CSCI 4041: Algorithms and Data Structures
Balanced Binary Search Trees (AVL trees).
Part 1

Last revision March 18, 2020

This is part one of a two-part series about balanced binary search trees. Maybe there will be more than two parts. Here's the agenda.

What's an AVL tree?
The AVL tree property.
How to search an AVL tree.
Transformations on AVL trees.

In the last lecture (or whatever these are) I discussed unbalanced binary search trees (BST's). In this lecture I'll discuss balanced BST's. Specifically I'll discuss a kind of balanced BST's called *AVL trees*, named for the two Soviet Russian mathematicians who invented them in 1962: Georgy Adelson-Velskii and Evgenii Landis.

AVL trees aren't in Cormen's book. Instead, he discusses another kind of balanced BST's called *red-black trees* (pages 308–338). The nodes in a red-black tree are imagined to have colors, either red or black, which is where their name comes from. I think Cormen's discussion of red-black trees is just insanely complicated, and I don't know anyone who claims to fully understand it. AVL trees, however, are easy to understand, and in my opinion they work better, so I'll discuss them instead. You don't have to know how red-black trees work! A few things from Cormen's book are relevant to AVL trees, however, so I'll mention them where appropriate.

My discussion of AVL trees is stolen from a book called *Algorithms + Data Structures = Programs* by Niklaus Wirth (Prentice Hall, 1976). This is one of my favorite computer science textbooks (there aren't many I like) but it's been out of print for more than forty years. I don't know if it's legally available online. It might be.

What's an AVL tree? The first thing to know is that an AVL tree is just a BST, but it has an extra property, which I'll call the *AVL tree property*. Suppose p points to a node in an AVL tree. Then $p.left$ and $p.right$ are p's left and right subtrees, respectively. The AVL tree property says that for all nodes p, the heights of $p.left$ and $p.right$ always differ by at most 1. Recall that the *height* of a binary tree is the length of its longest path. An empty subtree has height 0, and it doesn't matter if we compute heights by counting nodes, or by counting links.

For example, here's a BST that satisfies the AVL tree property. Its keys are integers, and its values aren't shown.

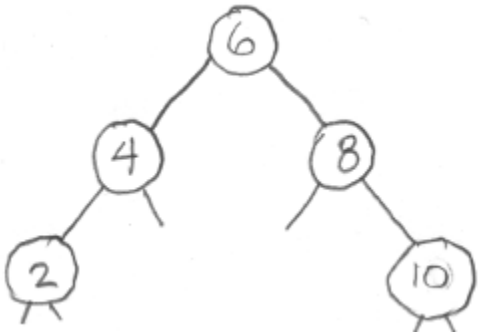

Fig 1.

Node 6's subtrees both have height 2, so their heights differ by at most 1. Node 4 has a left subtree of height 1 and a right subtree of height 0, so their heights differ by at most 1. Node 10 has two subtrees, both of height 0, so their heights also differ by at most 1. I'll stop here and hope you get the point.

An AVL tree with a given set of keys isn't necessarily unique. There may be many AVL trees with the same keys. For example, Fig. 2 shows another AVL tree, shaped differently, but with the same keys as in Fig. 1. Node 8's left subtree has height 2, and its right subtree has height 1, so again the heights of both subtrees differ by at most 1.

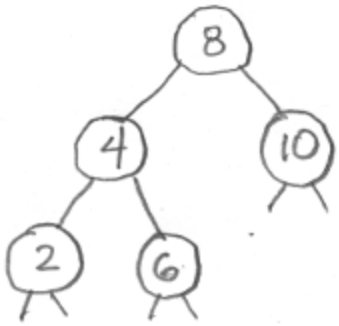

Fig. 2.

An AVL tree isn't always perfectly balanced for all nodes—note that we can find key 10 faster in Fig. 2 than we can in Fig. 1—but it's usually close to being perfect.

How to search an AVL tree. AVL trees are just BST's, with an extra constraint on their shapes. That means we can search an AVL tree using the same algorithm we'd use to search an ordinary BST. For example, the procedure BST-GET from the previous lecture could search both Fig. 1 and Fig. 2, without change.

Transformations on AVL trees. However, we do need a different algorithm for adding key-value pairs to an AVL tree. This is because the algorithm must take the AVL property into account. When it adds a node to an AVL tree, it must reorganize the tree if the AVL property would be violated. This is done by applying transformations to the tree.

A *transformation* is a statement of the form $L \Rightarrow R$, where the arrow '⇒' means ''turns into.'' It says that whenever we see something that matches L, we can turn it into something else that matches R. This course is about designing algorithms, and one way to design an algorithm is to have it apply transformations to some data structure, repeatedly, until we get a data structure that we want. That's how the AVL addition algorithm works.

There are four transformations. They're called *rotations* because they ''rotate'' the positions of some nodes around other nodes. The first is a *single left left rotation*, and is shown in Fig. 3. Here A and B (in circles) are keys in nodes; α, β, and γ (in triangles) are subtrees. Some or all of the subtrees may be empty.

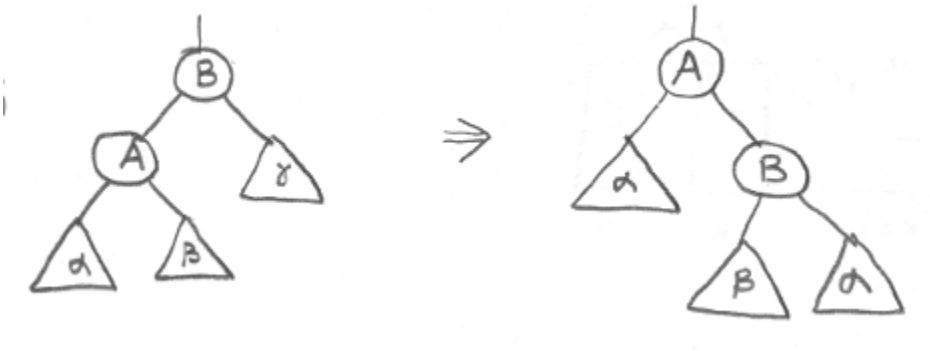

Fig. 3. Single Left Left Rotation.

The transformation gets its name because node A on the left is ''rotated'' around node B to its right. Personally, I don't find these names very helpful in visualizing what the transformations do. You might want to consider their names as just arbitrary labels.

There are two important things about this transformation (and the others also). First, if the tree on the left of '⇒' satisfies the BST property, then the tree on the right also satisfies it. For example, the tree on the left says that $A < B$, so A is in B's left subtree. The tree on the right says $B > A$, so B is in A's right subtree. Similarly, in the tree on the left, if all the nodes in the subtree α are less than A, and less than B, then this is also true of the tree on the right. As before, I won't show this for each part of the tree, but I hope you get the point.

The second important thing is that the transformation changes the heights of the subtrees α, β, and γ. Suppose we know

that *A*, *B*, α, β, and γ are parts of some larger tree, and we know that the larger tree is unbalanced because α's height is too large. We could apply this transformation to reduce α's height (moving it up) to make the tree better balanced.

The remaining transformations work in similar ways, and we could say much the same things about them as we said about Fig. 3. Fig. 4 is a *single right right rotation*, which is a mirror image of the single left left rotation in Fig. 3.

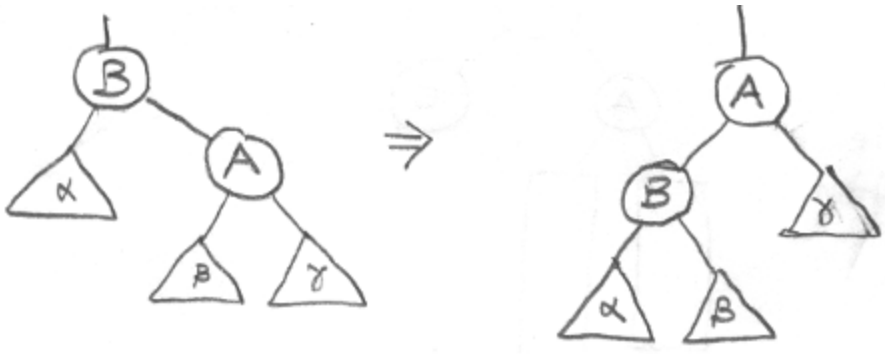

Fig. 4. Single Right Right Rotation.

Fig. 5 is a *double left right rotation*, in which we do a left rotation followed by a right rotation.

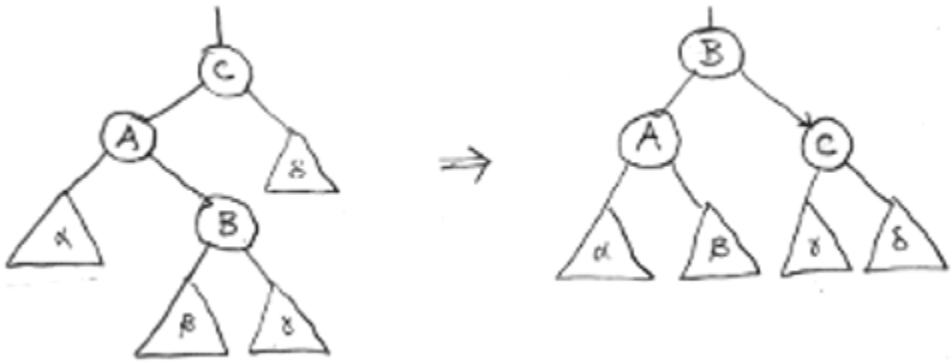

Fig. 5. Double Left Right Rotation.

And Fig. 6 is a *double right left rotation*, in which we do a right rotation followed by a left rotation. It's a mirror image of the double left right rotation in Fig. 5.

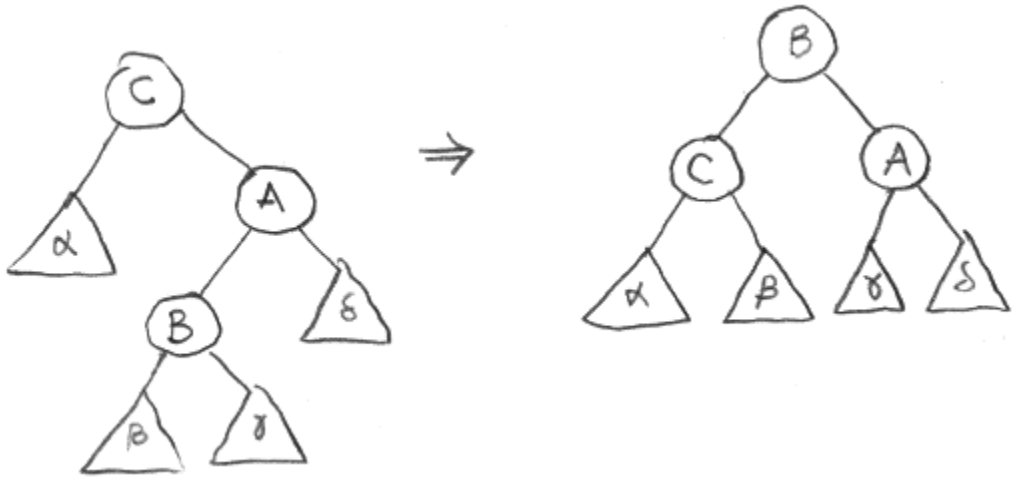

Fig. 6. Double Right Left Rotation.

As an aside, note that other balanced tree algorithms use transformations that are much like these. Red-black trees (which we're not discussing) do that. See pages 312–313 of Cormen for details.

We'll see an example later this week about how an addition algorithm for AVL trees can use these transformations. It uses them to take an unbalanced BST and reduce its height so it's better balanced. The trick is to figure out what transformations to use, and when to use them.