# AVL Trees

CMPUT 275 - Winter 2018

University of Alberta

## Balancing A Binary Search Tree

When we insert or delete, we will do just a bit of extra work to help maintain **balance** in the tree.

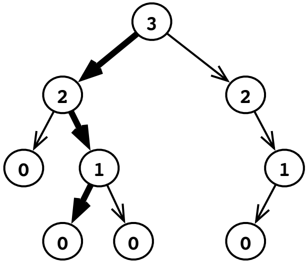
Roughly speaking, a tree is **balanced** when approximately the same number of nodes in either subtree of any node.

If so, then we expect the maximum height to be $O(\log n)$.

Let's make this precise!

## AVL Trees

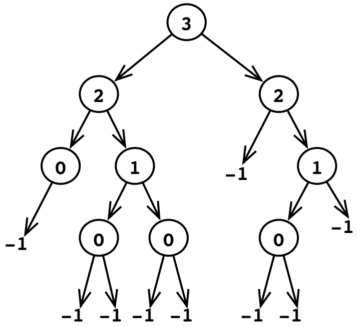
The **height** of any node $v$ is the maximum number of edges between $v$ and a leaf in its subtree. (**Depicted**: numbers $\equiv$ heights)



The depicted path determines the root's height. Leaves have height 0.

Let's refer to this as $v.height$ for each node $v$. We will explicitly maintain this quantity at each node in our data structure.

## AVL Trees



**Again**: numbers indicate height, not keys.
Say `null`.*height* is -1.

So $v.\mathit{height} = 1 + \max\{v.\mathit{left}.\mathit{height}, v.\mathit{right}.\mathit{height}\}$ if $v \neq \texttt{null}$.

## AVL Trees

We say a node $v$ has the **AVL property** if

$$|v.left.height - v.right.height| \leq 1.$$

**In words**
The children of $v$ have almost the same height (off by at most 1).

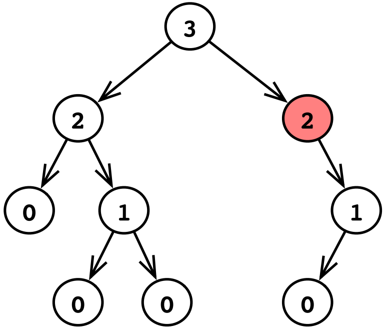
A binary search tree is an **AVL tree** if **all** nodes have the AVL property.

Named for the two Soviet scientists who discovered it in 1962: Adelson-Velsky and Landis.

## AVL Trees

This is not an AVL tree. The highlighted vertex has

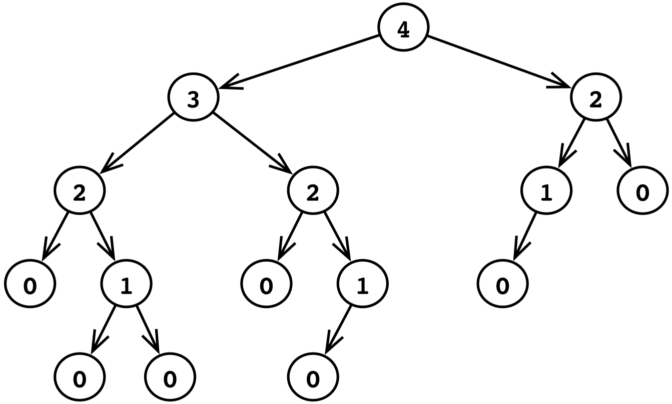
$$|v.left.height - v.right.height| = |(-1) - (1)| = 2.$$



(`null` nodes not depicted)

## AVL Trees

This is an AVL tree (once again, numbers indicate heights, not keys).

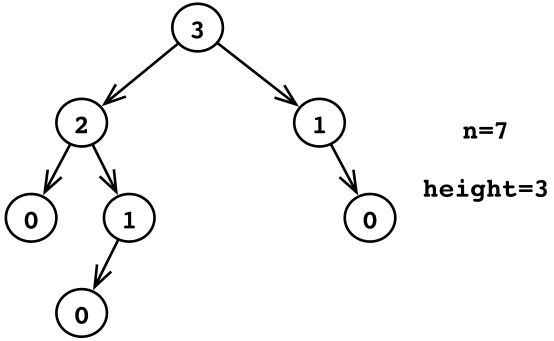


## AVL Trees Are Balanced

**Claim**: The height of any AVL tree is logarithmic!

This would follow if any node $v$ has (# nodes under $v$) $\geq 2^{v.height}$.

Unfortunately, this is not true for every AVL tree.



## AVL Trees Are Balanced

But it is still true that

$$\#\text{ nodes under } v \geq c^{v.height}$$

for some constant $c > 1$. But what constant $c$?

**Let's discover it by assuming it is true**!

$$\begin{aligned}(\#\text{ nodes under } v) &= 1 + (\#\text{ nodes under } v.left \textbf{ and } v.right) \\ &\geq c^{v.left.height} + c^{v.right.height} \\ &\geq c^{v.height-1} + c^{v.height-2}\end{aligned}$$

The last bound holds because $v$ has the AVL property.

$$(\#\text{ nodes under } v) \geq c^{v.height-1} + c^{v.height-2}$$

We want this to be $\geq c^{v.height}$.

For this, we must have:

$$c^{-1} + c^{-2} \geq 1$$

a.k.a. $c^2 \leq c + 1$.

By the quadratic formula, this holds for $c := \frac{1+\sqrt{5}}{2} \approx 1.618$.

**Conclusion**: $(\#\text{ nodes under } v) \geq c^{v.heght}$.
(Technically one should prove it by induction on $v.height$ and verify explicitly it holds for heights 0 and 1).

Thus, the height of an AVL tree with $n$ nodes is $O(\log n)$.

## Summary

If we ensure our tree always has

$$|v.left.height - v.right.height| \leq 1$$

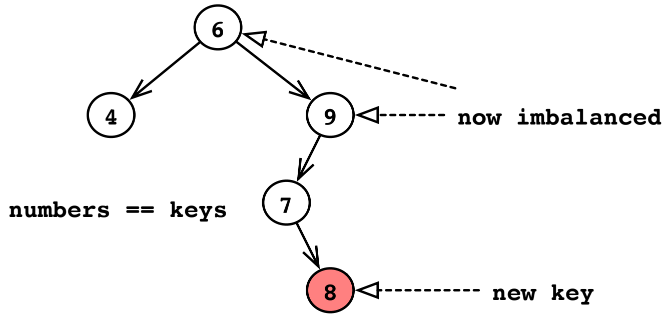
for each vertex $v \in V$, then the height is bounded by $O(\log n)$.

Thus, searching for a key takes $O(\log n)$ time!

We will slightly modify the insertion and deletion routines to fix up any violation of the AVL property after changing the tree.

## Maintaining an AVL Tree

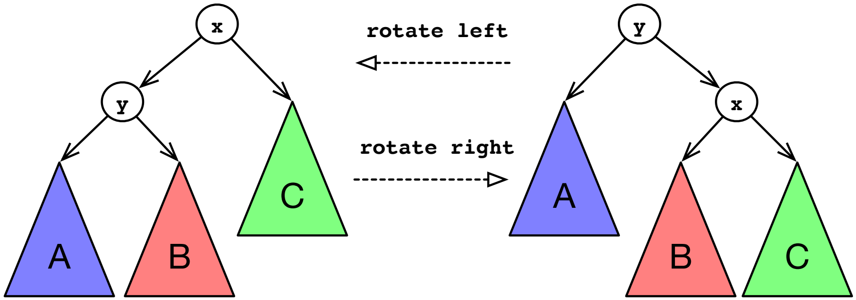
If we insert or delete a node using the standard Binary Search Tree algorithm, we may destroy the AVL property at some node.



The heights of some nodes change, perhaps causing an imbalance of $\pm 2$ on some nodes between the new node and the root.

## Rotations

We use **rotations** to rebalance. There are two types (that are symmetries of each other).



## Rebalancing

Now let's see how to fix an imbalanced node after an insertion.

**Assumptions** (satisfied by the imbalanced nodes after insertion)

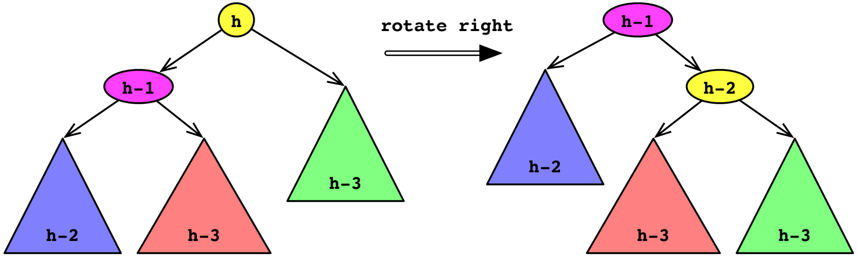
- $\mathit{balance} := |v.\mathit{left}.\mathit{height} - v.\mathit{right}.\mathit{height}| = 2$.
  That is, the balance of $v$ is only off by 1 from the AVL property.
- Every other node in the subtree under $v$ is balanced.
  They all have the AVL property.

In $O(1)$ time we can fix the balance of this subtree using rotations so all nodes in this subtree have the AVL property.

## Rebalancing

Suppose $v's$ balance is off by 2 and $v.height = v.left.height + 1$.

**Subcase #1**: $v.left.left.height \geq v.left.right.height$
Then right-rotate $v$.

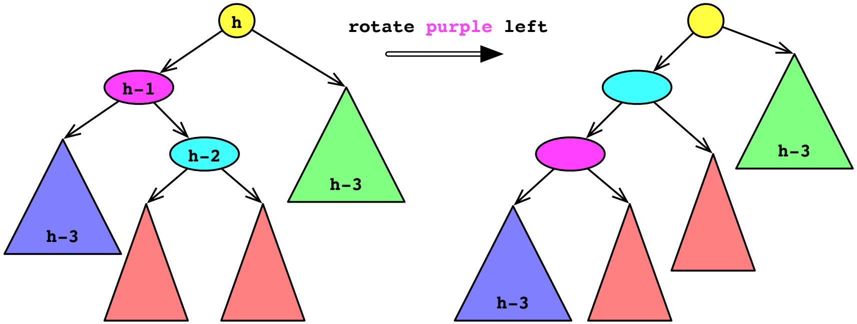


**Not depicted**: It could be the red subtree has height $h-2$, in which case the root keeps height $h$ but the AVL property is still fixed.

## Rebalancing

Suppose $v's$ balance is off by 2 and $v.height = v.left.height + 1$.

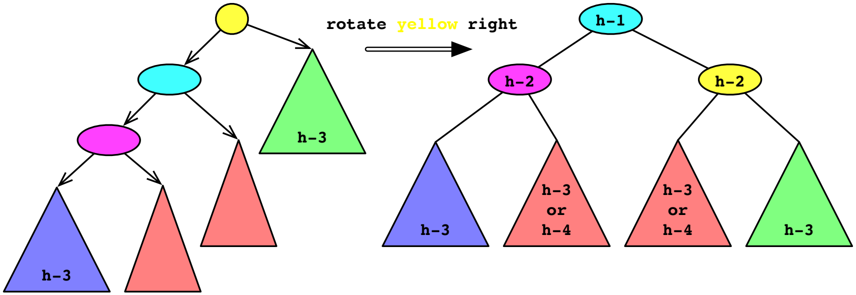
**Subcase #2**: $v.left.left.height < v.left.right.height$
First left-rotate $v.left$.



## Rebalancing

Suppose $v's$ balance is off by 2 and $v.height = v.left.height + 1$.

**Subcase #2**: $v.left.left.height < v.left.right.height$
Then right-rotate $v$.



The case where $v.right.height = v.height - 1$ is symmetric. So just a brief summary.

**Subcase #1**: $v.right.right.height \geq v.right.left.height$
Left-rotate $v$.

**Subcase #2**: $v.right.right.height < v.right.left.height$
First right-rotate $v.right$ and then left-rotate $v$.

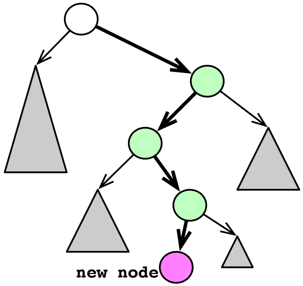
**In Any Case**
Don't forget to update the appropriate child pointer of the node that originally pointed to $v$. The rotations change the root of the subtree.

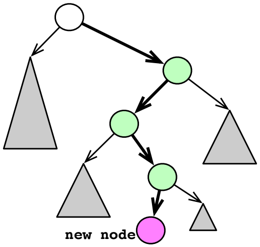
**A Better Reference**
The code we will see has all cases explicitly handled. Also, a TA is creating an **AVL Quick-Reference Sheet** which will be posted soon.

## Putting It Together

Insert the new key using standard binary search tree insertion.

The use rotations to fixed imbalanced nodes (depicted in green): from the bottom-up.

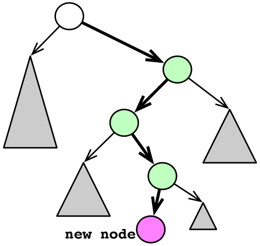




**Algorithm 1** Rebalancing after inserting a new node $v$.

```
while $v \neq$ null do
    $par \leftarrow parent(v)$
    if $|balance(v)| = 2$ then
        rebalance $v$ using rotations          # as described earlier
    $v \leftarrow par$
```

What invariants let us see why this actually works?



In each iteration of the pseudocode, let $P_v$ denote the path between the current node $v$ and the root. The following can be verified.

**Invariants**

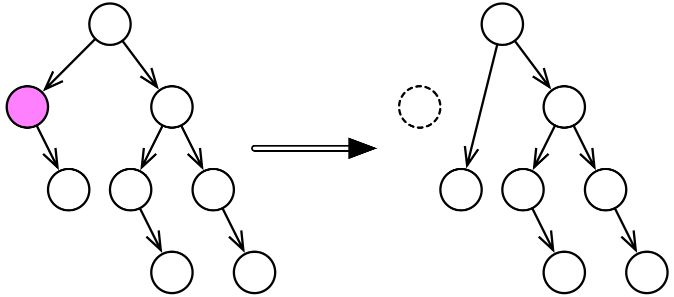
- Every node in the subtree under $v$ has the AVL property except, perhaps, $v$.
- Any imbalanced nodes lies on $P_v$ and has imbalance of $\pm 2$.
- If $u \in P_v$ is imbalanced, its highest child also lies on $P_v$.

## Removal

Removing a node is similar.

Recall there were two cases:

**Case 1**

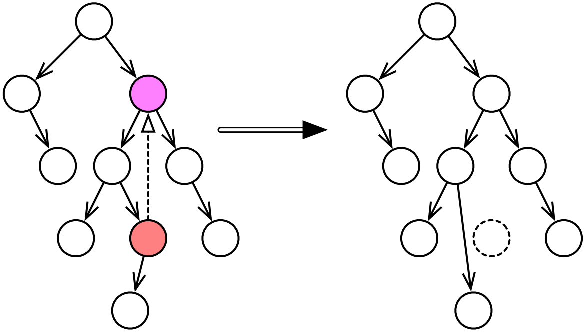


## Removal

Removing a node is similar.

Recall there were two cases:

**Case 2**



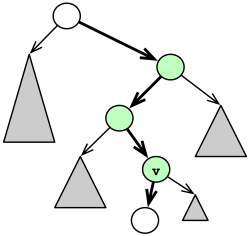
## Removal

In either case, let $v$ be the parent of the node that was deleted (which may or may not have been the node with the key).

**Note**

- All nodes in the subtree under $v$ except, perhaps, $v$ itself have the AVL property.
- The height of $v$ may have decreased, but only by 1.
- Any node whose height decreases lies on $P_v$ (path from $v$ to the root).
- If $u \in P_v$ is now imbalanced, its balance is $\pm 2$ and the shorter child also lies on $P_v$.

The green nodes are the imbalanced ones.



Recall $v$ is the parent of the node that was just deleted.

**Algorithm 2** Rebalancing after deleting a key.

```
while $v \neq$ null do
    $par \leftarrow parent(v)$
    if $|balance(v)| = 2$ then
        rebalance $v$ using rotations                # as described earlier
    $v \leftarrow par$
```

The properties from last slide are invariants that hold throughout.

## Implementation Considerations

Recall that after rotating from $v$ that $v$ is no longer the root of this subtree. So you should also update the child pointer from $v$'s old parent (or the root pointer for the entire tree itself if $v$ was the root).

The trees we are using do not explicitly store parent pointers. This is burdensome to maintain and takes extra space.

**Cute Solution**
Insertions and removals should use recursion and the recursion stack will hold the parents of the vertices found in the search.

See the implementation for details.

## Summary

Before balancing using AVL trees, binary search tree operations took $O(\mathit{height})$ time, which could be as bad as $O(n)$.

With AVL trees, we guarantee the heights remain logarithmic, specifically $\mathit{height} \leq \log_c(n)$ where $c = \frac{1+\sqrt{5}}{2}$.

More work is needed to maintain this property, but this overhead runs in $O(\log n)$ time. Ultimately, all searches, insertions, and removals take $O(\log n)$ time in the worst case.

**Further Reading (optional)**
**Red-Black** trees are another way to balance binary search trees. They use a completely different approach. The ordered sets and maps in the `C++` standard template library use them.

## Further Discussion

Memory is smaller with AVL trees. Hash tables need noticeable empty space in order to perform well.

Some reports that balanced trees work faster in practice than hash tables if items are constantly added and removed (not just a lookup table), especially when hashing is a bit expensive (long strings).

Can simultaneously manage a min-heap and a max-heap with a balanced tree: find min and find max each take $O(\log n)$ time.