[TAP:PMUHE] Stack vs Queue

• A Singly Linked List can be used to implement which of the following efficiently
  A. List
  B. Stack
  C. Queue
  D. B and C
  E. Whatever
Administrative Details

- Midterm and lab scores (4 and 5) will be released on Wednesday
- 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
Today’s Outline

• Iterators
  • Iterator interface
  • AbstractlIterator abstract class (structure5)
  • Aside: For-each and Iterable interface
Traversing a Structure

- `numOccurs()` counts the number of times a particular (non-null) Object appears in a List.

```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;
}
```
Problems with our implementation

- generality
  - \texttt{get(i)} not defined on some structures
- efficiency
  - \texttt{get(i)} is “slow” on some structures

\texttt{Stack, Queue}

\texttt{Linked List:} $O(n)$ \texttt{\rightarrow} \texttt{numOccurs():} $O(n^2)$
Goals

• We want a mechanism to traverse data in structures, such that:
  • use same *interface* for *generality*
  • data structure-specific *implementation* for efficiency
Iterator

• **Iterator** is a general purpose mechanism for efficiently traversing data (structures)

• An Iterator:
  • Provides generic methods to dispense values
    • Traversal of elements: *Iteration*
    • Production of values: *Generation*
  • Uses different implementations for each structure
Today’s Outline

- Iterators
  - Iterator interface
  - AbstractIterator abstract class (structure5)
  - Aside: For-each and Iterable interface
Iterator interface

public interface Iterator<E> {
    boolean hasNext(); // are there remaining elements?
    E next(); // returns the next element
    default void remove();
    default void forEachRemaining(Consumer<? super E> action)
}
Recall: Fibonacci Numbers

• Definition
  • $F_1 = 1$
  • $F_2 = 1$
  • $F_n = F_{n-1} + F_{n-2}$

1 1 2 3 5 8 ...
A FibonacciNumbers Iterator

• 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 + next;

        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 for general structures

• Define an iterator class for the structures, 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(){ ... }
```
public int numOccurs (List<E> data, E o) {
    int count = 0;
    Iterator<E> iter = data.iterator();
    for (int i = 0; i < data.size(); i++) {
        if (o.equals(data.get(i)))
            count++;
        while (iter.hasNext())
            iter.next();
    }
    return count;
}
Today’s Outline

• Iterators
  • Iterator interface
  • AbstractlIterator abstract class (structure5)
  • Aside: For-each and Iterable interface
AbstractIterator

- structure5 defines AbstractIterator
- AbstractIterator
  - partially implements Iterator interface
  - adds two methods
    - get() – peek at (but don’t take) next element, and
    - reset() – reinitialize iterator for reuse
public class VectorIterator<E> extends AbstractIterator<E>{
    protected Vector<E> v;
    protected int cur;

    public VectorIterator(Vector<E> v){
        this.v = v;
        reset();
    }
    public void reset() { cur = 0; }
    public boolean hasNext(){ return cur < v.size(); }
    public E next() { return v.get(cur++); }
    public E get() { return v.get(cur); }
}

In Vector.java:

class Vector{
    public Iterator<E> iterator() {
        return new VectorIterator<E>(this);
    }
}
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(); }
}

public Iterator<E> iterator() {
    return new SinglyLinkedListIterator<E>(head);}

In SinglyLinkedList.java:

public Iterator<E> iterator() {
    return new SinglyLinkedListIterator<E>(head);
Iterator Use: numOccurs

- AbstractIterator allows the use of get() and reset() (but requires a cast to AbstractIterator)

```java
public int numOccurs (List<E> data, E o, E o2) {
    int count = 0;
    int count2 = 0;
    AbstractIterator<E> i =
        (AbstractIterator<E>) data.iterator();
    while(i.hasNext()){
        if(o.equals(i.get()))
            count++;
        if(o2.equals(i.get()))
            count2++;
        i.next();
    }
    return count;
}
```
More Iterator Examples

- We can also make “specialized iterators”
  - ReverseIterator.java
  - SkipIterator.java
Today’s Outline

- Iterators
  - Iterator interface
  - AbstractIterator abstract class (structure5)
  - Aside: For-each and Iterable interface
The Iterable Interface

• For-each construct uses iterators.
  
  ```java
  for( E elt : data ) { ... }
  ```

  is essentially the same as
  
  ```java
  for(Iterator<E> iter = data.iterator();
      iter.hasNext();) {
      E elt = iter.next();
      ...
  }
  ```

• Thus, we can use the “for-each” if `data` implements the `Iterable` interface

  ```java
  public interface Iterable<T> {
      public Iterator<T> iterator();
  }
  ```
General Rules for Iterators

1. Always call hasNext() before calling next()
2. In general, don’t add to structure while iterating
3. Use remove() with caution