

Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Review: Arrays

In Java, an array is an indexed collection of data values of the same type.

- Array declaration and creation

```
<data type> [ ] <variable>  
<variable> = new <data type> [ <size> ]
```

- Example:

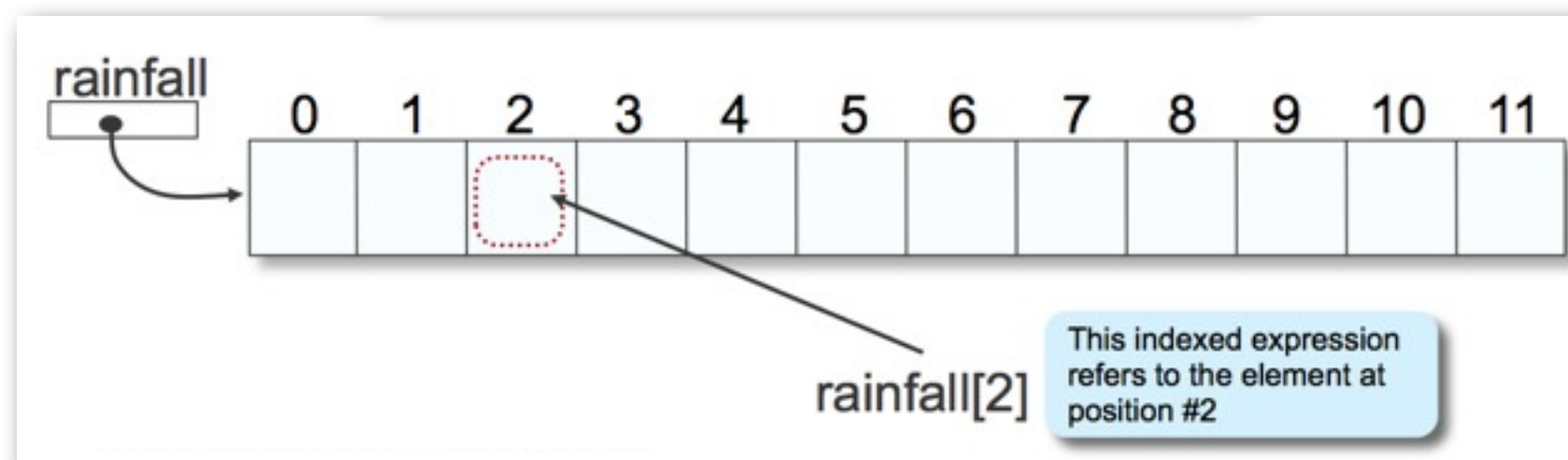
```
double[ ] rainfall;  
rainfall = new double[12];
```

An array is like an object

- Like other data types, it is possible to declare and initialize an array at the same time:

```
int[] number = { 2, 4, 6, 8 };
```

- Individual elements in an array are accessed with an indexed expression.



Review: Arrays

Index out of bounds

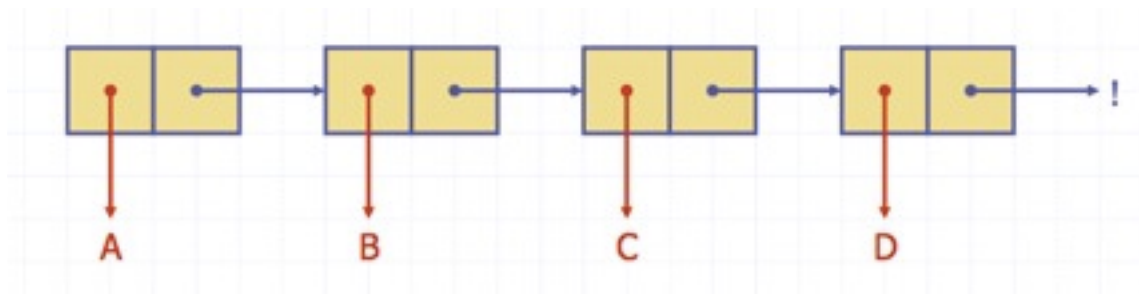
- The index for an array `a`, must evaluate to a value between 0 and `a.length-1`.
- If it does not, then an “`ArrayIndexOutOfBoundsException`” is thrown
- This run time exception does not need to be caught -- but will cause the program to terminate if not caught.

Arrays of Objects

- In Java, in addition to arrays of primitive data types, we can declare arrays of objects
- The use of an array of objects allows us to model the application more cleanly and logically.

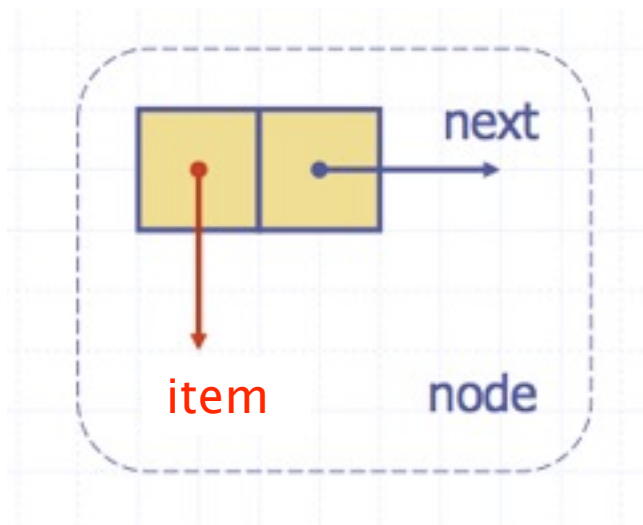
Review: Linked lists

A singly linked list is a data structure consisting of a sequence of nodes



Each node stores:

- Item
- Link to the next node



```
public class Node {
    private Object item;        // Instance variable
    private Node next;          // Instance variable
    /** Creates a node with null references. */
    public Node(){
        this(null, null);
    }
    /** Creates a node with specified contents. */
    public Node(Object e, Node n){
        item = e;
        next = n;
    }

    // Accessor methods:
    public Object getItem() { return item; }
    public Node getNext() { return next; }

    // Modifier methods:
    public void setItem(Object newItem){
        item = newItem;
    }
    public void setNext(Node newNext){
        next = newNext;
    }
}
```

Review: Linked list

Creating a linked list

- Need a reference variable, **first**, that identifies the first node in the list

Traversing a linked list

- Once you are at the first node, you can use **node.getNext()** to get to the next node
- Scan a linked list by assigning a variable **curr** to the value of **first**, then use the **node.getNext()** method of each node to proceed down the list
- Conclude when **curr == null**

Other details

- If a linked list is empty, then **first** value is **null**
- Inserting or deleting an element at the front of the list is easy, because the list maintains a reference that points to the first element

Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Stacks and queues

Fundamental data types.

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

Stack. Remove the item most recently added.

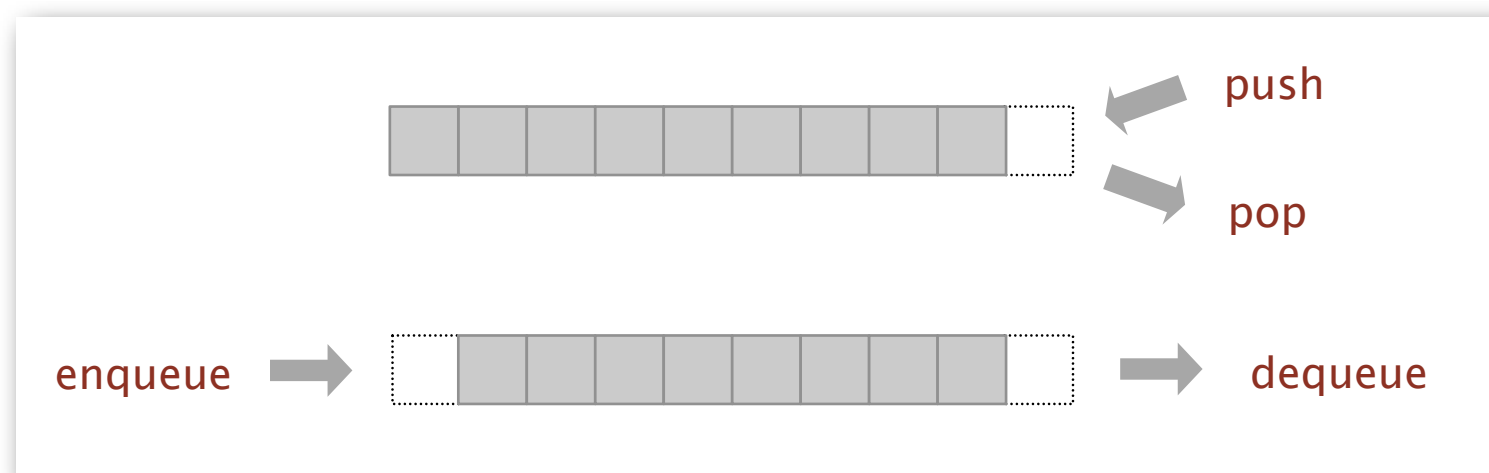
LIFO = "last in first out"

Analogy. Cafeteria trays, Web surfing.

Queue. Remove the item least recently added.

FIFO = "first in first out"

Analogy. Registrar's line.



Client, implementation, interface

Separate interface and implementation.

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

Benefits.

- Client can't know details of implementation \Rightarrow
can easily substitute new and improved implementation.
- 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: refers to a program that calls a method in another library.

Implementation: refers to the Java code that implements the methods in an API.

Interface: refers to a list of methods with names, signatures, basic descriptions.

Stack API

Warmup. Stack of strings objects.

```
public class StackOfStrings
```

```
    StackOfStrings()           create an empty stack
```

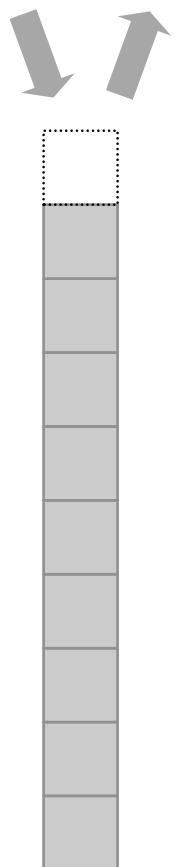
```
    void push(String s)       insert a new item onto stack
```

```
    String pop()              remove and return the item  
                              most recently added
```

```
    boolean isEmpty()         is the stack empty?
```

```
    int size()                number of items on the stack
```

push pop



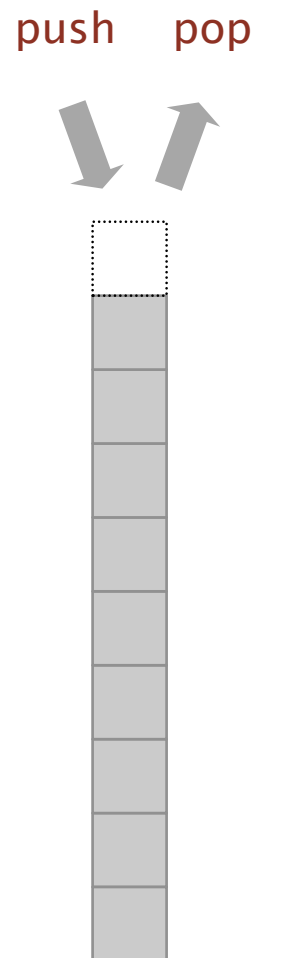
Challenge. Reverse sequence of strings from standard input.

Stack test client

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

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

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



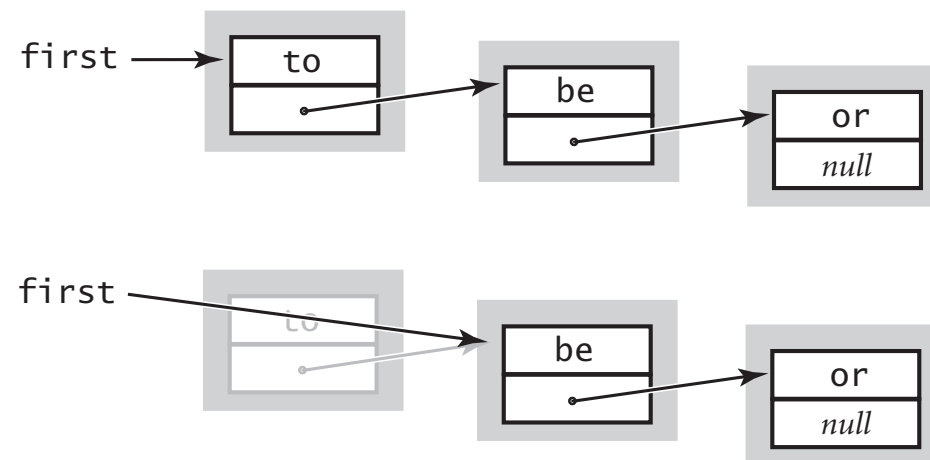
Stack pop: linked-list implementation

save item to return

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

change top-of-stack pointer

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



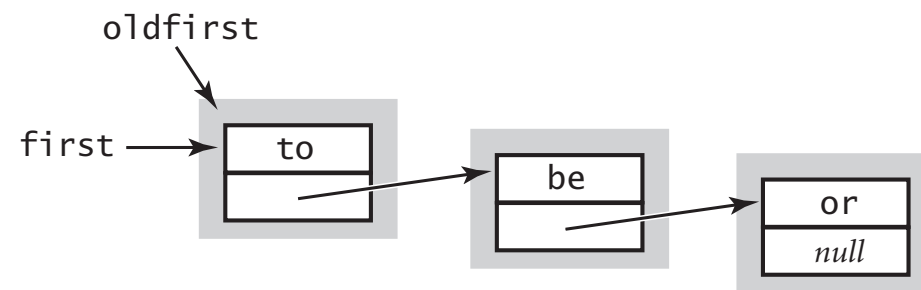
return saved item

```
return item;
```

Stack push: linked-list implementation

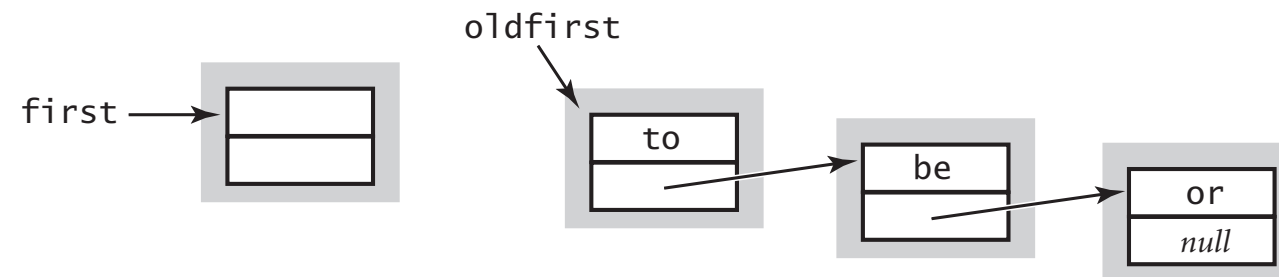
save a link to the list

```
Node oldfirst = first;
```



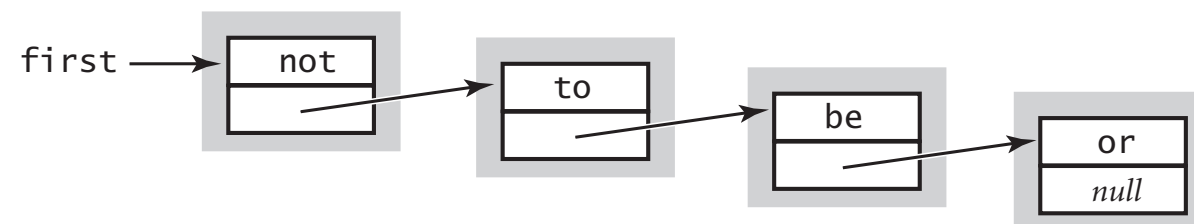
create a new node for the beginning

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



set the instance variables in the new node

```
first.item = "not";  
first.next = oldfirst;
```



Stack: linked-list implementation in Java

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

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

← inner class

```
    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()
    {
        if (isEmpty()) throw new RuntimeException();
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

← stack underflow

Stack: linked-list implementation performance

Proposition. Using a linked-list implementation of a stack, every operation takes constant time in the worst case.

Proposition. Uses $\sim 16 N$ bytes to represent a stack with N items.

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

8 bytes (object overhead)

← 4 bytes (reference to String)

← 4 bytes (reference to Node)

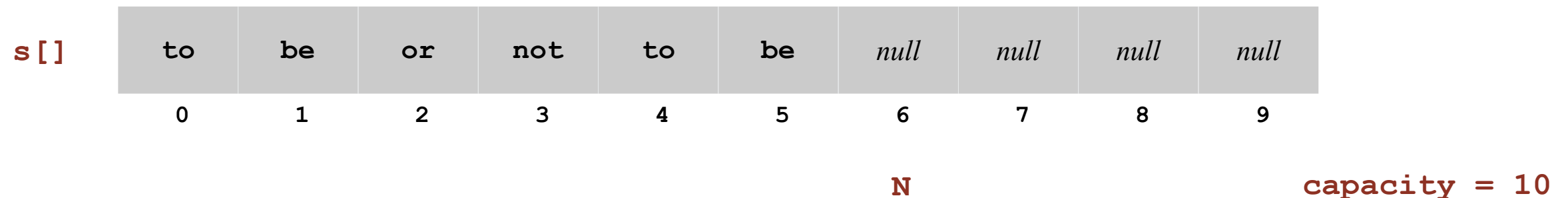
16 bytes per stack item

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

Stack: array implementation

Array implementation of a stack.

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



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

Stack: array implementation

```
public class StackOfStrings
{
    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]; }
}
```

a cheat (stay tuned)



decrement N;
then use to index into array

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

this version avoids "loitering":
garbage collector reclaims memory
only if no outstanding references

Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Stack: dynamic-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 **s[]** by 1.
- **pop()**: decrease size of **s[]** by 1.

Too expensive.

- Need to copy all item to a new array.
- Inserting first N items takes time proportional to $1 + 2 + \dots + N \sim N^2 / 2$.

↑
infeasible for large N

Challenge. Ensure that array resizing happens infrequently.

Stack: dynamic-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 StackOfStrings() { 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;
}
```

cost of array resizing is now

$$2 + 4 + 8 + \dots + N \sim 2N$$

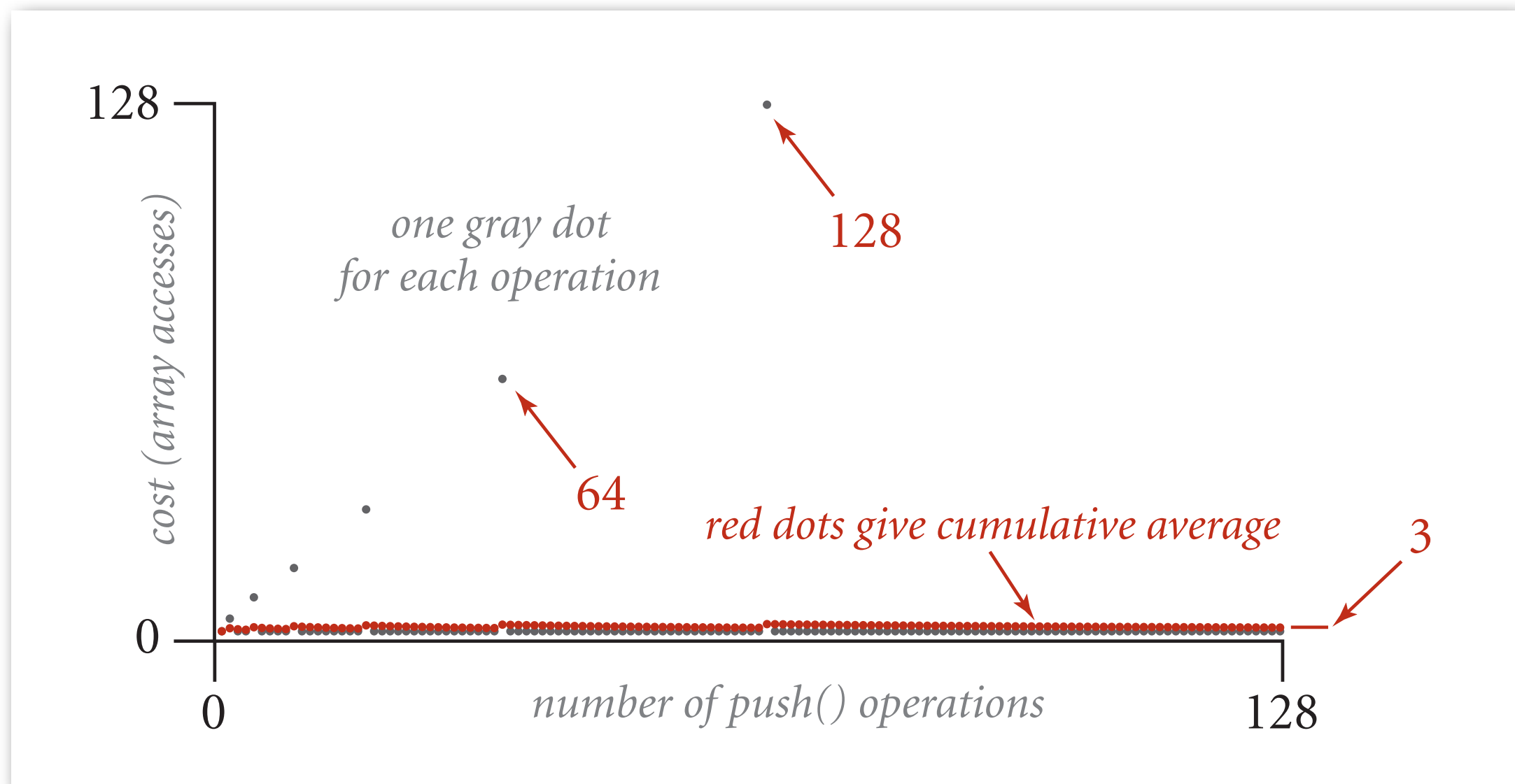
Consequence. Inserting first N items takes time proportional to N (not N^2).

Stack: amortized cost of adding to a stack

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

\uparrow 1 array accesses
per push

\uparrow k array accesses
to double to size k



Stack: dynamic-array implementation

Q. How to shrink array?

First try.

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

Too expensive.

"thrashing"



- Consider push-pop-push-pop-... sequence when array is full.
- Takes time proportional to N per operation in worst case.

N = 5

to	be	or	not	to	<i>null</i>	<i>null</i>	<i>null</i>
----	----	----	-----	----	-------------	-------------	-------------

N = 4

to	be	or	not
----	----	----	-----

N = 5

to	be	or	not	to	<i>null</i>	<i>null</i>	<i>null</i>
----	----	----	-----	----	-------------	-------------	-------------

N = 4

to	be	or	not
----	----	----	-----

Stack: dynamic-array implementation

Q. How to shrink array?

Efficient solution.

- **push()**: double size of **s[]** when array is full.
- **pop()**: halve size of **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: dynamic-array implementation trace

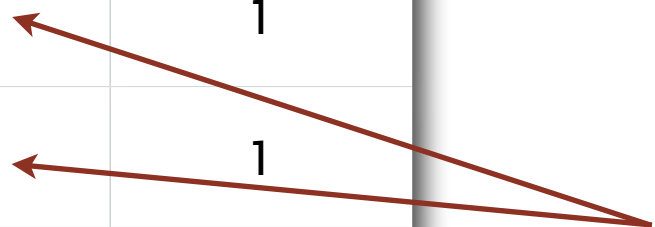
StdIn	StdOut	N	a.length	a							
				0	1	2	3	4	5	6	7
		0	1	<i>null</i>							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	<i>null</i>				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	<i>null</i>	<i>null</i>	<i>null</i>
-	to	4	8	to	be	or	not	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
be		5	8	to	be	or	not	be	<i>null</i>	<i>null</i>	<i>null</i>
-	be	4	8	to	be	or	not	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
-	not	3	8	to	be	or	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
that		4	8	to	be	or	that	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
-	that	3	8	to	be	or	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
-	or	2	4	to	be	<i>null</i>	<i>null</i>				
-	be	1	2	to	<i>null</i>						
is		2	2	to	is						

Stack dynamic-array implementation: performance

Amortized analysis. Average running time per operation over a worst-case sequence of operations. [stay tuned]

Proposition. Starting from empty stack (with dynamic resizing), any sequence of M push and pop operations takes time proportional to M .

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



doubling and shrinking

running time for doubling stack with N items

Stack dynamic array implementation: memory usage

Proposition. Uses between $\sim 4 N$ and $\sim 16 N$ bytes to represent a stack with N items.

- $\sim 4 N$ when full.
- $\sim 16 N$ when one-quarter full.

```
public class DoublingStackOfStrings
{
    private String[] s;
    private int N = 0;
    ...
}
```

← 4 bytes × array size
← 4 bytes

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

Stack implementations: dynamic array vs. linked List

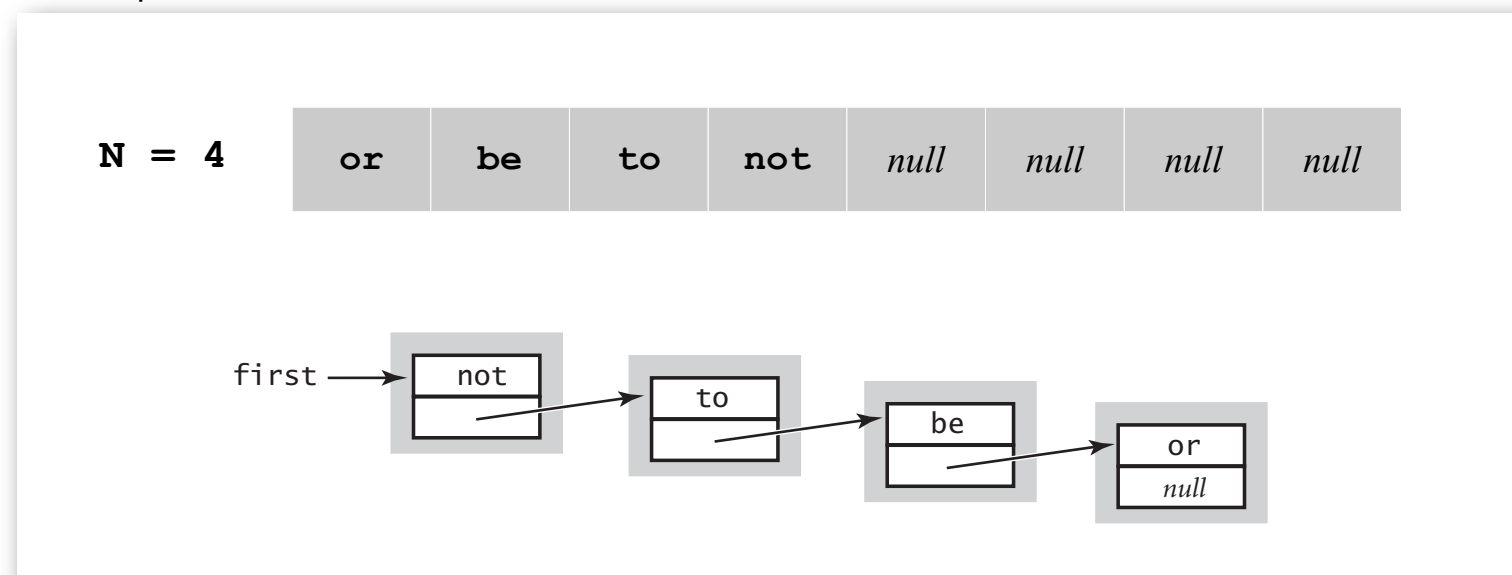
Tradeoffs. Can implement a stack with either dynamic array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

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

Dynamic-array implementation.

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



Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings ()
```

create an empty queue

```
    void enqueue (String s)
```

insert a new item onto queue

```
    String dequeue ()
```

*remove and return the item
least recently added*

```
    boolean isEmpty ()
```

is the queue empty?

```
    int size ()
```

number of items on the queue

enqueue



dequeue



Queue test client

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

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

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

Queue dequeue: linked-list implementation

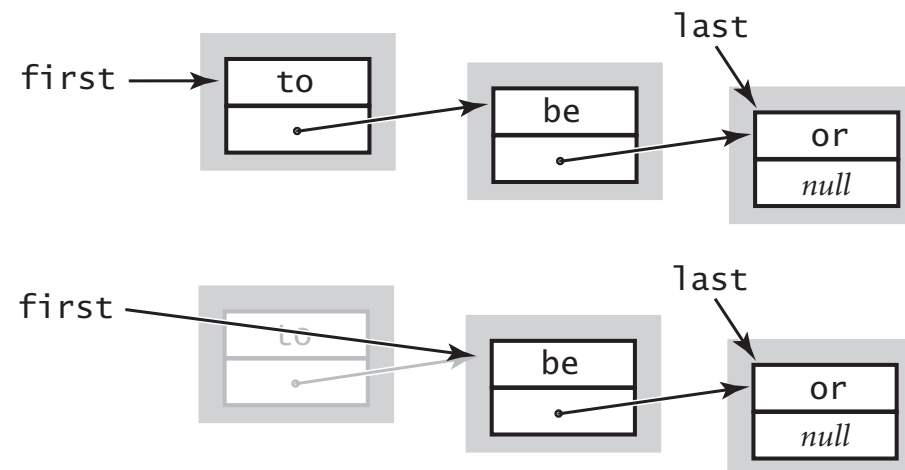
save item to return

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

change head-of-queue pointer

save item to return

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



return saved item

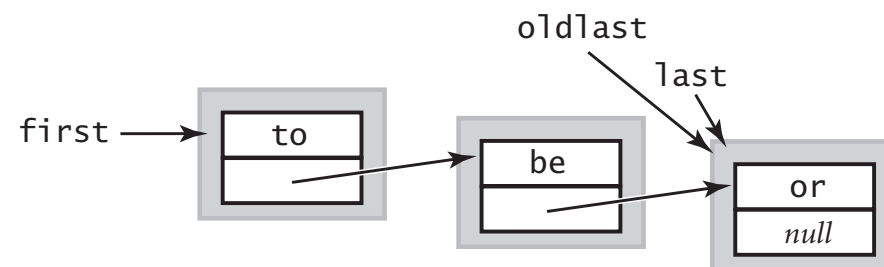
```
return item;
```

Remark. Identical code to linked-list stack **pop()**.

Queue enqueue: linked-list implementation

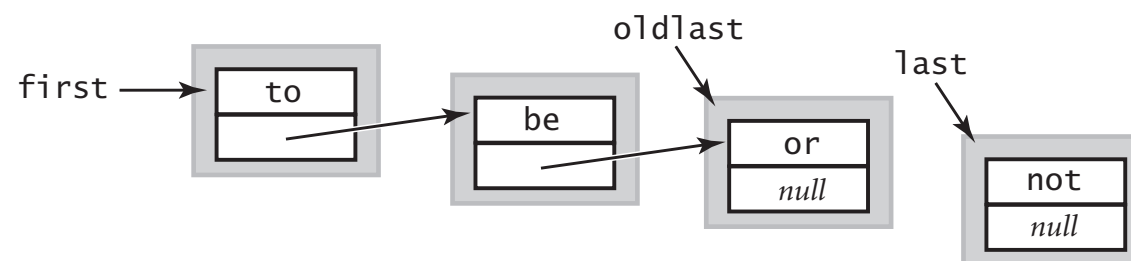
save a link to the last node

```
Node oldlast = last;
```



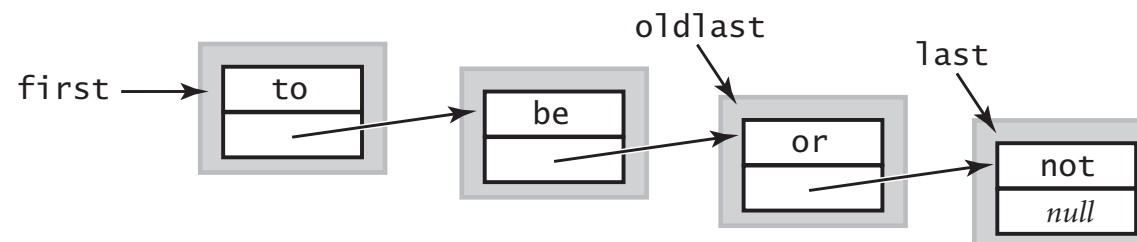
create a new node for the end

```
Node last = new Node();  
last.item = "not";  
last.next = null;
```



link the new node to the end of the list

```
oldlast.next = last;
```



Queue: linked-list implementation in Java

```
public class QueueOfStrings
{
    private Node first, last;

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

    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

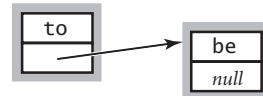
Queue: linked-list trace

StdIn StdOut

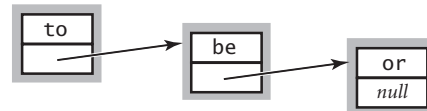
to



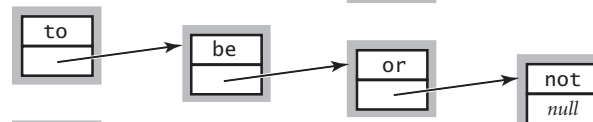
be



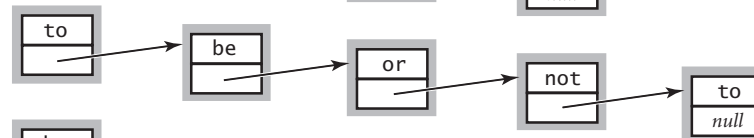
or



not

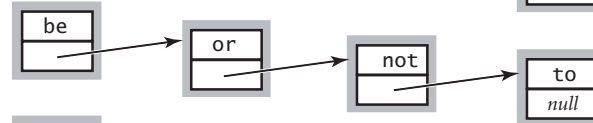


to

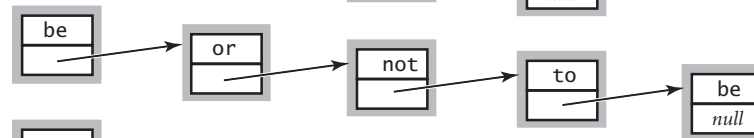


-

to

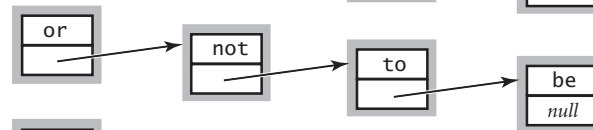


be



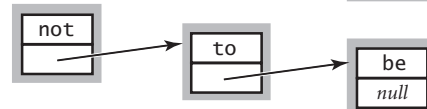
-

be

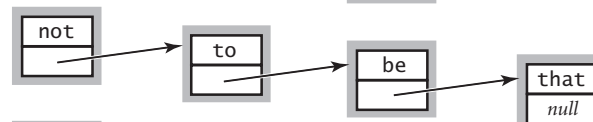


-

or

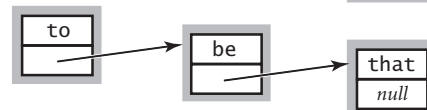


that



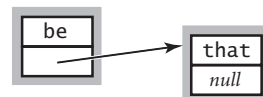
-

not



-

to

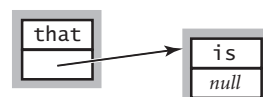


-

be



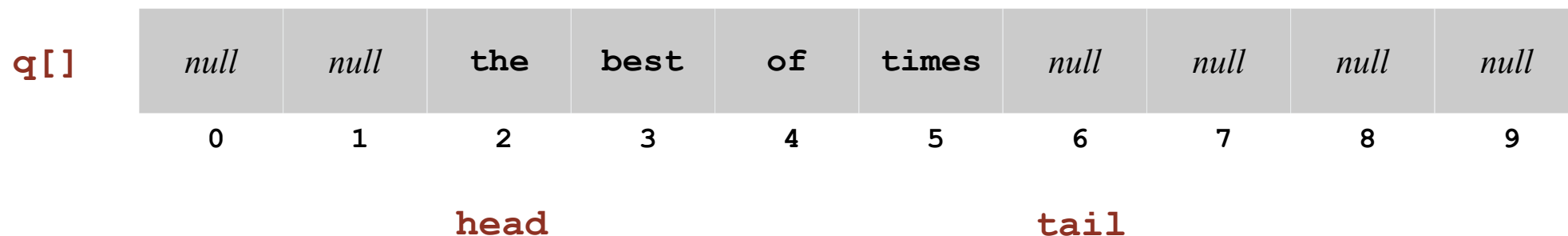
is



Queue: dynamic array implementation

Array implementation of a queue.

- 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 dynamic resizing.



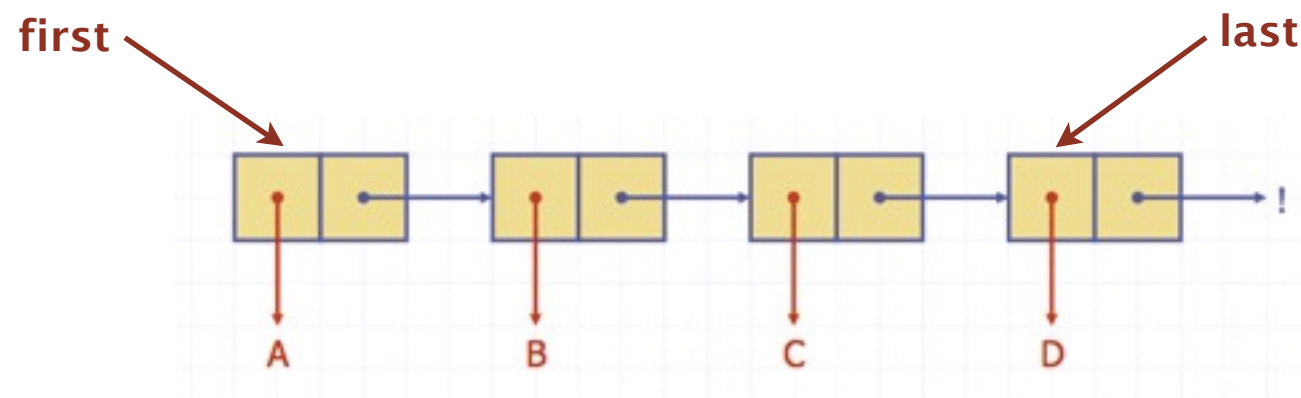
Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

More on singly-linked lists

Note that the queue implementation needed a pointer to the end of the list (last) as well as the beginning of the list (first)

- It is easy to maintain the pointers to both ends of the list if all we do is remove from the front and insert at the end



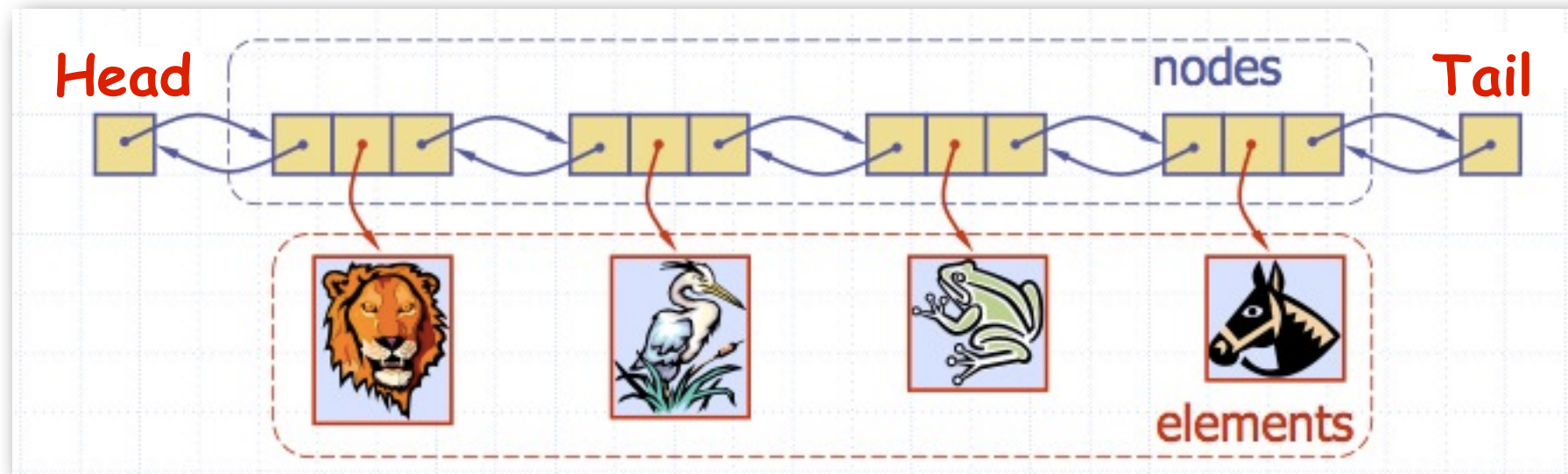
But what if we want to delete at the end of the list?

- There is no constant time way to update the tail to point to the previous!

Doubly-linked lists

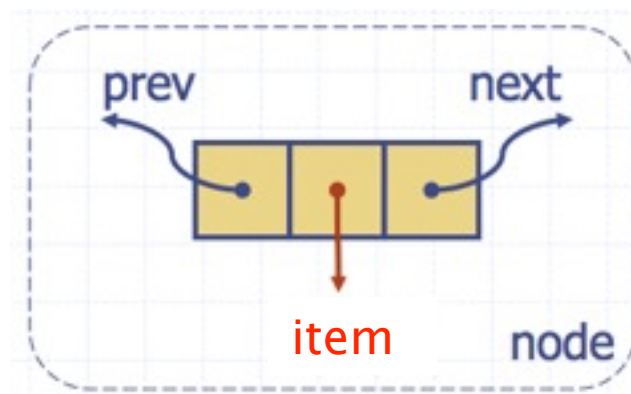
A doubly linked list is a data structure consisting of a sequence of nodes

- Special head and tail nodes



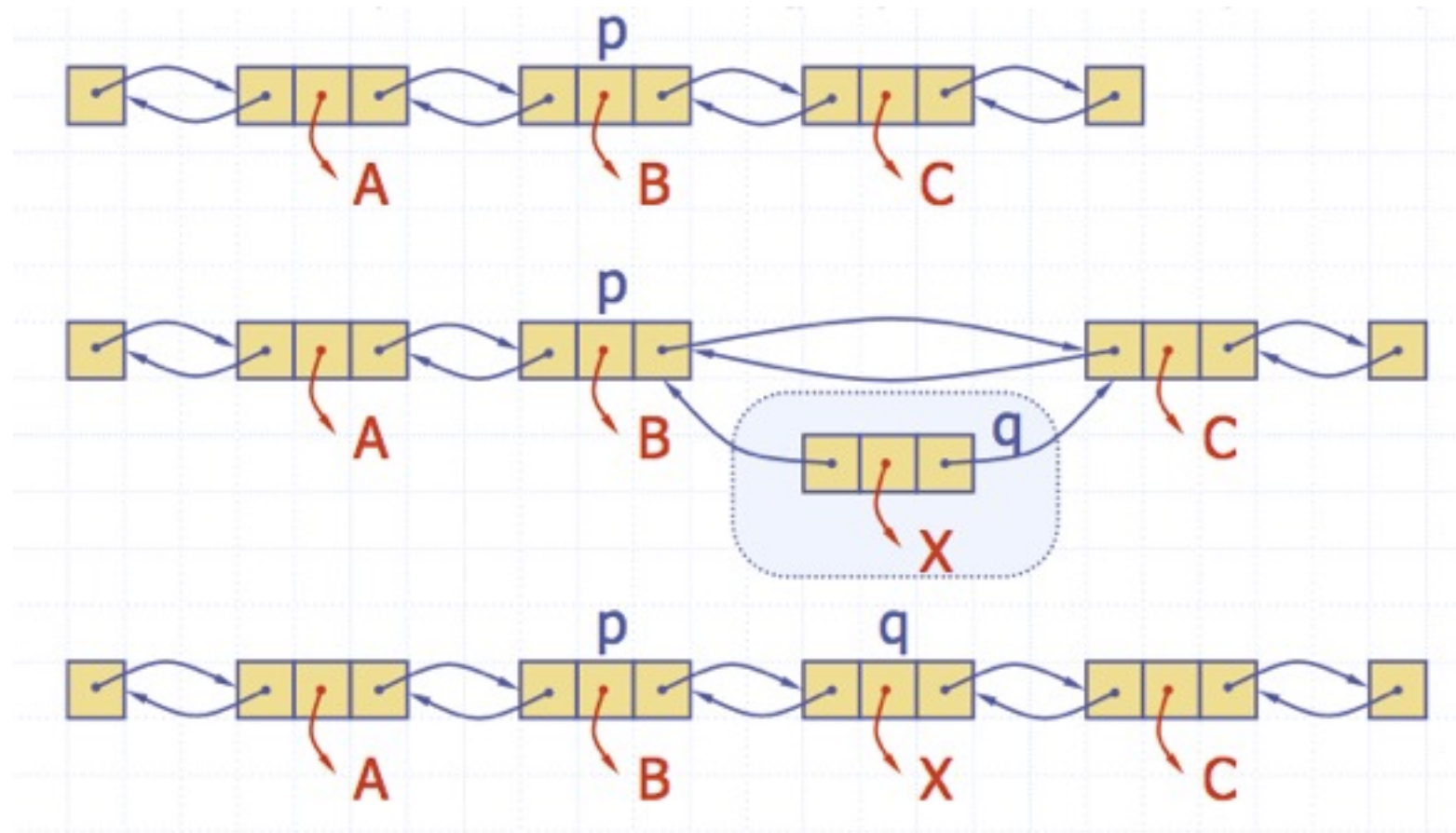
Each node stores:

- Item
- Link to the next node
- Link to the previous node



Insertion

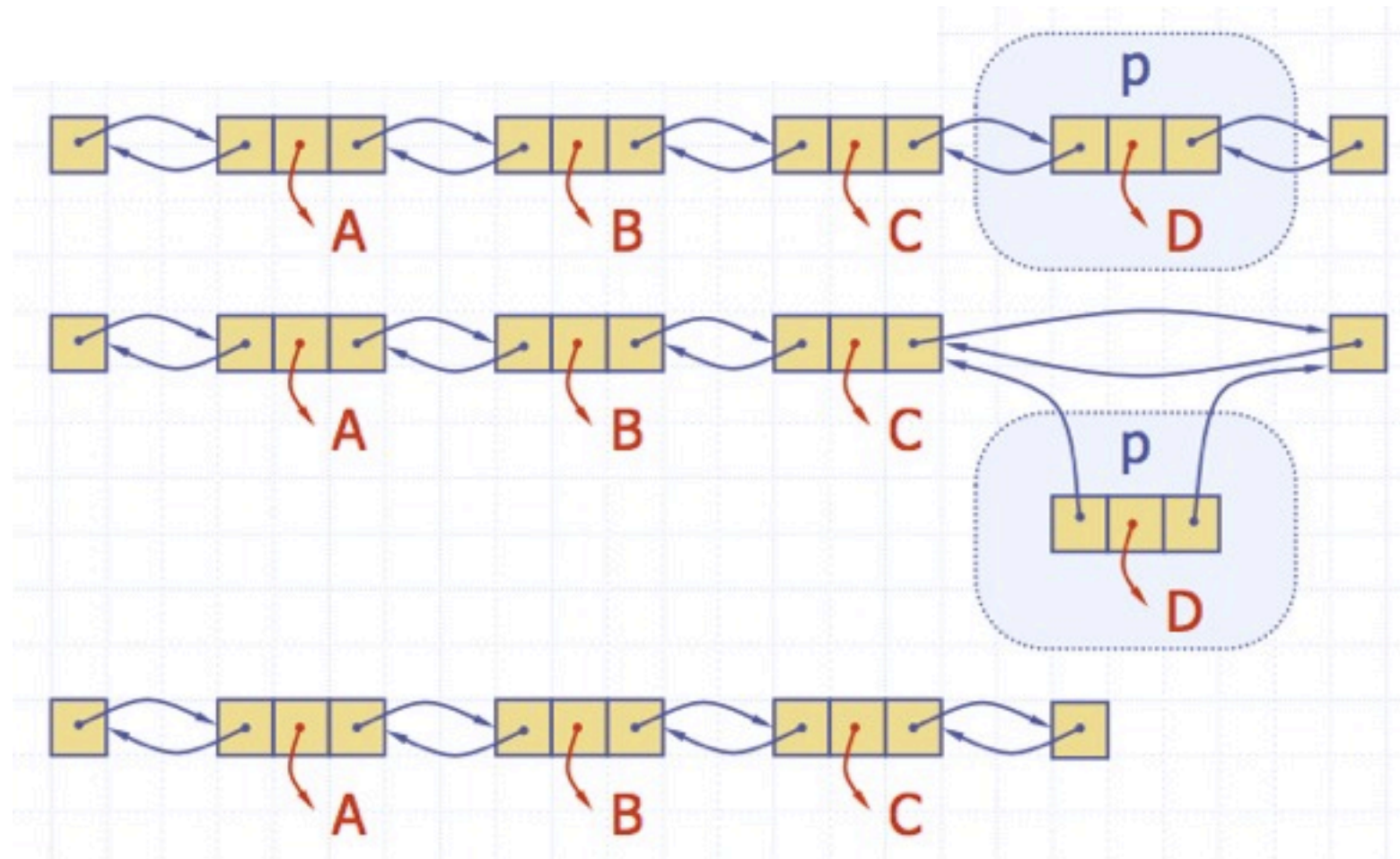
Below we visualize the operation **insertAfter(p,X)**



```
public static void insertAfter(p,x):  
    Node v = new Node()           //Create a new node v  
    v.setItem(x)  
    v.setPrev(p)                   //link v to its predecessor  
    v.setNext(p.getNext())         //link v to its successor  
    (p.getNext()).setPrev(v)       //link p's old successor to v  
    p.setNext(v)                   //link p to its new successor
```

Deletion

Below we visualize the operation **remove(p)**, where $p = \text{last}()$



```
public static Object remove(p):  
    t = p.item                                //tmp variable to hold the return value  
(p.getPrev()).setNext(p.getNext())           //linking over p  
(p.getNext()).setPrev(p.getPrev())  
p.setPrev(null)                               //invalidating the position p  
p.setNext(null)  
return t
```


Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, etc.?

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`, etc.?

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



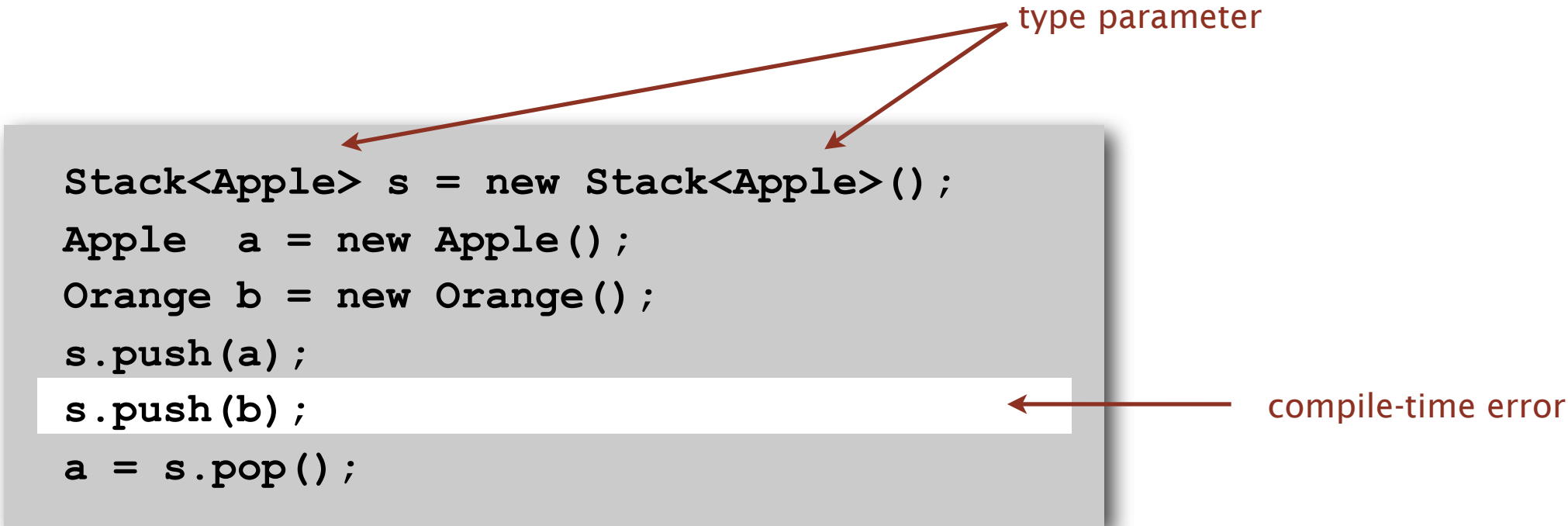
Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, etc.?

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();
```

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;
    }
}
```

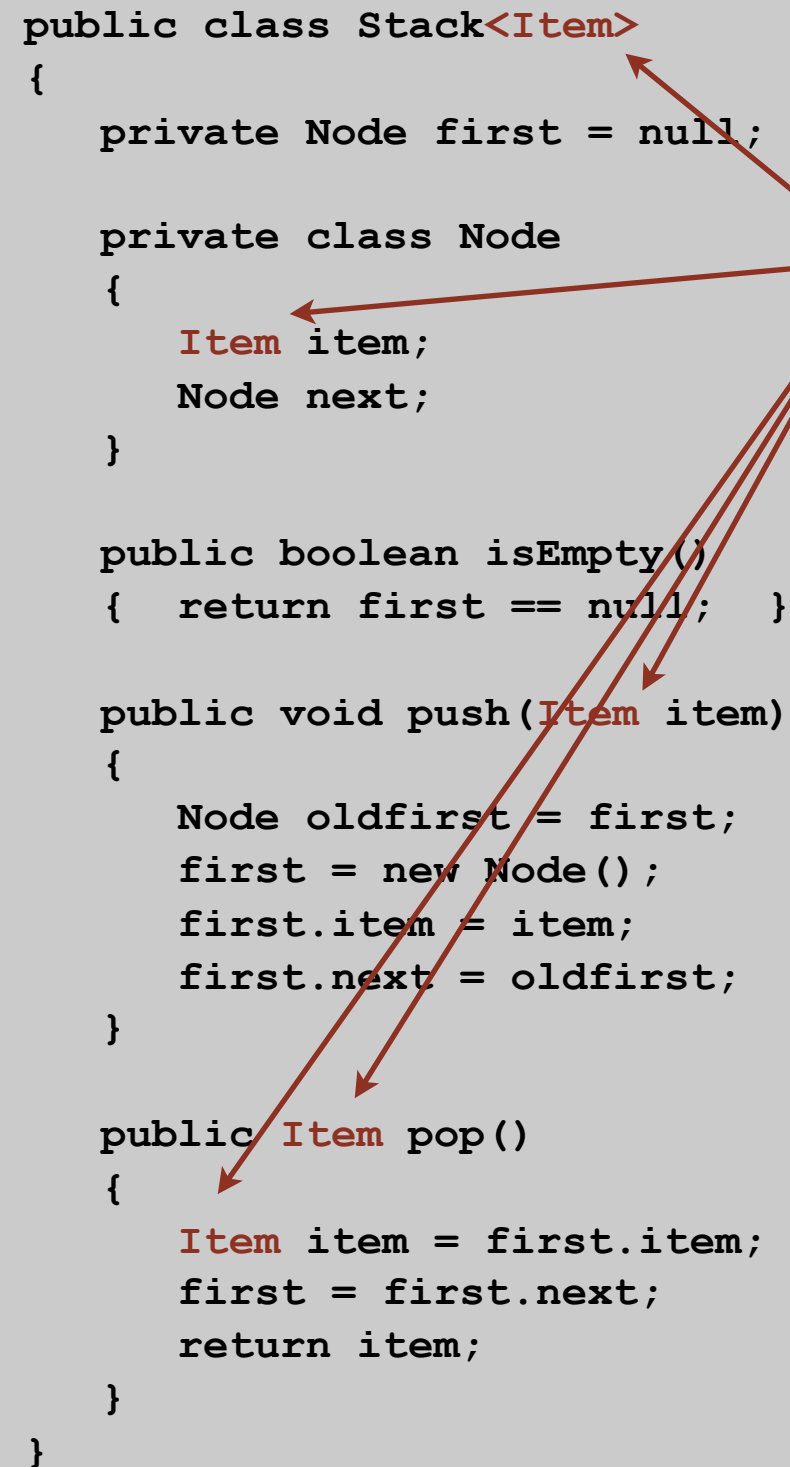
```
public class Stack<Item>
{
    private Node first = null;

    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;
    }
}
```



The diagram illustrates the use of the generic type name 'Item' in the Stack class. Red arrows point from the text 'generic type name' to the following occurrences of 'Item': the class declaration 'Stack<Item>', the 'item' field in the 'Node' class, the 'item' parameter in the 'push' method, the 'Item' return type in the 'pop' method, and the 'Item' variable in the 'pop' method body.

generic type name

Generic stack: array implementation

the way it should be

```
public class ArrayStackOfStrings
{
    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]; }
}
```

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

    public Stack(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

the way it is

```
public class ArrayStackOfStrings
{
    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]; }
}
```

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

    public Stack(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

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.

Syntactic sugar. Behind-the-scenes casting.

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

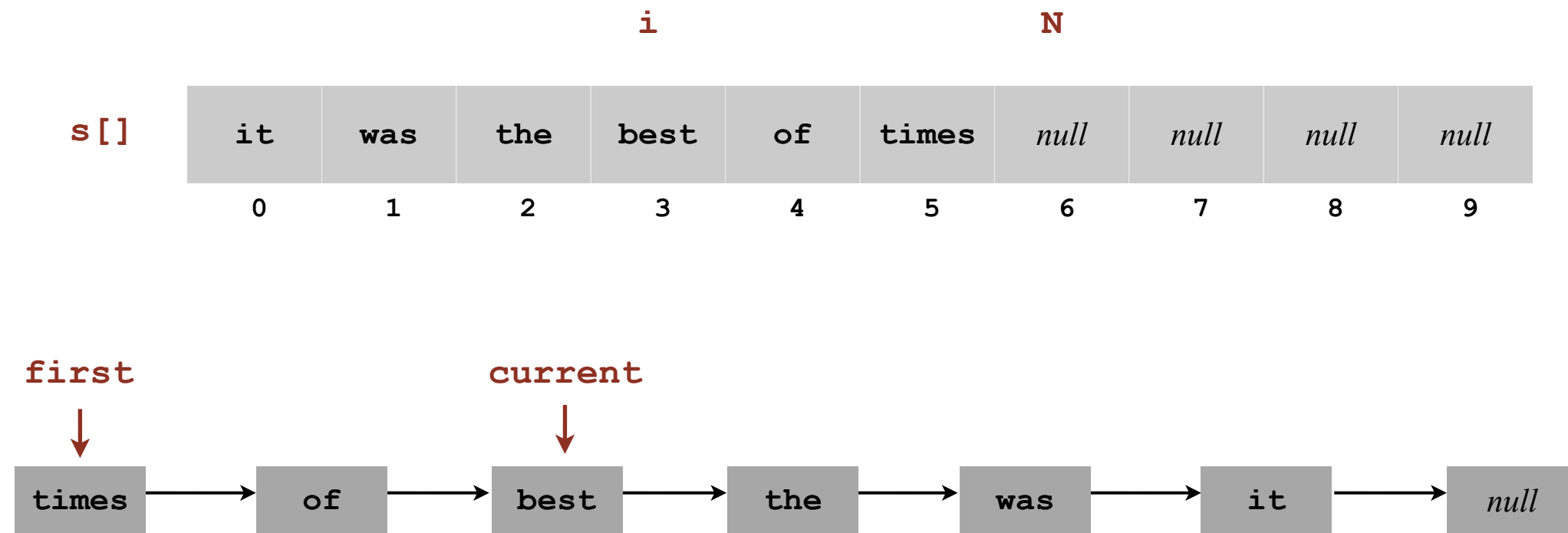
Bottom line. Client code can use generic stack for **any** type of data.

Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- 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 `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.

Iterable interface

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

Iterator interface

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

“foreach” statement

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

equivalent code

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

Stack iterator: linked-list implementation

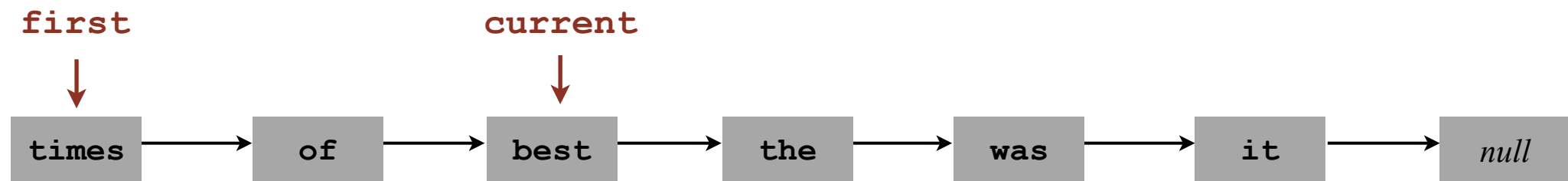
```
import java.util.Iterator;

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;
        }
    }
}
```



Stack iterator: array implementation

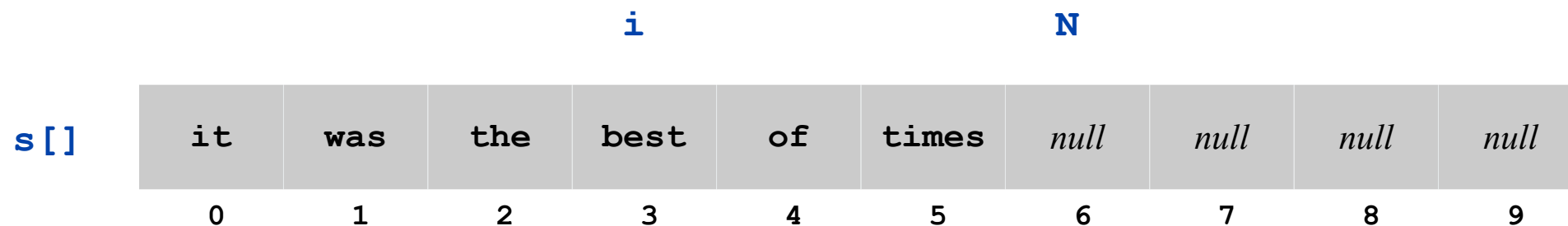
```
import java.util.Iterator;

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

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

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

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



Iteration: concurrent modification

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

A. A fail-fast iterator throws a **ConcurrentModificationException**.

concurrent modification

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

To detect:

Count total number of **push()** and **pop()** operations in **Stack**.

Save current count in ***Iterator** subclass upon creation.

Check that two values are still equal when calling **next()** and **hasNext()**.

Bag API

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

Stacks and Queues

- review: lists, linked lists
- stacks
- dynamic resizing
- queues
- doubly-linked lists
- generics
- iterators
- applications

Java collections library

List interface. `java.util.List` is API for ordered collection of items.

```
public interface List<Item> implements Iterable<Item>
```

```
    List() create an empty list
```

```
    boolean isEmpty() is the list empty?
```

```
    int size() number of items
```

```
    void add(Item item) append item to the end
```

```
    Item get(int index) return item at given index
```

```
    Item remove(int index) return and delete item at given index
```

```
    boolean contains(Item item) does the list contain the given item?
```

```
    Iterator<Item> iterator() iterator over all items in the list
```

```
    ...
```

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

`java.util.LinkedList` uses linked list.

Java collections library

`java.util.Stack`.

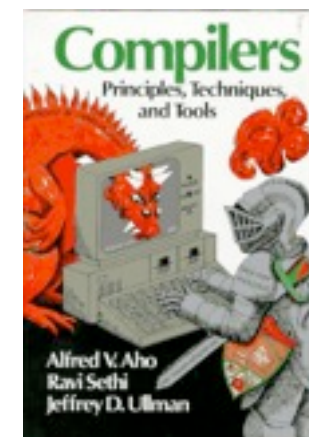
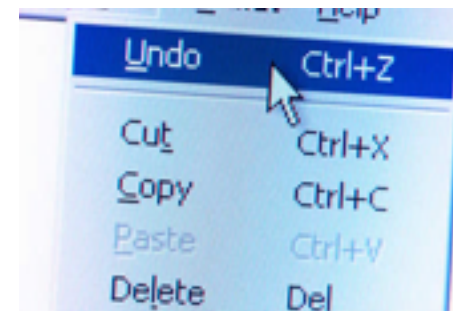
- Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
- Also implements `java.util.List` interface from previous slide, including, `get()`, `remove()`, and `contains()`.
- Bloated and poorly-designed API \Rightarrow don't use.

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

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

Stack applications

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



Function calls

How a compiler implements a function.

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

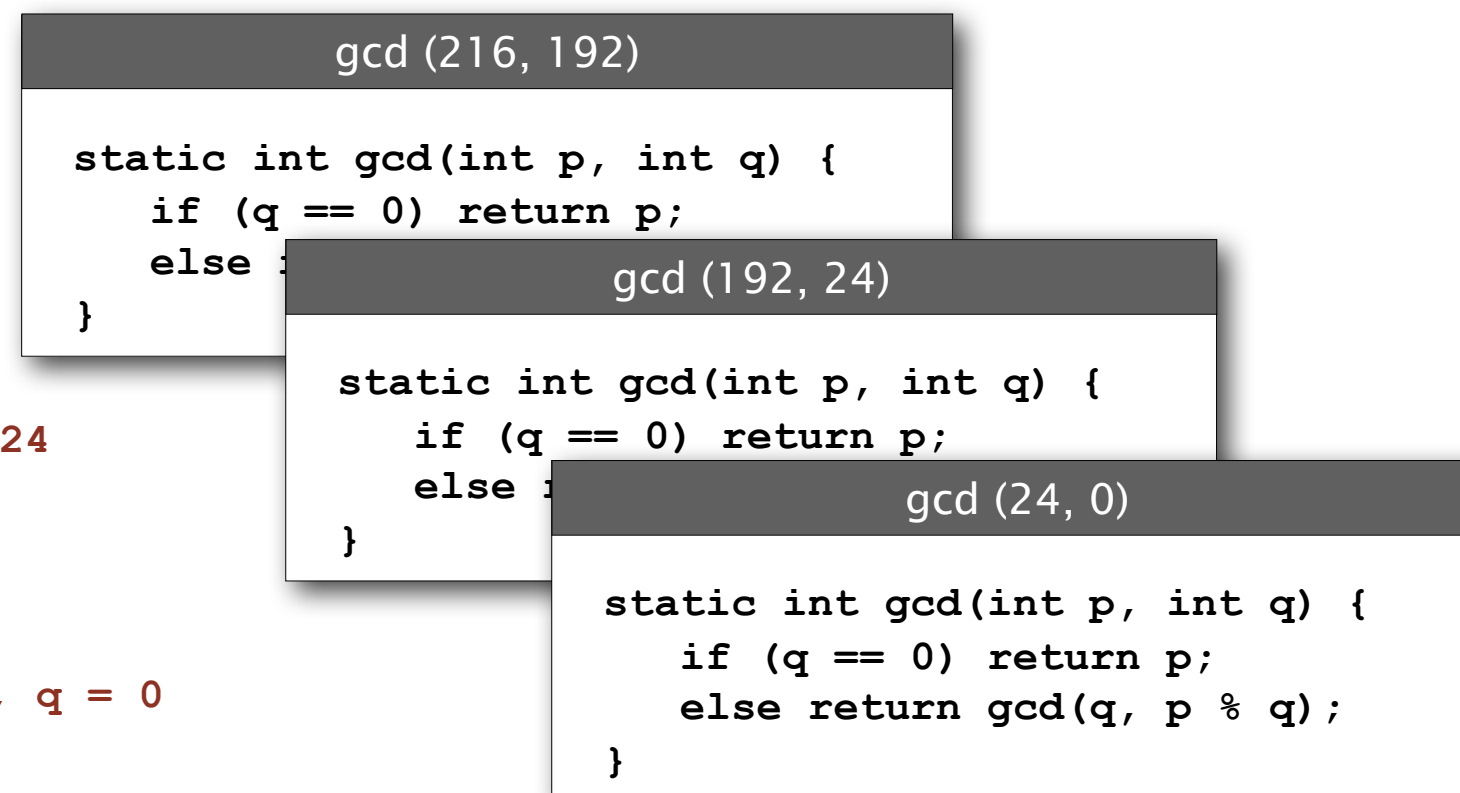
Recursive function. Function that calls itself.

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

p = 216, q = 192

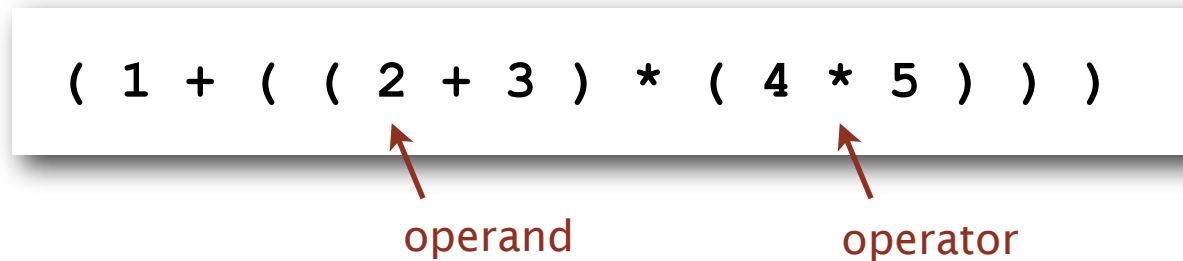
p = 192, q = 24

p = 24, q = 0



Arithmetic expression evaluation

Goal. Evaluate infix expressions.



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

Value: push onto the value stack.

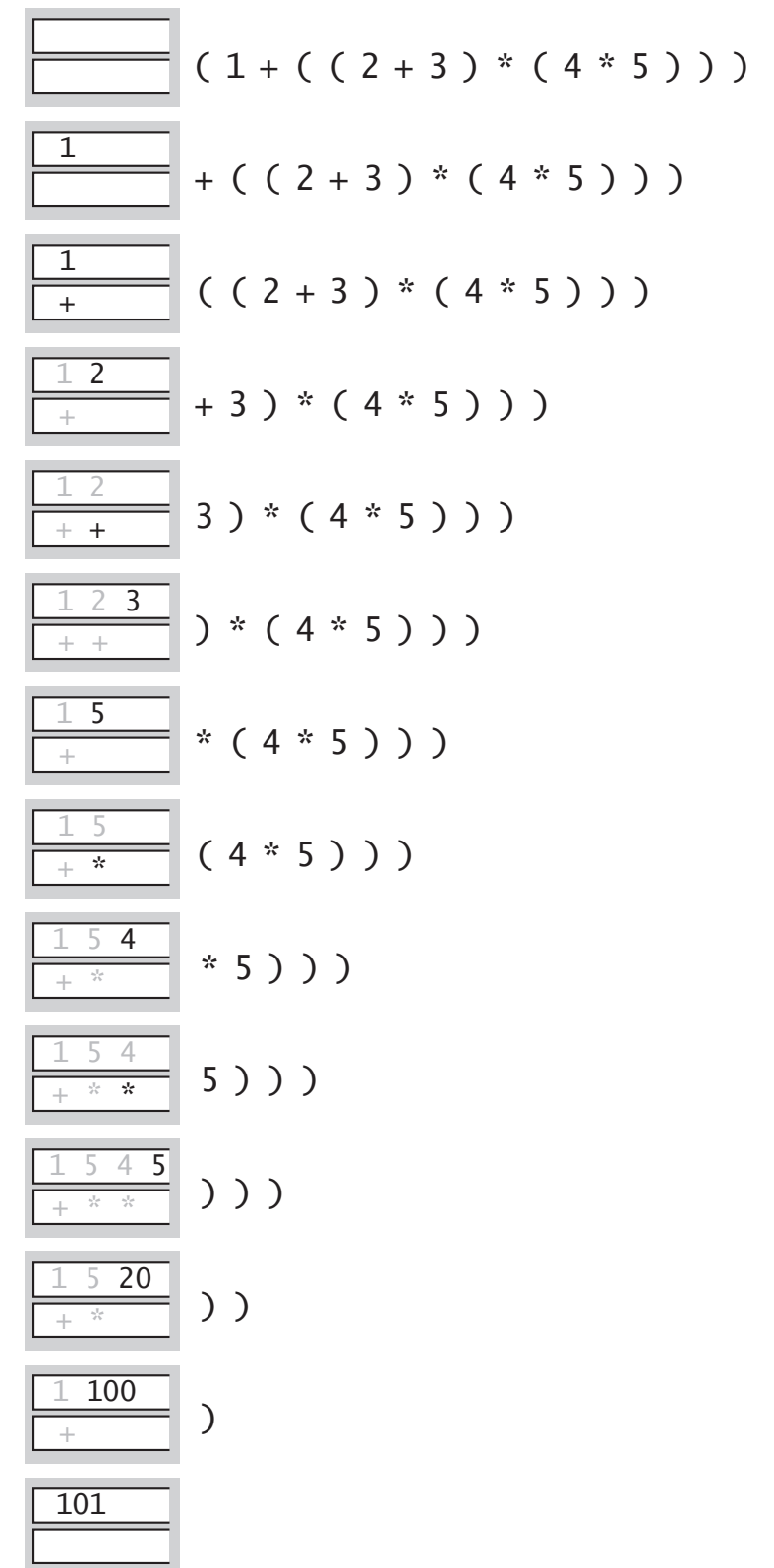
Operator: push onto the operator stack.

Left parens: ignore.

Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

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. The 2-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, ...

PostScript

PostScript. [Warnock-Geschke 1980s]

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

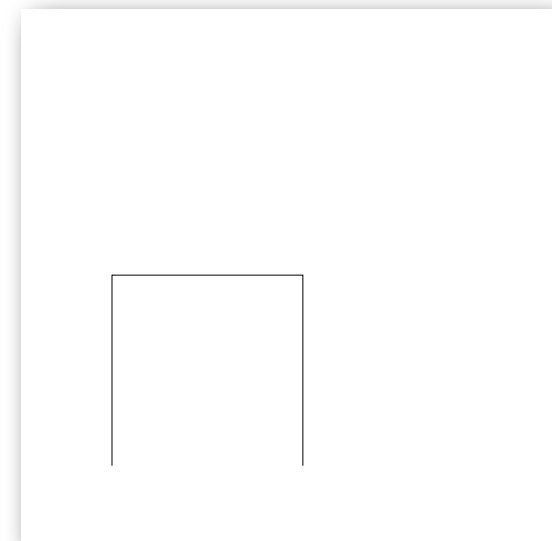
units are points
(72 per inch)

```
%!  
100 100 moveto  
100 300 lineto  
300 300 lineto  
300 100 lineto  
stroke
```

a PostScript program

define a path

draw the path



its output

Simple virtual machine, but not a toy.

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



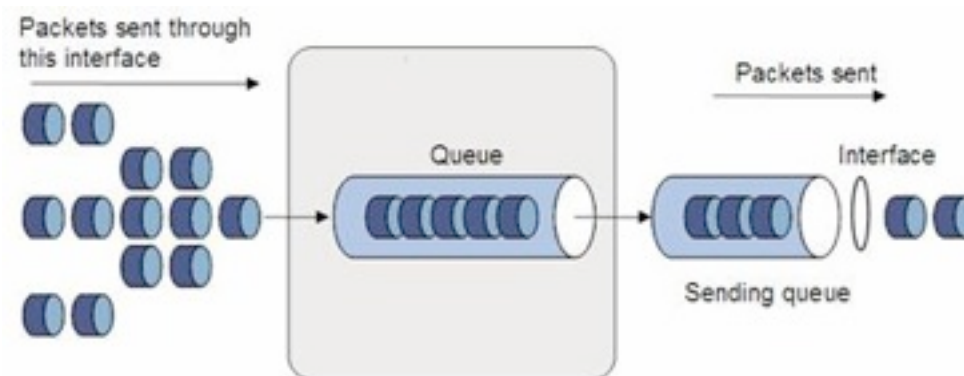
Queue applications

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.

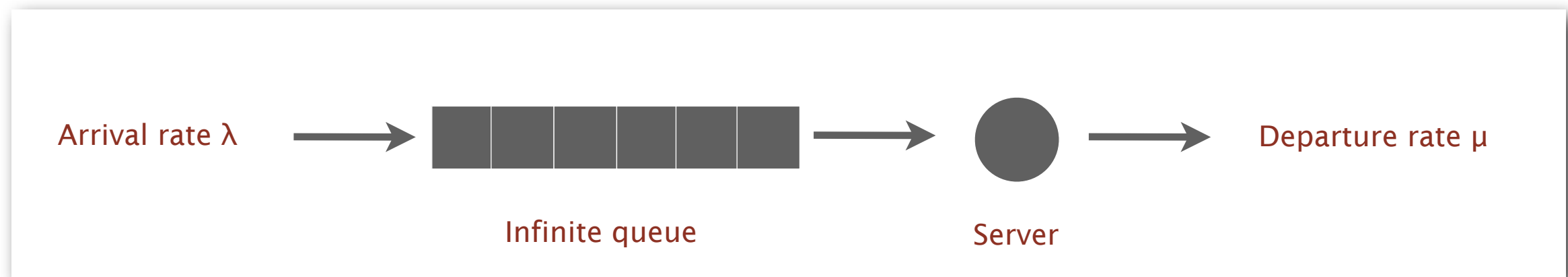


M/M/1 queuing model

M/M/1 queue.

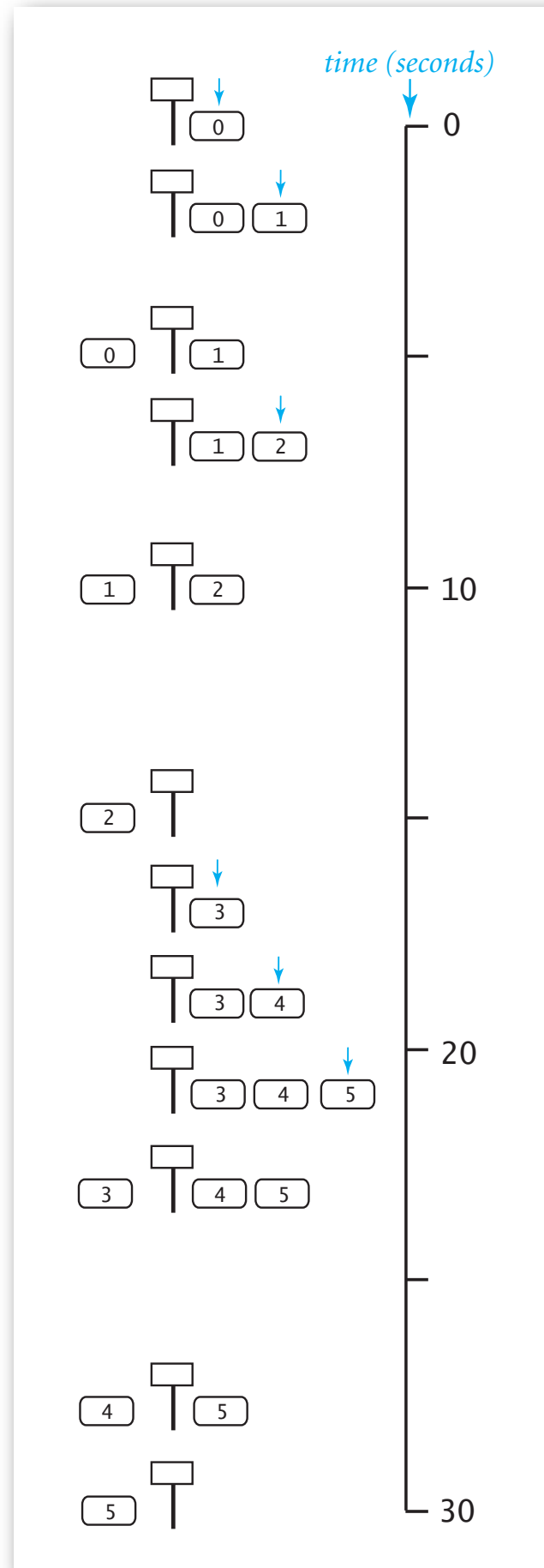
- Customers arrive according to **Poisson process** at rate of λ per minute.
- Customers are serviced with rate of μ per minute.

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



	<i>arrival</i>	<i>departure</i>	<i>wait</i>
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

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);

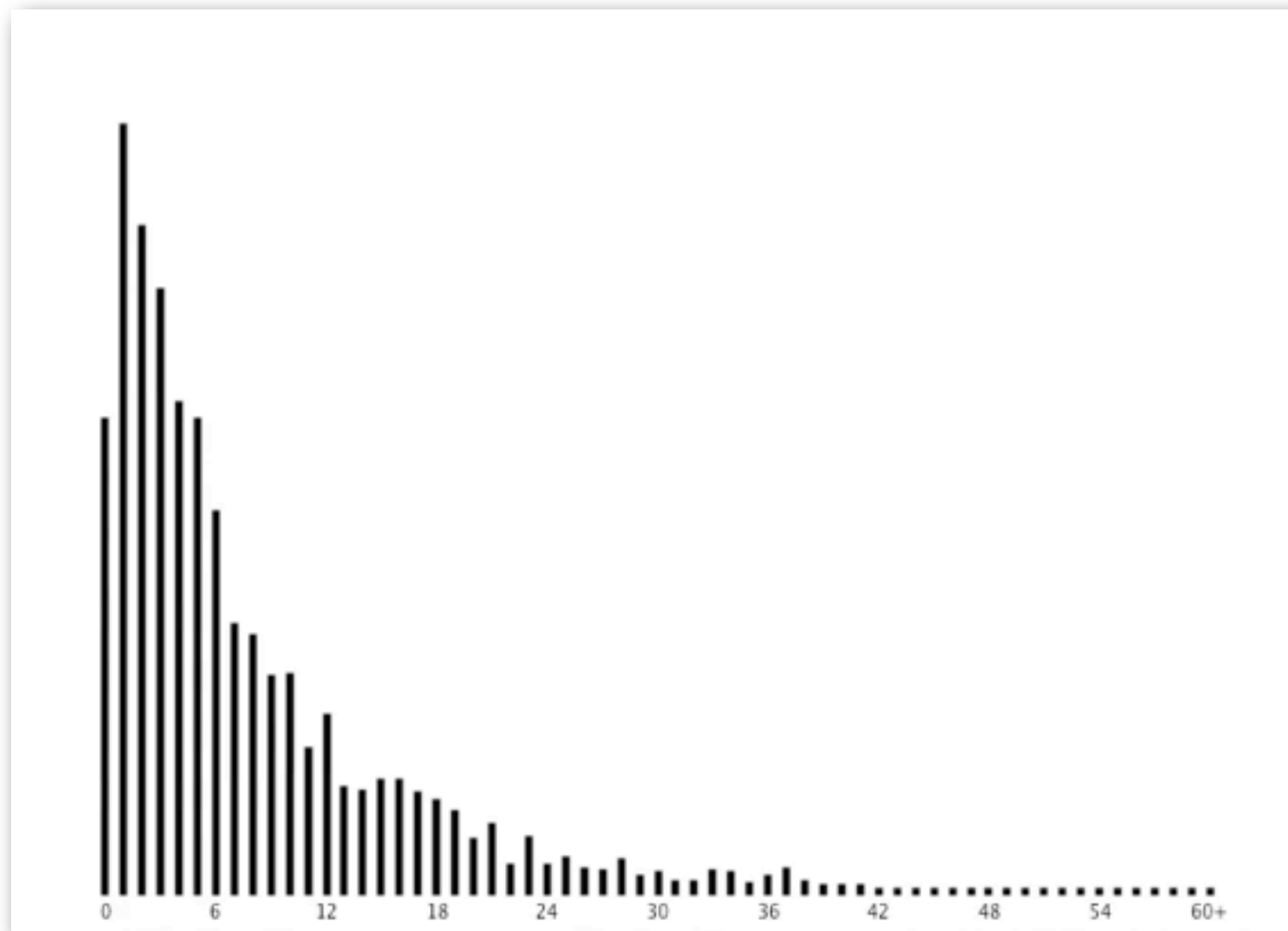
        while (true)
        {
            while (nextArrival < nextService)            next event is an arrival
            {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            double arrival = queue.dequeue();              next event is a service
            double wait = nextService - arrival;          completion
            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 μ is much larger than arrival rate λ , customers gets good service.

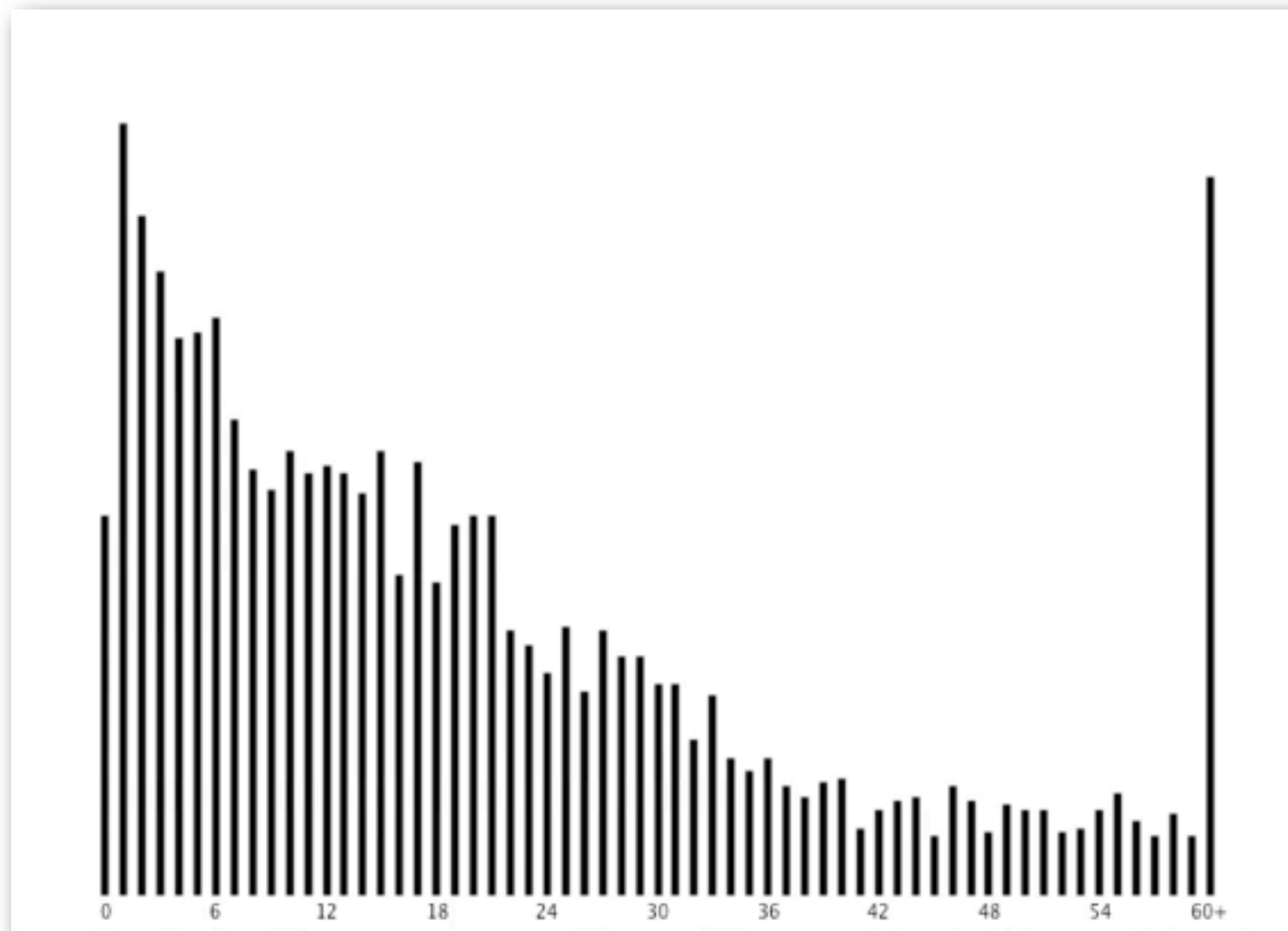
```
% java MM1Queue .2 .333
```



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ , services goes to h^{**} .

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



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ , services goes to h^{**} .

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



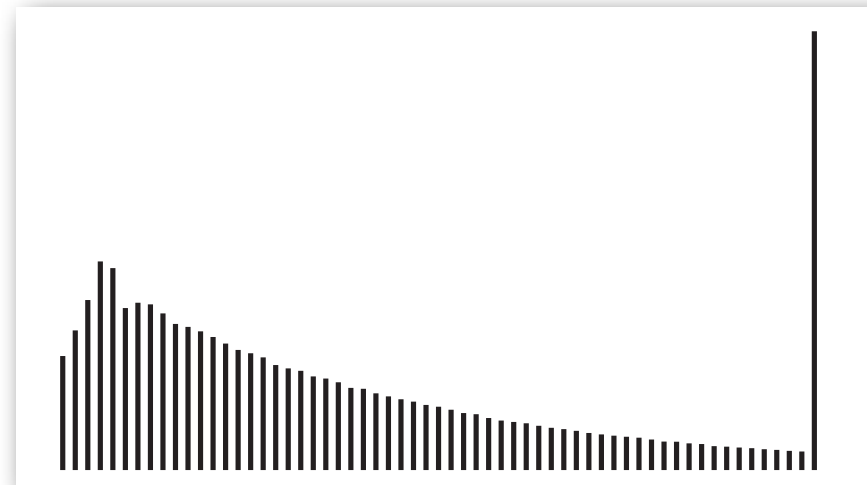
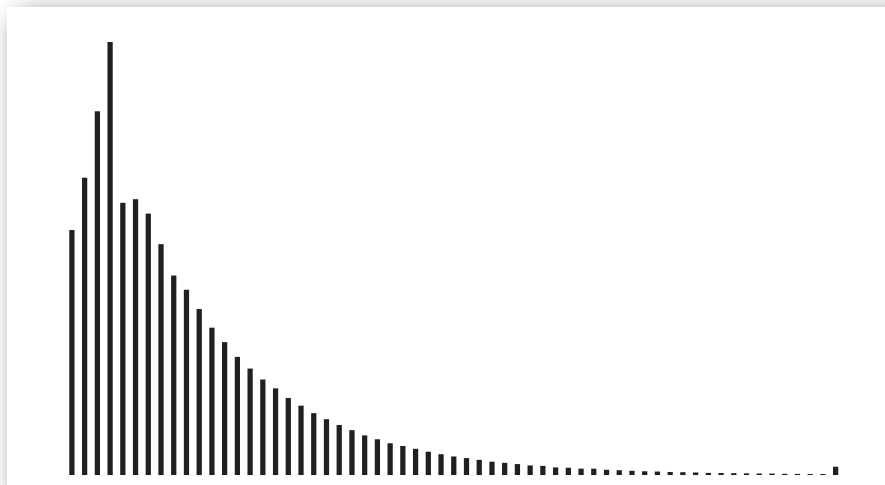
M/M/1 queuing model: analysis

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!