CSCI 136
Data Structures & Advanced Programming

Lecture 15
Fall 2018
Instructor: Bills
Announcements

• Mid-Term Review Session
  • Tonight: 7:00-8:00 pm in TPL 203
  • No prepared remarks, so bring questions!
• Mid-term exam is Wednesday, October 17
  • During your normal lab session
  • You’ll have 1 hour & 45 minutes (if you come on time!)
  • Closed-book
  • Covers Chapters 1-7 & 9 and all topics up through Linked Lists
• A “sample” mid-term and study sheet are available online
  • [See Handouts & Problem Sets](#)
Last Time: Linear Structures

- Stack applications
  - Arithmetic Expressions
  - Postscript
  - Mazerunning (Depth-First-Search)
Today: Linear Structures

- Stacks
  - (Implicit) program call stack
- Queues
  - Implementations Details
  - Applications
- Iterators
Recursive “Pseudo-Code” Sketch

Boolean RecSolve(Maze m, Position current)
   If (current equals finish) return true
   Mark current as visited
   next ← some unvisited neighbor of current (or null if none left)
   While (next does not equal null && recSolve(m, next) is false)
       next ← some unvisited neighbor of current (or null if none left)
   Return next ≠ null

• To solve maze, call: Boolean recSolve(m, start)
• To prove correct: Induction on distance from current to finish
• How could we generate the actual solution?
Method Call Stacks

- In JVM, need to keep track of method calls
- JVM maintains stack of method invocations (called frames)
- Stack of frames
  - Receiver object, parameters, local variables
- On method call
  - Push new frame, fill in parameters, run code
- Exceptions print out stack
- Example: StackEx.java
- Recursive calls recurse too far: StackOverflowException
  - Overflow.java
Stacks vs. Queues

- **Stacks are LIFO (Last In First Out)**
  - Methods: push, pop, peek, empty
  - Sample Uses:
    - 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 declare E[]
    int head;
    int count; // better than storing 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
}
```

All three of these also extend AbstractQueue
QueueArray

• Let’s look at an example…
• How to implement?
  • enqueue(item), dequeue(), size()

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>tail</td>
<td></td>
</tr>
</tbody>
</table>

enqueue(C)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>tail</td>
<td></td>
</tr>
</tbody>
</table>

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;
    }
}
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)
Routing With Queues

Slides by Stephen Freund
The Network

moo.cs.williams.edu (137.165.8.3)

www.google.com (216.239.37.99)

Message:

| 137.165.8.3 | 216.239.37.99 | "Search for ..." |
Routing Algorithm
1. Receive message
2. Look up Destination Address
   a) Deliver message to Dest
   b) Forward to next Router
Router Internals

R1

R4

Lookup

Dest Addr

137.165.8.3
216.239.37.99
...
R1
R4

R2

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
Priority Scheduling

Priority of Source/Dest

Scheduler

Lookup Dest Addr

R1

R4

high
med
low

70%
20%
10%
Bandwidth Shaper

Classify Message

Limit(100)

Scheduler

Lookup Dest Addr

music

other
Choosing The Best Route

moo
(137.165.8.3)

R1

R2

R3

R4

google
(216.239.37.99)
Choosing Routes

- Routers exchange information periodically
  - Attempt to route on "best" path to destination
- Not easy to determine:
  - Network congestion varies (evening vs. morning)
  - Hardware added/removed or failures
- Dijkstra's algorithm (later)
Visiting Data from a Structure

• 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 $\text{numOccurs} = O(n^2)$ 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

- An Iterator:
  - Provides generic methods to dispense values
    - Traversal of elements: *Iteration*
    - Production of values: *Generation*
  - Abstracts away details of how elements are retrieved
  - 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
A Simple Iterator

- Example: FibonacciNumbers

```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;
    }
}
```