# AVL Trees

## Summary

AVL trees are Binary Search Trees, with the additional AVL balance condition to maintain the balanced property of the tree. When doing an insertion or deletion in the tree, we do some rotations to maintain the balance property. Because we do the rotations as we insert and delete nodes, we can assume some nice things about the height of the tree, which allows us to analyze the runtime of the operations in worst case.

**We didn't cover AVL deletion or the proof by induction** that proves the height of the AVL tree is worst case log(N) where N is the number of nodes in the tree, but we use that fact to make claims about the runtimes of the insert, delete, and find operations on an AVL tree. We did talk about the insertion into an AVL tree and how to maintain the balance condition when inserting. There are resources linked from the lectures section of the website if you're curious about how deletion in an AVL tree works. It is similar to insert, but requires more rotations.

## AVL Balance Condition

**Left and right subtrees of *every node* have heights differing by at most 1**

This condition, checked on insert and delete, requires that we keep track of the height of every node in the tree. We can maintain that as we insert and delete from the tree. We use the height to check that at **every node** the heights of the subtrees differ by at most 1. The height of a node is defined as the path from that node to the bottom of the tree (the leaves). Leaves have a height of 0. You can think of null subtrees as having a height of -1, or you can have a special case for it in your code to deal with null subtrees.

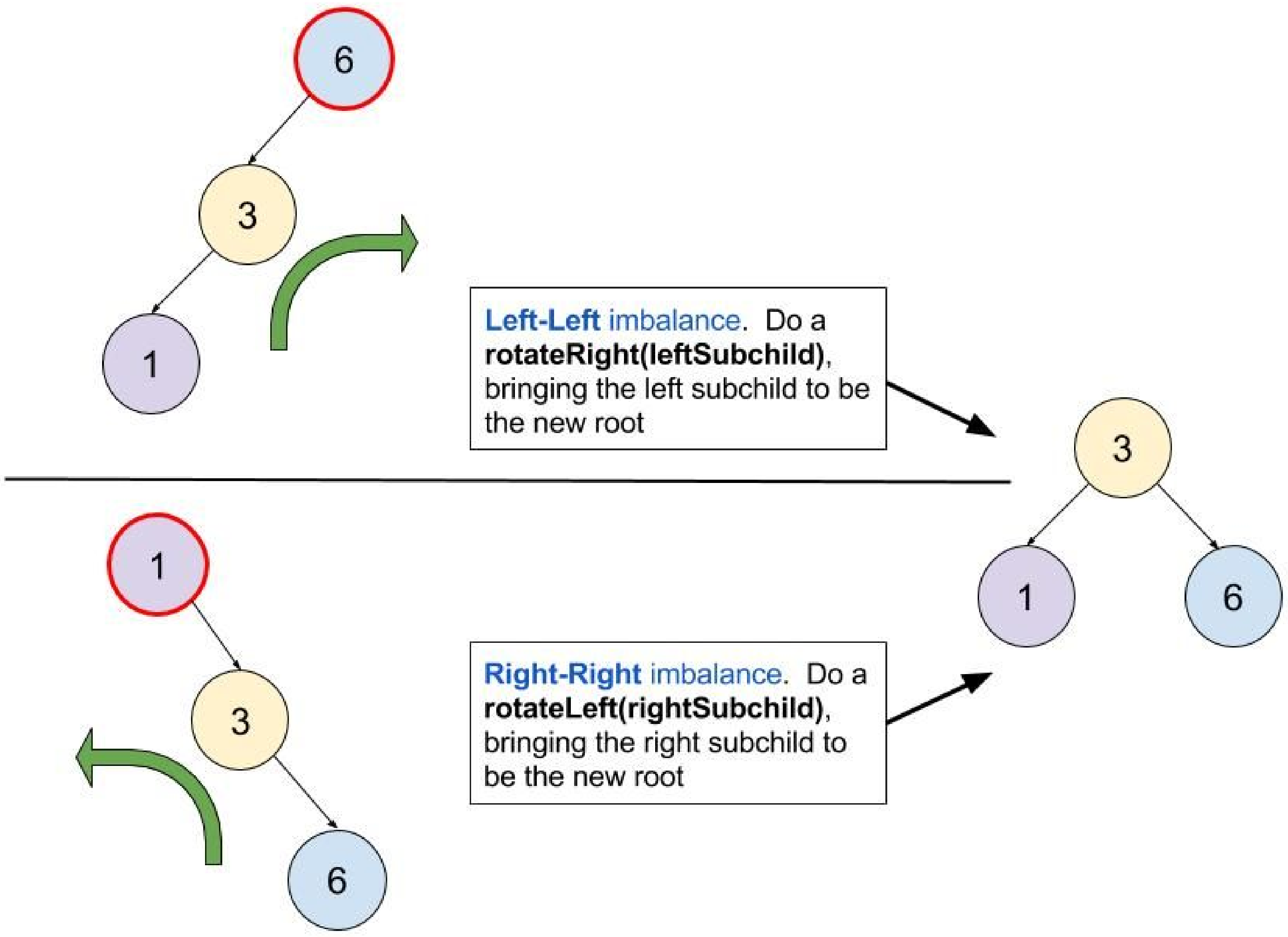
## Find and Insert

**Find:** To insert, or delete, we have to find where the element belongs in the tree. We use the BST property of the tree to traverse to where the node should exist if it were in the tree. The BST property says we go left for values less than the root, and right for values greater than the root. Your BST can deal with duplicates arbitrarily. Since the height of the AVL tree is log(N) based on the balance property and inductive reasoning, we can argue that the find operation costs **O(log(N))**.

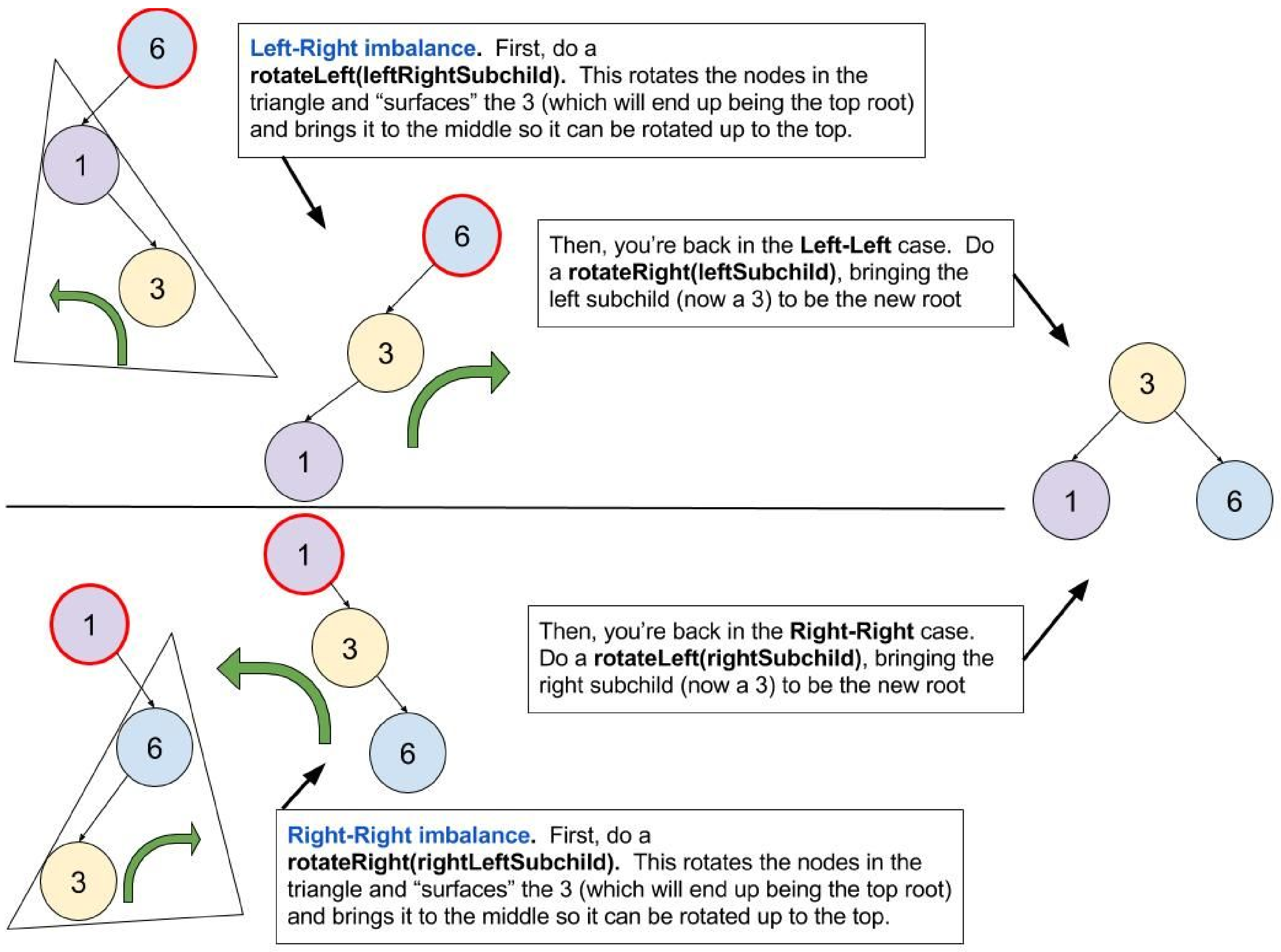
**Insert:** To insert an element, first we do an O(log(N)) traversal to where the node goes. Then we add a new node at that place and start traversing back up the tree, updating the heights as we go. We also check on this traversal that the AVL balance condition has not been violated. We will only have to change heights on small parts of the tree on each insert and delete, so we don't have to traverse every node (or N nodes) whenever we insert or delete. If we find that there is a node with an imbalance we will have to do either a single or double rotation. There are four cases for imbalance: left-left, left-right, right-right and right-left. The left-left and right-right cases can be solved with a single rotation. The left-right and right-left cases are trickier and take two rotations. The first rotation transforms the tree into one of the first two cases, and then the second rotation actually fixes the balance condition. We only have to perform at most a double rotation, because after fixing the height of the imbalanced node, we have changed the height of that imbalanced node back to the same height it was before we inserted the new element. That means we no longer have to update heights or do more rotations at a higher place up the tree. Because of this, and because the height maintenance is only necessary on one path or one small subtree, rotations and height maintenance are an asymptotic constant cost operation. We can reason then that insert is **O(log(N))** worst case: O(log(N)) traversal for find, and O(1) for rotating and height maintenance.

## Rotations

The left-left and right-right cases require a single rotation. The node with a red border indicates where the imbalance was detected.


6
3
1
Left-Left imbalance. Do a **rotateRight(leftSubchild)**, bringing the left subchild to be the new root
3
1
6
1
3
6
Right-Right imbalance. Do a **rotateLeft(rightSubchild)**, bringing the right subchild to be the new root

The left-right and right-left cases require a double rotation. The node with a red border indicates where the imbalance was detected. The first rotation gets us into one of the first two cases and the second rotation fixes the balance condition:



**Left-Right imbalance.** First, do a **rotateLeft(leftRightSubchild).** This rotates the nodes in the triangle and "surfaces" the 3 (which will end up being the top root) and brings it to the middle so it can be rotated up to the top.

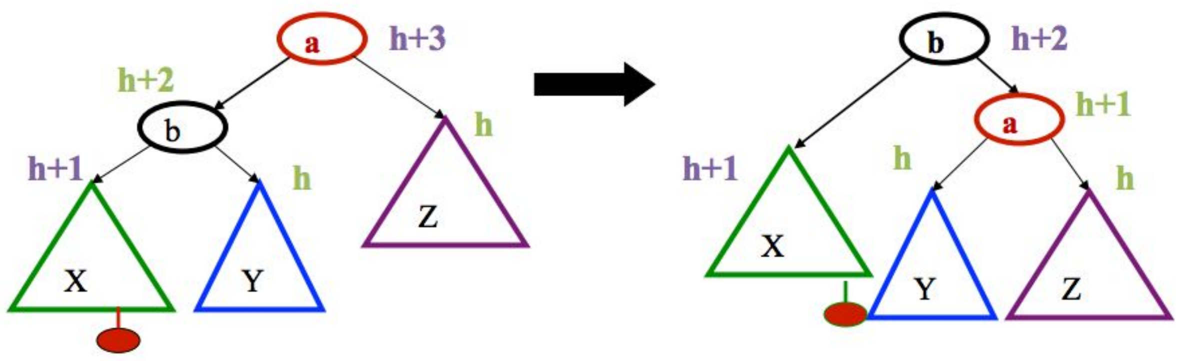
Then, you're back in the **Left-Left** case. Do a **rotateRight(leftSubchild)**, bringing the left subchild (now a 3) to be the new root

Then, you're back in the **Right-Right** case. Do a **rotateLeft(rightSubchild)**, bringing the right subchild (now a 3) to be the new root

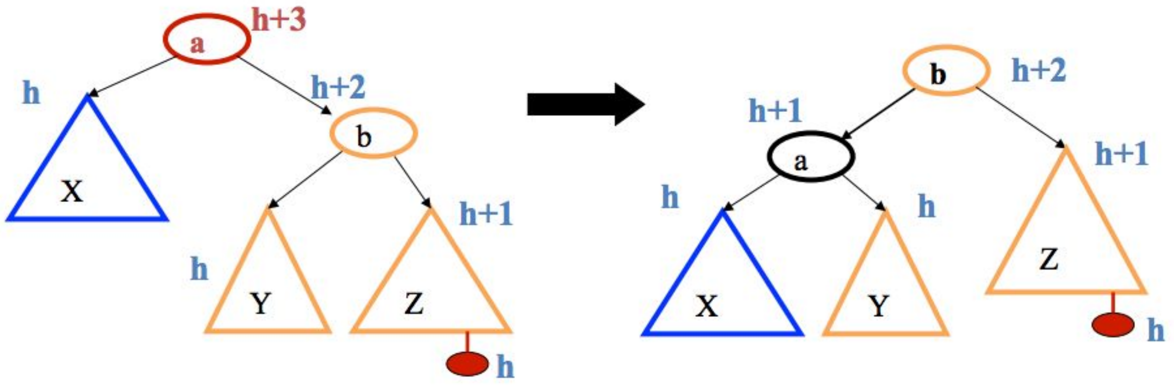
**Right-Right imbalance.** First, do a **rotateRight(rightLeftSubchild).** This rotates the nodes in the triangle and "surfaces" the 3 (which will end up being the top root) and brings it to the middle so it can be rotated up to the top.

These example rotations are the simplest AVL trees that need rotations. However, the imbalanced node will not always be the overall root of the tree, and there could be subtrees to juggle around in these rotations. For the following images of general rotations, see detailed explanations in the AVL tree slides.

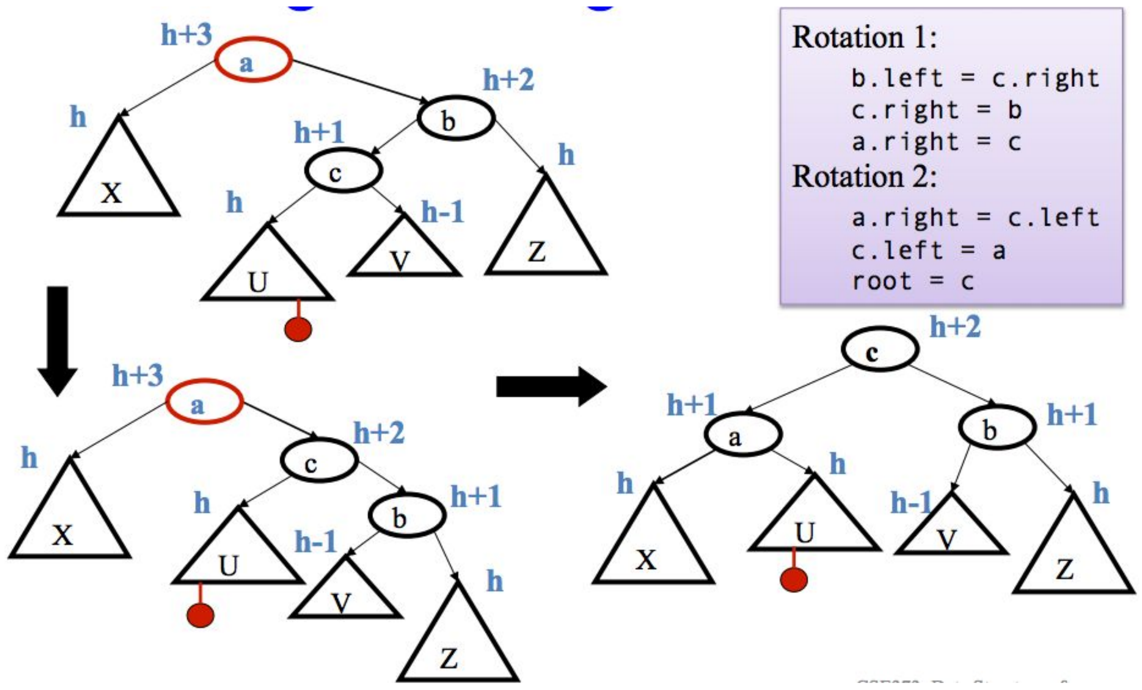
## General Left-Left Case



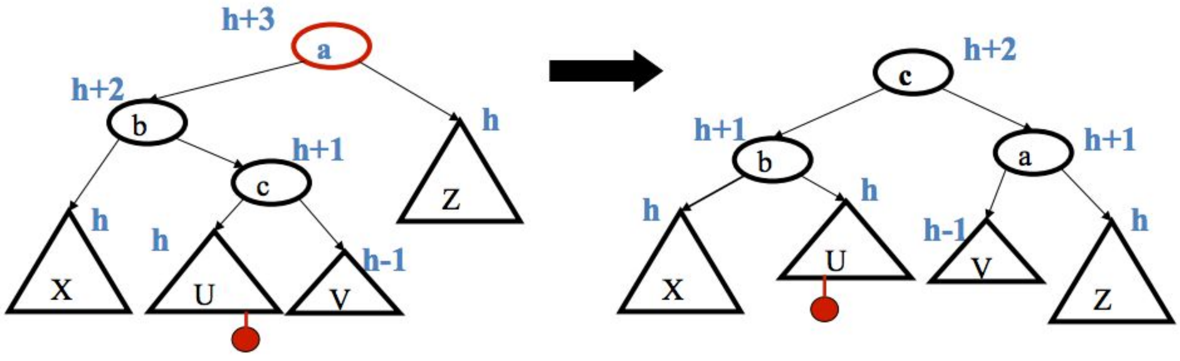
General Right-Right Case



General Right-Left Case


```
Rotation 1:
    b.left = c.right
    c.right = b
    a.right = c
Rotation 2:
    a.right = c.left
    c.left = a
    root = c
```

General Left-Right Case


h+3
a
h+2
b
h
h+1
c
Z
h
h
h-1
X
U
V
c
h+2
h+1
b
a
h+1
h
h
h-1
h
X
U
V
Z