AVL Trees

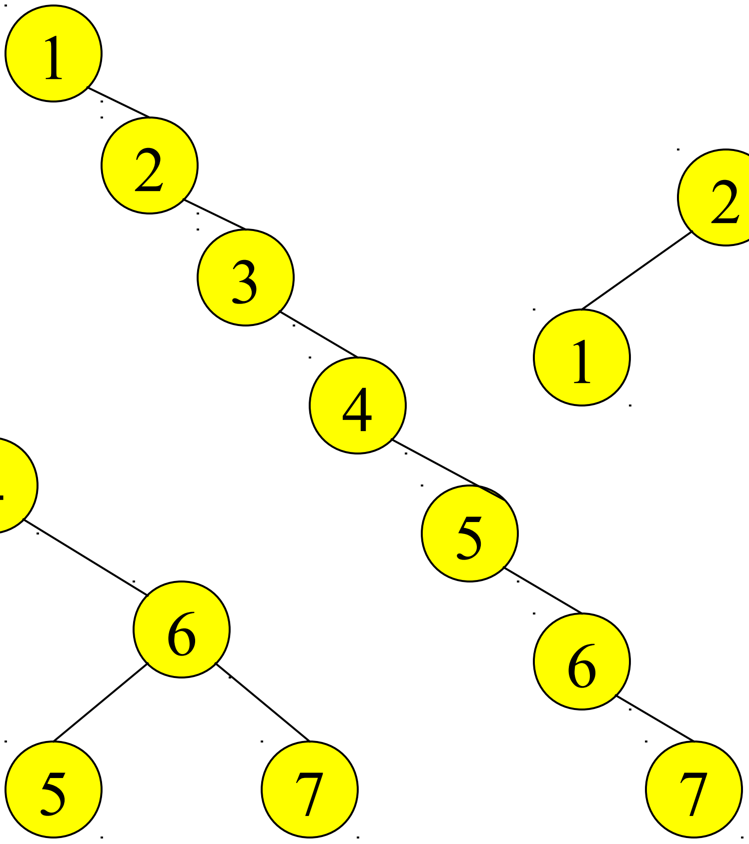
Binary Search Tree - Best Time

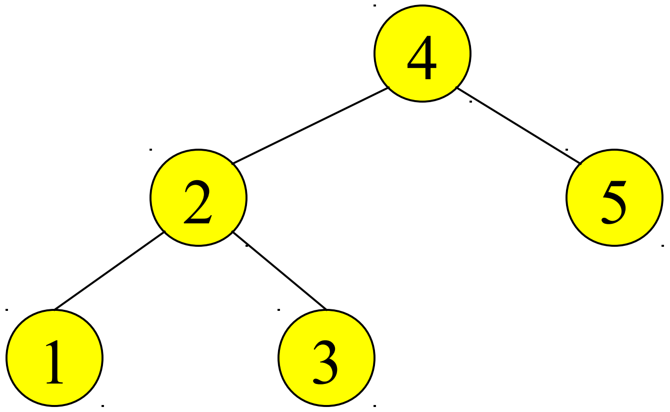
- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

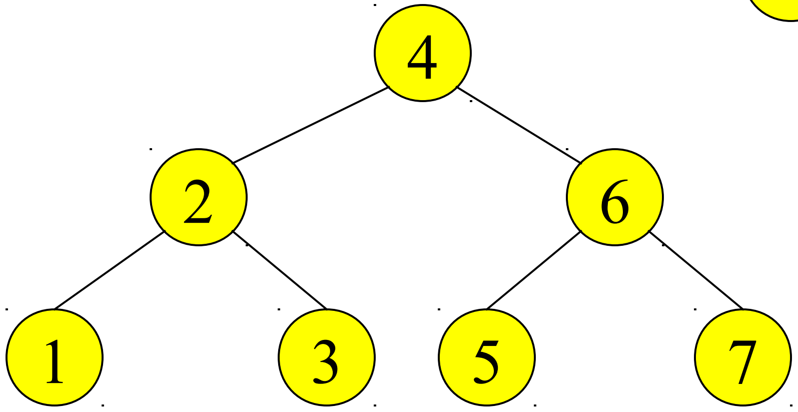
Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree
  - Unbalanced degenerate tree

Balanced and unbalanced BST







Is this “balanced”?

Approaches to balancing trees

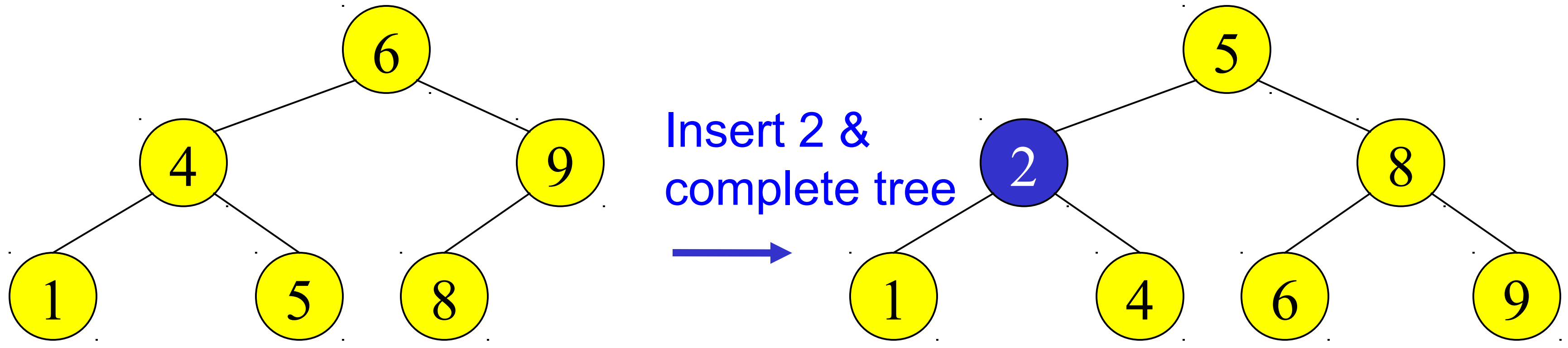
- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly(might not be possible in many cases.)
- Pretty good balance
  - Only allow a little out of balance(gives O(log(n) as we'll prove.)
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - › Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - › Splay trees and other self-adjusting trees
  - › B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree

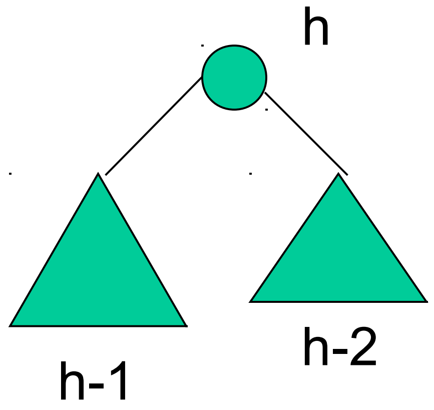


AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

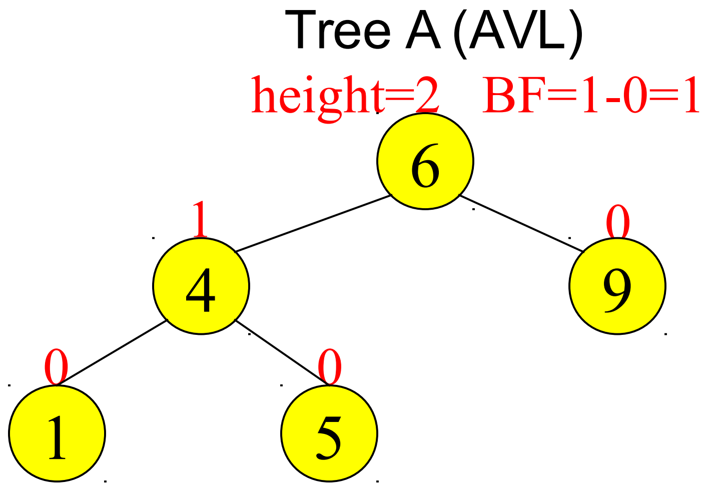
- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - $N(0) = 1$, $N(1) = 2$
- Induction
  - $N(h) = N(h-1) + N(h-2) + 1$
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$

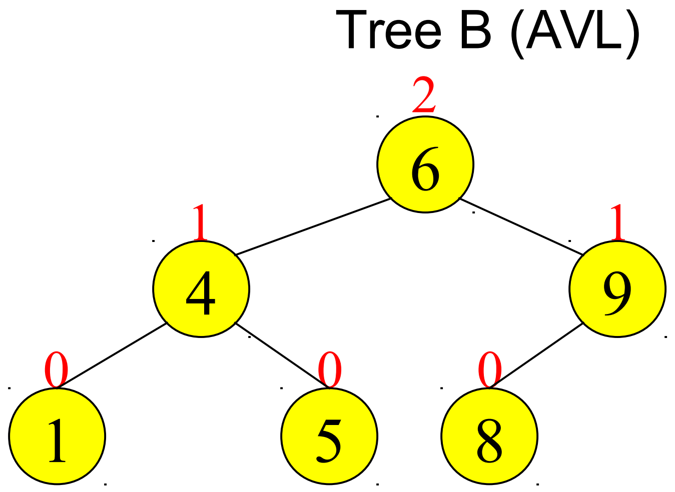


Height of an AVL Tree

- $N(h) \geq \phi^h$ ($\phi \approx 1.62$)
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(logn))

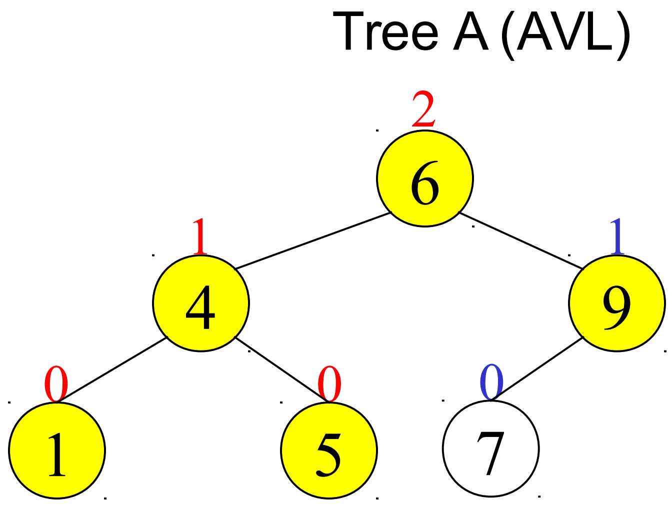
Node Heights

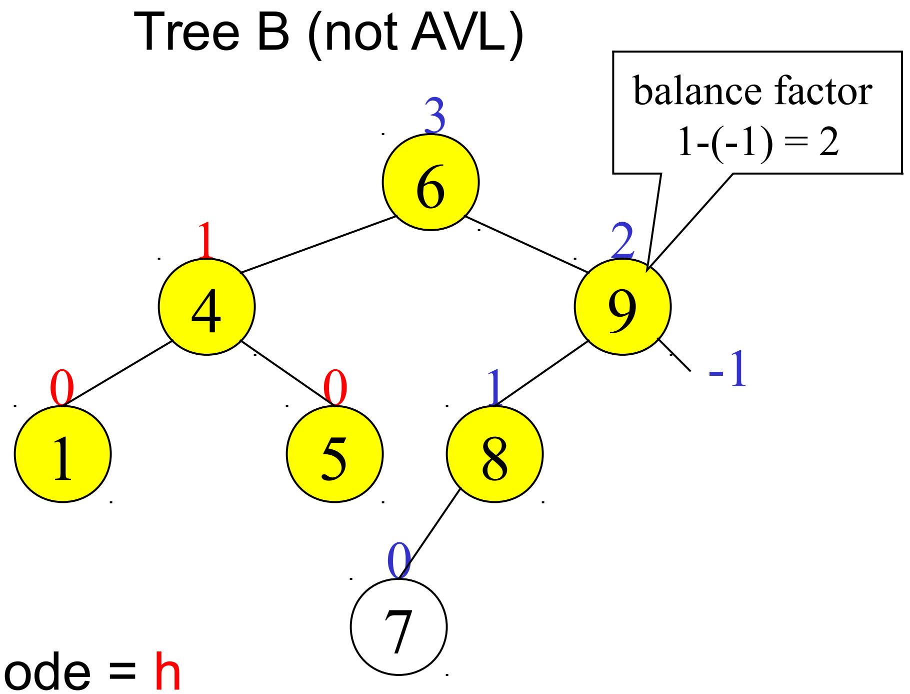




height of node = h
balance factor = $h_{left} - h_{right}$
empty height = -1

Node Heights after Insert 7



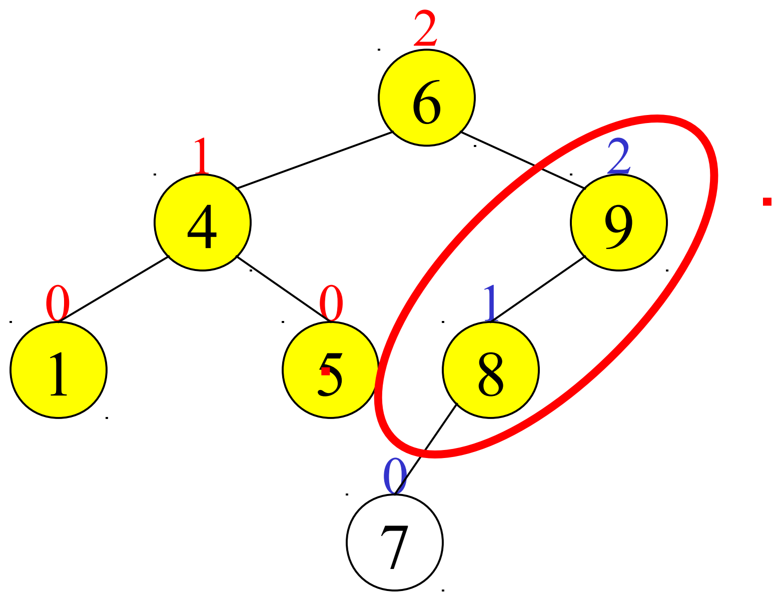


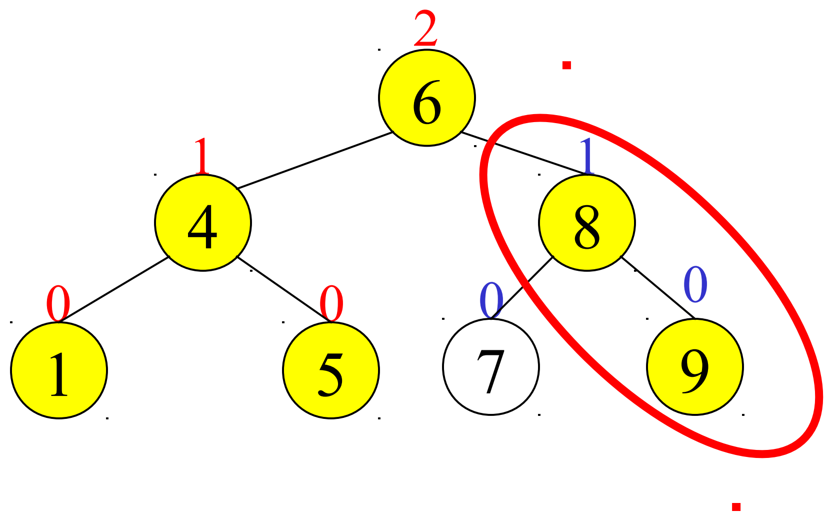
height of node = h
balance factor = $h_{left}-h_{right}$
empty height = -1

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree





Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

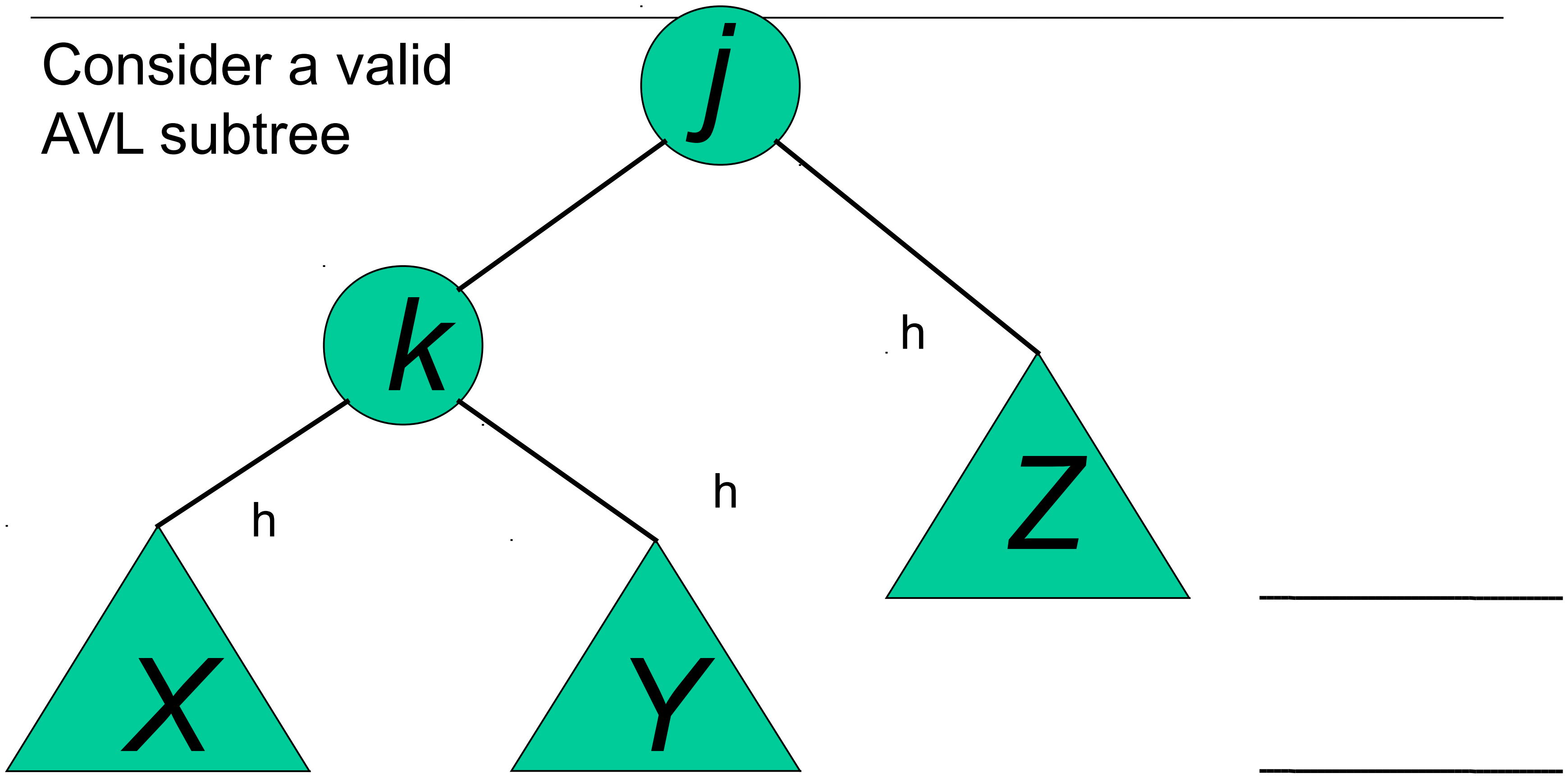
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

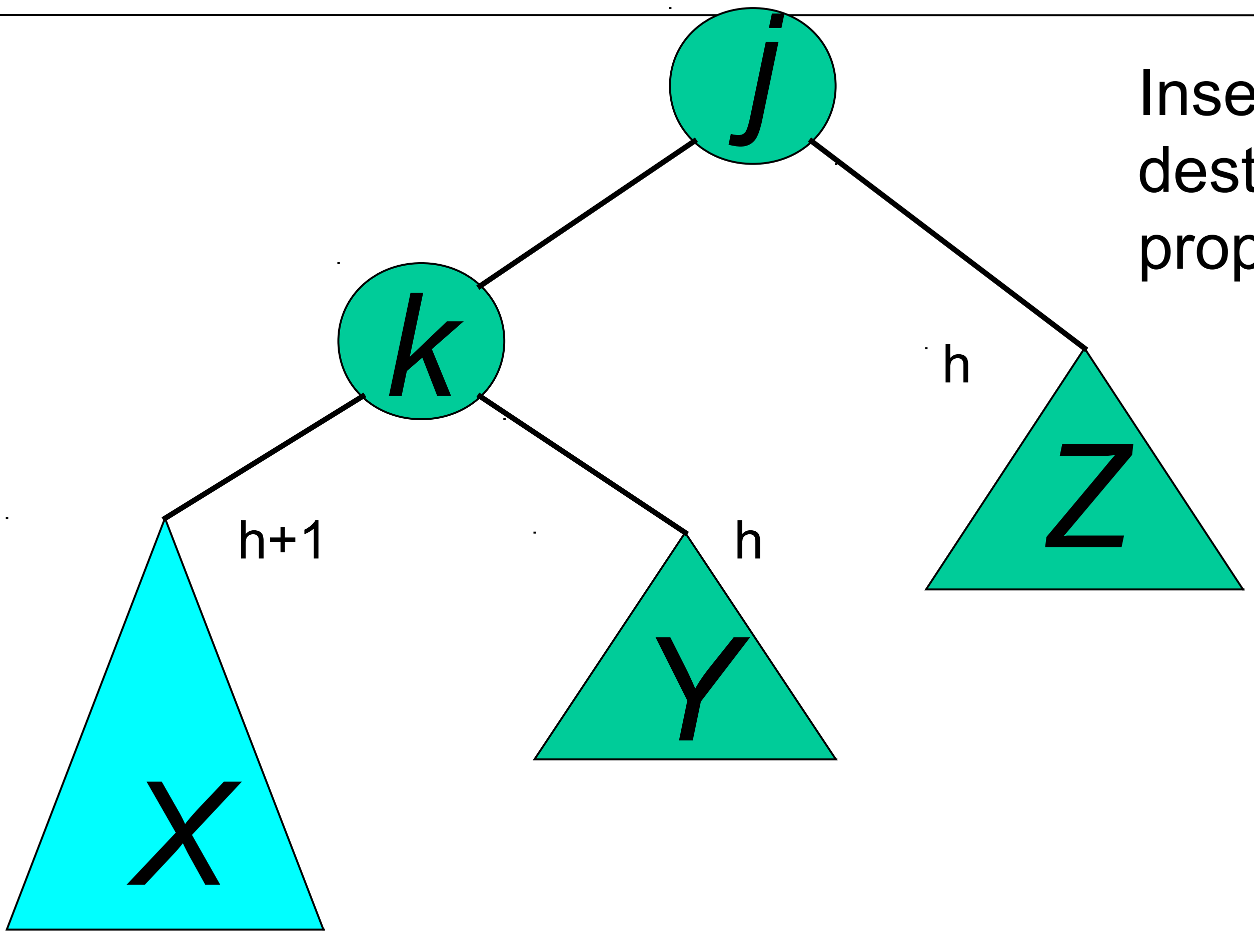
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid AVL subtree

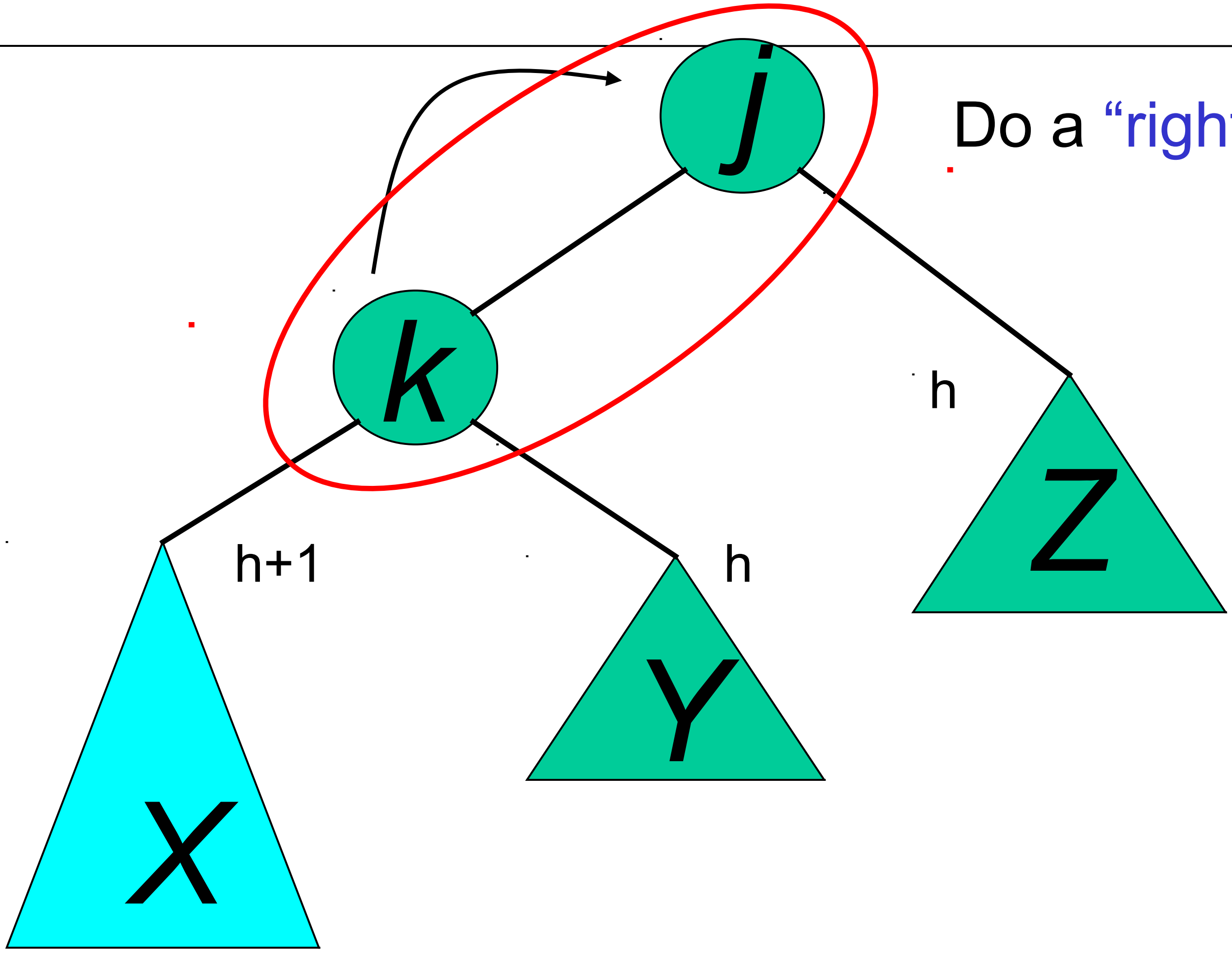


AVL Insertion: Outside Case



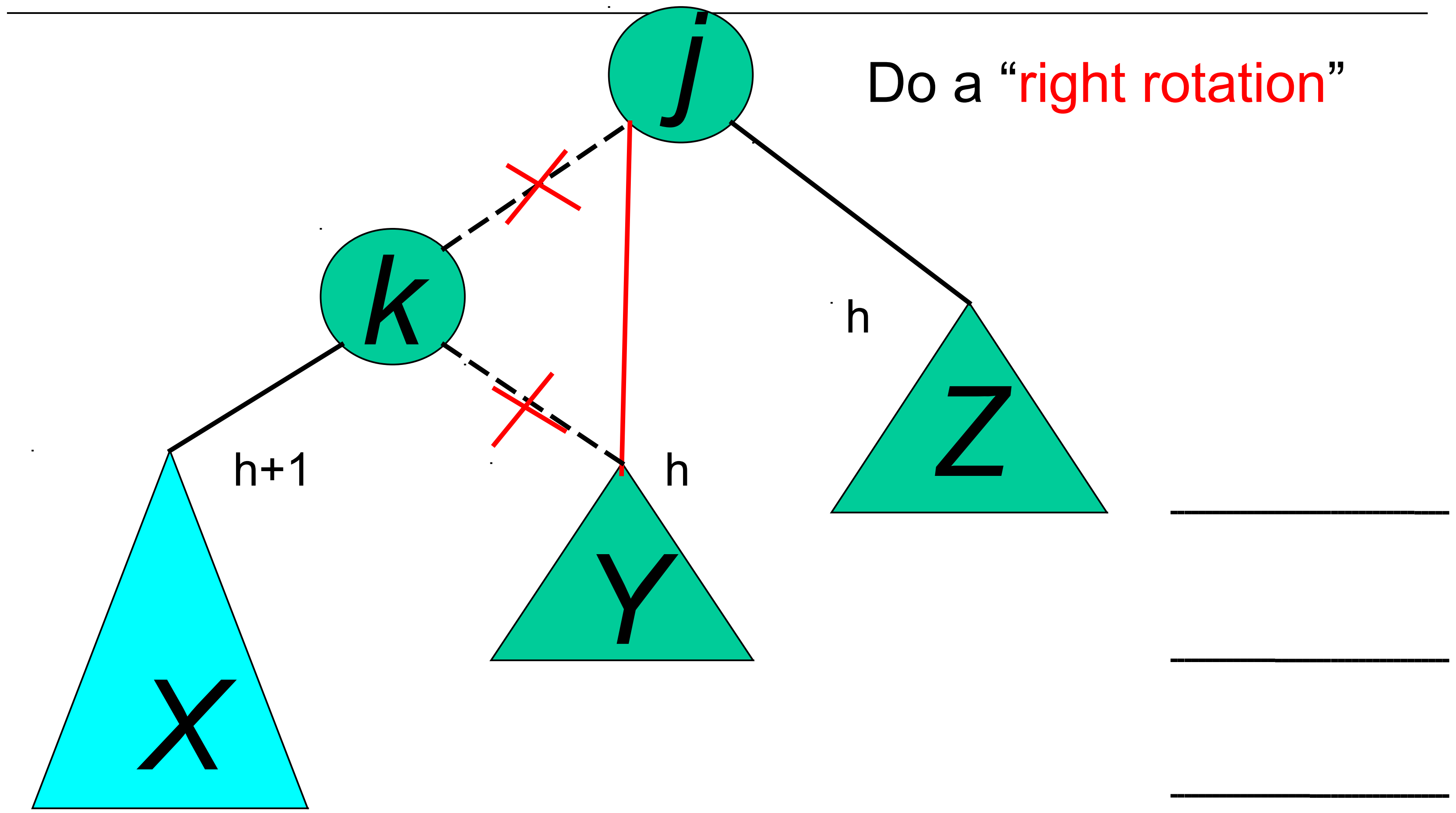
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



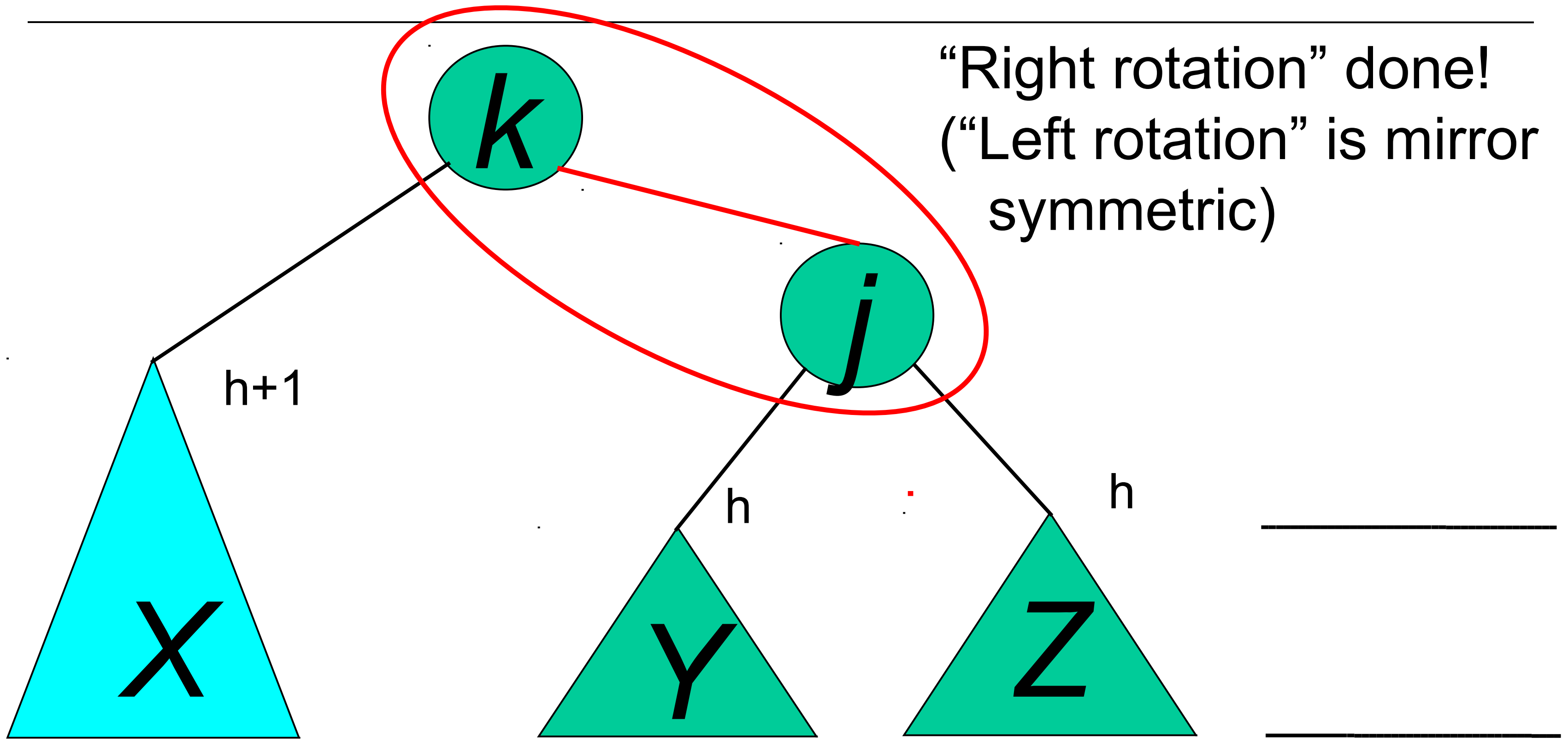
Do a "right rotation"

Single right rotation



Do a "right rotation"

Outside Case Completed

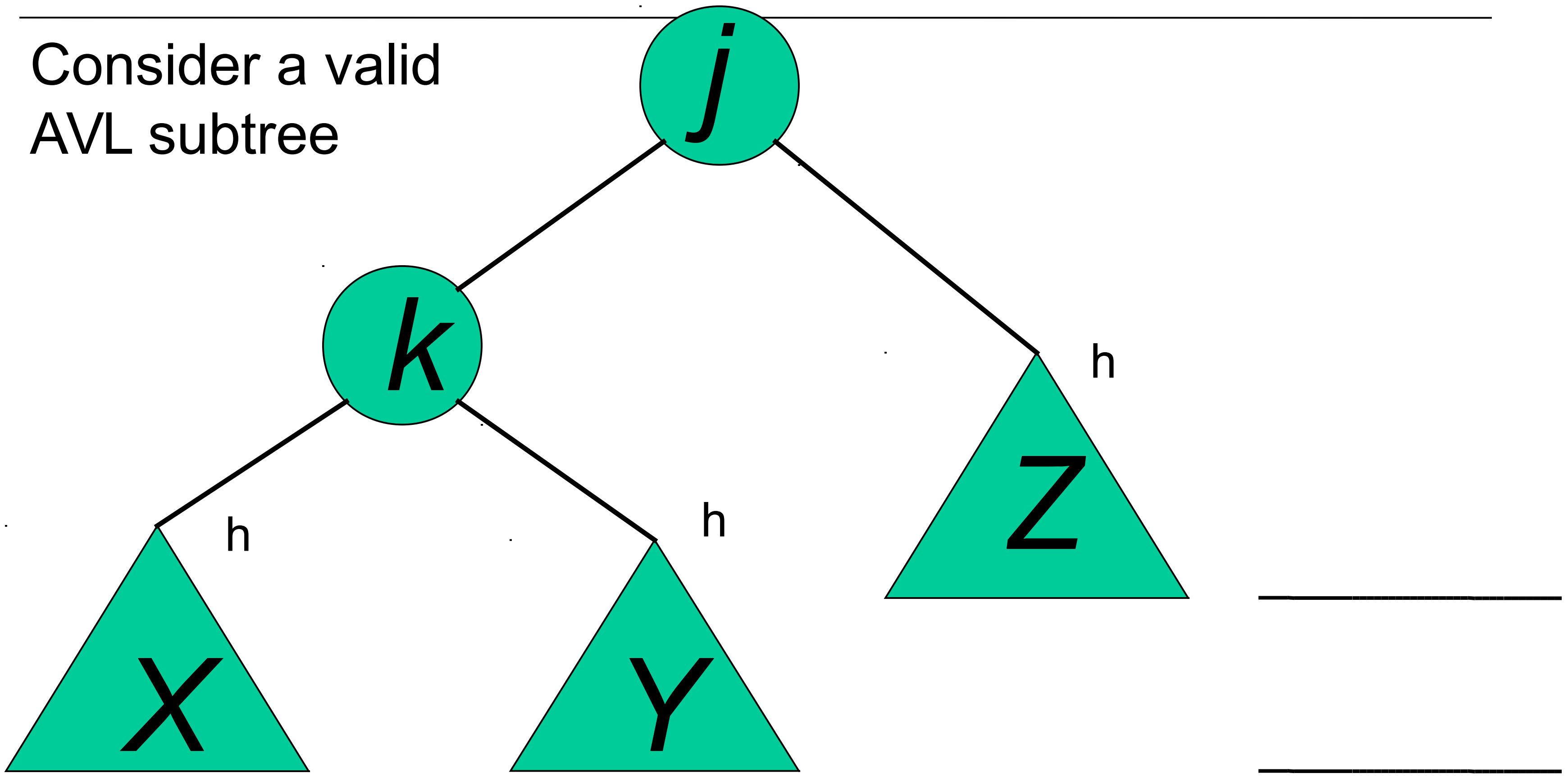


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

AVL Insertion: Inside Case

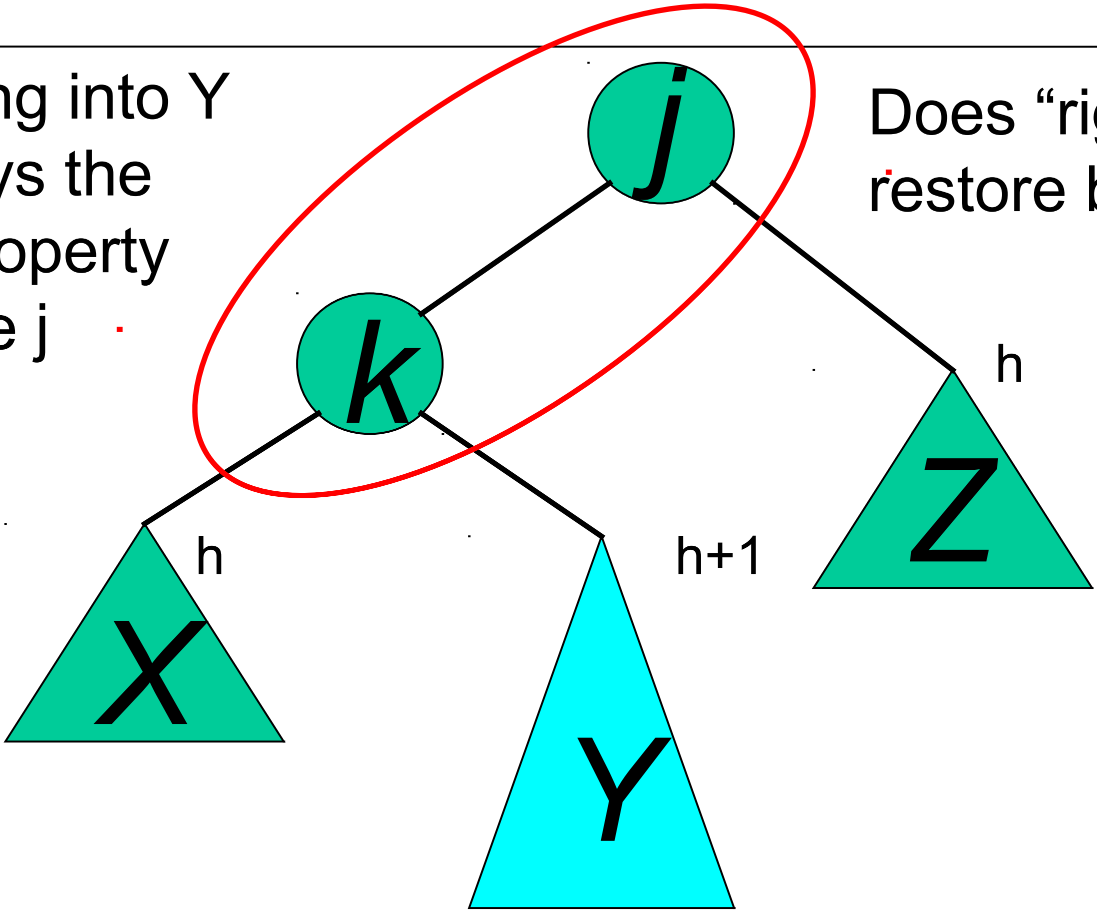
Consider a valid AVL subtree



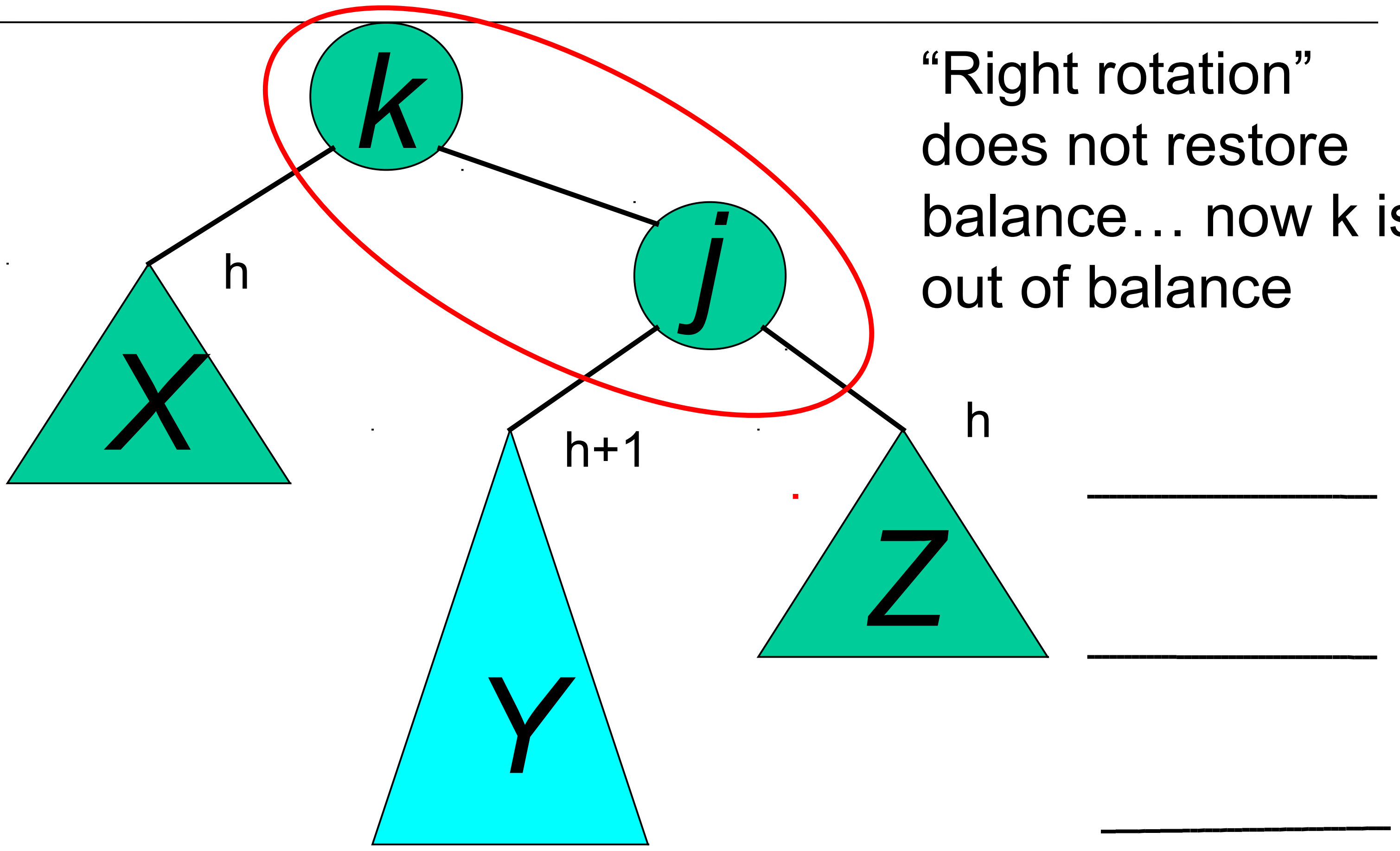
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does "right rotation" restore balance?



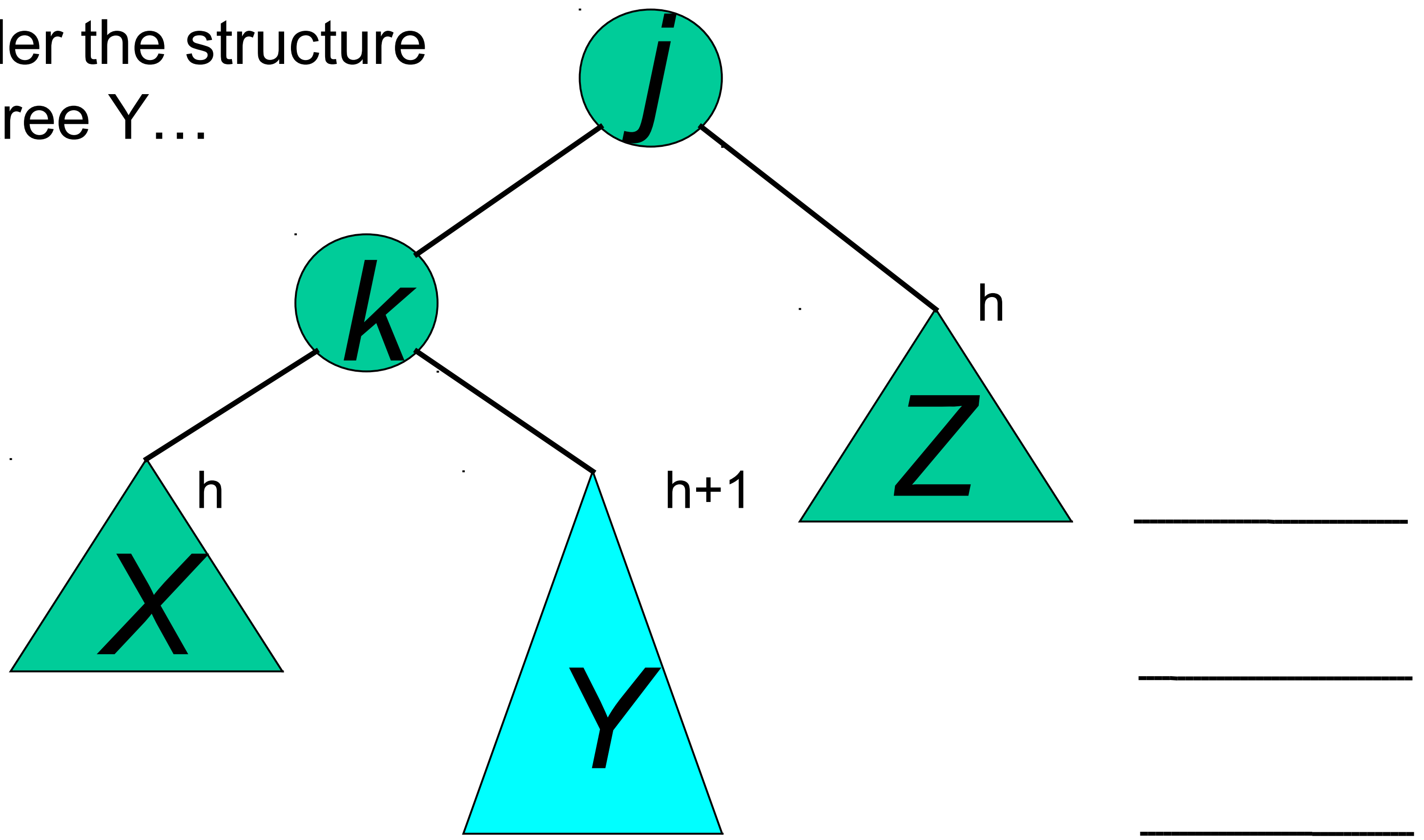
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

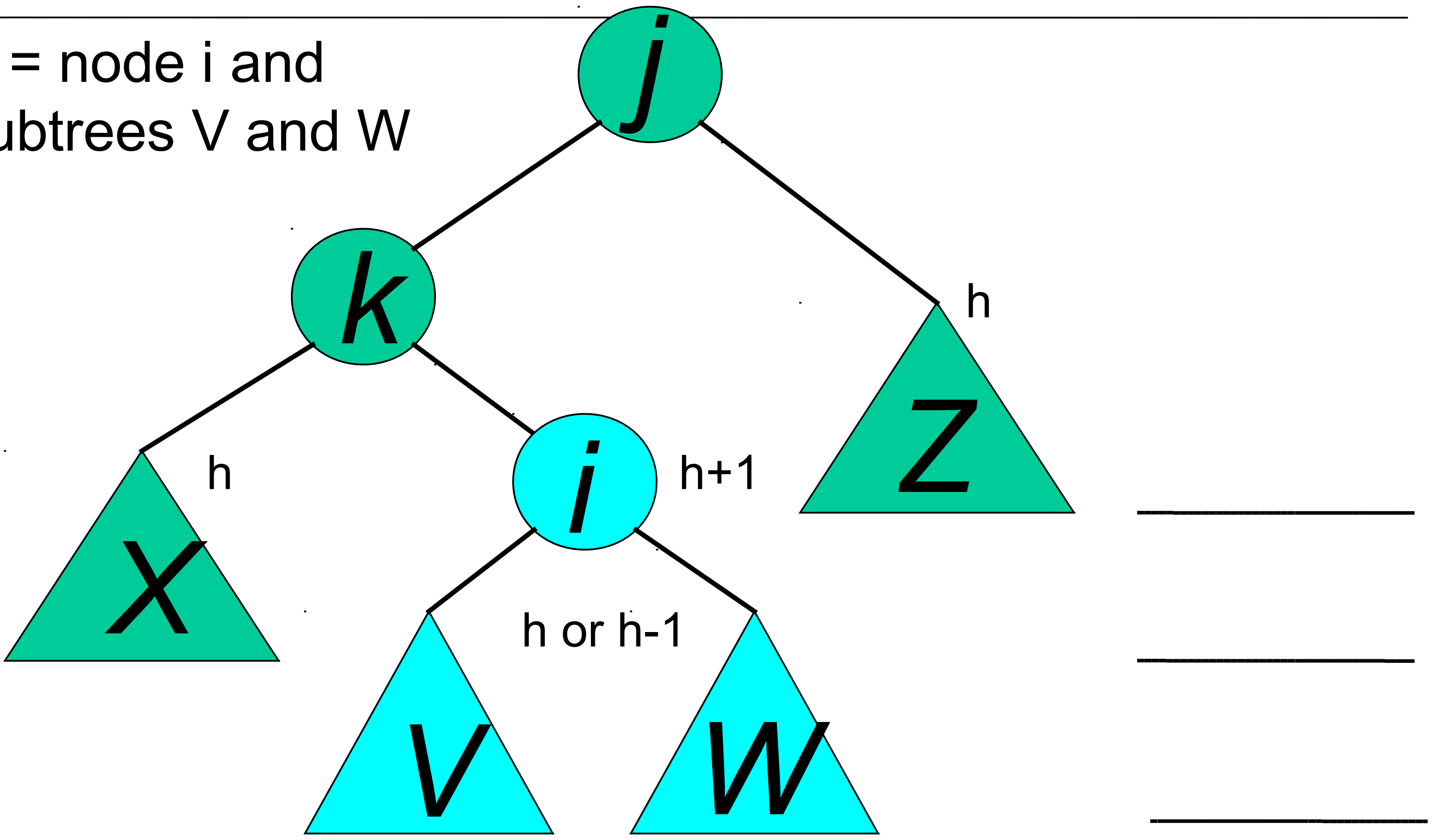
AVL Insertion: Inside Case

Consider the structure of subtree Y…

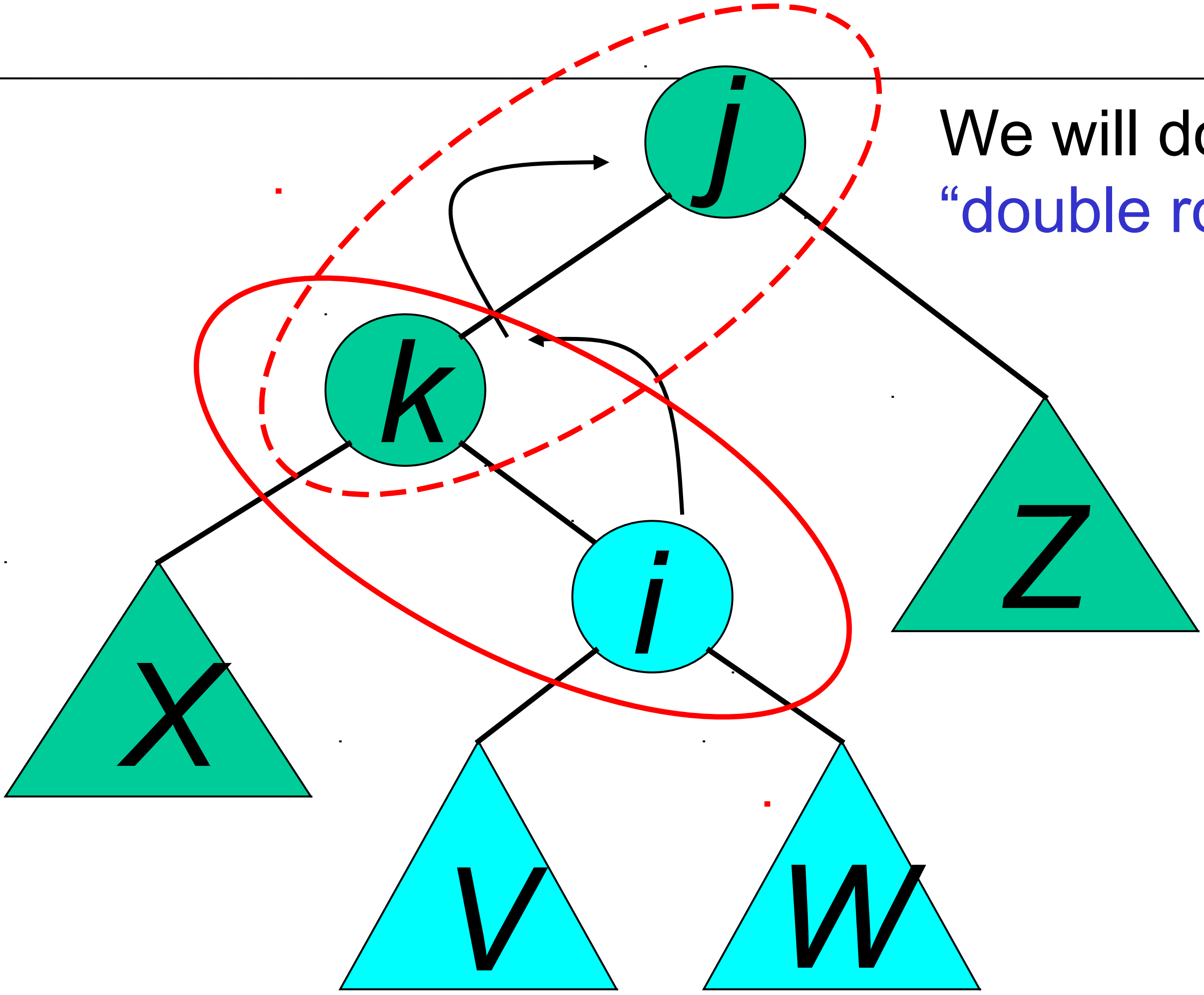


AVL Insertion: Inside Case

Y = node i and subtrees V and W

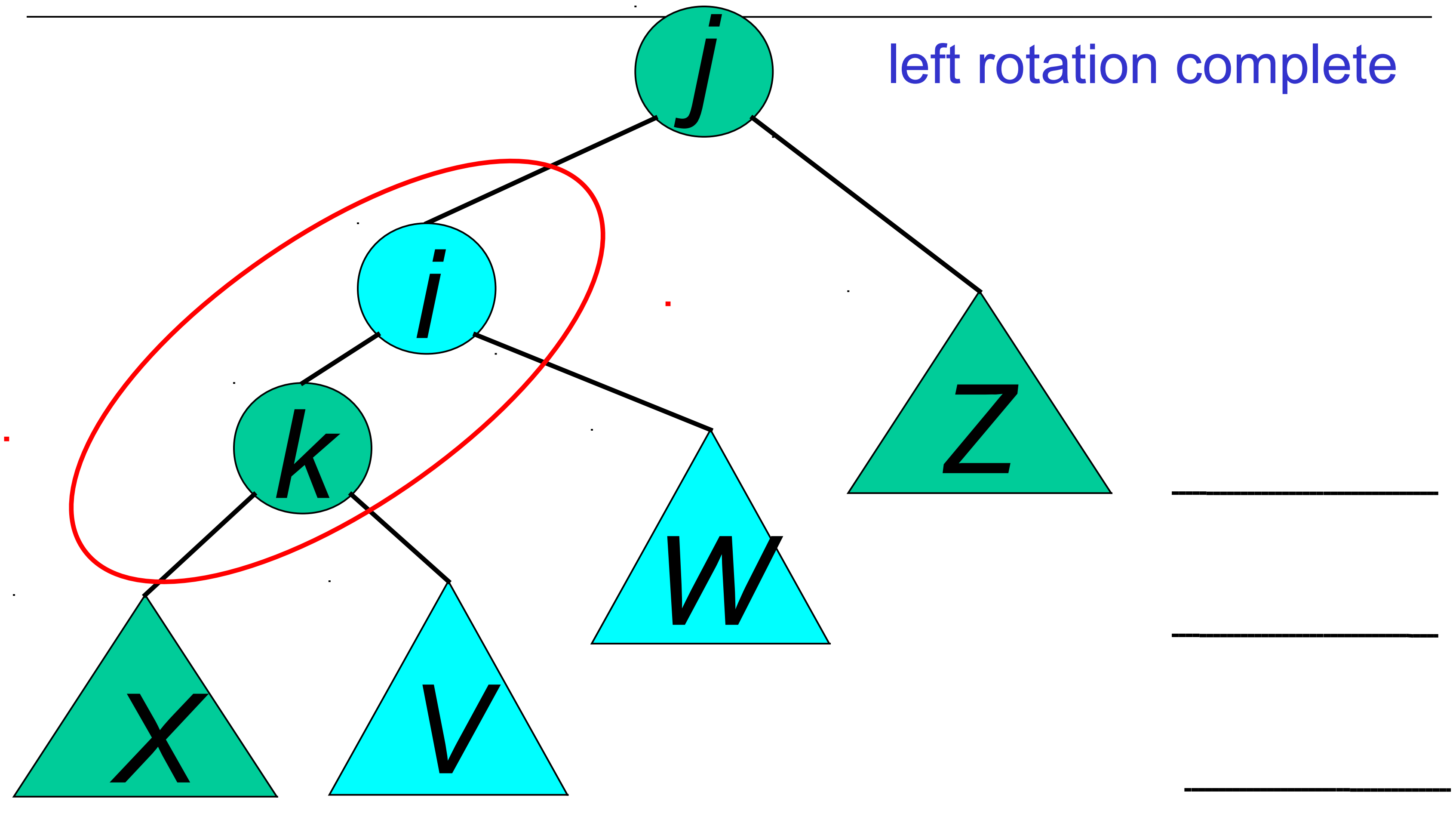


AVL Insertion: Inside Case

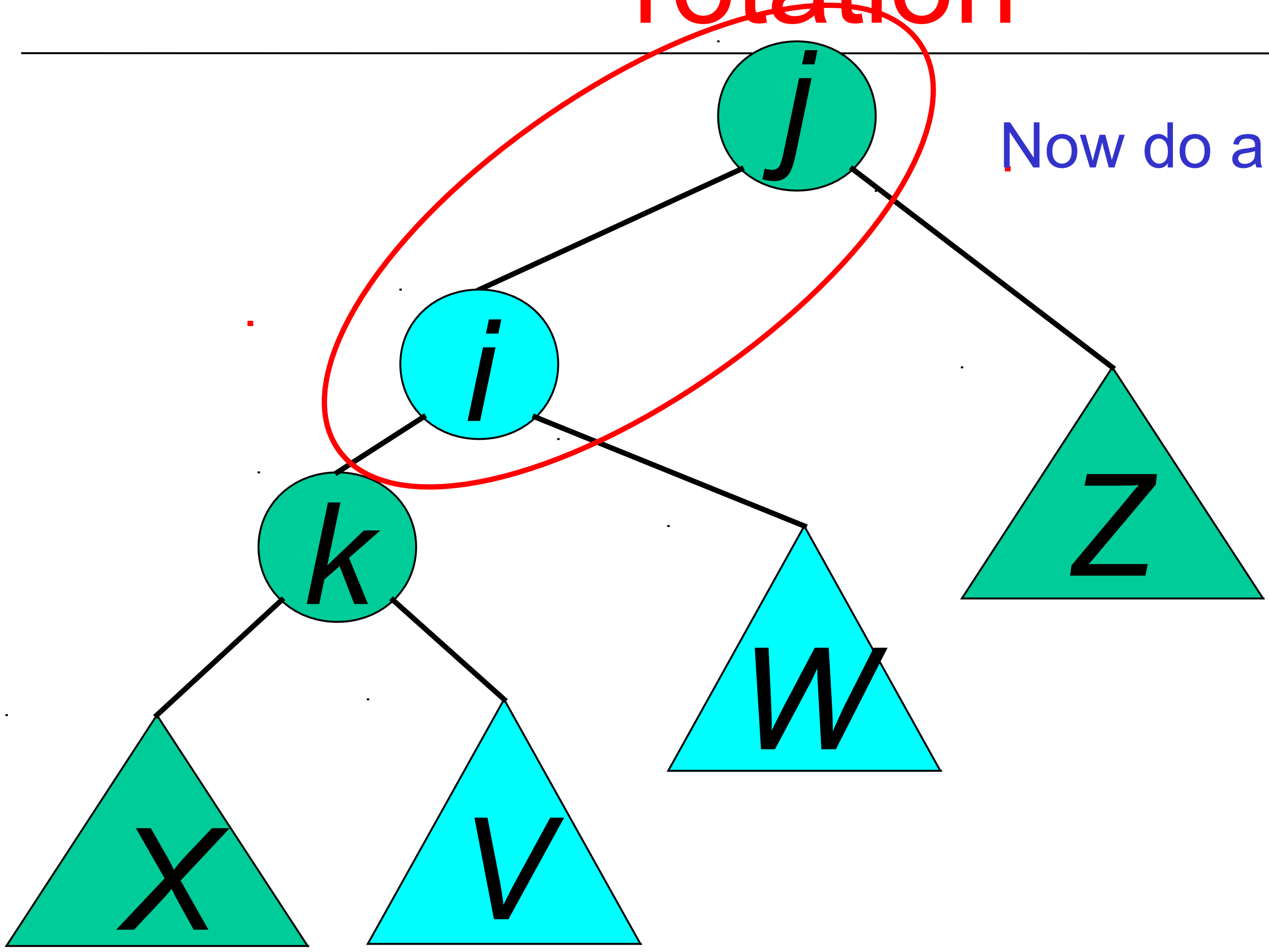


We will do a left-right "double rotation" . . .

Double rotation : first rotation



Double rotation : second rotation

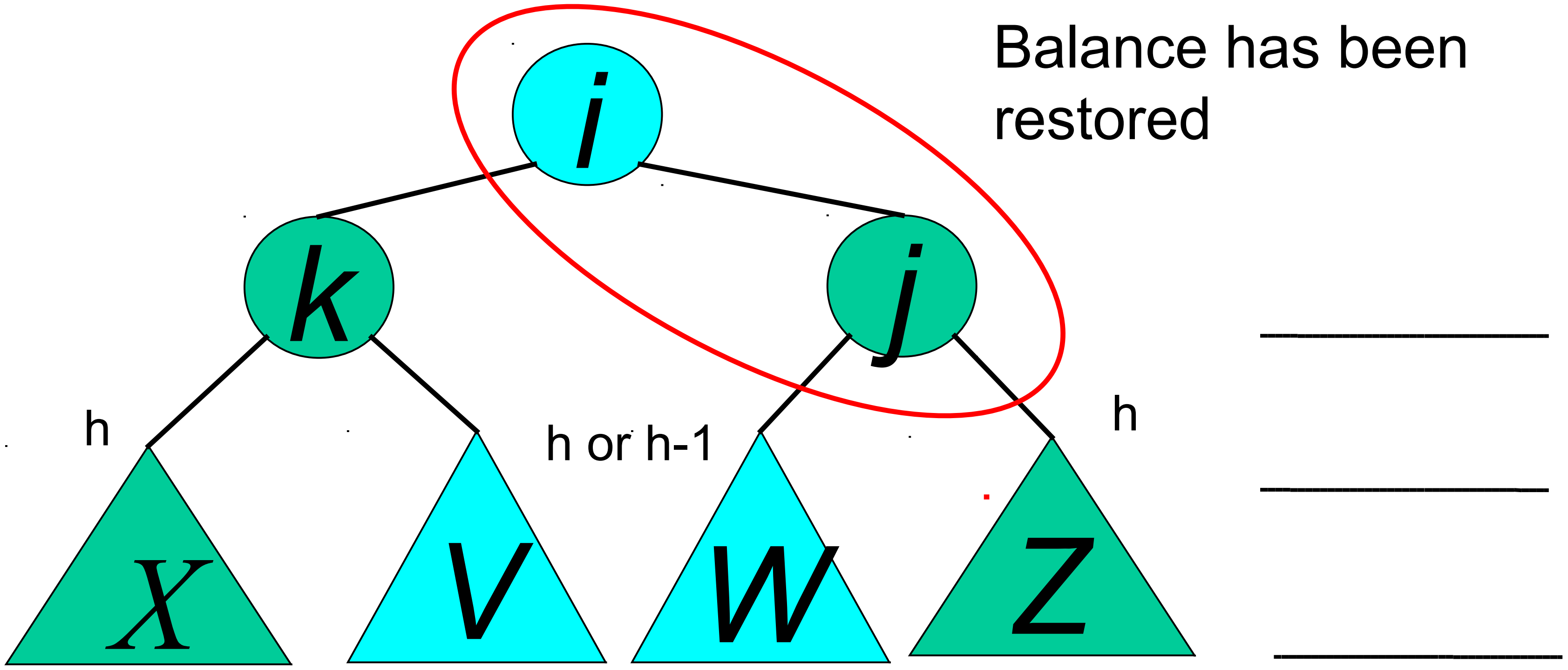


Now do a right rotation

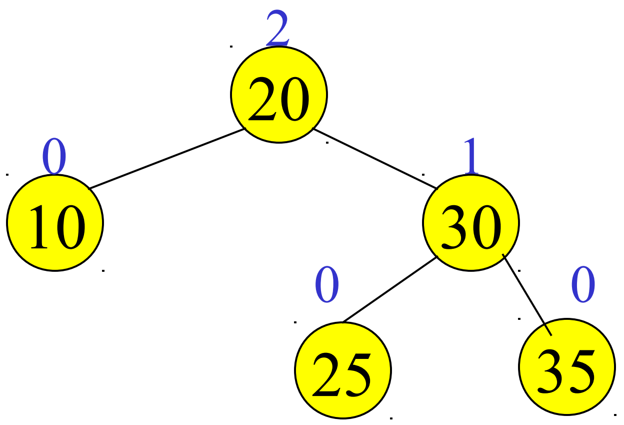
Double rotation : second rotation

right rotation complete

Balance has been restored

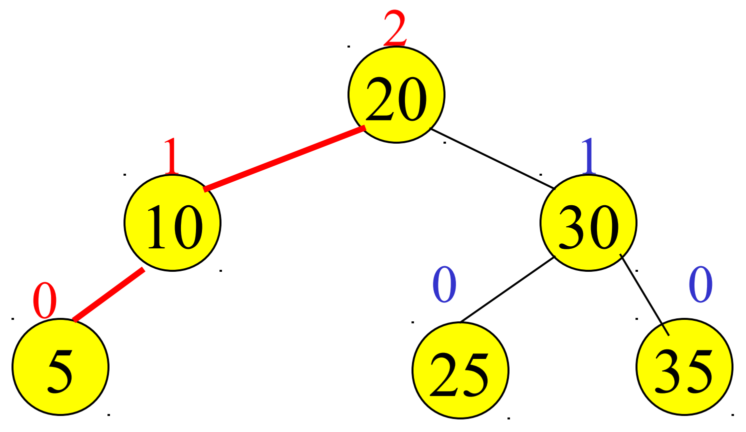


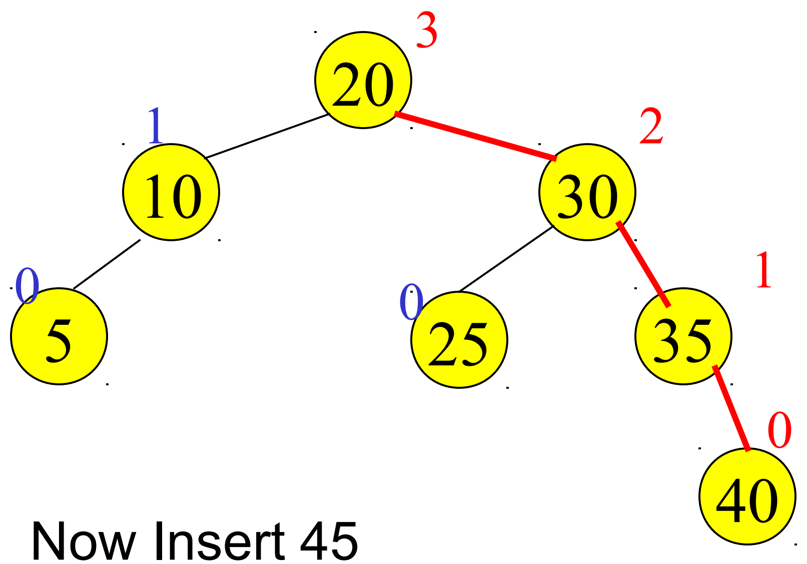
Example of Insertions in an AVL Tree



Insert 5, 40

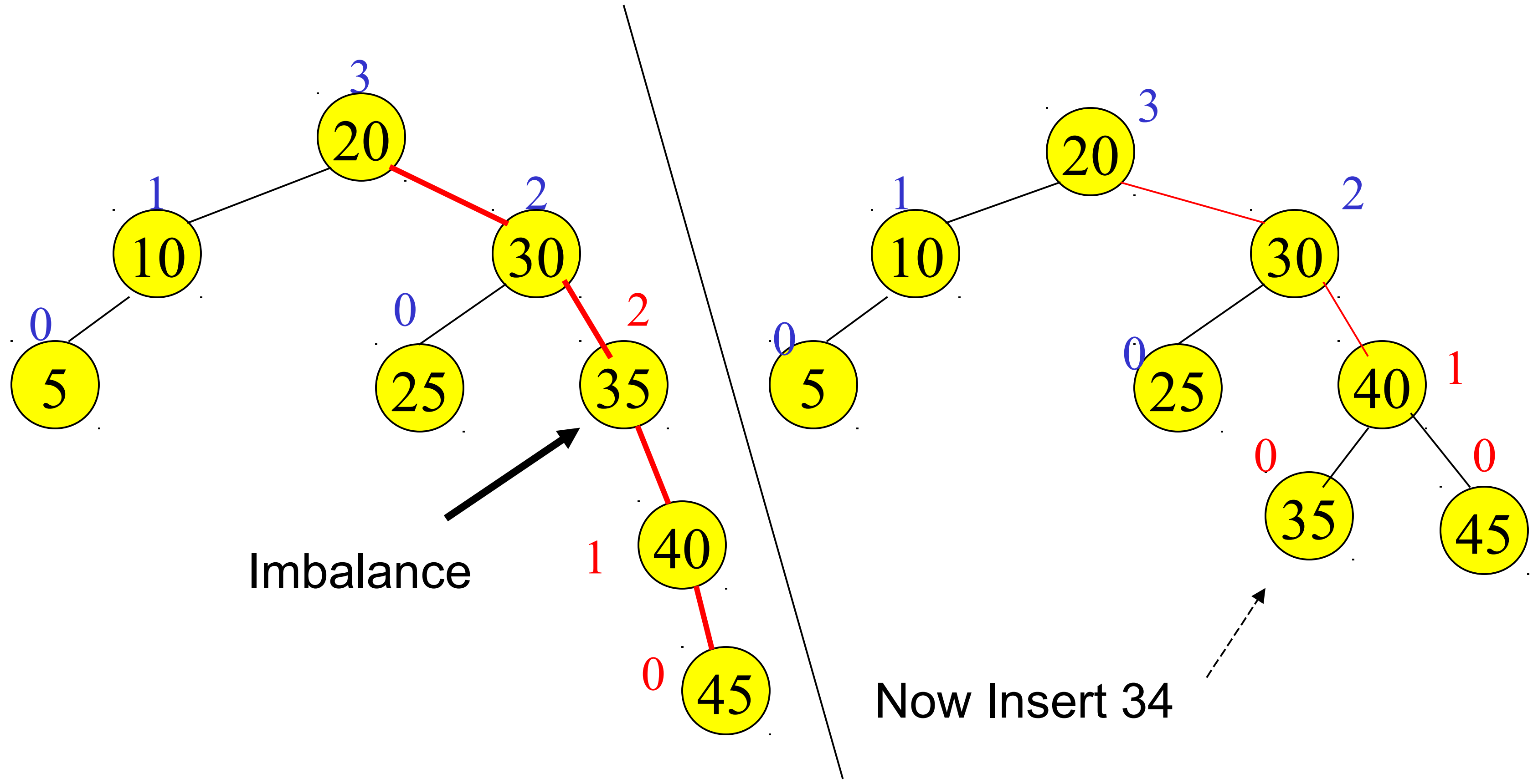
Example of Insertions in an AVL Tree



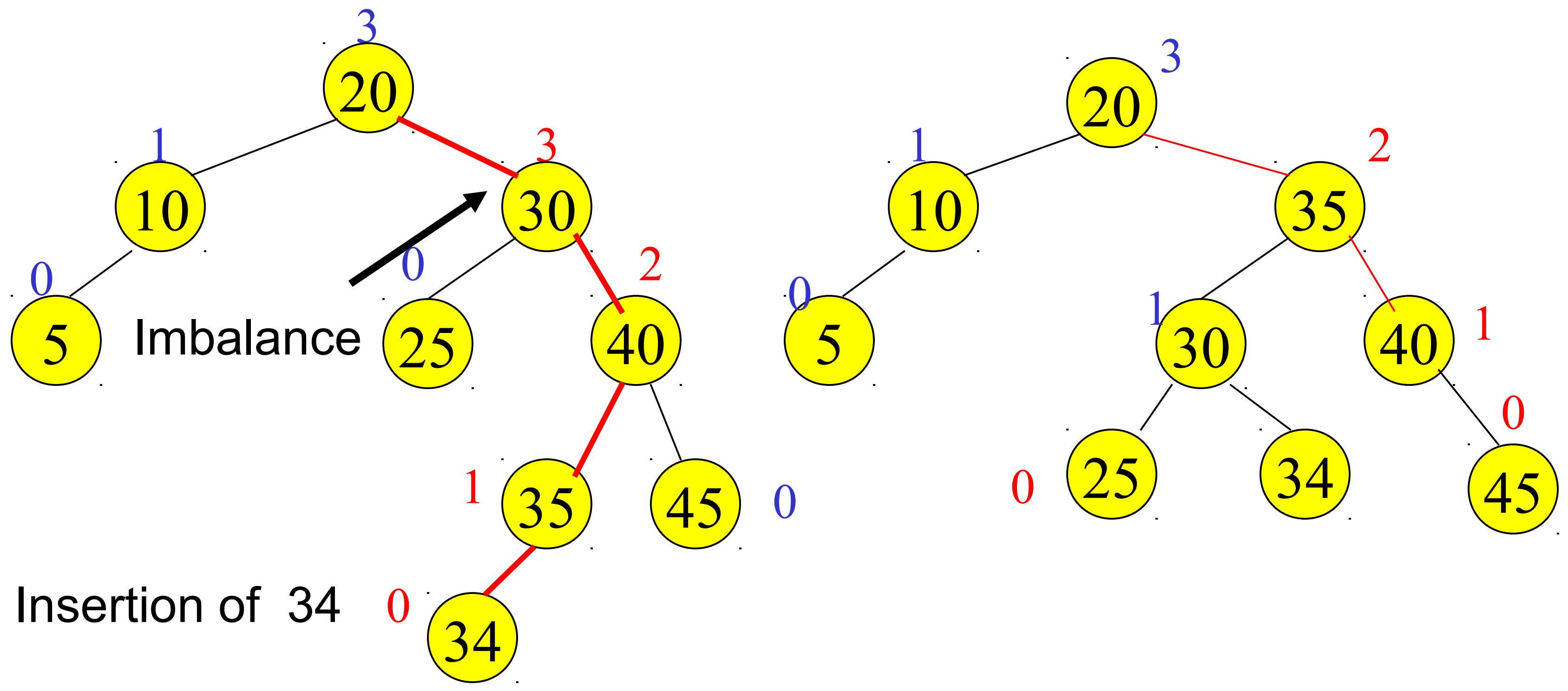


Now Insert 45

Single rotation (outside case)



Double rotation (inside case)

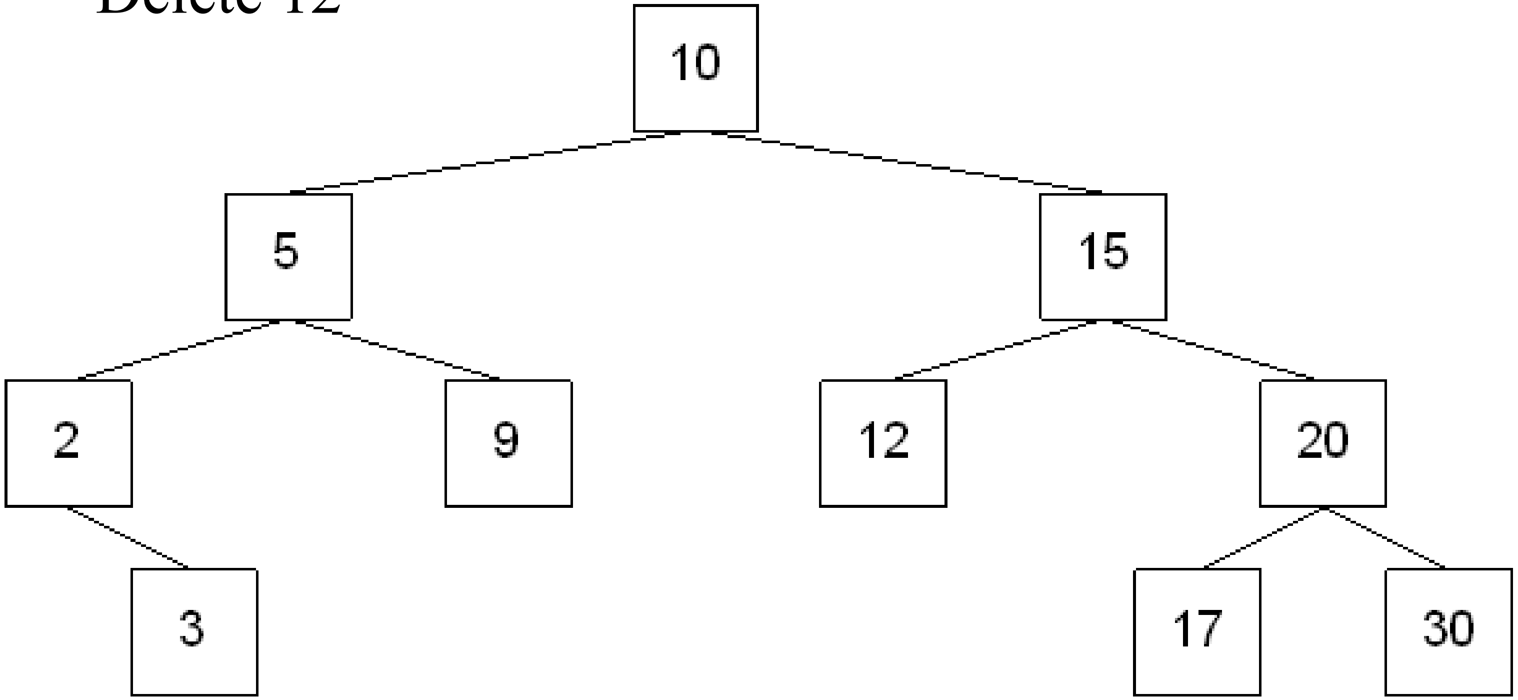


Deletion in BST

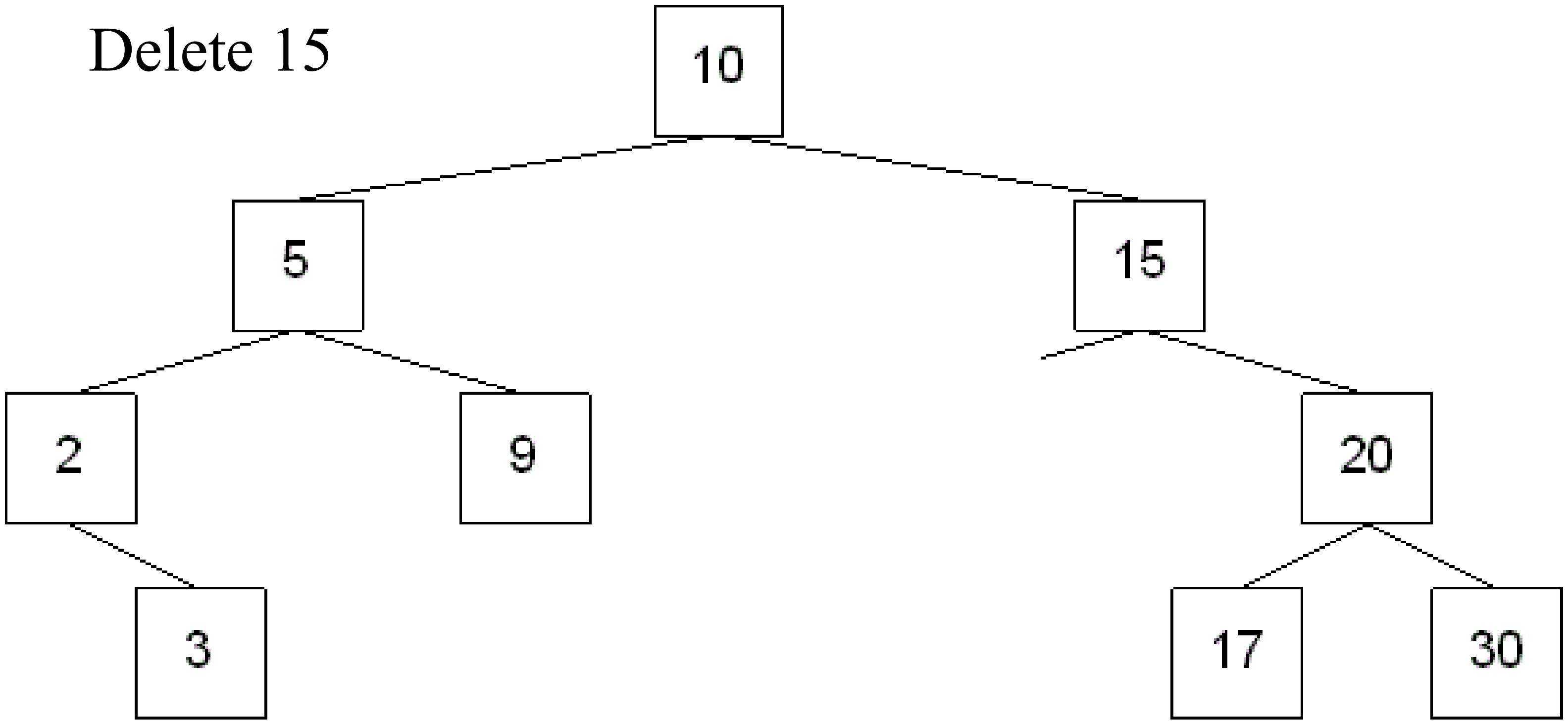
There are 3 cases.

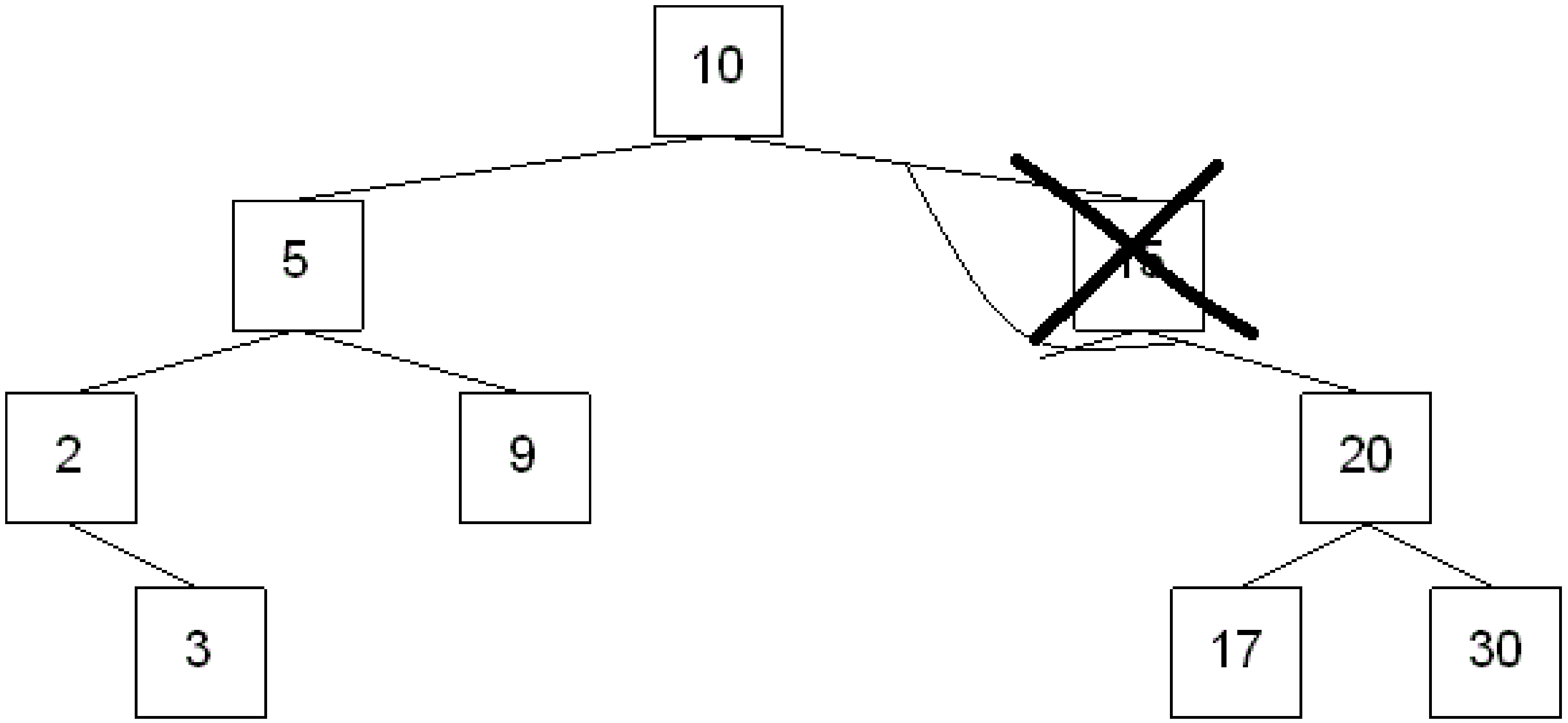
1. Node to be deleted is a leaf-simply remove the node.
2. When node has only one child-attach the child to the parent.
3. When the node to be deleted has 2 children-

Delete 12

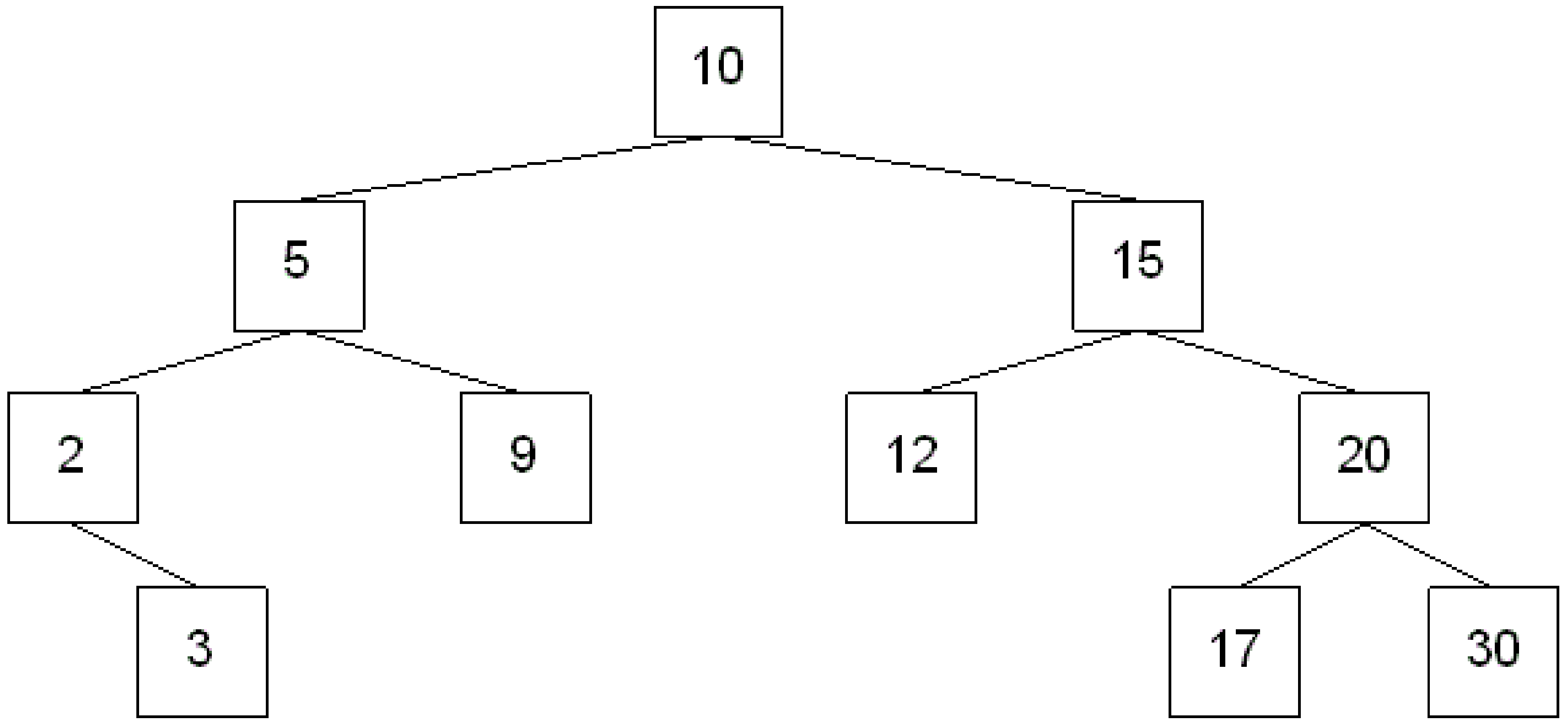


Delete 15

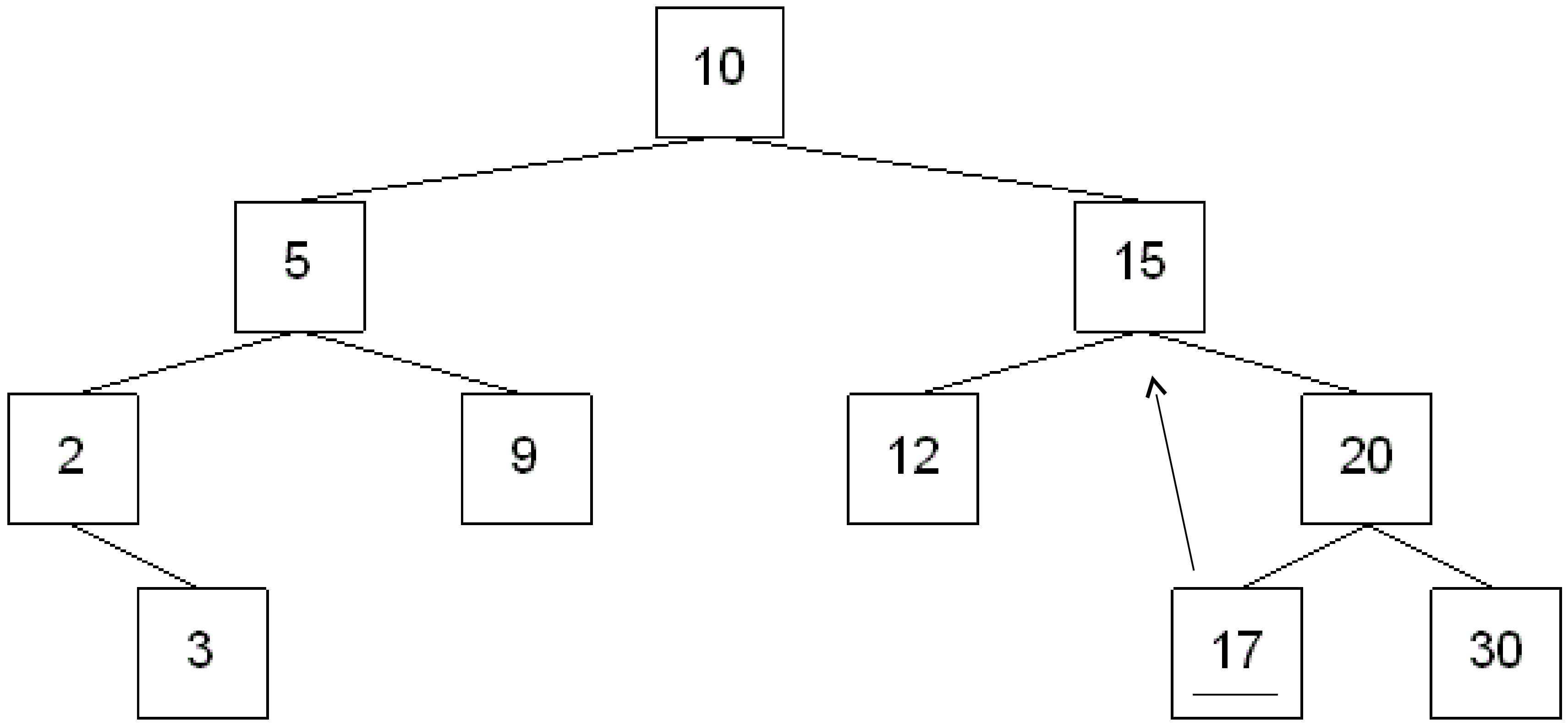


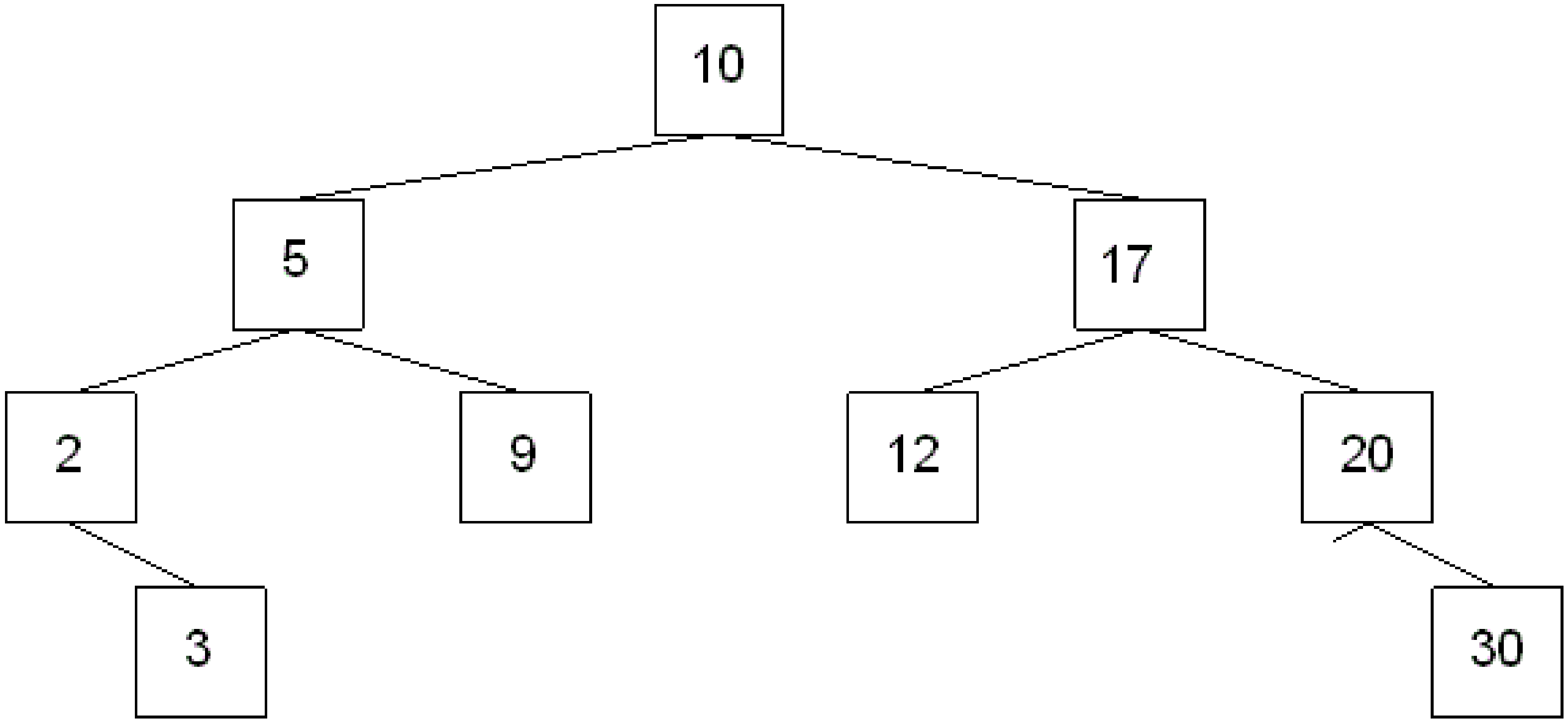

10
5
15
2
9
20
3
17
30

Delete 15



1. Copy the contents of inorder successor of the node and delete it.


10
5
15
2
9
12
20
3
17
30


10
5
17
2
9
12
20
3
30

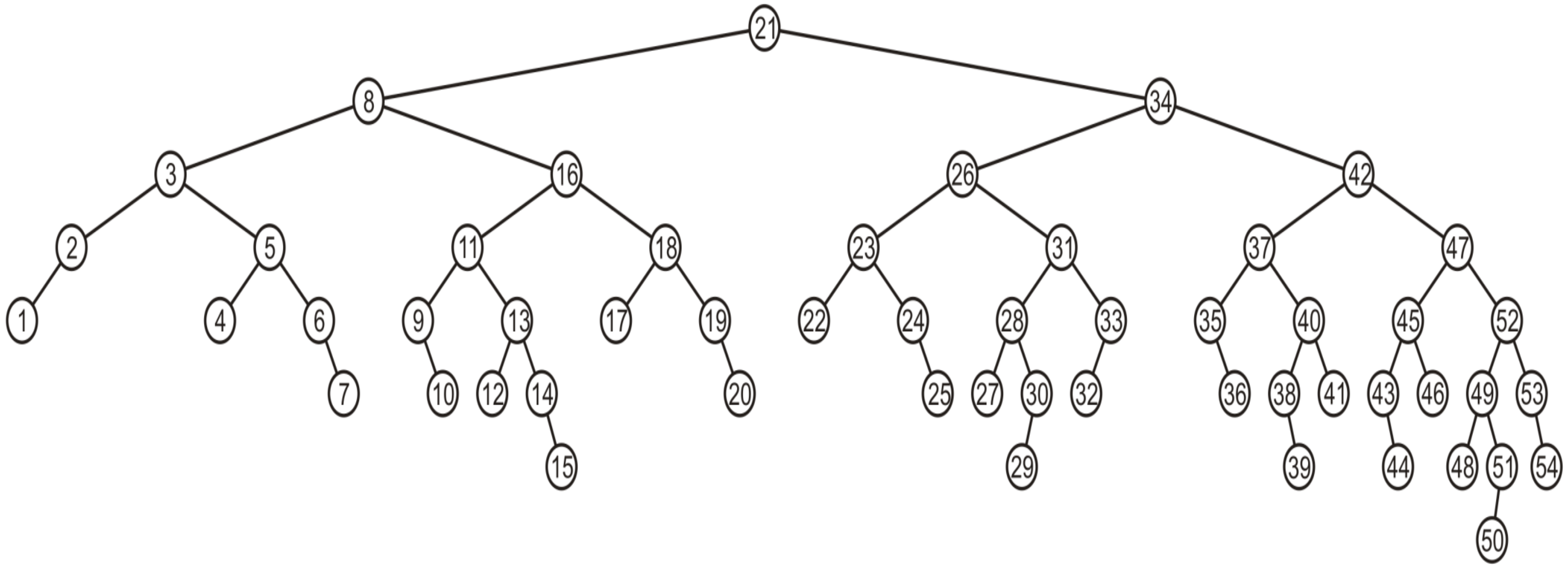
AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

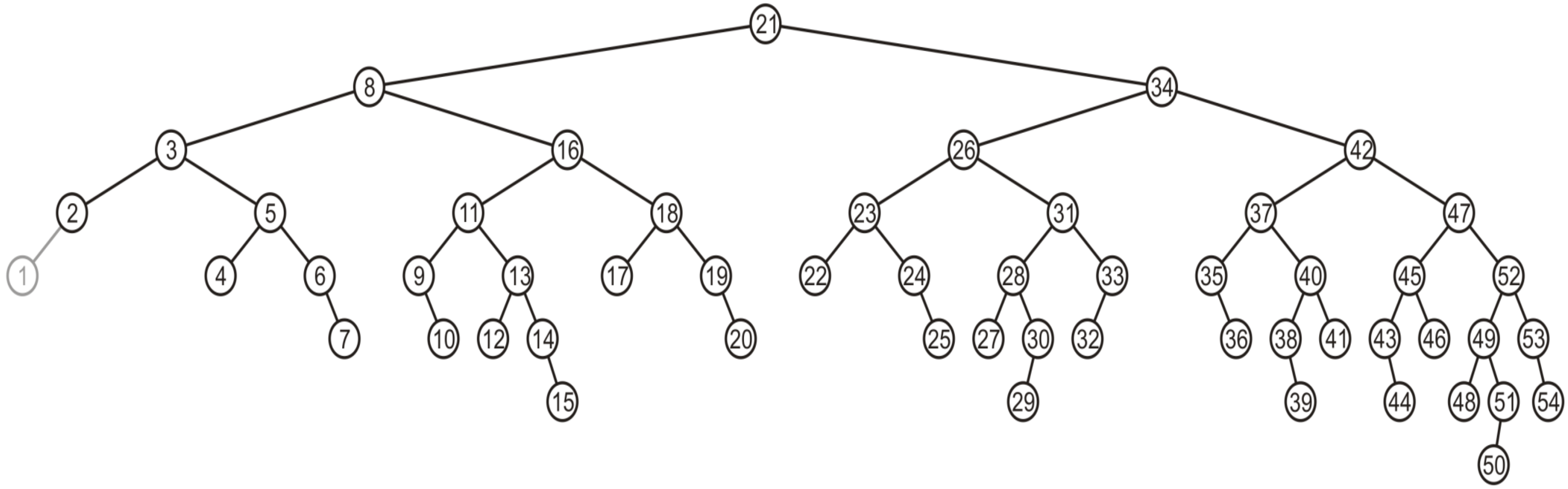
- Removing a node from an AVL tree may cause more than one AVL imbalance
  - Like insert, delete must check after it has been successfully called on a child to see if it caused an imbalance
  - Unfortunately, it may cause multiple imbalances that must be corrected
    - Insertions will only cause one imbalance that must be fixed

Deletion

Consider the following AVL tree

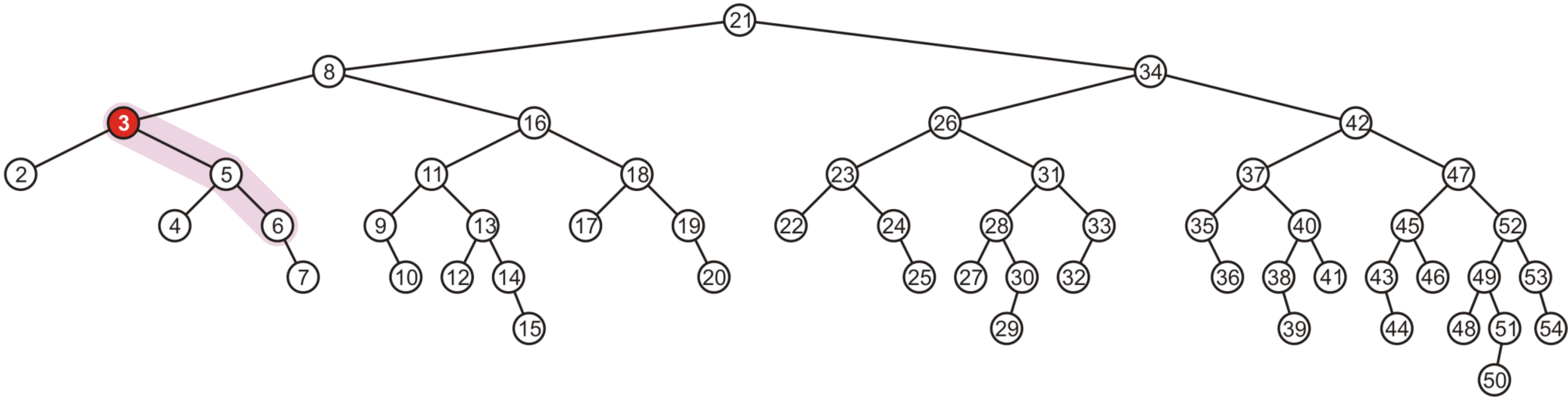


Suppose we delete the front node: 1



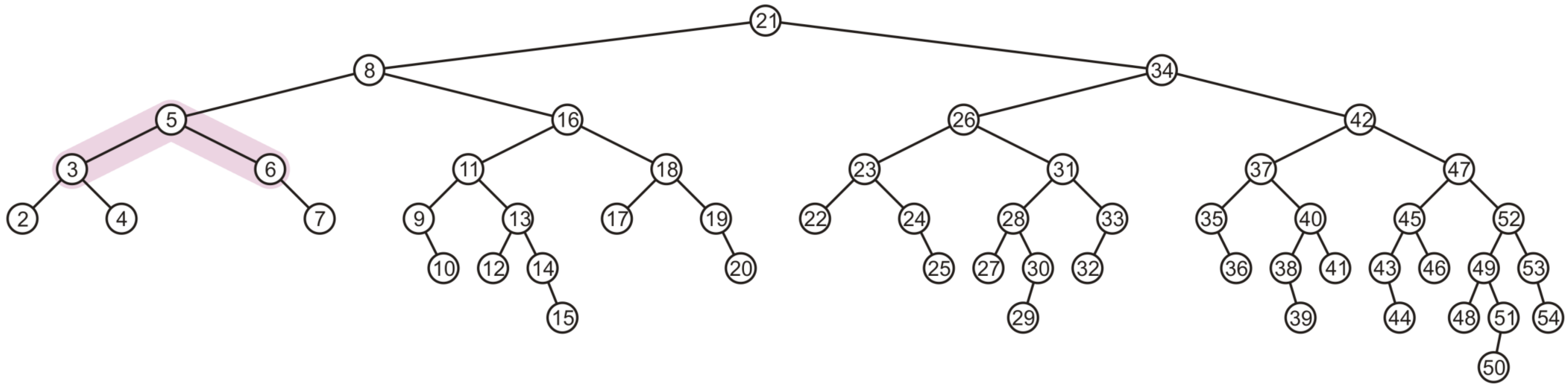
While its previous parent, 2, is not unbalanced, its grandparent 3 is

– The imbalance is in the right-right subtree



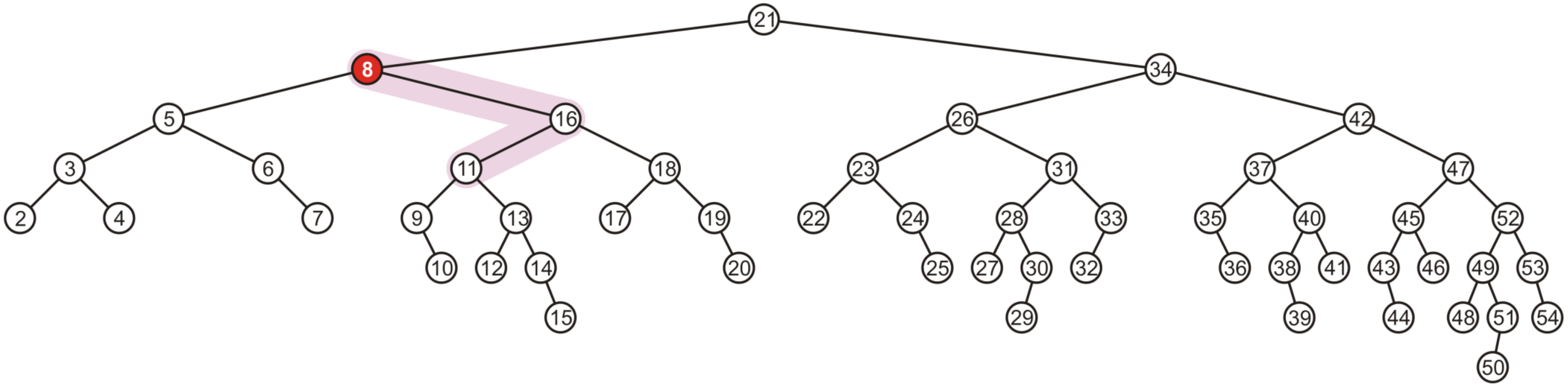
Erase

We can correct this with a simple balance



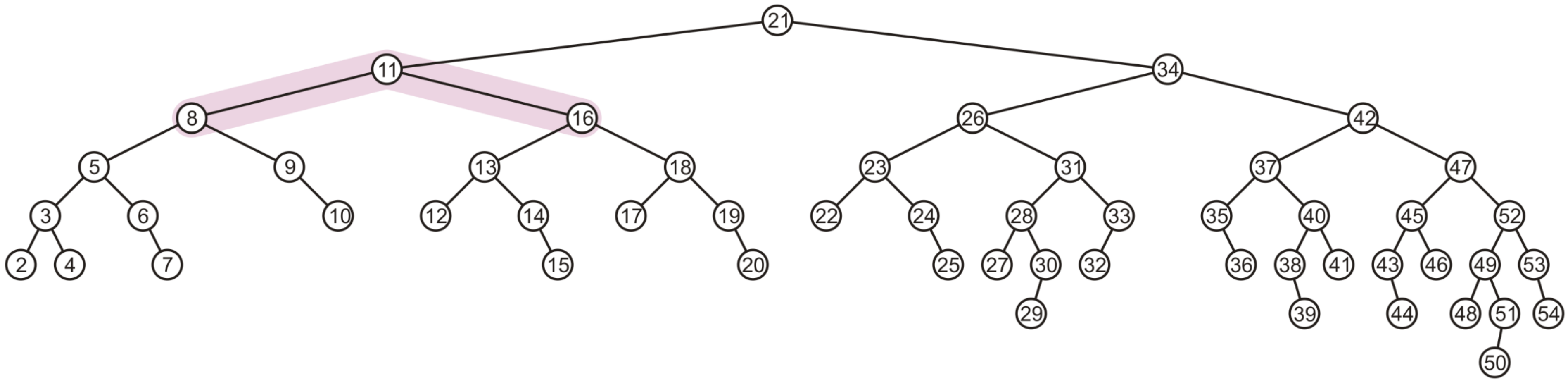
Erase

- Recursing to the root, however, 8 is also unbalanced
  - This is a right-left imbalance



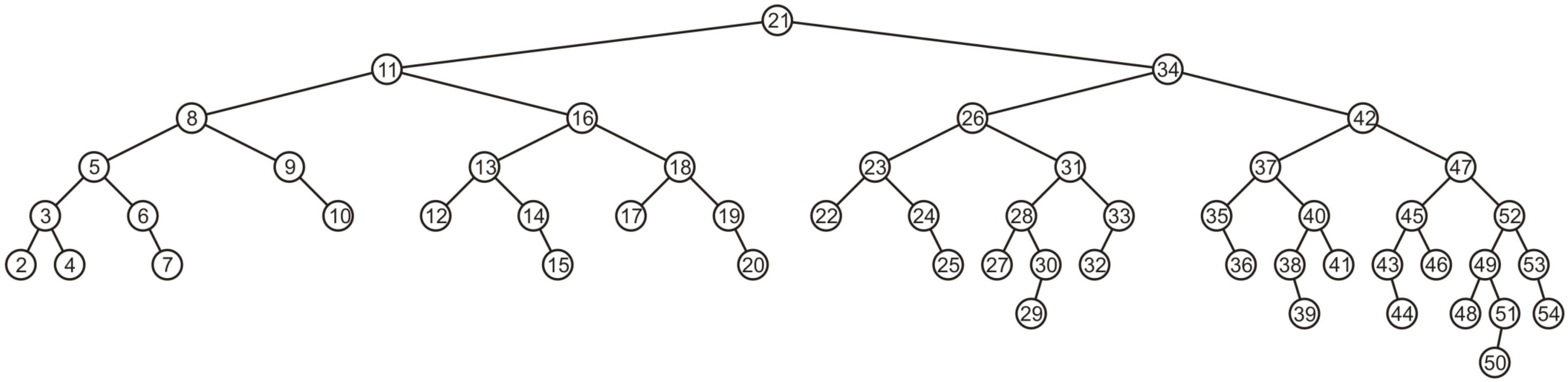
Erase

Promoting 11 to the root corrects the imbalance



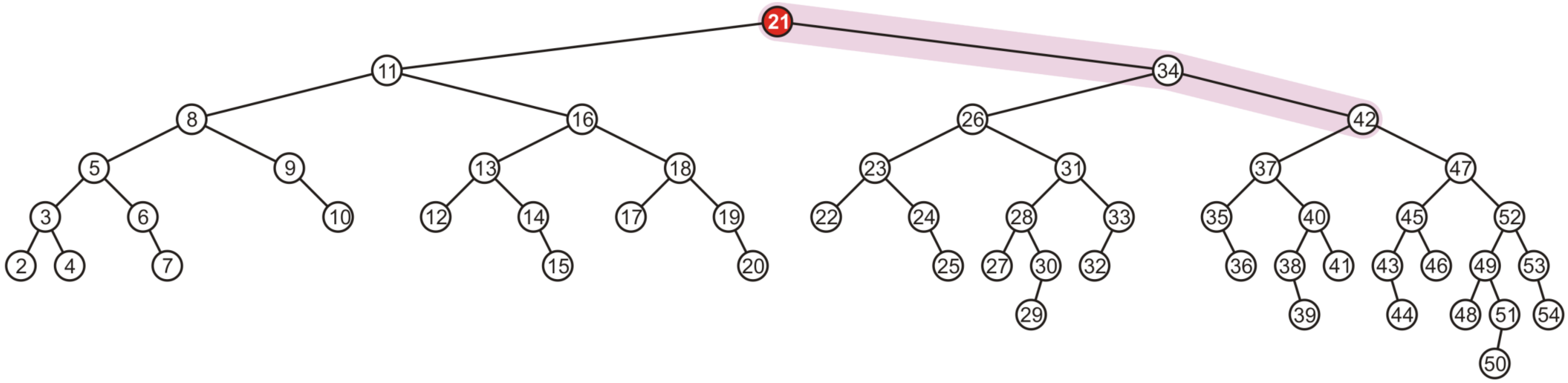
Erase

At this point, the node 11 is balanced



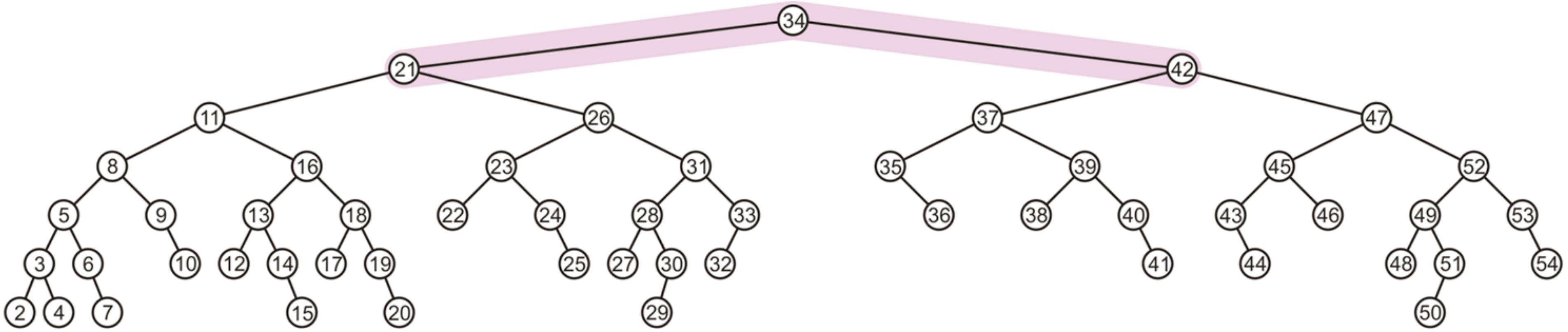
Erase

- Still, the root node is unbalanced
  - This is a right-right imbalance



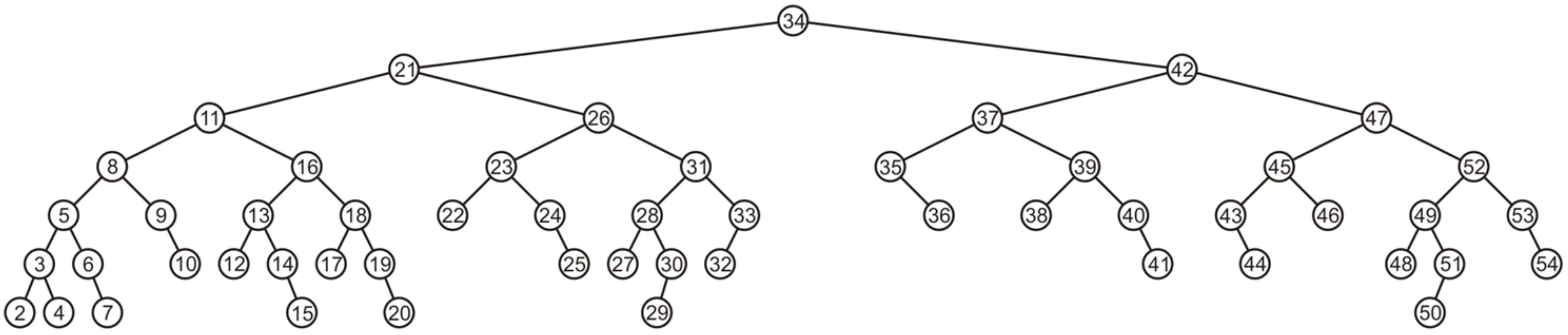
Erase

Again, a simple balance fixes the imbalance



Erase

The resulting tree is now AVL balanced


34
21
42
11
26
37
47
8
16
23
31
35
39
45
52
5
9
13
18
22
24
28
33
36
38
40
43
46
49
53
3
6
10
12
14
17
19
25
27
30
32
41
44
48
51
54
2
4
7
15
20
29
50