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 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.

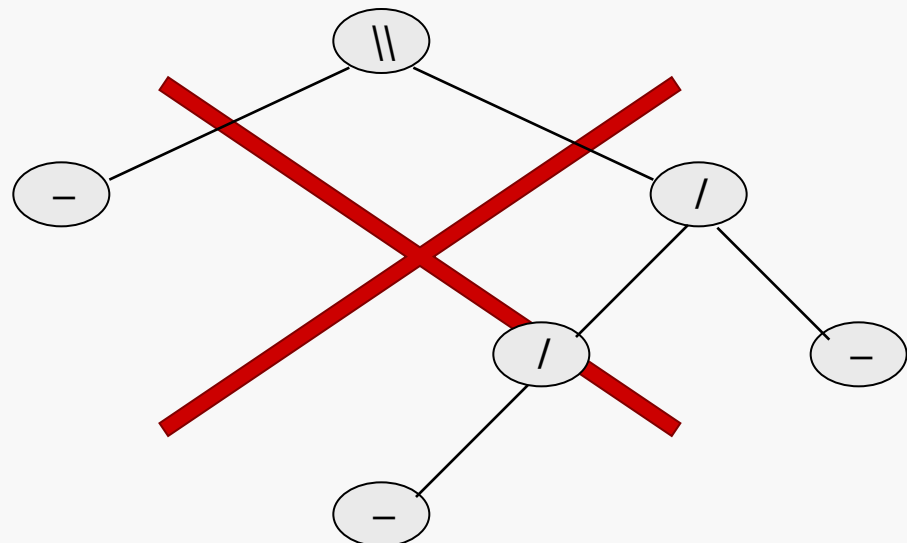*G. M. Adelson-Velskii and E. M. Landis, 1962.
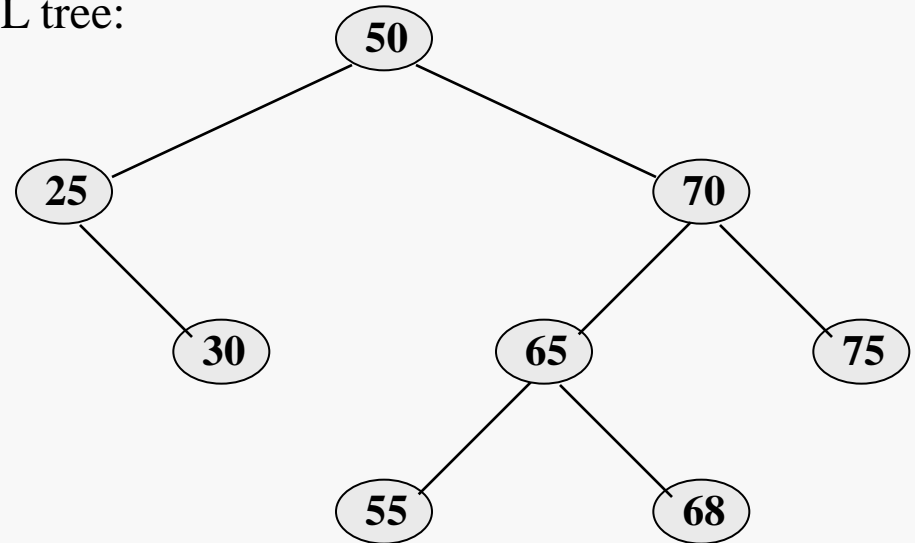
This is an AVL tree. . .
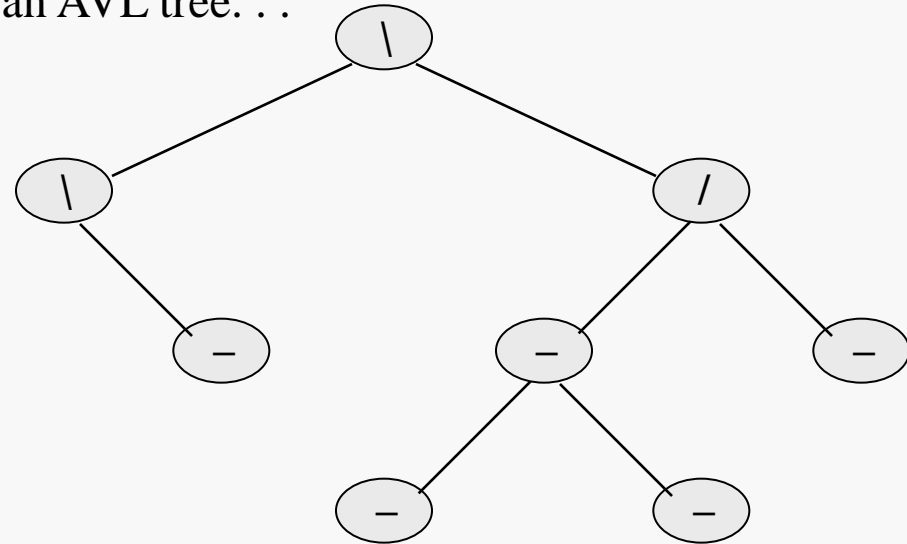
Notation:
- – means the subtrees of the node are the same height
- \ means that the right subtree of the node is higher (by one) than the left subtree (right-higher)
- / means that the left subtree of the node is higher (by one) than the right subtree (left-higher)
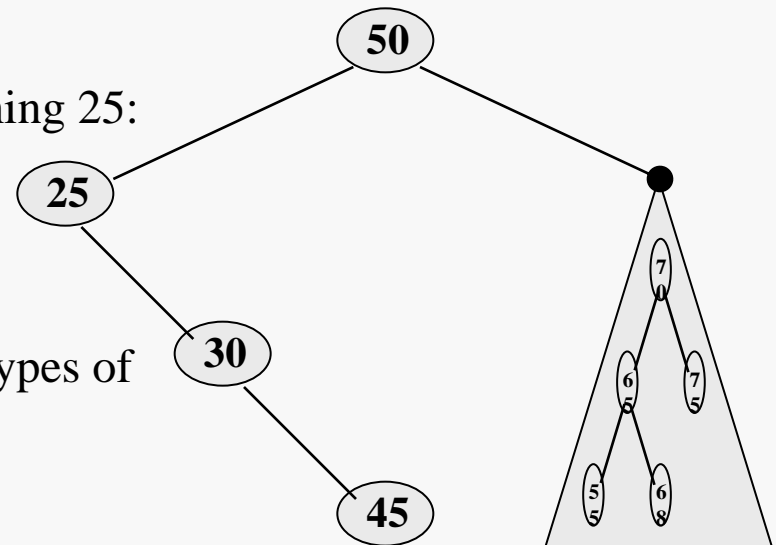
. . .and this is **NOT** an AVL tree.

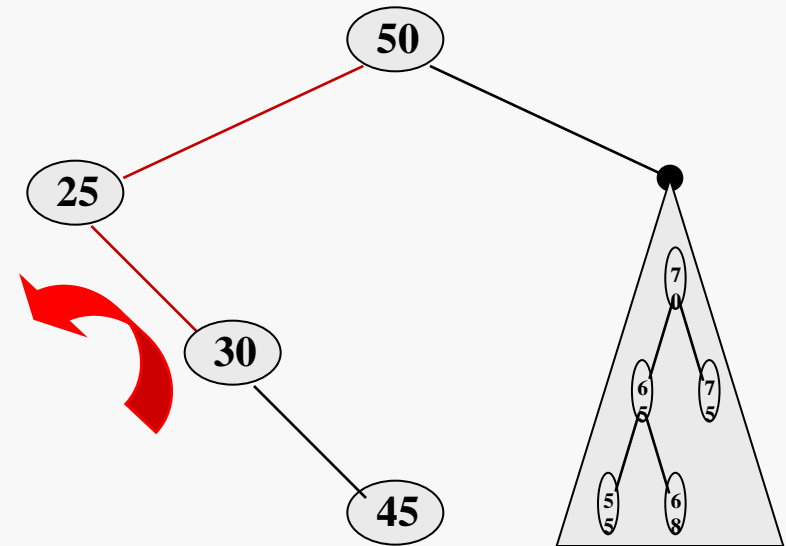Consider inserting the value 45 into the AVL tree:
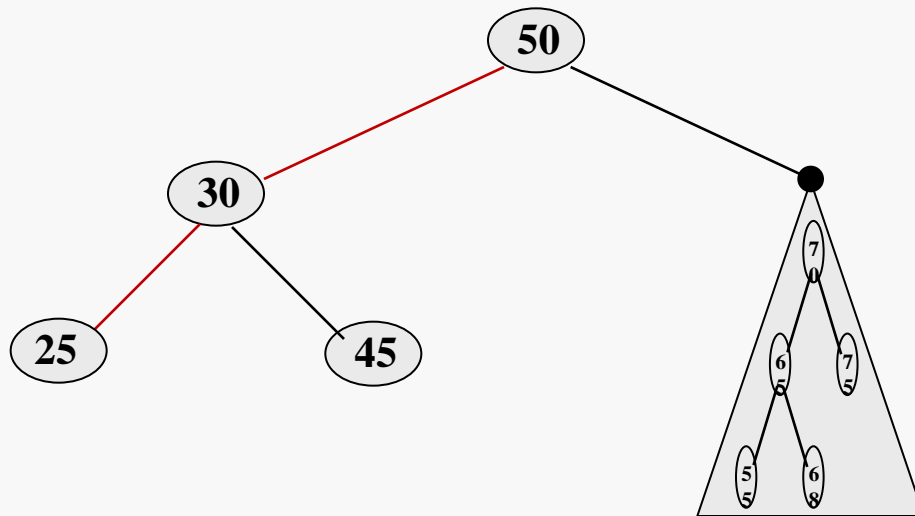
The result would be unbalanced at the node containing 25:

The unbalance is repaired by applying one of two types of *"rotation"* to the unbalanced subtree…
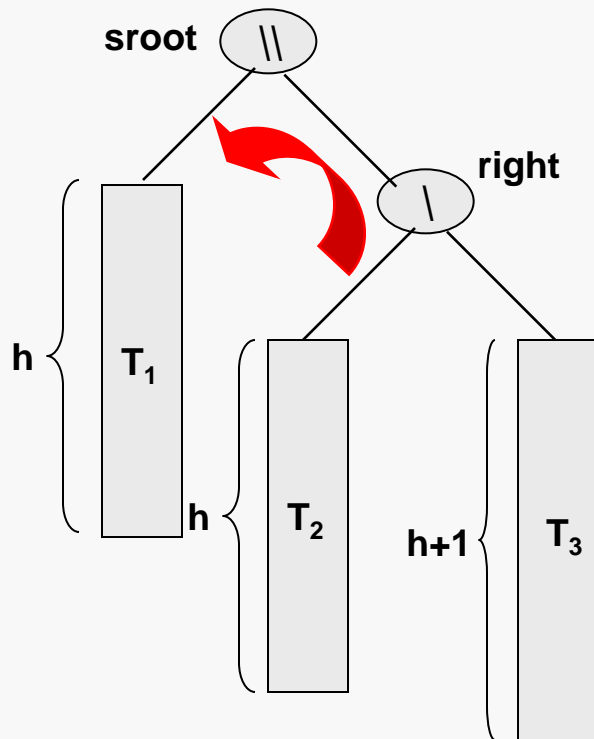
The subtree rooted at 25 is right-higher.

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.

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.

The manipulation just described is known as a "*left rotation*" and the result is:
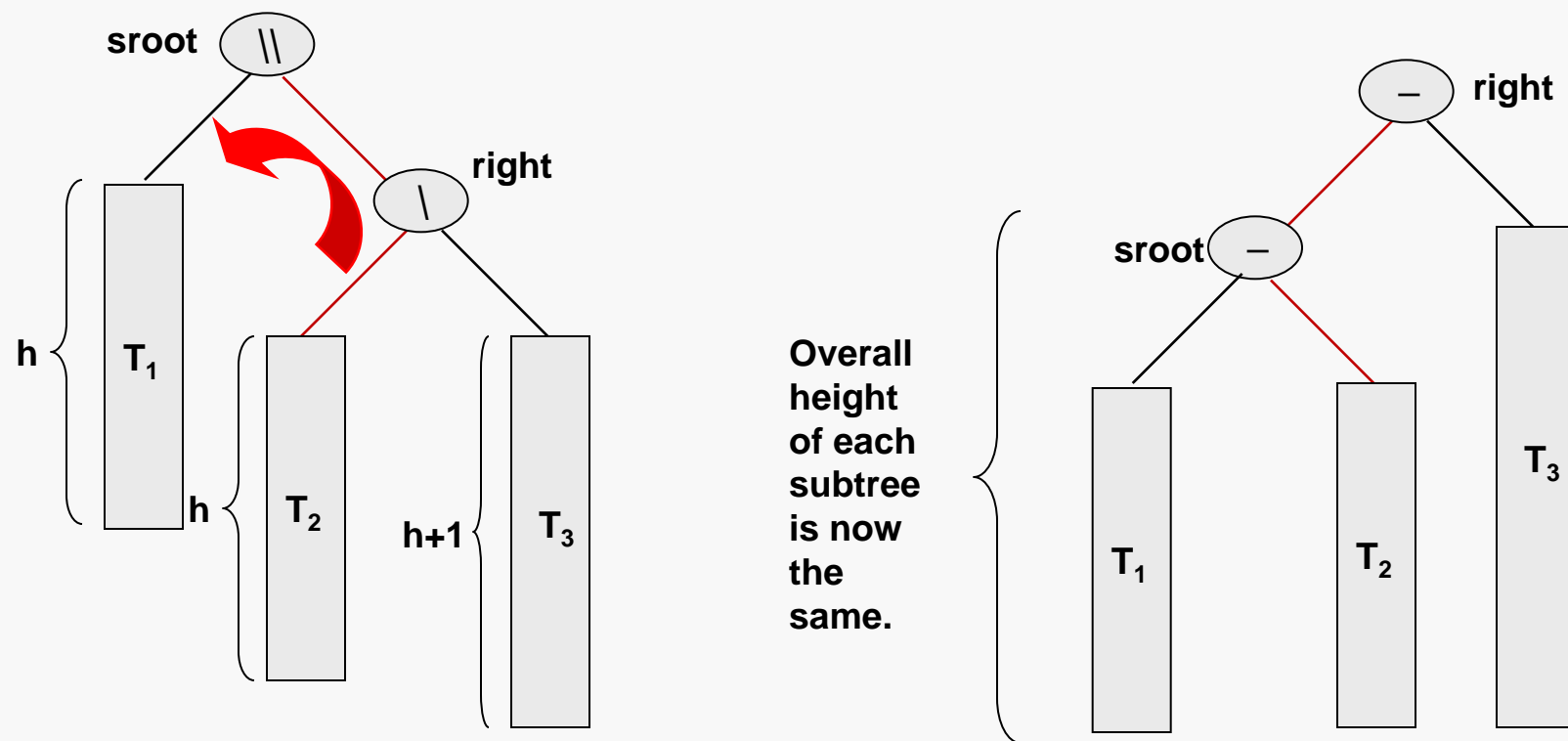
That covers the *outside* case where the right subtree has become right-higher… the other *outside* case where the left subtree has become left-higher is analogous and solved by a *right rotation*.
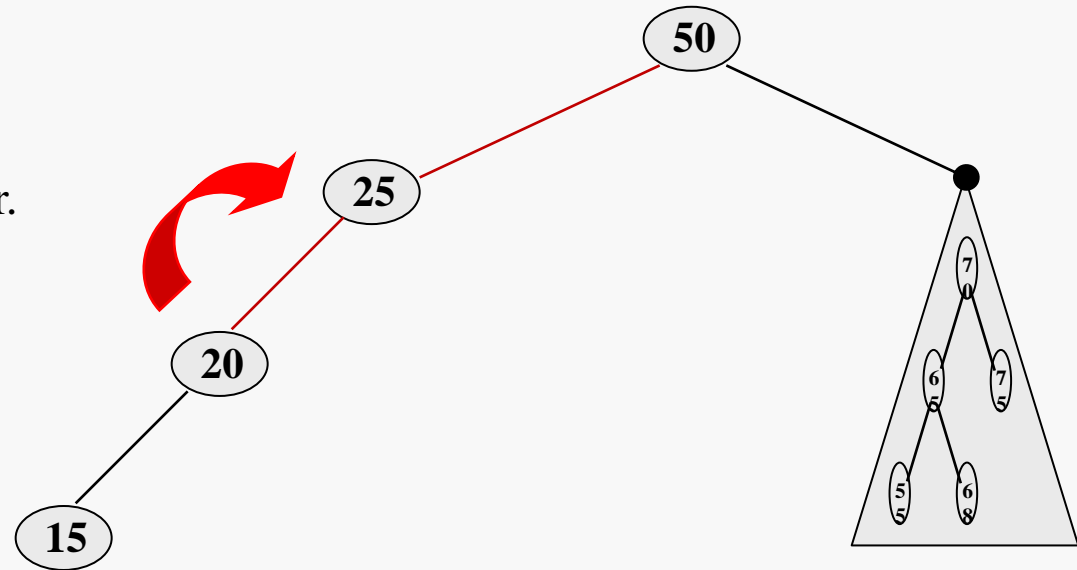
Consider inserting the value 15 into this AVL tree:
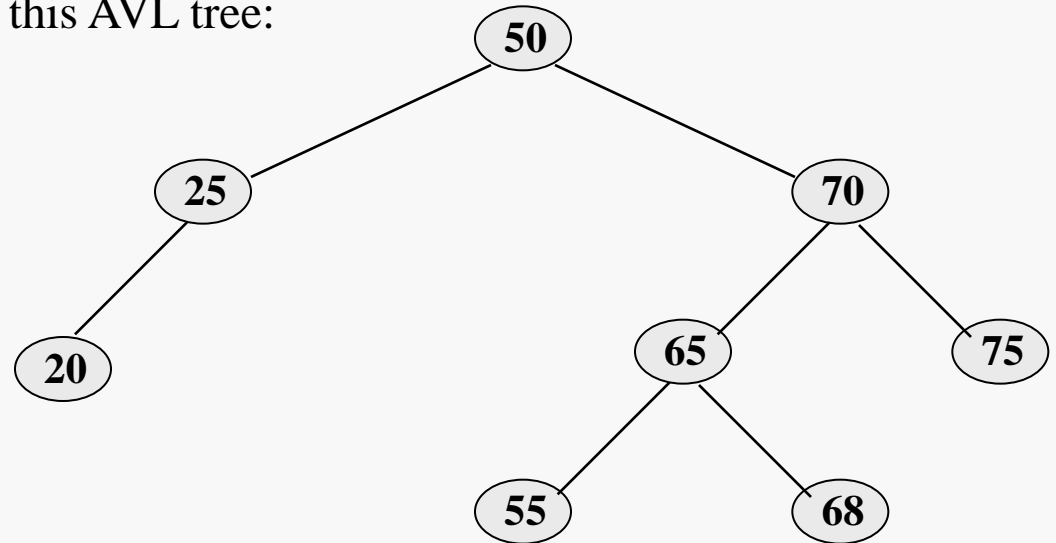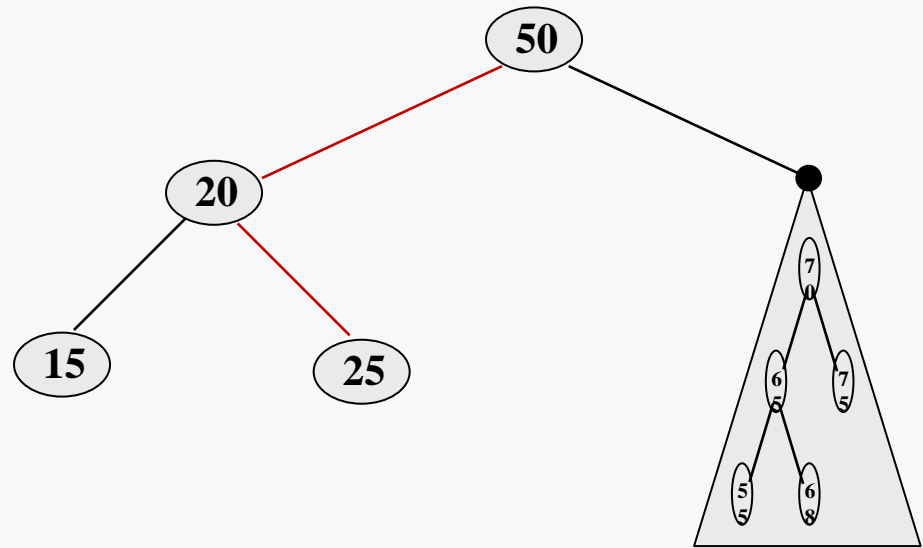
The result would be unbalanced at the node containing 25:

The unbalance is repaired by applying one of two types of "*rotation*" to the unbalanced subtree…
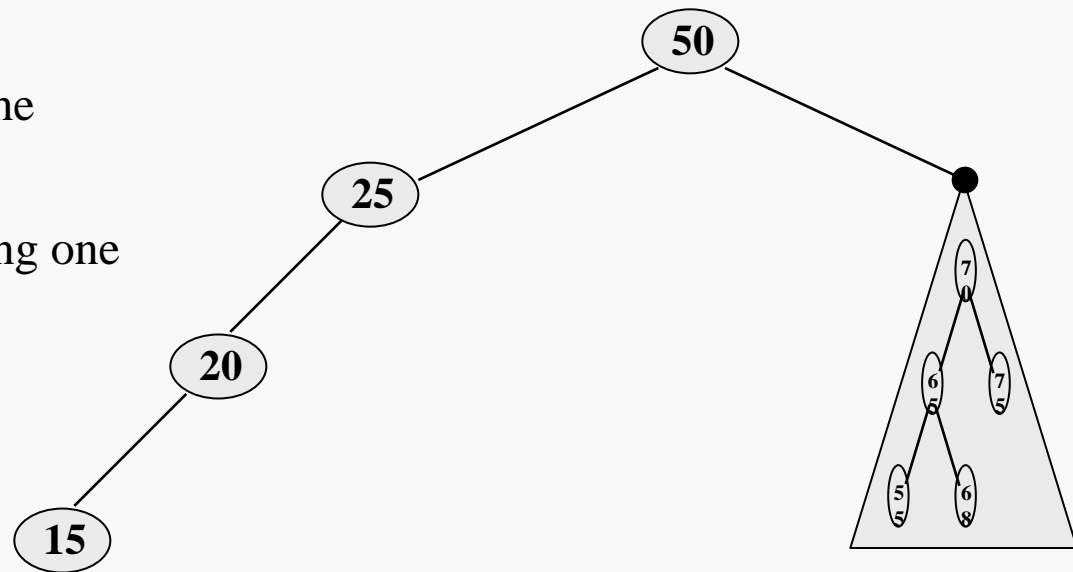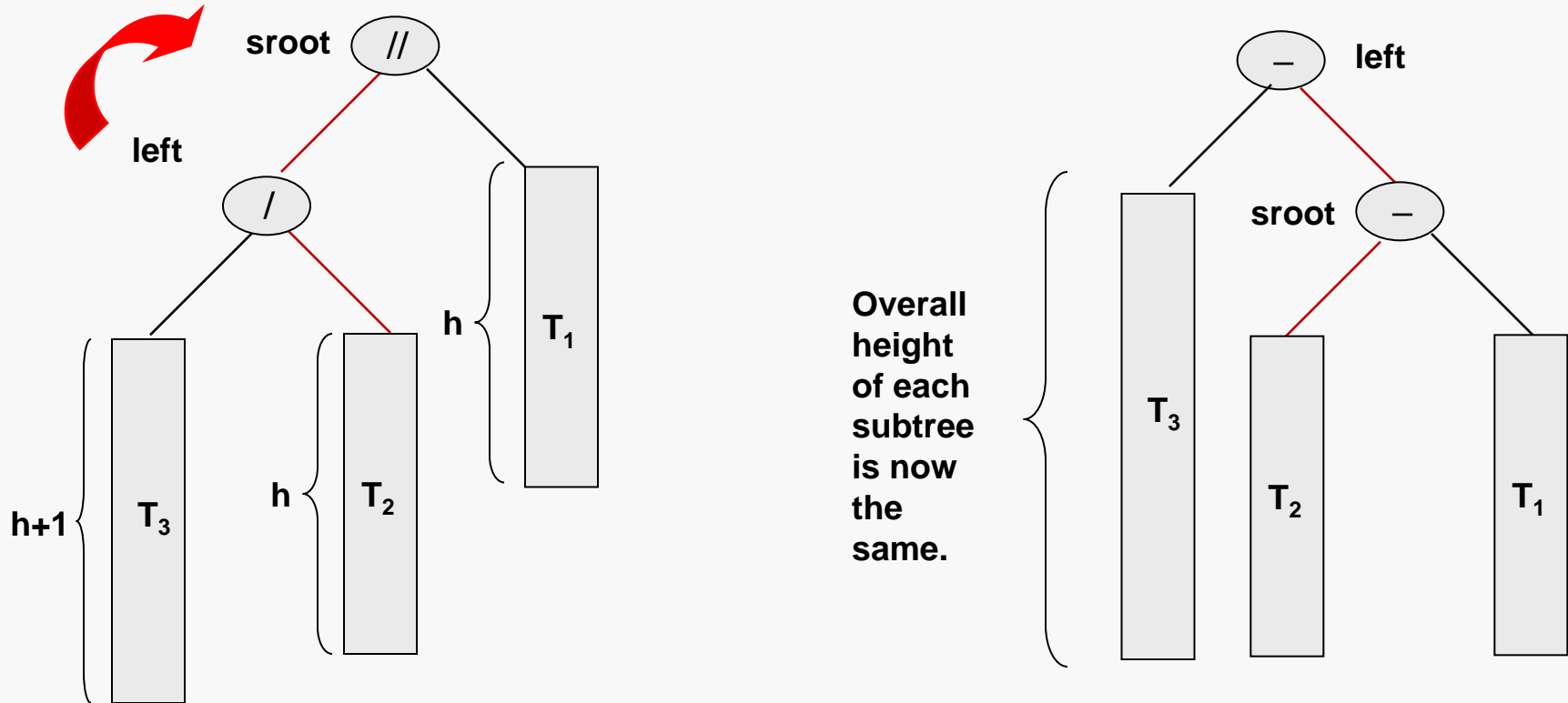
The subtree rooted at 25 is left-higher.

We restructure the subtree, resulting in a balanced subtree:
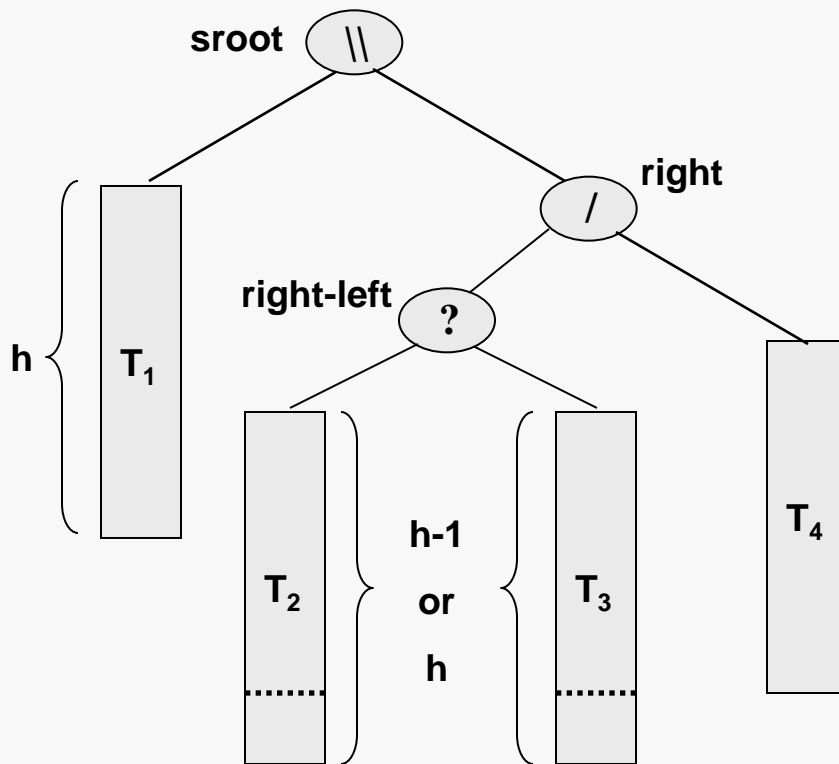
The transformation is again relatively simple, requiring only a few operations, and results in a subtree that has equal balance.

The previous manipulation is known as a "*right rotation*" and the result is:

That covers the case where the left subtree has become left-higher.

Now suppose that the right subtree has become left-higher (*inner case*) :

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.

Applying a single *right rotation* to the subtree rooted at right produces…
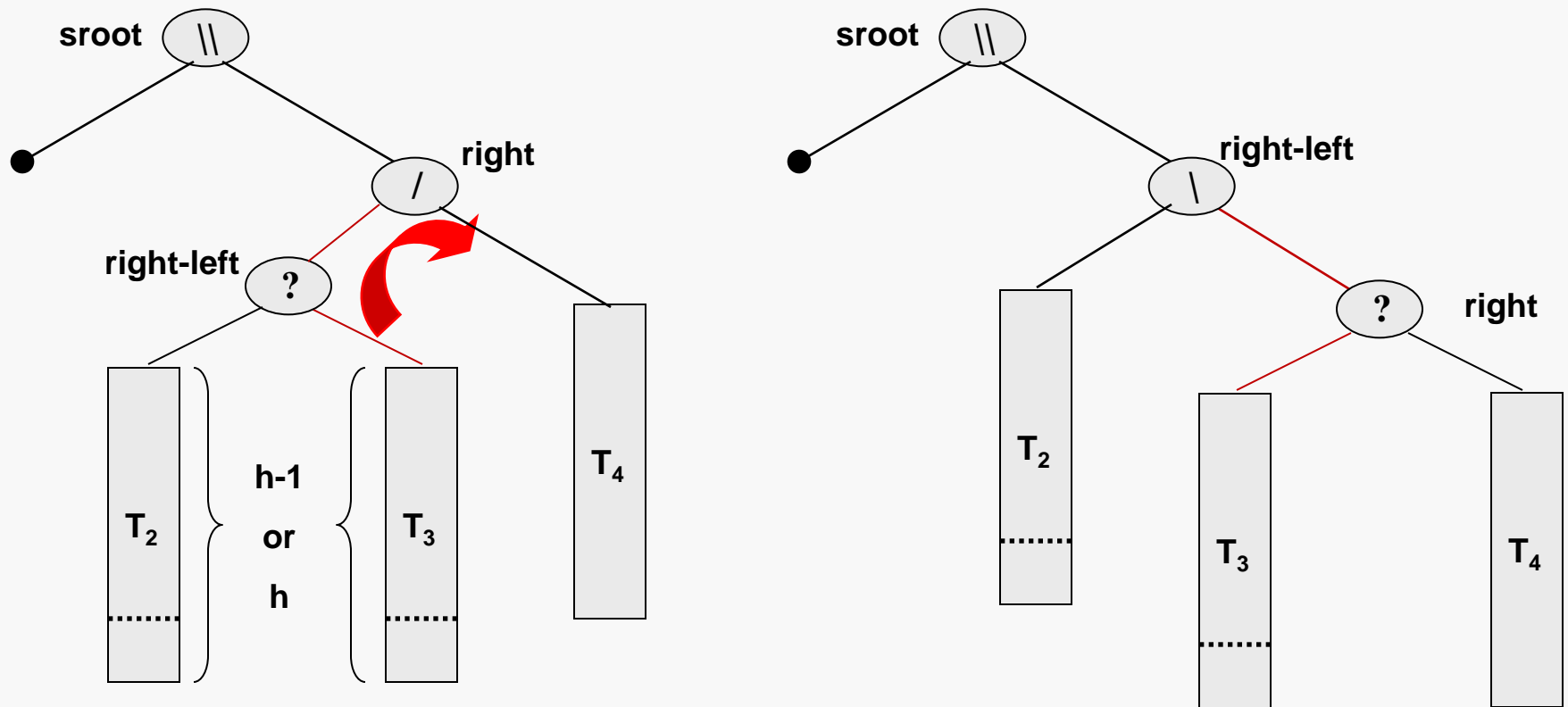
…a subtree rooted at right-left that is now right-higher…

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

Balance factors here depend on <u>original</u> balance factor of right-left.
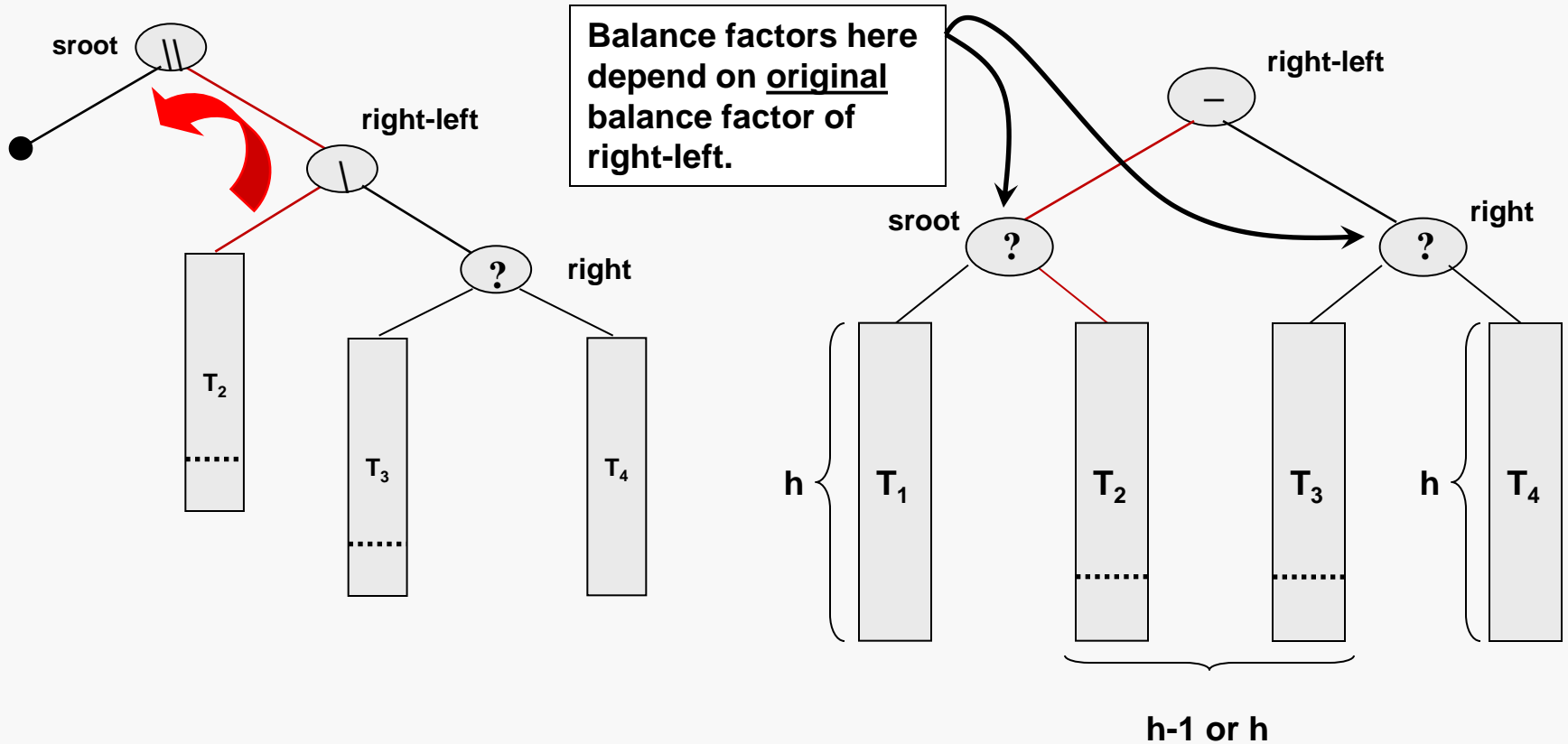
…a balanced subtree.

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

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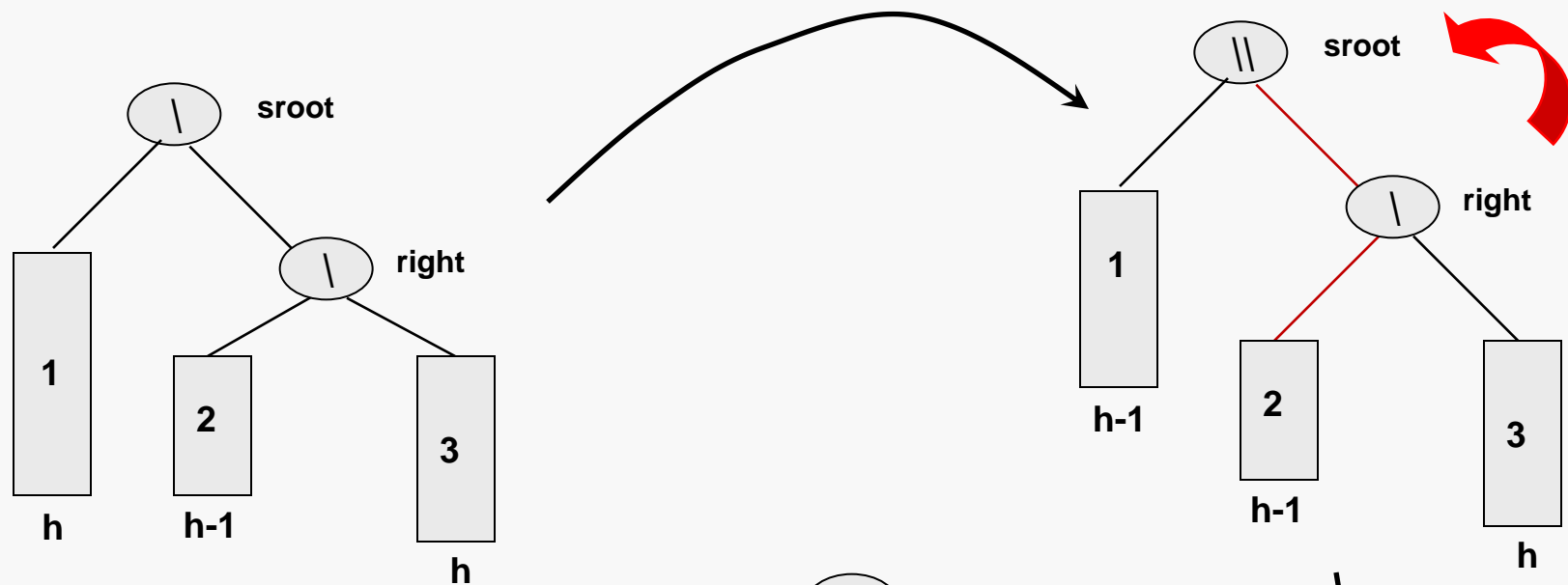
However, rebalancing after a deletion may require applying single or double rotations at **more than one point** along that path.

As usual, there are cases…

Here, we will make the following assumptions:

- the lowest imbalance occurs at the node `sroot` (a subtree root)

- the deletion occurred in the left subtree of `sroot`

Suppose we have the subtree on the left <u>prior</u> to deletion and that on the right <u>after</u> deletion:

sroot \

right \

1
h

2
h-1

3
h

|| **sroot**

1
h-1

\ **right**

2
h-1

3
h

— **right**

— **sroot**

1
h-1

2
h-1

3
h

Note: "*right-higher*" refers to the balance factor of the root of the right subtree (labeled right here).
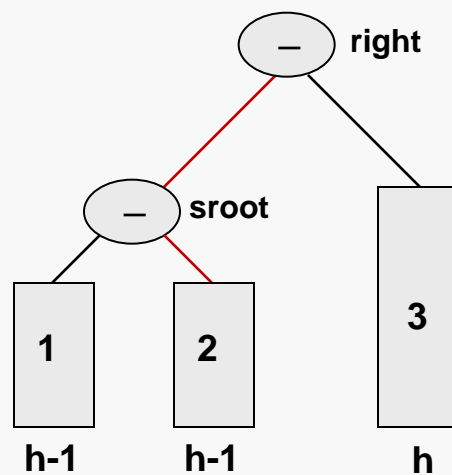
Then a single *left rotation* at sroot will rebalance the subtree.

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

Again, a single left rotation at sroot will rebalance the subtree.
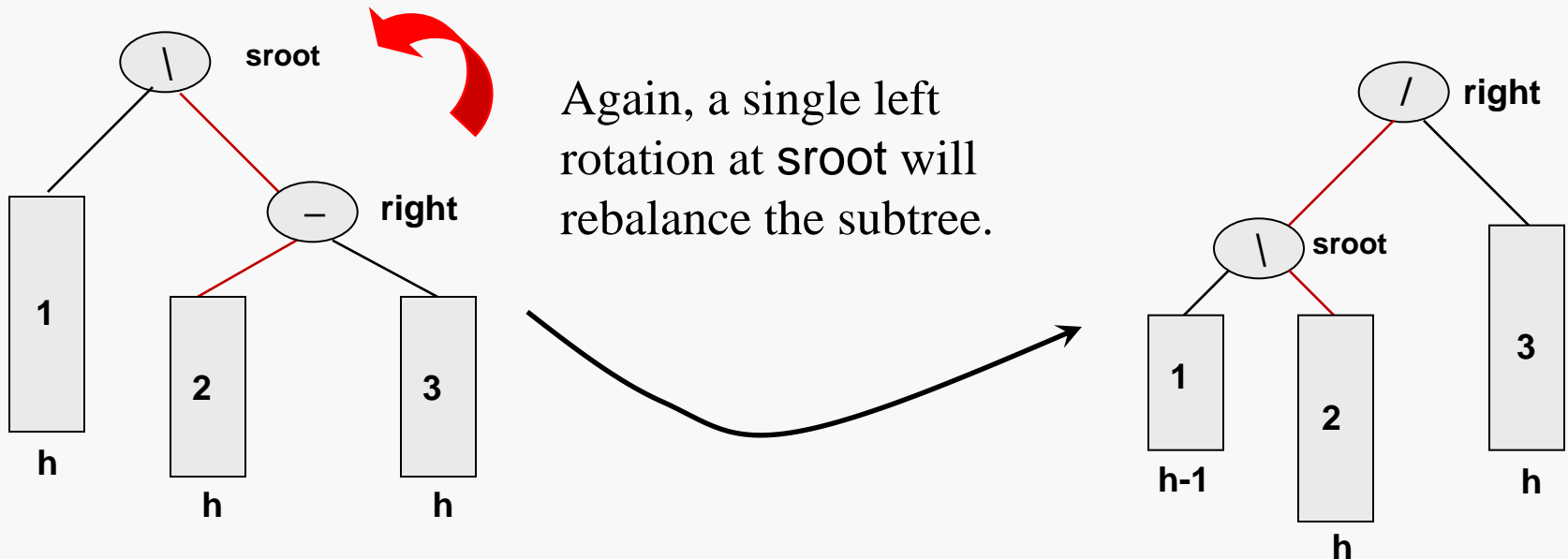
The difference is the resulting balance factor at the old subtree root node, sroot, which depends upon the original balance factor of the node right.

If the right subtree root was left-higher, we have the following situation:
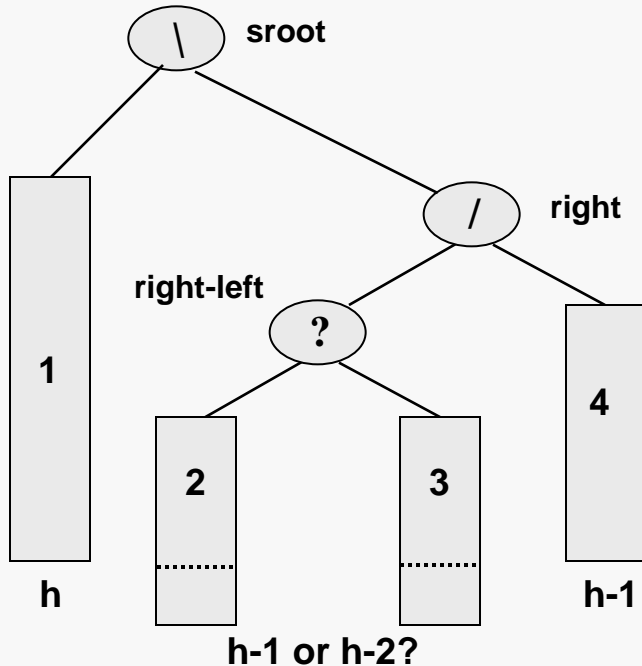
Deleting a node from the left subtree of sroot now will cause sroot to become double right higher.

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…

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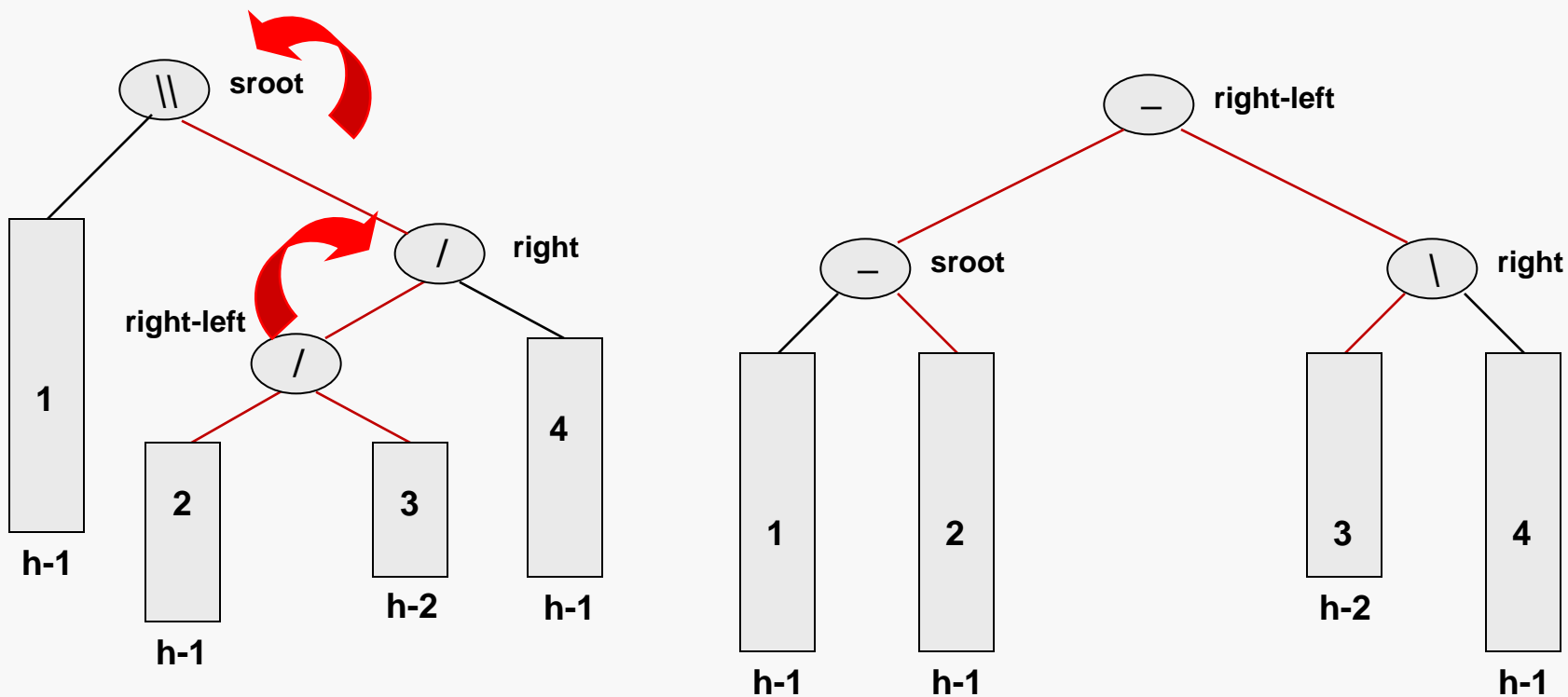
If the right-left subtree root was right-higher, we obtain:
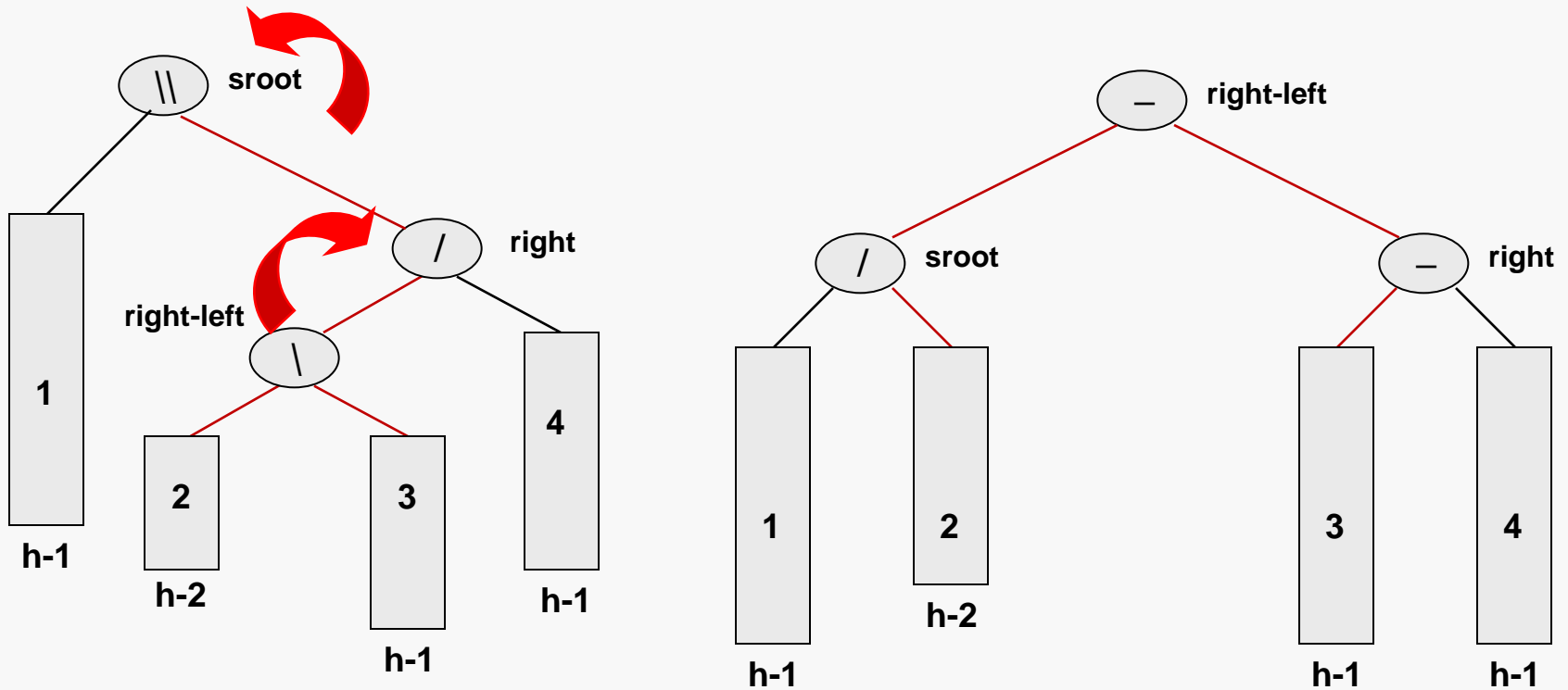
And, finally, if the right-left subtree root was equal-height, we'd obtain a tree where all three of the labeled nodes have equal-height.

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.