



# 数据结构与算法 (Python版)

## AVL树的定义和性能

陈斌 北京大学 gischen@pku.edu.cn

# 平衡二叉查找树：AVL树的定义

❖ 我们来看看能够在key插入时一直保持平衡的二叉查找树：AVL树

AVL是发明者的名字缩写：G.M. Adelson-Velskii and E.M. Landis

❖ 利用AVL树实现ADT Map，基本上与BST的实现相同

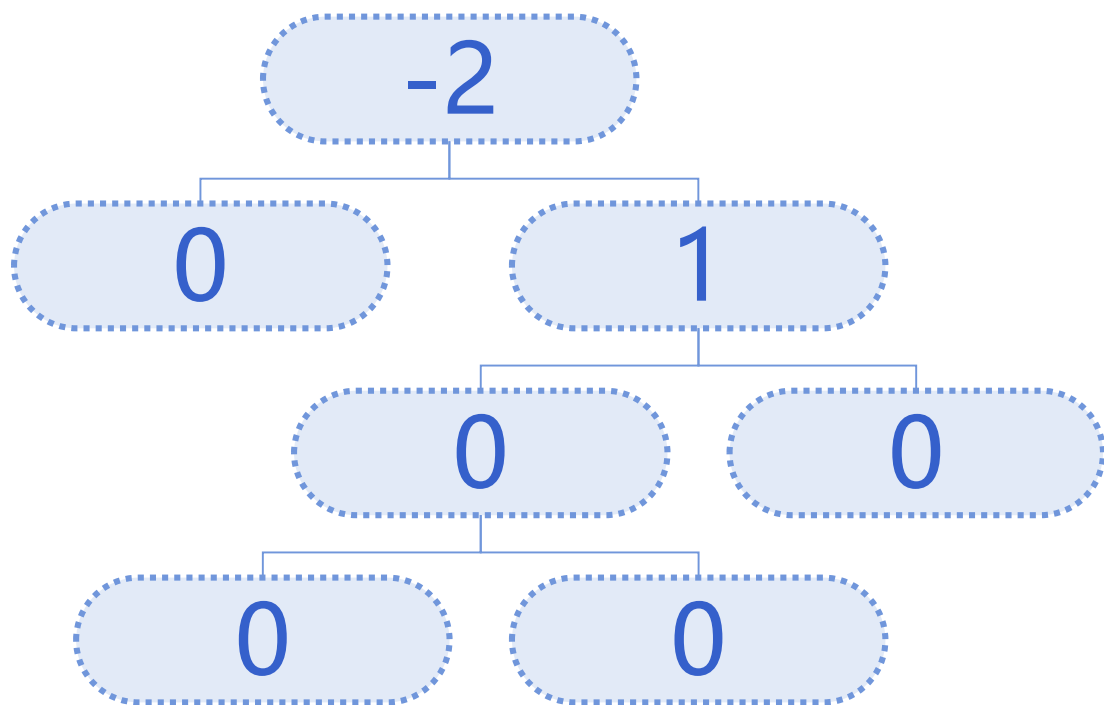
❖ 不同之处仅在于二叉树的生成与维护过程

# 平衡二叉查找树：AVL树的定义

- ❖ AVL树的实现中，需要对每个节点跟踪“平衡因子balance factor”参数
- ❖ 平衡因子是根据节点的左右子树的高度来定义的，确切地说，是左右子树高度差：  
$$\text{balanceFactor} = \text{height}(\text{LeftSubTree}) - \text{height}(\text{rightSubTree})$$
  
如果平衡因子大于0，称为“左重left-heavy”，  
小于零称为“右重right-heavy”  
平衡因子等于0，则称作平衡。

# 平衡二叉查找树：平衡因子

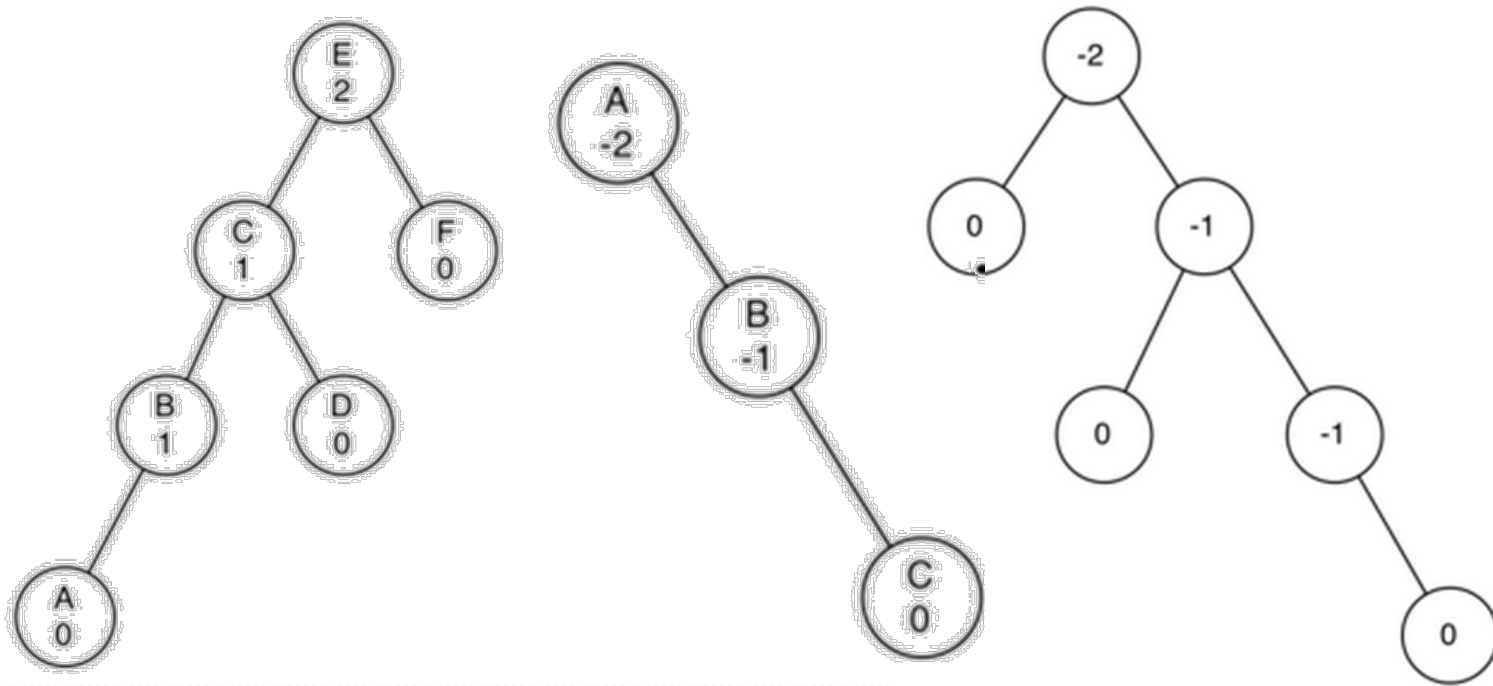
❖ 如果一个二叉查找树中每个节点的平衡因子都在-1, 0, 1之间, 则把这个二叉搜索树称为平衡树



# 平衡二叉查找树：AVL树的定义

❖ 在平衡树操作过程中，有节点的平衡因子超出此范围，则需要一个**重新平衡**的过程  
要保持BST的性质！

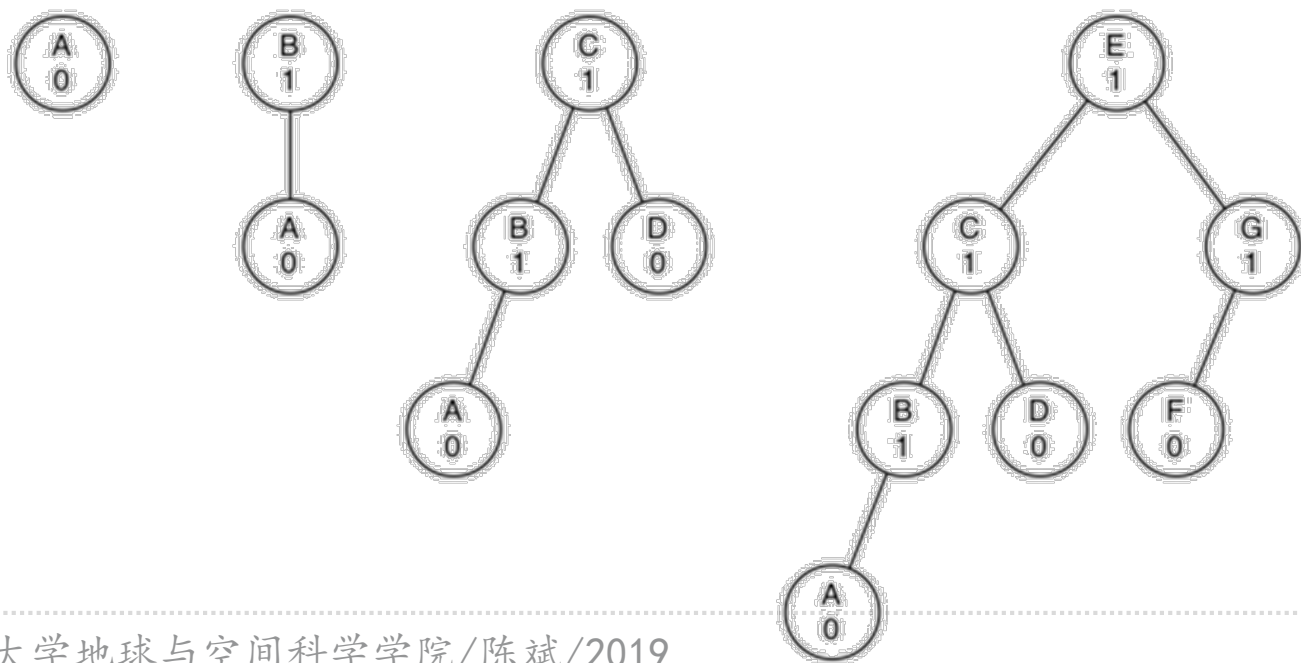
思考：如果重新平衡，应该变成什么样？



# 先来看看AVL树的性能

## ❖ 我们来看看AVL树最差情形下的性能：即平衡因子为1或者-1

下图列出平衡因子为1的“左重”AVL树，树的高度从1开始，来看看问题规模（总节点数N）和比对次数（树的高度h）之间的关系如何？



# AVL树性能分析

❖ 观察上图 $h=1 \sim 4$ 时，总节点数 $N$ 的变化

$$h=1, N=1$$

$$h=2, N=2=1+1$$

$$h=3, N=4=1+1+2$$

$$h=4, N=7=1+2+4$$

$$N_h = 1 + N_{h-1} + N_{h-2}$$

❖ 观察这个通式，很接近斐波那契数列！

# AVL树性能分析

## ❖ 定义斐波那契数列 $F_i$

利用 $F_i$ 重写 $N_h$

$$\begin{aligned} F_0 &= 0 & N_h &= 1 + N_{h-1} + N_{h-2} \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ for all } i \geq 2 & N_h &= F_{h+2} - 1, h \geq 1 \end{aligned}$$

## ❖ 斐波那契数列的性质： $F_i/F_{i-1}$ 趋向于黄金分割 $\Phi$

可以写出 $F_i$ 的通式

$$\Phi = \frac{1+\sqrt{5}}{2} \quad F_i = \Phi^i / \sqrt{5}$$



# AVL树性能分析

❖ 将 $F_i$ 通式代入到 $N_h$ 中，得到 $N_h$ 的通式

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

❖ 上述通式只有 $N$ 和 $h$ 了，我们解出 $h$

$$\log N_h + 1 = (H + 2) \log \Phi - \frac{1}{2} \log 5$$

$$h = \frac{\log N_h + 1 - 2 \log \Phi + \frac{1}{2} \log 5}{\log \Phi}$$

$$h = 1.44 \log N_h$$

❖ 最多搜索次数 $h$ 和规模 $N$ 的关系，可以说  
AVL树的搜索时间复杂度为 $O(\log n)$

