



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

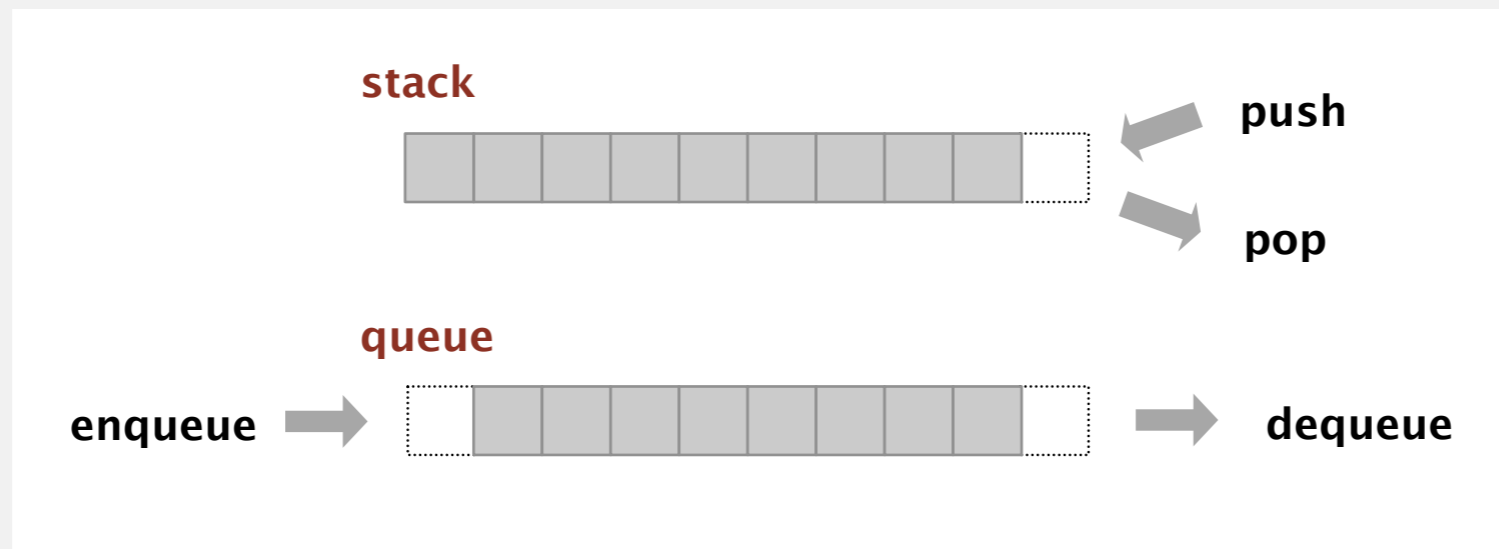
1.3 BAGS, QUEUES, AND STACKS

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

Stacks and queues

Fundamental data types.

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

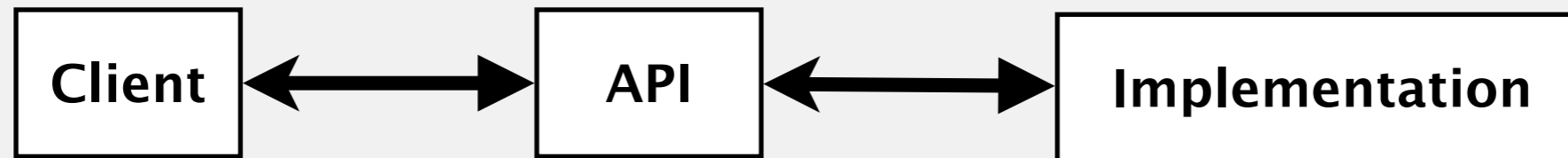


Stack. Examine the item most recently added. ← LIFO = "last in first out"

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,



<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)
```

add a new string to stack

```
    String pop()
```

*remove and return the string
most recently added*

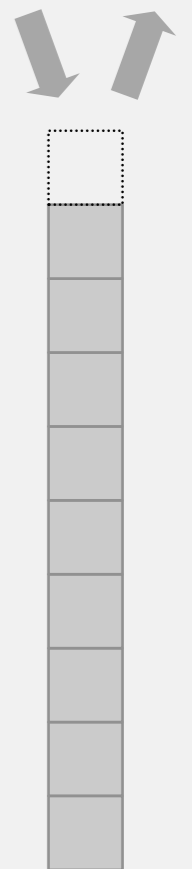
```
    boolean isEmpty()
```

is the stack empty?

```
    int size()
```

number of strings on the stack

push pop



Warmup client. Reverse sequence of strings from standard input.

Sample client

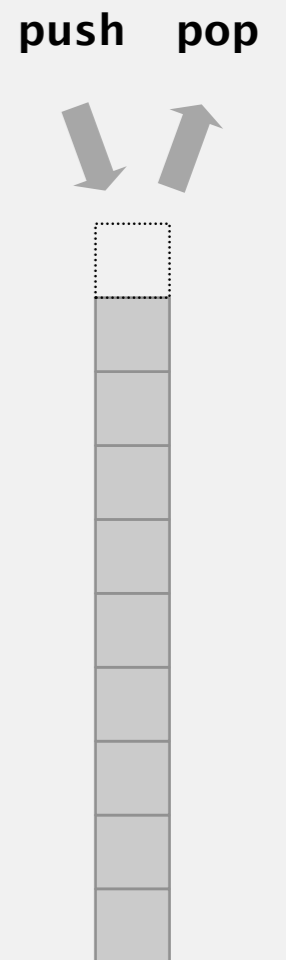
Warmup client. Reverse sequence of strings from standard input.

- Read string and push onto stack.
- Pop string and print.

```
public class ReverseStrings
{
    public static void main(String[] args)
    {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readString());
        while (!stack.isEmpty())
            StdOut.println(stack.pop());
    }
}
```

```
% more tinyTale.txt
it was the best of times ...
```

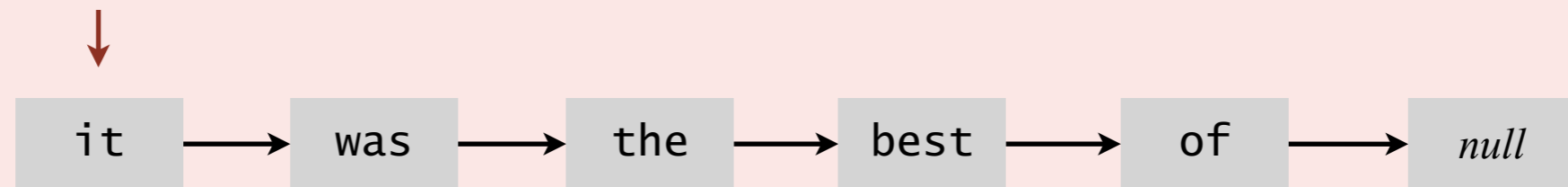
```
% java ReverseStrings < tinyTale.txt
... times of best the was it
[ignoring newlines]
```



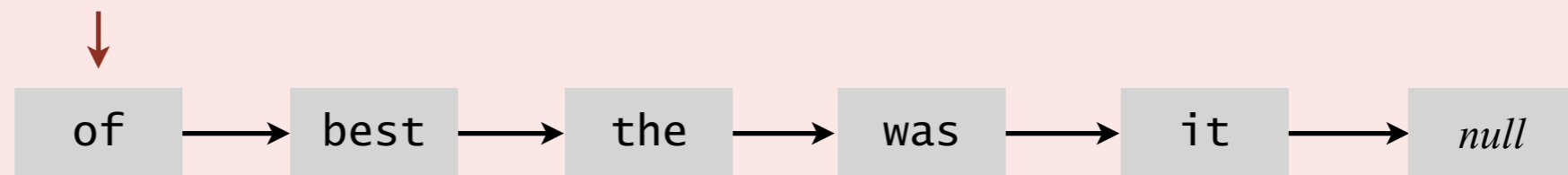
Stacks quiz 1

How to implement a stack with a singly-linked list?

A. least recently added



B. most recently added

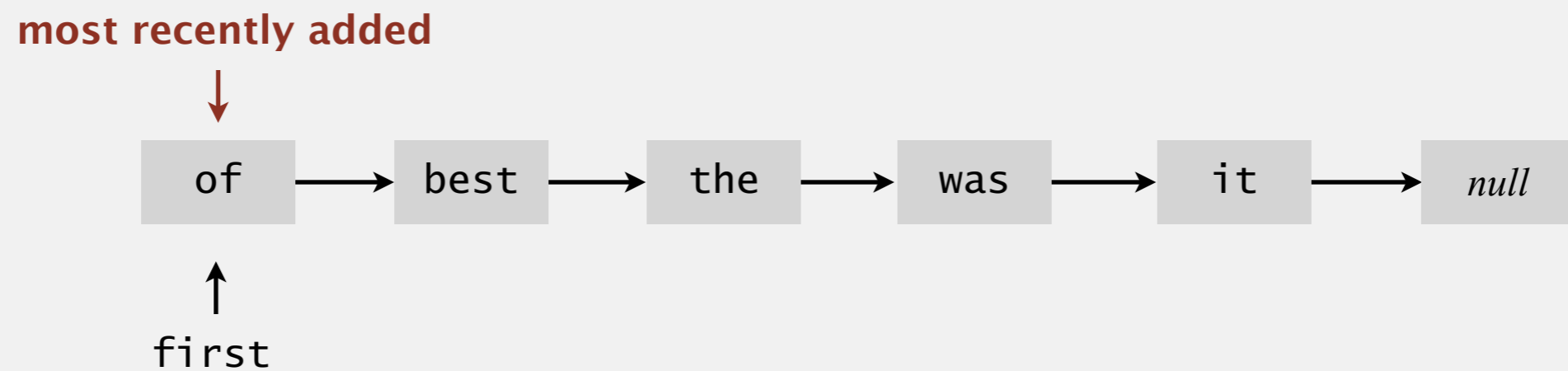


C. *None of the above.*

D. *I don't know.*

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

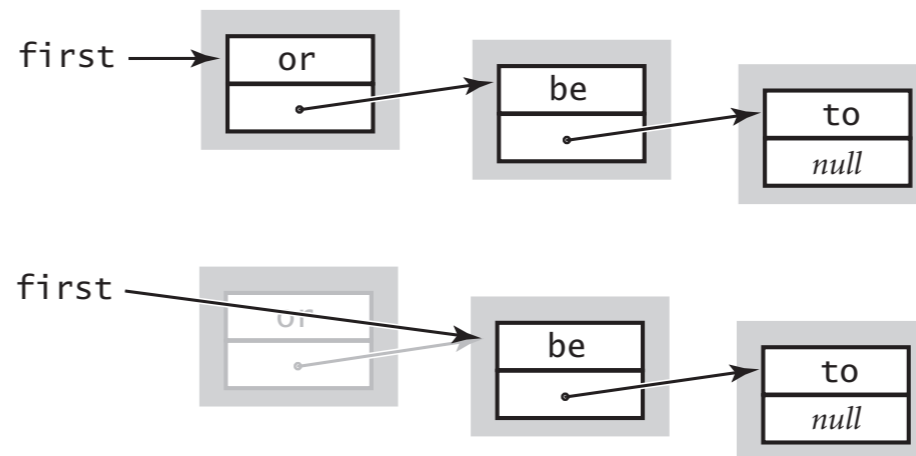
```
private class Node
{
    String item;
    Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

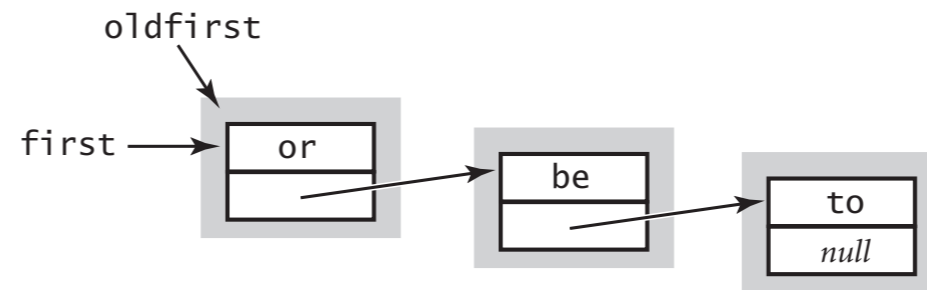
Stack push: linked-list implementation

inner class

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

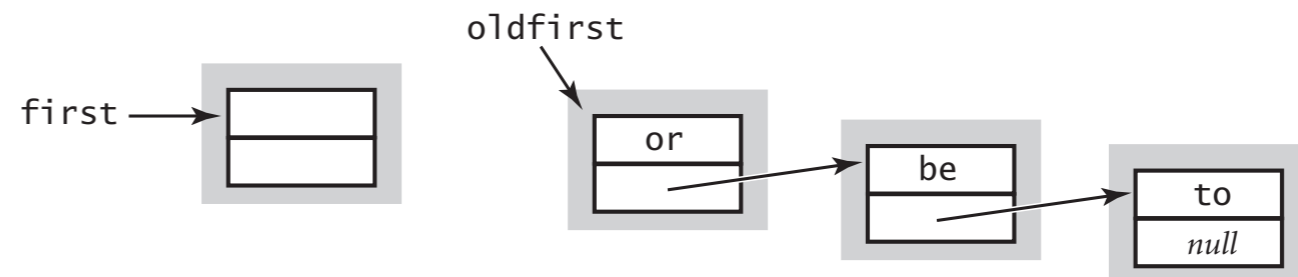
save a link to the list

```
Node oldfirst = first;
```



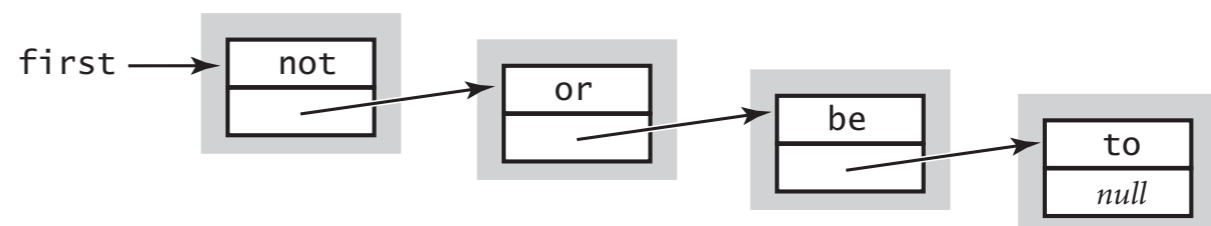
create a new node for the beginning

```
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)



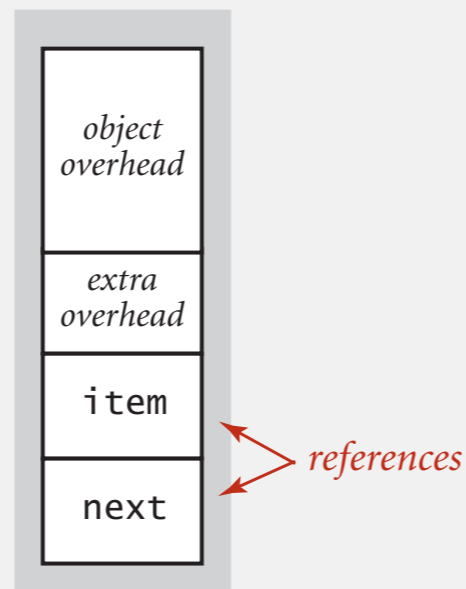
Stack: linked-list implementation performance

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 item String)

8 bytes (reference to next Node)

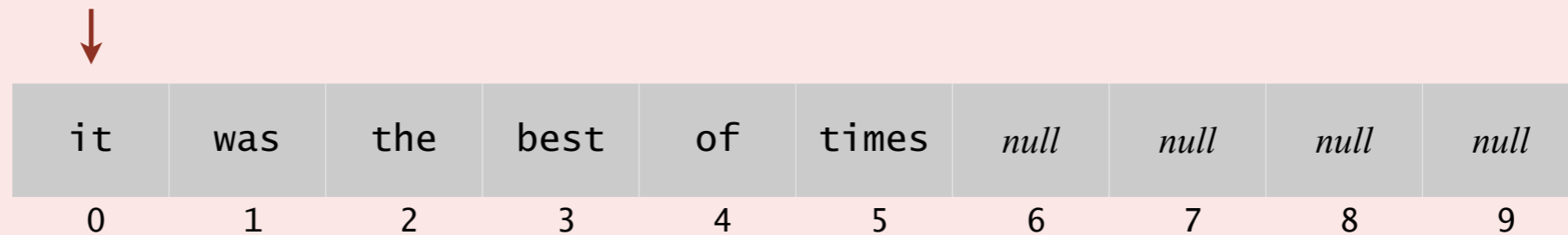
40 bytes allocated per stack node

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

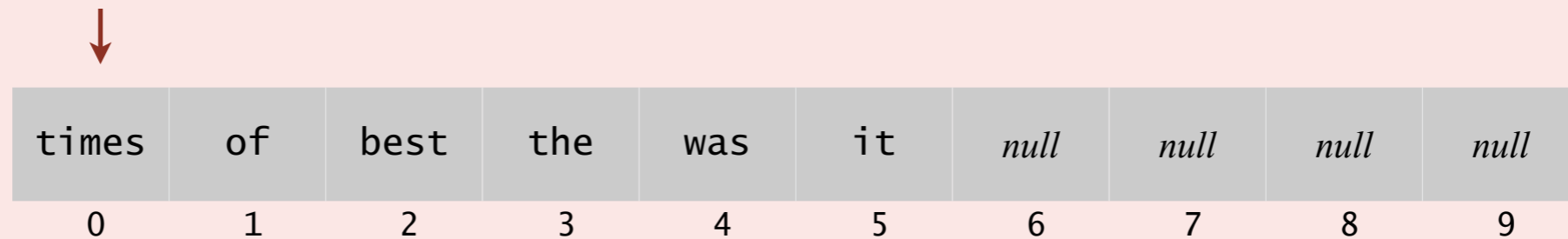
Stacks quiz 2

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

A. **least recently added**



B. **most recently added**

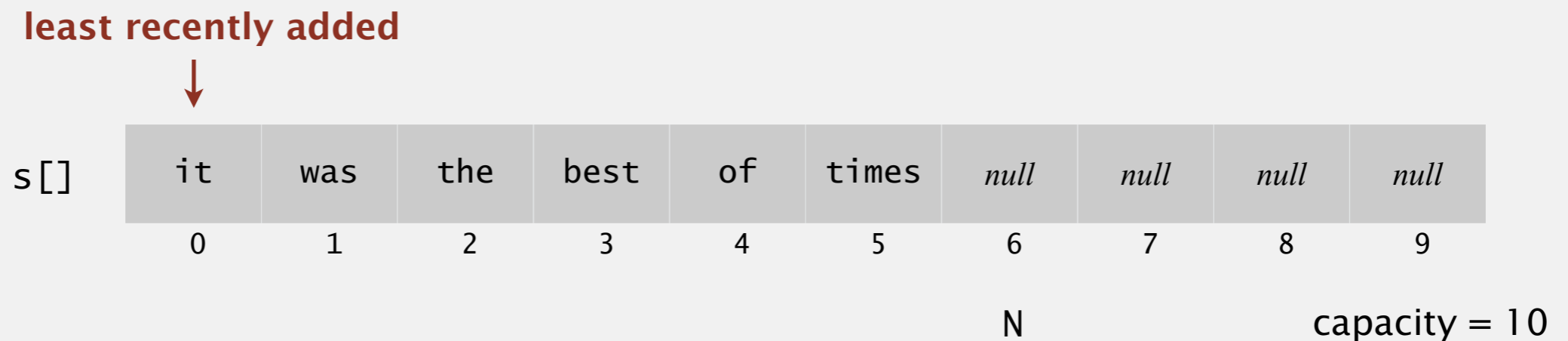


C. *None of the above.*

D. *I don't know.*

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; }

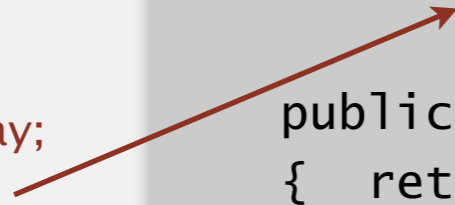
    public void push(String item)
    { s[N++] = item; }

    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]

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**



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

1.3 BAGS, QUEUES, AND STACKS

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

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

↑
1 array access
per push

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

Stack: resizing-array implementation

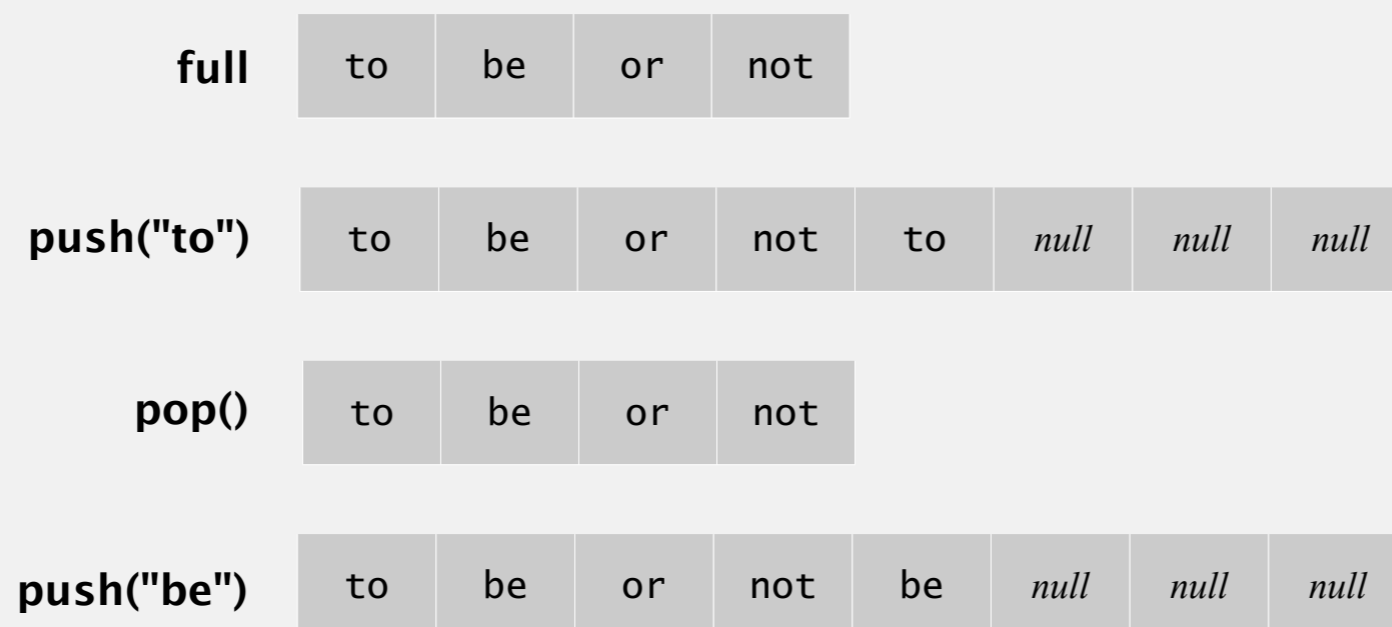
Q. How to shrink array?

First try.

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

Too expensive in worst case.

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



Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

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

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

Invariant. Array is between 25% and 100% full.

Stack resizing-array implementation: performance

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

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

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

doubling and halving operations

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

Stack resizing-array implementation: memory usage

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

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

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

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

Stack implementations: resizing array vs. linked list

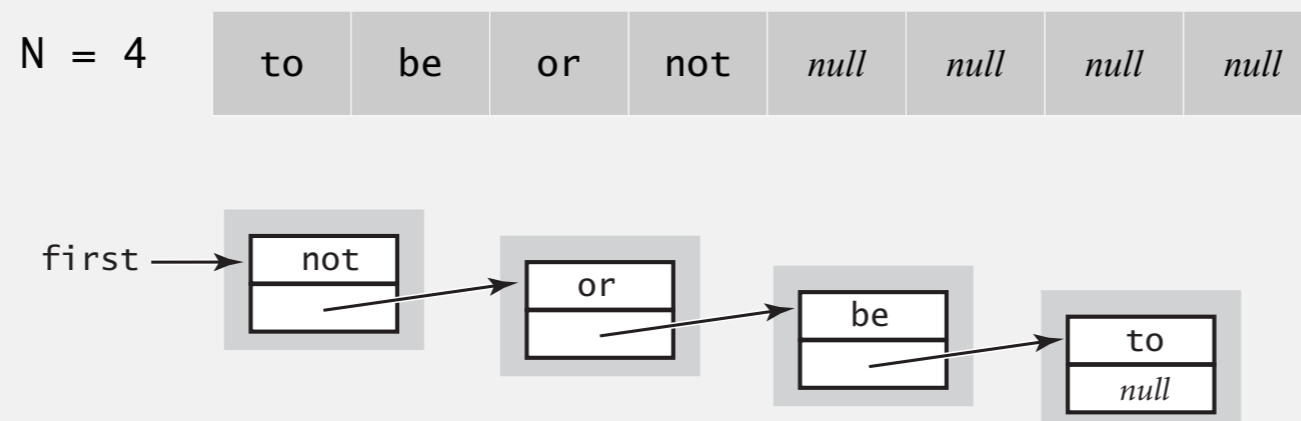
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the **worst case**.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant **amortized** time.
- Less wasted space.





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

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)
```

add a new string to queue

```
    String dequeue()
```

*remove and return the string
least recently added*

```
    boolean isEmpty()
```

is the queue empty?

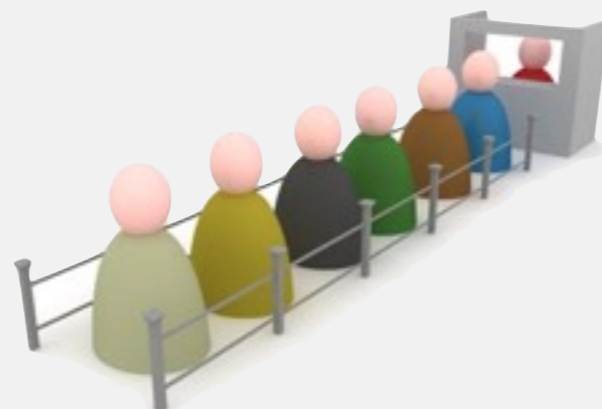
```
    int size()
```

number of strings on the queue

enqueue

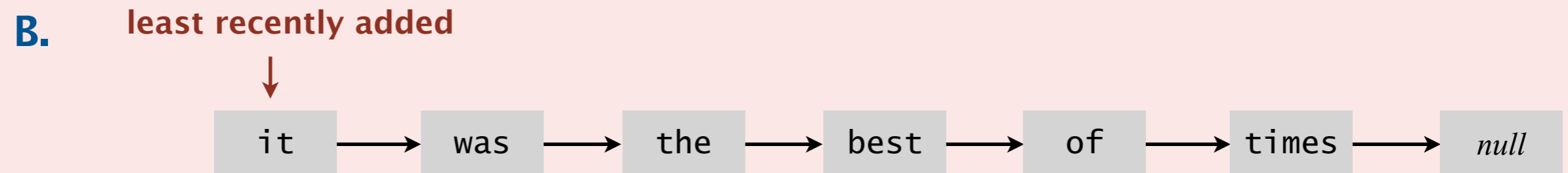
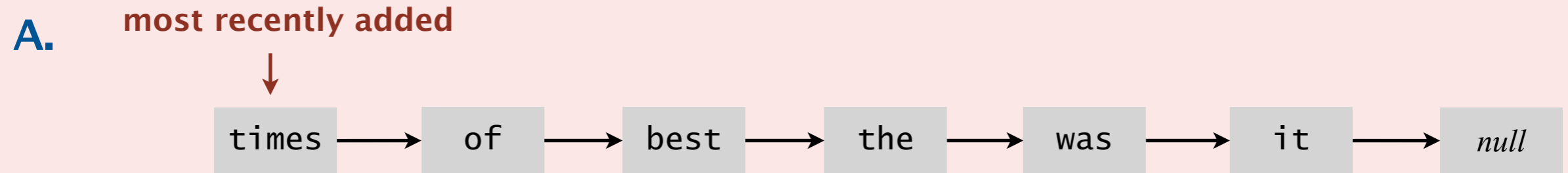


dequeue



Queues quiz 1

How to implement a queue with a singly-linked linked list?

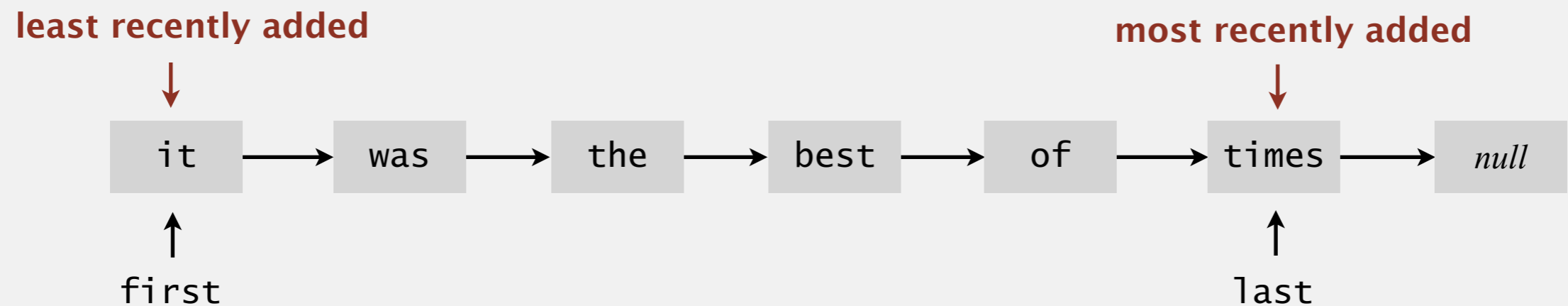


C. *None of the above.*

D. *I don't know.*

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

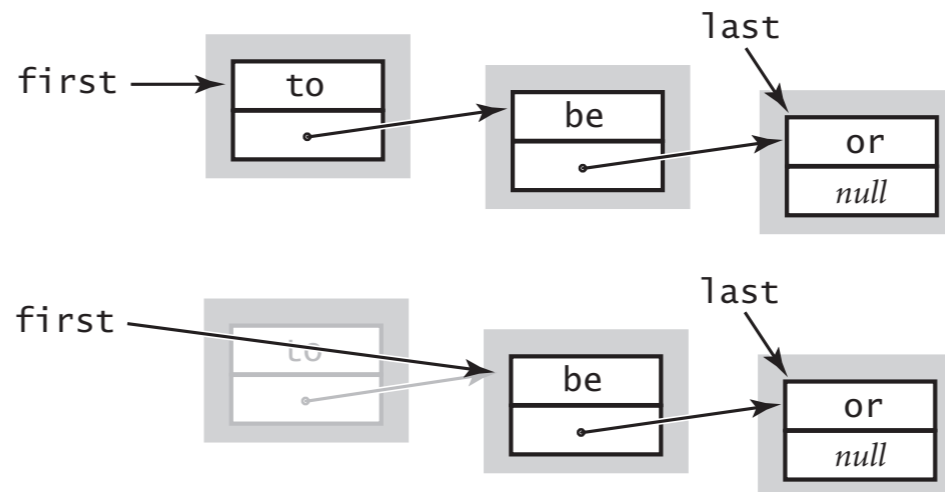
```
private class Node
{
    String item;
    Node next;
}
```

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().

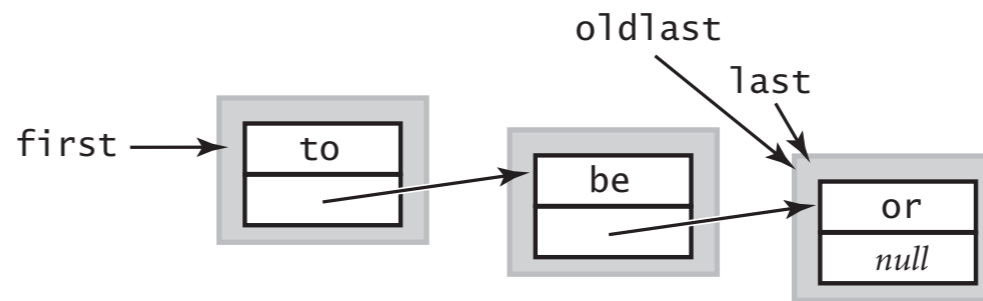
Queue enqueue: linked-list implementation

inner class

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

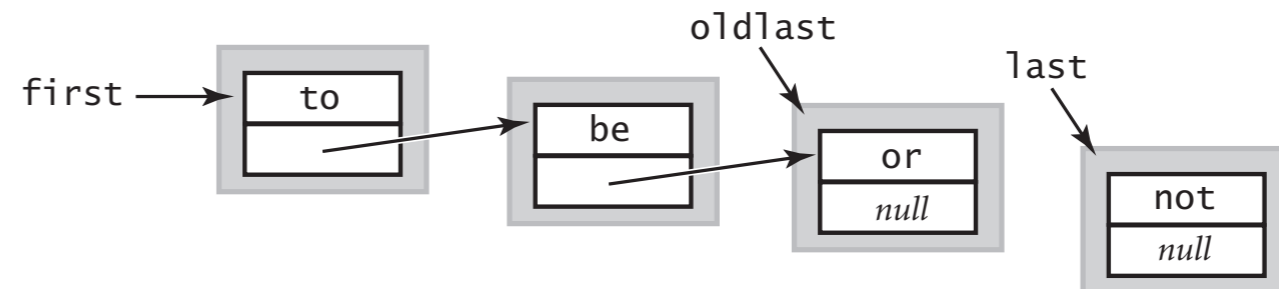
save a link to the last node

```
Node oldlast = last;
```



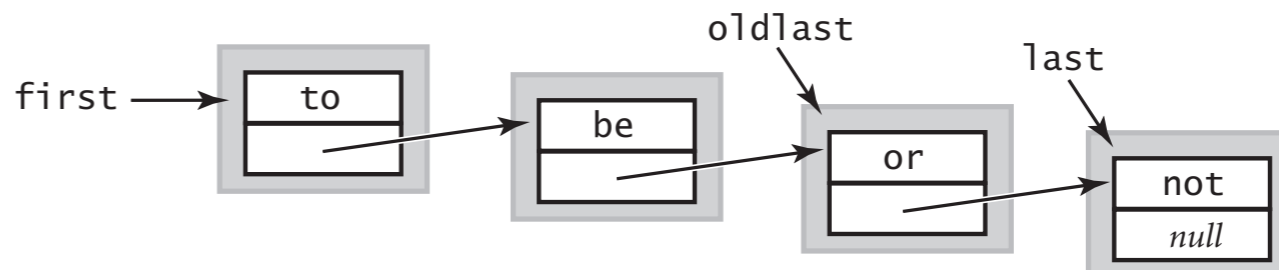
create a new node for the end

```
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;

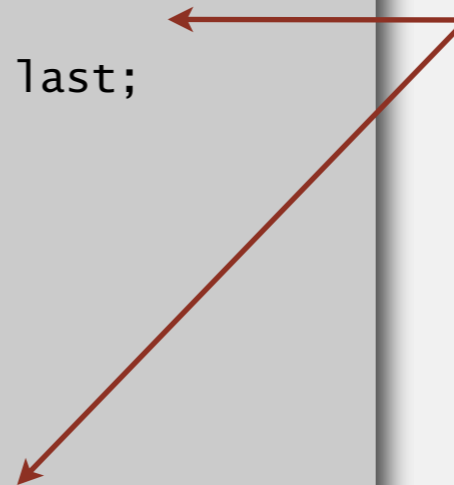
    private class Node
    { /* same as in LinkedStackOfStrings */ }

    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



Queues quiz 2

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

A. least recently added



it	was	the	best	of	times	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
0	1	2	3	4	5	6	7	8	9

B.

most recently added



times	of	best	the	was	it	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
0	1	2	3	4	5	6	7	8	9

C. *None of the above.*

D. *I don't know.*

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**: “circular buffer”
- Add resizing array.



Q. How to resize?



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

1.3 BAGS, QUEUES, AND STACKS

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

Parameterized stack

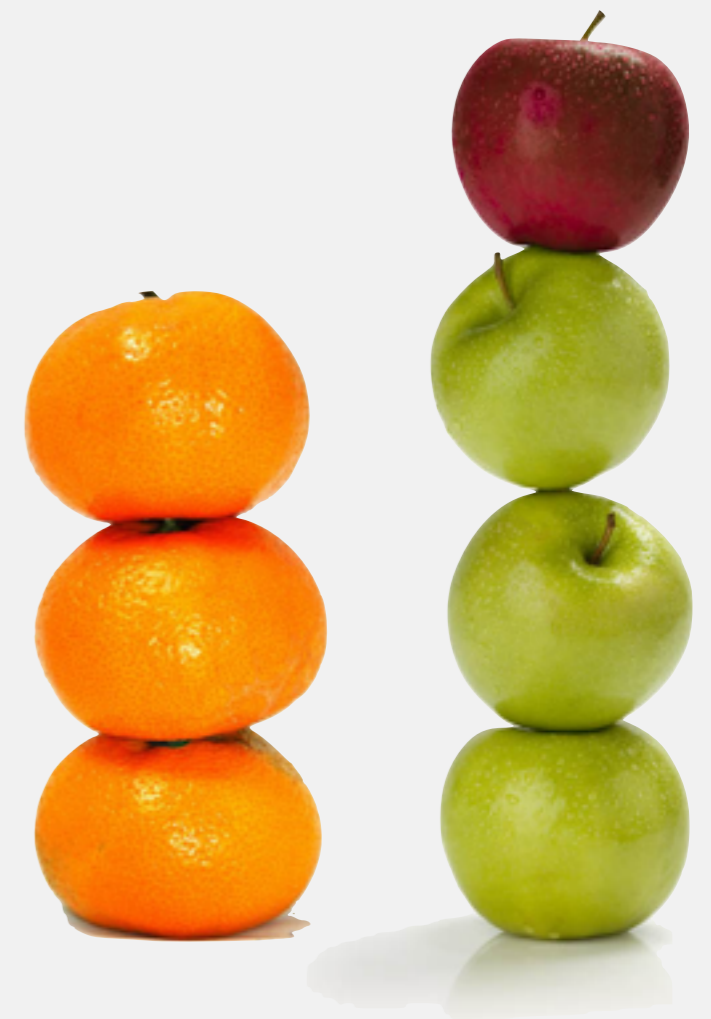
We implemented: StackOfStrings.

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;
    }
}
```

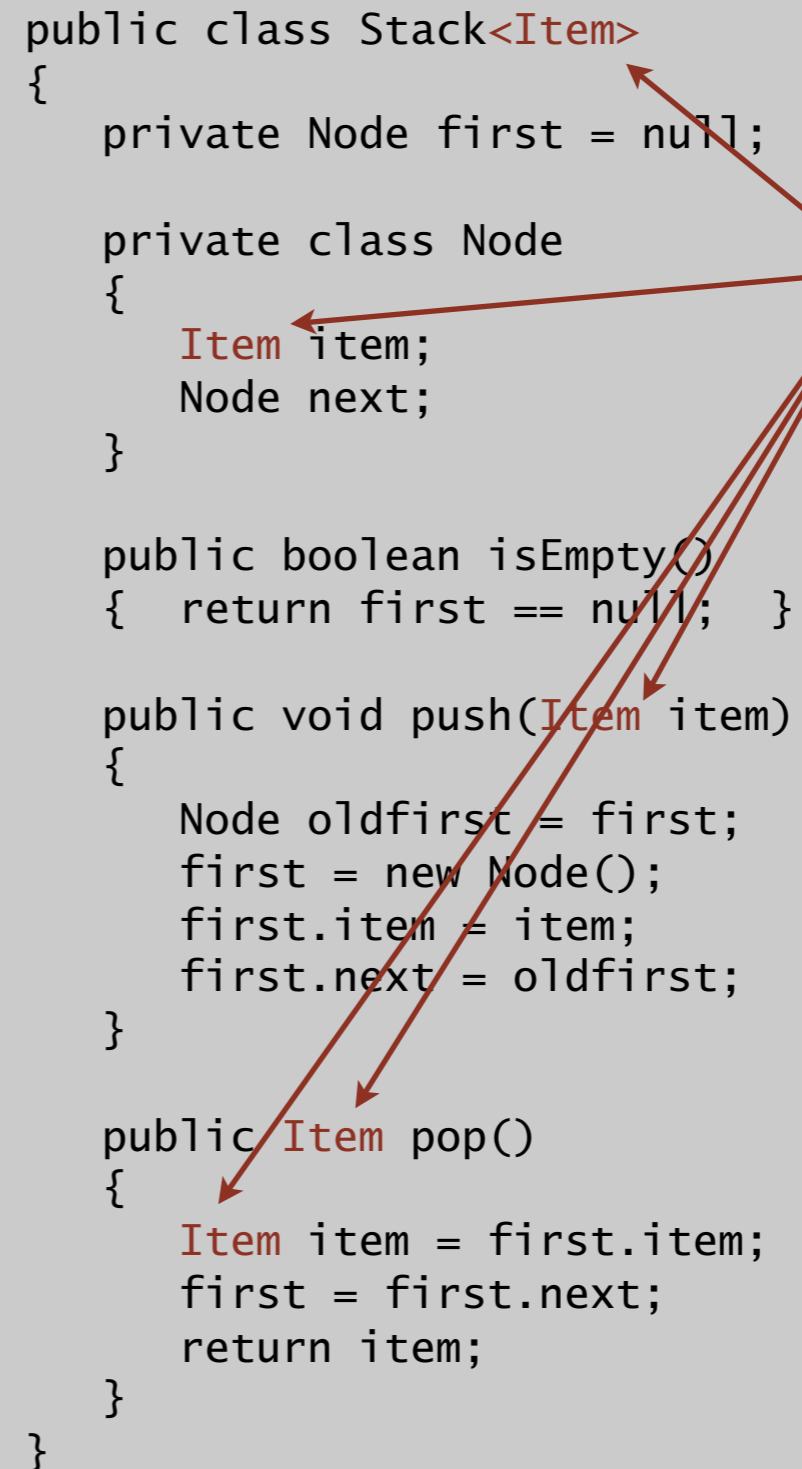
```
public class Stack<Item>
{
    private Node first = null;

    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.

Stacks quiz 3

Which of the following is the correct way to declare and initialize an empty stack of characters?

- A. `Stack<Character> stack = new Stack();`
- B. `Stack stack = new Stack<Character>();`
- C. `Stack<Character> stack = new Stack<Character>();`
- D. `Stack<char> stack = new Stack<char>();`
- E. *None of the above.*



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

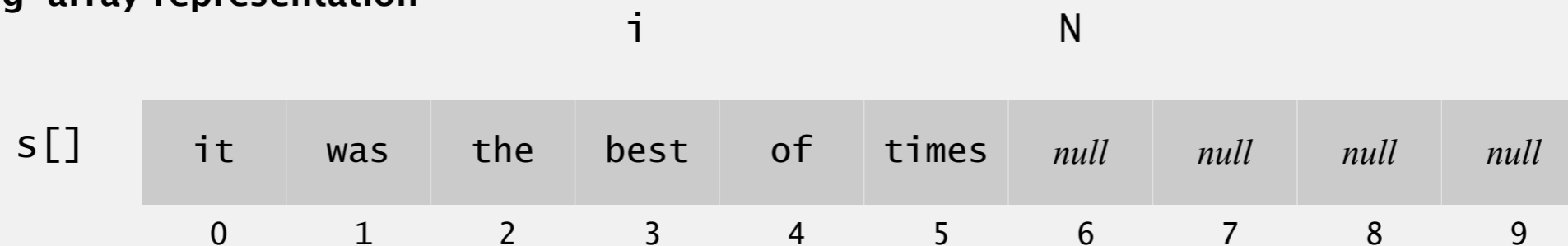
1.3 BAGS, QUEUES, AND STACKS

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

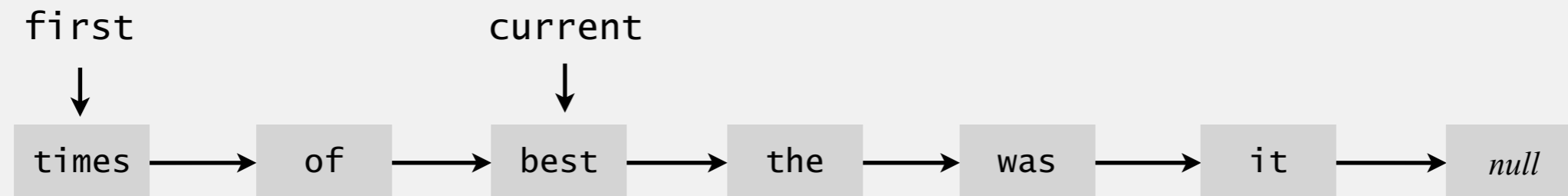
Iteration

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

resizing-array representation



linked-list representation



Java solution. Use a “foreach” loop.

Foreach loop

Java provides elegant syntax for iteration over collections.

“foreach” loop (shorthand)

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

equivalent code (longhand)

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

To make user-defined collection support foreach loop:

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

Iterators

To support foreach loops, Java provides two interfaces.

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

java.util.Iterator interface

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

java.lang.Iterable interface

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

Type safety.

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

Stack iterator: linked-list implementation

```
import java.util.Iterator;

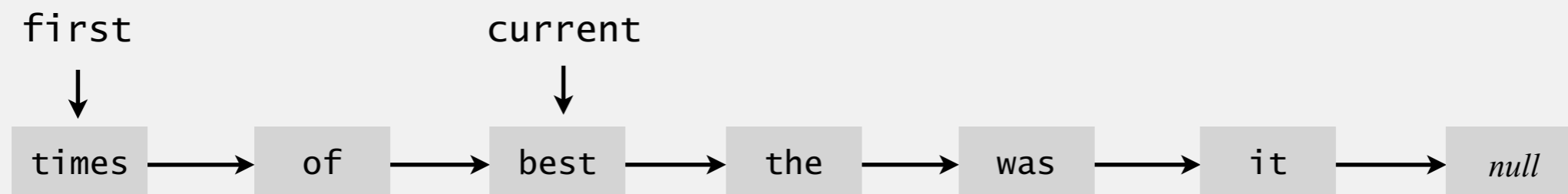
public class Stack<Item> implements Iterable<Item>
{
    ...

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

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

        public boolean hasNext() { return current != null; }
        public void remove() { /* not supported */ }
        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

throw UnsupportedOperationException
throw NoSuchElementException
if no more items in iteration



Bag API

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

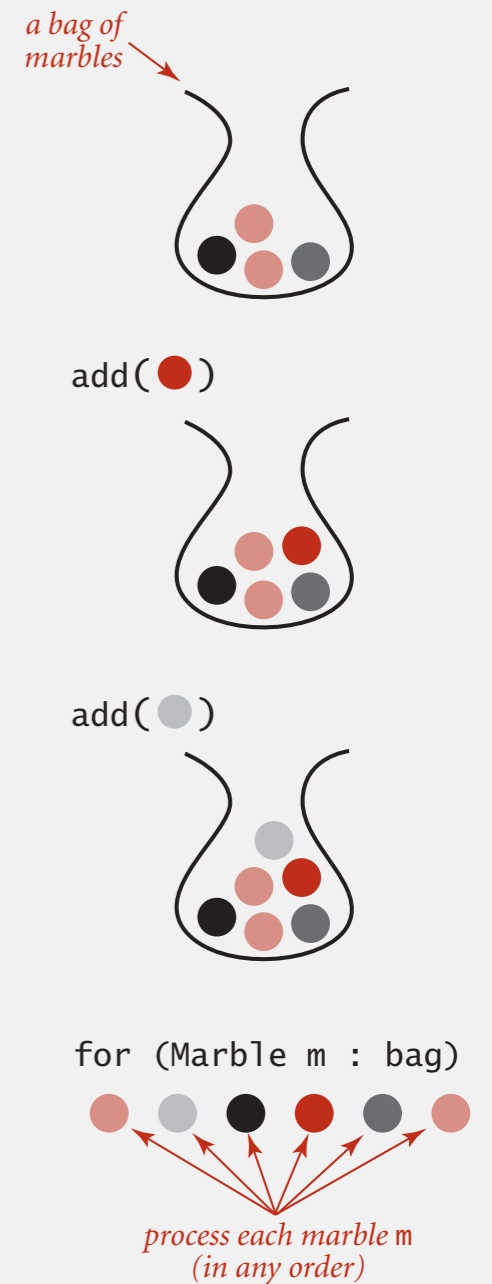
```
public class Bag<Item> implements Iterable<Item>
```

```
    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).



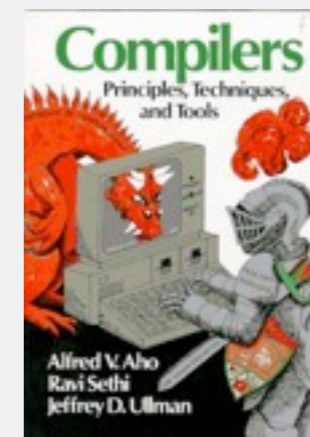
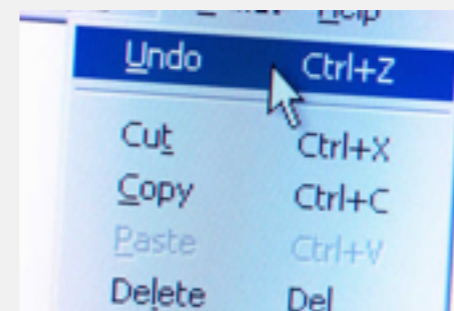
<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

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

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.
- ...



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.

p = 216, q = 192

```
gcd (216, 192)
static int gcd(int p, int q) {
    if (q == 0) return p;
    else
}
```

p = 192, q = 24

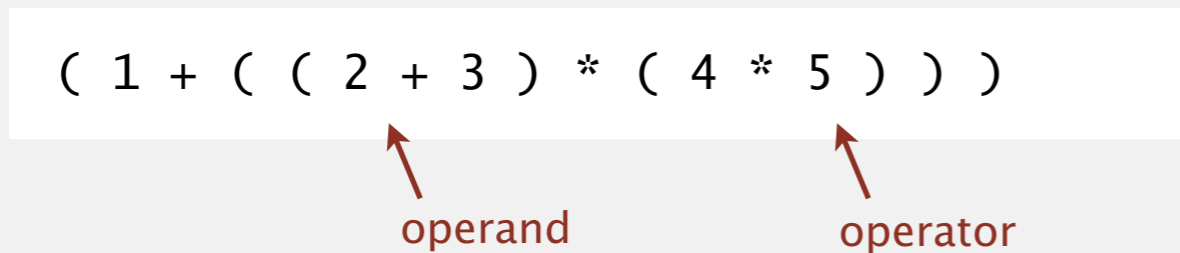
```
gcd (192, 24)
static int gcd(int p, int q) {
    if (q == 0) return p;
    else
}
```

p = 24, q = 0

```
gcd (24, 0)
static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}
```

Arithmetic expression evaluation

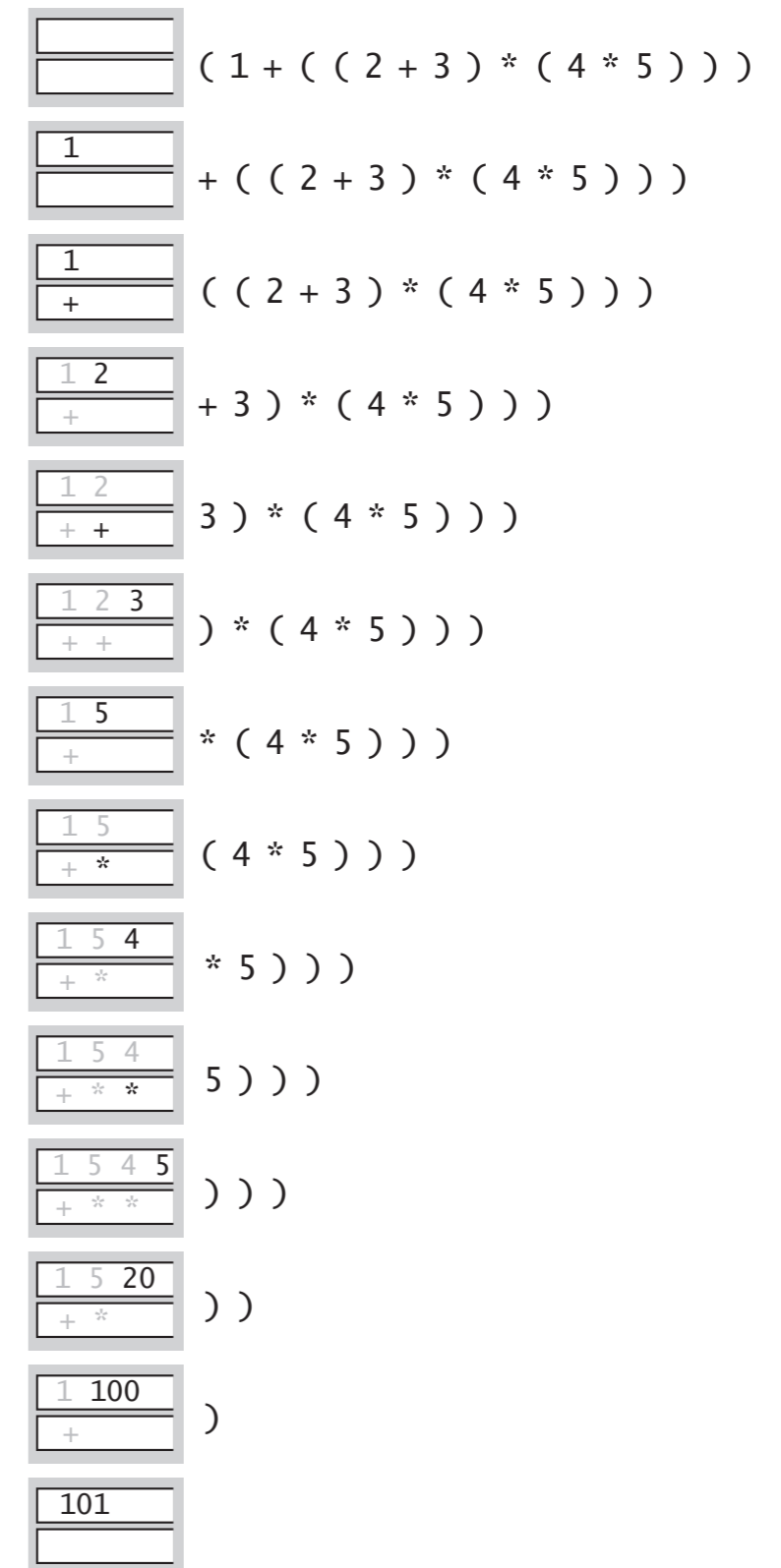
Goal. Evaluate infix expressions.



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.

value stack
operator 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
```

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 * * +
```

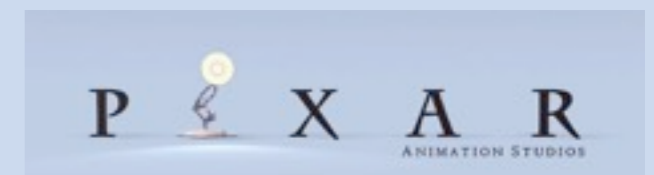
Bottom line. Postfix or "reverse Polish" notation.

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



Jan Łukasiewicz

TECHNICAL INTERVIEW QUESTIONS



QUEUE WITH TWO STACKS

Problem. Implement a queue with two stacks so that:

- Each queue op uses a constant **amortized** number of stack ops.
- At most constant extra memory (besides two stacks).

Applications.

- Job interview.
- Implement an **immutable** or **persistent** queue.
- Implement a queue in a (purely) **functional programming language**.

