# CSCI 136
# Data Structures &
# Advanced Programming

Lecture 15

Fall 2019

Instructor: B&S

# Announcements

- Mid-Term Review Session
  - Monday, Oct. 14 from 9:00-11:00 am
  - No prepared remarks, so bring questions!
- Mid-term exam is Wednesday, October16
  - During your normal lab session
  - You'll have 1 hour & 45 minutes (if you come on time!)
  - Closed-book
  - Covers Chapters 1-7 & 9 and all topics up through Linked Lists
  - A "sample" mid-term and study sheet are available online
    - See Handouts & Problem Sets

# Last Time : Linear Structures

- Stack applications
  - Arithmetic Expressions
  - Postscript
  - Mazerunning (Depth-First-Search)

# Today: Linear Structures

- Stacks
  - (Implicit) program call stack
- Queues
  - Implementations Details
  - Applications
- Iterators

# Recursive "Pseudo-Code" Sketch

*Boolean RecSolve(Maze* m, *Position* current*)*

*If* (current *eqauls* finish*) return* true

*Mark* current *as visited*

next ← *some unvisited neighbor of* current *(or* null *if none left)*

*While* (next *does not equal* null *&& recSolve(m, next) is* false)

next ← *some unvisited neighbor of* current*(or* null *if none left)*

*Return* next ≠ null

- To solve maze, call: *Boolean recSolve(*m, start*)*
- To prove correct: Induction on distance from *current* to *finish*
- How could we generate the actual solution?

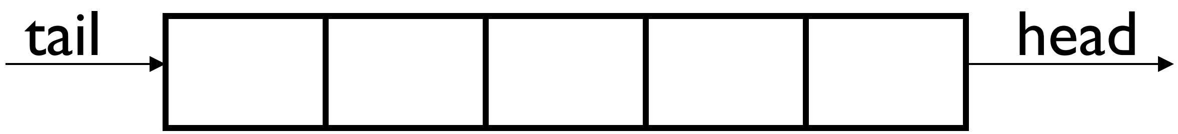
# Method Call Stacks

- In JVM, need to keep track of method calls
- JVM maintains stack of method invocations (called frames)
- Stack of frames
  - Receiver object, parameters, local variables
- On method call
  - Push new frame, fill in parameters, run code
- Exceptions print out stack
- Example: StackEx.java
- Recursive calls recurse too far: StackOverflowException
  - Overflow.java

# Stacks vs. Queues

- Stacks are LIFO (Last In First Out)
  - Methods: push, pop, peek, empty
  - Sample Uses:
    - Evaluating expressions (postfix)
    - Solving mazes
    - Evaluating postscript
    - JVM method calls
- Queues are FIFO (First In First Out)
  - Another linear data structure (implements Linear interface)
  - Queue interface methods: enqueue (add), dequeue (remove), getFirst (get), peek (get)

# Queues




- Examples:
  - Lines at movie theater, grocery store, etc
  - OS event queue (keeps keystrokes, mouse clicks, etc, in order)
  - Printers
  - Routing network traffic (more on this later)

# Queue Interface

```
public interface Queue<E> extends Linear<E> {
  public void enqueue(E item);
  public E dequeue();
  public E getFirst(); //value not removed
  public E peek();  //same as get()
}
```

# Implementing Queues

As with Stacks, we have three options:

QueueArray

```
class QueueArray<E> implements Queue<E> {
protected Object[] data; //can’t declare E[]
int head;
int count; // better than storing tail...
}
```

QueueVector

```
class QueueVector<E> implements Queue<E> {
protected Vector<E> data;
}
```

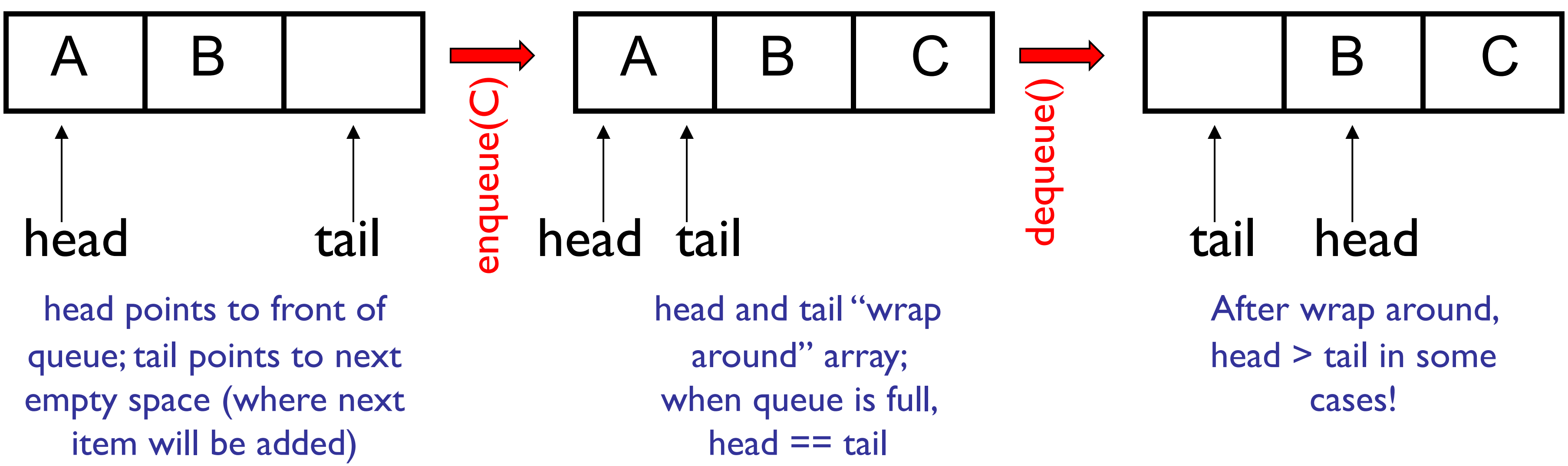
QueueList

```
class QueueList<E> implements Queue<E> {
protected List<E> data; //uses a CircularList
}
```

All three of these also extend AbstractQueue

# QueueArray

- Let's look at an example…
- How to implement?
  - enqueue(item), dequeue(), size()



```
public class queueArray<E> {

    protected Object[] data;      // Must use object because...
    protected int head;
    protected int count;

   public queueArray(int size) {
        data = new Object[size];  // ... can’t say “new E[size]”
   }

   public void enqueue(E item) {
       Assert.pre(count<data.length,”Queue is full.");
       int tail = (head + count) % data.length;
       data[tail] = item;
       count++;
   }

   public E dequeue() {
        Assert.pre(count>0,"The queue is empty.");
        E value = (E)data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        return value;
   }

    public boolean empty() {
        return count>0;
    }
```

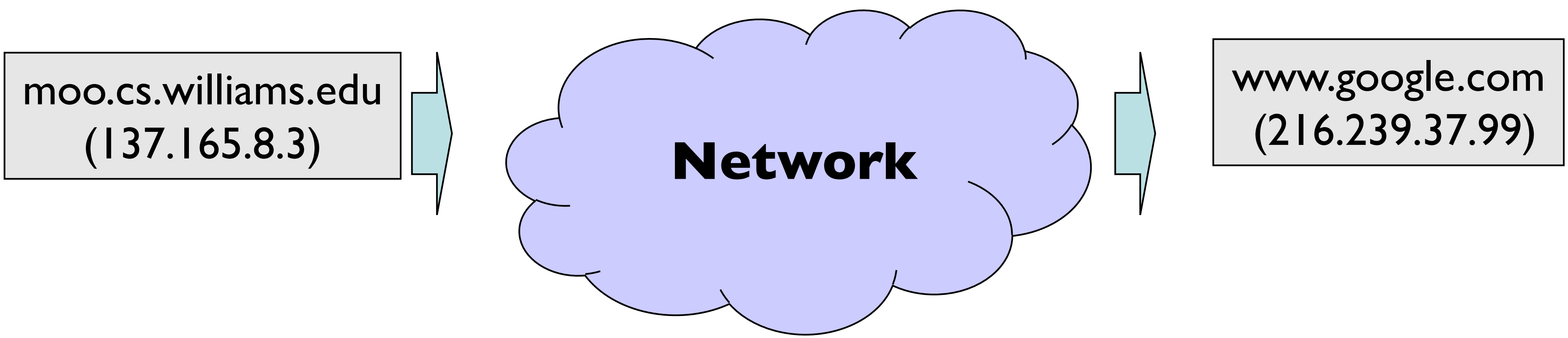
# Tradeoffs:

- QueueArray:
  - enqueue is O(1)
  - dequeue is O(1)
  - Faster operations, but limited size
- QueueVector:
  - enqueue is O(1) (but O(n) in worst case - ensureCapacity)
  - dequeue is O(n)
- QueueList:
  - enqueue is O(1) (addLast)
  - dequeue is O(1) (CLL removeFirst)

# Routing With Queues

Slides by Stephen Freund

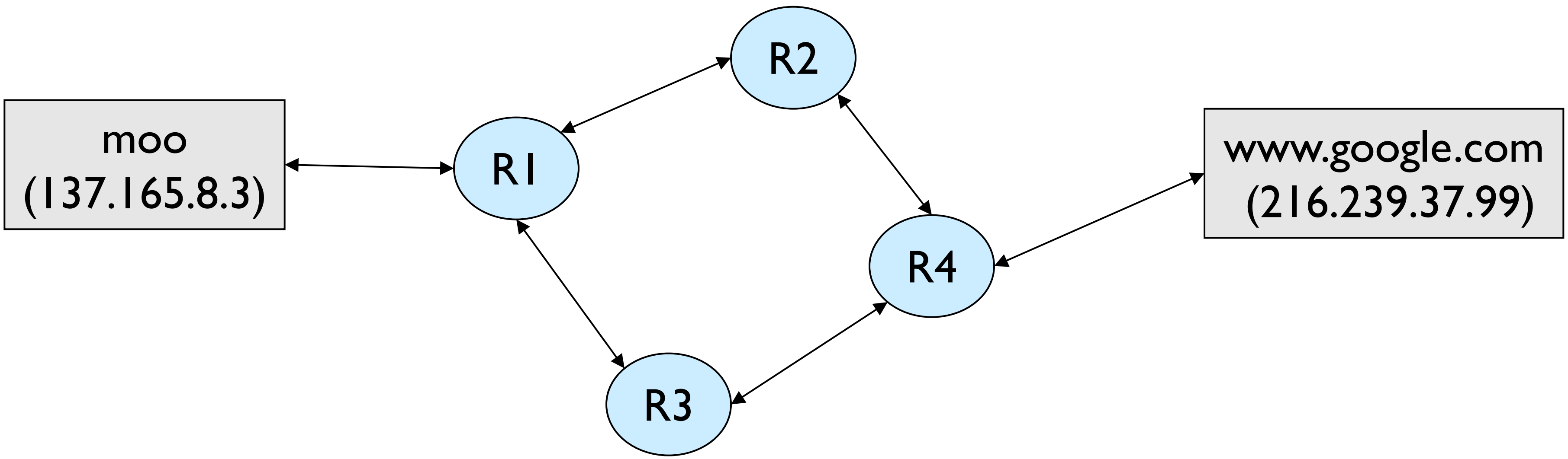
# The Network




Message:

| | | |
|---|---|---|
| 137.165.8.3 | 216.239.37.99 | "Search for ..." |

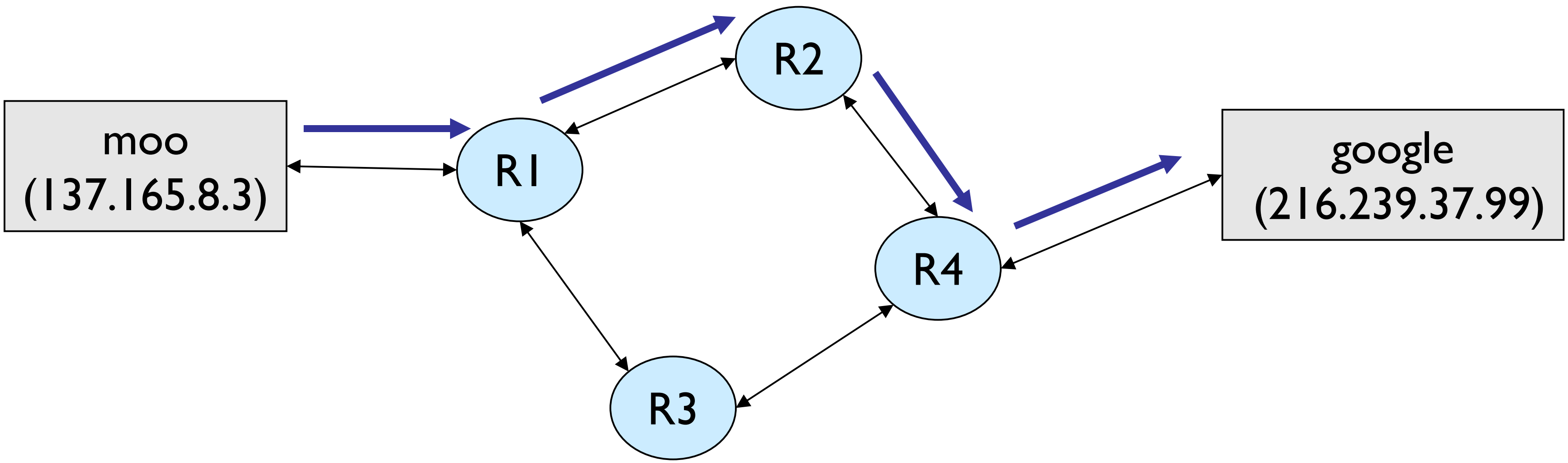
# Routers




Message:

| | | |
|---|---|---|
| 137.165.8.3 | 216.239.37.99 | "Search for ..." |

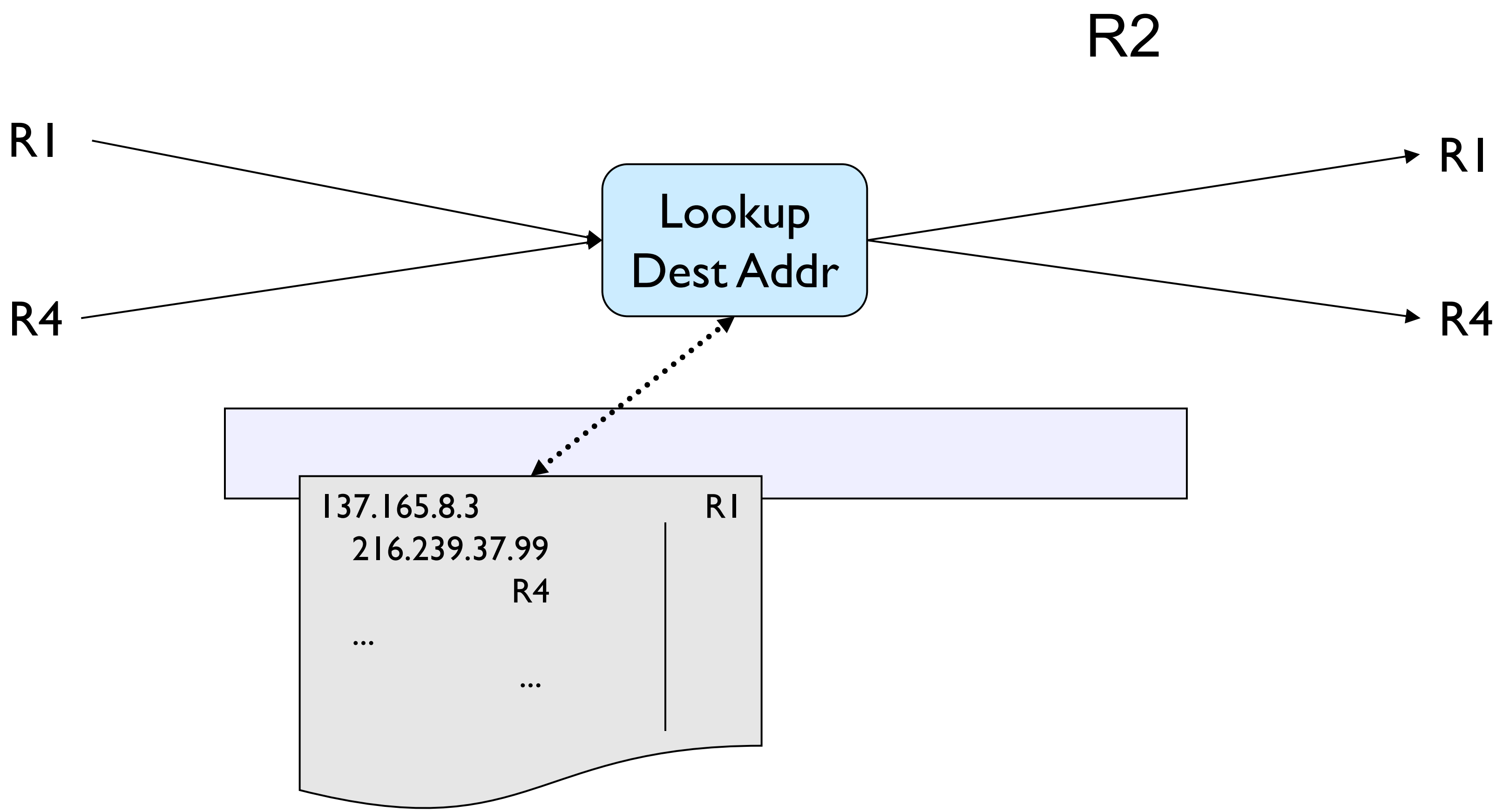
# Routers




Routing Algorithm

1. Receive message
2. Look up Destination Address
   a) Deliver message to Dest
   b) Forward to next Router

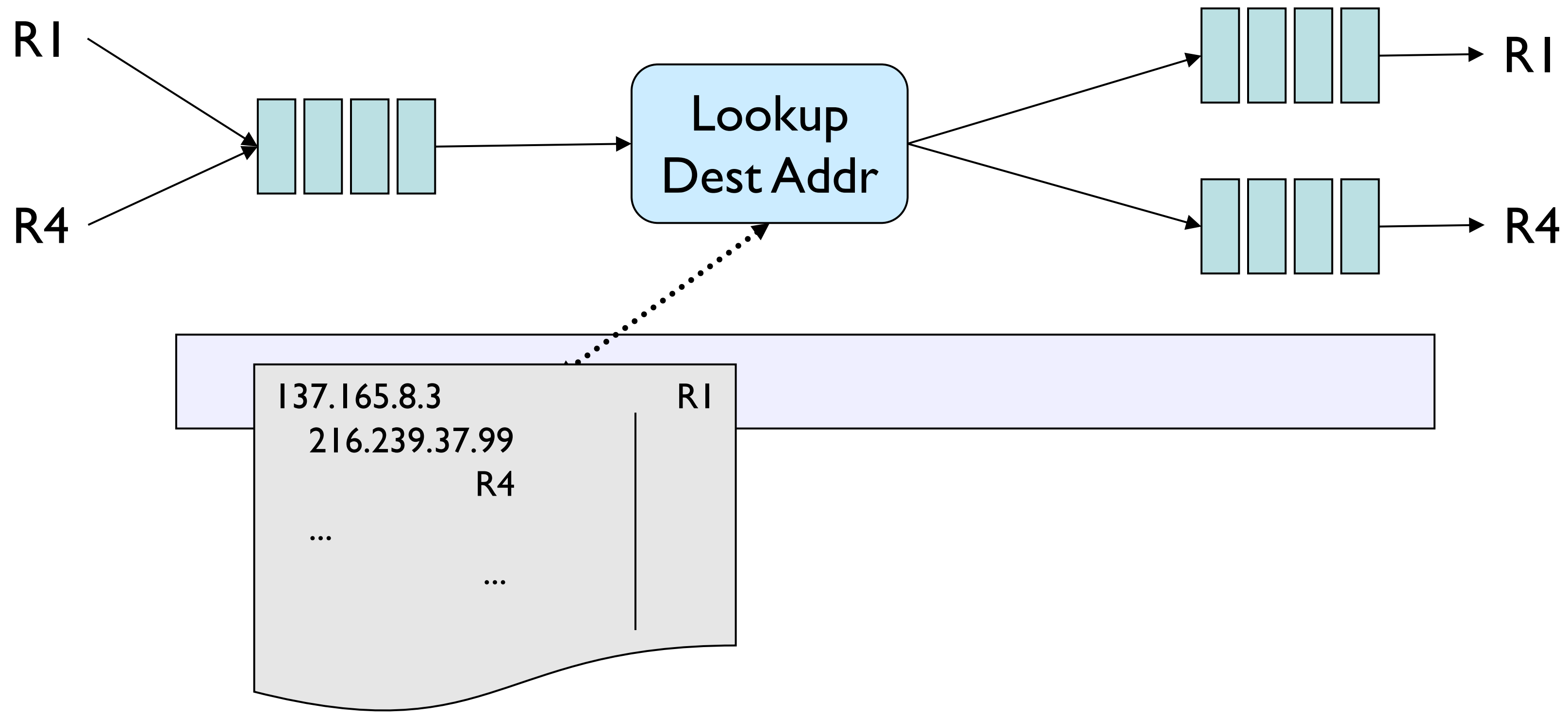
# Router Internals



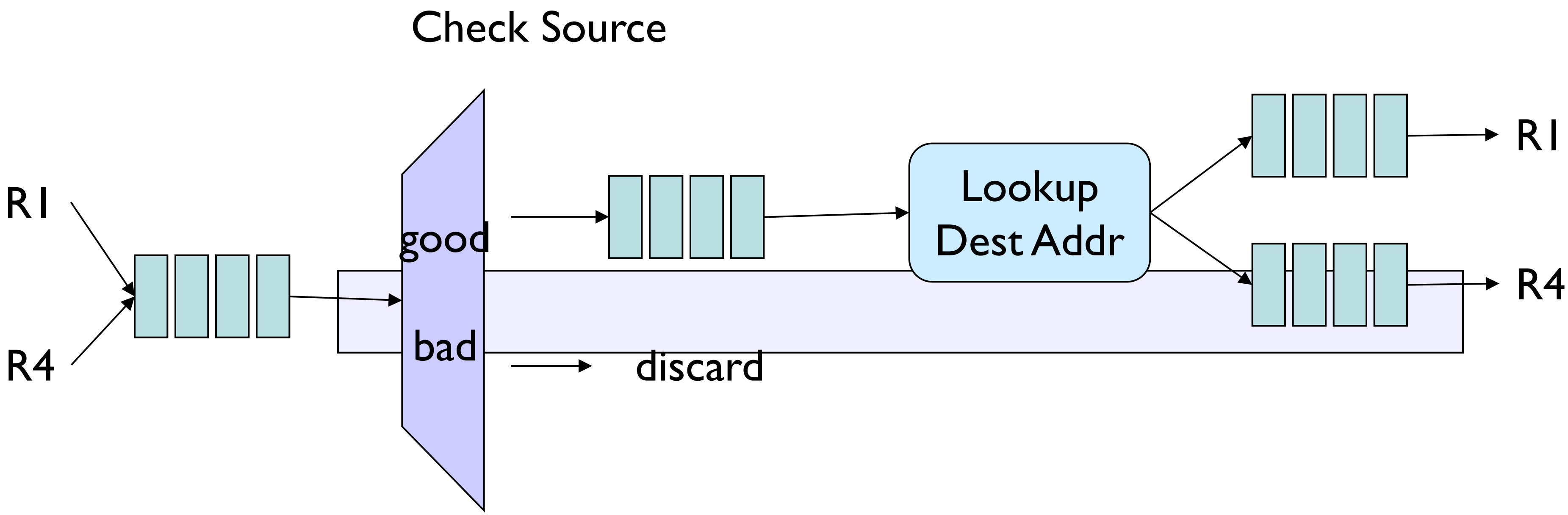
# Buffering Messages

- There may be delays
  - Router receives messages faster than it can process and send
  - Some links are slower than others
    - Common speeds: 10 Mbs, 100Mbs, 1Gbs.
    - Wireless, satellite, infra-red, telephone line, ...
  - Hardware problems
- Want to be able to handle short-term congestion problems

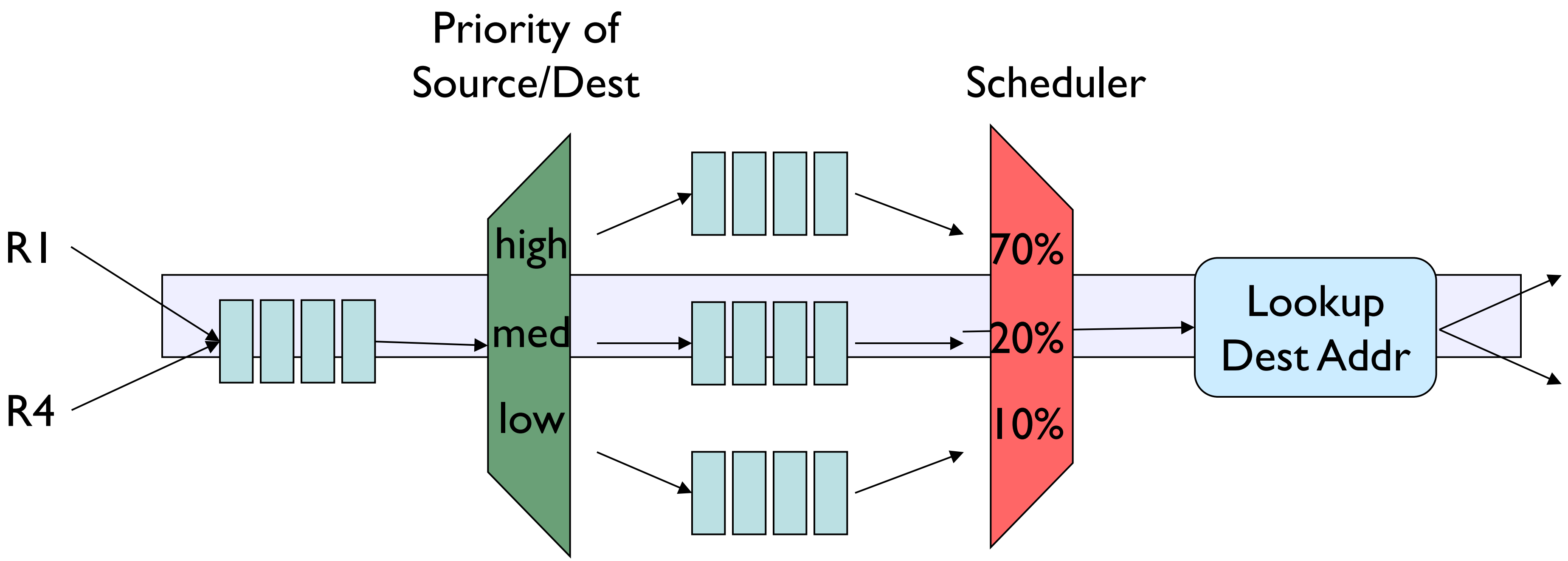
# Router Internals



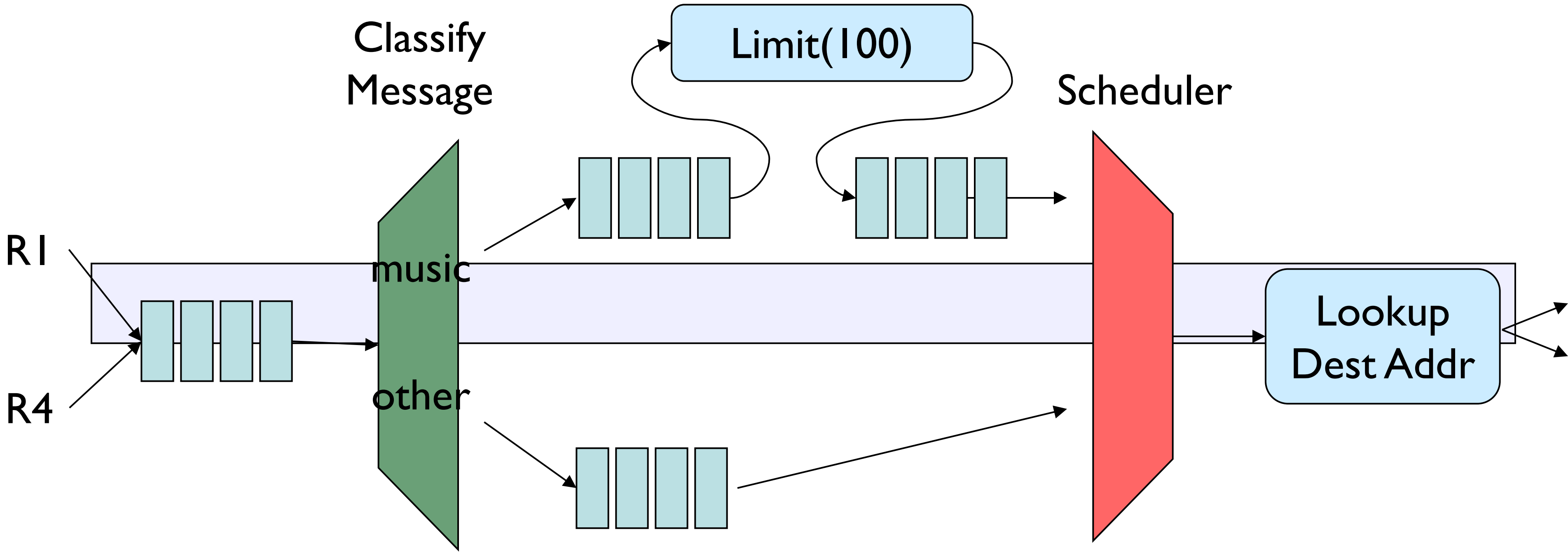
# Firewalls



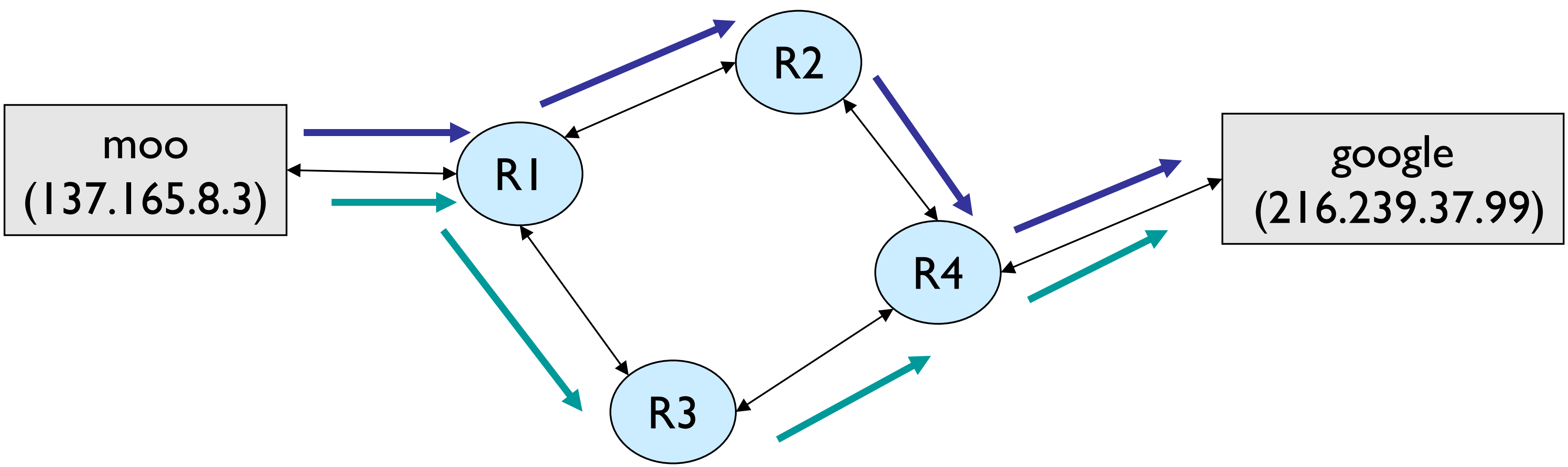
# Priority Scheduling



# Bandwidth Shaper



# Choosing The Best Route



# Choosing Routes

- Routers exchange information periodically
  - Attempt to route on "best" path to destination
  - Not easy to determine:
    - Network congestion varies (evening vs. morning)
    - Hardware added/removed or failures
- Dijkstra's algorithm (later)

# Visiting Data from a Structure

- Write a method (numOccurs) that counts the number of times a particular Object appears in a structure

```
public int numOccurs (List data, E o) {
     int count = 0;
     for (int i=0; i<data.size(); i++) {
          E obj = data.get(i);
          if (obj.equals(o)) count++;
     }
     return count;
}
```

- Does this work on all structures (that we have studied so far)?

# Problems

- get() not defined on Linear structures (i.e., stacks and queues)
- get() is "slow" on some structures
  - O(n) on SLL (and DLL)
  - So numOccurs = $O(n^2)$ for linked lists
- How do we traverse data in structures in a general, efficient way?
  - Goal: data structure-specific for efficiency
  - Goal: use same interface to make general

# Recall : Structure Operations

- `size()`
- `isEmpty()`
- `add()`
- `remove()`
- `clear()`
- `contains()`

- But also
  - Method for efficient data traversal
    - `iterator()`

# Iterators

- **Iterators** provide support for *efficiently* visiting all elements of a data structure
- An Iterator:
  - Provides generic methods to dispense values
    - Traversal of elements : *Iteration*
    - Production of values : *Generation*
  - Abstracts away details of how elements are retrieved
  - Uses different implementations for each structure

```
public interface Iterator<E> {
   boolean hasNext() – are there more elements in iteration?
   E next() – return next element
   default void remove() – removes most recently returned value
```

- Default : Java provides an implementation for remove
  - It throws an UnsupportedOperationException exception

# A Simple Iterator

- Example: FibonacciNumbers

```
public class FibonacciNumbers implements Iterator<Integer> {
    private int next= 1, current = 1;
    private int length= 10;  // Default

    public FibonacciNumbers() {}
    public FibonacciNumbers(int n) {length= n;}
    public boolean hasNext() { return length>=0;}
    public Integer next() {
            length--;
            int temp = current;
            current = next;
            next = temp + current;
            return temp;
    }
}
```

# Why Is This Cool? (it is)

- We could calculate the $i^{th}$ Fibonacci number each time, but that would be slow
  - Observation: to find the $n^{th}$ Fib number, we calculate the previous n-1 Fib numbers…
  - But by storing some state, we can easily generate the next Fib number in O(1) time
- Knowledge about the structure of the problem helps us traverse the Fib space *efficiently* one element at a time
  - Let's do the same for data structures

# Iterators Of Structures

Goal: Have data structures produce iterators that return the values of the structure in some order.

How?

- Define an iterator class for the structure, e.g.

```
public class VectorIterator<E>
        implements Iterator<E>;
public class SinglyLinkedListIterator<E>
        implements Iterator<E>;
```

- Provide a method in the structure that returns an iterator

```
public Iterator<E> iterator(){ … }
```

# Iterators Of Structures

The details of hasNext() and next() depend on the specific data structure, e.g.

- VectorIterator holds an array reference and index of next element
  - A reference to the data array of the Vector
  - The index of the next element whose value to return
- SinglyLinkedListIterator holds
  - a reference to the head of the list
  - A reference to the next node whose value to return

# Iterator Use : numOccurs

```
public int numOccurs (List<E> data, E o) {
      int count = 0;
      Iterator<E> iter = data.iterator();
      while (iter.hasNext())
            if(o.equals(iter.next())) count++;
      return count;
}
// Or...

public int numOccurs (List<E> data, E o) {
      int count = 0;
      for(Iterator<E> i = data.iterator());
      i.hasNext();)
            if(o.equals(i.next())) count++;
      return count;
}
```

# Implementation Details

- We use both an Iterator interface and an AbstractIterator class
- All concrete classes in structure5 extend AbstractIterator
  - AbstractIterator partially implements Iterator
- Importantly, AbstractIterator *adds* two methods
  - get() – peek at (but don't take) next element, and
  - reset() – reinitialize iterator for reuse
- Methods are specialized for each data structure

# Iterator Use : numOccurs

Using an AbstractIterator allows more flexible coding
(but requiring a cast to AbstractIterator)

Note: It has the form of a standard 3-part for statement

```
public int numOccurs (List<E> data, E o) {
      int count = 0;
      for(AbstractIterator<E> i =
            (AbstractIterator<E>) data.iterator();
                  i.hasNext(); i.next())
            if(o.equals(i.get())) count++;
      return count;
}
```

# Implementation : SLLIterator

```
public class SinglyLinkedListIterator<E> extends AbstractIterator<E> {

    protected Node<E> head, current;

    public SinglyLinkedListIterator(Node<E> head) {
         this.head = head;
         reset();
    }

    public void reset() { current = head;}

    public E next() {
         E value = current.value();
         current = current.next();
         return value;
    }

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

    public E get() { return current.value(); }
}
```

## In SinglyLinkedList.java:

```
public Iterator<E> iterator() {
     return new SinglyLinkedListIterator<E>(head);
}
```

# More Iterator Examples

- How would we implement VectorIterator?
- How about StackArrayIterator?
  - Do we go from bottom to top, or top to bottom?
  - Doesn't matter! We just have to be consistent…

- We can also make "specialized" iterators
  - Another SLL Example: SkipIterator.java
  - ReverseIterator.java

# Iterators and For-Each

Recall: with arrays, we can use a simplified form of the for loop

```
for( E elt : arr) {System.out.println( elt );}
```

Or, for example

```
// return number of times o appears in data
public int numOccurs (E[] data, E o) {
        int count = 0;
        for(E current : data)
                if(o.equals(current)) count++;
        return count;
}
```

We can do this with classes that provide an iterator() method…

# The Iterable Interface

We can use the "for-each" construct…

```
for( E elt : boxOfStuff ) { ... }
```

…as long as `boxOfStuff` implements the *Iterable* interface

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

Since Structure<E> extends Iterable<E>, we can write

```
public int numOccurs (List<E> data, E o) {
        int count = 0;
        for(E current : data)
                if(o.equals(current)) count++;
        return count;
}
```

# General Rules for Iterators

1. Understand order of data structure
2. **Always call hasNext() before calling next()!!!**
3. Use remove with caution!
   1. Don't use remove….
4. Don't add to structure while iterating: TestIterator.java

- Take away messages:
  - Iterator objects capture state of traversal
  - They have access to internal data representations
  - They should be fast and easy to use