파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

BST의 균형을 유지하는

AVL 트리 알고리즘에 대해 학습한다

Data Structures in Python
Chapter 7 - 2

- Binary Search Tree(BST)
- BST Algorithms
- **AVL Tree**
- AVL Algorithms

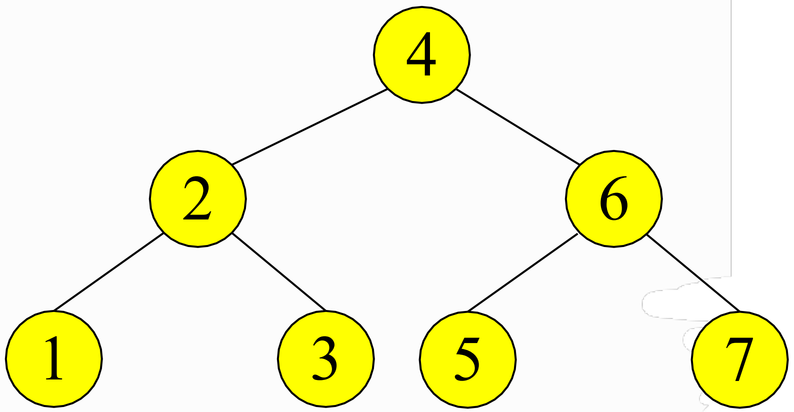
Agenda & Readings

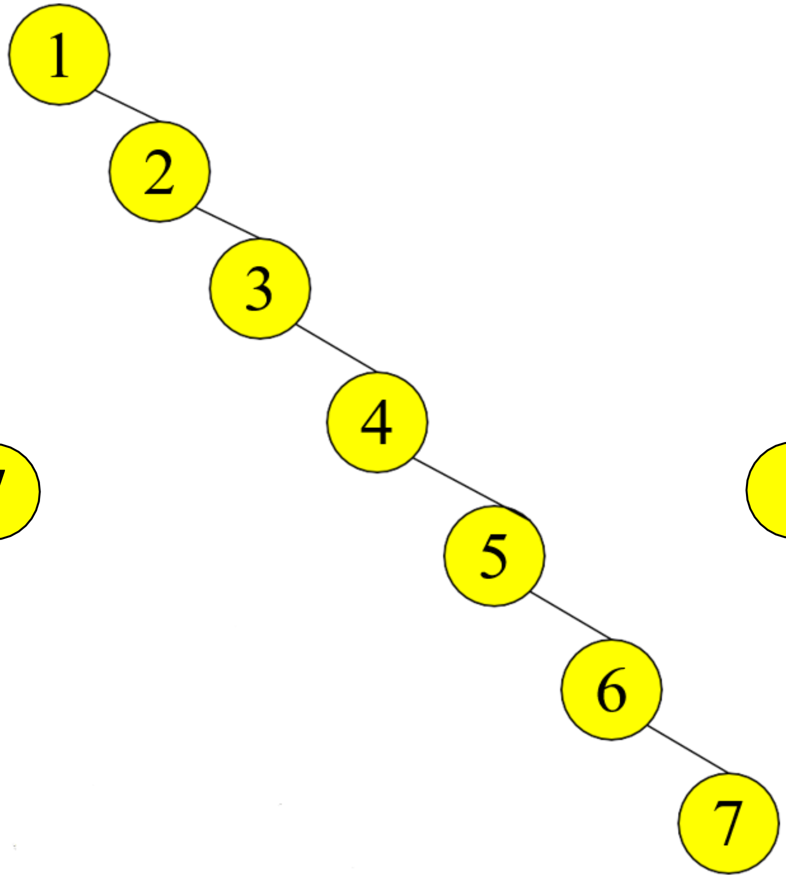
- AVL Tree Introduction
  - Binary Search Tree Review
  - AVL Tree Introduction
  - Balance factor
  - Single/Double Rotation

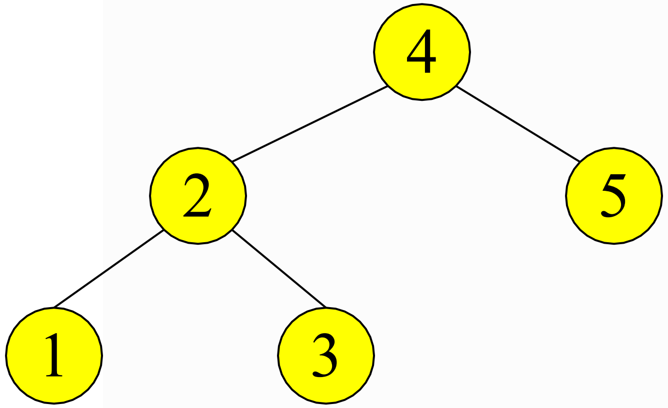
- Reference:
  - Problem Solving with Algorithms and Data Structures
    Chapter 6 - Tree
  - Wikipedia: AVL tree

Binary search trees - Review

- Balanced and unbalanced BST
  - **The best-case** time complexity of BST operations is $O(\log_2 N)$, and **the worst-case** $O(N)$.







Binary search trees - Review

- Many algorithms exist for keeping BST balanced
  - Adelson-Velskii and Landis (AVL) tree - (height-balanced tree)
  - 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

AVL Tree - Good but not Perfect Balance

AVL Tree (1962)

- Named after two Russian mathematicians
- Georgii Adelson-Velsky (1922 - 2014)
- Evgenii Mikhailovich Landis (1921-1997)

AVL Tree - Good but not Perfect Balance

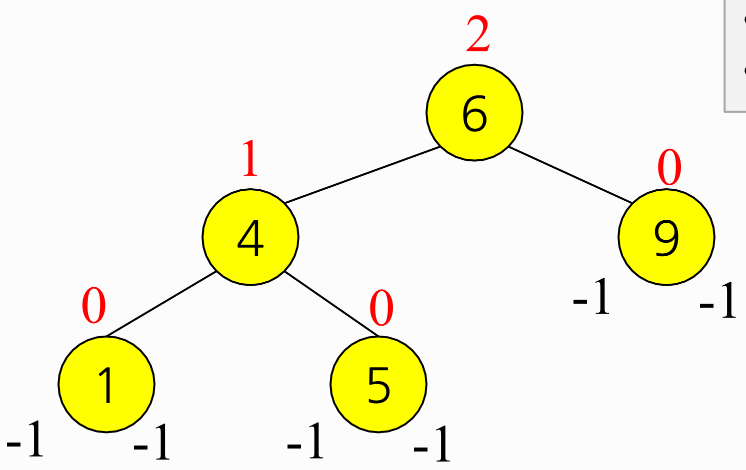
AVL Tree Algorithm:

- Named after two Russian mathematicians in 1962
- Georgii **A**delson-**V**elsky (1922 - 2014)
- Evgenii Mikhailovich **L**andis (1921-1997)

- AVL tree is a height-balanced binary search tree.
  - Balance factor of a node
    - bf = height(left subtree) - height(right subtree)
    - May store current heights in each node or compute it on the fly
  - For every node, heights of left and right subtree can differ **by no more than one.**

Balance Factor

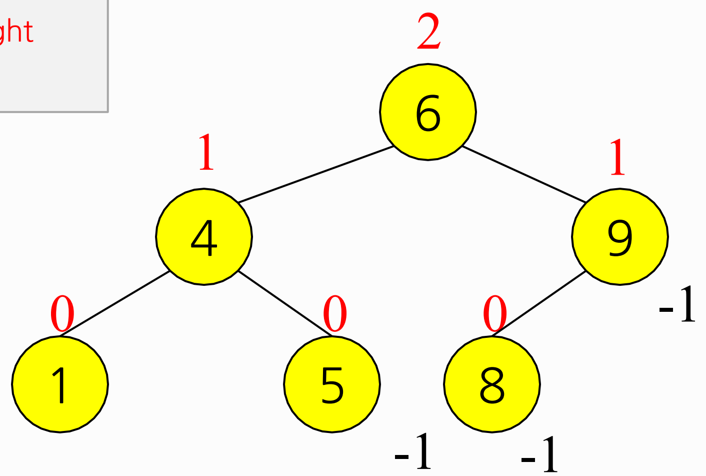
Tree A (AVL)



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

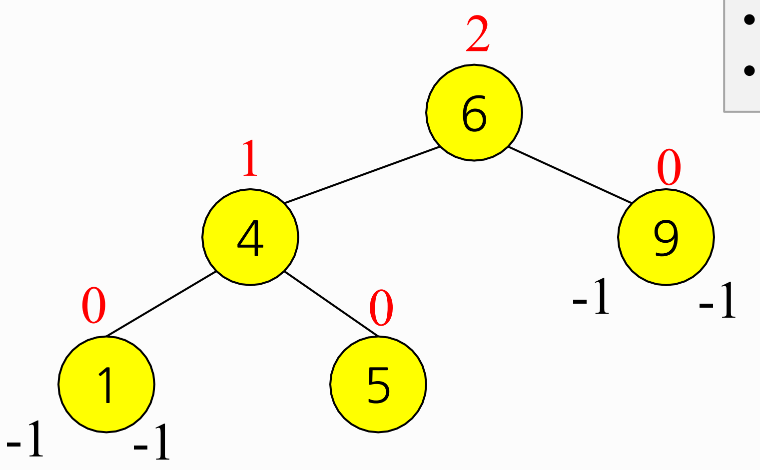
Node heights
before inserting 7

Tree B (AVL)



Balance Factor

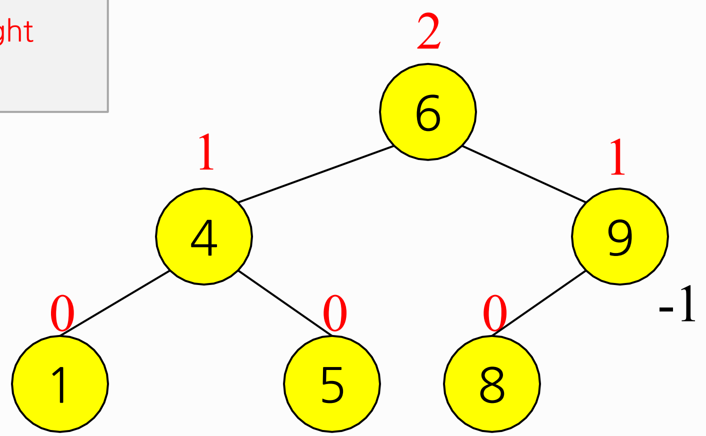
Tree A (AVL)

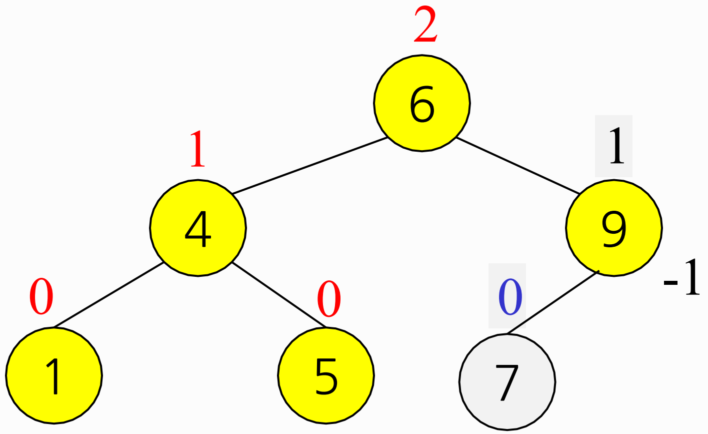


- height of node = h
- balance factor = h_{left} - h_{right}
- empty height = -1

Node heights
before inserting 7

Tree B (AVL)

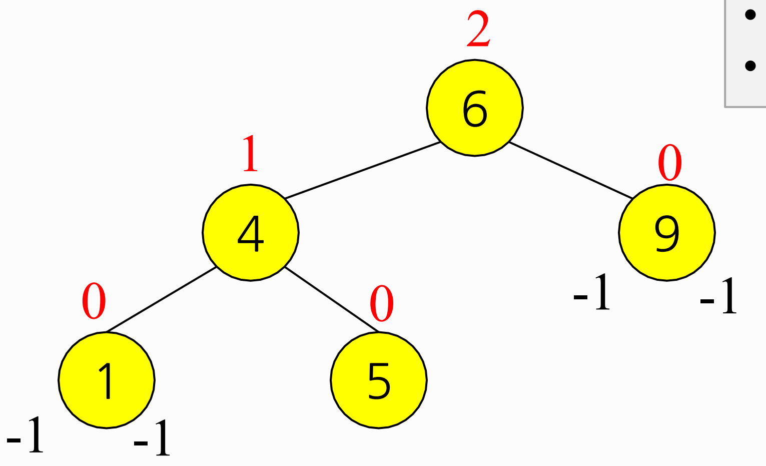




Node heights
after inserting 7

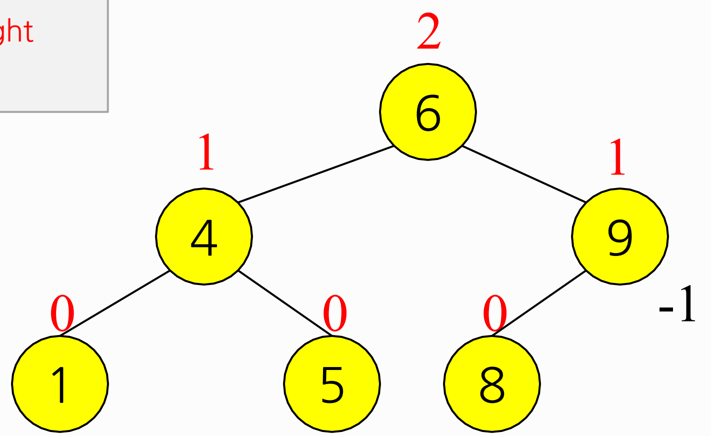
Balance Factor

Tree A (AVL)

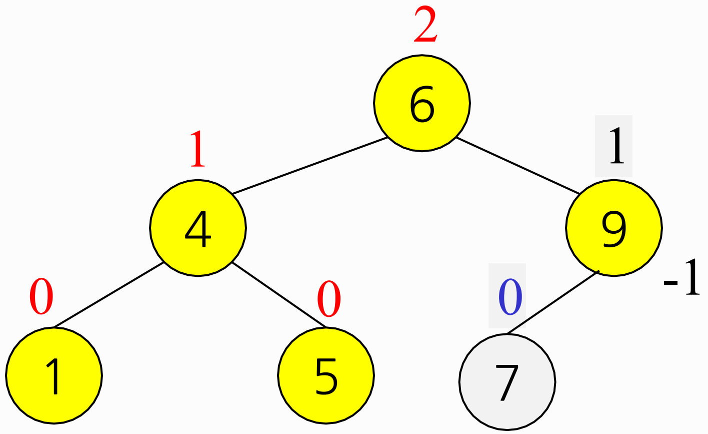


- height of node = h
- balance factor = h_{left} - h_{right}
- empty height = -1

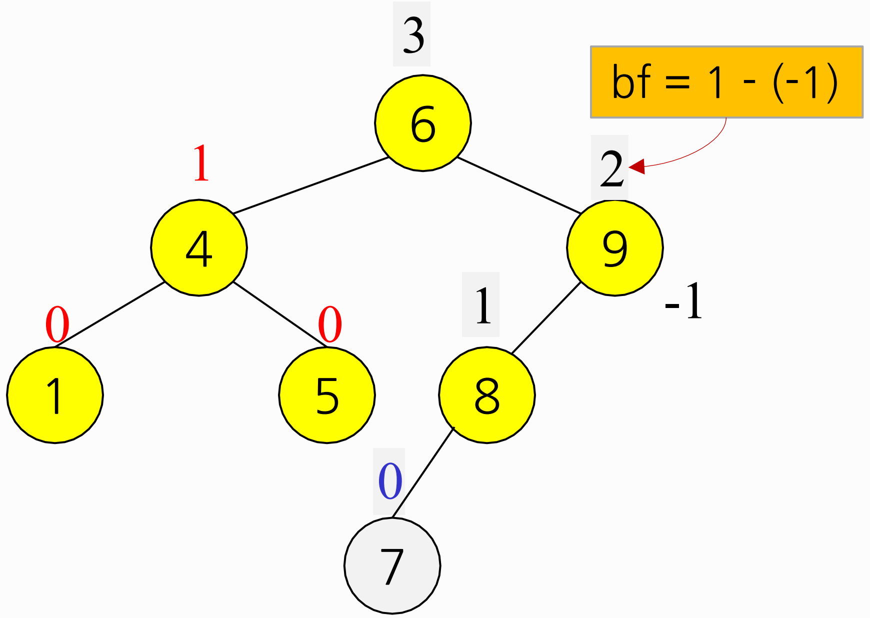
Tree B (AVL)



Node heights
before inserting 7



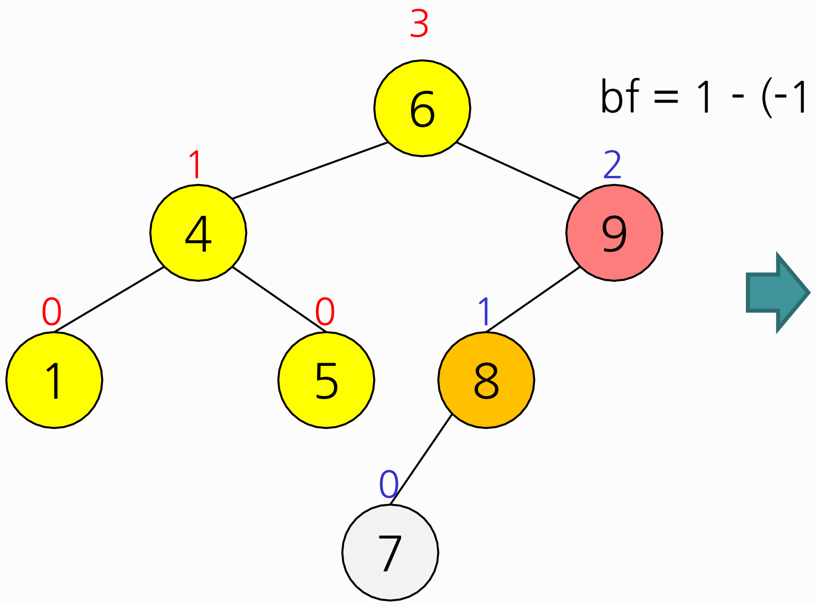
Node heights
after inserting 7



Rotation in AVL Trees

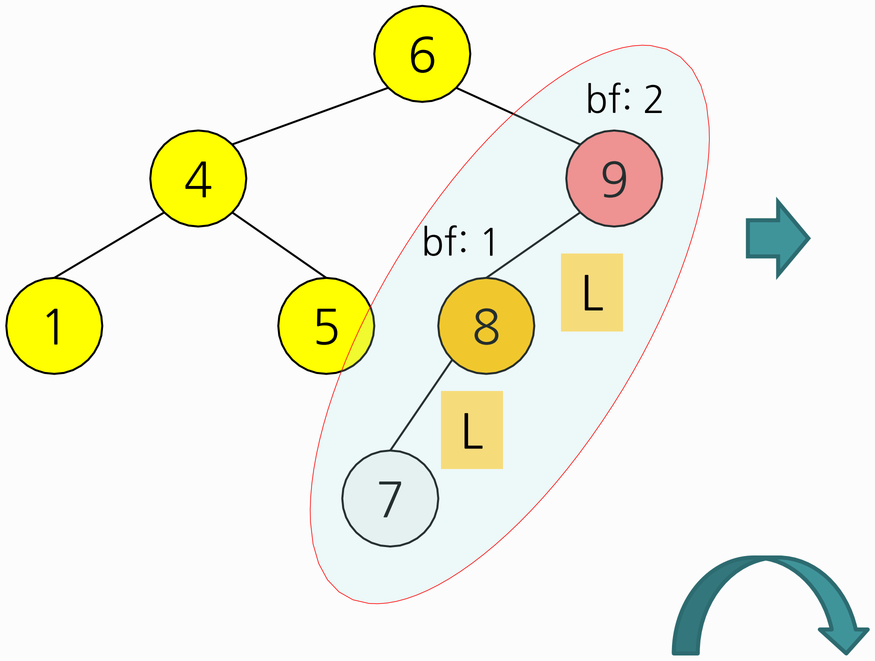
- Insert operation may cause balance factor to become 2 or -2 for some node
  - Only nodes on the path from insertion point to the root node have possibly changed in height.
  - So, after the insertion, **go back up** to the root node by node.
  - If a new balance factor (the difference $h_{left} - h_{right}$)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

Single Rotation Example



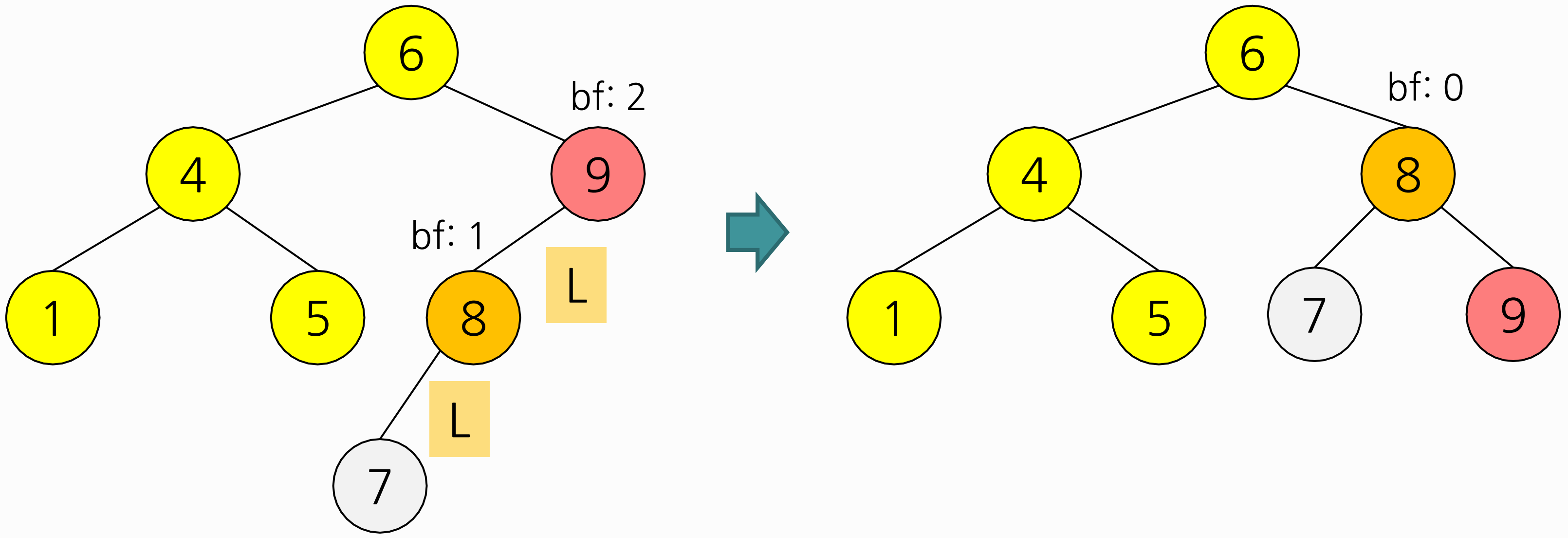
bf = 1 - (-1) = 2

Single Rotation Example



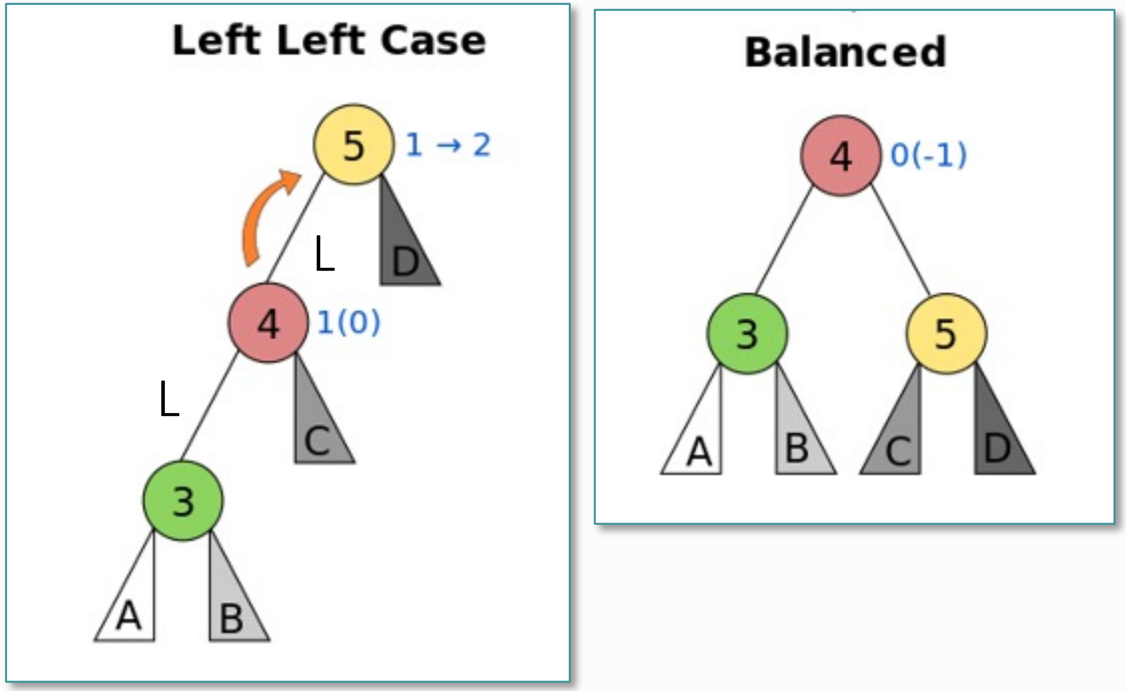
LL case
Rotate right at 9

Single Rotation Example



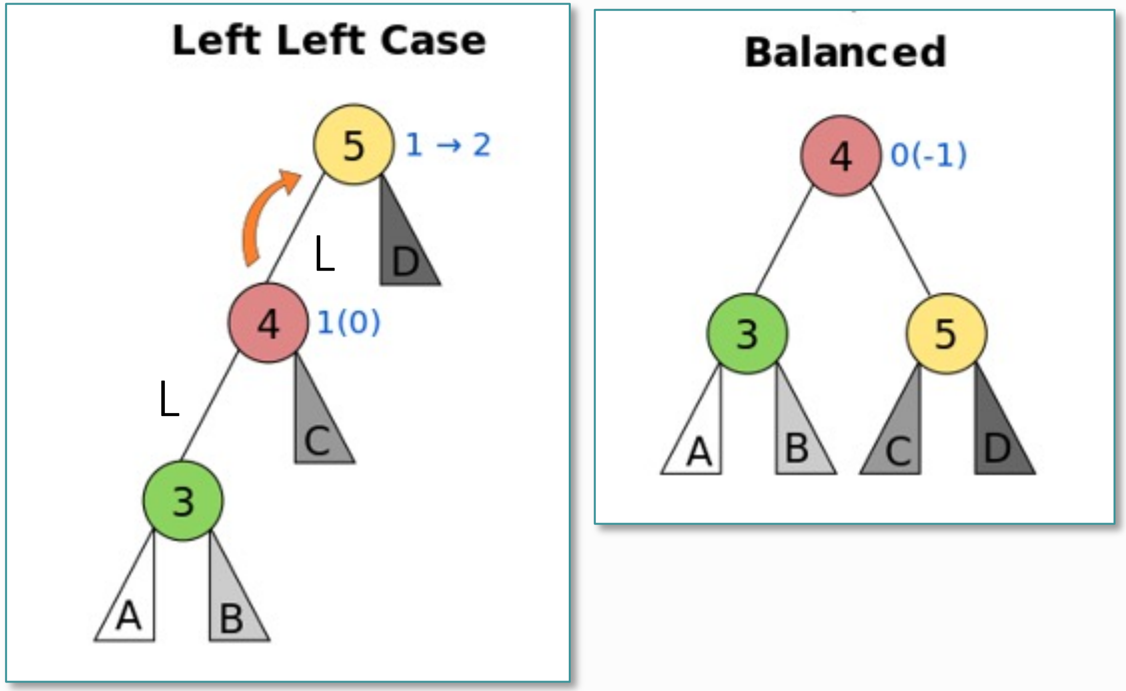
LL case
Rotate right at 9

Single Rotation Example

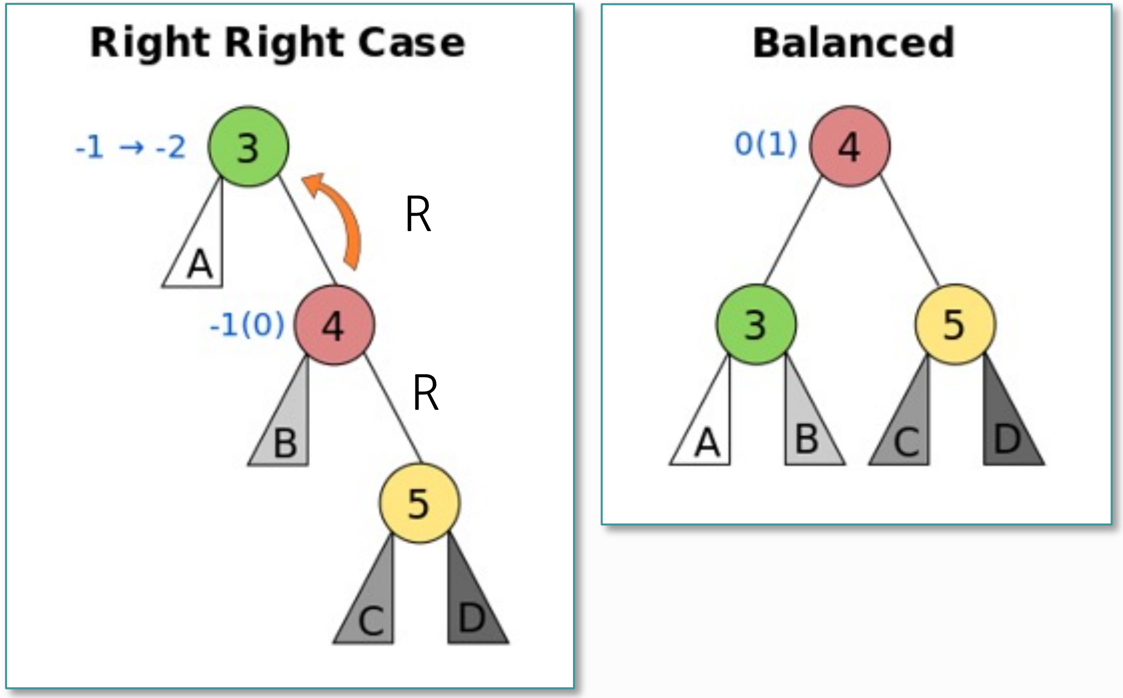


LL Case - Single Right Rotation

Single Rotation Example



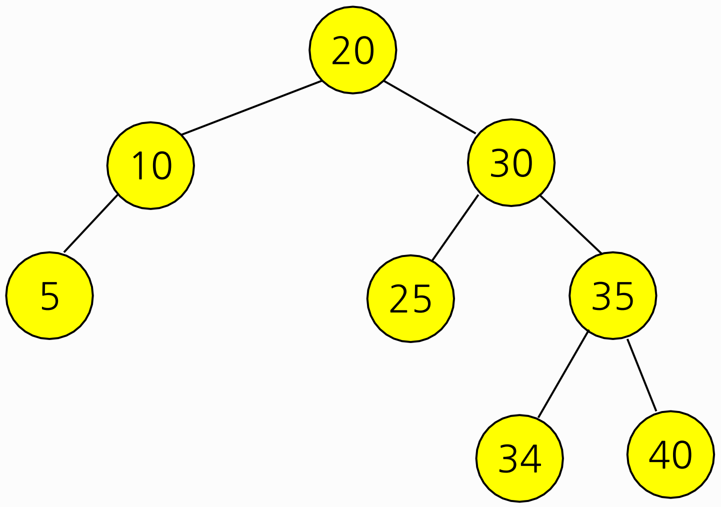
LL Case - Single Right Rotation



RR Case - Single Left Rotation

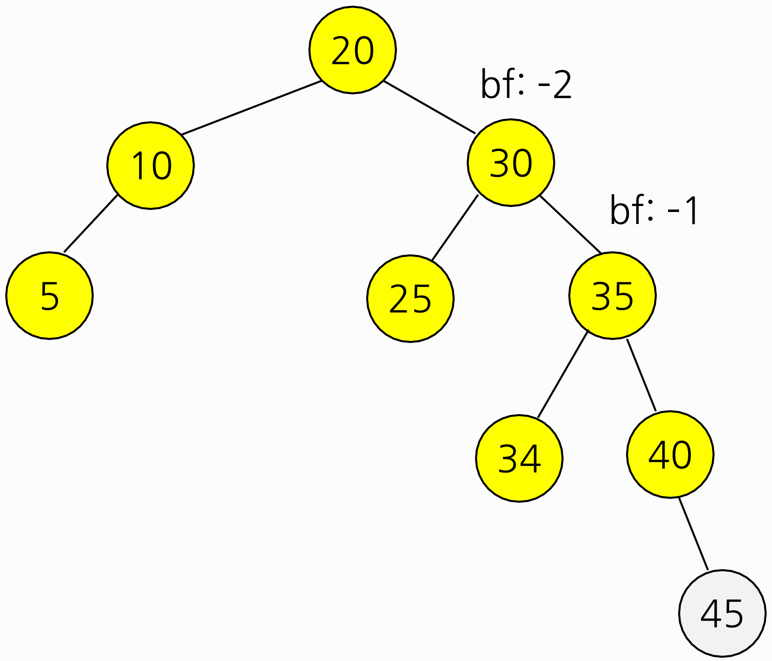
Single Rotation Exercise:

AVL tree balanced?



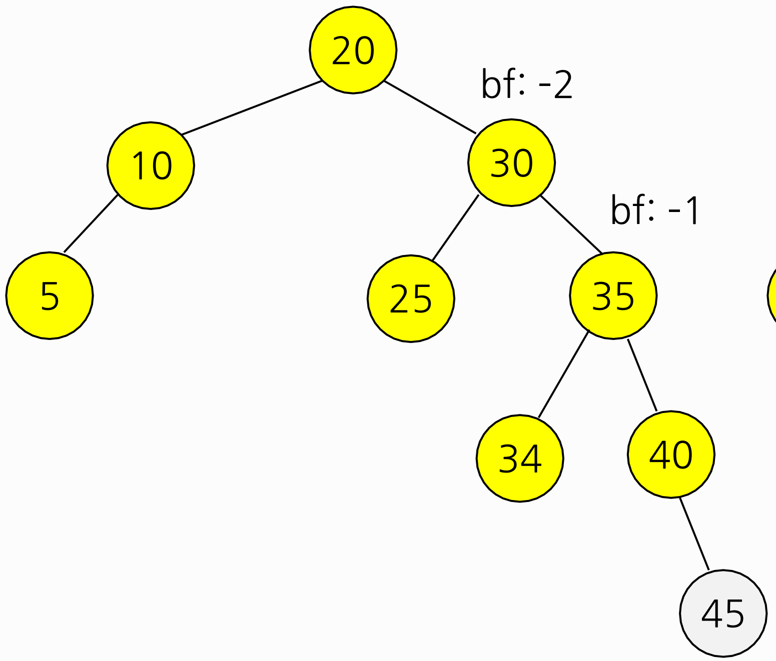
Single Rotation Exercise:

AVL tree balanced
after adding 45?

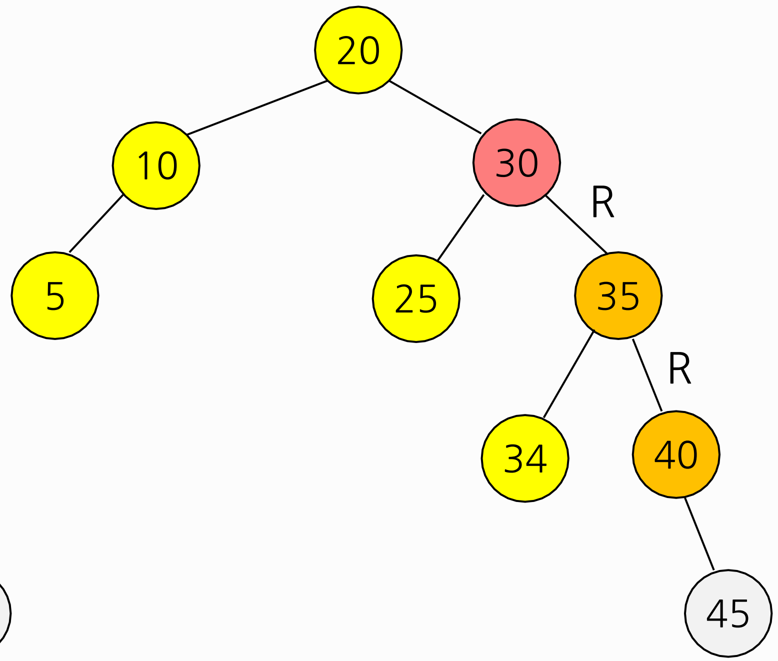


Single Rotation Exercise:

AVL tree balanced
after adding 45?

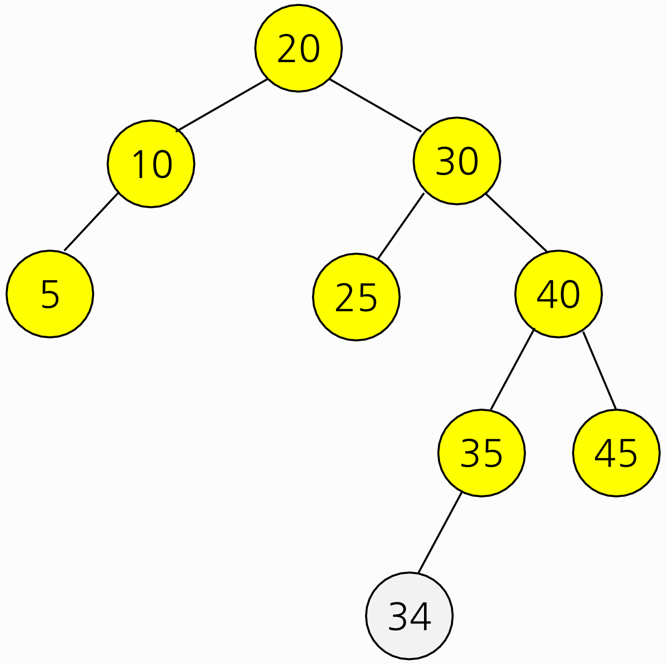


RR case
Rotate left at 30



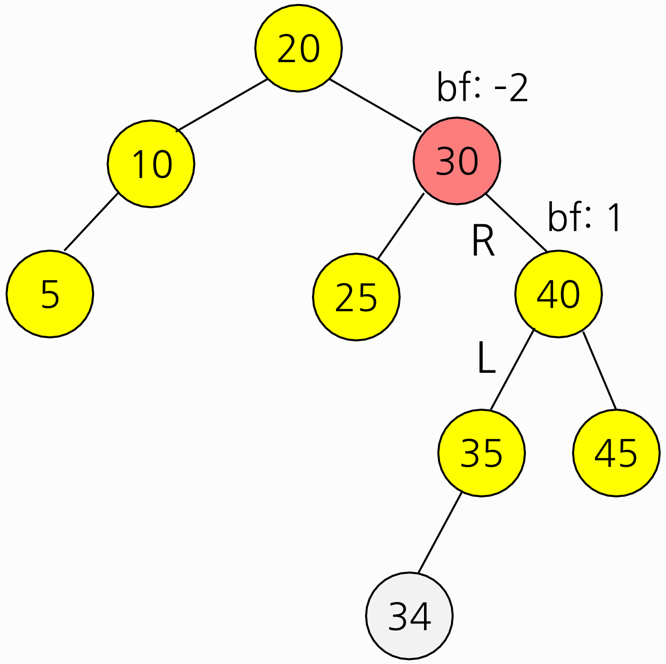
AVL Tree Balanced?

- **Insertion of 34**
- Imbalance at?
- Balance factor?



Double rotation RL case

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

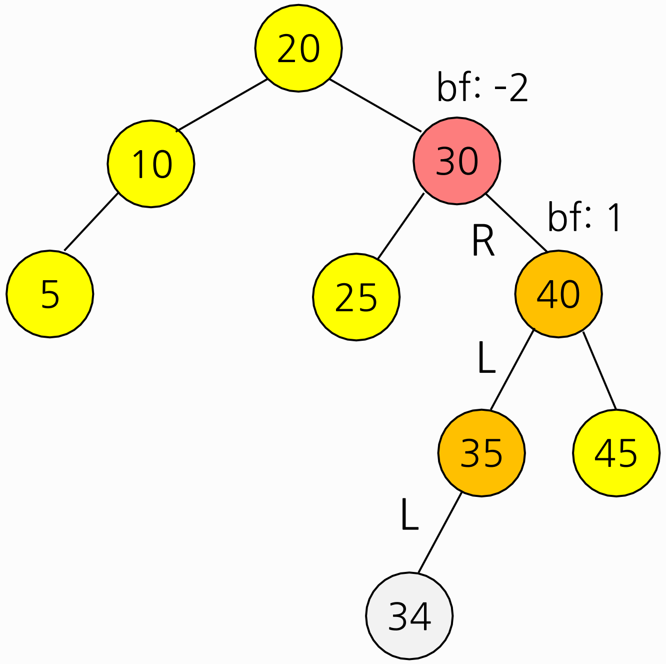


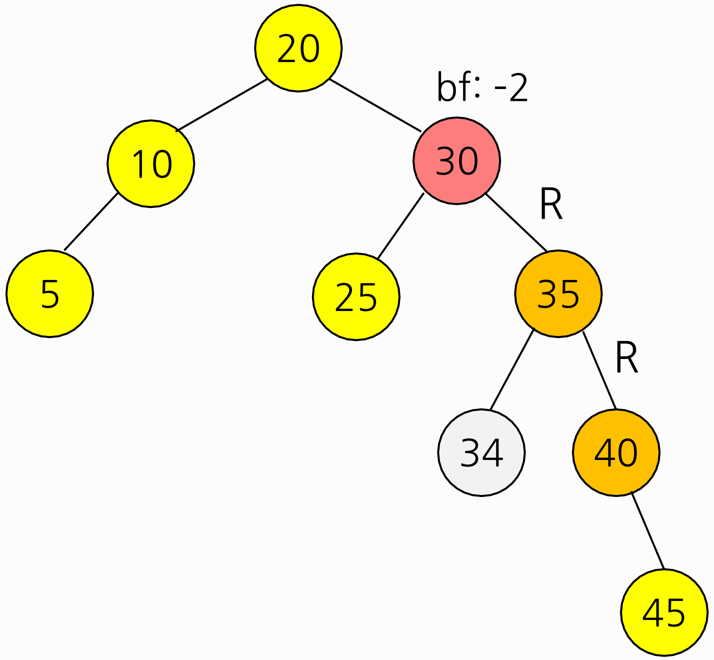
Double rotation RL case

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

- **RL case** (RR + LL cases)
  - Rotate at 40, LL case
  - Rotate at 30, RR case

Double rotation



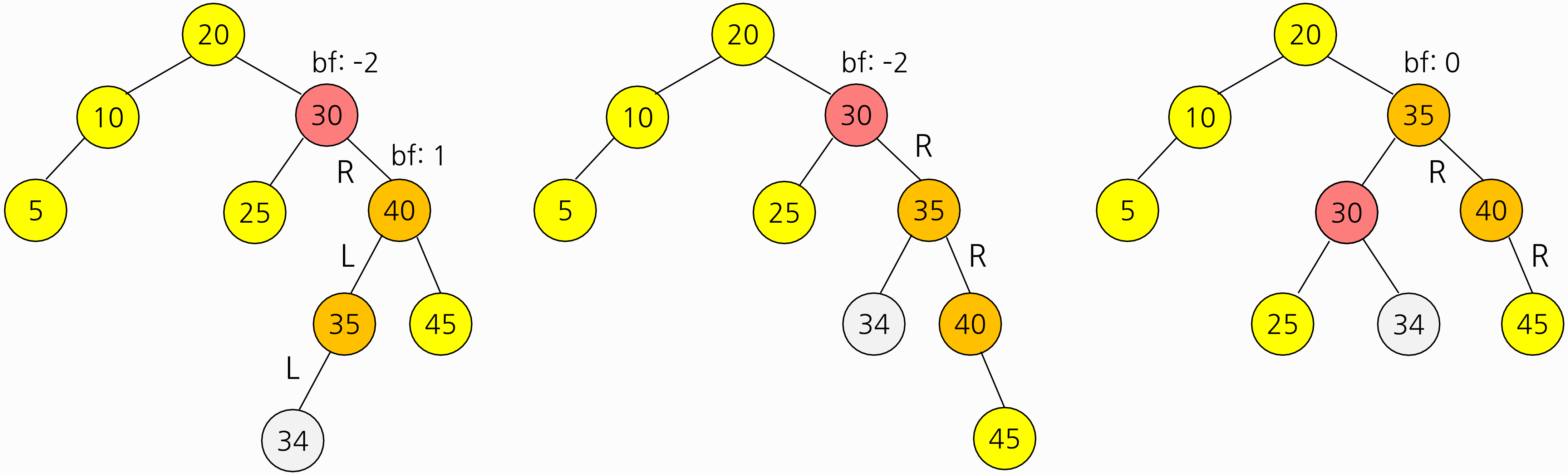


Double rotation RL case

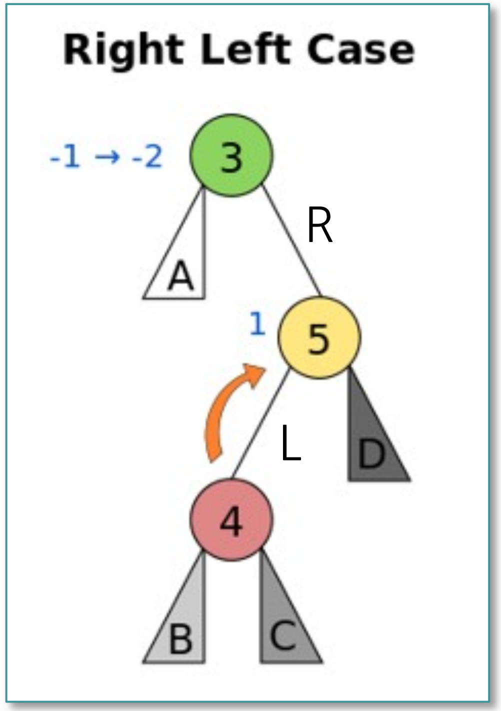
- **Insertion of 34**
- Imbalance at 30
- Balance factor - 2

- **RL case** (RR + LL cases)
  - Rotate at 40, LL case
  - Rotate at 30, RR case

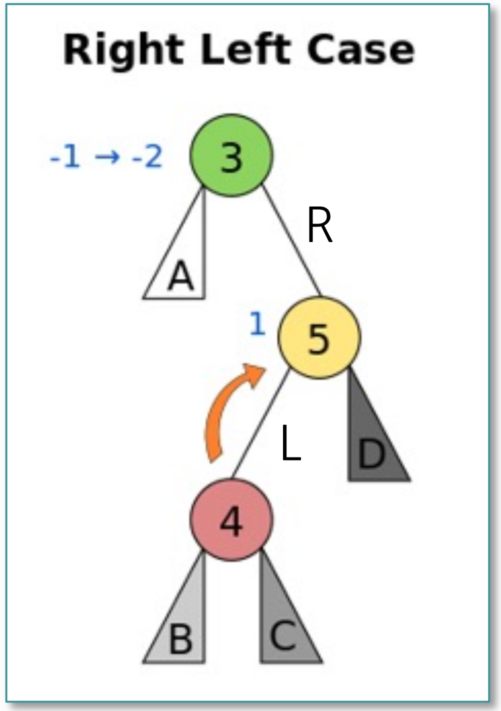
Double rotation

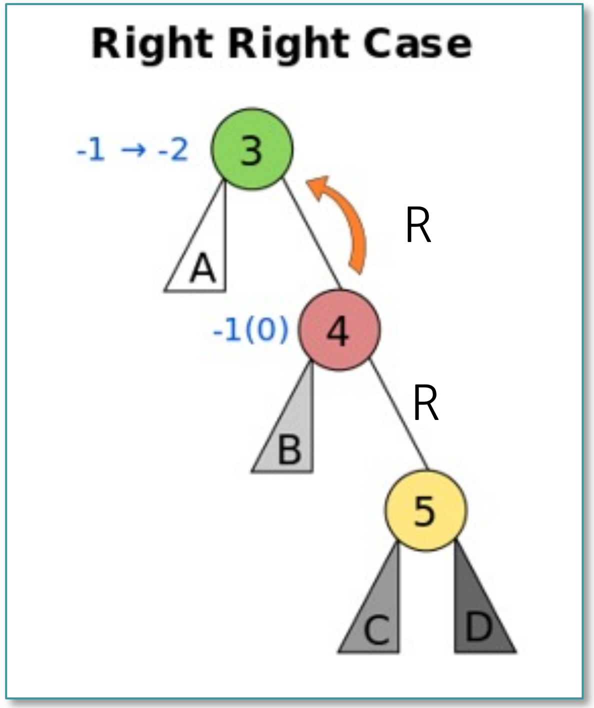


Double rotation - RL Case

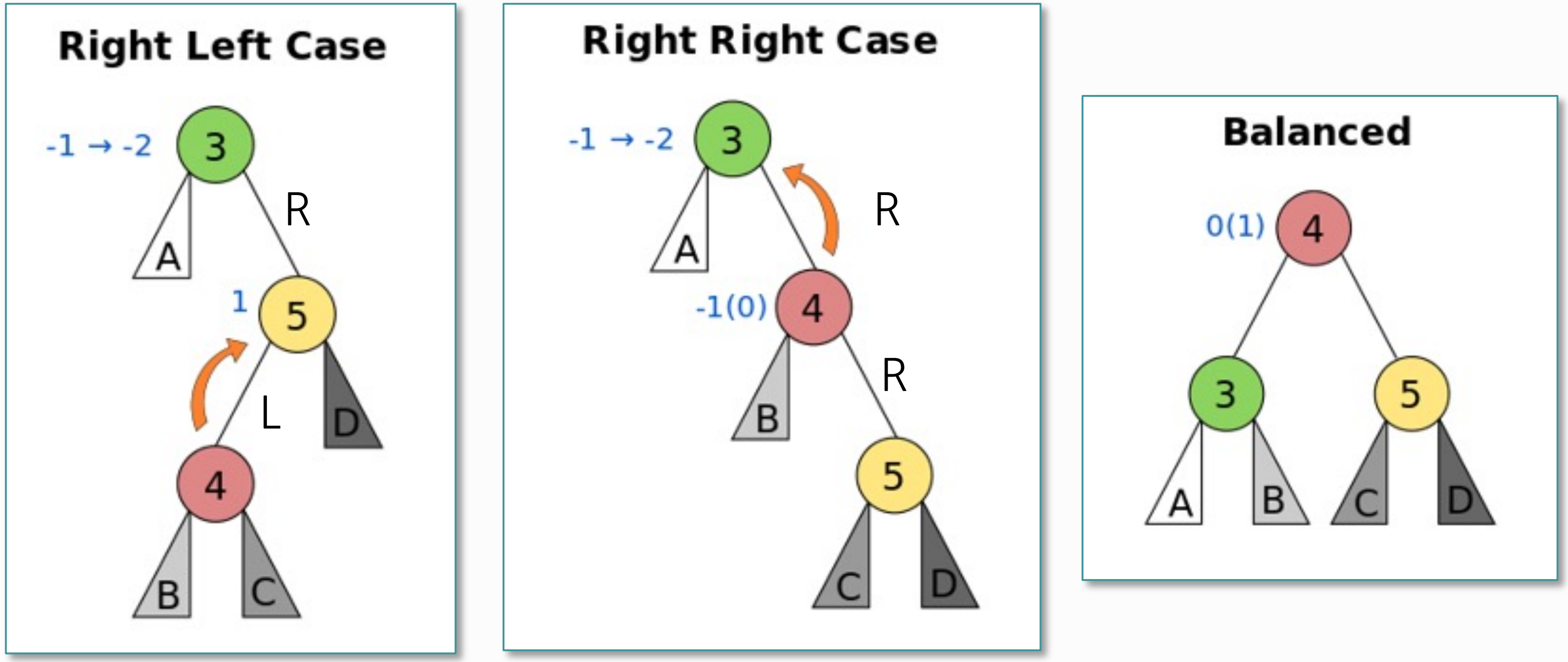


Double rotation - RL Case

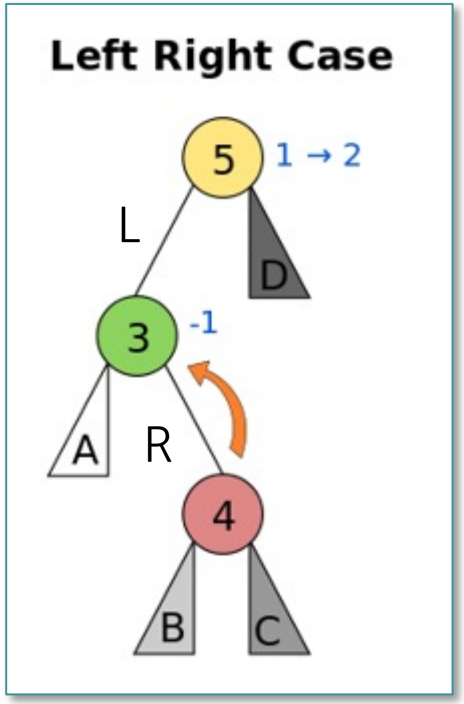


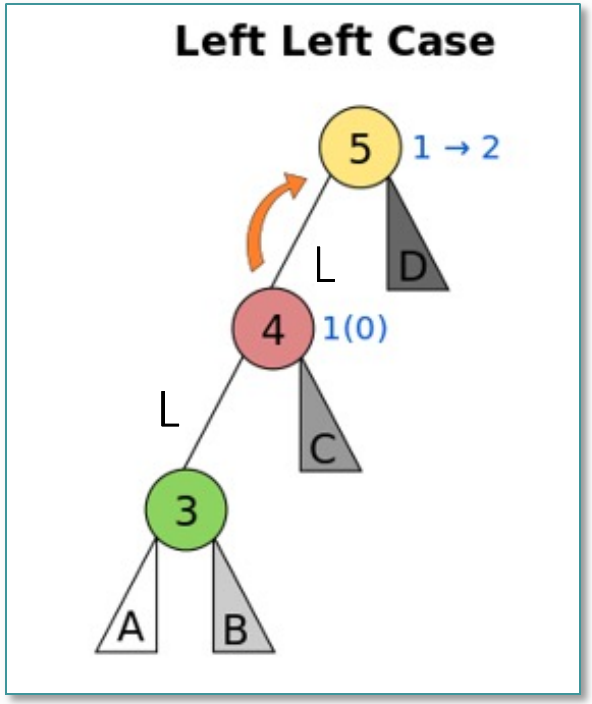


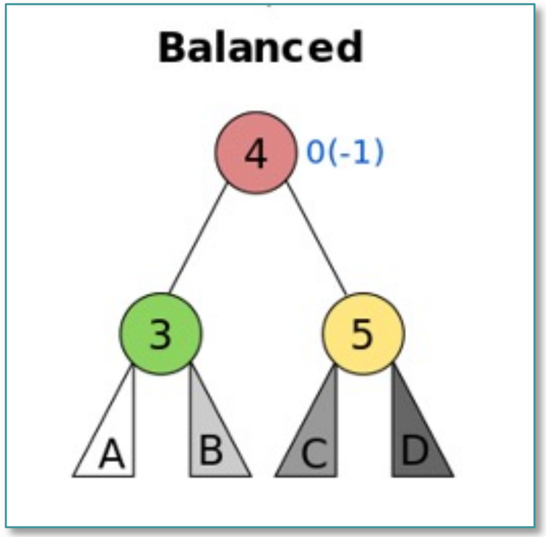
Double rotation - RL Case



Double rotation - LR Case





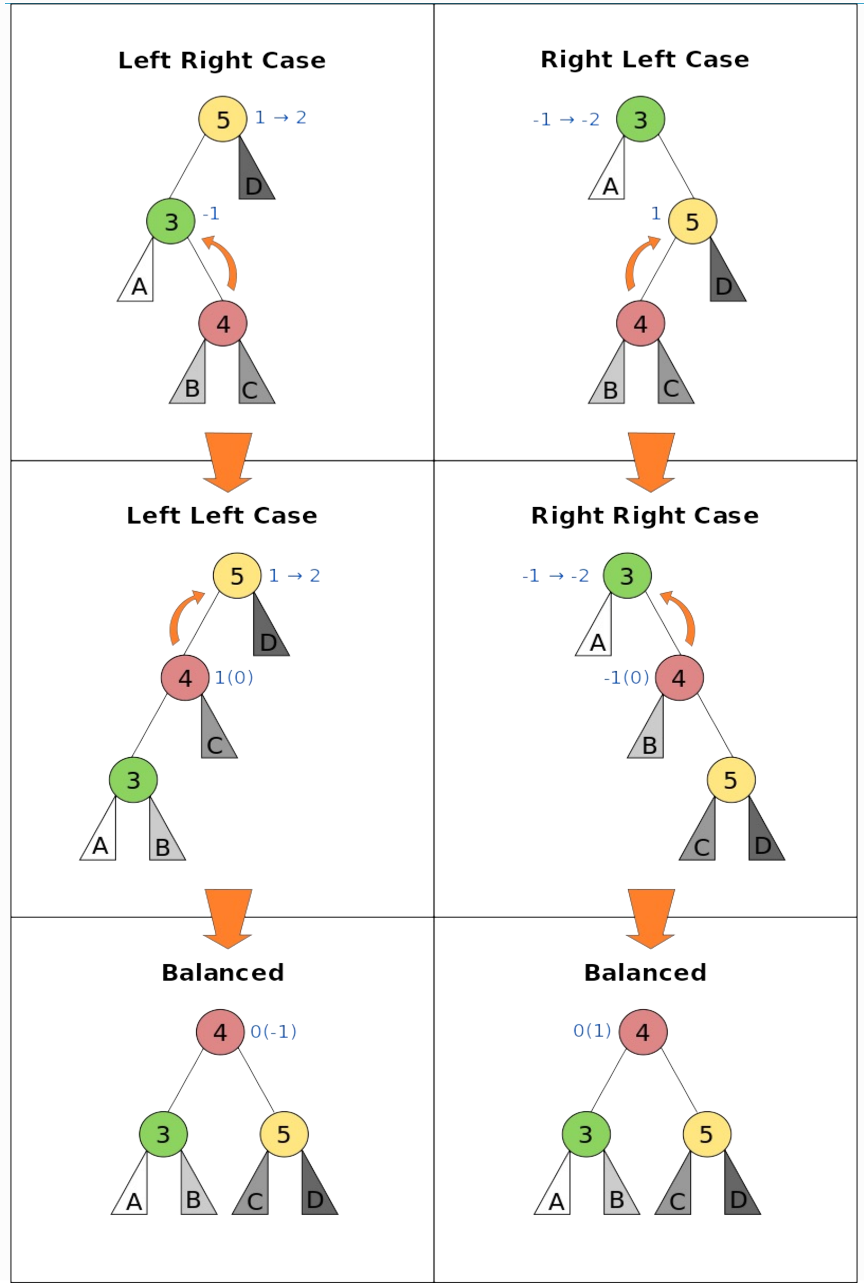


Time Complexity

- Since AVL trees are always balanced, the time complexity of AVL tree shows $O(\log_2 n)$ for most operations.

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	$O(n)$	$O(1)$	$O(n)$	$O(log_2 n)$
get	$O(log_2 n)$	$O(1)$	$O(n)$	$O(log_2 n)$
in	$O(log_2 n)$	$O(1)$	$O(n)$	$O(log_2 n)$
del	$O(n)$	$O(1)$	$O(n)$	$O(log_2 n)$

Summary (1/2)



- 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

Summary (2/2)

- AVL tree is a height-balanced binary search tree(BST).
- Arguments **for** AVL tree:
  - The time complexity of AVL tree shows $O(\log_2 n)$ for most operations.
  - The height balancing adds no more than a constant factor to the speed of insertion or deletion.
- Arguments **against** using AVL tree:
  - **Difficult** to program & debug

학습 정리

1) AVL 트리는 자동적으로 균형을 잡아주는 고전적인 이진탐색트리 알고리즘이다

2) Balance factor를 이용해 트리의 균형을 맞추는 작업의 시간복잡도는 O($\log n$)이다

3) 단일회전(Single rotation)에는 LL, RR case가 있고, 이중회전(Double rotation)에는 LR, RL case가 있다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다