CSCI 136
Data Structures & Advanced Programming

Lecture 18
Spring 2018
Profs Bill & Jon
Administrative Details

• Lab 6 is online
  • 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 : Linear Structures

- Linear Interface, AbstractLinear
- Stack Interface, AbstractStack
  - StackArray
  - StackList
  - StackVector
- Queue Interface, AbstractQueue
  - QueueArray
  - QueueVector
  - QueueList
Today: Iterators

• Iterators
  • A general purpose mechanism for efficiently traversing data (structures)
Visiting Data from a Structure

• Take a minute and write a method (numOccurs) that counts the number of times a particular non-null Object appears in a data structure.

```java
public int numOccurs (List<E> data, E o) {
    int count = 0;
    for (int i = 0; i < data.size(); i++) {
        if (o.equals(data.get(i)))
            count++;
    }
    return count;
}
```

• Does this work on all structures (that we have studied so far)?
Problems

• \texttt{get(i)} not defined on Linear structures (i.e., stacks and queues)
• \texttt{get(i)} is “slow” on some structures
  • \(O(n)\) on SLL (and DLL)
  • So \(\text{numOccurs(data, o)}\) is \(O(n^2)\)
• How \textit{should} we traverse data in structures?
  • Goal 1: data structure-specific for \textit{efficiency}
  • Goal 2: use same interface for \textit{generality}
Recall: **Structure** Operations

- `size()`  
- `isEmpty()`  
- `add()`  
- `remove()`  
- `clear()`  
- `contains()`  

But also: a 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
    - **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 the next element, step forward
    default void remove(); // removes most recently returned value
}
```

- **Default**: Java provides an implementation for remove
  - It throws an `UnsupportedOperationException` exception
Recall: Fibonacci Numbers

- We have previously seen Fibonacci numbers during recursion, where
- \( F_1 = 1, \ F_2 = 1, \ F_n = F_{n-1} + F_{n-2} \)

```java
public fib(int n) {
    if (n <= 2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```
A Simple Iterator

• Example: FibonacciNumbers. An iterator for the first $n$ Fibonacci numbers.

```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
Iterators Of Structures

Goal: Have data structures produce their own iterators so we can ask for it when we need it. How?

- Define an iterator class for the structure, e.g.
  ```java
  public class VectorIterator<E>
  implements Iterator<E>;
  public class SinglyLinkedListIterator<E>
  implements Iterator<E>;
  ```

- Provide a method in the structure that returns an iterator
  ```java
  public Iterator<E> iterator(){ ... }
  ```
Iterators Of Structures

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

- **VectorIterator** holds:
  - A reference to 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
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 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
Using an AbstractIterator allows more flexible coding (but requiring a cast to AbstractIterator)

Note: It has the form of 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;
}
```

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

```java
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?
• 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
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:
   see TestIterator.java

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