AVL Trees

In order to have a worst case running time for insert and delete operations to be O(log n), we must make it impossible for there to be a very long path in the binary search tree. The first balanced binary tree is the AVL tree, named after it's inventors, Adelson-Velskii and Landis. A binary search tree is an AVL tree iff each node in the tree satisfies the following property:

The height of the left subtree can differ from the height of the right subtree by at most 1.

Based on this property, we can show that the height of an AVL tree is logarithmic with respect to the number of nodes stored in the tree.

In particular, for an AVL tree of height H, we find that it must contain at least F_{H+3} -1 nodes. (F_i is the ith Fibonacci number.) To prove this, notice that the number of nodes in an AVL tree is the 1 plus the number of notes in the left subtree plus the number of nodes in the right subtree. If we let S_H represent the minimum number of nodes in an AVL tree with height H, we get the following recurrence relation:

$$S_H = S_{H-1} + S_{H-2} + 1$$

We also know that S_0=1 and S_1=2. Now we can prove the assertion above through induction.

For those of you who haven't seen induction yet, I won't "test" on it in this class. I'll try to explain the major steps of induction as best as I can, very briefly. Induction is used to prove that some statement or formula is true for all positive integers. Sometimes, it is difficult to prove a formula for all positive integers outright though.

In these cases, it may be easier to prove that IF the formula is true for an integer, say, 10 (we can call this k), then it MUST BE true for the next integer 11 (this would be k+1).

Finally, if we can prove that, AND we can show that the formula is true when you plug in 1 into it, it follows that the formula is true for all positive integers.

Here's a simple example you can hopefully relate to (I apologize to women who have small wardrobes!!!):

Assumptions: A female's wardrobe increases by 15% a year. A male's wardrobe increases by 10% a year. At the age of 20, a female has 50 pieces of clothing, while a male has 45.

We will prove: That for all years over 20 years of age, females own more pieces of clothing than males.

It is true for age 20 based on the given information.
Assume it's true for age k, where $k \geq 20$.

Now, under that assumption we will prove it for k+1. Let the number of clothes a female has at the age of k be F. Let the number of clothes a male has at the same age be M. At age k+1, a female must have 1.15F pieces of clothing. This is greater than 1.1F. Using the inductive hypothesis, this is greater than 1.1M, which is the number of pieces of clothing a male owns at age k+1.

Consider the following example:

Given that the sum of the first k integers is k(k+1)/2, I will prove that the sum for the first k+1 integers is (k+1)(k+2)/2.

1 + 2 + 3 + ... + (k+1) = (1 + 2 + 3 + .. + k) + (k+1)

$$= \frac{k(k+1)}{2} + (k+1)$$

because we are ASSUMING that the formula I have works for the first k integers.

Now, add this up with a common denominator:

$$= \frac{k(k+1)+2(k+1)}{2} = \frac{k^2+k+2k+2}{2}$$

$$= \frac{k^2+3k+2}{2} = \frac{(k+1)(k+2)}{2}$$

Now, what I have written above is a "template proof" that you can plug in any value for k. The problem being of course, we don't know if the formula I have actually holds for any value of k. BUT, I can simply check that it works for one by plugging k=1 into the equation below and verifying its truth:

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

The left hand side and right hand side both evaluate to 1, when plugging in k=1. So, this formula above is true when k=1. But, based on our template proof, if it is true for 1, it is true for k=2. And if it's true for k=2, it's true for k=3, etc.
Now, consider this example about binary trees:

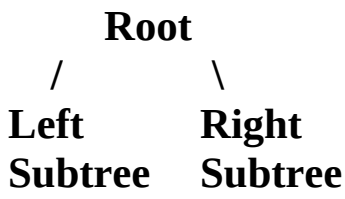
We will prove that an inorder traversal of a binary search tree of any height (positive integer) yields the elements in numerical

order. Certainly this is true for a binary tree of height 0, which has only one element in it.

Now, assume this is true for a binary tree of height k or less. (This is a strong inductive hypothesis. For now, don't worry about how it differs from a normal inductive hypothesis.)

Consider proving it for a binary search tree of height k+1:

The tree must look like this:



It also follows that both the left and right subtrees have a height of k or less. During an inorder traversal, all the nodes in the left subtree are printed using an inorder traversal, then the root node is printed, then all the values in the right subtree are printed using an inorder traversal.

Since the tree is a binary search tree, all the values in its left subtree are less than the root. These all print before the root prints. Furthermore, because this left subtree has height k or less, these all print in the proper order. Then the root is printed, which still maintains the proper order of nodes. Finally, all the values in the right subtree are printed. Since this subtree is of height k or less, these are ALSO printed in the right order, so that the WHOLE list is in the correct order, finishing the proof.

Problem: Prove that $S_H = F_{H+3} - 1$.

We will use induction on H, the height of the AVL tree.

Base Cases H=0: LHS = 1, RHS = F_3 - 1 = 2 - 1 = 1
H=1: LHS = 2, RHS = F_4 - 1 = 3 - 1 = 2

Inductive hypothesis: For an arbitrary integer k <= H, assume that $S_k = F_{k+3} - 1$.

Inductive step: Under the assumption above, prove for H=k+1 that $S_{k+1} = F_{k+1+3} - 1$.

$S_{k+1} = S_k + S_{k-1} + 1$ (because to form an AVL tree with the min. number of nodes of height k+1, one side of the root must have height k and the other k-1. This is because we need the sides to be within one, but we want to minimize the number of nodes. The only other option would have been k and k, which would NOT minimize the desired value.)

$= (F_{k+3} - 1) + (F_{k+2} - 1) + 1$, using the I.H. twice
$= (F_{k+3} + F_{k+2}) - 1$
$= F_{k+4} - 1$, using the defn. of Fibonacci numbers, to complete proof.

It can be shown through recurrence relations, that

$$F_n \approx 1/\sqrt{5}\,[(1 + \sqrt{5})/2]^n$$

So now, we have the following:

$$S_n \approx 1/\sqrt{5}\,[(1 + \sqrt{5})/2]^{n+3}$$

This says that when the height of an AVL tree is n, the minimum number of nodes it contains is $1/\sqrt{5}\,[(1 + \sqrt{5})/2]^{n+3}$.

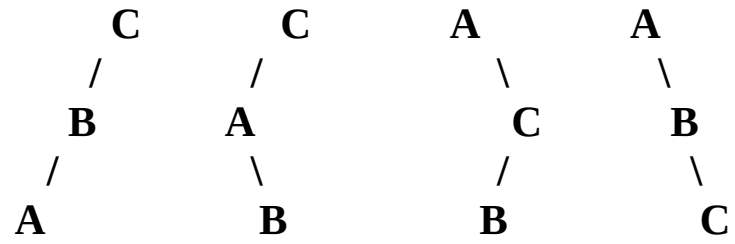
So, in order to find the height of a tree with n nodes, we must replace S_n with n and replace n with h? Why is this the case?

$$n \approx 1/\sqrt{5}\,[(1 + \sqrt{5})/2]^{h+3}$$
$$n \approx (1.618)^h$$
$$h \approx \log_{1.618} n$$
$$h = O(\log_2 n)$$

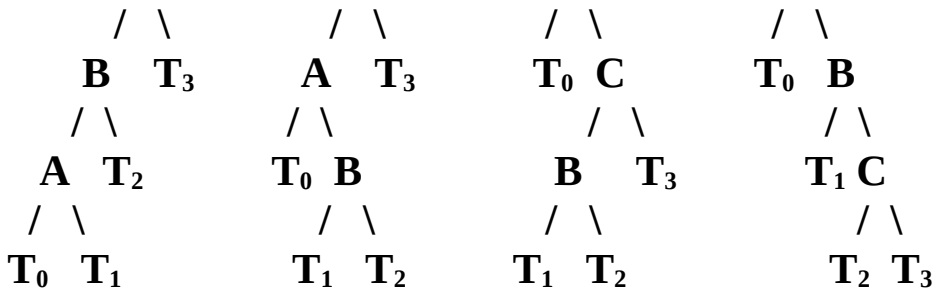
Now the question remains, how do we maintain an AVL tree? What extra work do we have to do to make sure that the AVL property is maintained?

Basically whenever an insertion or deletion is done, it is possible that the new node added or taken away destroys the AVL property. In these sitiuations, we have to "rework" the tree so that the binary search tree and AVL properties are satisfied.

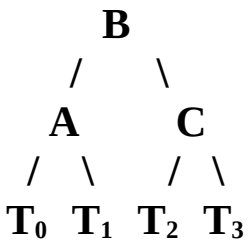
When an imbalance is introduced to a tree, it is localized to three nodes and their four subtrees. Denote these three nodes as A, B, and C, in their inorder listing. Structurally, they may appear in various configurations. Here are the four possibilities:



Denote the four subtrees as T_0, T_1, T_2, and T_3, also listed in their inorder listing. Here is where these would lie in the trees drawn above:



No matter which of these structural imbalances exist, they can all be fixed the same way:



Another way we can view these transformations is through two separate types or restructuring operations: a single rotation and a double rotation.

Let's look at how both of these work.

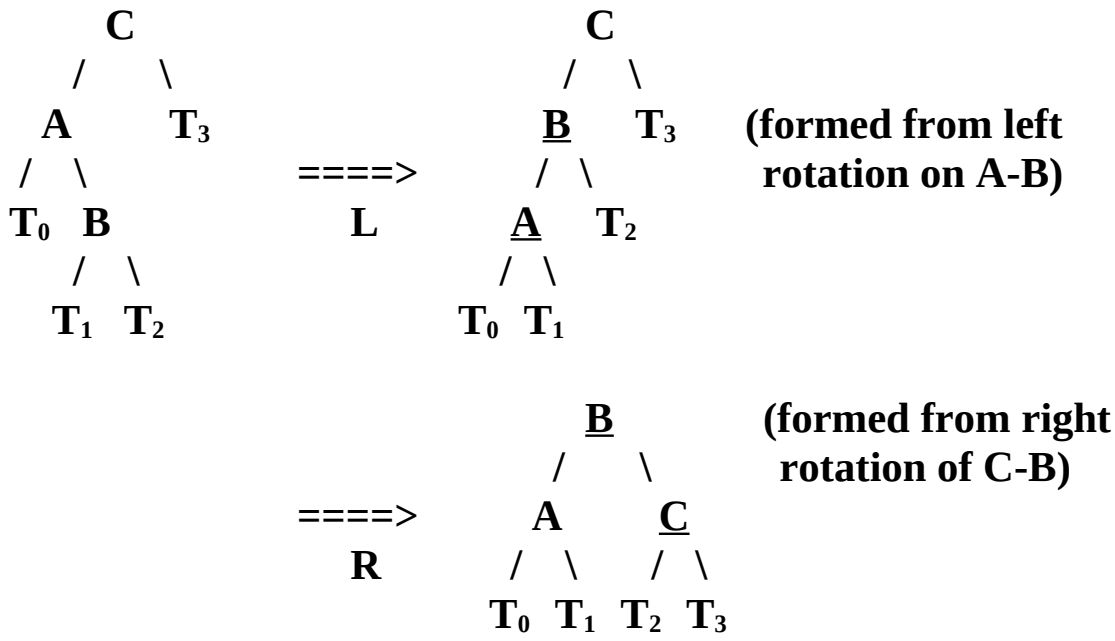
Here are the four cases we will look at:

1) insertion into the left subtree of the left child of the root.
2) insertion into the right subtree of the left child of the root.
3) insertion into the left subtree of the right child of the root.
4) insertion into the right subtree of the right child of the root.

Technically speaking, cases 1 and 4 are symmetric as are 2 and 3.

For cases 1 and 4, we will perform a single rotation, and for 2 and 3 we will do a double rotation.

In the pictures I have above, pictures are in the order 1, 2, 3, 4, from left to right. Why are case 2 and case 3 called double rotations? Because we can achieve both by performing two rotations on the root node:



Case 3 works symmetrically.

It should be fairly easy to see that case 1 visually looks like a "right rotation" and case 4 looks like a "left rotation".

Insertion into an AVL Tree

So, now the question is, how can we use these rotations to actually perform an insert on an AVL tree?

Here are the basic steps involved:

1) Do a normal binary tree insert.
2) Restoring the tree based on this leaf node.

This restoration is more difficult than just following the steps above. Here are the steps involved in the restoration of a node:

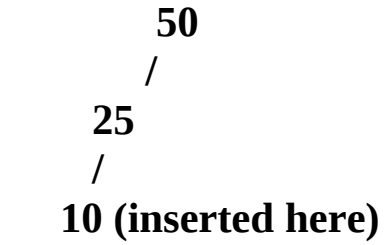
1) Calculate the heights of the left and right subtrees, use this to set the potentially new height of the node.
2) If they are within one of each other, recursively restore the parent node.
3) If not, then perform the appropriate restructuring described above on that particular node, THEN recursively call the method on the appropriate parent node.

Note: No recursive call is made if the node in question is the root node and has no parents.

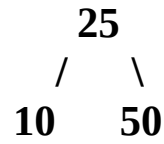
Also, one rebalancing will always do the trick, though we must make the recursive calls to move up the tree so that the heights stored at each node are properly recalculated.

AVL Tree Insert Examples

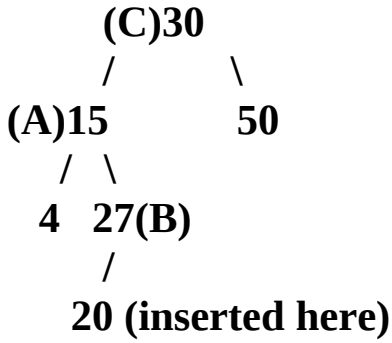
1) The most simple insert into an AVL Tree that causes a rebalance is inserting a third node into an AVL tree that creates a tree of height two. In this example, consider inserting the value 10:



After the insert, we trace up the tree, from 10 to 25 (which is also balanced), to 50. This node is unbalanced since the left subtree has height 1 and the right has height -1. The labels for the nodes are 50(C), 25(B), and 10(A). The restructure is as follows:

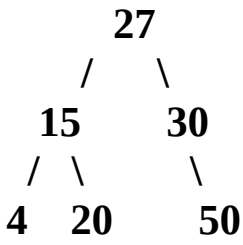


2) Consider inserting 20 into the following AVL Tree:

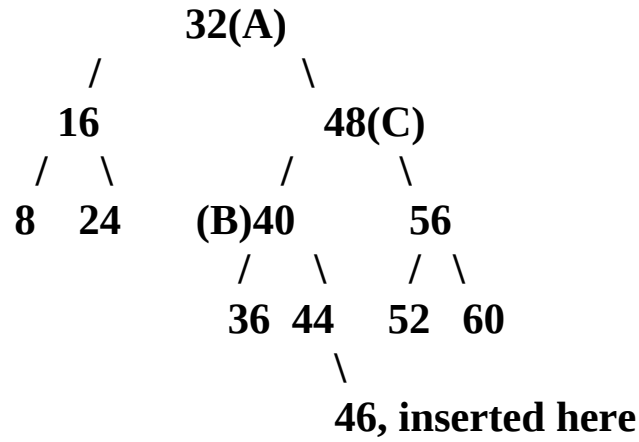


In this situation, the nodes 27 and 15 are balanced and we don't discover an imbalance until we trace up to 30. At this point, we label the nodes A, B and C based on our trace up the tree. The three values we passed were 27, 15 and 30, respectively. Thus, our labels are A = 15, B = 27, and C = 30.

Our resulting tree after rebalancing is as follows:



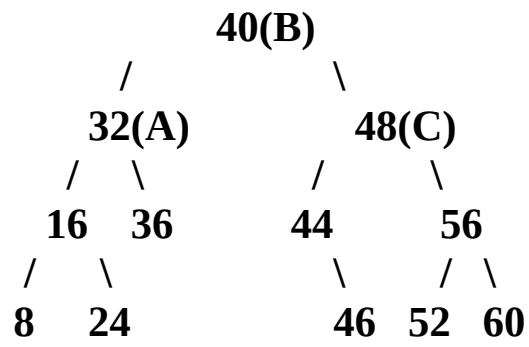
3) Consider inserting 46 into the following AVL Tree:



Initially, using the standard binary search tree insert, 46 would go to the right of 44. Now, let's trace through the rebalancing process from this place.

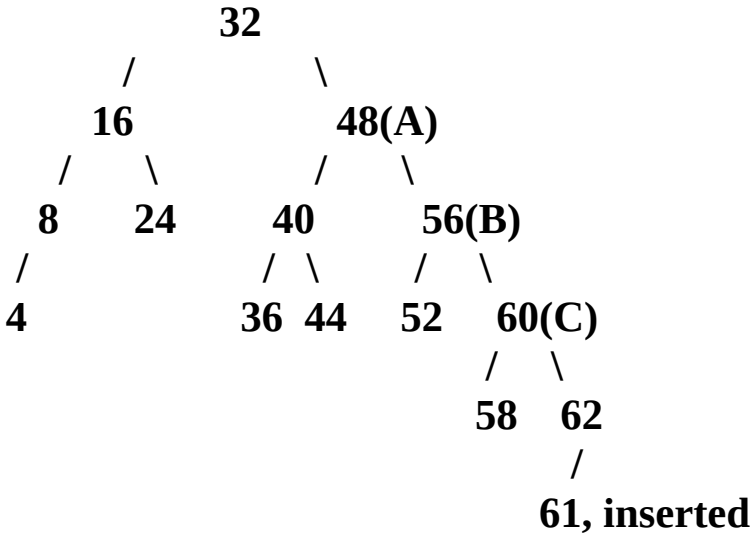
In this case, the node is balanced, so we march up to the parent node that stores 44. Then decide that the nodes storing 40 and 48 are balanced as well. Finally, when we reach the root node storing 32, we realize that our tree is imbalanced.

Now we identify A, B and C by looking at the last three nodes visited up the ancestral path. These are 40, 48 and 32, respectively. We have A = 32, B = 40 and C = 48. The corresponding restructuring is:

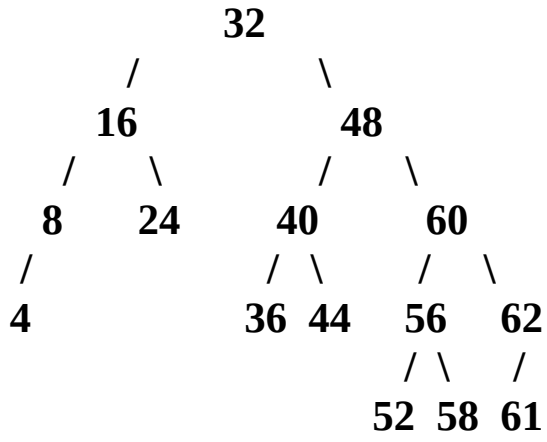


Using the variables from the last lecture, the node storing 40 is B, the node storing 32 is A, and the node storing 48 is C. T_0 is the subtree rooted at 16, T_1 is the subtree rooted at 36, T_2 is the subtree rooted at 44, and T_3 is the subtree rooted at 56.

4) Now, for the fourth example, consider inserting 61 into the following AVL Tree:



Tracing through the code, we find the first place an imbalance occurs tracing up the ancestry of the node storing 61 is at the noce storing 56. This time, we have that node A stores 56, node B stores 60, and node C stores 62. Using our restructuring algorithm, we find the tallest grandchild of 56 to be 62, and rearrange the tree as follows:



T_0 is the subtree rooted at 52, T_1 is the subtree rooted at 58, T_2 is the subtree rooted at 61, and T_3 is a null subtree.