CSCI 136
Data Structures &
Advanced Programming

Lecture 16
Fall 2017
The Bills
Administrative Details

• Lab 7: PostScript
  • Will be posted this weekend
  • Can’t wait!?  
    • Read about it in Java Structures: Section 10.5
• No partners this time
• Review before lab & come to lab with design doc
Last Time: Linear Structures

- Stack applications
  - Postscript
  - Mazerunning (Depth-First-Search)
  - (Implicit) program call stack
Today: Linear Structures

• Queues
  • Implementations Details
  • Applications
• Iterators
Stacks vs. Queues

• Stacks are LIFO (Last In First Out)
  • Methods: push, pop, peek, empty
  • Used for:
    • Evaluating expressions (postfix)
    • Solving mazes
    • Evaluating postscript
    • JVM method calls

• Queues are FIFO (First In First Out)
  • Another linear data structure (implements Linear interface)
  • Queue interface methods: enqueue (add), dequeue (remove), getFirst (get), peek (get)
Queues

- Examples:
  - Lines at movie theater, grocery store, etc.
  - OS event queue (keeps keystrokes, mouse clicks, etc, in order)
  - Printers
  - Routing network traffic (more on this later)
public interface Queue<E> extends Linear<E> {
    public void enqueue(E item);
    public E dequeue();
    public E getFirst(); //value not removed
    public E peek(); //same as get()
}
Implementing Queues

As with Stacks, we have three options:

**QueueArray**

```java
class QueueArray<E> implements Queue<E> {
    protected Object[] data; // can’t instantiate E[]
    int head;
    int count; // can be used to determine tail...
}
```

**QueueVector**

```java
class QueueVector<E> implements Queue<E> {
    protected Vector<E> data;
}
```

**QueueList**

```java
class QueueList<E> implements Queue<E> {
    protected List<E> data; // uses a CircularList
}
```
Tradeoffs:

- **QueueArray:**
  - enqueue is $O(1)$
  - dequeue is $O(1)$
  - Faster operations, but limited size

- **QueueVector:**
  - enqueue is $O(1)$ (but $O(n)$ in worst case - `ensureCapacity`)
  - dequeue is $O(n)$

- **QueueList:**
  - enqueue is $O(1)$ (`addLast`)
  - dequeue is $O(1)$ (`CLL removeFirst`)
• Let’s look at an example…
• How to implement?
  • enqueue(item), dequeue(), size()

QueueArray

head points to front of queue; tail points to next empty space (where next item will be added)

head and tail “wrap around” array; when queue is full, head == tail

After wrap around, head > tail in some cases!
public class queueArray<E> {

    protected Object[] data;    // Must use object because...
    protected int head;
    protected int count;

    public queueArray(int size) {
        data = new Object[size];    // ... can’t say “new E[size]”
    }

    public void enqueue(E item) {
        Assert.pre(count < data.length, "Queue is full.");
        int tail = (head + count) % data.length;
        data[tail] = item;
        count++;
    }

    public E dequeue() {
        Assert.pre(count>0,"The queue is empty.");
        E value = (E)data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        return value;
    }

    public boolean empty() {
        return count>0;
    }
}
Routing With Queues

Slides by Stephen Freund
The Network

moo.cs.williams.edu (137.165.8.3) -> Network -> www.google.com (216.239.37.99)

Message: 137.165.8.3 | 216.239.37.99 | "Search for ..."
Message:  

<table>
<thead>
<tr>
<th>Routers</th>
<th></th>
<th></th>
<th>&quot;Search for ...&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>moo</td>
<td></td>
<td></td>
<td>137.165.8.3</td>
</tr>
<tr>
<td>(137.165.8.3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.google.com">www.google.com</a></td>
<td></td>
<td>R4</td>
<td>216.239.37.99</td>
</tr>
<tr>
<td>(216.239.37.99)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Routers

Routing Algorithm
1. Receive message
2. Look up Destination Address
   a) Deliver message to Dest
   b) Forward to next Router
Router Internals

R1

R2

Lookup

Dest Addr

137.165.8.3
216.239.37.99
...

R1
R4
...

R1
R4
Buffering Messages

• There may be delays
  • Router receives messages faster than it can process and send
  • Some links are slower than others
    • Common speeds: 10 Mbs, 100Mbs, 1Gbs.
    • Wireless, satellite, infra-red, telephone line, ...
  • Hardware problems

• Want to be able to handle short-term congestion problems
Router Internals

lookup table:

137.165.8.3
216.239.37.99
...

R1 -> Lookup
Dest Addr -> R1

R4 -> Lookup
Dest Addr -> R4
Firewalls

Check Source

good

bad

discard

Lookup Dest Addr

R1

R4
Priority Scheduling

Priority of Source/Dest

Scheduler

R1

R4

high

med

low

70%

20%

10%

Lookup Dest Addr
"The Click Modular Router", Eddie Koller and Robert Morris, Jr.
Visiting Data from a Structure

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

```java
public int numOccurs(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() not defined on Linear structures (i.e., stacks and queues)

• get() is “slow” on some structures
  • $O(n)$ on SLL (and DLL)
  • So numOccurs = $O(n^2)$

• 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

- 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 next element
    default void remove() – removes most recently returned value
}
```

- Default: Java provides an implementation for `remove`
  - It throws an `UnsupportedOperationException` exception
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 iterators. 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 an array reference and index of next element
  - A reference to the data array of 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