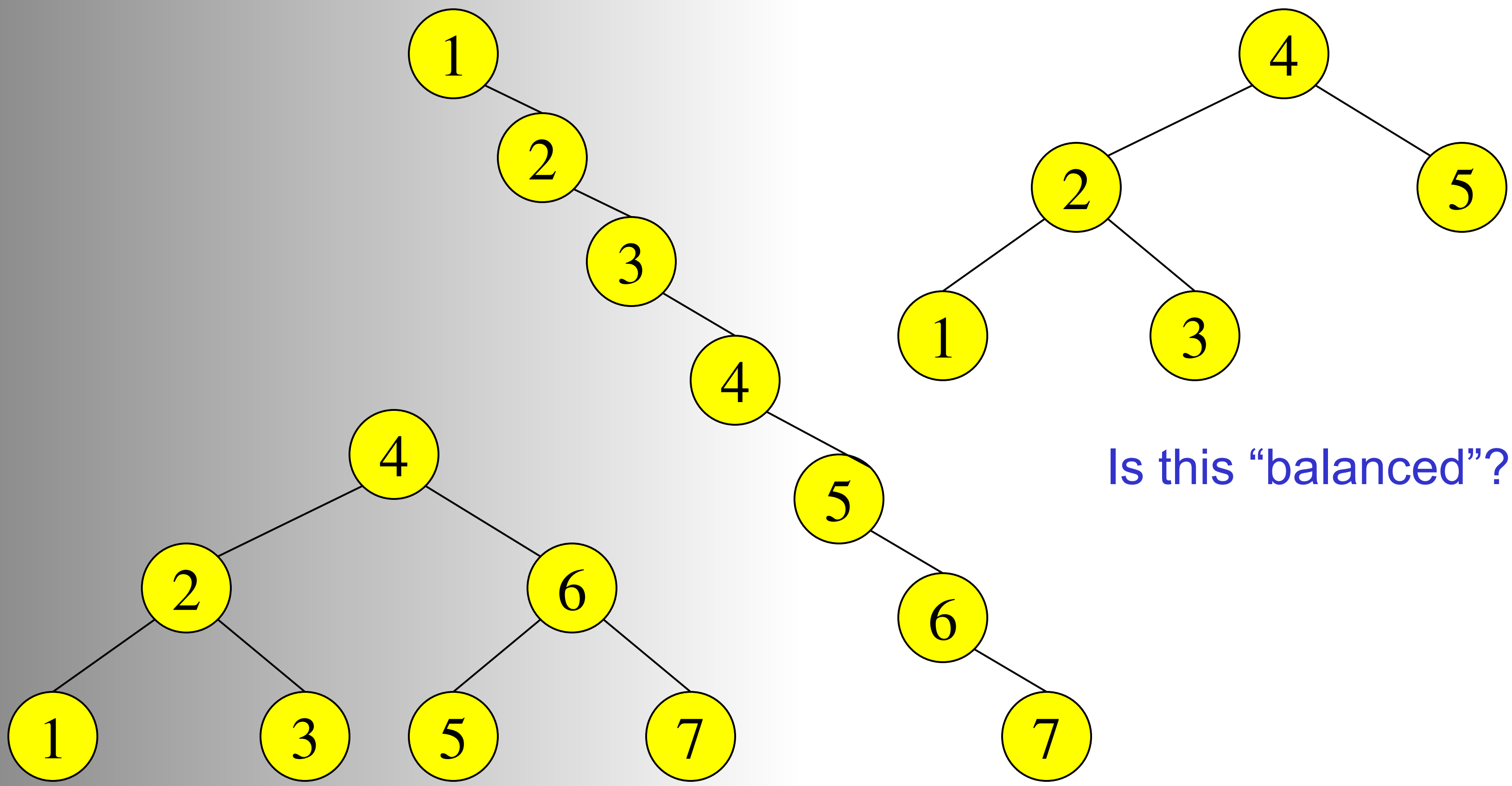
Data Structures and Algorithms

AVL Search Tree

By
Ravi Kant Sahu
Asst. Professor,
Lovely Professional University, Punjab

Balanced and Unbalanced BST



Is this “balanced”?

AVL Search Tree

- Skewed Binary Search Tree:

  Worst case time complexity is O(n).

- Adelson-Velskii and Landis introduced height balanced tree in 1962.

- Balance factor of a node =

  height(left sub-tree) - height(right sub-tree)

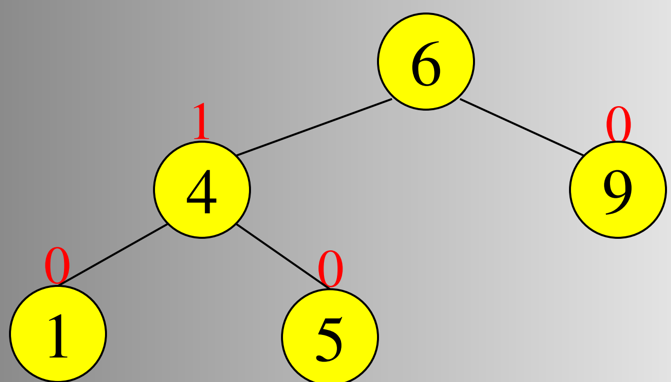
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees.
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right sub-tree can differ by no more than 1
  - Balance Factor of a node is -1, 0 or 1 in AVL.

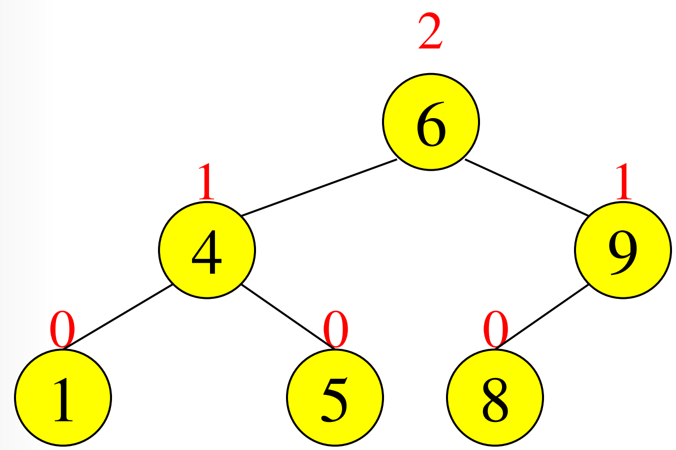
Node Heights

Tree A (AVL)

height=2   BF=1-0=1



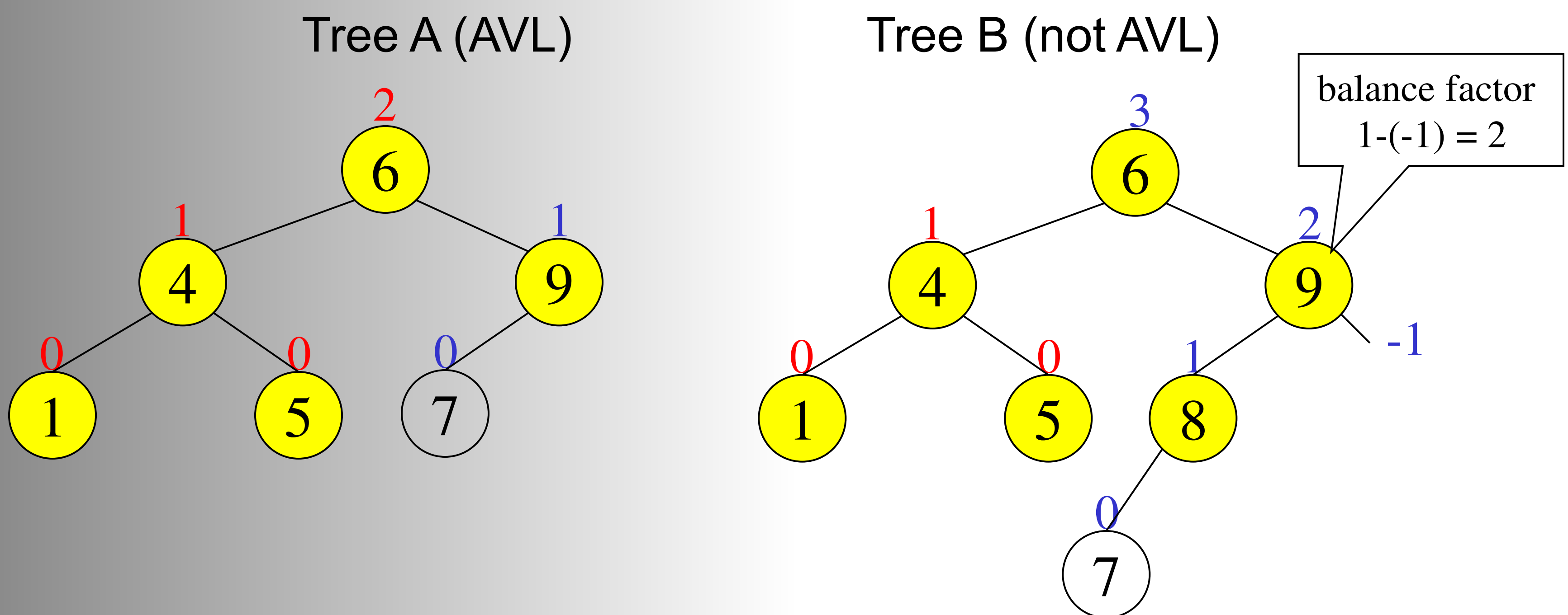
Tree B (AVL)



height of node = h

balance factor = h_{left}-h_{right}

Node Heights after Insert 7



Basic Concepts

LR and RL Rotation

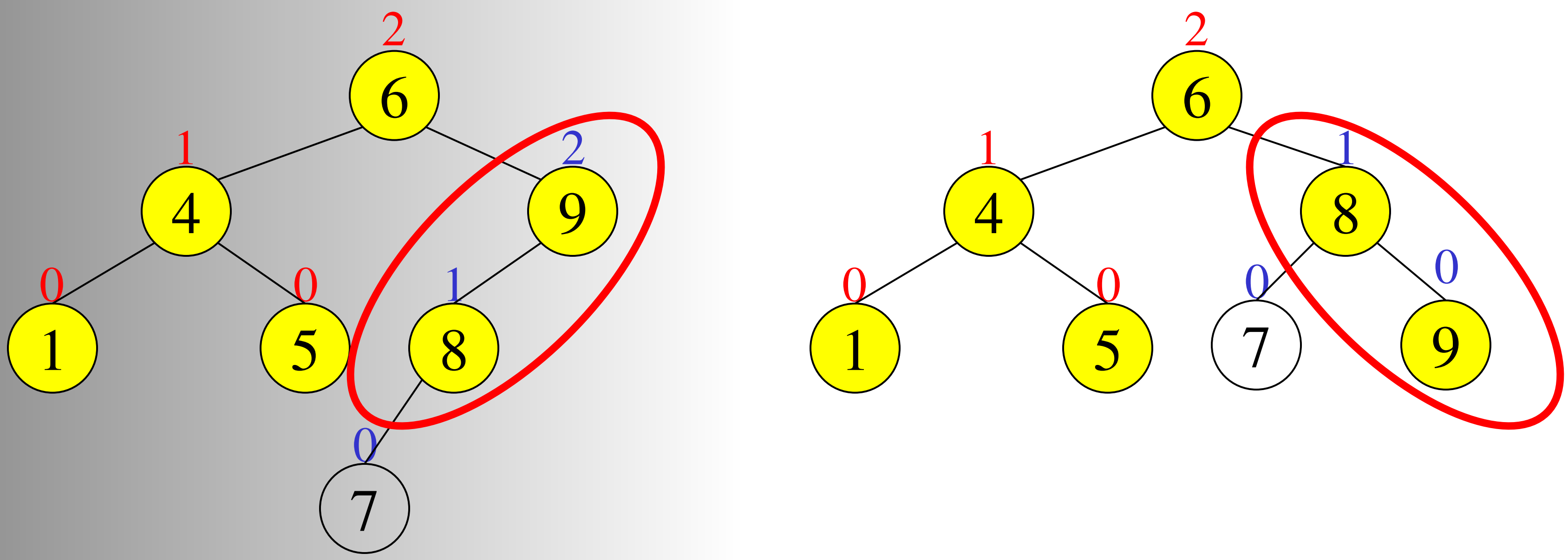
- Find Out the first Node from the bottom which has BF other than 1, 0, -1, call it A and its descendent towards the newly inserted node as B.

- LR Rotation: If newly inserted node is in the right subtree of left subtree of A.
    - Apply RR rotation on B
    - Then Apply LL rotation on A
- RL Rotation: If newly inserted node is in the left subtree of right subtree of A.
    - Apply LL rotation on B
    - Then Apply RR rotation on A

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.

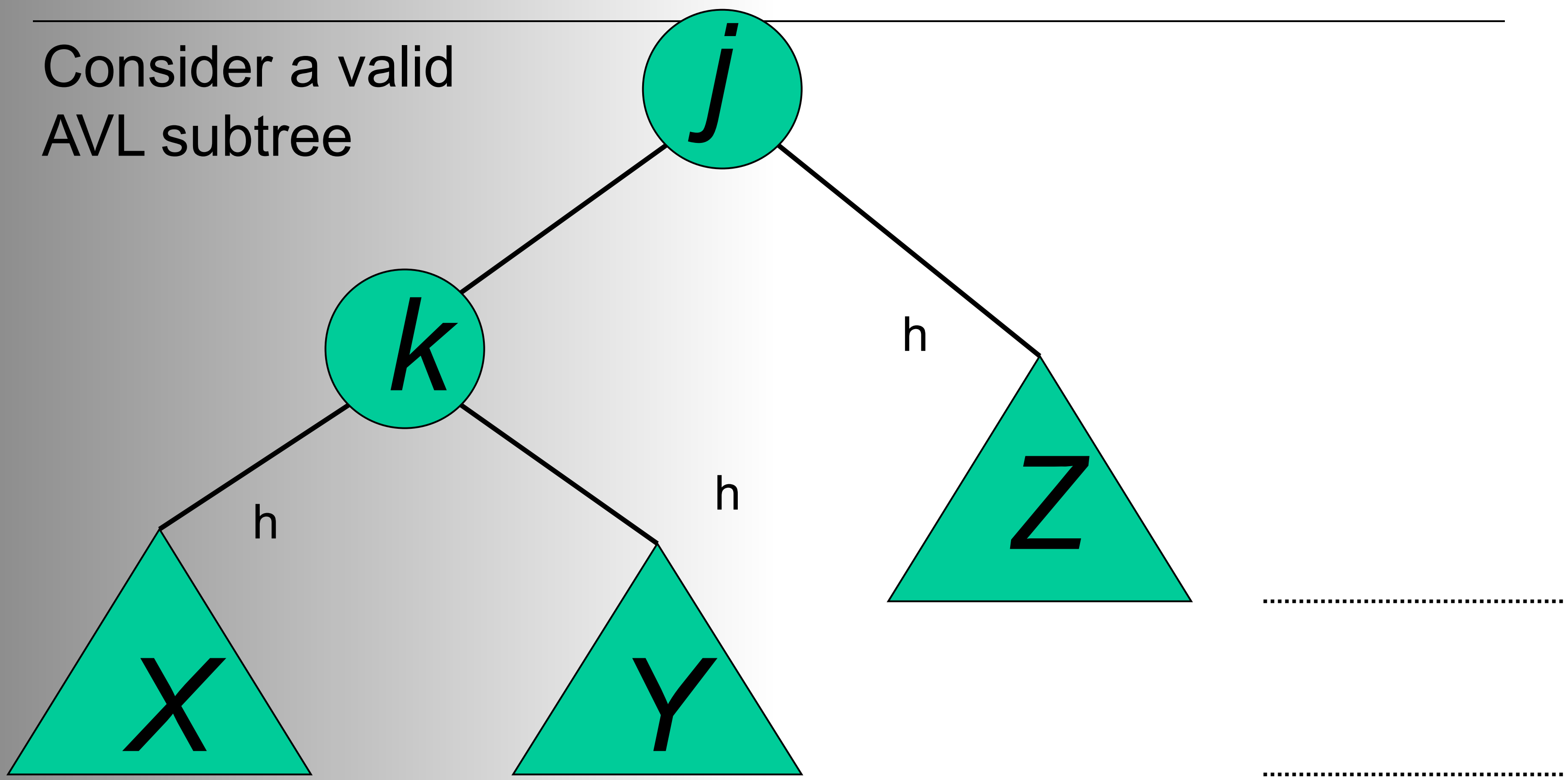
Inside Cases (require double rotation) :

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

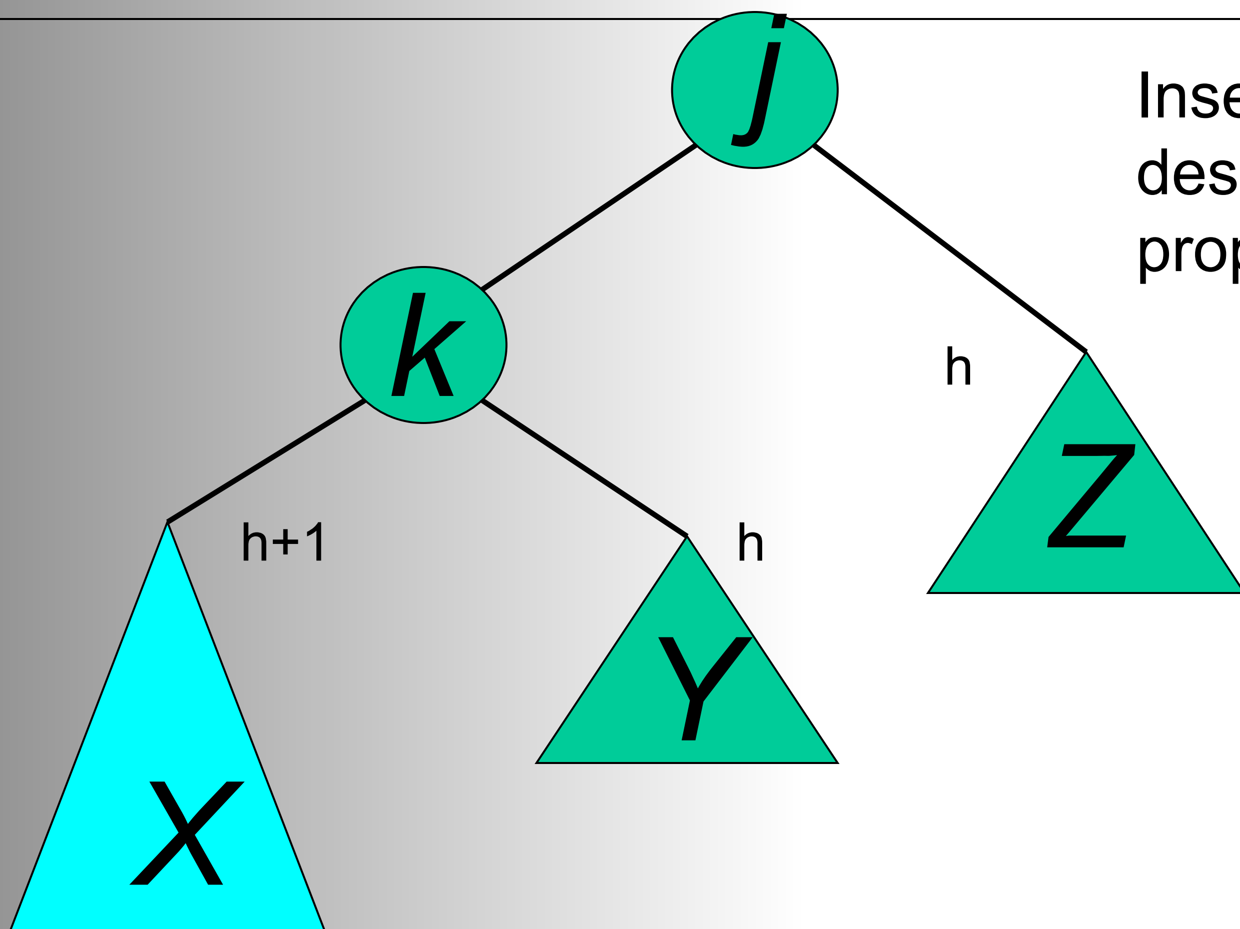
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid
AVL subtree



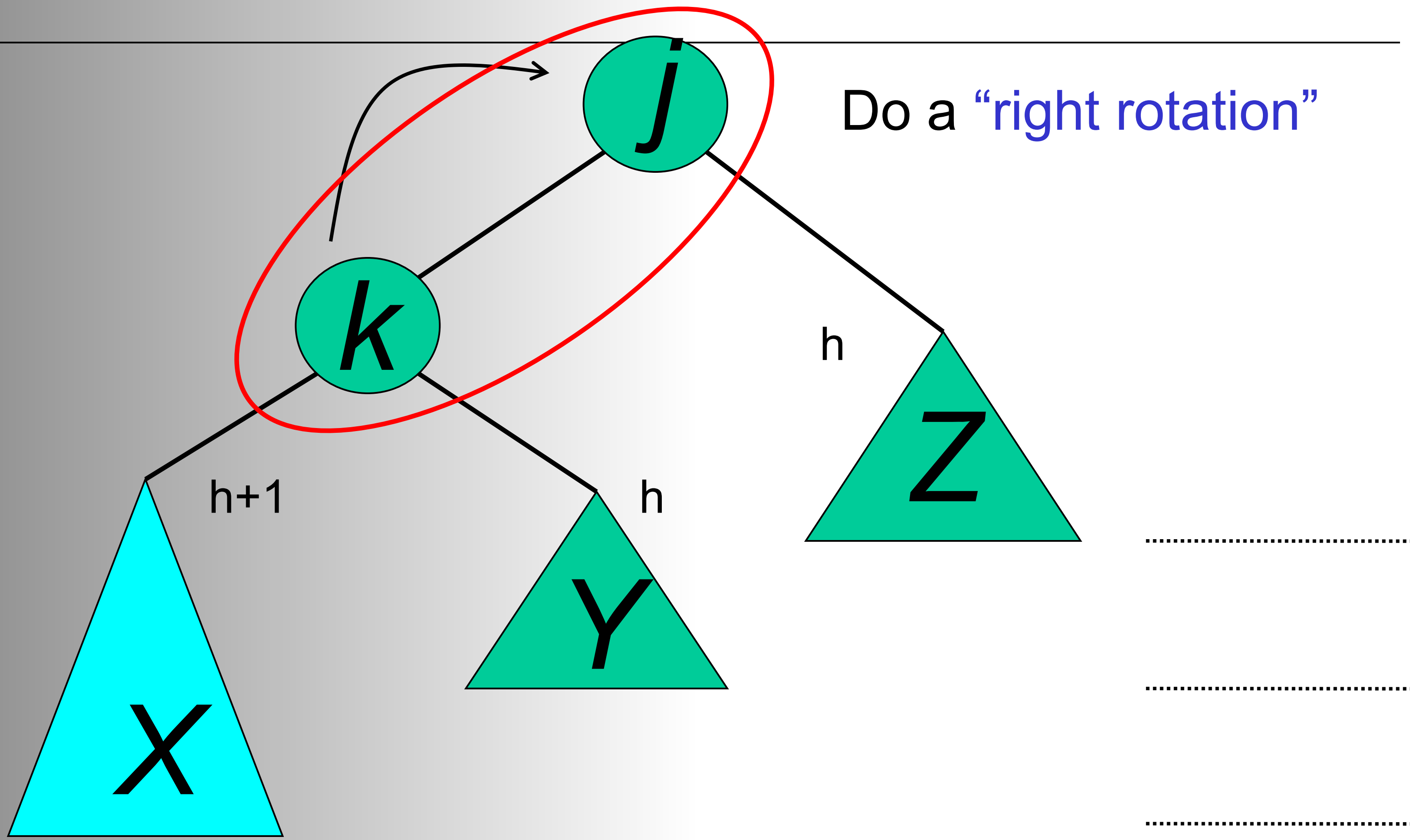
AVL Insertion: Outside Case



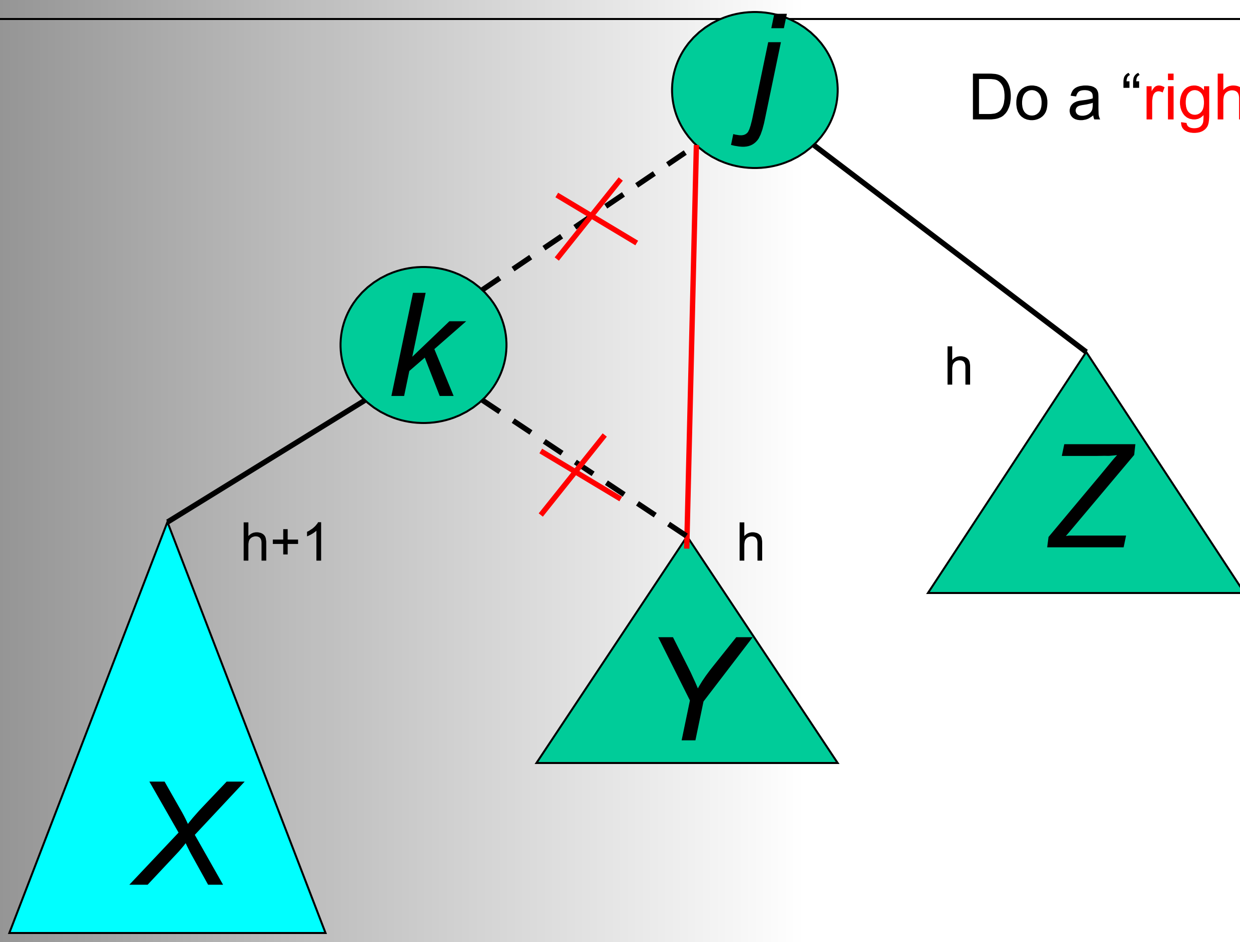
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

Do a "right rotation"



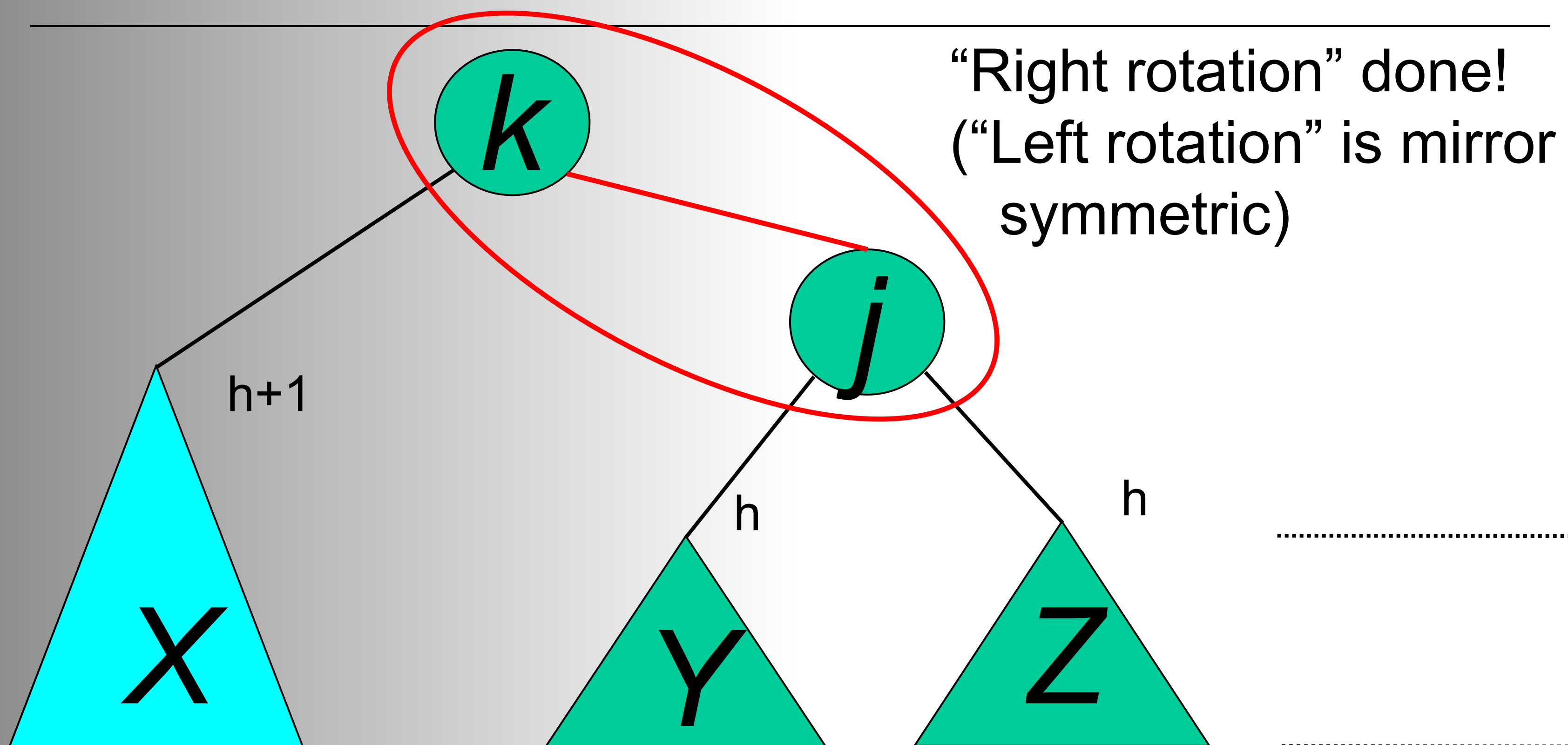
Single right rotation



Do a "right rotation"

Outside Case Completed

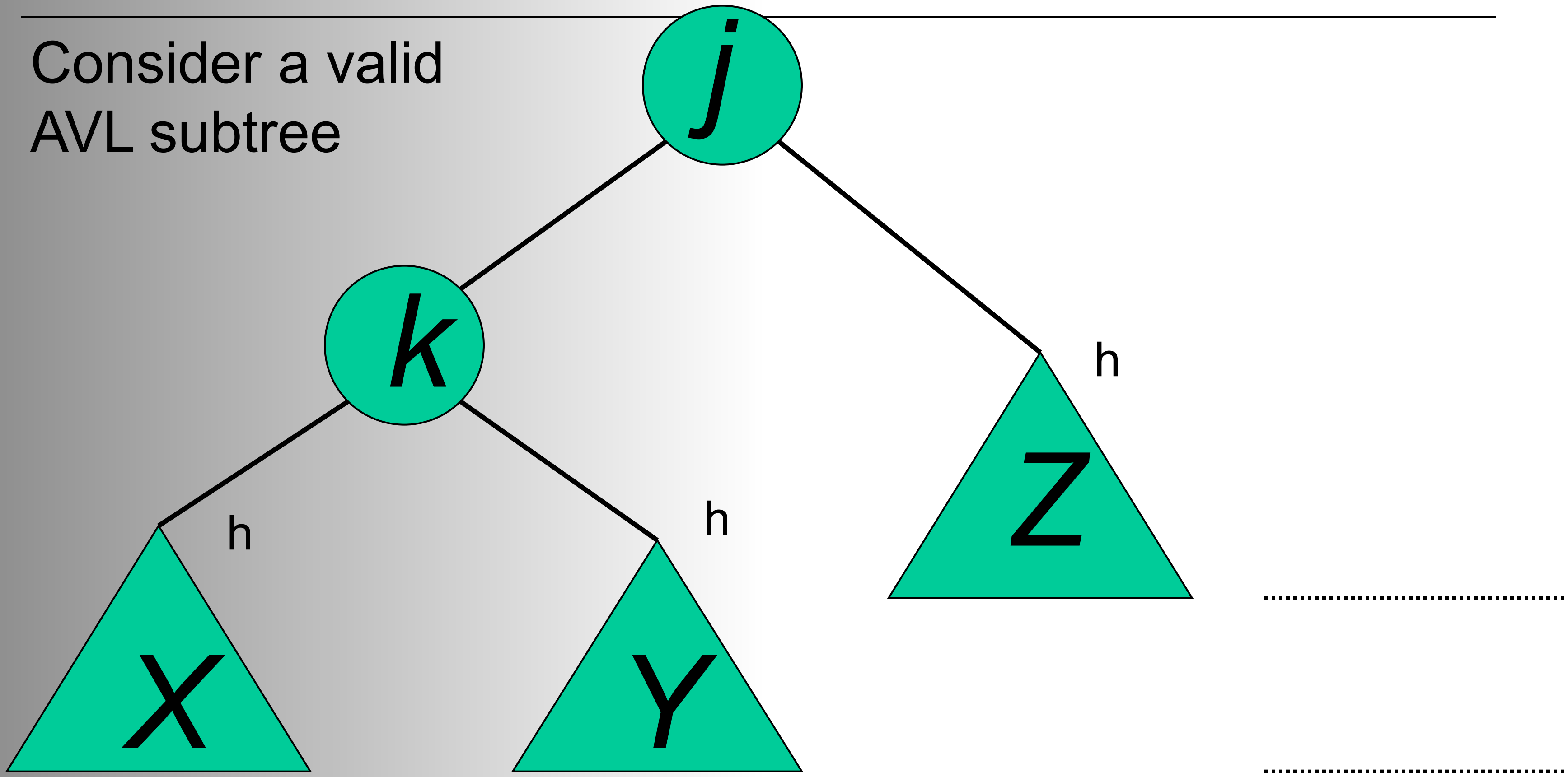
“Right rotation” done!
(“Left rotation” is mirror symmetric)



AVL property has been restored!

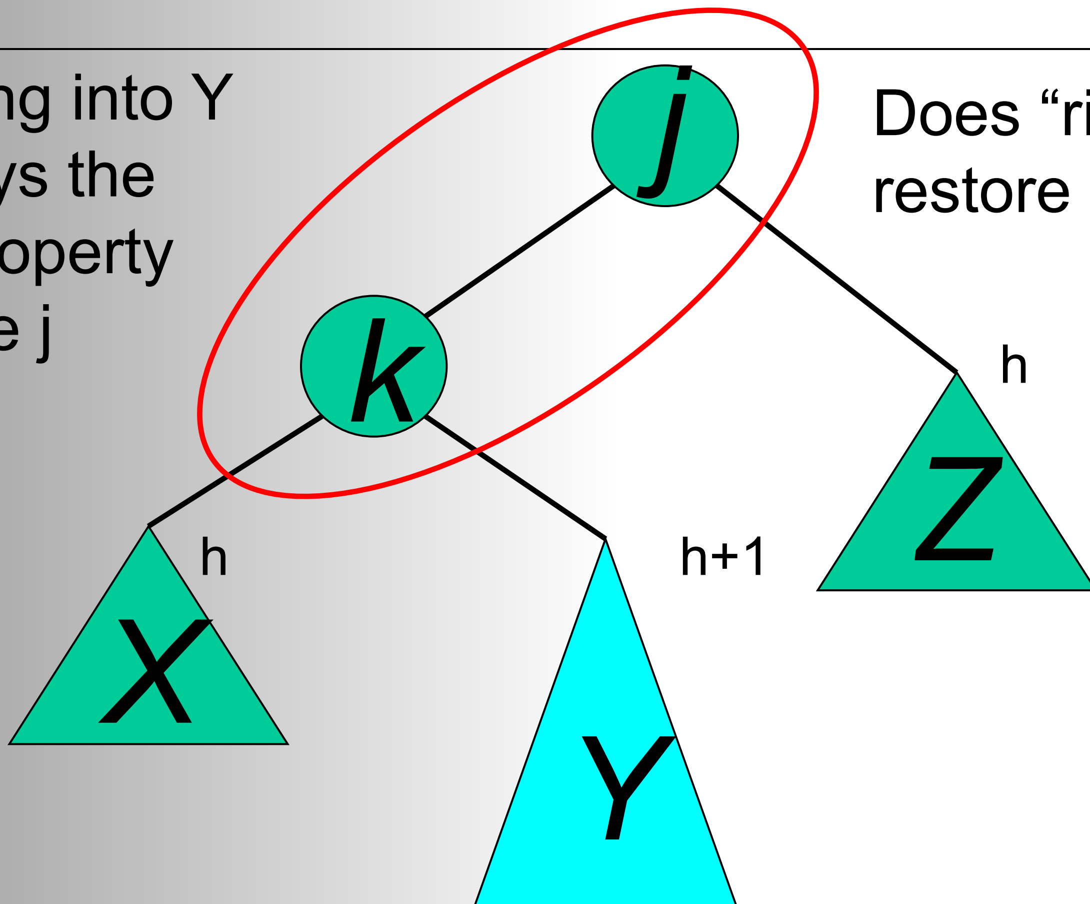
AVL Insertion: Inside Case

Consider a valid AVL subtree



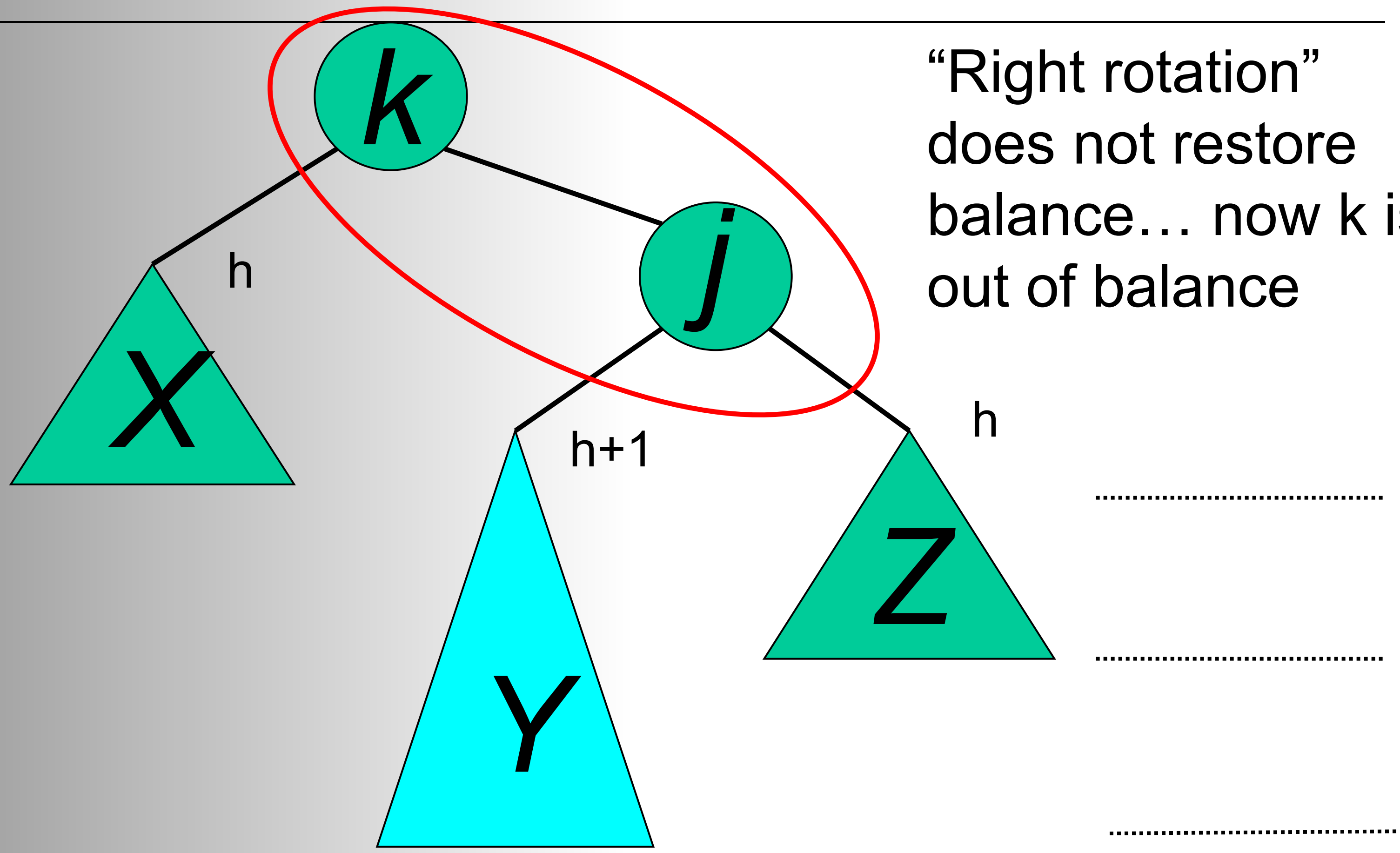
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does “right rotation” restore balance?

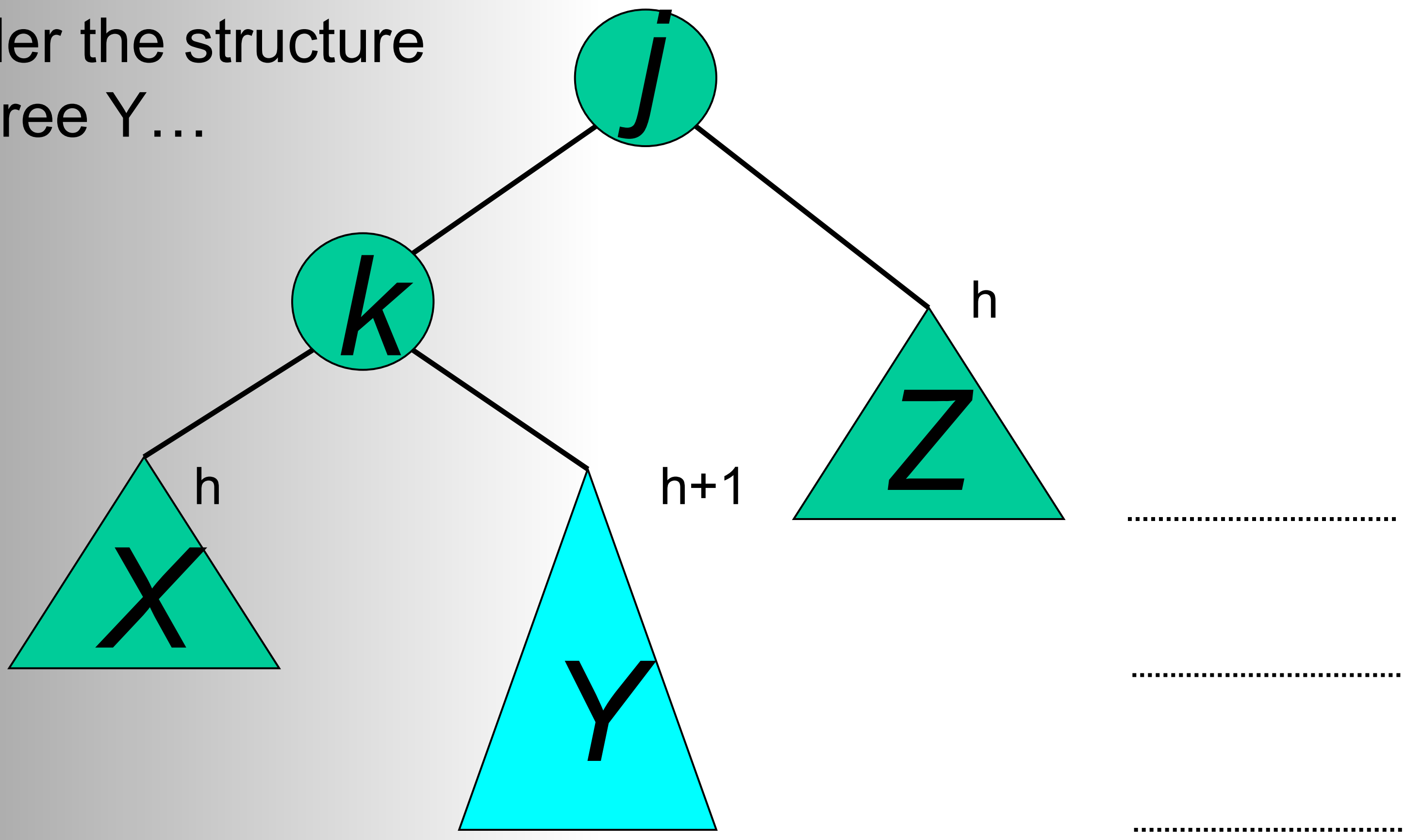
AVL Insertion: Inside Case



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

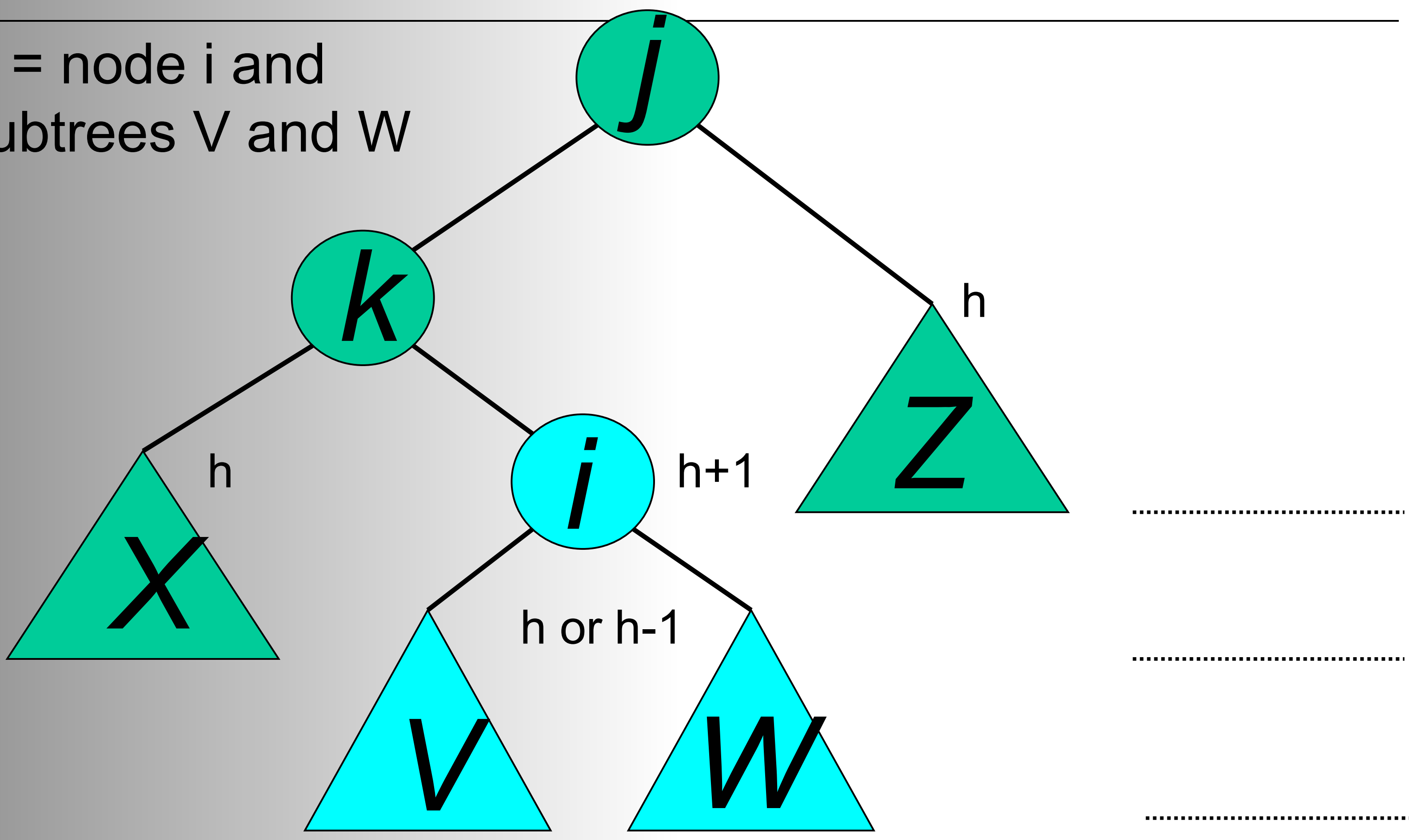
AVL Insertion: Inside Case

Consider the structure of subtree Y…

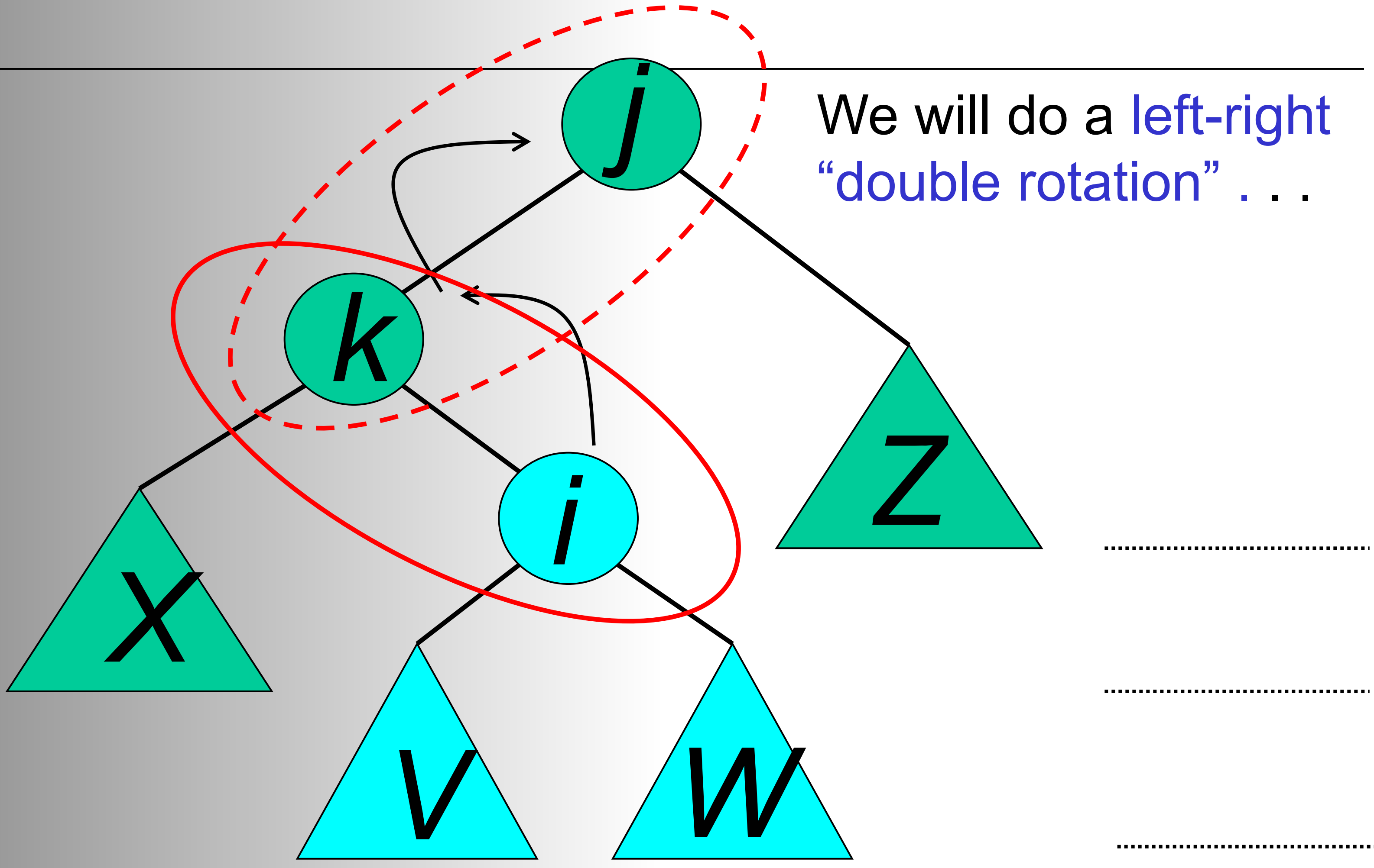


AVL Insertion: Inside Case

Y = node i and subtrees V and W

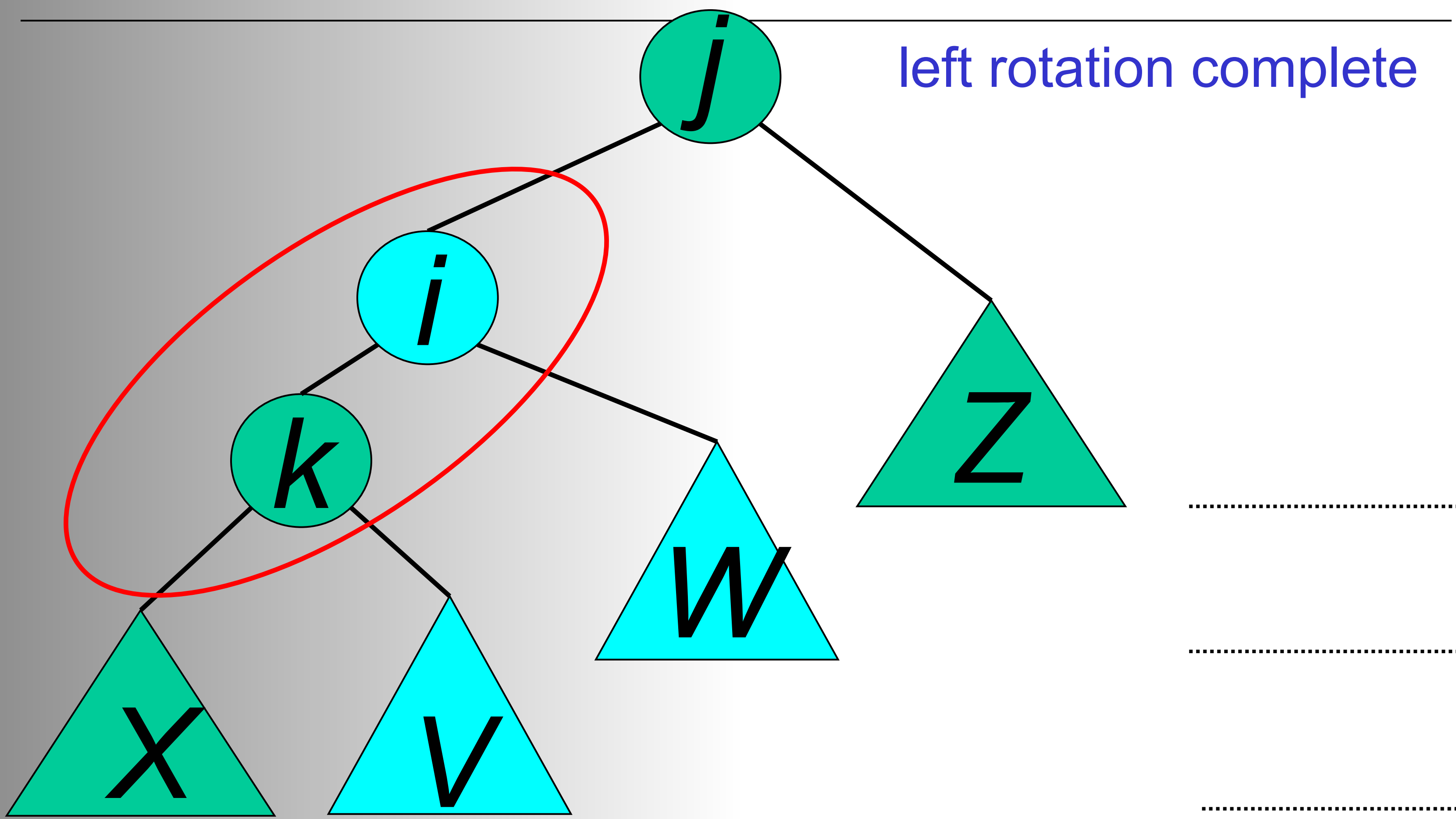


AVL Insertion: Inside Case

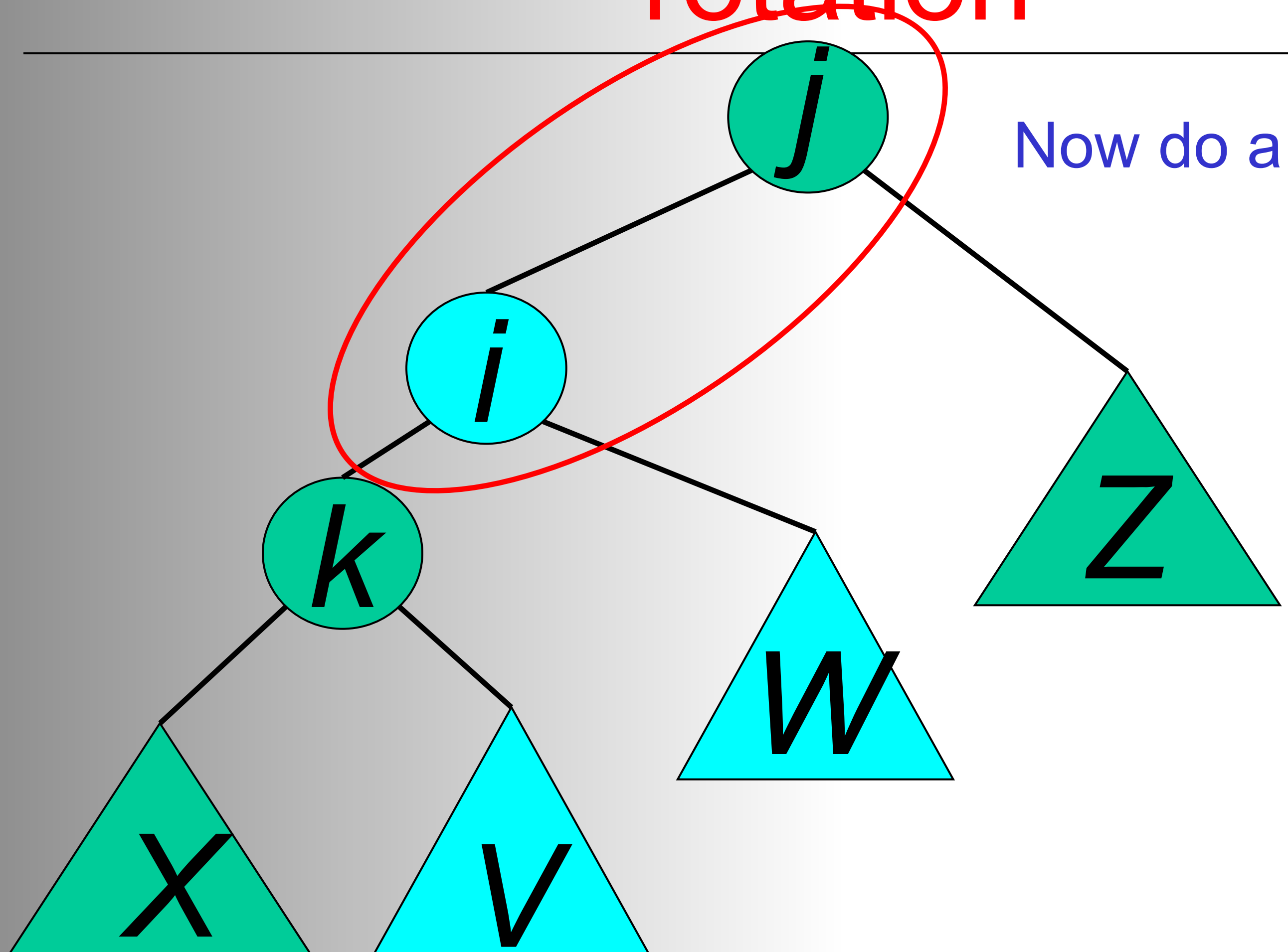


We will do a left-right "double rotation" . . .

Double rotation : first rotation



Double rotation : second rotation

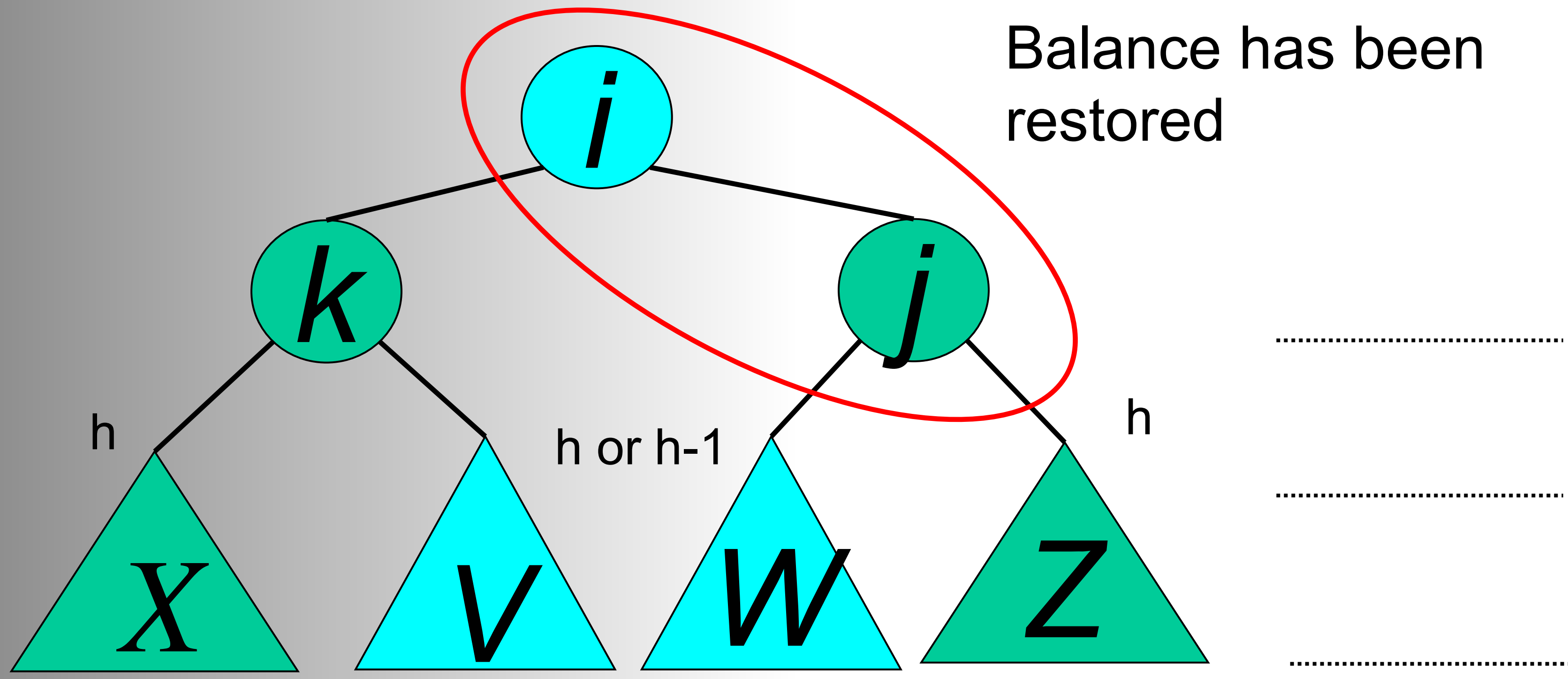


Now do a right rotation

Double rotation : second rotation

right rotation complete

Balance has been restored

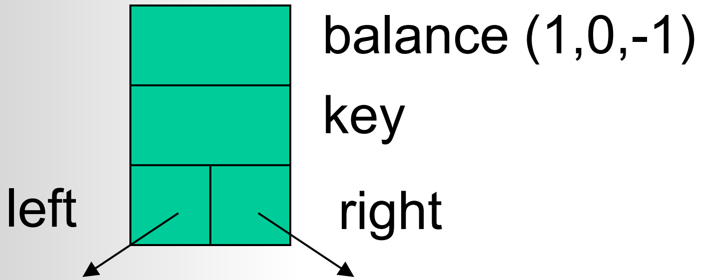


Exercise

- Construct an AVL Search Tree by inserting the following elements:

- 50, 20, 80, 10, 30, 5, 15, 17, 19, 14, 16, 18
- F, C, E, T, J, Z, D, B, A, Y

Implementation



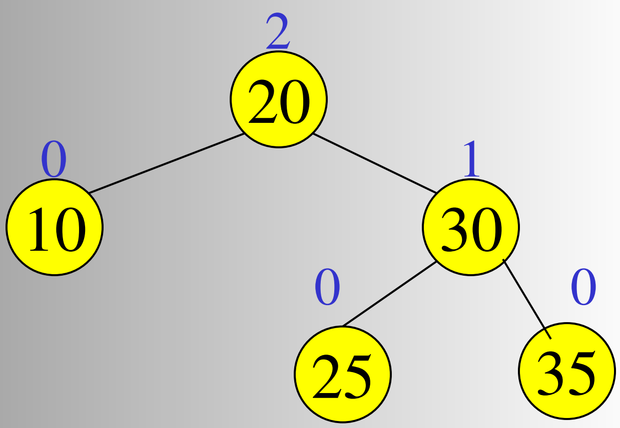
No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

Insertion in AVL Trees

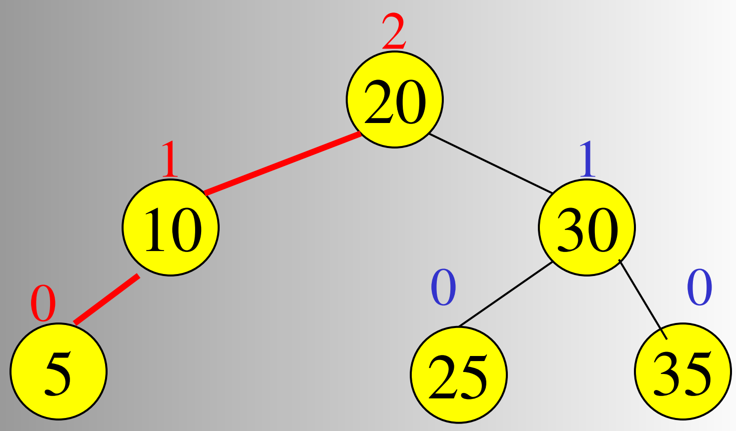
- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by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

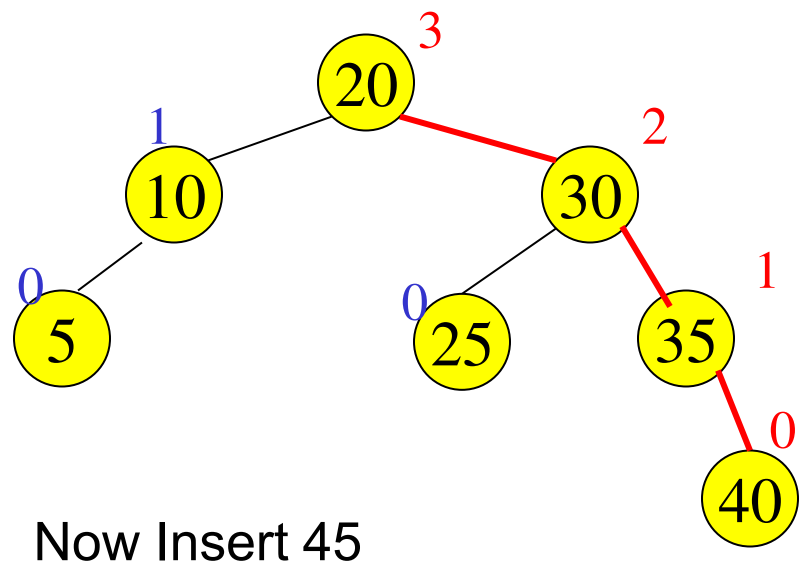
Example of Insertions in an AVL Tree



Insert 5, 40

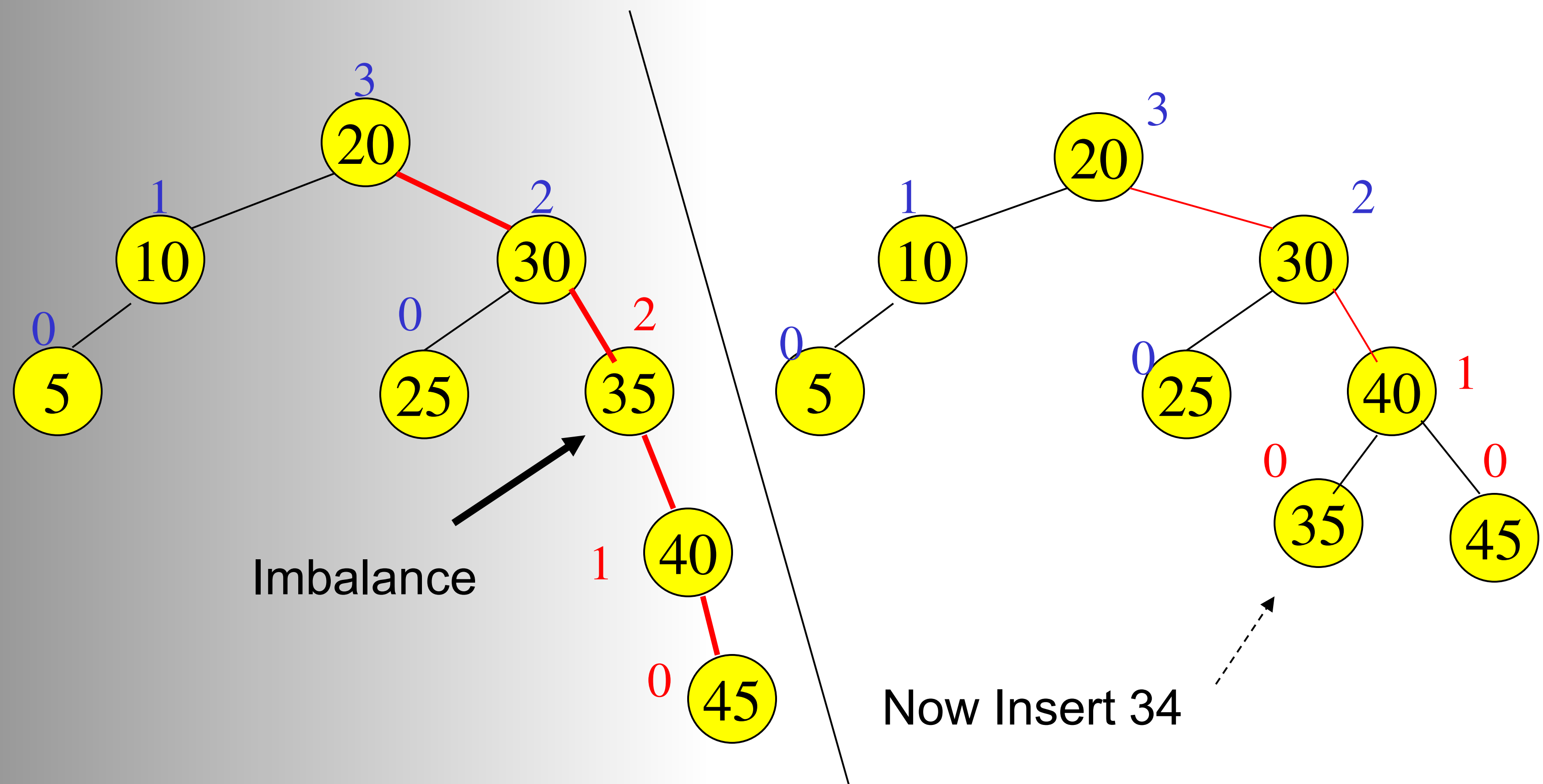
Example of Insertions in an AVL Tree



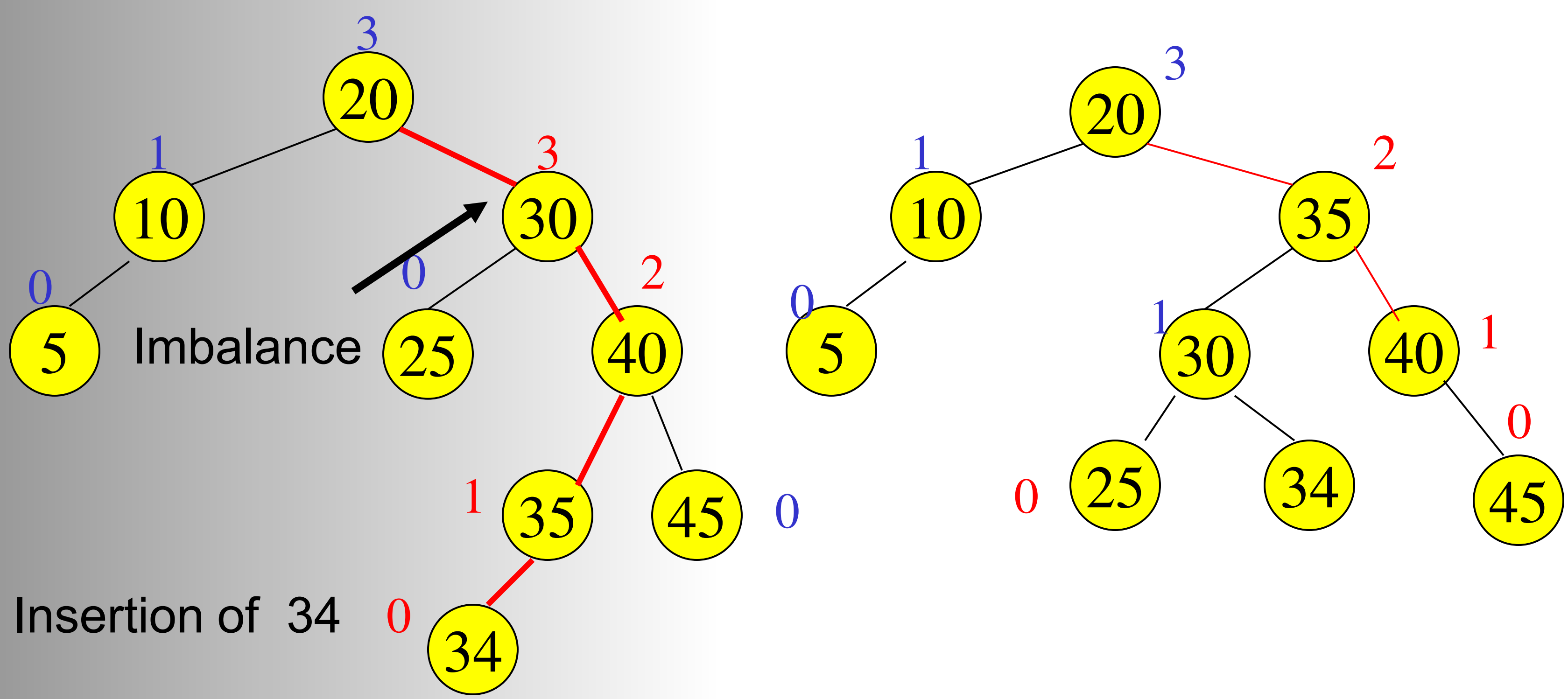


Now Insert 45

Single rotation (outside case)



Double rotation (inside case)



Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
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 & 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).