CS 240

Data Structures and Algorithms I

Alex Vondrak

ajvondrak@csupomona.edu

November 14, 2011

Linked Lists

```
class List<E> {
    private class Node<E> {
        E       data;
        Node<E> link;

        public Node(E data, Node<E> link) {
            this.data = data;
            this.link = link;
        }
    }

    private Node<E> head;

    // ...
}
```

Iterative Algorithms

We're used to thinking in terms of iteration:

- Set up initial state (variables, etc.)
- Repeatedly perform a process until it reaches a desired goal
- **`for`**-loops, **`while`**-loops, etc.

Example (`toString` method)

We've seen an iterative way to "step through" each element of a `List<E>` in the `LinkedList.java` file.

Example (`length` method—Worked Out In Class)

Let's implement the `length` method of the `List<E>` class iteratively.

Recursive Algorithms

Another natural way to think is in terms of recursion (or self-reference):

- Start with base cases—the "simplest" cases that needn't be defined with self-reference
- Recursive cases (or inductive cases) solve an instance of the problem by referring to a "simpler" case of the same problem (until we reach a base case)

Example (Recursive Multiplication)

We may define multiplication of nonnegative integers recursively:

$$m \cdot 0 = 0 \qquad \text{(base case)}$$

$$m \cdot n = m + m \cdot (n - 1) \qquad \text{(recursive case)}$$

Example (`length` method—Worked Out In Class)

`length` can be defined recursively, too.

A Closer Look At Recursion

Internally, recursive methods are handled by stacks of call frames (or activation records):

- Every time a method is invoked, we allocate space to store
  - Input parameters
  - The return address
  - Local variables
- Upon allocating the frame, we push it to the call stack
- When we finish executing the method we
  - Restore certain portions of memory
  - Pop the activation record
  - Jump to the code at the frame's return address