Stacks and Queues

CS 121: Data Structures

START RECORDING

Outline

- Attendance quiz
- Overview of stacks and queues
- Implementation of stacks and queues

Attendance Quiz

Attendance Quiz: Linked Lists

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced

```
public class IntLinkedList {

    private class Node {
        int val;
        Node next;

        public Node(int v) {
            val = v;
            next = null;
        }
    }

    private Node head; // the first node
    private Node tail; // the last node
    private int length; // number of nodes in the list

    // constructor initializes an empty linked list
    public IntLinkedList() {
        head = null;
        tail = null;
        length = 0;
    }

    // TODO
    // public void addFirst(int val) { }
    // public void addLast(int val) { }
}
```

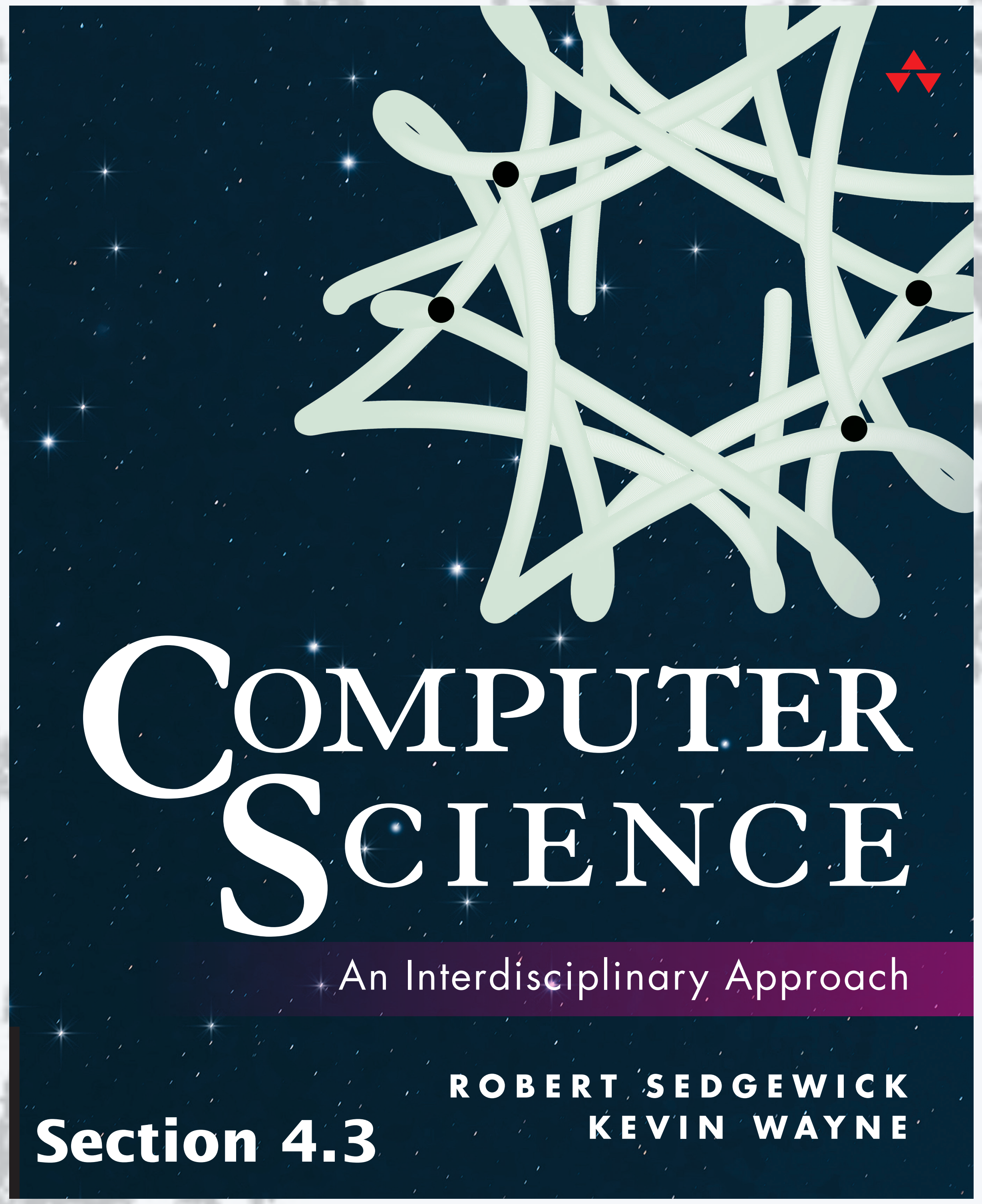
Attendance Quiz: Linked Lists

- Write your name
- Write code for addFirst() and addLast()

```
public class IntLinkedList {

    private class Node {
        int val;
        Node next;

        public Node(int v) {
            val = v;
            next = null;
        }
    }

    private Node head; // the first node
    private Node tail; // the last node
    private int length; // number of nodes in the list

    // constructor initializes an empty linked list
    public IntLinkedList() {
        head = null;
        tail = null;
        length = 0;
    }

    // TODO
    // public void addFirst(int val) { }
    // public void addLast(int val) { }
}
```

COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES



http://introcs.cs.princeton.edu

12. Stacks and Queues

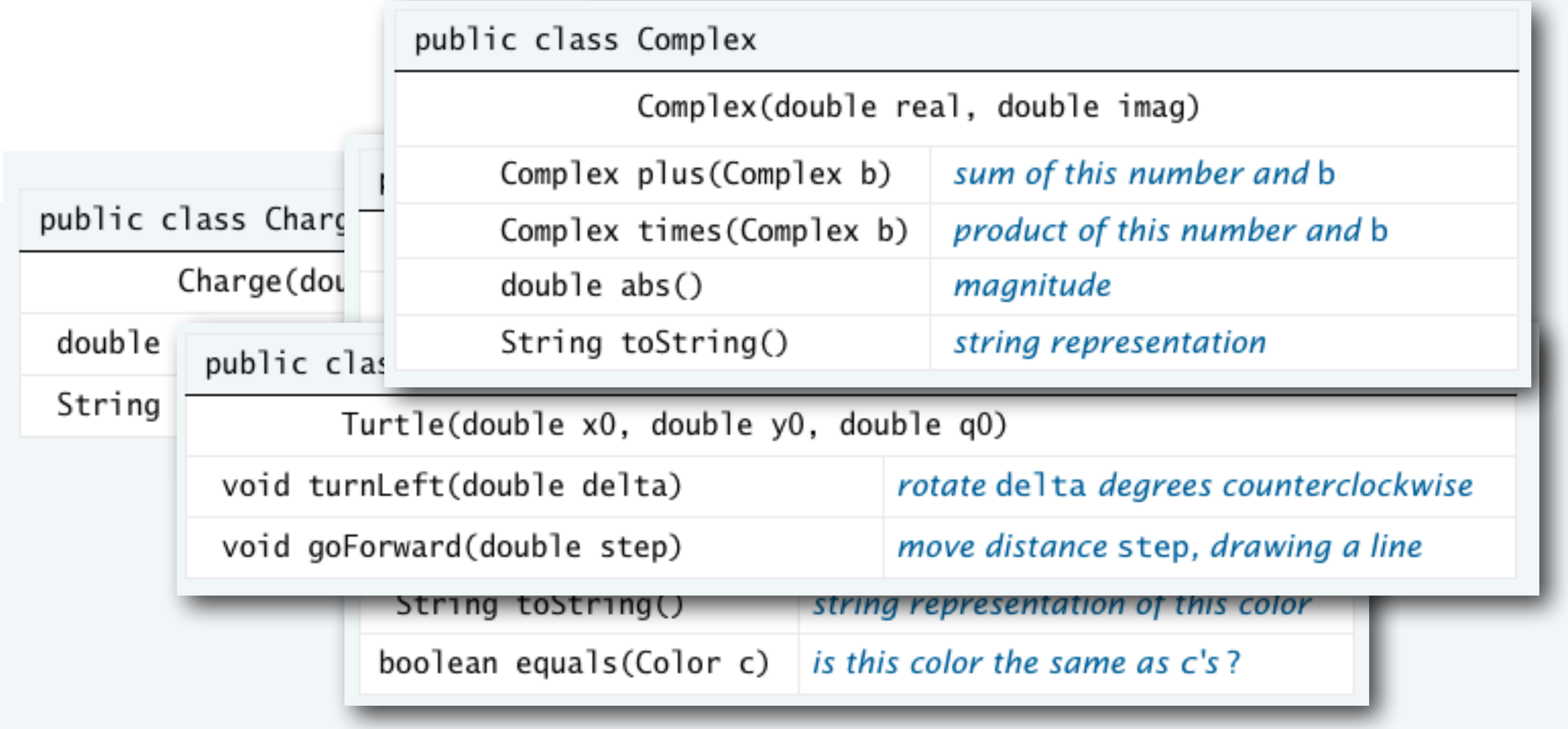
12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations

Choosing appropriate data structures

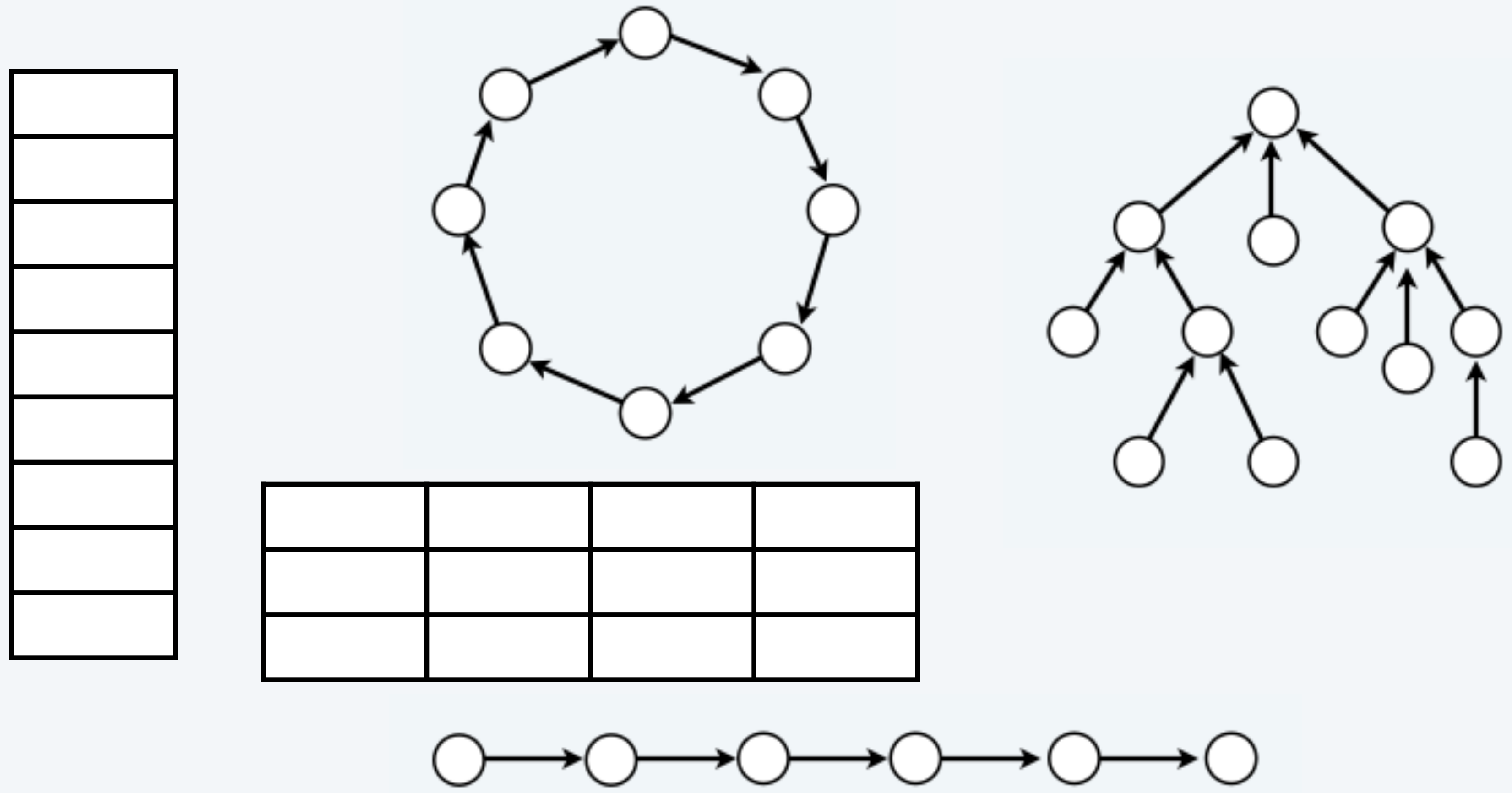
When implementing a Java class: Which data structures structures to use?

- Resource 1: How much memory is needed?
- Resource 2: How much time do data-type methods use?



Data structures

- Represent data.
- Represent relationships among data.
- Some are built in to Java: 1D arrays, 2D arrays, . . .
- Most are not: linked list, circular list, tree, . . .



Data structure comparison: arrays vs linked lists

- Arrays allow constant-time access of any element, but growth requires linear time
- Linked lists allow constant-time access of the first and last elements, and constant-time growth

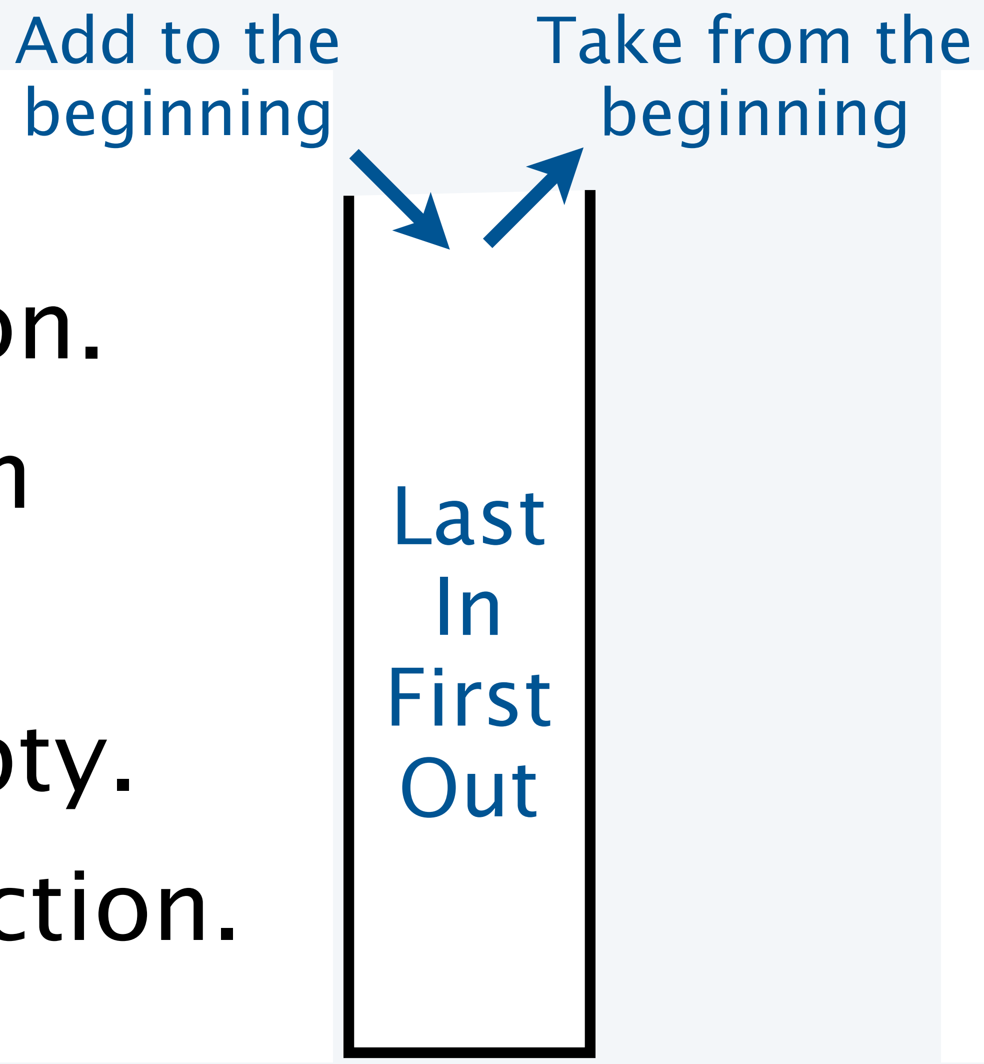
Stack and Queue APIs

A collection is an ADT whose values are a multiset of items, all of the same type.

Two fundamental collection ADTs differ in just a detail of the specification of their operations.

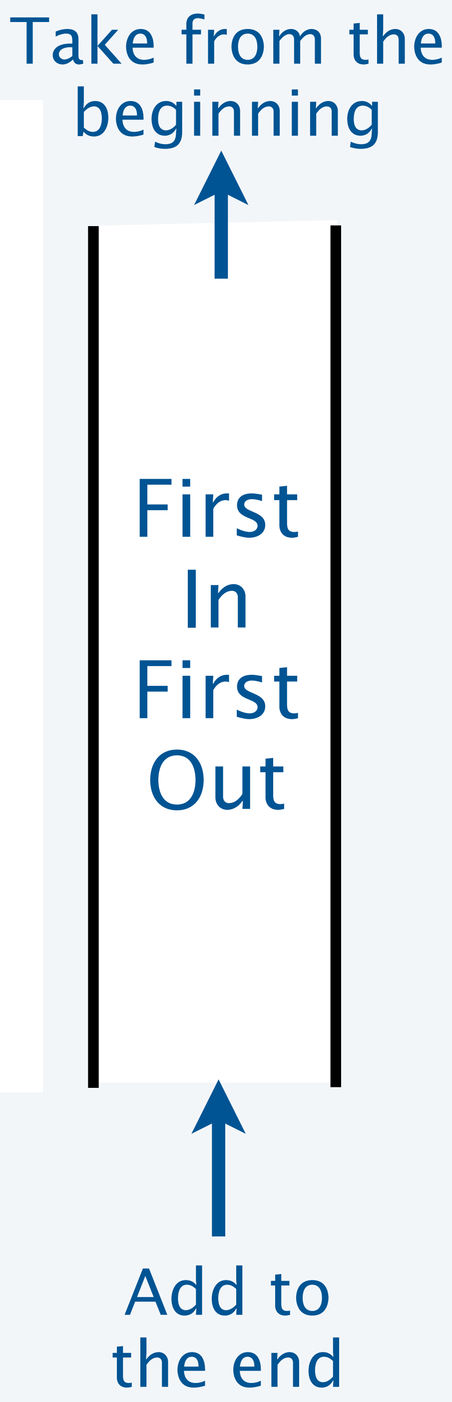
Stack operations

- Add an item to the collection.
- Remove and return the item *most* recently added (LIFO).
- Test if the collection is empty.
- Return the size of the collection.



Queue operations

- Add an item to the collection.
- Remove and return the item *least* recently added (FIFO).
- Test if the collection is empty.
- Return the size of the collection.



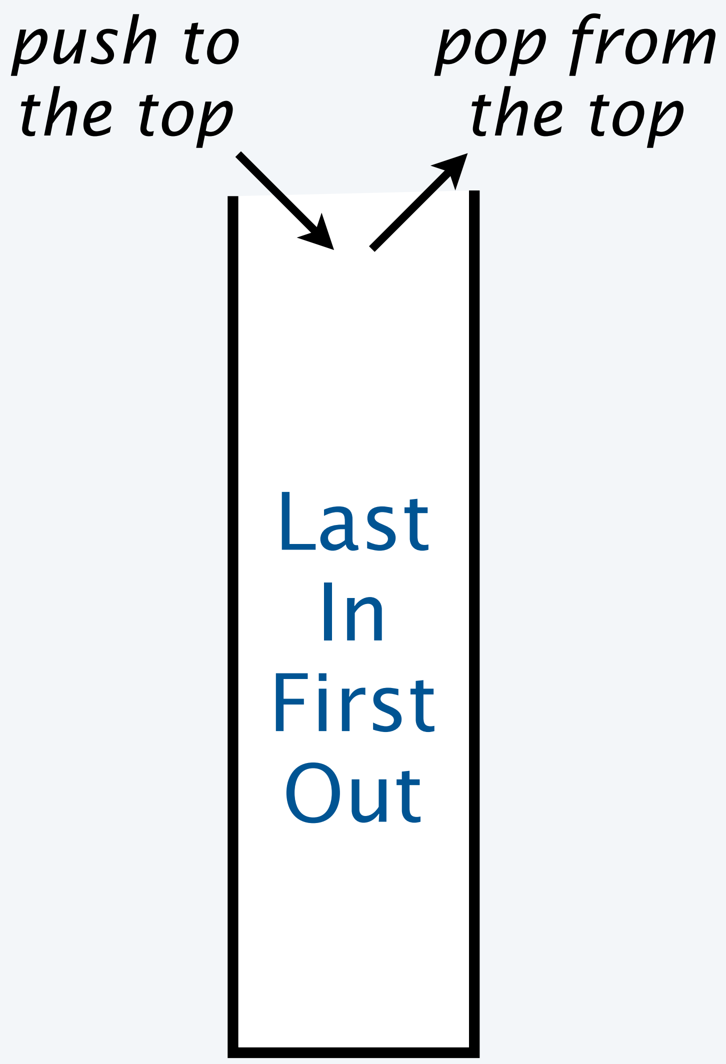
Stacks and queues both arise naturally in countless applications.

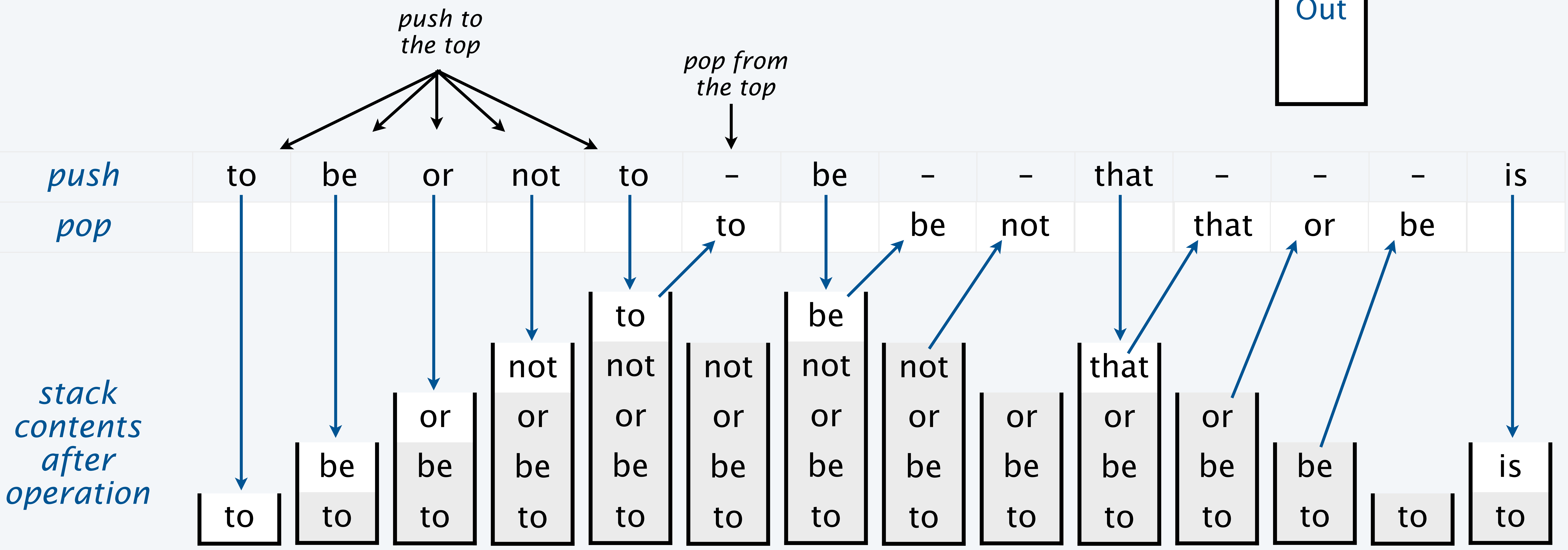
A key characteristic. No limit on the size of the collection.

Example of stack operations

Push. Add an item to the collection.

Pop. Remove and return the item *most* recently added.





push	to	be	or	not	to	-	be	-	-	that	-	-	-	is
pop						to		be	not		that	or	be	

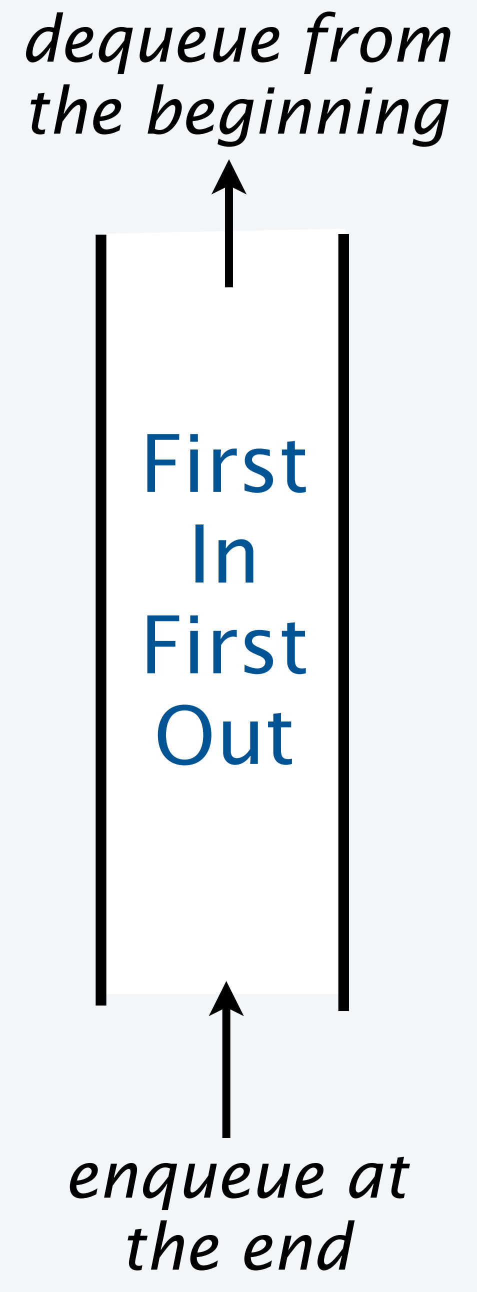
Stack

https://www.webstaurantstore.com/

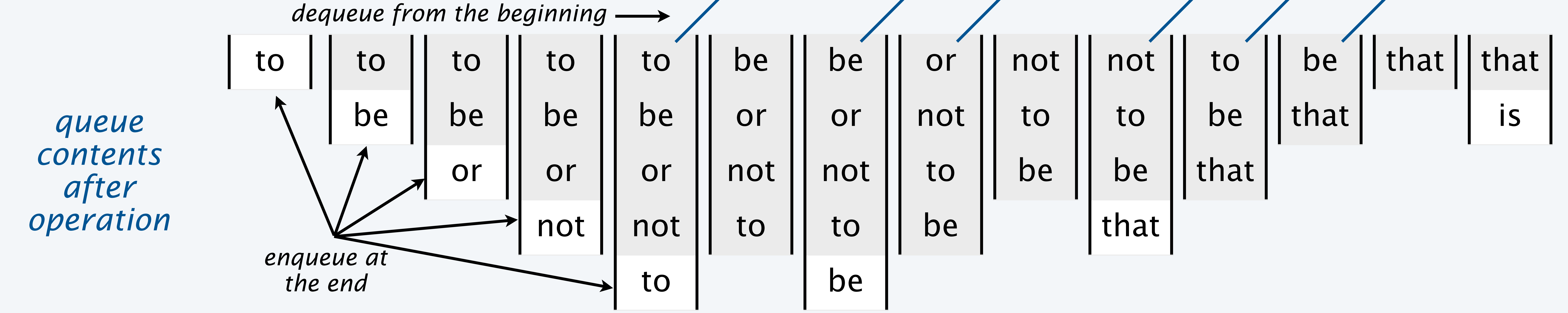
Example of queue operations

Enqueue. Add an item to the collection.

Dequeue. Remove and return the item *least* recently added.



enqueue	to	be	or	not	to	–	be	–	–	that	–	–	–	is
dequeue						to		be	or		not	to	be	



Queue

https://commons.wikimedia.org/wiki/File:People_waiting_a_train_of_Line_13_to_come_02.JPG

Parameterized data types

Goal. Simple, safe, and clear client code for collections of any type of data.

Java approach: Parameterized data types (generics)
- Use placeholder type name in definition.
- Substitute concrete type for placeholder in clients. ← stay tuned for examples

Stack API

public class Stack<Item>	
Stack<Item>()	*create a stack of items, all of type* Item
void push(Item item)	*add* item *to stack*
Item pop()	*remove and return the item most recently pushed*
boolean isEmpty()	*is the stack empty?*
int size()	*# of objects on the stack*

Queue API

public class Queue<Item>	
Queue<Item>()	*create a queue of items, all of type* Item
void enqueue(Item item)	*add* item *to queue*
Item dequeue()	*remove and return the item least recently enqueued*
boolean isEmpty()	*is the queue empty?*
int size()	*# of objects on the queue*

Performance specifications

Challenge. Provide guarantees on performance.

Goal. Simple, safe, clear, and *efficient* client code.

Typically required for client code to be *scalable*

Performance specifications

- All operations are constant-time.
- Memory use is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Java. Any implementation of the API implements the stack/queue abstractions.

This course: Implementations that do not meet performance specs *do not* implement the abstractions.

12. Stacks and Queues

- APIs
- **Clients**
- Strawman implementation
- Linked lists
- Implementations

Stack and queue applications

Queues

- First-come-first-served resource allocation.
- Asynchronous data transfer (`StdIn`, `StdOut`).
- Dispensing requests on a shared resource.
- Simulations of the real world.

Stacks

- Last-come-first-served resource allocation.
- Function calls in programming languages.
- Basic mechanism in interpreters, compilers.
- Fundamental abstraction in computing.

Queue client example: Read all strings from StdIn into an array

Challenge

- Can't store strings in array before creating the array.
- Can't create the array without knowing how many strings are in the input stream.
- Can't know how many strings are in the input stream without reading them all.

Solution: Use a `Queue<String>`.

Note: StdIn has this functionality

```
public class QEx
{
    public static String[] readAllStrings()
    { /* See next slide. */ }

    public static void main(String[] args)
    {
        String[] words = readAllStrings();
        for (int i = 0; i < words.length; i++)
            StdOut.println(words[i]);
    }
}
```

```
% more moby.txt
moby dick
herman melville
call me ishmael some years ago never
mind how long precisely having
little or no money
...
```

```
% java QEx < moby.txt
moby
dick
herman
melville
call
me
ishmael
some
years
...
```

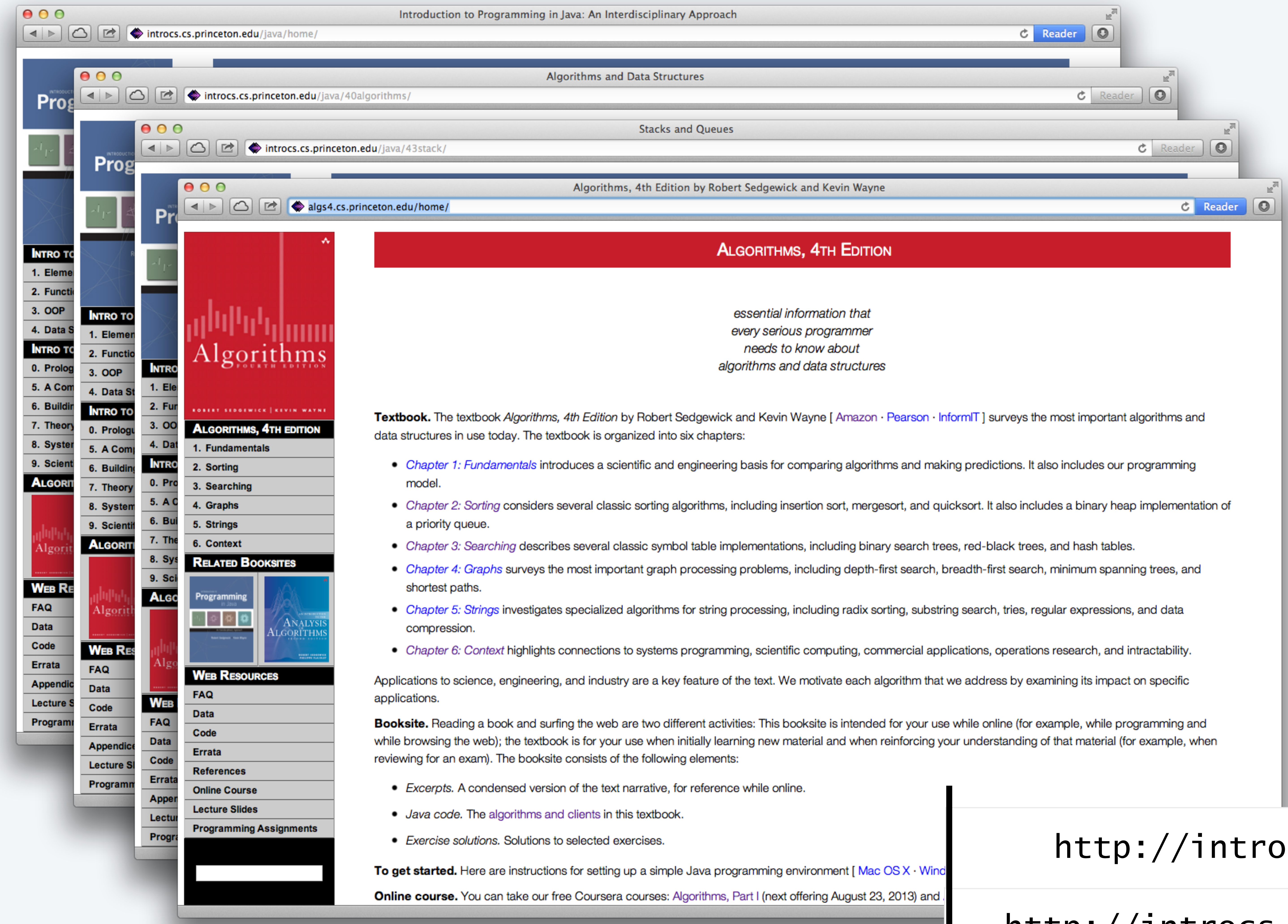
Queue client example: Read all strings from StdIn into an array

Solution: Use a Queue<String>.

- Store strings in the queue.
- Get the size when all have been read from StdIn.
- Create an array of that size.
- Copy the strings into the array.

```
public class QEx
{
    public static String[] readAllStrings()
    {
        Queue<String> q = new Queue<String>();
        while (!StdIn.isEmpty())
            q.enqueue(StdIn.readString());
        int N = q.size();
        String[] words = new String[N];
        for (int i = 0; i < N; i++)
            words[i] = q.dequeue();
        return words;
    }
    public static void main(String[] args)
    {
        String[] words = readAllStrings();
        for (int i = 0; i < words.length; i++)
            StdOut.println(words[i]);
    }
}
```

Stack example: "Back" button in a browser



Typical scenario

- Visit a page.
- Click a link to another page.
- Click a link to another page.
- Click a link to another page.
- Click "back" button.
- Click "back" button.
- Click "back" button.

http://introcs.cs.princeton.edu/java/43stack/

http://introcs.cs.princeton.edu/java/40algorithms/

http://introcs.cs.princeton.edu/java/home/

Autoboxing

Challenge. Use a *primitive* type in a parameterized ADT.

Wrapper types

- Each primitive type has a wrapper reference type.
- Wrapper type has larger set of operations than primitive type.
  Example: `Integer.parseInt()`.
- Instances of wrapper types are objects.
- Wrapper type can be used in a parameterized ADT.

primitive type	*wrapper type*
`int`	`Integer`
`char`	`Character`
`double`	`Double`
`boolean`	`Boolean`

Autoboxing. Automatic cast from primitive type to wrapper type.

Auto-unboxing. Automatic cast from wrapper type to primitive type.

Simple client code (no casts) →

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);        // Autobox    (int -> Integer)
int a = stack.pop();   // Auto-unbox (Integer -> int)
```

Stack client example: Postfix expression evaluation

Infix. Standard way of writing arithmetic expressions, using parentheses for precedence.

Example. (1 + ((2 + 3) * (4 * 5))) = (1 + (5 * 20)) = 101

Jan Łukasiewicz
1878–1956

Postfix. Write operator *after* operands (instead of in between them).

Example. 1 2 3 + 4 5 * * + ← also called "reverse Polish" notation (RPN)

Remarkable fact. No parentheses are needed!

There is only one way to parenthesize a postfix expression.

1 2 3 + 4 5 * * +

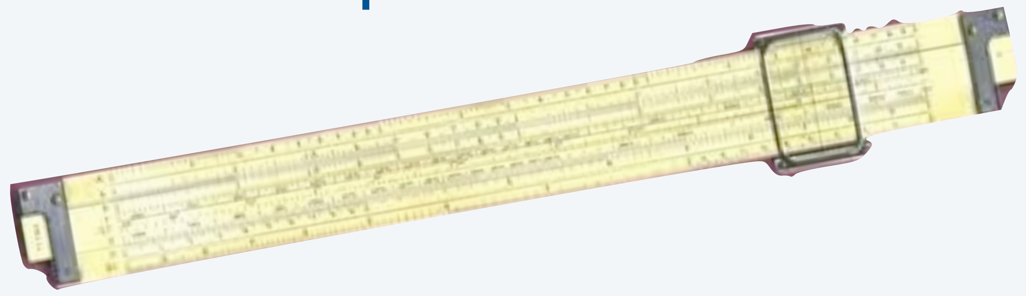
1 (2 + 3) 4 5 * * + ← find first operator, convert to infix, enclose in ()

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

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

iterate, treating subexpressions in parentheses as atomic

HP-35 (1972)
First handheld calculator.
"Enter" means "push".
No parentheses.



Made slide rules obsolete (!)

Next. With a stack, postfix expressions are easy to evaluate.

Postfix arithmetic expression evaluation

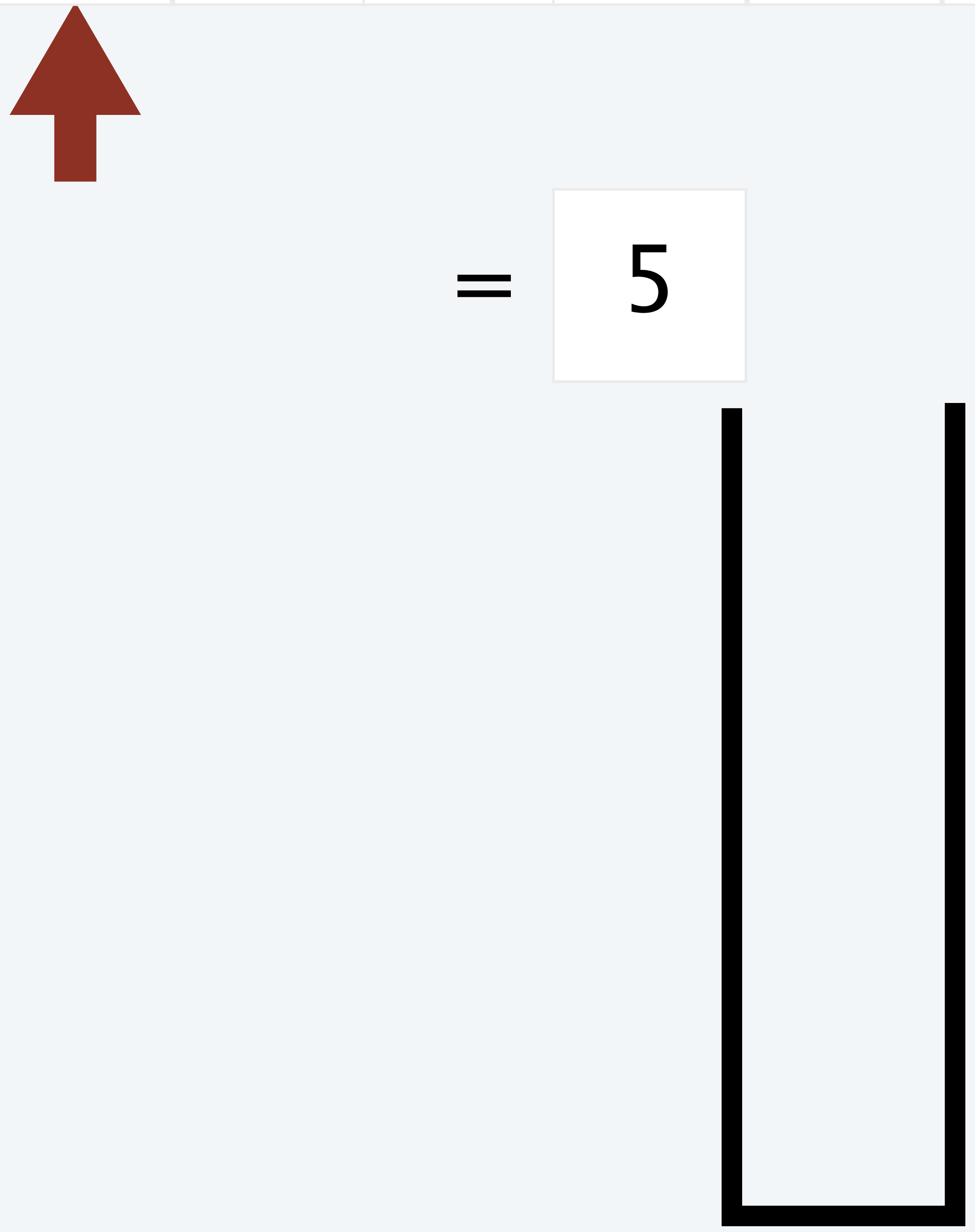
Algorithm

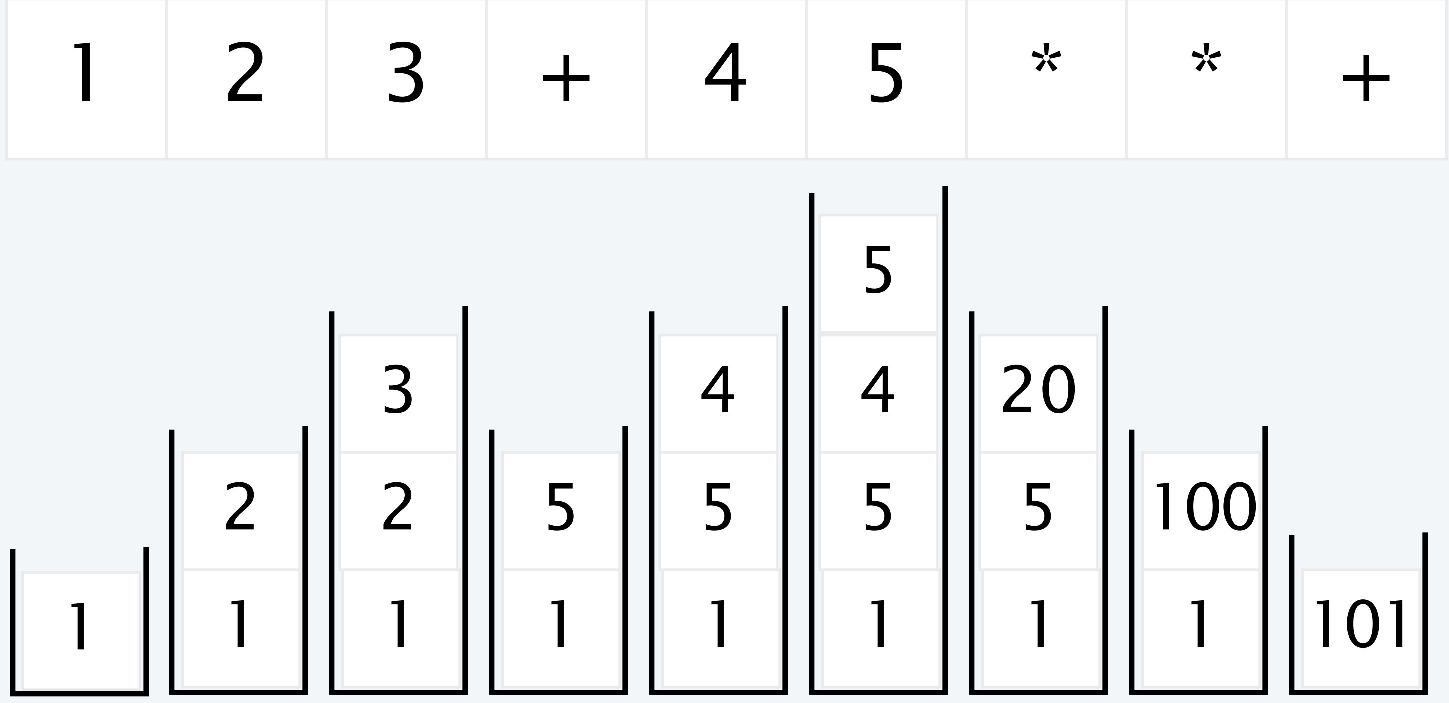
- While input stream is nonempty, read a token.
- Value: Push onto the stack.
- Operator: Pop operand(s), apply operator, push the result.

1	2	3	+	4	5	*	*	+

= 5

= 20 100 101





Stack client example: Postfix expression evaluation

```
public class Postfix
{
   public static void main(String[] args)
   {
      Stack<Double> stack = new Stack<Double>();
      while (!StdIn.isEmpty())
      {
         String token = StdIn.readString();
         if (token.equals("*"))
             stack.push(stack.pop() * stack.pop());
         else if (token.equals("+"))
             stack.push(stack.pop() + stack.pop());
         else if (token.equals("-"))
             stack.push(-stack.pop() + stack.pop());
         else if (token.equals("/"))
             stack.push((1.0/stack.pop()) * stack.pop());
         else if (token.equals("sqrt"))
             stack.push(Math.sqrt(stack.pop()));
         else
             stack.push(Double.parseDouble(token));
      }
      StdOut.println(stack.pop());
   }
}
```

```
% java Postfix
1 2 3 + 4 5 * * +
101.0
```

```
% java Postfix
1 5 sqrt + 2 /
1.618033988749895
```

$$\frac{1+\sqrt{5}}{2}$$

Perspective

- Easy to add operators of all sorts.
- Can do infix with two stacks (see text).
- Could output machine language code.
- Indicative of how Java compiler works.

Stack client example: Infix expression evaluation

Infix. Standard way of writing arithmetic expressions, using parentheses for precedence.

Example. (1 + ((2 + 3) * (4 * 5))) = (1 + (5 * 20)) = 101

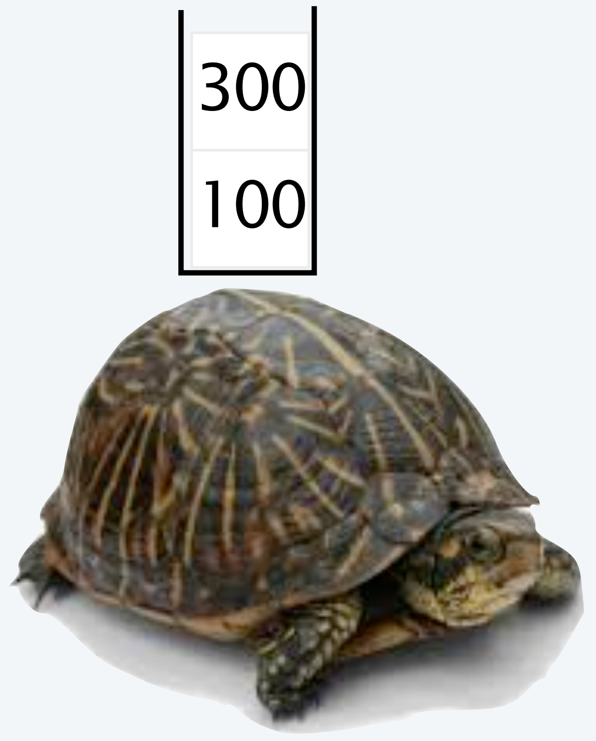
Edsger Dijkstra
1878–1956

Dijkstra. With *two* stacks, infix expressions are easy to evaluate.

Dijkstra's 2-stack algorithm

- While input stream is nonempty, read a token.
- Value: Push onto the value stack.
- Operator: Push onto the operator stack.
- Left paren: Ignore.
- Right paren: Pop two values, pop operator, apply operator to values, push the result.

Real-world stack application: PostScript



PostScript (Warnock-Geschke, 1980s): A turtle with a stack.

- Postfix program code (push literals; functions pop arguments).
- Add commands to drive virtual graphics machine.
- Add loops, conditionals, functions, types, fonts, strings....

PostScript code

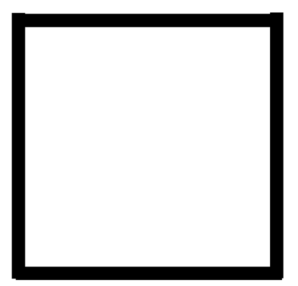
push(100)

call "moveto" (takes args from stack)

```
100 100 moveto
100 300 lineto
300 300 lineto
300 100 lineto
stroke
```

define a path

draw the path



A simple virtual machine, but not a toy

- Easy to specify published page.
- Easy to implement on various specific printers.
- Revolutionized world of publishing.

Another stack machine: The JVM (Java Virtual Machine)!

Image sources

http://pixabay.com/en/book-stack-learn-knowledge-library-168824/

http://upload.wikimedia.org/wikipedia/commons/2/20/Cars_in_queue_to_enter_Gibraltar_from_Spain.jpg

12. Stacks and Queues

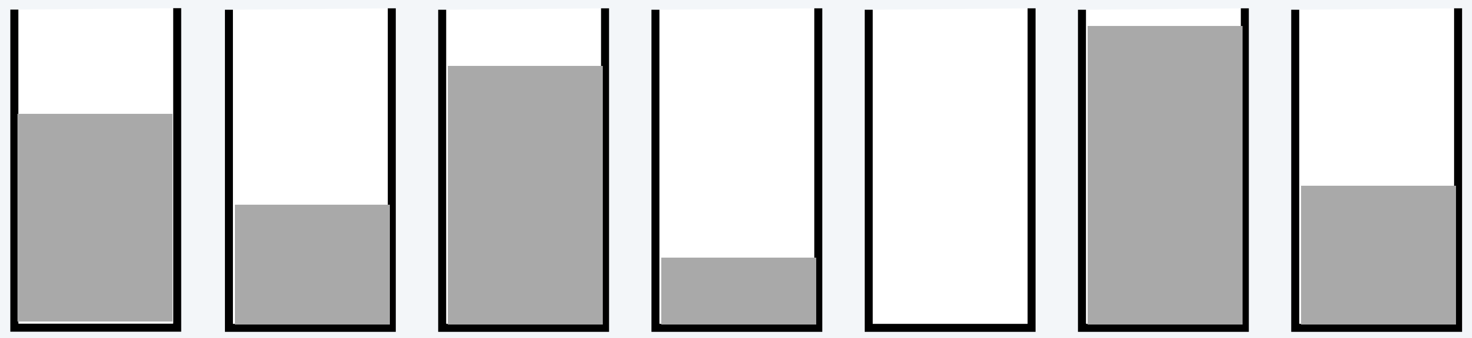
- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations

Strawman ADT for pushdown stacks

Warmup: simplify the ADT

- Implement only for items of type `String`.
- Have client provide a stack *capacity* in the constructor.

values



Strawman API

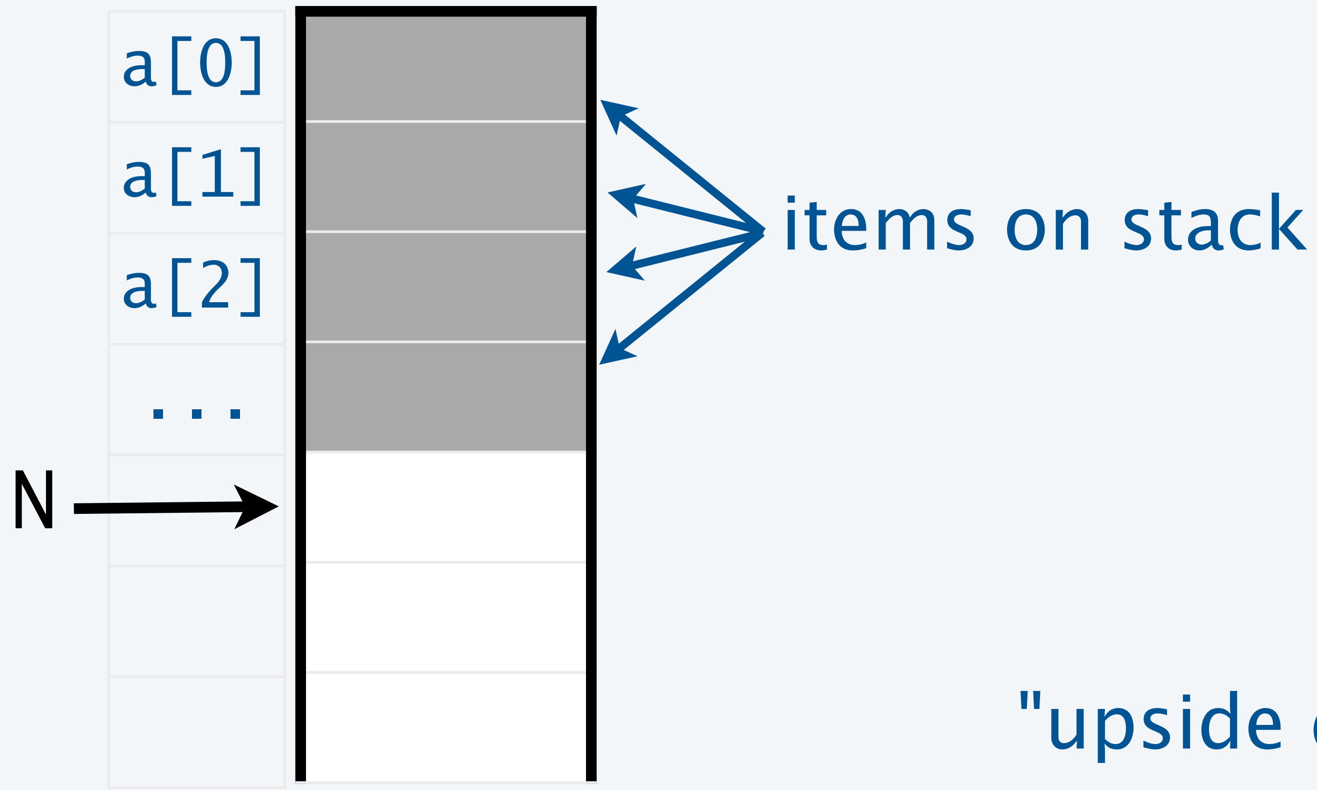
public class StrawStack	
`StrawStack(int max)`	*create a stack of capacity* `max`
`void push(String item)`	*add* `item` *to stack*
`String pop()`	*return the string most recently pushed*
`boolean isEmpty()`	*is the stack empty?*
`int size()`	*number of strings on the stack*

Rationale. Allows us to represent the collection with an array of strings.

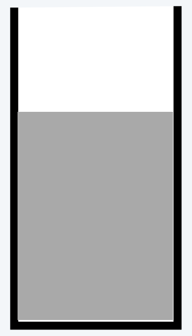
Strawman implementation: Instance variables and constructor

Data structure choice. Use an array to hold the collection.


```
public class StrawStack
{
   private String[] a;
   private int N = 0;

   public StrawStack(int max)
   {  a = new String[max];  }
...
}
```



"upside down"
representation of



Strawman stack implementation: Test client

```
public static void main(String[] args)
{
   int max = Integer.parseInt(args[0]);
   StrawStack stack = new StrawStack(max);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-"))
         StdOut.print(stack.pop());
      else
         stack.push(item);
   }
   StdOut.println();
}
```

instance variables

constructors

methods

test client

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

% java StrawStack 20 < tobe.txt
to be not that or be
```

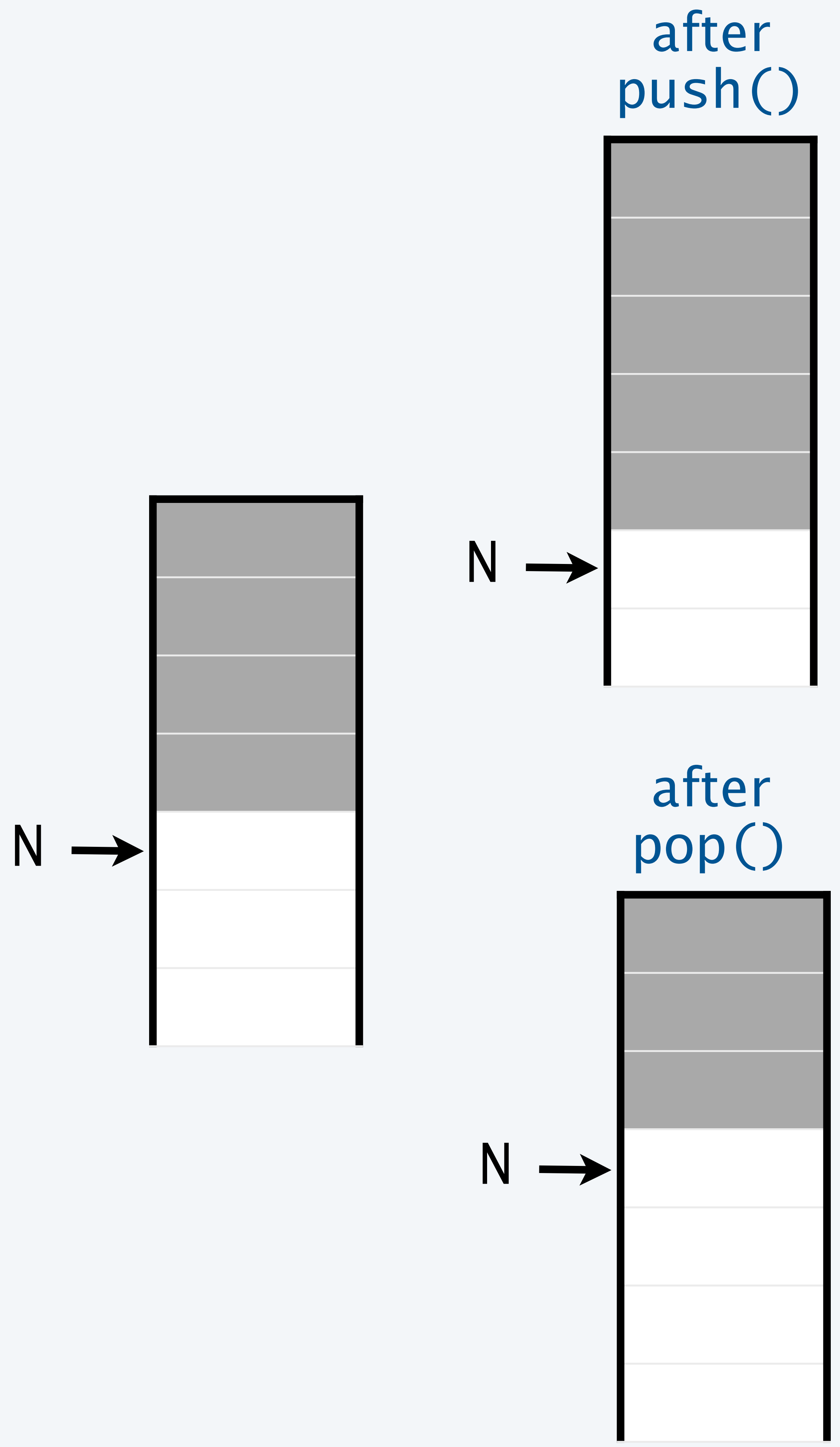
What we *expect*, once the implementation is done.

Strawman implementation: Methods

Methods define data-type operations (implement APIs).

```
public class StrawStack
{
...
   public boolean isEmpty()
   {  return (N == 0); }

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

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

   public int size()
   {  return N;   }
...
}
```

all constant-time one-liners!



Strawman pushdown stack implementation

```
public class StrawStack
{
   private String[] a;
   private int N = 0;

   public StrawStack(int max)
   {  a = new String[max];  }

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

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

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

   public int size()
   {  return N;  }

   public static void main(String[] args)
   {
      int max = Integer.parseInt(args[0]);
      StrawStack stack = new StrawStack(max);
      while (!StdIn.isEmpty())
      {
         String item = StdIn.readString();
         if (item.equals("-"))
            StdOut.print(stack.pop() + " ");
         else
            stack.push(item);
      }
      StdOut.println();
   }
}
```

instance variables

constructor

methods

test client

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

% java StrawStack 20 < tobe.txt
to be not that or be
```

Trace of strawman stack implementation (array representation)

push	to	be	or	not	to	–	be	–	–	that	–	–	–	is
pop						to		be	not		that	or	be	

stack contents after operation

a[0]	to	to	to	to	to	to	to	to	to	to	to	to	to	to
a[1]		be	be	be	be	be	be	be	be	be	be	be	be	is
a[2]			or	or	or	or	or	or	or	or	or	or	or	or
a[3]				not	not	not	not	not	not	that	that	that	that	that
a[4]					to	to	be	be	be	be	be	be	be	be
a[5]														
a[6]														
a[7]														
a[8]														
a[9]														
a[10]														
a[11]														
a[12]														
a[13]														
a[14]														
a[15]														
a[16]														
a[17]														
a[18]														
a[19]														

↑ N

Significant wasted space when stack size is not near the capacity (typical).

Benchmarking the strawman stack implementation

StrawStack implements a *fixed-capacity collection that behaves like a stack* if the data fits.

It does *not* implement the stack API or meet the performance specifications.

StrawStack requires client to provide capacity

Stack API

public class Stack<Item>	
Stack<Item>() ✗	*create a stack of items, all of type* Item
void push(Item item)	*add* item *to stack*
✗ Item pop()	*remove and return the item most recently pushed*
boolean isEmpty()	*is the stack empty?*
int size()	*# of items on the stack*

StrawStack works only for strings

Performance specifications

- All operations are constant-time. ✓
- Memory use is linear in the size of the collection, when it is nonempty. ✗
- No limits within the code on the collection size. ✗

Nice try, but need a new *data structure.*

CS.12.C.StacksQueues.Strawman

12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- **Linked lists**
- Implementations

Data structures: sequential vs. linked

Sequential data structure

- Put objects next to one another.
- Machine: consecutive memory cells.
- Java: array of objects.
- Fixed size, arbitrary access. ⟵ *i*th element

Linked data structure

- Associate with each object a link to another one.
- Machine: link is memory address of next object.
- Java: link is reference to next object.
- Variable size, sequential access. ⟵ *next* element
- Overlooked by novice programmers.
- Flexible, widely used method for organizing data.

Array at C0

addr	*value*
C0	"Alice"
C1	"Bob"
C2	"Carol"
C3	
C4	
C5	
C6	
C7	
C8	
C9	
CA	
CB	

Linked list at C4

addr	*value*
C0	"Carol"
C1	null
C2	
C3	
C4	"Alice"
C5	CA
C6	
C7	
C8	
C9	
CA	"Bob"
CB	C0

Simplest singly-linked data structure: linked list

Linked list

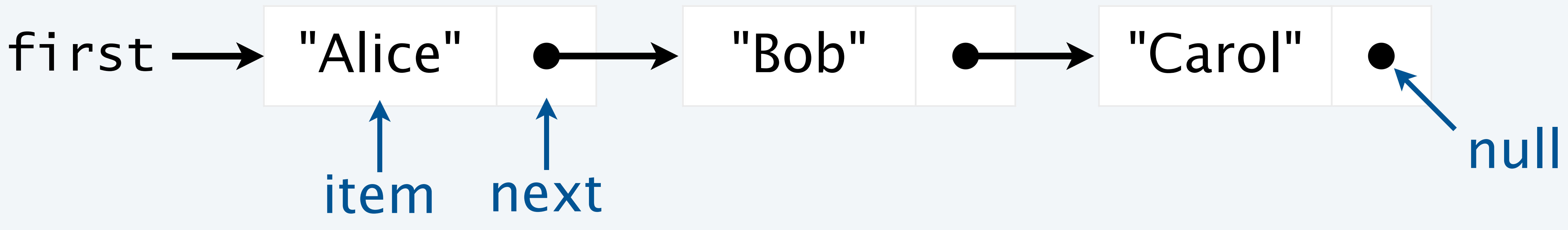
- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

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

Representation

- Use a private nested class Node to implement the node abstraction.
- For simplicity, start with nodes having two values: a `String` and a Node.

A linked list



14. Stacks and Queues

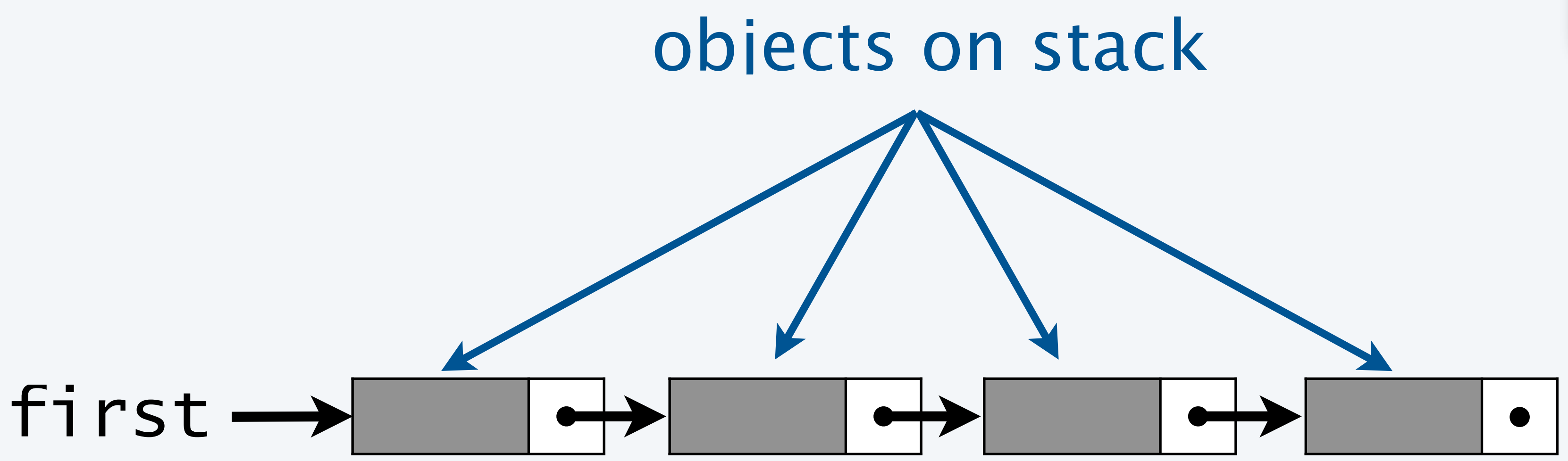
- APIs
- Clients
- Strawman implementation
- Linked lists
- **Implementations**

Pushdown stack implementation: Instance variables and constructor

Data structure choice. Use a linked list to hold the collection.


```
public class Stack<Item>
{
   private Node first = null;
   private int N = 0;

   private class Node
   {
      private Item item;
      private Node next;
   }
...
}
```

use in place of concrete type



Annoying exception (not a problem here).

Can't declare an array of `Item` objects (don't ask why).

Need cast: `Item[] a = (Item[]) new Object[N]`

Stack implementation: Test client


```
public static void main(String[] args)
{
   Stack<String> stack = new Stack<String>();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-"))
         System.out.print(stack.pop() + " ");
      else
         stack.push(item);
   }
   StdOut.println();
}
```

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

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

What we *expect*, once the implementation is done.

Stack implementation: Methods

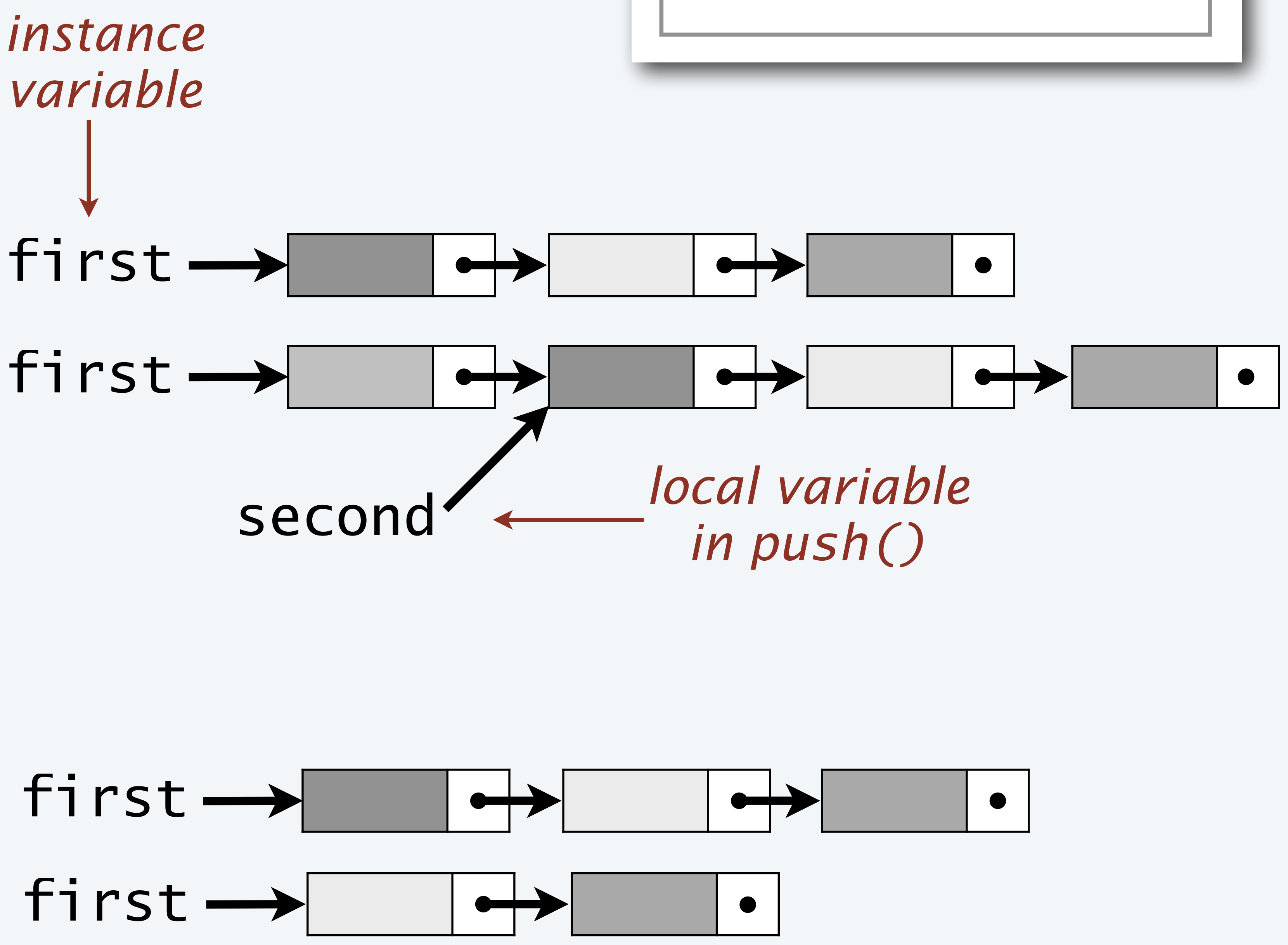
Methods define data-type operations (implement the API).

```
public class Stack<Item>
{
...
   public boolean isEmpty()
   {  return first == null;  }
   public void push(Item item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
      N++;
   }
   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      N--;
      return item;
   }
   public int size()
   {  return N;  }
...
}
```

might also use N == 0

add a new node to the beginning of the list

remove and return first item on list



Stack implementation

```
public class Stack<Item>
{
   private Node first = null;
   private int N = 0;
   private class Node
   {
      private Item item;
      private Node next;
   }
   public boolean isEmpty()
   {  return first == null;  }
   public void push(Item item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
      N++;
   }
   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      N--;
      return item;
   }
   public int size()
   {  return N;  }
   public static void main(String[] args)
   { // See earlier slide   }
}
```

instance variables

nested class

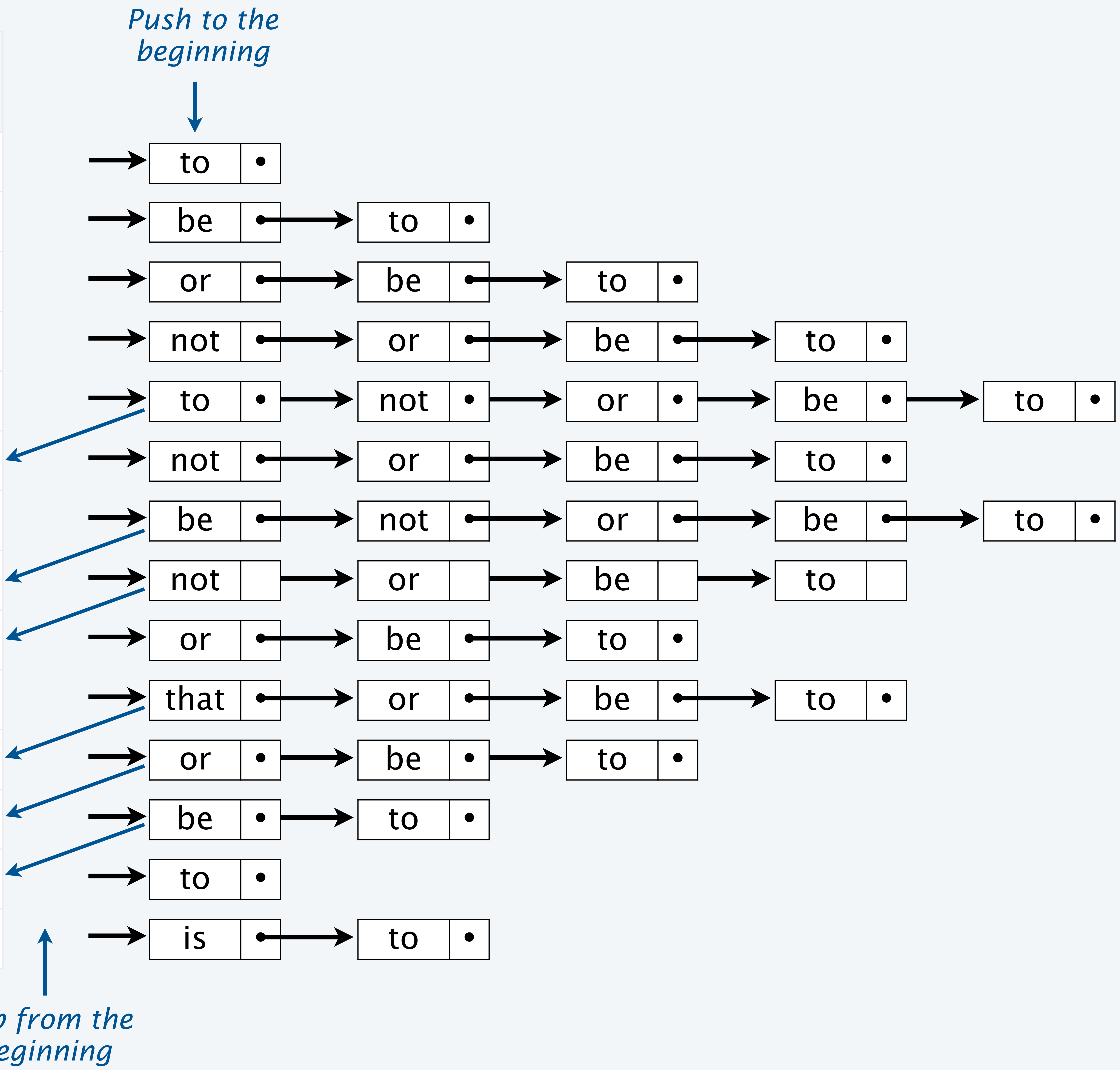
methods

test client

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

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

Trace of stack implementation (linked list representation)

push	*pop*
to	
be	
or	
not	
to	
–	to
be	
–	be
–	not
that	
–	that
–	or
–	be
is	



Benchmarking the stack implementation

Stack implements the stack abstraction.

It *does* implement the API and meet the performance specifications.

Stack API

public class Stack<Item>	
Stack<Item>()	*create a stack of items, all of type* Item
void push(Item item)	*add* item *to stack*
Item pop()	*remove and return the item most recently pushed*
boolean isEmpty()	*is the stack empty?*
int size()	*# of items on the stack*

✓

Performance specifications

- All operations are constant-time. ✓
- Memory use is linear in the size of the collection, when it is nonempty. ✓
- No limits within the code on the collection size. ✓

Made possible by *linked data structure.*

dequeue(): same code as pop()
enqueue(): slightly more complicated

Also possible to implement the *queue* abstraction with a singly-linked list (see text).

ADT for queues

A queue is an idealized model of a FIFO storage mechanism.

An ADT allows us to write Java programs that use and manipulate queues.

API

public class Queue<Item>	
`Queue<Item>()`	*create a queue of objects, all of type* Item
`void enqueue(Item item)`	*push* `item` *onto the queue*
`Item dequeue()`	*remove and return the object most recently enqueued*
`boolean isEmpty()`	*is the queue empty?*
`int size()`	*# of objects on the queue*

Performance specs

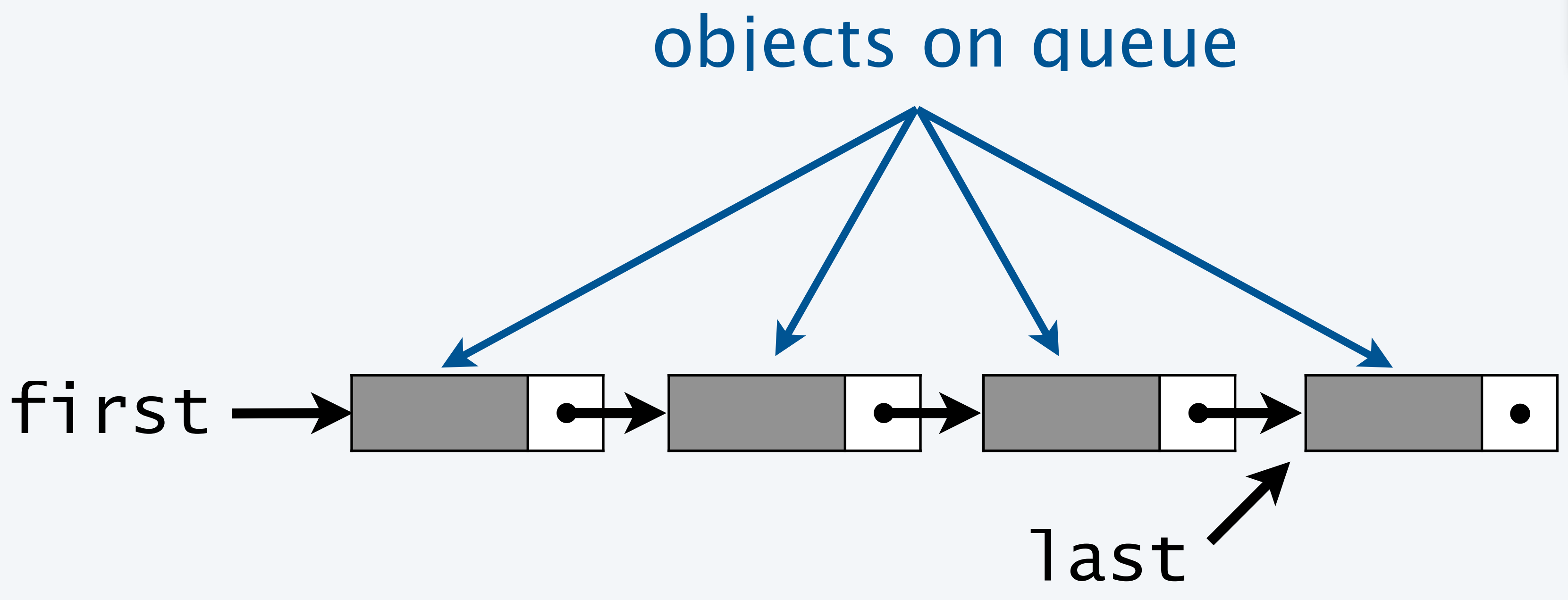
- All operations are constant-time.
- Memory use is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Queue implementation: Instance variables and constructor

Data structure choice. Use a linked list to hold the collection.

```
public class Queue<Item>
{
   private Node first = null;
   private Node last = null;
   private int N = 0;

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

   public Queue()
   {
      first = null;
      last = null
      N = 0;
   }
...
}
```

default constructor
redundant code
not needed



Queue implementation: Test client

```
public static void main(String[] args)
{
   Queue<String> q = new Queue<String>();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.compareTo("-") != 0)
         q.enqueue(item);
      else
         System.out.print(q.dequeue());
   }
   System.out.println();
}
```

instance variables

constructors

methods

test client

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

% java Queue < tobe.txt
to be or not to be
```

What we *expect*, once the implementation is done.

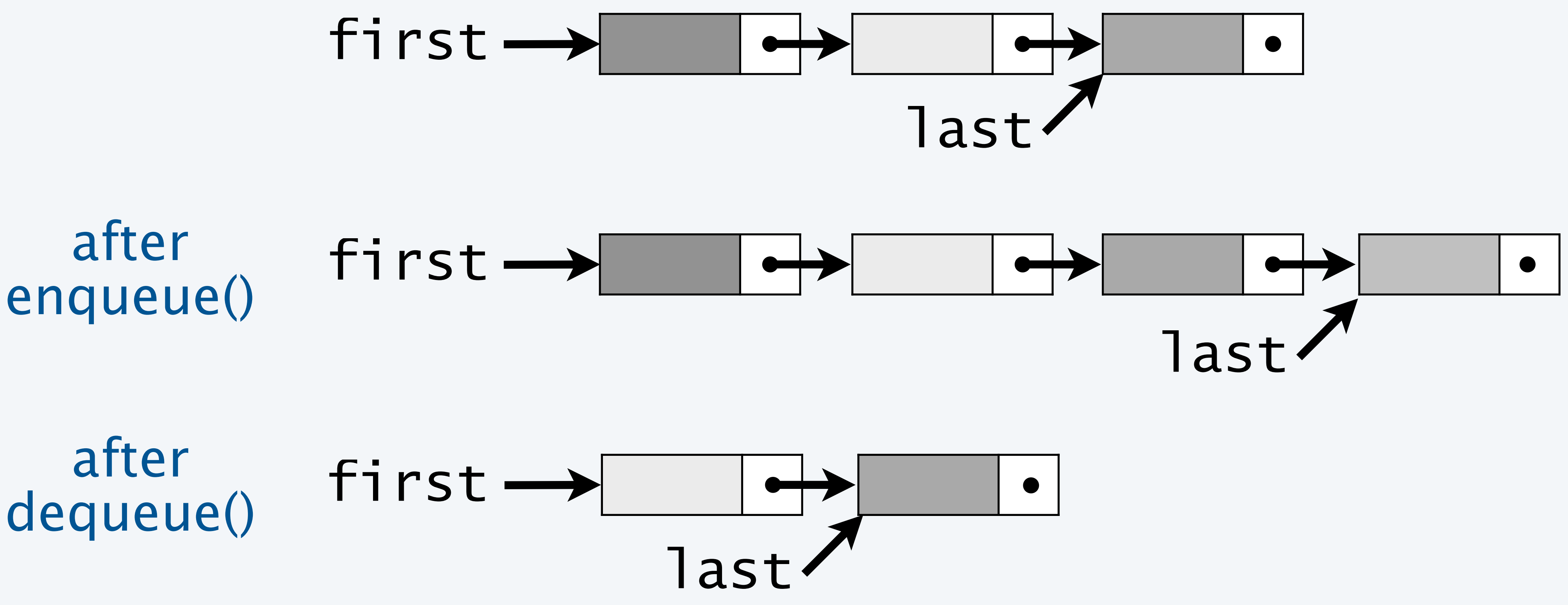
Queue implementation: Methods

Methods define data-type operations (implement the API).

```
public class Queue<Item>
{
...
   public boolean isEmpty()
   {  return first == null;   }
   public void enqueue(Item item)
   {
      last.next = new Node();
      last = last.next;
      last.item = item;
      N++;
   }
   public Item dequeue()
   {
      String item = first.item;
      first = first.next;
      N--;
      return item;
   }
   public int size()
   {  return N;   }
...
}
```

add a new node to the end of the list

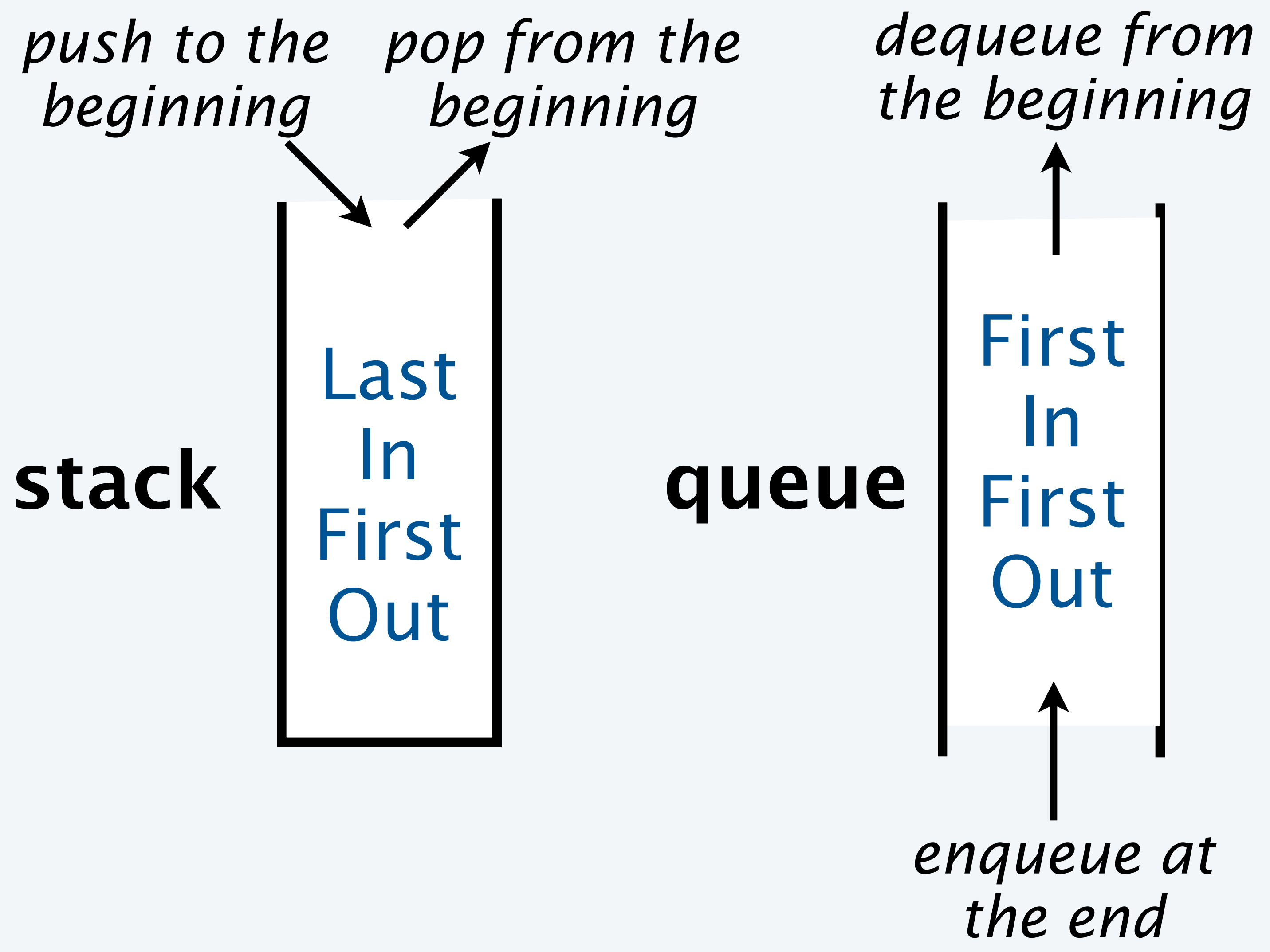
remove and return first item on list (same as stack pop)



Summary

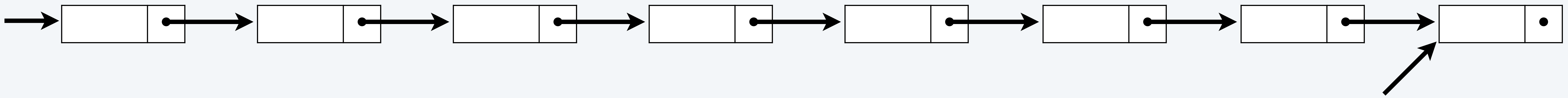
Stacks and queues

- Fundamental collection abstractions.
- Differ only in order in which items are removed.
- Performance specifications: Constant-time for all operations and space linear in the number of objects.



Linked structures

- Fundamental alternative to arrays.
- Enable implementations of the stack/queue abstractions *that meet performance specifications*.



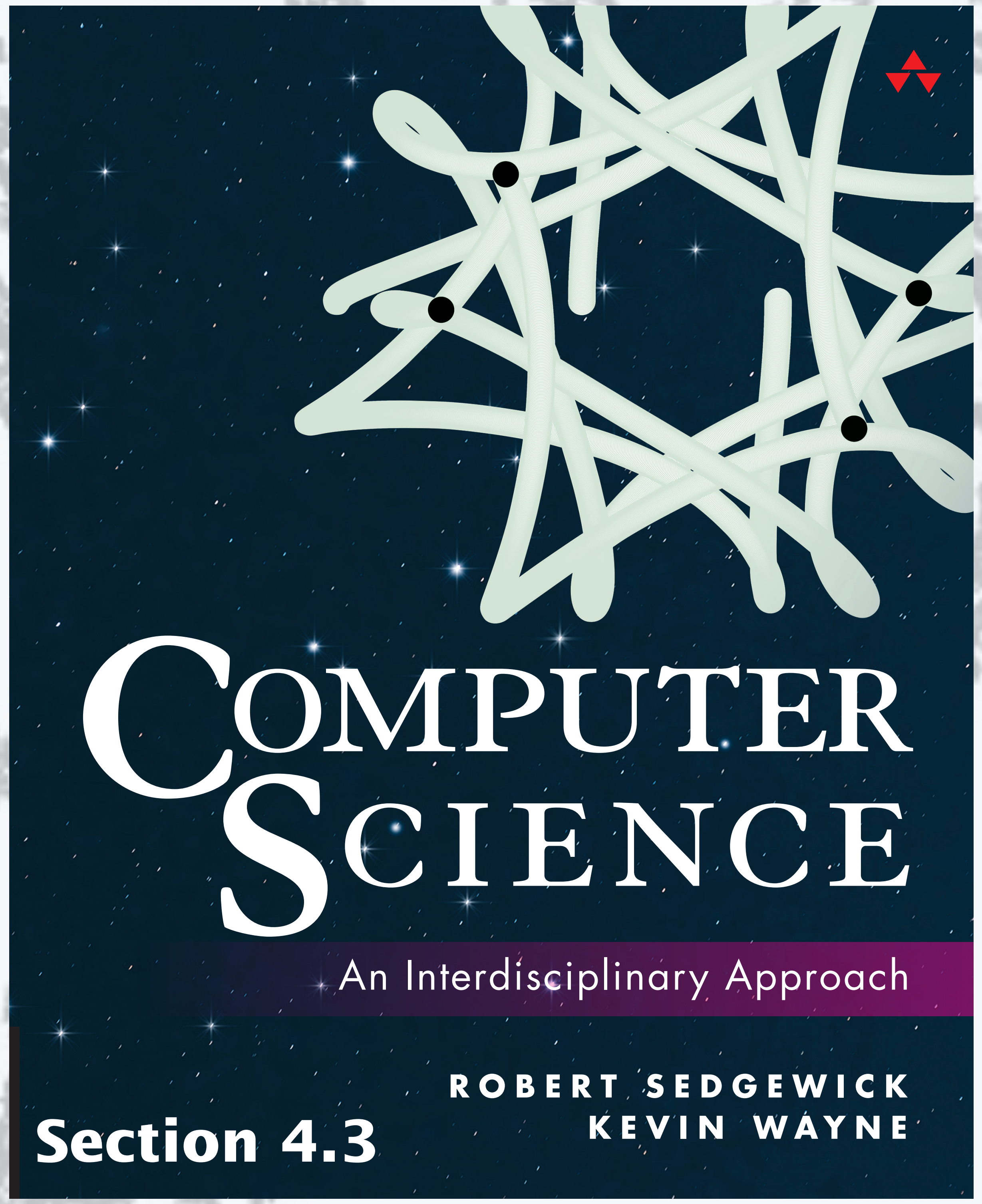
Next: *Symbol tables*

CS.12.E.StacksQueues.Implementations

COMPUTER SCIENCE

SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES



http://introcs.cs.princeton.edu

12. Stacks and Queues

Introduce Homework