



ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

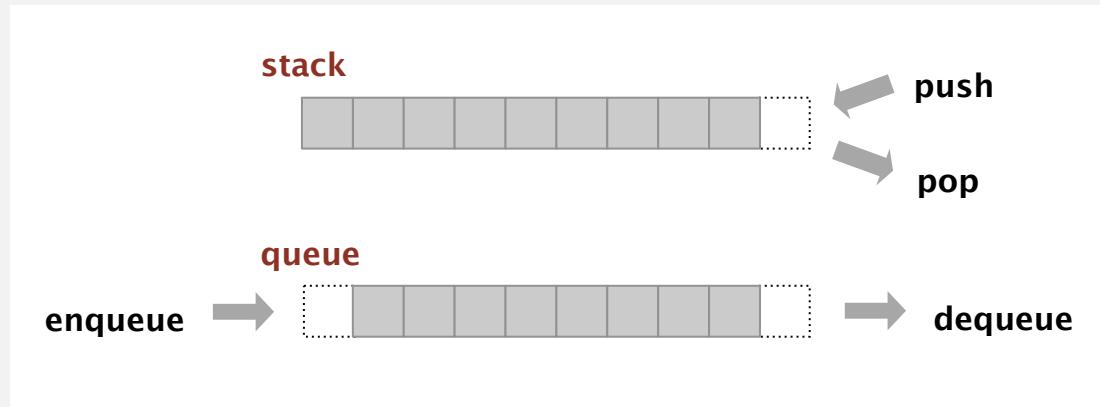
- ***stacks***
- ***resizing arrays***
- ***queues***
- ***generics***
- ***iterators***
- ***applications***

# Stacks and queues

---

## Fundamental data types.

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



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

**Queue.** Examine the item least recently added. ← FIFO = "first in first out"

# Client, implementation, interface

---

Separate interface and implementation.

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

Benefits.

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

**Client:** program using operations defined in interface.

**Implementation:** actual code implementing operations.

**Interface:** description of data type, basic operations.



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<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()	<i>create an empty stack</i>
------------------	------------------------------

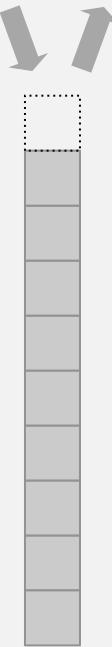
void push(String item)	<i>insert a new string onto stack</i>
------------------------	---------------------------------------

String pop()	<i>remove and return the string most recently added</i>
--------------	---

boolean isEmpty()	<i>is the stack empty?</i>
-------------------	----------------------------

int size()	<i>number of strings on the stack</i>
------------	---------------------------------------

push    pop



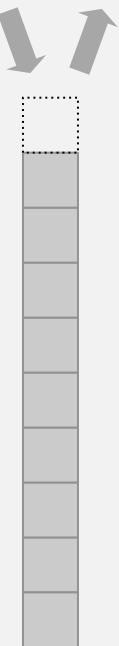
Warmup client. Reverse sequence of strings from standard input.

## Sample client

Warmup client. Reverse sequence of strings from standard input.

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

push    pop



```
public class ReverseStrings
{
    public static void main(String[] args)
    {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readString());
        while (!stack.isEmpty())
            StdOut.print(stack.pop());
    }
} % more tinyTale.txt
it was the best of times ...
```

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

## Stack test client

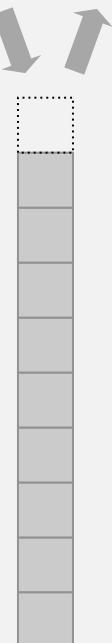
---

Read strings from standard input.

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

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

push    pop



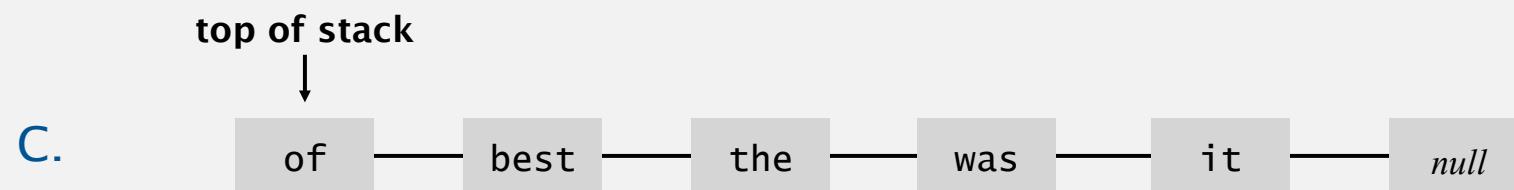
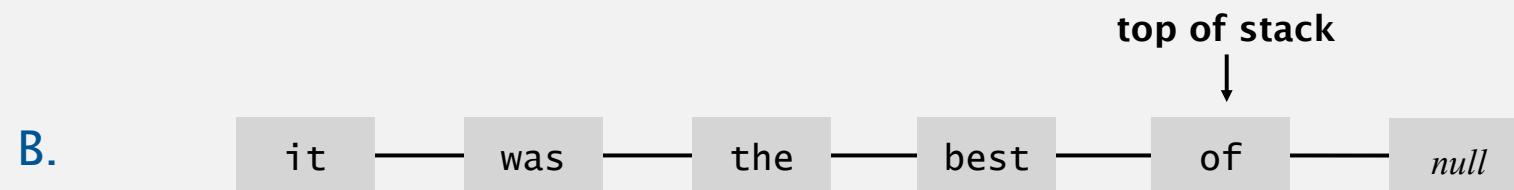
```
% more tobe.txt
to be or not to - be - -
```

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

# How to implement a stack with a linked list?

---

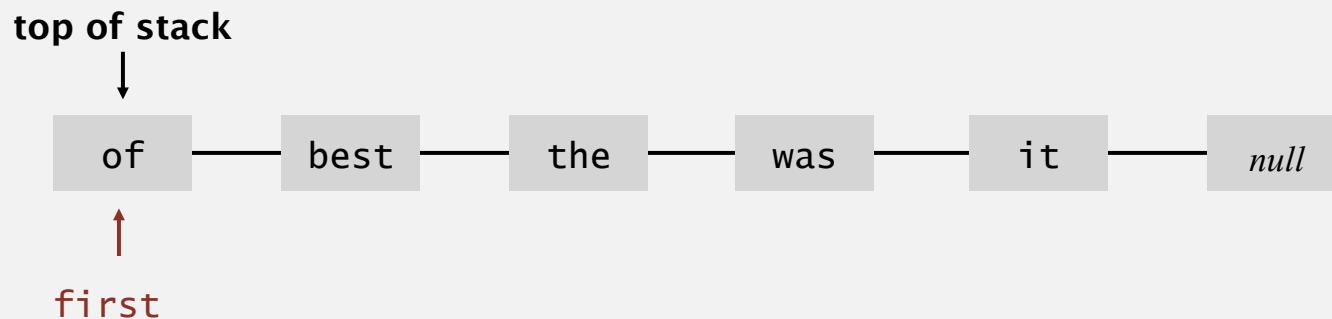
- A. Can't be done efficiently with a singly-linked list.



## Stack: linked-list implementation

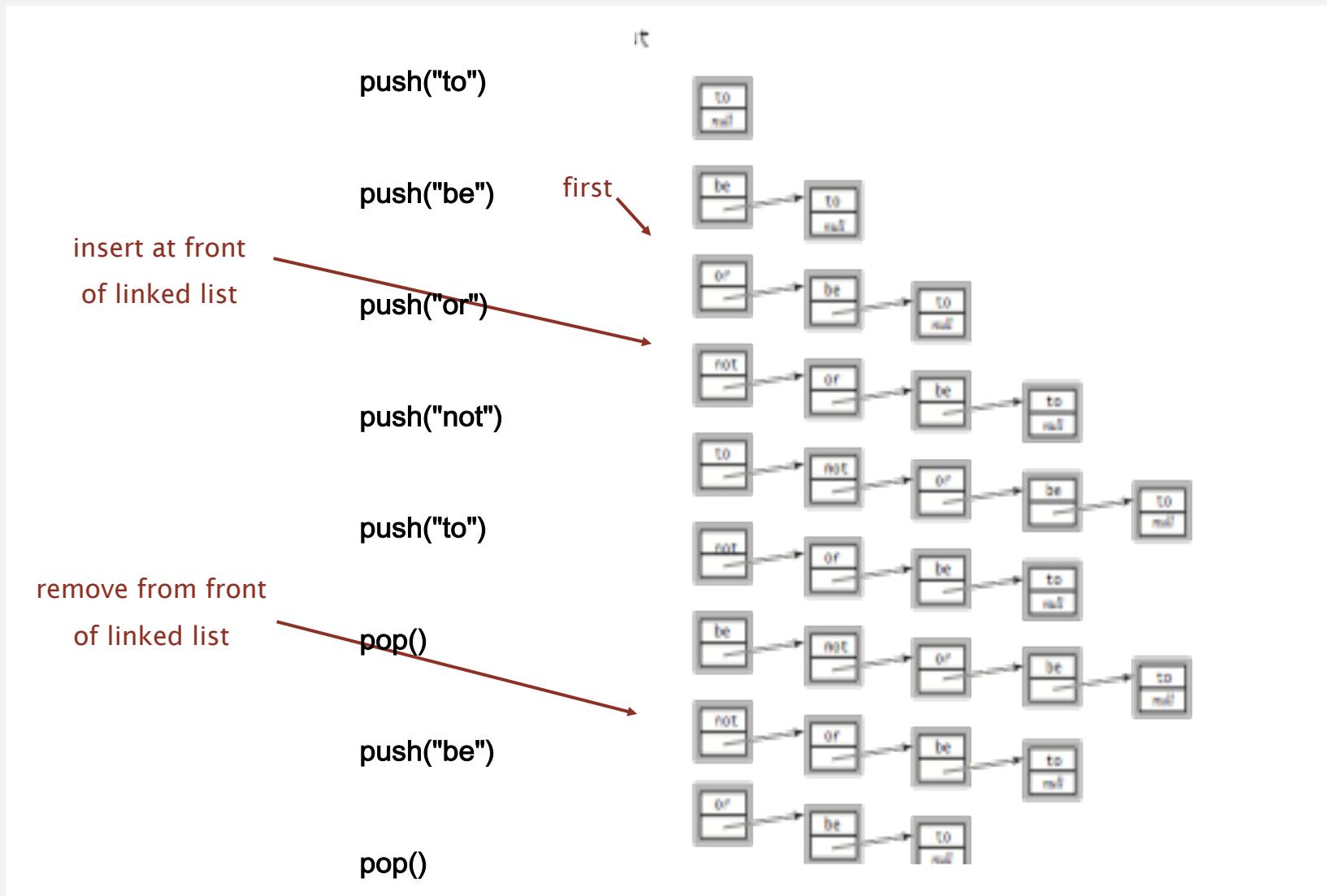
---

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.



# Stack: linked-list representation

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



# Stack pop: linked-list implementation

```
save item to return  
String item = first.item;  
  
delete first node  
  
inner class  
private class Node  
{  
    String item;  
    Node next;  
}  
  
first = first.next;  
  
return saved item  
return item;
```



# Stack push: linked-list implementation

inner class

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

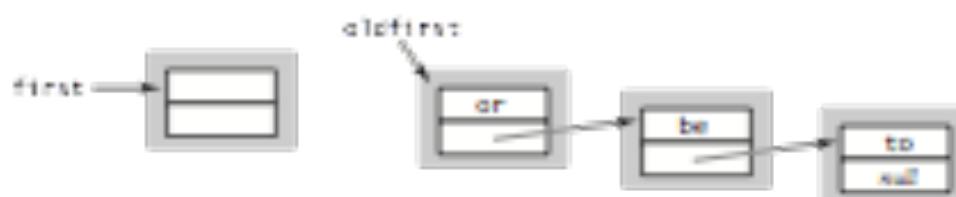
save a link to the list

```
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;
```



Inserting a new node at the beginning of a linked list

# Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

    public void push(String item)
    {
        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  $\sim 40N$  bytes.

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



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

---

40 bytes per stack node

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

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

---

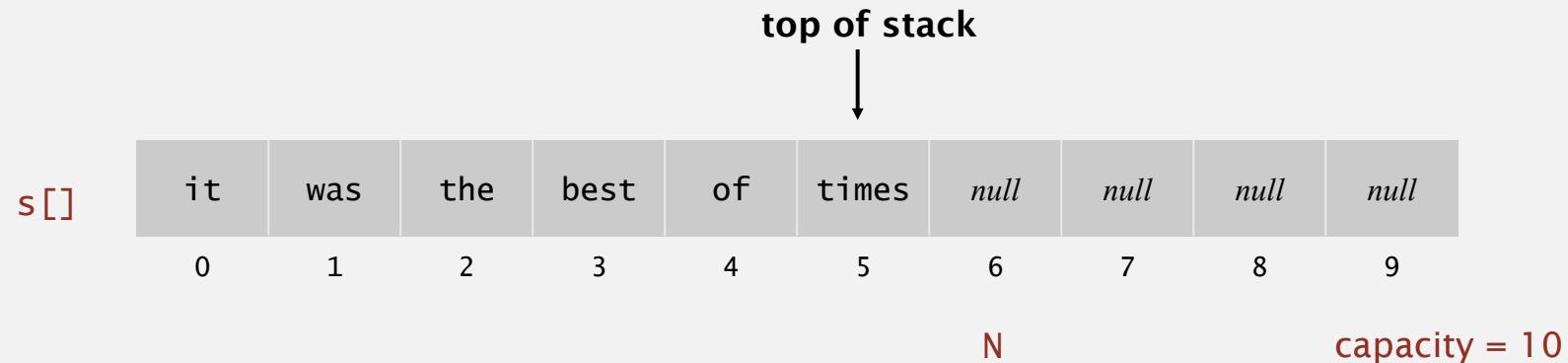
- A. Can't be done efficiently with an array.



## Fixed-capacity stack: array implementation

---

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



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

# 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];   }
}
```

use to index into array  
then increment N

a cheat  
(stay tuned)

decrement N;  
then use to index into array

# Stack considerations

---

## Overflow and underflow.

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

**Null items.** We allow null items to be inserted.

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

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

loitering

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

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## **1.3 BAGS, QUEUES, AND STACKS**

---

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## **1.3 BAGS, QUEUES, AND STACKS**

---

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

## Stack: resizing-array implementation

---

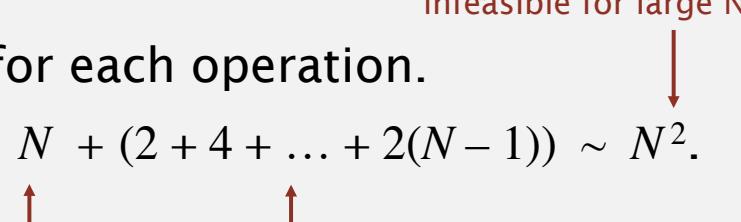
Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

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

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first  $N$  items =  $N + (2 + 4 + \dots + 2(N-1)) \sim N^2$ .  


$\uparrow$                      $\uparrow$   
1 array access      2( $k-1$ ) array accesses to expand to size  $k$   
per push                (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

## Stack: resizing-array implementation

Q. How to grow array?

A. If array is full, create a new array of **twice** the size, and copy items.

"repeated doubling"

```
public ResizingArrayStackOfStrings()
{   s = new String[1]; }

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

Array accesses to insert first  $N = 2^i$  items.  $N + (2 + 4 + 8 + \dots + N) \sim 3N$ .



1 array access  
per push

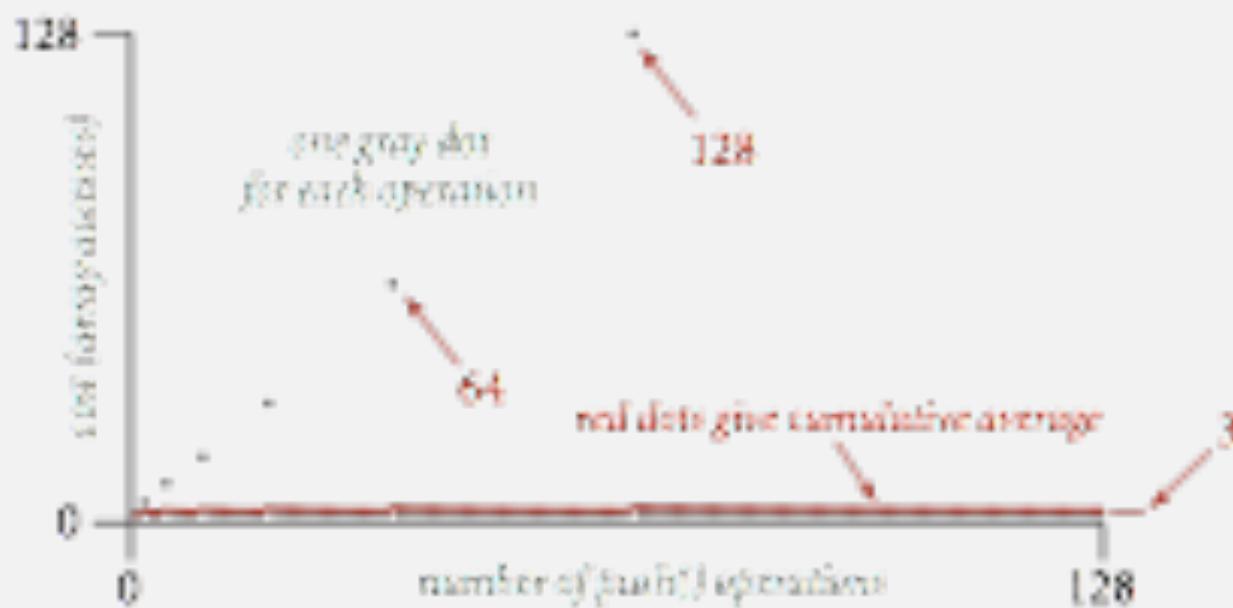


k array accesses to double to size k  
(ignoring cost to create new array)

## Stack: amortized cost of adding to a stack

Cost of inserting first  $N$  items.  $N + (2 + 4 + 8 + \dots + N) \sim 3N.$

$\uparrow$                              $\uparrow$   
1 array access                k array accesses to double to size k  
per push                      (ignoring cost to create new array)



# Stack: resizing-array implementation

---

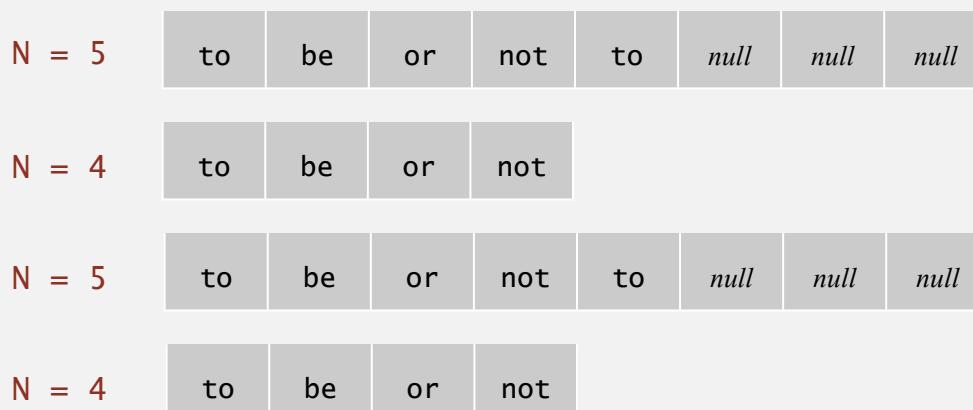
Q. How to shrink array?

First try.

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

Too expensive in worst case.

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



## Stack: resizing-array implementation

---

Q. How to shrink array?

Efficient solution.

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

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

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

## Stack: resizing-array implementation trace

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

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

## Stack resizing-array implementation: performance

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

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

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

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

*doubling and halving operations*

The diagram illustrates the amortized cost of push and pop operations. It shows two red arrows originating from the 'worst' column for 'push' and 'pop', both pointing to the 'amortized' column value of 1. A text label 'doubling and halving operations' is positioned to the right of these arrows, explaining the nature of the amortization strategy used in a resizing array implementation.

## Stack resizing-array implementation: memory usage

---

Q. How much memory does a `ResizingArrayList` use to store  $N$  strings in the best case? Worst case?

Count only the memory owned by the stack (not the strings themselves).

## Stack resizing-array implementation: memory usage

---

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

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

```
public class ResizingArrayStackOfStrings
{
    private String[] s; ← 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

---

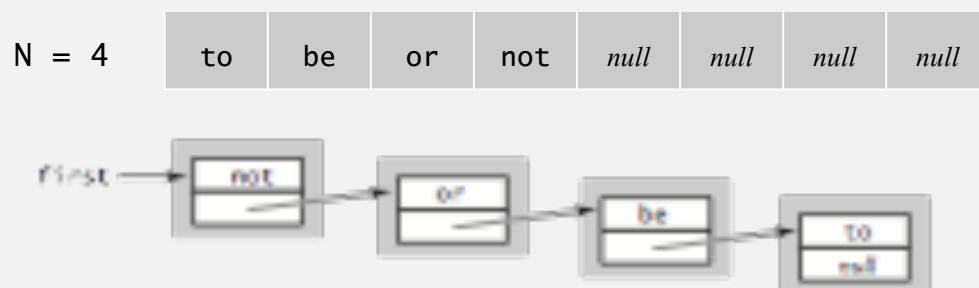
**Tradeoffs.** Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

### 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.



Inserting a new node at the beginning of a linked list



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## **1.3 BAGS, QUEUES, AND STACKS**

---

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<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)
```

*insert a new string onto queue*

```
    String dequeue()
```

*remove and return the string  
least recently added*

```
    boolean isEmpty()
```

*is the queue empty?*

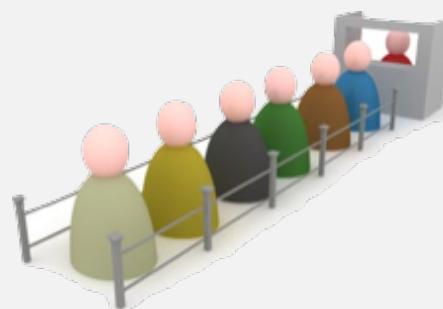
```
    int size()
```

*number of strings on the queue*

enqueue



dequeue



## Queue test client

---

Read strings from standard input.

- If string equals "-", dequeue string and print.
- Otherwise, enqueue string.

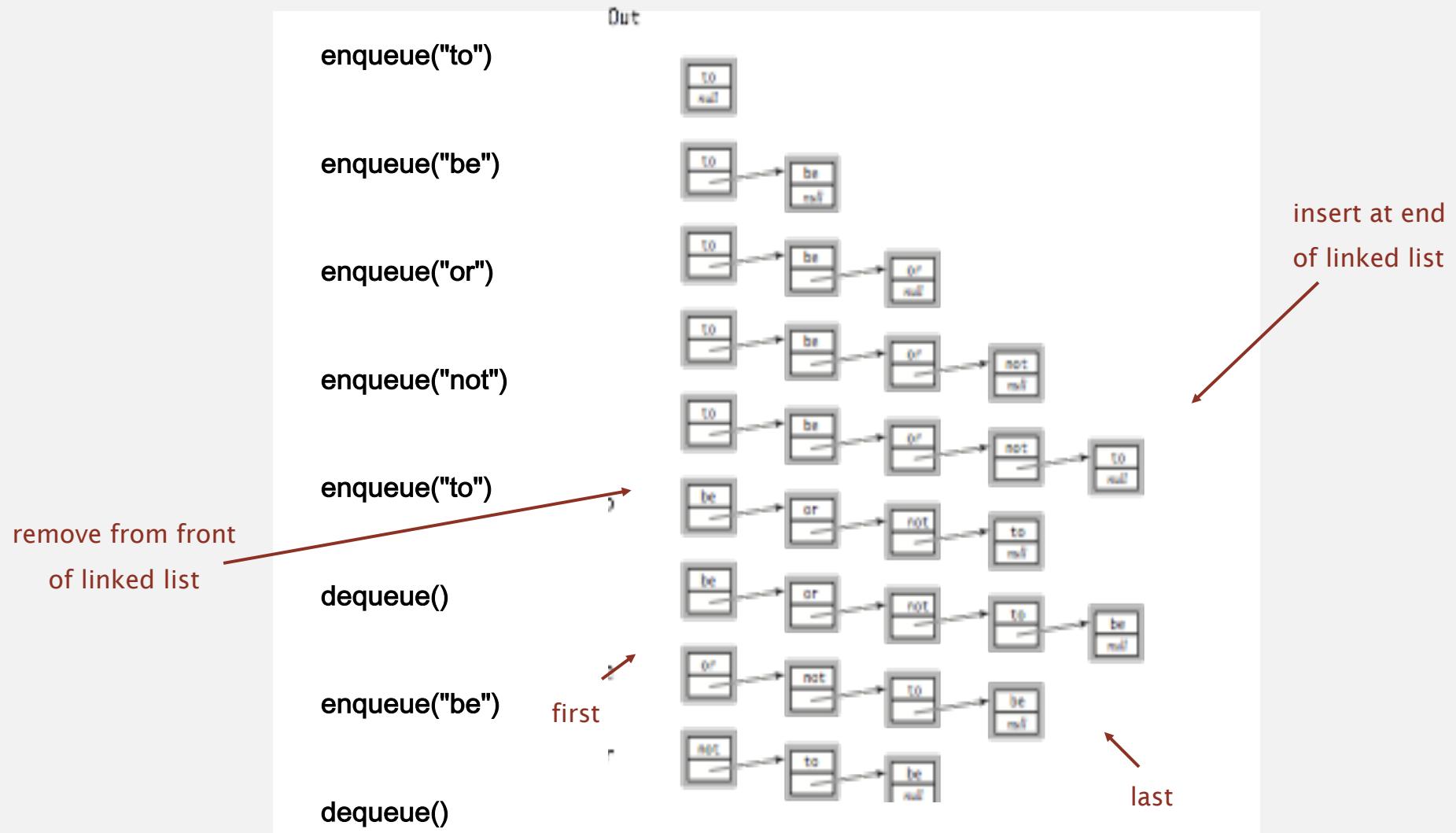
```
public static void main(String[] args)
{
    QueueOfStrings q = new QueueOfStrings();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("-")) StdOut.print(q.dequeue());
        else             q.enqueue(s);
    }
}
```

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

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

# Queue: linked-list representation

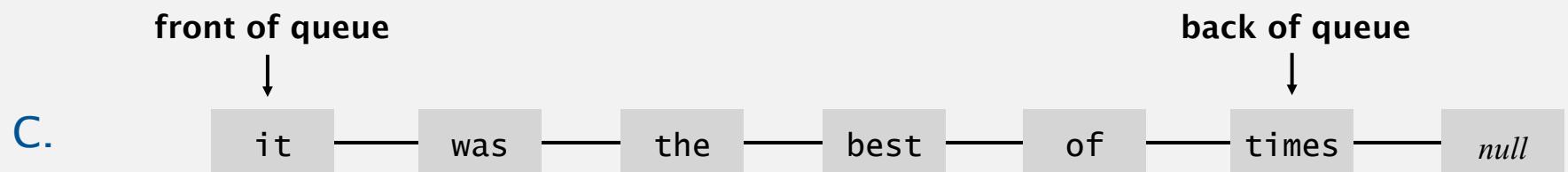
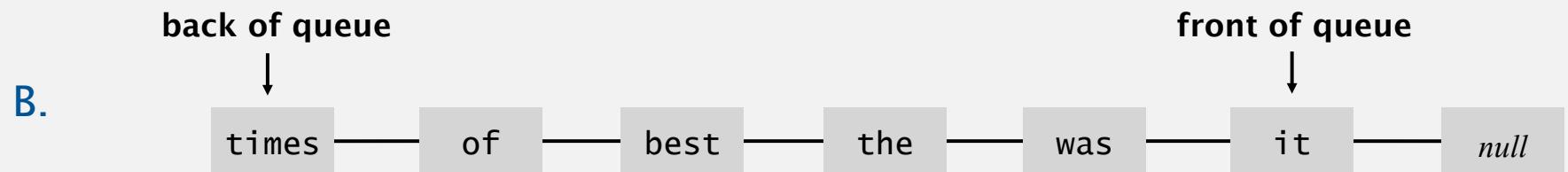
Maintain pointer to first and last nodes in a linked list;  
remove from front; insert at end.



# How to implement a queue with a linked list?

---

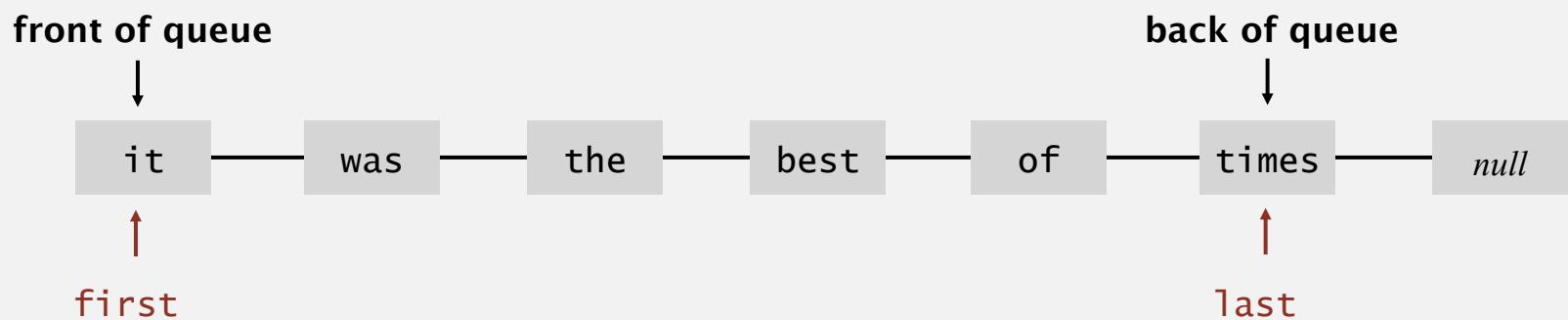
- A. Can't be done efficiently with a singly-linked list.



## 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`.

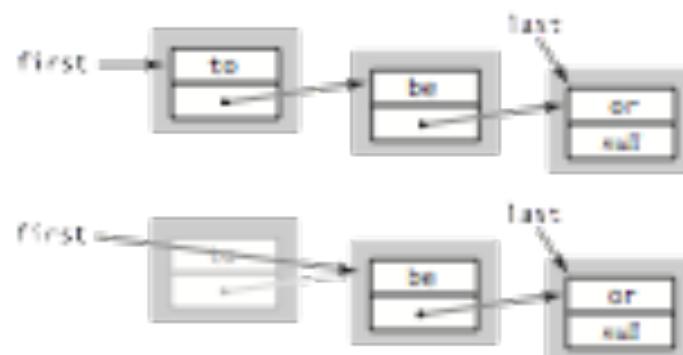


## Queue dequeue: linked-list implementation

```
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().

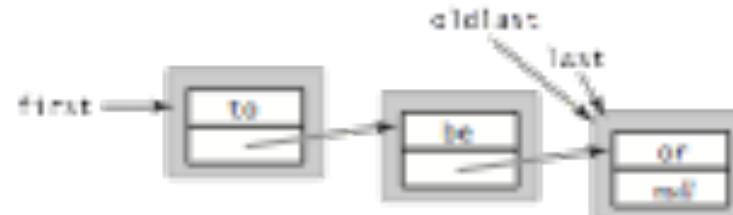
## Queue enqueue: linked-list implementation

### inner class

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

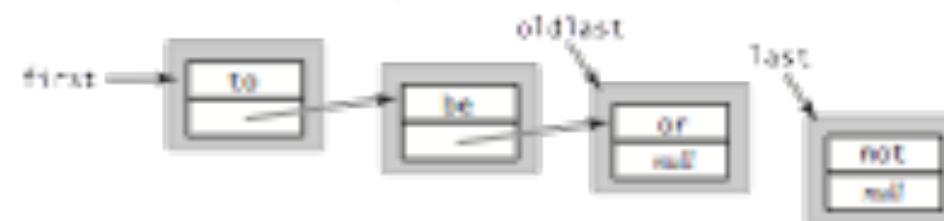
save a link to the last node

```
Node oldlast = last;
```



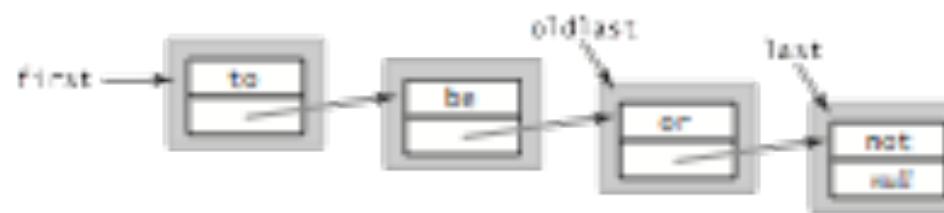
create a new node for the end

```
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;
```

```
}
```

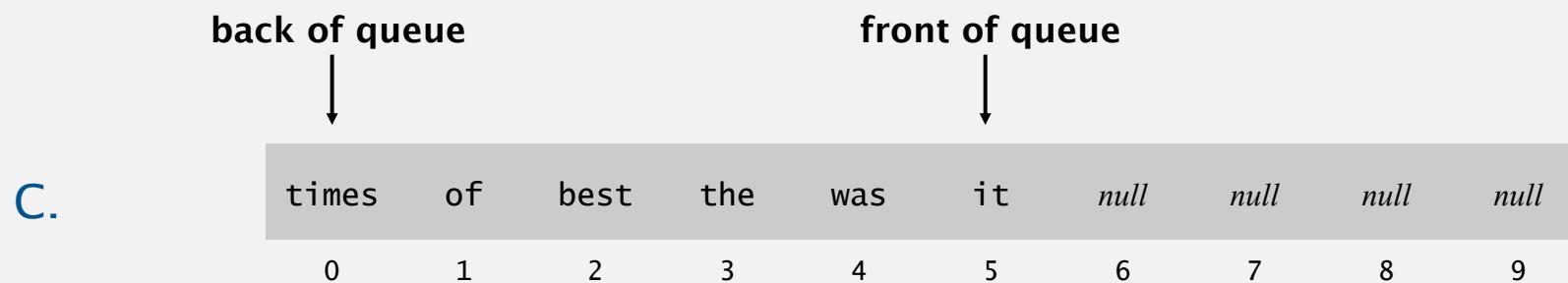
```
}
```

special cases for  
empty queue

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

---

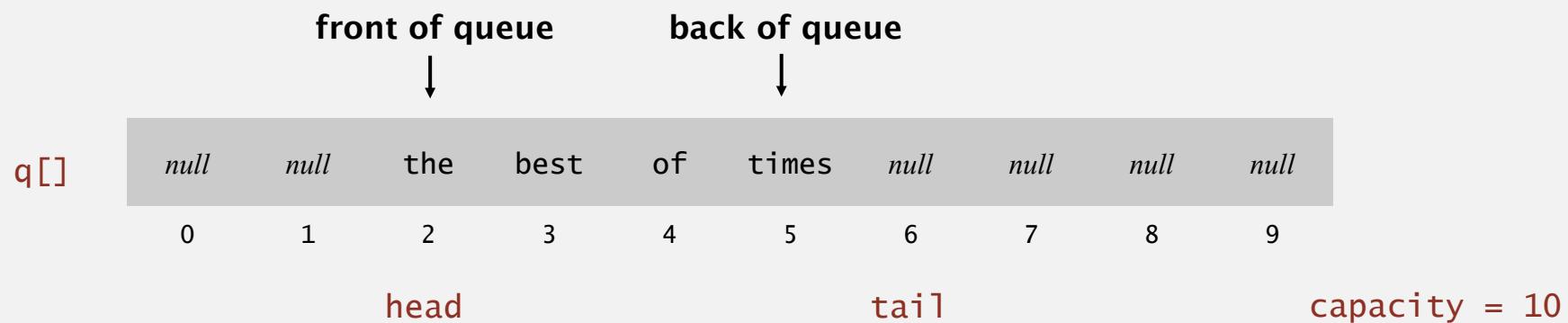
- A. Can't be done efficiently with an array.



## Queue: resizing-array implementation

---

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



Q. How to resize?



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<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, StackOfVans, ....

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

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

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



## Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

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

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

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

run-time error



## Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.

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

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

type parameter

compile-time error

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

# Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

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

    private class Node
    {
        Item item;
        Node next;
    }

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

    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

generic type name

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public . . . StackOfStrings(int capacity)
    {   s = new String[capacity];   }

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

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

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

the way it should be

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = new Item[capacity];   }

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

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

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

@#\$\*! generic array creation not allowed in Java

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public . . . StackOfStrings(int capacity)
    {   s = new String[capacity];   }

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

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

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

the way it is

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = (Item[]) new Object[capacity];   }

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

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

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

the ugly cast

## Unchecked cast

---

```
% javac FixedCapacityStack.java
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: Item[]
        a = (Item[]) new Object[capacity];
                           ^
1 warning
```

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.



# Arrays in Java

---

Arrays in Java are covariant. `String[]` is subclass of `Object[]`.

```
String[] a = new String[N];
Object[] b = a;
b[0] = new Integer("17");
a[0] = a[0].toUpperCase();
```

- ← legal because arrays are covariant
- ← compiles, but `ArrayStoreException` at runtime
- ← would break type-safety

Consequence. Need to check type of array entries at runtime.

Why are Java arrays covariant? To support "generic" methods on arrays, such as `Arrays.sort()`, before language had generics.

# How are generic implemented?

**Design requirement.** Maintain backward compatibility with old JVMs.

**Type erasure.** Generics are implemented entirely in the compiler.

No generic type information is available at runtime.

```
Stack<String> s = new Stack<String>();  
s.push("hello");  
String a = s.pop();
```

**programmer uses parameterized types;  
compiler checks for type safety**

```
Stack s = new Stack();  
s.push("hello");  
String a = (String) s.pop();
```

**JVM sees only "raw type"**

**Consequence.** No generic array creation. ← because JVM needs to check type  
of array entries at runtime

**Lesson.** Hard to evolve language while maintaining backward compatibility.

## Generic data types: autoboxing

---

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a **wrapper** object type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast between a primitive type and its wrapper.

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

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

# Autoboxing challenge

---

Q. What does the following program print?

```
public class Autoboxing {

    public static void cmp(Integer a, Integer b) {
        if      (a <  b) StdOut.printf("%d < %d\n", a, b);
        else if (a == b) StdOut.printf("%d == %d\n", a, b);
        else            StdOut.printf("%d > %d\n", a, b);
    }

    public static void main(String[] args) {
        cmp(new Integer(42), new Integer(42));
        cmp(43, 43);          % java Autoboxing
        cmp(142, 142);
    }
}
```

```
42 > 42
43 == 43
142 > 142
```

Best practice. Avoid using wrapper types whenever possible.

# Generics

Caveat. Java generics can be mystifying at times.

```
public class Collections
{
    ...
    public static<T> void copy(List<? super T> dest, List<? extends T> src)
    {
        for (int i = 0; i < src.size(); i++)
            dest.set(i, src.get(i));
    }
}
```

mixing generics with inheritance



This course. Restrict attention to "pure generics."

avoid mixing generics with inheritance



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<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.



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

# Iterators

---

Q. What is an **Iterable** ?

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

Q. What is an **Iterator** ?

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

Q. Why make data structures **Iterable** ?

A. Java supports elegant client code.

## java.lang.Iterable interface

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

## java.util.Iterator interface

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

## “foreach” statement (shorthand)

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

## equivalent code (longhand)

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

# Stack iterator: linked-list implementation

```
import java.util.Iterator;

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

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

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

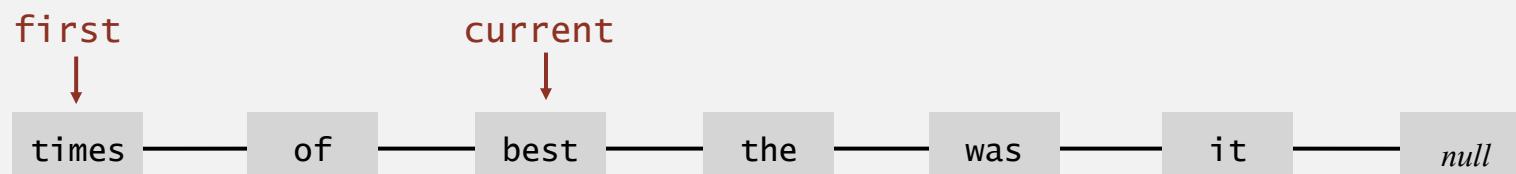
        public boolean hasNext() { return current != null; }

        public void remove() { /* not supported */ }

        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

Annotations for the ListIterator class:

- `remove()`: throw UnsupportedOperationException
- `next()`:
  - throw NoSuchElementException if no more items in iteration



## Stack iterator: array implementation

---

```
import java.util.Iterator;

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

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

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove()    { /* not supported */ }
        public Item next()      { return s[--i]; }
    }
}
```

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

## Iteration: concurrent modification

---

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

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

### concurrent modification

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

Q. How to detect?

A.

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

# 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)

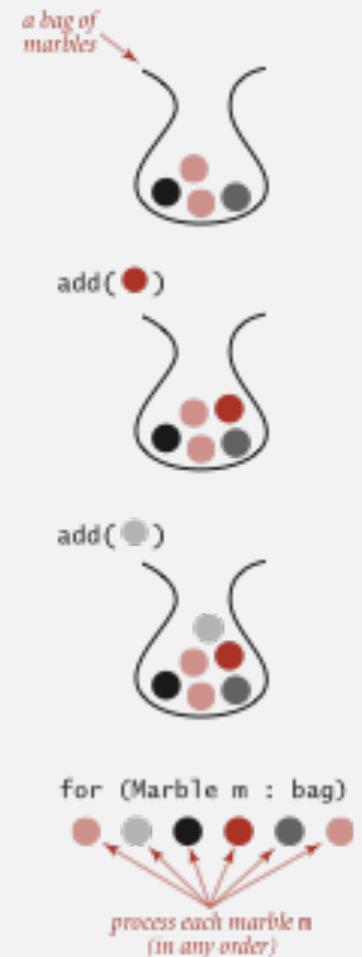
*insert a new item onto bag*

int size()

*number of items in bag*

Iterable<Item> iterator()

*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

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

# Java collections library

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

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

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

`java.util.LinkedList` uses linked list.

caveat: only some operations are efficient

# Java collections library

---

## `java.util.Stack`.

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

### **Java 1.3 bug report (June 27, 2001)**

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

### **status (closed, will not fix)**

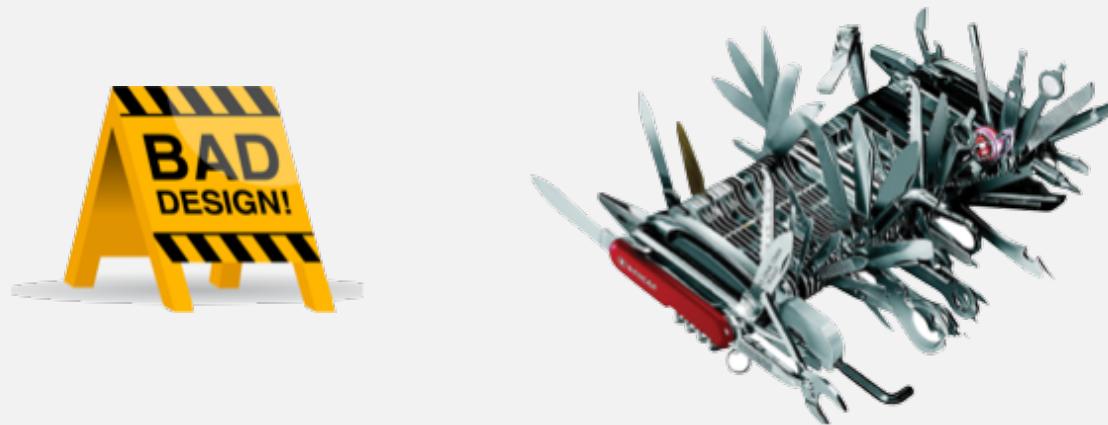
It was an incorrect design decision to have `Stack` extend `Vector` ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

## Java collections library

---

### `java.util.Stack`.

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



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

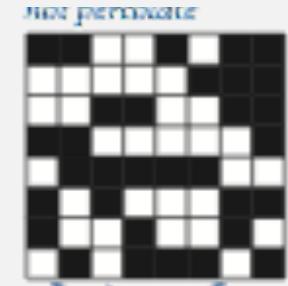
**Best practices.** Use our implementations of Stack, Queue, and Bag.

## War story (from Assignment 1)

---

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

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



Why is my program so slow?



Kenny

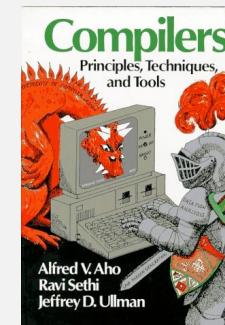
Lesson. Don't use a library until you understand its API!

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

# Stack applications

---

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



# Function calls

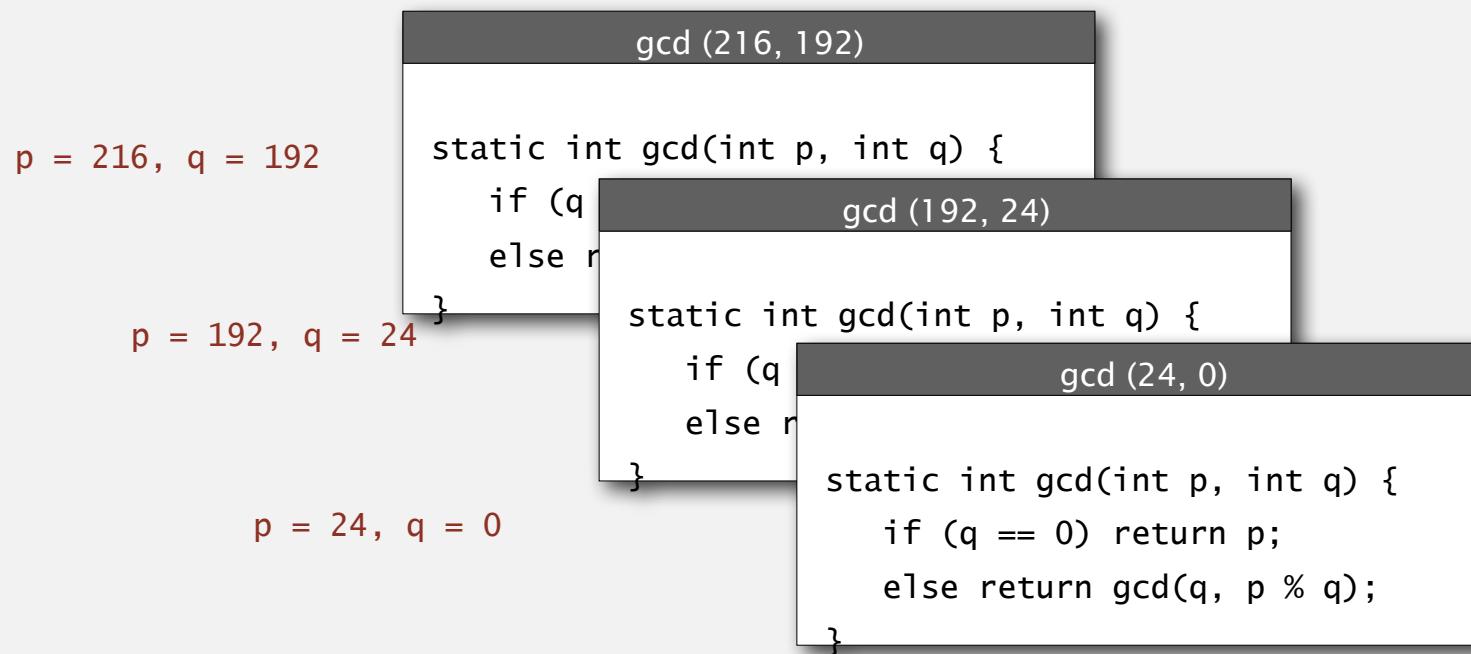
---

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.



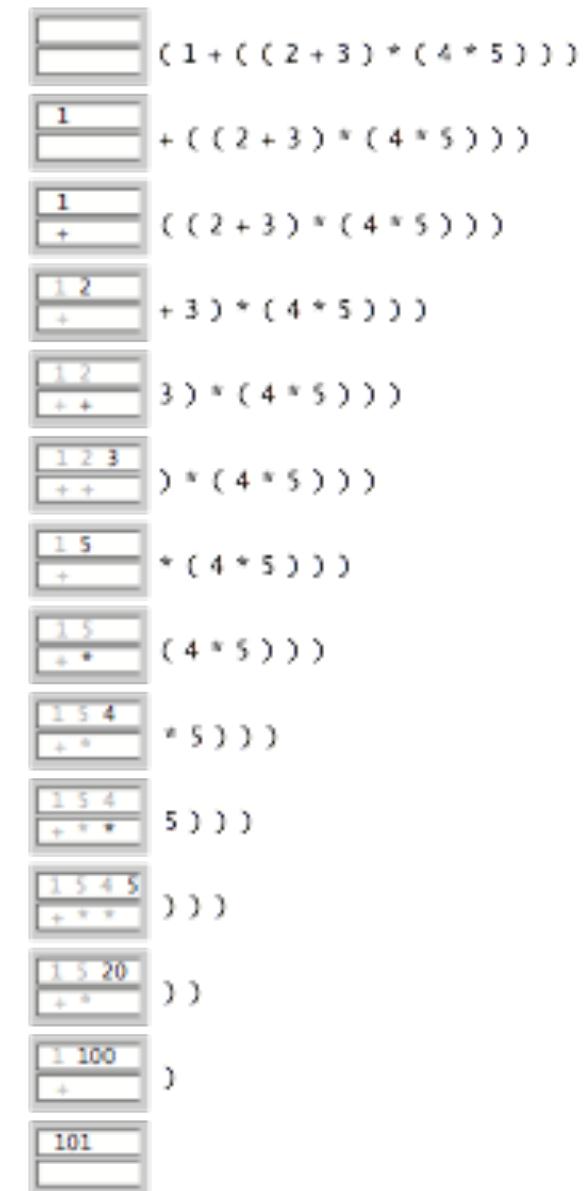
# Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

operand      operator

value stack  
operator stack

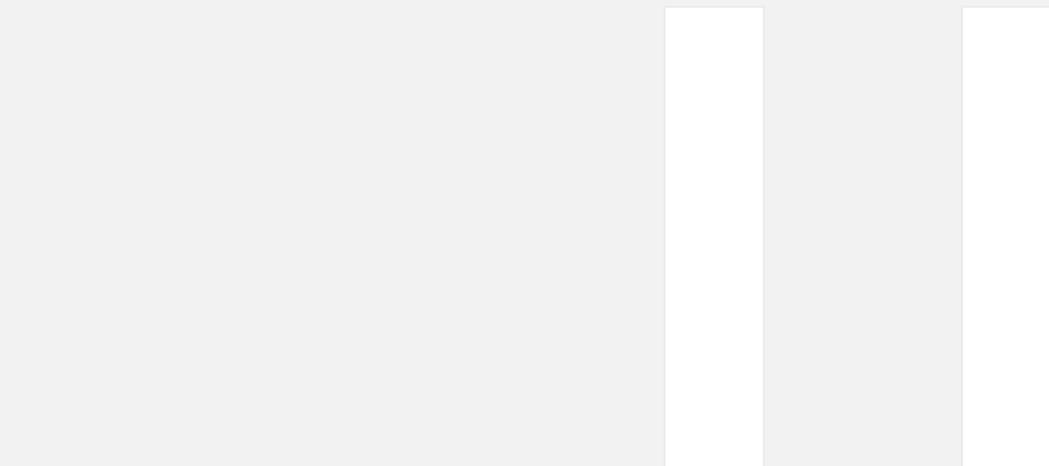


Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

# Dijkstra's two-stack algorithm demo

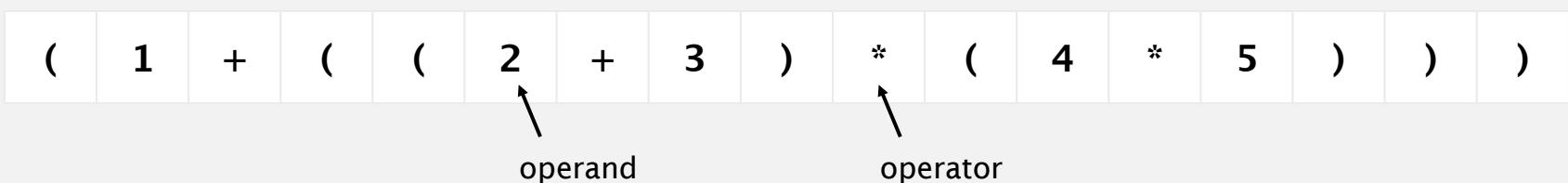


**infix expression**

(fully parenthesized)

value stack

operator stack



# Arithmetic expression evaluation

---

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

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

## Correctness

---

Q. Why correct?

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

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

as if the original input were:

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

Repeating the argument:

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

Extensions. More ops, precedence order, associativity.

# Stack-based programming languages

---

**Observation 1.** Dijkstra's two-stack algorithm computes the same value if the operator occurs **after** the two values.

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

**Observation 2.** All of the parentheses are redundant!

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



Jan Lukasiewicz

**Bottom line.** Postfix or "reverse Polish" notation.

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



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

---

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



ROBERT SEDGEWICK | KEVIN WAYNE

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

## 1.3 BAGS, QUEUES, AND STACKS

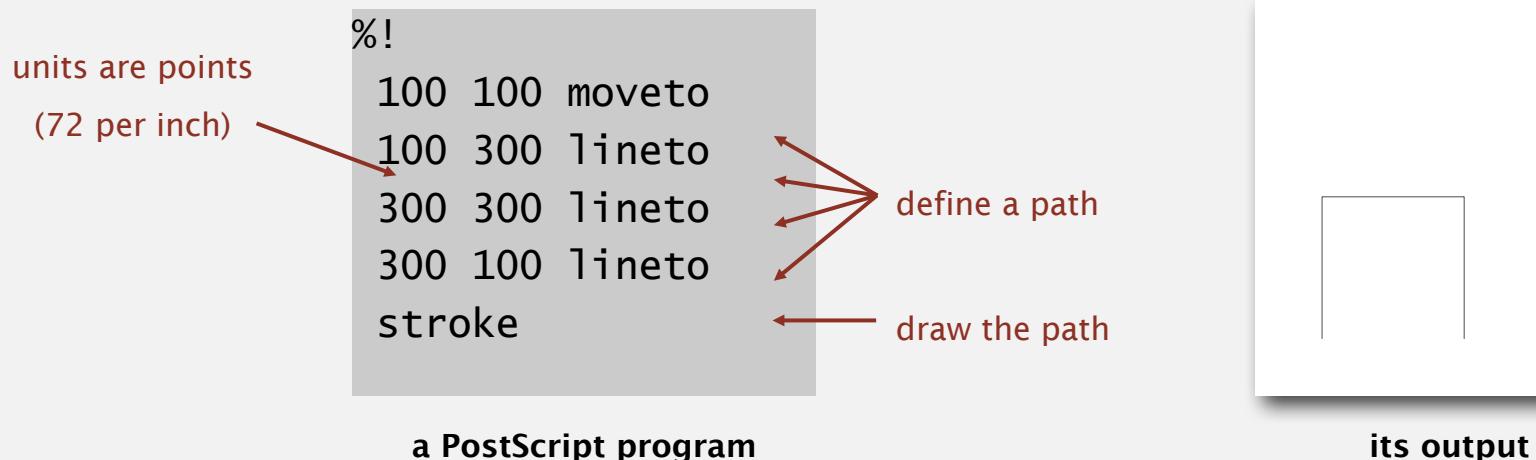
---

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

# PostScript

PostScript. [Warnock-Geschke 1980s]

- Postfix program code.
- Turtle graphics commands.
- Variables, types, text, loops, conditionals, functions, ...



Simple virtual machine, but not a toy.

- Easy to specify published page.
- Easy to implement in printers.
- Revolutionized the publishing world.



# PostScript

---

## Page description language.

- Explicit stack.
- Full computational model
- Graphics engine.

## Basics.

- %!: “I am a PostScript program.”
- Literal: “push me on the stack.”
- Function calls take arguments from stack.
- Turtle graphics built in.

a PostScript program

```
%!
72 72 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
-72 0 rlineto
2 setlinewidth
stroke
```

its output



# PostScript

---

## Data types.

- Basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ....
- Full set of built-in operators.

## Text and strings.

- Full font support.
- show (display a string, using current font).
- cvs (convert anything to a string).

System.out.print()

toString()

```
%!  
/Helvetica-Bold findfont 16 scalefont setfont  
72 168 moveto  
(Square root of 2:) show  
72 144 moveto  
2 sqrt 10 string cvs show
```

Square root of 2:  
1.41421

# PostScript

---

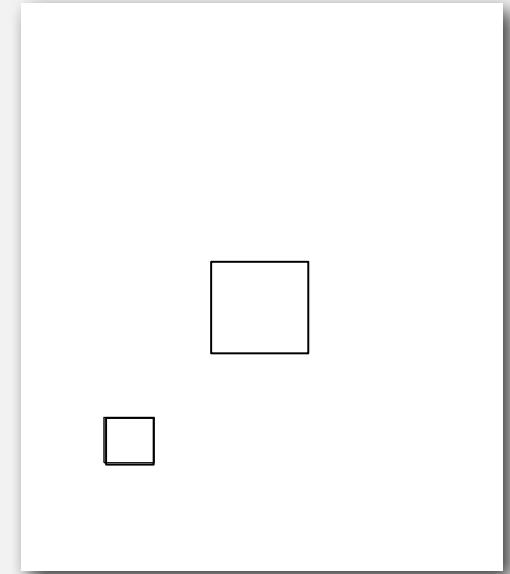
## Variables (and functions).

- Identifiers start with /.
- def operator associates id with value.
- Braces.
- args on stack.

```
%!
function      → /box
definition
{
    /sz exch def
    0 sz rlineto
    sz 0 rlineto
    0 sz neg rlineto
    sz neg 0 rlineto
} def

72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```

function  
calls



# PostScript

---

## For loop.

- “from, increment, to” on stack.
- Loop body in braces.
- `for` operator.

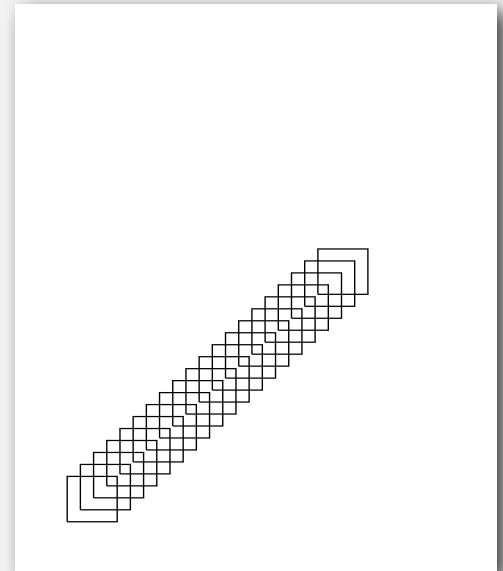
```
%!
\box
{
    ...
}

1 1 20
{ 19 mul dup 2 add moveto 72 box }
for
stroke
```

## If-else conditional.

- Boolean on stack.
- Alternatives in braces.
- `if` operator.

... (hundreds of operators)



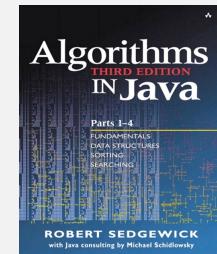
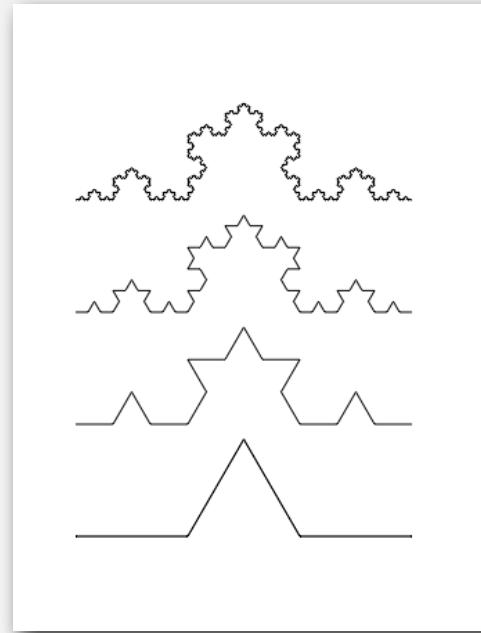
# PostScript applications

Algorithms, 3<sup>rd</sup> edition. Figures created directly in PostScript.

```
%!
72 72 translate

/kochR
{
    2 copy ge { dup 0 rlineto }
    {
        3 div
        2 copy kochR 60 rotate
        2 copy kochR -120 rotate
        2 copy kochR 60 rotate
        2 copy kochR
    } ifelse
    pop pop
} def

0 0 moveto 81 243 kochR
0 81 moveto 27 243 kochR
0 162 moveto 9 243 kochR
0 243 moveto 1 243 kochR
stroke
```



see page 218

Algorithms, 4<sup>th</sup> edition. Figures created using enhanced version of StdDraw that saves to PostScript for vector graphics.



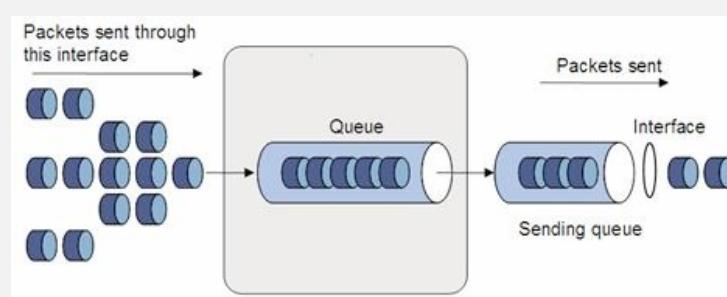
# Queue applications

## Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

## Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

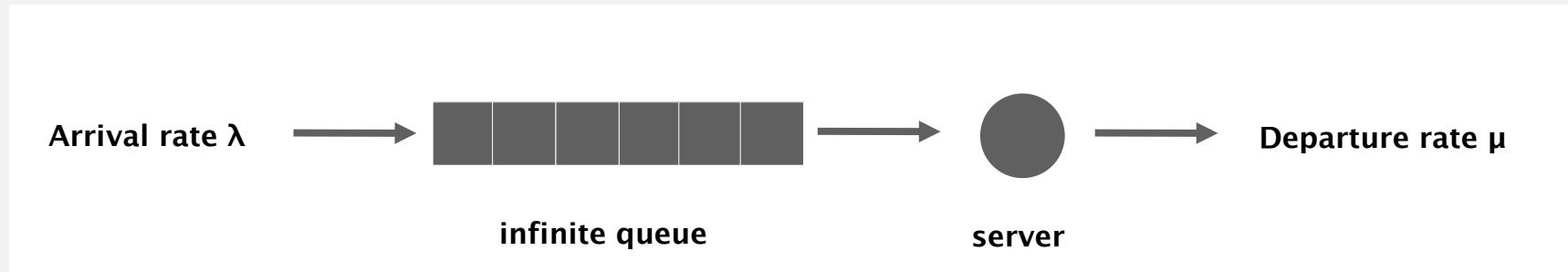


# M/M/1 queuing model

## M/M/1 queue.

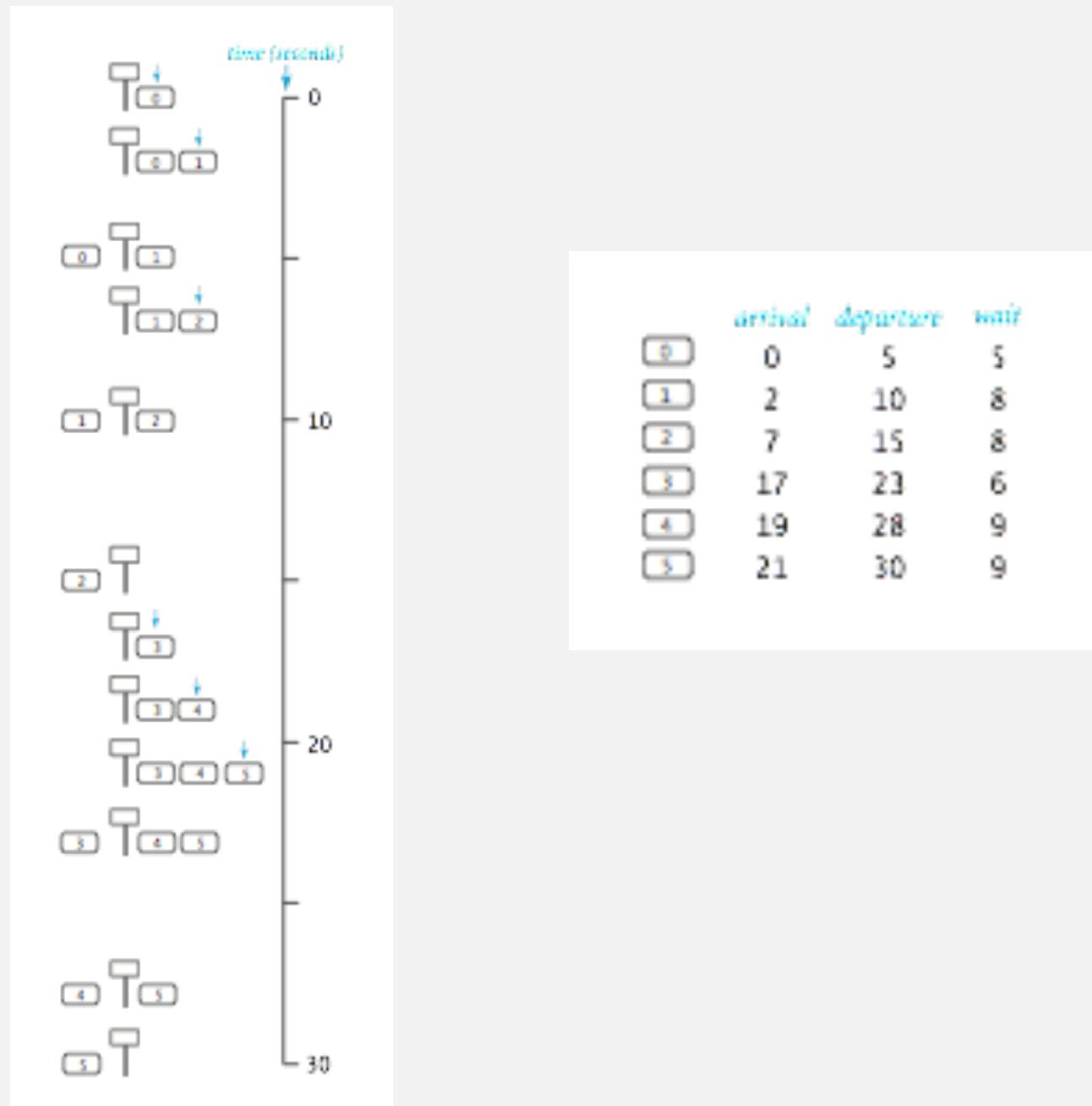
- Customers arrive according to **Poisson process** at rate of  $\lambda$  per minute.
- Customers are serviced with rate of  $\mu$  per minute.

interarrival time has exponential distribution  $\Pr[X \leq x] = 1 - e^{-\lambda x}$   
service time has exponential distribution  $\Pr[X \leq x] = 1 - e^{-\mu x}$



- Q. What is average wait time  $W$  of a customer in system?  
Q. What is average number of customers  $L$  in system?

# M/M/1 queuing model: example simulation



# M/M/1 queuing model: event-based simulation

```
public class MM1Queue
{
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);      // arrival rate
        double mu     = Double.parseDouble(args[1]);      // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);

        Queue<Double> queue = new Queue<Double>();          queue of arrival times
        Histogram hist = new Histogram("M/M/1 Queue", 60);

        while (true)
        {
            while (nextArrival < nextService)
            {
                queue.enqueue(nextArrival);                  next event is an arrival
                nextArrival += StdRandom.exp(lambda);
            }

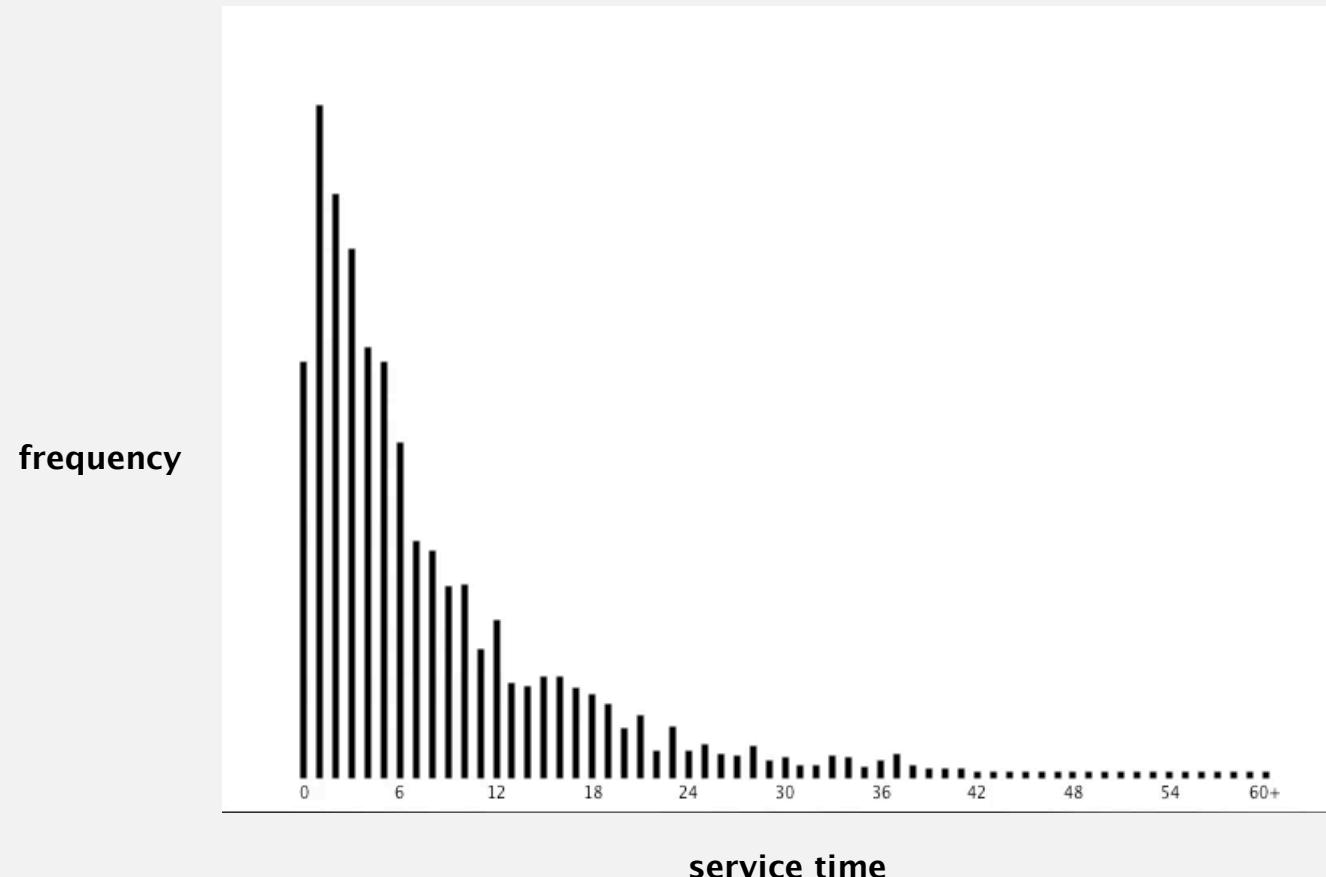
            double arrival = queue.dequeue();           next event is a service completion
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60, (int) (Math.round(wait)))); 
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else                  nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```

## M/M/1 queuing model: experiments

---

**Observation.** If service rate  $\mu$  is much larger than arrival rate  $\lambda$ , customers gets good service.

```
% java MM1Queue .2 .333
```

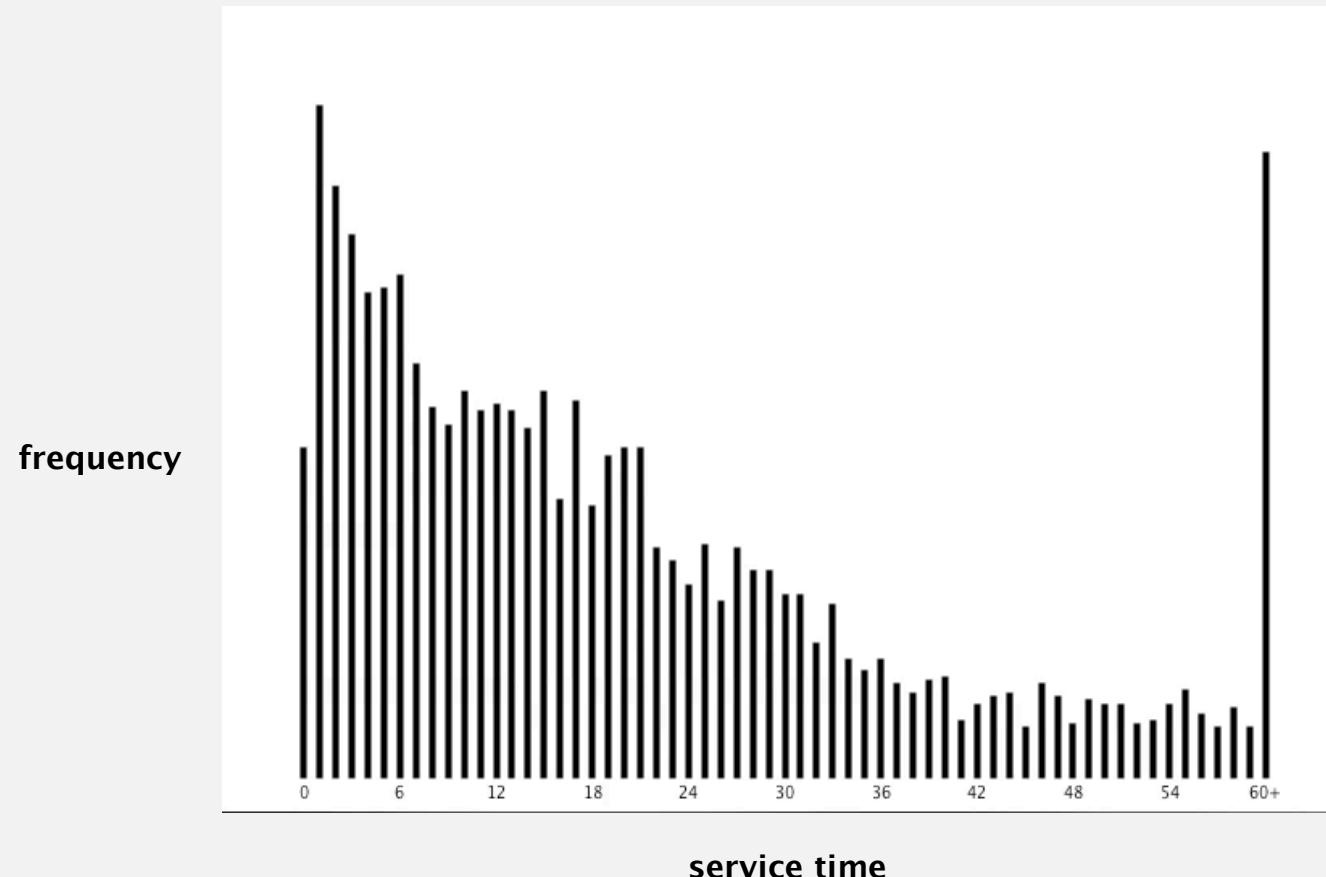


## M/M/1 queuing model: experiments

---

**Observation.** As service rate  $\mu$  approaches arrival rate  $\lambda$ ,  
customer service goes to hell\*\*\*.

```
% java MM1Queue .2 .25
```

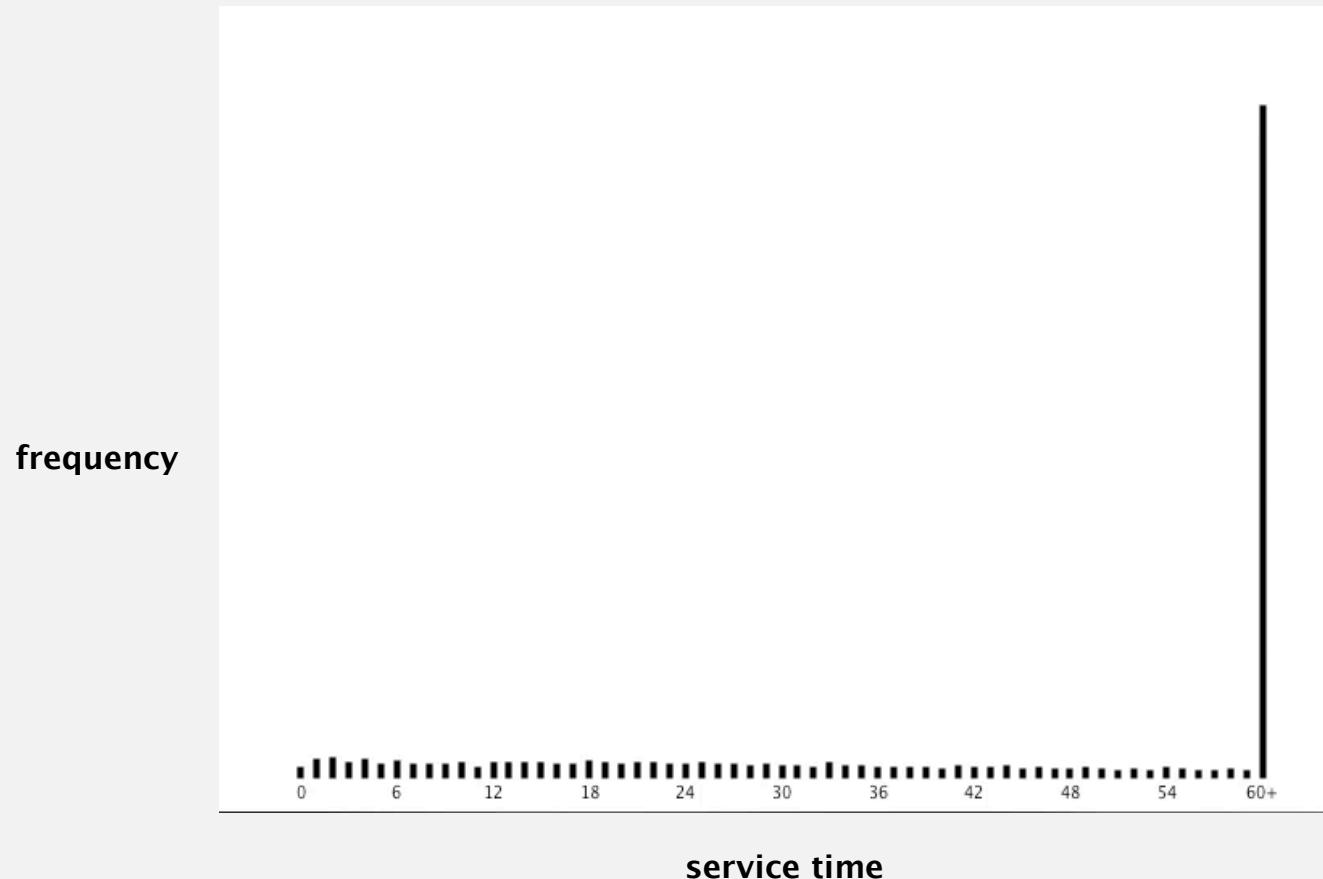


## M/M/1 queuing model: experiments

---

Observation. As service rate  $\mu$  approaches arrival rate  $\lambda$ ,  
customer service goes to  $h^{***}$ .

```
% java MM1Queue .2 .21
```



# M/M/1 queuing model: analysis

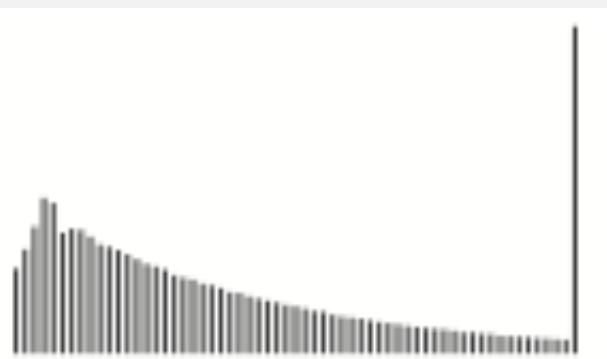
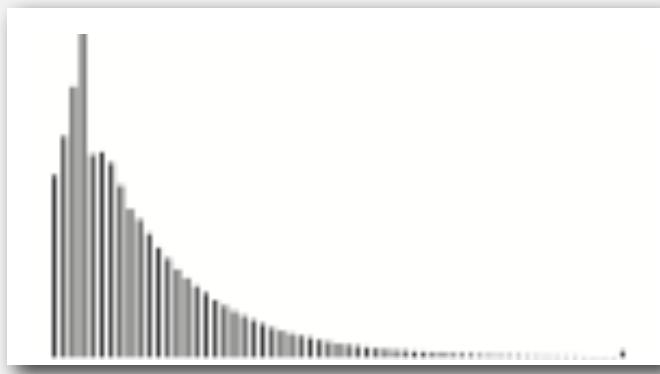
M/M/1 queue. Exact formulas known.

wait time  $W$  and queue length  $L$  approach infinity

as service rate approaches arrival rate

Little's Law

$$W = \frac{1}{\mu - \lambda}, \quad L = \lambda W$$



More complicated queueing models. Event-based simulation essential!