CSCI 136
Data Structures & Advanced Programming

Iterators

Iterators : Dispensing Data

Iterators

- Iterators
  - The problem: Efficient and uniform dispensing of values from data structures
  - The solution: The Iterator interface
    - Iterators as dispensers
    - Iterators as generators
    - Iterators as filters
  - Iterators that iterate over other Iterators ?!
    - Yep, it's a thing
  - Iterators and for loops: The Iterable interface
    - Allows use of iterators with for-each

Visiting Data from a Structure

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

```
public int count(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(int) not defined on Linear structures (i.e., stacks and queues)
- get(int) is "slow" on some structures
  - O(n) on SLL (and DLL)
  - count() = $O(n^2)$ 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
  - *Provides common methods* to dispense values for
    - Traversal of elements : *Iteration*
    - Production of values : *Generation*
    - Selection of values : *Filtering*
  - *Abstracts away details* of how to access elements
  - *Customizes implementation* based on 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
  - *Even the Java folks are hesitant to remove from a structure during iteration!*

Iterators as *Generators*

- Simple 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

Iterating Over Structures

Goal: Have a data structure produce an iterator 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 data structure that returns an iterator

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

Iterator Example : Counting

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

Iterating Over Structures

Why provide a method in the data structure that returns an iterator?

Why not just pass the data structure to the constructor for the iterator? E.g.

```
public SLLIterator<E>(SLL<E> v) {
      // code to construct the iterator
}
```

From with the data structure, we can access the instance variables of the structure so the we pass access to those variables to the iterator

- We'll see other benefits soon

Iterating Over Structures

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

- SinglyLinkedListIterator holds
  - a reference to the head of the list
  - A reference to the next node whose value to return

But not *always*…

- VectorIterator holds a reference to the Vector and index of next element

Note: The Iterator class for a structure often has *privileged access* to the implementation of the structure.

Technical Detail : AbstractIterators

- We use both the Iterator (java.util) *interface* and the AbstractIterator (structure5) *class*
- All concrete iterator implementations in structure5 *extend* AbstractIterator
  - AbstractIterator *partially implements* Iterator
  - [Aside: *Very* partially]
- 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

AbstractIterator Use : Counting

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

Note: Can now write a ‘standard’ 3-part **for** statement

```
// Only works if data.iterator() returns
// an AbstractIterator!

public int count (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

- Structure5 provides an ArrayIterator
  - It will iterate over the entire array or any slice
- How do we implement a StackArrayIterator?
  - Do we go from bottom to top, or top to bottom?
  - Doesn't matter! We just need to be consistent…
    - Structure5 is *not* consistent!
    - StackArrayIterator starts at bottom, StackListIterator at top!
- We can also make iterators that *filter* the output of other iterators
  - SkipIterator.java : skips over a given value
  - ReverseIterator.java : Dispenses elements in the reverse order given by another iterator
  - EvenFib.java : Only produce even Fibonacci numbers

SkipIterator

Problem: How can we filter out unwanted elements from an iterator Iter?

Solution: Create another iterator that takes Iter as a parameter its constructor and uses that the methods of Iter (with some extra steps)

- The SkipIterator will ensure that the next element that Iter would dispense is *not* the one we want to skip over!

SkipIterator

```
// An iterator that filters out a value from another iterator
public class SkipIterator<E> extends AbstractIterator<E> {

protected AbstractIterator<E> elems;
E value;

public SkipIterator(Iterator<E> iter, E skipMe) {
        elems = (AbstractIterator<E>) iter;
        value = skipMe;
        reset();
}

public E get() { return elems.get(); }

public boolean hasNext() { return elems.hasNext(); }
```

SkipIterator

```
public void reset() {
          elems.reset();
          skip();
}

public E next() {
          E returnVal = elems.next();
          skip();
          return returnVal;
}

private void skip() {
    while(elems.hasNext() && elems.get().equals(value))
elems.next();
}
```

Iterator Hack : ReverseIterator

Problem: How can dispense the elements from an iterator Iter *in the opposite order* from which Iter would dispense them?

Solution: Create another iterator that

- Creates a SinglyLinkedList secretSLL
- Fills it with the elements dispensed by Iter
  - But stores them in reverse order
- Asks secretSLL for an iterator to itself
- Uses that iterator for dispensing values

ReverseIterator

```
// An iterator that reverses the order of elements
// returned from another iterator.

class ReverseIterator<E> extends AbstractIterator<E> {

    protected AbstractIterator<E> elems;

public ReverseIterator(Iterator<E> iter) {
        SinglyLinkedList<E> list = new SinglyLinkedList<E>();
        while (iter.hasNext()) {
            list.addFirst(iter.next());
        }
        elems = (AbstractIterator<E>)list.iterator();
    }
```

ReverseIterator

```
// All other methods dispatch to the underlying iterator.

    public boolean hasNext() { return elems.hasNext(); }

    public void reset() { elems.reset(); }

    public E next() { return elems.next(); }

    public E get() { return elems.get(); }
```

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 count (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 count (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. Take care when adding to structure while iterating

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