Data Structures and Algorithms

Spring 2008-2009

Outline

1. AVL Trees
   - Worst-Case Analysis
2. Priority Queues
   - The PQ ADT

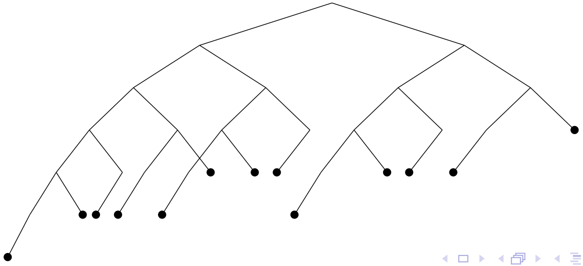
Outline

1. AVL Trees
   - Worst-Case Analysis

2. Priority Queues
   - The PQ ADT

How Bad Can an AVL Tree Get?

- What is the worst height that an AVL tree on *n* nodes can achieve?
- To answer this we firstly ask the inverse: what is the smallest number of nodes that can be in an AVL tree of height *h*?
- Because it's imbalanced everywhere, here's the most sparse tree with height 6:



How Bad Can an AVL Tree Get?

- We will denote the smallest number of nodes in a tree of height h by n_h
- Then

$$n_h = n_{h-1} + n_{h-2} + 1, \qquad n_1 = 2, n_0 = 1;$$

- This is very similar to the Fibonacci series and we use the same technique to solve it, by assuming a solution of the form

$$n_h = cr^h, \qquad \text{where } c \text{ and } r \text{ are constants}$$

- Then substituting gives

$$cr^h = cr^{h-1} + cr^{h-2} + 1$$

- Standard approach: solve the homogeneous part first and build upon this for full (inhomogeneous) equation

How Bad Can an AVL Tree Get? (contd.)

Solving the homogeneous part of this:

$$cr^h = cr^{h-1} + cr^{h-2}$$

gives

$$cr^{h-2}(r^2 - r - 1) = 0$$

Therefore, $n_h = cr^h$ is a solution if $c = 0$ or $r = 0$ or, (the interesting case), if $r^2 - r - 1 = 0$. That is,

$$r = \frac{1}{2}(1 + \sqrt{5}) \quad \text{or} \quad r = \frac{1}{2}(1 - \sqrt{5})$$

Since the sum of two solutions of a homogeneous recurrence relation is also a solution,

$$n_h = c_1 \frac{1}{2^h}(1 + \sqrt{5})^h + c_2 \frac{1}{2^h}(1 - \sqrt{5})^h$$

How Bad Can an AVL Tree Get? (contd.)

To solve the inhomogeneous equation $cr^h = cr^{h-1} + cr^{h-2} + 1$, since the inhomogeneous term is a constant, we try a solution of the form

$$n_h^* = B = n_{h-1}^* + n_{h-2}^* + 1 = B + B + 1$$

giving

$$B = -1$$

Therefore,

$$n_h = c_1 \frac{1}{2^h}(1+\sqrt{5})^h + c_2 \frac{1}{2^h}(1-\sqrt{5})^h - 1$$

Using the initial conditions and noting that $\left(\frac{1-\sqrt{5}}{2}\right)^h \longrightarrow 0, h \rightarrow 0$ we get

How Bad Can an AVL Tree Get? (contd.)

$$
\begin{aligned}
n_h &= \left(1+\frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1-\frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1 \\
&\approx 1.9\left(\frac{1+\sqrt{5}}{2}\right)^h
\end{aligned}
$$

(See also, Wikipedia's entry.)
Inverting this last expression we get

$$
\begin{aligned}
h &= \frac{1}{\log(1+\sqrt{5})/2}\log n_h + O(1) \\
&\approx 1.44\log n_h
\end{aligned}
$$

How Bad Can an AVL Tree Get? (contd.)

Therefore, the search time in the worst possible AVL tree is

- approx. 44% longer than the best search time of $\log n$ for the completely balanced binary search tree seen earlier, and
- about the same as the *average* search time in randomly constructed binary search trees
- can do *average-case* analysis on most-skewed tree (above) to demonstrate that avg. search cost is $1.04 \log n + O(1)$

Outline

Jumping the Queue

- In normal queue, the mode of selection is “first in, first out”
- In many situations (e.g. an OS) *turnaround* can often be improved by selecting the smallest job as the one to do next
- A *priority queue* enables finding efficiently the smallest job in a set
- Two operations supported by a PQ:
  - *insert*: put an item in the queue
  - *delete_min*: find the smallest item in the queue, remove it from the queue and return it to the calling function

Implementing a PQ

- Implementation alternatives:
  - Linked list with insertions at front; `insert`: $O(1)$-time cost; `delete_min`: $O(n)$-time cost
  - Sorted linked list where insertions maintains sortedness; `insert`: $O(n)$; `delete_min`: $O(1)$
  - Binary Search Tree (BST): height-balanced (HB) BST will guarantee $O(\log n)$-time for both ops; but this is more than is needed
- Data structure we construct that supports the PQ ADT is a *heap*
- Running times will be $O(\log n)$-time (worst case) for `delete_min` and $O(1)$-time (average case) for `insert`