AVL Trees

AVL Trees are a type of **balanced binary search tree**.

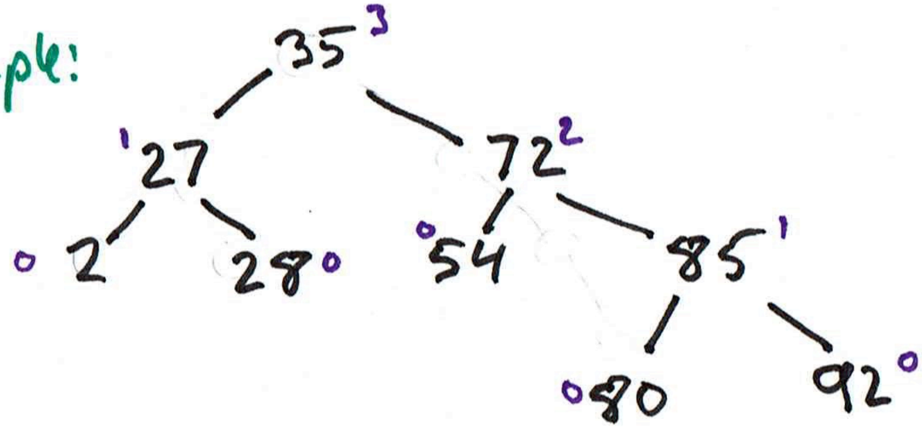
Named after its inventors, Adel'son-Vel'skii and Landis.

Start with a basic binary search tree and add a balance condition, the height-balance property.

The Height-Balance Property

For every internal node v of a tree T, the heights of the children of v differ by at most one.

Example:



Height is small, purple number.

At each node, the left and right heights differ by at most one.

Theorem: The height of an AVL tree storing n entries is $O(\log n)$.

Recall that in a BST, search, insertion, and deletion are all $O(h)$. Therefore, for an AVL tree these operations are all $O(\log n)$.

This is, asymptotically, the best we can hope for from a BST.

Sketch of Proof

Let $n(h)$ be the minimum number of internal nodes for an AVL tree w/height h.

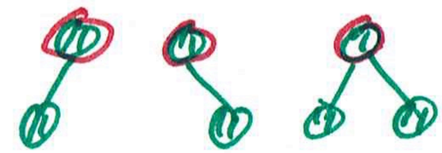
Examples:

$\boxed{h=0}$

No internal nodes.

$n(0) = 0$

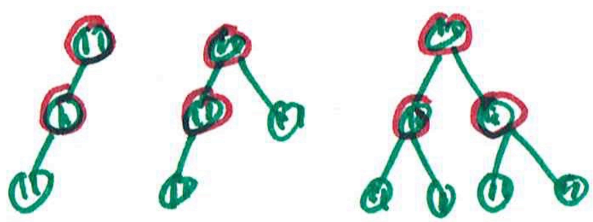
$\boxed{h=1}$



One internal node.

$n(1) = 1$

$\boxed{h=2}$

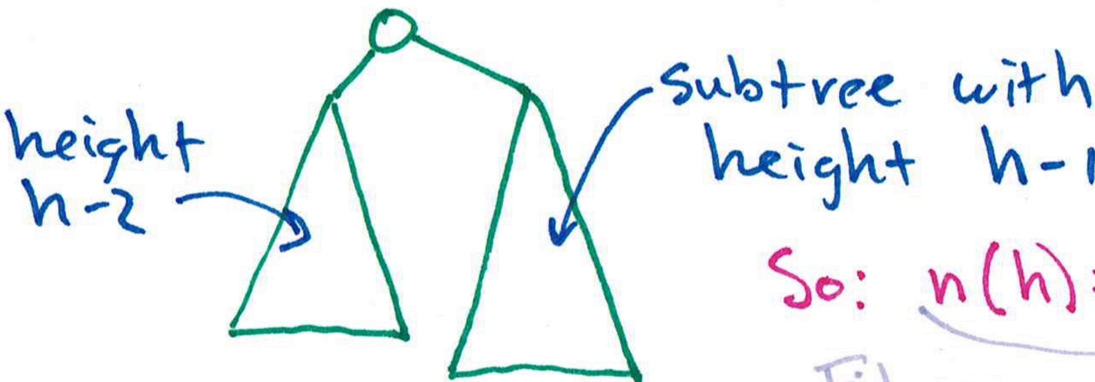


etc.

At least two internal nodes

$n(2) = 2$

Proof (cont.) Now, suppose $h \geq 3$. The tree of height h with the least number of internal nodes must look something like this:



So: $\underbrace{n(h) = 1 + n(h-1) + n(h-2)}_{\text{Fibonacci sequence!}}$

Why?

* At least one subtree must have height $h-1$ for the tree to have height h.
* If both had height $h-1$, we could shrink one.

Proof Conclusion ...

Prove by Induction: $n(h) > 2^{h/2 - 1}$

Then $\log(n(h)) > \frac{h}{2} - 1$

$h < 2 \log n(h) + 2$

And so h is $O(\log n(h))$.

Since the total number of nodes, n, is $\Theta(n(h))$, ~~so~~ we conclude that

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

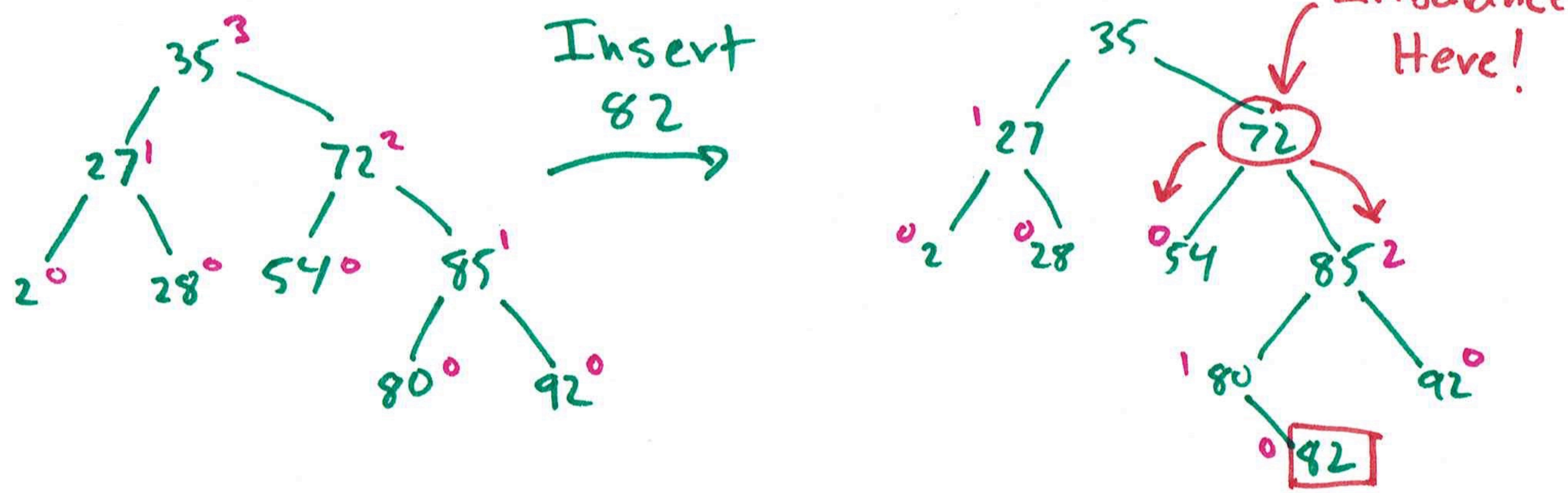
this is like "Q.E.D." $\blacksquare$

Insertion in an AVL Tree

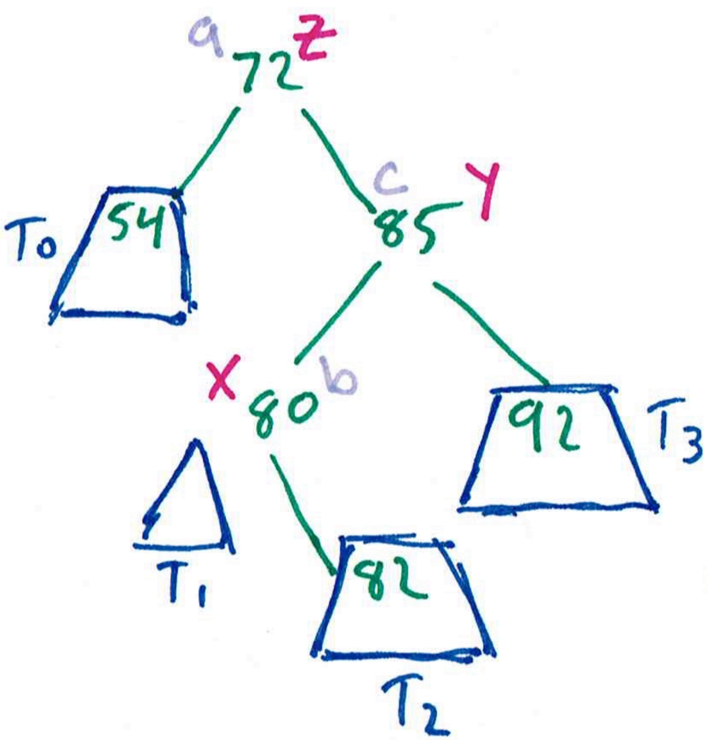
Suppose we have an AVL Tree and we want to insert a value

Insertion might break the height-balance property!

Example:



Trinode Restructure



z: location of imbalance

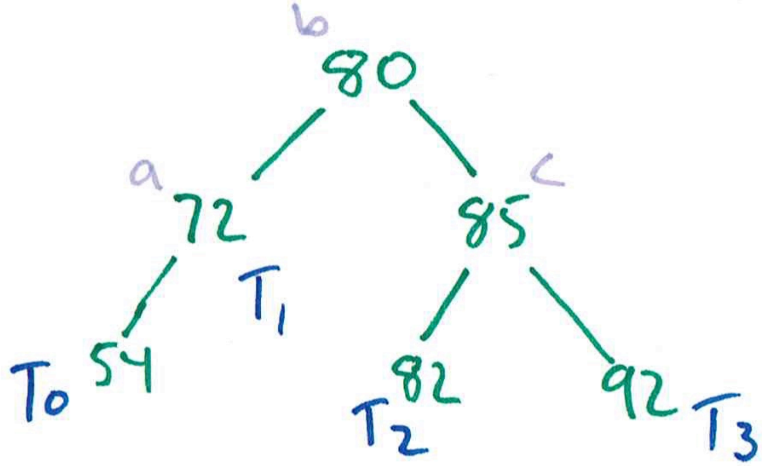
y: child of z with greater height

x: child of y with greater height

a, b, c: inorder labeling of x, y, z.

T_0, T_1, T_2, T_3: inorder labeling of subtrees of x, y, or z.

Trinode Restructure (cont.)



* Replace z with b.
* Make a left child of b; T_0, T_1 left and right subtrees of a
* Make c right child of b; T_2, T_3 left and right subtrees of c.

Imbalance is fixed!

It's a little tricky... takes practice.

Deletion from an AVL Tree

Deleting a node from an AVL tree can also cause an imbalance.

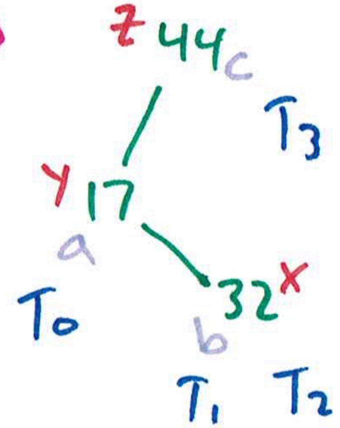
We can use trinode restructuring to fix these imbalances.

One wrinkle: imbalances may propogate <u>up</u> the tree: we may have to do multiple trinode restructures.

Example: build the AVL tree from the values 44, 17, 32, 78, 48, 88, 62, 50.

1

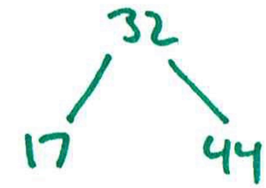
2



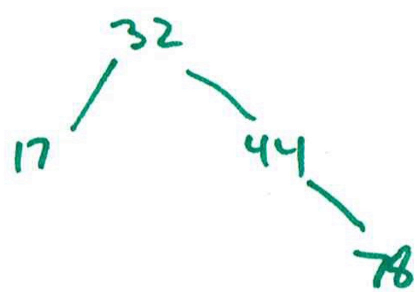
Imbalance at 44

T_0, T_1, T_2 T_3 are trivial

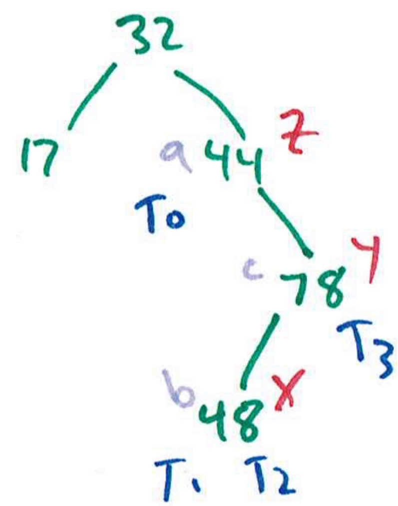
restructure



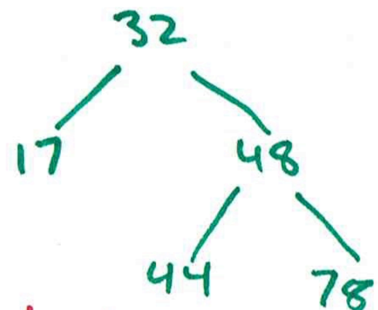
3



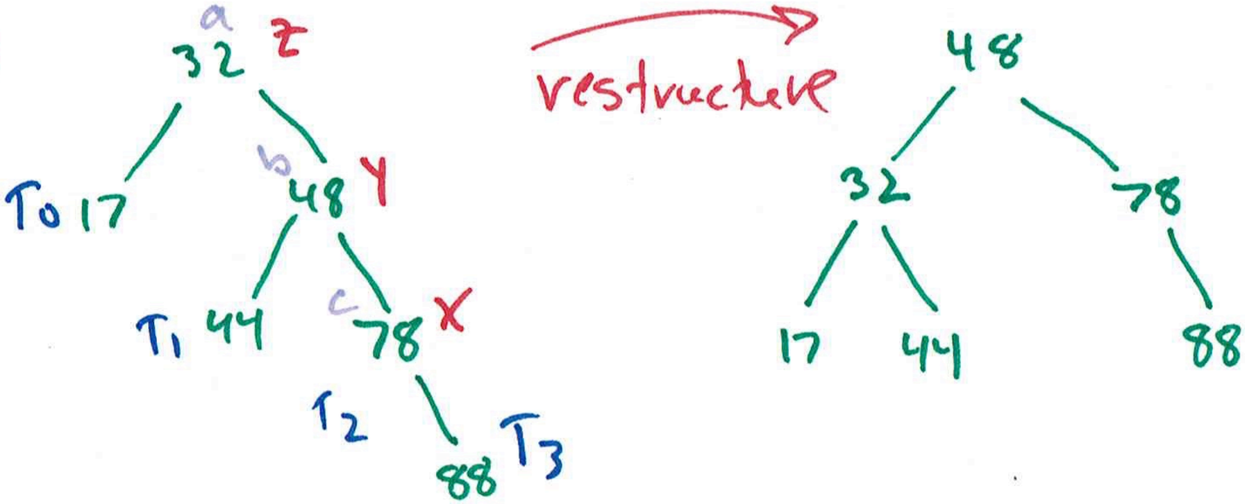
4



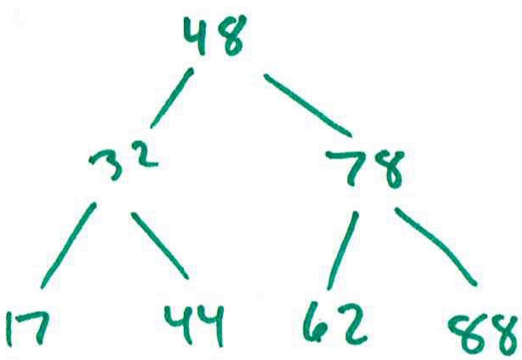
restructure



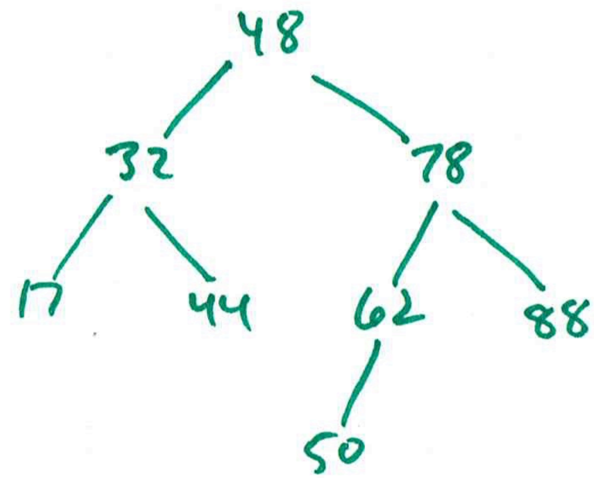
5



6

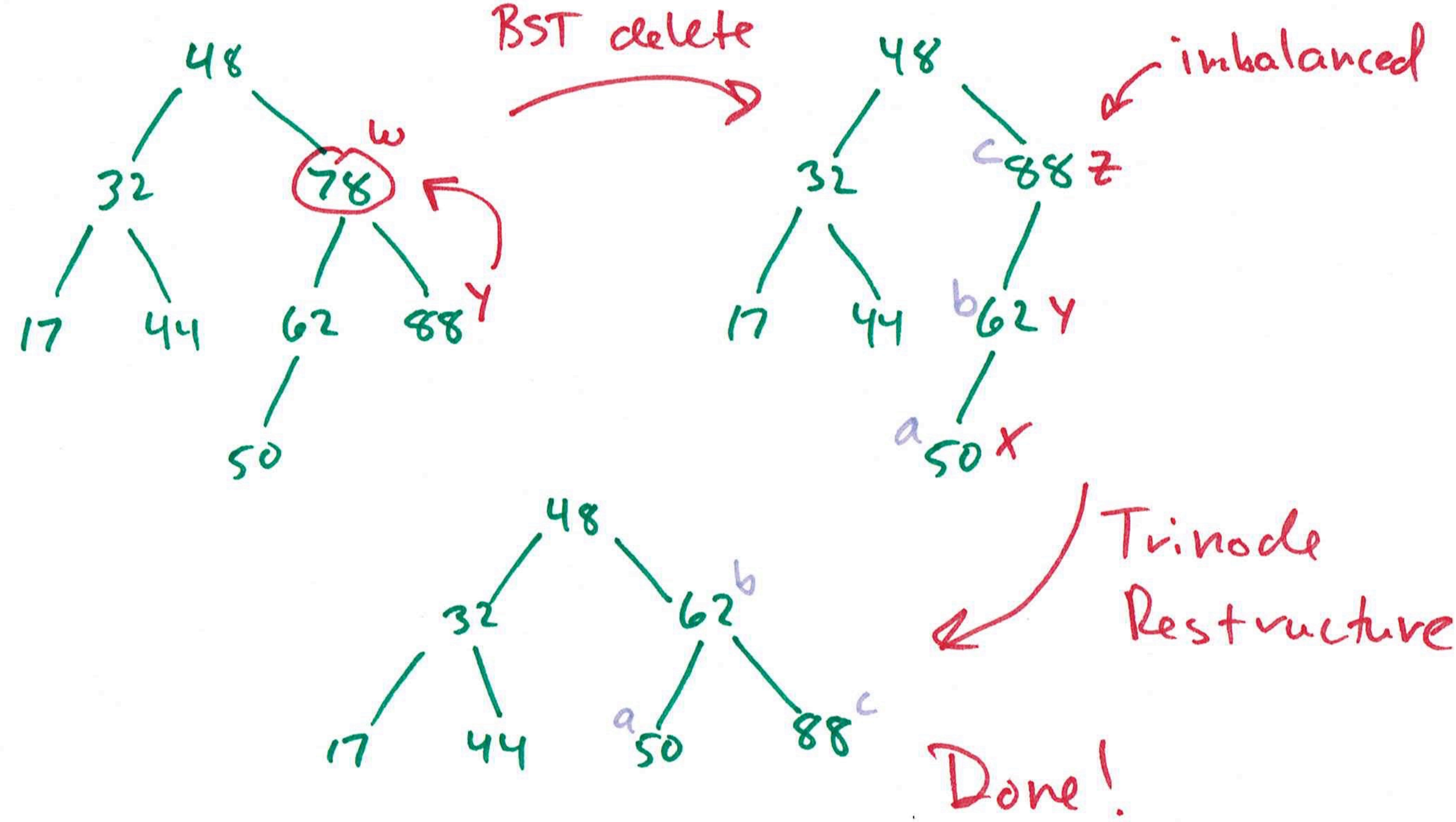


7



Done! Balanced!!

Now delete 78: start with regular BST deletion.



Running Times

Trinode Restructure is $O(1)$.

search: descend tree from root: at most $O(\log n)$ levels, so $O(\log n)$.

insert: descend tree from root; insert; at most $O(\log n)$ restructures, so $O(\log n)$.

delete: same idea; descend; BST delete; at most $O(\log n)$ restrucs. So, $O(\log n)$.