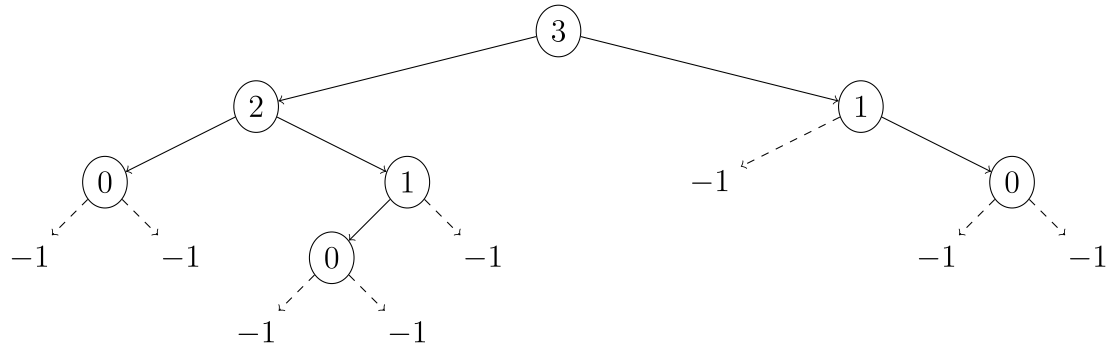
CMPUT 275 - Tangible Computing **Winter 2018**
Quick-Reference Sheet: AVL Trees

1 AVL Property

In a tree, the **height** of any node `v`—denoted `v.height`—is the maximum number of edges between `v` and any leaf of its subtree. For a `null` node, we will say that `null.height=-1`.



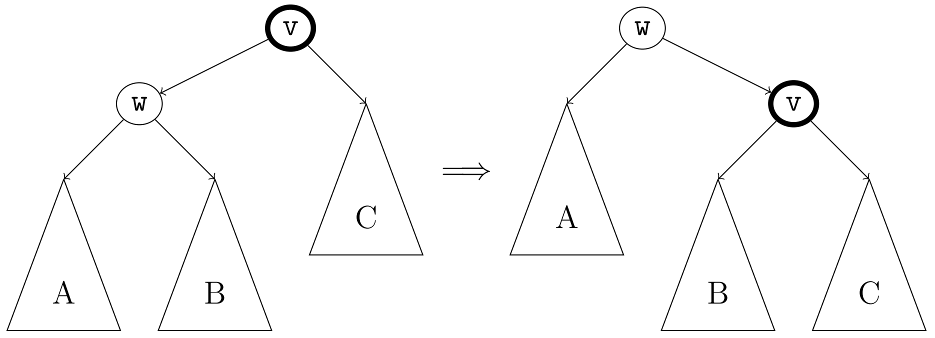
A tree node `v` is said to have the **AVL property** if

$$|\texttt{v.left.height} - \texttt{v.right.height}| \leq 1$$

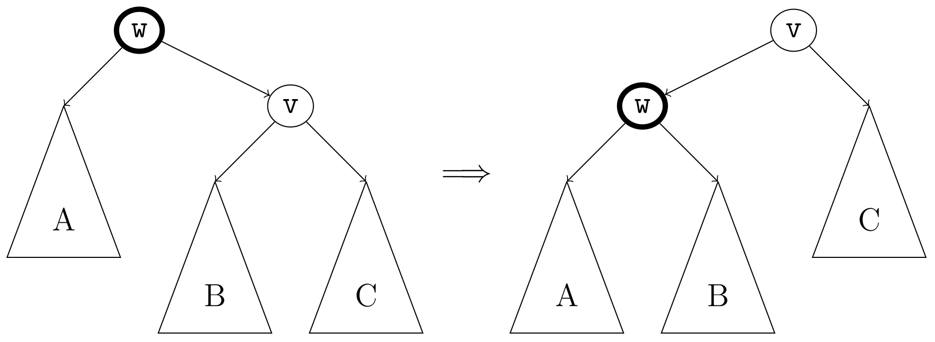
and the tree itself is an AVL tree if all nodes have this property (like the tree above).

2 Rebalancing

To **right-rotate** a tree node `v` pushes `v` to its right child with the middle subtree:



To **left-rotate** a tree node `w` is to do the opposite: push `w` to its left child with the middle subtree:

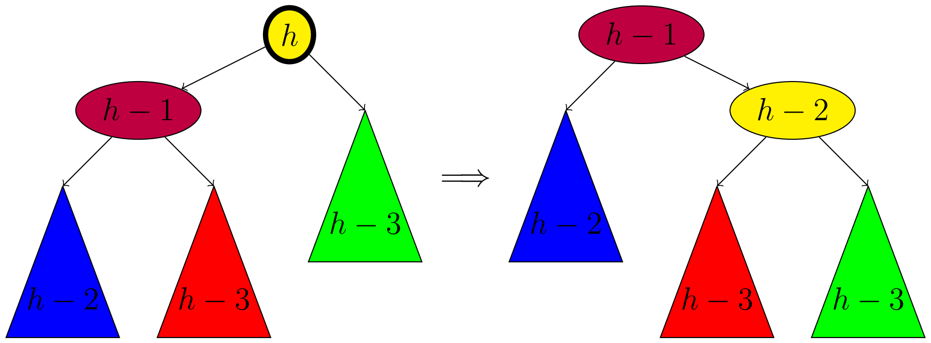


We will use these to adjust unbalanced nodes `v`. We can assume that the balance of `v` is off by exactly 2, then there are several cases:

Case 1: `v.height` $=$ `v.left.height` $+1$

Subcase 1.1: `v.left.left.height` $\geq$ `v.left.right.height`

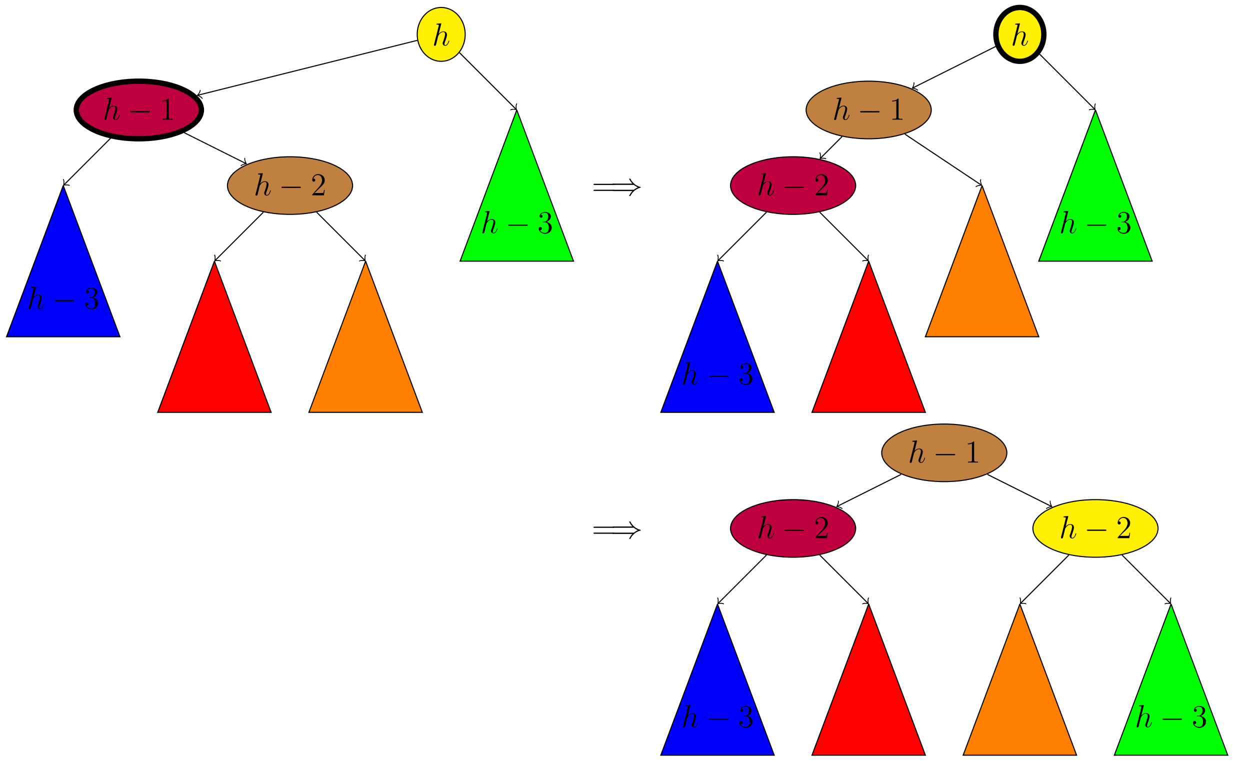
In this case, we right-rotate `v` (yellow):



(note that the red subtree could also have height $h-2$, which still causes no problem and would give the new root height h)

Subcase 1.2: `v.left.left.height` $<$ `v.left.right.height`

In this case, we left-rotate `v.left` (purple), and then right-rotate `v` (yellow):

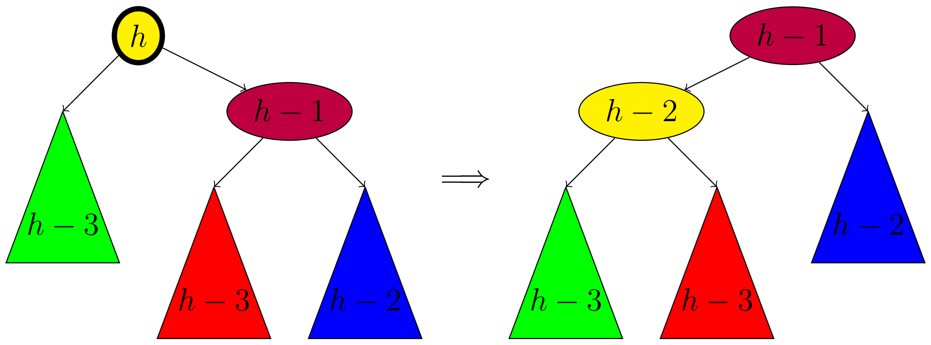


Case 2: `vheight` $=$ `v.right.height` $+1$

This is symmetric to case 1.

Subcase 2.1: `v.right.right.height` $\geq$ `v.right.left.height`

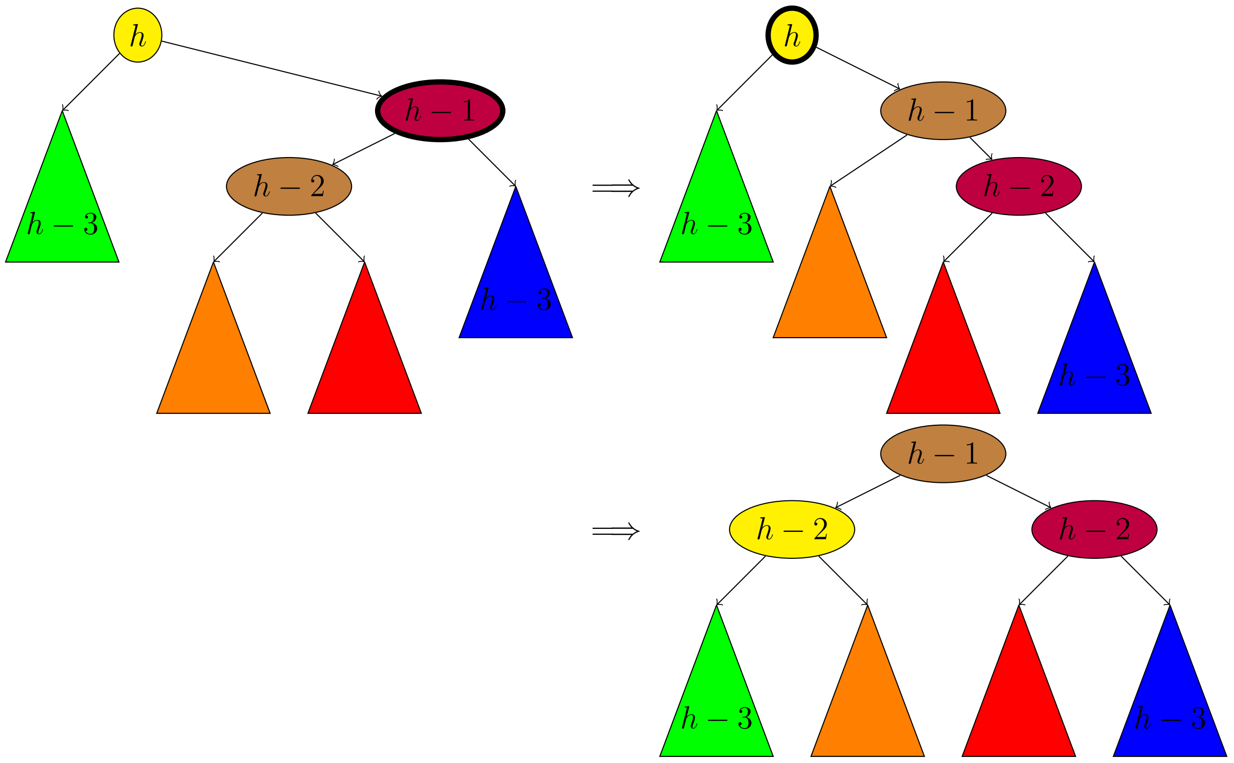
In this case, we left-rotate `v` (yellow):



(as before, note that the red subtree could also have height $h-2$, but this is fine, and the new root will have height h)

Subcase 2.2: `v.right.right.height` $<$ `v.right.left.height`

In this case, we right-rotate `v.right` (purple), then left-rotate `v` (yellow):



3 Insertion and Deletion

The aforementioned case-by-case analysis allows you to quickly rebalance a tree after inserting a new key or deleting a key. Perform insertion/deletion how you would normally do it for a binary search tree first. Now, starting at the parent of the altered node—for insertion, this is the parent of the inserted node; for deletion, this is the parent of the deleted node—backtrack up the tree the same way you entered. If at any point there is an imbalance, use the above procedure to rebalance the subtree and then continue backtracking.