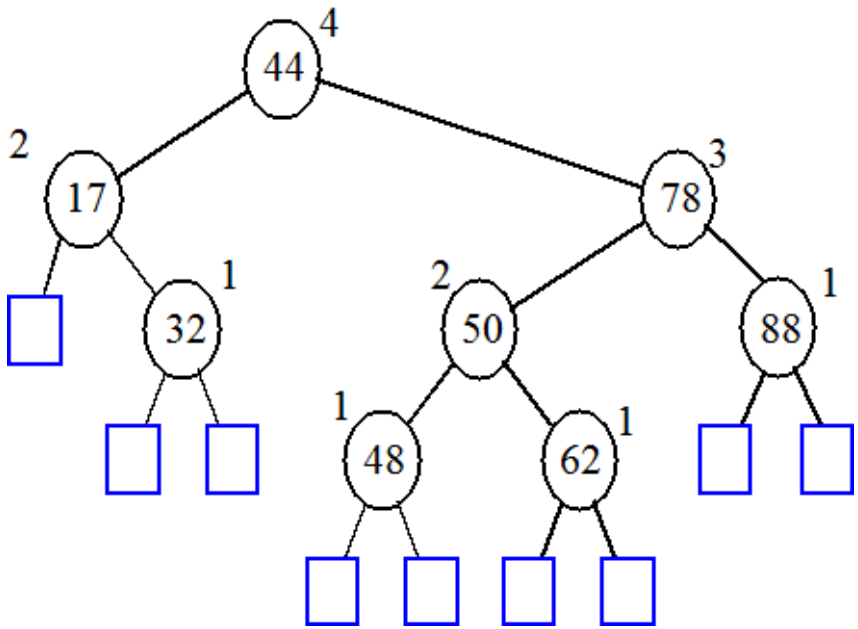
AVL Trees

Balanced Binary Tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree $O(logN)$
- Such trees are called *balanced binary search trees*. Examples are AVL tree, red-black tree.

AVL Tree (Georgy Adelson-Velsky and Evgenii Landis' tree, 1962)

- Height of a node
  - The height of a leaf is 1. The height of a null pointer is zero.
  - The height of an internal node is the maximum height of its children plus 1



Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

AVL Tree (Cont'd)

- An *AVL tree* is a binary search tree in which for every node in the tree, the height of the left and right subtrees differ by at most 1.

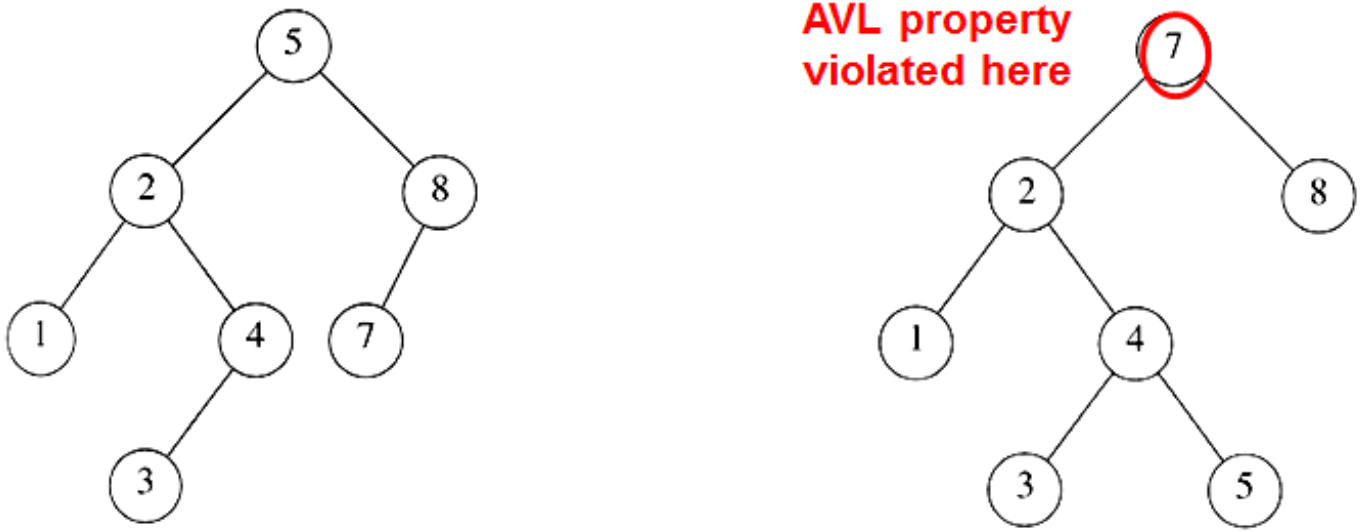


Figure 4.32 Two binary search trees. Only the left tree is AVL.

AVL Tree (Cont'd)

- Let x be the root of an AVL tree of height h
- Let N_h denote the minimum number of nodes in an AVL tree of height h
- Clearly, $N_i \geq N_{i-1}$ by definition
- We have

$$N_h \geq N_{h-1} + N_{h-2} + 1 \geq 2N_{h-2} + 1 > 2N_{h-2}$$

- By repeated substitution, we obtain the general form

$$N_h > 2^i N_{h-2i}$$

- The boundary conditions are: $N_1 = 1$ and $N_2 = 2$. This implies that $h = O(logN_h)$.
- More precisely, the height of an n-node AVL tree is approx. $1.44 \log_2 n$.
- Thus, many operations (searching, insertion, deletion) on an AVL tree will take $O(logN)$ time.

Smallest AVL Tree

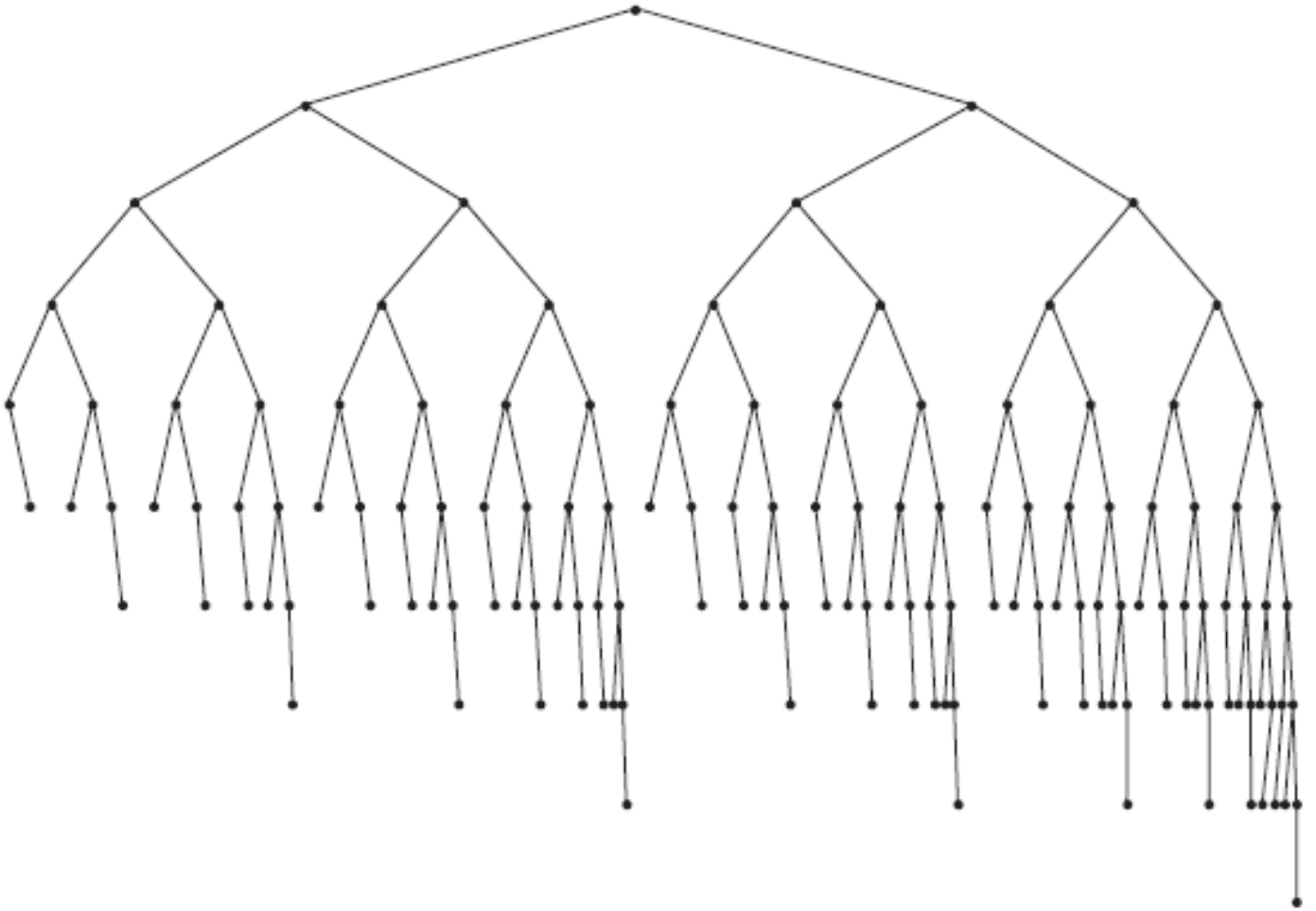


Figure 4.33 Smallest AVL tree of height 9

Rotations

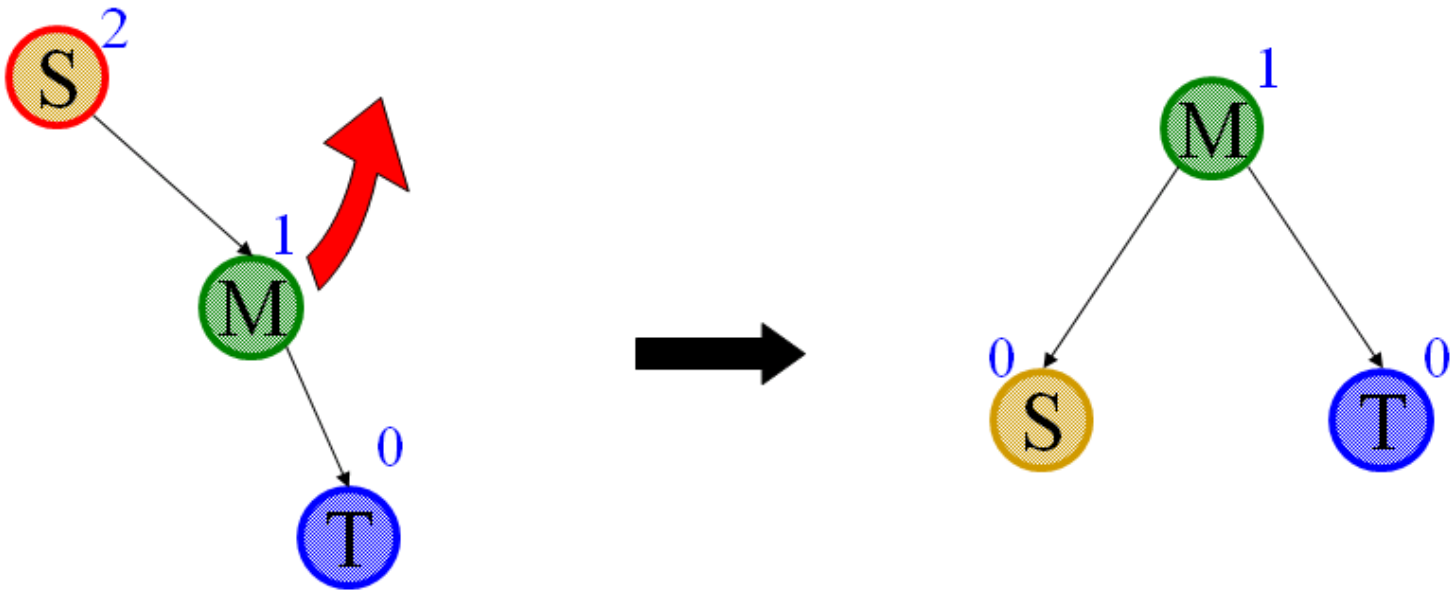
- When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- This is done using *single rotations* or *double rotations*.
- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

Insertion

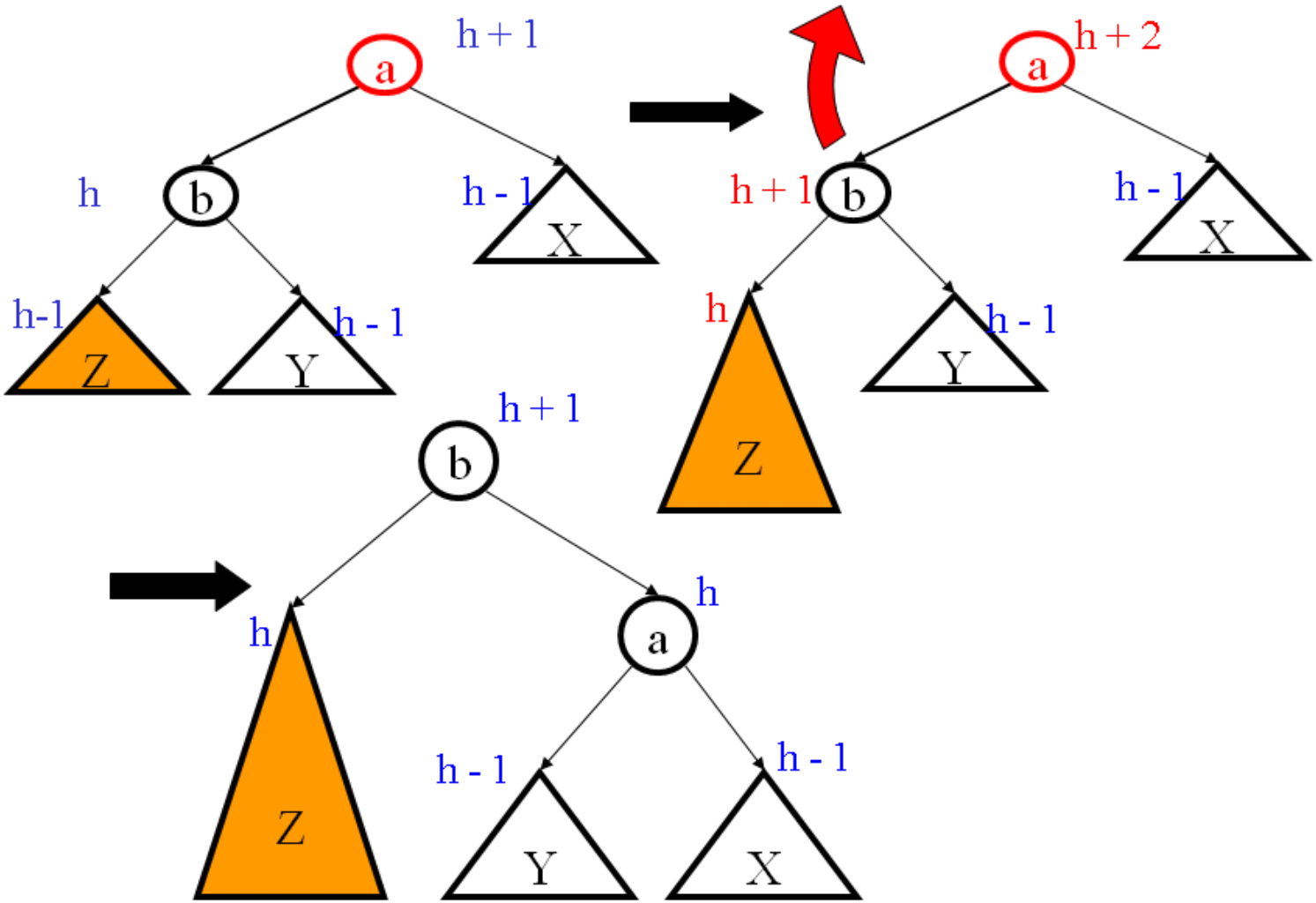
- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
- If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation [next slide].
- For insertion, once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

Single rotation

Basic operation used in AVL trees: A right child could legally have its parent as its left child.



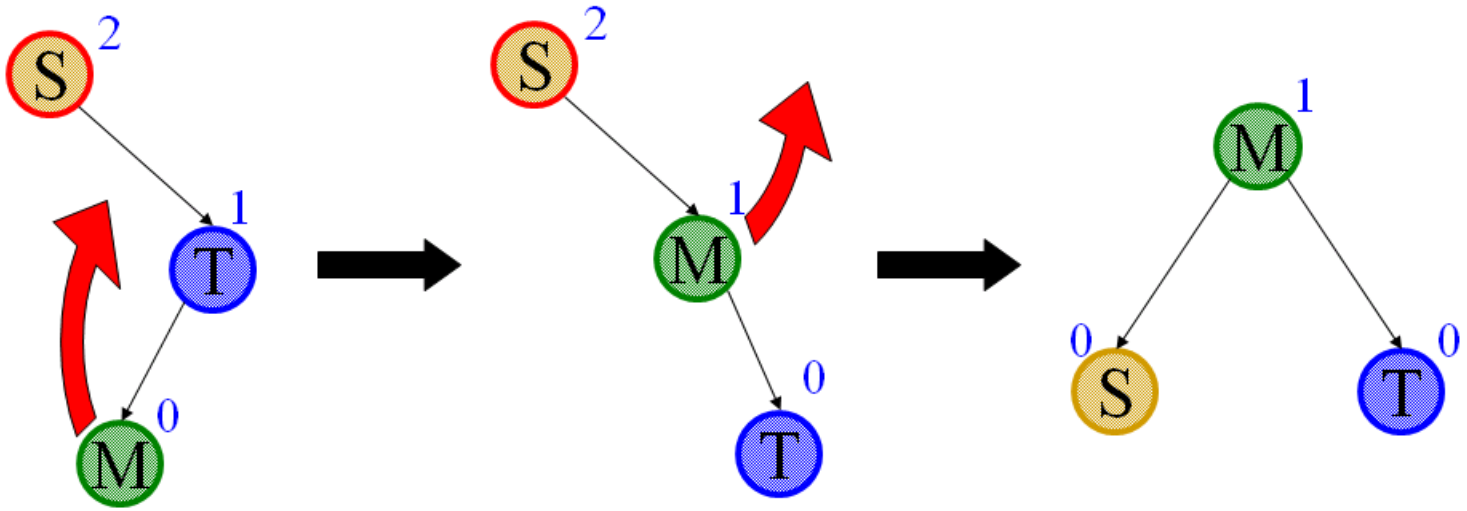
General Case: Insert Unbalances



Properties of General Insert + Single Rotation

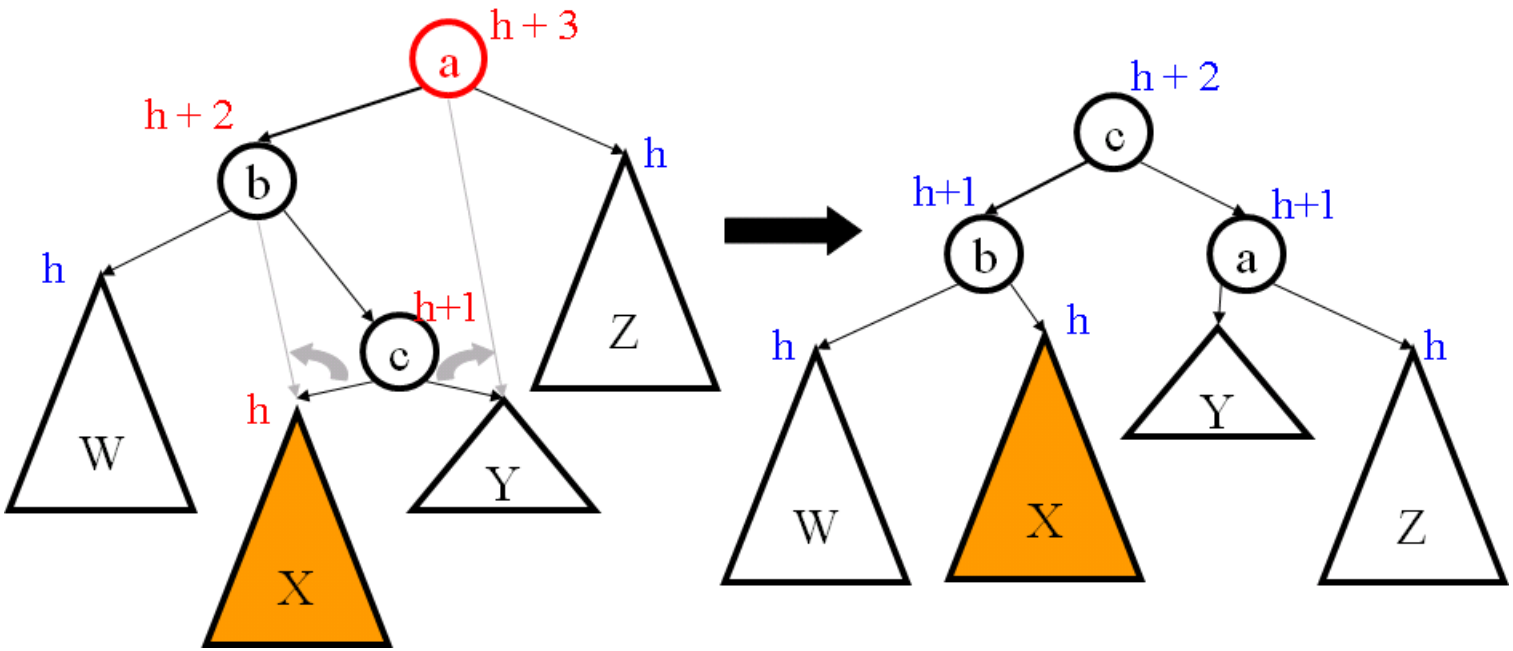
- Restores balance to a lowest point in tree where imbalance occurs
- After rotation, height of the subtree (in the example, h+1) is the same as it was before the insert that imbalanced it
- Thus, no further rotations are needed anywhere in the tree!

Double rotation



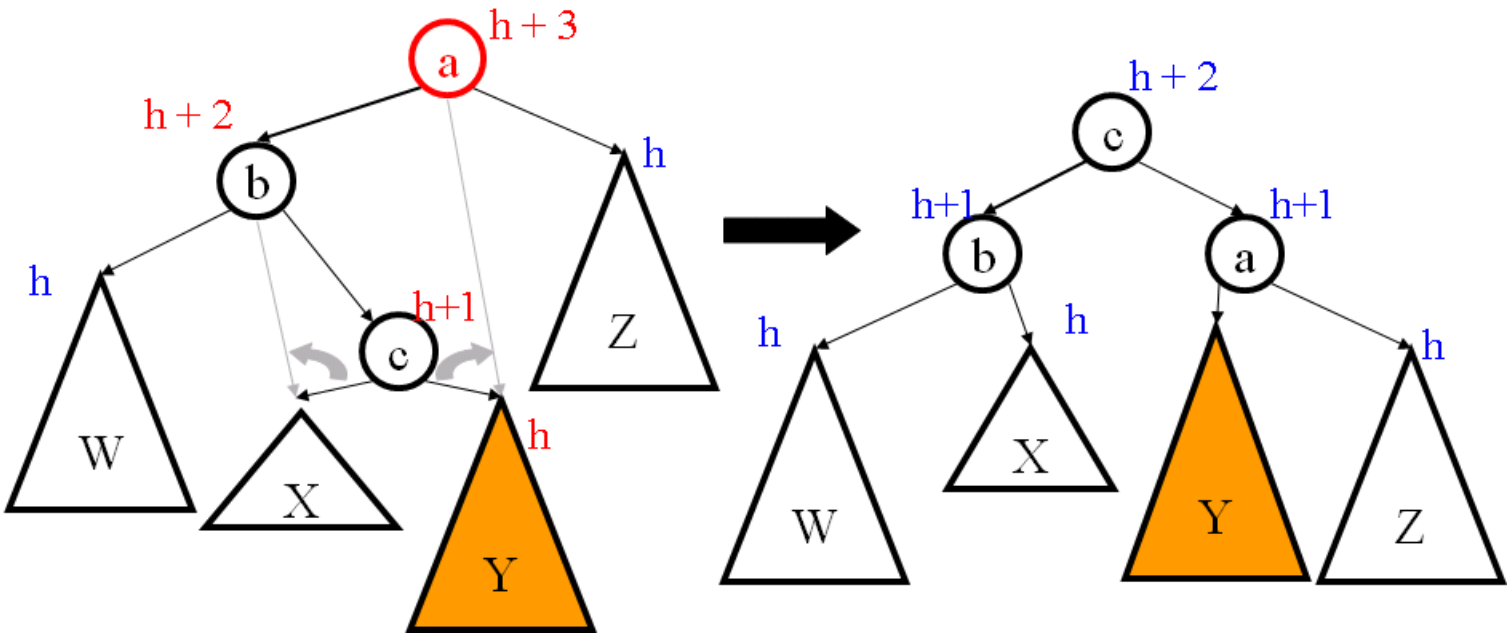
General Double Rotation

- Initially: inserting into X unbalances tree (root height goes to $h + 3$)
- "Zig-zag" to pull up c - restores root height to $h + 2$, left subtree height to h



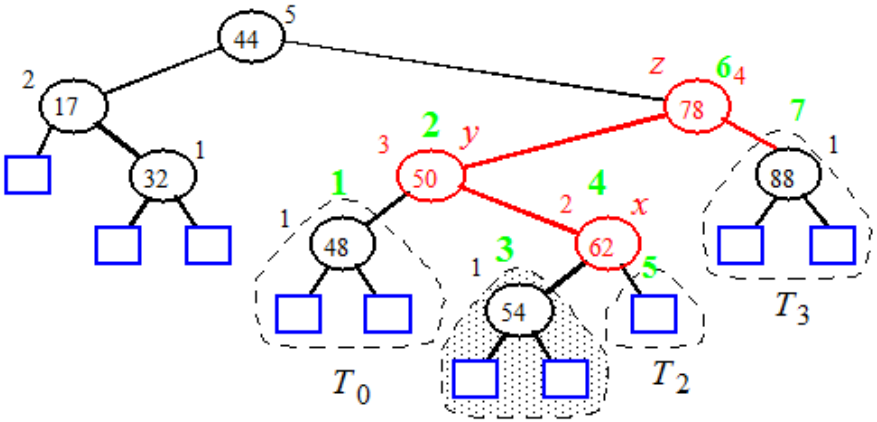
Another Double Rotation Case

- Initially: inserting into Y unbalances tree (root height goes to $h+2$)
- "Zig-zag" to pull up c - restores root height to $h+1$, left subtree height to h

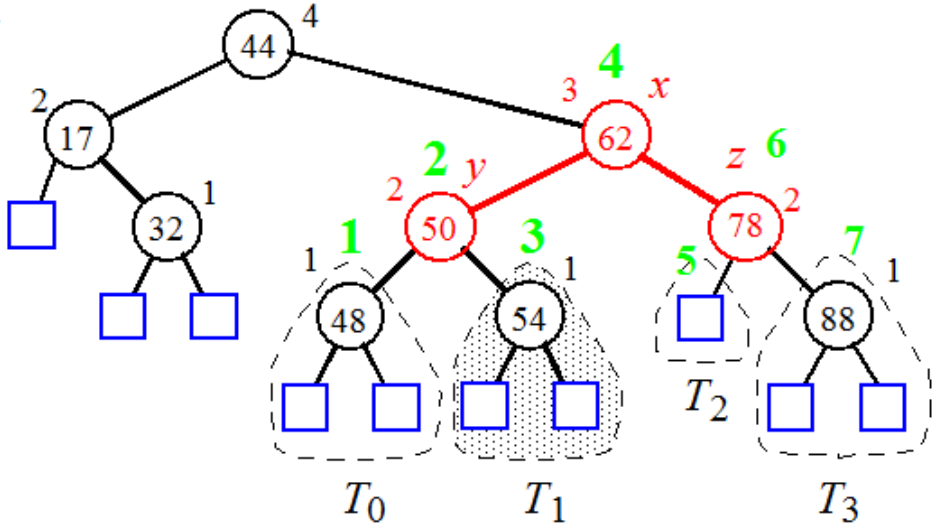


Example - Rebalance after Insertion

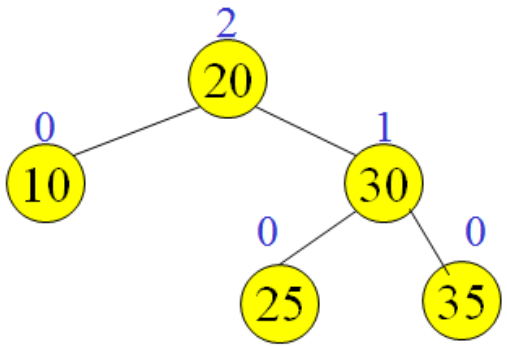
unbalanced...



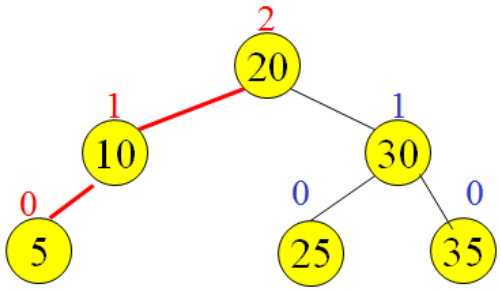
...balanced

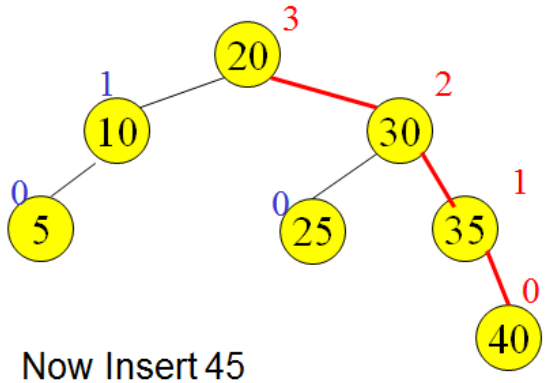


Another Example



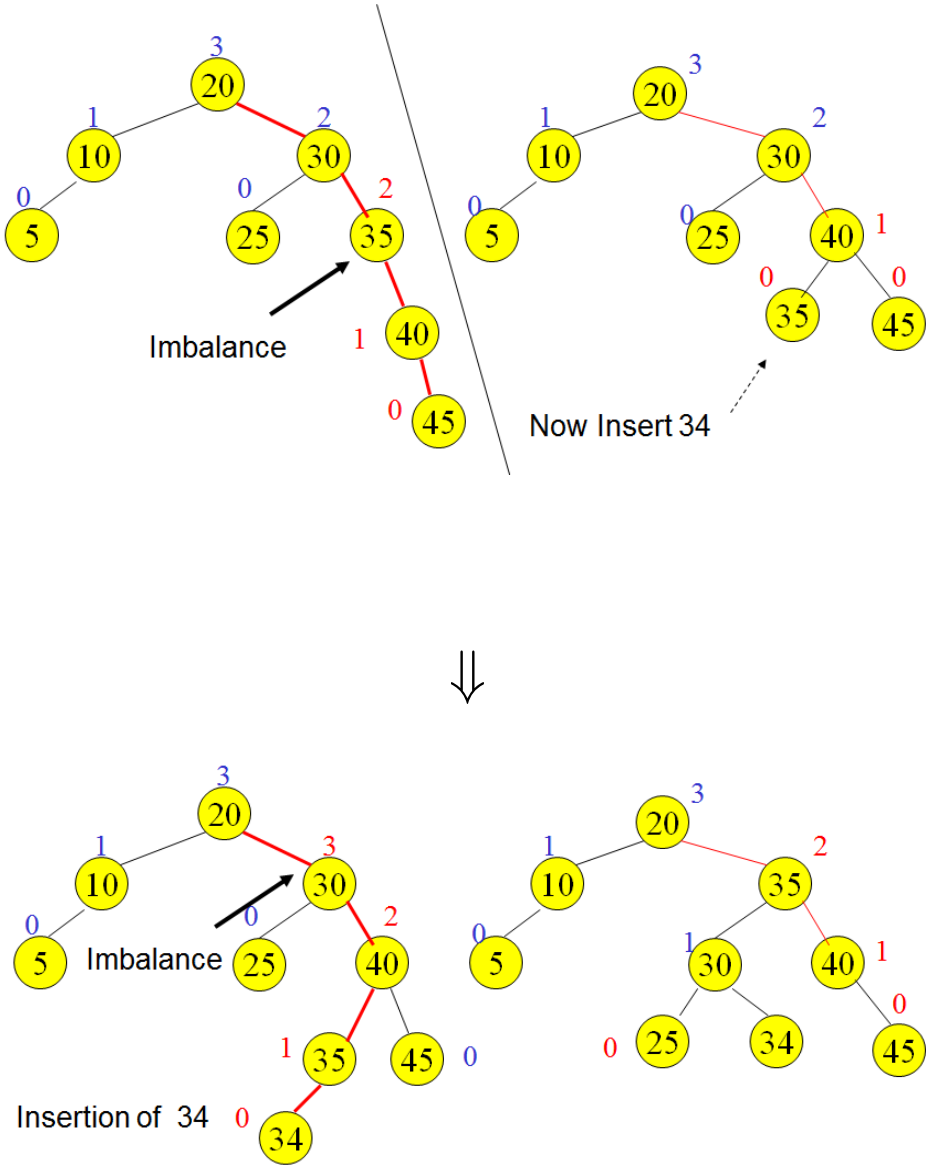
Insert 5, 40





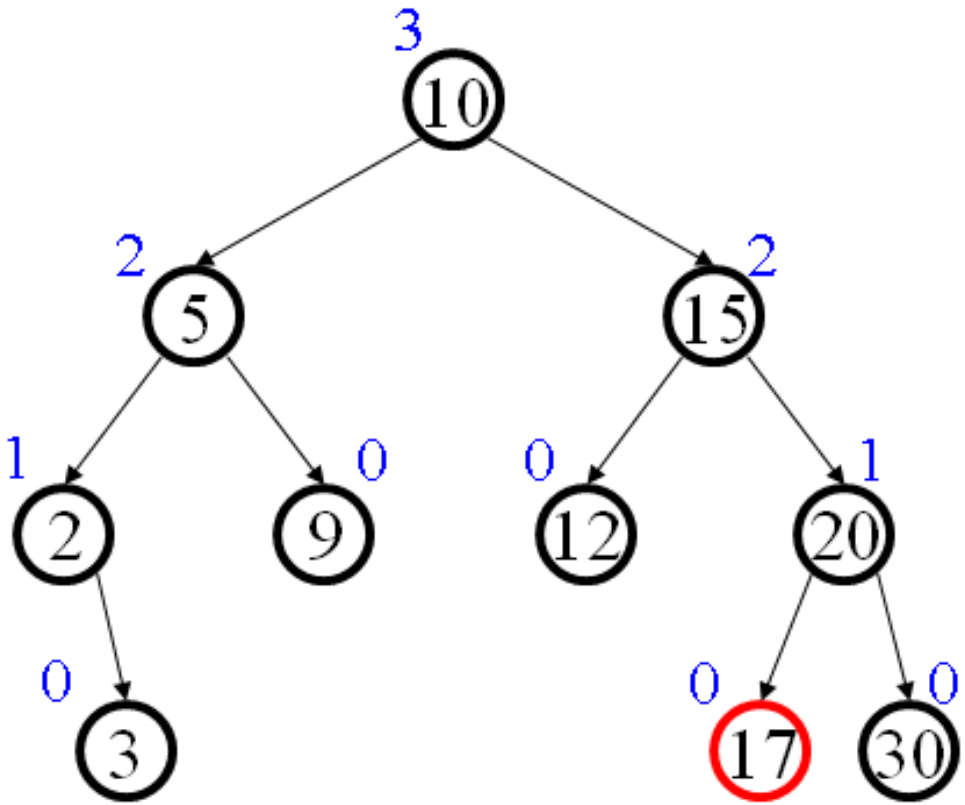
Now Insert 45

Another Example (Cont'd)



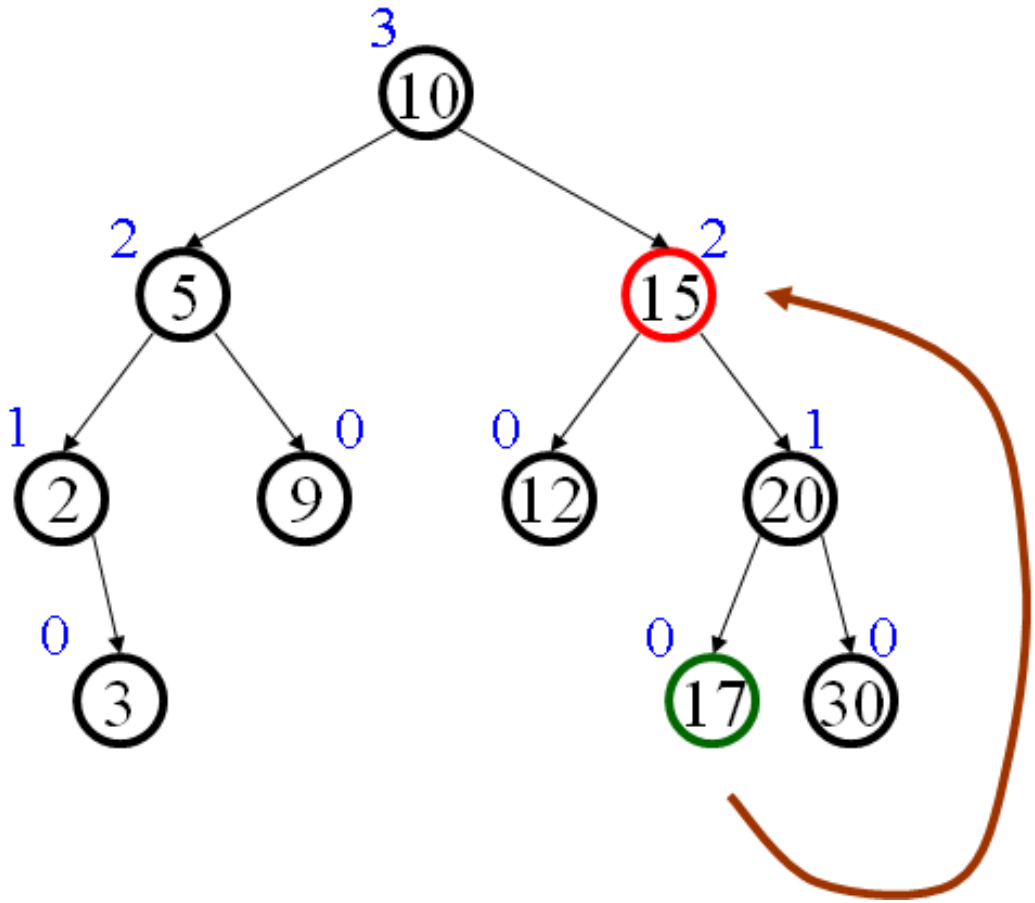
Deletion (Really Easy Case)

Delete(17)



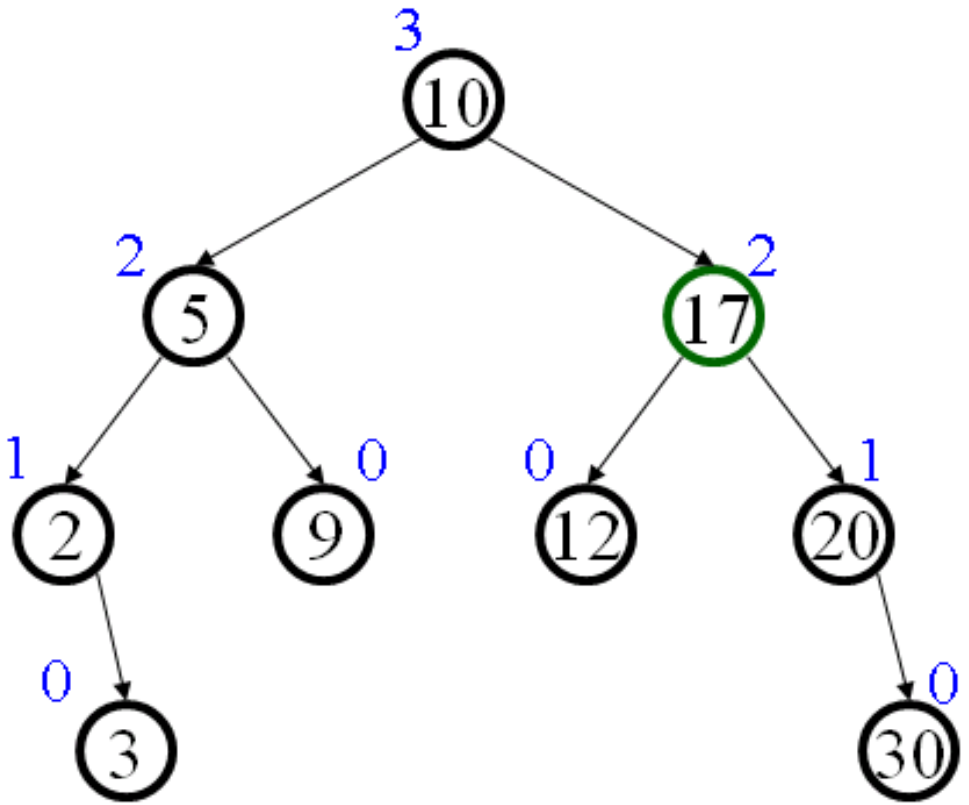
Deletion (Pretty Easy Case)

Delete(15)



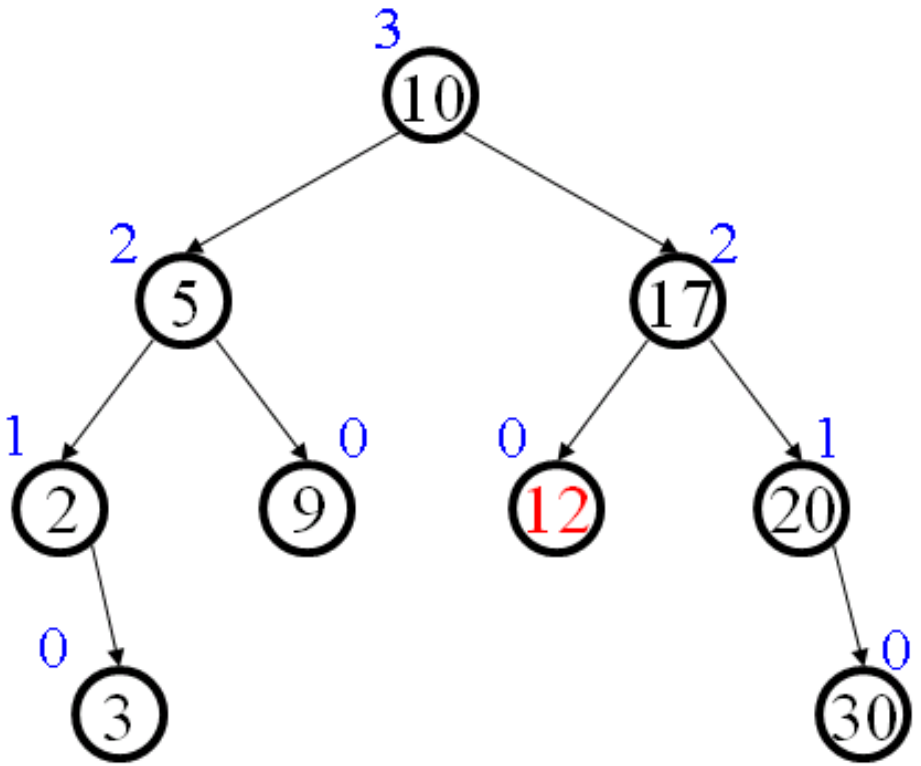
Deletion (Pretty Easy Case cont'd)

Delete(15)

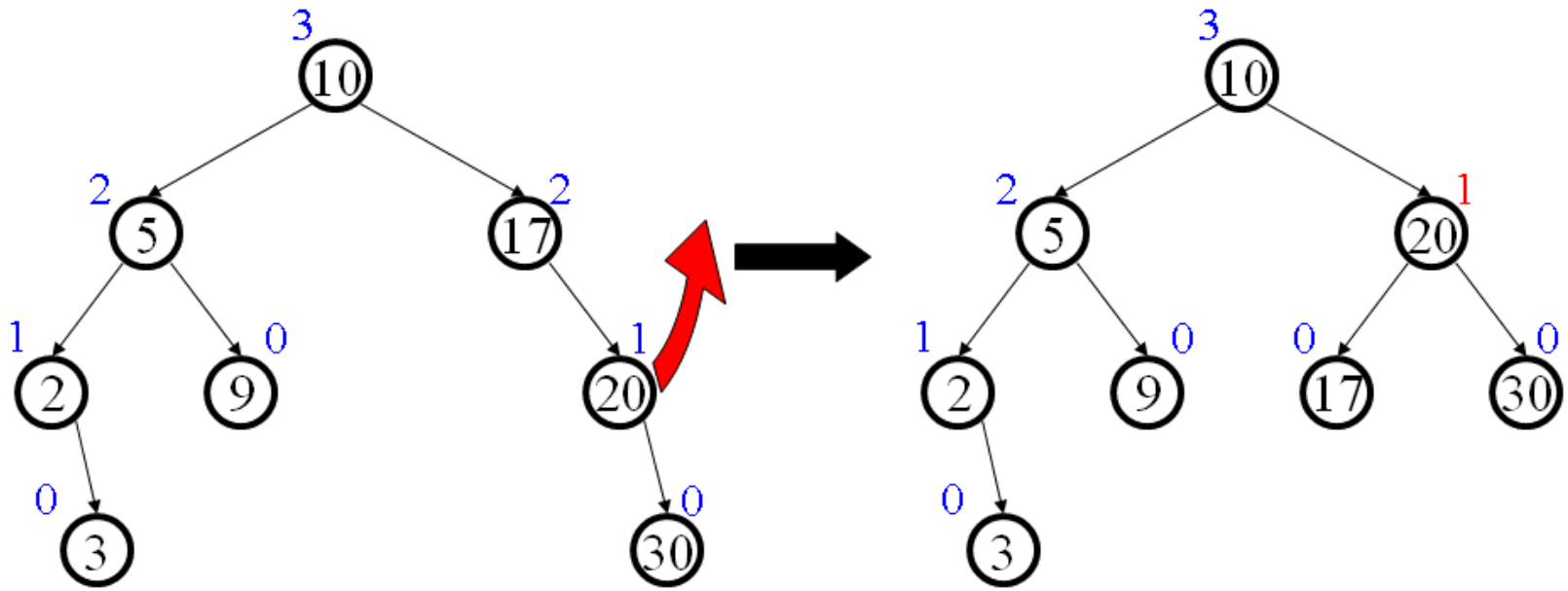


Deletion (Hard Case #1)

Delete(12)



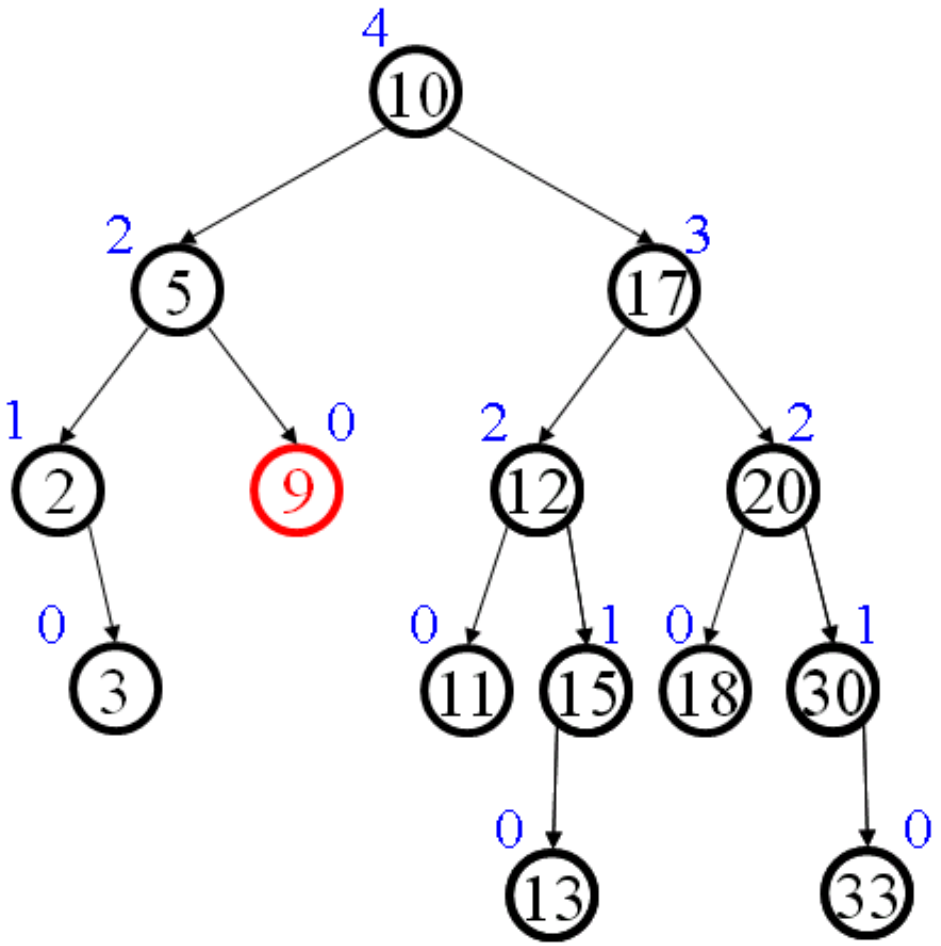
Single Rotation on Deletion



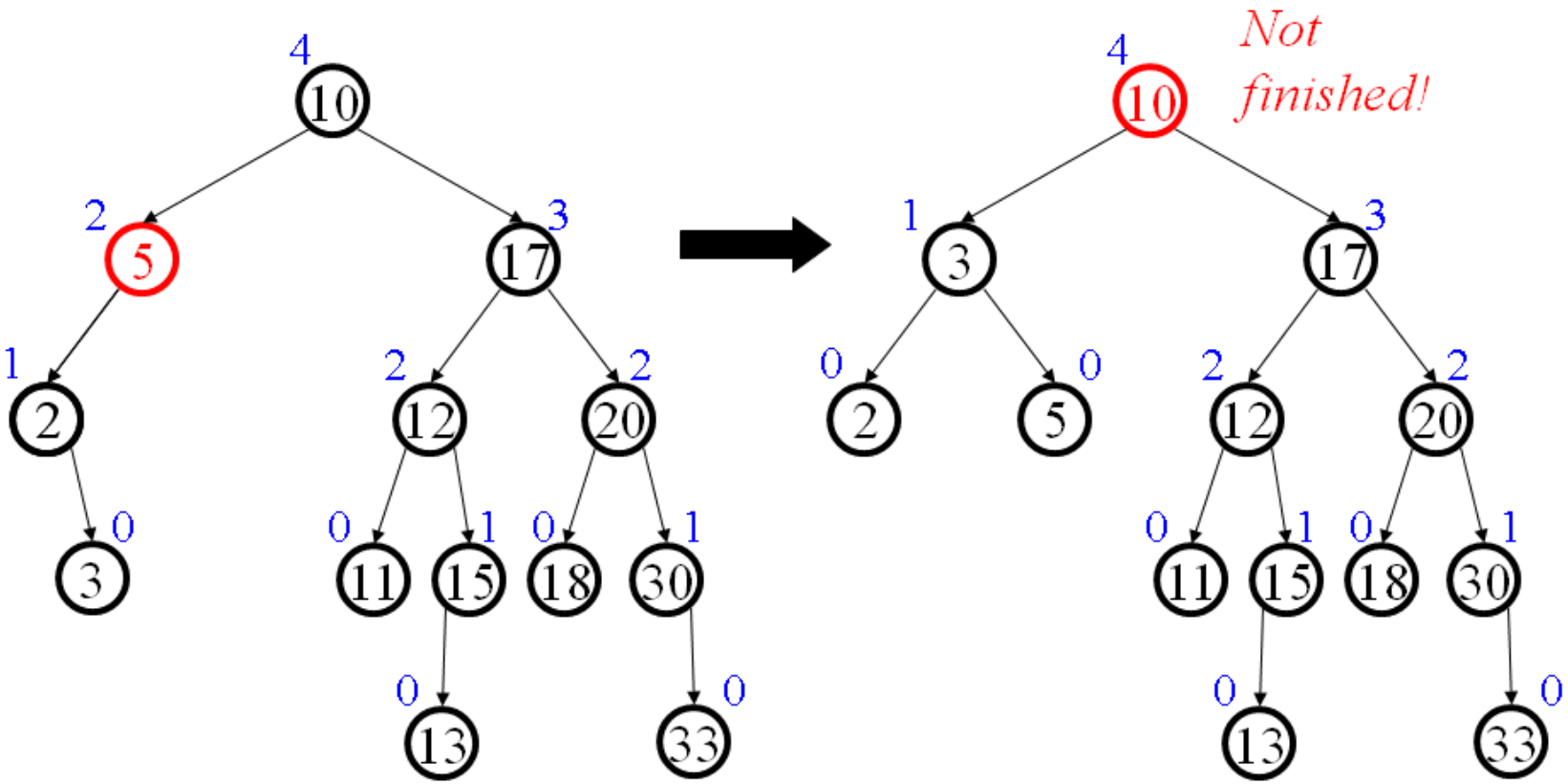
What is different about deletion than insertion?

Deletion (Hard Case)

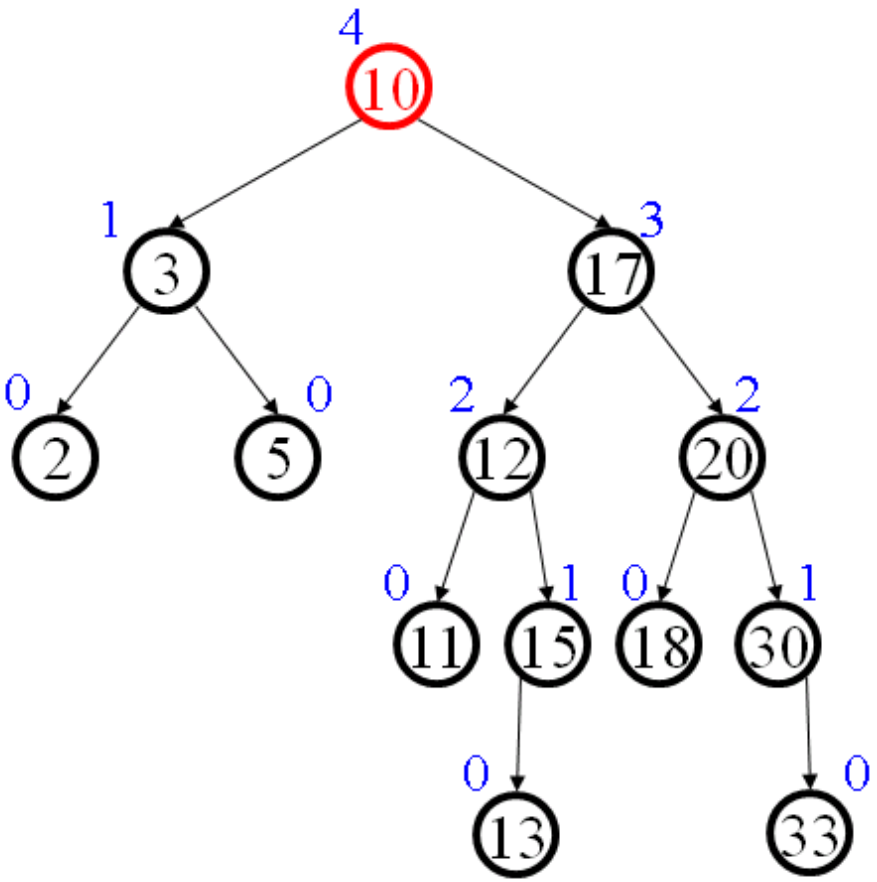
Delete(9)



Double Rotation on Deletion



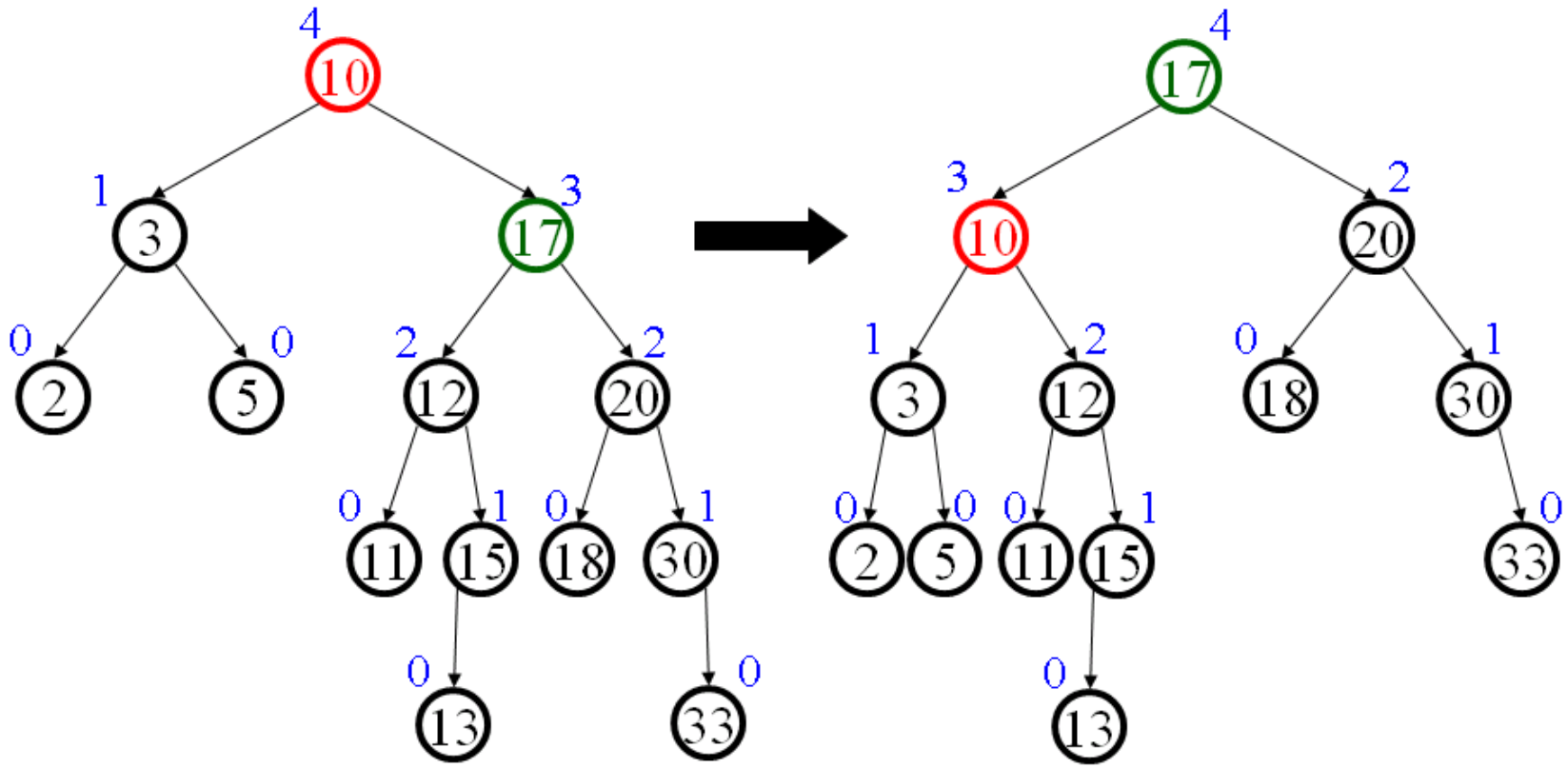
Deletion with Propagation



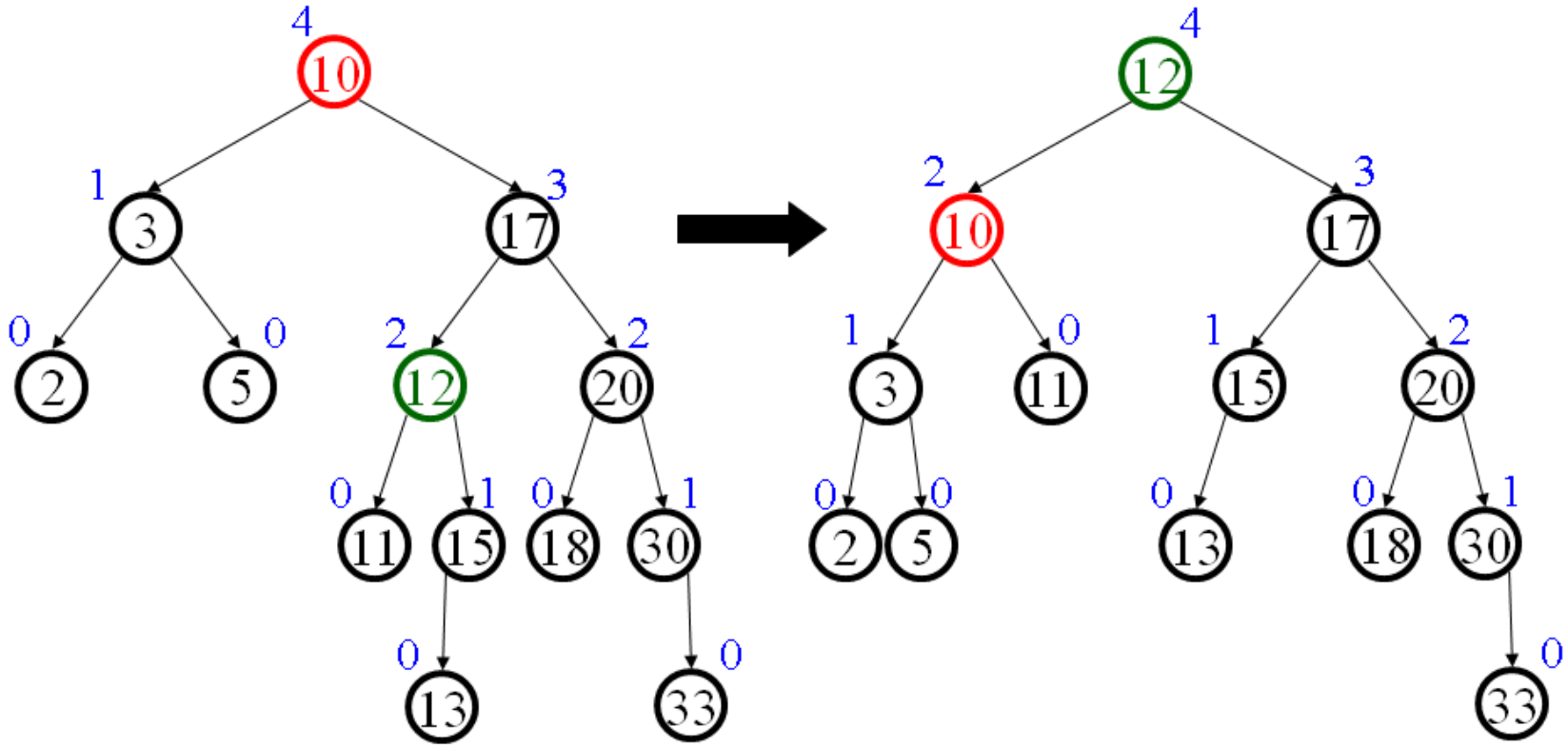
What different about this case?

We get to choose whether to single or double rotate!

Propagated Single Rotation



Propagated Double Rotation



Imbalance may propagate upward so that many rotations may be needed.

Pros and Cons of AVL Trees

- Pros and Cons of AVL Trees
  1. Search is O(log N) since AVL trees are always balanced.
  2. Insertion and deletions are also O(log n)
  3. The height balancing adds no more than a constant factor to the speed of insertion.
- Arguments against using AVL trees:
  1. Difficult to program and debug; more space for balance factor.
  2. Asymptotically faster but rebalancing costs time.
  3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
  4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Can we guarantee O(log N) performance with less overhead?