AVL Trees

Background

Complete Binary Trees

A complete tree is a tree of given depth with the maximum number of nodes. This tree has 2^(d + 1) - 1 nodes, where d is the depth. Alternatively, n items can be stored in a binary tree of depth floor(log(n)), where the log is to the base 2.

You can store 31 items in a tree of depth 4.

The Promise of Binary Search Trees

Binary trees can enable binary search with complexity O(log(n)) if we can ensure that the tree is close to being complete.

Is there an indication that binary trees are close to being complete?

How Complete Are BSTs?

If we use our method for adding items to the tree, we can end up with e.g. a tree of depth 9 instead of a tree of depth 3.

This is because the operations we defined were focused on doing as little work as possible while maintaining BST properties. This creates more work for when we want to search the structure (which we may also need to do when adding or removing items).

Rotation

[look at/copy in diagrams from dr. brown's notes]

Rotation changes between two equivalent binary trees, and decreases the height of one section while increasing the height of another.

```
x.rightchild = z.leftchild
z.leftchild.parent = x
z.leftchild = x
z.parent = x.parent
x.parent.?child = z
x.parent = z
```

This is O(1).

Double Rotation

If there's a zigzag in the longest path, we can do two rotations to end up at a more balanced tree. The first removes the zigzag, the second balances the tree.

Since it's just two rotations, it's still O(1).

Restructuring

A node x is unbalanced if children y and z are such that the difference between their heights is 2 or more, or if z only has one child, and the height of x is 2 or more (since these are the conditions under which our rotations help).

- If node x is unbalanced with higher rightchild z and z's rightchild is its higher child, then rotate z into x (from the right).
- If node x is unbalanced with higher rightchild z and z's leftchild w is its higher child, rotate w into z (from the left) and then into z (from the right).
- And symmetric versions.

AVL Trees

An AVL Tree (or a balanced tree) is a Binary Search Tree in which no node is unbalanced.

To implement an AVL Tree, each time an item is added or removed from the tree, we rebalance the tree.

Note that only ancestors of the added/deleted/moved node can become unbalanced, so we only need to search up the tree.

This is named after Georgy Adelson-Velsky and Evgenii Landis, who wrote a paper in 1962 on this.

Rebalance Method

```
rebalance()
    update the height
    if node is unbalanced
        restructure the node
        if node had a parent before restructuring
            parent.rebalance()
    else
        if height changed and node has a parent
            parent.rebalance()
```

To do this we're going to maintain a height variable for each BSTNode.

After each removal of an item:

```
if removed node was a leaf
    lead.parent.rebalance()
else if removed node was a semileaf
    semileaf.parent.rebalance()
else            # internal - [missed a bit]
    node = the original parent of biggest
    node.rebalance()
```

AVL Tree Properties

- The height of an AVL Tree for n items is O(log(n)), which we'll prove later.
- search is O(log(n))
- add is O(log(n))
- delete is O(log(n))
- find_min; find_max are O(log(n))
- Traversing all nodes is O(n)

So we've brought it from O(n) for BSTs down to O(log(n)), which is important for large trees.