Balanced Trees

- Average case search trees are good, but what about worst case?
  - some applications require a hard bound on worst case
  - use balanced search trees
  - with each insert/delete, tree may be rebalanced
  - balancing methods vary in ease of implementation/
    constants, but all yield $O(\log n)$ worst case

- AVL trees, Red-Black trees, 2-3 (or B) trees
  - AVL trees are easy to understand, and operations lead to
    red-black trees
  - For all nodes, heights of left and right subtrees differ by
    no more than 1: height balanced (*Adelson-Velskii/Landis*)
Rotations for balancing

- all nodes are height-balanced
- rebalancing only needed (potentially) on root-to-leaf path of inserted leaf
- rebalance the first unbalanced node: rotate left child up

```
Tree * x = y->left;
y->left = x->right;
x->right = y->left;
return x;
```

Add 1, unbalance

Details include:
- adjusting parent
- mirror right child up
- double rotation
Sometimes two rotations are needed

- left-left and right-right insertions fixed by single rotation
- left-right and right-left fixed by double rotation
- a double rotation is really two single rotations

- actual implementation is complicated especially if efficiency is important
- how are balance factors calculated?
- how to “stop” popping out of recursion?
Other balanced tree schemes

- **B-trees**, especially useful for indexing/searching items stored on disk
  - more than two-way branching, try to have lots of information in memory to avoid searching disk
  - B-trees have information stored only in leaves, internal nodes are used to guide the search to the leaves
  - all leaves at same level, sometimes the tree grows a new root

- **Red-black trees are an efficient way of implementing 2-3 trees**
  - good performance, not impossible to implement
  - basis for STL map class (Standard Template Library)
  - use rotations like AVL trees, also nodes that are either red or black