Self balancing BSTs (AVL trees)

BST recap

- Most operations are O(tree height) because they may require traversing from root to leaves
- A complete tree has hight lg(N + 1) - 1
- A bad tree has height N - 1 (basically a linked list)
- The naive BST's height depends on the order elements are inserted
- We can't rely on users of our BST to pick a nice order for us

Self balancing trees

- These BSTs have rules which when enforced guarantee that the tree is O(lg N) height
- The bounds are pretty tight, something like 2*lgN
- Tree modifying operations (insert remove) need to be updated to enforce these properties
- Typically nodes must store some extra data to check whether the rules are being followed
- Most standard libraries include Red Black trees as their balanced BST data structure (ie java TreeSet) but we'll look at AVL trees because they're a bit easier to understand

Rebalancing: Tree rotations

- A tree rotation is an operation that adjusts the relationships between nodes such that the binary search property is maintained and the tree height is reduced (or stays the same, but has some advantage)
- For a “single rotation” consider a node N which is the left child of its parent P. There are also subtrees NL, NR (the potential subtrees rooted at N's children) and PR (it's parent's right subtree)
- A “Rotation” moves N to the position it's parent had and updates moves the children of N and P so that the subtrees still have the BST property

AVL Trees

- AVL are the intials of the creators: Adelson-Velskii and Landis
- The rule that AVL trees enforce is that for each node, the height of the subtrees rooted at its children can differ by at most 1 (ie left can be 1 taller, 1 shorter, or equal in height to the right)
- AVL tree impls usually store the height of the subtree rooted at each node as a member var of that node. The “height” of a nullptr is −1
- Insert/Delete operations can affect the balance property and so may require rebalancing the tree through via rotations

Imbalance via insertion

- Consider an imbalanced node, when we insert, one of 4 subtrees can be “too tall”
  - left-left grandchild
  - left-right grandchild
  - right-left grandchild
  - right-right grandchild
- We call the first/last cases “zig-zig” and “zag-zag” cases since we go the same direction twice
- In the zig-zig case, rotating the left child up to the imbalanced node fixes the problem

Zig-Zag and double rotations

- We can't fix the height imbalance in the zig-zag and zag-zig cases with one rotation
- We basically need to rotate the “too-tall” subtree “up” twice
- We could implement this by calling “rotate” twice, or by adding a custom “double rotation” function
- This moves the unbalanced grandchild to it's grandparent's spot, and it's former grandparent and parent become its children

Imbalance via deletion

- Analagous to insertion: an imbalance at a node could only be caused by deleting from one of 4 subtrees (the same zig-zig, zig-zag, zag-zig and zag-zag subtrees)
- There are a couple of minor subtleties, but basically, can apply the same rebalancing algorithm to a tree that became unbalanced by deletion as to one that became inbalanced by insertion!

Implementation

- The simplest implementation (IMHO) uses a strategy that can be used for recursive delete from the unbalanced BST
- We define recursive add and remove methods: add(node, item) which return the new root of the subtree that was rooted at node after adding item and rebalancing that subtree if necessary
- In code that pretty much means wrapping the nodes we're returning with `return rebalance(nodeWeWantToReturn)`
- An iterative implementation is possible but trickier

Red Black Trees

- Red Black trees are more common in production than AVL trees because they store less state and may require fewer rotations because their height can be somewhat taller than AVL trees
- This means that inserts/deletes tend to be faster because they require fewer rotations, but searches will be slower because the trees are slightly taller
- Instead of storing heights in each node, nodes store whether they are “Red” or “black” which only takes 1 bit and can be stored basically for free with some bit fiddling tricks
- Iterative implementations of RB Tree methods are also supposedly simpler than iterative AVL implementations

BTrees

- We'll see these in CS6016
- BTrees are n-ary trees, meaning they nodes can have WAY more than 2 children (100s)
- They maintain their balance by forcing all leaves to be at exactly the same height. The tree actually grows by splitting the root into two and adding a new root on top.
- They have lots of great properties and get used in databases
- The rust standard library treeset is actually a btree, not a BST