Slide 1

Administrivia

- Reminder: Homework 7 design due Tuesday after break.

Slide 2

Tree-Based Data Structures — Review

- Last time we talked about trees in general, binary trees, and two special cases — binary search trees and heaps.
- (A comment about the readings: The chapters of the online textbook that are supposed to discuss some of this material are incomplete. I've posted links to Wikipedia articles that may help supplement the discussion from last time.)

Slide 3

Binary Search Trees — Review

- These are binary trees (at most two children per node) that store data of some sortable type, with the property that for each node, all the elements stored in its left subtree are smaller than the node's data, and all the elements in its right subtree are larger. (Usually simplest not to store duplicates.)
- So, they're a reasonable choice for storing a sorted list.
- Methods for adding, removing, and searching for elements sketched last time. Code today as time permits.

Slide 4

Binary Search Trees Versus Sorted Linked Lists

- For a sorted linked list with N elements, adding or removing an element is $O(N)$.
- For a BST with N elements, adding or removing an element is $O(D)$, where D is the depth of the tree. Best case is that D is about $\log_2 N$. (Why?) Worst case is that it's N. (Various optimizations to the basic add/remove methods discussed last time can be done to avoid the worst-case situation.)
- One more consideration might be storage requirements. Which takes more?

Slide 5

Heaps — Review

- These are binary trees that also store data of some sortable type, with two properties:
  - They're complete — i.e., they look as if each level of the tree had been filled left to right.
  - For each node, the stored data is less than or equal to the data stored at all children. ("Less than" here should be interpreted broadly, so you could also have a tree with the maximum element at the root instead of the minimum.)
- So, they're a good choice for storing a priority queue.

Slide 6

Heaps Versus List-Based Priority Queues

- For a priority queue N elements implemented as a linked list, adding an element is $O(N)$, while removing the minimum element is $O(1)$.
- For a heap with N elements, adding or removing an element is $O(D)$, where D is the depth of the tree — which for a heap is known to be about $\log_2 N$. (Why?)
- Heaps also have the nice property that they can be stored as arrays, rather than using an explicit tree data structure.

Slide 7

Heaps — Adding an Element (Review)

- Add the element in the position needed to maintain the completeness property (easily found if stored in an array, yes?). Of course, this may break the ordering property, so . . .
- Starting at the newly-added element, move up the tree, exchanging a node and its parent, until either the node being examined is not smaller than its parent or you reach the root. (Think about why we can be sure this is all we need to do.)

Slide 8

Heaps — Removing Smallest Element (Review)

- Remove the element at the root (this is the smallest, no?). This breaks the completeness property, so . . .
- Move the “last” element (rightmost element of lowest level) to the root. This may break the ordering property, so . . .
- Starting at the root, move down the tree. At each level, compare the node to all its children. If at least one is smaller, exchange the node with its smallest child, and recurse into the corresponding subtree. Continue until either the node being examined is smaller than (or equal to) all its children or it has no children. (Think about why we can be sure this is all we need to do.)

Slide 9

Homework 7 — 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:
  - Increase work being done by priority queue to the point where performance differences will show up — “dummy entities”.
  - Add something to code to measure performance — time every 100 calls to player’s `update` method.
  - Provide a way to test with both implementations of the priority queue ADT and varying workloads — command-line arguments. (More in a later slide.)

Slide 10

Homework 7 — Tips

- 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 11

Command-Line Arguments

- Many mechanisms for starting programs provide a way of passing them information without using files or standard input — “command-line arguments”. Example — when you type at the command line

  ```
  ls -l myfile
  ```

  `-l` and `myfile` are passed to the `ls` in this way.

- C programs can receive command-line arguments by declaring `main` as

  ```
  int main(int argc, char *argv[])
  ```

  or equivalent, where `argc` is the number of arguments and `argv` is an array of C-style strings. By convention the zero-th argument is something identifying the program (e.g., its name). So in the `ls` example above, there would be three arguments . . .

Slide 12

Command-Line Arguments, Continued

- Java `main` methods also receive command-line arguments via arguments passed to `main`. `main` must always be declared with an argument of type `String[]`, which is a Java array containing the arguments. A Java equivalent of `ls` would get only two arguments for the example of the previous slide.

- Eclipse unfortunately doesn’t make it that easy to invoke programs with command-line arguments that vary from execution to execution, but it’s possible. An alternative is to run the program from the command line:

  ```
  java MainClass arg1 arg2
  ```

  or for your game something like

  ```
  java -classpath bin:PAD2.jar MainClass arg1
  arg2
  ```

  (Replace “:” with “;” on Windows.)

Slide 13

Minute Essay

- None — quiz.