CIS 121, Fall 2016

Data Structures and Algorithms

aka Programming Languages & Techniques II

Chris Callison-Burch

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

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, ...

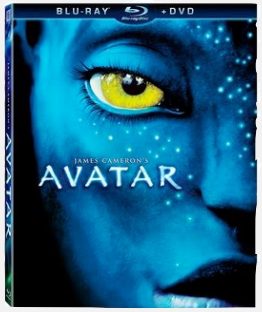
Security. Cell phones, e-commerce, voting machines, ...

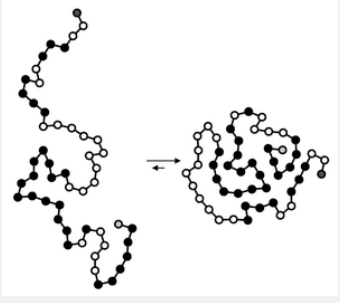
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

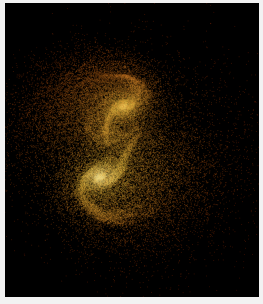
Social networks. Recommendations, news feeds, advertisements, ...

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

⋮







Why study algorithms?

Their impact is broad and far-reaching.

Mysterious algorithm was 4% of trading activity last week

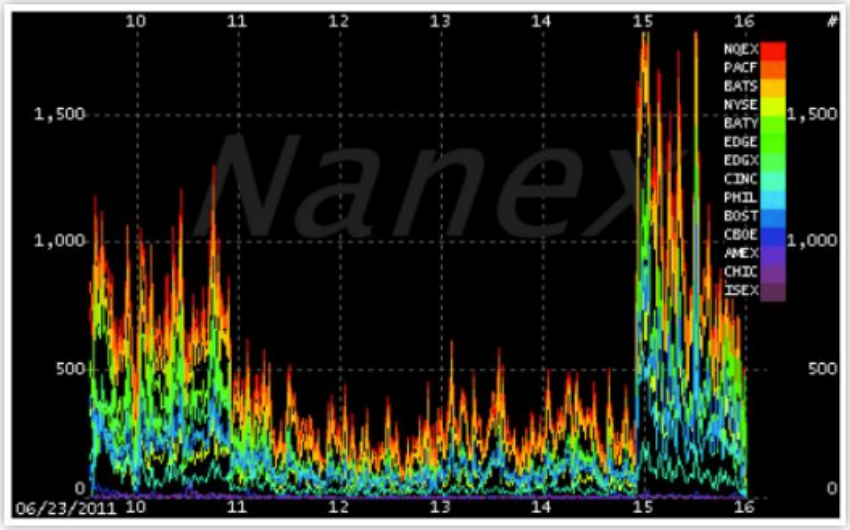
October 11, 2012

A single mysterious computer program that placed orders — and then subsequently canceled them — made up 4 percent of all quote traffic in the U.S. stock market last week, according to the top tracker of high-frequency trading activity.

The motive of the algorithm is still unclear, *CNBC* reports.

The program placed orders in 25-millisecond bursts involving about 500 stocks, according to Nanex, a market data firm. The algorithm never executed a single trade, and it abruptly ended at about 10:30 a.m. ET Friday.

[+]



Generic high frequency rrading chart (credit: Nanex)

"My guess is that the algo was testing the market, as high-frequency frequently does," says Jon Najarian, co-founder of TradeMonster.com. "As soon as they add bandwidth, the HFT crowd sees how quickly they can top out to create latency." (*Read More*: Unclear What Caused Kraft Spike: Nanex Founder.)

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. Sumerian cuneiform, one of the most ancient written records, contains partly of algorithm descriptions for reckoning in base 60. And I suppose we could claim that the Druid algorithm for estimating the start of summer is embodied in Stonehenge. (That's really hard hardware!)

Like so many other things that technology affects, algorithms have advanced in startling and unexpected ways in the 20th century—at least it looks that way to us now. The algorithms we chose for this issue have been essential for progress in communication, health care, manufacturing, economics, weather prediction, defense, and fundamental science. Conversely, progress in these areas has stimulated the search for ever-better algorithms. I recall one late-night ball session on the Maryland Shore when someone asked, "Who first ate a crab? After all, they don't look very appetizing." After the usual speculation about the observed behavior of sea gulls, someone grew what must be the right answer—namely, "A very hungry person first ate a crab."

The flip side to "necessity is the mother of invention" is "invention creates its own necessity." Our need for powerful machines always exceeds their availability. Each significant computation brings insights that suggest the next, usually much larger, computation to be done. New algorithms are an attempt to bridge the gap between the demand for cycles and the available supply of them. We've become accustomed to gaining the Moore's Law factor of two every 18 months. In effect, Moore's Law changes the constant in front of the estimate of running time as a function of problem size. Important new algorithms do not come along every 1.5 years, but when they do, they can change the exponent of the complexity!

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. A colleague recently claimed that he'd done only 15 minutes of productive work in his whole life. He wasn't joking, because he was referring to the 15 minutes during which he'd sketched out a fundamental optimization algorithm. He regarded the previous years of thought and investigation as a sunk cost that might or might not have paid off.

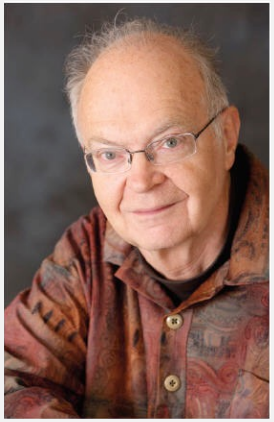
Researchers have cracked many hard problems since 1 January 1900, but we are passing some even harder ones on to the next century. In spite of a lot of good work, the question of how to extract information from extremely large masses of data is still almost untouched. There are still very big challenges coming from more "traditional" tasks, too. For example, we need efficient methods to tell when the result of a large floating-point calculation is likely to be correct. Think of the way that check sums function. The added computational cost is very small, but the added confidence in the answer is large. Is there an analog for things such as huge, multidisciplinary optimizations? At an even deeper level is the issue of reasonable methods for solving specific cases of "impossible" problems. Instances of NP-complete problems crop up in arbitrary to answer many practical questions. Are there efficient ways to attack them?

I suspect that in the 21st century, things will be ripe for another revolution in our understanding of the foundations of computational theory. Questions already arising from quantum computing and problems associated with the generation of random numbers seem to require that we somehow tie together theories of computing, logic, and the nature of the physical world.

The new century is not going to be very restful for us, but it is not going to be dull either.

2 COMPUTING IN SCIENCE & ENGINEERING

> *" An algorithm must be seen to be believed. "* — *Donald Knuth*

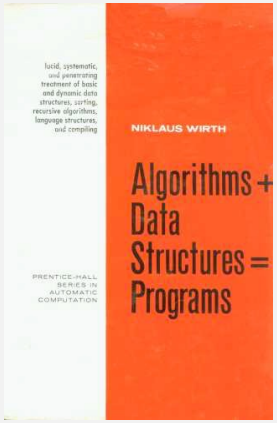


Why study algorithms?

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 programer 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?

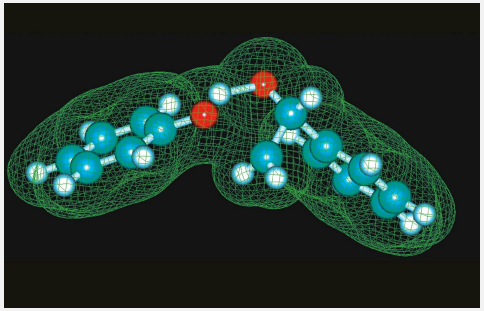
For the interview.

Why study algorithms?

They may unlock the secrets of life and of the universe.

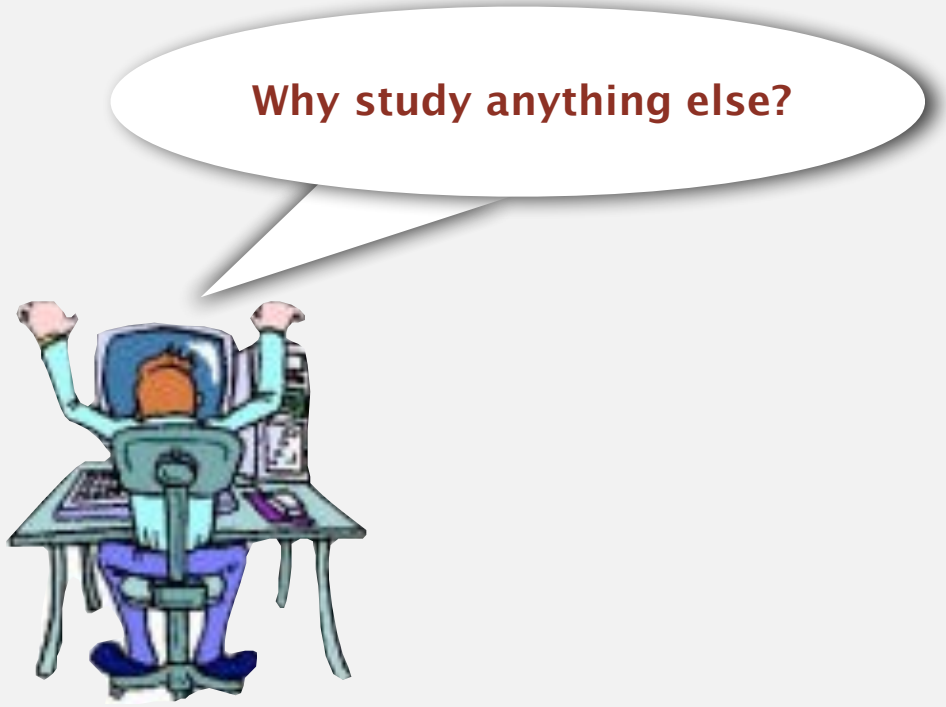
> *" Computer models mirroring real life have become crucial for most advances made in chemistry today.... Today the computer is just as important a tool for chemists as the test tube. "*
>
> — *Royal Swedish Academy of Sciences*
>
> (*Nobel Prize in Chemistry 2013*)

Martin Karplus, Michael Levitt, and Arieh Warshel



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.



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.

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

Office hours.

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

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

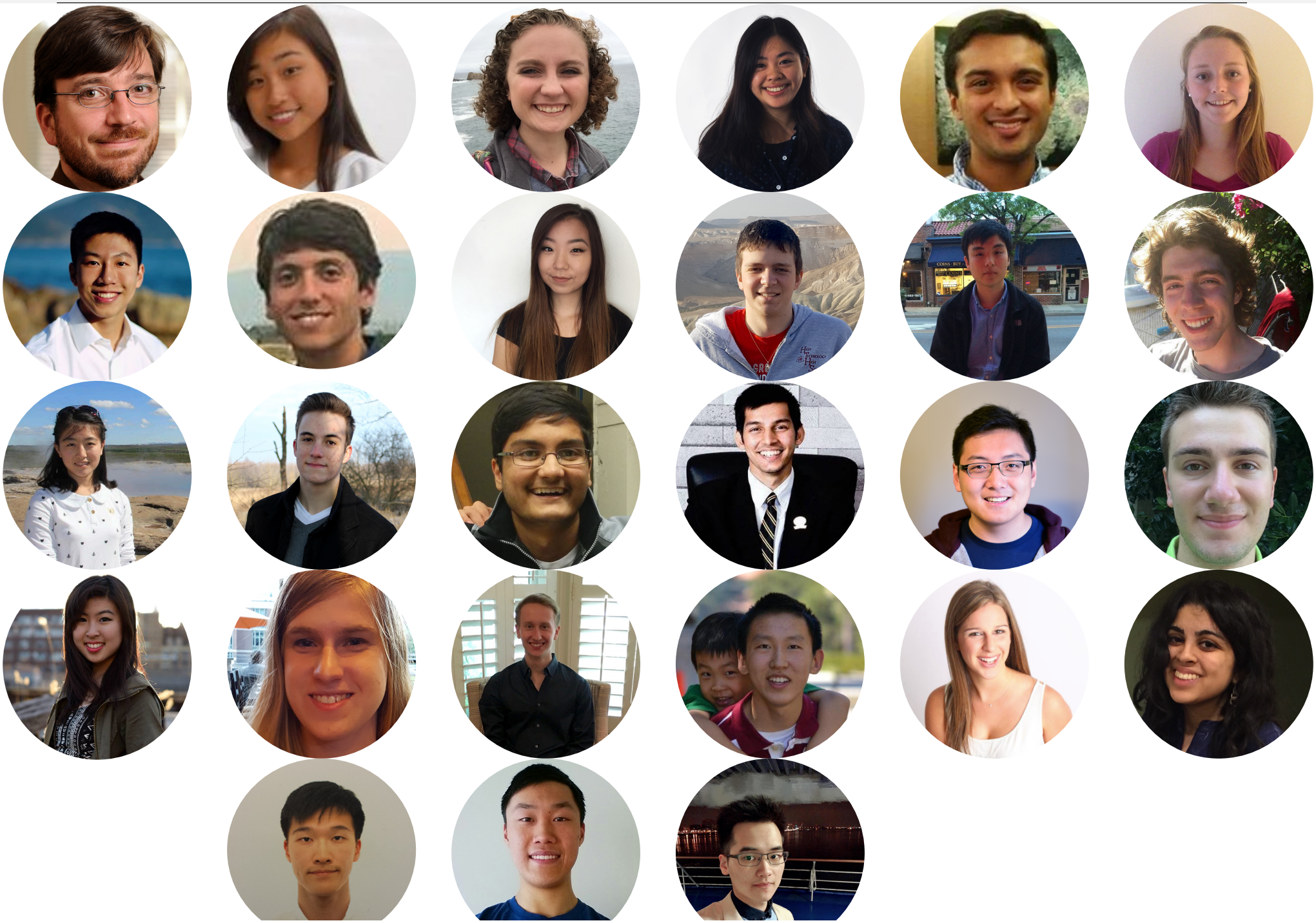
Tutoring.

- The Tutoring Center

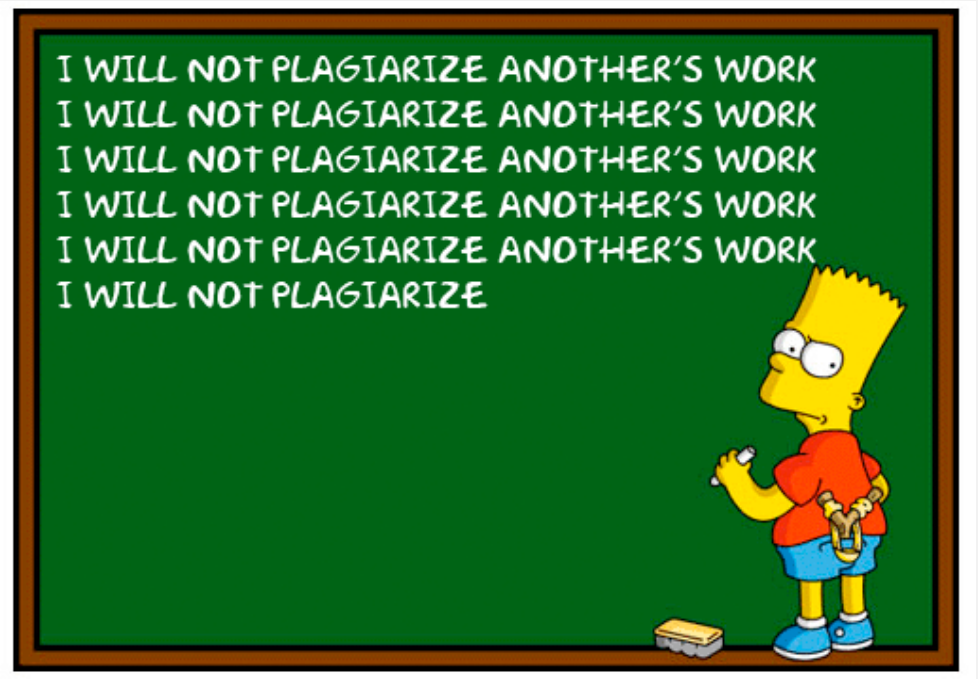
The Tutoring Center

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

Come to lectures and recitations

Read the textbook

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

http://algs4.cs.princeton.edu

1.3 STACKS AND QUEUES

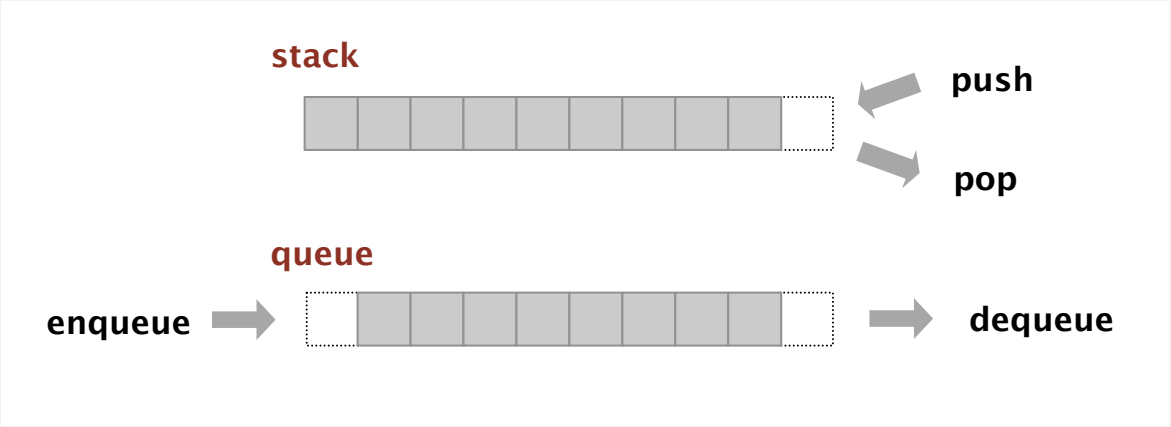
- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *applications*

Last updated on 8/27/16 3:29 PM

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?

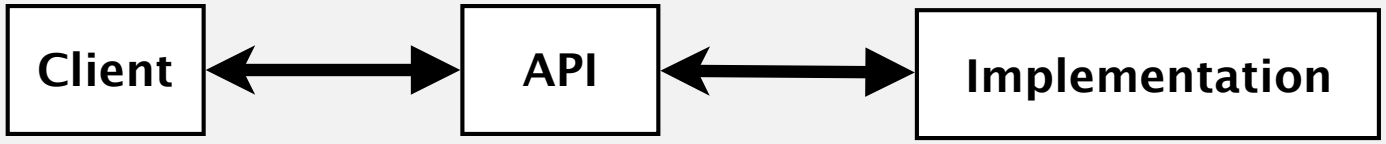


Stack. Examine the item most recently added. ← LIFO = "last in first out"

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, ...

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

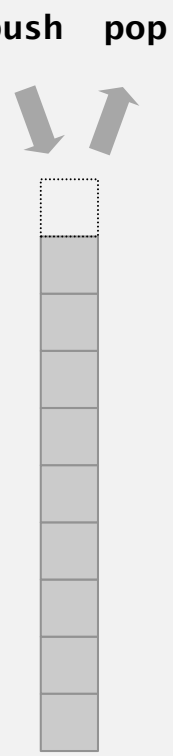
1.3 Bags, Queues, and Stacks

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

Stack API

Warmup API. Stack of strings data type.

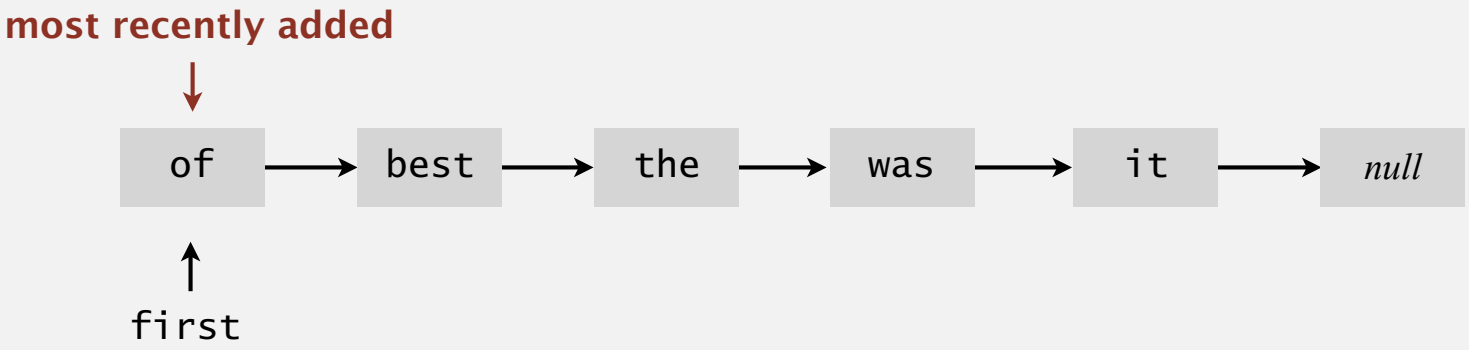
public class StackOfStrings		
	StackOfStrings()	*create an empty stack*
void	push(String item)	*add a new string to stack*
String	pop()	*remove and return the string most recently added*
boolean	isEmpty()	*is the stack empty?*
int	size()	*number of strings on the stack*



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

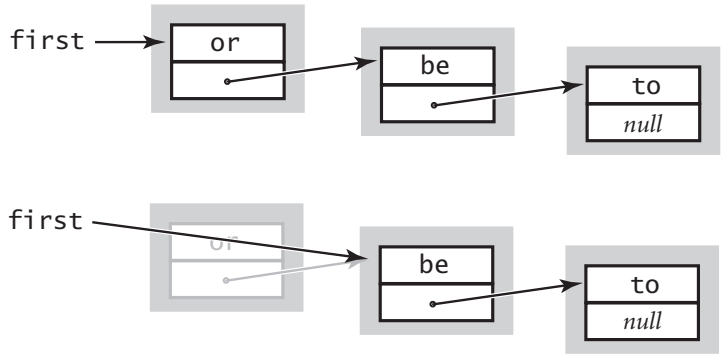
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

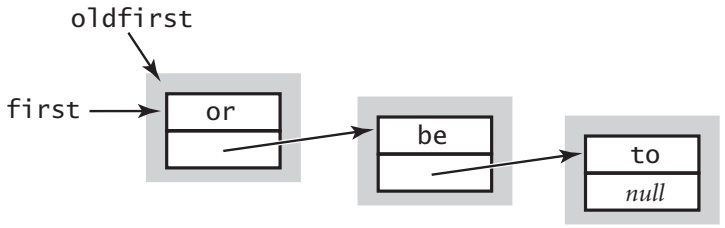
Stack push: linked-list implementation

inner class

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

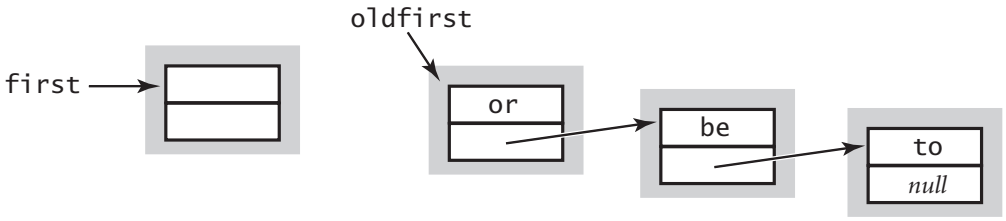
save a link to the list

```
Node oldfirst = first;
```



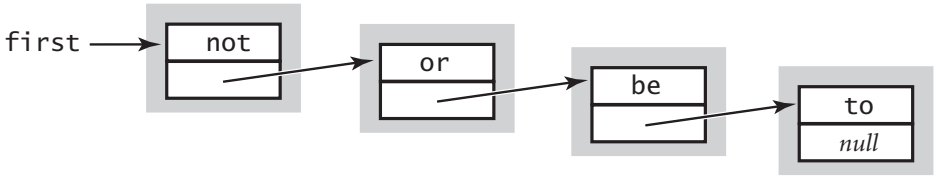
create a new node for the beginning

```
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)

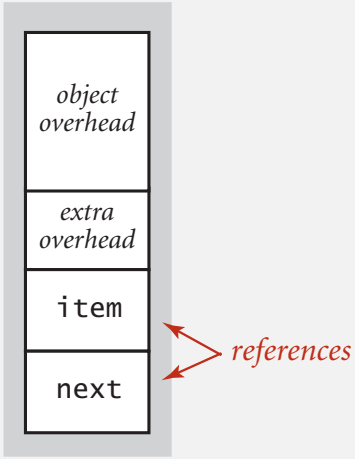
Stack: linked-list implementation performance

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

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

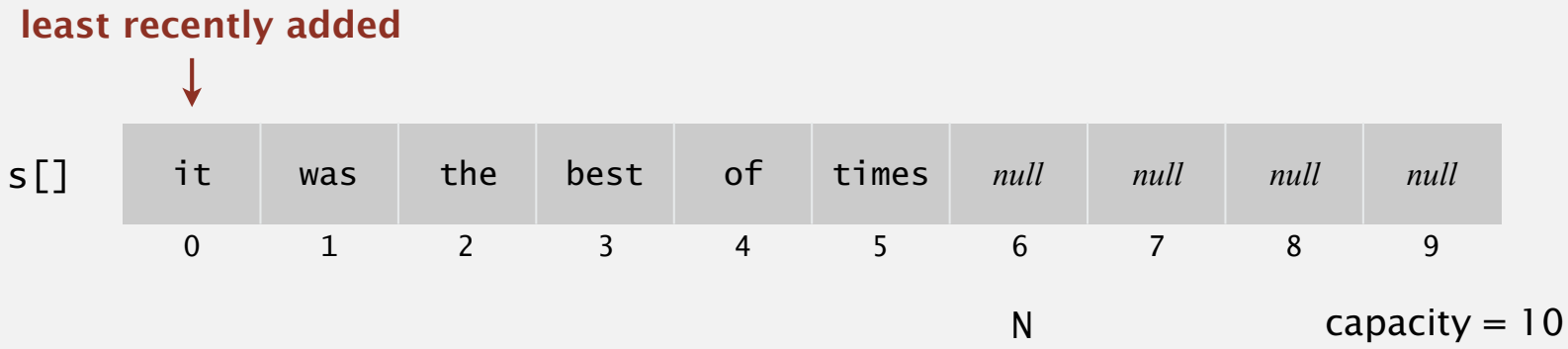
8 bytes (reference to Node)

40 bytes allocated per stack node

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.
- `push()`: add new item at `s[N]`.
- `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];  }
}
```

a cheat
(stay tuned)

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 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

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 = $N + (2 + 4 + \ldots + 2(N-1)) \sim N^2$.

quadratic time, which is infeasible for large N

1 array access per push

2(k–1) array accesses to expand to size k (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.

"repeated doubling"

```
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. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

1 array access per push

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

Stack: resizing-array implementation

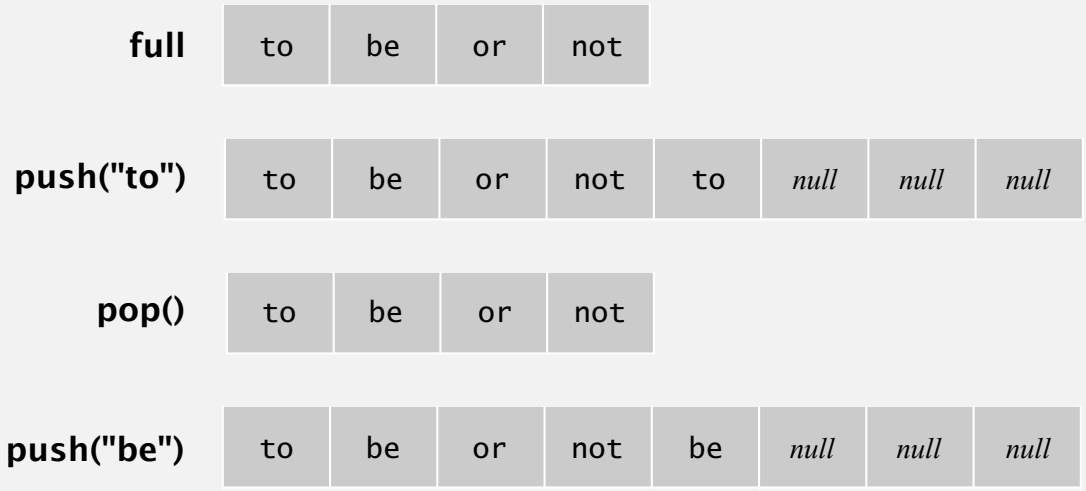
Q. How to shrink array?

First try.

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

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

doubling and halving operations

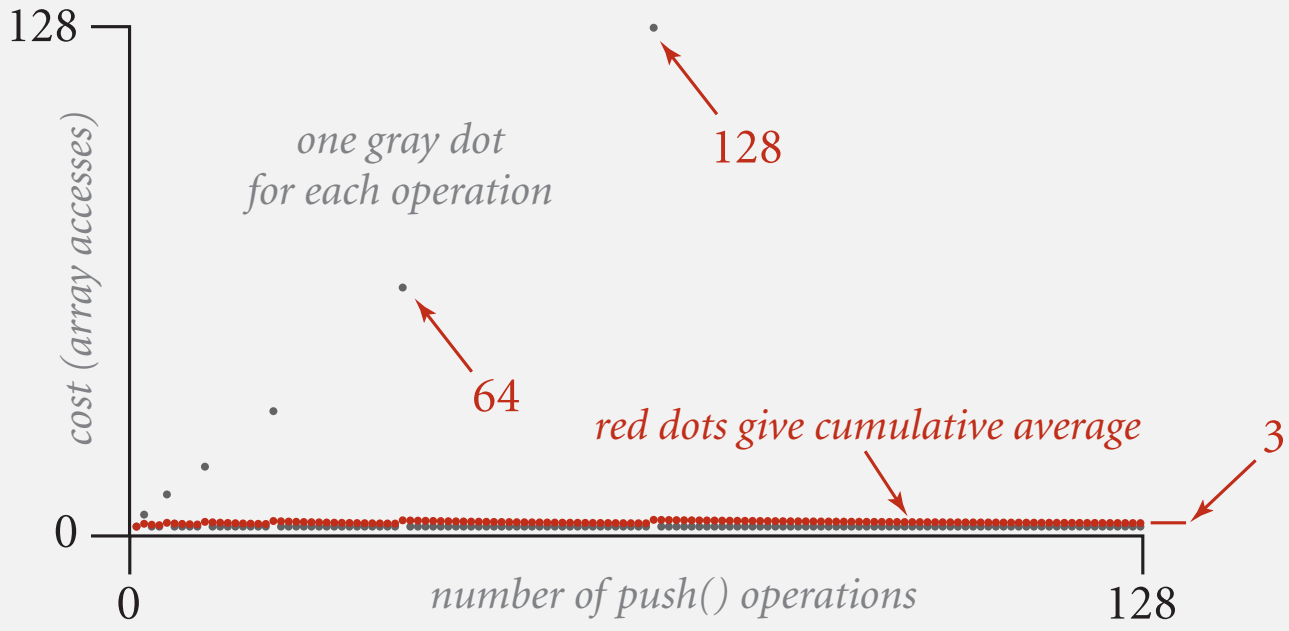
order of growth of running time for resizing array stack with N items

Stack: amortized cost of adding to a stack

Cost of adding first N 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: memory usage

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

- $\sim 8\,N$ when full.
- $\sim 32\,N$ 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

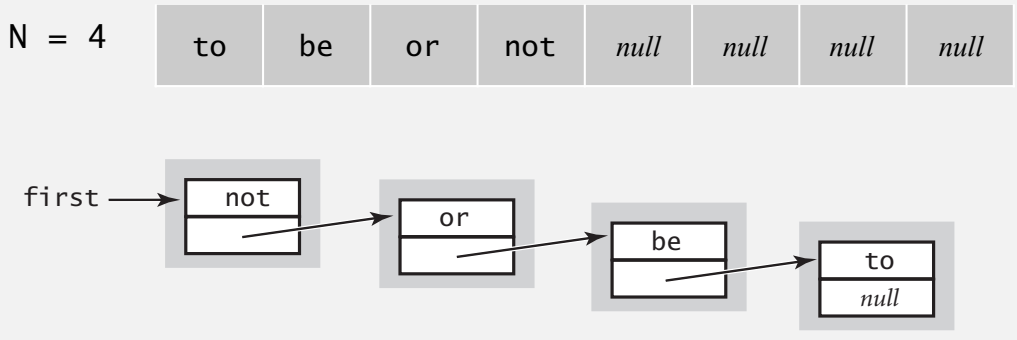
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

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.



Robert Sedgewick | Kevin Wayne

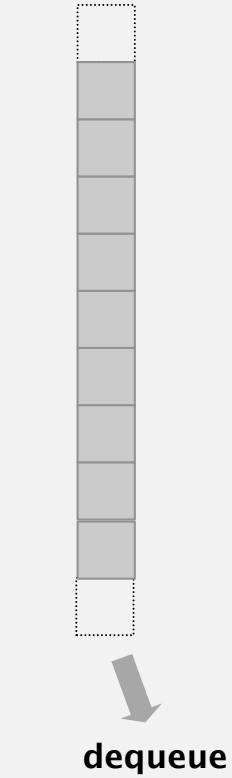
http://algs4.cs.princeton.edu

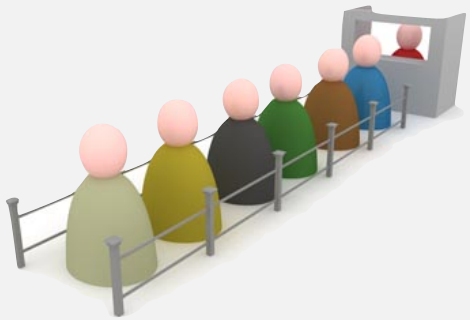
1.3 Bags, Queues, and Stacks

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

Queue API

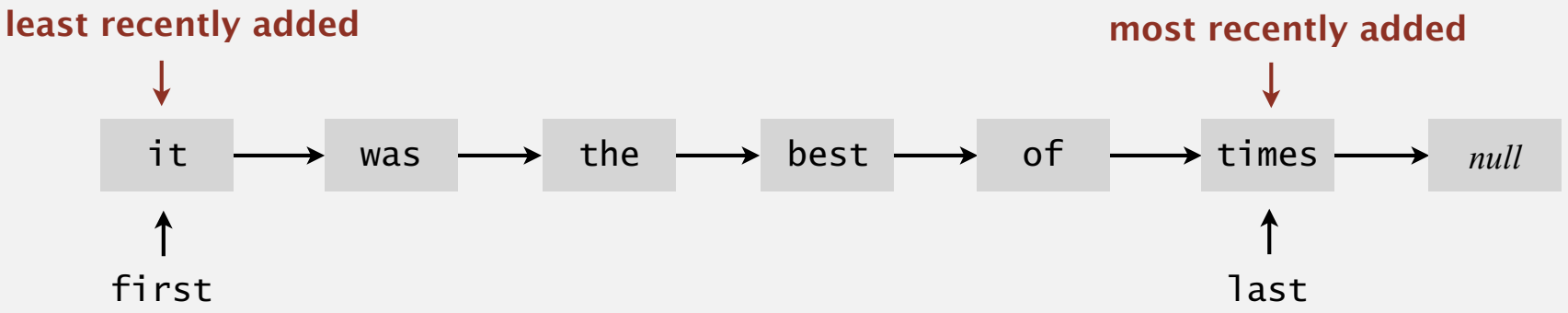
public class QueueOfStrings	
QueueOfStrings()	*create an empty queue*
void enqueue(String item)	*add a new string to queue*
String dequeue()	*remove and return the string least recently added*
boolean isEmpty()	*is the queue empty?*
int size()	*number of strings on the queue*





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

inner class

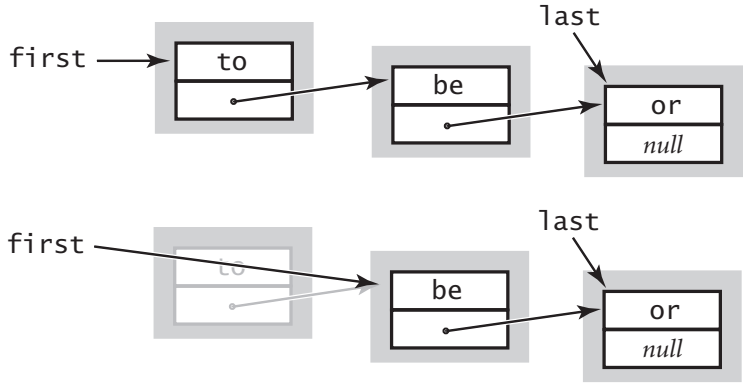
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

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

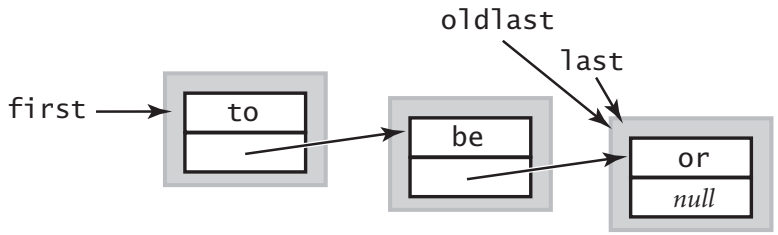
Queue enqueue: linked-list implementation

inner class

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

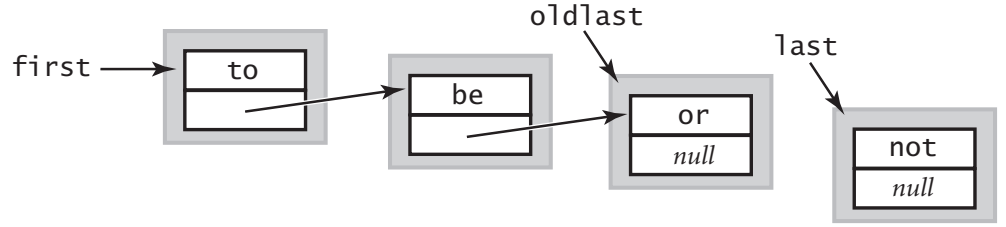
save a link to the last node

```
Node oldlast = last;
```



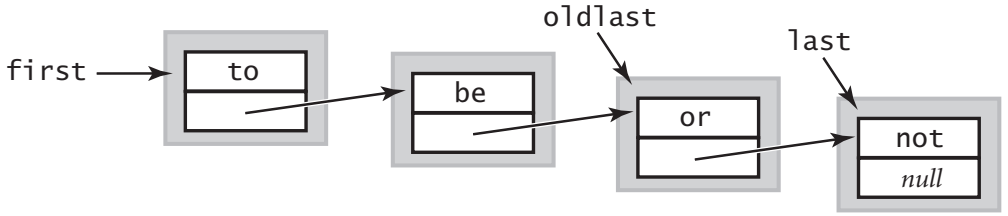
create a new node for the end

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



link the new node to the end of the list

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



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]`.
- Update `head` and `tail` modulo the `capacity`.
- Add resizing array.

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

1.3 Bags, Queues, and Stacks

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

Parameterized stack

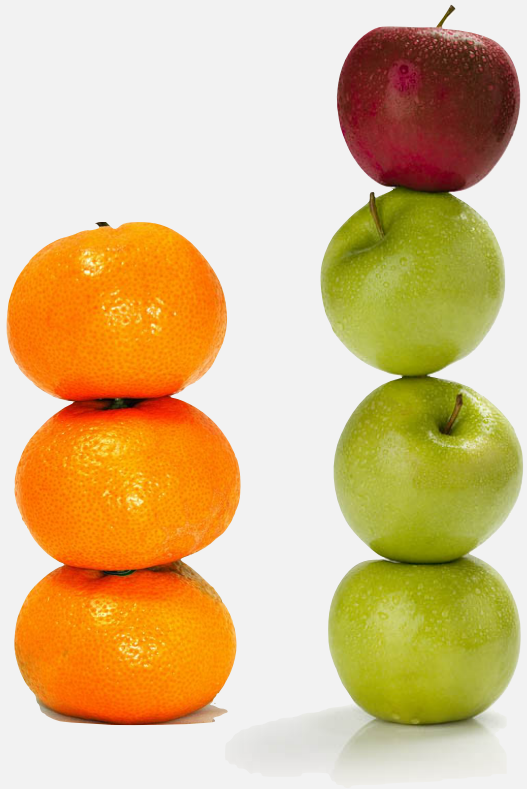
We implemented: `StackOfStrings`.

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.

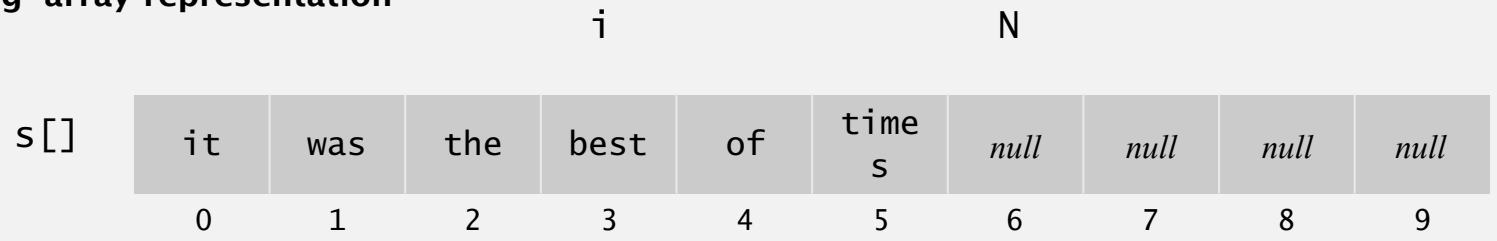
```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);          // stack.push(Integer.valueOf(17));
int a = stack.pop();     // int a = stack.pop().intValue();
```

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

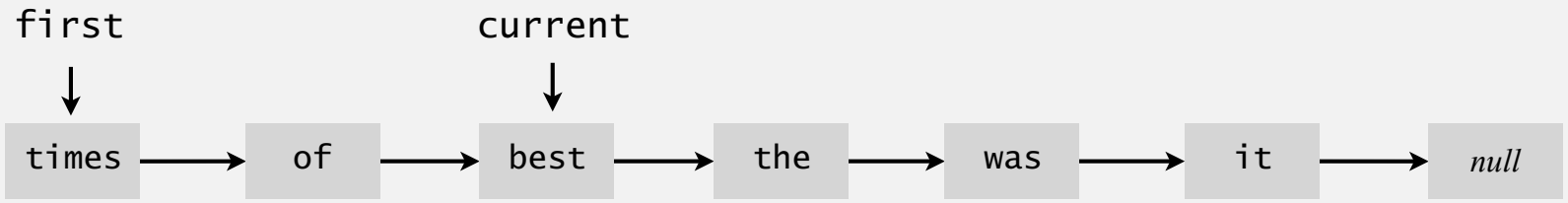
Iteration

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 method.
  - the `hasNext()` methods 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.

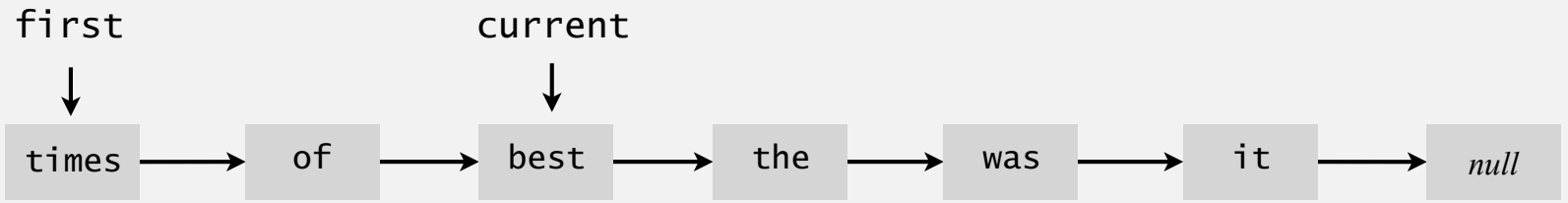
- Data type must use these interfaces to support foreach loop.
- Client program won't compile if implementation doesn't.

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;
        }
    }
}
```

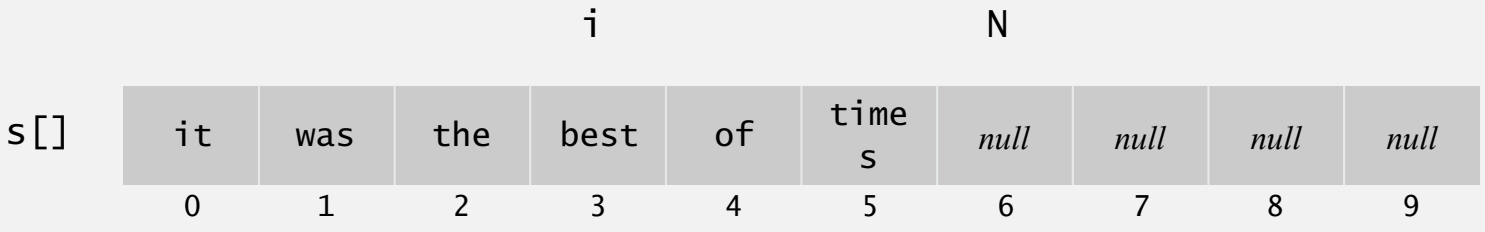
throw UnsupportedOperationException

throw NoSuchElementException if no more items in iteration



Stack iterator: array implementation

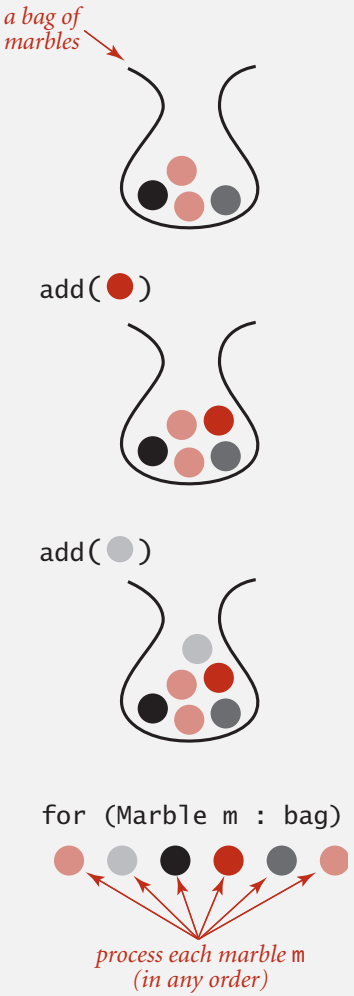
```
import java.util.Iterator;

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];        }
    }
}
```



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*



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

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>		
	List()	*create an empty list*
boolean	isEmpty()	*is the list empty?*
int	size()	*number of items*
void	add(Item item)	*add 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*
	⋮	

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`.

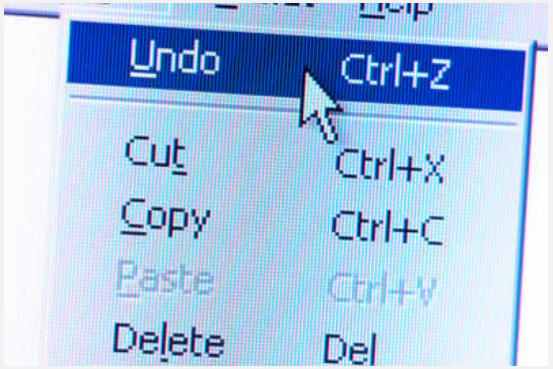
- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

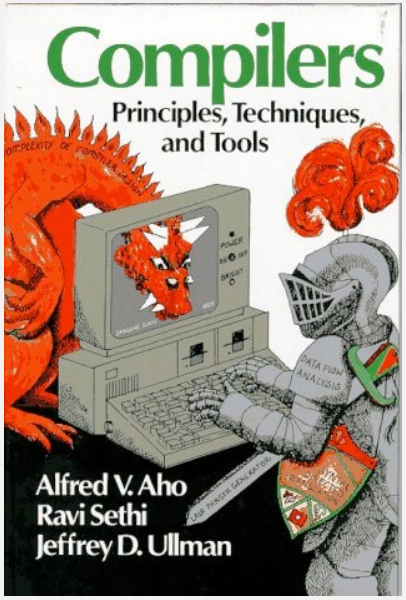
`java.util.Queue`. An interface, not an implementation of a queue.

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.
- ...





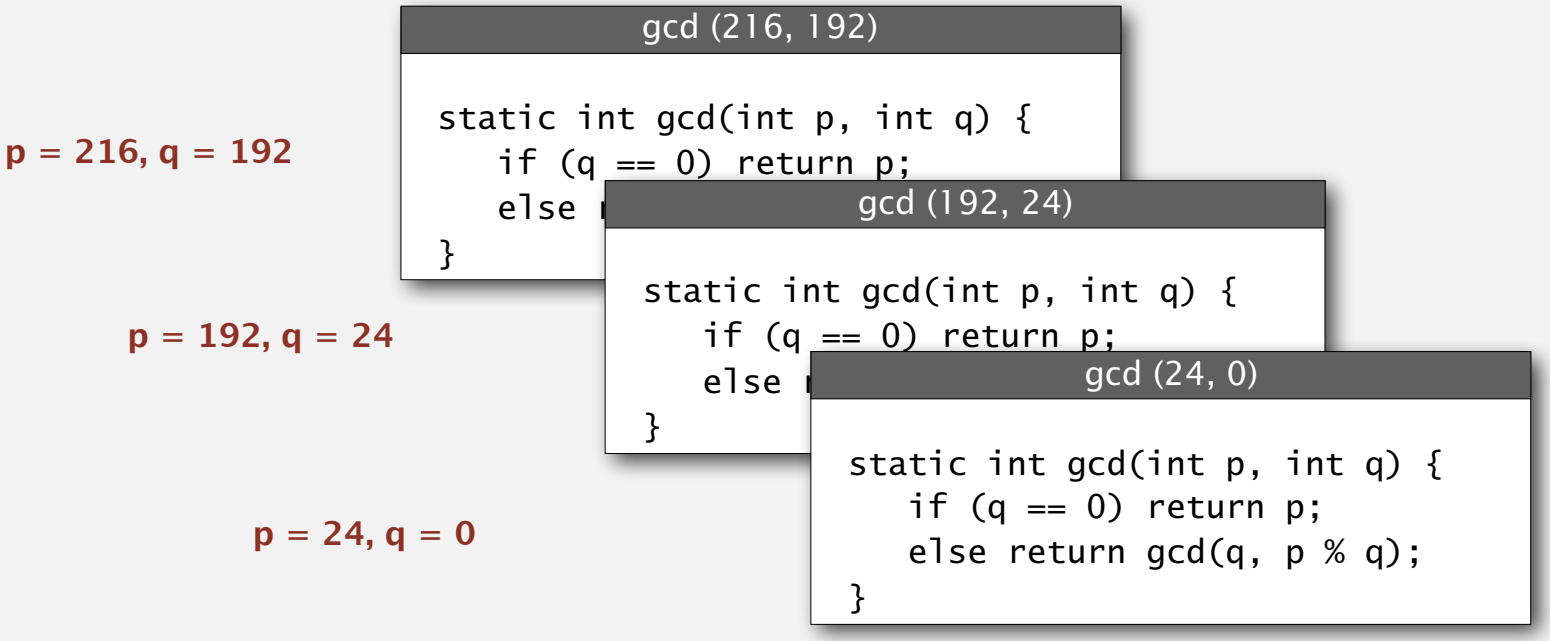
Function calls

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

value stack operator 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

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.

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

Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack operator stack

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

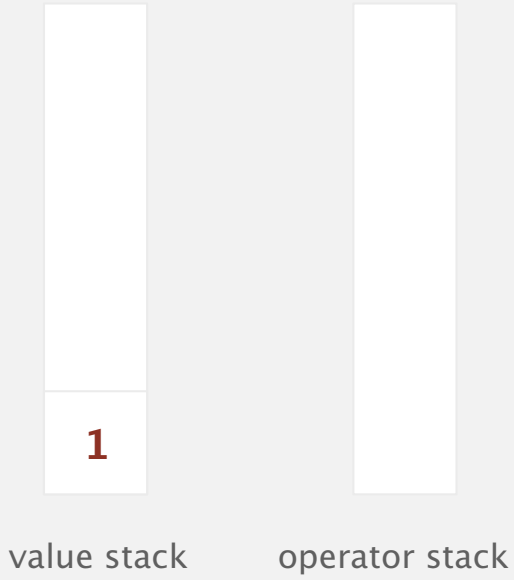
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



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

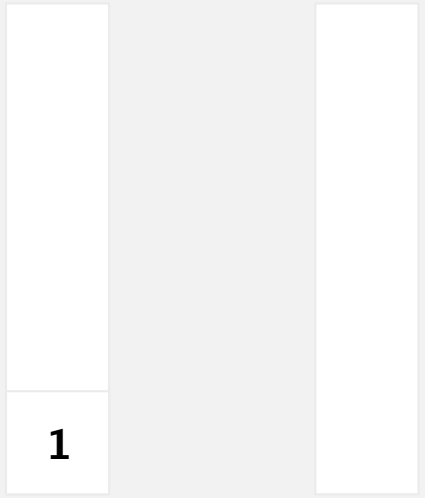
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



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

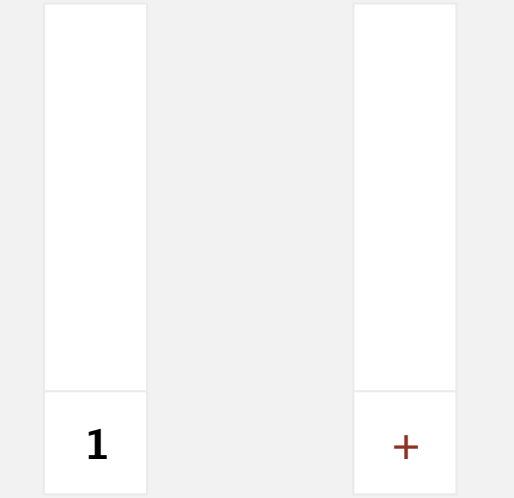
Dijkstra's two-stack algorithm

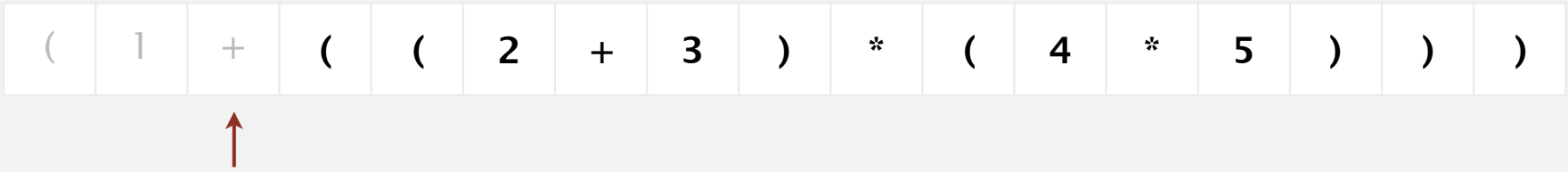
Value: push onto the value stack.

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.





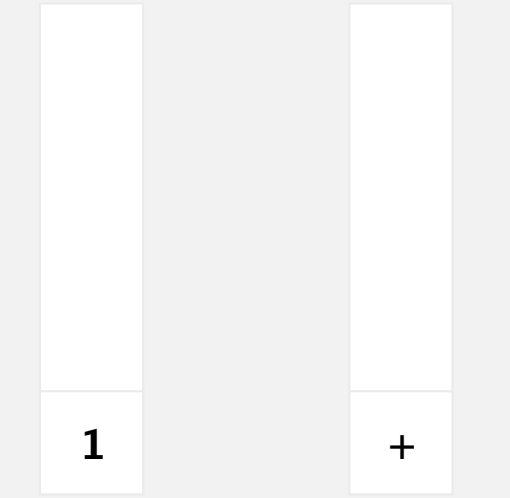
Dijkstra's two-stack algorithm

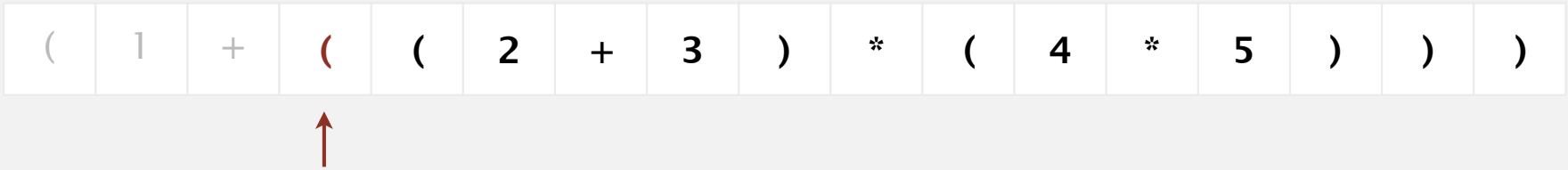
Value: push onto the value stack.

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.





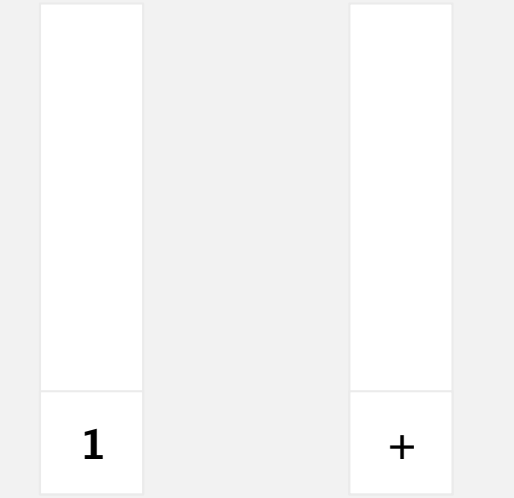
Dijkstra's two-stack algorithm

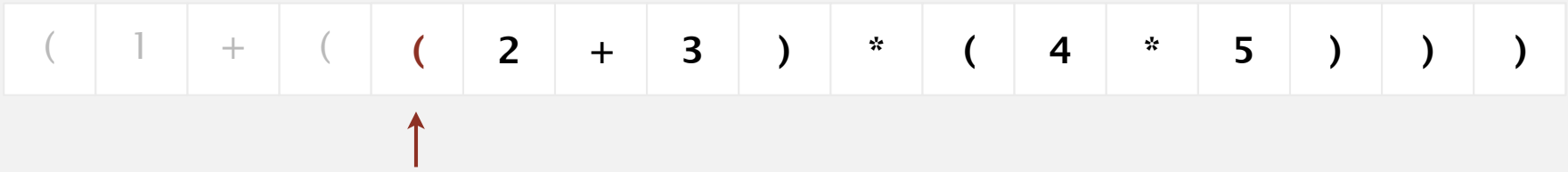
Value: push onto the value stack.

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.





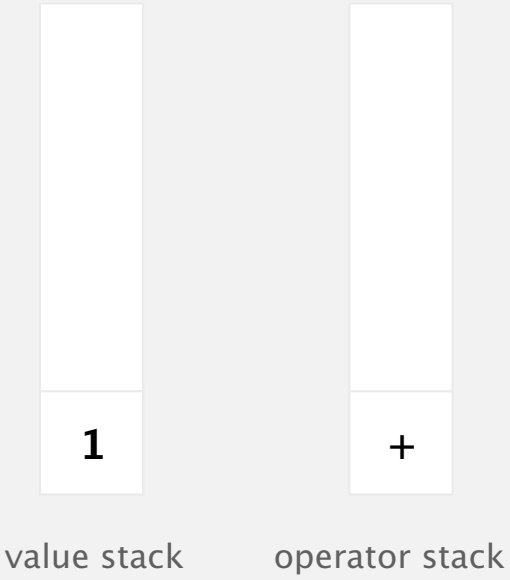
Dijkstra's two-stack algorithm

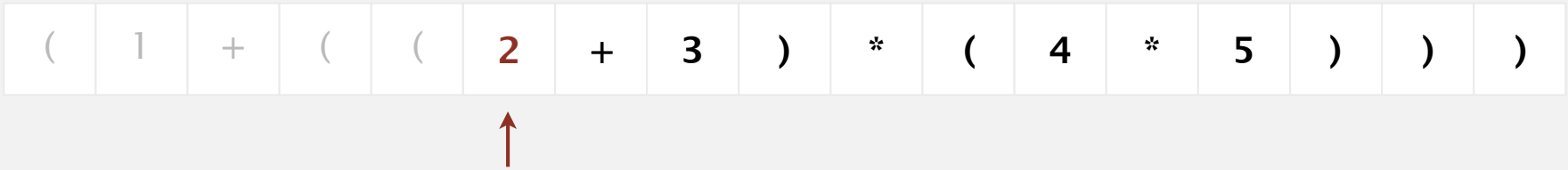
Value: push onto the value stack.

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.





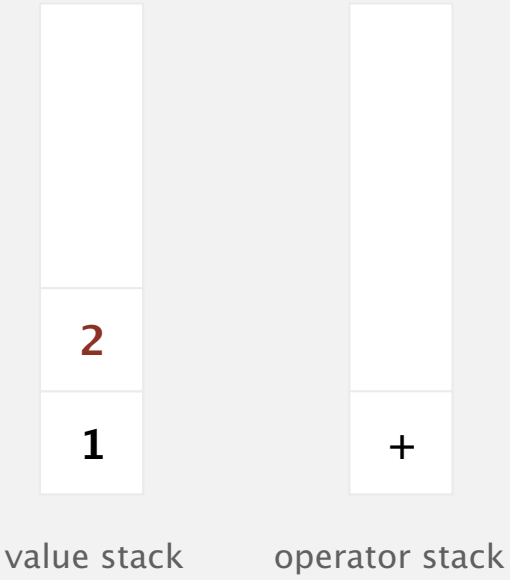
Dijkstra's two-stack algorithm

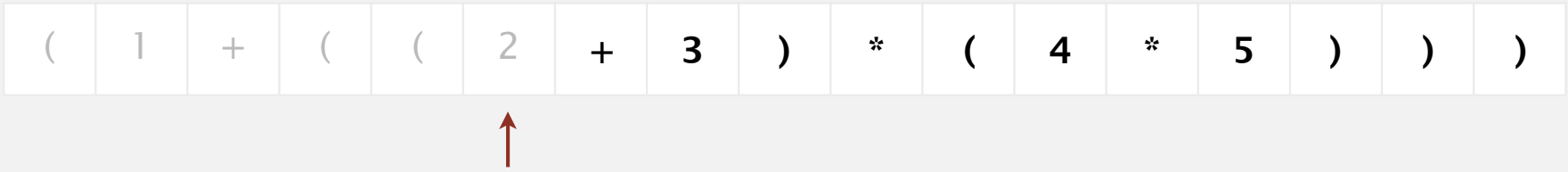
Value: push onto the value stack.

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.





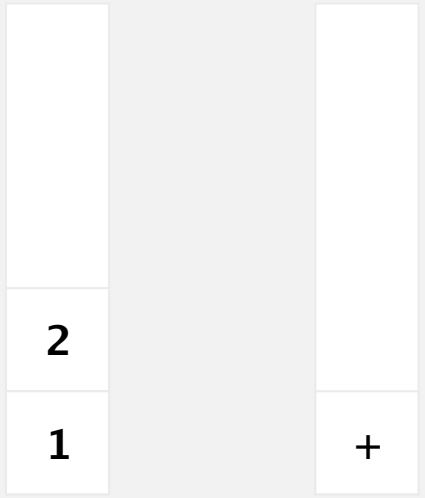
Dijkstra's two-stack algorithm

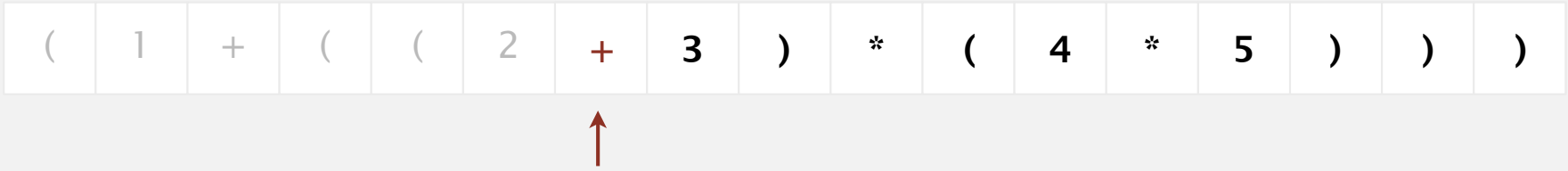
Value: push onto the value stack.

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.





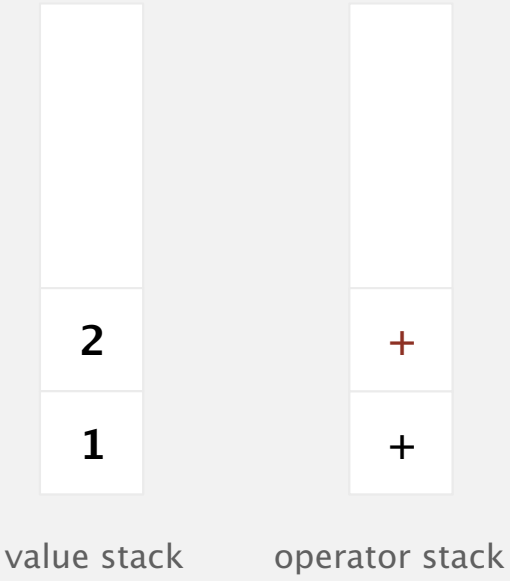
Dijkstra's two-stack algorithm

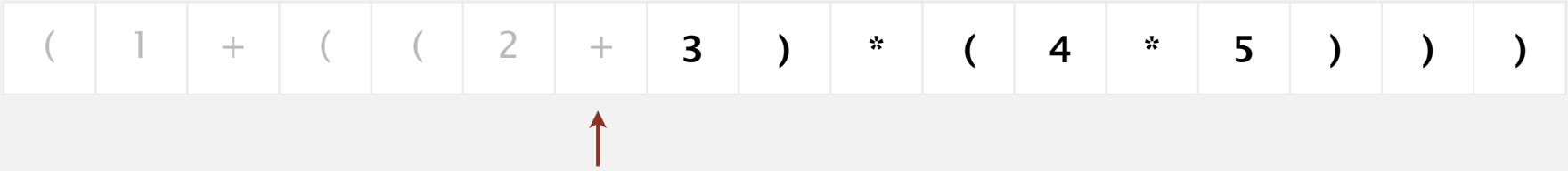
Value: push onto the value stack.

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.





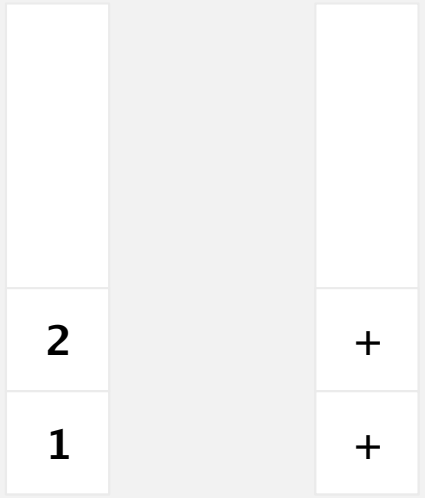
Dijkstra's two-stack algorithm

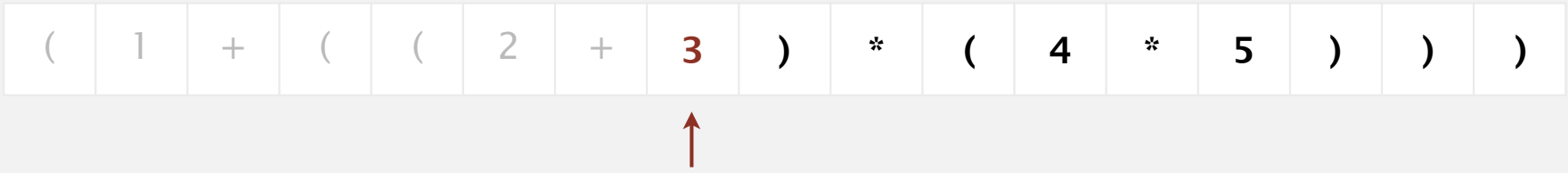
Value: push onto the value stack.

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.





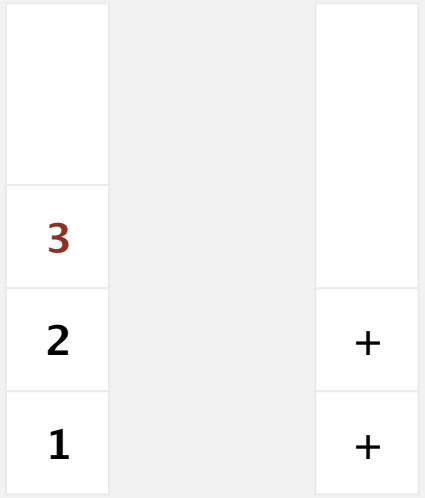
Dijkstra's two-stack algorithm

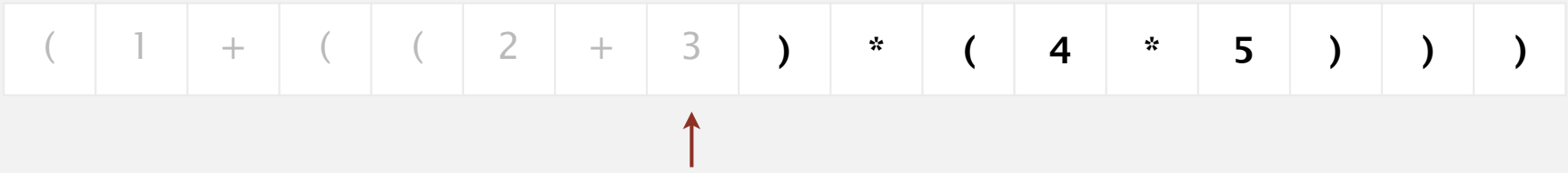
Value: push onto the value stack.

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.





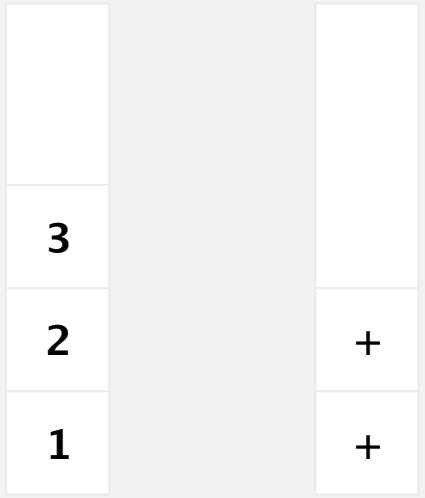
Dijkstra's two-stack algorithm

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.



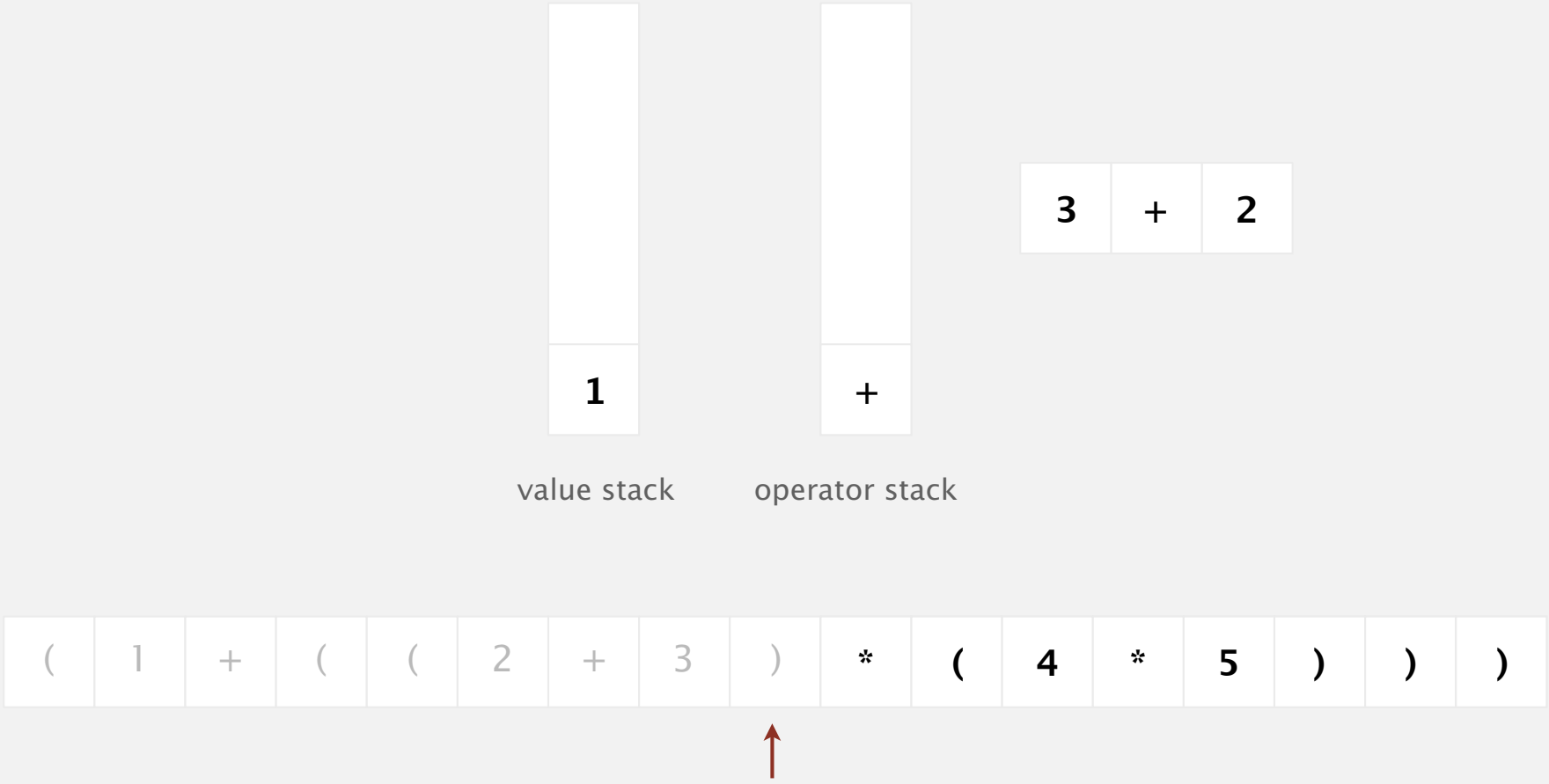
Dijkstra's two-stack algorithm

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.



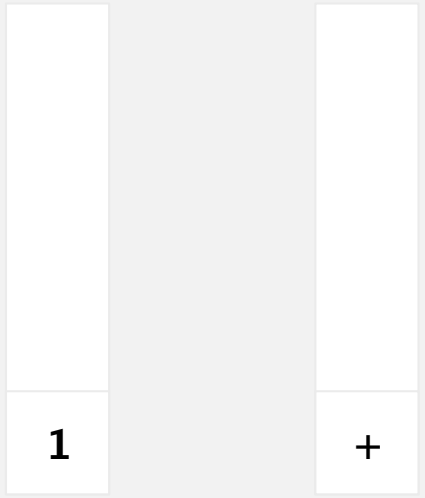
Dijkstra's two-stack algorithm

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.



3	+	2	=	5

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

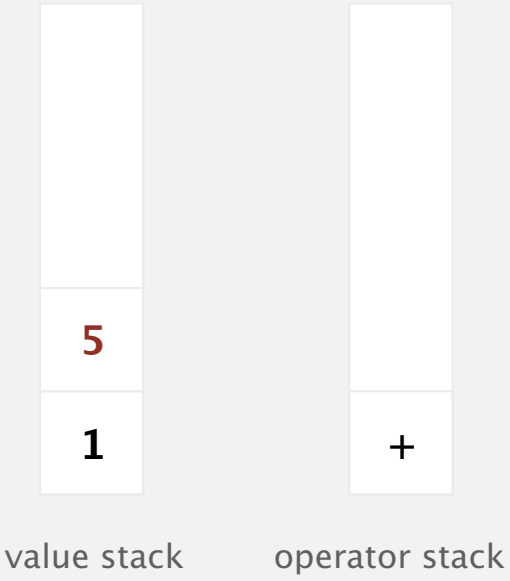
Dijkstra's two-stack algorithm

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.



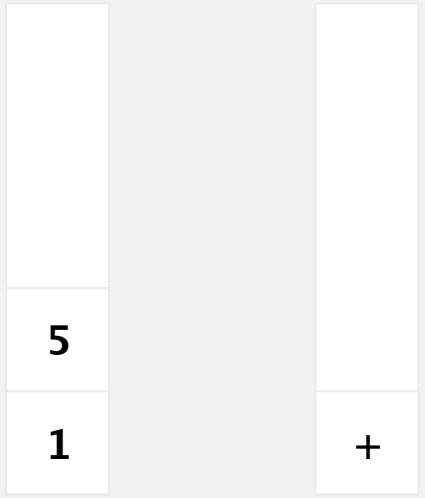
Dijkstra's two-stack algorithm

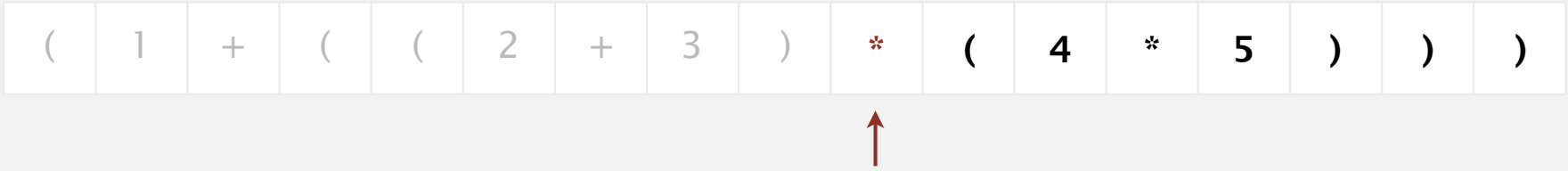
Value: push onto the value stack.

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.





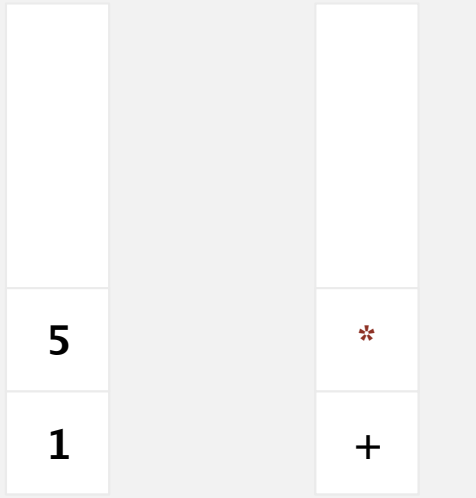
Dijkstra's two-stack algorithm

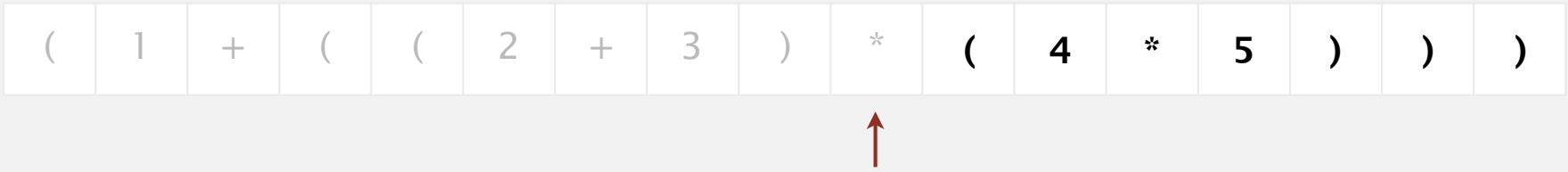
Value: push onto the value stack.

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.





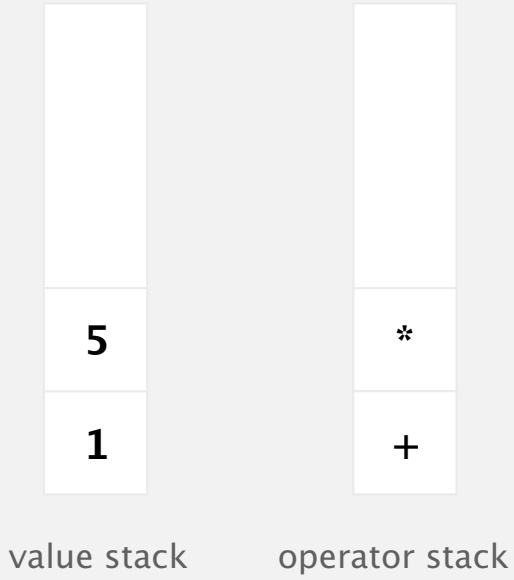
Dijkstra's two-stack algorithm

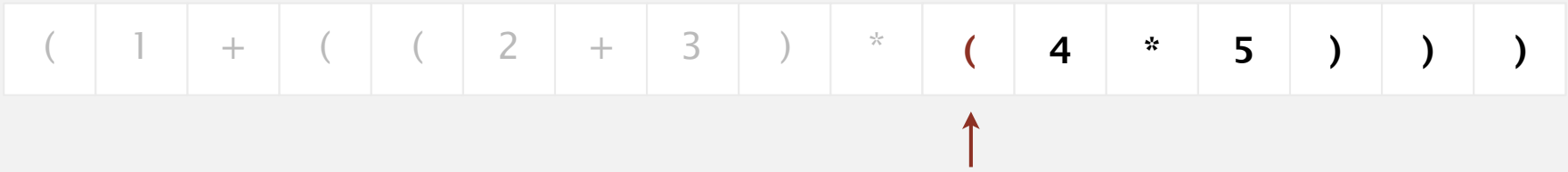
Value: push onto the value stack.

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.





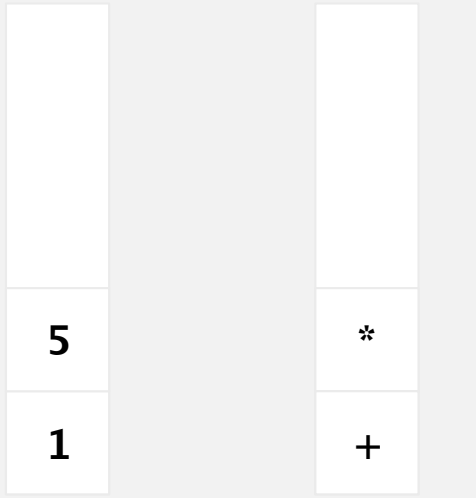
Dijkstra's two-stack algorithm

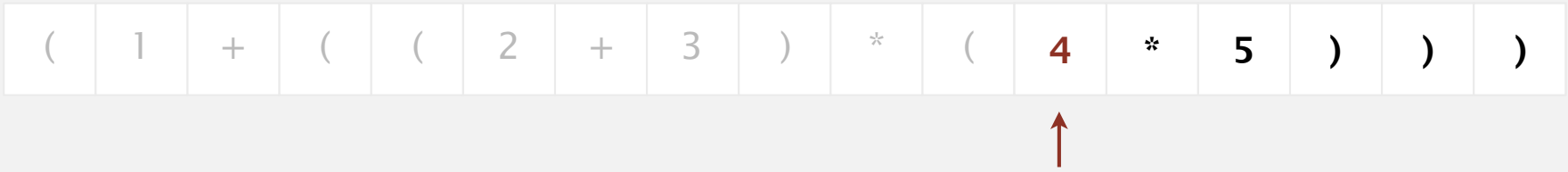
Value: push onto the value stack.

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.





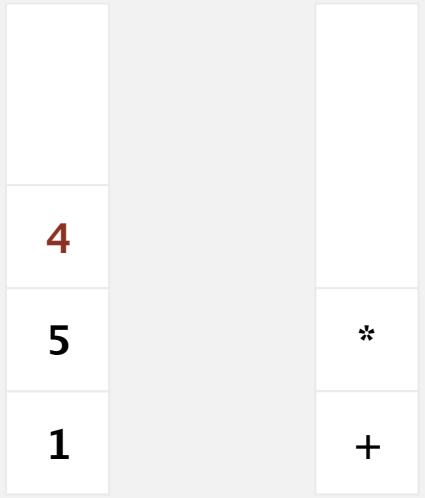
Dijkstra's two-stack algorithm

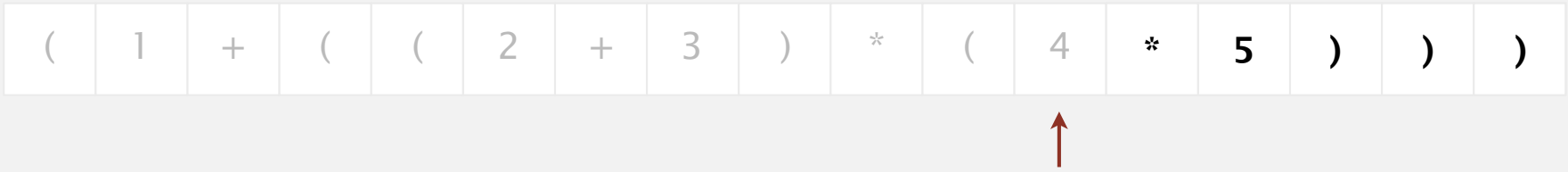
Value: push onto the value stack.

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.





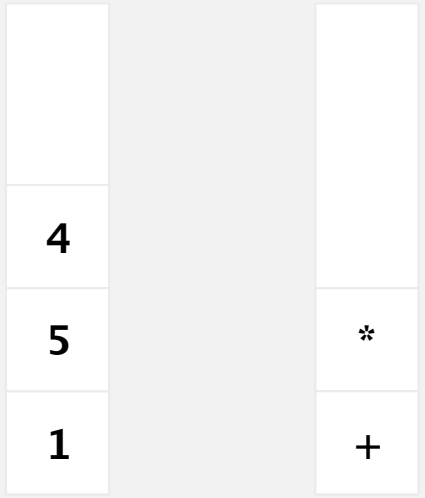
Dijkstra's two-stack algorithm

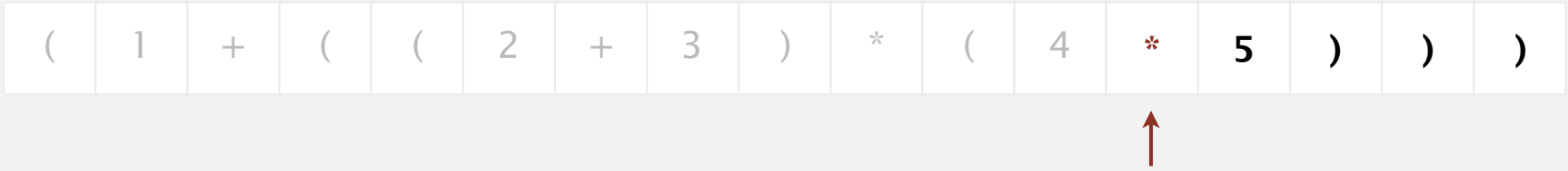
Value: push onto the value stack.

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.





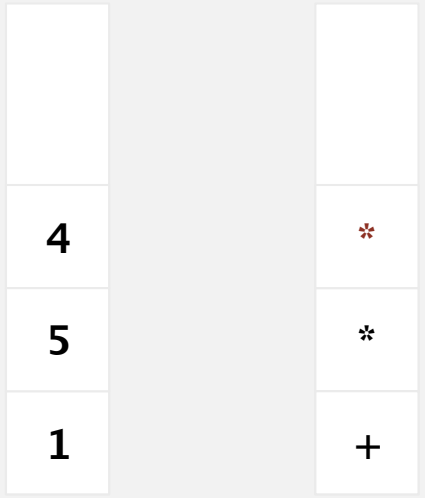
Dijkstra's two-stack algorithm

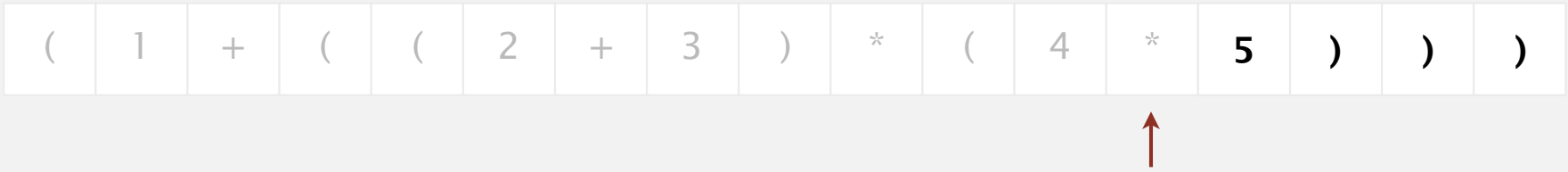
Value: push onto the value stack.

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.





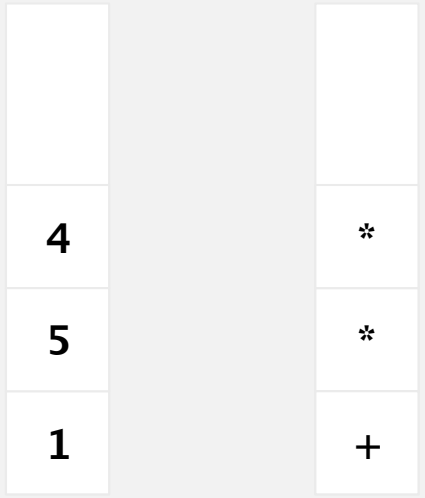
Dijkstra's two-stack algorithm

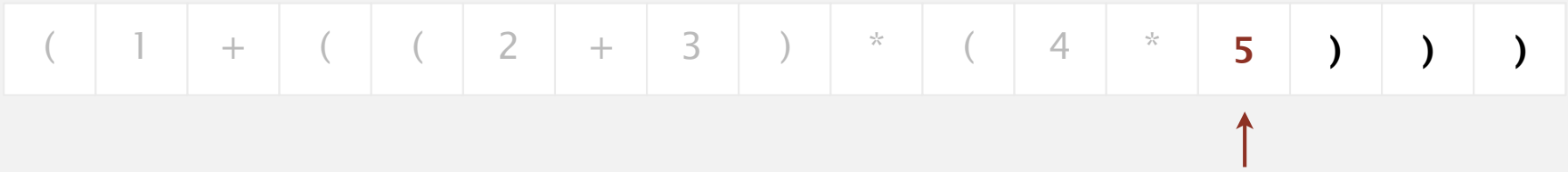
Value: push onto the value stack.

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.





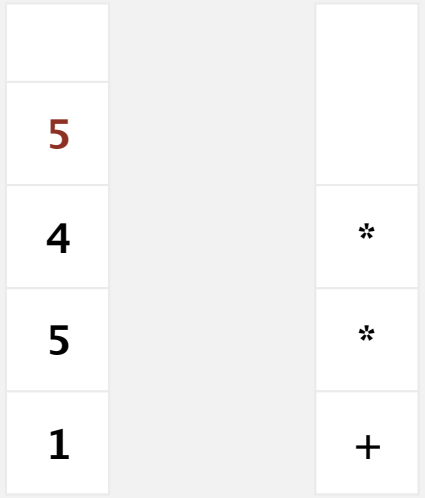
Dijkstra's two-stack algorithm

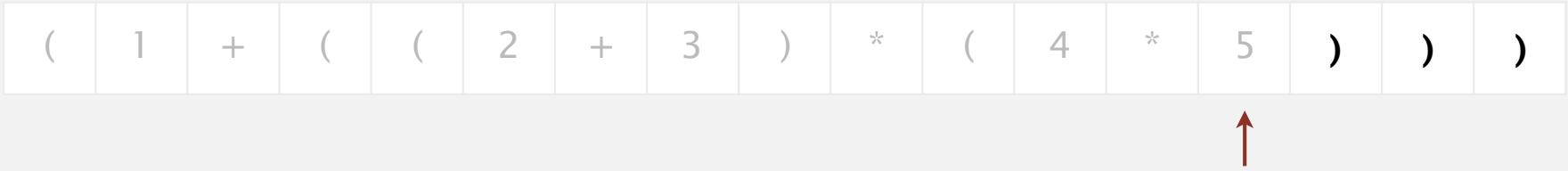
Value: push onto the value stack.

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.





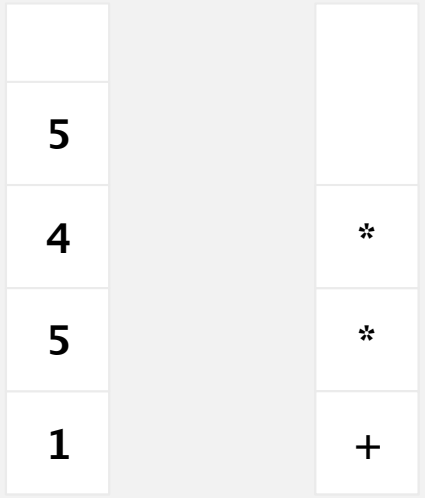
Dijkstra's two-stack algorithm

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.



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

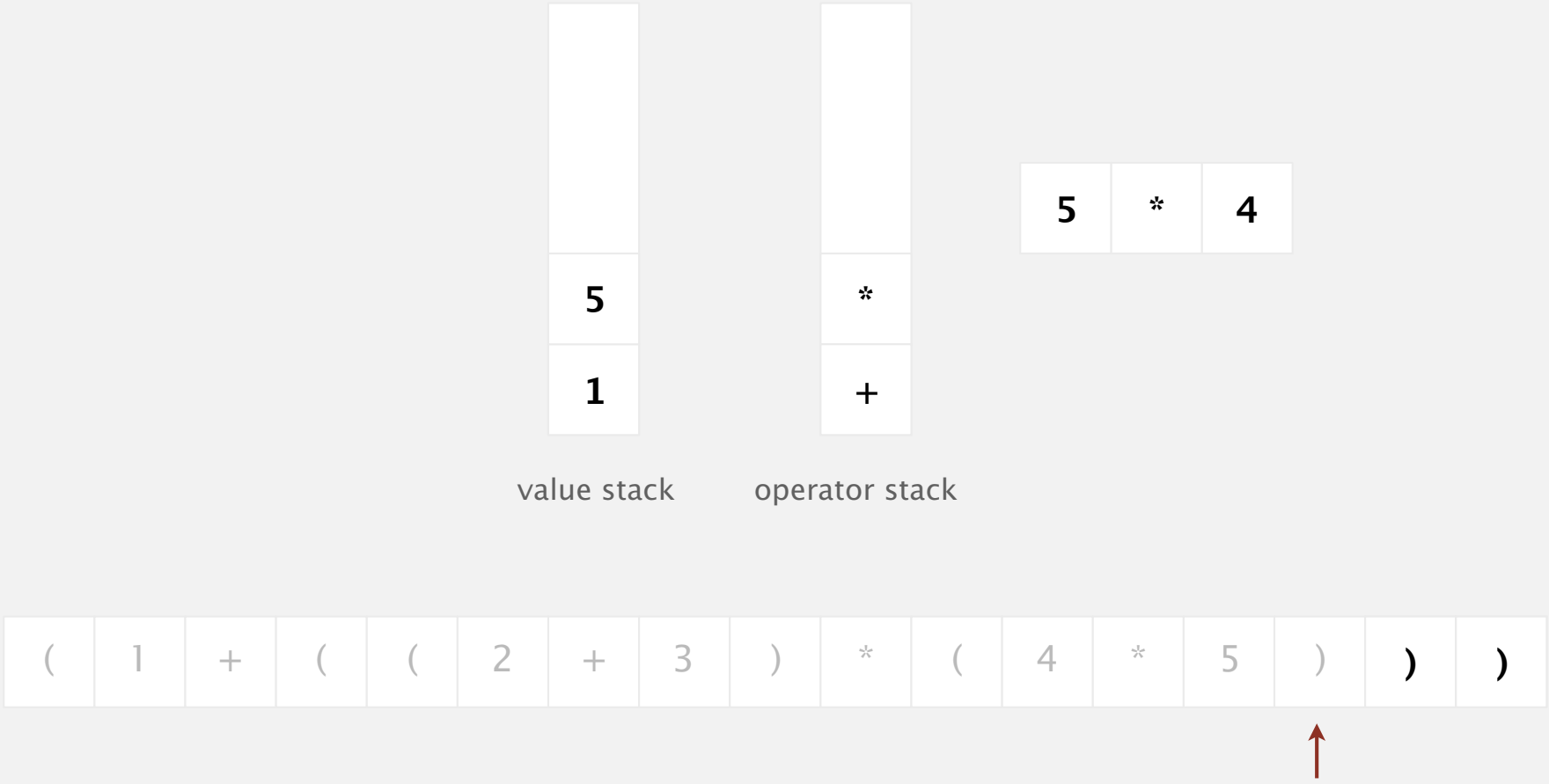
Dijkstra's two-stack algorithm

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.



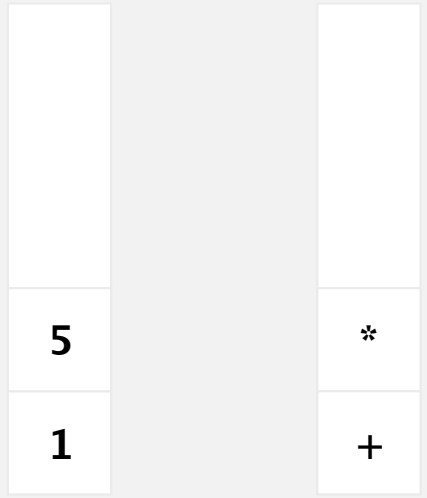
Dijkstra's two-stack algorithm

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	=	20

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

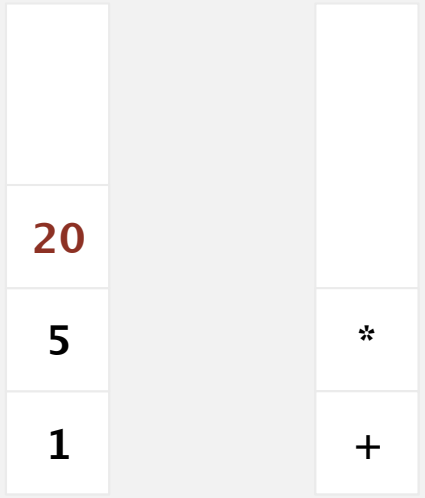
Dijkstra's two-stack algorithm

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.



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

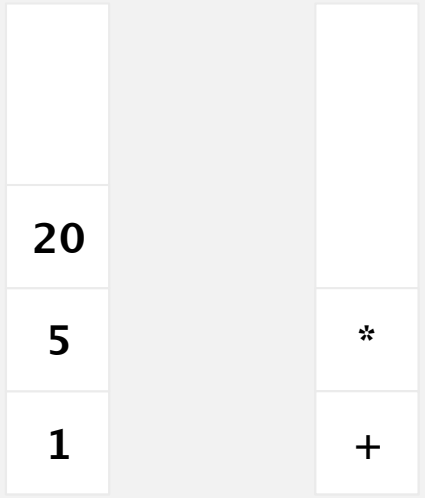
Dijkstra's two-stack algorithm

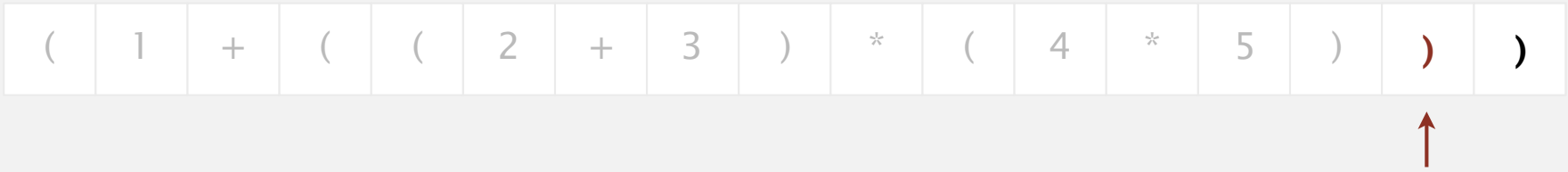
Value: push onto the value stack.

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.





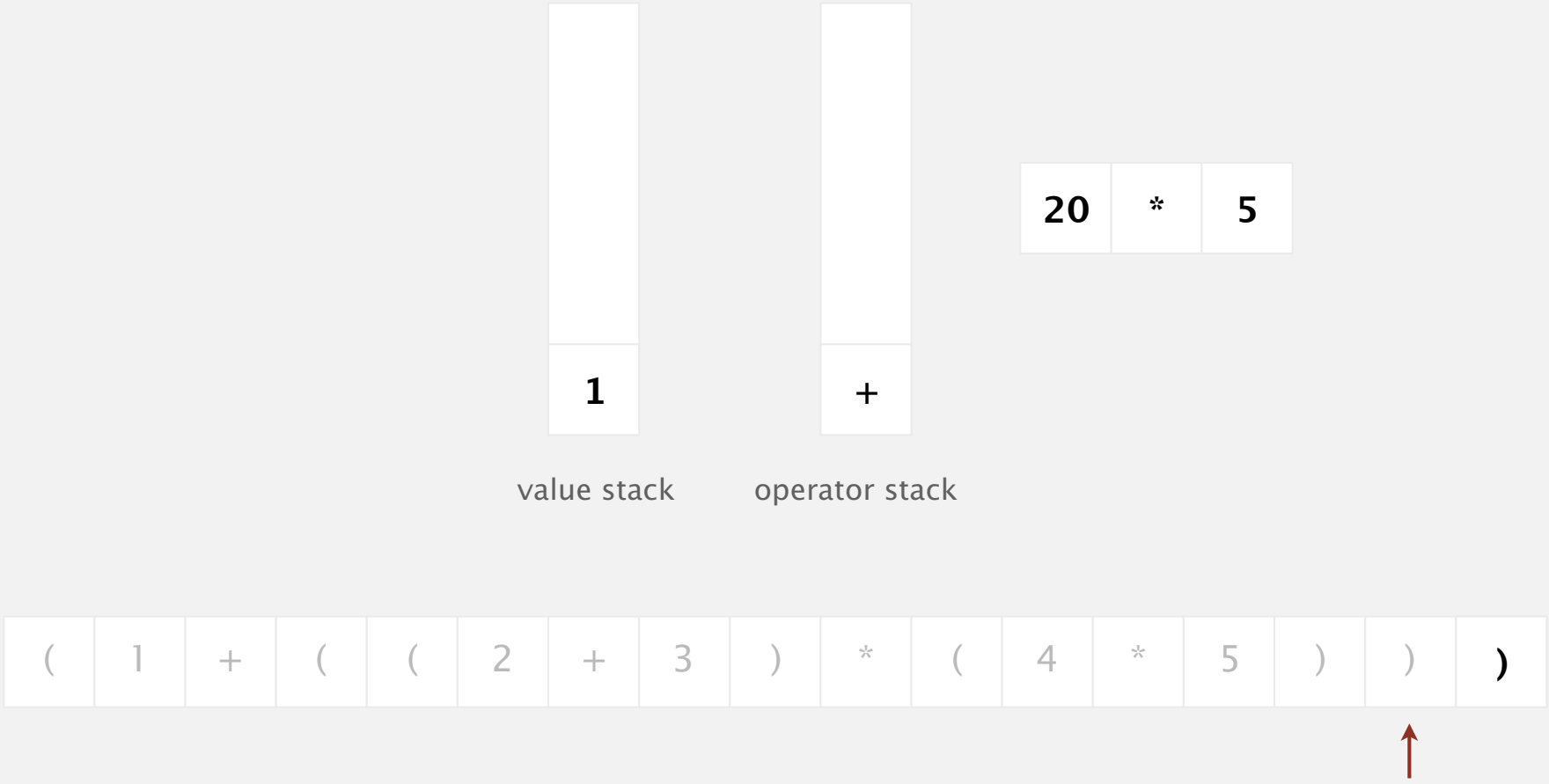
Dijkstra's two-stack algorithm

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.



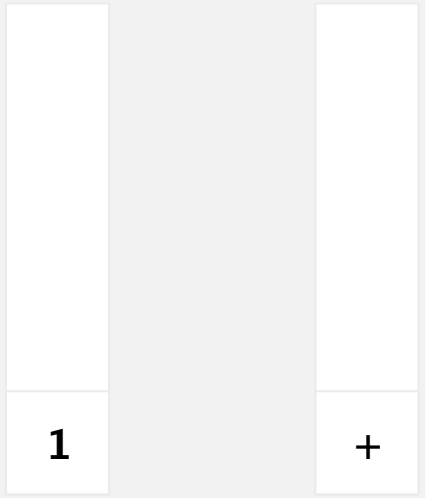
Dijkstra's two-stack algorithm

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.



20	*	5	=	100

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

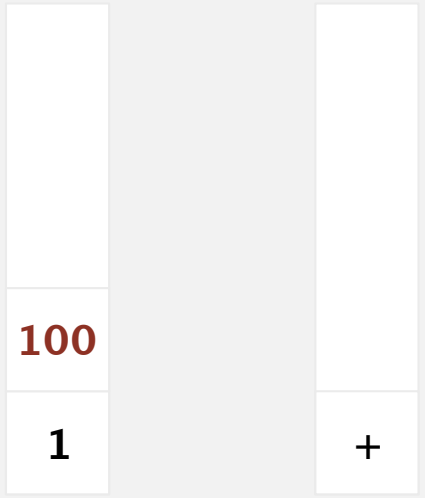
Dijkstra's two-stack algorithm

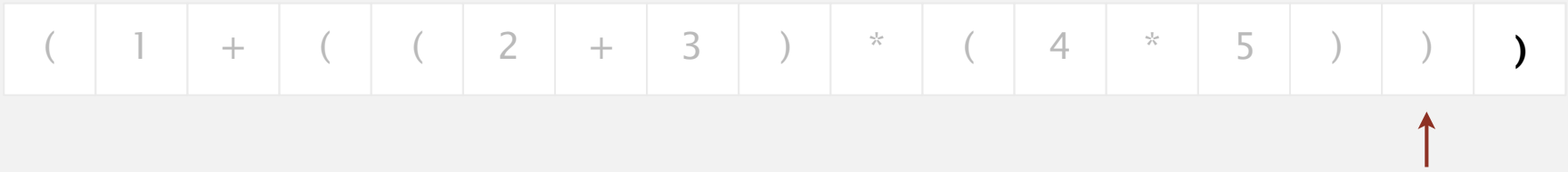
Value: push onto the value stack.

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.





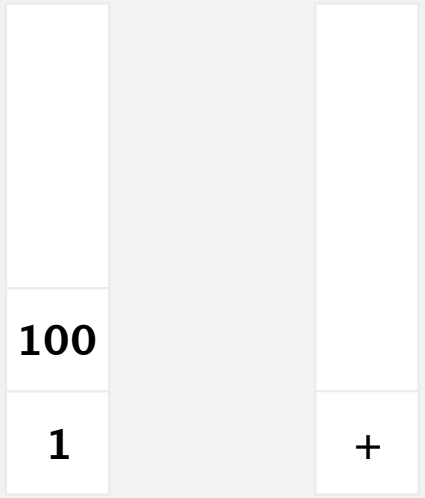
Dijkstra's two-stack algorithm

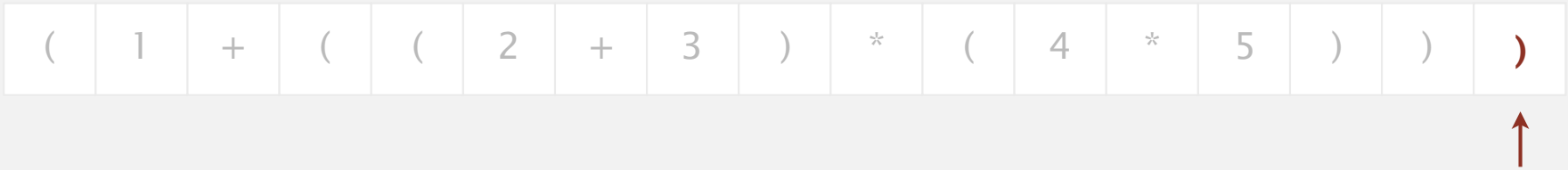
Value: push onto the value stack.

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.

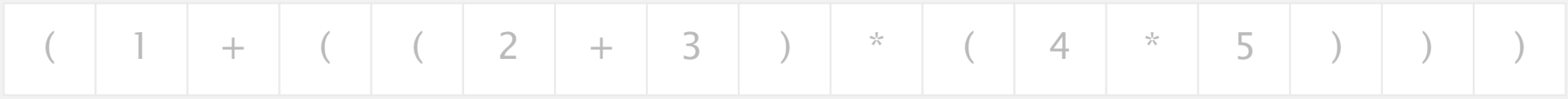
Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack

operator stack



Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

value stack operator stack

100	+	1	=	101

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

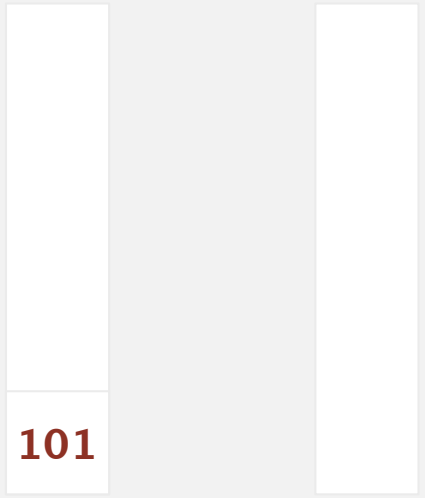
Dijkstra's two-stack algorithm

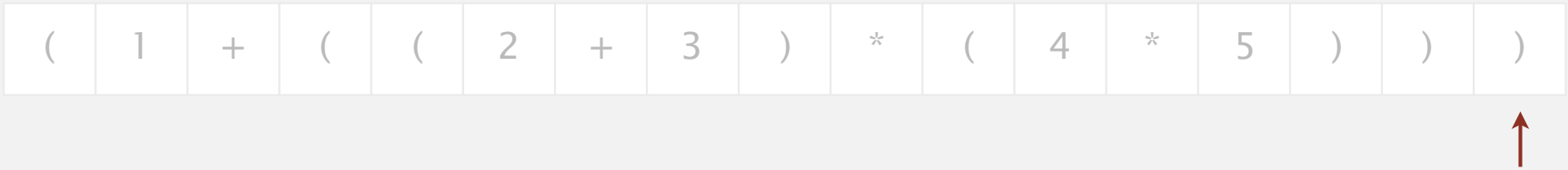
Value: push onto the value stack.

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.





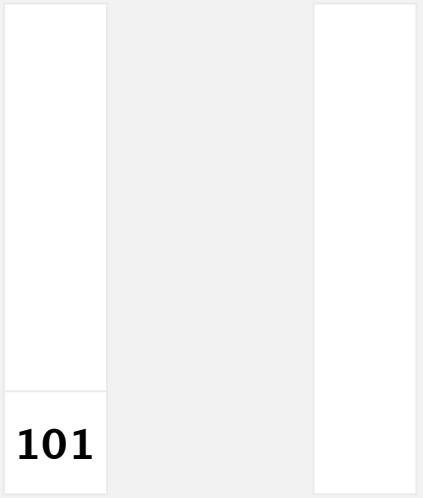
Dijkstra's two-stack algorithm

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.



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

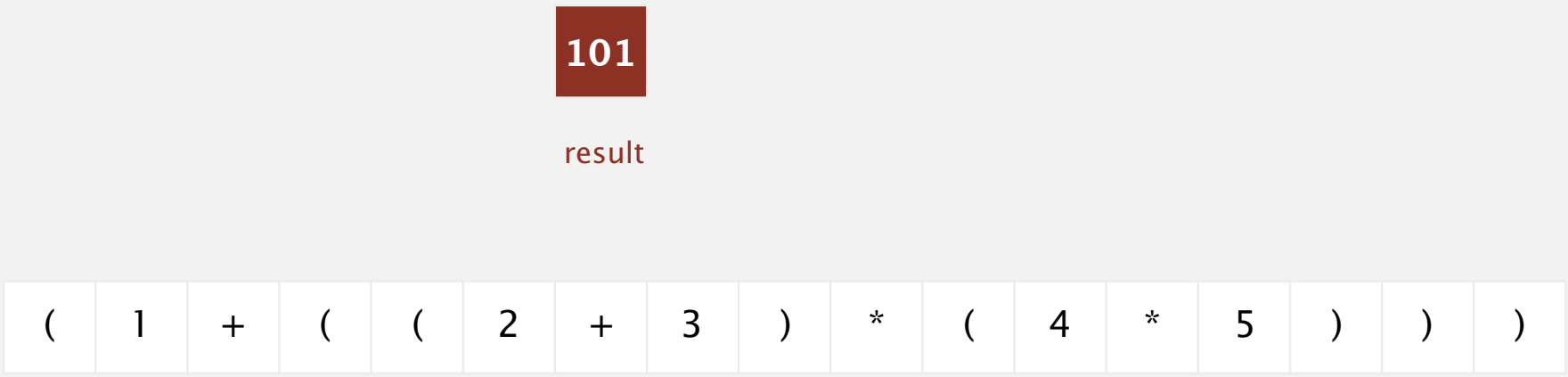
Dijkstra's two-stack algorithm

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.



Arithmetic expression evaluation

Goal. Evaluate infix expressions.

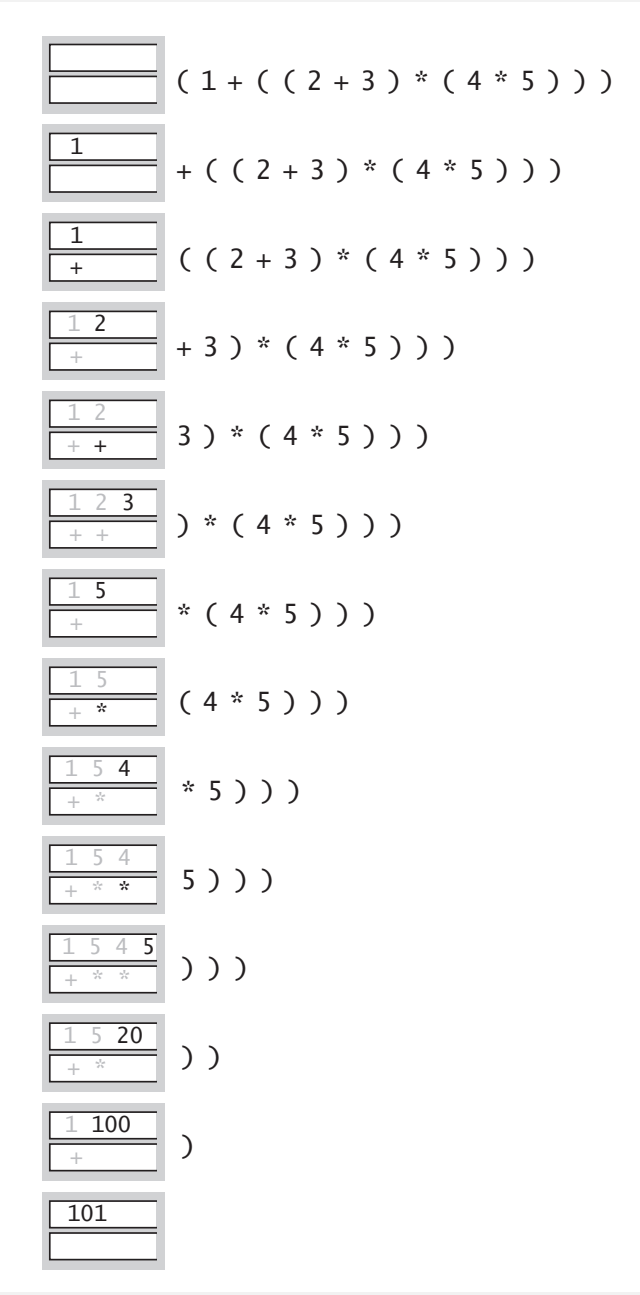
```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

operand operator

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

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

Context. An interpreter!



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, ...