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
- Iterator motivation and Iterator API

This Time: Iterators & Ordered Structures

- Iterator Recap
- Iterator Implementations
- 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;
}
```

No increment step because i.next() "**consumes**" an element

Implementation Details

- We use both an Iterator interface and an 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 for 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;
}
```

`Iterator`'s **`next()`** consumes a value. To reuse that value, either create a temporary variable, or use `AbstractIterator`'s **`get()`**

Implementation : SLLIterator

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

    protected Node<E> head;
    protected Node<E> 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 use this syntax with classes that provide an iterator() method because…

The Iterable Interface

We can use the "for-each" construct…

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

…as long as `stuff` 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
- You are Provided:
  - `Reader`, `Token`, and `SymbolTable` classes
  - You write an `Interpreter` class
- Try out GhostScript: (Unix command: gs)
  - It will pop up a graphics window – ignore that window

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 that 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, average, and median values

Ordering Structures

- The key to establishing order is being able to compare objects and rank them
- 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!

OrderedStructure Interface

```
public interface OrderedStructure<K extends
                                  Comparable<K>>
          extends Structure<K>
{
}
```

- Recall: a `Structure` supports adding and removing elements, and membership checks
- An `OrderedStructure` is a `Structure` that stores `Comparable` elements
- We have the API we want, and the "sortability" we want

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