Data Structures with C++ : CS189

Lecture 8-1: Balancing Trees

Recap

- A binary search tree speeds up lookups by keeping all data less than a node to the left and all data greater to the right
- This is log n because every decision you make cuts half of the tree off from being considered
- BST is what's inside a Set, but a BST is not itself an STL container

Balancing

- If we took the tree we have so far and entered a set of data that was in order, you wouldn't have a tree
  - 1 2 3 4 5 6 7 8 9 makes a straight line down the right side
- That's O(n) list-searching. We want O(log n) so we need to do something to change the tree
- To "balance" a tree is to try to make it as even as possible

Types of Balancing

- There are two different times you could balance a tree
  - As you add or remove data
  - All at once when the tree's not being used
- There are two paradigms for how to balance
  - Fast but okay
  - More overhead but better (more tightly packed)
- All of the choices are right, they just depend on the program and the data

AVL (p 257)

- This is a faster-with-overhead method that has the benefit of being easier to write
  - I'll explain when I talk about the homework why this isn't scary
- The idea is that if the height* of the two sides of a node differ by two, we will even the two sides out
  - *Height = Greatest number of nodes to get to a leaf
- Since we do this on every add and delete, the tree is always kept *mostly* balanced

Out of Balance

- This is another example of this class being impossible without drawing pictures
- Picture A is in balance.
  - The number in the circle is the difference in height between the sides
  - "h" means "0 or more nodes that I don't care about" They aren't part of the rotate, except to note that their biggest height is h
- In picture B, a node has been added to the rightmost subtree, and now P is out of balance
  - Left is h, right is h + 2 (h + 1 + Q)

Rotation (p 257, fig 6.41)

- Picture c is the result of balancing
- We take P, Q, and P's left and rotate the three nodes counter-clockwise
  - Note the tricky move Q's left makes
- We have maintained BST-ness as left data is still less and right data is still greater
  - That middle tree went from "Greater than P and less than Q" to "Less than Q and greater than P"
- You can see this code in the files provided
  - The assignment is going to be optimizing that file, not writing it from scratch

Double Rotation (p 258, fig 6.42)

- That rotation works if the node added was right-right or left-left (outside)
- If the add went right-left or left-right (inside) then we need to rotate twice
- (a) is in balance, and (b)'s right-left add put it out of balance
- (c) is doing nothing. It is just observing that Q's left could be called R
- (d) Q and R rotate clockwise
- (e) P and R rotate counter-clockwise
- And again, I wrote this. I just wrote it badly for you.

My Implementation

- In my code, you will see that I recursively balance every node in the tree at every add.
- You will also see that I recursively set the heights of every node twice for every add
- But this algorithm never forks
  - If you can use a loop, use a loop
  - The path of adding a node is a straight line
- A rotate on an add means it is done, but a delete always has to check all the way up

Your Task

I also summarize this in the assignment.

1. Keep track of the path you take to do your Add or Remove
2. Only update the heights of those nodes
    a. Remember that two-child delete needs to consider the path to the actual deleted node too
3. Only check for balance on those nodes
4. If you make a rotation, update the heights of the nodes involved
5. If you make a rotation on an add, you are done
    a. Remove could make a chain of rotates

End

Draw this out, and ask questions I can draw on the board