Maps Holy Grail

• want something which looks and performs like an array
  • destination = capitals[‘IL’]
  • application programmer sees [ ] expects O(1) behavior

• so far have had mix of O(1) and O(n) for the five essential map behaviors
  • different implementations, different tradeoffs
Binary Search Trees

- sorted and searching fall-out naturally
- complexities $O(h)$
  - best case: $h = \log n$
  - worst case: $h = n$

Q: How to control $h$?

A: Rotations, which are $O(1)$ each
  - set number of nodes, 3
  - set number of rotations, 1 or 2
  - set number of re-linkings, 3 or 5

Main decision is when to do the rotations
AVL Trees

• Invented by Adel’son-Vel’skii, Landis (1962)

• Height-Balance Property
  • For every node in the tree, the heights of the children differ by at most one
  • every subtree of an AVL tree is also an AVL tree

• a node is balanced if $|r_h - l_h| = 0$ or $1$

• unbalanced or skewed otherwise

http://www.cs.usfca.edu/~galles/visualization/AVLtree.html
AVL Tree Height

• $h$ is $O(\log n)$

• sketch of proof
  • consider most left-heavy AVL tree possible for heights 1, 2, 3, 4, ...
  • observe pattern of heights $\approx$ Fibonacci sequence
  • Fibonacci sequence bounded by golden ratio
  • (arithmetic happens)
  • height bounded by $\log n$
AVL Tree Insertions

• as regular BST, search for spot then insert as leaf
• ascend towards root
• if unbalanced ancestor found, label it z
  • label taller child of z, y
  • label taller child of y, x
  • tri-node restructure x, y, z
• what’s the most restructurings we’d have to do?
AVL Tree Deletions

• delete as plain BST
  • move beforehand node into vacant spot
  • ascend and rebalance as AVL insertion
  • however, may require multiple restructurings
AVL Complexities

- Complexity chart in webnotes/textbook
- 5 essential map behaviors (and others): $O(\log n)!$
  - Generally, $O(\log n)$ for all better than mix of $O(1)/O(n)$
- `avl_tree.py`
Splay Trees

• Invented by Sleator, Tarjan (1985)
• Splaying - move a node to root
• zig-zig
  • rotate parent, then current
• zig-zag
  • rotate current twice
• zig
  • rotate current once
• O(1) each, but multiple might be required
When to Splay? Almost Always!

- Splay when key found
- If key not found, splay last node accessed
- Splay after insertion
- Splay after deletion
  - textbook – regular BST deletion, then splay deleted’s parent
  - UCSFA – splay delete target to root, then delete

http://www.cs.usfca.edu/~galles/visualization/SplayTree.html
Splay Tree Complexities

- Amortized $O(\log n)$
  - requires most activity to be near root so average number of splays allows cost of occasional deep access to be spread out
  - applications need to have temporal locality
- `splay_tree.py`