CSCI 136
Data Structures & Advanced Programming

Lecture 17

Fall 2017

Instructor: Bills

Administrative Details

- Lab 7 is now available
  - No partners this week
  - Review before lab; come to lab with design doc
  - Check out the javadoc pages for the 3 provided classes
    - Token – A wrapper for semantic PS elements,
    - Reader – An iterator to produce a stream of Tokens from standard input or a List of Tokens,
    - SymbolTable – A dictionary with String keys and Token values: For user-defined names

Last Time: Queues & Iterators

- Queues: Implementations Recap
- Queues: Applications
- Iterators

This Time: Iterators & Ordered Structures

- Iterators Recap
- 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

```
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

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 specific 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;
}
```

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!

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;
}
```

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!
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

Lab 7: PostScript Interpreter

- PostScript is a *stack-based* programming language
  - designed for vector graphics & printing
- Lab 7: Implement a small portion of a PS interpreter
  - Read a stream of "tokens"
  - Evaluate expressions using a stack
  - Allow for creation of variables (and procedures!) using a symbol table
- Provided:
  - Reader, Token, and SymbolTable class
  - You write an interpreter class
- Try out GhostScript: unix command: gs
  - It will pop up a graphics window – ignore it

Lab 7: Concept Overview

- Basic input unit: the *token*: There are multiple types
  - Number, Boolean, Symbol, Procedure (sorry, no Strings)
  - Implemented with class Token
- A PostScript program is a sequence of tokens
  - Tokens are processed as received
    - Numbers, booleans, procedures go on stack
    - A symbol should
      - Be put on stack (if preceded by /), or
      - Cause an operation to be performed if it is a built-in symbol (add, pstack, …), or
      - Cause its value to be looked up in symbol table and appropriate action taken
  - The SymbolTable class provides a symbol table
  - The Reader class provides in iterator for producing a stream of tokens
    - Stream can come from standard input, a single Token, or a List of Tokens
- Your job: Write code to carry out the processing
  - Driven by a method (you write) *interpret(Reader r)*

Lab 7: Suggested Approach

1. Read Lab handout and description in text carefully
2. Read the Javadoc pages for the 3 provided classes:
   Using these classes well will help you a great deal!
3. Develop a plan. Here are some starting steps
   1. Write your interpret method so that it just reads a token stream from standard input and prints out each token.
   2. Handle numbers, booleans, and pstack/pop operators
   3. Follow the steps in the text in order
4. Debug as you go, use gs program to clarify expected behavior

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
- 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…
```