



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

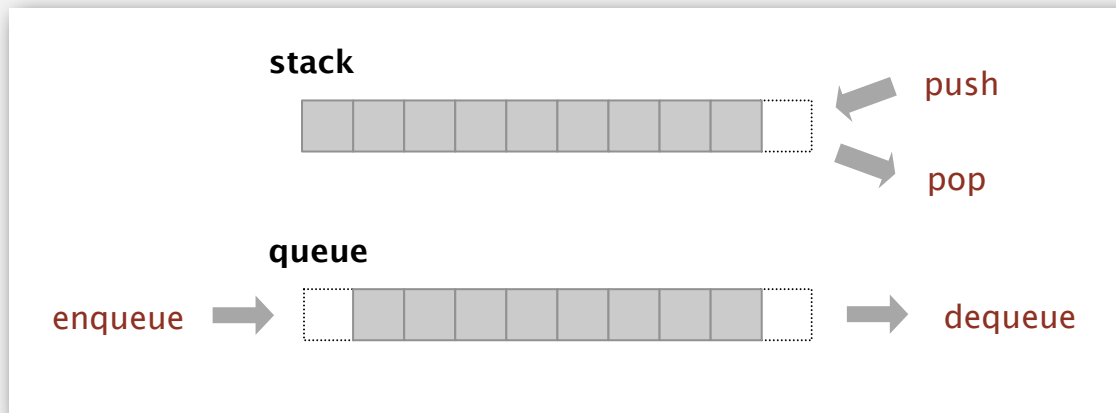
1.3 BAGS, QUEUES, AND STACKS

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

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: **insert**, **remove**, **iterate**, test if empty.
- Intent is clear when we insert.
- 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, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find,

Benefits.

- Client can't know details of implementation \Rightarrow client has many implementation from which to choose.
- Implementation can't know details of client needs \Rightarrow many clients can re-use the same implementation.
- **Design:** creates modular, reusable libraries.
- **Performance:** use optimized implementation where it matters.

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Interface: description of data type, basic operations.



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

1.3 BAGS, QUEUES, AND STACKS

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

Stack API

Warmup API. Stack of strings data type.

```
public class StackOfStrings
```

```
    StackOfStrings()
```

create an empty stack

```
    void push(String item)
```

insert a new string onto stack

```
    String pop()
```

*remove and return the string
most recently added*

```
    boolean isEmpty()
```

is the stack empty?

```
    int size()
```

number of strings on the stack

push pop



Warmup client. Reverse sequence of strings from standard input.

Stack test client

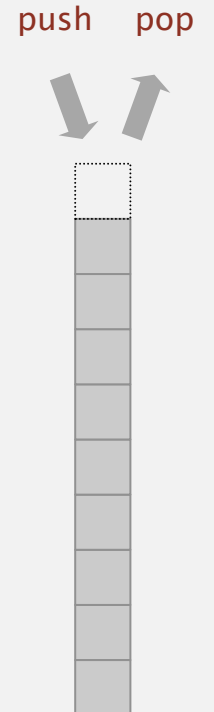
Read strings from standard input.

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

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

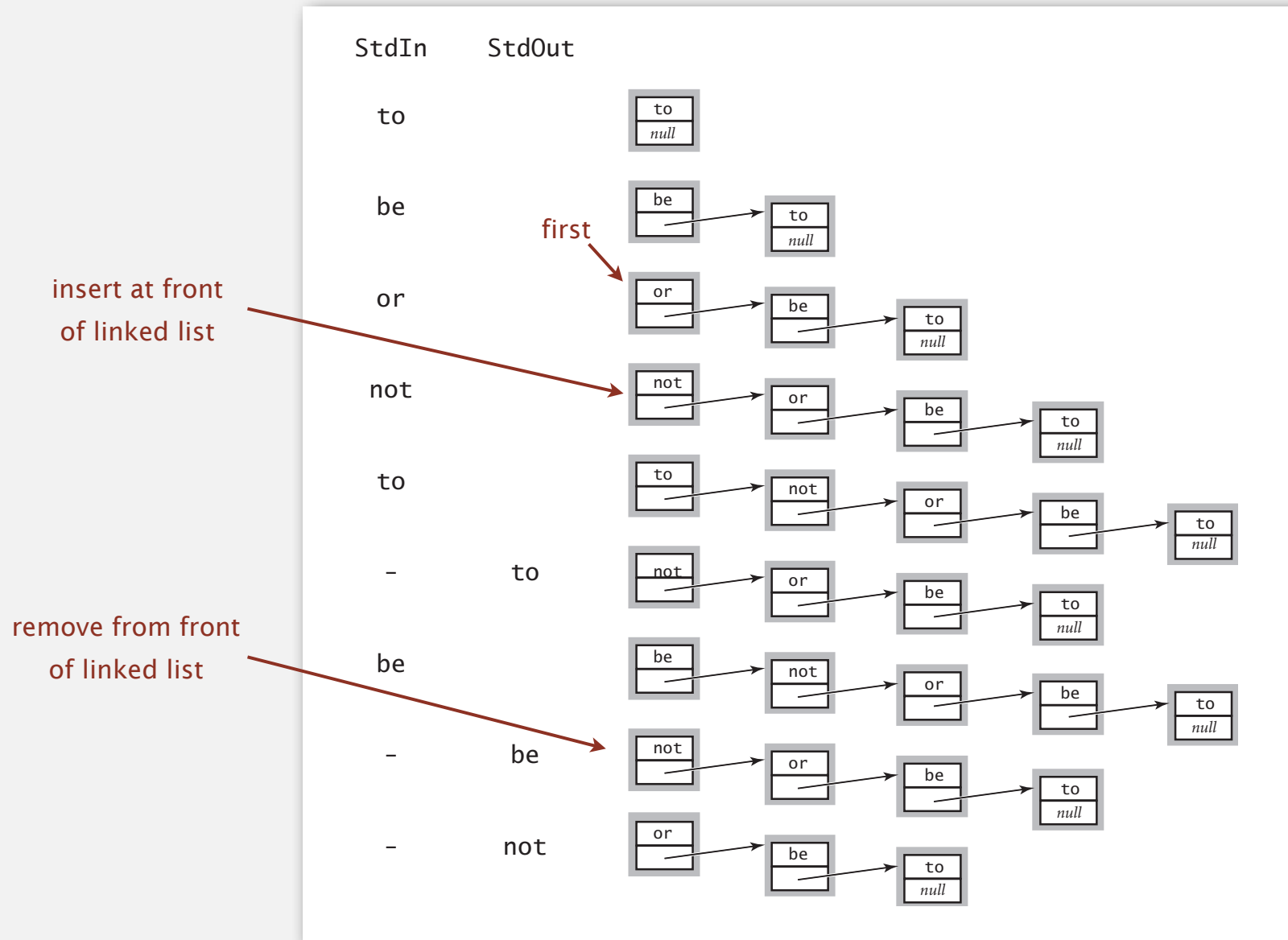
```
% more tobe.txt
to be or not to - be - - that - - - is

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



Stack: linked-list representation

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



Stack pop: linked-list implementation

inner class

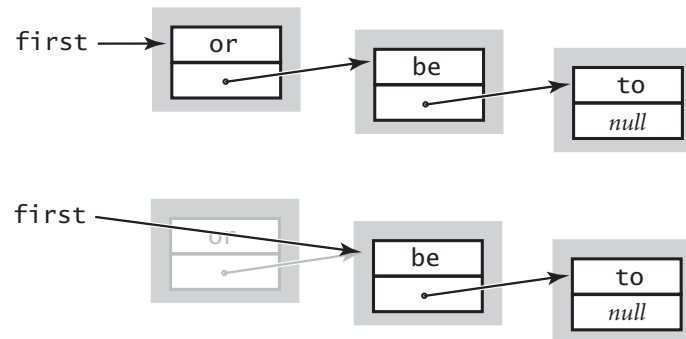
```
private class Node
{
    String item;
    Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

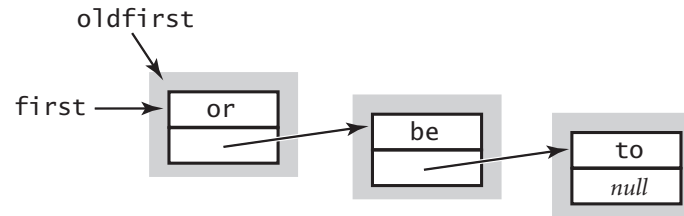

Stack push: linked-list implementation

inner class

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

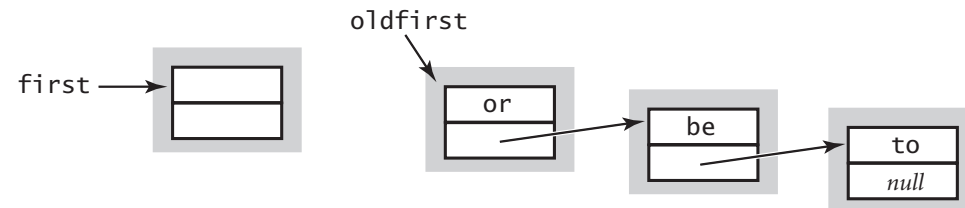
save a link to the list

```
Node oldfirst = first;
```



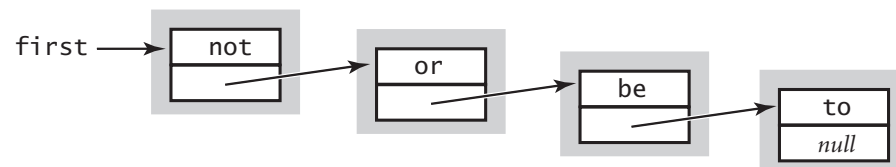
create a new node for the beginning

```
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
    {
        String item;
        Node next;
    }

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

    public void push(String item)
    {
        ...
    }

    public String pop()
    {
        ...
    }
}
```

← private inner class
(access modifiers don't matter)

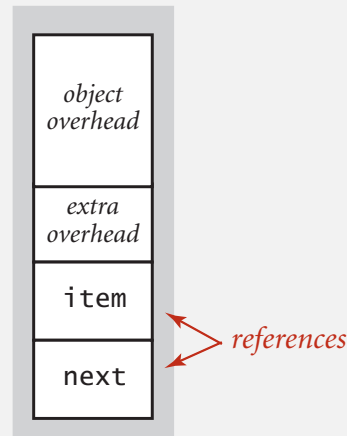
Stack: linked-list implementation performance

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)

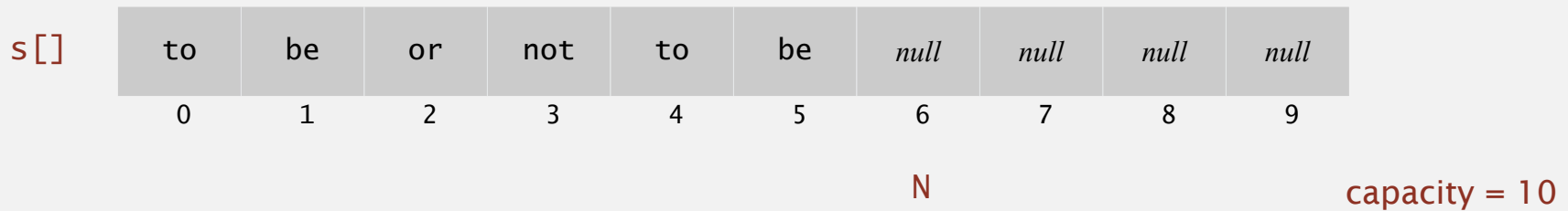
40 bytes per stack node

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

Stack: array implementation

Array implementation of a stack.

- 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]

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; }

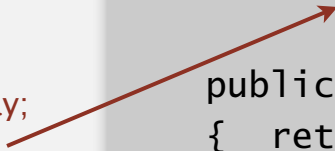
    public void push(String item)
    { s[N++] = item; }

    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 inserted.

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

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

loitering

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

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



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

1.3 BAGS, QUEUES, AND STACKS

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

Stack: resizing-array implementation

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

Q. How to grow and shrink array?

First try.

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

Too expensive.

- Need to copy all items to a new array.
- Inserting first N items takes time proportional to $1 + 2 + \dots + N \sim N^2 / 2$.

↑
infeasible for large N

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;
}
```

see next slide



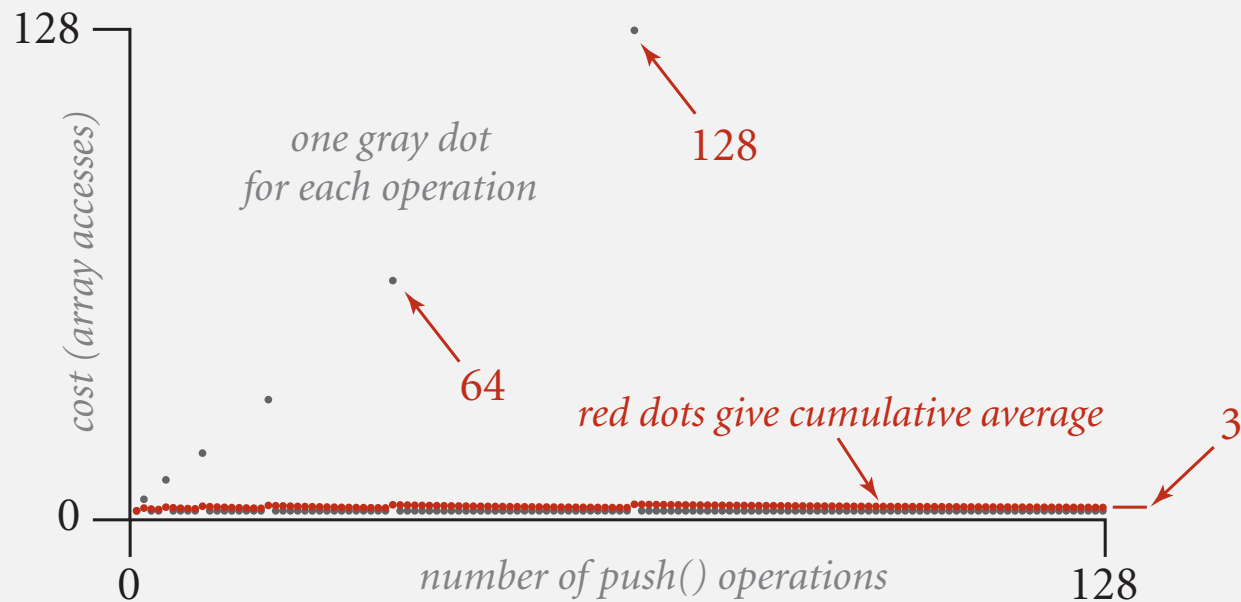
Consequence. Inserting first N items takes time proportional to N (not N^2).

Stack: amortized cost of adding to a stack

Cost of inserting first N 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

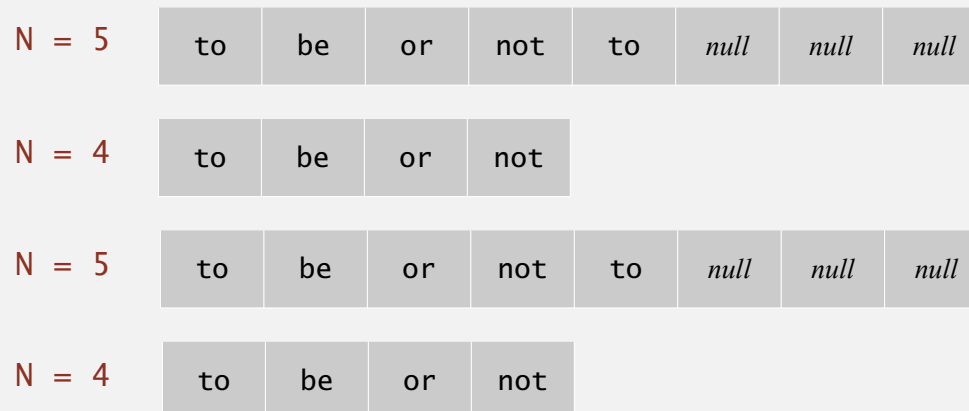
Q. How to shrink array?

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 trace

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

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

Stack resizing-array implementation: performance

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

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

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

doubling and halving operations

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

Stack resizing-array implementation: memory usage

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

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

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

8 bytes (reference to array)
24 bytes (array overhead)
8 bytes \times array size
4 bytes (int)
4 bytes (padding)

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

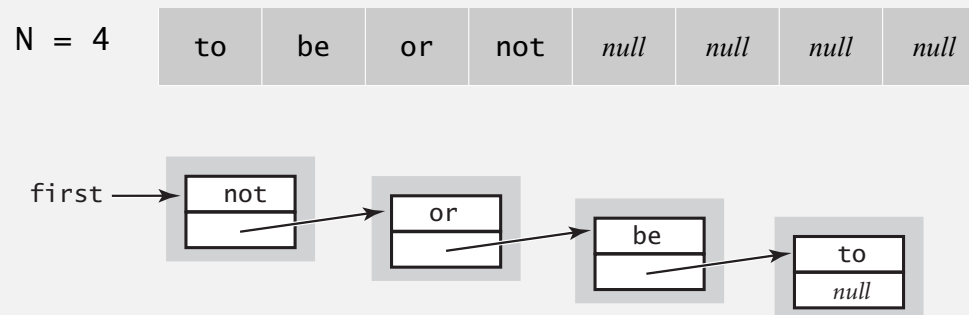
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

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.





1.3 BAGS, QUEUES, AND STACKS

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

Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings()
```

create an empty queue

```
    void enqueue(String item)
```

insert a new string onto queue

```
    String dequeue()
```

*remove and return the string
least recently added*

```
    boolean isEmpty()
```

is the queue empty?

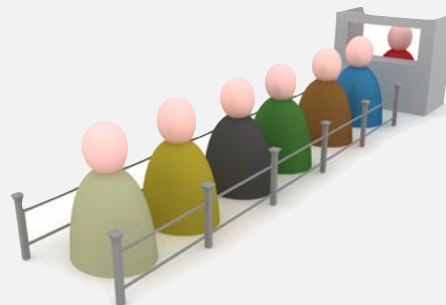
```
    int size()
```

number of strings on the queue

enqueue

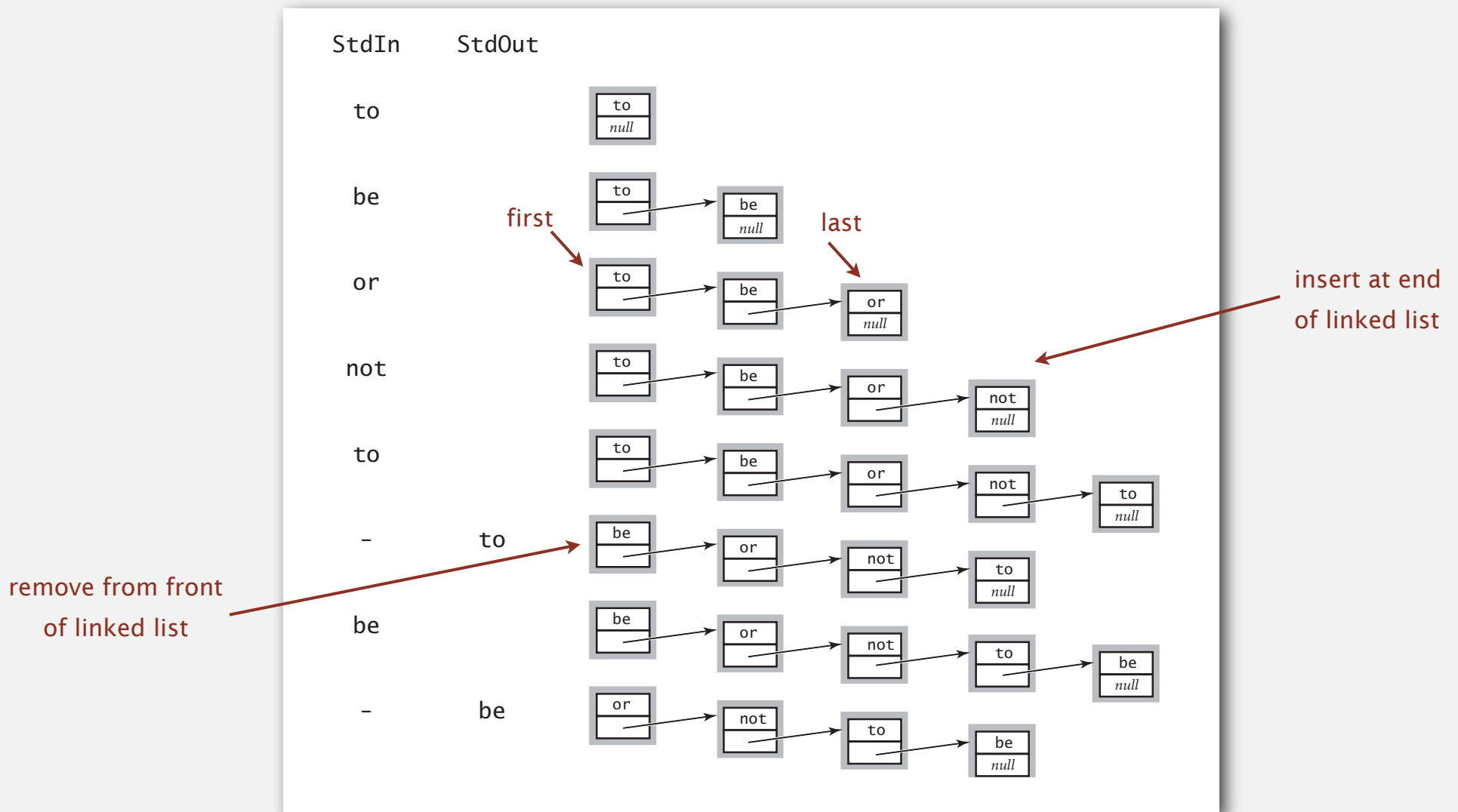


dequeue



Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list;
insert/remove from opposite ends.



Queue dequeue: linked-list implementation

inner class

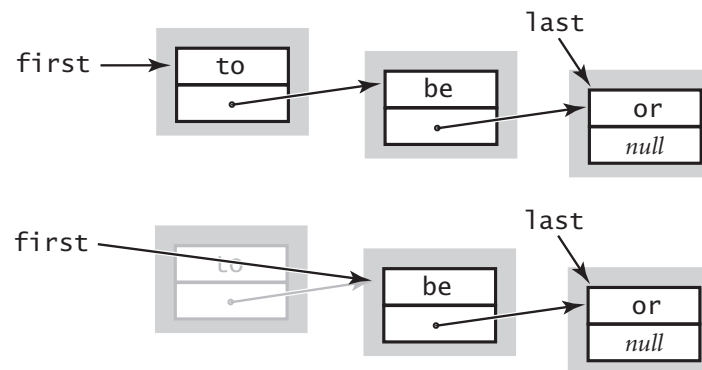
```
private class Node
{
    String item;
    Node next;
}
```

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().

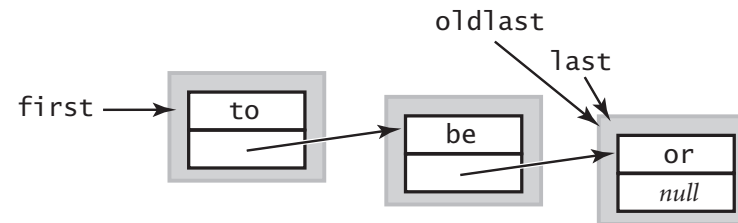
Queue enqueue: linked-list implementation

inner class

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

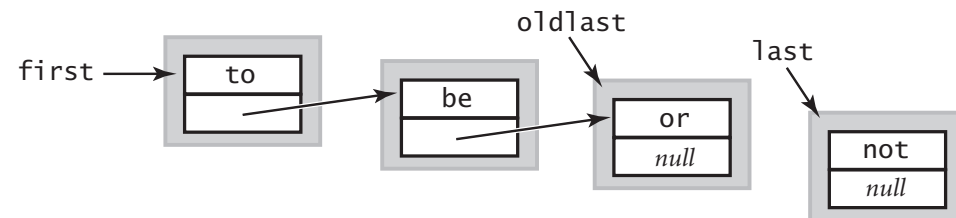
save a link to the last node

```
Node oldlast = last;
```



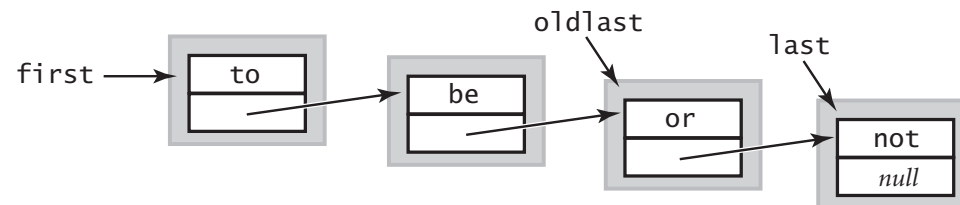
create a new node for the end

```
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;

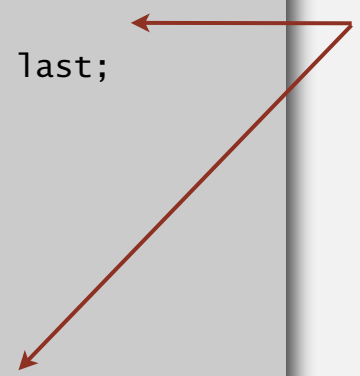
    private class Node
    { /* same as in LinkedStackOfStrings */ }

    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





1.3 BAGS, QUEUES, AND STACKS

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

Parameterized stack

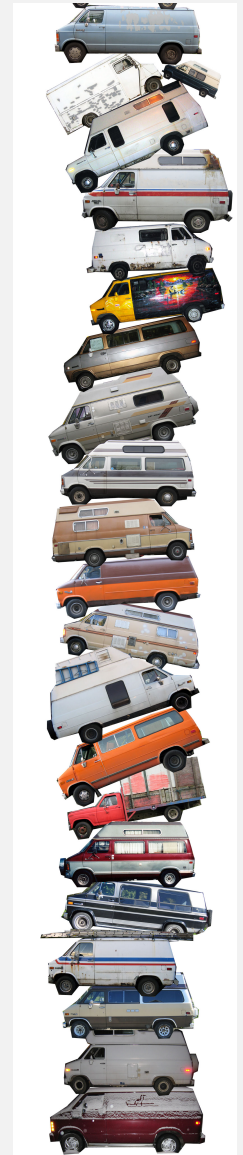
We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans,

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#*\$! most reasonable approach until Java 1.5.



Parameterized stack

We implemented: StackOfStrings.

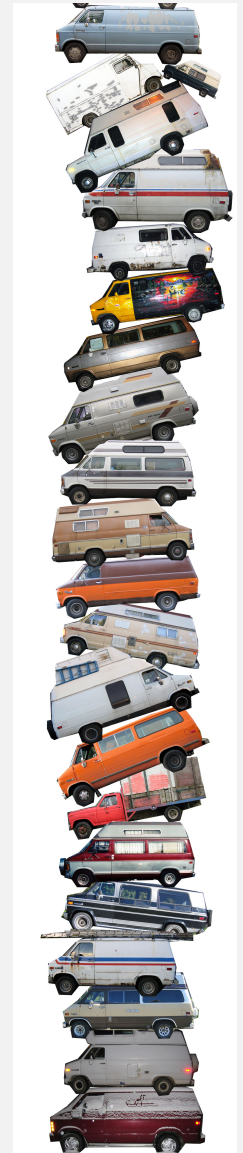
We also want: StackOfURLs, StackOfInts, StackOfVans,

Attempt 2. Implement a stack with items of type Object.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = (Apple) (s.pop());
```

run-time error



Parameterized stack

We implemented: StackOfStrings.

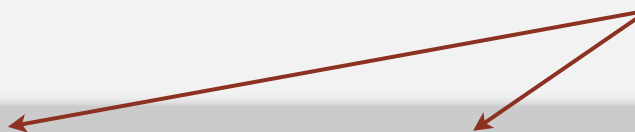
We also want: StackOfURLs, StackOfInts, StackOfVans,

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

```
Stack<Apple> s = new Stack<Apple>();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = s.pop();
```

type parameter



compile-time error



Guiding principles. Welcome compile-time errors; avoid run-time errors.

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
```

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

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

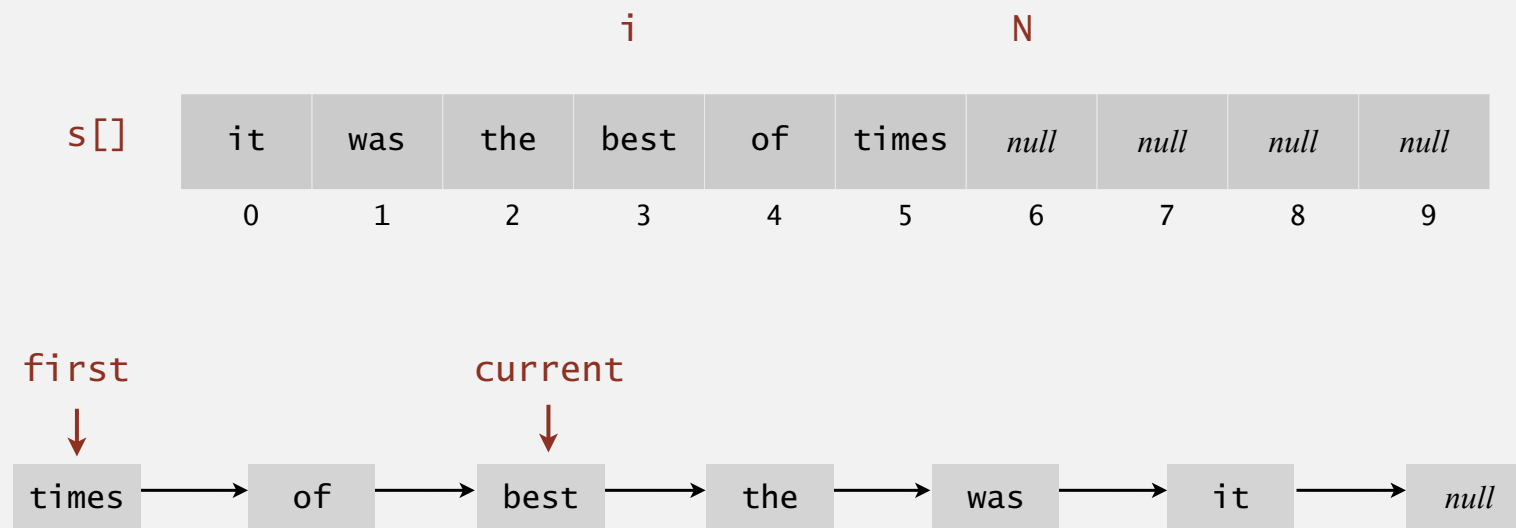


1.3 BAGS, QUEUES, AND STACKS

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

Iteration

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



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

Iterators

Q. What is an `Iterable` ?

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

Q. What is an `Iterator` ?

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

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

Iterable interface

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

Iterator interface

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

“foreach” statement (shorthand)

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

equivalent code (longhand)

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

Stack iterator: linked-list implementation

```
import java.util.Iterator;

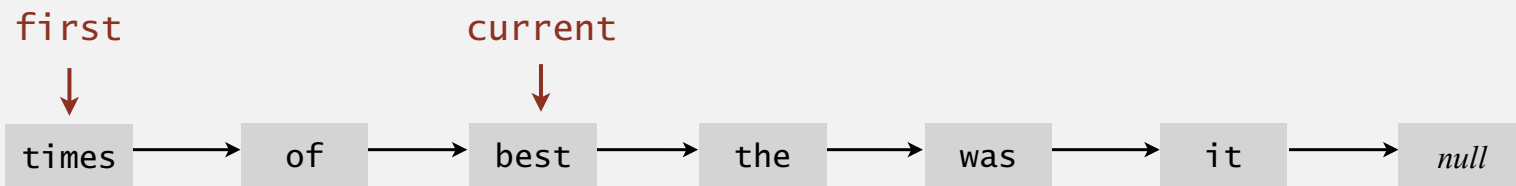
public class Stack<Item> implements Iterable<Item>
{
    ...

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

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

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



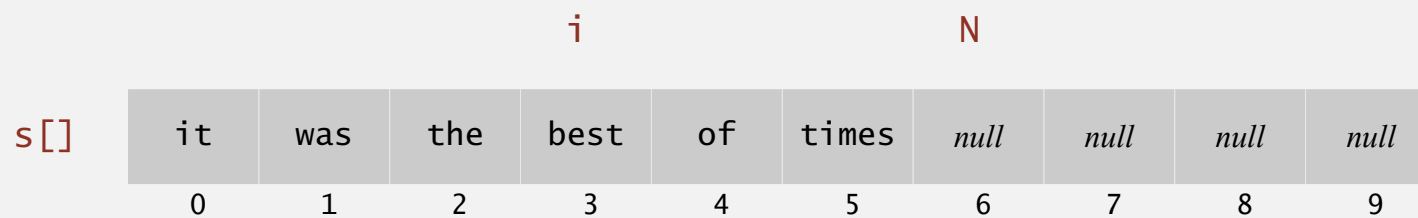
Stack iterator: array implementation

```
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]; }
    }
}
```



Iteration: concurrent modification

Q. What if client modifies the data structure while iterating?

A. A fail-fast iterator throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
    stack.push(s);
```

To detect:

- Count total number of `push()` and `pop()` operations in Stack.
- Save counts in `*Iterator` subclass upon creation.
- If, when calling `next()` and `hasNext()`, the current counts do not equal the saved counts, throw exception.



1.3 BAGS, QUEUES, AND STACKS

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


Java collections library

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

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

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

`java.util.LinkedList` uses linked list.

 caveat: only some
operations are efficient

Java collections library

`java.util.Stack`.

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

Java 1.3 bug report (June 27, 2001)

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

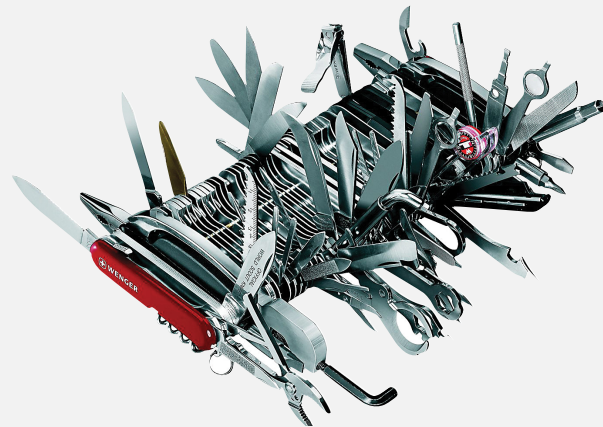
status (closed, will not fix)

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

Java collections library

`java.util.Stack`.

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



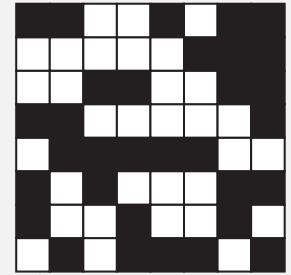
`java.util.Queue`. An interface, not an implementation of a queue.

Best practices. Use our implementations of Stack, Queue, and Bag.

War story (from Assignment 1)

Generate random open sites in an N -by- N percolation system.

- Jenny: pick (i, j) at random; if already open, repeat.
Takes $\sim c_1 N^2$ seconds.
- Kenny: create a `java.util.ArrayList` of N^2 closed sites.
Pick an index at random and delete.
Takes $\sim c_2 N^4$ seconds.



Why is my program so slow?



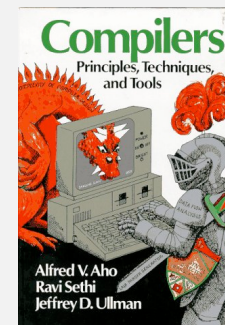
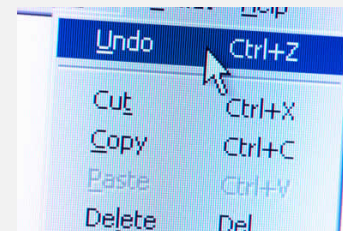
Kenny

Lesson. Don't use a library until you understand its API!

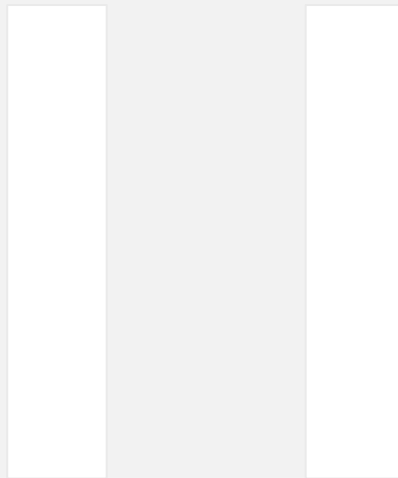
This course. Can't use a library until we've implemented it in class.

Stack applications

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



Dijkstra's two-stack algorithm demo



value stack

operator stack

infix expression
(fully parenthesized)

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

operand

operator

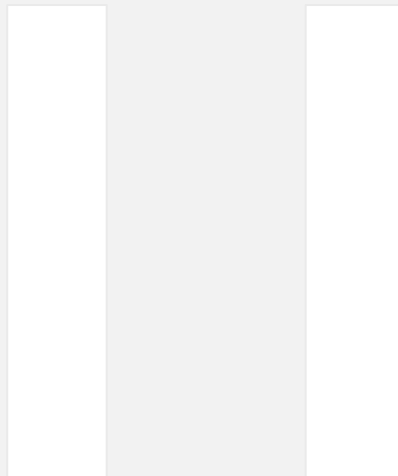
Dijkstra's two-stack algorithm

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)



operand

operator

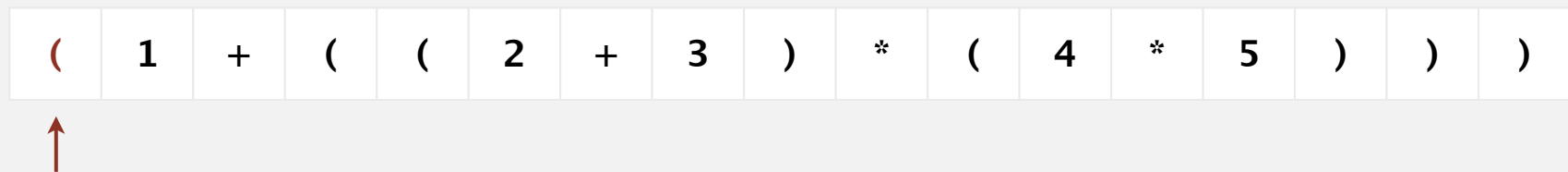
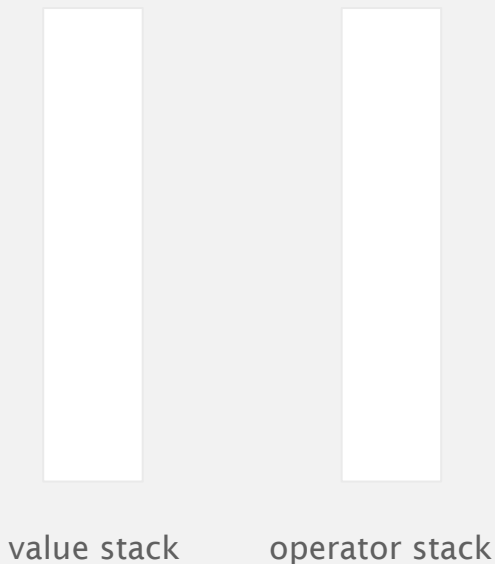
Dijkstra's two-stack algorithm

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.



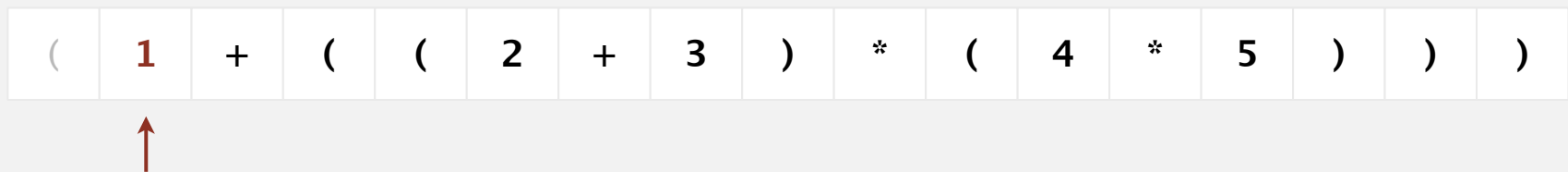
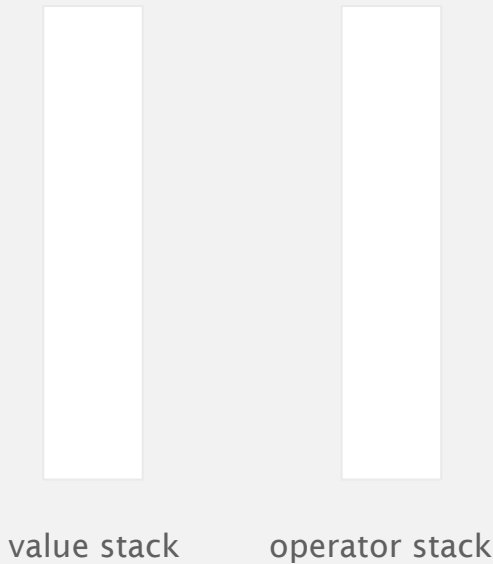
Dijkstra's two-stack algorithm

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.



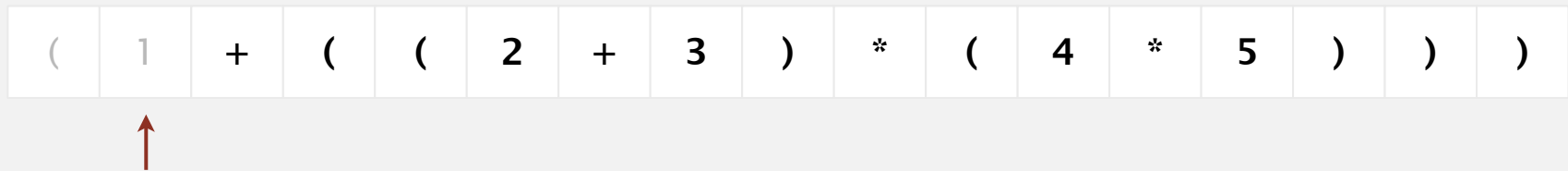
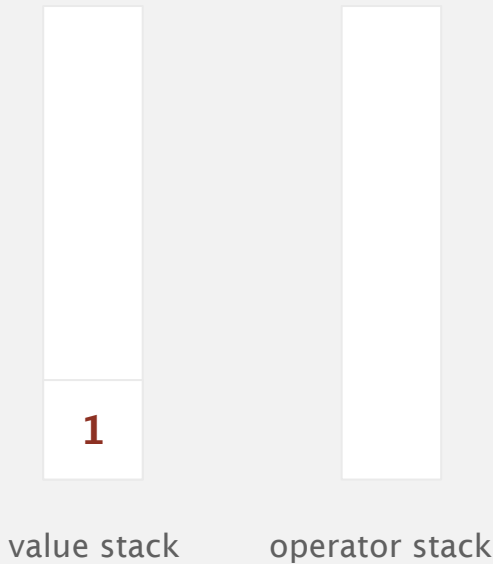
Dijkstra's two-stack algorithm

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.



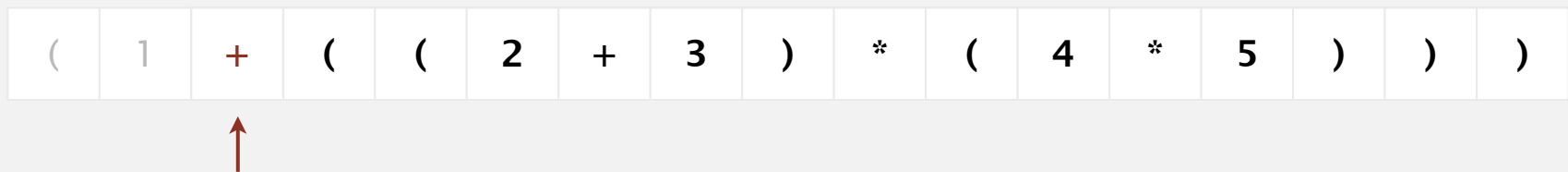
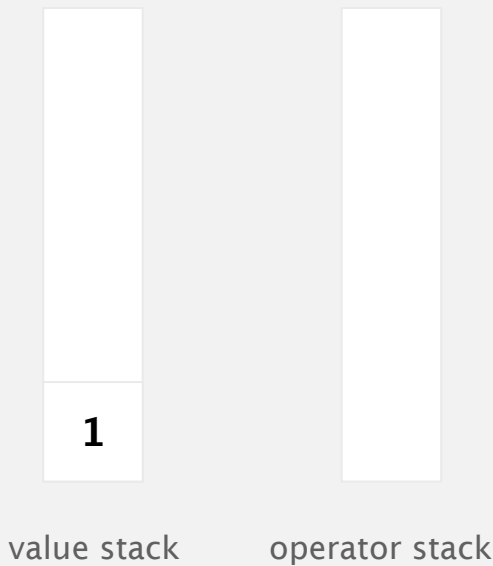
Dijkstra's two-stack algorithm

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.



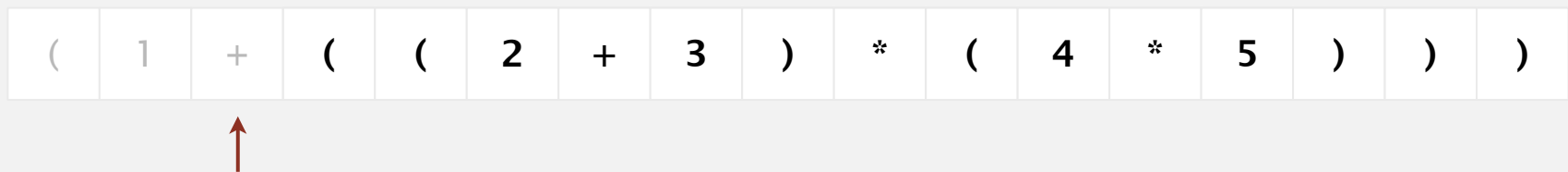
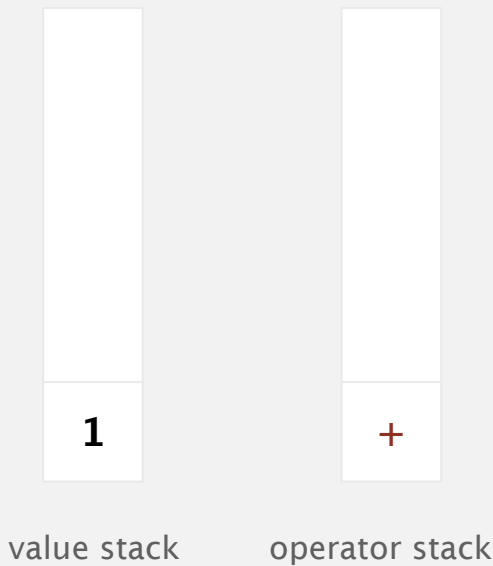
Dijkstra's two-stack algorithm

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.



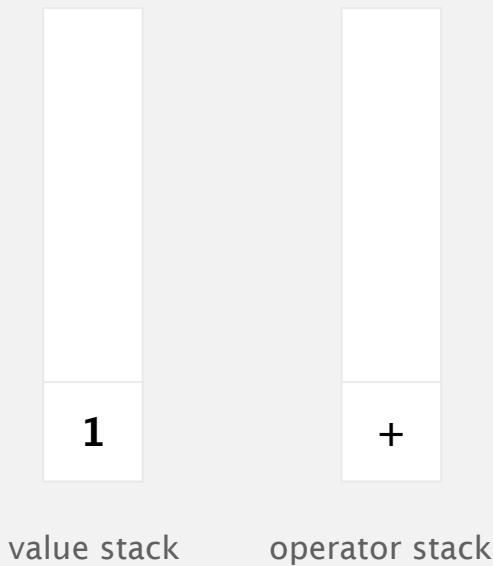
Dijkstra's two-stack algorithm

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.



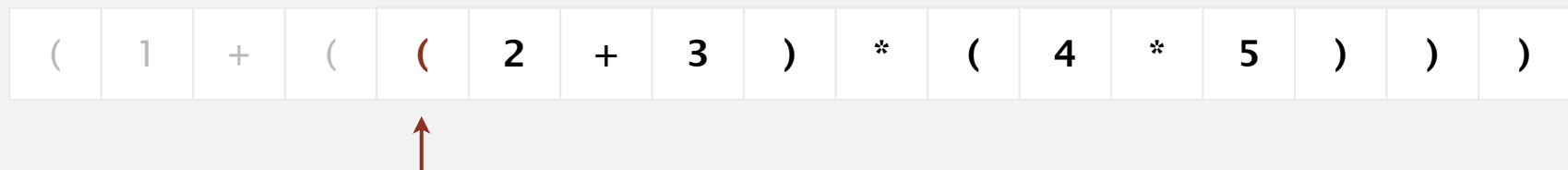
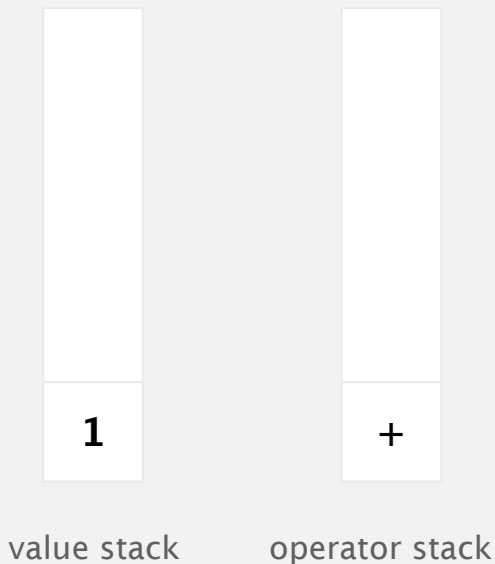
Dijkstra's two-stack algorithm

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.



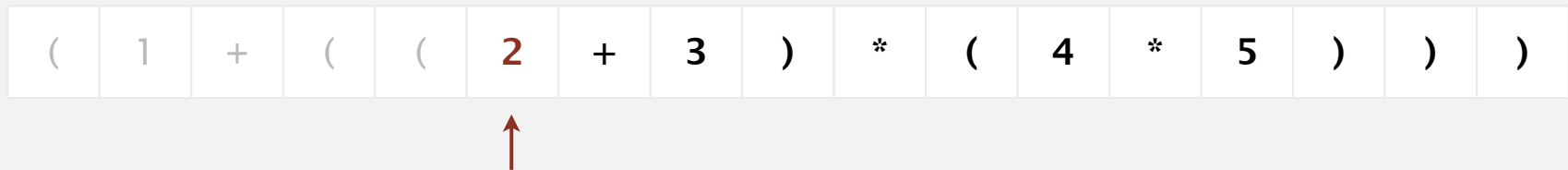
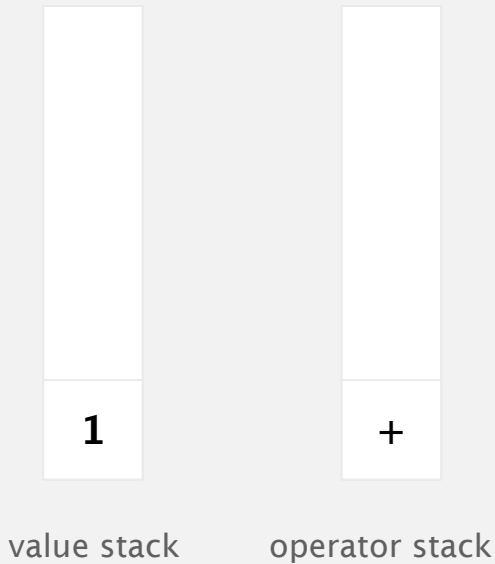
Dijkstra's two-stack algorithm

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.



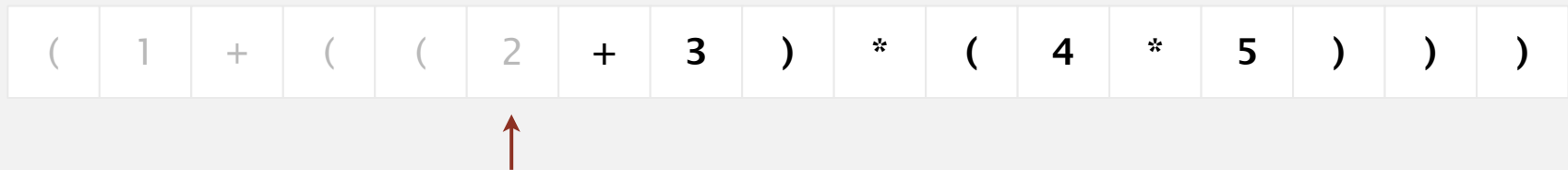
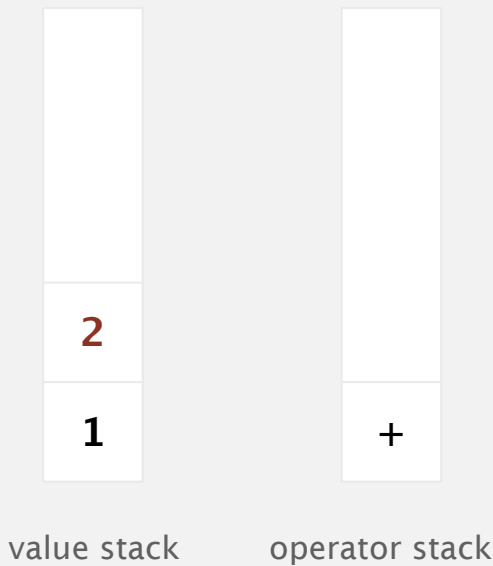
Dijkstra's two-stack algorithm

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.



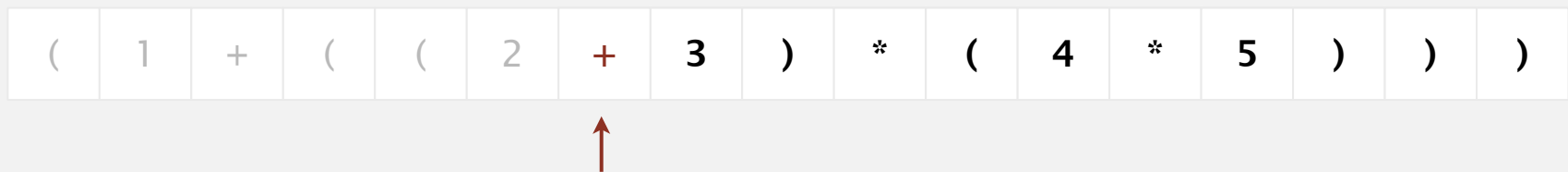
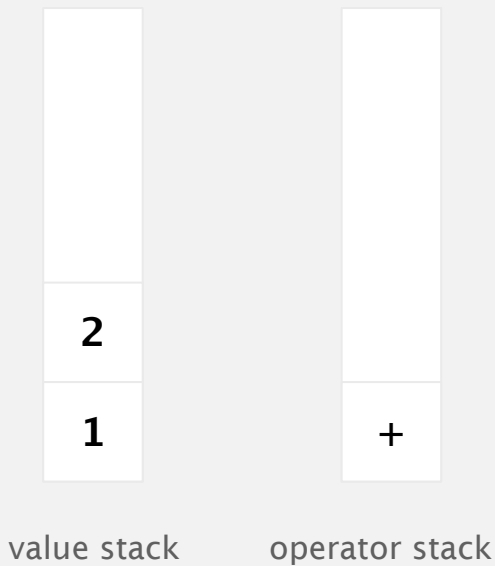
Dijkstra's two-stack algorithm

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.



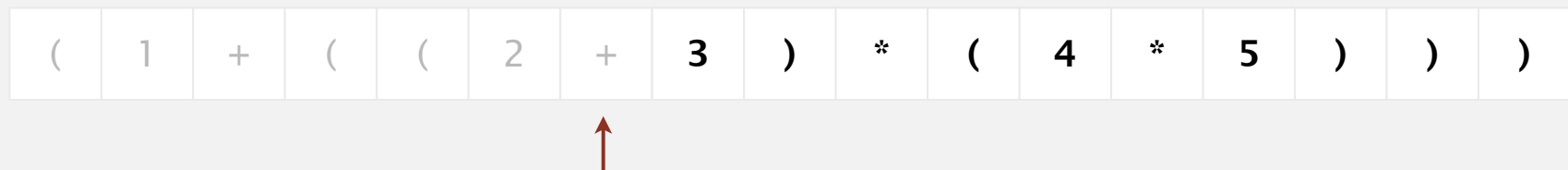
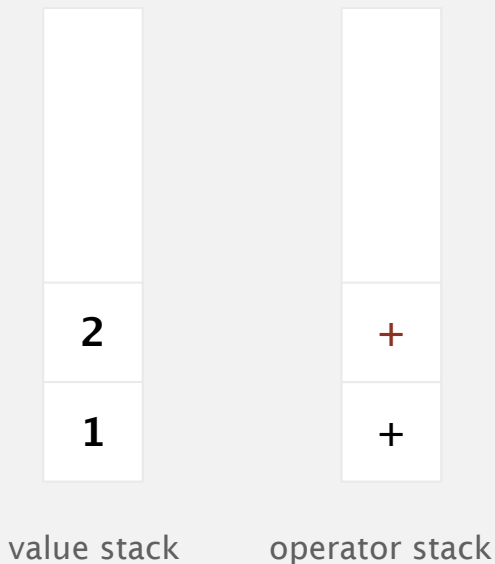
Dijkstra's two-stack algorithm

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.



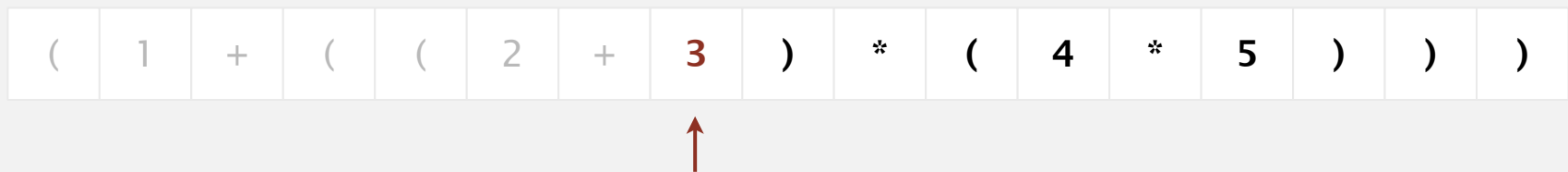
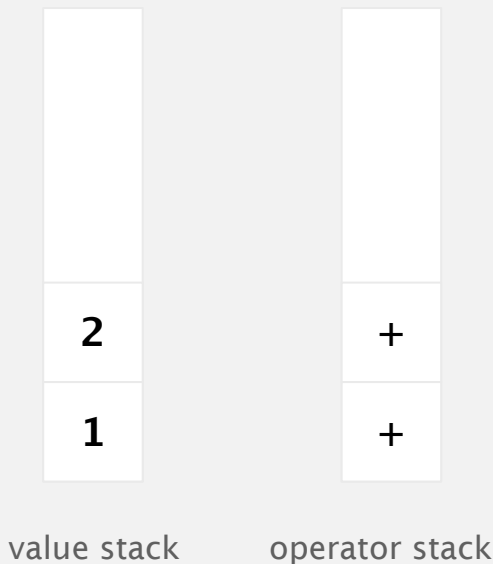
Dijkstra's two-stack algorithm

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.



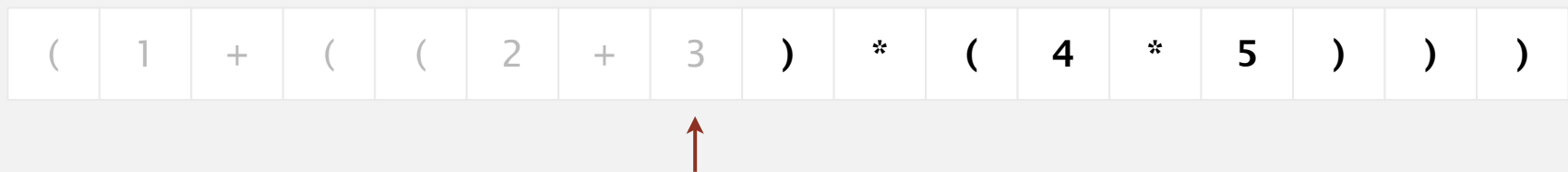
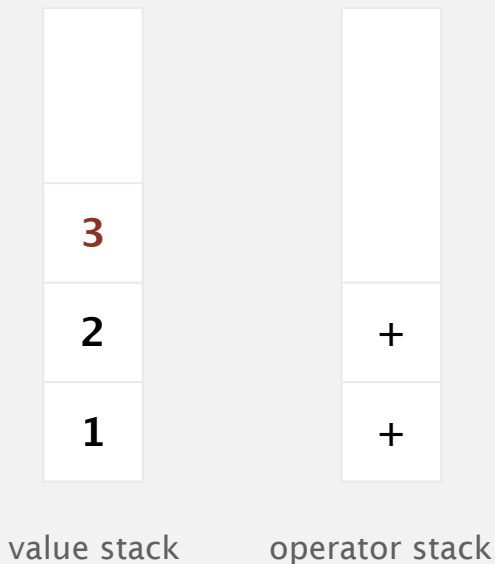
Dijkstra's two-stack algorithm

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.



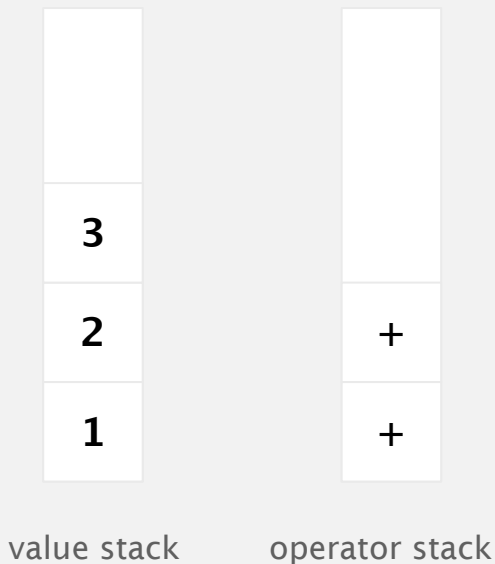
Dijkstra's two-stack algorithm

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.



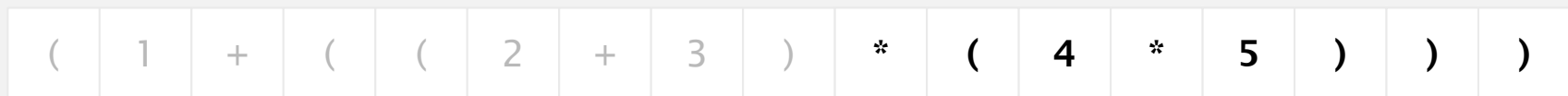
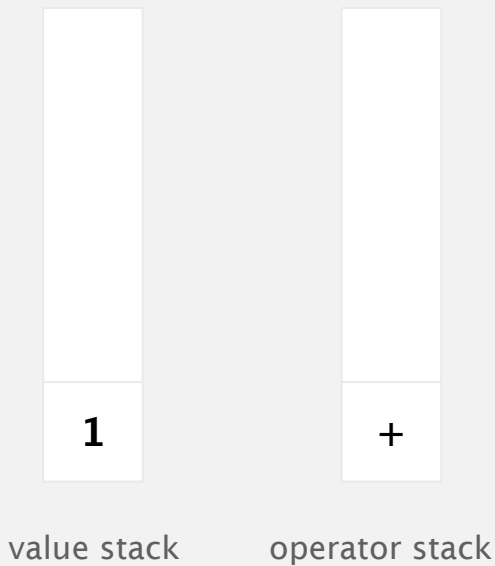
Dijkstra's two-stack algorithm

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.



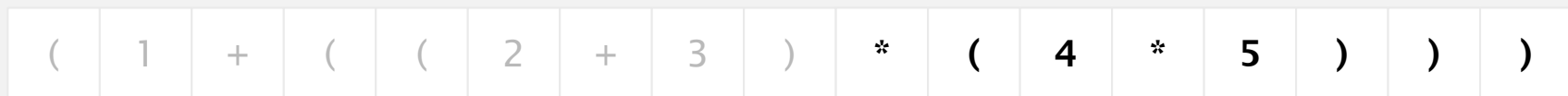
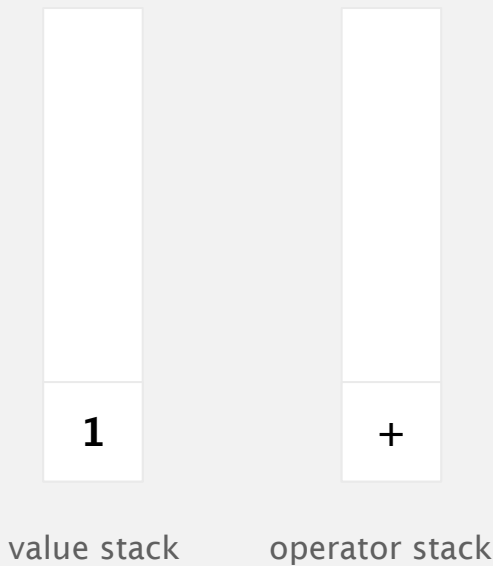
Dijkstra's two-stack algorithm

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.



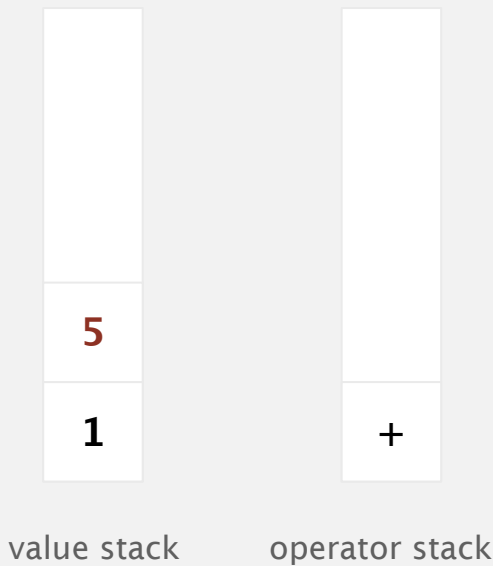
Dijkstra's two-stack algorithm

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.



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



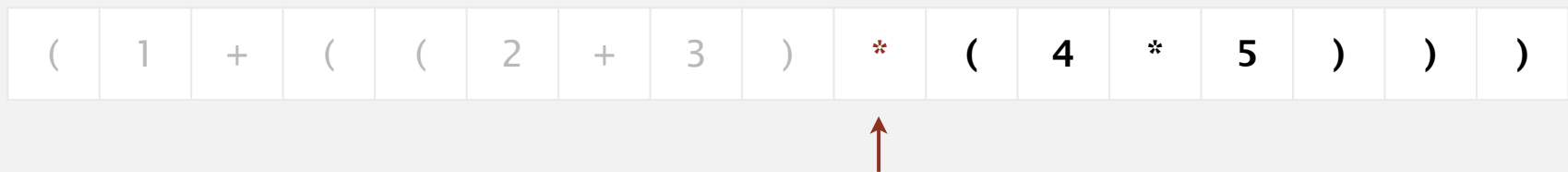
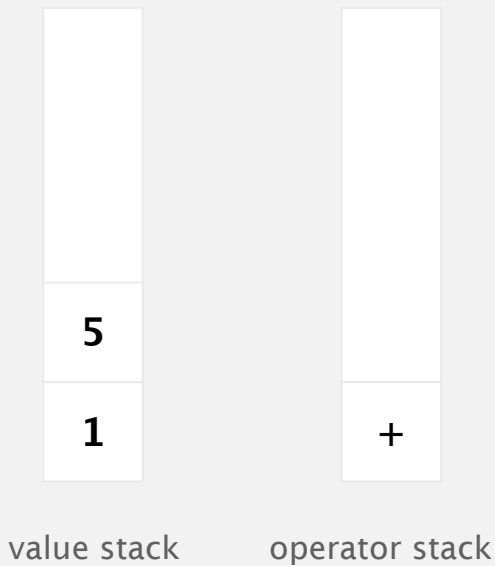
Dijkstra's two-stack algorithm

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.



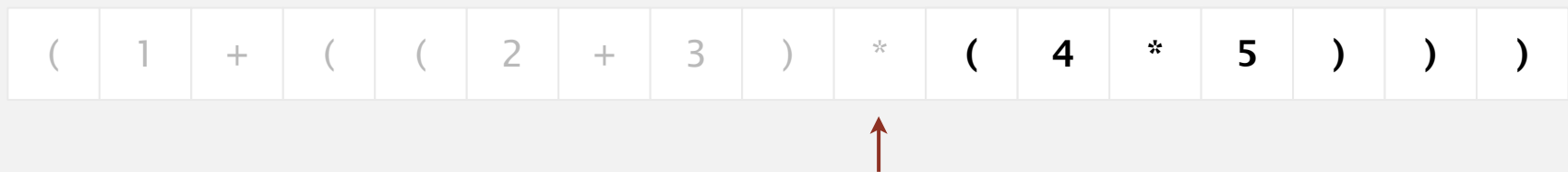
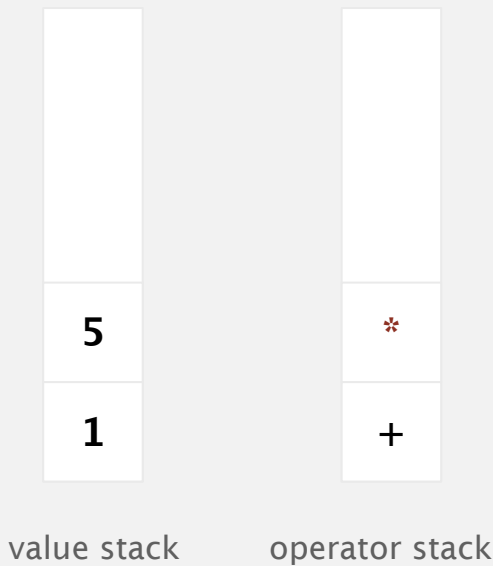
Dijkstra's two-stack algorithm

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.



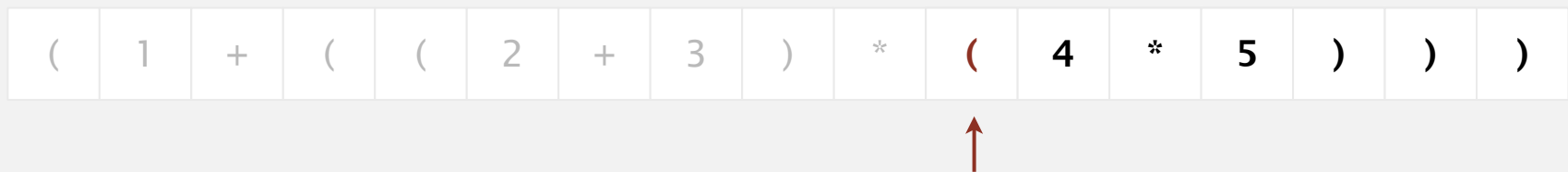
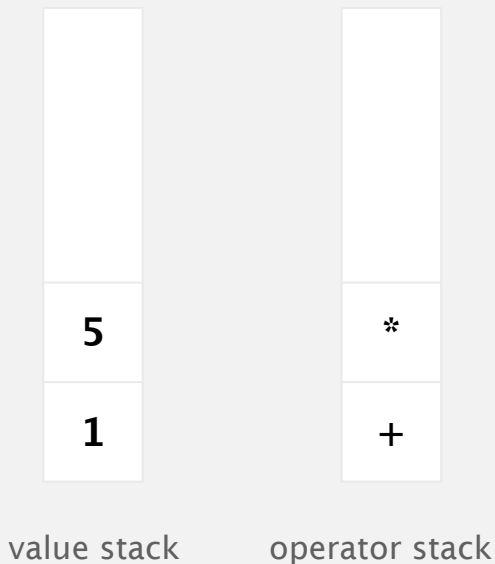
Dijkstra's two-stack algorithm

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.



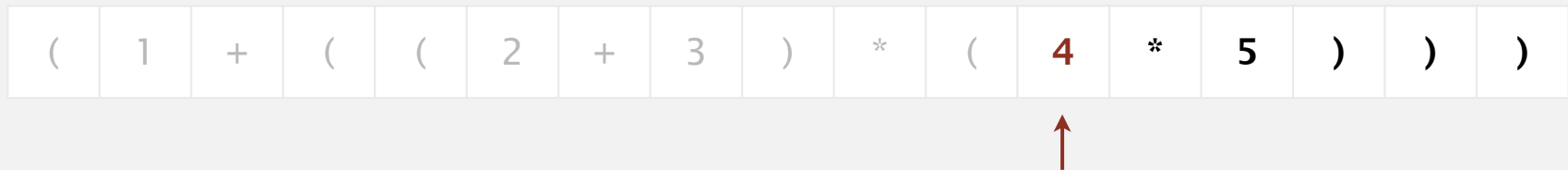
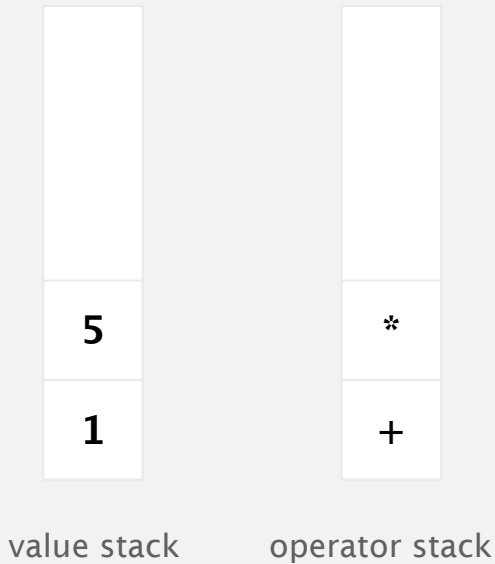
Dijkstra's two-stack algorithm

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.



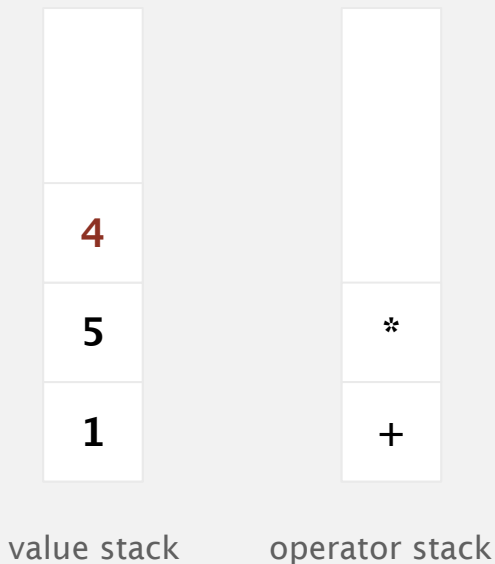
Dijkstra's two-stack algorithm

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.



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



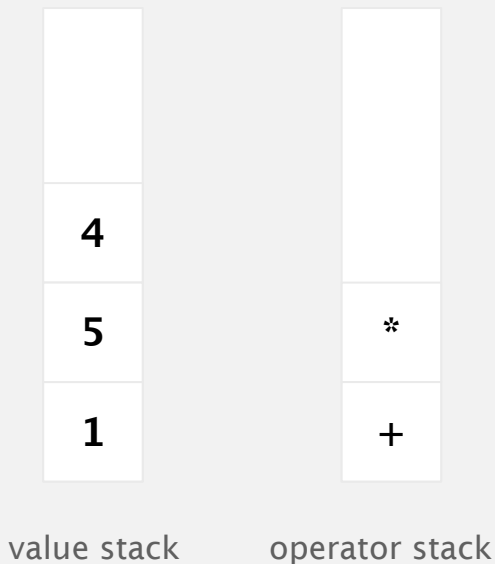
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

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



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



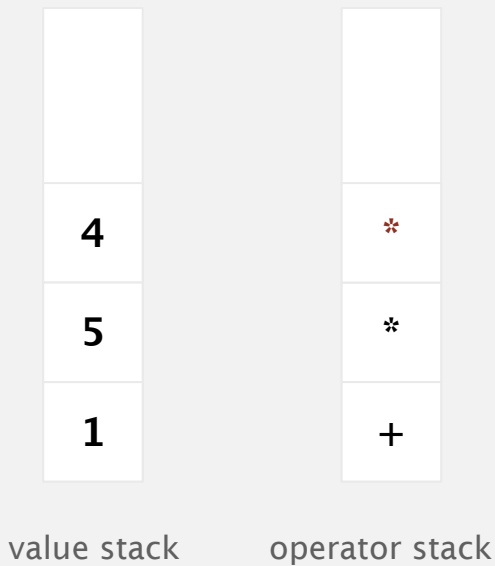
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

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



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



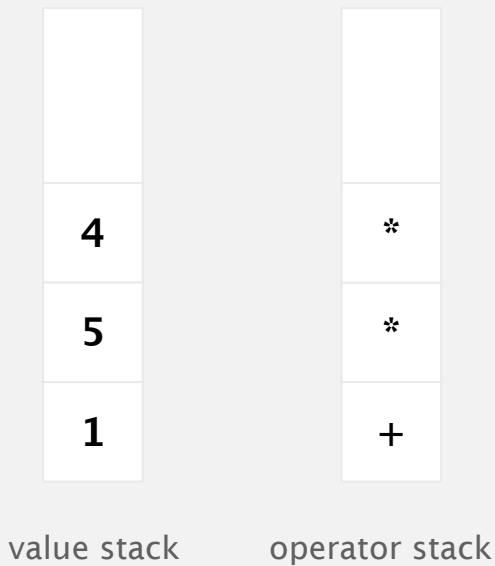
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

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



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



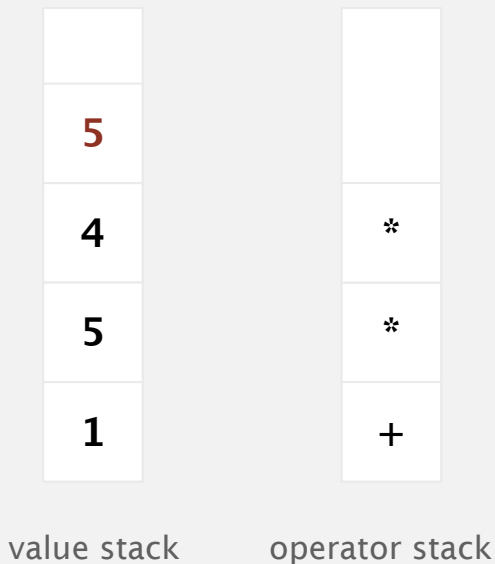
Dijkstra's two-stack algorithm

Value: push onto the value stack.

Operator: push onto the operator stack.

Left parenthesis: ignore.

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



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



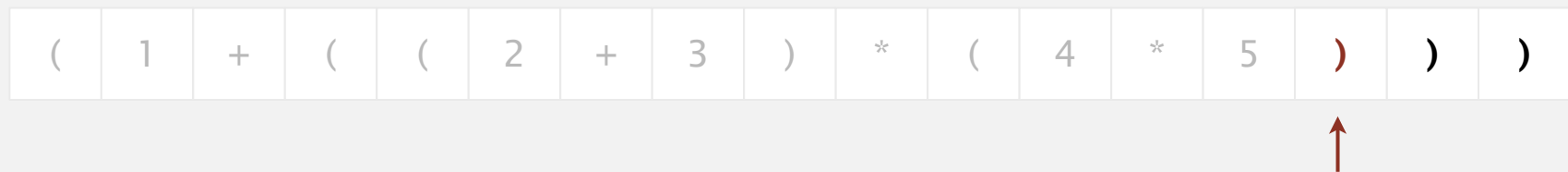
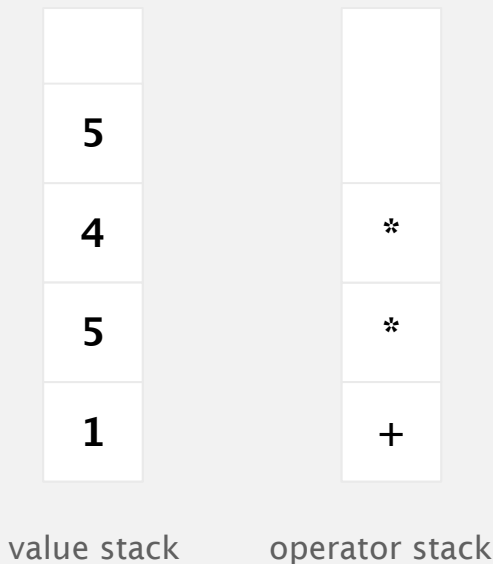
Dijkstra's two-stack algorithm

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.



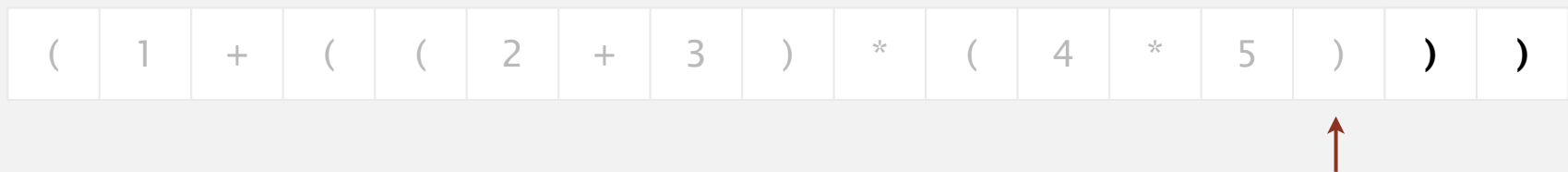
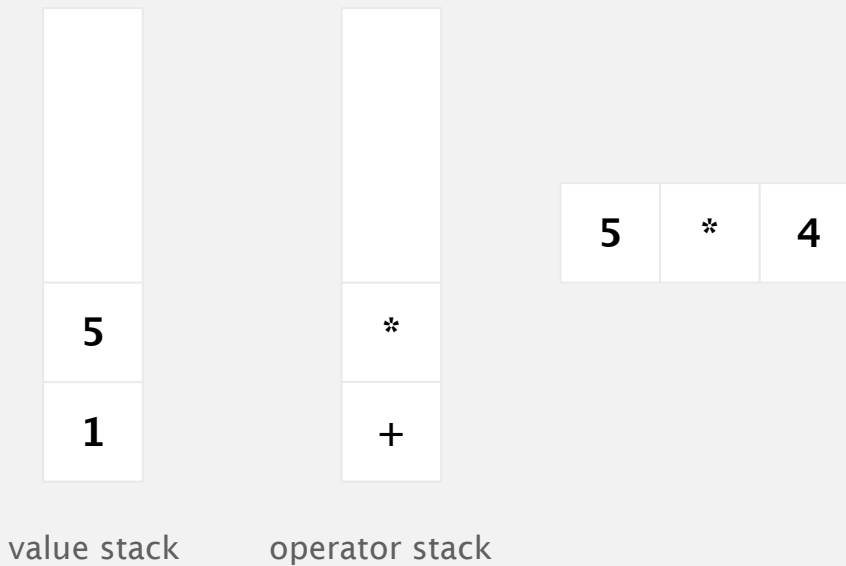
Dijkstra's two-stack algorithm

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.



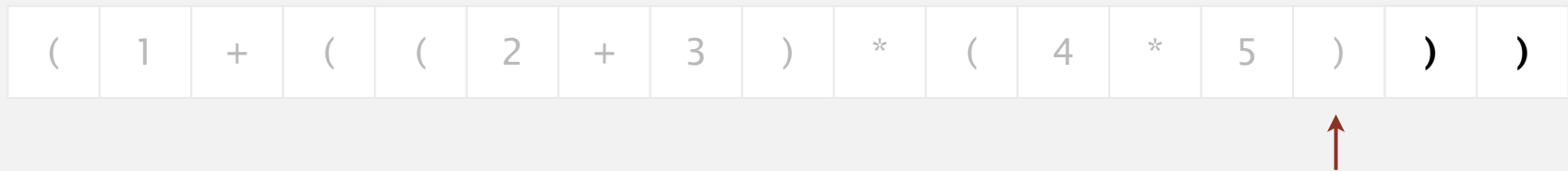
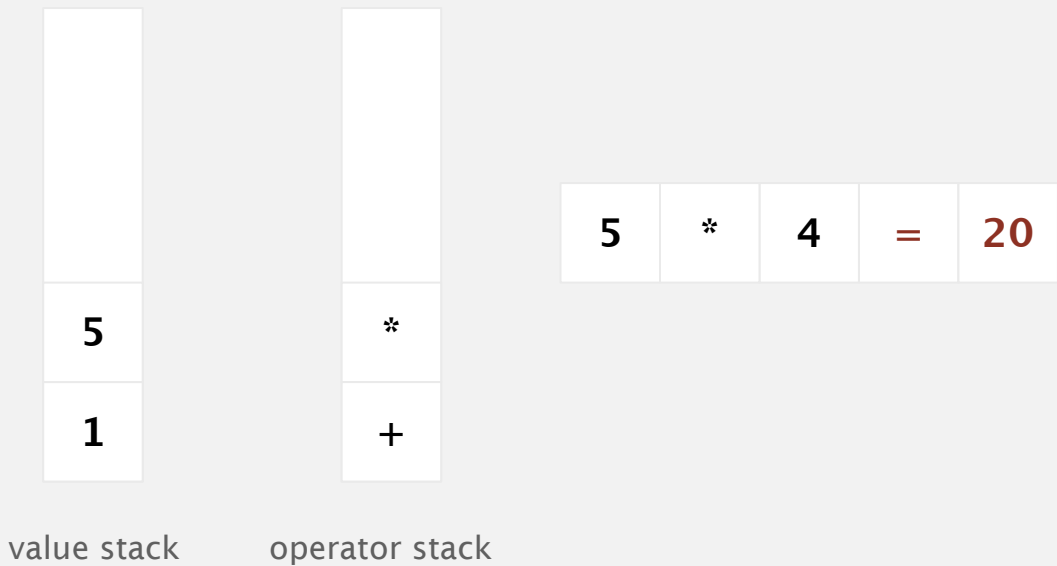
Dijkstra's two-stack algorithm

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.



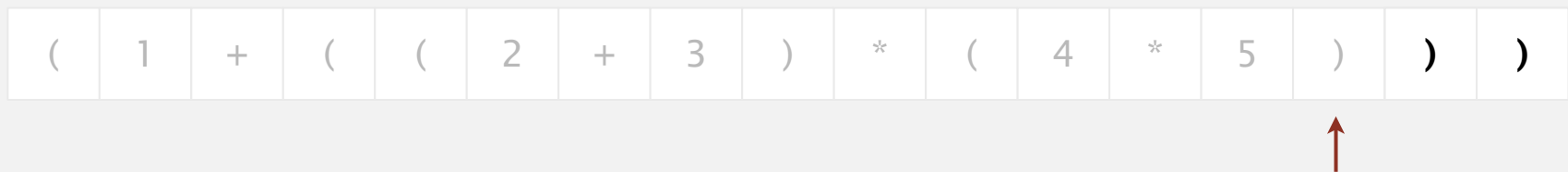
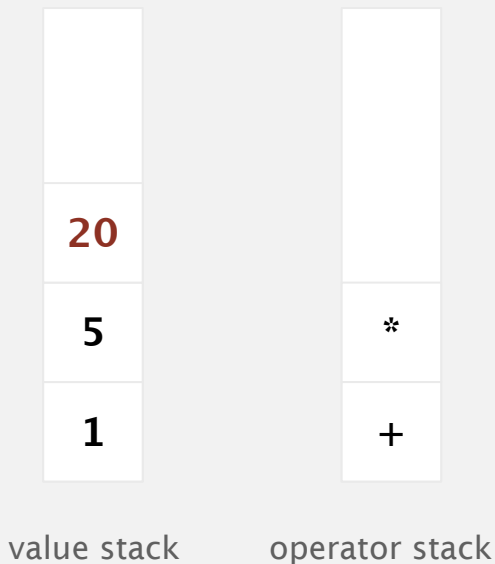
Dijkstra's two-stack algorithm

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.



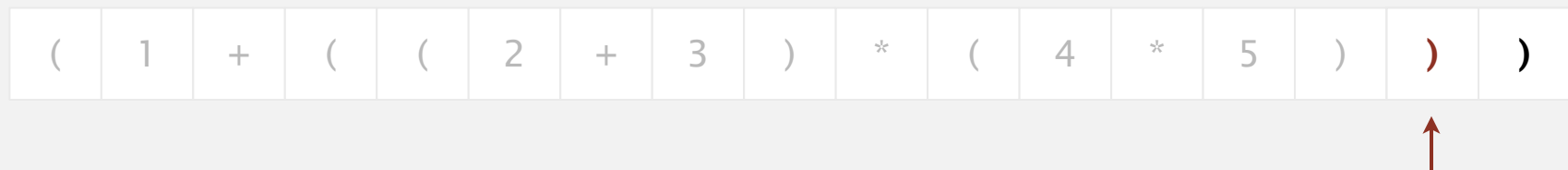
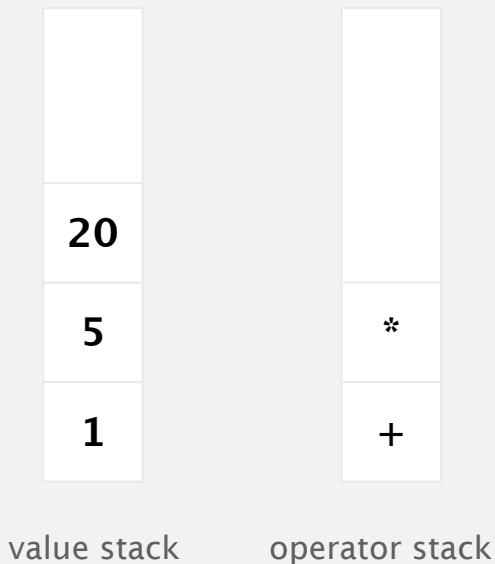
Dijkstra's two-stack algorithm

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.



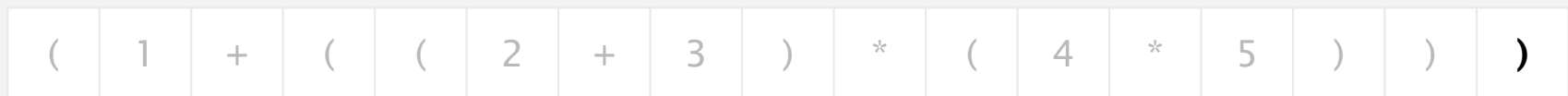
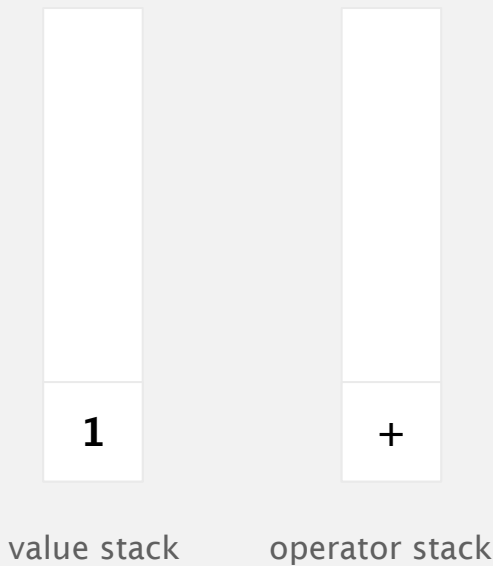
Dijkstra's two-stack algorithm

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.



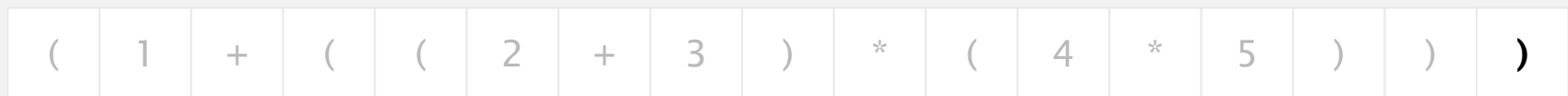
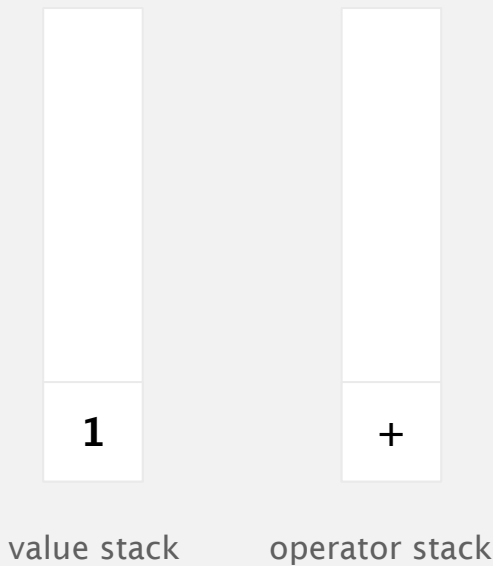
Dijkstra's two-stack algorithm

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.



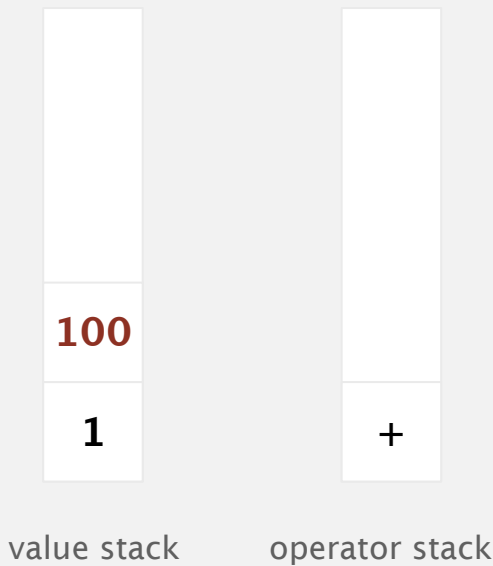
Dijkstra's two-stack algorithm

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.



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



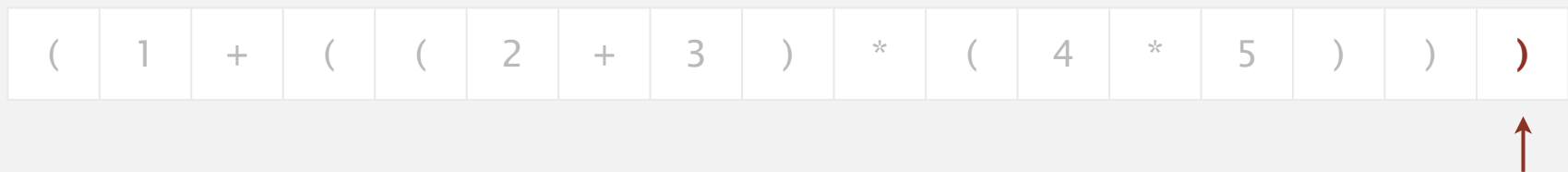
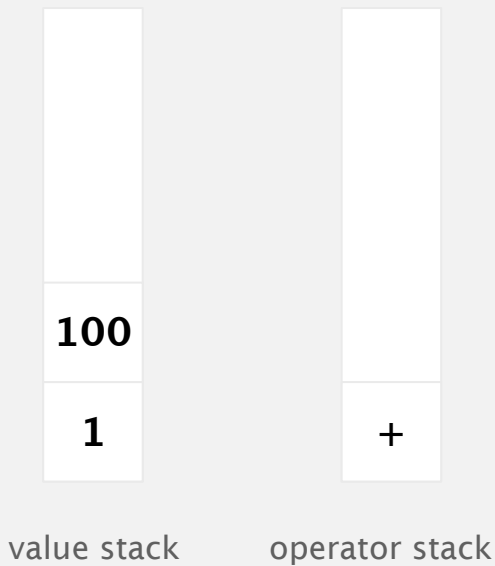
Dijkstra's two-stack algorithm

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.



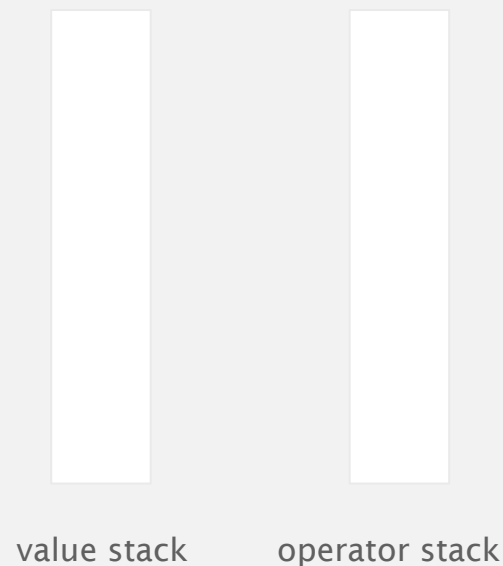
Dijkstra's two-stack algorithm

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)))



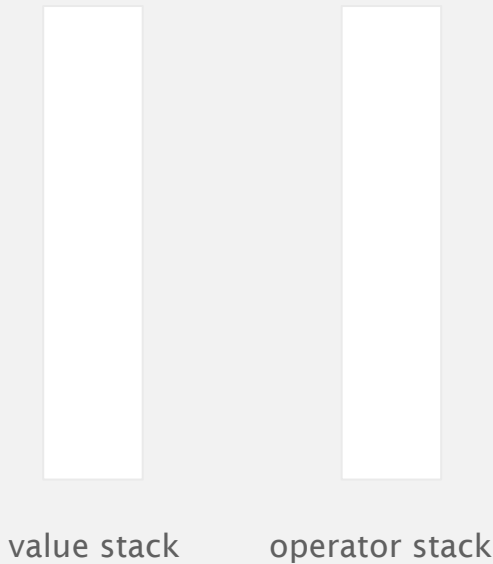
Dijkstra's two-stack algorithm

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.



100	+	1	=	101
-----	---	---	---	-----

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



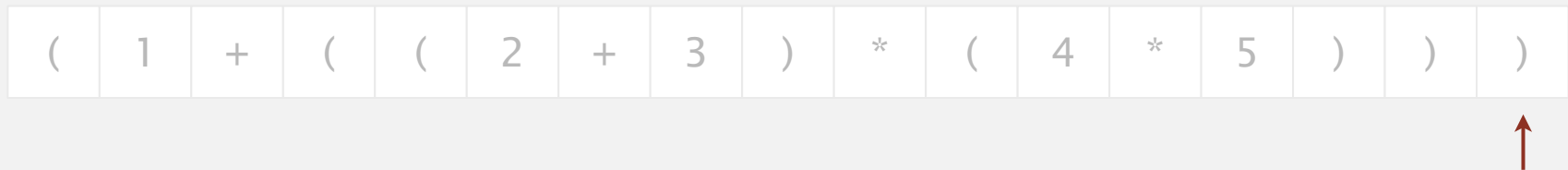
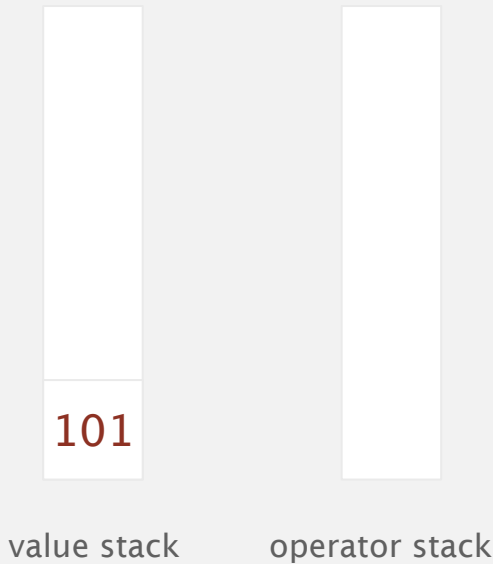
Dijkstra's two-stack algorithm

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.



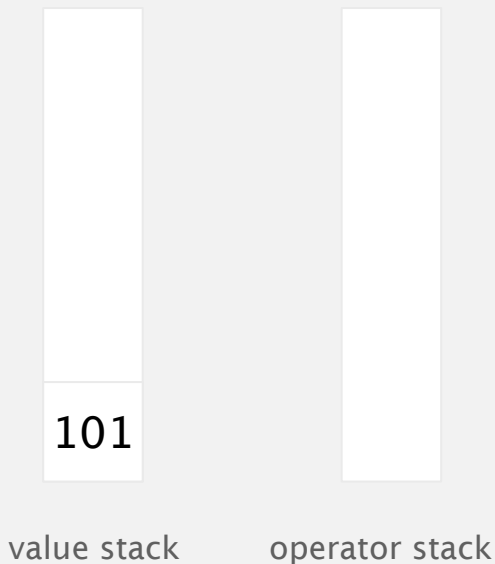
Dijkstra's two-stack algorithm

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.

101

result

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