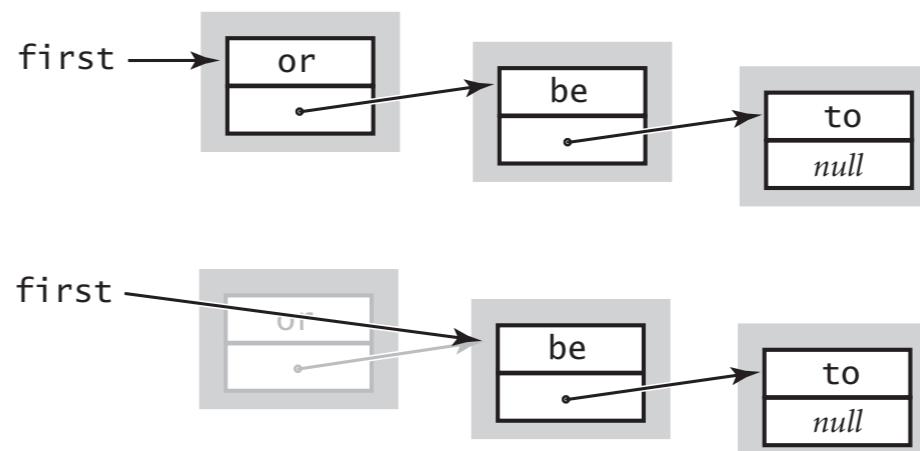
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

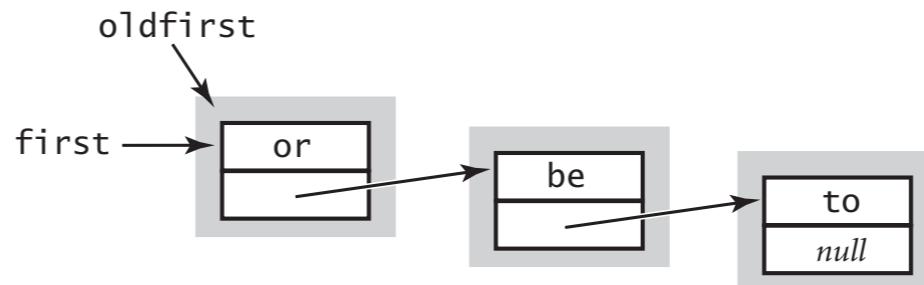
Stack push: linked-list implementation

inner class

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

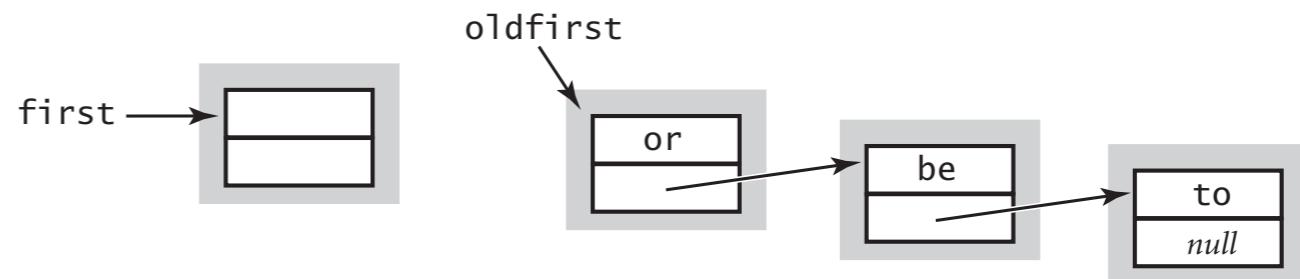
save a link to the list

```
Node oldfirst = first;
```



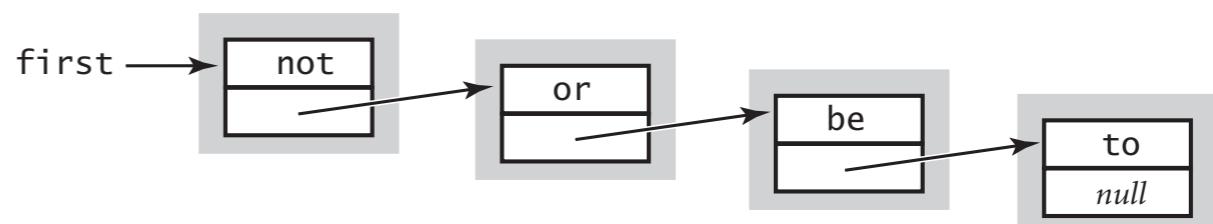
create a new node for the beginning

```
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
    {
        private String item;
        private 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)



Stack: linked-list implementation performance

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

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

inner class

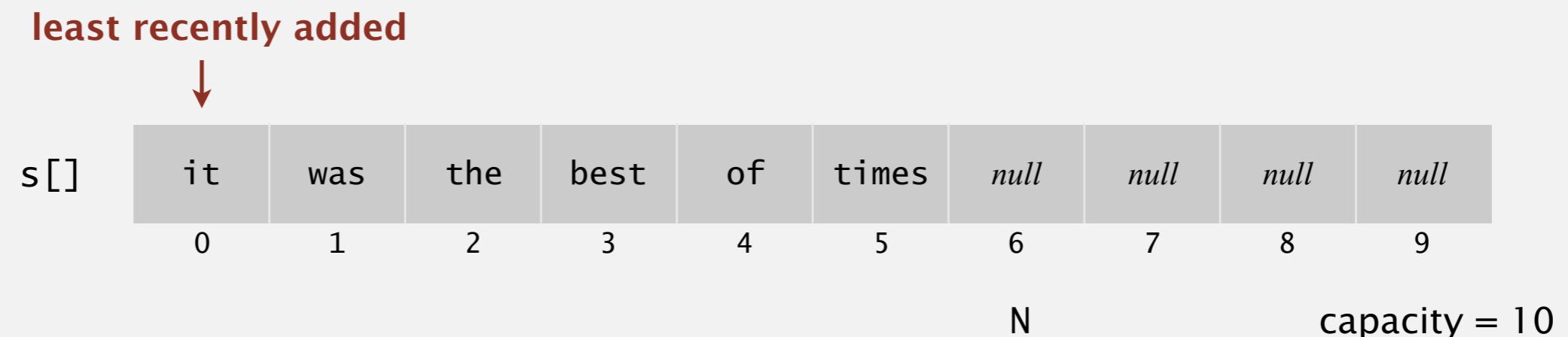
```
private class Node
{
    String item;
    Node next;
}
```



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

Fixed-capacity stack: array implementation

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



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

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];   }
}
```

use to index into array;
then increment N

a cheat
(stay tuned)

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 added.

Duplicate items. We allow an item to be added more than once.

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 remaining references

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *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.
- Array accesses to add first N items = $N + (2 + 4 + \dots + 2(N-1)) \sim N^2$.

quadratic time, which is
infeasible for large N

\uparrow \uparrow
1 array access 2($k-1$) array accesses to expand to size k
per push (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?

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 add first $N = 2^i$ items. $N + (2 + 4 + 8 + \dots + N) \sim 3N$.

↑
1 array access
per push

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

Stack: resizing-array implementation

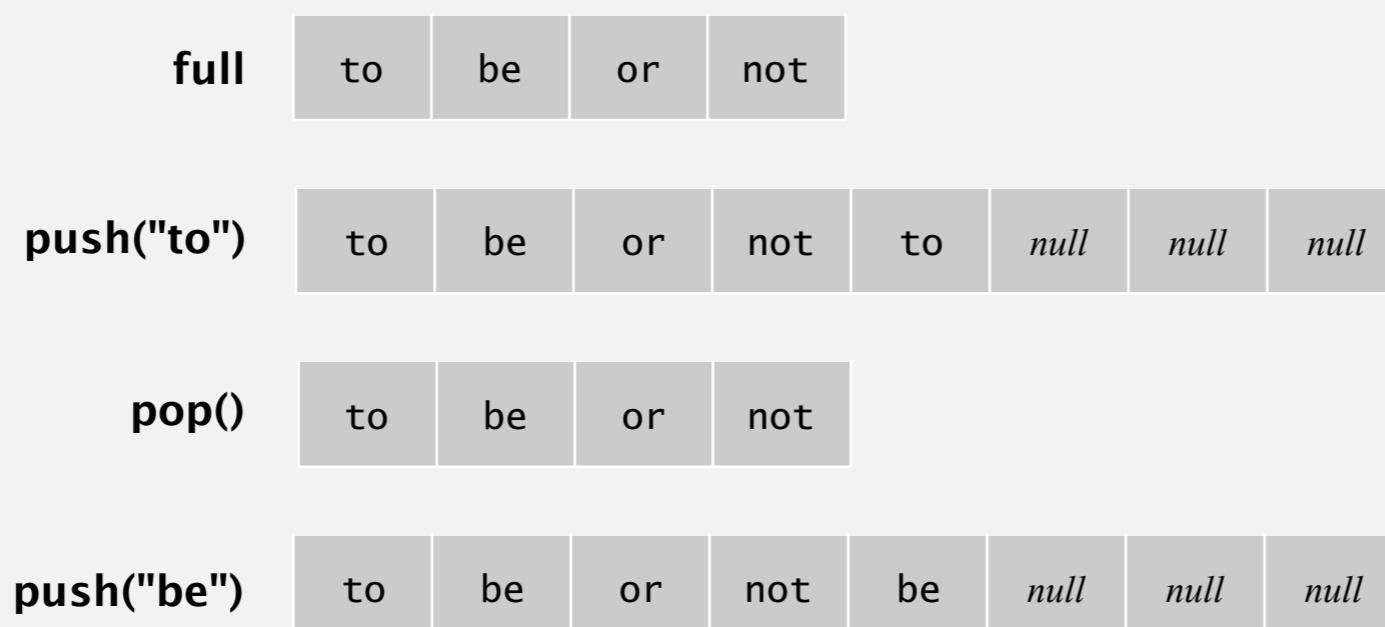
Q. How to shrink array?

First try.

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

Too expensive in worst case.

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



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: 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 .

	typical	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

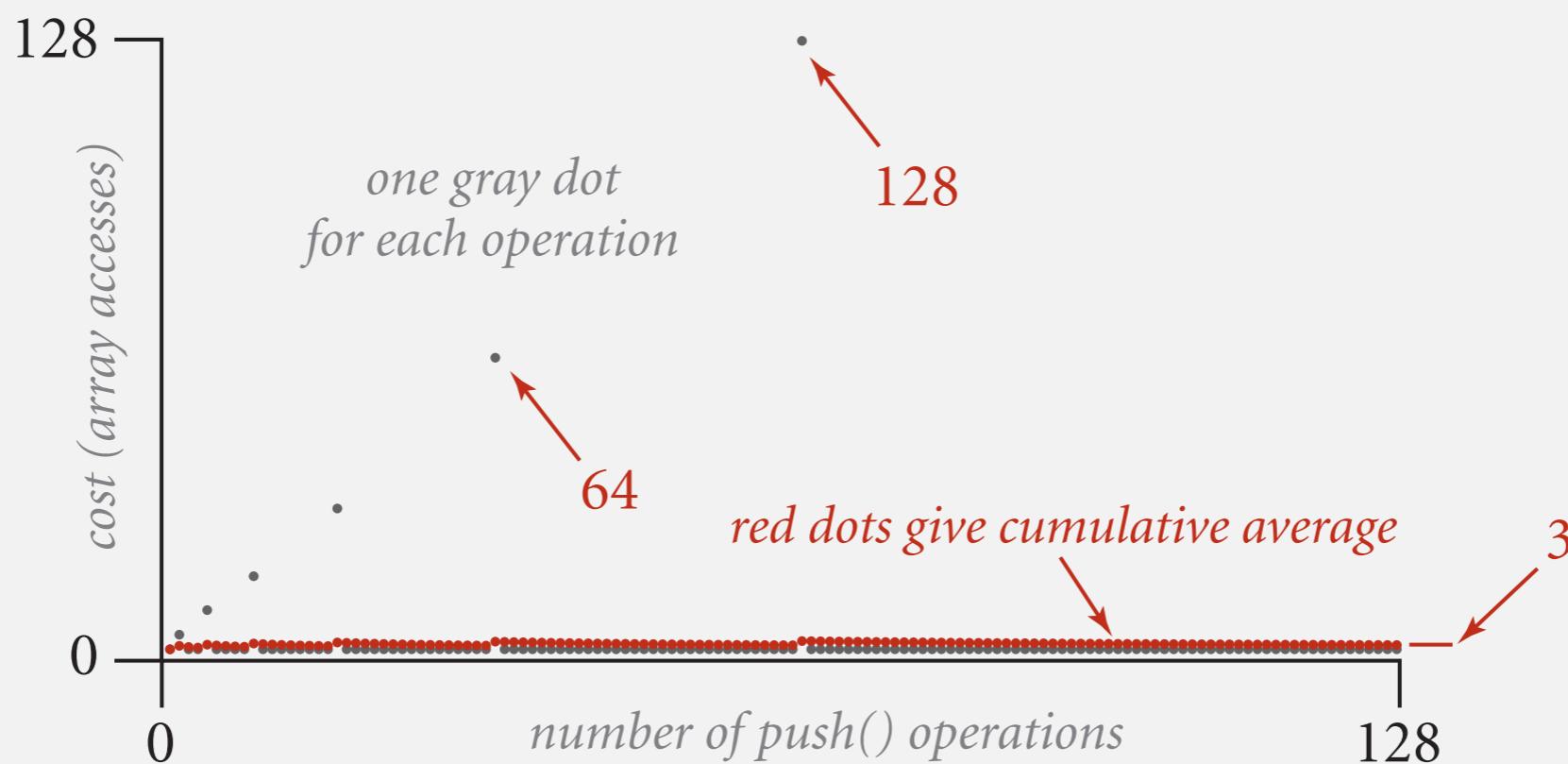
**order of growth of running time
for resizing array stack with N items**

Two red arrows point from the 'worst' column to the 'N' values in the 'push' and 'pop' rows. A red callout bubble points to these two 'N' values with the text "doubling and halving operations".

Stack: amortized cost of adding to a stack

Cost of adding first N items. $N + (2 + 4 + 8 + \dots + N) \sim 3N.$

\uparrow
1 array access
per push \uparrow
k array accesses to double to size k
(ignoring cost to create new array)



Stack resizing-array implementation: memory usage

Proposition. A `ResizingArrayStackOfStrings` uses $\sim 8N$ to $\sim 32N$ bytes of memory for a stack with N items.

- $\sim 8N$ when full.
- $\sim 32N$ when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
    private String[] s; ← 8 bytes × array size
    private int N = 0;
    ...
}
```

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

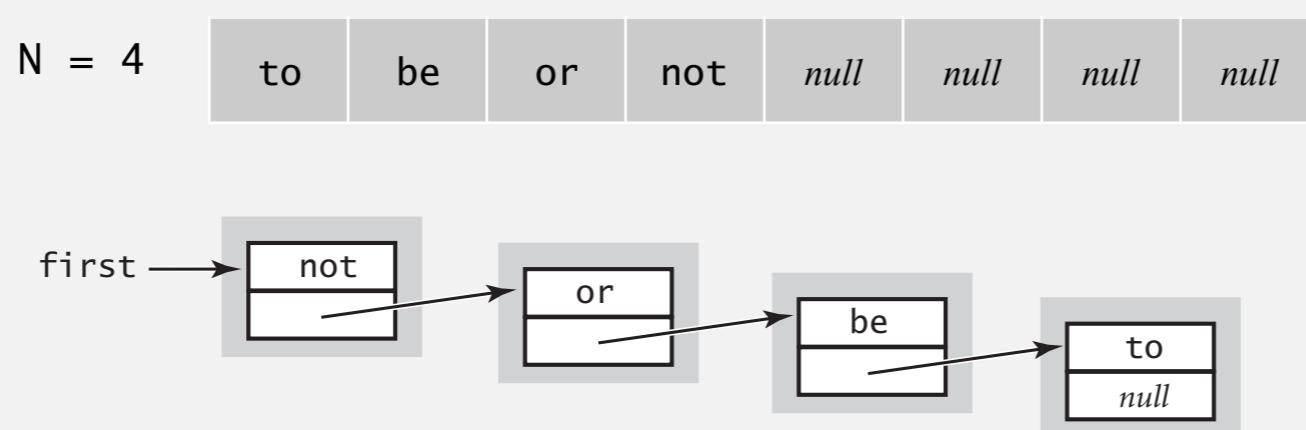
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

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 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ **queues**
- ▶ *generics*
- ▶ *applications*

Queue API

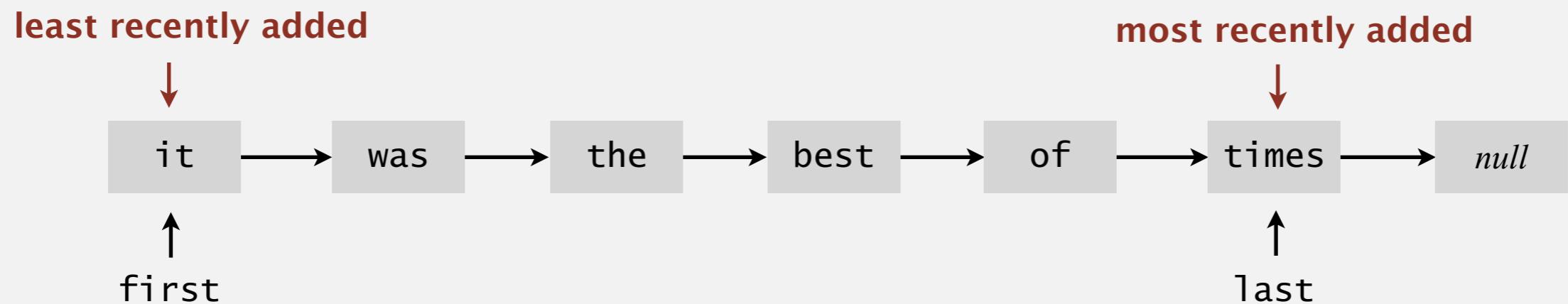
		enqueue
<pre>public class QueueOfStrings</pre>		
	<code>QueueOfStrings()</code>	<i>create an empty queue</i>
	<code>void enqueue(String item)</code>	<i>add a new string to queue</i>
	<code>String dequeue()</code>	<i>remove and return the string least recently added</i>
	<code>boolean isEmpty()</code>	<i>is the queue empty?</i>
	<code>int size()</code>	<i>number of strings on the queue</i>



dequeue

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

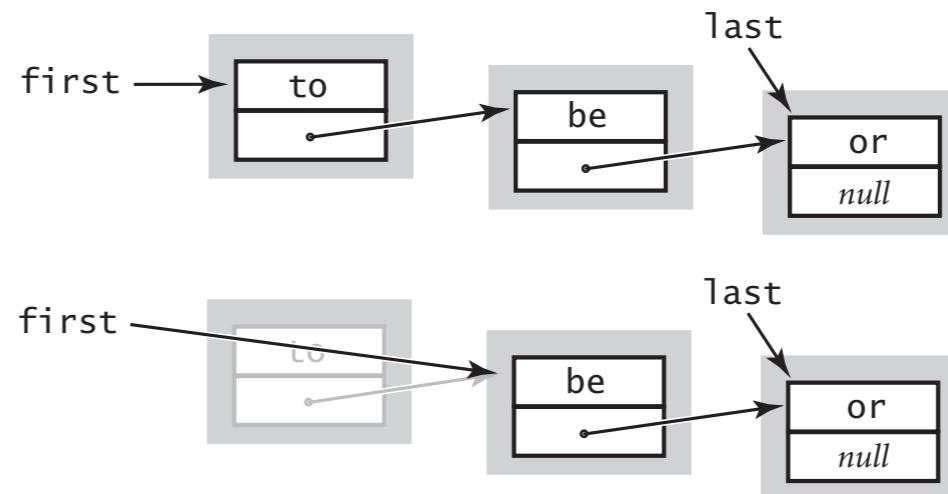
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

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

delete first node

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



return saved item

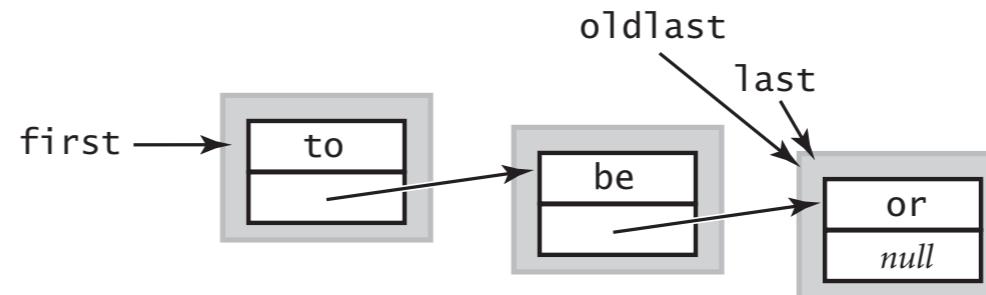
```
return item;
```

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

Queue enqueue: linked-list implementation

save a link to the last node

```
Node oldlast = last;
```

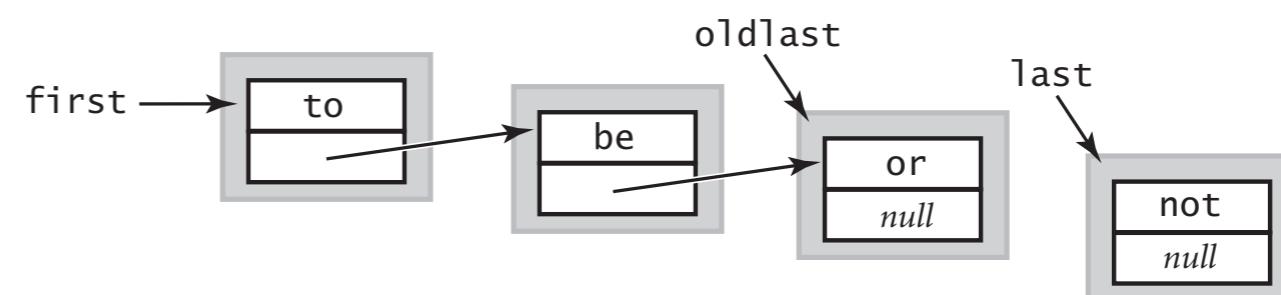


inner class

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

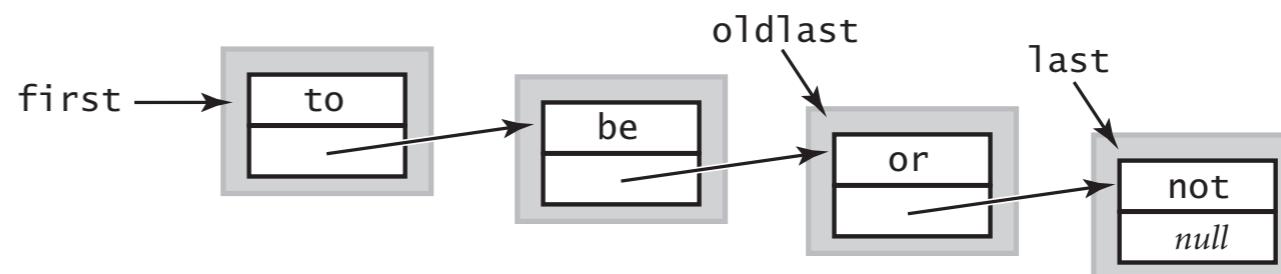
create a new node for the end

```
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;

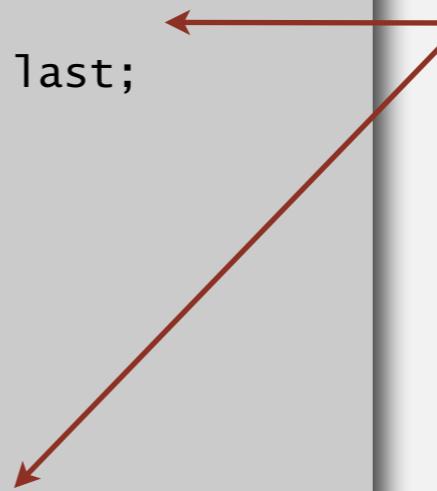
    private class Node
    { /* same as in LinkedStackOfStrings */ }

    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.
 - $\text{enqueue}()$: add new item at $q[\text{tail}]$.
 - $\text{dequeue}()$: remove item from $q[\text{head}]$.
 - Update head and tail modulo the capacity.
 - Add resizing array.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

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

Parameterized stack

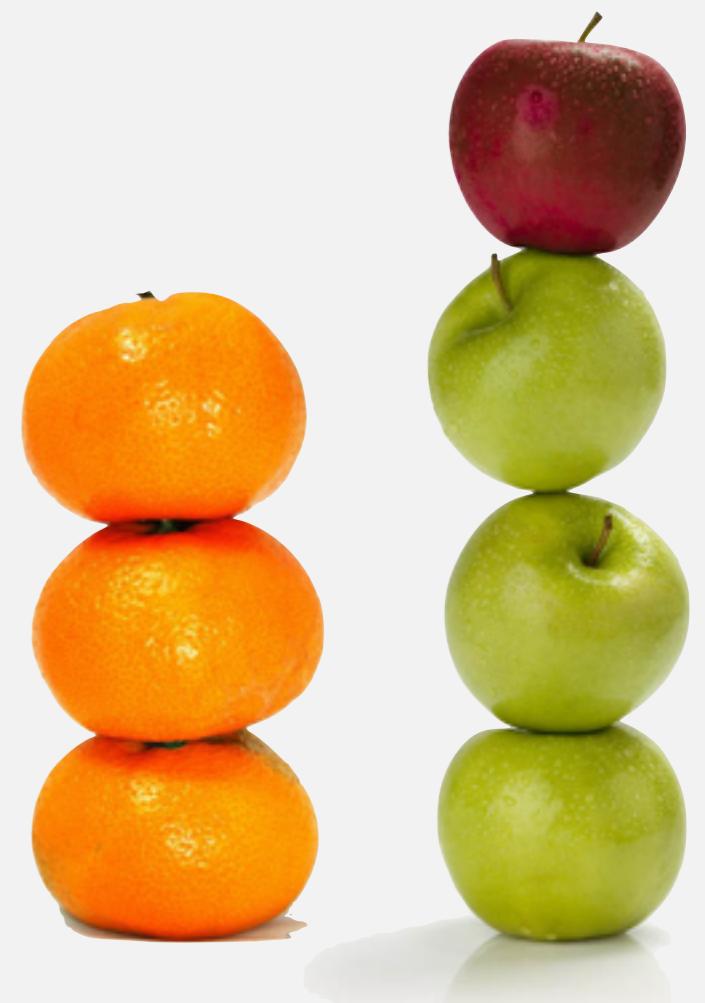
We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfApples, StackOfOranges,

Solution in Java: generics.

type parameter
(use both to specify type and to call constructor)

```
Stack<Apple> stack = new Stack<Apple>();  
Apple apple = new Apple();  
Orange orange = new Orange();  
stack.push(apple);  
stack.push(orange); ← compile-time error  
...
```



Generic stack: linked-list implementation

```
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;
    }
}
```

```
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

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

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

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

generic type name

Generic stack: array implementation

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

    public ..StackOfStrings(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];   }
}
```

the way it should be

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = new Item[capacity];   }

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

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

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

@#\$*! generic array creation not allowed in Java

Generic stack: array implementation

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

    public ..StackOfStrings(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];   }
}
```

the way it is

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = (Item[]) new Object[capacity];   }

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

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

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

the ugly cast

Unchecked cast

```
% javac FixedCapacityStack.java
```

Note: FixedCapacityStack.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

FixedCapacityStack.java:26: warning: [unchecked] unchecked cast

found : java.lang.Object[]

required: Item[]

```
    a = (Item[]) new Object[capacity];  
           ^
```

1 warning

Q. Why does Java make me cast (or use reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about **type erasure** and **covariant arrays**.



Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a **wrapper** object type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast between a primitive type and its wrapper.

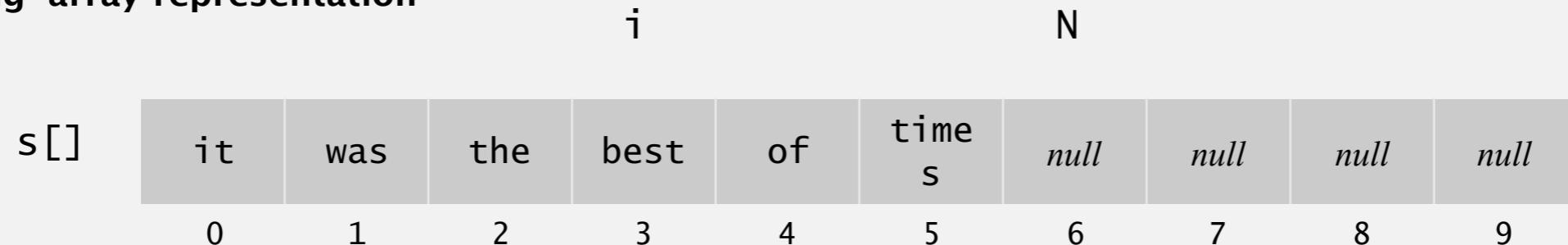
```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);           // stack.push(Integer.valueOf(17));
int a = stack.pop();     // int a = stack.pop().intValue();
```

Bottom line. Client code can use generic stack for **any** type of data.

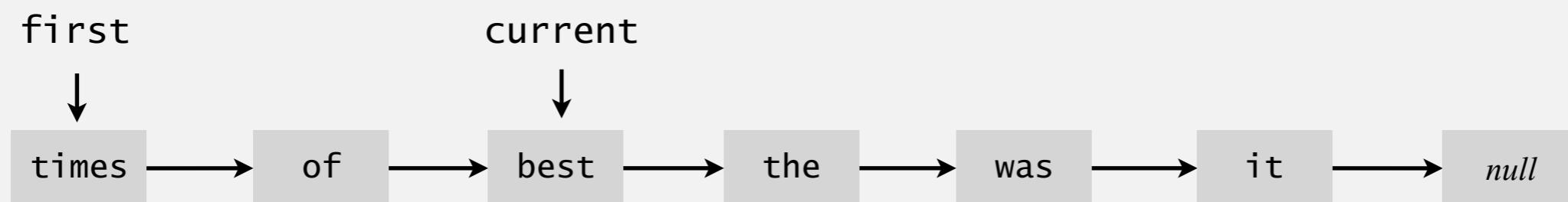
Iteration

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

resizing-array representation



linked-list representation



Java solution. Use a **foreach** loop.

Foreach loop

Java provides elegant syntax for iteration over collections.

“foreach” loop (shorthand)

```
Stack<String> stack;  
...  
  
for (String s : stack)  
    ...
```

equivalent code (longhand)

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

To make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core methods.
 - the `hasNext()` method returns `false` when there are no more items
 - the `next()` method returns the next item in the collection

Iterators

To support foreach loops, Java provides two interfaces.

- Iterator interface: next() and hasNext() methods.
- Iterable interface: iterator() method that returns an Iterator.
- Both should be used with generics.

java.util.Iterator interface

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove(); ← optional; use
                    at your own risk
}
```

java.lang.Iterable interface

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

Type safety.

- Data type must use these interfaces to support foreach loop.
- Client program won't compile if implementation doesn't.

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;

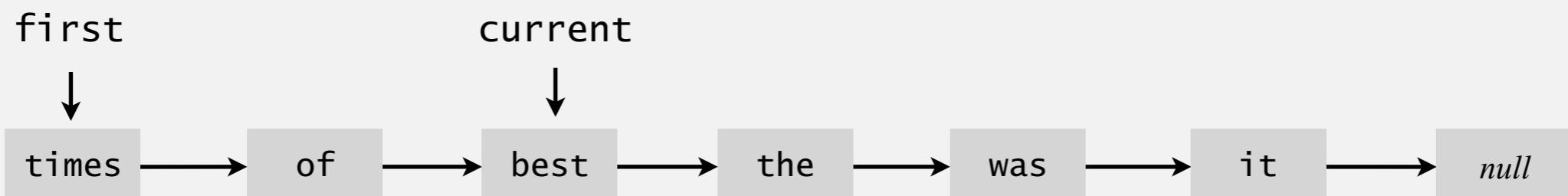
        public boolean hasNext() { return current != null; }

        public void remove() { /* not supported */ }

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

Diagram annotations for the ListIterator code:

- An arrow points from the `remove()` method to the note: "throw UnsupportedOperationException".
- An arrow points from the `next()` method to the note: "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 void remove()    { /* not supported */ }
        public Item next()      { return s[--i]; }
    }
}
```

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

Bag API

Main application. Adding items to a collection and iterating
(when order doesn't matter).

```
public class Bag<Item> implements Iterable<Item>
```

Bag()

create an empty bag

void add(Item x)

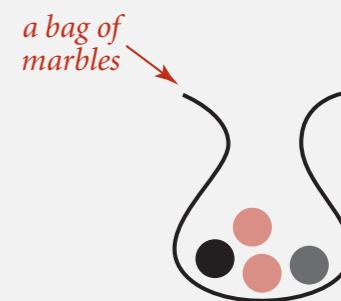
add a new item to bag

int size()

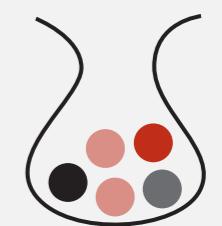
number of items in bag

Iterator<Item> iterator()

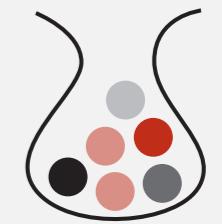
iterator for all items in bag



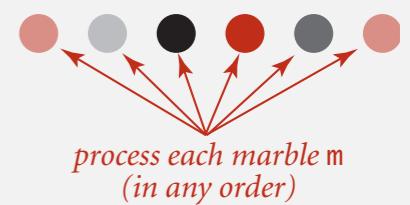
add(●)



add(○)



for (Marble m : bag)



Implementation. Stack (without pop) or queue (without dequeue).

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ ***applications***

Java collections library

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

```
public interface List<Item> extends Iterable<Item>
```

<code>List()</code>	<i>create an empty list</i>
<code>boolean isEmpty()</code>	<i>is the list empty?</i>
<code>int size()</code>	<i>number of items</i>
<code>void add(Item item)</code>	<i>add item to the end</i>
<code>Item get(int index)</code>	<i>return item at given index</i>
<code>Item remove(int index)</code>	<i>return and delete item at given index</i>
<code>boolean contains(Item item)</code>	<i>does the list contain the given item?</i>
<code>Iterator<Item> iterator()</code>	<i>iterator over all items in the list</i>
<code>:</code>	

Implementations. `java.util.ArrayList` uses a resizing array;

`java.util.LinkedList` uses doubly-linked list. ← Caveat: only some operations are efficient.

Java collections library

[java.util.Stack](#)

- Supports push(), pop(), and iteration.
- Inherits from java.util.Vector, which implements java.util.List interface.



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.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

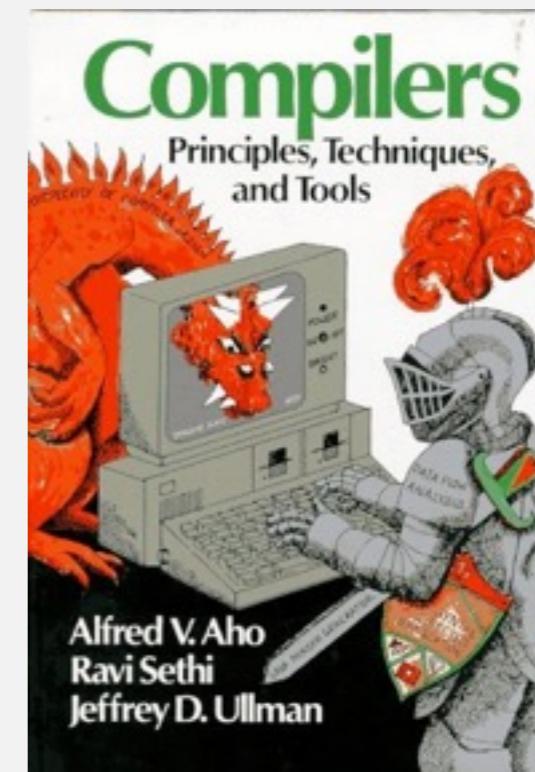
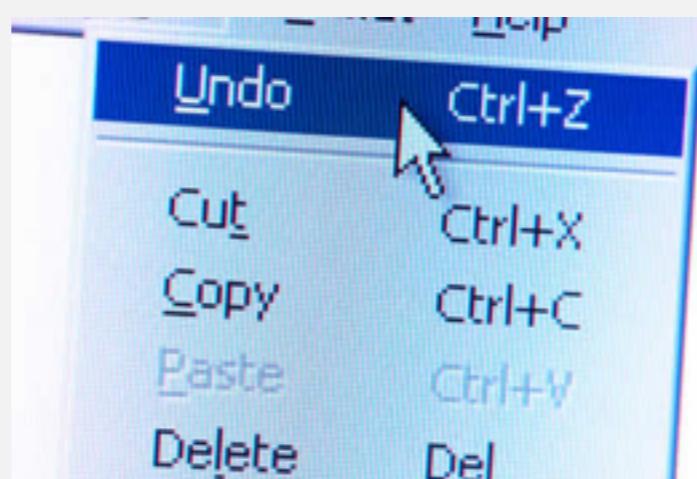


`java.util.Queue`. An interface, not an implementation of a queue.

We suggest. Implement your own Stack and Queue following the textbook to be sure you understand what's going on.

Stack applications

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



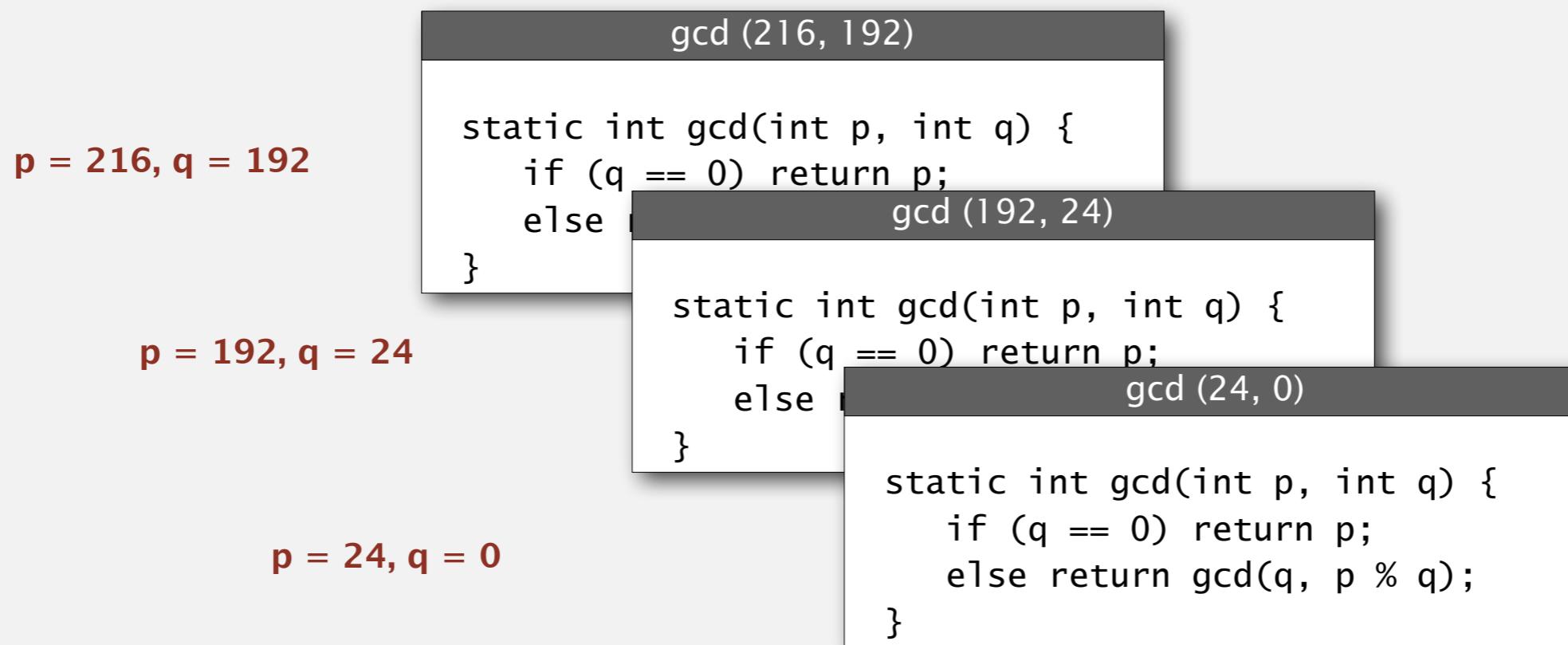
Function calls

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.

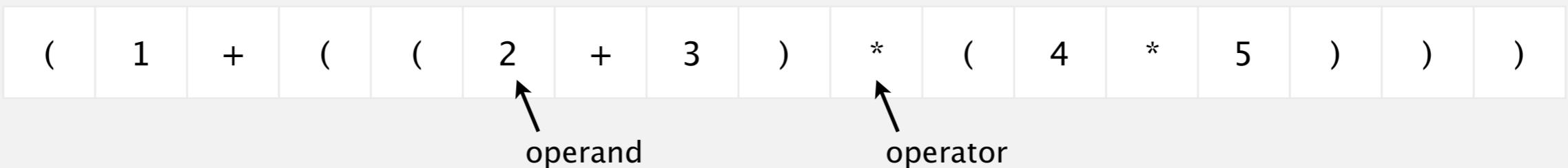
Note. Can always use an explicit stack to remove recursion.



Dijkstra's two-stack algorithm demo



infix expression (fully parenthesized)



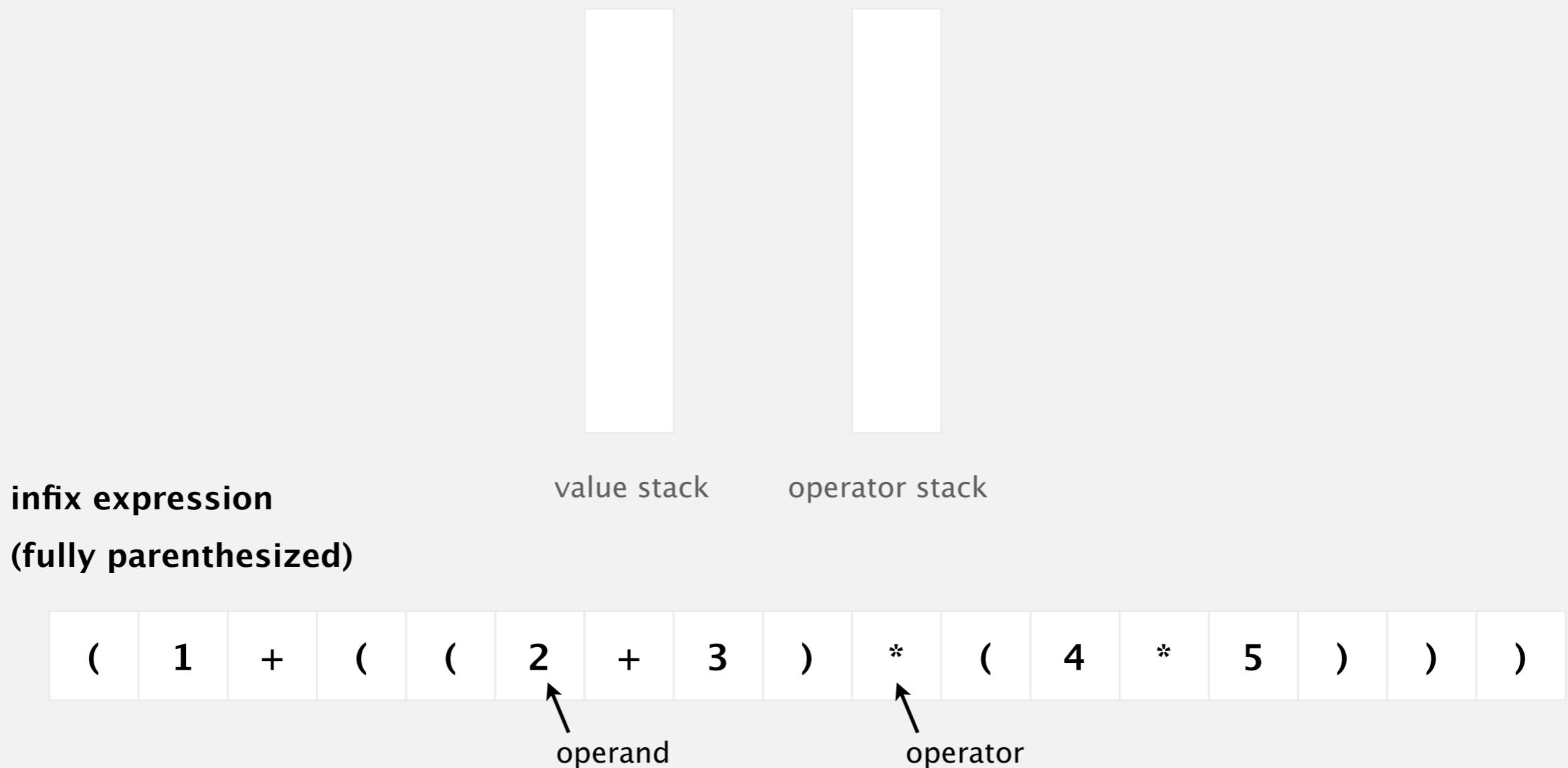
Dijkstra's two-stack algorithm

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.



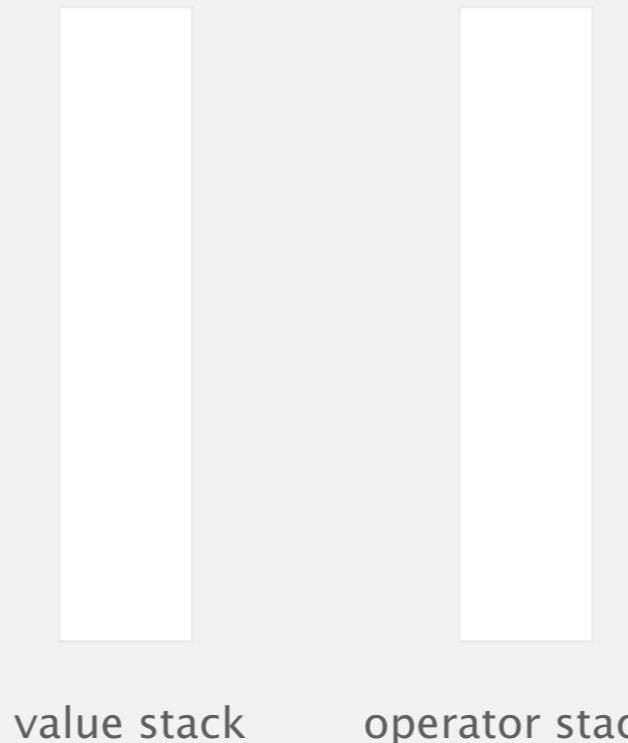
Dijkstra's two-stack algorithm

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.



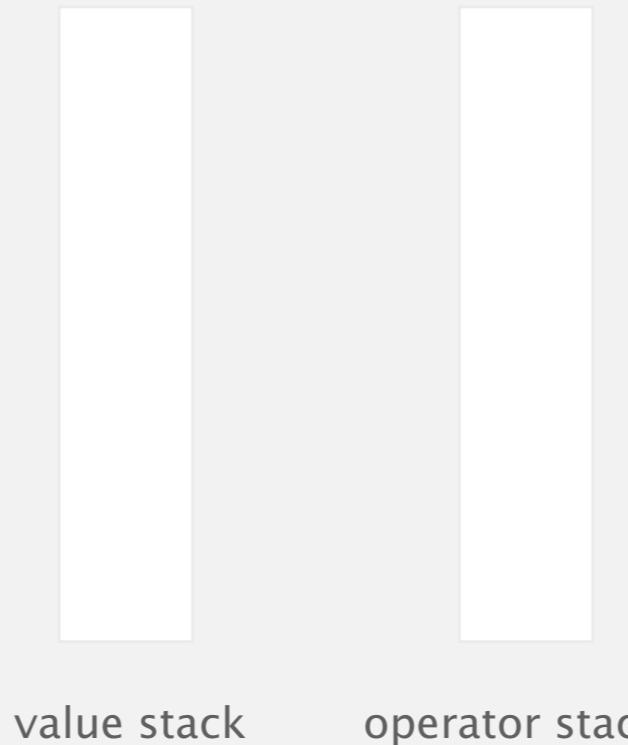
Dijkstra's two-stack algorithm

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.



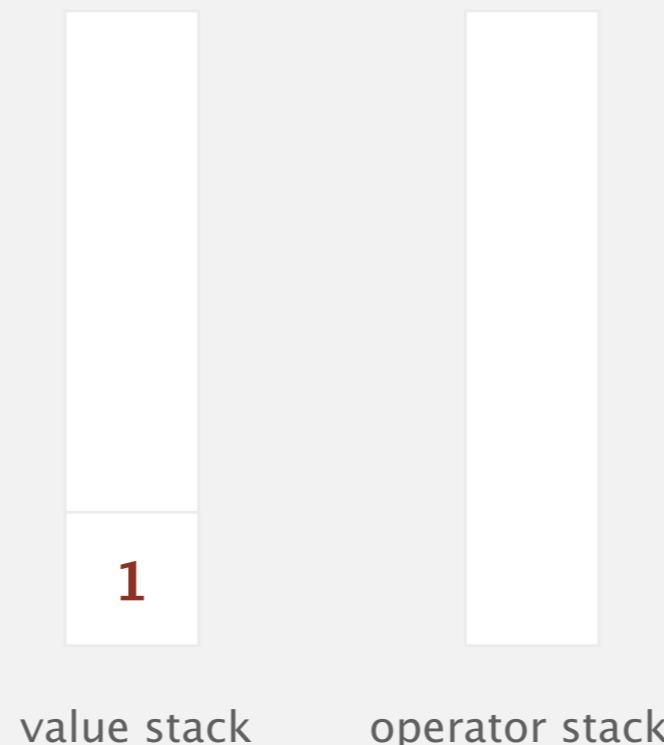
Dijkstra's two-stack algorithm

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.



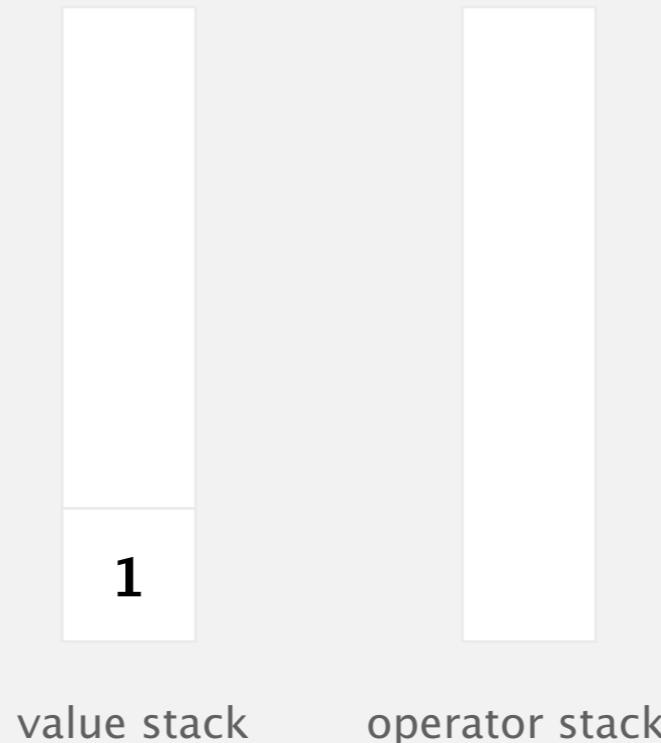
Dijkstra's two-stack algorithm

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.



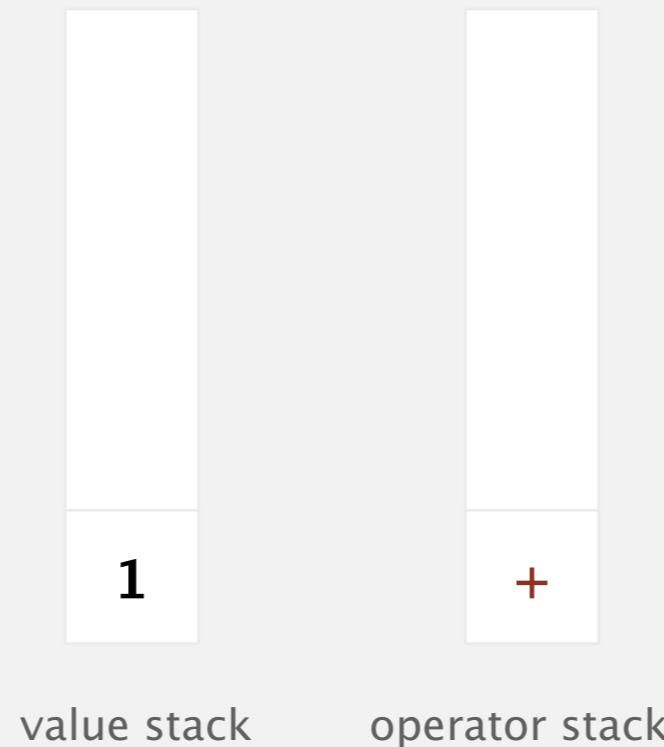
Dijkstra's two-stack algorithm

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.



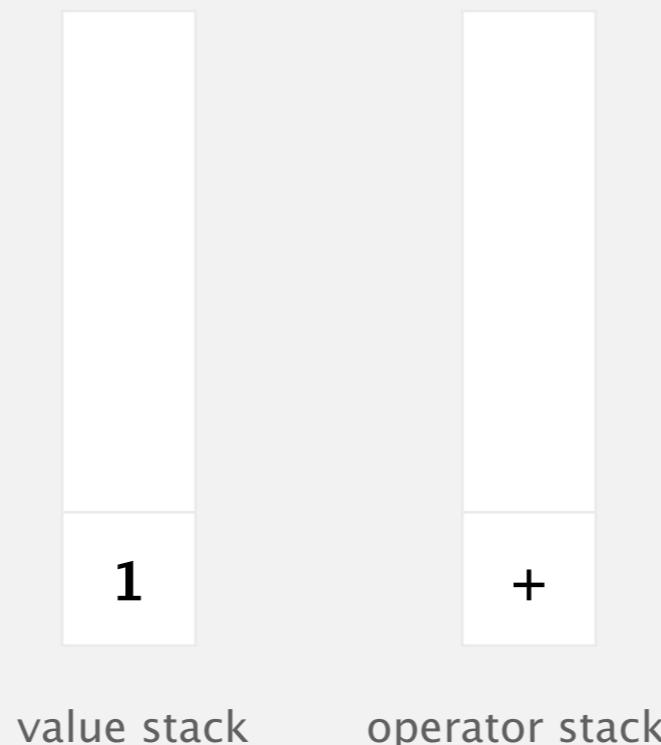
Dijkstra's two-stack algorithm

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.



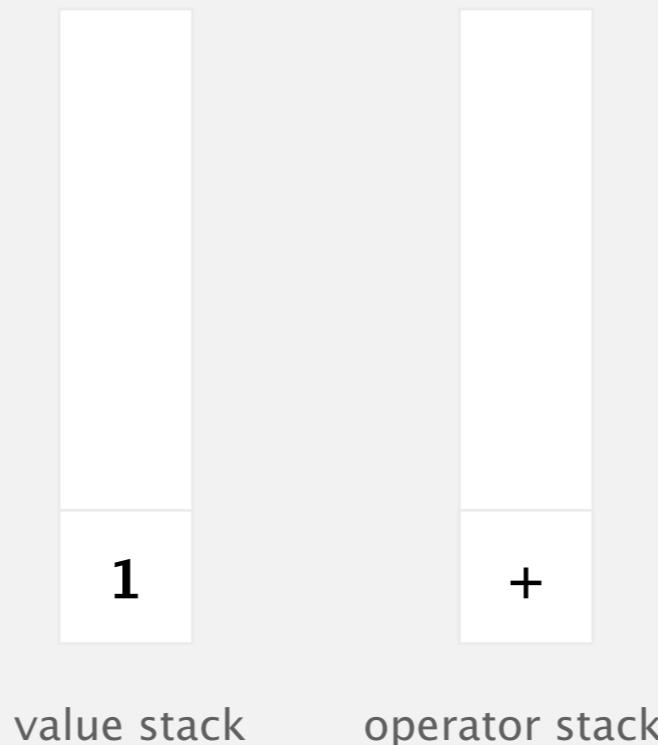
Dijkstra's two-stack algorithm

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.



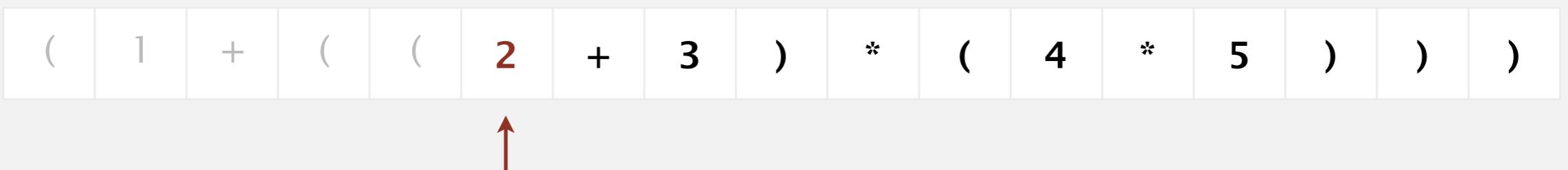
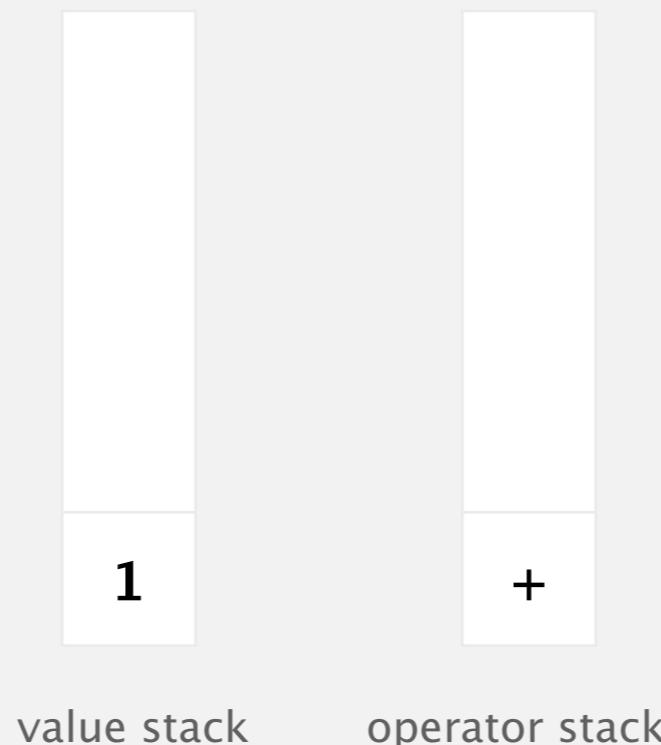
Dijkstra's two-stack algorithm

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.



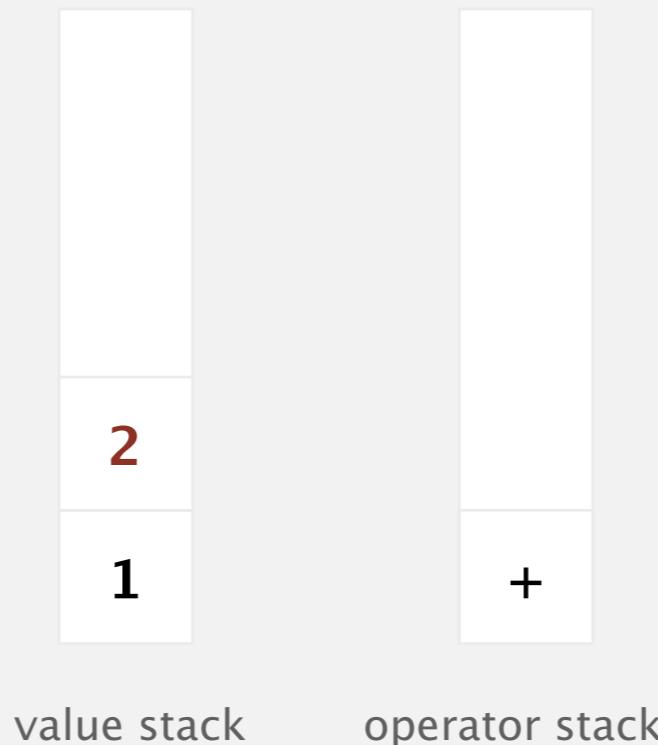
Dijkstra's two-stack algorithm

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.



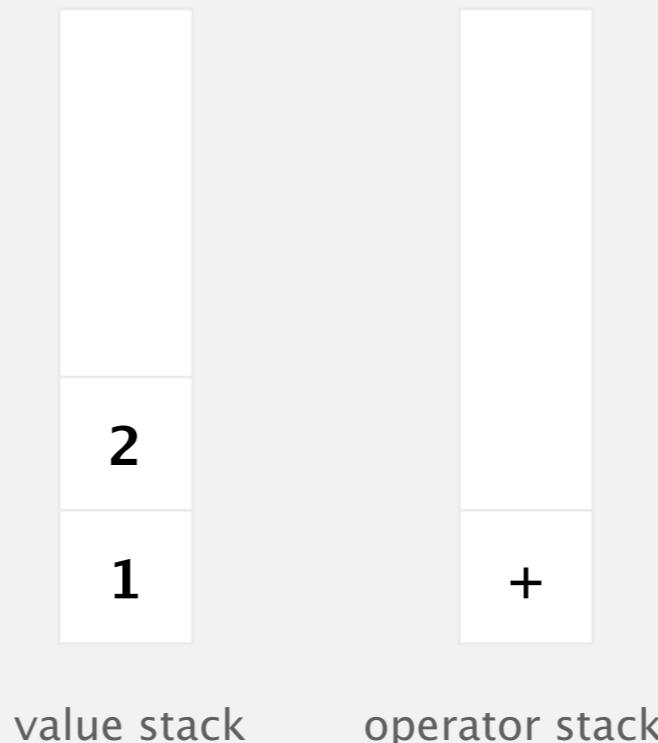
Dijkstra's two-stack algorithm

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.



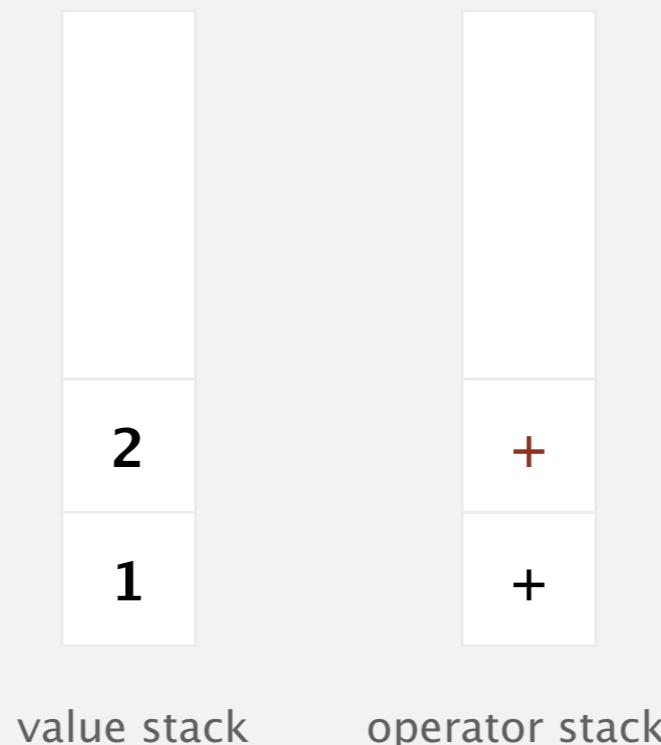
Dijkstra's two-stack algorithm

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.



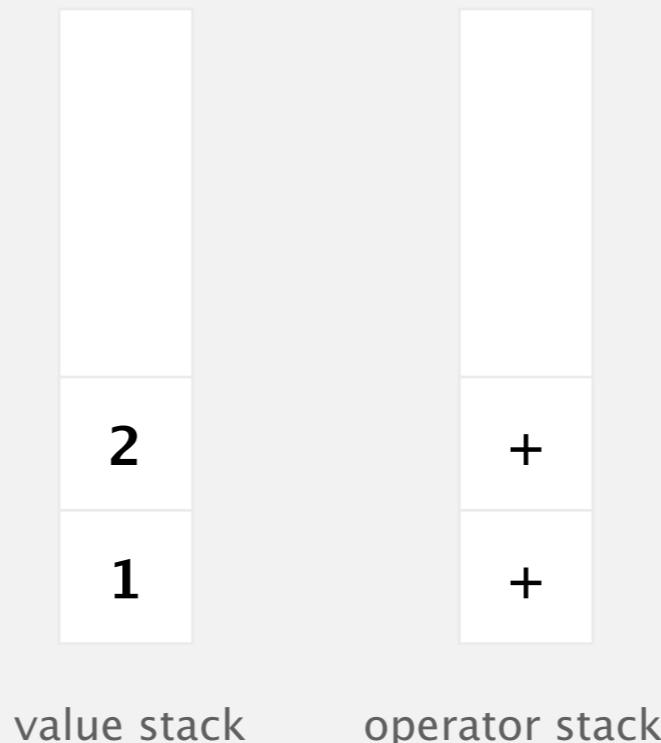
Dijkstra's two-stack algorithm

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.



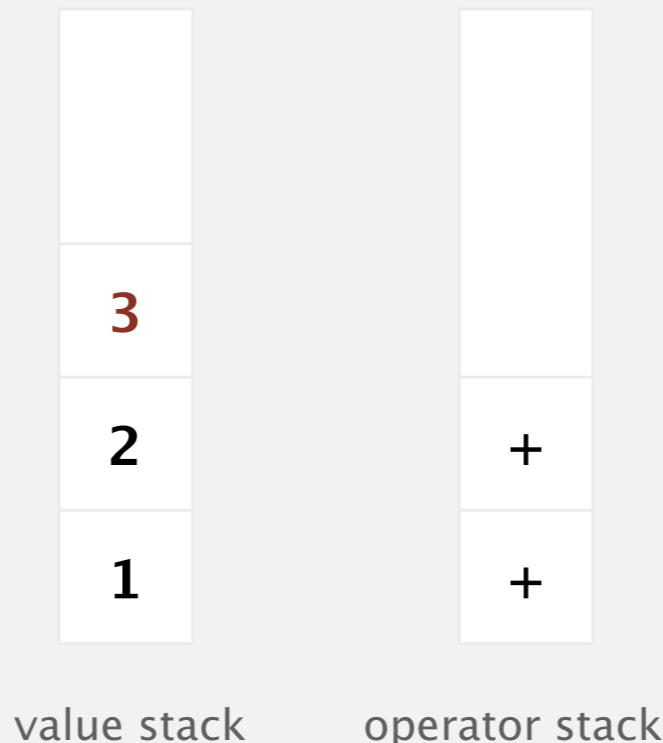
Dijkstra's two-stack algorithm

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.



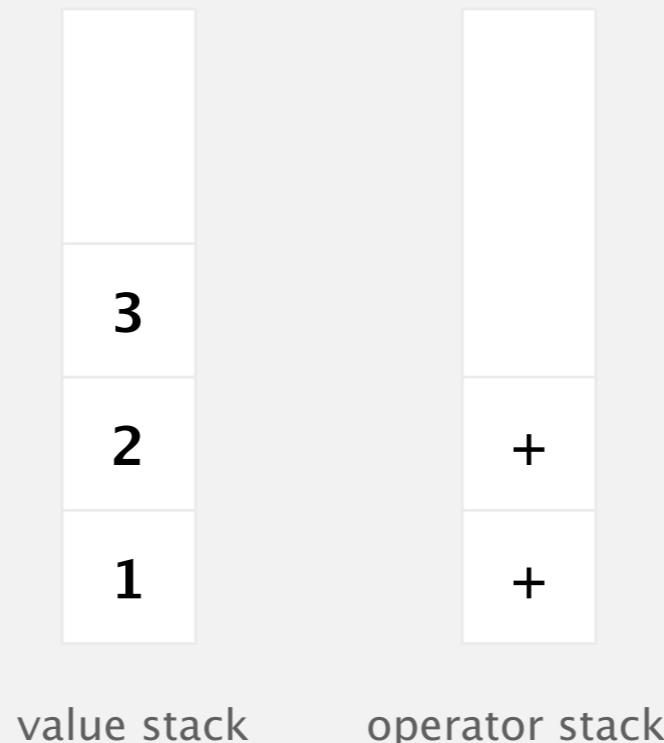
Dijkstra's two-stack algorithm

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.



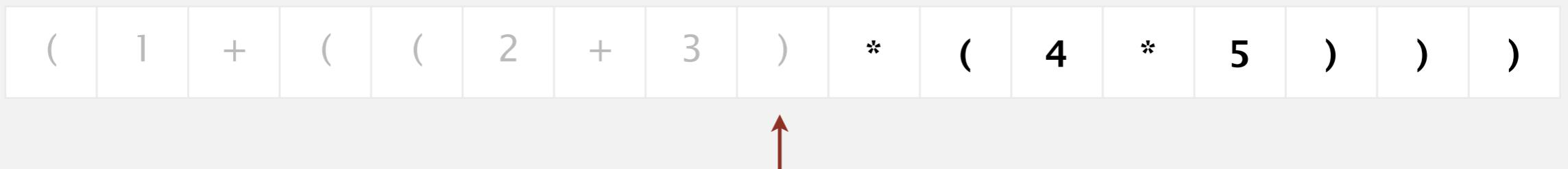
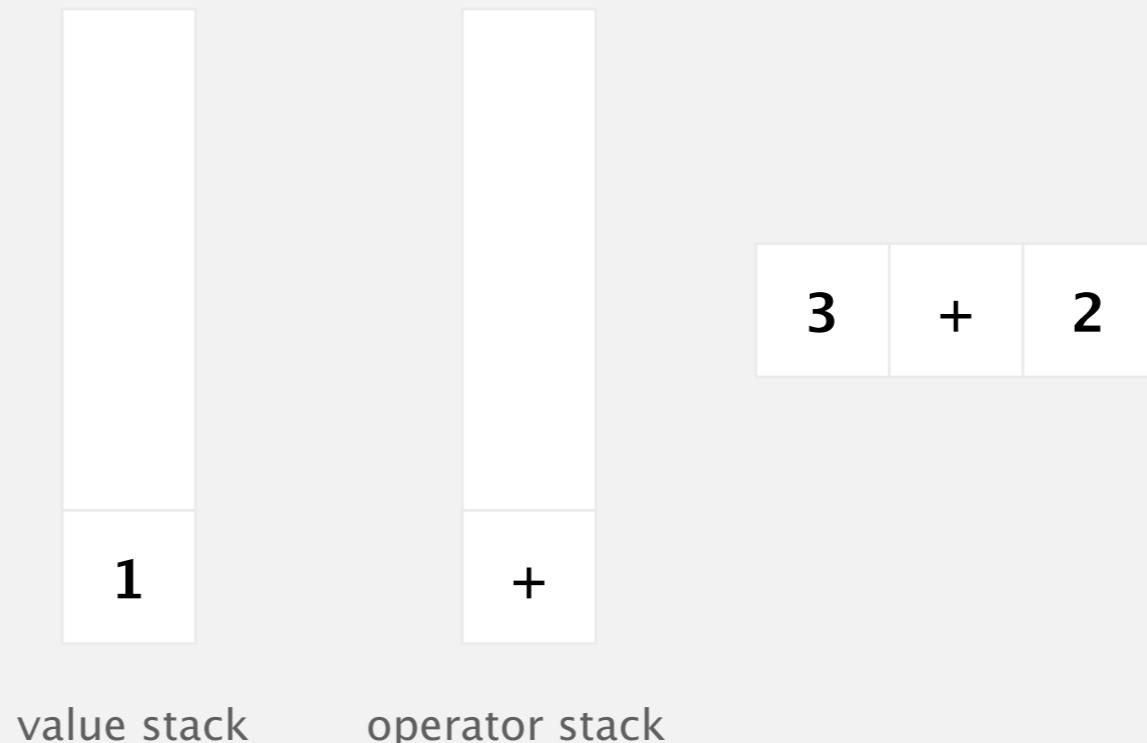
Dijkstra's two-stack algorithm

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.



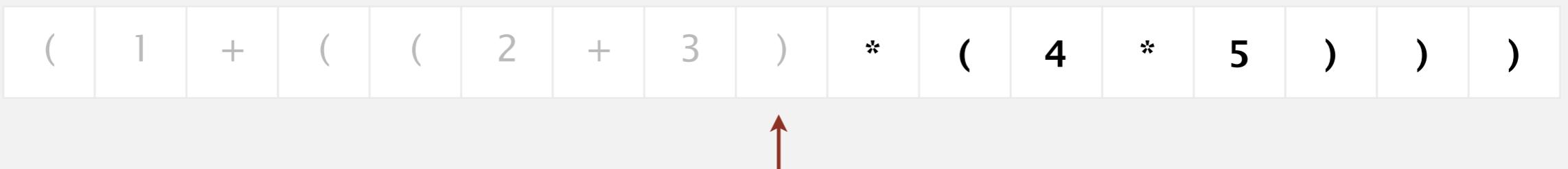
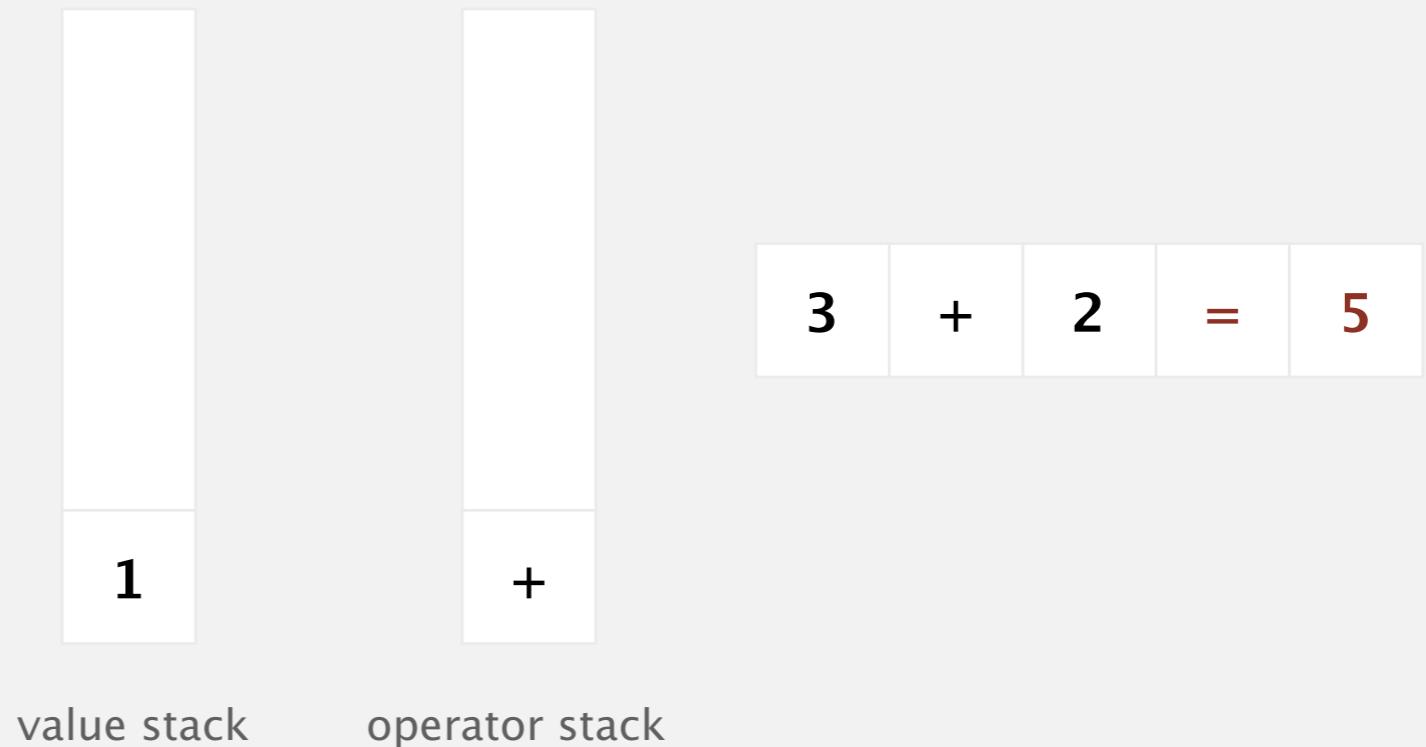
Dijkstra's two-stack algorithm

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.



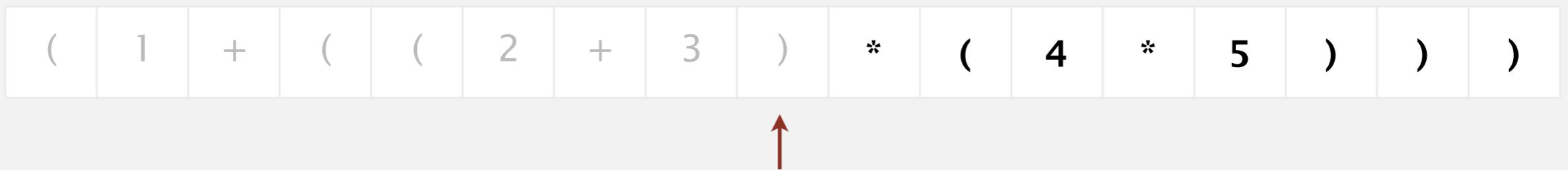
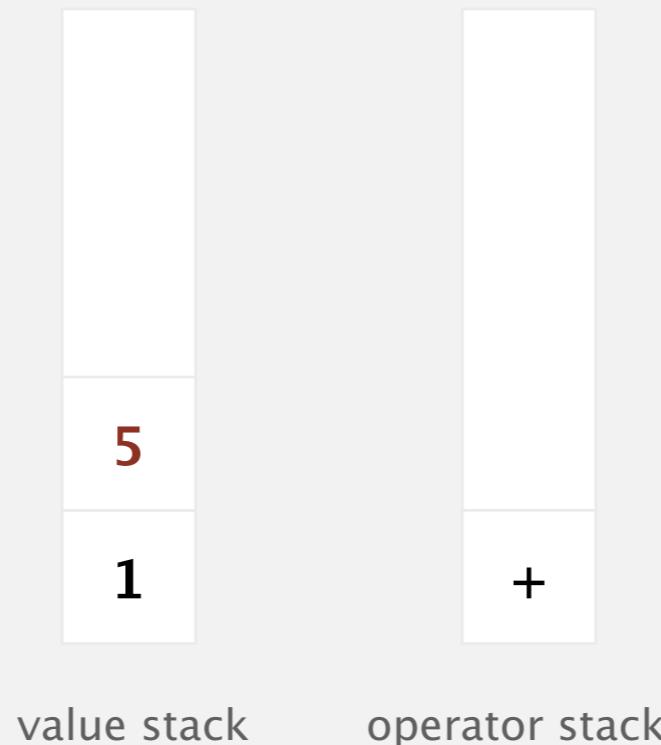
Dijkstra's two-stack algorithm

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.



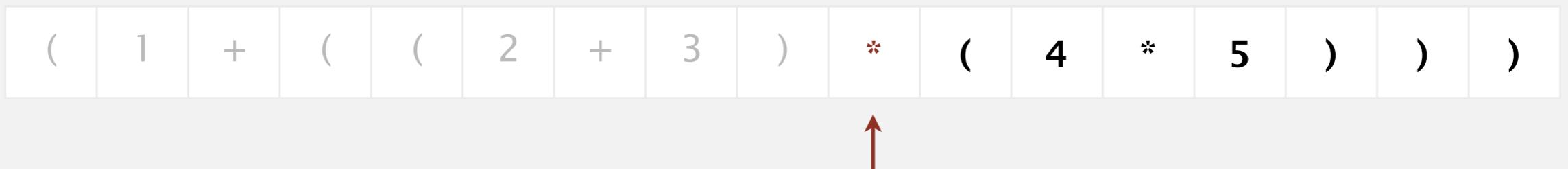
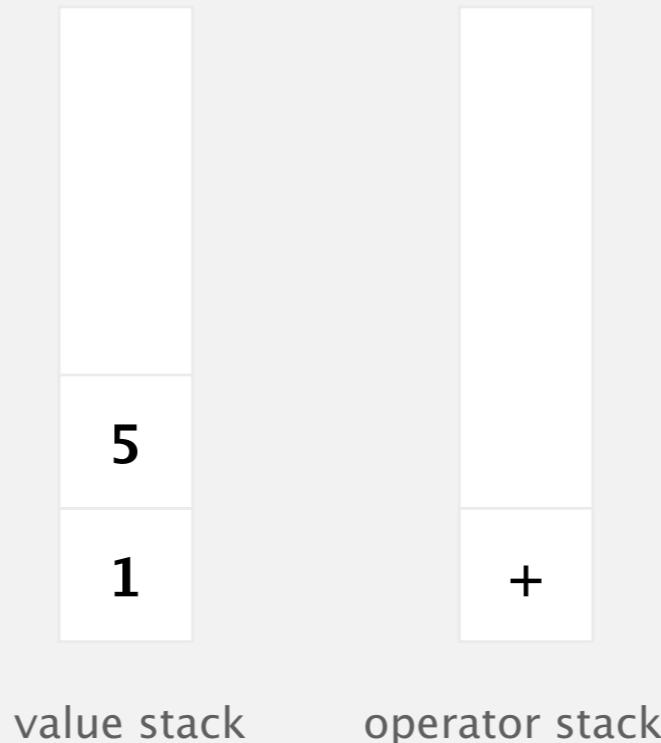
Dijkstra's two-stack algorithm

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.



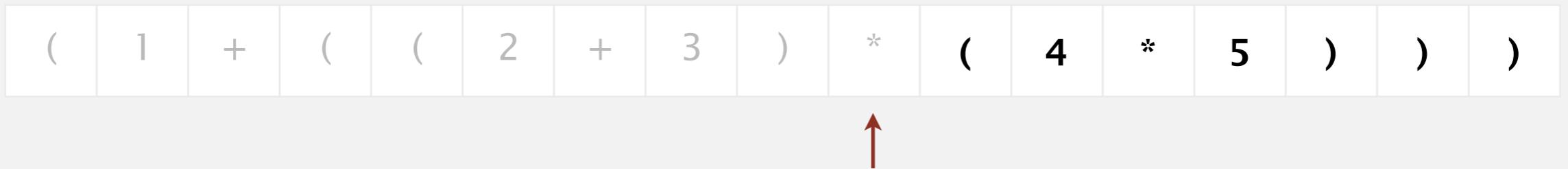
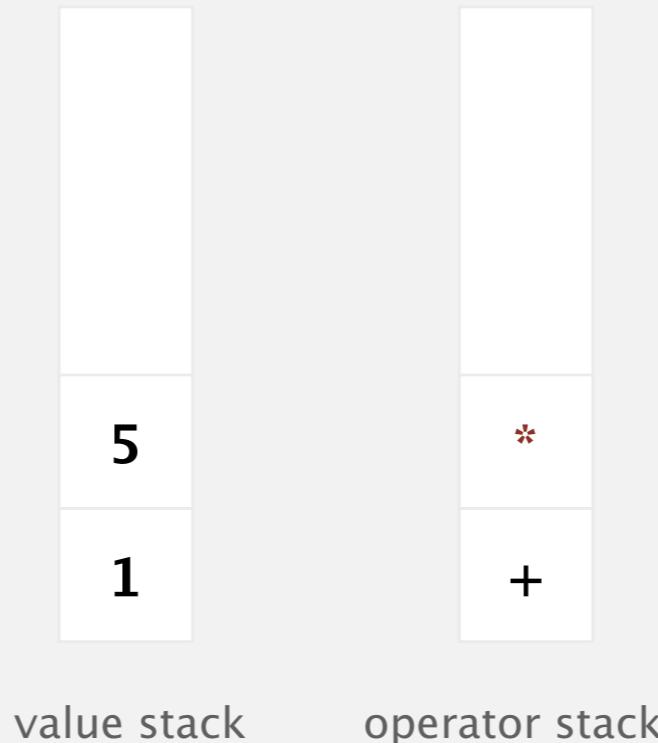
Dijkstra's two-stack algorithm

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.



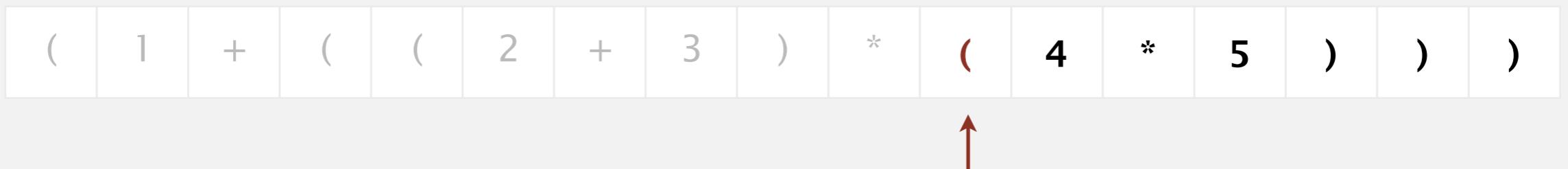
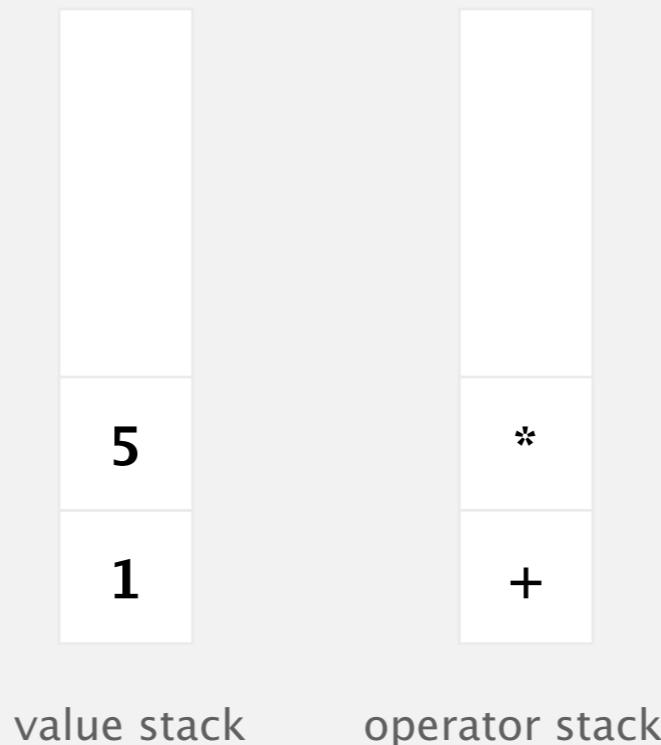
Dijkstra's two-stack algorithm

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.



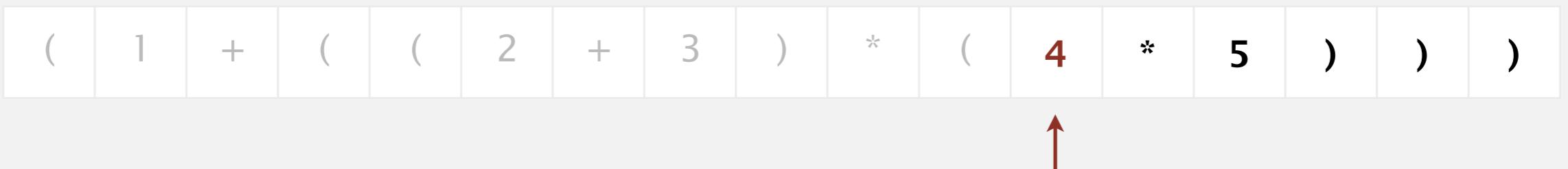
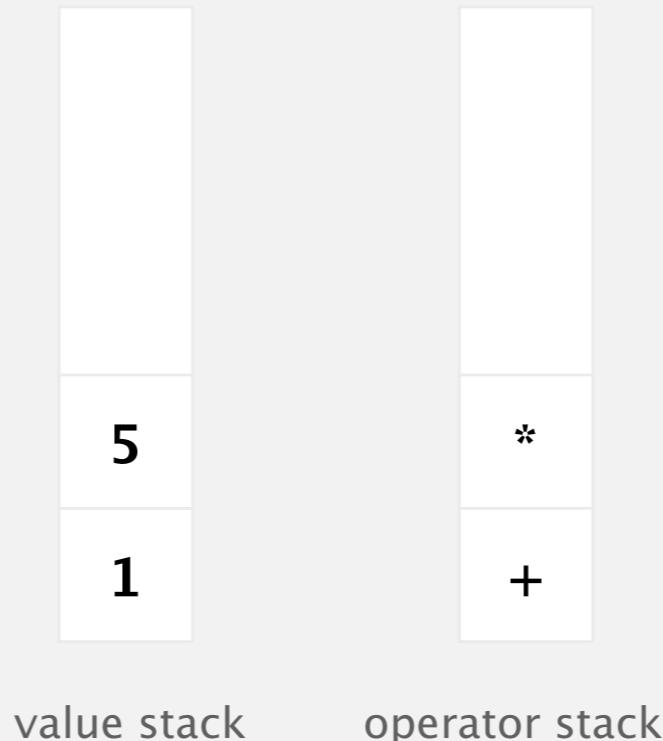
Dijkstra's two-stack algorithm

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.



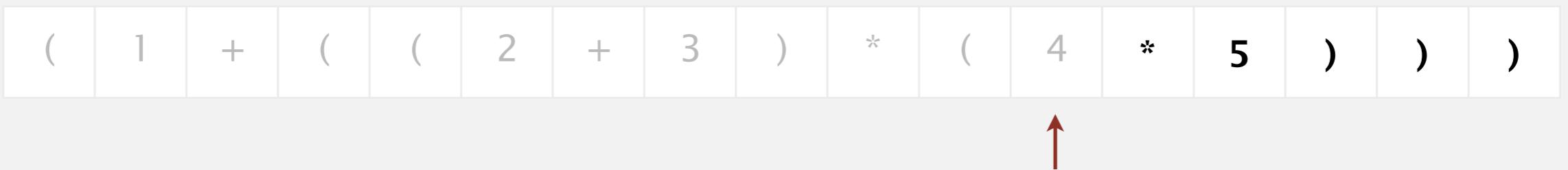
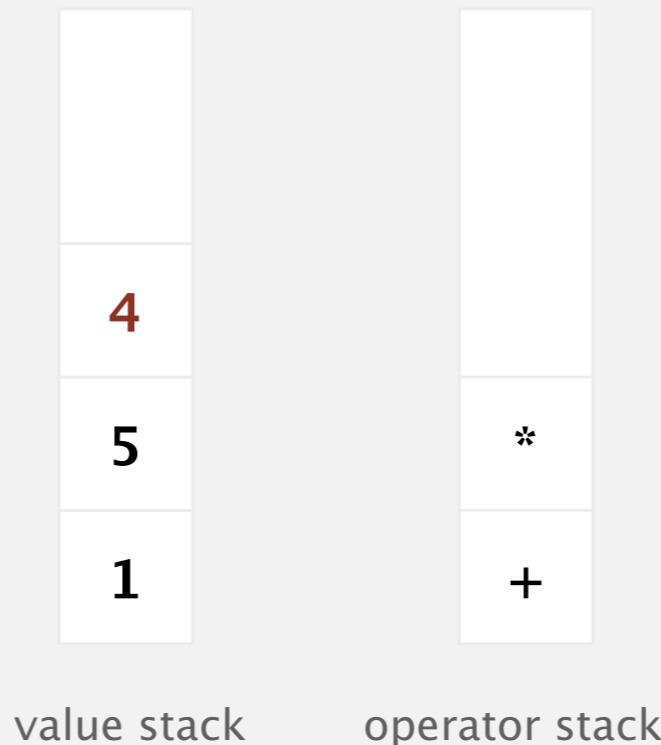
Dijkstra's two-stack algorithm

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.



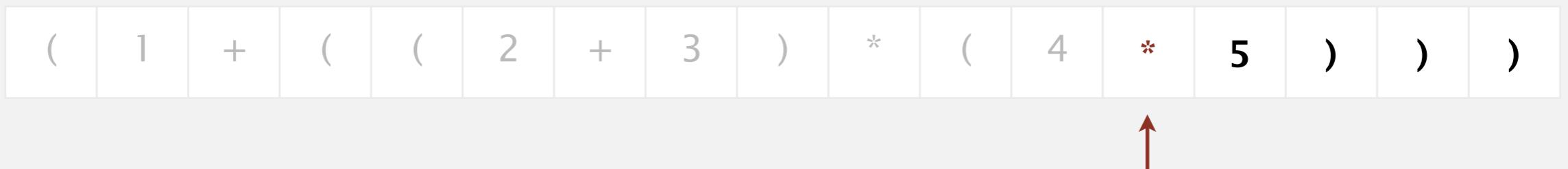
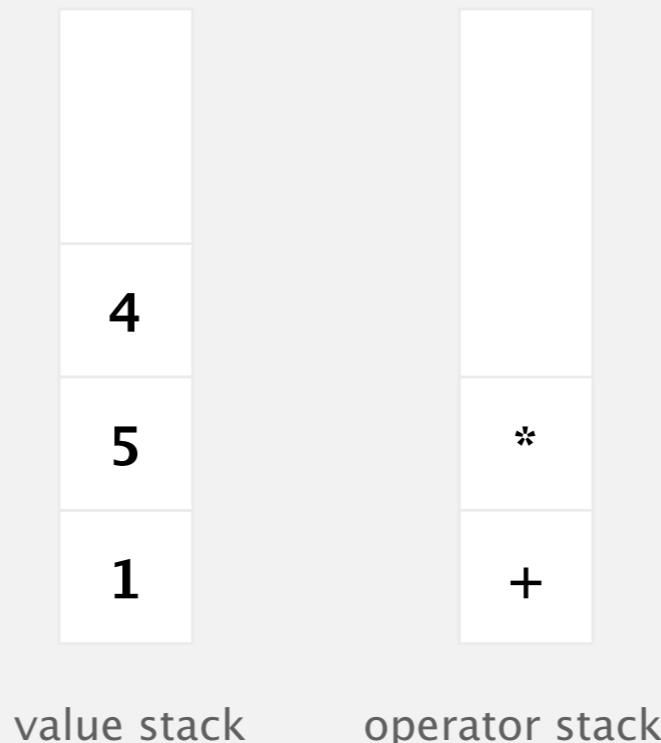
Dijkstra's two-stack algorithm

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.



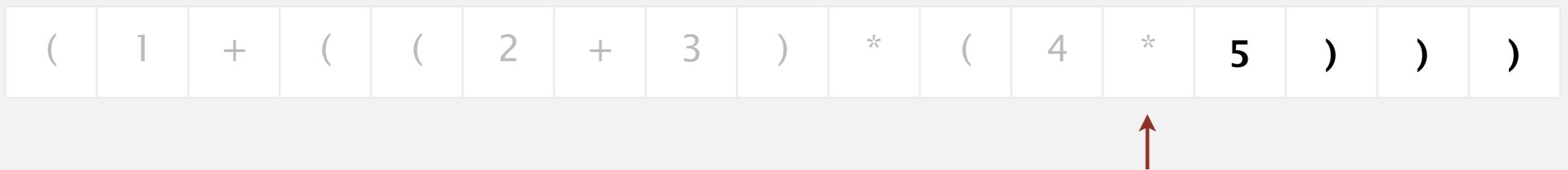
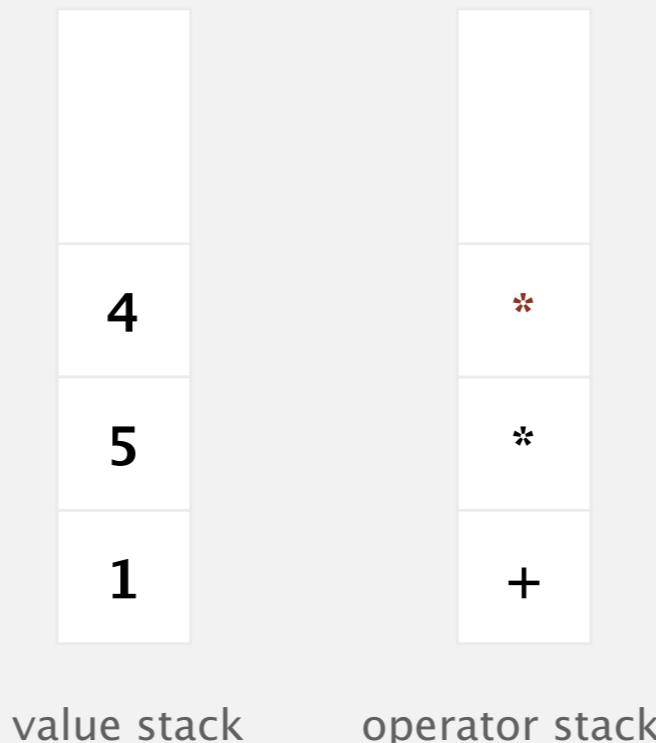
Dijkstra's two-stack algorithm

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.



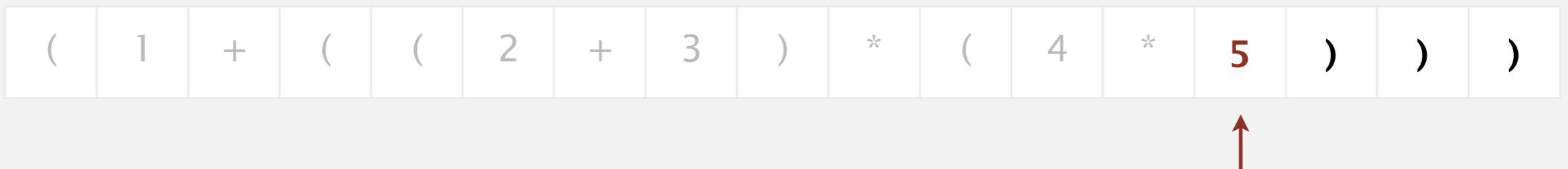
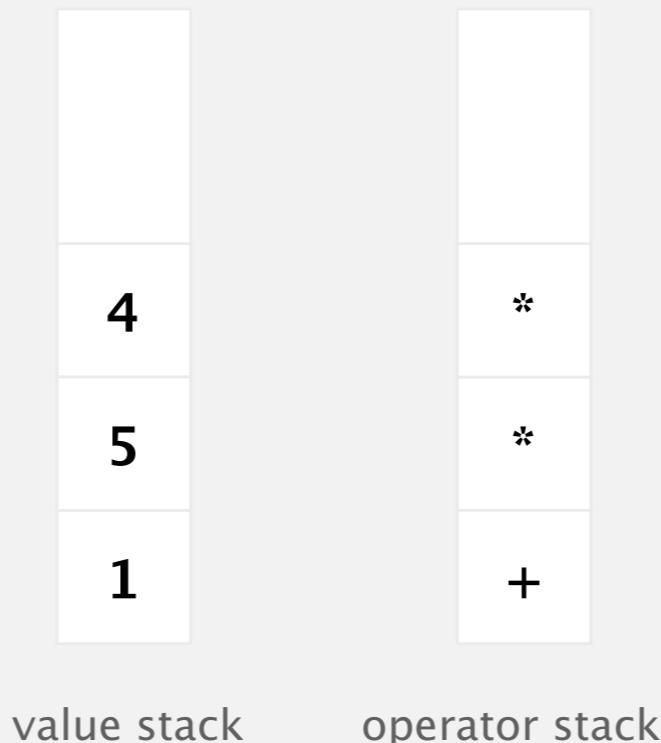
Dijkstra's two-stack algorithm

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.



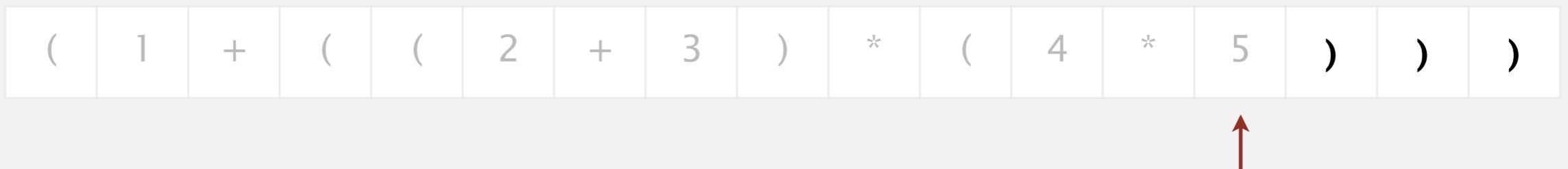
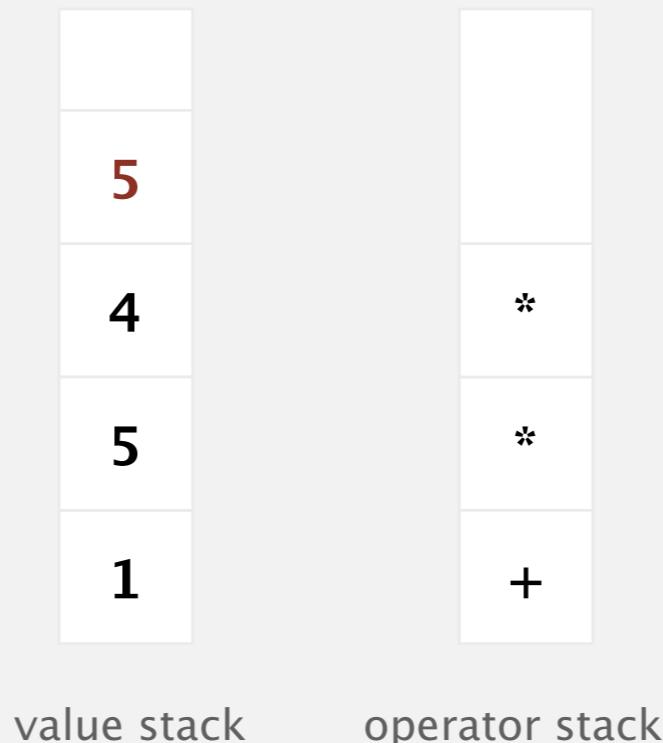
Dijkstra's two-stack algorithm

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.



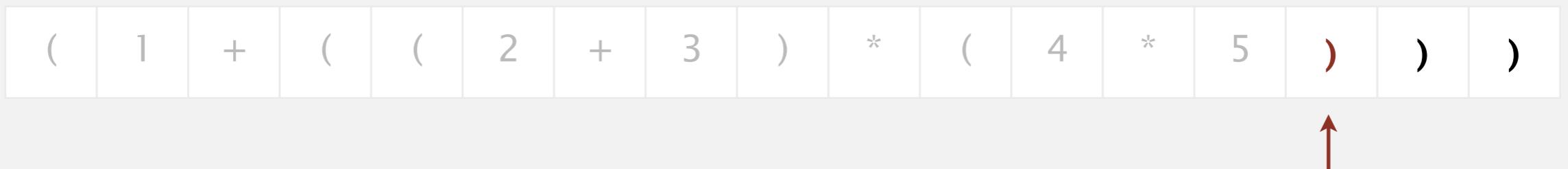
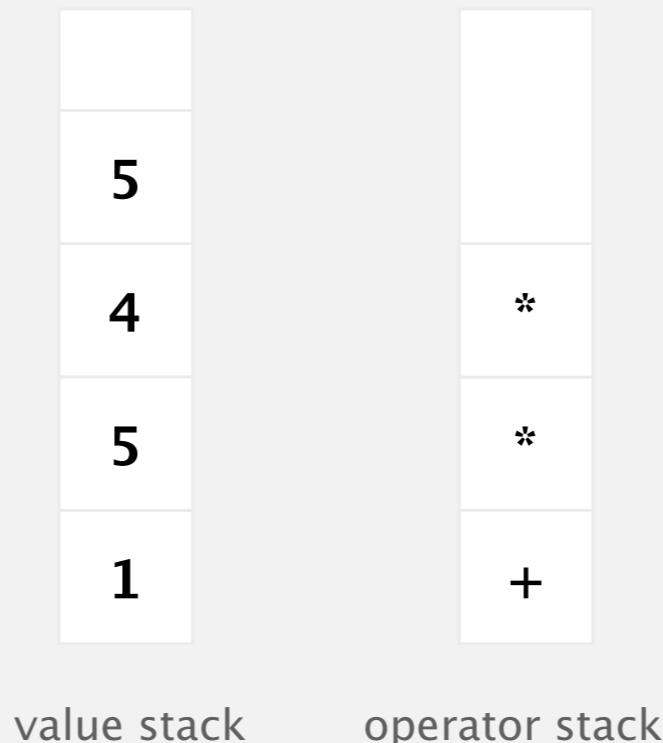
Dijkstra's two-stack algorithm

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.



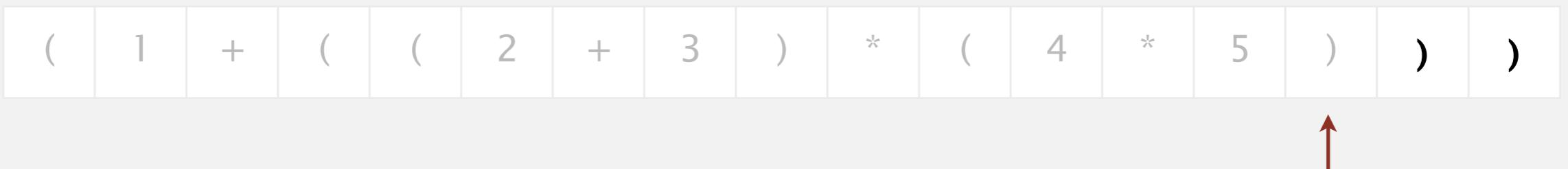
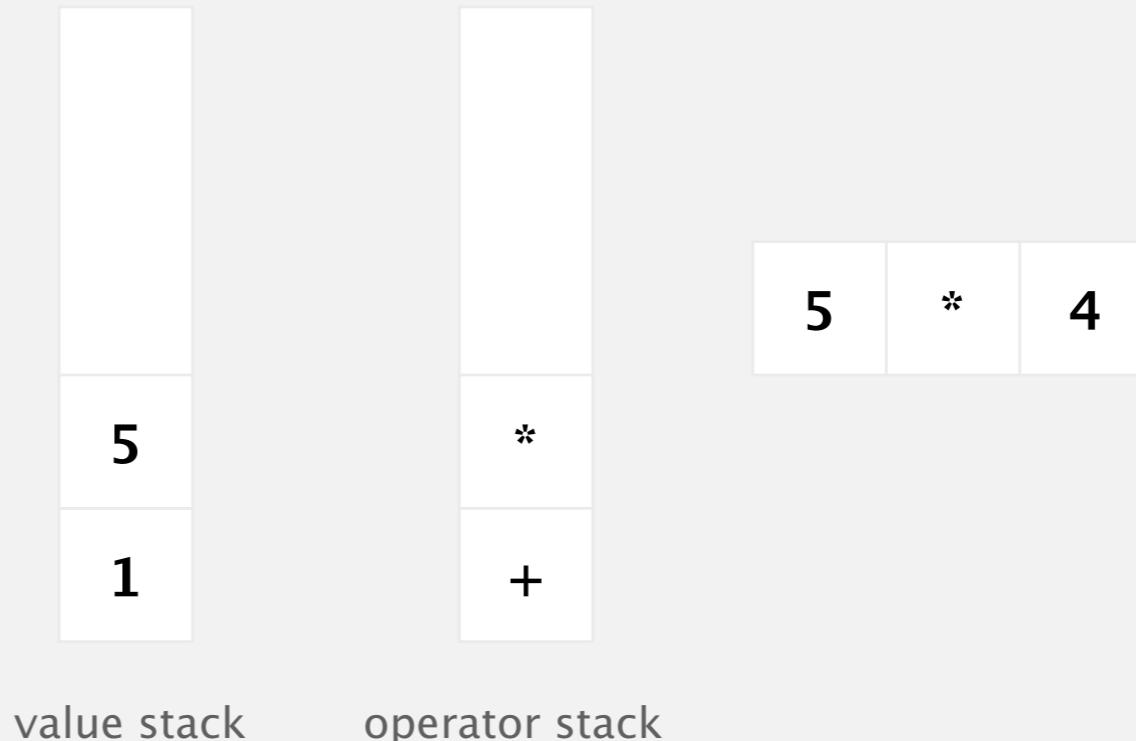
Dijkstra's two-stack algorithm

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.



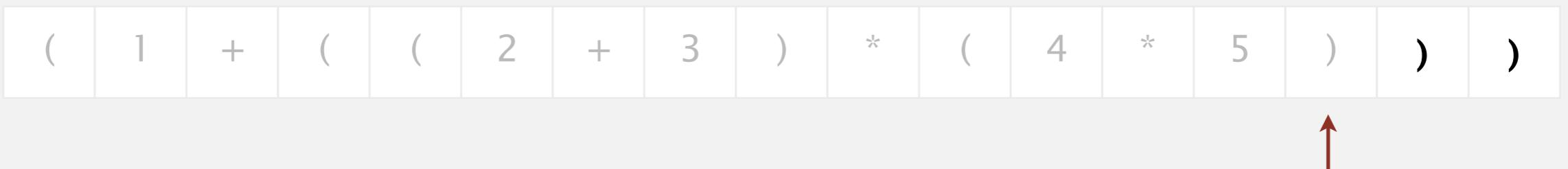
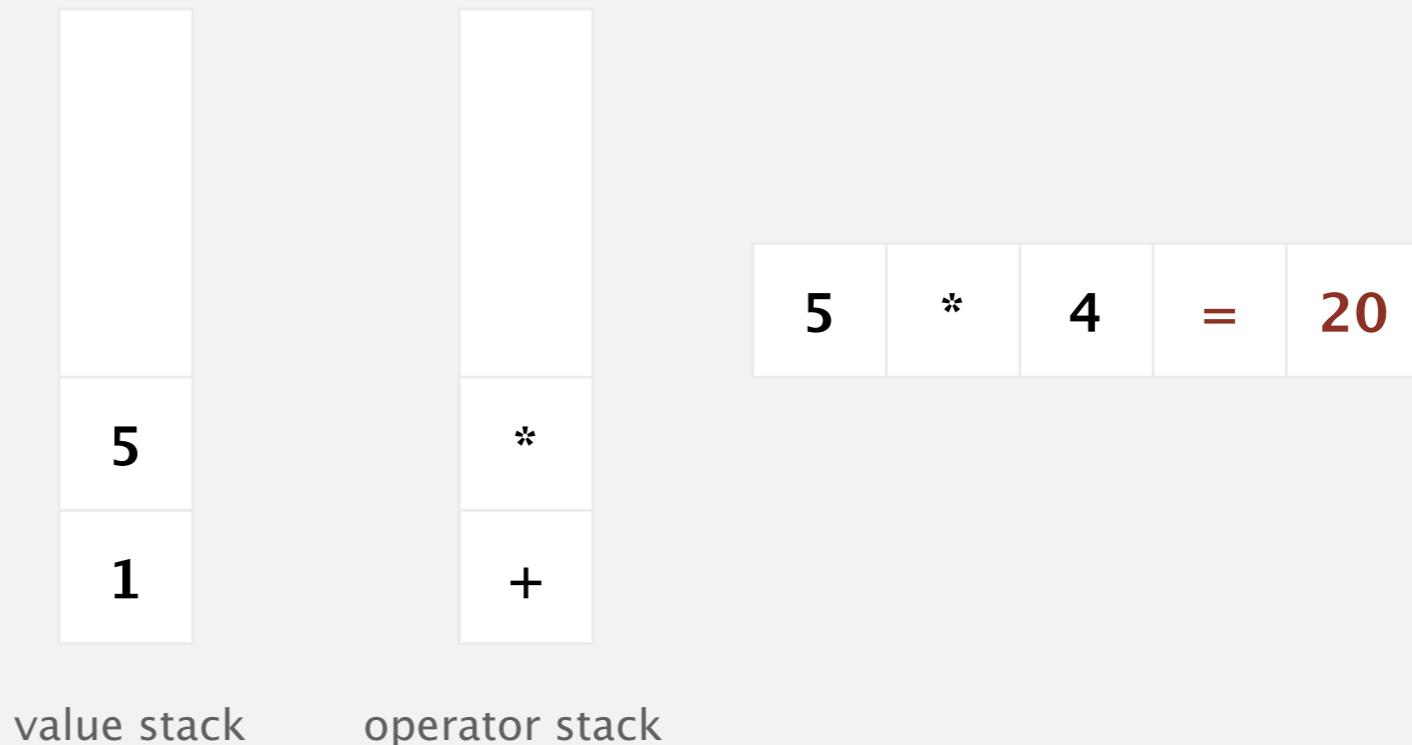
Dijkstra's two-stack algorithm

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.



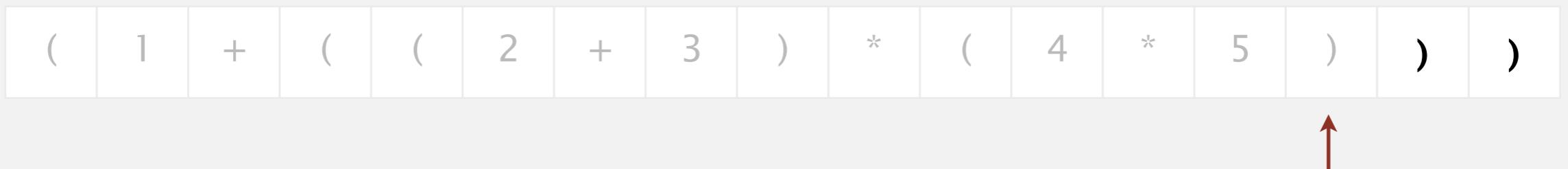
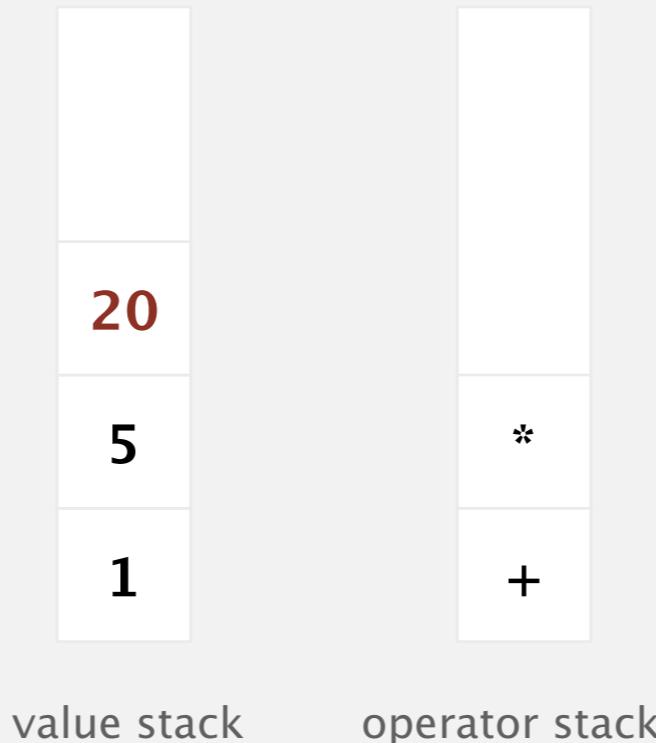
Dijkstra's two-stack algorithm

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.



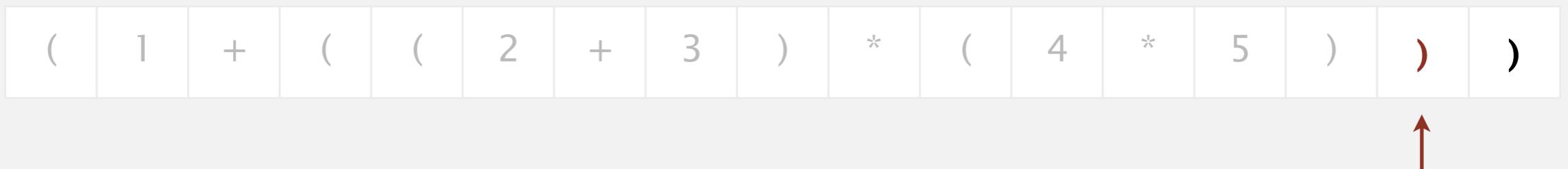
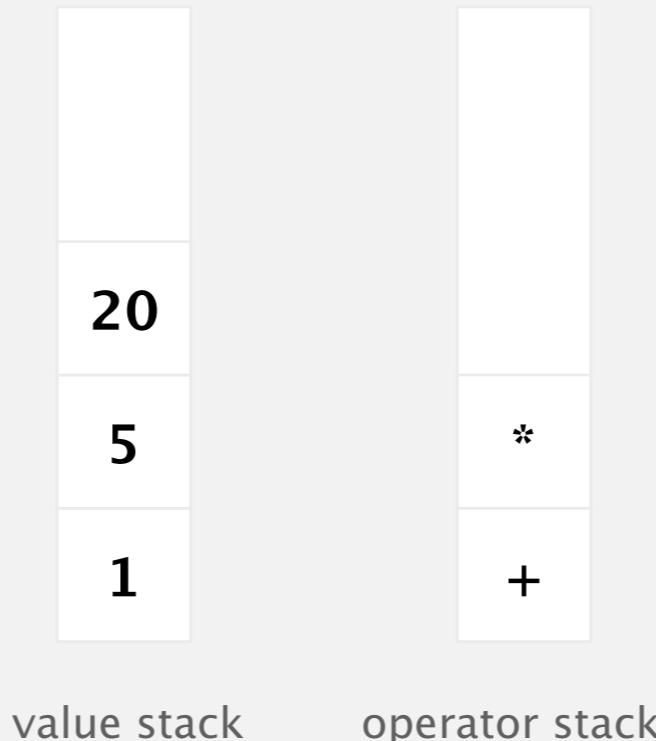
Dijkstra's two-stack algorithm

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.



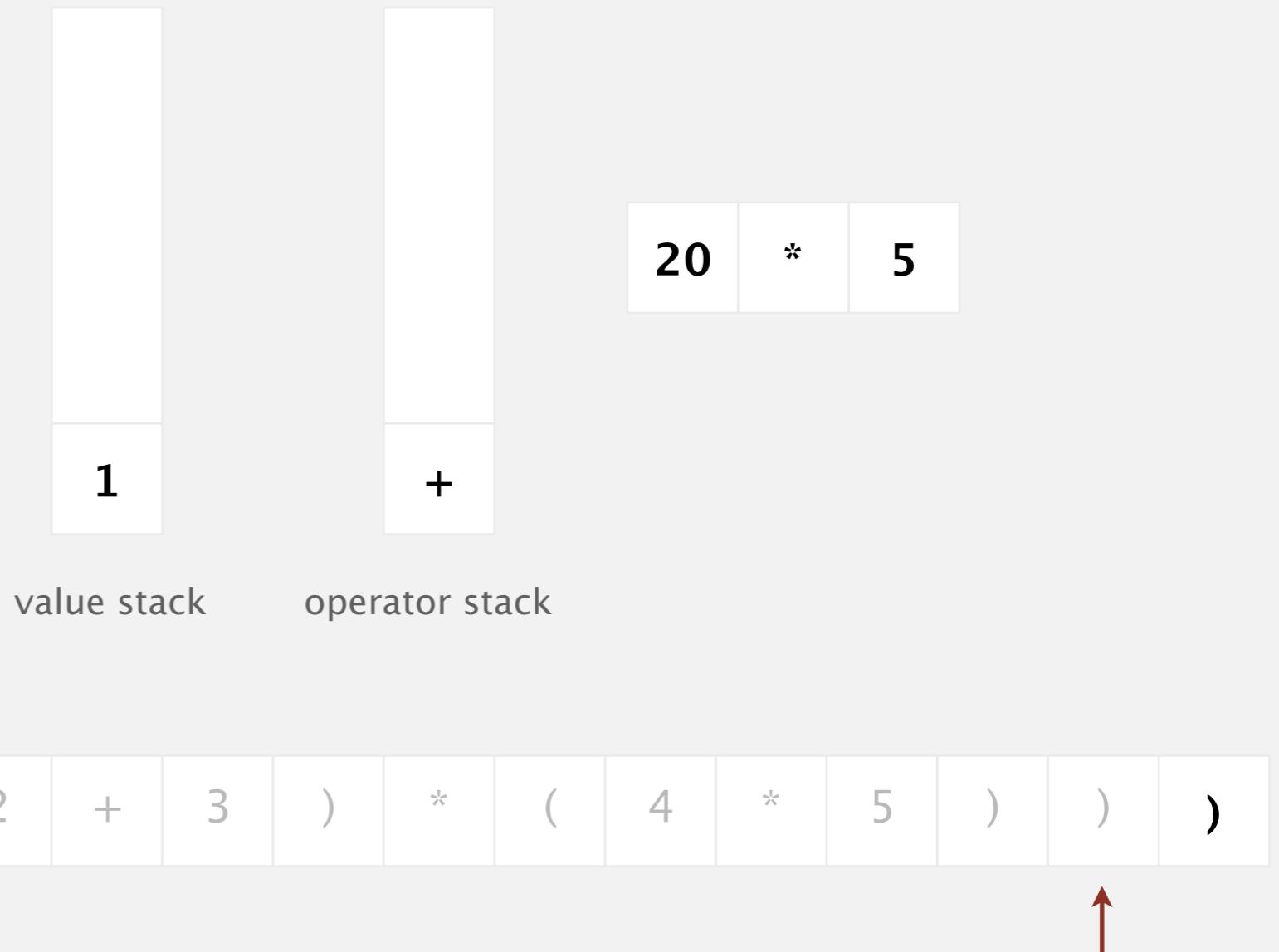
Dijkstra's two-stack algorithm

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.



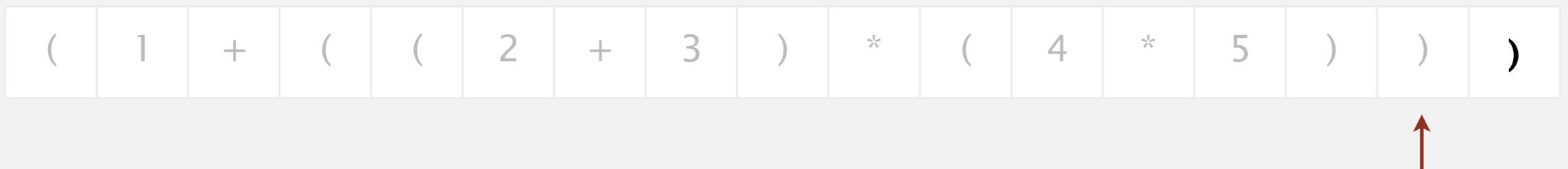
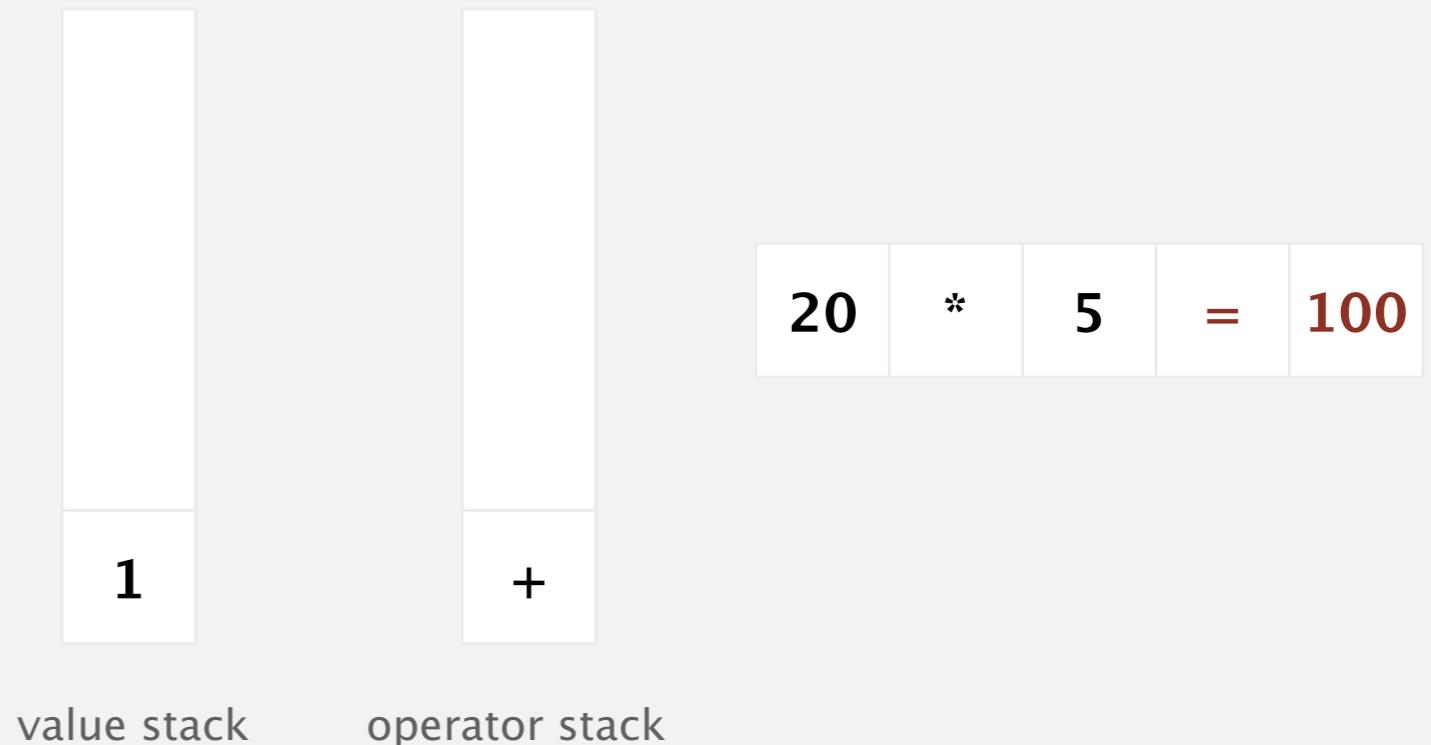
Dijkstra's two-stack algorithm

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.



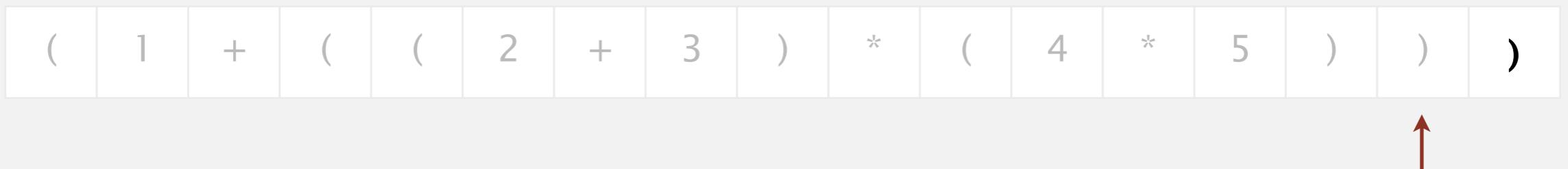
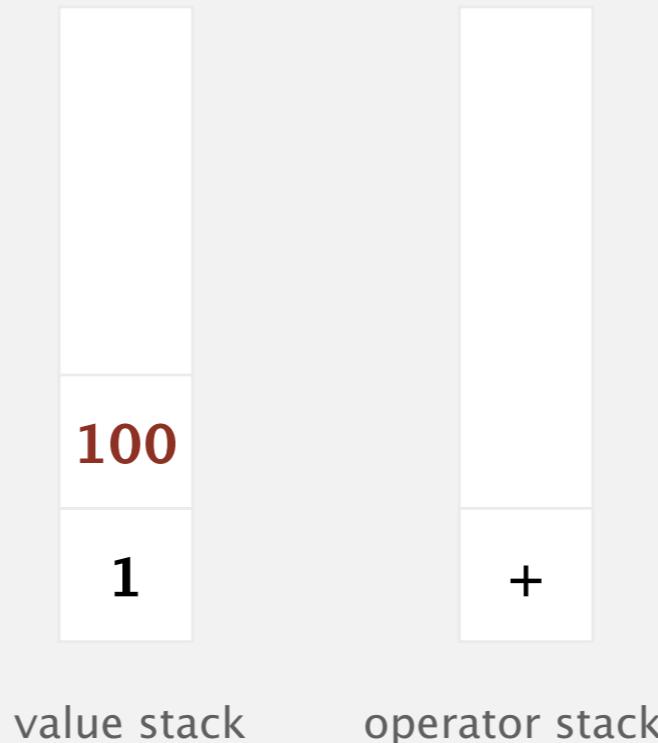
Dijkstra's two-stack algorithm

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.



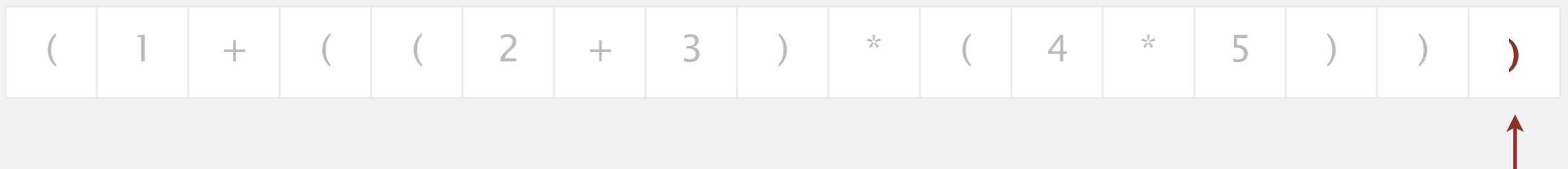
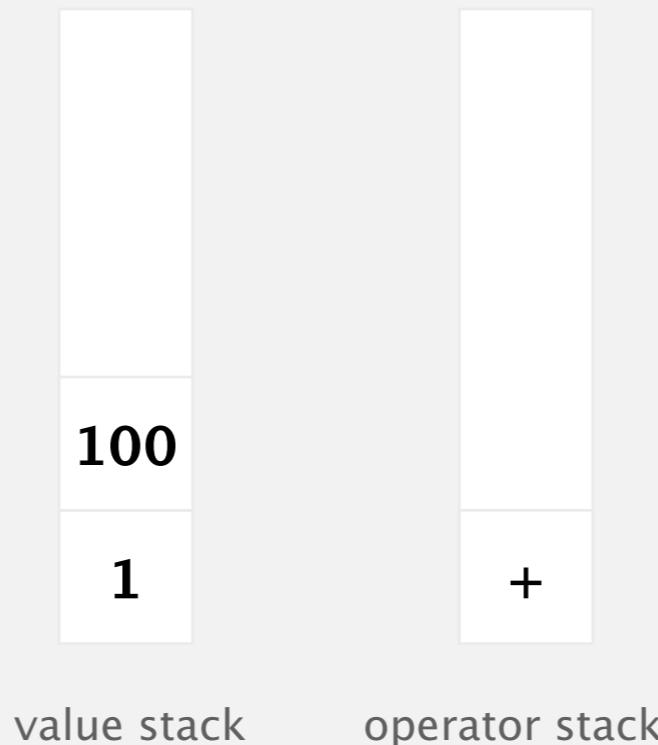
Dijkstra's two-stack algorithm

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.



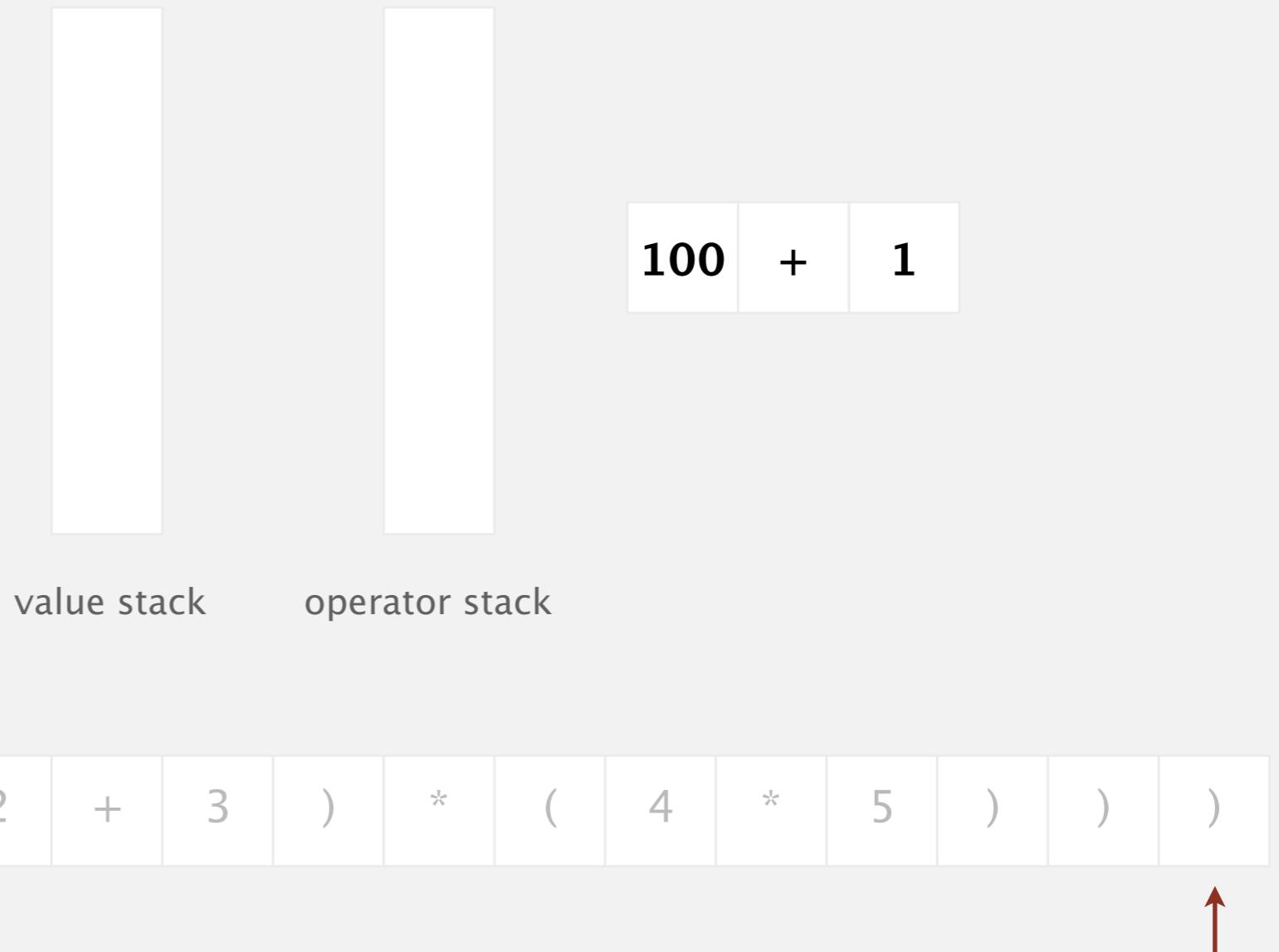
Dijkstra's two-stack algorithm

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.



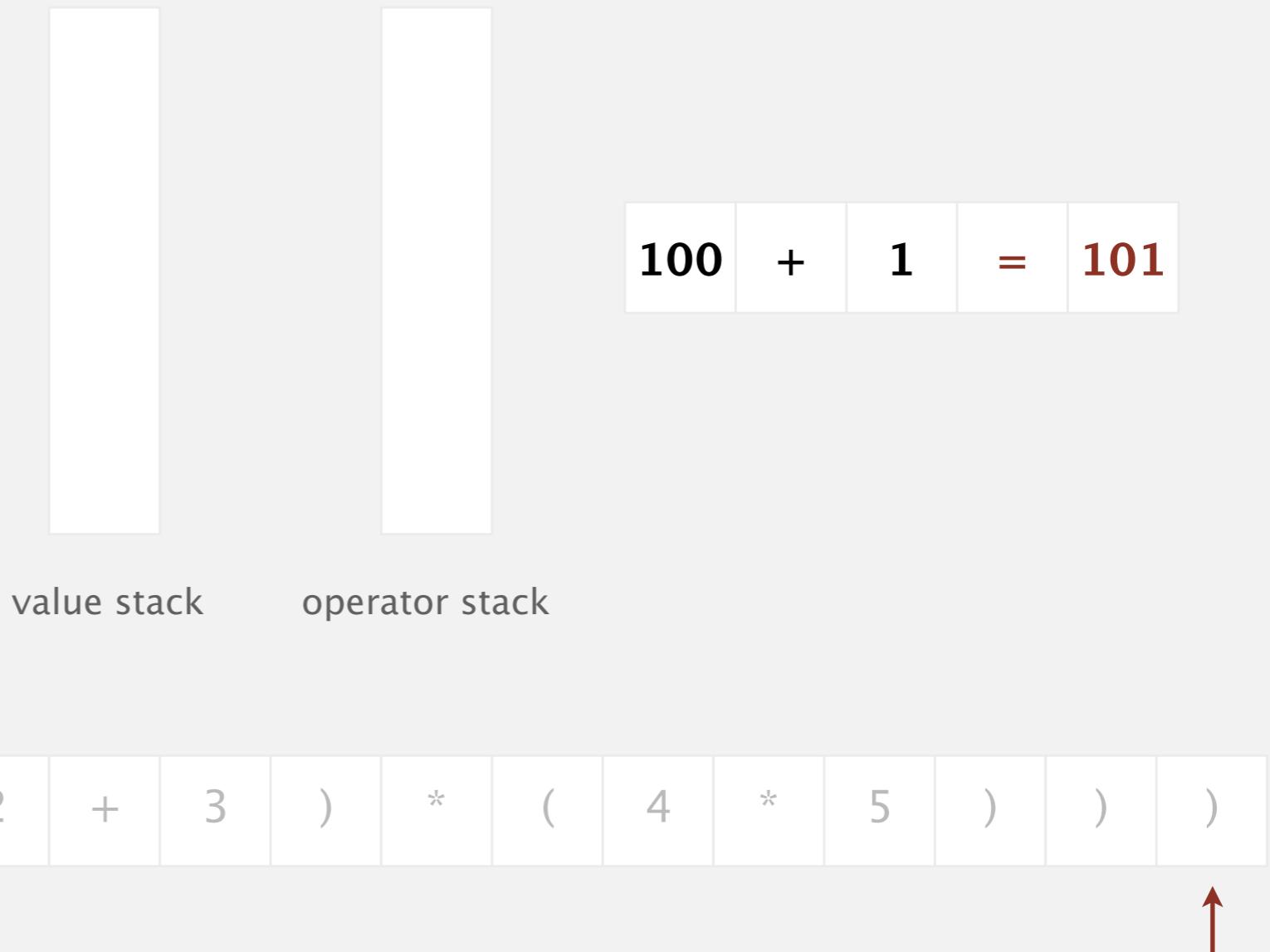
Dijkstra's two-stack algorithm

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.



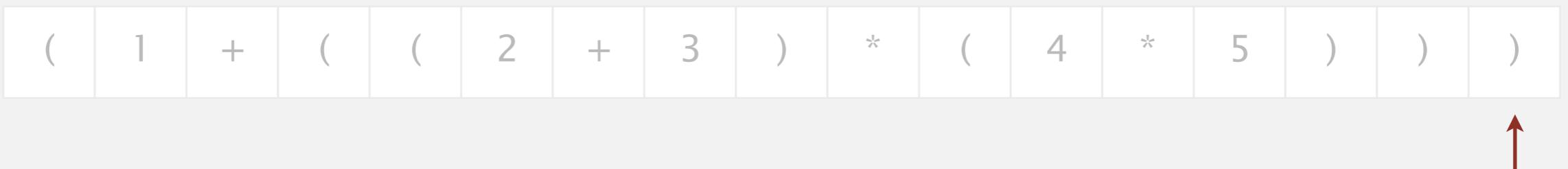
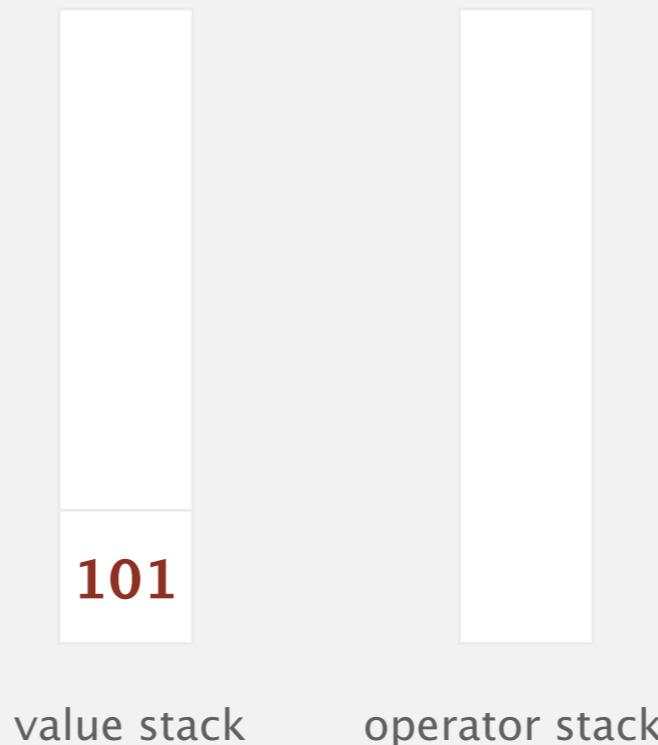
Dijkstra's two-stack algorithm

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.



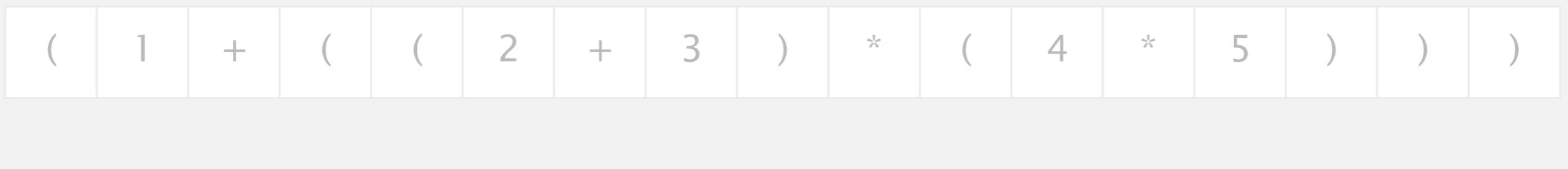
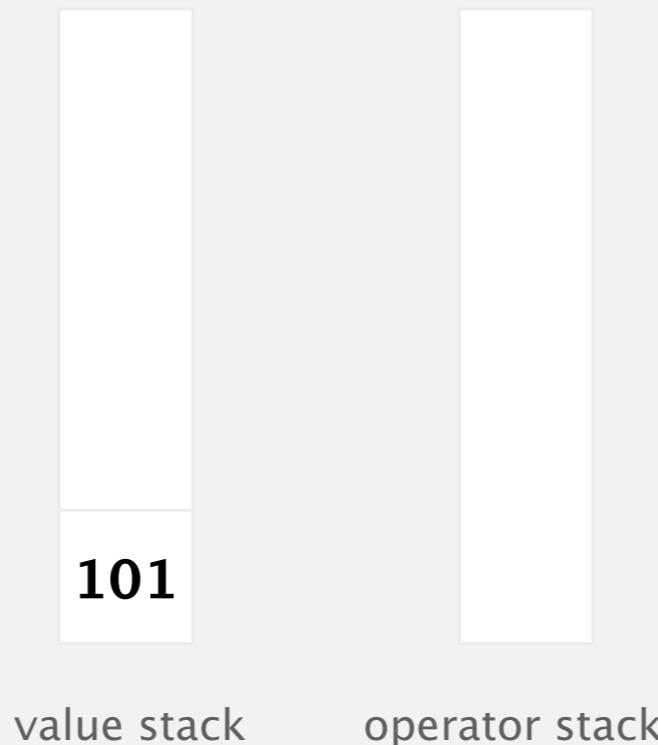
Dijkstra's two-stack algorithm

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.



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.

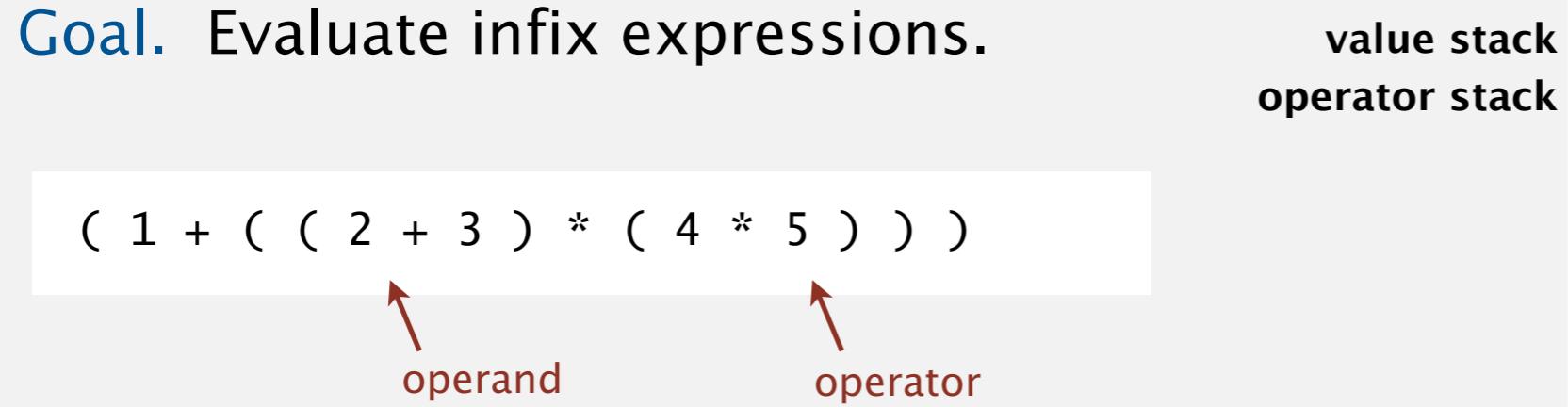
101

result

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

Arithmetic expression evaluation

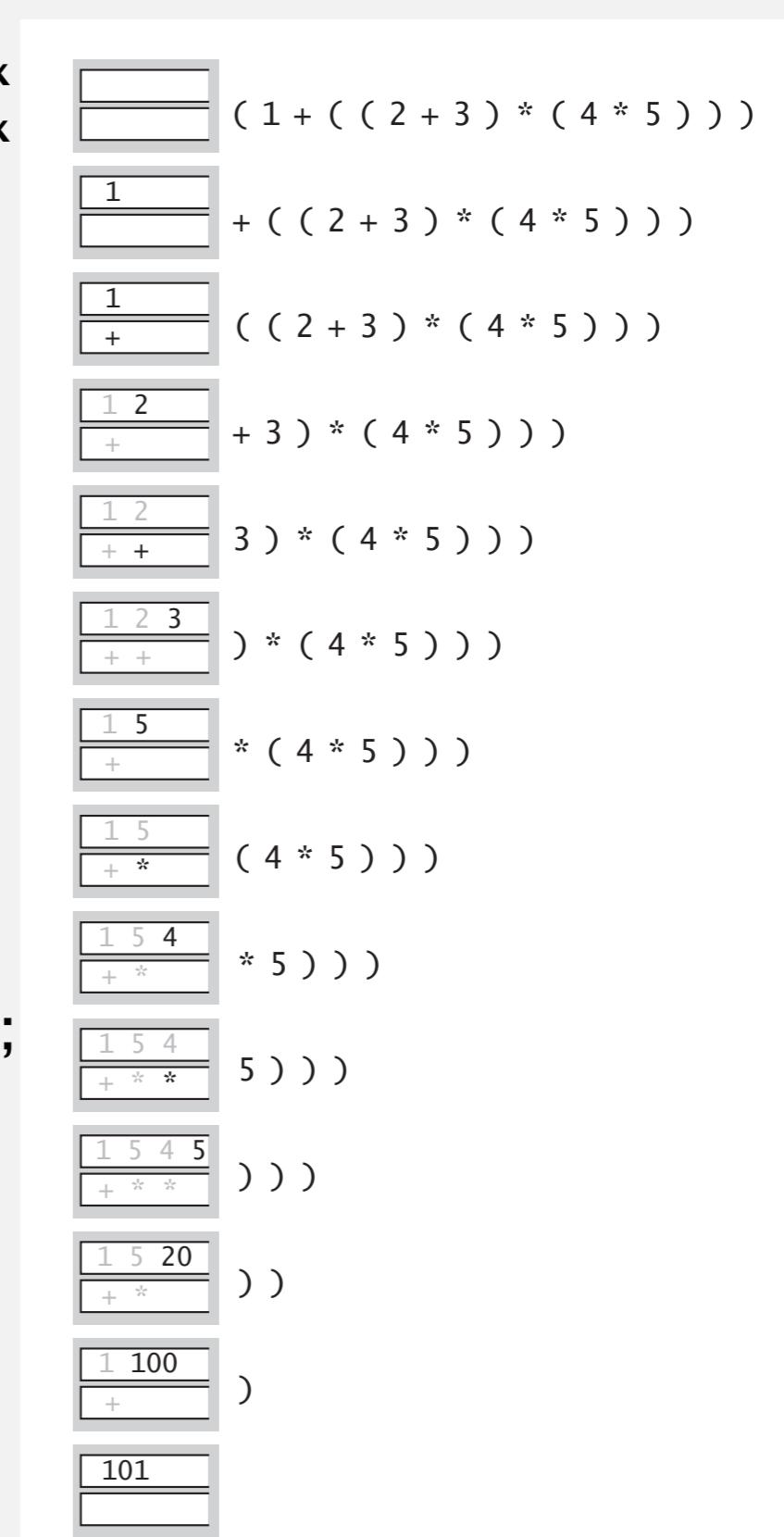
Goal. Evaluate infix expressions.



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

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

Context. An interpreter!



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("(")) /* noop */;
            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, associativity.

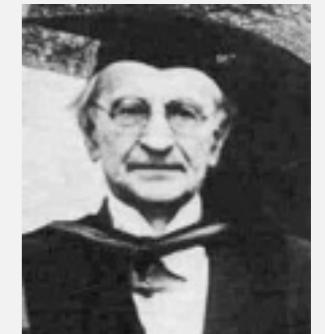
Stack-based programming languages

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

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

Observation 2. All of the parentheses are redundant!

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



Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, ...