

Ch15: AVL Tree

305234

Algorithm Analysis and Design

Jiraporn Pooksook
Naresuan University

Review: Binary Search Tree

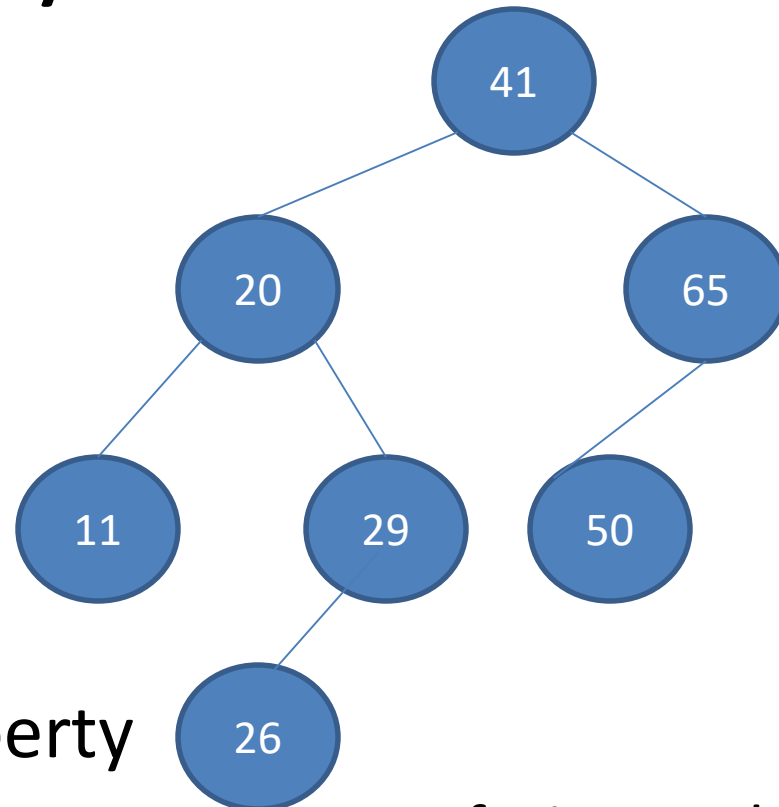
- Rooted binary tree

- Each node has

- Key
- Left pointer
- Right pointer
- Parent pointer

- Binary search tree property

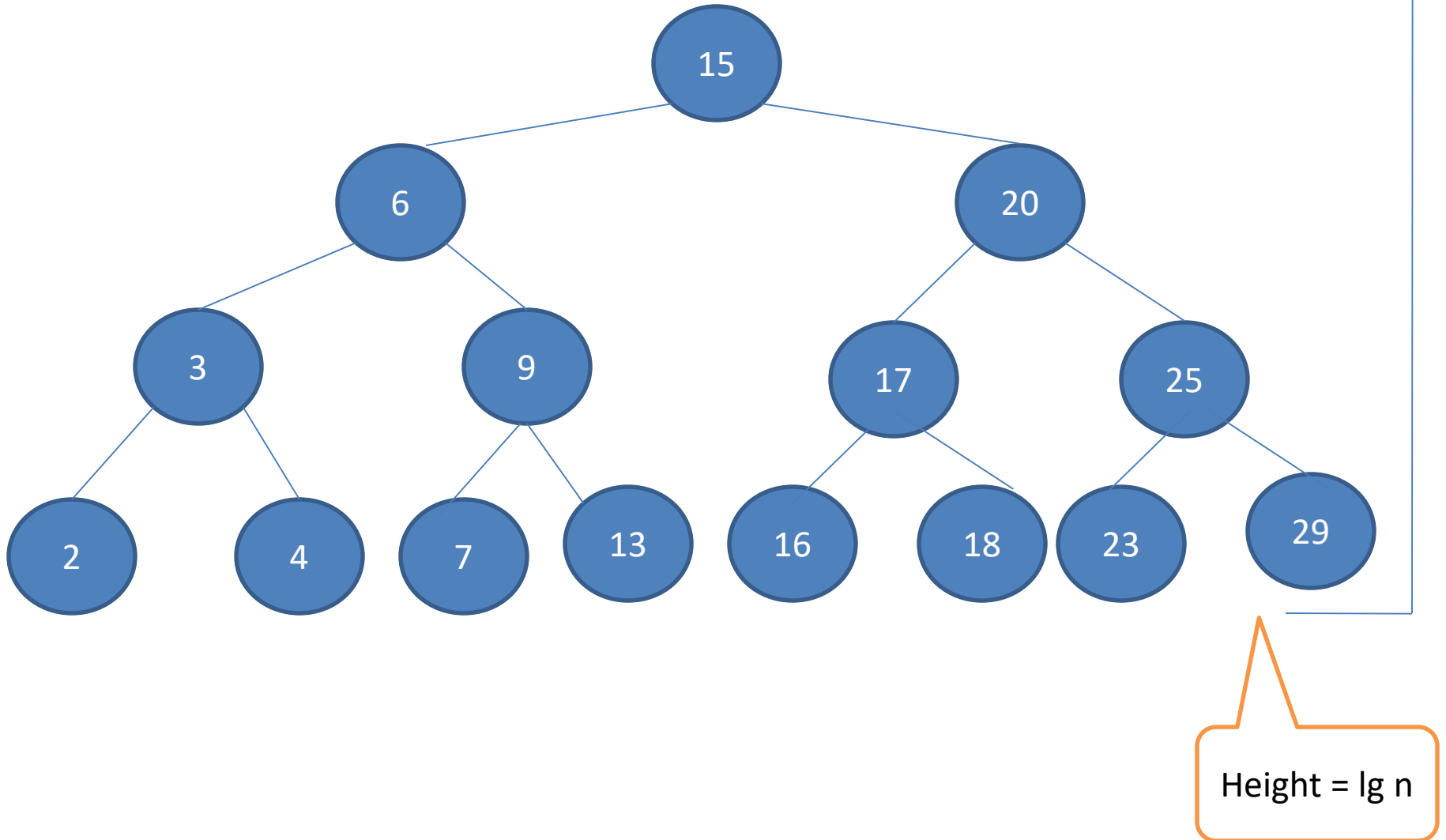
- Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $\text{key}[y] \leq \text{key}[x]$. If y is a node in the right subtree of x , then $\text{key}[x] \leq \text{key}[y]$.



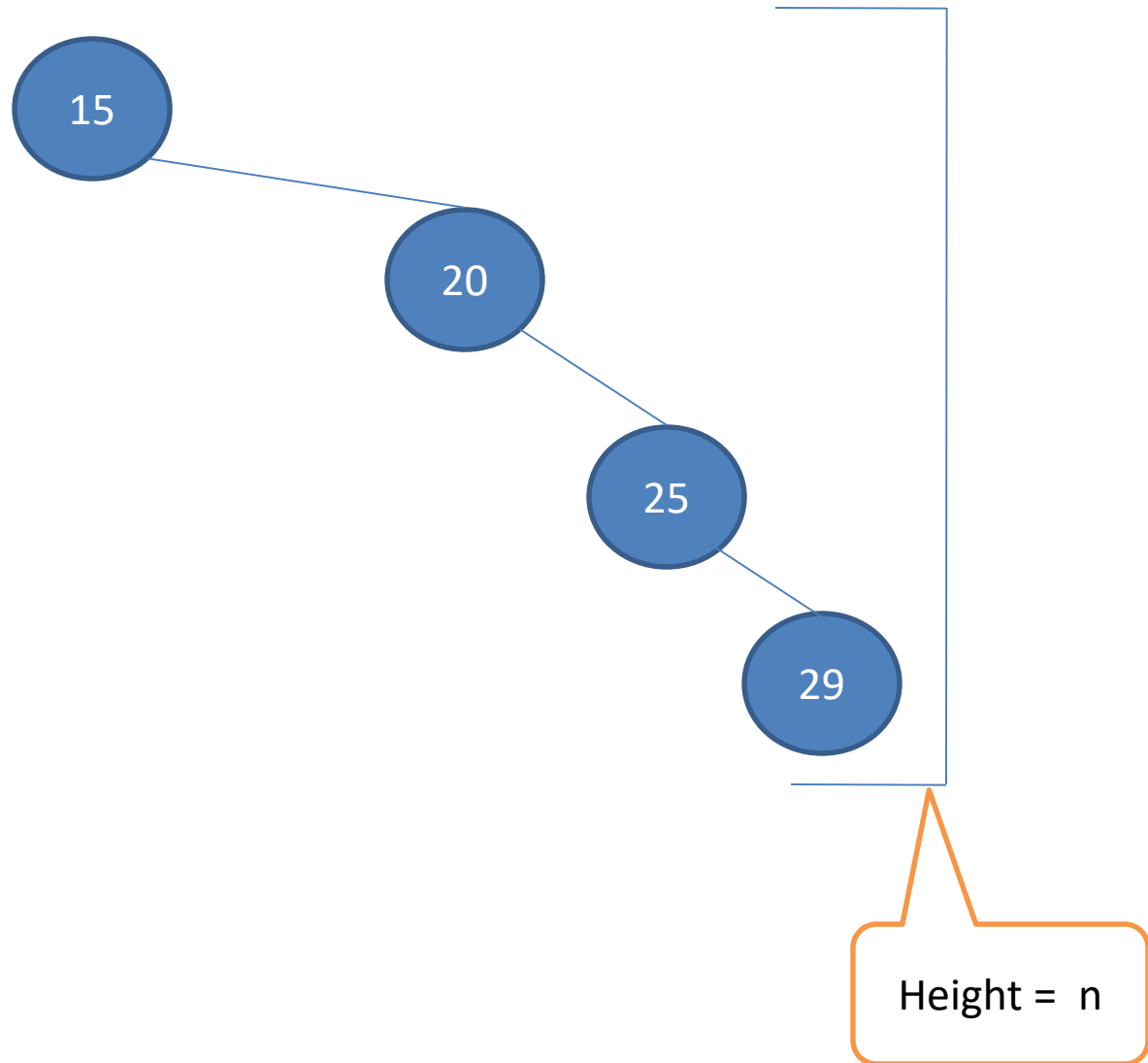
Review: Binary Search Tree

- Binary search tree supports
 - Insert
 - Delete
 - Minimum
 - Maximum
 - Successor
- In $O(h)$ time where h is the height of a binary search tree

Height Balanced



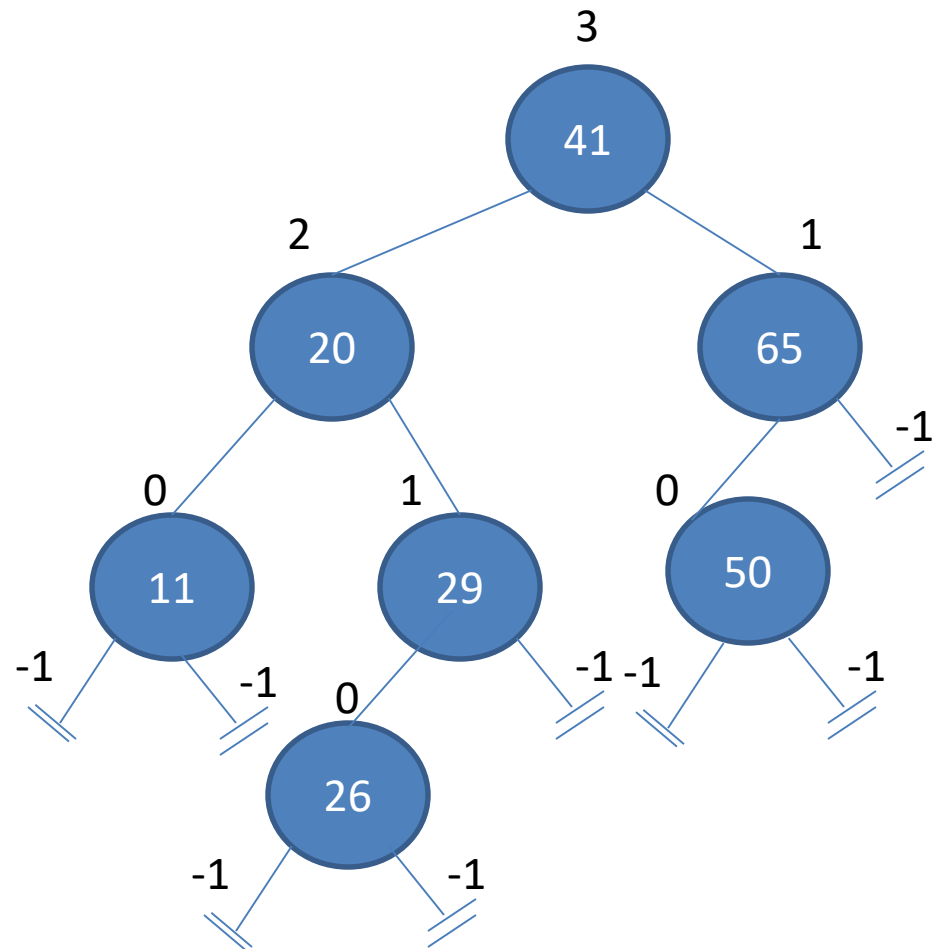
Height Unbalanced



Height

- **Height of a tree** is the length of longest path of the root down to a leaf.
- **Height of a node** is the length of longest path of it node down to a leaf.

Example: Height

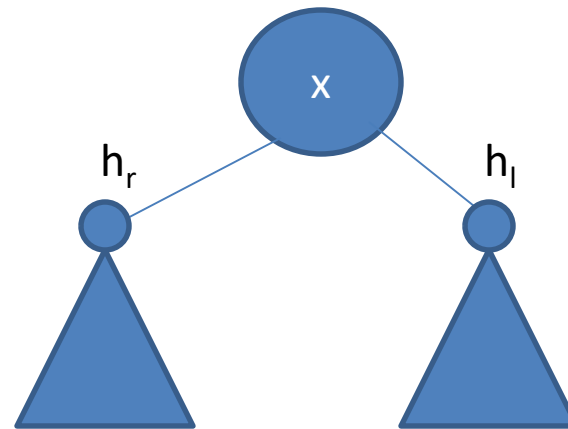


Height of node = max {height of left child , height of right child}

AVL Tree

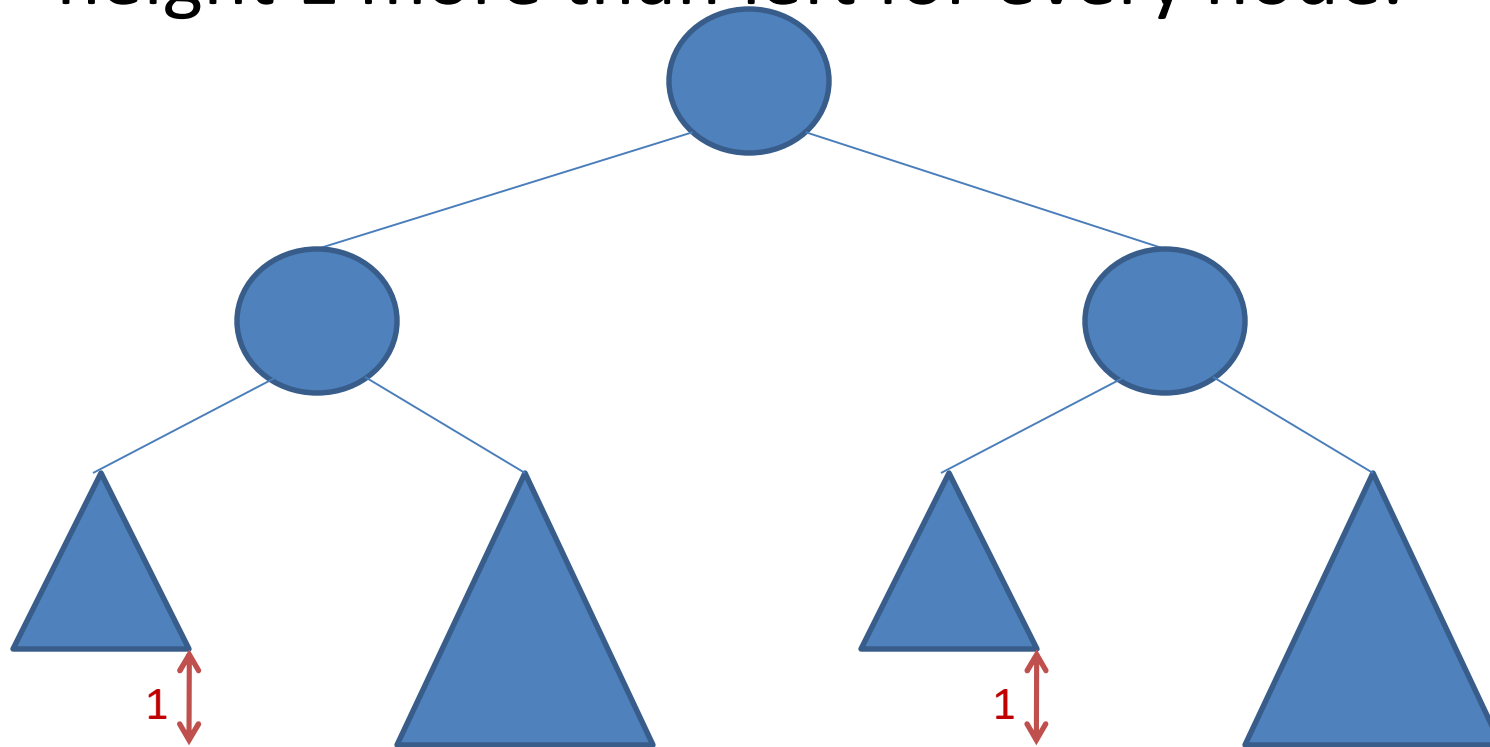
- An AVL tree is a binary search tree that is height balanced.
- For each node x , the heights of the left and right subtrees of x differ by at most 1.
- An AVL tree with n nodes has height $O(\lg n)$.

$$|h_r - h_l| \leq 1$$



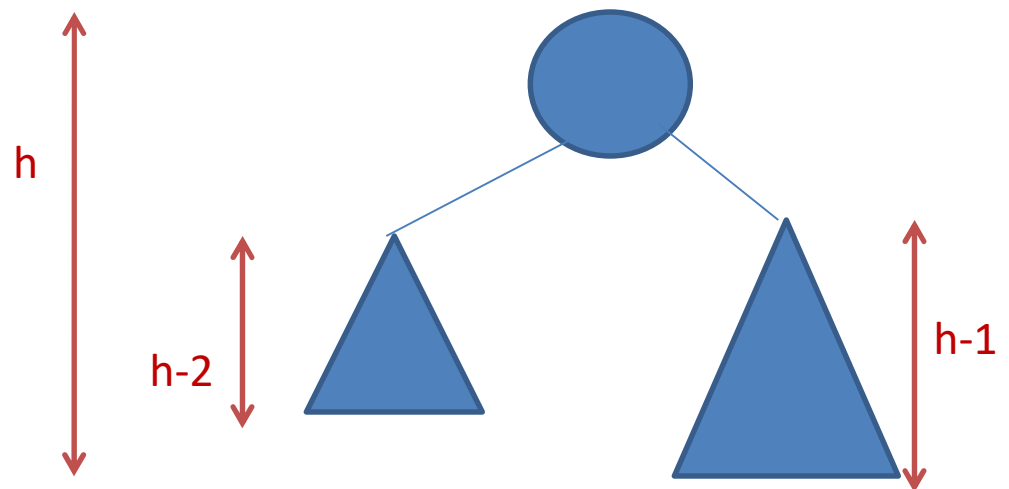
AVL Tree

- AVL trees are balanced.
- The worst case is when the right subtree has height 1 more than left for every node.



Analyze the Height of AVL Tree

- Let N_h to be a minimum number of nodes in an AVL tree of height h .
- $N_h = 1 + N_{h-1} + N_{h-2}$
- $N_h > F_h$ where F_h is a Fibonacci function
- $N_h > 1.618^h / 5^{1/2}$
- Denote $N_h = n$
- $1.618^h / 5^{1/2} < n$
- $h < 1.440 \lg n$

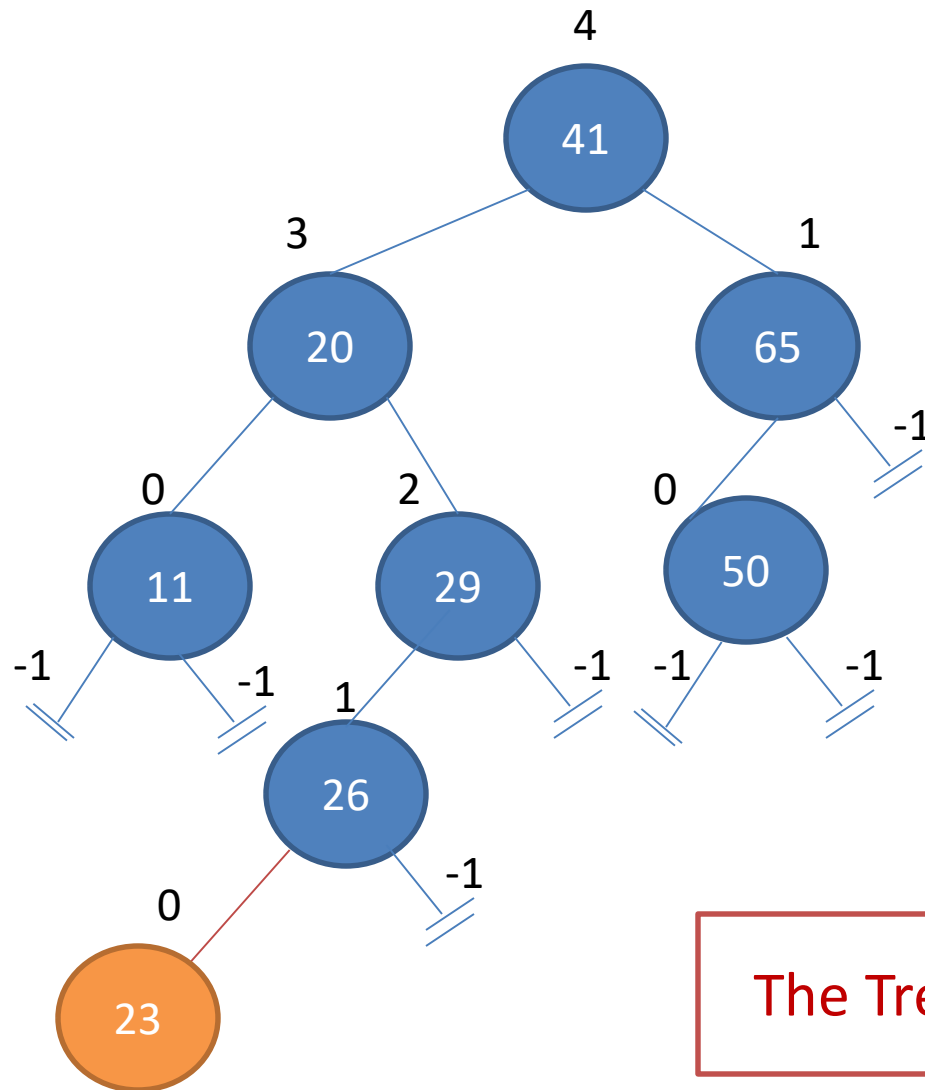


Height of AVL tree = $\lg n$

AVL Insert

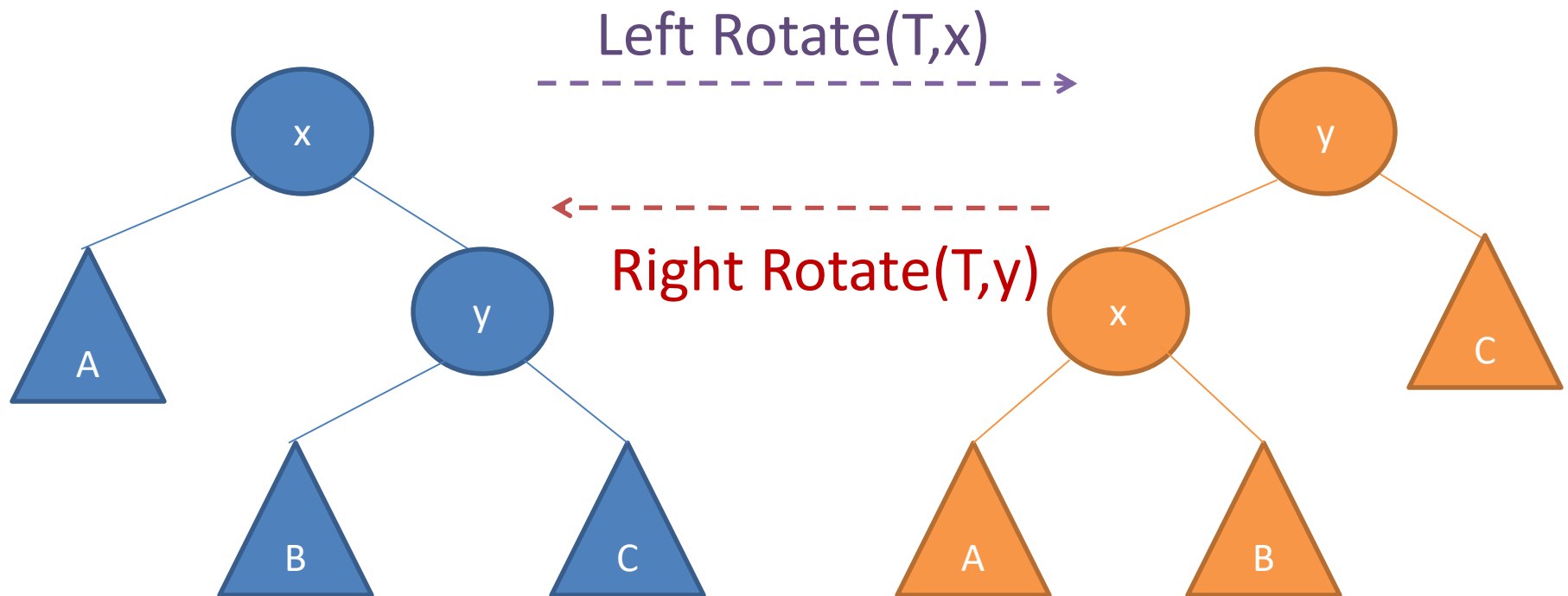
- Use a simple binary search tree insert
- Fix the AVL property using rotations

Example: AVL Insert(T,23)



The Tree is unbalanced !!!

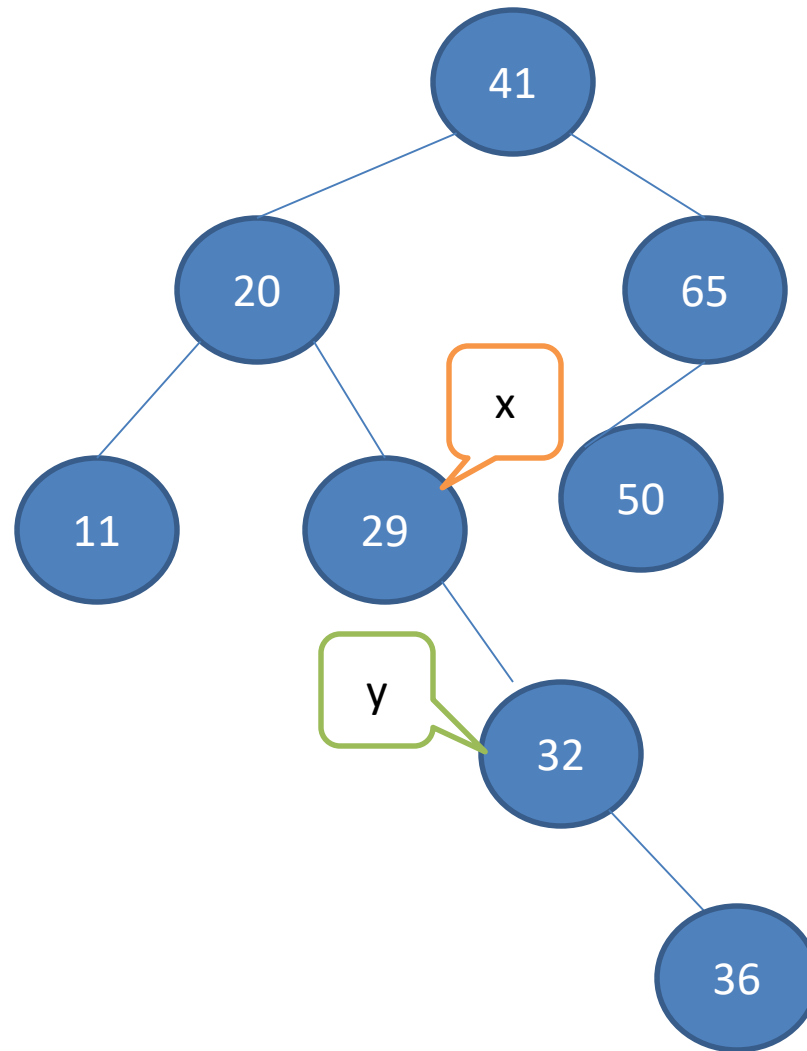
Rotations



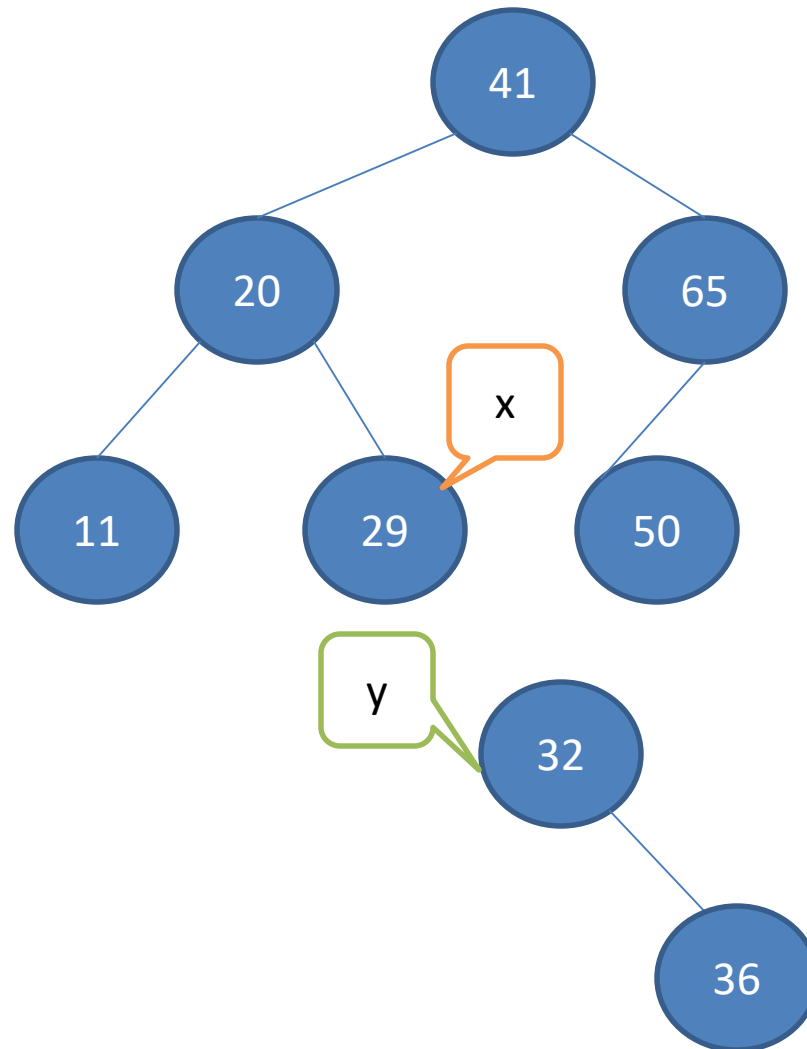
Left-Rotate(T,x)

```
y = right[x]
right[x] = left[y]
if left[y] != nil[T]
    then p[left[y]] = x
p[y] = p[x]
if p[x] = nil[T]
    then root[T] = y
else if x = left[p[x]]
    then left[p[x]] = y
else right[p[x]] = y
left[y] = x
p[x] = y
```

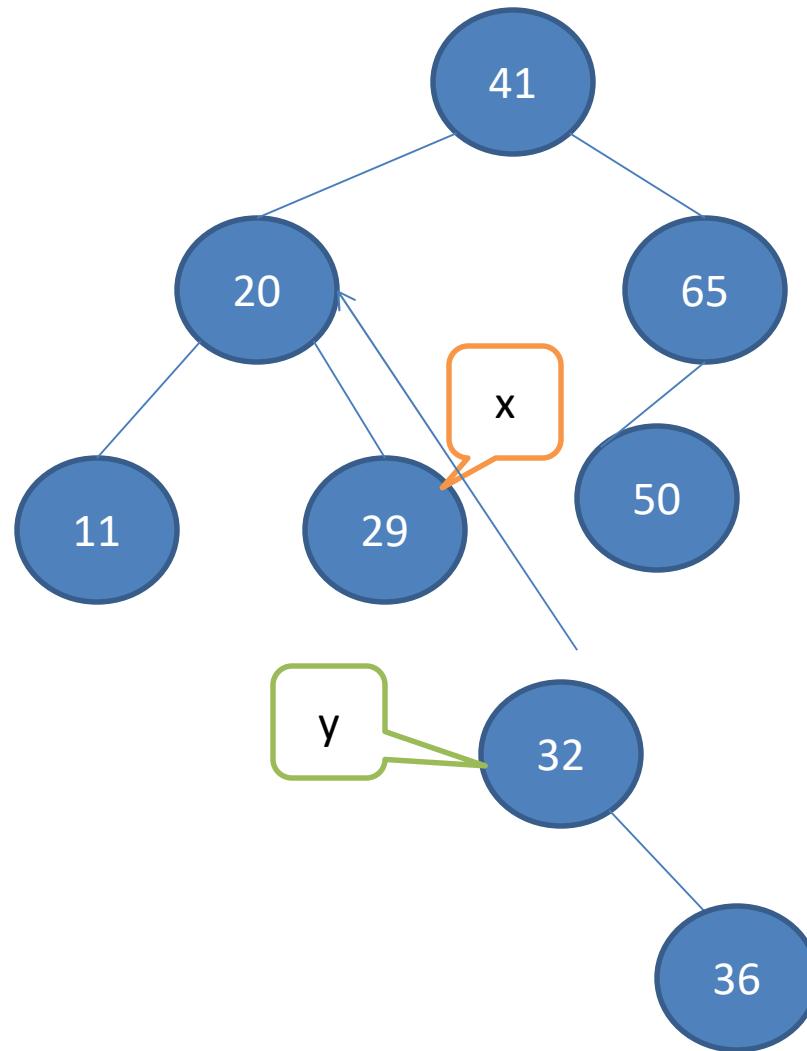
Example: Left-Rotate(T,29)



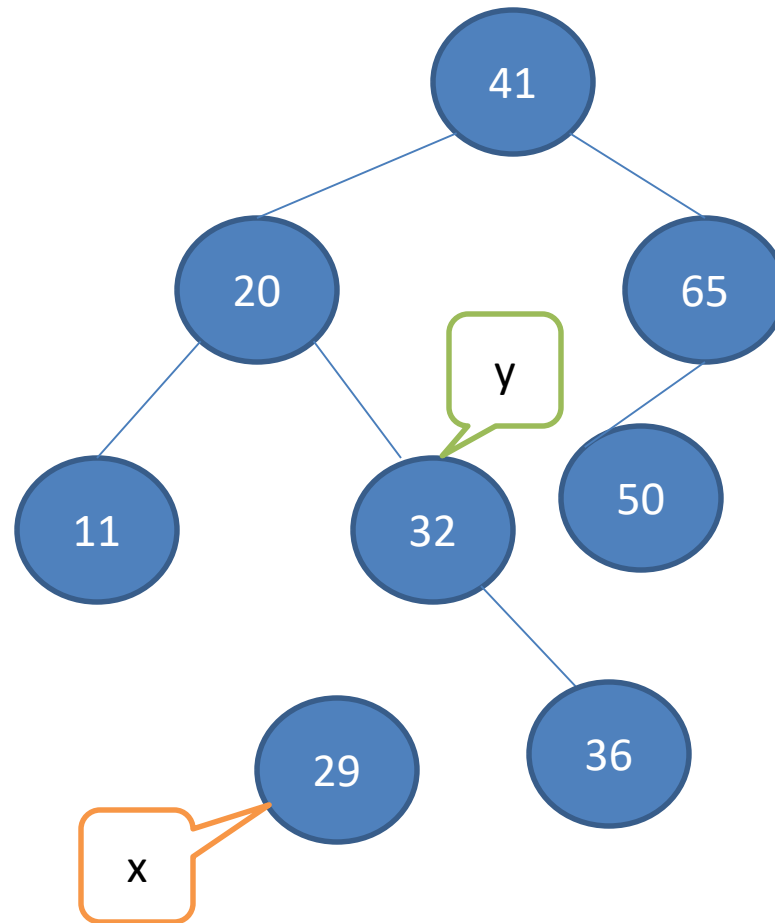
Example: Left-Rotate(T,29)



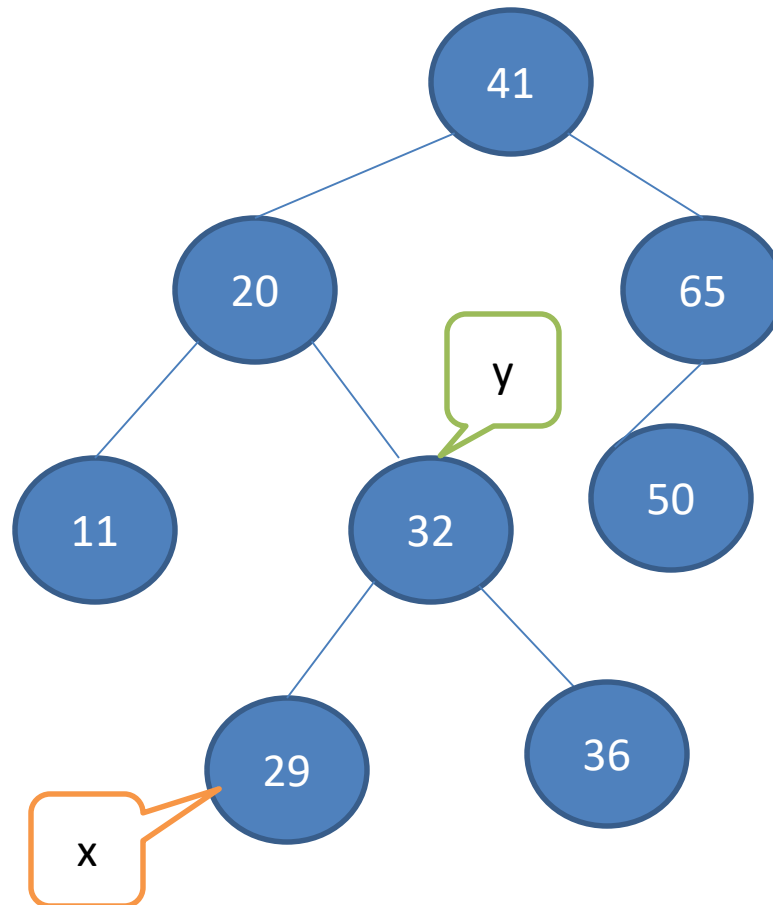
Example: Left-Rotate(T,29)



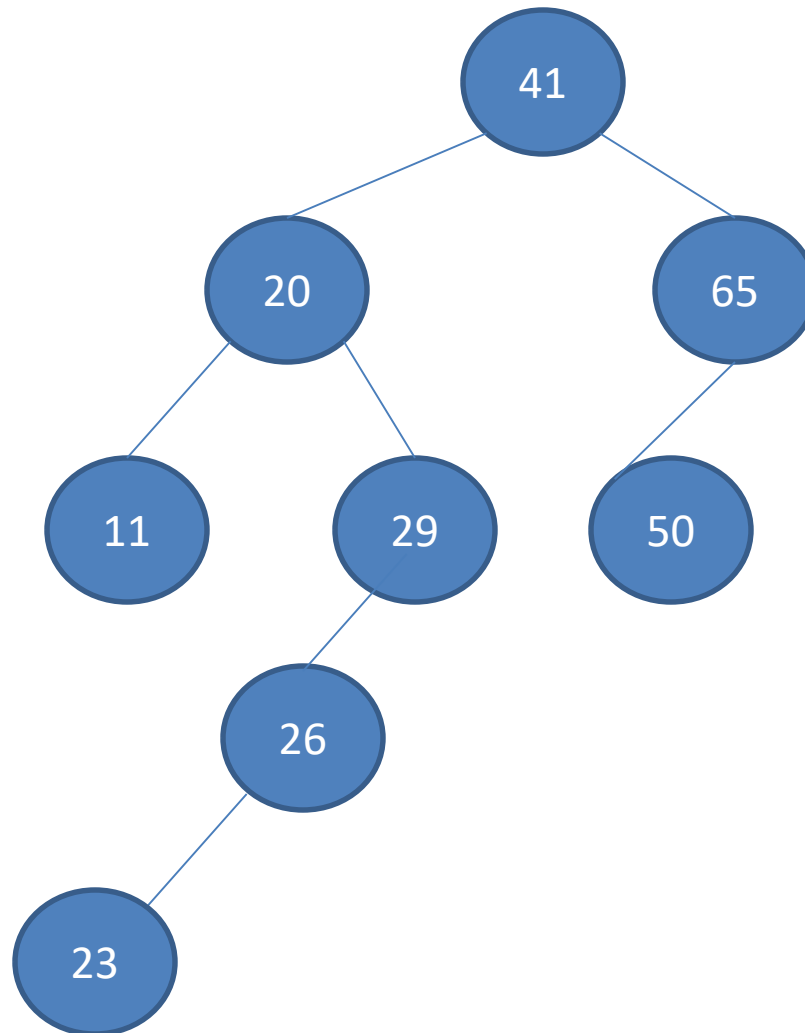
Example: Left-Rotate(T,29)



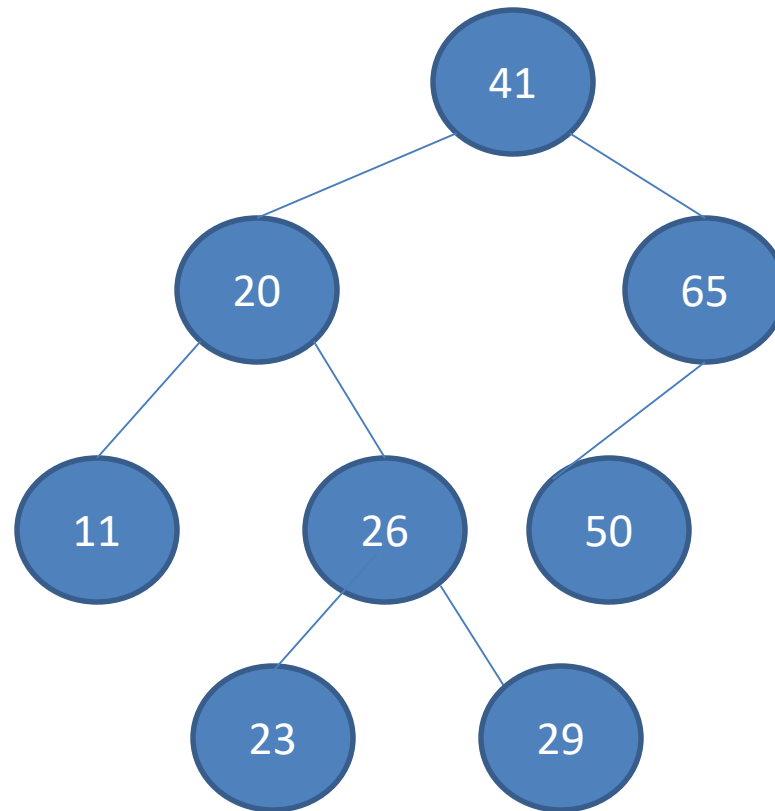
Example: Left-Rotate(T,29)



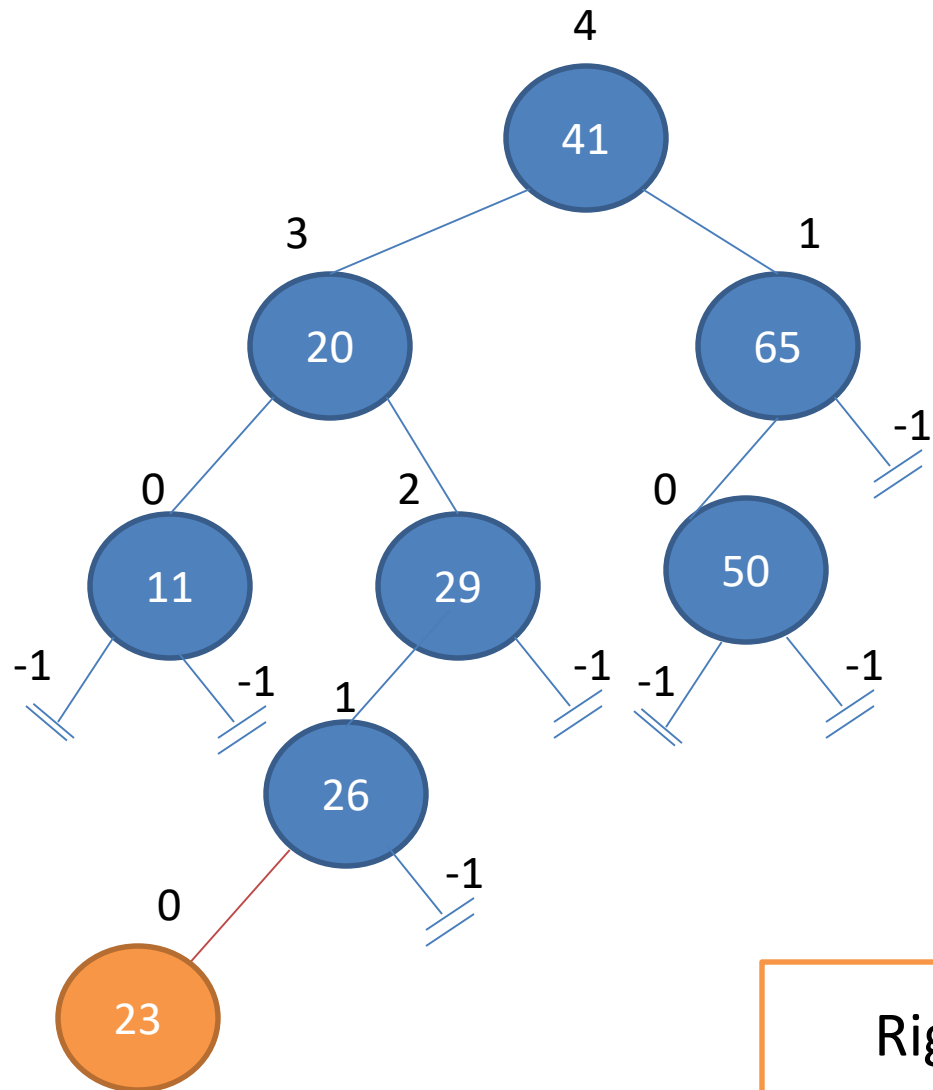
Example: Right-Rotate(T,29)



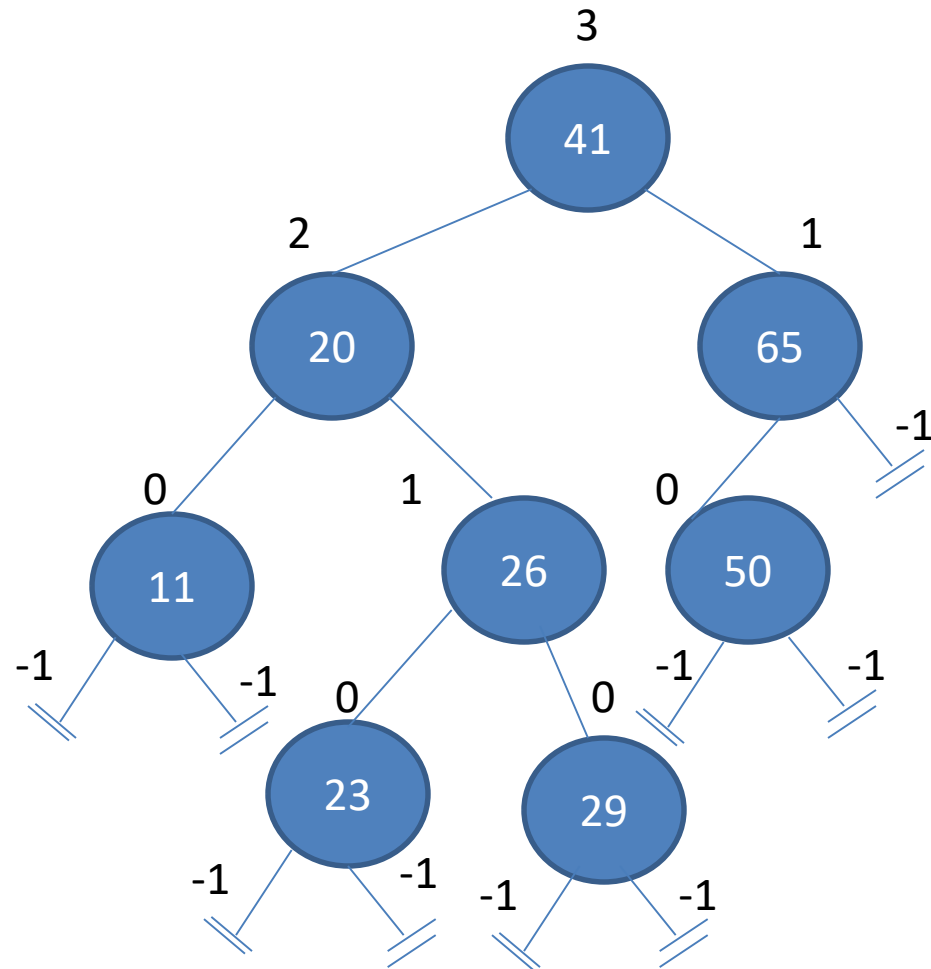
Example: Right-Rotate(T,29)



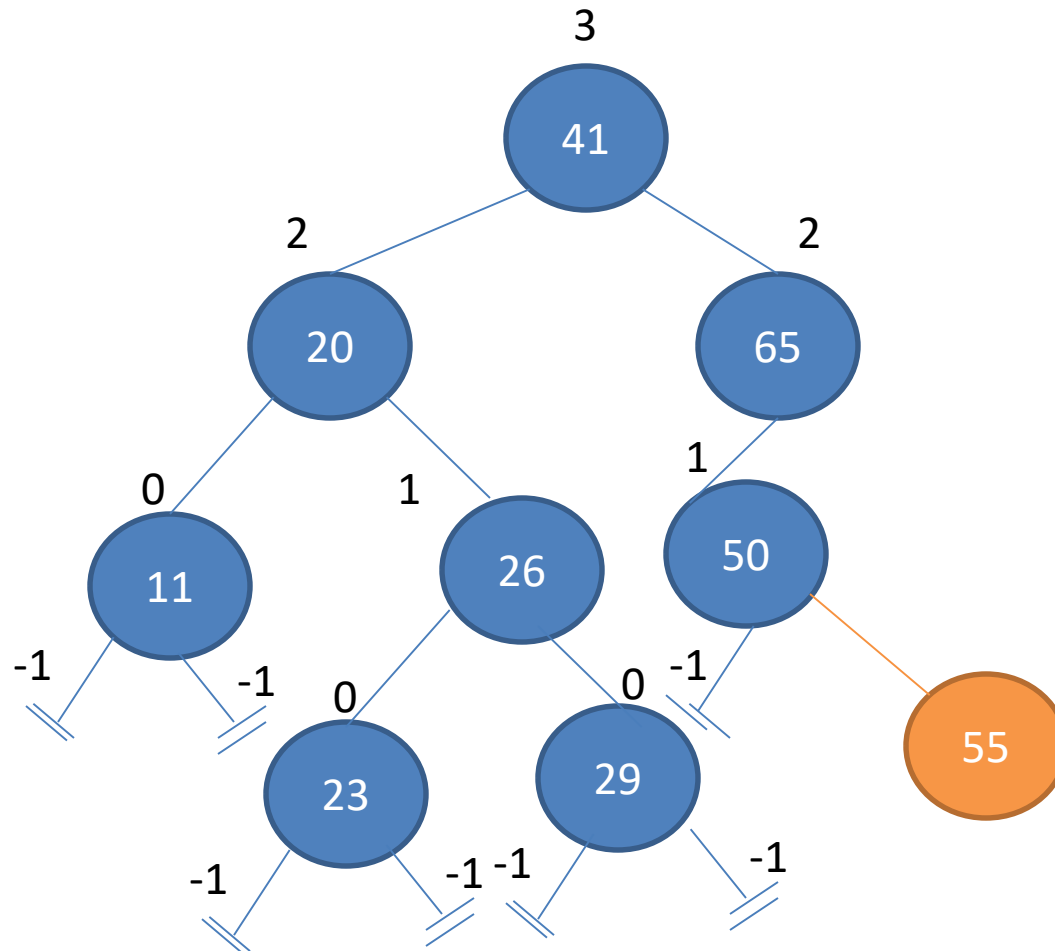
Example: AVL Insert(T,23)



Example: AVL Insert(T,23)

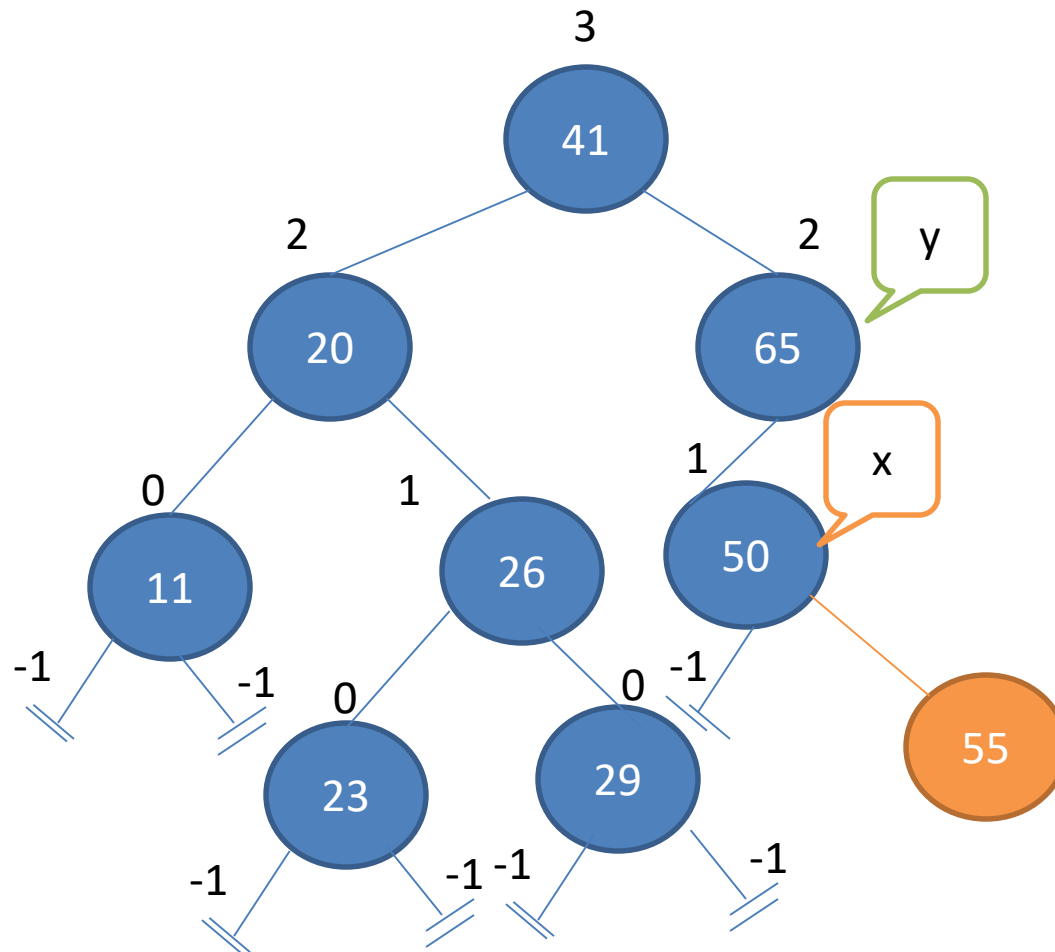


Example: AVL Insert(T,55)



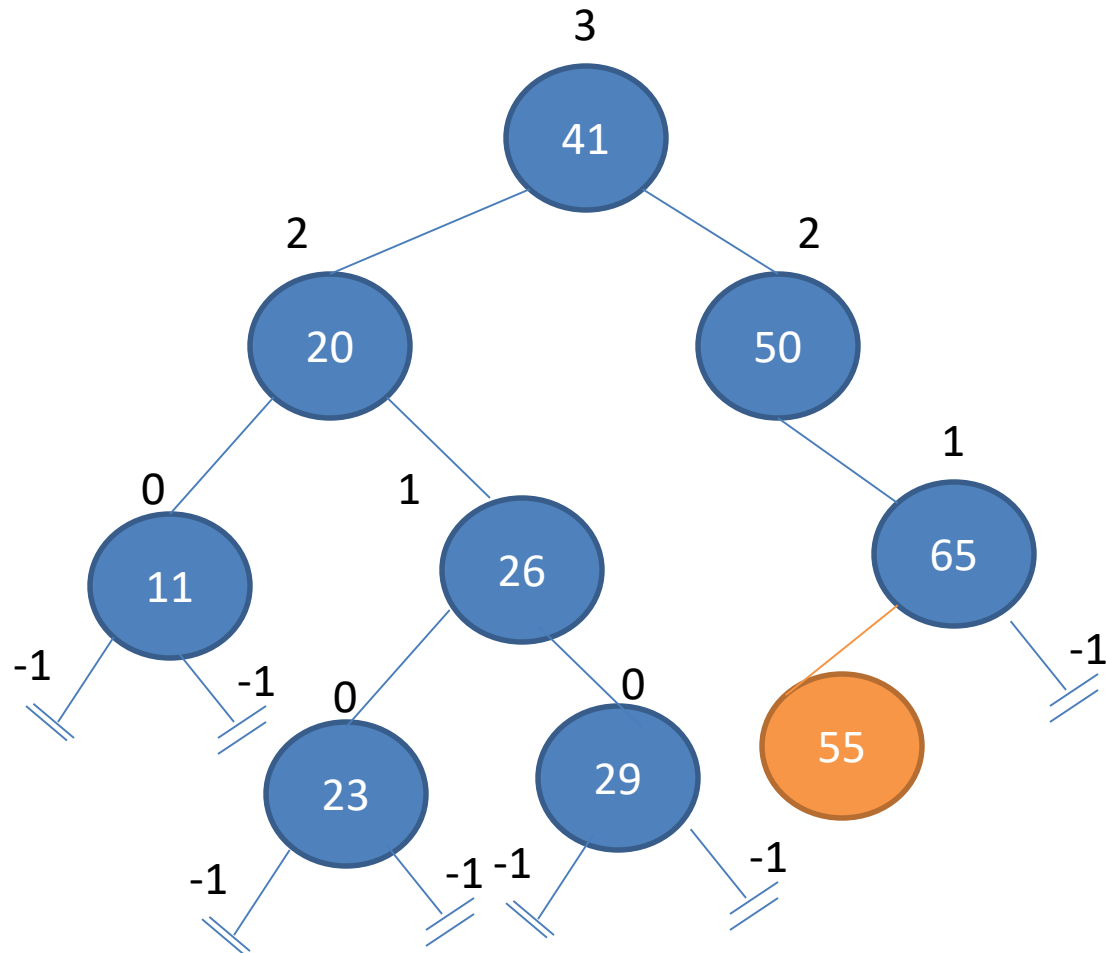
The Tree is unbalanced !!!

Example: AVL Insert(T,55)



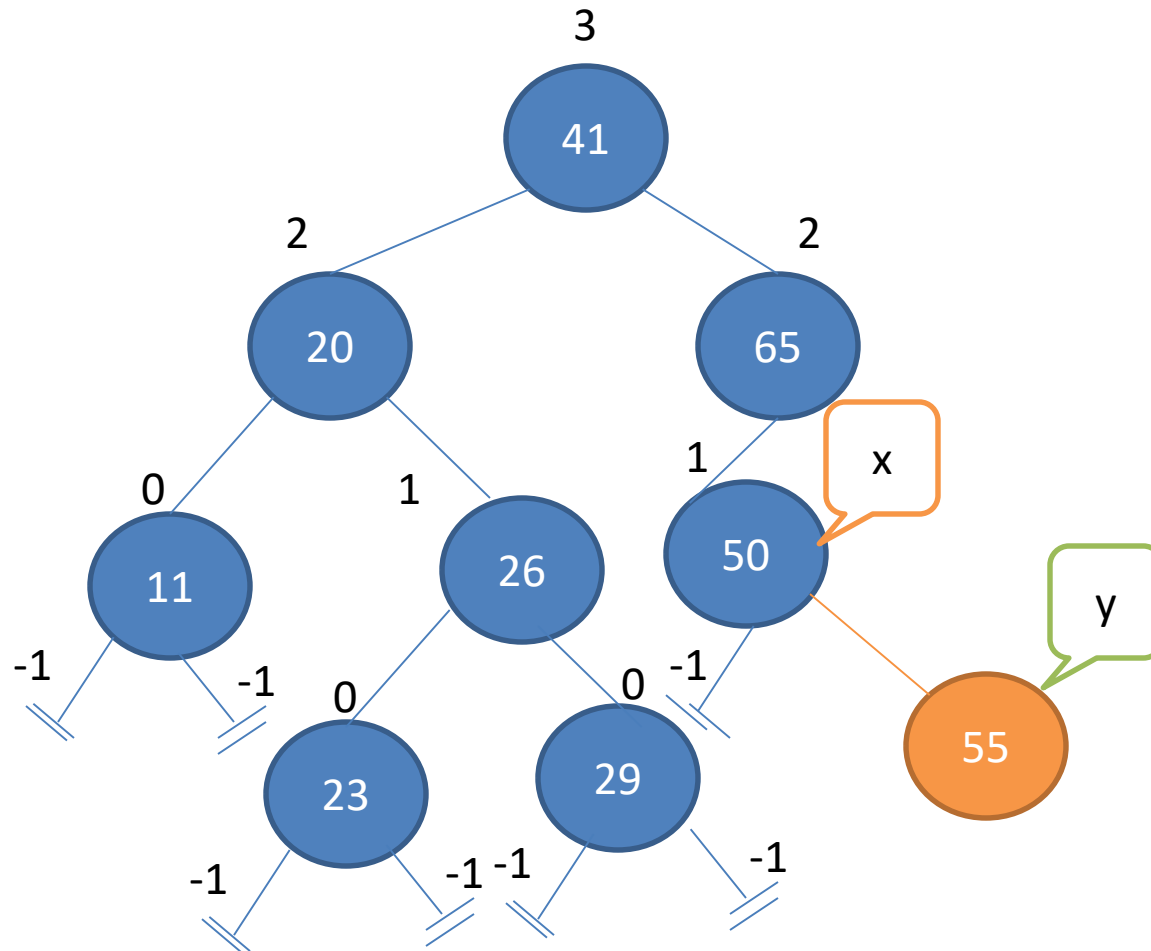
Right Rotate(T,65)

Example: AVL Insert(T,55)



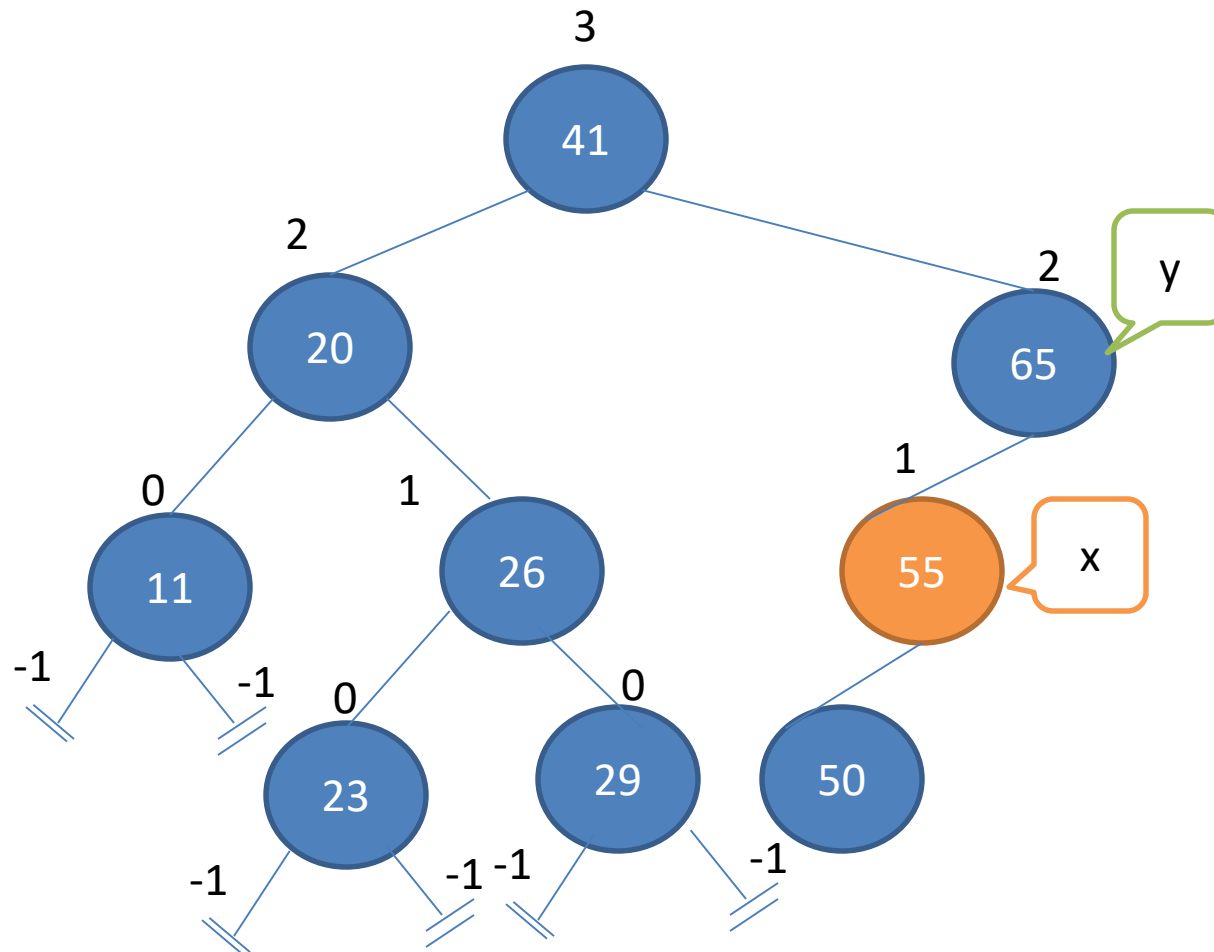
The Tree is unbalanced !!!

Example: AVL Insert(T,55)



Left-Rotate(T,50)

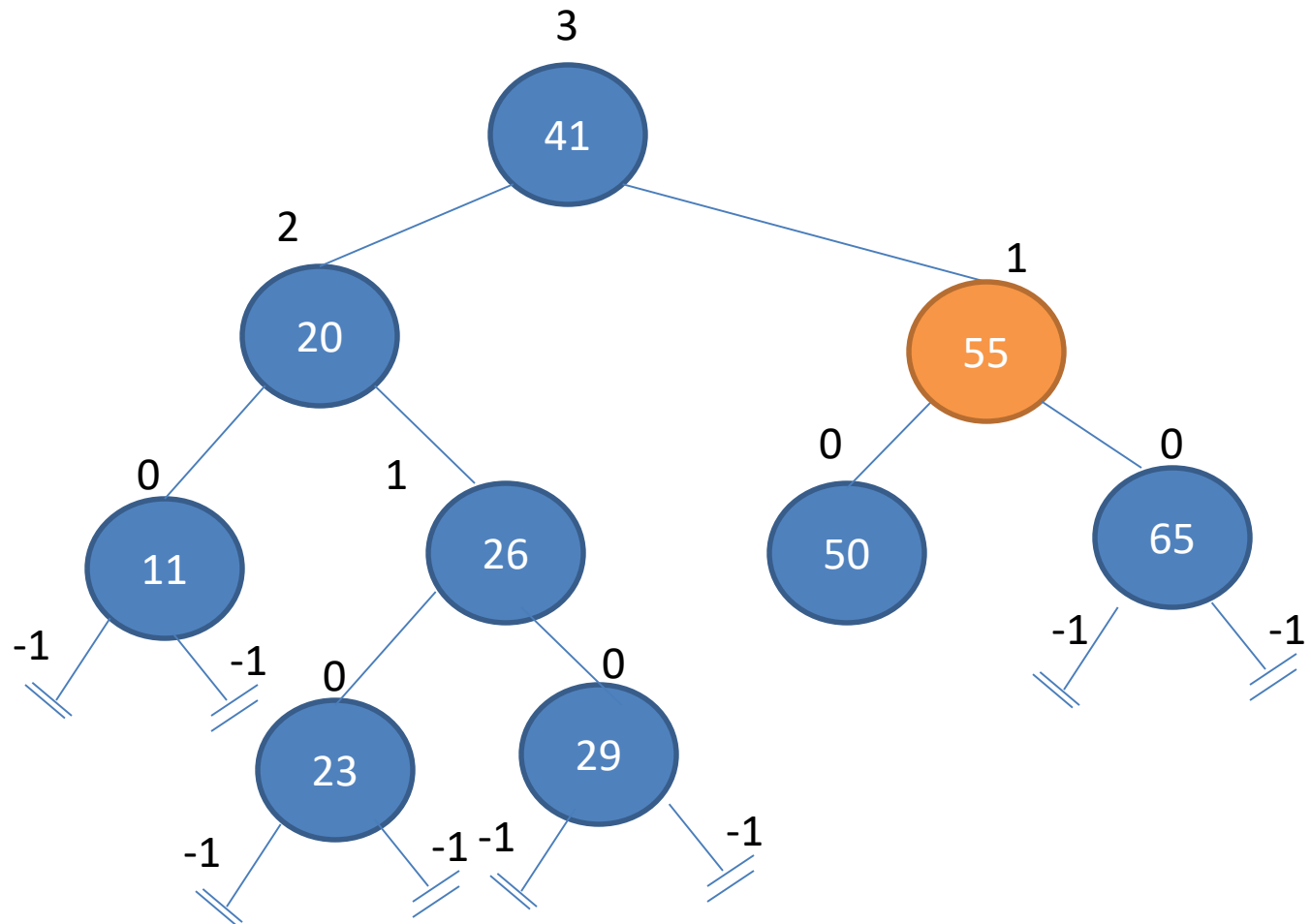
Example: AVL Insert(T,55)



Right-Rotate(T,65)

The Tree is unbalanced !!!

Example: AVL Insert(T,55)

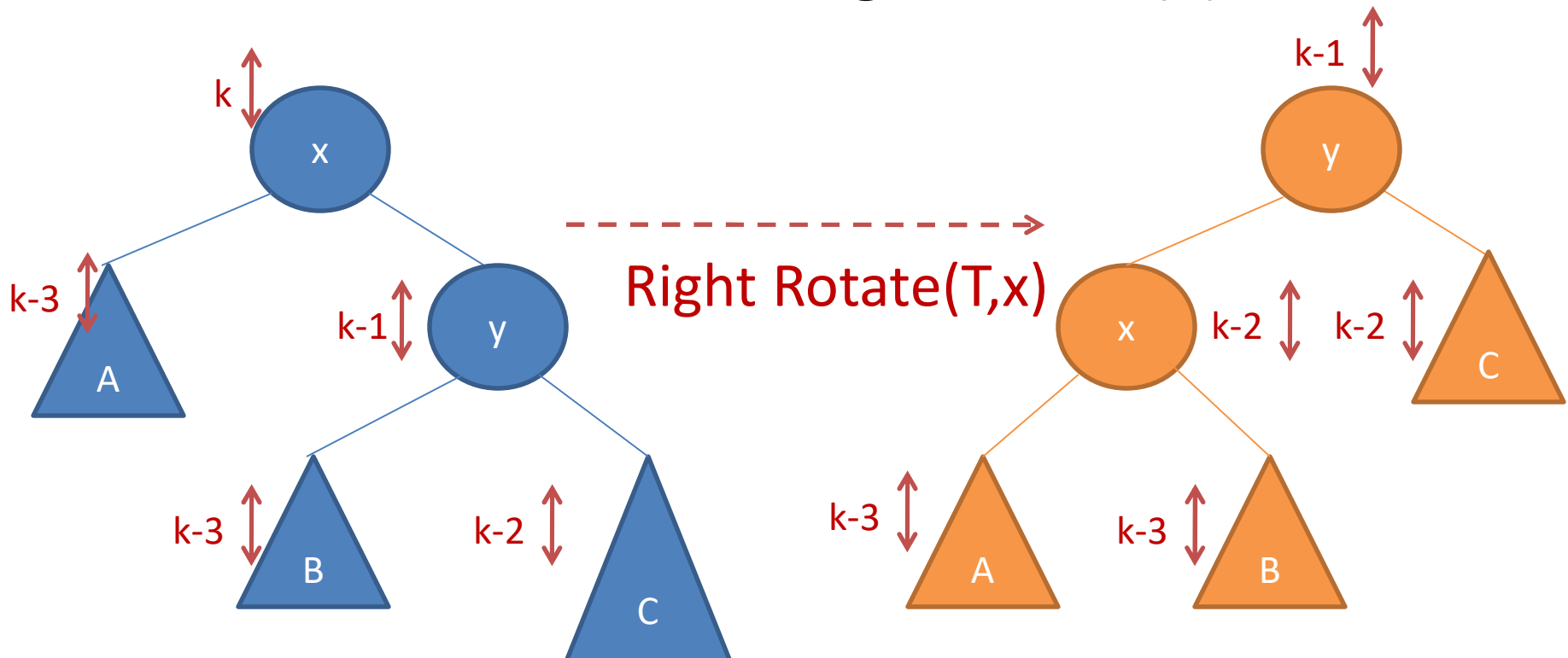


AVL Insert

- Use a simple binary search tree insert
- Fix the AVL property from changed node up
 - Suppose x is lowest node violating AVL property

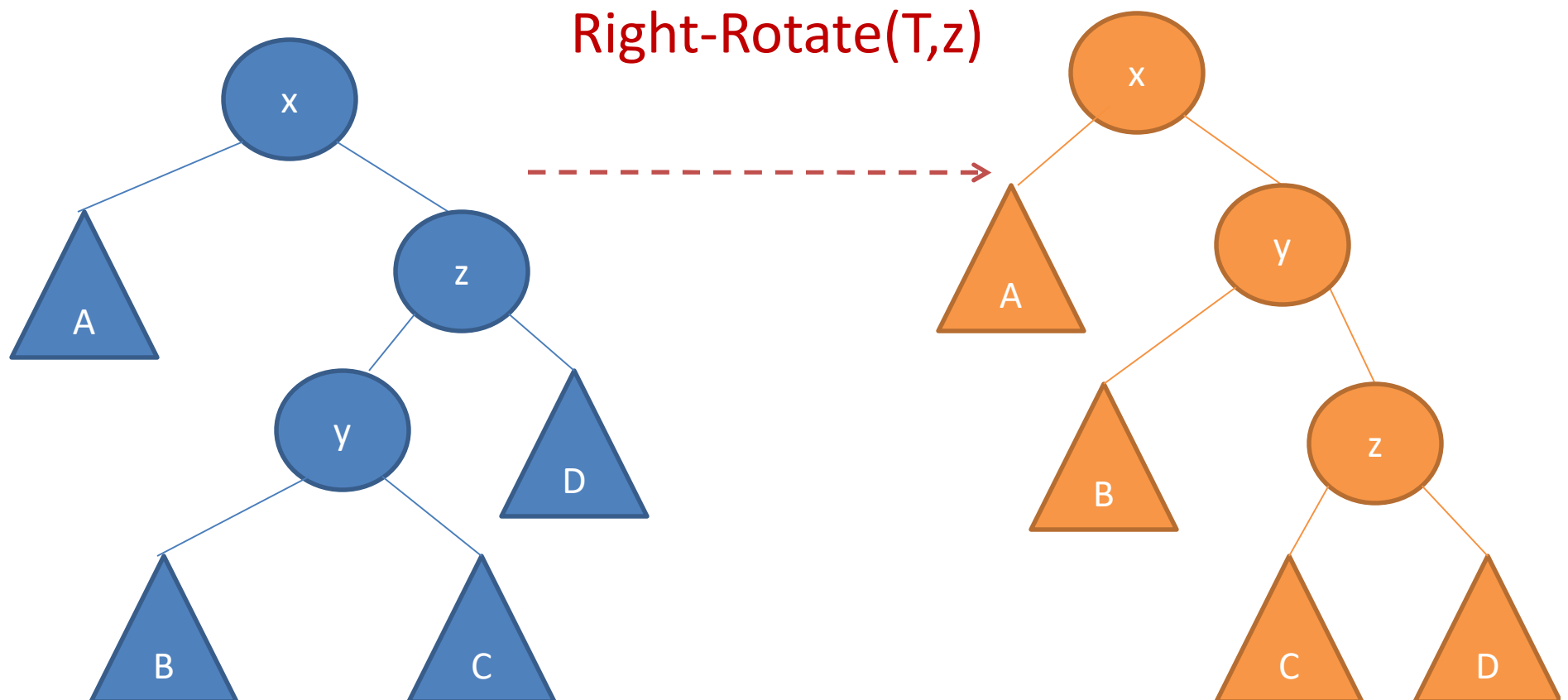
AVL Insert

- Assume the right child of x is higher
- If the right child of x is right-heavy or balanced, then we do $\text{right-rotate}(x)$



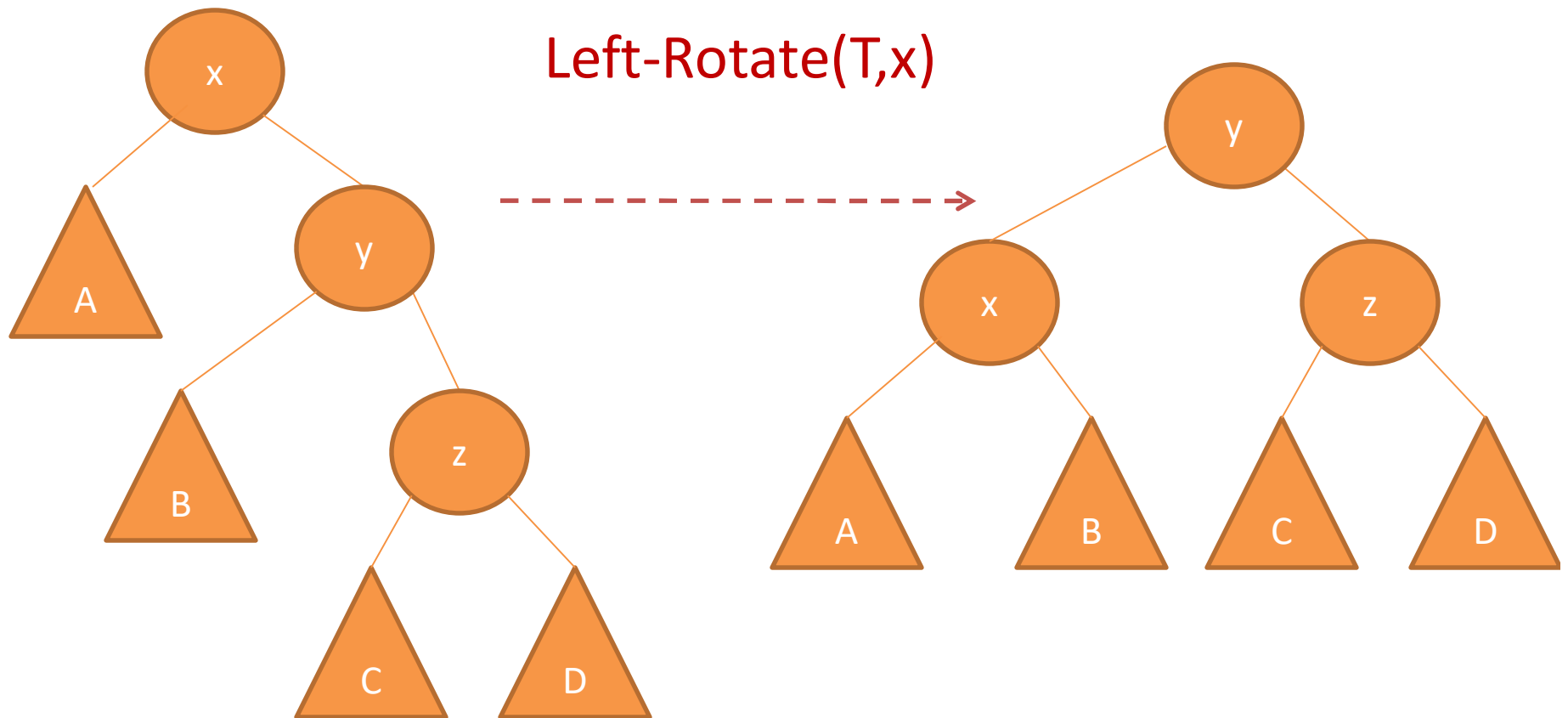
AVL Insert

- Assume the right child of x is higher
- Else



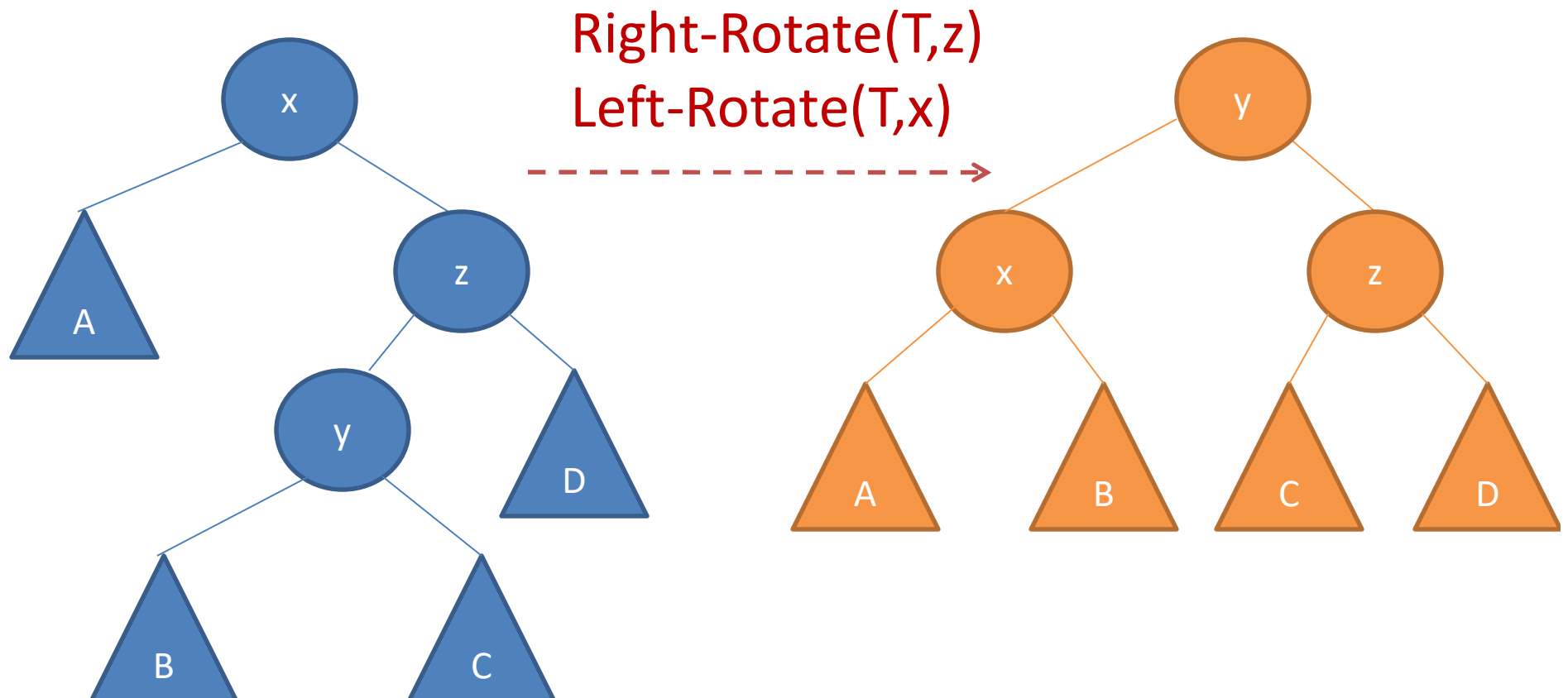
AVL Insert

- Assume the right child of x is higher
- Else



AVL Insert

- Assume the right child of x is higher
- Else



AVL Sort

- Insert n items
 - Take $O(n \lg n)$ time
- In-order tree traversal
 - Take $O(n)$ time

Summary

- Abstract Data Types
- Insert , Delete
- Min
- Successor/ predecessor

Priority queue

Data Structure

- Heap or AVL
- Balanced BST

Practice: AVL Insert

4	7	5	10	23	65	73
---	---	---	----	----	----	----