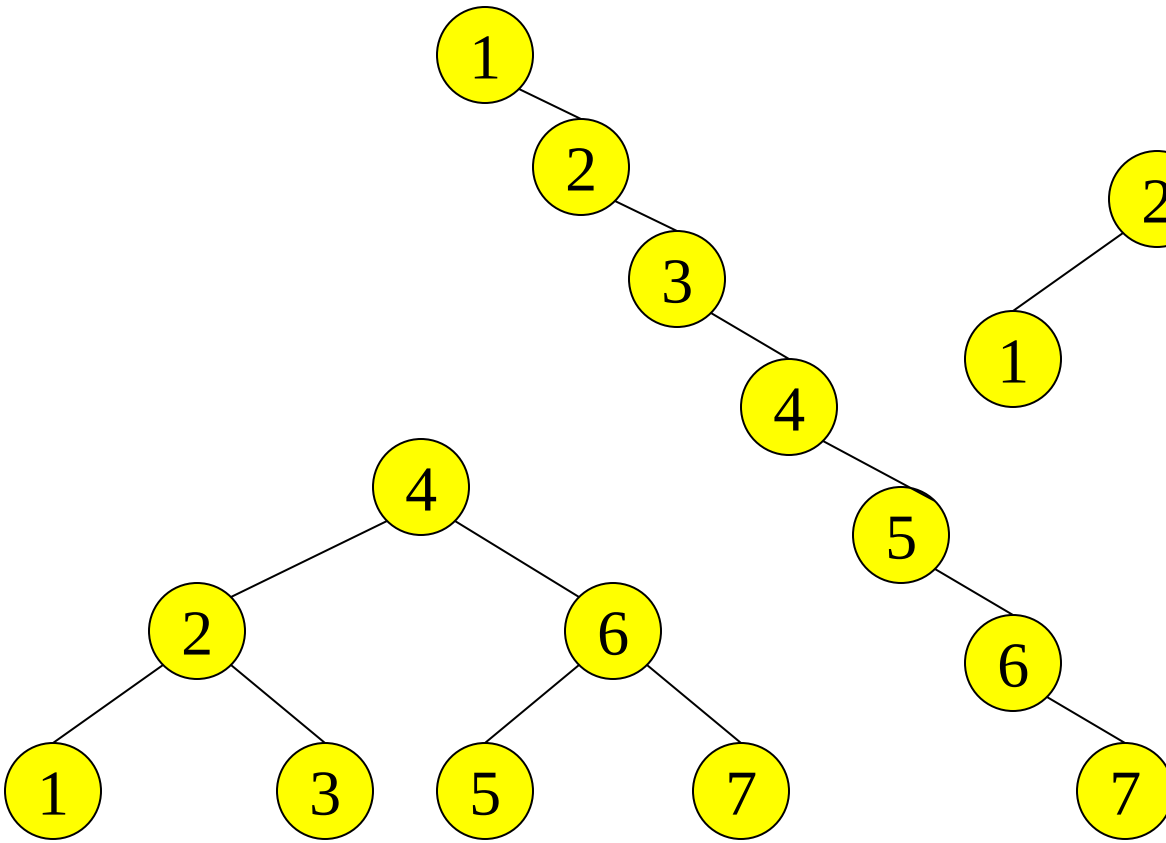
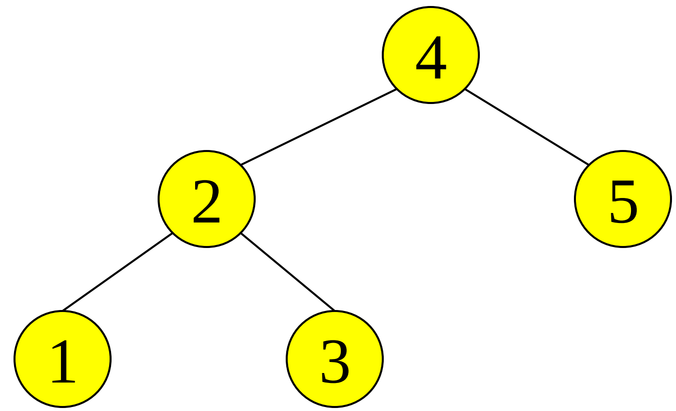
National University of Computer & Emerging Sciences

AVL Trees

Georgy **Adelson-Velsky and Landis**' tree

Balanced and unbalanced BST

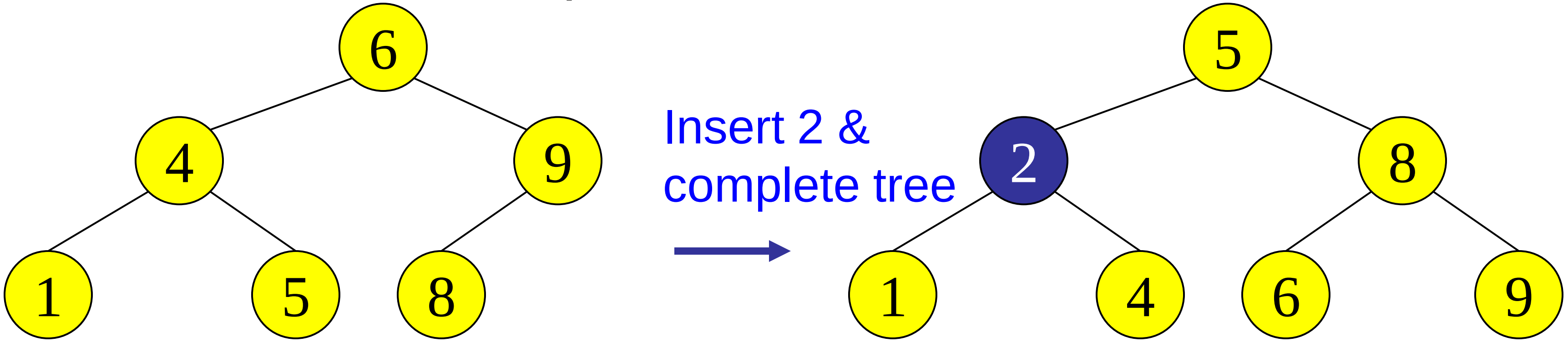




Is this “balanced”?

Perfect Balance

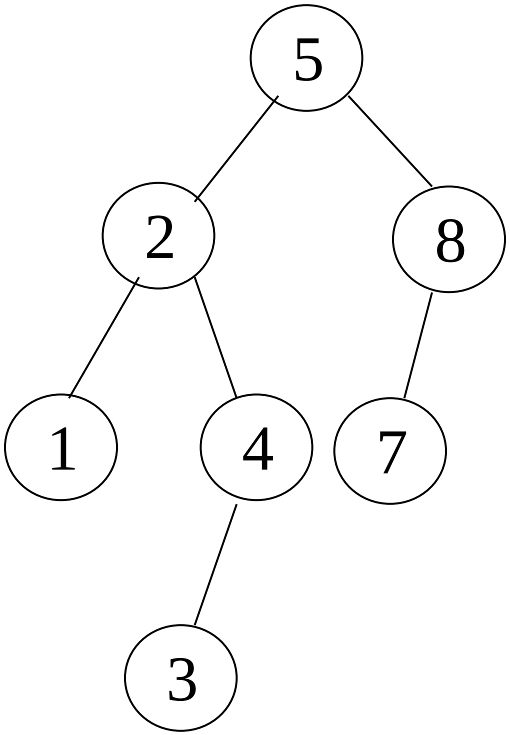
- Want a (almost) complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



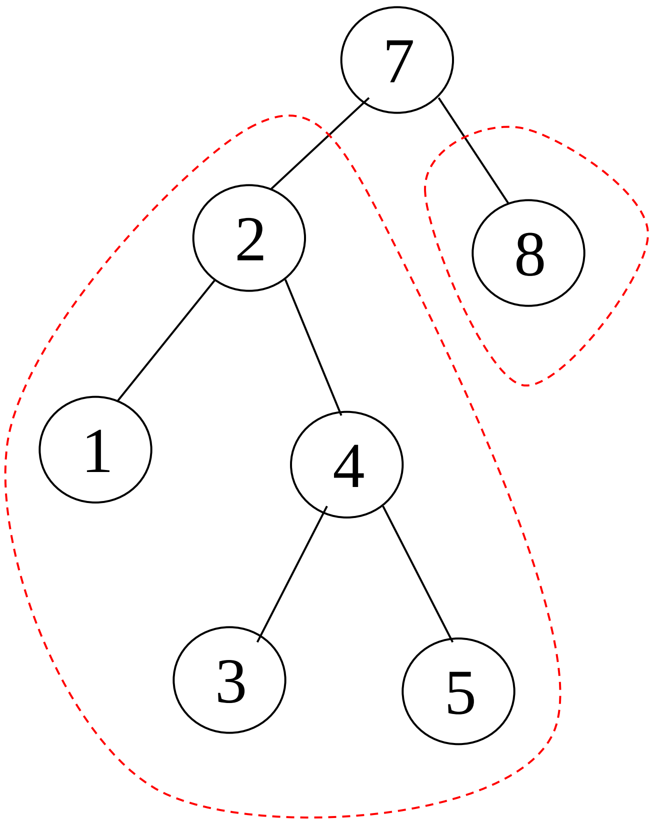
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- 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

AVL Trees



An AVL Tree



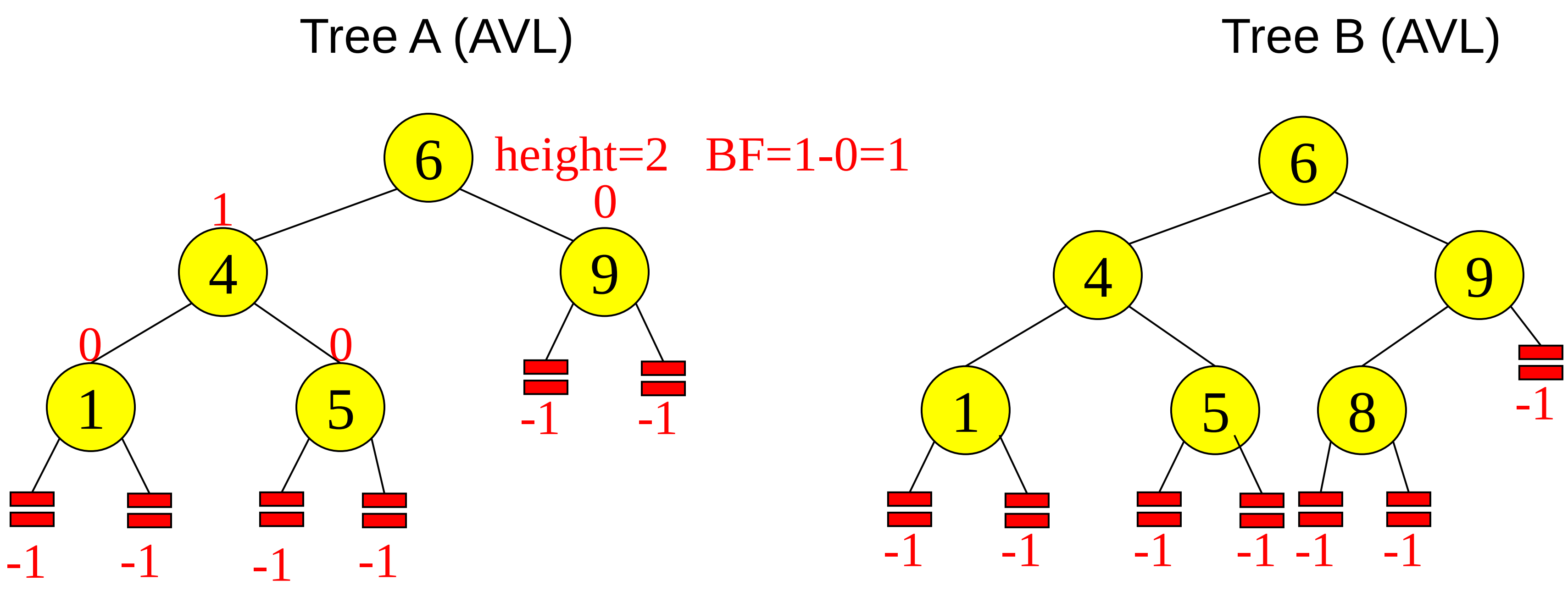
Not an AVL Tree

AVL Trees

- The height of the left subtree minus the height of the right subtree of a node is called the *balance of the node*. For an AVL tree, the balances of the nodes are always -1, 0 or 1.
  - The height of an empty tree is defined to be -1.

Node Height and Balance Factor

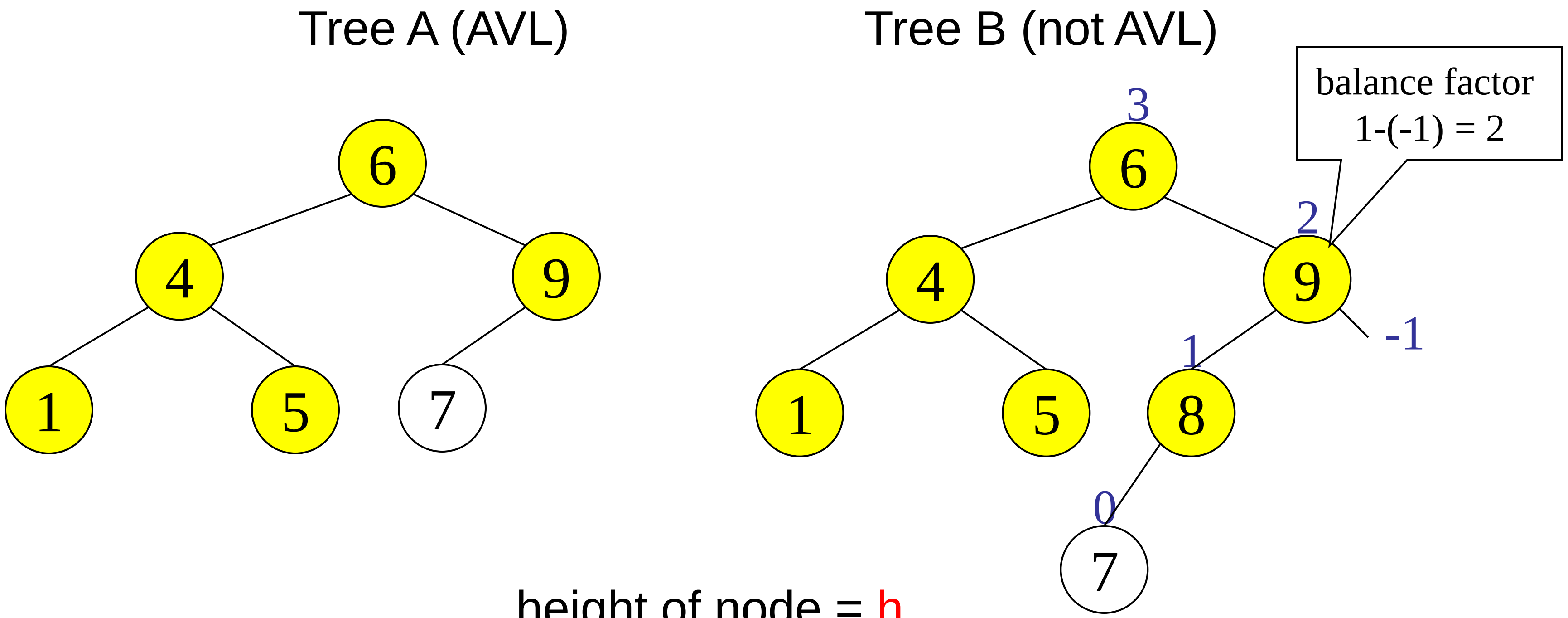
height of node = h
balance factor = $h_{left} - h_{right}$
empty height = -1



AVL Trees

- Given an AVL tree, if insertions or deletions are performed, the AVL tree *may not* remain height balanced.

Node Heights after Insert 7



height of node = h

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

empty height = -1

AVL Trees

To maintain the height balanced property of the AVL tree after insertion or deletion, it is necessary to perform a *transformation* on the tree so that

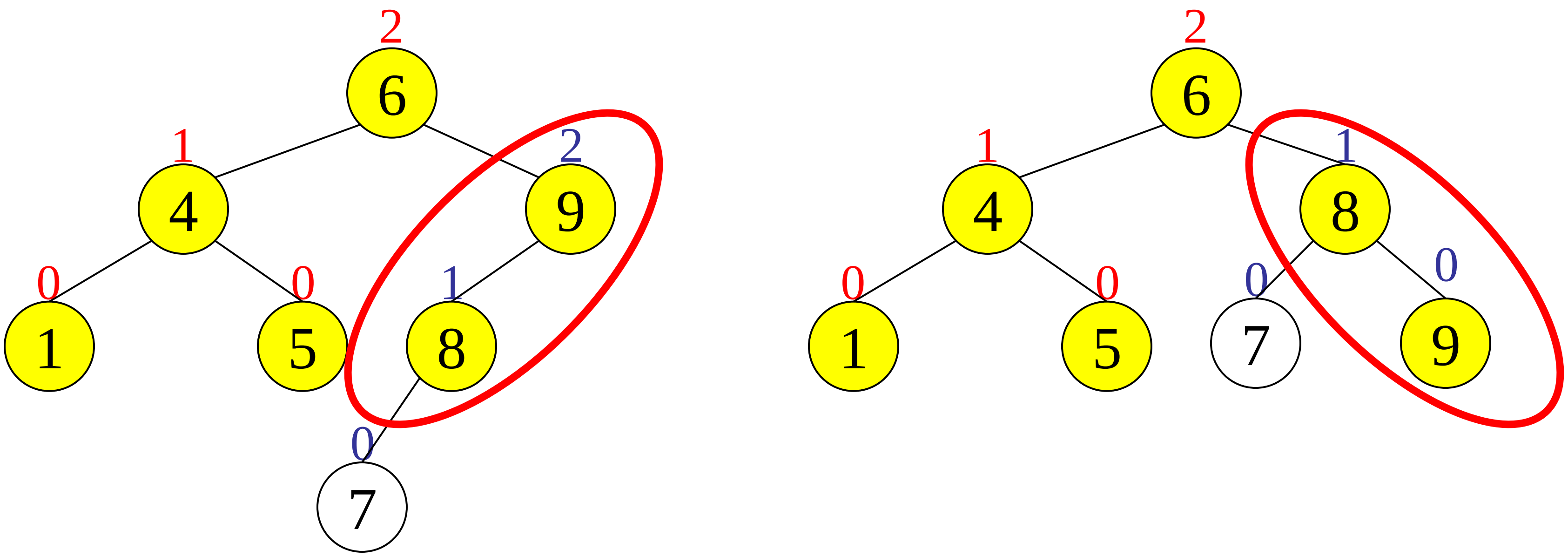
1) the in-order traversal of the transformed tree is the same as for the original tree (i.e., the new tree remains a binary search tree).

2) the tree after transformation is height balanced.

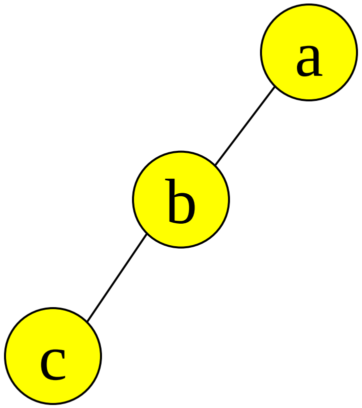
Insert and Rotation in AVL Trees

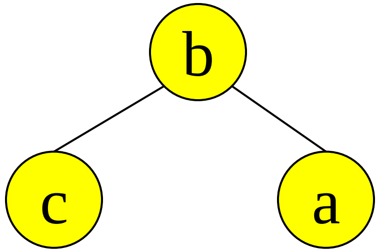
- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - Follow the path up to the root, find the first node (i.e., deepest) whose new balance violates the AVL condition. Call this node *a*
  - If a's 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



Right Rotation (RR) in an AVL Tree



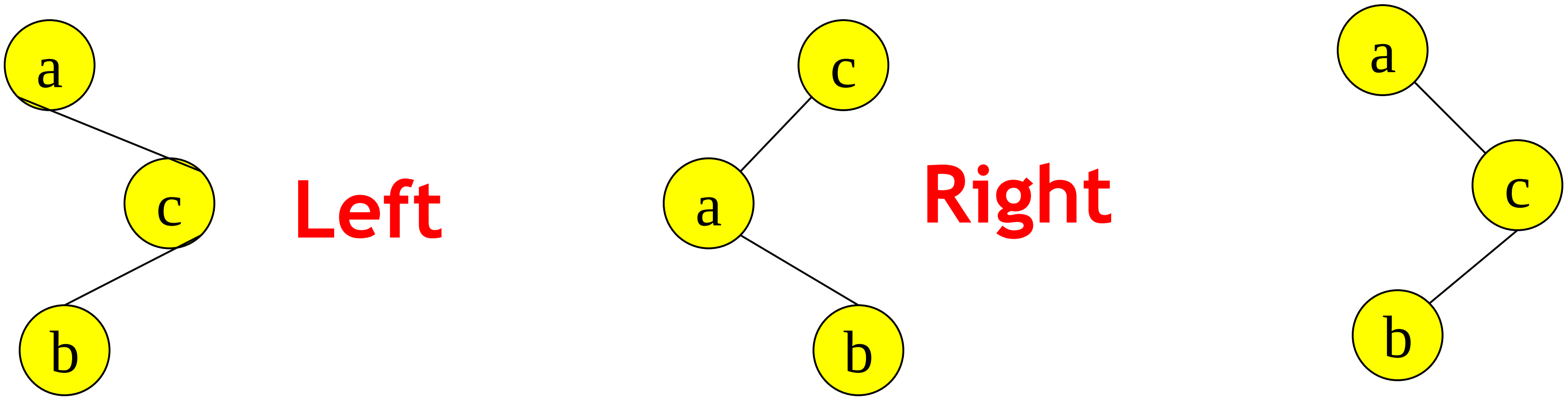


- b becomes the new root.
- a takes ownership of b's right child, as its left child, or in this case, null.
- b takes ownership of a, as it's right child.

Left Rotation (LL) in an AVL Tree

- b becomes the new root.
- a takes ownership of b's left child as its right child, or in this case, null.
- b takes ownership of a as its left child.

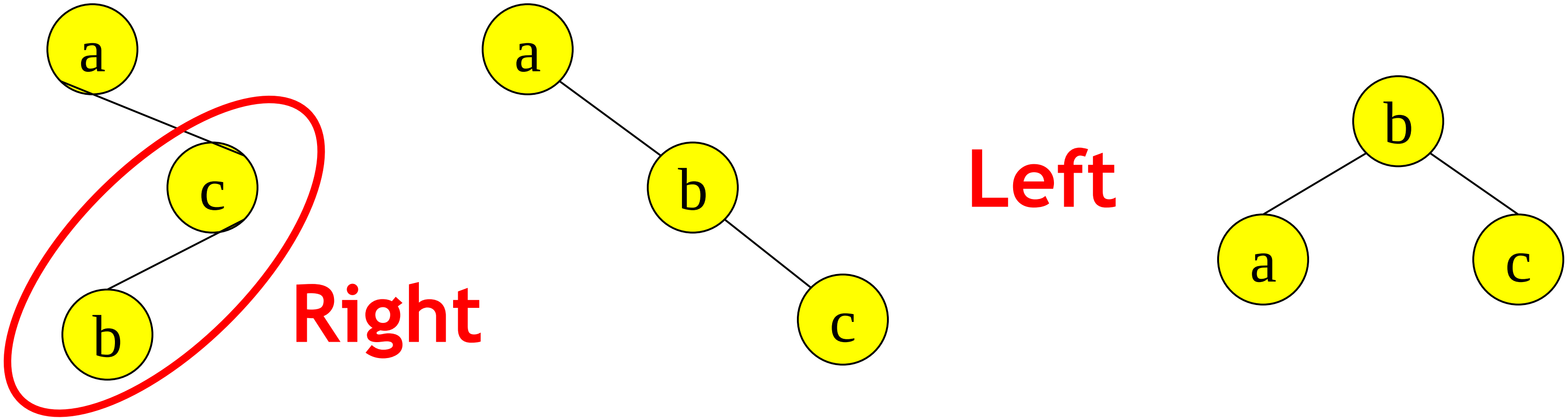
Single Rotation may be Insufficient



- *c becomes the new root.*
- *a takes ownership of c's left child as its right child, in this case, b.*
- *c takes ownership of a as its left child.*

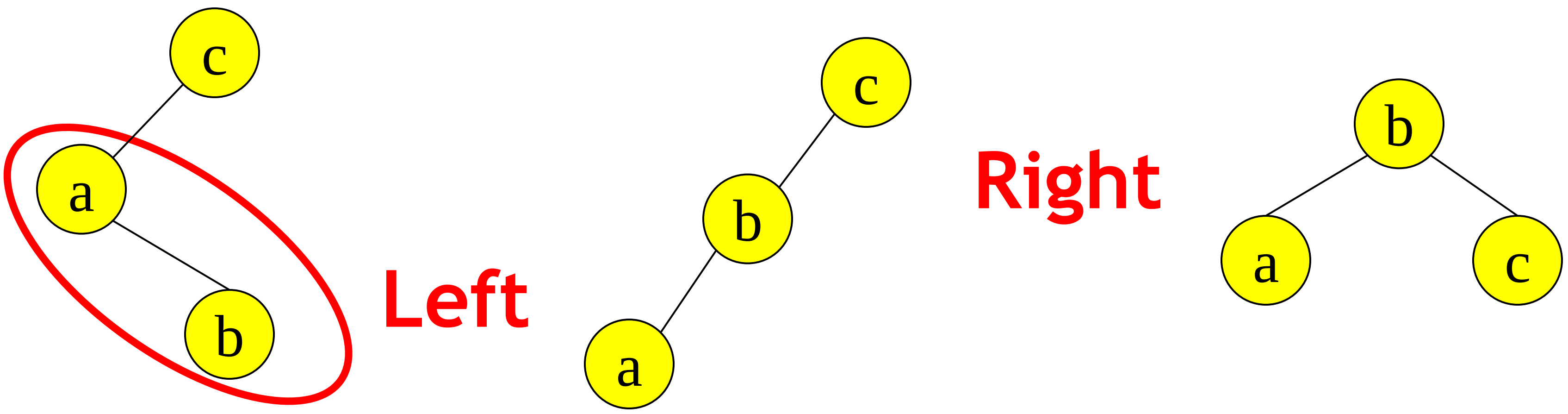
- *a becomes the new root.*
- *c takes ownership of a's right child as its left child, b.*
- *a takes ownership of c as its right child.*

Left-Right Rotation (LR) or "Double left"



- *perform a right rotation on the right subtree.*
- *b becomes the new root.*
- *a takes ownership of b's left child as its right child, in this case null.*
- *b takes ownership of a as its left child.*

Right-Left Rotation (RL) or "Double right"



- *perform a left rotation on the left subtree.*
- *b becomes the new root.*
- *c takes ownership of b's right child as its left child, in this case null.*
- *b takes ownership of c as its right child.*

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
  - Follow the path up to the root, find the first node (i.e., deepest) whose new balance violates the AVL condition. Call this node *a*
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

How and when to rotate?

Let the node that needs rebalancing be *a*.
In general, violation may occur for following 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of *a* (RR).
2. Insertion into right subtree of right child of *a* (LL).

Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of *a* (RL).
4. Insertion into left subtree of right child of *a* (LR).

How and when to rotate?

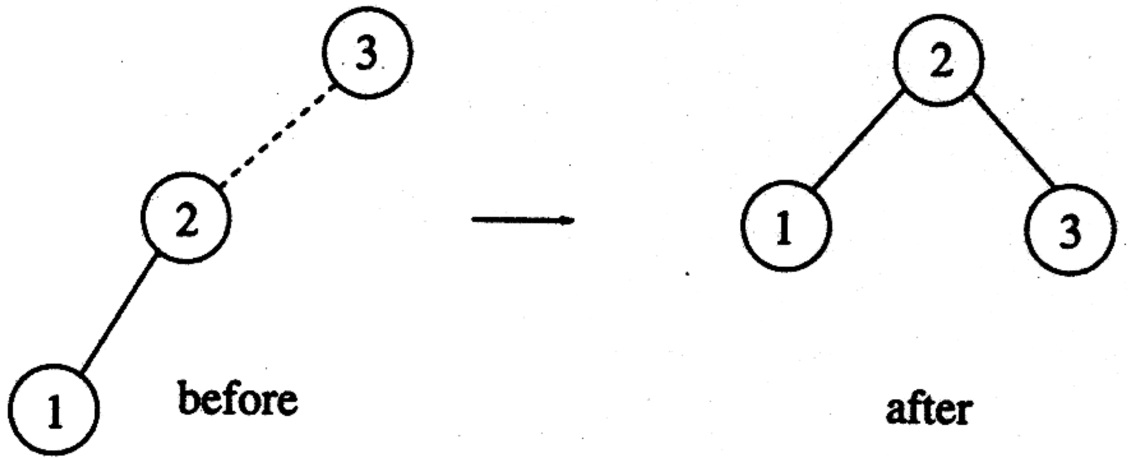
```
IF tree is right heavy
{
  IF tree's right subtree is left heavy
  {
     Perform Double Left rotation
  }
ELSE {
     Perform Single Left rotation
  }
}
…
```

```
…
ELSE IF tree is left heavy
{
  IF tree's left subtree is right heavy
  {
     Perform Double Right rotation
  }
ELSE {
     Perform Single Right rotation
  }
}
```

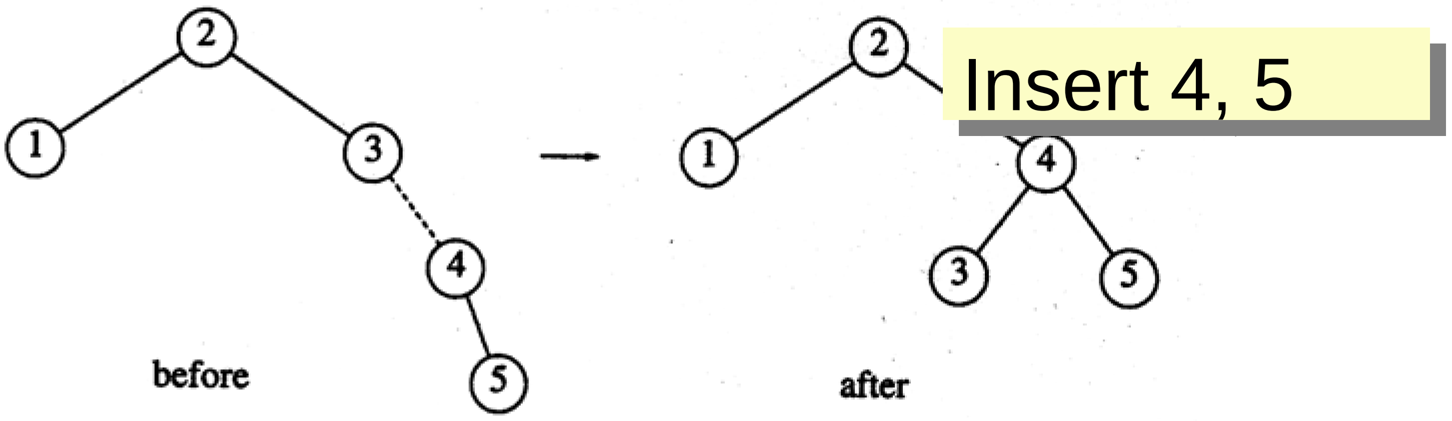
- Example: 3 2 1 4 5 6 7
- construct binary search tree without height balanced restriction
- depth of tree = 4

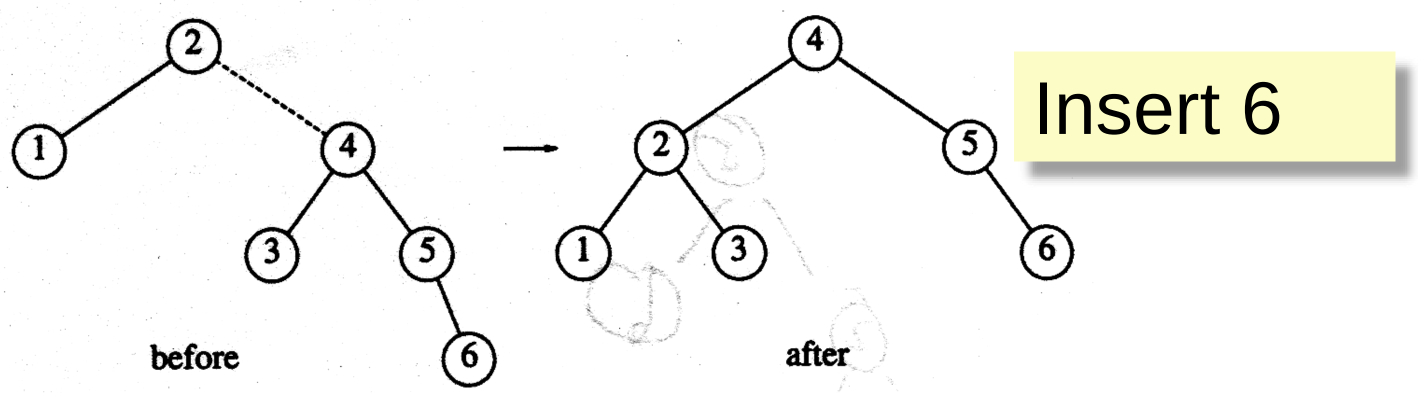
AVL Trees: Single Rotation

- Construct AVL tree (height balanced)

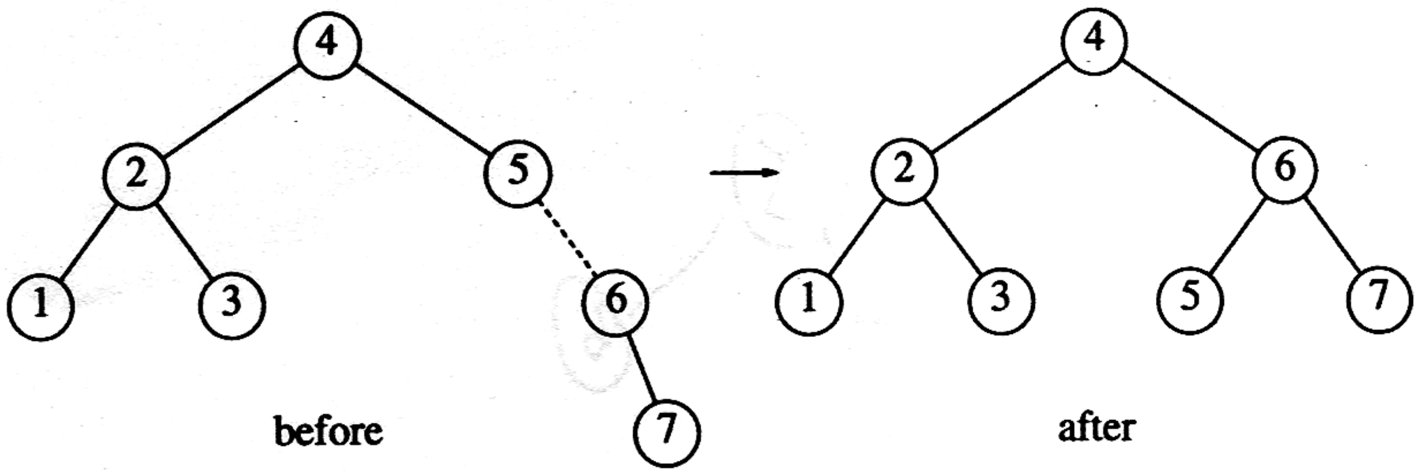


AVL Trees: Single Rotation





AVL Trees: Single Rotation



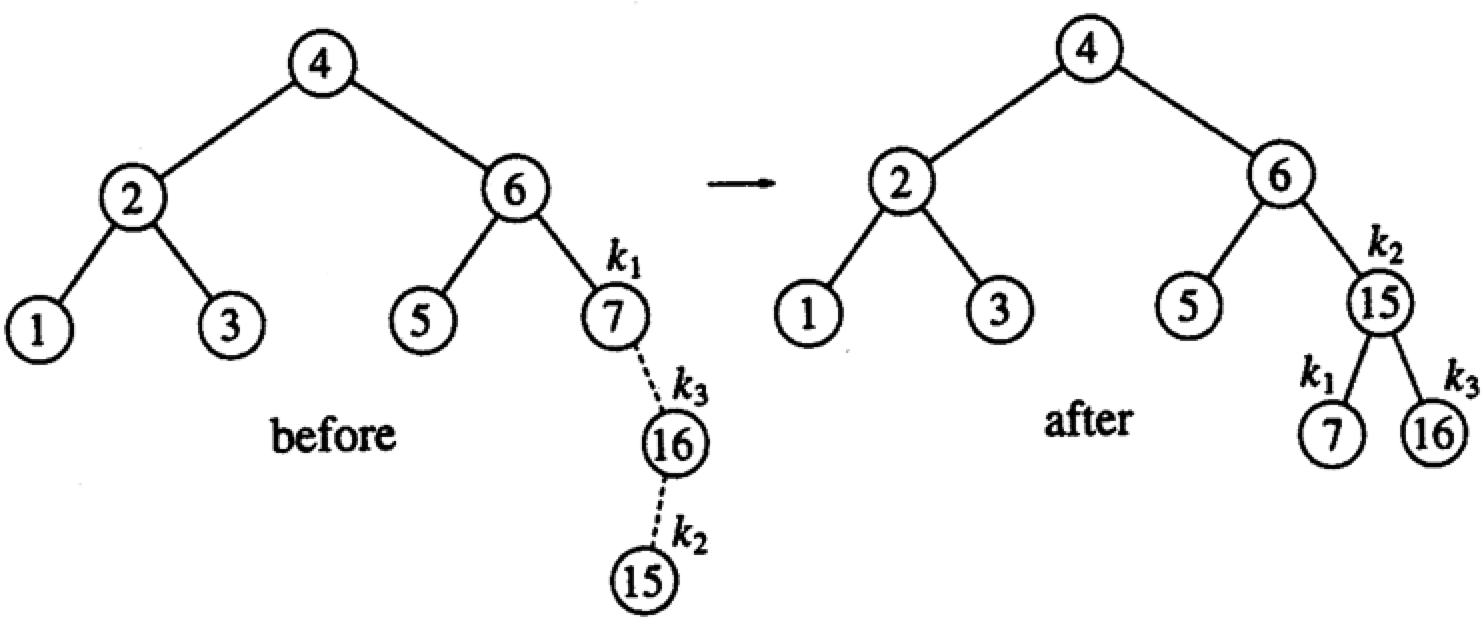
Insert 7

- So far so good, what about inserting the following numbers

  16, 15, 14, 13, 12, 11, 10, 8

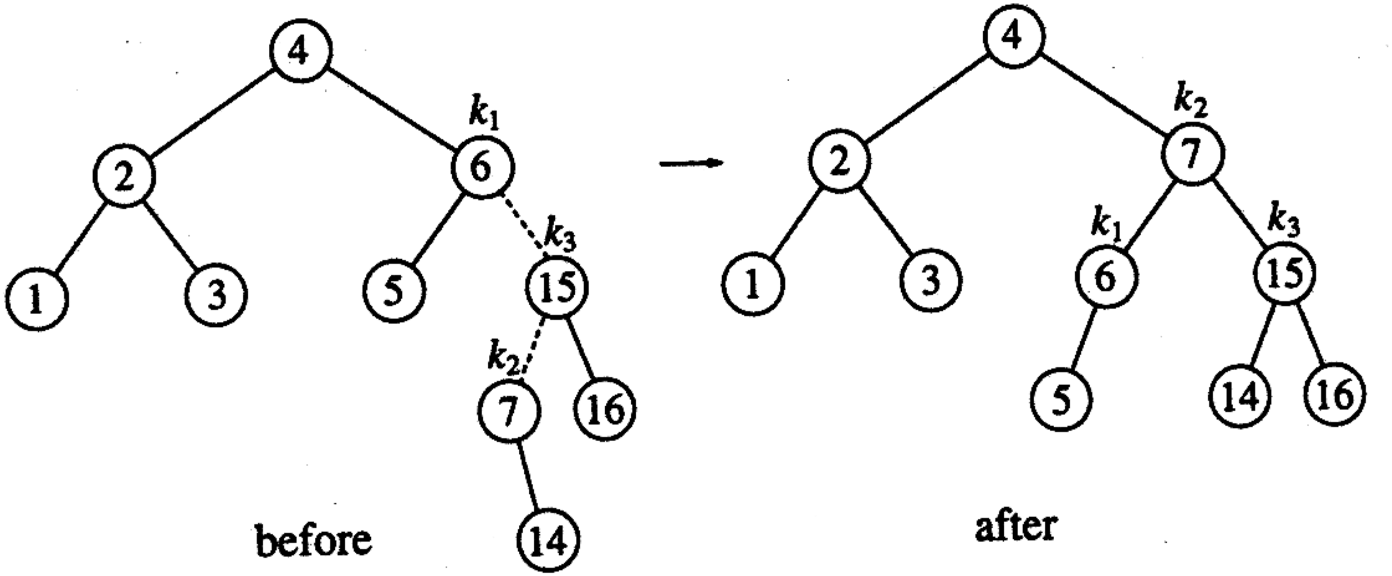
AVL Trees: Double Rotation

- Example:



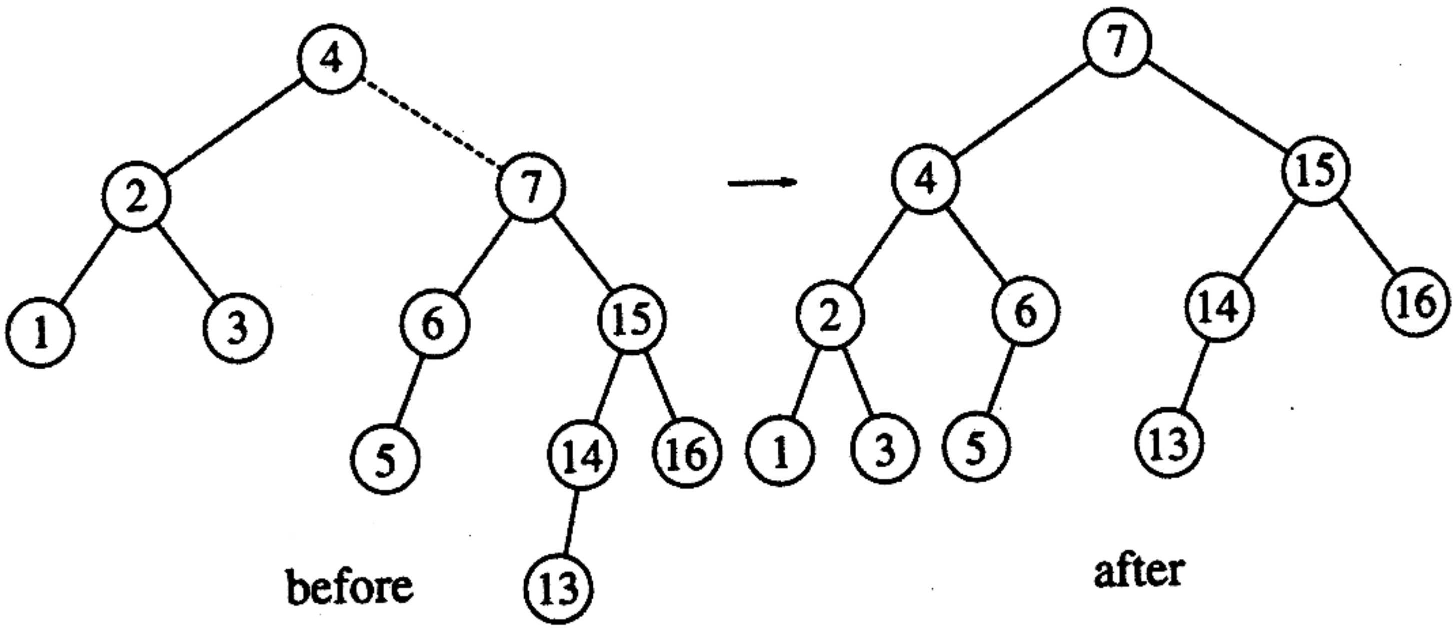
AVL Trees: Double Rotation

- Insert 14



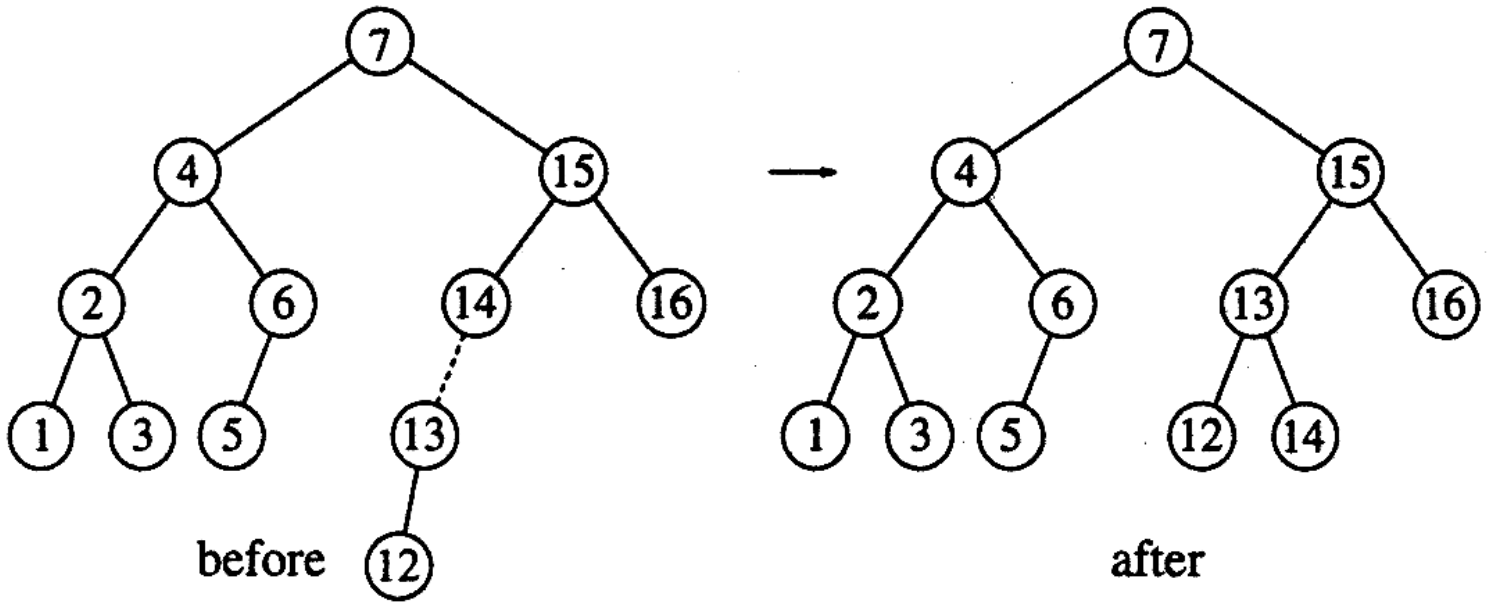
AVL Trees: Double Rotation

- Insert 13 (This is single rotation: LL Case)



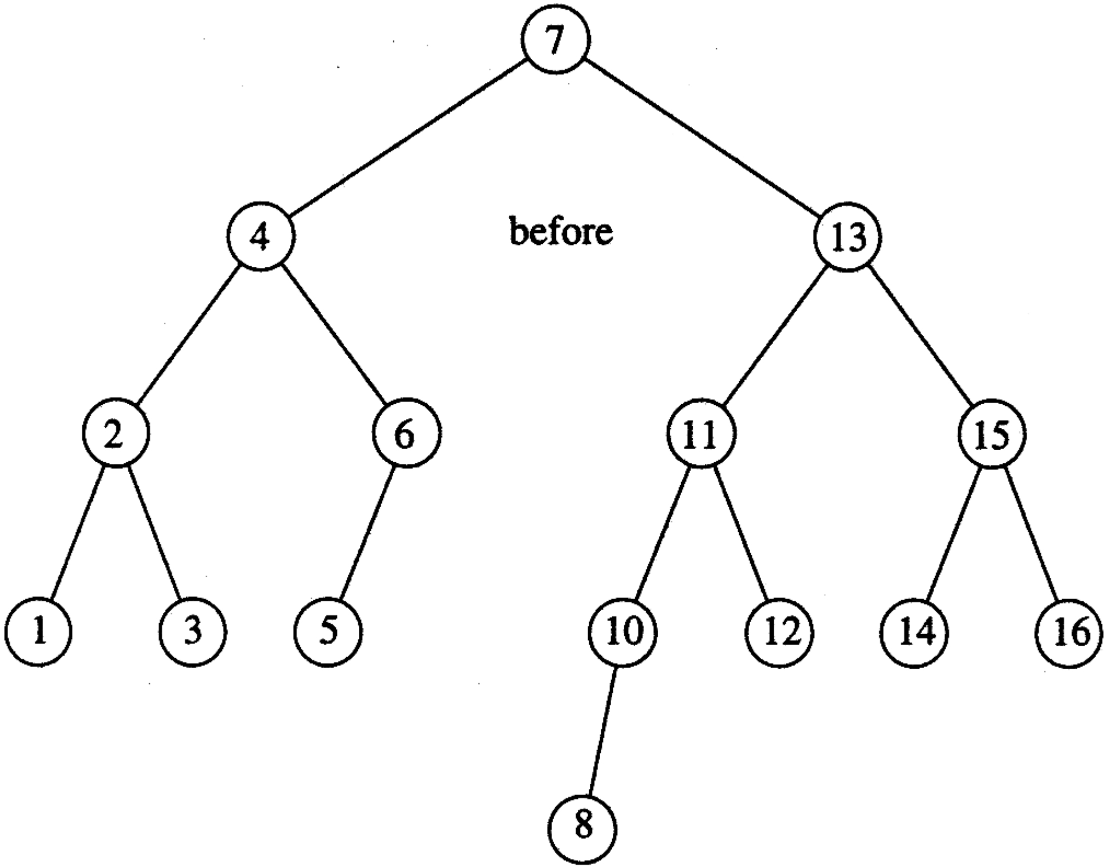
AVL Trees: Double Rotation

- Insert 12



AVL Trees: Double Rotation

- Insert 11 and 10 (single rotation), then 8



AVL Trees: Double Rotation

- Inserting 9