





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 its 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 i th Fibonacci number.) To prove this, notice that the number of nodes in an AVL tree is the 1 plus the number of nodes 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.

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 $k \leq H$, assume that $S_k = F_{k+3} - 1$.

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

$$\begin{aligned} S_{k+1} &= S_k + S_{k-1} + 1 \\ &= (F_{k+3} - 1) + (F_{k+2} - 1) + 1, \text{ using the I.H. twice} \\ &= (F_{k+3} + F_{k+2}) - 1 \\ &= F_{k+4} - 1, \text{ using the defn. of Fibonacci numbers, to complete proof.} \end{aligned}$$

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?





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

$$n \approx (1.618)^h$$

$$h \approx \log_{1.618} n$$

$$h = O(\log_2 n)$$

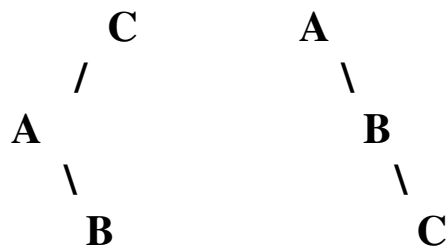


Now the question remains, how do we maintain an AVL tree?

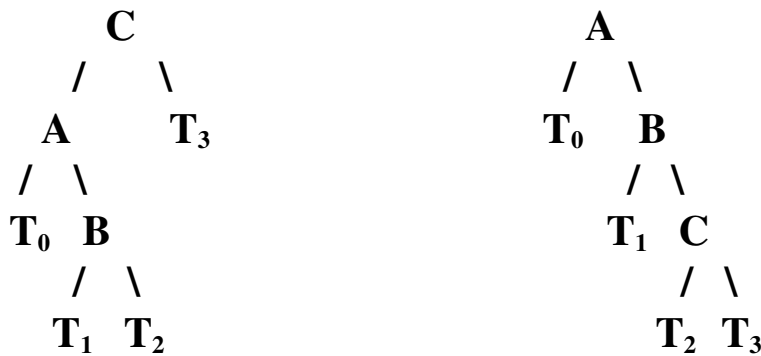
What extra work do we have to do to make sure that the AVL property is maintained?

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 situations, we have to "rework" the tree so that the binary search tree and AVL properties are satisfied.

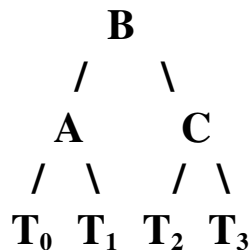
When an imbalance is introduced to a tree, it is localized to three nodes and their four subtrees. Denote these three nodes as A, B, and C, in their inorder listing. Structurally, they may appear in various configurations. A couple of these are listed below:



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:

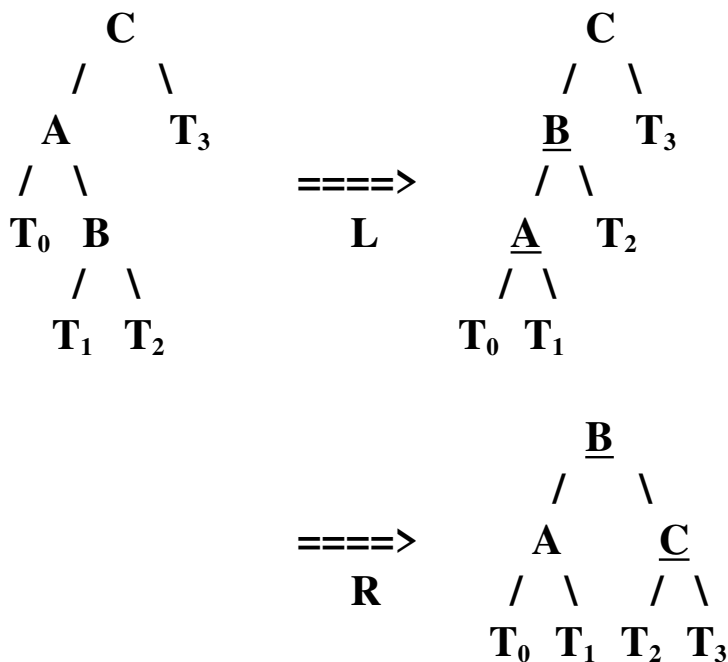
- 1) insertion into the left subtree of the left child of the root.
- 2) insertion into the right subtree of the left child of the root.
- 3) insertion into the left subtree of the right child of the root.
- 4) insertion into the right subtree of the right child of the root.

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, the left picture is case 2 of this description, and the right picture is case 4. Why is the case on the left called a double rotation? Because we can achieve it by performing two rotations on the root node:



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, just go up to the parent node and continue.
- 3) If not, then perform the appropriate restructuring described above on that particular node, THEN go to the parent node and continue.
- 4) Stop when you've reached the root node.

With insertion, we are guaranteed that we will at most rebalance the tree once.

 When we march up the tree, at each step we are updating the height of that node, if necessary. The only nodes that need to be updated are those on the ancestral "lineage" of the inserted node.

Deletion from an AVL Tree



First we will do a normal binary search tree delete. Note that structurally speaking, all deletes from a binary search tree delete nodes with zero or one child. For deleted leaf nodes, clearly the heights of the children of the node do not change. Also, the heights of the children of a deleted node with one child do not change either. Thus, if a delete causes a violation of the AVL Tree height property, this would HAVE to occur on some node on the path from the parent of the deleted node to the root node.



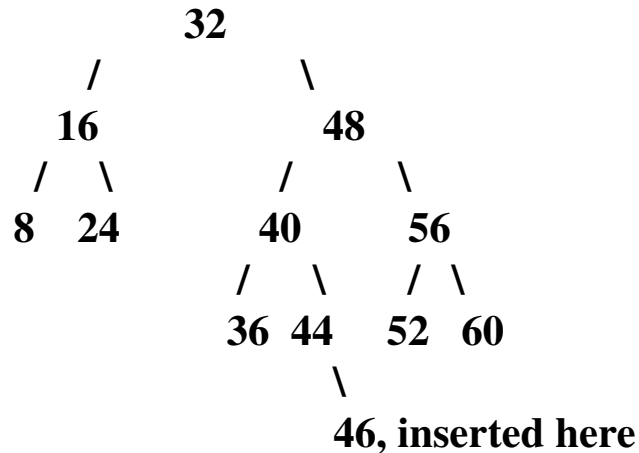
Thus, once again, as above, to restructure the tree after a delete we will call the restructure method on the parent of the deleted node. One thing to note: whereas in an insert there is at most one node that needs to be rebalanced, there may be multiple nodes in the delete that need to be rebalanced. Technically speaking, at any point in the restructuring algorithm ONLY one node will ever be unbalanced. But, what may happen is when that node is fixed, it may propagate an error to an ancestor node. But, this is NOT a problem because our restructuring algorithm goes all the way to the root node.



Let's trace through a couple examples each for inserts and deletes.

AVL Tree Examples

1) Consider inserting 46 into the following AVL Tree:



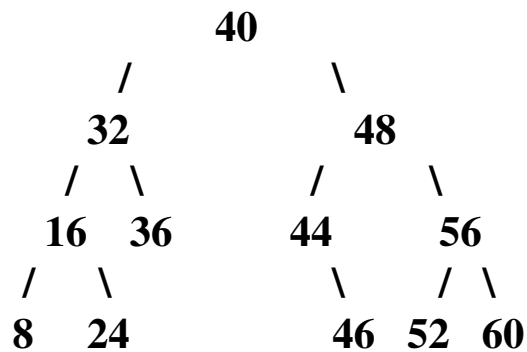
Initially, using the standard binary search tree insert, 46 would go to the right of 44. Now, let's trace through the rebalancing process from this place.

First, we call the method on this node. Once we set its height, we check to see if the node is balanced. (This simply looks up the heights of the left and right subtrees, and decides if the difference is more than 1.) In this case, the node is balanced, so we march up to the parent node, that stores 44.

We will trace through the same steps here, setting the new height of this node (this is important!) and determining that this node is balanced, since its left subtree has a height of -1 and the right subtree has a height of 0.

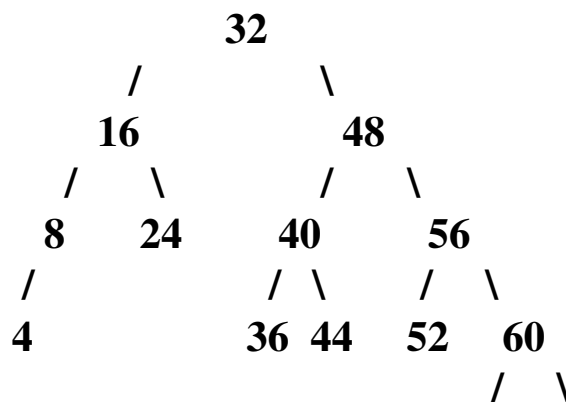
Similarly, we set the height and decide that the nodes storing 40 and 48 are balanced as well. Finally, when we reach the root node storing 32, we realize that our tree is imbalanced.

Now, we finally get to execute the code inside the if statement in the rebalance method. Here we set xPos to be the tallest grandchild of the root node. (This is the node storing 40, since its height is 2.) Thus, the restructuring occurs on the nodes containing the 32, 48 and 40. Using the method described from last lecture, we will restructure the tree as follows:



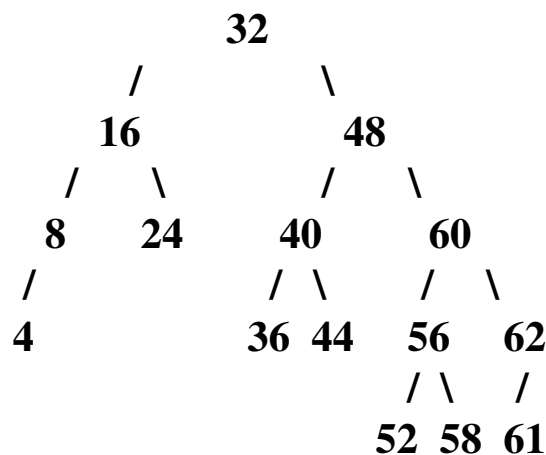
Using the variables from the last lecture, the node storing 40 is B, the node storing 32 is A, and the node storing 48 is C. T_0 is the subtree rooted at 16, T_1 is the subtree rooted at 36, T_2 is the subtree rooted at 44, and T_3 is the subtree rooted at 56.

2) Now, for the second example, consider inserting 61 into the following AVL Tree:



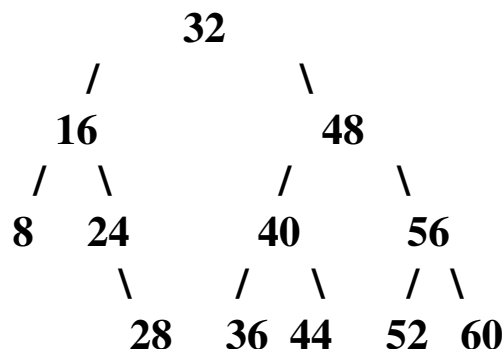
58 62
/
61, inserted

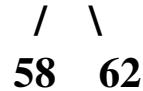
Tracing through the code, we find the first place an imbalance occurs tracing up the ancestry of the node storing 61 is at the node storing 56. This time, we have that node A stores 56, node B stores 60, and node C stores 62. Using our restructuring algorithm, we find the tallest grandchild of 56 to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



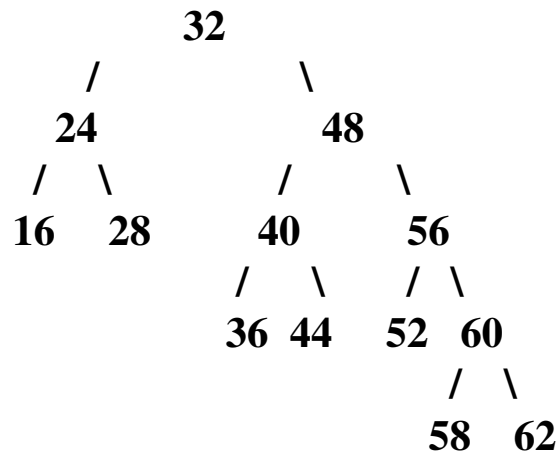
T_0 is the subtree rooted at 52, T_1 is the subtree rooted at 58, T_2 is the subtree rooted at 61, and T_3 is a null subtree.

3) *For this example, we will delete the node storing 8 from the AVL tree below:*



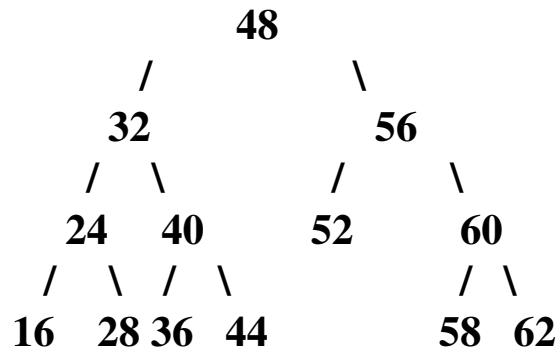


Tracing through the code, we find that we must first call the rebalance method on the parent of the deleted node, which stores 16. This node needs rebalancing and gets restructured as follows:

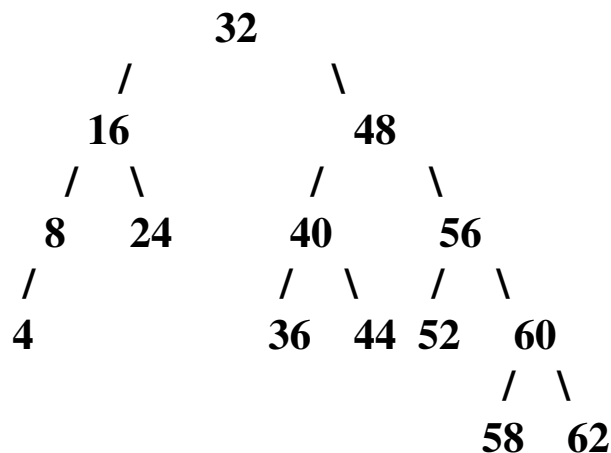


Notice that all four subtrees for this restructuring are null, and we only use the nodes A, B, and C. Next, we march up to the parent of the node storing 24, the node storing 32. Once again, this node is imbalanced. The reason for this is that the restructuring of the node with a 16 reduced the height of that subtree. By doing so, there was an INCREASE in the difference of height between the subtrees of the old parent of the node storing 16. This increase could propagate an imbalance in the AVL tree.

When we restructure at the node storing the 32, we identify the node storing the 56 as the tallest grandchild. Following the steps we've done previously, we get the final tree as follows:



4) The final example, we will delete the node storing 4 from the AVL tree below:



When we call rebalance on the node storing an 8, (the parent of the deleted node), we do NOT find an imbalance at an ancestral node until we get to the root node of the tree. Here we once again identify the node storing 32 as node A, the node storing

48 as node B and the node storing 56 as node C. Accordingly, we restructure as follows:

