ITP20001/ECE20010 Data Structures

Chapter 5, 7, 9, 10

- *introduction*
- *binary tree*
- *complete binary tree*
  - *max heap, min heap*
  - *Chapter 7 – heap sorting*
  - *Chapter 9 - priority queues*
- *binary search tree(bst)*
- ***AVL tree -*** *Chapter 10 – Efficient BST*

Major references:

1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

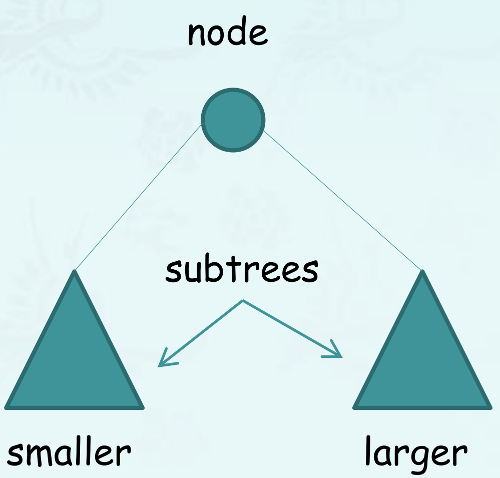
BST

- **Definition: A binary search tree is a binary tree in symmetric order.**

- A binary tree is either
  - empty
  - a key-value pair and two binary trees [neither of which contain that key]

equal keys ruled out

- Symmetric order means that
  - every node has a key
  - every node's key is
    larger than all keys in its left subtree
    smaller than all keys in its right subtree



BST

- **Definition: A binary search tree is a binary tree in symmetric order.**

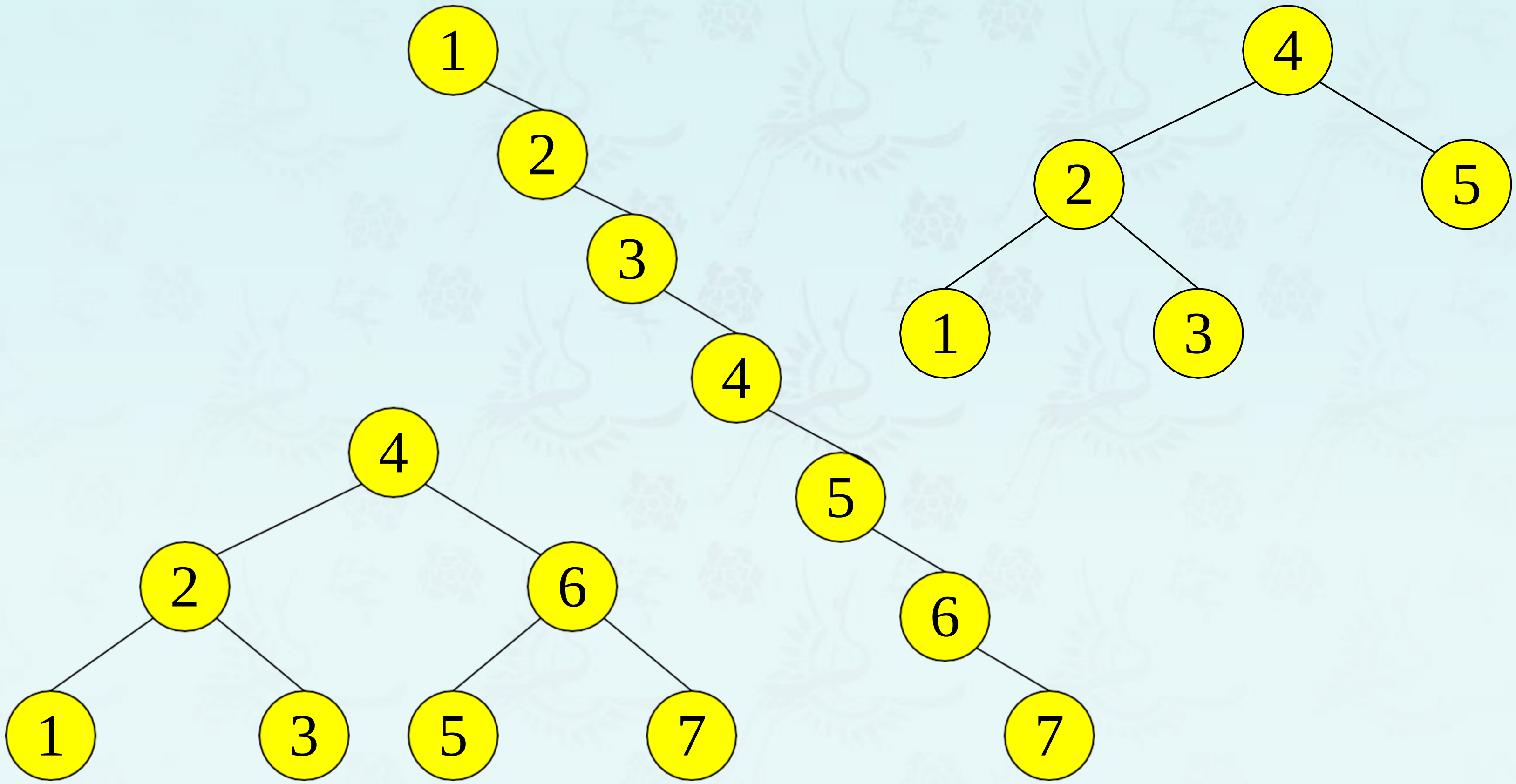
- All BST operations are O(d), where d is tree depth
- Minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

BST

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



Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

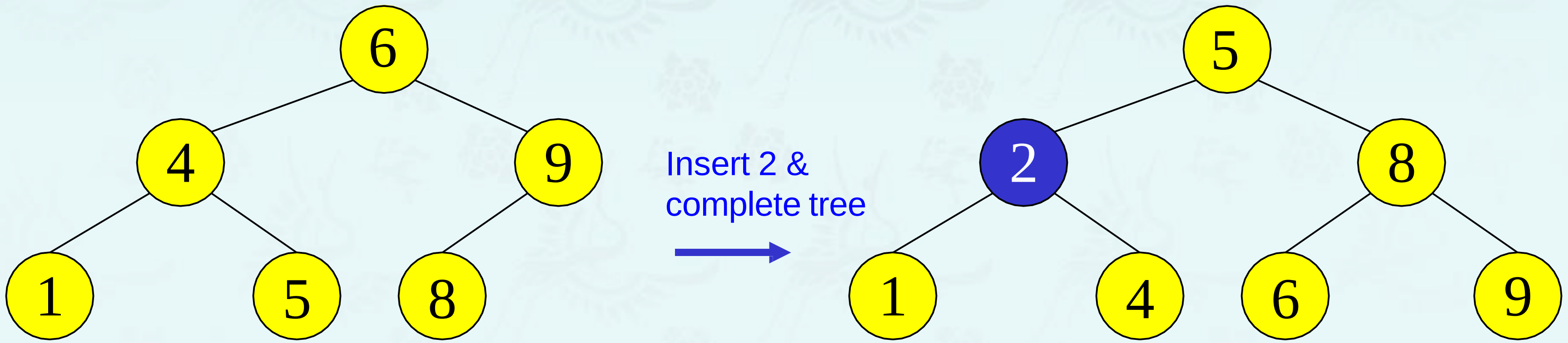
Balancing Binary Search Trees

Many algorithms exist for keeping BST balanced

- Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
- Weight-balanced trees
- Red-black trees;
- Splay trees and other self-adjusting trees
- B-trees and other (e.g. 2-4 trees) multiway search trees

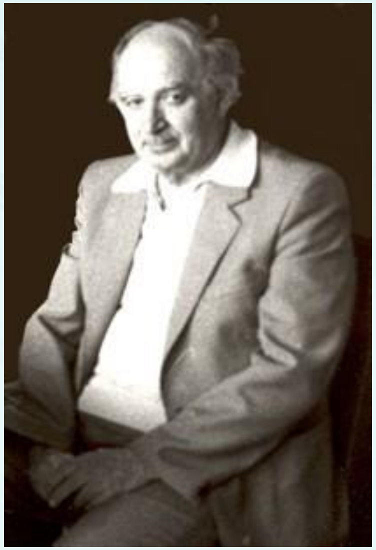
Perfect Balance

- 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 Trees (1962)

- Named after 2 Russian mathematicians
- Georgii Adelson-Velsky (1922 - ?)
- Evgenii Mikhailovich Landis (1921-1997)

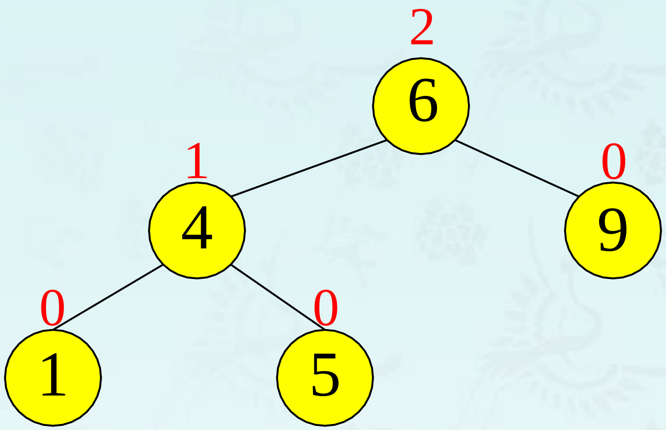


AVL - Good but not Perfect Balance

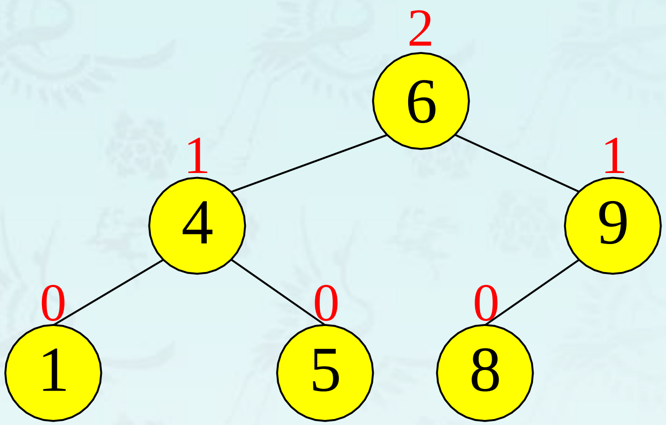
- Height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node or compute it on the fly

Node Heights

Tree A (AVL)



Tree B (AVL)

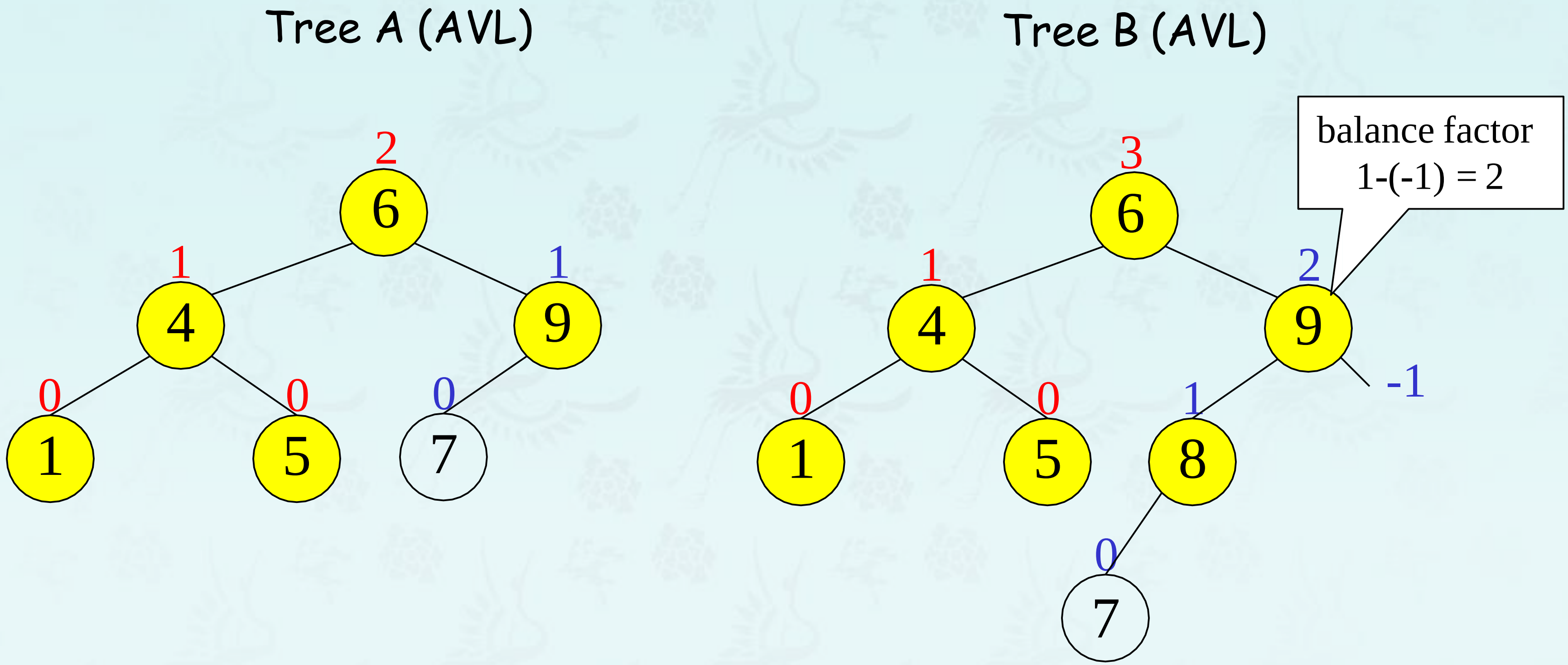


height of node = h

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

empty height = -1

Node Heights after Insert 7

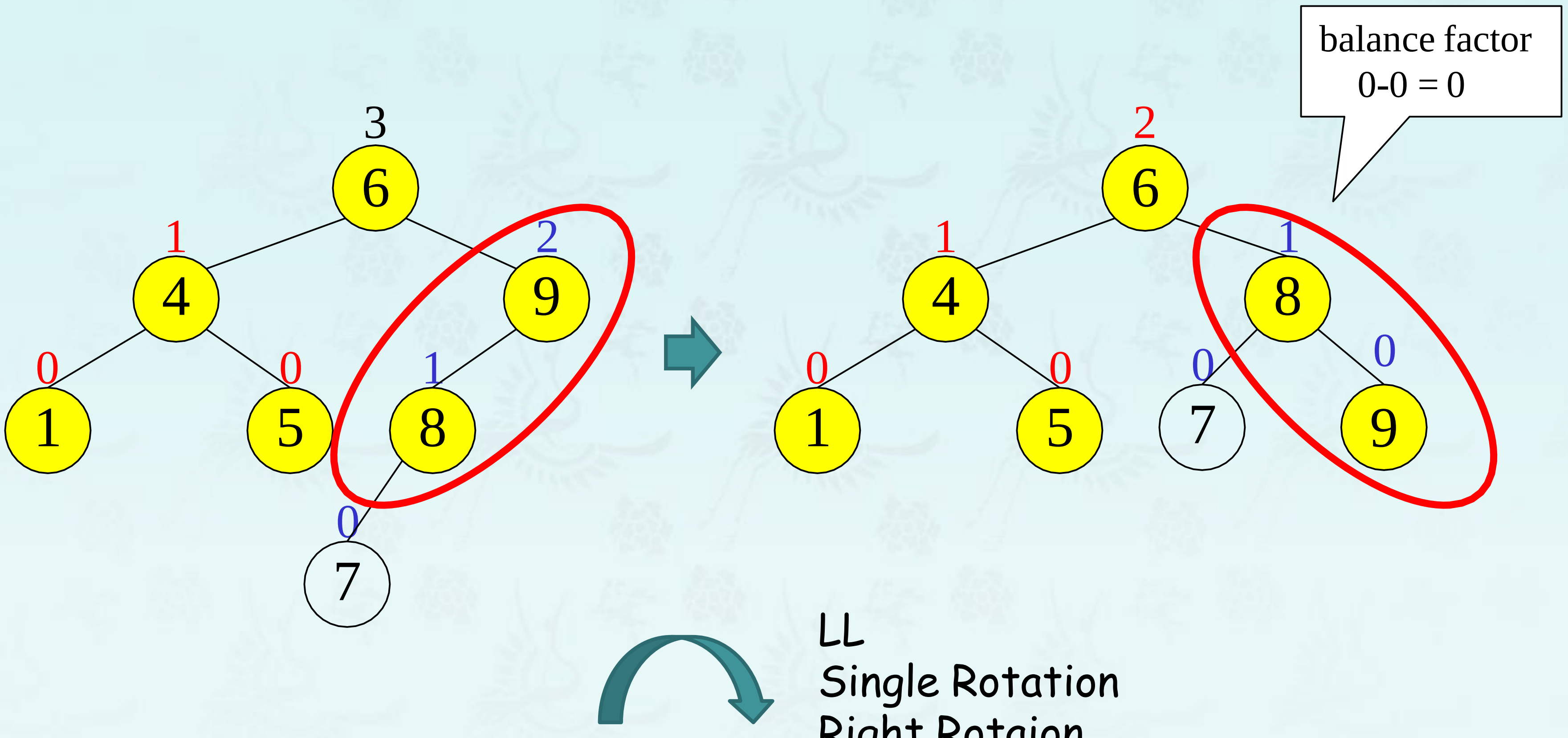


height of node = h
balance factor = $h_{left} - h_{right}$
empty height = -1

Insert and Rotation in AVL Trees

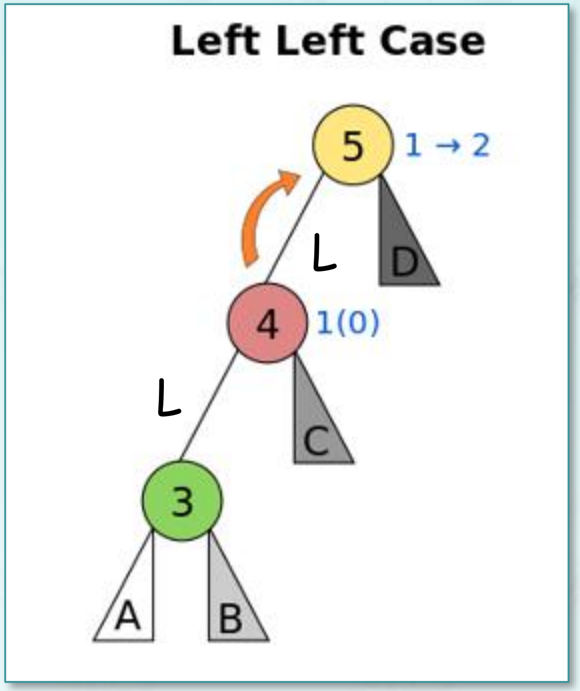
- Insert operation may cause balance factor to become 2 or –2 for some node
  - Only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, **go back up** to the root node by node, updating heights
  - If a new balance factor (the difference $h_{left}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

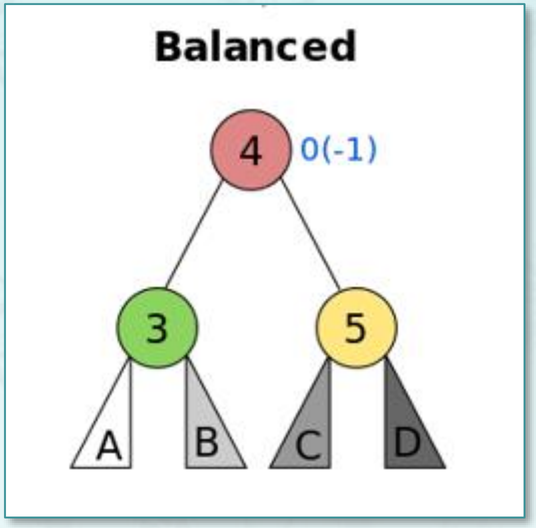
Single Rotation in an AVL Tree



LL
Single Rotation
Right Rotaion

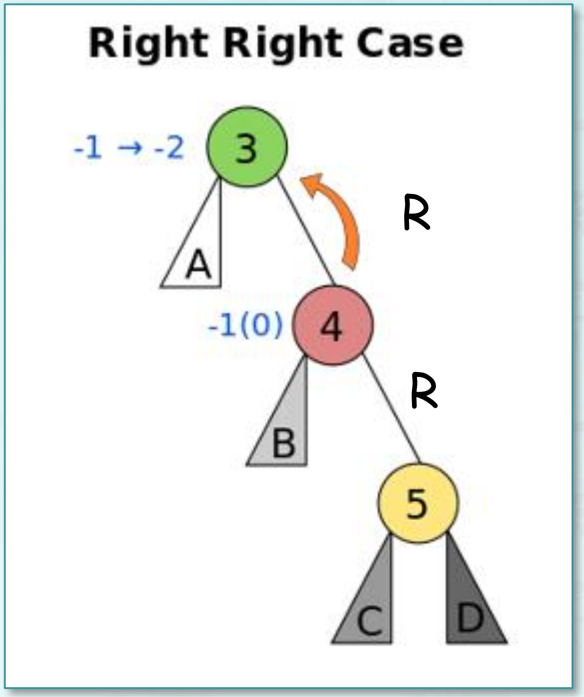
Single Rotation in an AVL Tree

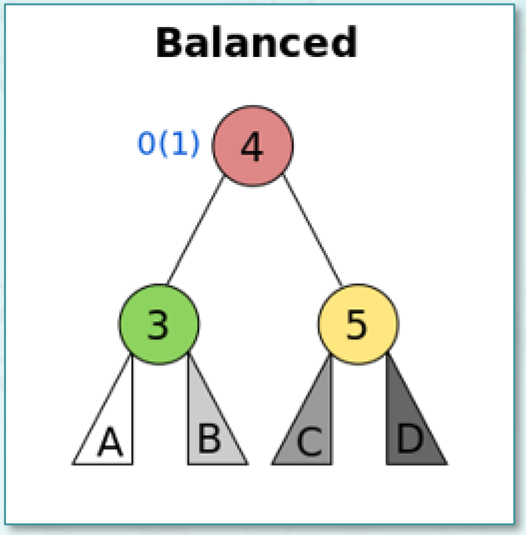




LL Case
Single Right Rotation

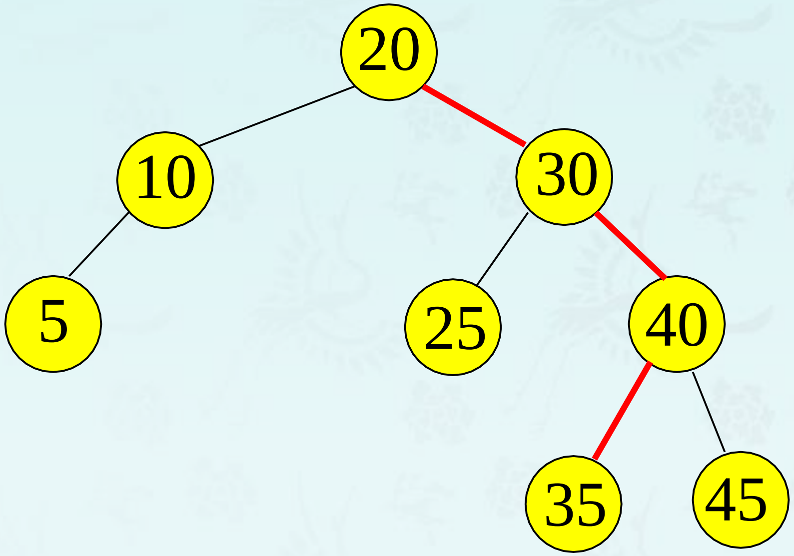
Single Rotation in an AVL Tree





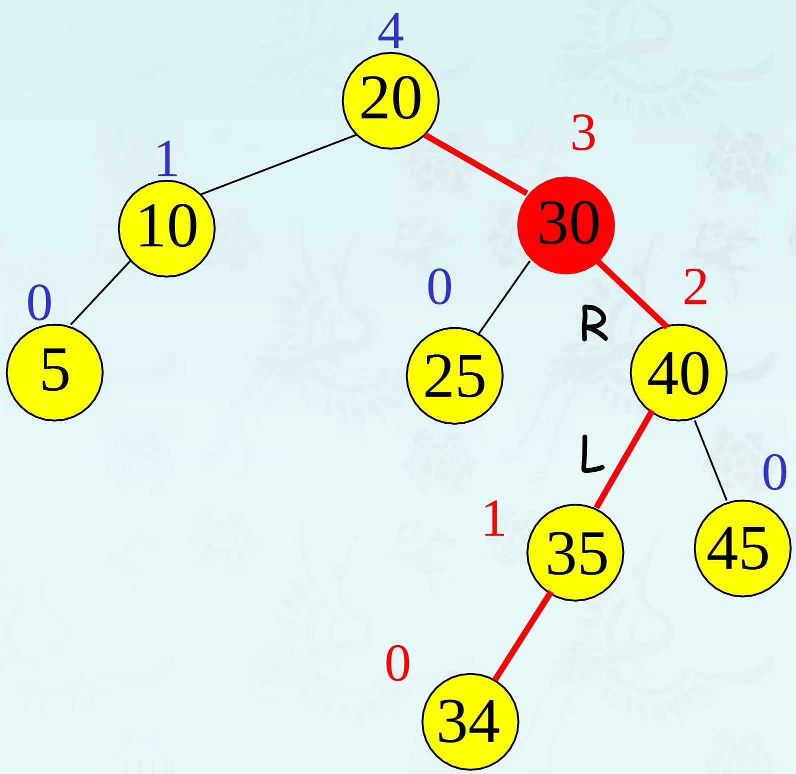
RR Case
Single Left Rotation

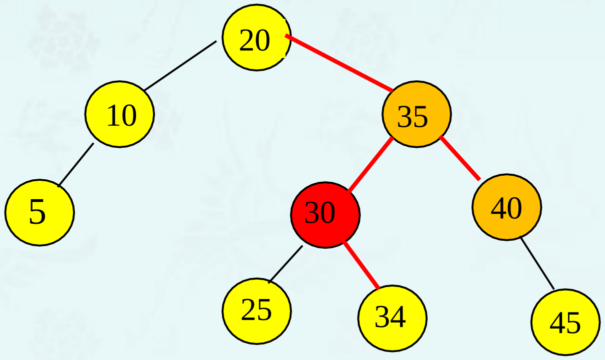
AVL Tree Balanced?



AVL Tree Balanced?

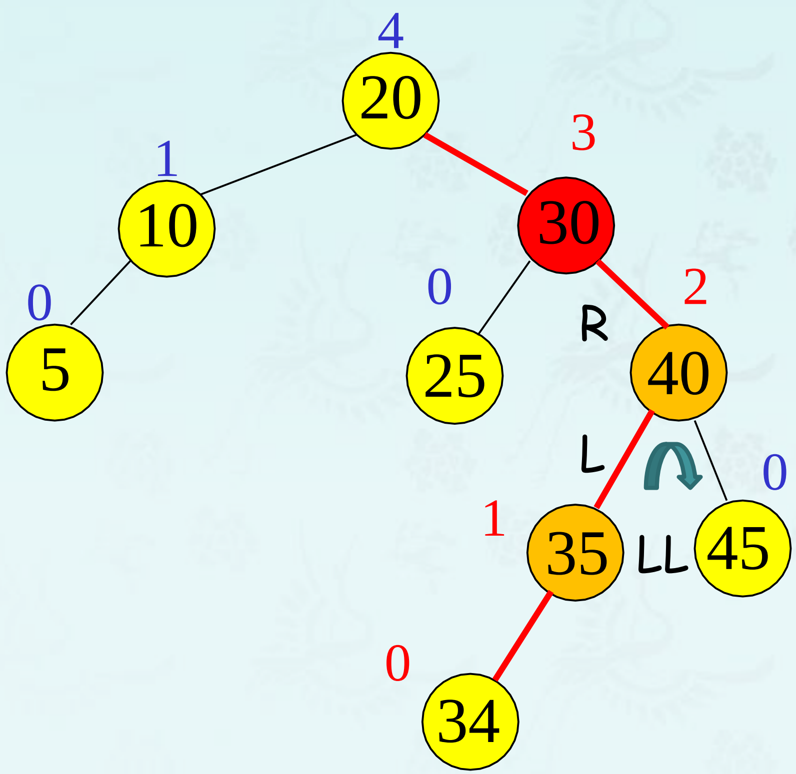
Insertion of 34
Imbalance at 30
Balance factor at 30 = -2

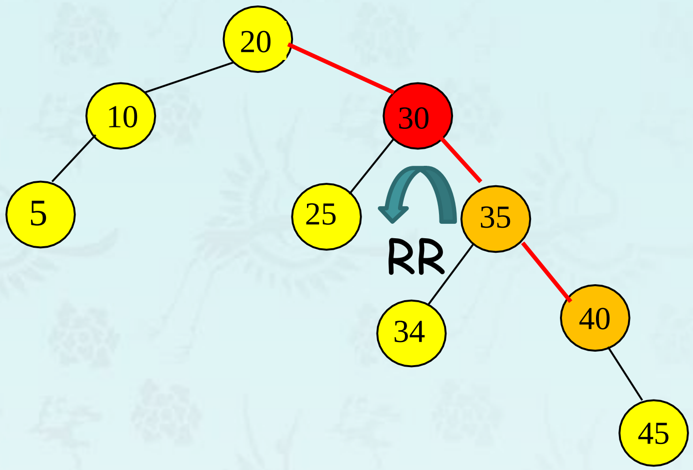


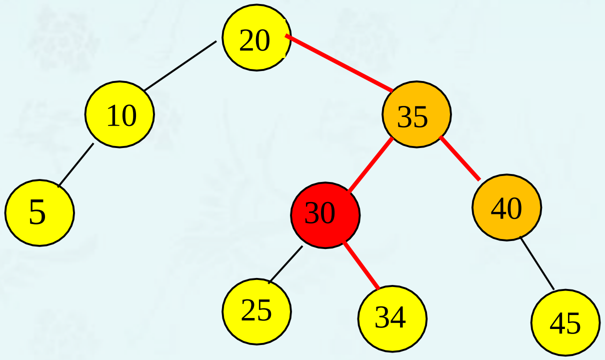


Double rotation RL

Insertion of 34
Imbalance at 30
Balance factor at 30 = -2

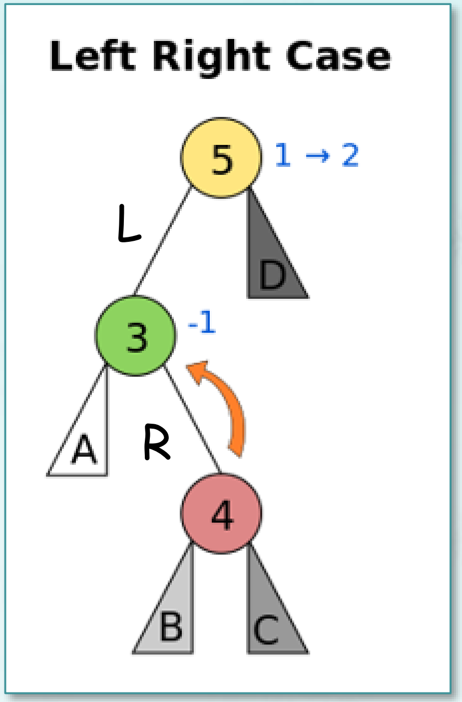


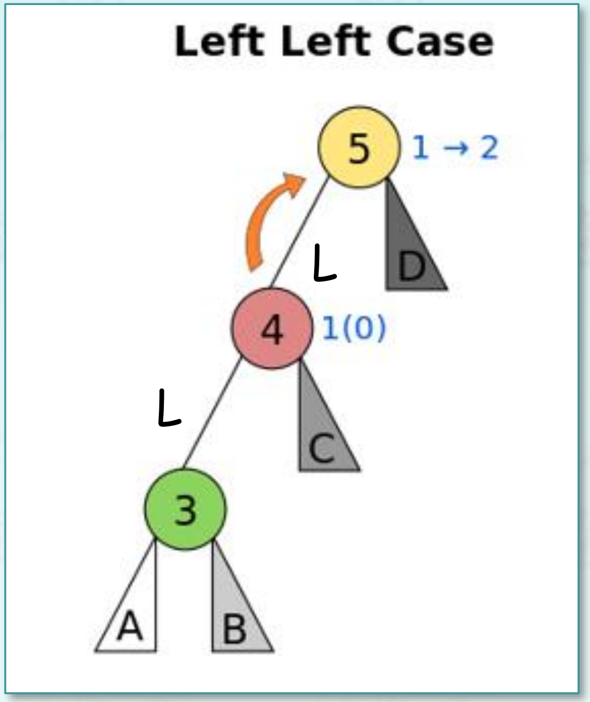


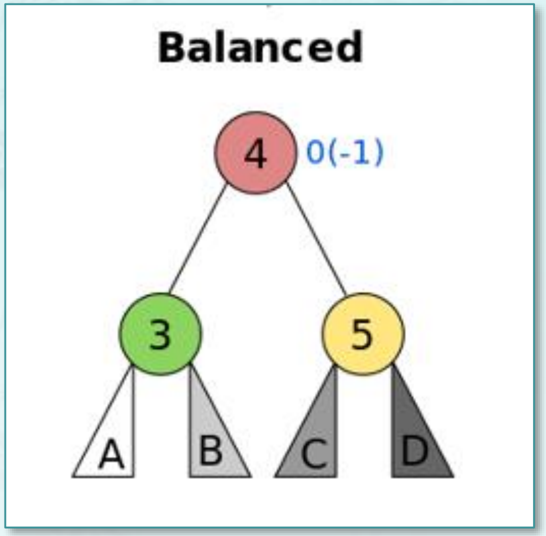


RL
double rotation
LL rotation + RR rotation

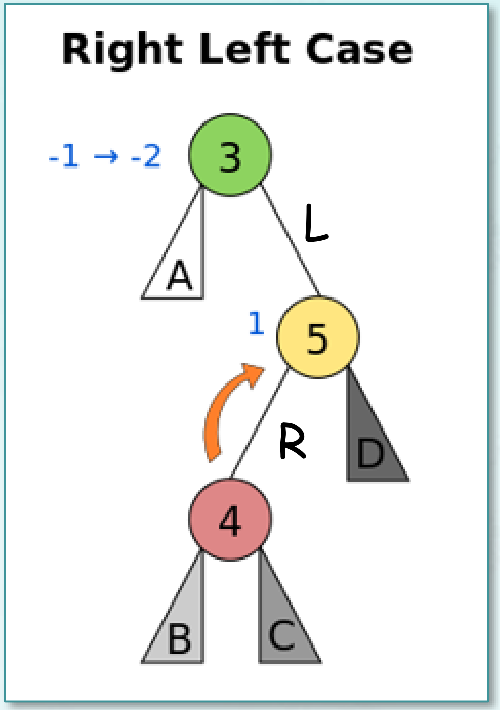
Double rotation – LR Case

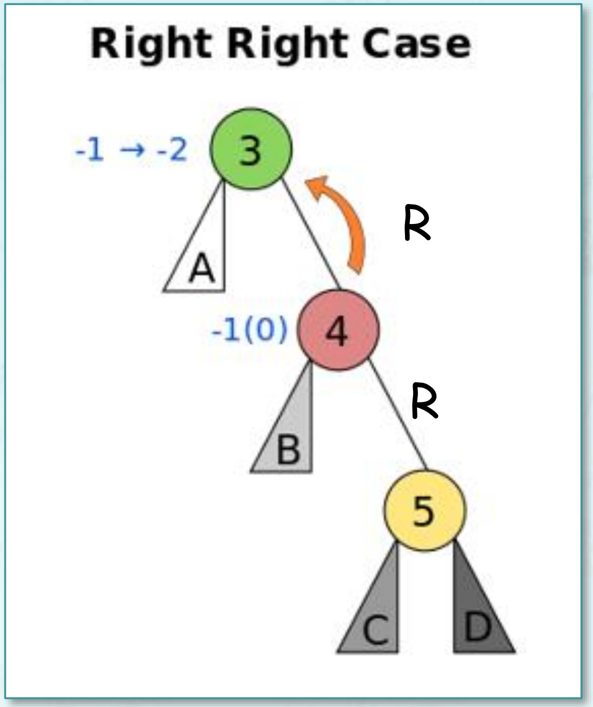


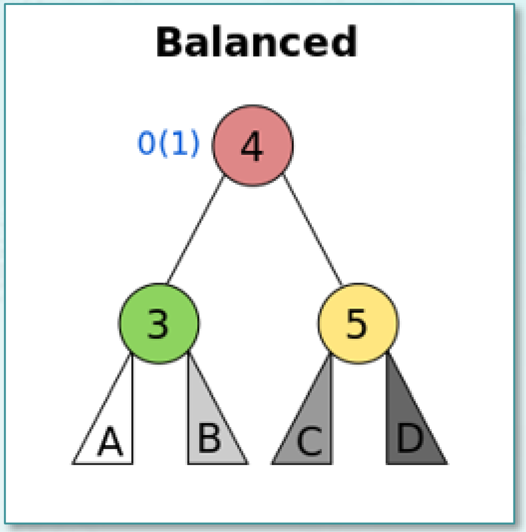




Double rotation - RL Case







Insertions in AVL Trees

Let the node that needs rebalancing be α.

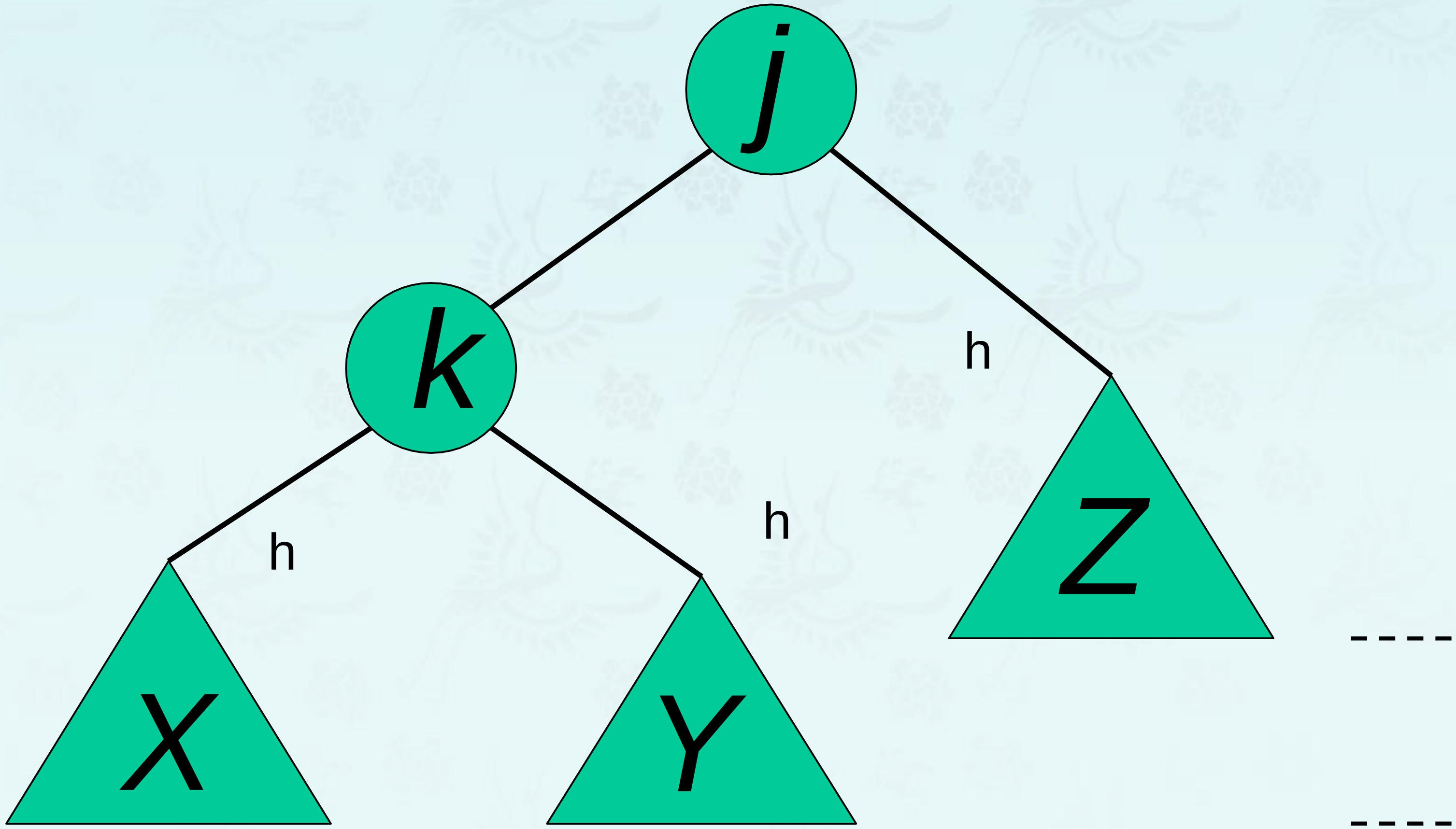
There are 4 cases:

- 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) :
  1. Insertion into right subtree of left child of α.
  2. Insertion into left subtree of right child of α.

The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

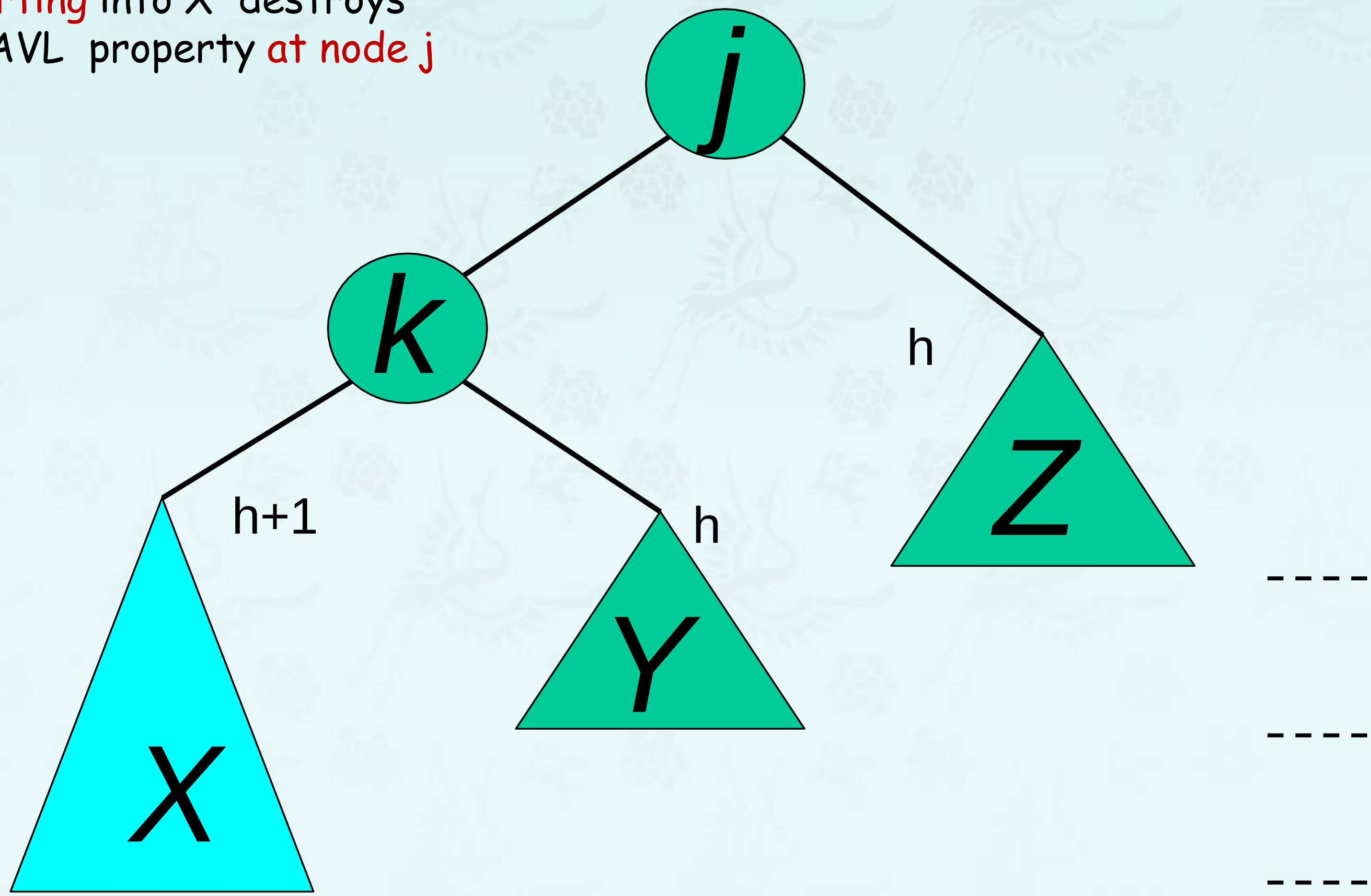
Consider a valid AVL subtree



AVL Insertion: Outside Case

Consider a valid AVL subtree

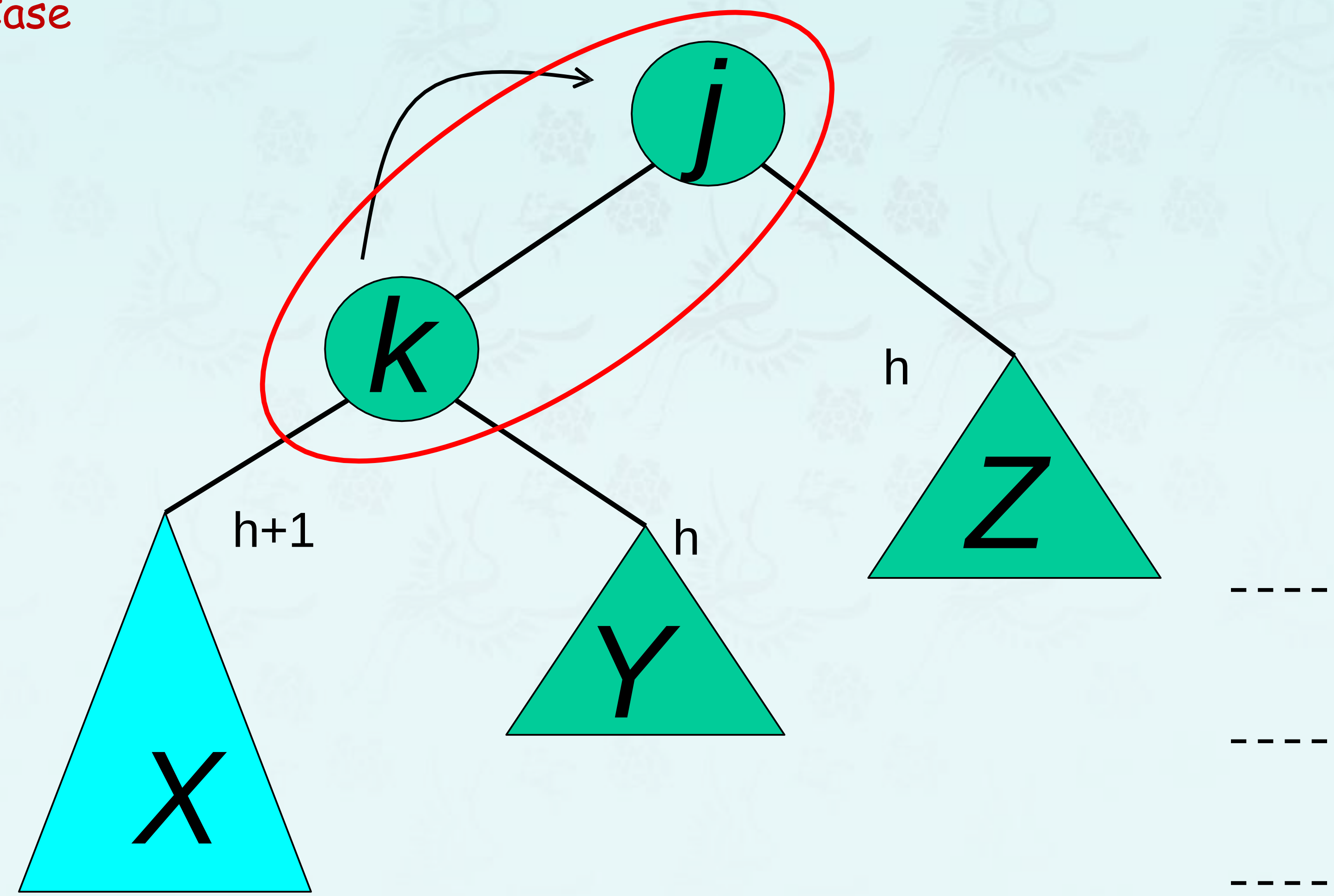
Inserting into X destroys the AVL property at node j



AVL Insertion: Outside Case

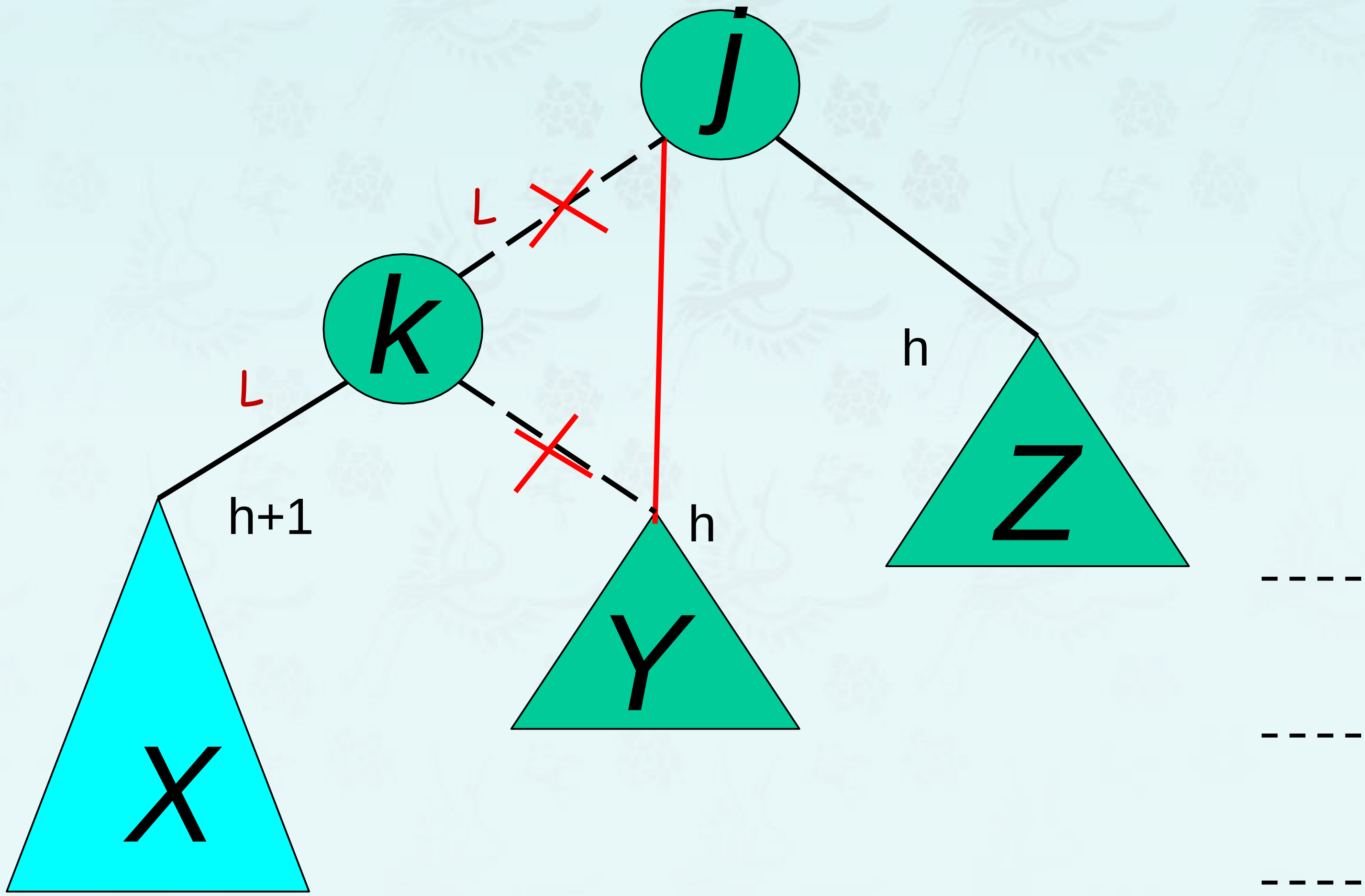
Do a "right rotation"

LL Case



Single right rotation

Do a "right rotation"

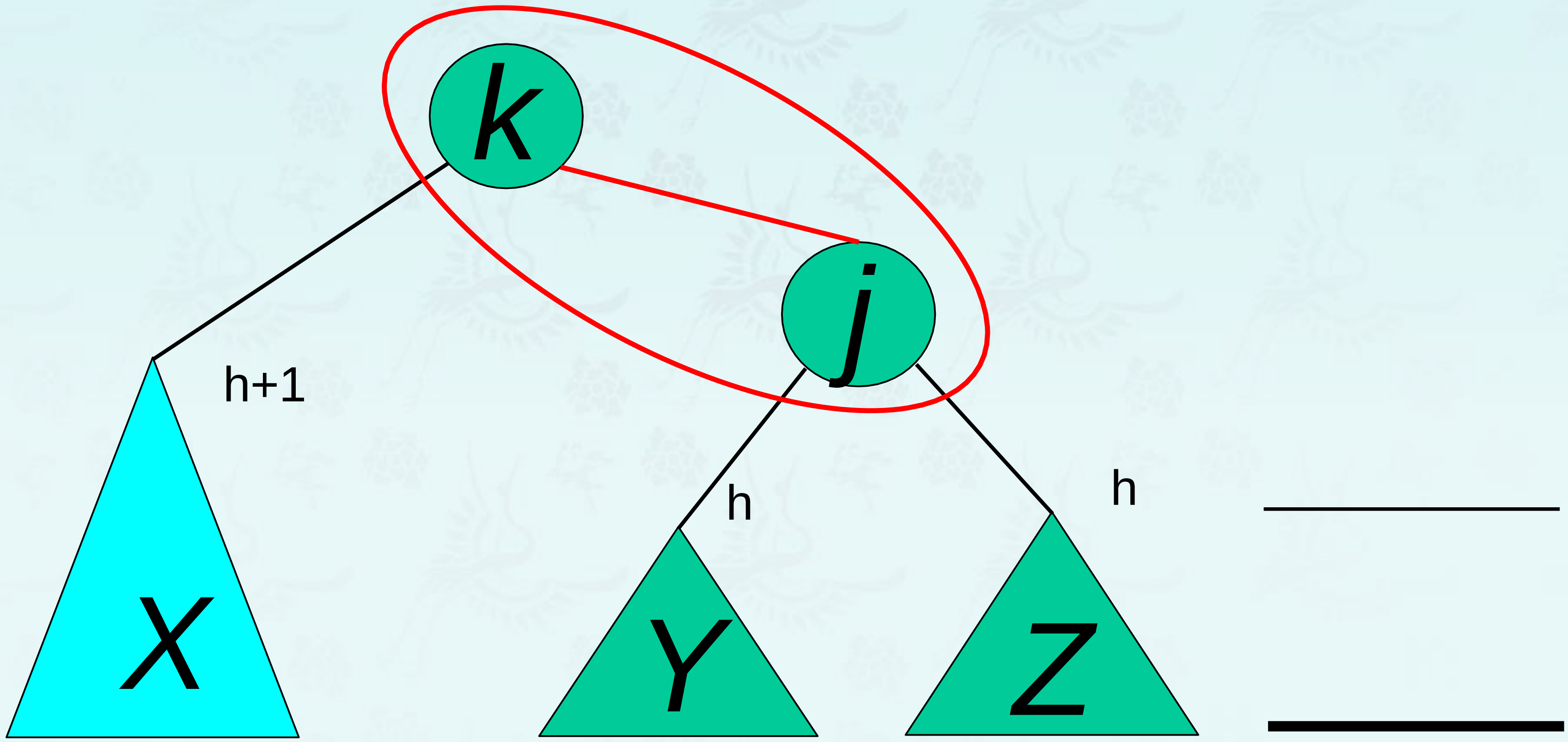


Outside Case Completed

AVL property has been restored!

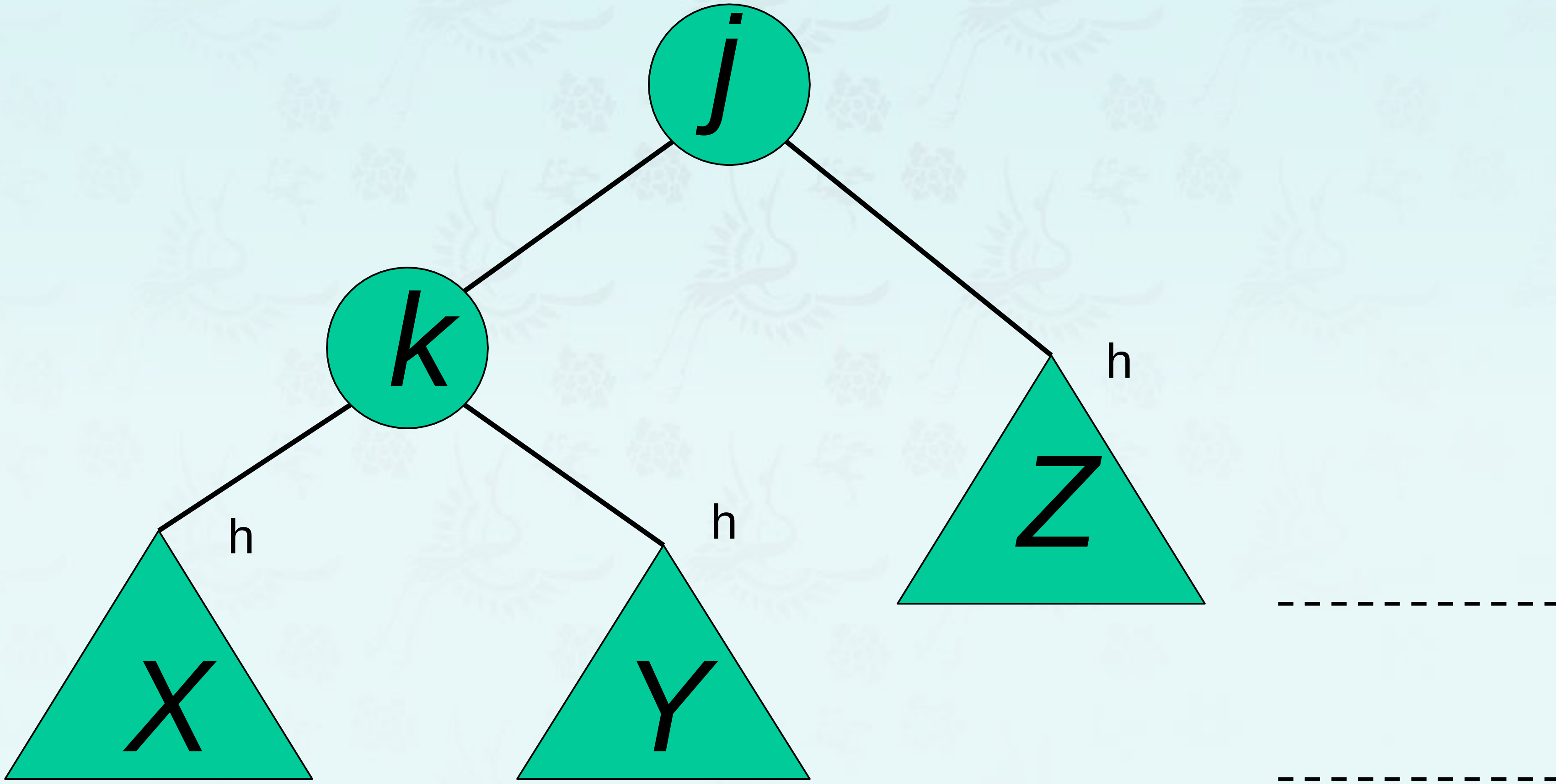
LL Case – Single Rotation

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



AVL Insertion: Inside Case

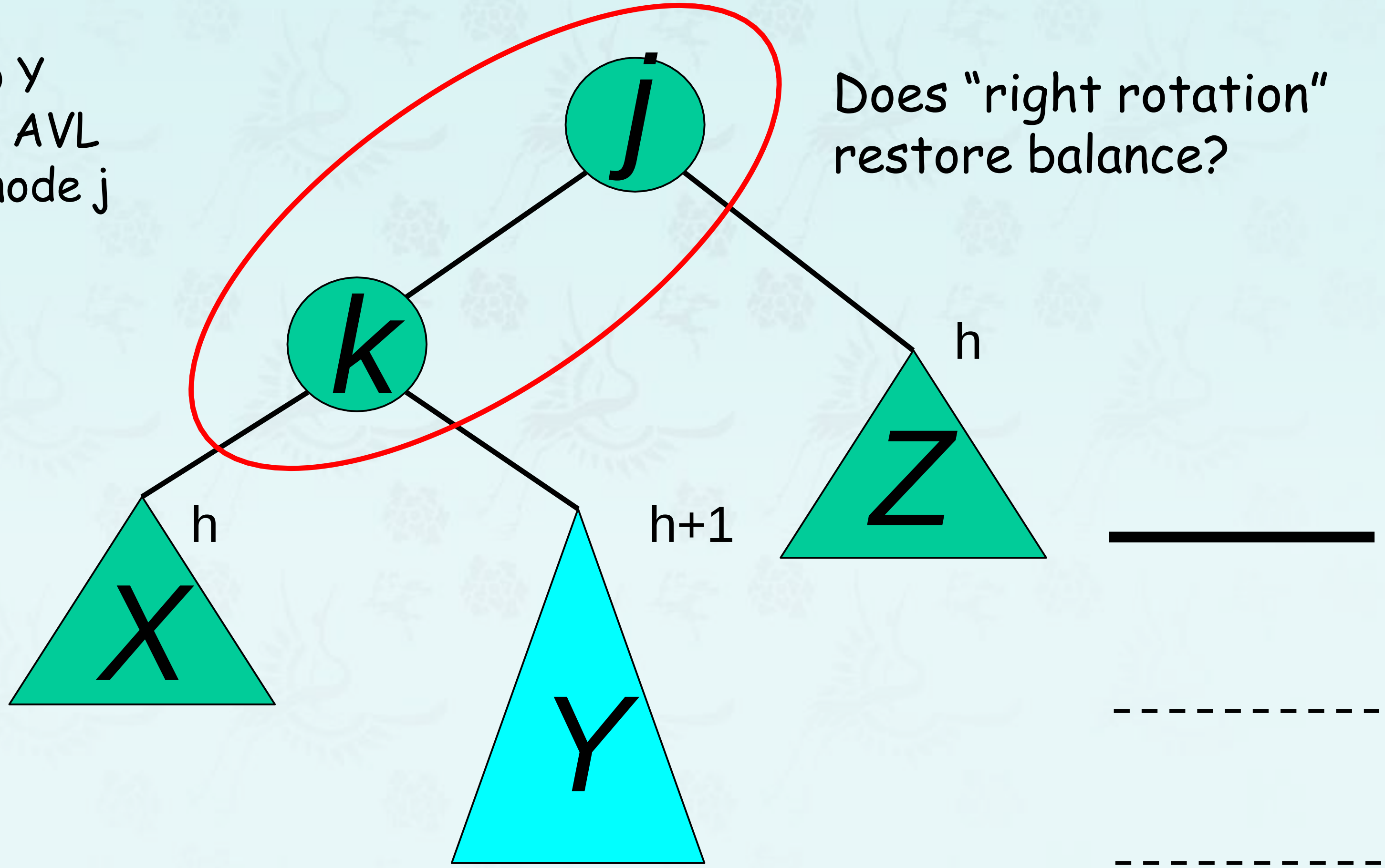
Consider a valid AVL subtree



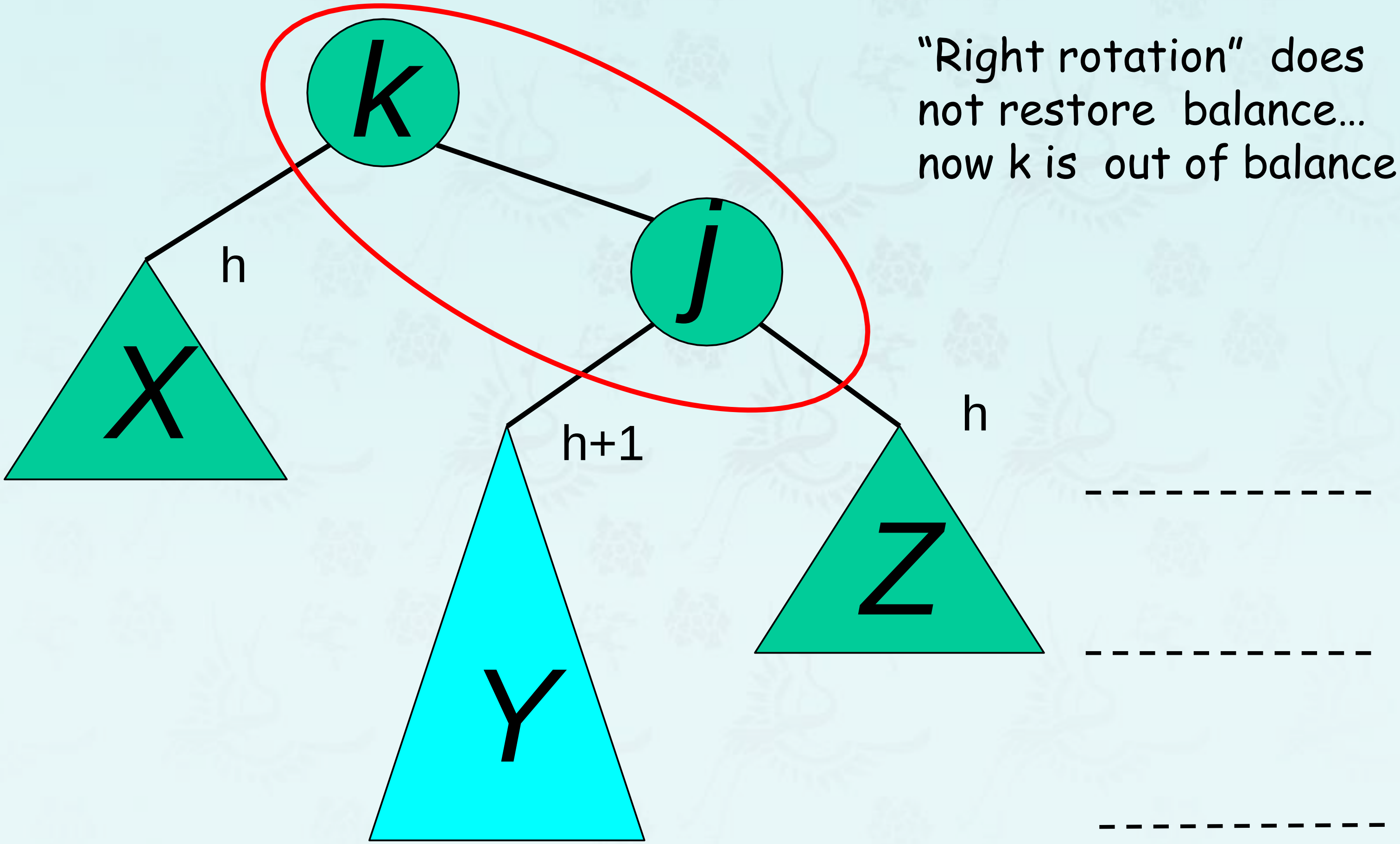
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does "right rotation" restore balance?



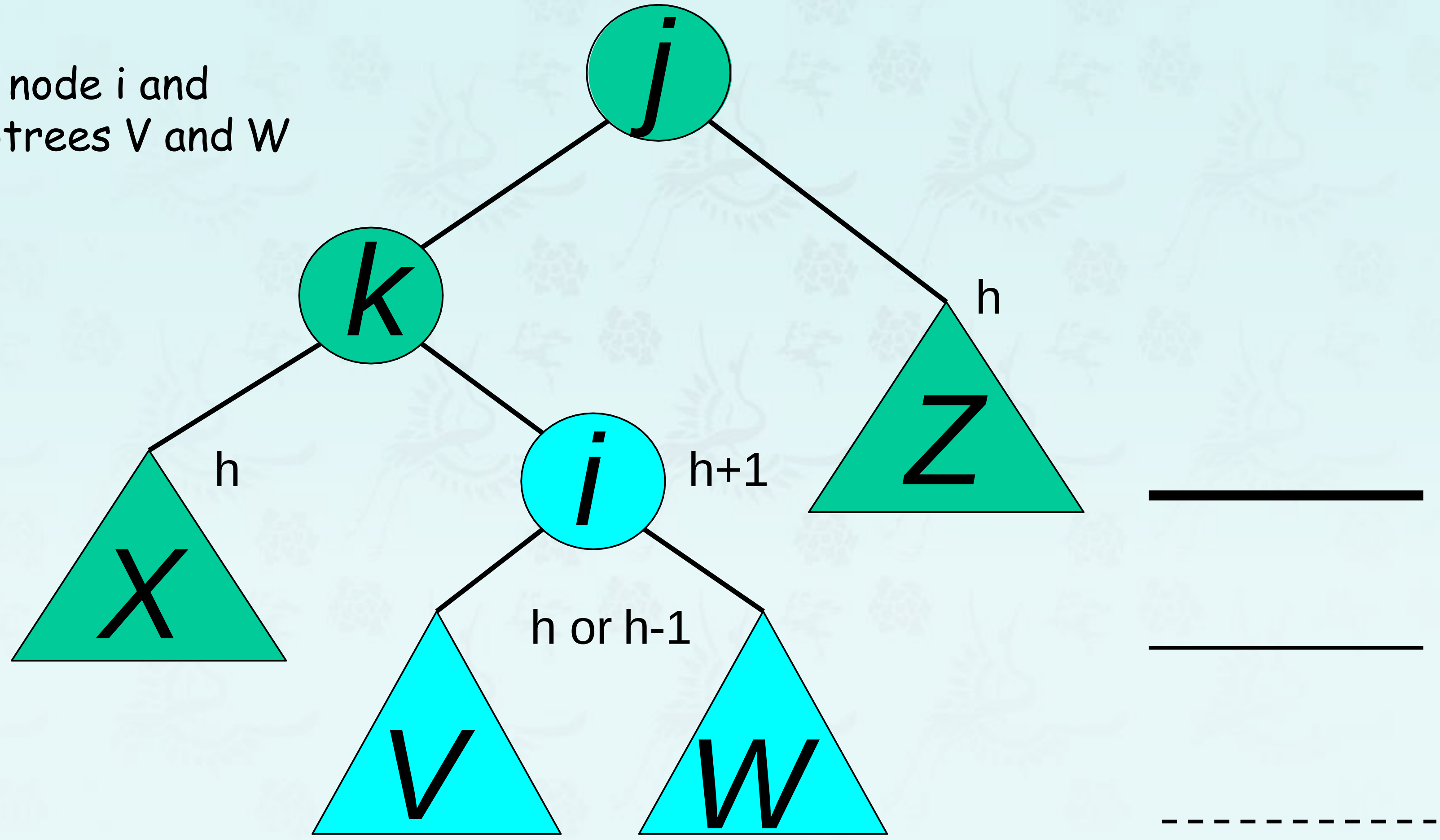
AVL Insertion: Inside Case



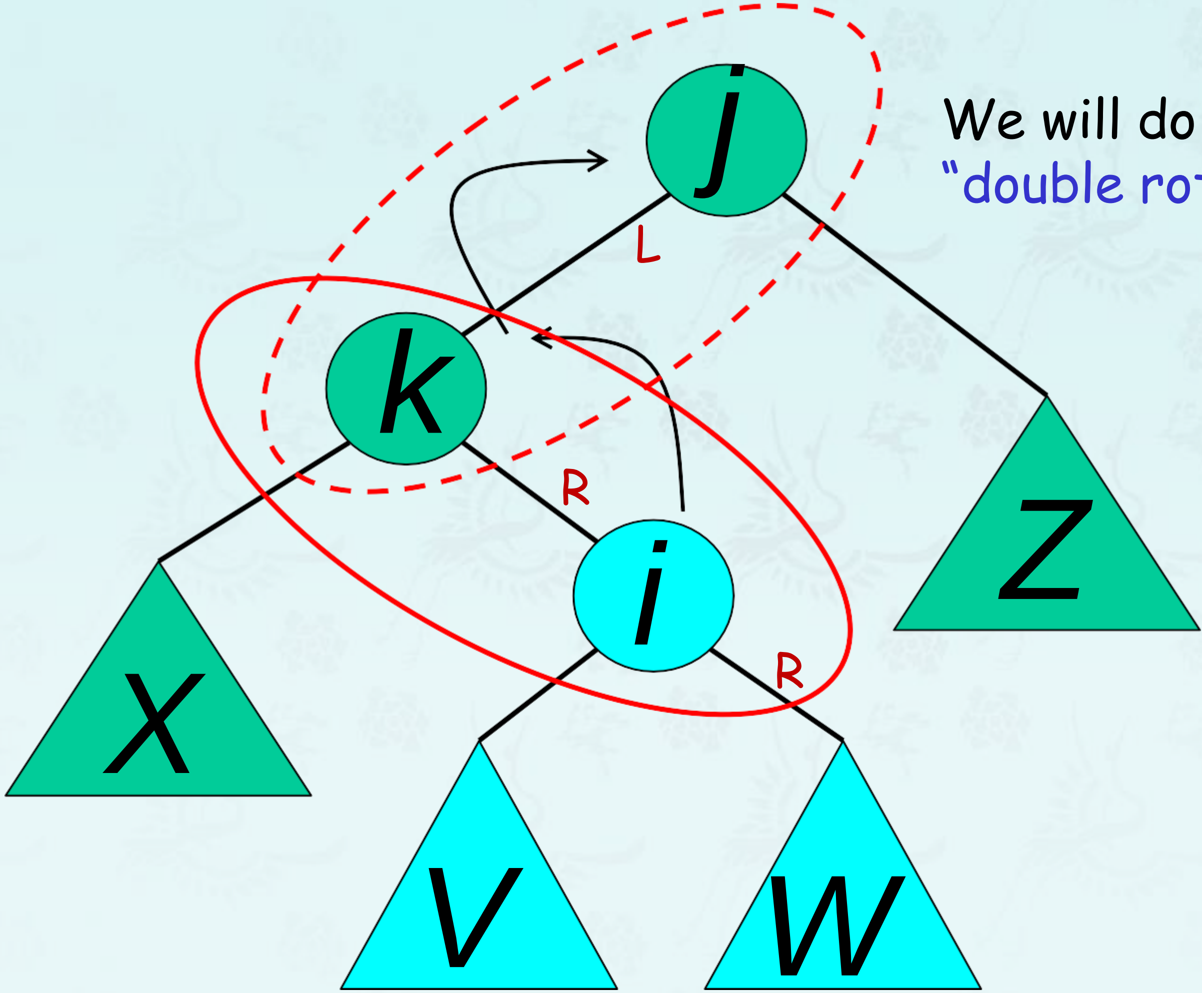
"Right rotation" does not restore balance… now k is out of balance

AVL Insertion: Inside Case

Y = node i and
subtrees V and W



AVL Insertion: Inside Case

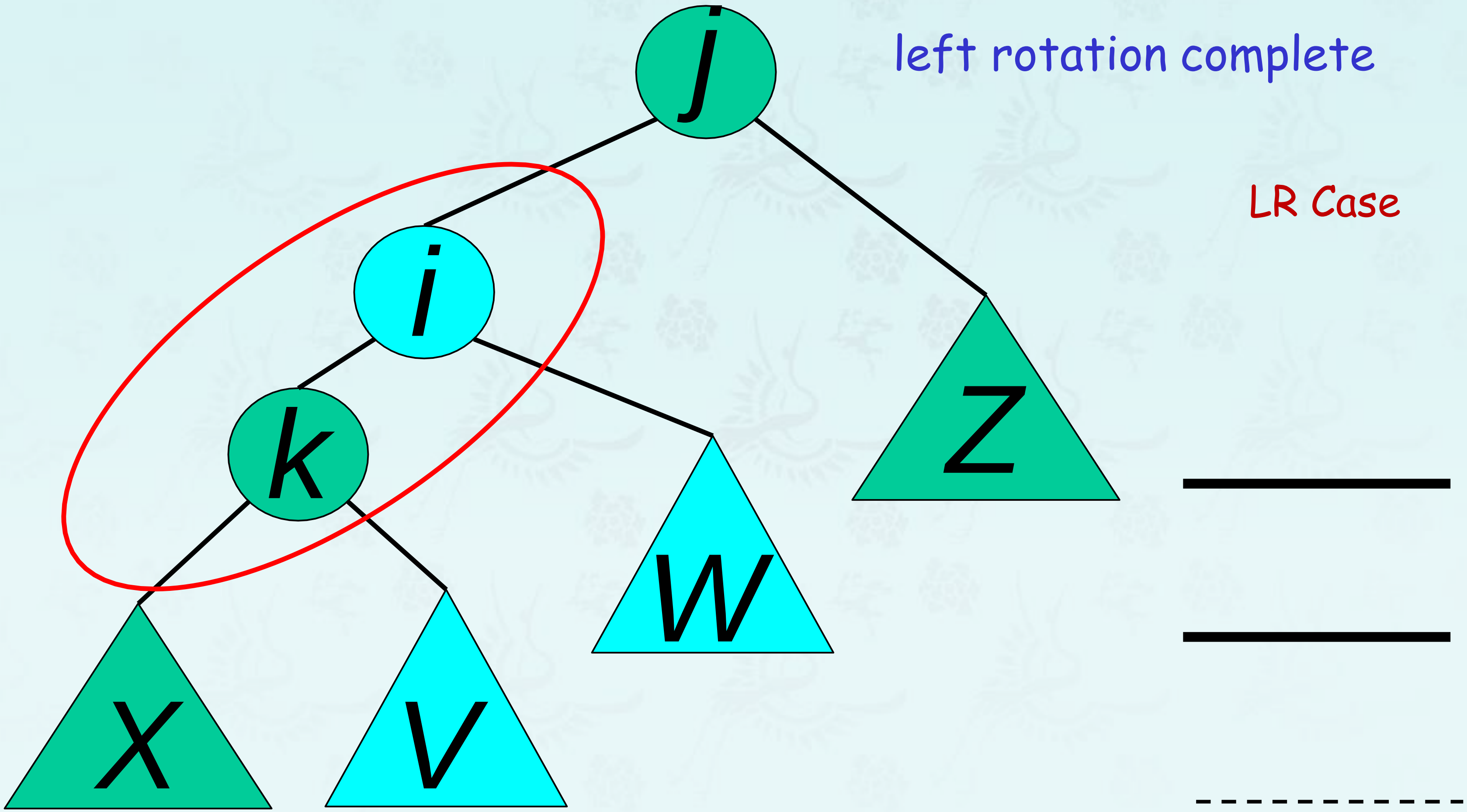


We will do a left-right "double rotation" . . .

LR Case

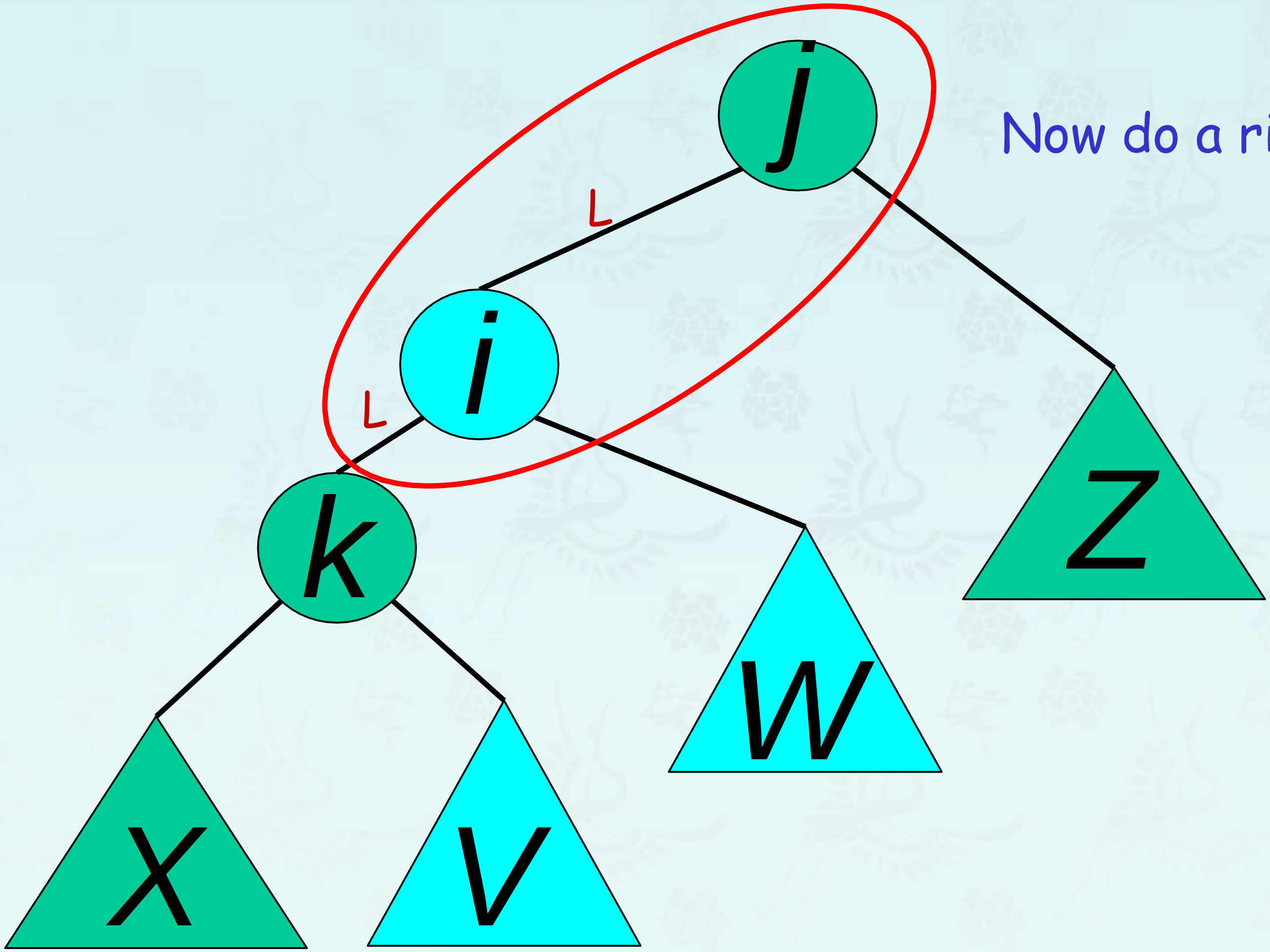
Double rotation : first rotation

left rotation complete



LR Case

Double rotation : second rotation



Now do a right rotation

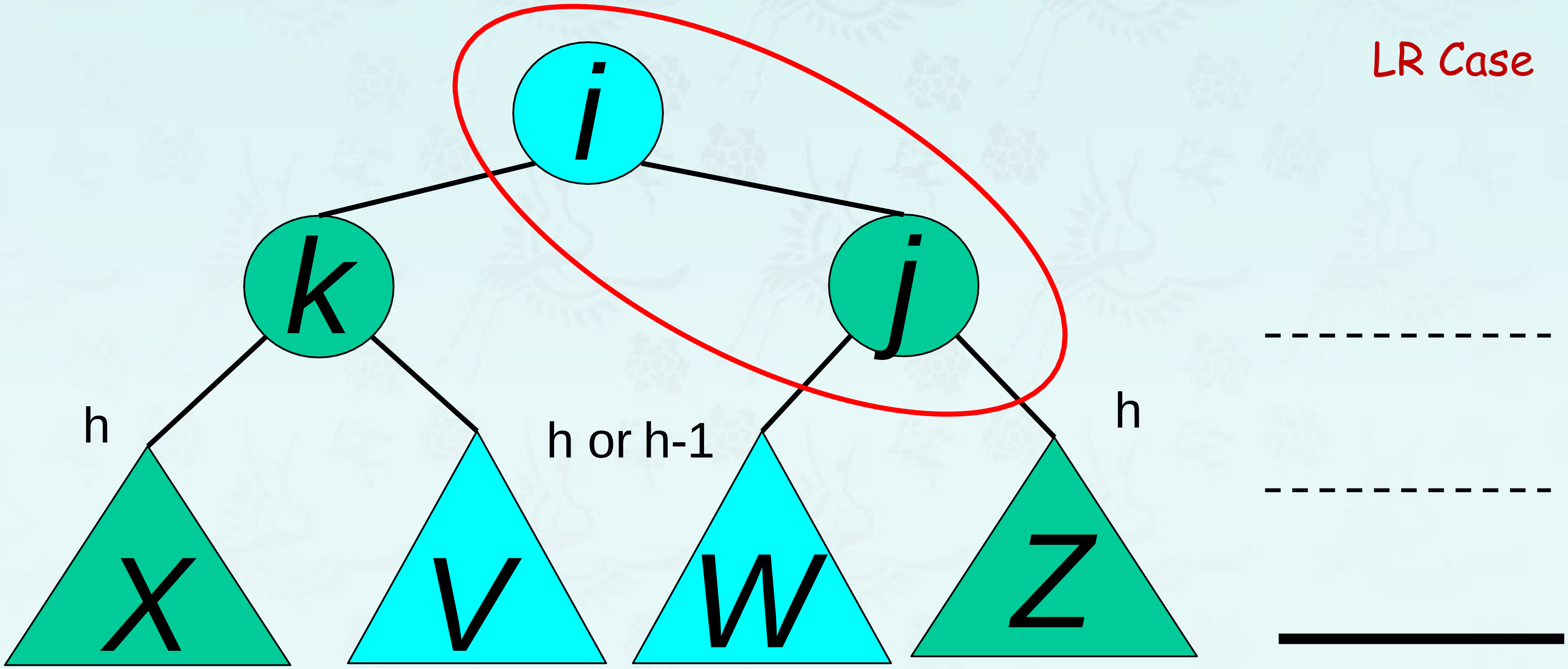
LR Case

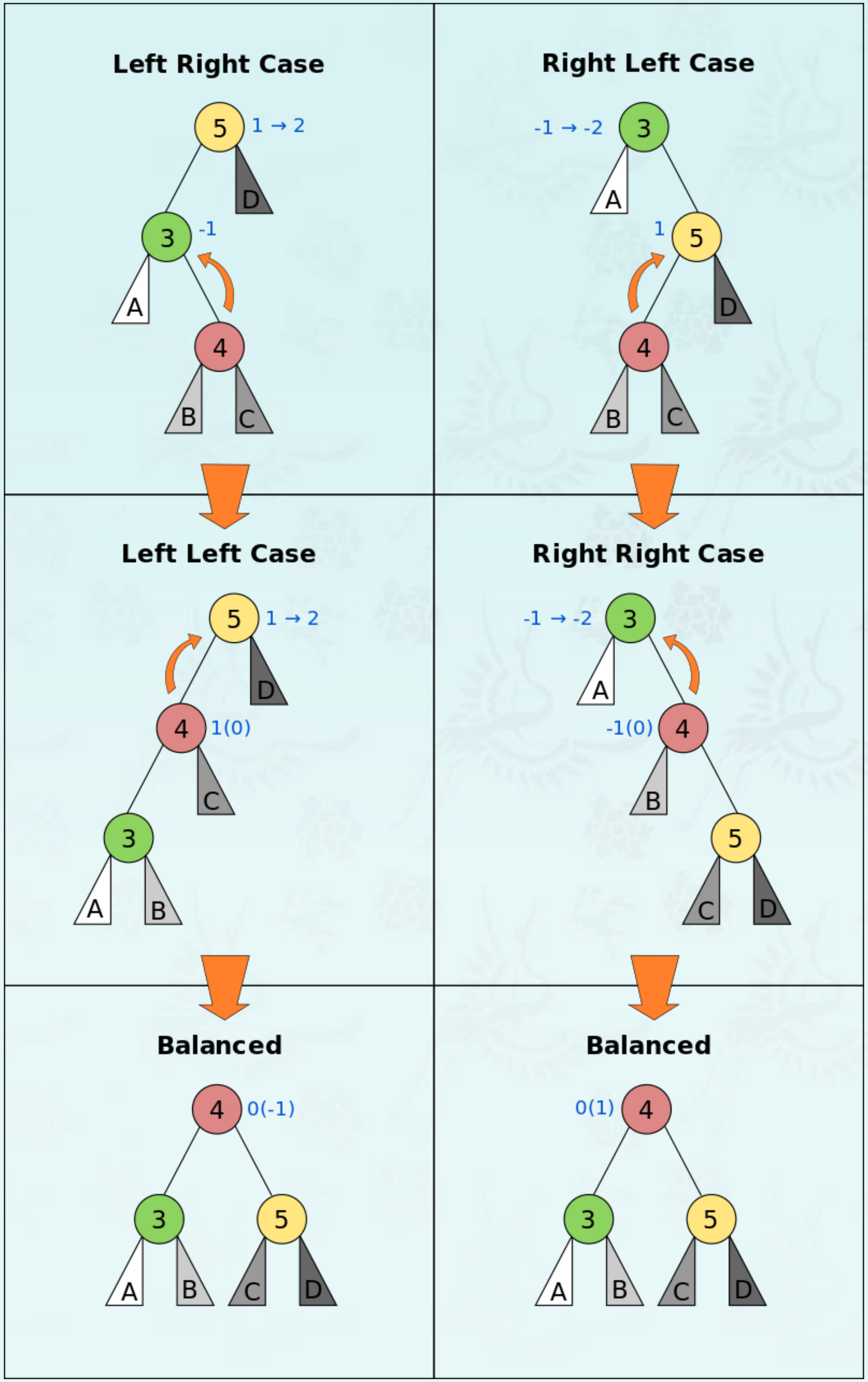
Double rotation : second rotation

right rotation complete

Balance has been restored

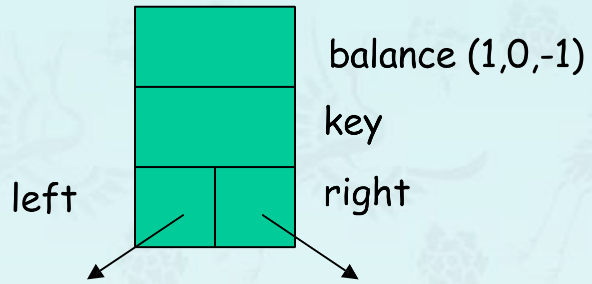
LR Case





- The numbered circles represent the nodes being rebalanced.
- The lettered triangles represent subtrees which are themselves balanced AVL trees.
- A blue number next to a node denotes possible balance factors
- (those in parentheses occurring only in case of deletion).
- Source: www.wikipedia.com

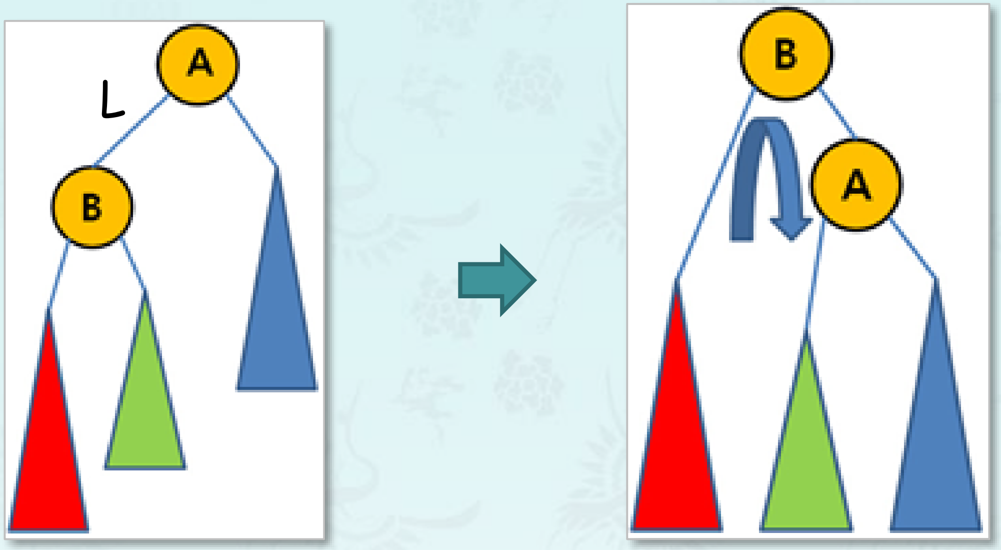
Implementation



- You can either keep the height or 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
- You may compute the balance factor on the fly after the insert is done during the recursion.

Single Rotation - LL case

outside case

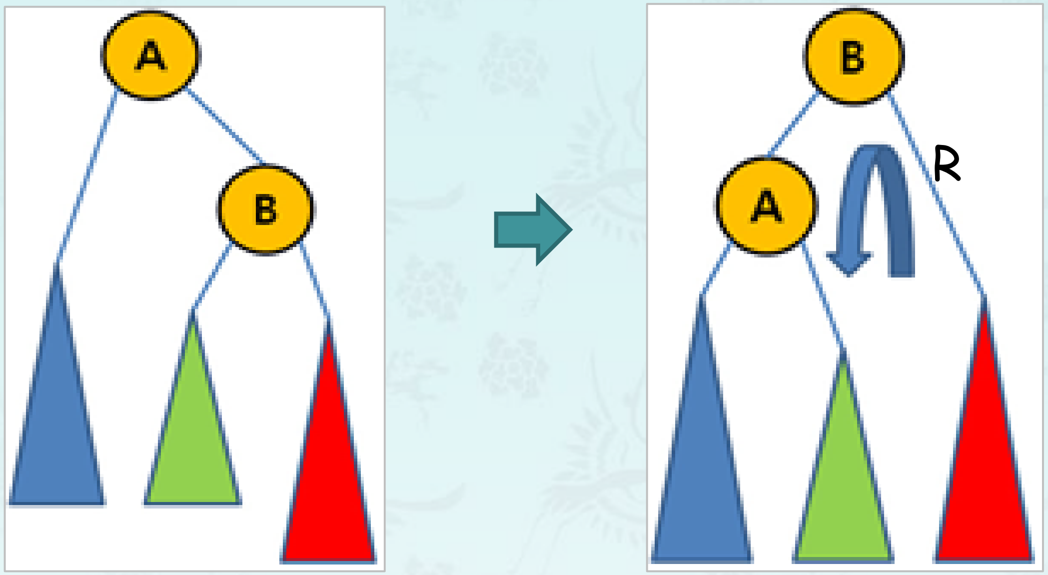



```
node rotateLL(node A)
{
  node B    =
  A->left   =
  B->right  =
  return
}
```

Why?

Single Rotation - RR case

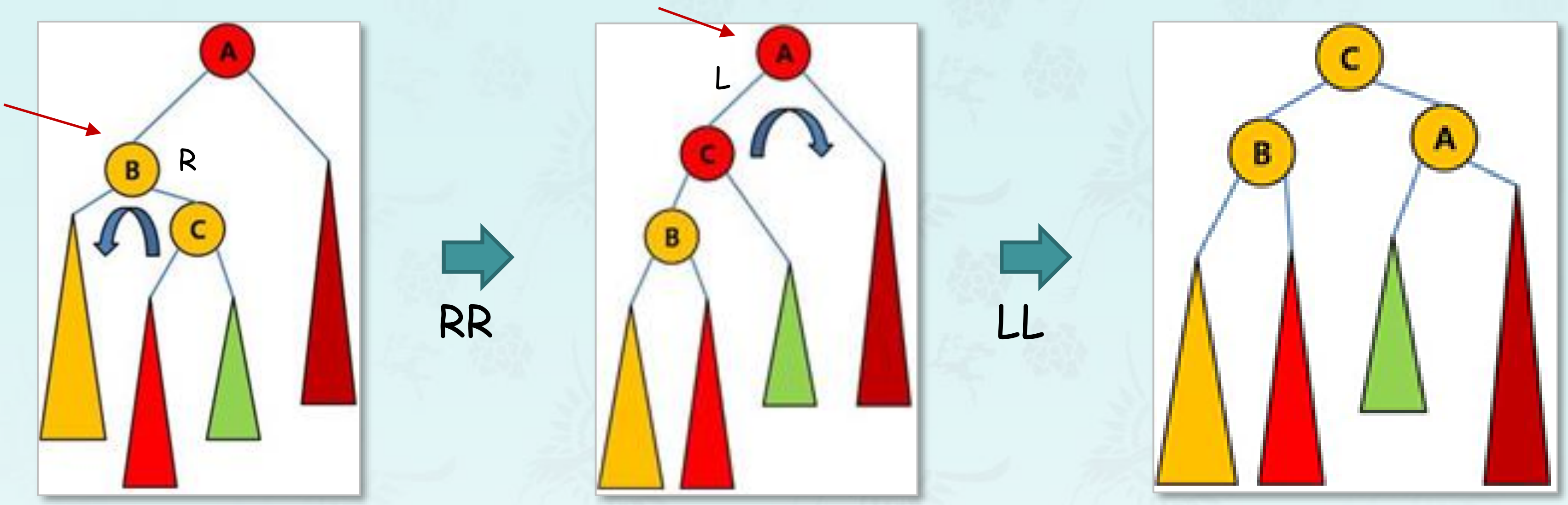
outside case




```
node rotateRR(node A)
{
  node B    =
  A->right =
  B->left  =
  return
}
```

Double Rotation - LR

inside case

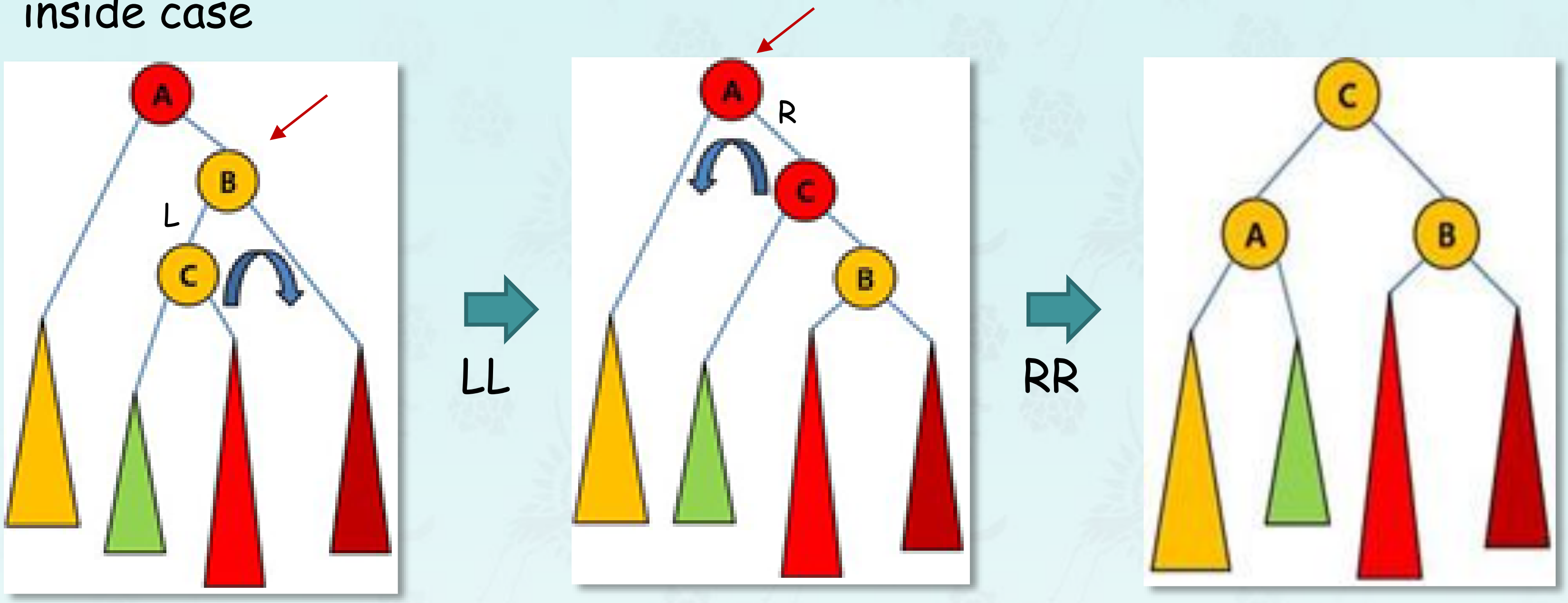



```
node rotateLR(node A) // RR and LL
{
  node B  =
  A->left = rotateRR(  );
  return rotateLL(  );
}
```

What will return eventually?

Double Rotation - RL

inside case



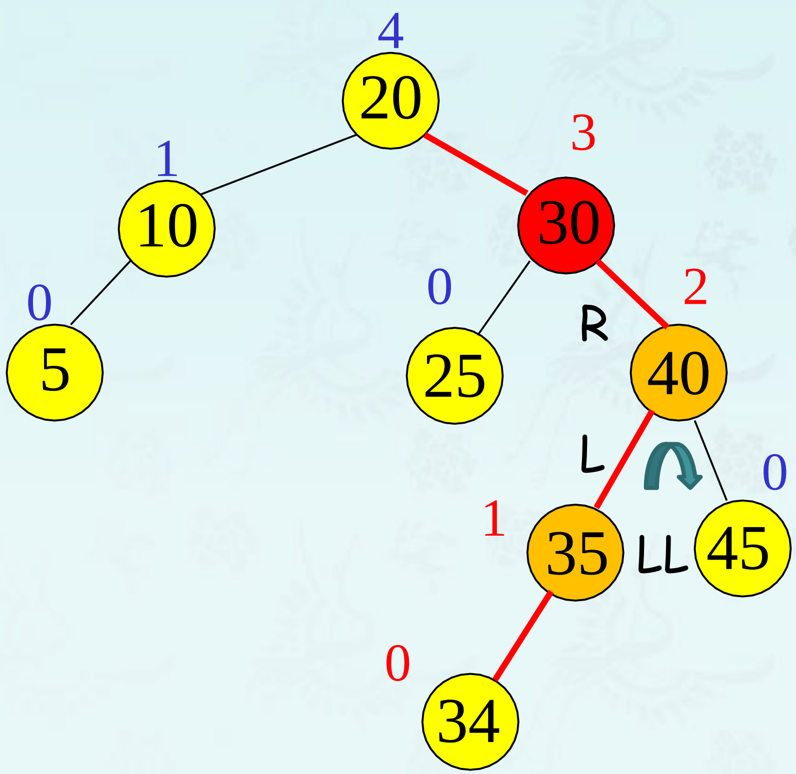

```
node rotateRL(node A) { // LL and RR
{
   node B   = A->right;
   A->right = rotateLL(  );
   return rotateRR(  );
}
```

Insertion of 34

Imbalance at 30

Balance factor at 30 = -2

Double rotation RL

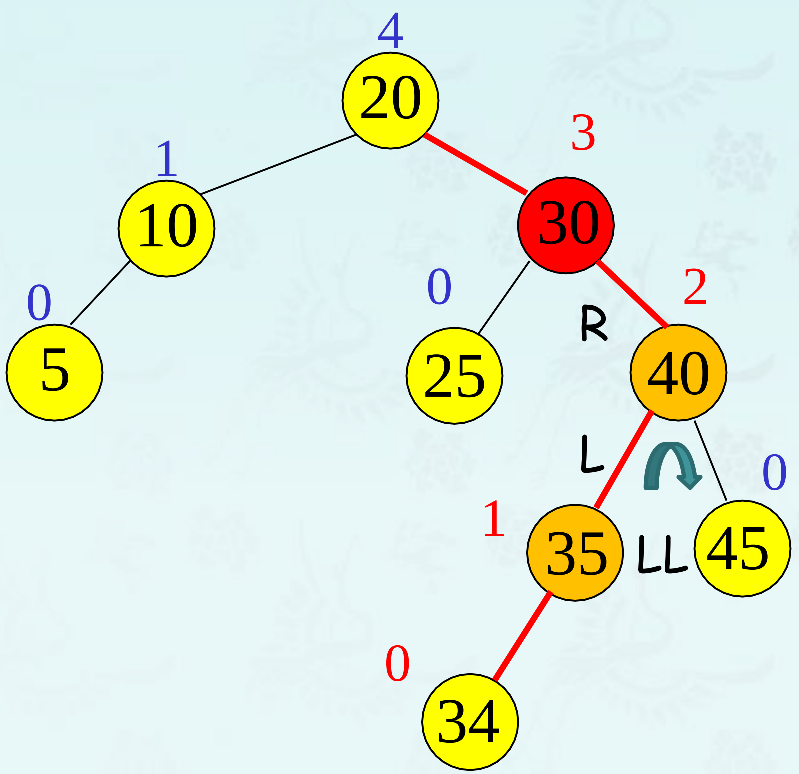


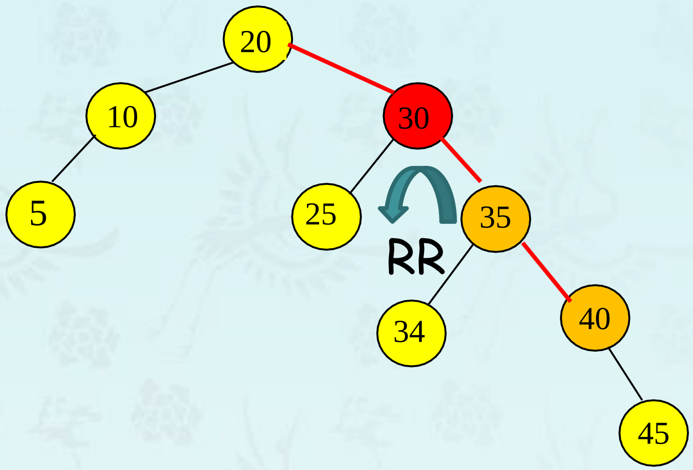
Insertion of 34

Imbalance at 30

Double rotation RL

Balance factor at 30 = -2



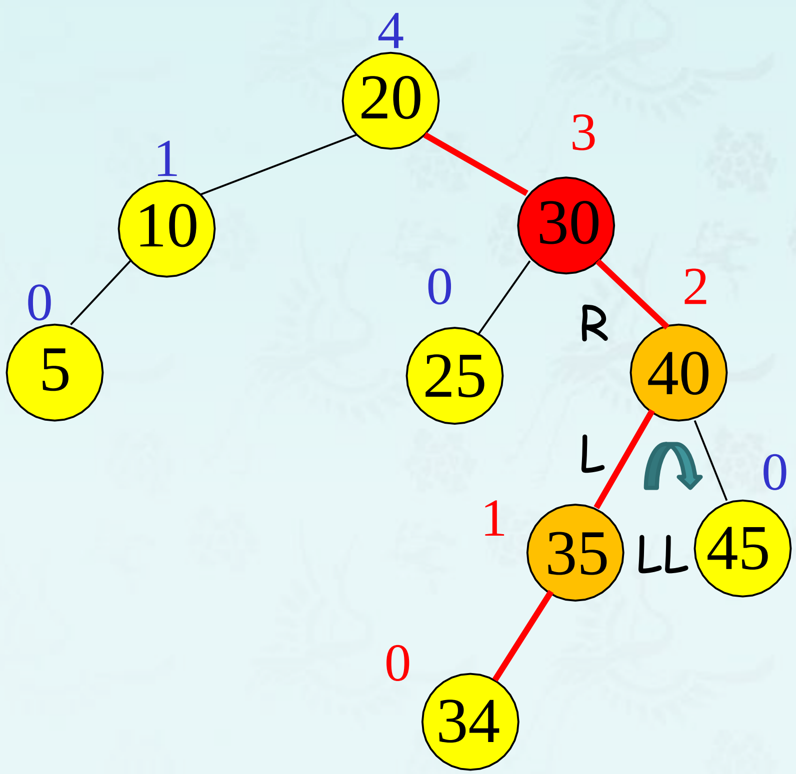


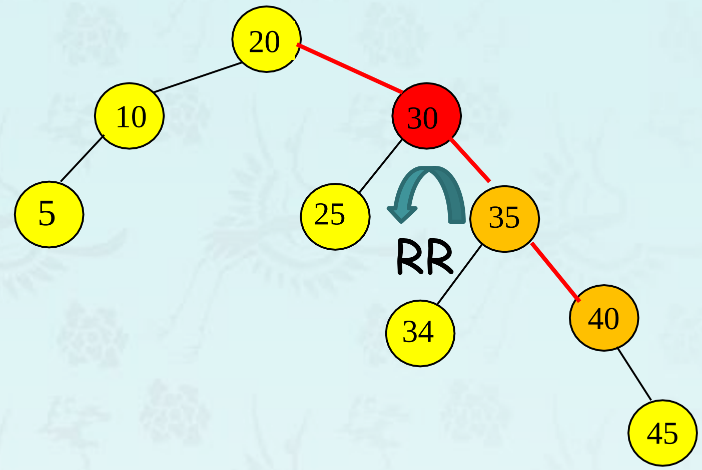
Insertion of 34

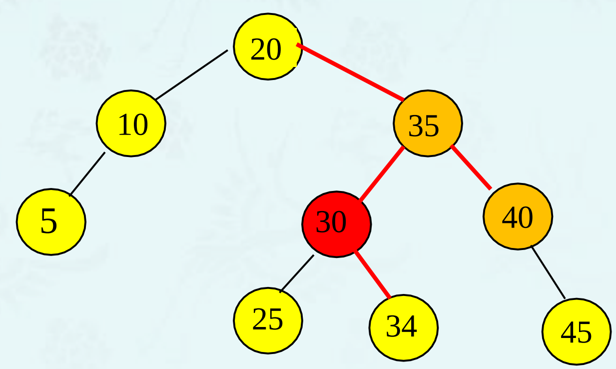
Imbalance at 30

Balance factor at 30 = -2

Double rotation RL







Balance Factor and Height

```
int getHeight(node node) {
  if (node == NULL) return 0;
  int left  = leftHeight (node->left);
  int right = rightHeight(node->right);
  return (left > right) ? left + 1 : right + 1;
}
```

```
int balanceFactor(node node) {
  if (node == NULL) return 0;
  int left  = leftHeight (node->left);
  int right = rightHeight(node->right);
  return left - right;
}
```

Rebalance

```
node rebalance(node node)
{
  bf = balanceFactor(node);
  if (bf >= 2) {
    if (balanceFactor(node->left) >= 1) {
      node = rotateLL(node);     // LL
    else
      node = rotateLR(node);     // LR
  }
  else if (bf <= -2) {
    if (balanceFactor(node->right) <= -1)
      node = rotateRR(node);
    else
      node = rotateRL(node);
  }
  return node;
}
```

checking single or double rotation

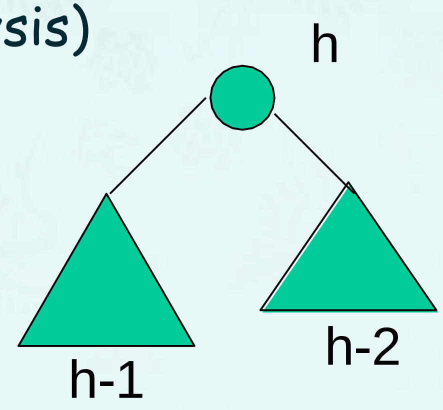
outside case

inside case

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 (compare it with Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



47

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.

- $n \geq N(h)$
- $n \geq \phi^h$ hence $log_\phi n \geq h$
  (relatively well balanced tree!!)
- $h \leq 1.44\, log_2 n$ (i.e., 'Find' operation takes O(log n))

Pros and Cons of AVL Trees

Arguments **for** AVL trees:

- Search is O(log N) since AVL trees are always balanced.
- Insertion and deletions are also O(logn)
- The height balancing adds no more than a constant factor to the speed of insertion.

Arguments **against** using AVL trees:

- Difficult to program & debug; more space for balance factor.
- Asymptotically faster but rebalancing costs time.
- Most large searches are done in database systems on disk an d use other structures (e.g. B-trees).
- May be OK to have O(N) for a single operation if total run ti me for many consecutive operations is fast (e.g. Splay trees).