Data Structures with C++ : CS189

Lecture 8-2: Balancing Trees

Recap

- We need to keep a tree balanced to prevent bad data from slowing us down
- AVL balances a tree by checking if the two sides of any node have a height differing by 2
- You only check the nodes on the path you took since a node can't affect the other side of the tree
- Once we balance a node, we are done since there can't be two nodes out of balance

Red-Black

- This is an alternative to using AVL on adds and removes
  - It's fast but okay
- The root is black
- A nullptr off the bottom is considered black
- A red node can't have a red child
- Every path from root to leaf has the same number of blacks
- I thought this was way weirder to write

Red-Black vs AVL

- Like everything in this class, all strategies are valid but are good at different things
- Red-black needs to store a color bit per node, AVL stores an int per node
  - RB is smaller
- AVL makes sure that no two sides of a node are off by two, Red-Black only guarantees the nodes are off by double
  - AVL is faster at looking up
- AVL's Remove might need to rotate every node, RB is constant
  - RB is faster at removing

DSW

- This is an entirely different balancing method than AVL
- It's perfect but slow, and operates on the whole tree
  - So the very first add or remove puts it out of DSW
- Pull all of the data out in to a Vector
  - You know it's already sorted in a BST
- Make the middle item the head, and then keep breaking halves in half to make a perfect symmetrical tree
  - Only practical with small data or long down times

Splaying

- Doesn't balance in absolute terms like AVL or RB
- Another case of "know your program", this one is good if you know some items are accessed more than others
- Every time you do a Find, make that node the root
- As you travel down the path, you break the tree off in to Lower and Higher
- Data becomes root, and those two subtrees are its children

End

Even the stuff that isn't in your homework is important