# CSCI 136
# Data Structures &
# Advanced Programming

Lecture 16

Fall 2019

Instructor: B&S

# Last Time: Queues

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

# This Time: Iterators & Ordered Structures

- Iterators
  - The goal: Efficient and uniform dispensing of values from data structures
  - The solution: The Iterator interface
  - Iterators as dispensers and as generators
  - Iterating over Iterators
  - The Iterable interface : Using iterators with for-each
- Ordered Structures : Introduction

# 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 for
    - Traversal of elements : *Iteration*
    - Production of values : *Generation*
  - Abstracts away details of how to access elements
  - 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(){ … }
```

  - Not needed for Vector, but will be for other structures….

# 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 the Iterator interface and the AbstractIterator class
- All concrete implementations 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 specific data structures

# Iterator Use : numOccurs

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

Note: Can now write 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?
  - Do we store the Vector or the underlying array?
- 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
  - SkipIterator.java
    - next() post-work: skip elts until new next found
  - ReverseIterator.java
    - A massive cheat!
  - EvenFib.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 (List<E> data, E o) {
      int count = 0;
      for(E current : data)
            if(o.equals(current)) count++;
      return count;
}
```

Why did that work?!

List provides an iterator() method and…

# 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();
```

Duane's Structure interface extends Iterable, so we can use it:

```
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. [Opinion: 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

# Ordered Structures

- Until now, we have not required a specific ordering to the data stored in our structures
  - If we wanted the data ordered/sorted, we had to do it ourselves
- We often want to keep data ordered
  - Allows for faster searching
  - Easier data mining - easy to find best, worst, and median values, as well as rank (relative position)

# Ordering Structures

- The key to establishing order is being able to compare objects
- We already know how to compare two objects…how?
- Comparators and `compare(T a, T b)`
- Comparable interface and `compareTo(T that)`
- Two means to an end: which should we use?

BOTH!

# Ordered Vectors

- We want to create a Vector that is always sorted
  - When new elements are added, they are inserted into correct position
  - We still need the standard set of Vector methods
    - add, remove, contains, size, iterator, …
- Two choices
  - Extend Vector (as we did in sorting lab)
  - Create new class
    - Allows for more focused interface
    - Can have a Vector as an instance variable
    - Avoid corrupting order by controlled access to Vector
- We will implement a new class (OrderedVector)
  - Start with Comparables
  - Generalize to use Comparators instead of Comparables

# OrderedVector Methods

```
public class OrderedVector<E extends Comparable<E>>
  implements OrderedStructure<E> {
   protected Vector<E> data;

   public OrderedVector() {
        data = new Vector<E>();
   }

   public void add(E value) {
        int pos = locate(value);
        data.add(pos, value);
   }

   protected int locate(E value) {
   //use modified binary search to find position of value
   //if not found, returns position where add should occur
   //uses iterative version of binary search (see text)
   }
```

# OrderedVector Methods

```
public boolean contains(E value) {
    int pos = locate(value);
    return pos < size() && data.get(pos).equals(value);
}

public Object remove (E value) {
    if (contains(value)) {
        int pos = locate(value);
        return data.remove(pos);
    }
    else return null;
}
```

Performance:

- add - O(n)
- contains - O(log n)
- remove - O(n)

# Adding Flexibility with Comparators

- We would like to be able to allow ordered structures to use different orders
- Idea: Add constructor that has a Comparator parameter
- Q: How does structure know whether to use the Comparator or the Comparable ordering?
- A: The NaturalComparator class....

# An Aside: Natural Comparators

- NaturalComparators bridge the gap between Comparators and Comparables

```
class NaturalComparator<E extends Comparable<E>>
implements Comparator<E> {
    public int compare(E a, E b) {
        return a.compareTo(b);
    }
}
```

# Generalizing OrderedVector

```
public class OrderedVector<E extends Comparable<E>>
    implements OrderedStructure<E> {
   protected Vector<E> data;
   protected Comparator<E> comp;

   public OrderedVector() {
        data = new Vector<E>();
        this.comp = new NaturalComparator<E>();
   }

   public OrderedVector(Comparator<E> comp) {
        data = new Vector<E>();
        this.comp = comp;
   }

   protected int locate(E value) {
        //use modified binary search to find position of value
        //return position
        //use comp.compare instead of compareTo
   }

   //rest stays same…
```

# Ordered Lists

- Similar to OrderedVector
- Can't easily use SinglyLinkedList like OrderedVector used Vector (Why?)
- So, we just build a SinglyLinkedList-like structure
- add, contains, remove runtime?
  - All O(n)…why?

# OrderedList Methods

```
public class OrderedList<E extends Comparable<E>>
    extends AbstractStructure<E> implements
OrderedStructure<E> {

    protected Node<E> data; // smallest value
    protected int count;    // size of list
    protected Comparator<? super E> ordering;

    public OrderedList() {
        this(new NaturalComparator<E>());
    }
    public OrderedList(Comparator<? super E> ordering){
        this.ordering = ordering;
        clear();
    }
```

# OrderedList Methods

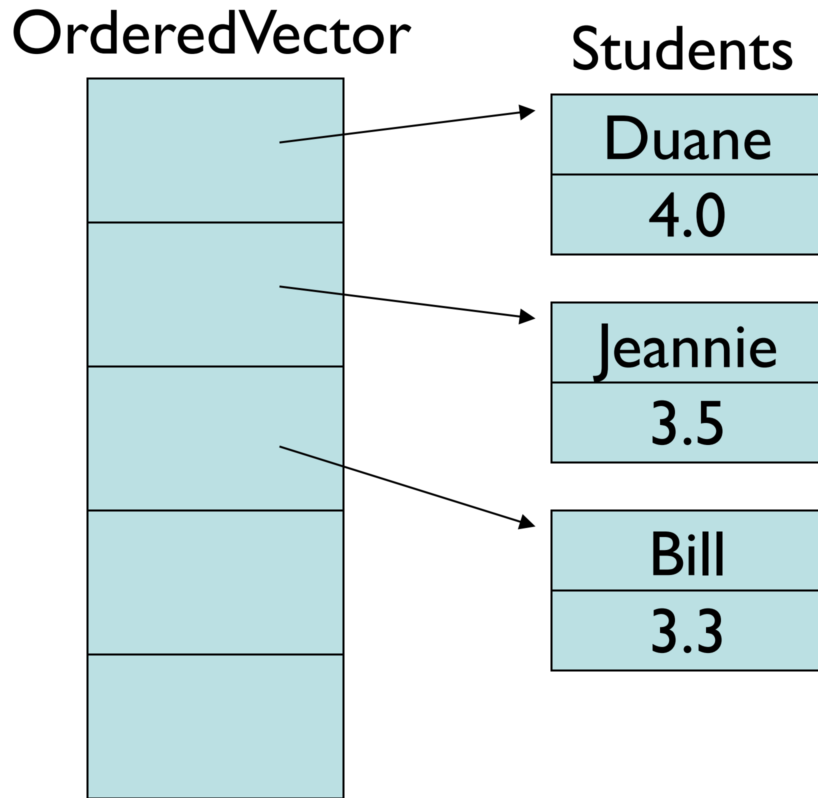
```
public void clear() {
      data = null;
      count = 0;
}
public boolean contains(E value) {
   Node<E> finger = data; // target

   while ((finger != null) &&
      ordering.compare(finger.value(),value)<0)

      finger = finger.next();

  return finger!=null && value.equals(finger.value());
}
```

# What Could Go Wrong?




- Students compared to each other by GPA
- Suppose next semester I get a 3.7 and Jeannie gets a 3.3

# What’s the problem?

- We have to recompute GPAs each semester
- What happens if the values are allowed to change?
- We may need to resort vector
  - But since this isn’t part of the interface, it may be forgotten
- Options:
  - Avoid changing values in OrderedStructures
  - Incorporate an update method that repositions element
  - Incorporate a resort method
    - This invites adding a “setComparator” method....
  - Always update a value by removing and re-adding

# Type Safety & Generic Types

- Question: Since String extends Object, does List<String> extend List<Object>?
  - I.e., can I say List<Object> = new List<String>()?
- No. It would compromise the type system:

```
List<String> slist = new List<String>();
List<Object> olist = slist;    // If this were possible
olist.add(new Object());       // This would be bad!
```

- It generates a compiler error.
- On the other hand…

```
String[] sa = {“I”, “love”, “java”, “!”};
Object[] oa = sa;
oa[1] = new Object()); // This would be bad!
```

- …actually compiles
  - But causes a run-time error!