Warm-Up

An AVL tree is a binary search tree that also meets the following rule:

**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

pollEV.com/cse373su19

Is this a valid AVL tree?

Height of a tree:
Maximum number of edges on a path from the root to a leaf.
A tree with one node has height 0
A null tree (no nodes) has height -1
Are These AVL Trees?
Avoiding the Degenerate Tree

An AVL tree is a binary search tree that also meets the following rule:

**AVL condition**: For every node, the height of its left subtree and right subtree differ by at most 1.

This will avoid the $\Theta(n)$ behavior! We have to check:
1. We must be able to maintain this property when inserting/deleting
2. Such a tree must have height $O(\log n)$. 
Let $m(h)$ be the minimum number of nodes in an AVL tree of height $h$.

If we can say $m(h)$ is big, we’ll be able to say that a tree with $n$ nodes has a small height.

So...what’s $m(h)$?

$$m(h) = \begin{cases} 
1 & \text{if } h = 0 \\
2 & \text{if } h = 1 \\
(m(h - 1) + m(h - 2) + 1 & \text{otherwise}
\end{cases}$$
A simpler recurrence

Hey! That’s a recurrence!

Recurrences can describe any kind of function, not just running time of code!

\[
m(h) = \begin{cases} 
1 & \text{if } h = 0 \\
2 & \text{if } h = 1 \\
m(h - 1) + m(h - 2) + 1 & \text{otherwise}
\end{cases}
\]

We could use tree method, but it’s a little...weird.

It’ll be easier if we change things just a bit:

\[
m(h) \geq \begin{cases} 
1 & \text{if } h = 0 \\
2 & \text{if } h = 1 \\
m(h - 2) + m(h - 2) + 1 & \text{otherwise}
\end{cases}
\]
Answer the following questions:
1. What is the size of the input on level \( i \)?
2. What is the work done by each node on the \( i^{th} \) recursive level?
3. What is the number of nodes at level \( i \)?
4. What is the total work done at the \( i^{th} \) recursive level?
5. What value of \( i \) does the last level occur?
6. What is the total work across the base case level?

\[
m(h) \geq \begin{cases} 
1 & \text{if } h = 0 \\
2 & \text{if } h = 1 \\
(m(h - 2) + m(h - 2) + 1) & \text{otherwise}
\end{cases}
\]
Tree Method Practice

1. What is the size of the input on level $i$?  $h - 2i$

2. What is the work done by each node on the $i^{th}$ recursive level?

3. What is the number of nodes at level $i$?  $2^i$

4. What is the total work done at the $i^{th}$ recursive level?  $2^i \cdot 1$

5. What value of $i$ does the last level occur?

   $$h - 2i = 0 \Rightarrow i = \frac{h}{2}$$

6. What is the total work across the base case level?  $2^{h/2}$

$$m(h) \geq \sum_{i=0}^{\frac{h-1}{2}} 2^i + 2^{h/2} = 2^{h/2} - 1 + 2^{h/2}$$

$$m(h) \geq 2^{h/2}$$
Finishing the Argument

Suppose I give you an AVL tree of height $h$, how many nodes, $n$ can it have?

Every AVL tree of height $h$ has at least $m(h)$ nodes (that’s how $m()$ was defined)

So $n \geq m(h) \geq 2^{h/2}$

This inequality is true for all AVL trees of height $h$. So we can reverse the logic, i.e. it’s still true to say “if we have $n$ nodes, what can the height be? It must satisfy:

$n \geq 2^{h/2}$

$\log_2 n \geq h/2$

$h \leq 2 \log_2 n$     AVL trees are always short, just like we wanted!
Avoiding the Degenerate Tree

An AVL tree is a binary search tree that also meets the following rule

**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

This will avoid the $\Theta(n)$ behavior! We have to check:

1. We must be able to maintain this property when inserting/deleting
2. Such a tree must have height $O(\log n)$.
What happens if when we do an insertion, we break the AVL condition?
Left Rotation

Rest of the tree

UNBALANCED
Right subtree is 2 longer

Rest of the tree

BALANCED
Right subtree is 1 longer
Left subtree whose height differs by at most 1

Somewhat

-Perfectly balanced...

...As all things should be
Right rotation

Just like a left rotation, just reflected.
There’s a “kink” in the tree where the insertion happened.

Can’t do a left rotation
Do a “right” rotation around 3 first.

Now do a left rotation.
Right Left Rotation

Rest of the tree

UNBALANCED
Right subtree is 2 longer

Left subtree is 1 longer

BALANCED
Right subtree is 1 longer
Four Types of Rotations

<table>
<thead>
<tr>
<th>Insert location</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left subtree of left child (A)</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>Right subtree of left child (B)</td>
<td>Double (left-right) rotation</td>
</tr>
<tr>
<td>Left subtree of right child (C)</td>
<td>Double (right-left) rotation</td>
</tr>
<tr>
<td>Right subtree of right child (D)</td>
<td>Single left rotation</td>
</tr>
</tbody>
</table>
AVL Example: 8,9,10,12,11
AVL Example: 8, 9, 10, 12, 11
AVL Example: 8,9,10,12,11
AVL Example: 8, 9, 10, 12, 11
AVL Example: 8,9,10,12,11
How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree. How do we find an unbalanced node?

How many rotations might we have to do?
How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?
- Just go back up the tree from where we inserted.

How many rotations might we have to do?
- Just a single or double rotation on the lowest unbalanced node.
- A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion.
Deletion

There is a similar set of rotations that will always let you rebalance an AVL tree after a deletion.

The textbook (or Wikipedia) can tell you more.

We won’t test you on deletions, beyond the following facts:
- Deletion is similar to insertion.
- It takes $\Theta(\log n)$ time on a dictionary with $n$ elements.
- We won’t ask you to perform a deletion.
Avoiding the Degenerate Tree

An AVL tree is a binary search tree that also meets the following rule:

**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

This will avoid the $\Theta(n)$ behavior! We have to check:

1. We must be able to maintain this property when inserting/deleting.
2. Such a tree must have height $O(\log n)$.
Three Asides

Three related topics, we’ll cover what we can:

Lazy deletion
How fast is $\Theta(\log n)$
Traversals
Aside: Lazy Deletion

Deleting things is hard. Let’s be lazy.

Instead of actually removing elements from a data structure add a flag for whether the element is “really there.”

Then when you call delete, just find the item you’re looking for, and set the flag.

Should you “really delete” or just lazy delete?

This sounds like a design decision! What are the tradeoffs?
Aside: Lazy Deletion

Pros:
Much easier to write delete function.
Delete takes the same time as find (much faster for our data structures that shift everything!)

Cons:
All code now has to check that entries are really there
Running time of all operations is now in terms of elements ever inserted, not number of elements currently in the data structure.

Takeaways:
General technique (not just for dictionaries).
But only use it if you have a good reason.
Aside: How Fast is $\Theta(\log n)$?

If you just looked at a list of common running times

<table>
<thead>
<tr>
<th>Class</th>
<th>Big O</th>
<th>If you double N...</th>
<th>Example algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>Add to front of linked list</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log n)$</td>
<td>Increases slightly</td>
<td>Binary search</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
<td>doubles</td>
<td>Sequential search</td>
</tr>
<tr>
<td>“n log n”</td>
<td>$O(n \log n)$</td>
<td>Slightly more than doubles</td>
<td>Merge sort</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
<td>quadruples</td>
<td>Nested loops traversing a 2D array</td>
</tr>
</tbody>
</table>

You might think this was a small improvement.
It was a HUGE improvement!
Logarithmic vs. Linear

If you double the size of the input,
- A linear time algorithm takes twice as long.
- A logarithmic time algorithm has a constant \texttt{additive} increase to its running time.

To make a logarithmic time algorithm take twice as long, how much do you have to increase \( n \) by?

You have to square it \( \log(n^2) = 2\log(n) \).

A gigabyte worth of integer keys can fit in an AVL tree of height 60.

It takes a ridiculously large input to make a logarithmic time algorithm go slowly. Log isn’t “that running time between linear and constant” it’s “that running time that’s barely worse than a constant.”
This identity is so important, one of my friends made me a cross-stitch of it.

Two lessons:
1. Log running times are REALLY REALLY REALLY FAST.
2. $O(\log(n^3))$ is not simplified, it’s just $O(\log n)$
Aside: Traversals

What if the heights of subtrees were corrupted. How could we calculate from scratch?

We could use a “traversal”
- A process that visits every piece of data in a data structure.

```java
int height(Node curr){
    if(curr==null) return -1;
    int h = Math.max(height(curr.left),height(curr.right));
    return h+1;
}
```
Three Kinds of Traversals

InOrder(Node curr) {
    InOrder(curr.left);
    doSomething(curr);
    InOrder(curr.right);
}

PreOrder(Node curr) {
    doSomething(curr);
    PreOrder(curr.left);
    PreOrder(curr.right);
}

PostOrder(Node curr) {
    PostOrder(curr.left);
    PostOrder(curr.right);
    doSomething(curr);
}
Traversal Practice

For each of the following scenarios, choose an appropriate traversal:

1. Print out all the keys in an AVL-Dictionary in sorted order.

2. Make a copy of an AVL tree

3. Determine if an AVL tree is balanced (assume height values are not stored)
Traversals

If we have \( n \) elements, how long does it take to calculate height? \( \Theta(n) \) time.

The recursion tree (from the tree method) IS the AVL tree!

We do a constant number of operations at each node

In general, traversals take \( \Theta(n \cdot f(k)) \) time, where \( \text{doSomething}() \) takes \( \Theta(f(k)) \) time.

Common question on technical interviews!
Aside: Other Self-Balancing Trees

There are lots of flavors of self-balancing search trees.

“Red-black trees” work on a similar principle to AVL trees.

“Splay trees”
- Get $O(\log n)$ amortized bounds for all operations.

“Scapegoat trees”

“Treaps” – a BST and heap in one (!)

Similar tradeoffs to AVL trees.

Wednesday: A completely different idea for a dictionary

Goal: $O(1)$ operations in practice, in exchange for $\Theta(n)$ worst case.