AVL Trees

Recall the operations (e.g. `find`, `insert`, `delete`) of a binary search tree. The runtime of these operations were all $O(h)$ where h represents the height of the tree, defined as the length of the longest branch. In the worst case, all the nodes of a tree could be on the same branch. In this case, $h = n$, so the runtime of these binary search tree operations are $O(n)$. However, we can maintain a much better upper bound on the height of the tree if we make efforts to balance the tree and even out the length of all branches. AVL trees are binary search trees that balances itself every time an element is inserted or deleted. **Each node of an AVL tree has the property that the heights of the sub-tree rooted at its children differ by at most one.**

Upper Bound of AVL Tree Height

We can show that an AVL tree with n nodes has $O(\log n)$ height. Let N_h represent the minimum number of nodes that can form an AVL tree of height h.

If we know N_{h-1} and N_{h-2}, we can determine N_h. Since this N_h-noded tree must have a height h, the root must have a child that has height $h-1$. To minimize the total number of nodes in this tree, we would have this sub-tree contain N_{h-1} nodes. By the property of an AVL tree, if one child has height $h-1$, the minimum height of the other child is $h-2$. By creating a tree with a root whose left sub-tree has N_{h-1} nodes and whose right sub-tree has N_{h-2} nodes, we have constructed the AVL tree of height h with the least nodes possible. This AVL tree has a total of $N_{h-1}+N_{h-2}+1$ nodes (N_{h-1} and N_{h-2} coming from the sub-trees at the children of the root, the 1 coming from the root itself).

The base cases are $N_1 = 1$ and $N_2 = 2$. From here, we can iteratively construct N_h by using the fact that $N_h = N_{h-1} + N_{h-2} + 1$ that we figured out above.

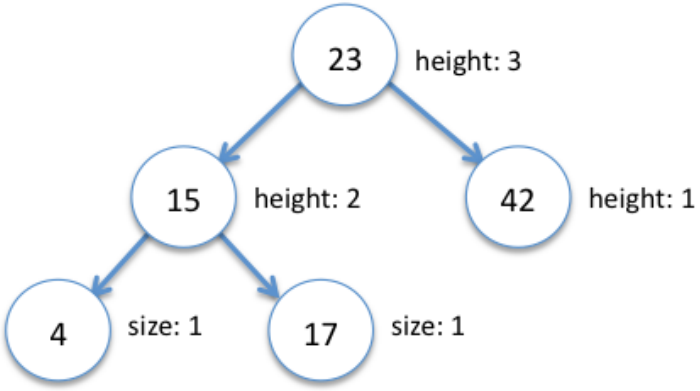
Using this formula, we can then reduce as such:

$$N_h = N_{h-1} + N_{h-2} + 1 \tag{1}$$
$$N_{h-1} = N_{h-2} + N_{h-3} + 1 \tag{2}$$
$$N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \tag{3}$$
$$N_h > 2N_{h-2} \tag{4}$$
$$N_h > 2^{\frac{h}{2}} \tag{5}$$
$$\log N_h > \log 2^{\frac{h}{2}} \tag{6}$$
$$2\log N_h > h \tag{7}$$
$$h = O(\log N_h) \tag{8}$$

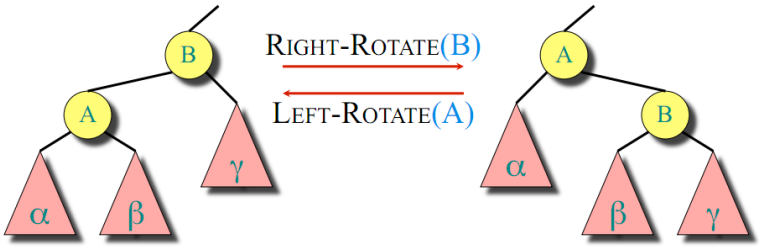
Showing that the height of an AVL tree is indeed $O(\log n)$.

AVL Rotation

We've shown that if we can indeed keep the tree balanced, we can keep the height of the tree at $O(\log n)$, which speeds up the worst case runtime of the tree operations. The next step is to show how to keep the tree balanced as we insert and delete nodes from the tree.



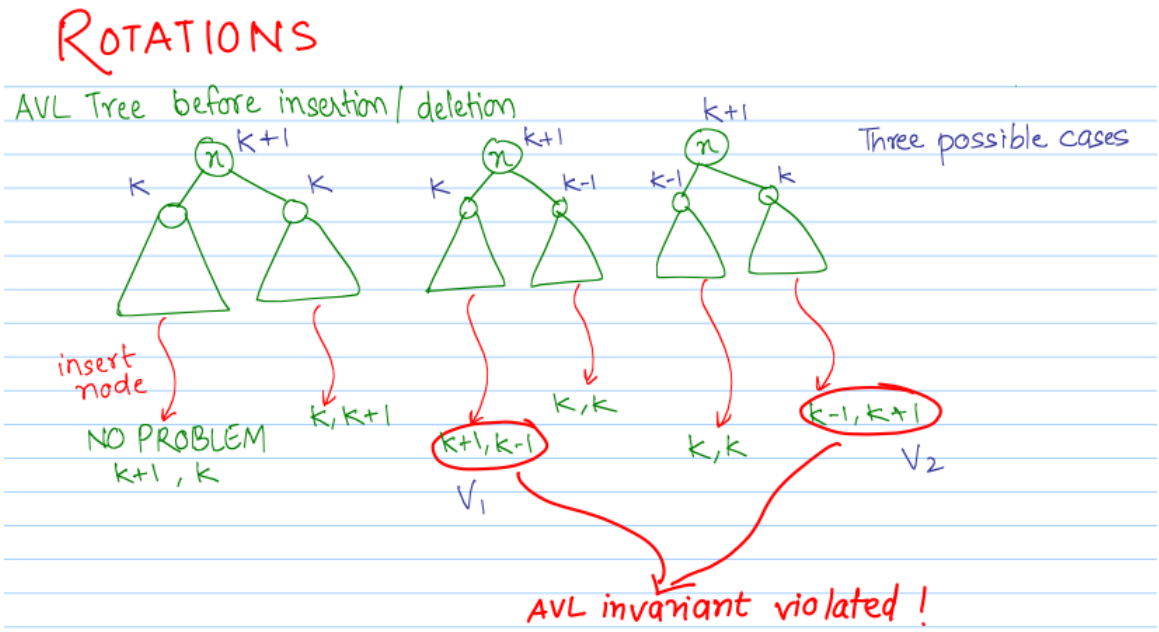
Since we need to maintain the property that the height of the children must not differ by more than 1 for every node, it would be useful if we could access a node's height without needing to examine the entire length of the branch that it's on. Recall that for a binary search tree, each node contained a `key`, `left`, `right`, and a `parent`. AVL trees will also contain an additional parameter, `height` to help us keep track of balance.



There are two operations needed to help balance an AVL tree: a left rotation and a right rotation. Rotations simply re-arrange the nodes of a tree to shift around the heights while maintaining the order of its elements. Making a rotation requires re-assigning `left`, `right`, and `parent` of a few nodes, but nothing more than that. Rotations are $O(1)$ time operations.

AVL Insertion

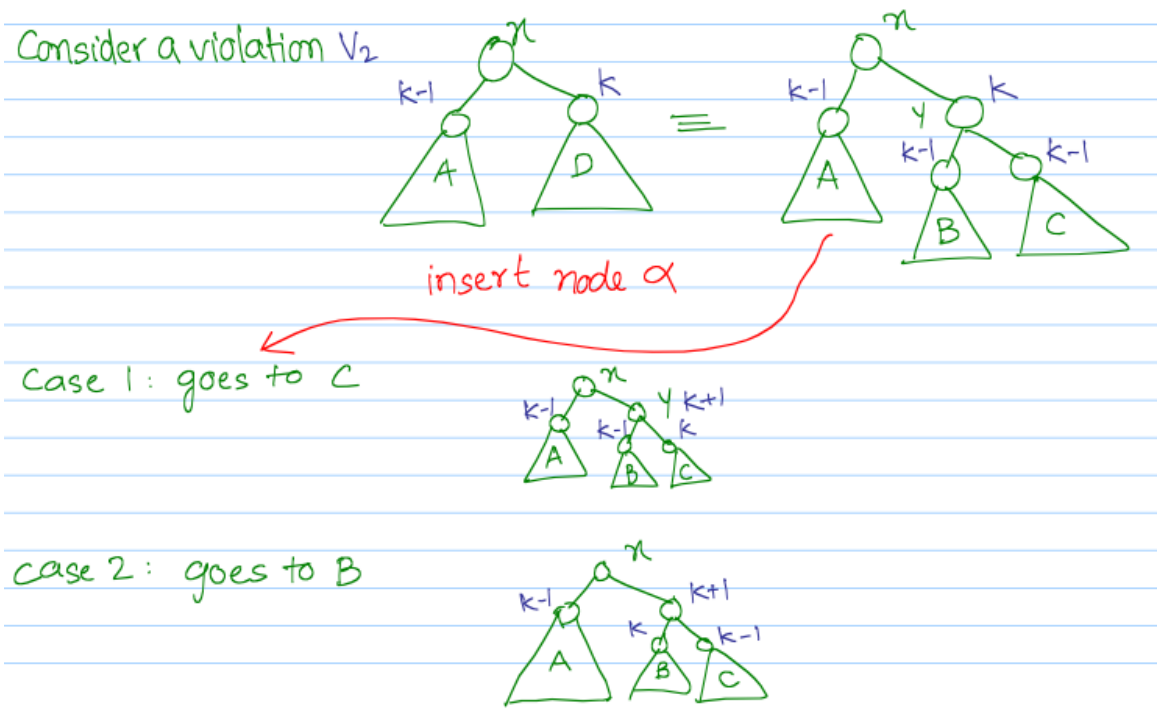
Now delegating to recitation notes from fall of 2009:



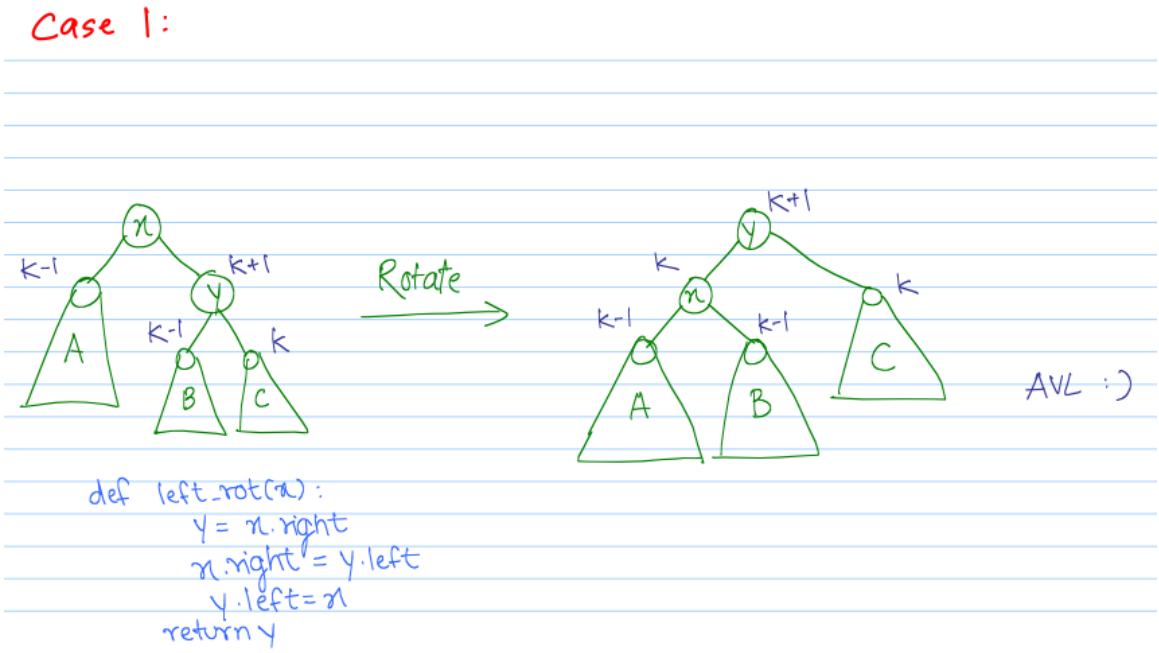
When we insert a node into node n, we have three possible cases.

1. Children of n have same height k. Inserting into either sub-tree will still result in a valid AVL tree
2. The left child of n is heavier than the right child. Inserting into the left child may imbalance the AVL tree
3. The right child of n is heavier than the left child. Inserting into the right child may imbalance the AVL tree

When the AVL tree gets imbalanced, we must make rotations in the tree to re-arrange the nodes so that the AVL tree becomes balanced once again. Note that adding a node into a k height tree does not always turn it into a $k+1$ height tree, since we could have inserted the node on a shorter branch of that tree. However, for now, we are looking specifically at the situations where adding a node into a k height tree does turn it into a $k+1$ height tree. Let's examine the case where we insert a node into a heavy right child.

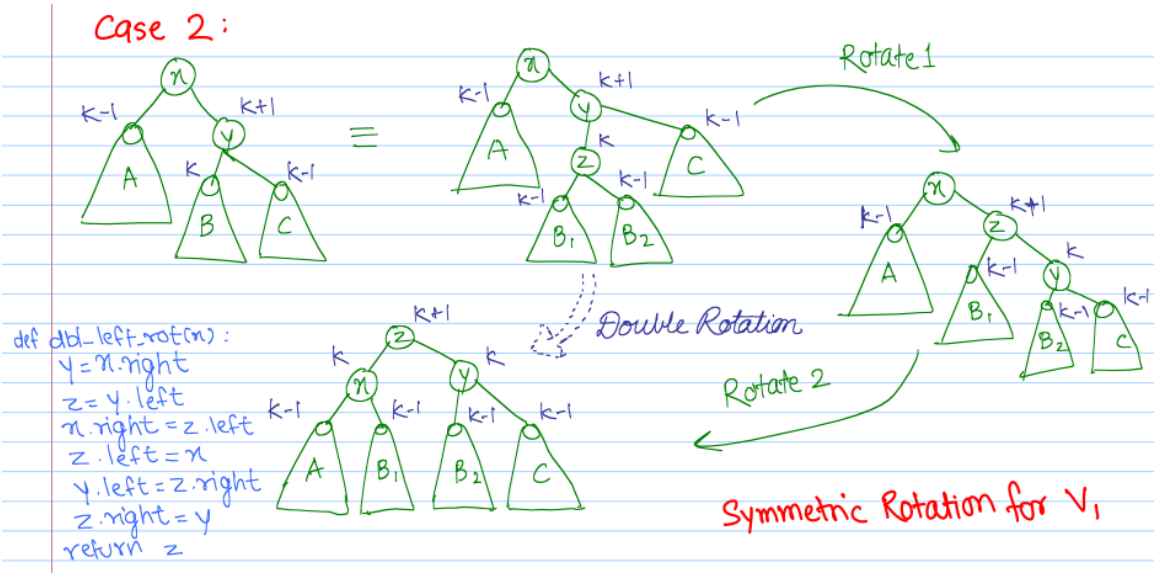


There are two cases here that will imbalance the AVL tree. We will once again look at the problem on a case by case basis.



In the first case, B had height $k-1$, C had height $k-1$, and a node was inserted into C, making its current height k. We call a left rotation on n to make the y node the new root and

shifting the B sub-tree over to be n's child. The order of the elements are preserved (In both trees, $A < n < B < y < C$), but after the rotation, we get a balanced tree.



In the second case, B had height $k-1$, C had height $k-1$, and a node was inserted into B, making its current height k. In this case, no single rotation on a node will result in a balanced tree, but if we make a right rotation on y and then a left rotation on x, we end up with a happy AVL tree.

If we insert a node into a heavy left child instead, the balancing solutions are flipped (i.e. right rotations instead of left rotations and vice versa), but the same concepts apply.

AVL insertions are binary search tree insertions plus at most two rotations. Since binary search tree insertions take $O(h)$ time, rotations are $O(1)$ time, and AVL trees have $h = O(\log n)$, AVL insertions take $O(\log n)$ time.

There are only a finite number of ways to imbalance an AVL tree after insertion. **AVL insertion is simply identifying whether or not the insertion will imbalance the tree, figuring out what imbalance case it causes, and making the rotations to fix that particular case**.

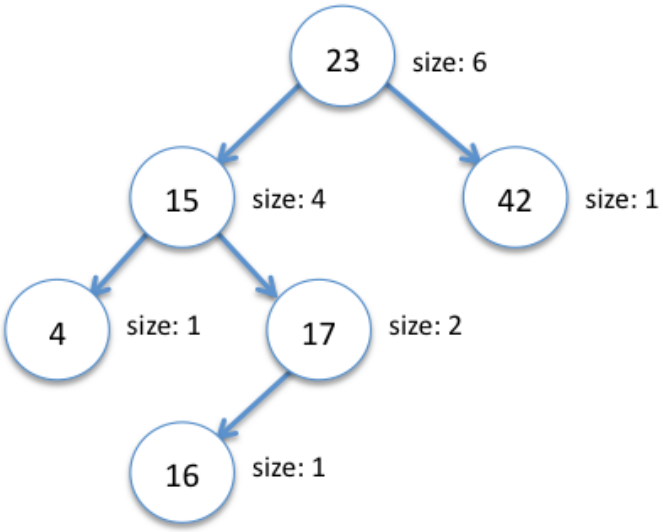
AVL Deletion

AVL deletion is not too different from insertion. The key difference here is that unlike insertion, which can only imbalance one node at a time, a single deletion of a node may imbalance several of its ancestors. Thus, when we delete a node and cause an imbalance of that node's parent, not only do we have to make the necessary rotation on the parent, but we have to traverse up the ancestry line, checking the balance, and possibly make some more rotations to fix the AVL tree.

Fixing the AVL tree after a deletion may require making $O(\log n)$ more rotations, but since rotations are $O(1)$ operations, the additional rotations does not affect the overall $O(\log n)$ runtime of a deletion.

Tree Augmentation

In AVL trees, we augmented each node to maintain the node's height and saw how that helped us maintain balance. Augmentation is a very useful tool to help us solve problems that a vanilla binary search tree cannot solve efficiently. We will learn about another useful augmentation, subtree size augmentation.



In subtree size augmentation, each node maintains the size of the subtree rooted at that node in a parameter `size`. The root of a tree containing n elements will have `size` $= n$ and each leaf of a tree will have `size` $= 1$.

The operations of this tree must maintain the `size` value of every node so that it is always correct.

For insertion, as we traverse down a branch to find where to insert a node, we need to increment `size` for every node that we visit. This is because going through a node during insertion means we will be inserting a node in the tree rooted at that node.

Deletion will also require some maintenance. Every time we remove a node, we must traverse up its ancestry and decrement `size` of all its ancestors.

If we wanted to augment an AVL tree with subtree size, we would also have to make sure that the rotation operations maintain `size` of all the nodes being moved around (Hint: the fact that x.size = x.left.size + x.right.size + 1 is useful here).

As you'll see in the problem set, using subtree augmentation can help speed up operations that normally would be slow on a regular binary search tree or AVL tree.