Balanced Binary Trees

Binary search trees provide O(log N) search times provided that the nodes are distributed in a reasonably “balanced” manner. Unfortunately, that is not always the case and performing a sequence of deletions and insertions can often exacerbate the problem.

When a BST becomes badly unbalanced, the search behavior can degenerate to that of a sorted linked list, O(N).

There are a number of strategies for dealing with this problem; most involve adding some sort of restructuring to the insert/delete algorithms.

That can be effective only if the restructuring reduces the average depth of a node from the root of the BST, and if the cost of the restructuring is, on average, O(log N).

We will examine one such restructuring algorithm…

AVL Trees

AVL tree*: a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1, and in which the left and right subtrees are themselves AVL trees.

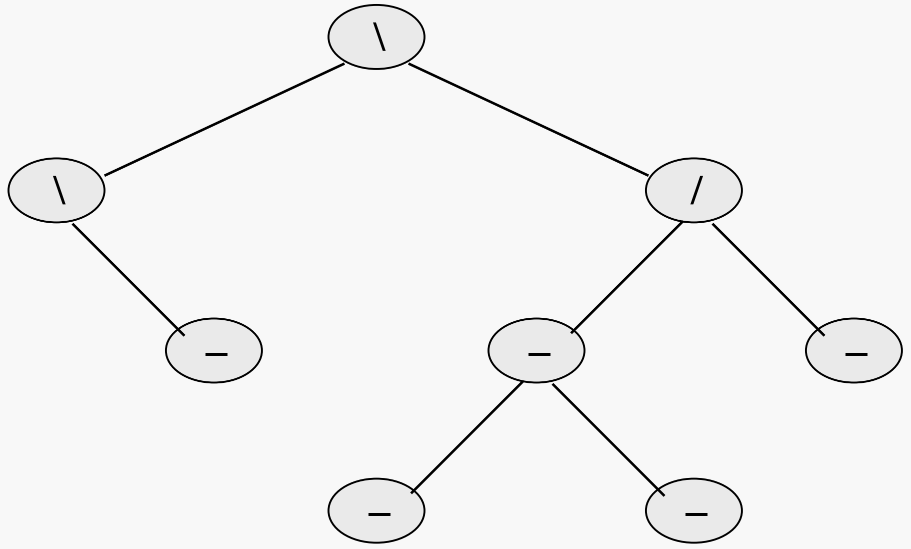
Each AVL tree node has an associated balance factor indicating the relative heights of its subtrees (left-higher, equal, right-higher). Normally, this adds one data element to each tree node and an enumerated type is used.

How effective is this? The height of an AVL tree with N nodes never exceeds 1.44 log N and is typically much closer to log N.

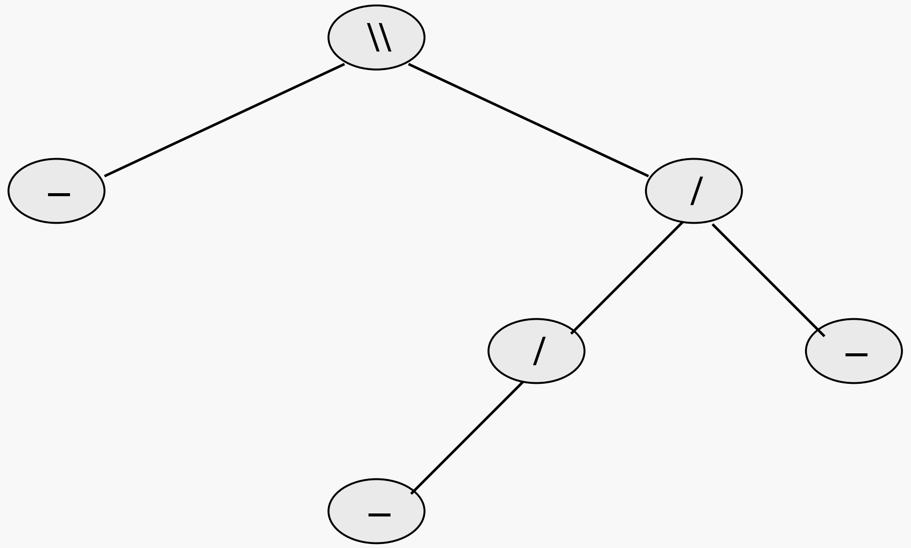
*G. M. Adelson-Velskii and E. M. Landis, 1962.

Examples

This is an AVL tree. . .

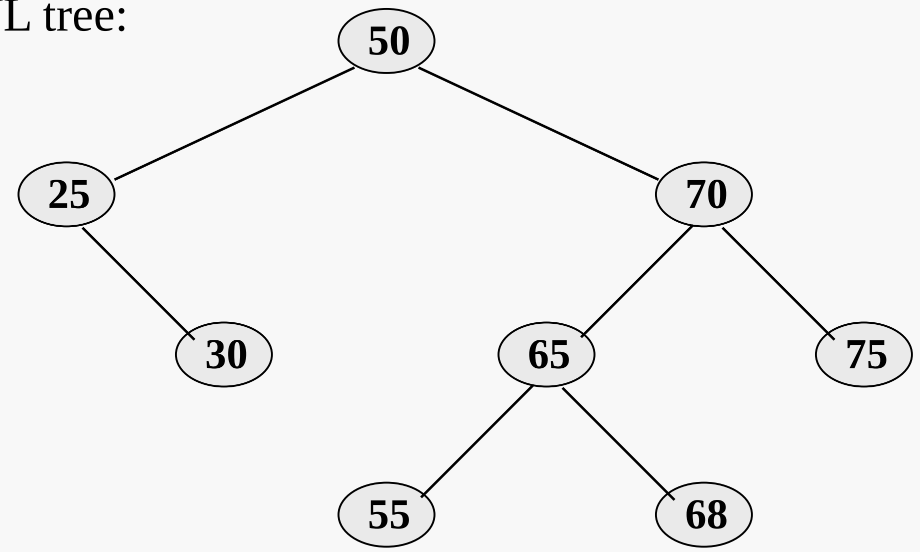


. . .and this is not.

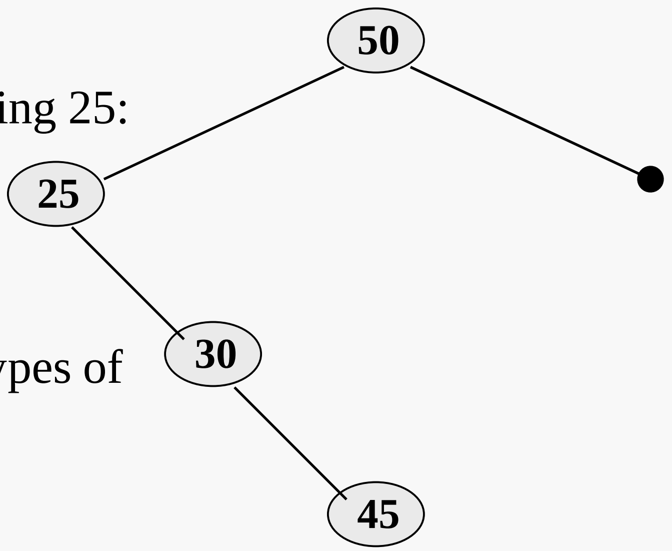


Unbalance from Insertion

Consider inserting the value 45 into the AVL tree:



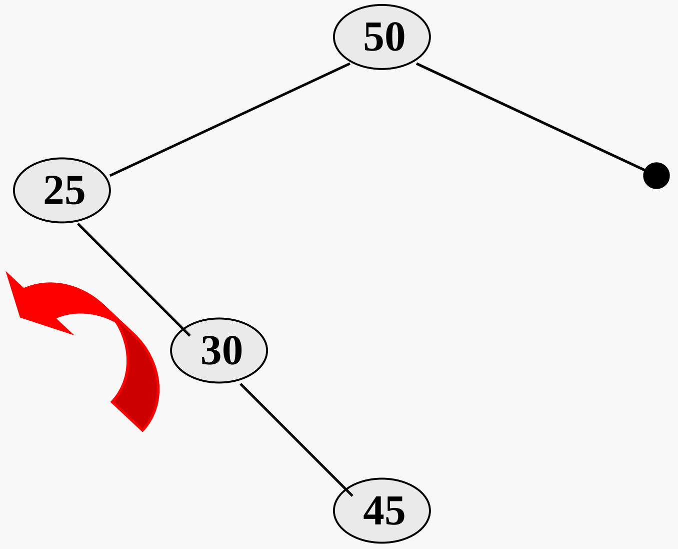
The result would be unbalanced at the node containing 25:



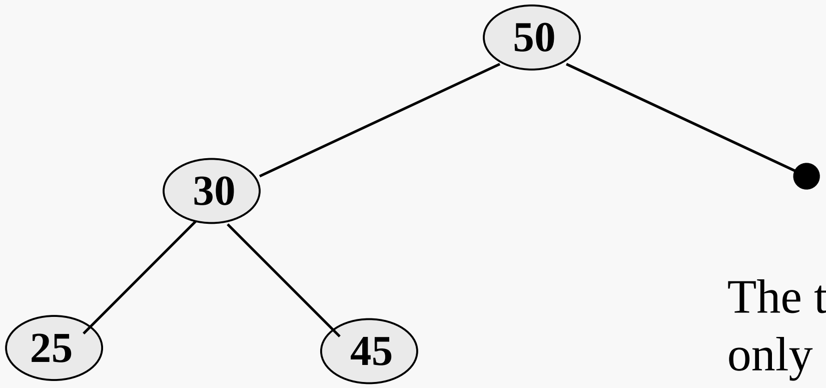
The unbalance is repaired by applying one of two types of "rotation" to the unbalanced subtree…

Rebalancing via Subtree Restructuring

The subtree rooted at 25 is right-higher.



We restructure the subtree, resulting in a balanced subtree:

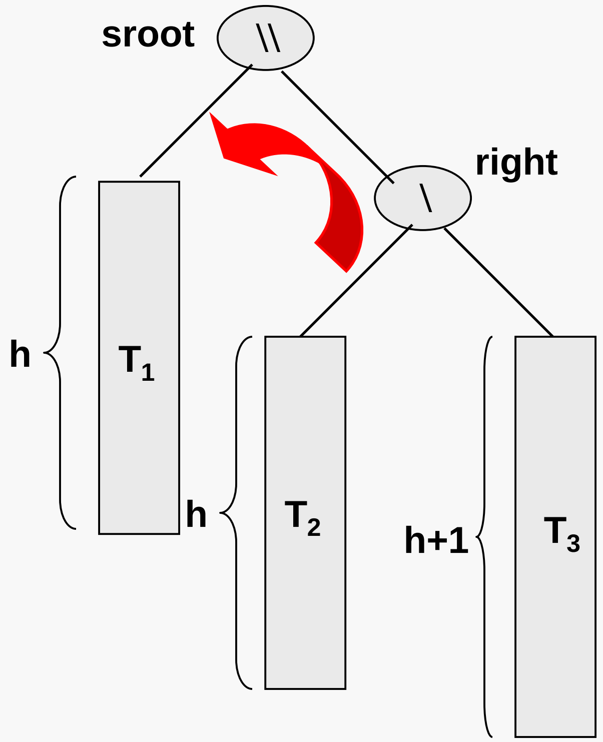


The transformation is relatively simple, requiring only a few operations, and results in a subtree that has equal balance.

AVL Insertion Case: right-higher

There are two unbalance cases to consider, each defined by the state of the subtree that just received a new node. For simplicity, assume for now that the insertion was to the right subtree (of the subtree).

Let sroot be the root of the newly unbalanced subtree, and suppose that its right subtree is now right-higher:



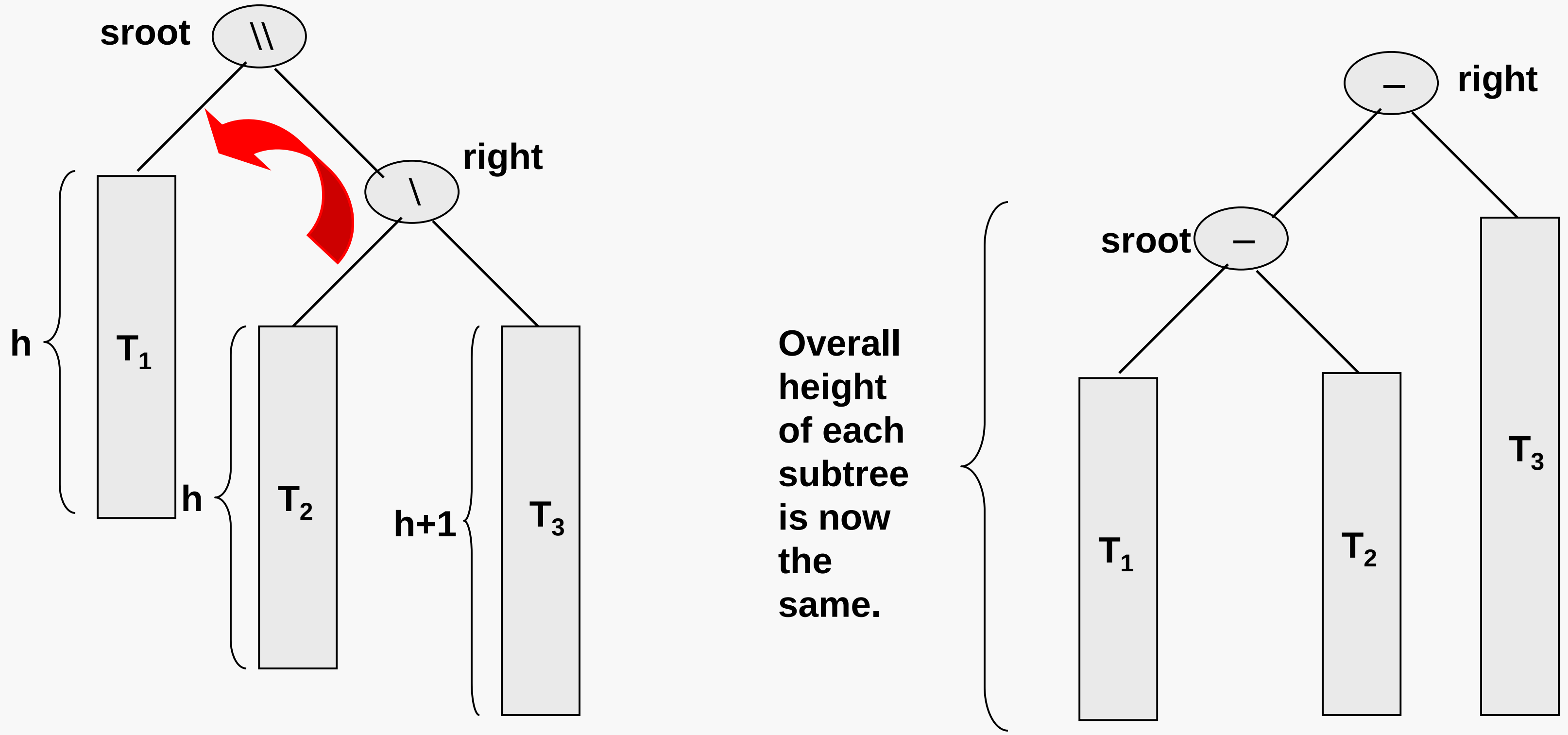
In this case, the subtree rooted at right was previously equally balanced (why?) and the subtree rooted at sroot was previously right-higher (why?).

The height labels follow from those observations.

Balance can be restored by "rotating" the values so that right becomes the subtree root node and sroot becomes the left child.

AVL Left Rotation

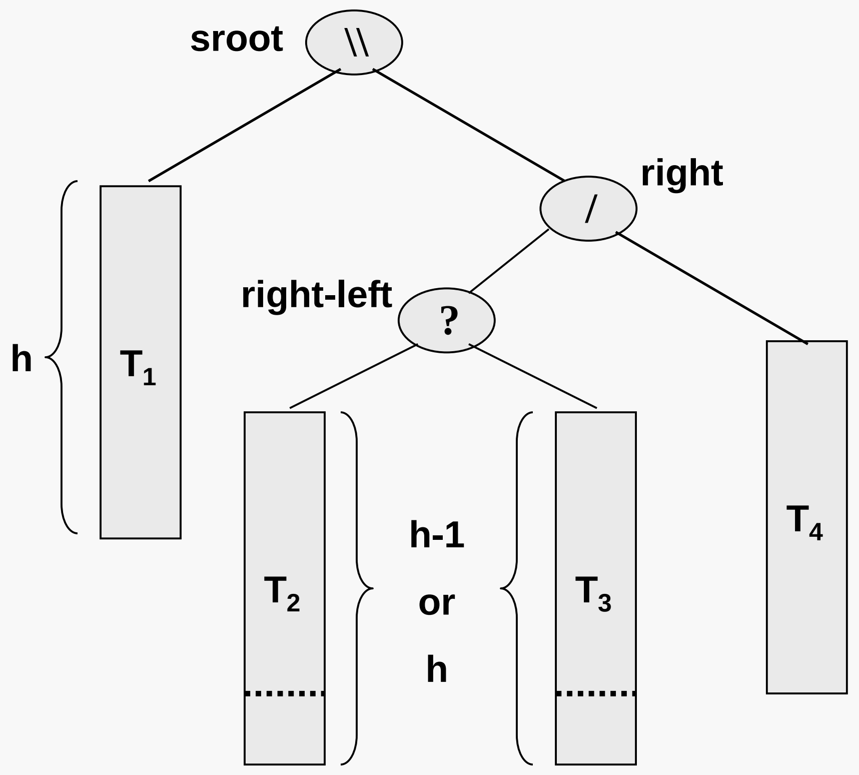
The manipulation just described is known as a “left rotation” and the result is:



That covers the case where the right subtree has become right-higher… the case where the left subtree has become left-higher is analogous and solved by a right rotation.

AVL Insertion Case: left-higher

Now suppose that the right subtree has become left-higher:



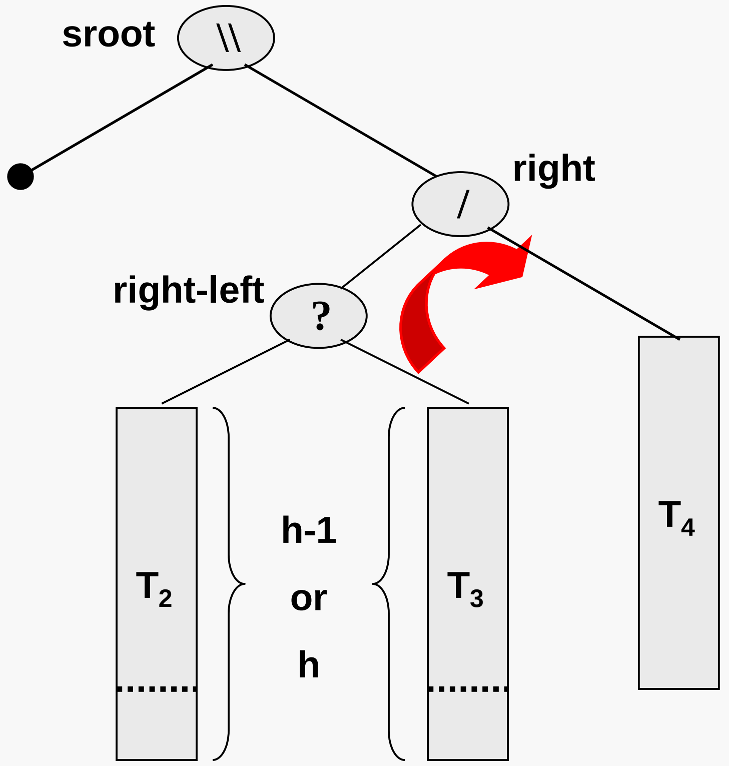
The insertion occurred in the left subtree of the right subtree of sroot.

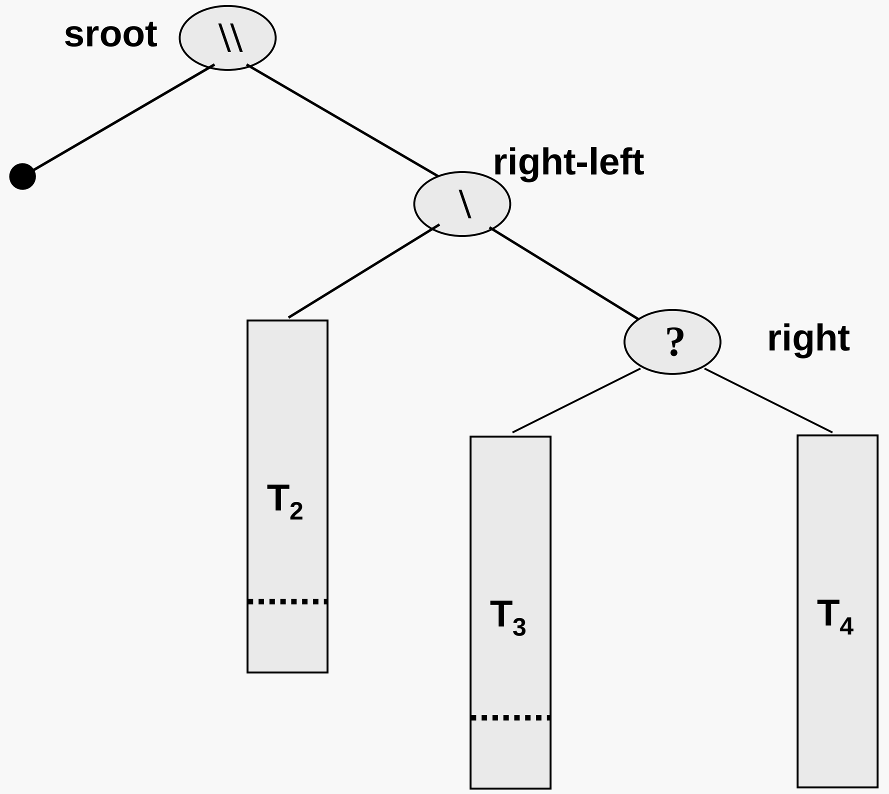
In this case, the left subtree of the right subtree (rooted at right-left) may be either left-higher or right-higher, but not balanced (why?).

Surprisingly (perhaps), this case is more difficult. The unbalance cannot be removed by performing a single left or right rotation.

AVL Double Rotation

Applying a single right rotation to the subtree rooted at right produces…

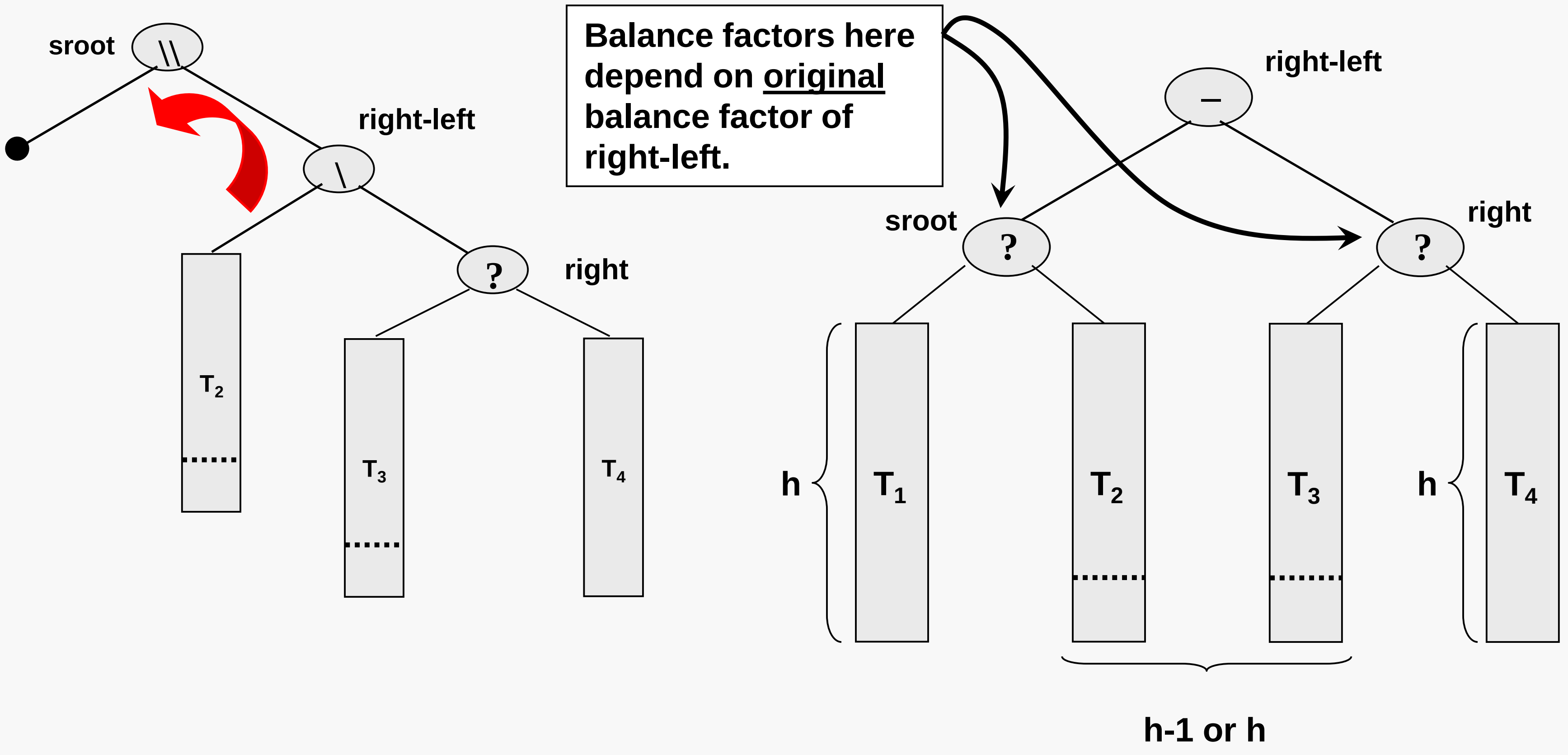




…a subtree rooted at right-left that is now right-higher…

AVL Double Rotation

Now, applying a single left rotation to the subtree rooted at sroot produces…



…a balanced subtree.

The case where the left subtree of sroot is right-higher is handled similarly (by a double rotation).

Unbalance from Deletion

Deleting a node from an AVL tree can also create an imbalance that must be corrected.

The effects of deletion are potentially more complex than those of insertion.

The basic idea remains the same: delete the node, track changes in balance factors as the recursion backs out, and apply rotations as needed to restore AVL balance at each node along the path followed down during the deletion.

However, rebalancing after a deletion may require applying single or double rotations at more than one point along that path.

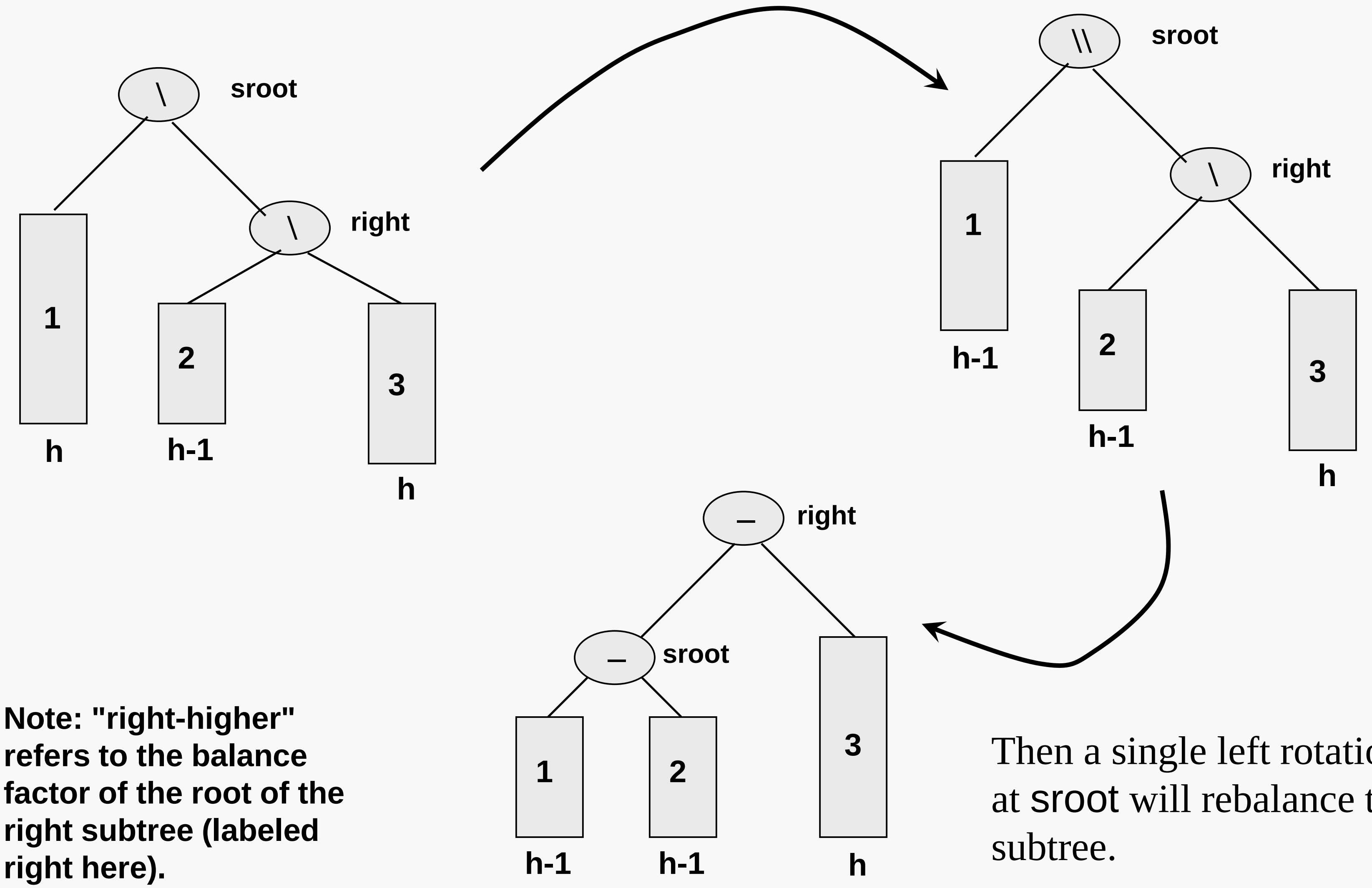
As usual, there are cases…

Here, we will make the following assumptions:

- the lowest imbalance occurs at the node `root` (a subtree root)
- the deletion occurred in the left subtree of `root`

AVL Deletion Case: right-higher

Suppose we have the subtree on the left prior to deletion and that on the right after deletion:

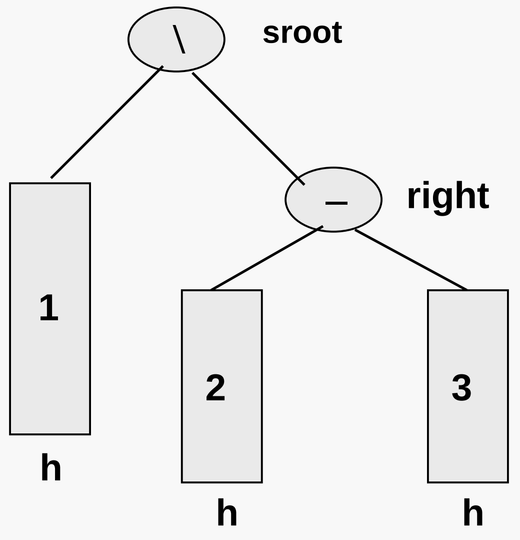


Note: "right-higher" refers to the balance factor of the root of the right subtree (labeled right here).

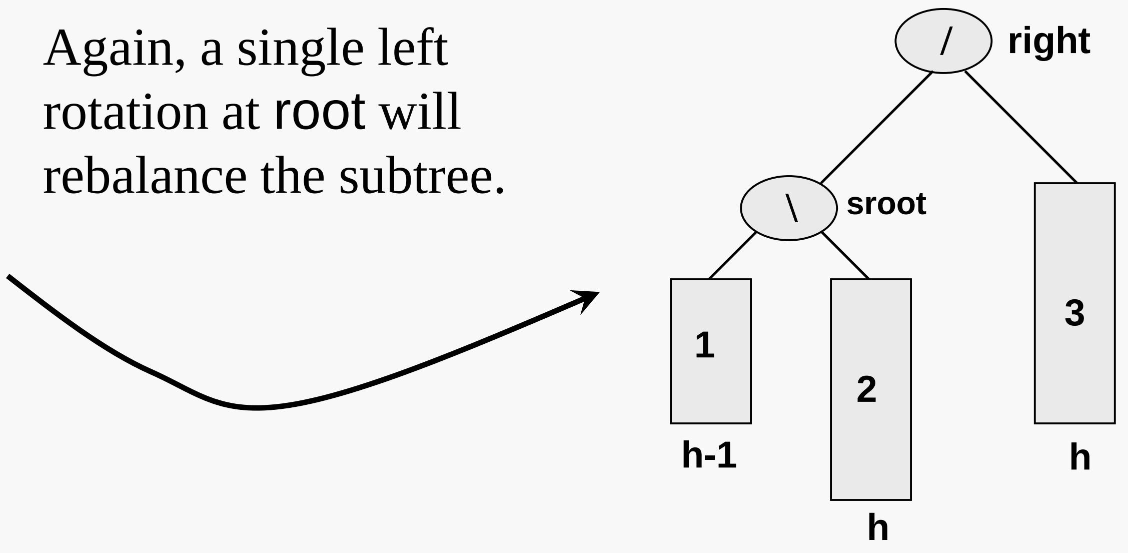
Then a single left rotation at sroot will rebalance the subtree.

AVL Deletion Case: equal-height

Suppose the right subtree root has balance factor equal-height:



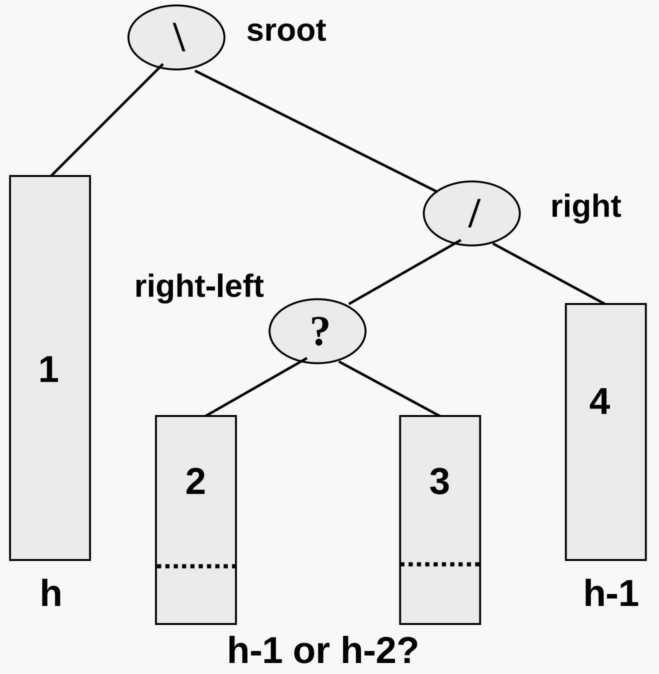
Again, a single left rotation at root will rebalance the subtree.



The difference is the resulting balance factor at the old subtree root node, sroot, which depends upon the original balance factor of the node right.

AVL Deletion Case: left-higher

If the right subtree root was left-higher, we have the following situation:



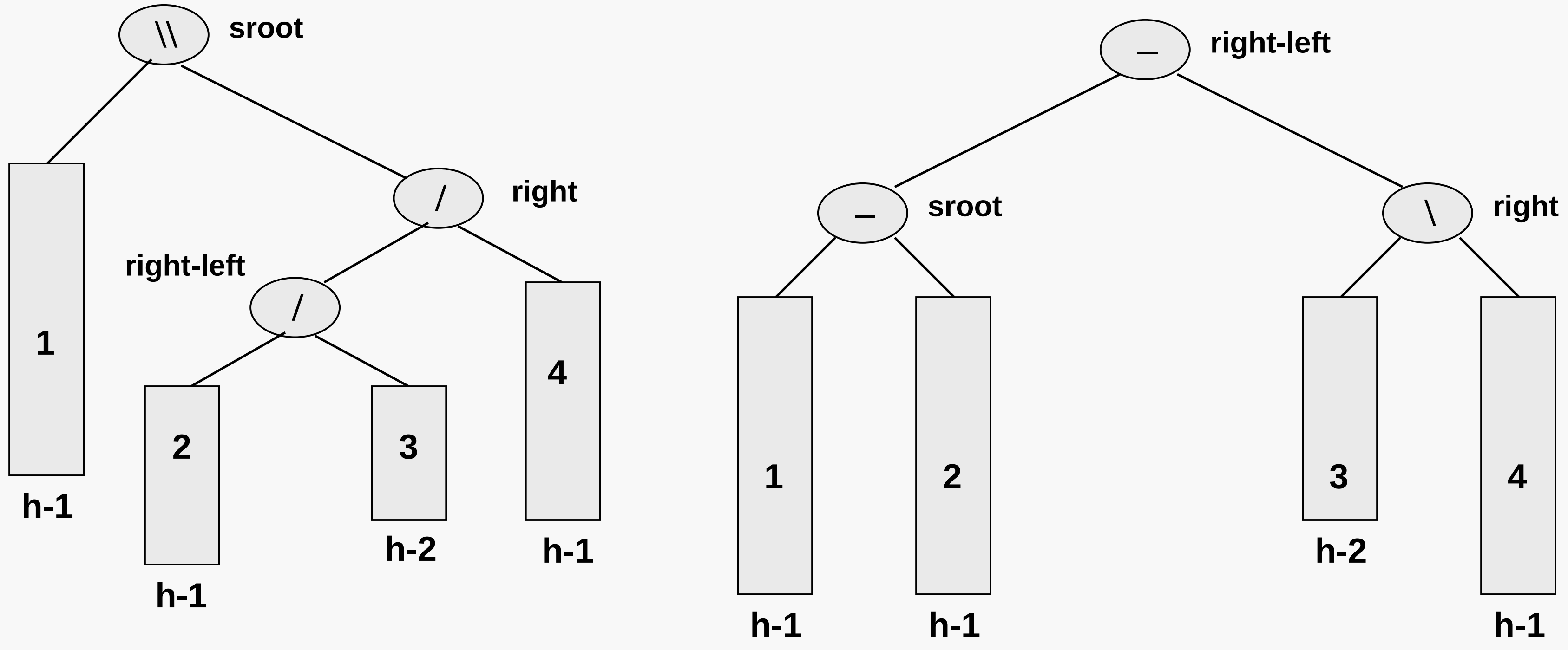
Deleting a node from the left subtree of sroot now will cause sroot to become double right higher.

As you should expect, the resulting imbalance can be cured by first applying a right rotation at the node right, and then applying a left rotation at the node sroot.

However, we must be careful because the balance factors will depend upon the original balance factor at the node labeled right-left…

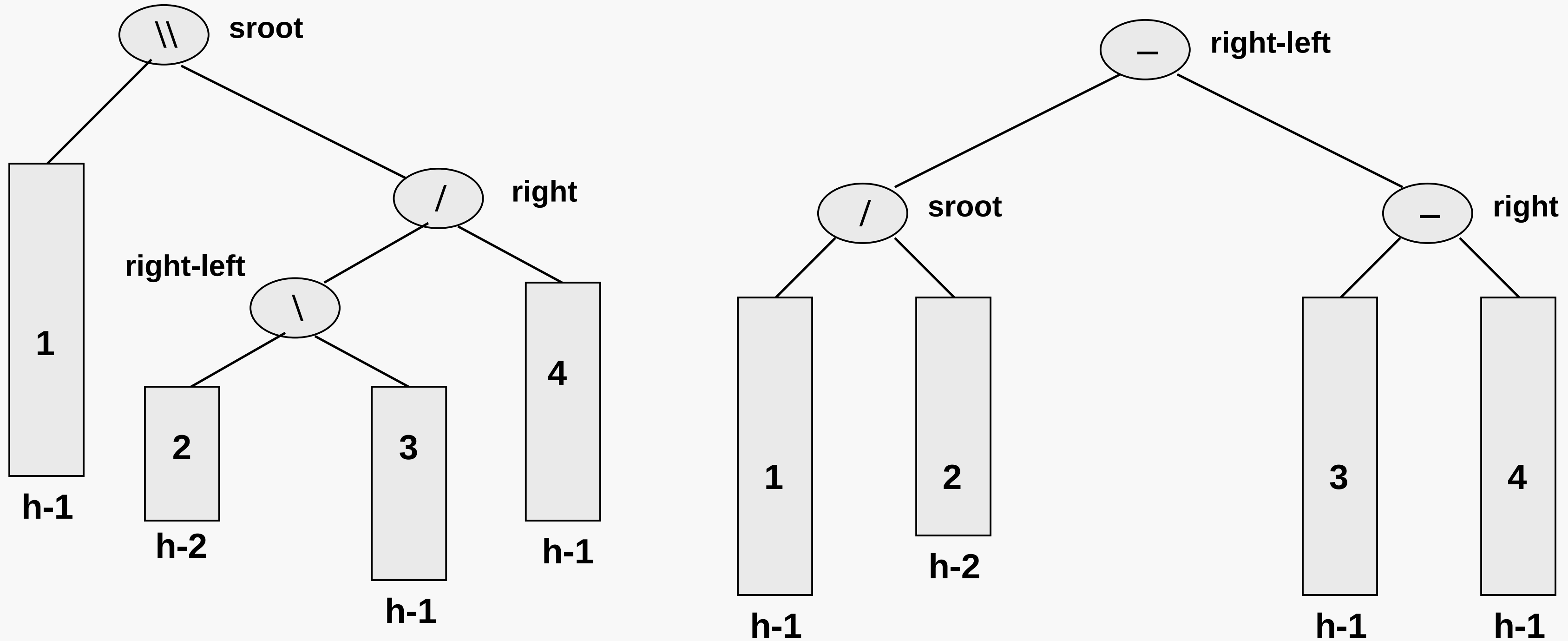
AVL Deletion Case: left-higher, left-higher

If the right-left subtree root was also left-higher, we obtain:



AVL Deletion Case: left-higher, right-higher

If the right-left subtree root was right-higher, we obtain:



And, finally, if the right-left subtree root was equal-height, we'd obtain a tree where all three of the labeled nodes have equal-height.

AVL Deletion Cases: Summary

We have considered a number of distinct deletion cases, assuming that the deletion occurred in the left subtree of the imbalanced node.

There are an equal number of entirely similar, symmetric cases for the assumption the deletion was in the right subtree of the imbalanced node.

Drawing diagrams helps…

This discussion also has some logical implications for how insertion is handled in an AVL tree. The determination of the balance factors in the tree, following the rotations, involves similar logic in both cases.

Implementation Ideas

An enumerated type is useful for dealing with the balance factors:

```
public enum BFactor {DBLLEFTHI, LEFTHI, EQUALHT,
                                RIGHTHI, DBLRIGHTHI};
```

Enumerated type values can be compared with the == operator:

```
. . .
if ( Grew == BFactor.RIGHTHI ) {
   . . .
```

Enumerated type values can also be used as cases for a `switch` statement.

Implementation Ideas

AVL nodes add a representation for the nodes's balance:

```
private static class AVLNode {
   . . .
   BFactor    balance;
   T          element;
   AVLNode    left;
   AVLNode    right;
}
```

Because we need pointers to AVL nodes, we do not derive `AVLNode` from `BSTNode`.

That can be made to work, but it is ugly and inefficient.

Implementation Ideas

```
public class AVLTree<T extends Comparable<? super T>> {

   . . .
   AVLNode root;
   . . .
   // single rotations
   private AVLNode rotateRight( AVLNode sroot ) {. . .}
   private AVLNode rotateLeft( AVLNode sroot ) {. . .}
   // double rotations
   private AVLNode rotateRightLeft( AVLNode sroot ) {. . .}
   private AVLNode rotateLeftRight( AVLNode sroot ) {. . .}

   // rebalance managers
   private AVLNode rightBalance( AVLNode sroot ) {. . .}
   private AVLNode leftBalance( AVLNode sroot) {. . .}
   . . .
```

Complexity

Let N_h be the minimum number of nodes an AVL tree with h levels can have.

Then:

$$N_1 = 1, N_2 = 2$$
$$N_h = N_{h-1} + N_{h-2} + 1, for\, N > 2$$

This can be solved to show that:

$$N_h = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1$$

And, from this:

$$h < \log_{\frac{1+\sqrt{5}}{2}}(N_h + 1) \approx 1.44 \log(N_h)$$