Slide 1

Administrivia

- Remaining homeworks on Web with due dates. Some of these need to be enforced more strictly than I've been enforcing due dates this semester. Here are the "not accepted past" deadlines:
  - Homework 7 (design and code) no later than 11:59pm December 7. Usual late penalties apply.
  - Homework 8 design accepted without penalty through 11:59pm December 7; not accepted later.
  - Homework 8 due at the time of the final; not accepted past 11:59pm that day.

Slide 2

Trees — Recap of 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 3

Implementing Trees

- Define `Node` data structure, analogous to linked list, with reference to data and references to children (linked list or `Vector` 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 4

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 (unless you did that last time?).

Slide 5

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, so we'll just draw pictures. (You did these last time, right?)

Slide 6

Priority Queues, Revisited

- Several data structures we could use to implement priority queue ADT:
  - Unsorted linked list.
  - Sorted linked list.
  - Sorted binary tree.

  Compare how much work to add/remove if N elements. Can we do better? Maybe!

Slide 7

Heaps

- Heap is another tree-based data structure, with two properties:
  - A node is always "bigger than" both its children.
  - Tree is "complete".
- For a priority queue, we want to retrieve the "biggest" thing (for game problem, smallest update time). Does this seem useful?
- Note also that we can store a complete binary tree in an array.
- How to insert and remove? Compare running times.

Slide 8

Homework 7

- Writeup should be complete, but a short overview:
  - Objective is to write an alternate implementation for priority queue ADT and compare its performance to that of first implementation.
  - To compare performance, need to (1) add something to code to measure execution time, and (2) increase work being done by priority queue to the point where performance differences will show up.
- You can find code for a heap-based priority queue lots of places, but you will probably learn more if you write your own.
- Be sure to save a copy of your existing code before doing this, because you shouldn't include most of these changes in what you turn in as a "final" game (Homework 8).

Slide 9

Minute Essay

- None — quiz.