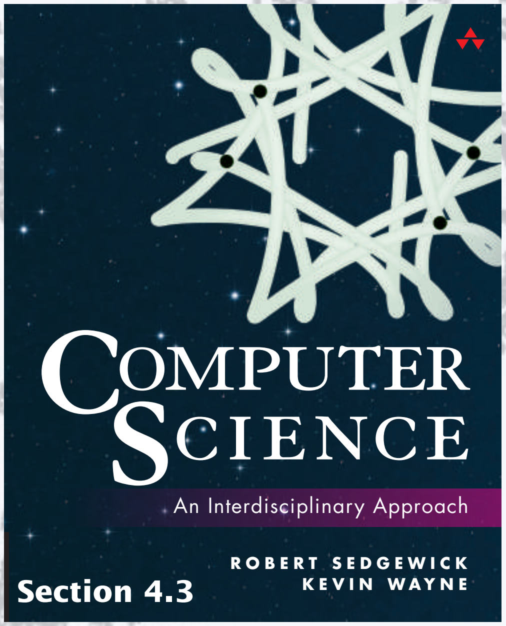
COMPUTER SCIENCE
SEDGEWICK/WAYNE

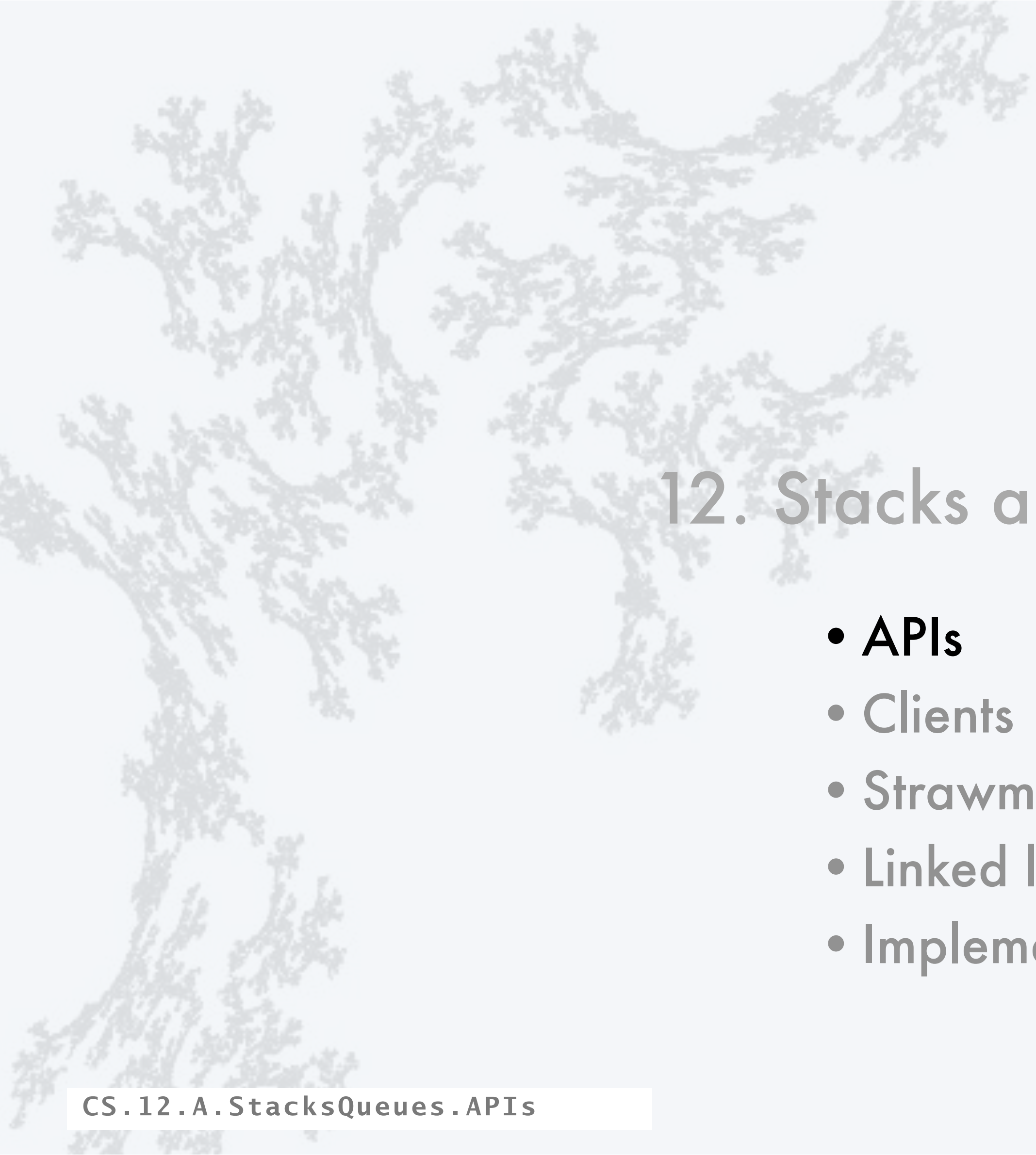
PART II: ALGORITHMS, THEORY, AND MACHINES



Section 4.3

http://introcs.cs.princeton.edu

12. Stacks and Queues



COMPUTER SCIENCE
SEDGEWICK/WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES

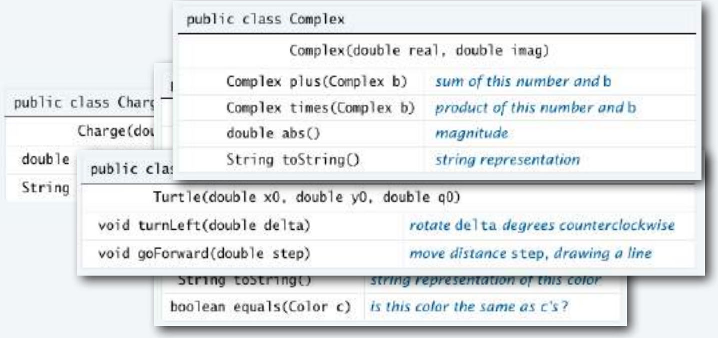
12. Stacks and Queues

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

Data types and data structures

Data types

- Set of values.
- Set of operations on those values.
- Some are built in to Java: `int`, `double`, `String`, . . .
- Most are not: `Complex`, `Picture`, `Charge`, . . .



public class Complex	
Complex(double real, double imag)	
Complex plus(Complex b)	*sum of this number and* b
Complex times(Complex b)	*product of this number and* b
double abs()	*magnitude*
String toString()	*string representation*

public clas	
Turtle(double x0, double y0, double q0)	
void turnLeft(double delta)	*rotate* delta *degrees counterclockwise*
void goForward(double step)	*move distance* step, *drawing a line*

String toString()	*string representation of this color*
boolean equals(Color c)	*is this color the same as c's?*

public class Charg

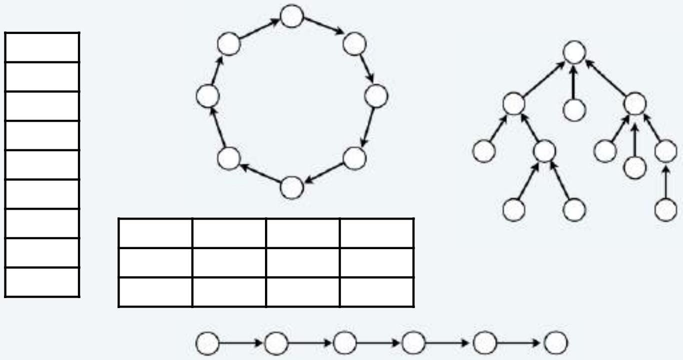
Charge(dou

double

String

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, . . .



Design challenge for every data type: Which data structure to use?

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

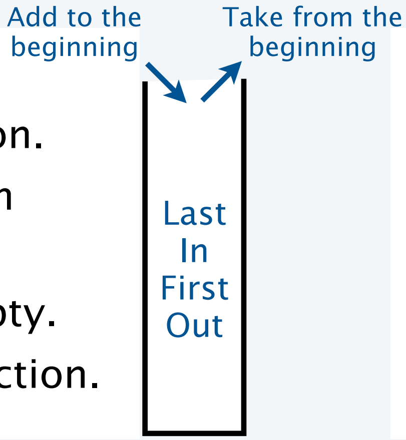
Stack and Queue APIs

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.

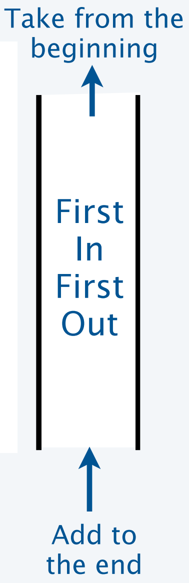
Stack operations

- 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.



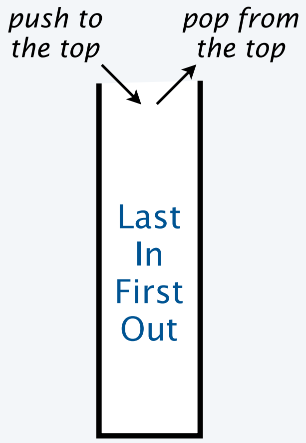
Stacks and queues both arise naturally in countless applications.

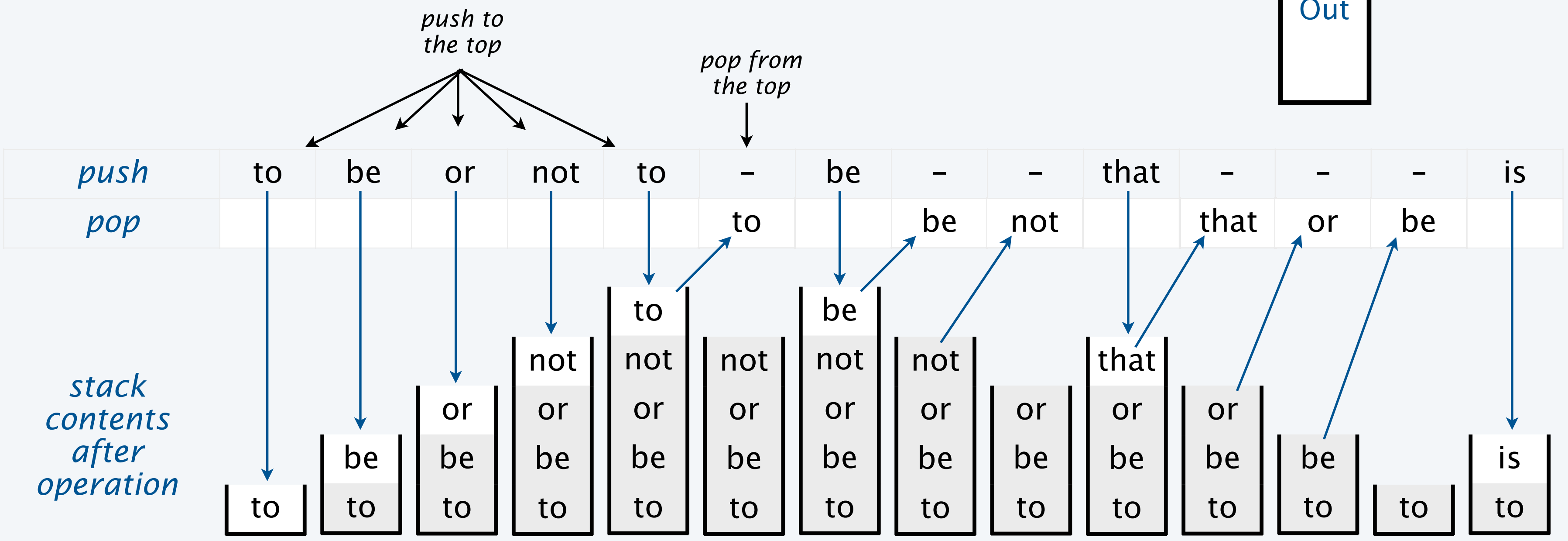
A key characteristic. No limit on the size of the collection.

Example of stack operations

Push. Add an item to the collection.

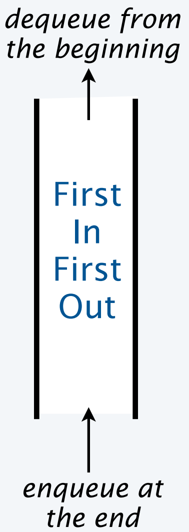
Pop. Remove and return the item *most* recently added.





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

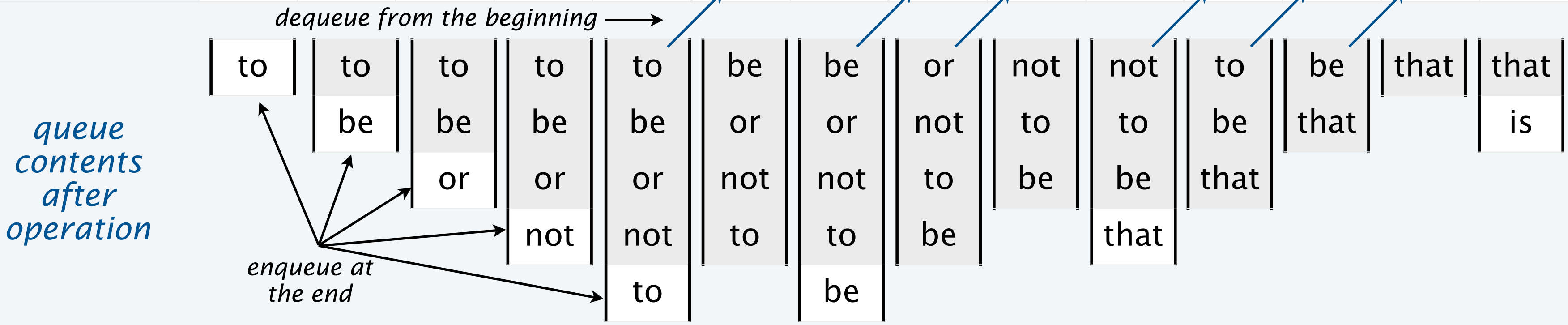
Example of queue operations



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	



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.

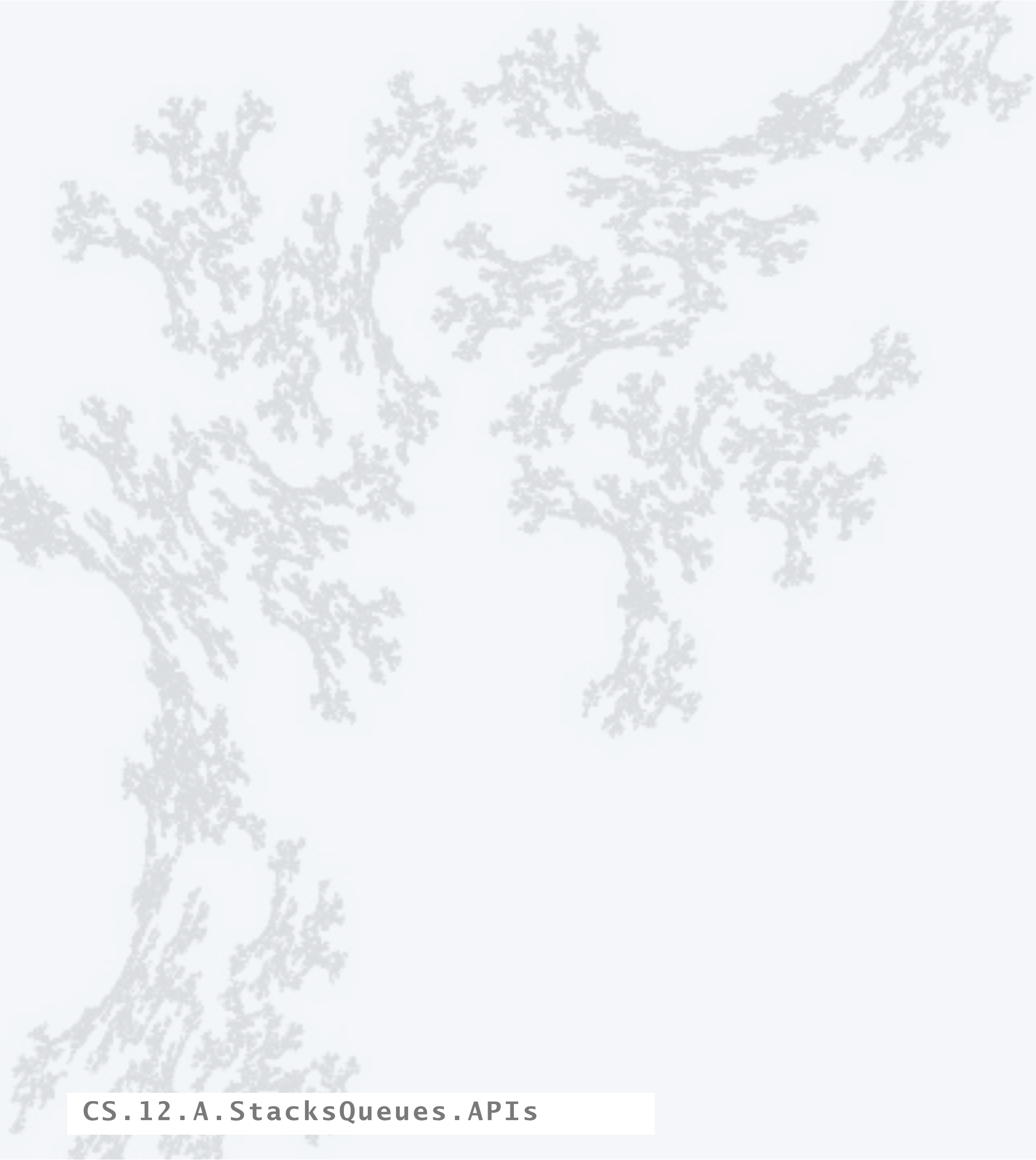
Typically required for client code to be *scalable*

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.

RS+KW. Implementations that do not meet performance specs *do not* implement the abstractions.



COMPUTER SCIENCE

SEDGEWICK/WAYNE

PART I: PROGRAMMING IN JAVA

CS.12.A.StacksQueues.APIs

COMPUTER SCIENCE
SEDGEWICK / WAYNE
PART II: ALGORITHMS, THEORY, AND MACHINES

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
...
```

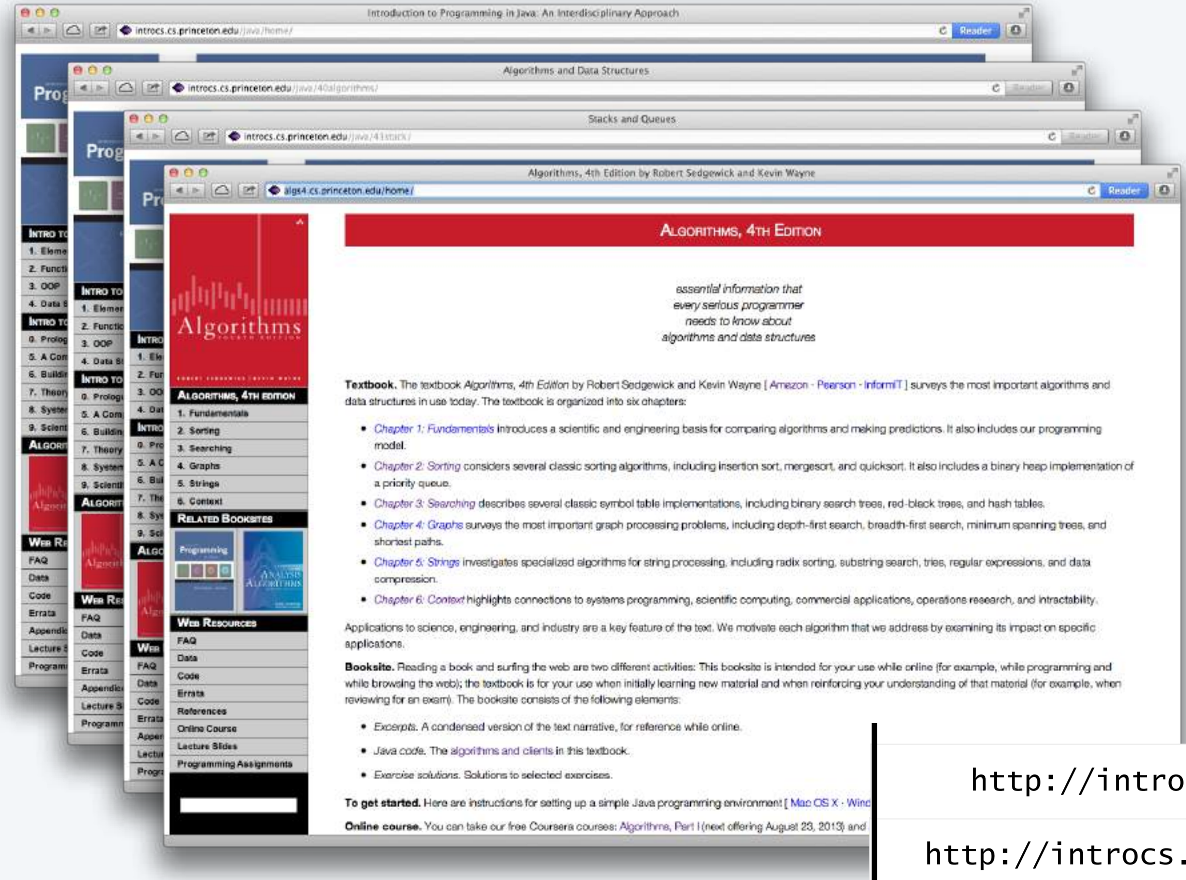
Queue client example: Read all strings from StdIn into an array

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



ALGORITHMS, 4TH EDITION

essential information that
every serious programmer
needs to know about
algorithms and data structures

Textbook. The textbook *Algorithms, 4th Edition* by Robert Sedgewick and Kevin Wayne [Amazon · Pearson · InformIT] surveys the most important algorithms and data structures in use today. The textbook is organized into six chapters:

- *Chapter 1: Fundamentals* introduces a scientific and engineering basis for comparing algorithms and making predictions. It also includes our programming model.
- *Chapter 2: Sorting* considers several classic sorting algorithms, including insertion sort, mergesort, and quicksort. It also includes a binary heap implementation of a priority queue.
- *Chapter 3: Searching* describes several classic symbol table implementations, including binary search trees, red-black trees, and hash tables.
- *Chapter 4: Graphs* surveys the most important graph processing problems, including depth-first search, breadth-first search, minimum spanning trees, and shortest paths.
- *Chapter 5: Strings* investigates specialized algorithms for string processing, including radix sorting, substring search, tries, regular expressions, and data compression.
- *Chapter 6: Context* highlights connections to systems programming, scientific computing, commercial applications, operations research, and intractability.

Applications to science, engineering, and industry are a key feature of the text. We motivate each algorithm that we address by examining its impact on specific applications.

Booksite. Reading a book and surfing the web are two different activities: This booksite is intended for your use while online (for example, while programming and while browsing the web); the textbook is for your use when initially learning new material and when reinforcing your understanding of that material (for example, when reviewing for an exam). The booksite consists of the following elements:

- *Excerpts.* A condensed version of the text narrative, for reference while online.
- *Java code.* The algorithms and clients in this textbook.
- *Exercise solutions.* Solutions to selected exercises.

To get started. Here are instructions for setting up a simple Java programming environment [Mac OS X · Wind

Online course. You can take our free Coursera courses: Algorithms, Part I (next offering August 23, 2013) and

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 * * +

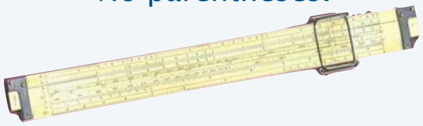
1 (2 + 3) 4 5 * * + ← find first operator, convert to infix, enclose in ()

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.



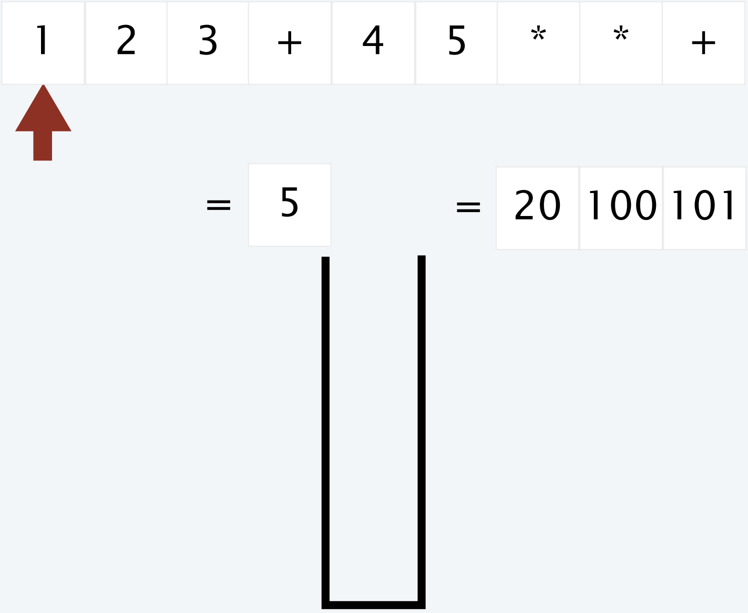
Made slide rules obsolete (!)

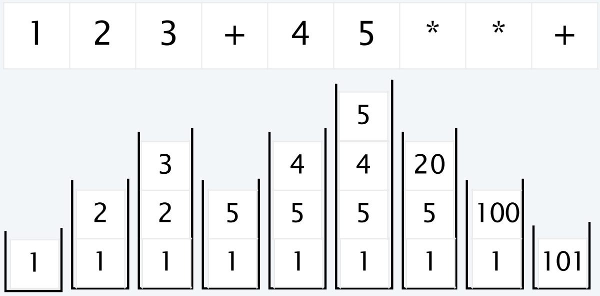
Next. With a stack, postfix expressions are easy to evaluate.

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.





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
```

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

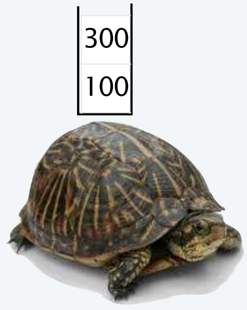
Perspective

- 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.

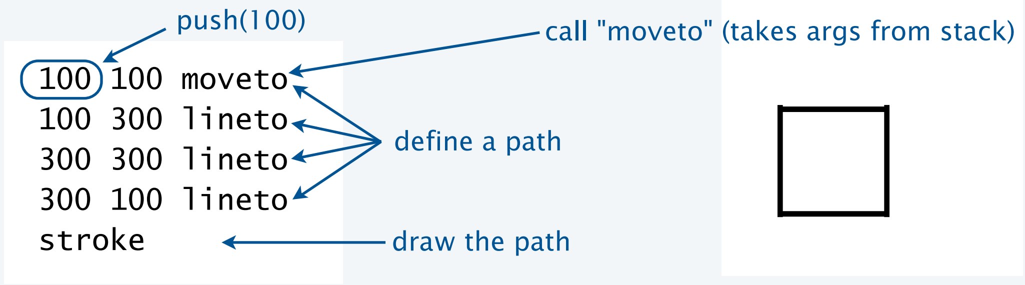
Real-world stack application: PostScript

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



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

push(100)

call "moveto" (takes args from stack)

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

COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES

12. Stacks and Queues

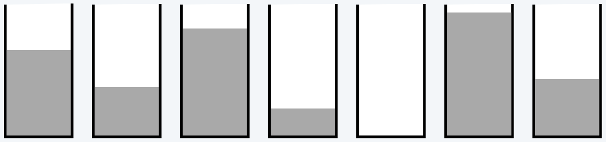
- APIs
- Clients
- **Strawman implementation**
- Linked lists
- Implementations

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.

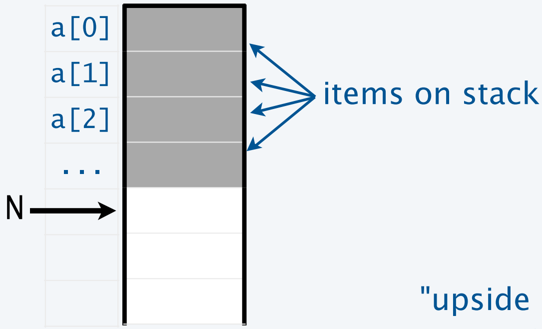
instance variables

constructor

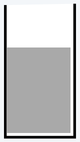
methods

test client

```
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();
}
```

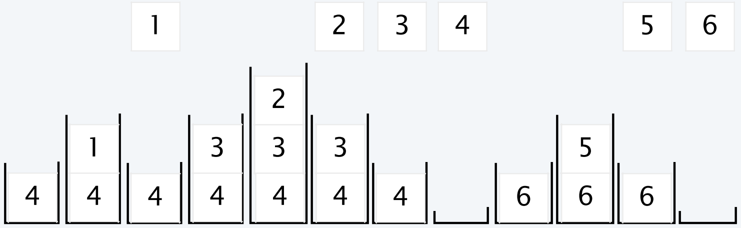
```
% more tobe.txt
to be or not to - be - - that - - - is

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

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

Pop quiz 1 on stacks

Q. Can we always insert pop() commands to make items come out in sorted order?

Example 1.	6	5	4	3	2	1	–	–	–	–	–	–
Example 2.	1	–	2	–	3	–	4	–	5	–	6	–
Example 3.	4	1	–	3	2	–	–	–	6	5	–	–

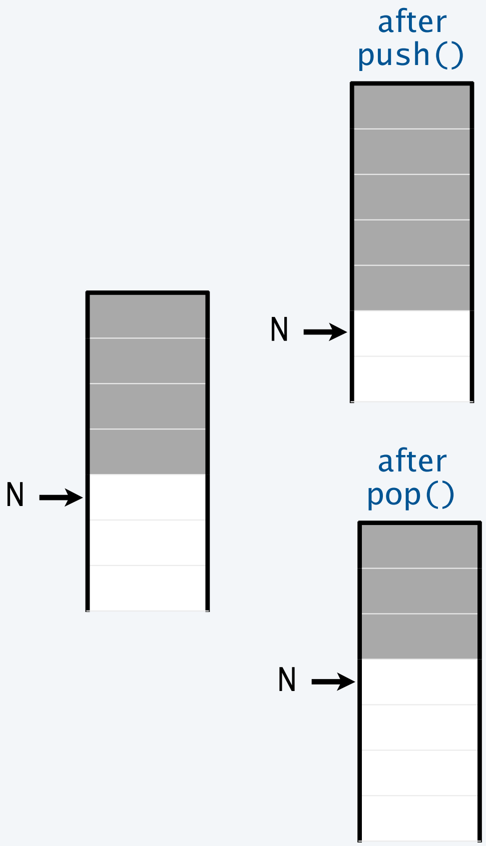


Strawman implementation: Methods

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

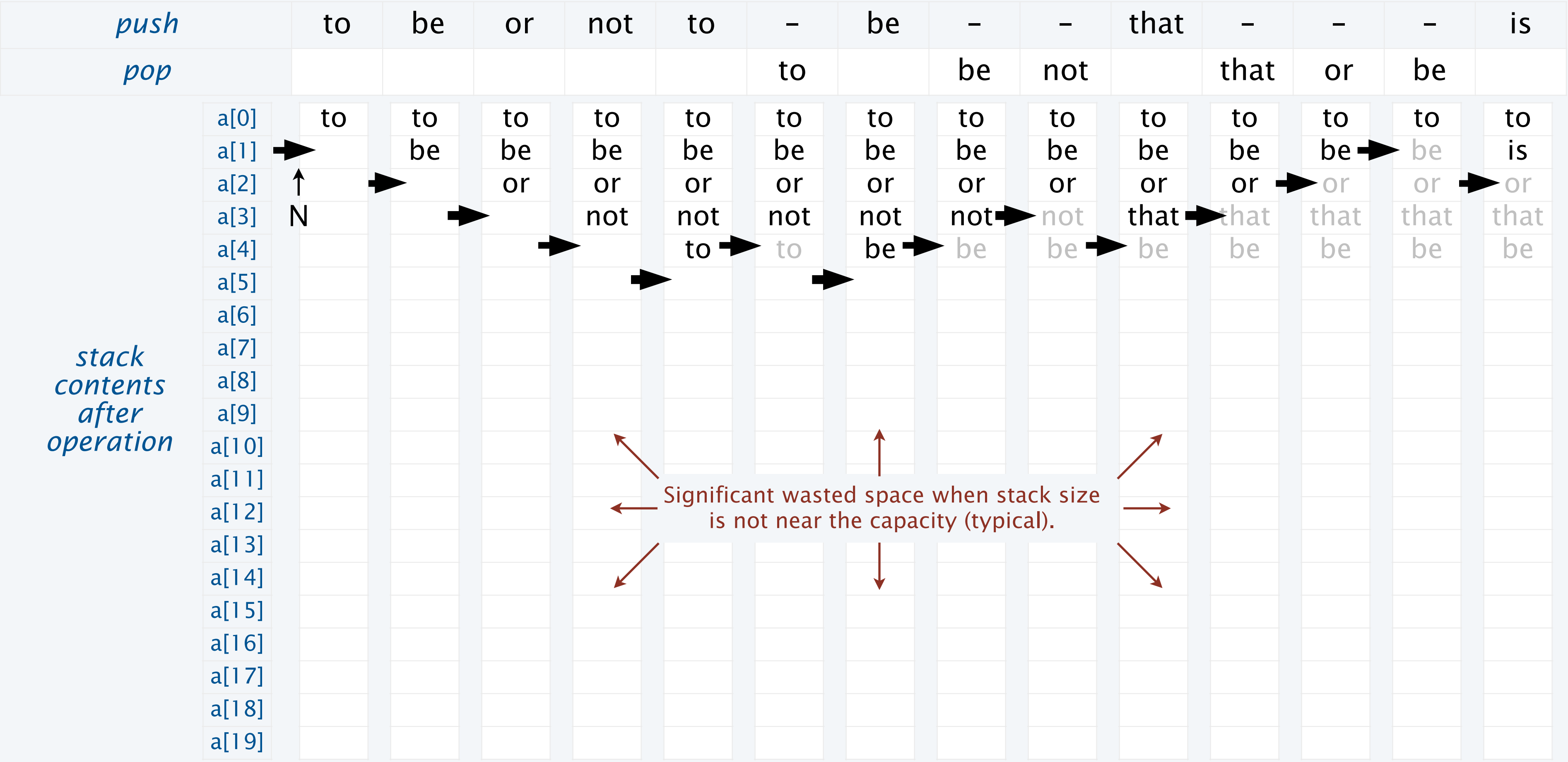
constructor

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]	N			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]														

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.*

COMPUTER SCIENCE

SEDGEWICK / WAYNE

PART I: PROGRAMMING IN JAVA

CS.12.C.StacksQueues.Strawman

12. Stacks and Queues

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

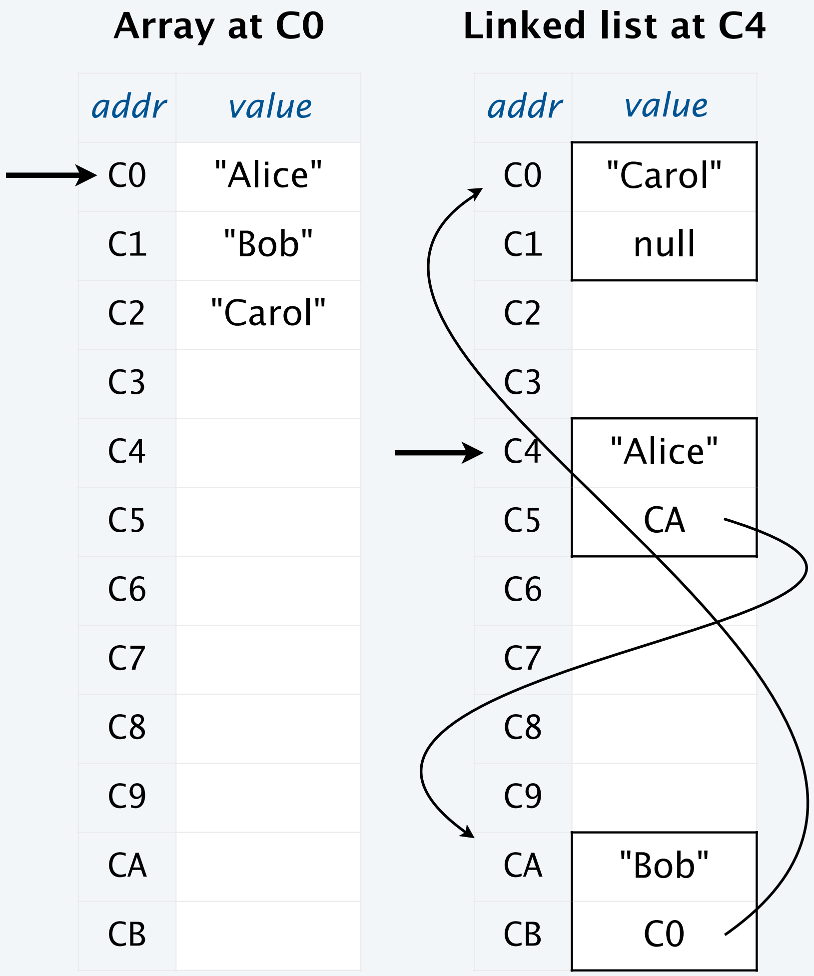
Data structures: sequential vs. linked

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

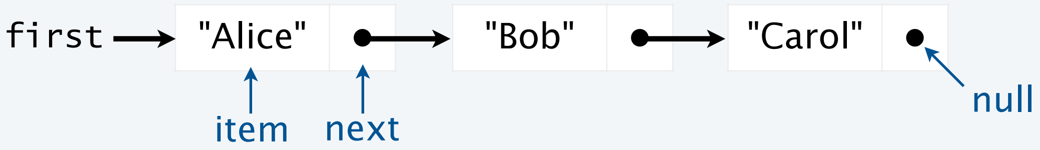
- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

```
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

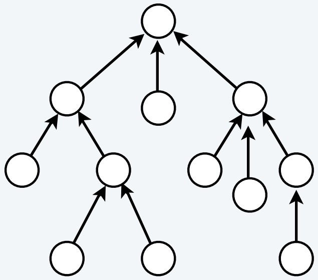


Singly-linked data structures

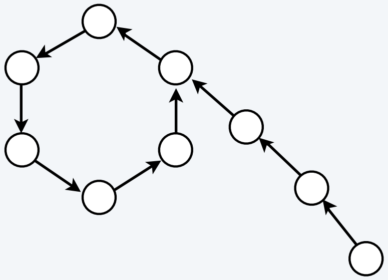
Even with just one link (○⟶) a wide variety of data structures are possible.

Linked list (this lecture)

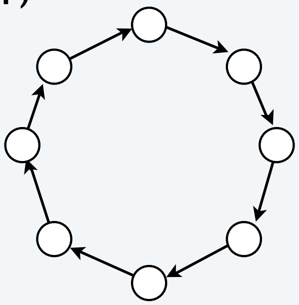
Tree



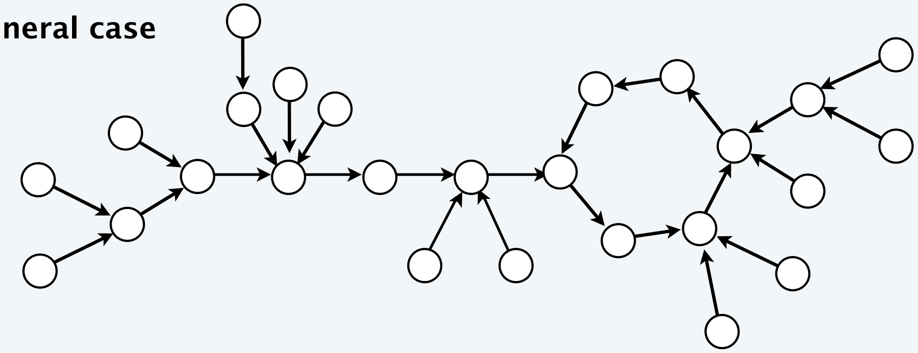
Rho



Circular list (TSP)



General case

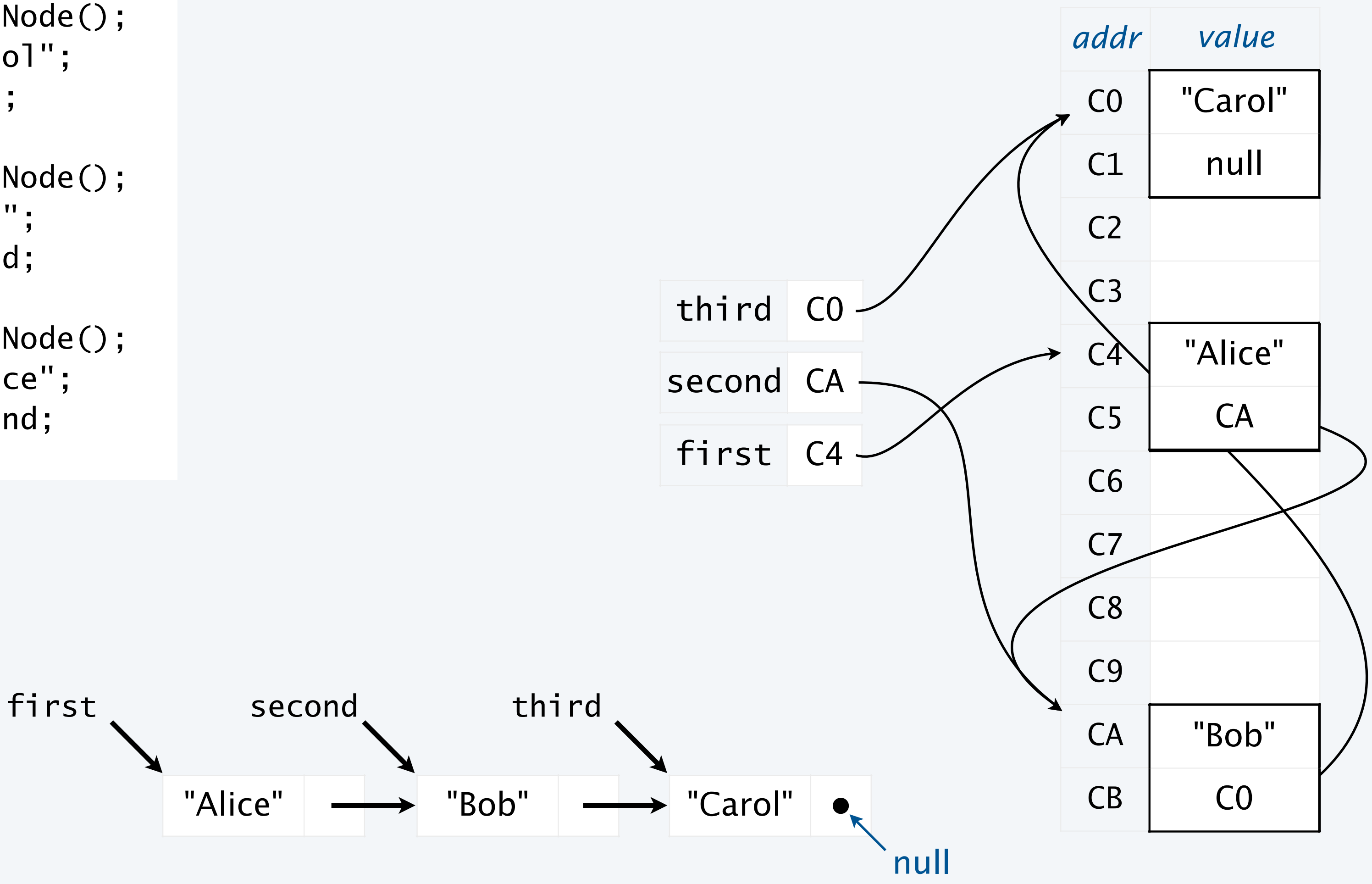


Multiply linked structures: many more possibilities!

From the point of view of a particular object, all of these structures look the same.

Building a linked list

```
Node third  = new Node();
third.item  = "Carol";
third.next  = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first  = new Node();
first.item  = "Alice";
first.next  = second;
```



List processing code

Standard operations for processing data structured as a singly-linked list

- Add a node at the beginning.
- Remove and return the node at the beginning.
- Add a node at the end (requires a reference to the last node).
- Traverse the list (visit every node, in sequence).

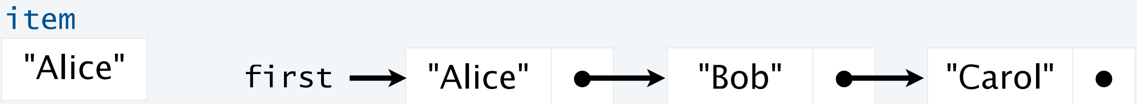
An operation that calls for a *doubly*-linked list (slightly beyond our scope)

- Remove and return the node at the end.

List processing code: Remove and return the first item

Goal. Remove and return the first item in a linked list `first`.

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



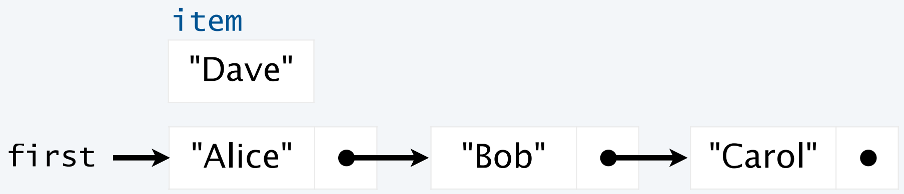
```
first = first.next;
```

available for garbage collection

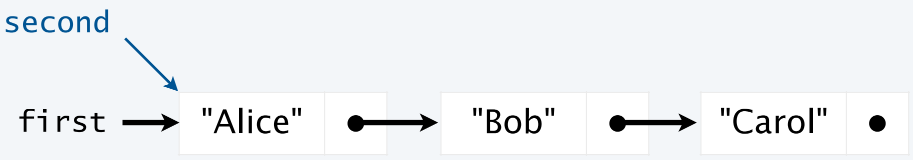
```
return item;
```

List processing code: Add a new node at the beginning

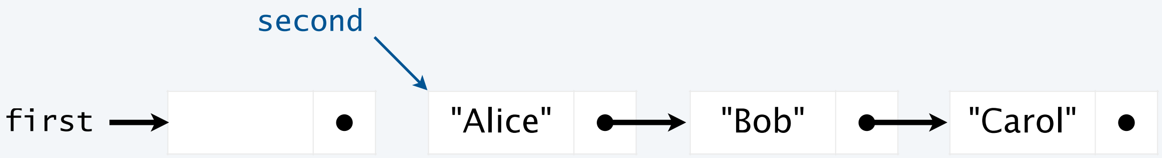
Goal. Add `item` to a linked list `first`.



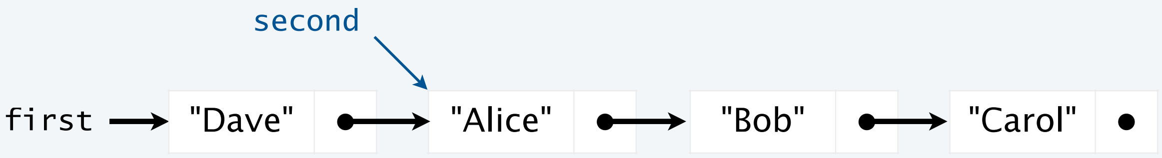
```
Node second = first;
```



```
first = new Node();
```



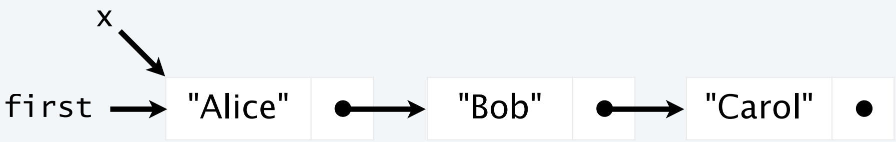
```
first.item = item;
first.next = second;
```



List processing code: Traverse a list

Goal. Visit every node on a linked list `first`.

```
Node x = first;
while (x != null)
{
   StdOut.println(x.item);
   x = x.next;
}
```



StdOut

```
Alice
Bob
Carol
```

Pop quiz 1 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
   Node old = list;
   list = new Node();
   list.item = StdIn.readString();
   list.next = old;
}
for (Node t = list; t != null; t = t.next)
   StdOut.println(t.item);
...
```

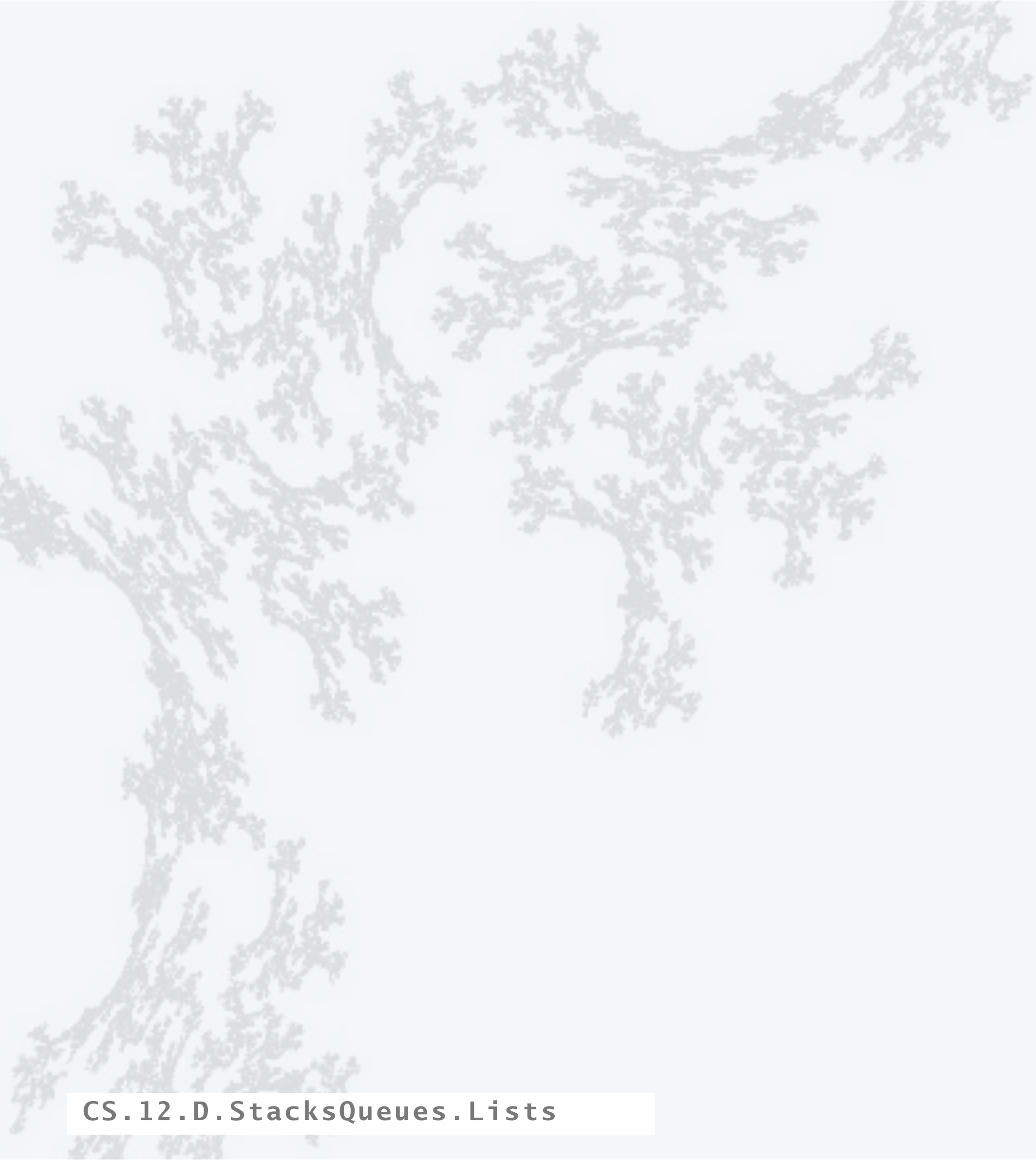
Pop quiz 2 on stacks

Q. Give code that uses a stack to print the strings from `StdIn` on `StdOut`, in reverse order.

Pop quiz 2 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
   last.next = new Node();
   last = last.next;
   last.item = StdIn.readString();
}
...
```



COMPUTER SCIENCE

SEDGEWICK/WAYNE

PART I: PROGRAMMING IN JAVA

CS.12.D.StacksQueues.Lists

COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES

14. Stacks and Queues

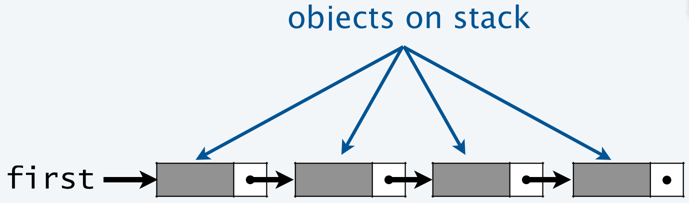
- APIs
- Clients
- Strawman implementation
- Linked lists
- **Implementations**

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

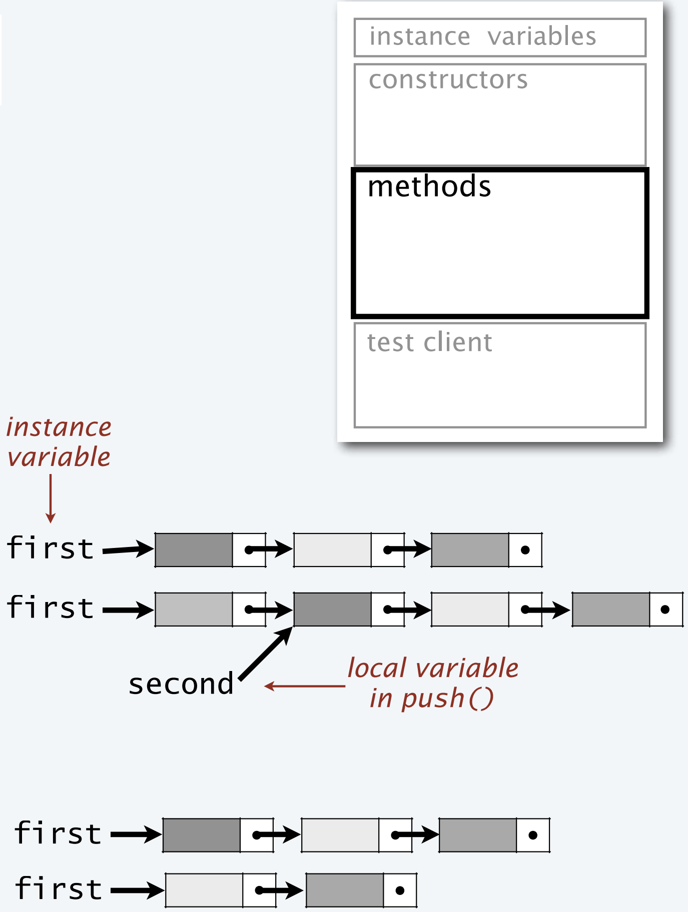
Methods define data-type operations (implement the API).

```
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;  }
...
}
```

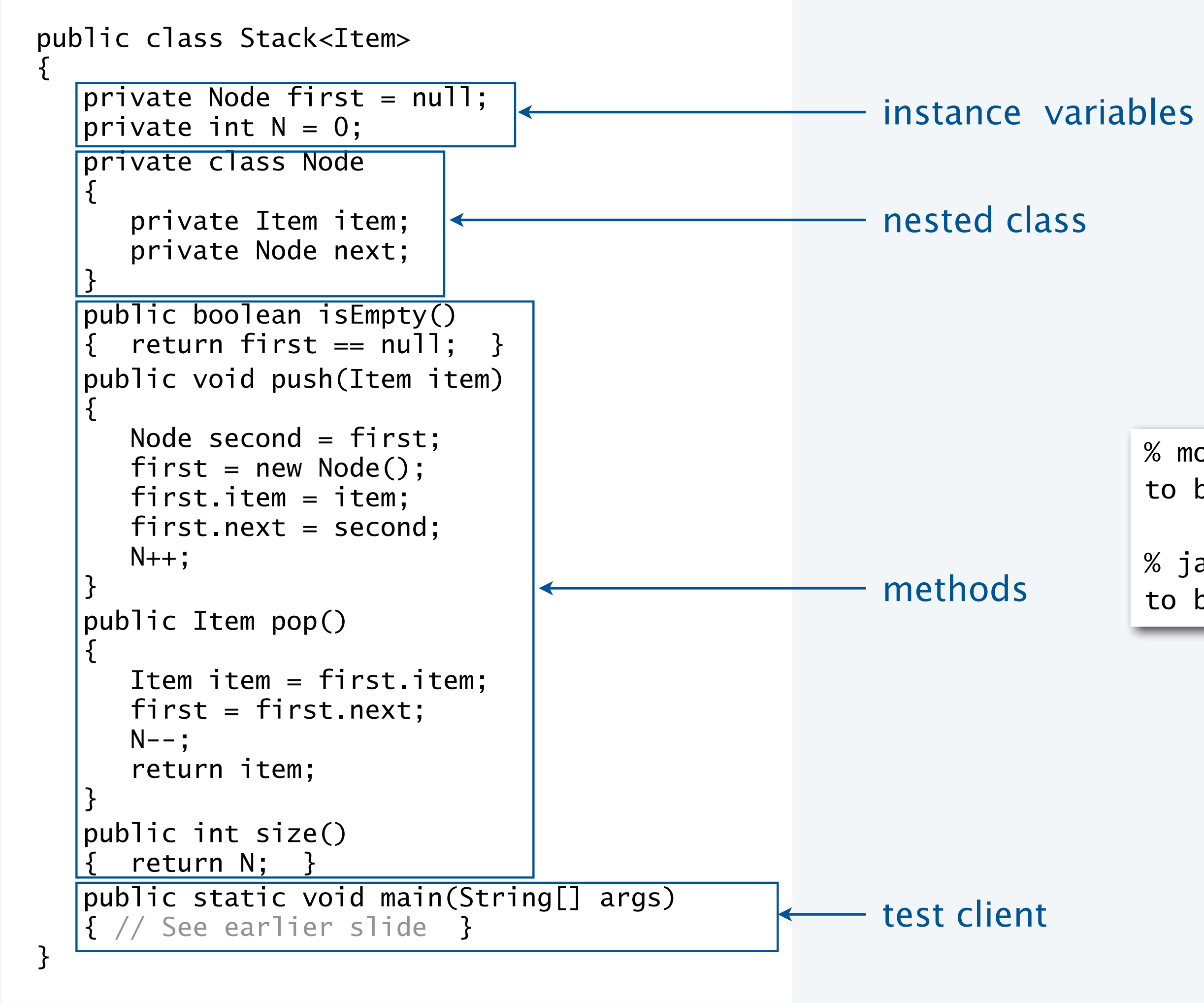
might also use N == 0

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

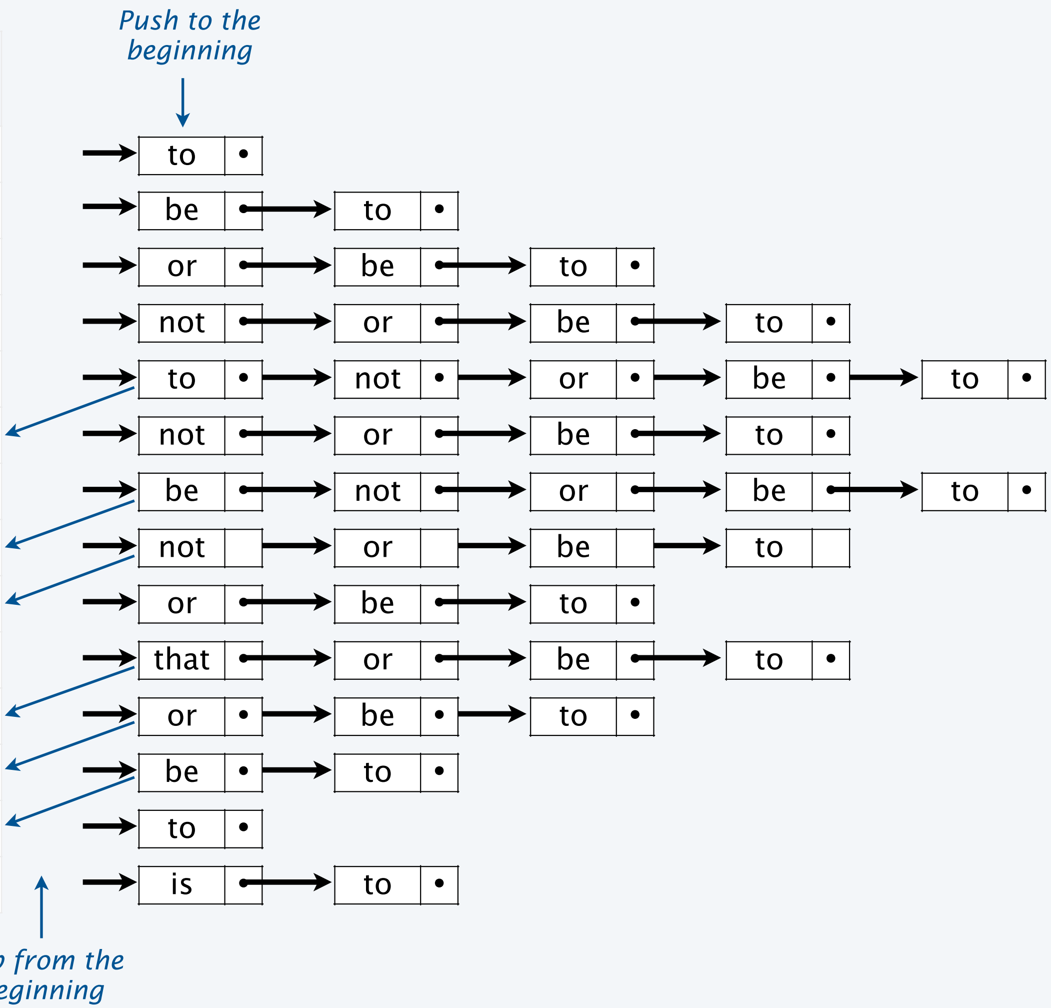
methods

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	



Push to the beginning

Pop from the beginning

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.*

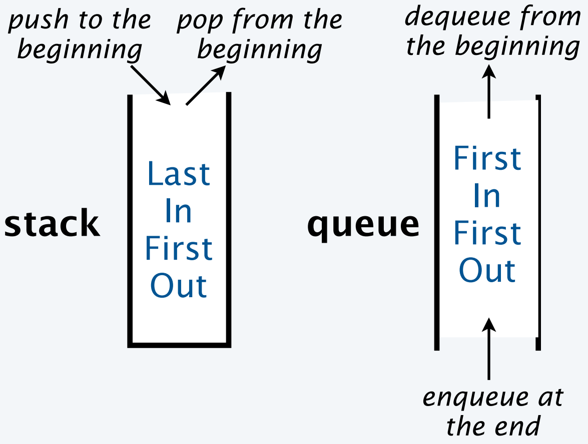
dequeue(): same code as pop()
enqueue(): slightly more complicated

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

Summary

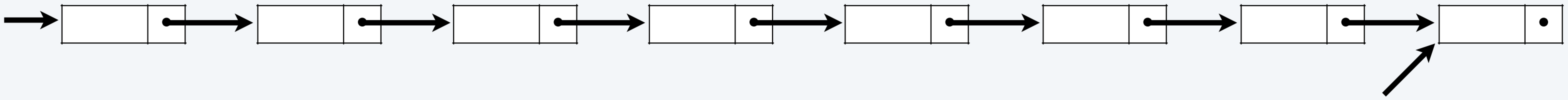
Stacks and queues

- 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*

COMPUTER SCIENCE

SEDGEWICK/WAYNE

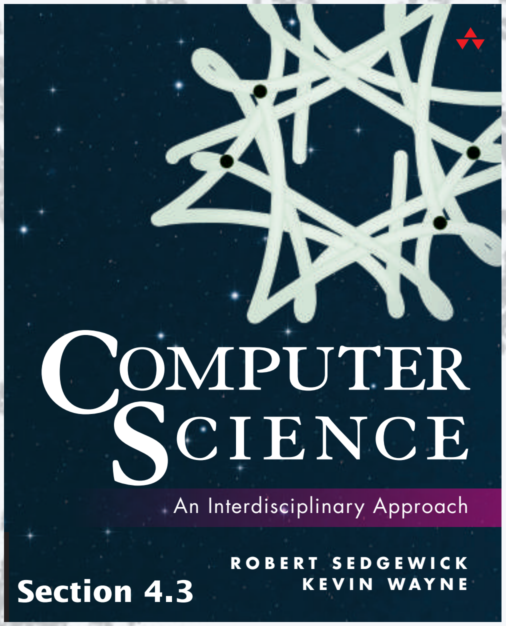
PART I: PROGRAMMING IN JAVA

CS.12.E.StacksQueues.Implementations

COMPUTER SCIENCE
SEDGEWICK / WAYNE

PART II: ALGORITHMS, THEORY, AND MACHINES

12. Stacks and Queues



http://introcs.cs.princeton.edu