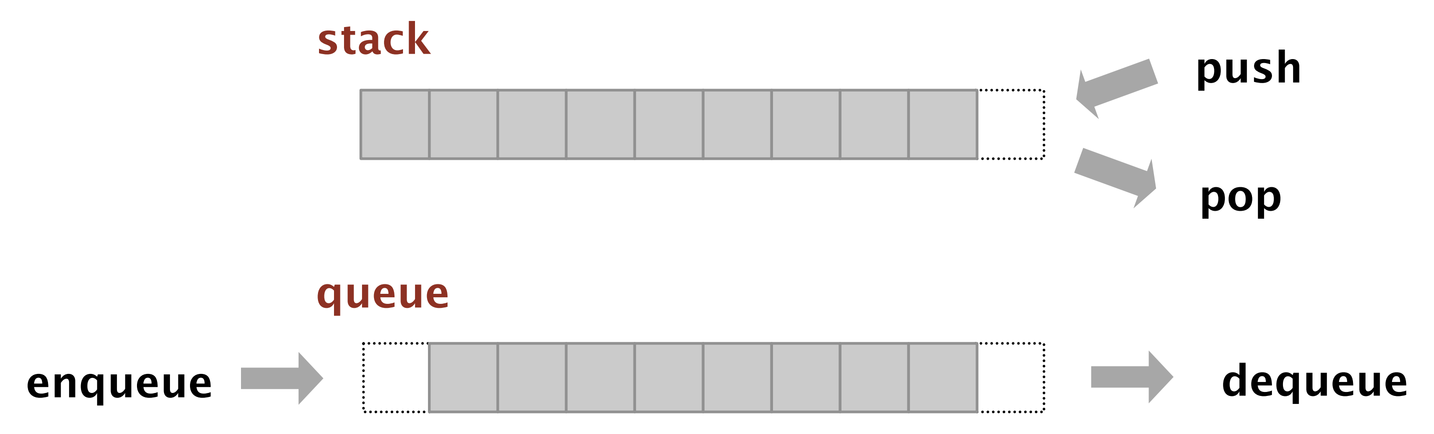
Stacks and queues

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 ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ 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.

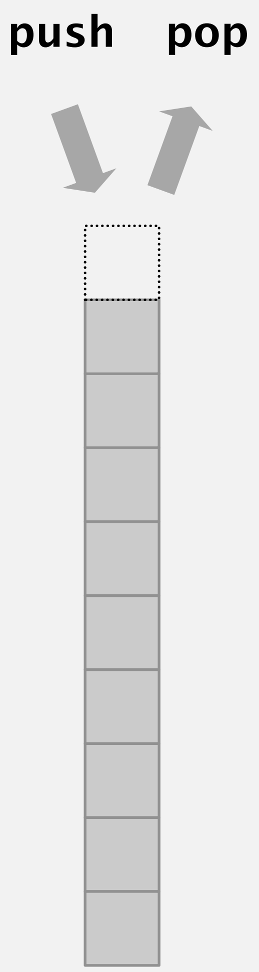
Implementation: actual code implementing operations.

Interface: description of data type, basic operations.

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*

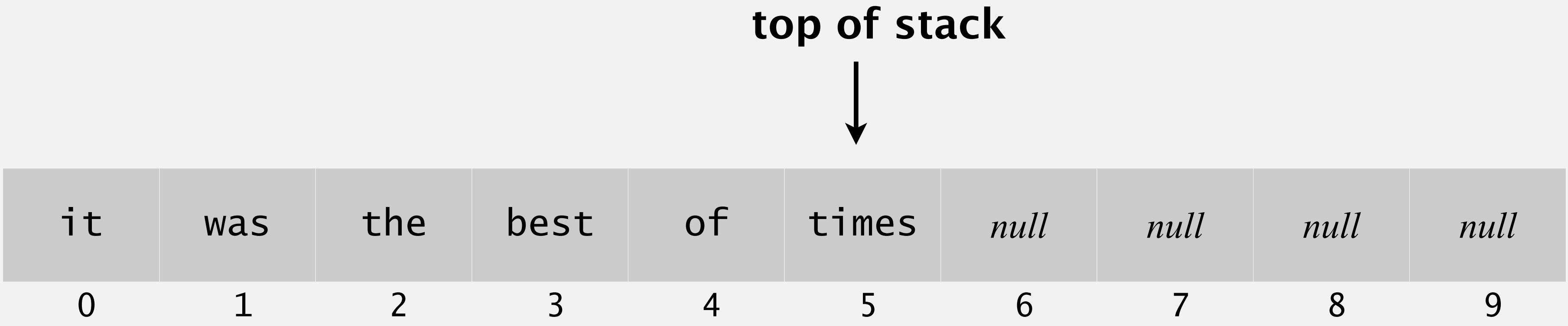


Warmup client. Reverse sequence of strings from standard input.

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

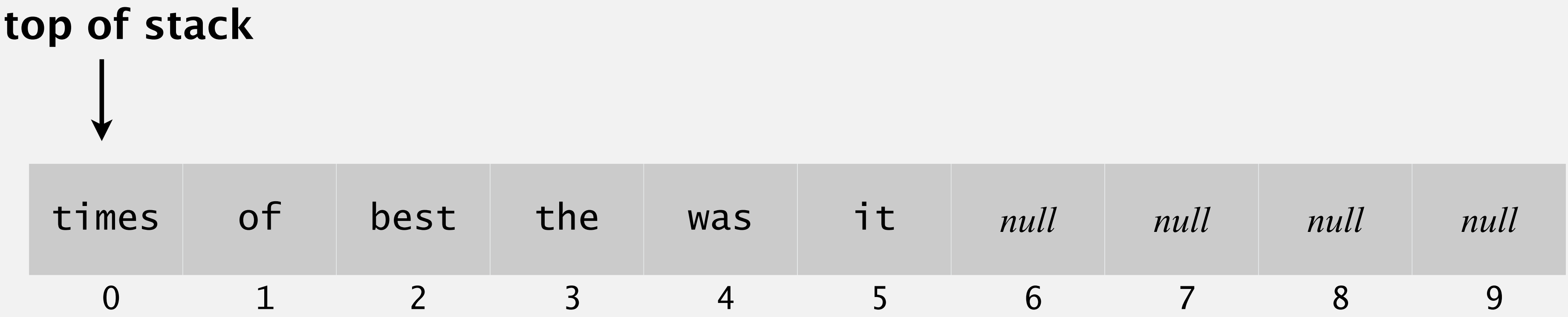
A. Can't be done efficiently with an array.

B.



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

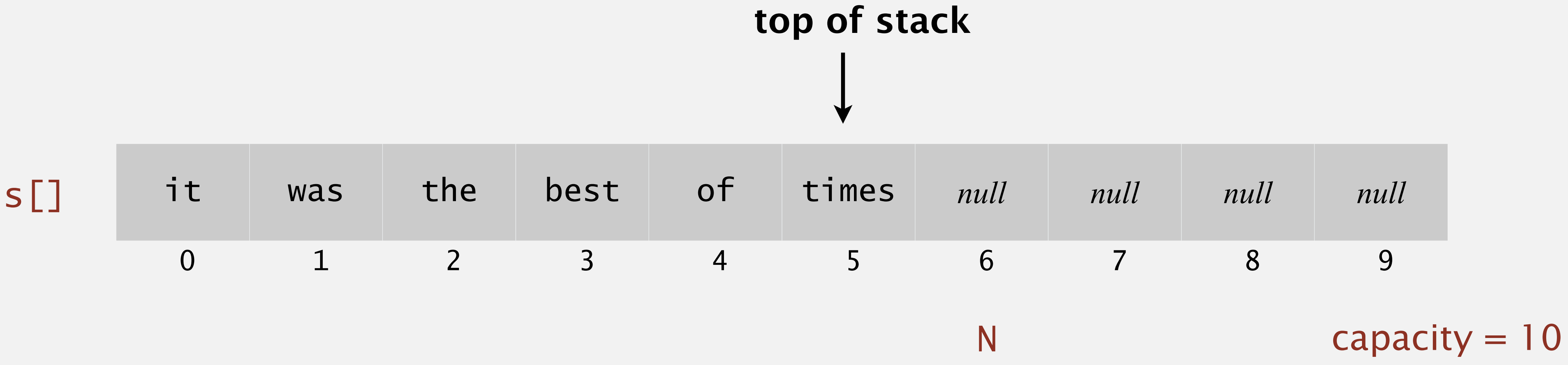
C.



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

Fixed-capacity stack: array implementation

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



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

Stack considerations

Overflow and underflow.

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

Null items. We allow null items to be inserted.

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

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

loitering

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

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

Stack: resizing-array implementation

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

Q. How to grow and shrink array?

First try.

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

Too expensive.

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

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

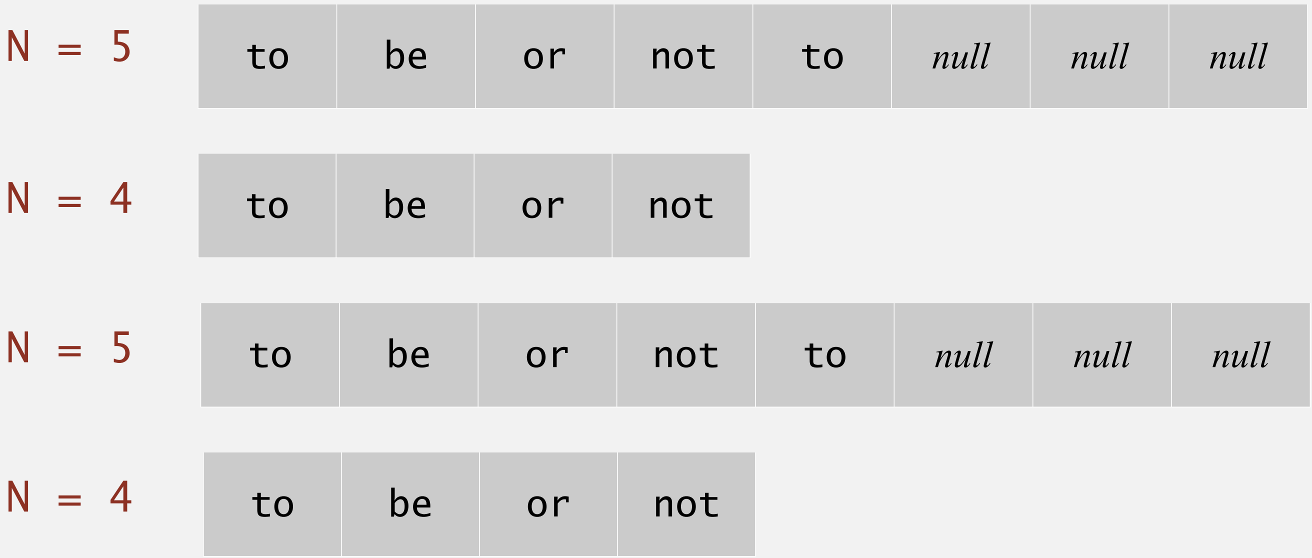
Q. How to shrink array?

First try.

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

Too expensive in worst case.

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



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

N = 4 `to be or not`

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

N = 4 `to be or not`

Stack resizing-array implementation: performance

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

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

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

doubling and halving operations

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

Stack resizing-array implementation: memory usage

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

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

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

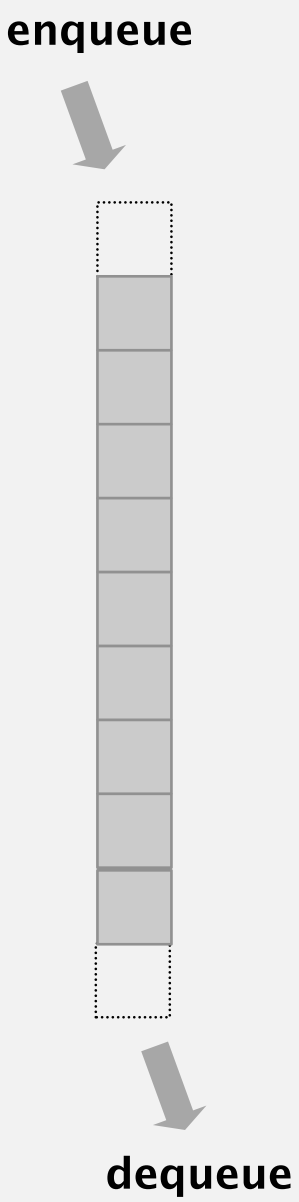
← 8 bytes × array size

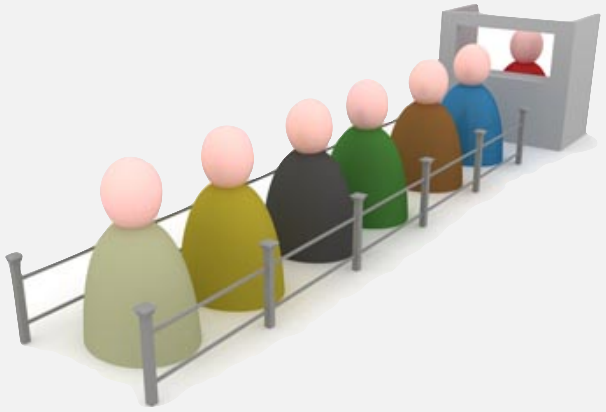
Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).

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*

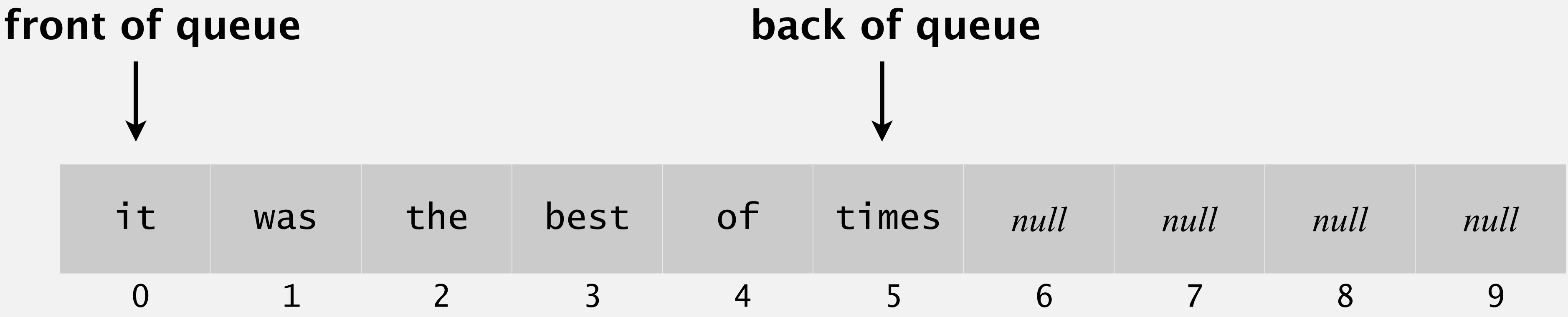




How to implement a fixed-capacity queue with an array?

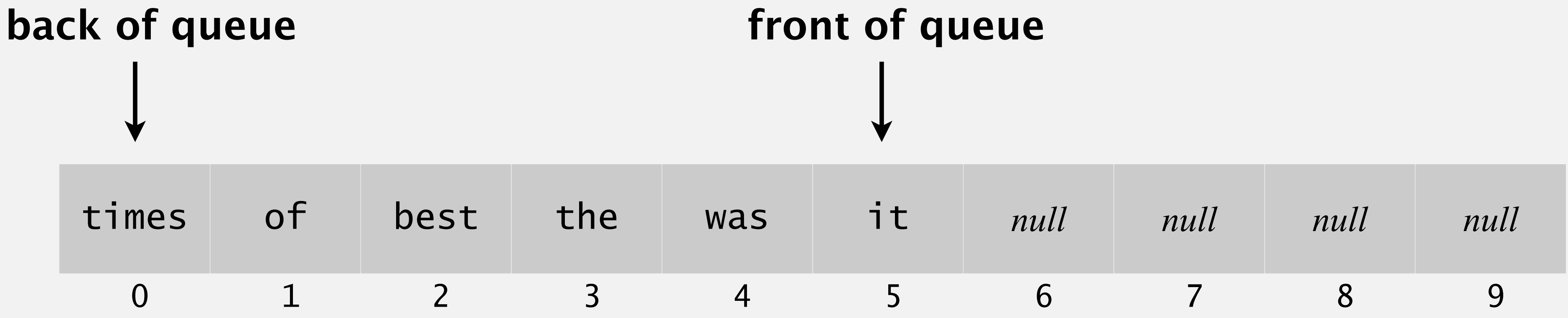
A. Can't be done efficiently with an array.

B.



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

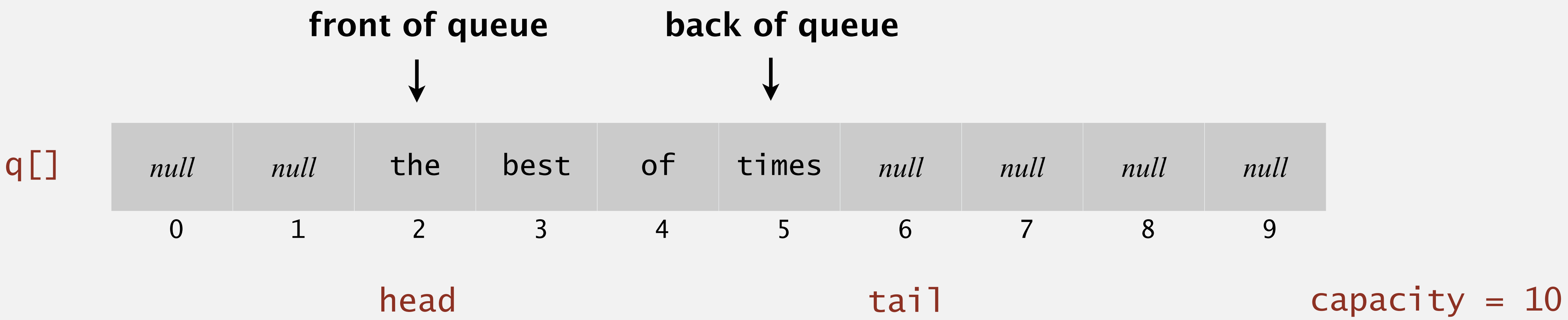
C.



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

Queue: resizing-array implementation

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



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

`capacity = 10`

Q. How to resize?

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.

type parameter

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

compile-time error

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

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.

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`.

java.lang.Iterable interface

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

Q. What is an `Iterator` ?

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

java.util.Iterator interface

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

← optional; use at your own risk

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement (shorthand)

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

equivalent code (longhand)

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

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);
```

Q. How to detect?

A.

- 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.

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.

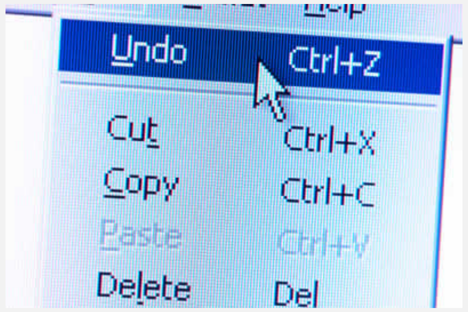
Best practices. Use our implementations of `Stack`, `Queue`, and `Bag`.

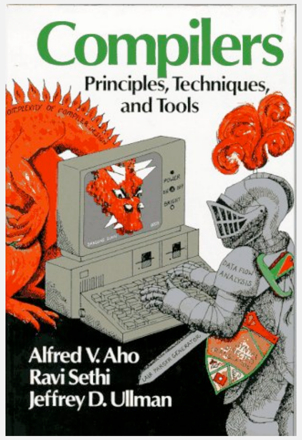
Stack applications

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

Java

Adobe® PostScript® 3™





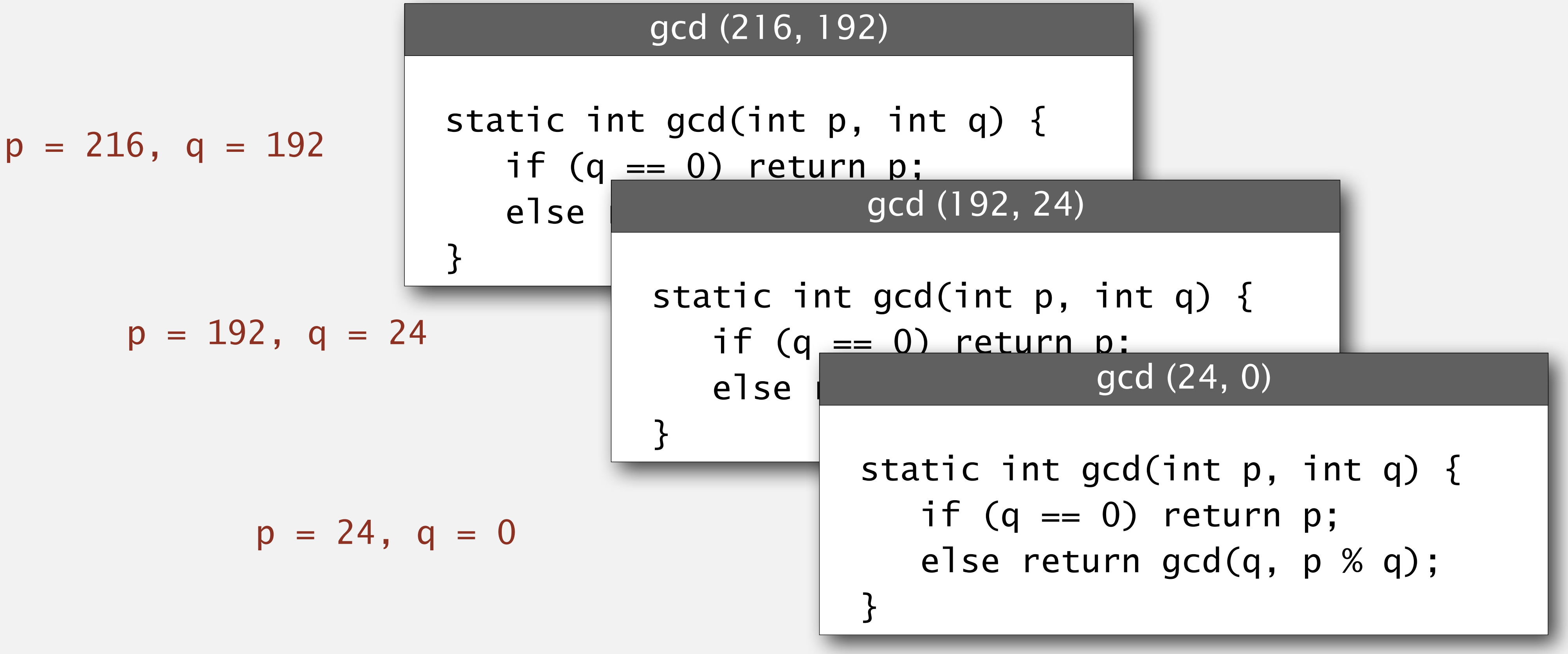
Function calls

How a compiler implements a function.

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

Recursive function. Function that calls itself.

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



gcd (216, 192)

p = 216, q = 192

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

gcd (192, 24)

p = 192, q = 24

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

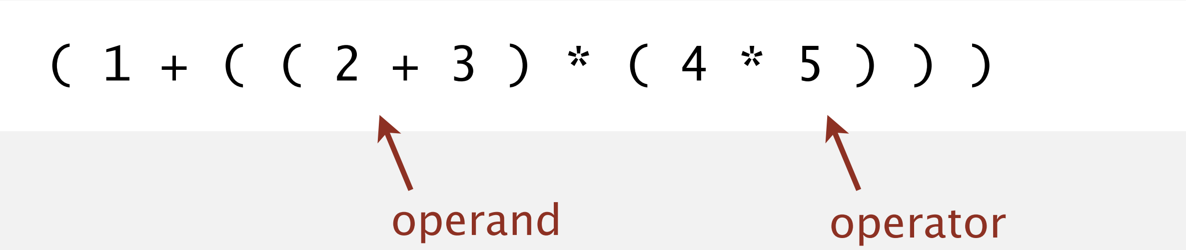
gcd (24, 0)

p = 24, q = 0

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

Arithmetic expression evaluation

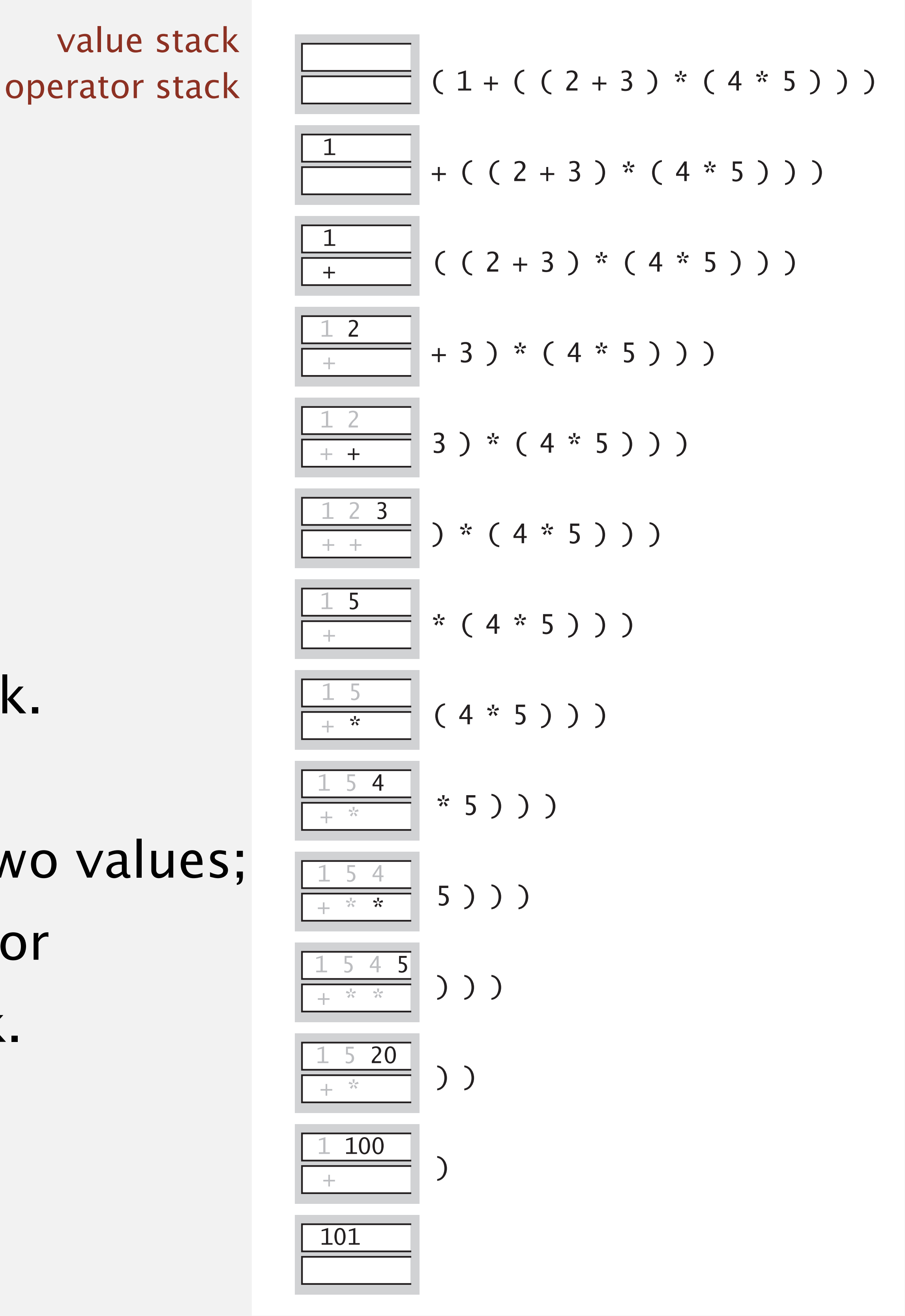
Goal. Evaluate infix expressions.



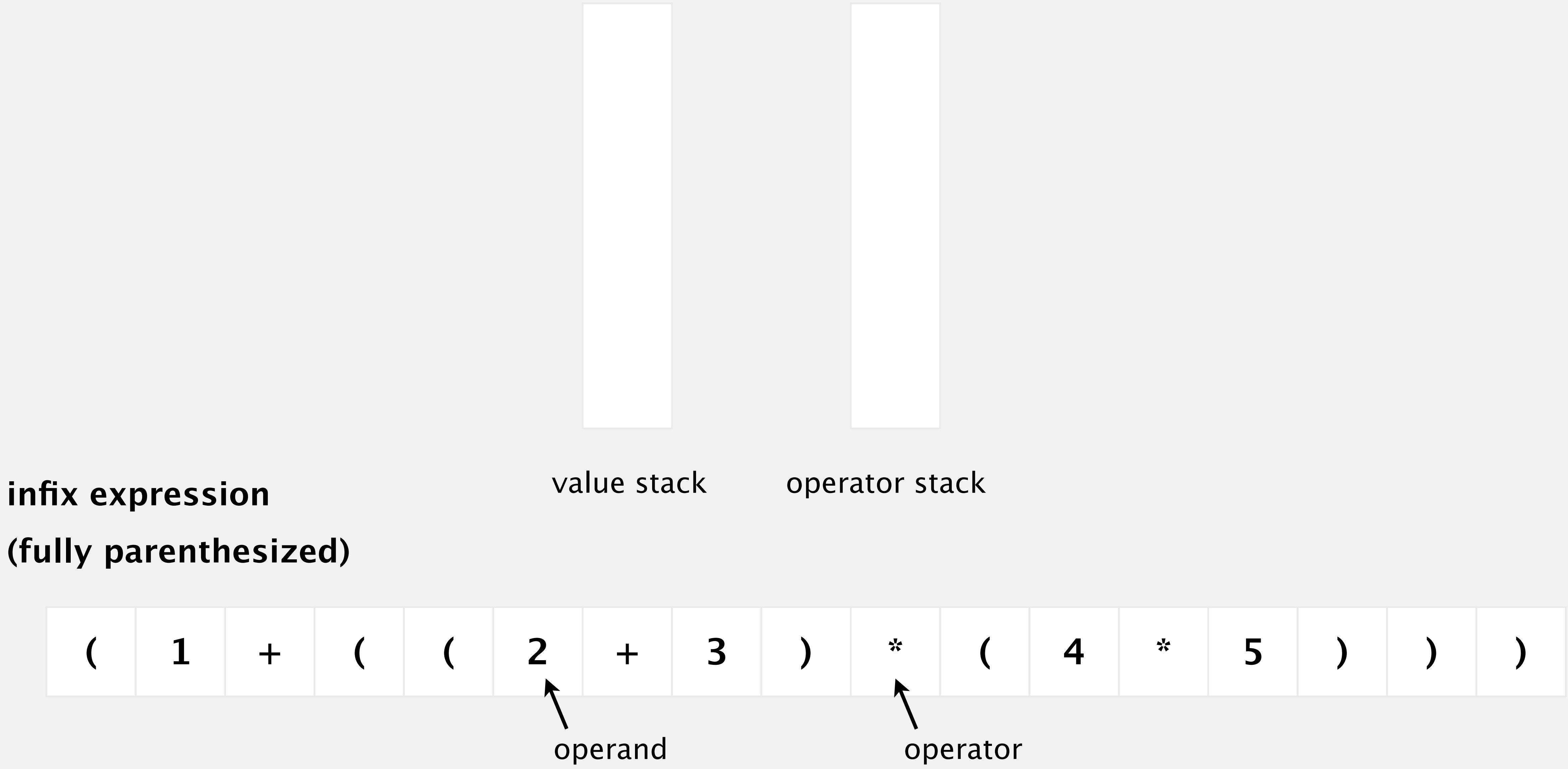
Two-stack algorithm. [E. W. Dijkstra]

- 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!



Dijkstra's two-stack algorithm demo



value stack

operator stack

infix expression

(fully parenthesized)

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

operand

operator

Arithmetic expression evaluation

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

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

Correctness

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

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

as if the original input were:

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

Repeating the argument:

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

Extensions. More ops, precedence order, 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, …