Slide 1

Administrivia

- Reminder: Homework 6 code due today.
- Homework 7 due dates next week.
- Quizzes 5 and 6 will be take-home not-for-credit. Solutions will be online.
- Sample programs page has additional examples of ways to use multithreading (for the interested?).

Slide 2

Recursion — Overview

- Basic approach:
  - Identify “base case” — something you can solve directly.
  - Figure out how to decompose non-base cases into “smaller” problems, and apply algorithm to smaller problems.
- How to think about “does it work?”
  - Does it work for base case(s)?
  - Assuming recursive calls work, does it work for other cases?
  - Does every recursive call get you at least one step closer to a base case?
- Implementation — conceptually (and usually in fact) involves a stack of calls-in-progress.
- Can be slower than iteration (though sometimes not), but can also be much easier to understand.

Slide 3

Recursion — Simple Examples

- Factorial function.
- Function to compute Fibonacci numbers (very slow!).

Slide 4

Recursion — More Examples

- Linked list implementation (if time permits).
- Quicksort — pick “pivot” element, split array into elements less than pivot and elements greater than pivot, and sort recursively. Why does this work?
- Mergesort — split array (or list) into two pieces of equal size, sort recursively, merge. Why does this work?

Slide 5

Trees — Mathematical Definition

- One definition —
  - Set of nodes, one called root.
  - Set of edges (directed connections between nodes).
  - Root has no incoming edges; all other nodes have exactly one (from parent).
  - Each node can have 0 or more outgoing edges (to children — if none, leaf node).
- Another, recursive definition — tree is one node connected by edges to 0 or more subtrees.
- This is a general tree — e.g., to represent hierarchy such as filesystem.

Slide 6

Implementing Trees

- Define `Node` data structure, analogous to linked list, with reference to data and references to children (array or linked list or . . .).
- Easier if number of children is limited to two, and this turns out to be sufficiently useful in practice — "binary tree". Then `Node` consists of pointers to data and left and right subtrees.

Slide 7

Tree Traversals

- For linked lists we defined a way to visit all elements — "iterator". Is there something analogous for trees?
- Well — three orders that are easy to define and implement:
  - Preorder — root first.
  - Postorder — root last.
  - Inorder — leftmost subtree first, then root, then remaining subtrees. (Admittedly a little weird for non-binary trees.)
- (Sketch some code for at least one of these.)

Slide 8

Sorted Binary Trees (Binary Search Trees)

- Key property — everything in the left subtree is smaller than the root, and everything in the right is bigger.
- Why is this useful? If you want a data structure to hold a collection that will be searched frequently, what are the choices? and how fast is each to search? to modify (insert/remove)? Compare approximate times for arrays (sorted and unsorted), linked lists (sorted and unsorted), sorted binary tree.
- (Sketch some code for `add` and `find`. `remove` is trickier . . .)

Slide 9

Minute Essay

- None — sign in.