CSCI 136
Data Structures &
Advanced Programming

Lecture 16
Fall 2018
Instructor: Bills
Last Time: Queues & Iterators

• Queues: Implementations Recap
• Queues: Applications
• Iterators: Preview
This Time: Iterators & Ordered Structures

- Iterators Continued
- Iterating over Iterators
- Ordered Structures
  - OrderedVector
  - OrderedList
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.

```java
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

```java
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
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

```java
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;
}
```
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
  • 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

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

Or, for example

```java
// 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;
}
```

Why did that work?!
List provides an iterator() method and…
The Iterable Interface

We can use the “for-each” construct...

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

...as long as `boxOfStuff` implements the `Iterable` interface

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

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

```java
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!
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