CS234 MODULE 13 – AVL TREE

- AVL Property
- Why it’s balanced
- Tree Rotations
- Inserting and Removing

Last Updated 06-30
What is an AVL Tree?

An AVL Tree is a self-balancing BST. That means it’s a regular BST (the search function works the same) but add and remove ensure that $h \in O(\log n)$

Name is an Initialism for its creators:

- Georgy Adelson-Velsky
- Evgenii Landis
Balance

Remember: Balance means $h \in O(\log n)$

That’s not possible to prove with actual numbers, as for any two numbers $h$ and $n$, you can always find appropriate constants to show $h = c \log n$

We need a property that implies balance, but can be measured!
The AVL Property

An AVL tree is a BST with an extra property, the AVL property.

It says:
For all nodes V, $|\text{height}(V.\text{left}) - \text{height}(V.\text{right})| \leq 1$

Or, for any node in the tree, the heights of its two children differ by no more than 1.

(A3Q2B)
AVL Implies Balance

An AVL Tree is balanced because of the AVL property. But why?

Let’s consider $C(h)$, the minimum number of nodes you can fit in an AVL Tree with height $h$.

$C(0) = 0$, $C(1) = 1$, $C(2) = 2$

...  

$C(h) = C(h-1) + C(h-2) + 1$
Oh Boy, Recurrence Relation!

• Can we just look that up on the 116 reference sheet?
  • NO! (It’s not on there anyway)

Anyway, a popular way of “solving” recurrence relations is repeated substitutions, then waving your hands around and saying “obviously this is the pattern”.

Solving the Relation

\[ C(h) = C(h-1) + C(h-2) + 1 \]
\[ = C(h-2) + 2C(h-3) + C(h-4) + 3 \]
\[ = C(h-3) + 3C(h-4) + 3C(h-5) + C(h-6) + 7 \]
\[ = \text{BARF} \]
Redo

Since we only care about a lower bound, not the real value, we can be extra pessimistic!

\[ C(h) \geq 2C(h-2) \]

That we can solve!

\[ 2C(h-2) = 4C(h-4) = 8C(h-6) = \ldots = 2^{0.5h} \]

That’s the minimum, remember.
Bringing it Home

\[ n \geq 2^{0.5h} \Rightarrow \]
\[ \log_2 n \geq 0.5h. \Rightarrow \]
\[ h \leq 2 \log_2 n \Rightarrow \]
\[ h \in O(\log n) \]

In fact it’s a little bit better than that!
A Tighter Bound

\[ C(1) = 1, \quad C(2) > 1, \quad C(h) > C(h-1) + C(h-2) \]

OH SNAP, FIBONACCI!

\[ n \geq C(h) > \text{Fib}_h = \varphi^h / \sqrt{5} \]

\[ \Rightarrow h \leq \log_{\varphi} \sqrt{5} n \]

\[ = \log_2 n / \log_2 \varphi + \log_{\varphi} \sqrt{5} \]

\[ \sim 1.44 \log_2 n + 1.7 \]
OK, So?

An AVL Tree is a balanced BST. It is known.

Alright, but that’s only any good if we can make one for any sequence of values.

It’s also only any good if insert is still $O(h)$, and therefore $O(\log n)$. If not, who cares, there are other inefficient balancing algorithms.
Two Birds, One Stone

If we demonstrate a O(h) insert, we also demonstrate that we can create an AVL tree for any sequence of values: start off empty, and insert every value in the sequence.

Oh, an empty tree is an AVL tree as a vacuous truth. (It has no nodes, so we can make any claim we want about “every node” and not be wrong). That’s good.
Drawrings

To help visualize the AVL property, I’ll use symbols to indicate the balance factor of a node. The balance factor is the difference between the left and right subtree heights.

= means “left and right have same height”
< means “left has 1 less height” (right-heavy)
> means “left has 1 greater height” (left-heavy)
<< , >> mean “UH OH!!!”
Drawrings (cont)
Insertion

We begin the same way as with a regular BST.

Insert(42)
Two (Four) Possibilities

1. $16 > 8$
2. $16 = 16$
3. $16 < 23$
4. $16 = 23$
Two Possibilities

If the parent of the new node used to be < or >, it is now = and we did not change the height!

If the parent of the new node used to be =, it will now be < or >, and we added 1 to the height! That’s bad news (maybe)
Not Bad News

This is fine
Bad News
The Problem

When we added 1 to the height of a node, that node itself was fine. But its parent, maybe not.

If any of its ancestors were already unbalanced, we could have made it worse!
The Problem (Another Example)

Insert 6
Different Situations

<table>
<thead>
<tr>
<th>If node is…</th>
<th>And we go…</th>
<th>Then it might become…</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Left</td>
<td>&gt;</td>
</tr>
<tr>
<td>=</td>
<td>Right</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>Right</td>
<td>&lt;&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Left</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right</td>
<td>=</td>
</tr>
</tbody>
</table>
How To Solve a Problem Like

\[
\begin{align*}
8 & \quad \rightarrow \quad 7 \\
\downarrow & \quad \rightarrow \\
7 & \quad \rightarrow \\
\downarrow & \quad \rightarrow \\
6 & \quad \rightarrow \\
\end{align*}
\]

\[
\begin{align*}
7 & \quad \rightarrow \\
\downarrow & \quad \rightarrow \\
6 & \quad \rightarrow \\
\downarrow & \quad \rightarrow \\
8 & \quad \rightarrow \\
\end{align*}
\]
You What?

That’s called a “Rotation”. Specifically, a “right rotation” as I rotated the 7 to the right.
Is That Legal?

- I will make it legal!
- Doesn’t break order

Before: Everything in Subtree #2 is: Less than P, but greater than C
After: Everything in Subtree #2 is: Greater than C, but less than P.
How Does it Fix Balance?

It only works if C is > and P is ≫

Why?
Mirror, Mirror

If C is < and P is <= then:
C is the right child of P instead of the left, and we can do a left rotation. Everything is the same, but mirrored.
What About the Other Ancestors?

In those two cases (out of four) they are all fixed
Why Did That Happen?

Before the insert: P’s height was h+2

After the insert + rotation: C’s height is h+2

Therefore: All ancestor’s will be back to their original balance factor
The Zigzag of Doom

• Rotating 4 will **not** fix the problem this time
I Heard You Like Rotations

![Diagram of AVL tree rotations]

- Left rotation
- Right rotation
- Double rotation
Change in Height

If P’s height was h+1 before insert (h+2 after): G’s height after rotation is h+1
## Rotation Cases

<table>
<thead>
<tr>
<th>P is</th>
<th>C is</th>
<th>Action</th>
<th>New Root (of subtree)</th>
<th>Change in Height (vs before insert)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>&lt;</td>
<td>Rotate C LEFT</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;</td>
<td>Rotate (C.Left) RIGHT first, then LEFT</td>
<td>C.Left</td>
<td>0</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>&lt;</td>
<td>Rotate (C.Right) LEFT first, then Right</td>
<td>C.Right</td>
<td>0</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>&gt;</td>
<td>Rotate C RIGHT</td>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
Meaning?

Once you rotate, there are no more issues above you in the tree.

Not just no more issues, but no more changes to balance factors.
Procedure

Use a **recursive** version of insert

When the recursion unwinds, look for the case where the current node was < and became <<, or was > and became >>

If that happens, rotate.

(It’s more trouble than its worth to communicate upward that no more changes are possible).
Code

I wrote this ahead of time. It worked the third time, I promise!
Removal

There’s nothing new to say here:

If you remove a value, you might decrease the height of that subtree.

You’ll have the same 4 cases, and the same 4 fixes.

Code: Recursive remove, rebalance during the unwind.
Rebalance: Complexity

A rotation is $O(1)$: Look at the code and tell me it isn’t!

The recursion itself is $O(h) = O(\log n)$

During unwind, we, at worst, do two rotations, $O(1)$ extra work per step => No change to the complexity.

Therefore: Insert and remove still $O(\log n)$