7，字典和集合-3

❖ 数据检索和字典，数据规模和检索，索引和字典
❖ 基于表和排序表实现字典和检索，二分检索
❖ 散列的概念，散列表和字典，散列函数和冲突消解，实现
❖ 集合数据结构，集合的实现，位向量实现
❖ Python 的字典和集合
❖ 二叉排序树的概念和实现
❖ 最佳二叉排序树，等概率和不等概率情况，构造算法
❖ 支持动态操作的排序树：AVL 树
❖ 支持字典实现的其他树形结构简介

最佳二叉排序树

- 基于平均检索长度评价二叉排序树的优或劣。需要参考
  - 各个关键码出现的概率
  - 成功和失败（关键码不存在）检索
- 概念：扩充二叉排序树的对称序列（中序遍历序列）：

  按中序遍历扩充二叉树得到的结点标记序列。其中内外部结点交叉排列，第 i 个内部结点位于第 i-1 个外部结点和第 i 个外部结点之间

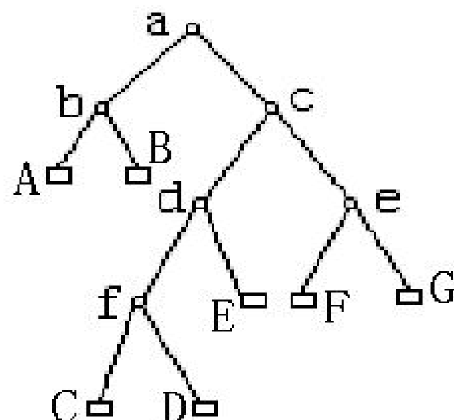
扩充的二叉排序树表示了一个字典的可能关键码集合：

内部结点代表已有元素的关键码

外部结点代表介于两个相邻内部结点的关键码之间的那些关键码

右图的中序周游序列：

AbBaCfDdEcFeG



最佳二叉排序树：平均检索长度

- 在扩充二叉排序树里，关键码的平均检索长度由下面公式给出

$$E(n) = \frac{1}{w}\left[\sum_{i=0}^{n-1} p_i(l_i+1) + \sum_{i=0}^{n} q_i l'_i\right]$$

其中 l_i 是内部结点 i 的层数，l'_i 是外部结点 i 的层数

p_i：检索内部结点 i 的关键码的频率，确定内部结点需比较层数加一次

q_i：被检索关键码属于外部结点 i 代表的关键码集合的频率

p_i，q_i 看作相应结点的权

w 是所有频率之和

$$w = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i$$

这样，p_i/w 就是检索内部结点 i 的关键码的概率

q_i/w 是被检索的关键码属于外部结点 i 的关键码集合的概率

最佳二叉排序树

- 最佳二叉排序树使检索的平均比较次数达到最少

  也即，它是使 $E(n)$ 的值达到最小的二叉排序树

- 问题：给定了一组数据及其分布，如何构造最佳二叉排序树？

先考虑简单情况：所有结点的检索概率相等

$$\frac{p_0}{w} = \frac{p_1}{w} = \cdots = \frac{p_{n-1}}{w} = \frac{q_0}{w} = \frac{q_1}{w} = \frac{q_2}{w} = \cdots = \frac{q_n}{w} = \frac{1}{2n+1}$$

$$\begin{aligned} E(n) &= \tfrac{1}{2n+1}\left[\textstyle\sum_{i=0}^{n-1} p_i(l_i+1) + \sum_{i=0}^{n} q_i l'_i\right] \\ &= (IPL + n + EPL)/(2n+1) \\ &= (2 \cdot IPL + 3n)/(2n+1) \end{aligned}$$

其中 $IPL = \sum_{k=1}^{n} \lfloor \log_2 k \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2$

内部路径长度 IPL 最小时这棵树达到最佳，也就是说，最低的树最好

最佳二叉排序树：简单情况的构造

- 等概率时一种构造方法是递归构造（左右子树结点均分法）

  设表 a 给定了按关键码排序的一组字典项

  0，令 low = 0, high = len(a)-1

  1，m = (high + low)//2;

  2，把 a[m] 存入所构造的二叉排序树的根结点 t，递归地：
  - 用同样方式由 a[low:m-1] 构造出二叉排序树作为 t 的左子树
  - 同样由 a[m+1:high] 构造出二叉排序树作为 t 的右子树
  - 如果切片为空就直接返回 None 表示空树
- 构造的时间代价：
  - 算法中构造最佳二叉排序树的部分的复杂性为 O(n)
  - 从有关排序的研究可知，n 个关键码排序的复杂性为 O(n log n)
  - 所以，整个构造过程的复杂性也是 O(n log n)

最佳二叉排序树：简单情况的算法

- 下面类的初始化方法从参数构造最佳二叉排序树，采用前述方法：

  参数可以是任何序列，只要求元素可以比较大小（< 运算符有定义）

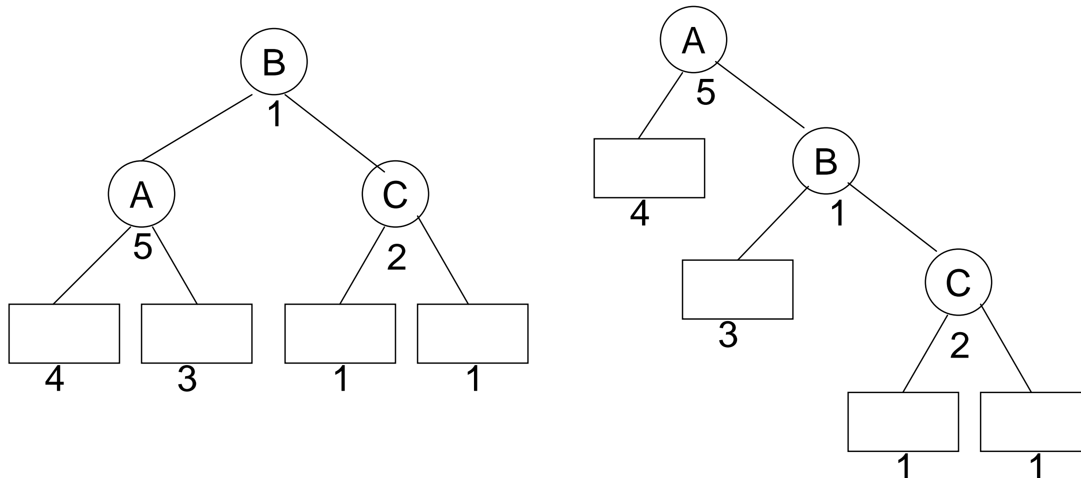
```
class DictOptBTree(DictBTree):
    def __init__(self, seq):
        DictBTree.__init__(self)
        data = sorted(seq)
        self._root = DictOptBTree.buildOBT(data, 0, len(data)-1)

    @staticmethod
    def buildOBT(data, start, end): # 注意，这里没有真做切片
        if start > end: return None
        mid = (end + start)//2
        left = DictOptBTree.buildOBT(data, start, mid-1)
        right = DictOptBTree.buildOBT(data, mid+1, end)
        return BiTNode(Assoc(*data[mid]), left, right)
```

- 其他操作继承自 DicBTree。简单插入新数据项则不再保证最佳性质

最佳二叉排序树：一般情况

- 一般情况中不同关键码的访问概率可能不同，构造过程复杂很多

  首先应看到，层数最小的树未必是最佳。例如：



E(n)=33/17 **E(n)=29/17**

结点处标的是频率，除以**17**得到检索达到各结点的“概率”

最佳二叉排序树：一般情况

- 问题描述：给定排序的关键码序列 $[key_0, key_1, \dots , key_{n-1}]$ 和两个权序列 $[p_0, \dots , p_{n-1}]$ 和 $[q_0, q_1, q_2, \dots , q_n]$
  - 其中的 p_i 是检索内部结点 **i** 的概率，q_j 是被检索关键码属于外部结点 **i** 的关键码集合的概率
  - 要求构造出一棵最佳二叉排序树，即构造出一棵二叉排序树，使下面的代价函数达到最小

$$E(0,n) = \sum_{i=0}^{n-1} p_i(l_i + 1) + \sum_{i=0}^{n} q_i l'_i$$

E(0, n) 表示包含 **n** 个内部结点和 **n+1** 个外部结点的树的代价

- 性质：最佳二叉排序树里的任何子树都是最佳的二叉排序树

  很容易证明（很简单，用反证法）

  易见：一棵最佳二叉排序树是两棵较小的最佳二叉排序树的组合。如果存在多种组合可能，就应该选其中的最佳组合

最佳二叉排序树：一般情况

- 设给定的排序关键码序列是 $key_0, key_1, \dots, key_{n-1}$，需构造的二叉排序树内部结点为 $v_0, \dots, v_{n-1}$，外部结点为 $e_0, e_1, \dots, e_n$。为讨论方便：
  - 用 **T(i, j)** 表示包含内部结点 $v_i, \dots, v_{j-1}$ 和外部结点 $e_i, e_{i+1}, \dots, e_j$ 的最佳二叉排序树（扩充二叉树的中序序列中的一段结点）
  - 例：**T(0, 1)** 表示包含内部结点 v_0 以及外部结点 e_0, e_1 的最佳二叉排序树；**T(2, 5)** 表示包含内部结点 v_2, v_3, v_4 以及外部结点 e_2, e_3, e_4, e_5 的最佳二叉排序树
- 下面先用一个例子说明构造过程，假设
  - 给定的关键码序列是 **[A, B, C]**
  - 内部和外部结点的权值序列分别为 **[5, 1, 2]** 和 **[4, 3, 1, 1]**
  - 需要构造的二叉排序树有三个内部结点，**4** 个外部结点
  - 要求做出相应的最佳二叉排序树

最佳二叉排序树：一般情况实例

关键码集合 {A, B, C}

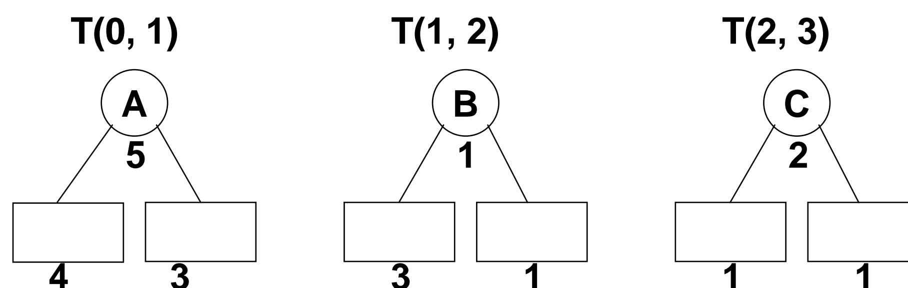
P = {5, 1, 2}

Q = {4, 3, 1, 1}

- 步骤 1：
  - 用 **T(0, 1)** 表示只含 e_0, v_0, e_1 的二叉排序树

    它自然最佳（这种二叉排序树只有一棵），代价是 $q_0+p_0+q_1$
  - **T(1, 2), … , T(n-1, n)** 自然也都是最佳二叉排序树（因为这些都是唯一的），相应的代价 **E(i, i+1)** 很容易计算

对实例，步骤 **1** 构造出 **3** 棵包含一个内部结点的最佳二叉排序树：



算出： **E(0, 1) = 12/17** **E(1, 2) = 5/17** **E(2, 3) = 4/17**

最佳二叉排序树：一般情况实例

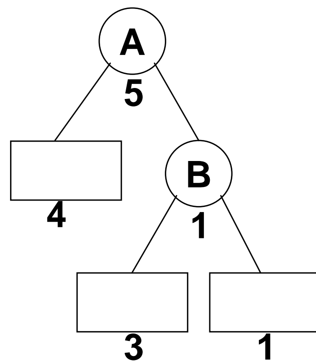
关键码集合 {A, B, C}

P = {5, 1, 2}

Q = {4, 3, 1, 1}

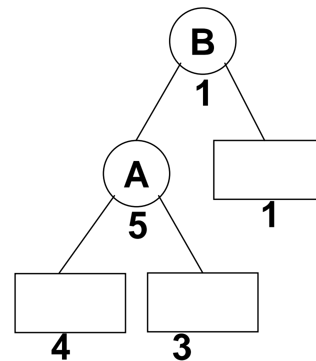
- 步骤 2：构造 T(0, 2)，T(1, 3)，各含两个内部结点
- 构造 T(0, 2) 有两种可能：
  - 以 v_0 为 根，e_0为左子树，T(1, 2) 为右子树
  - 以 v_1 为 根，T(0, 1) 为左子树，e_2为右子树
  - 比较哪种构造方式得到的树最佳，就选它作为 T(0, 2)

候选$_1$



$E(0, 2)_1 = 19/17$

候选$_2$



$E(0, 2)_2 = 26/17$

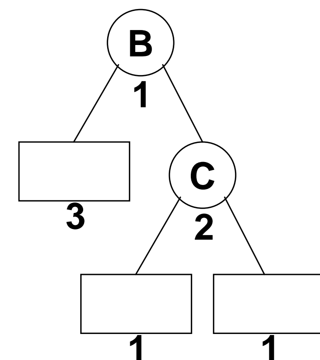
最佳二叉排序树：一般情况实例

关键码集合 {A, B, C}

P = {5, 1, 2}

Q = {4, 3, 1, 1}

- 可按类似方式的构造出 T(1, 3)
  - 两种可能性：
    - 以 v_1 为根，e_1 为左子树，T(2, 3) 为右子树
    - 以 v_2 为根，T(1, 2) 为左子树，e_3 为右子树
  - 前一个更优，就选它作为 T(1, 3)
  - 可算出 E(1, 3) = 12/17



- 每次都需要重头开始计算 E 值吗？
- 其实不必！可以递推计算。递推公式不难推出（下面会给出）

  信息的来源：两棵子树及其 E 值，根结点的相关信息

最佳二叉排序树：一般情况实例

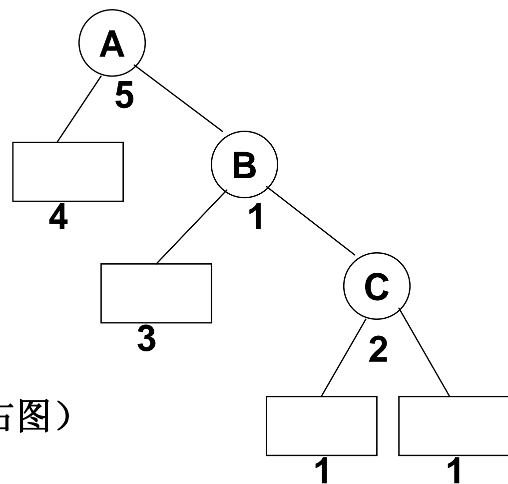
关键码集合 {A, B, C}

P = {5, 1, 2}

Q = {4, 3, 1, 1}

- 步骤 3，构造包含三个内部结点的最佳二叉排序树 T(0, 3)，即本问题最终结果。三棵候选树：
  - 以 v_0 为根，其左子树为 e_0，右子树为 T(1, 3)
  - 以 v_1 为根，其左子树为 T(0, 1) 右子树为 T(1, 3)
  - 以 v_2 为根，其左子树为 T(0, 2)，右子树为 e_3
- 计算这 3 棵二叉排序树的 E 值

  从中选出最佳，就得到了最终结果（右图）



- 对任意一组给定数据，
  - 总可以这样逐步构造一组最佳二叉排序树
  - 最终将构造出所需要的最佳二叉排序树

最佳二叉排序树：一般情况构造

关键码集合 {A, B, C}

P = {5, 1, 2}

Q = {4, 3, 1, 1}

- 一般步骤（第 m 步）：
  - 当时对所有的 0 < i < m，已构造好下面所有的最佳二叉排序树：

    T(0, i), T(1, 1+i), T(2, 2+i), …
  - 基于它们可构造出 n-m+1 棵包含 m 个结点的最佳二叉排序树：

    T(0, m), T(1, 1+m), …

    每次都是考虑所有可能构造的二叉排序树，选出最佳的一棵
- 下面算法的基本想法就是逐步构造最佳二叉排序树的子树，最终构造出包含所有结点的最佳二叉排序树。构造过程：
  - 步骤 1：构造所有包含一个内部结点的 T(0,1), T(1,2), …,T(n-1,n)
  - 步骤 2：构造所有包含二个内部结点 T(0,2), T(1,3), …,T(n-2,n)
  - ……
  - 步骤 n：构造T(0,n)

最佳二叉排序树：一般情况构造

现在考虑所构造的二叉排序树的代价的计算

- 考虑关键码为 $key_i, key_{i+1}, \ldots, key_{j-1}$ 的内部结点，相应内/外部结点的权值序列为 $q_i, p_i, q_{i+1}, \ldots, p_{j-1}, q_j$

  用 R(i, j) 表示最佳二叉排序树 T(i, j) 的根

  分段权和：$W(i, j) = p_i+\ldots+ p_{j-1} + q_i+ q_{i+1} +\ldots + q_j$ $(0 \leqslant i \leqslant j \leqslant n)$

  在构造 T(i, j) 时，对所有 i < k < j-1，T(i, k) 和 T(k+1, j) 都存在

  而且它们的代价已知，分别记为 C(i, k) 和 C(k+1, j)

- 对每个 k，以 key_k 的内部结点为根 T(i, k) 和 T(k+1, j) 为左右子树的二叉树权为 $C_k(i, j) = W(i, j) + C(i, k)+C(k+1, j)$ （所有结点增加一层）

  从所有可能 k 值中选择使 $C_k(i, j)$ 达到最小的那个 k，与之对应二叉树就是 T(i, j)，其根 R(i, j) = k

- 即，新树的代价可以从构造中使用的已知最佳二叉排序树的代价算出：

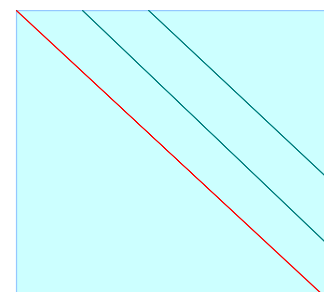
  $C(i, j) = W(i,j) + \min\{ C(i,k)+C(k+1,j) \mid i < k < j-1\}$

最佳二叉排序树：算法设计

- 考虑构造最佳二叉排序树的算法
- 下面的基本想法是用几个矩阵（二维表）记录构造过程中产生的信息，以表示数据段范围的 i 和 j 作为保存相应信息的矩阵元素下标
  - r[i][j] 记录构造出的最佳子树 T(i, j) 的根结点下标
  - c[i][j] 记录子树 T(i, j) 的代价
  - w[i][j] 表示子树的内外交错结点段 [i, j] 的权值之和
- 开始时知道 w 的对角线元素 w[i][i]
  - 首先算出 w 的其他元素（矩阵上三角的元素）
  - 算法逐步算出矩阵 r 和 c 左上各副对角线元素
  - 直至计算出

    r(0, n) 是构造出的结果树的根

    c(0, n) 是构造出的结果树的权



最佳二叉排序树：示例

关键码 [A, B, C]

p = [5, 1, 2]　　q = [4, 3, 1, 1]

w, r, c 为4×4数组

交替的权值序列 4 5 3 1 1 2 1

直接算出 w :

4	12	14	17
0	3	5	8
0	0	1	4
0	0	0	1

第一步

r :

0	0	0	0
0	0	1	0
0	0	0	2
0	0	0	0

c :

0	12	0	0
0	0	5	0
0	0	0	4
0	0	0	0

第二步

r :

0	0	0	0
0	0	1	1
0	0	0	2
0	0	0	0

c :

0	12	19	0
0	0	5	12
0	0	0	4
0	0	0	0

第三步

r :

0	0	0	0
0	0	1	1
0	0	0	2
0	0	0	0

c :

0	12	19	29
0	0	5	12
0	0	0	4
0	0	0	0

最佳二叉排序树：算法

- 数据安排：
  - 表参数 p (0..n-1) 和 q (0..n) 给定内部结点和外部结点的权
  - c, w, r 为 (n+1)*(n+1) 个元素的二维数组，只用上三角部分
  - w[i][j] 记录 W(i, j) 值（一段权之和），可直接算出
  - c[i][j] 和 r[i][j] 分别记录最佳二叉排序树 T(i,j) 的代价和根
- 构造好的最佳二叉排序树
  - 总代价保存在 c[0][n]
  - r[0][n] 是根结点编号。根据 r[0][n] 的值可确定树结构
- 例：假定内部结点为 v0 到 v7，r[0][8] 的值是 4，可知
  - 树根是结点 4，其左子树的根结点的编号保存在 r[0][4]，其右子树的根结点的编号保存在 r[5][8]
  - 按这种方式可以通过追溯得到构造出的树的结构

最佳二叉排序树：算法

```
def build_opt_btree(wp, wq):
    num = len(wp)+1
    w = [[0 for i in range(num)] for j in range(num)]
    c = [[0 for i in range(num)] for j in range(num)]
    r = [[0 for i in range(num)] for j in range(num)]
    for i in range(num): # 先算出所有 w[i][j]，0 <= i < j < num
        w[i][i] = wq[i]
        for j in range(i+1, num): w[i][j] = w[i][j-1] + wp[j-1] + wq[j]
    for i in range(0, num-1): # 设置一个结点的树
        c[i][i+1] = w[i][i+1]; r[i][i+1] = i
    for m in range(2, num):# 计算 m 结点的最佳二叉排序树（n–m+1棵）
        for i in range(0, num-m):
            k0, j, wmin = i, i+m, inf
            for k in range(i, j): # 在[i,j)里找使 C[i][k]+C[k+1][j] 最小的 k
                if c[i][k] + c[k+1][j] < wmin:
                    wmin = c[i][k] + c[k+1][j]; k0 = k
            c[i][j] = w[i][j] + wmin; r[i][j] = k0
    return (c, r)
```

最佳二叉排序树：算法分析

- **build_opt_btree** 就是前面算法的实现，其结果给出了树的全部信息
- 算法的时间复杂性为 $O(n^3)$，对较大的 n 非常耗时；空间为 $O(n^2)$
    - 任意权值的最佳二叉排序树有一定意义，但更多是理论价值，说明即使问题如此复杂，还是存在一般性的构造方法
    - 实际中使用很少。一是创建这种树的代价比较高，二是实际中很难得到有价值的访问分布情况
- 这个算法很典型，采用的算法模式称为“动态规划”
    - 动态规划算法的特点：在计算过程中维持一大批子问题的解（在这里维护的是最佳二叉树），基于对小的子问题的解，逐步构造更大子问题的解，最终构造出整个问题的解
    - 动态规划法在算法设计中使用广泛。实际上，前面介绍 Dijkstra 最短路径算法也应看作是一种动态规划算法
    - 这种方法在设计复杂的算法时非常有用

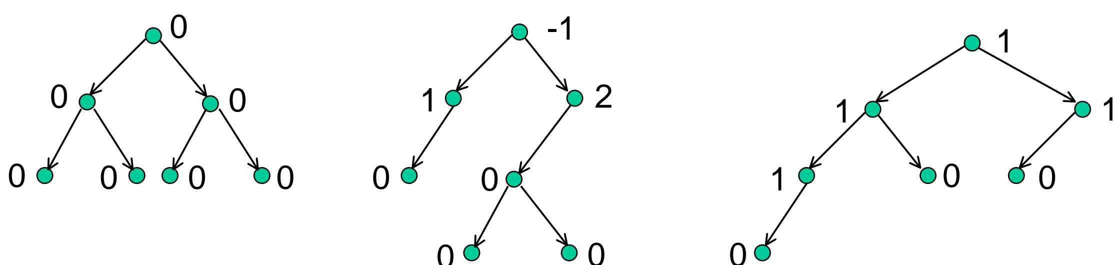
平衡二叉排序树

- 最佳二叉排序树的查询效率高。但这种结构需要根据所有元素的情况静态构造，适合作为静态字典表示，不能很好地支持动态变化
- 原因：最佳是全局性质，难以在局部把握、检查和调整
  - 对最佳二叉排序树做一系列插入删除，如果不维护，树结构就可能不断恶化而导致性能下降，最坏检索性能可能趋向于 **O(n)**
  - 在变动中，最佳性质很难维护，维护代价很高，因为一次插入删除可能造成大范围的影响（一定能做，但代价过高就不值得了）
- 为有效实现动态字典，人们提出了一些支持字典动态操作的树形结构
  - 共性都是放弃最佳性质，设法提供某些接近最佳的性质（树高与结点数成对数），换取在动态操作中较易通过局部调整进行维护
  - 设法通过局部调整维护树的较好结构（近似最佳的结构性质）
  - 各种操作（无论是检索还是插入删除）的最坏情况性能都有保证，操作都能在一条路径上完成（因此可以保证是 **O(log n)**）

平衡二叉排序树

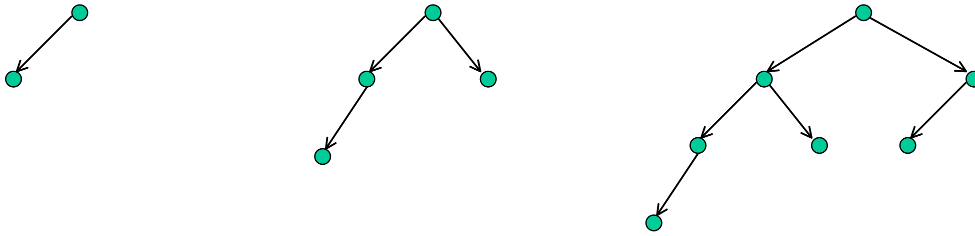
- 下面主要介绍平衡二叉排序树（**AVL**树，由前苏联 **Georgy Adelson-Velsky** 和 **E. M. Landis** 发明），类型结构还有红黑树，**B** 树等
- 平衡二叉排序树的基本想法：如果每个结点左右子树的高度差不多（平衡），整个树的结构也会比较好，不会出现特别长的路径
- 定义：平衡二叉排序树或是空树，或其左右子树都是平衡二叉排序树，而且左右子树的高度之差的绝对值不超过 **1**（局部性质，局部描述）

  平衡由各结点的情况刻画，用简单平衡因子 **BF**（**Balance Factor**）描述。**BF** 定义为结点左右子树的高度之差，可能取值只有 **-1, 0, 1**
- 平衡和不平衡的二叉排序树示例（结点旁所标数值为**BF**）



平衡二叉排序树

- AVL 树的形态分析，高度为 h 结点最少的（最接近失衡的） AVL树：



- 这种树的结点个数：$N_1 = 2$, $N_2 = 4$, $N_{h+2} = N_{h+1} + N_h + 1$

  很像斐波纳契序列 $F_0 = 0, F_1 = 1, F_{h+2} = F_{h+1} + F_h$

  易证（用数学归纳法） $N_h = F_{h+3} - 1$

- 对于斐波纳契序列，已知渐进公式 $F_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h$

  由此可以得到结论：n 个结点的 AVL 树，其高度 $h = O(\log_2 n)$。实际计算可知 $h \leqslant 3/2 \log_2(n+1)$

平衡二叉排序树

- 在 AVL 树上检索，时间代价的最坏情况受限于树中最长路径的长度，与最佳二叉排序树中最长路径的长度仅差一个常量因子
- AVL 树的检索与普通二叉排序树相同。问题是要在结构的动态变化中维持平衡，插入删除不但要保持树结构和结点序，还要维持平衡

  插入和删除操作可能破坏树的平衡，而后就需要调整恢复。如果这种调整是局部的，就可能保证操作的时间代价还是 O(log n)

- AVL树动态操作（插入/删除）的方式：
  - 按关键码顺序要求确定位置插入结点（或删除结点）；
  - 如果树失衡，则进行局部调整，恢复树的平衡
- 下面可以看到，插入和删除操作及其调整都可以在树中的一条路径上完成。由于 AVL 树的性质，插入删除的时间开销为 O(log n)
- 为实现 AVL 树，每个结点需要增加一个平衡因子记录

  下面考虑创建一个 AVL 树类，研究插入和删除操作并在类中实现

AVL 结点类和 AVL 树类

- AVL 树结点需要增加一个 bf 域，叶结点的 bf 值是 0

```
class AVLNode(BiTNode):
    def __init__(self, data):
        BiTNode.__init__(self, data)
        self.bf = 0
```

- AVL 树是一种二叉排序树，继承 DictBTree，许多方法都可以继承，但实现插入和删除操作的方法需要重新定义

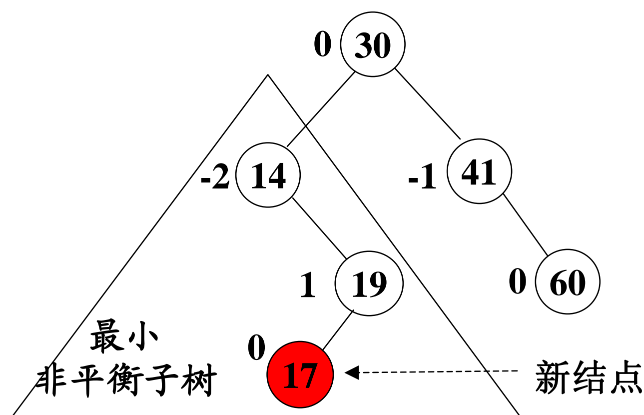
  初始化方法建立一棵空 AVL 树，这棵树自然是平衡的。如果需要，也可以给定一个初始化序列，但如果真的那样定义，就需要调用本类的 insert 方法（后面定义）把数据逐项插入树中

```
class DictAVL(DictBTree):
    def __init__(self):
        self._root = None
```

- 插入和删除方法的实现，是下面讨论的问题

插入中的失衡与调整

- 平衡的树插入一结点后可能使树中某个局部失衡，这时必须调整
- 最小非平衡子树：距插入结点最近且其根的 BF 非 0 的那棵子树。一般都存在（除非路径上所有结点的 BF 都是 0），设其根结点为 A
  - 如果通过调整可以使这棵最小非平衡子树恢复平衡，而且使其恢复到插入新结点之前的高度，那么整棵树也恢复平衡了
  - 下面说明如何局部恢复，使插入操作的复杂性不超过 O(log n)

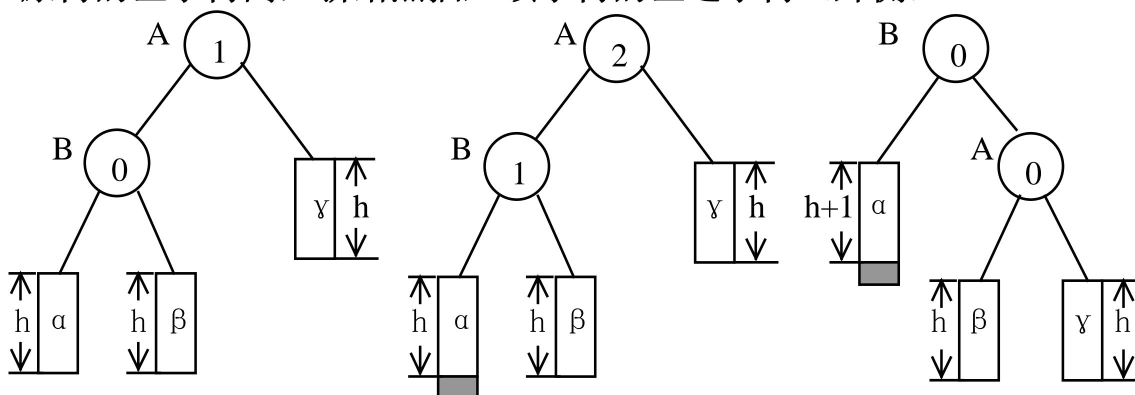


根据要调整的树的具体情况，调整动作分为四种：

1. LL型调整（左高插入左边）
2. RR型调整（右高插入右边）
3. LR型调整（左高插入右边）
4. RL型调整（右高插入左边）

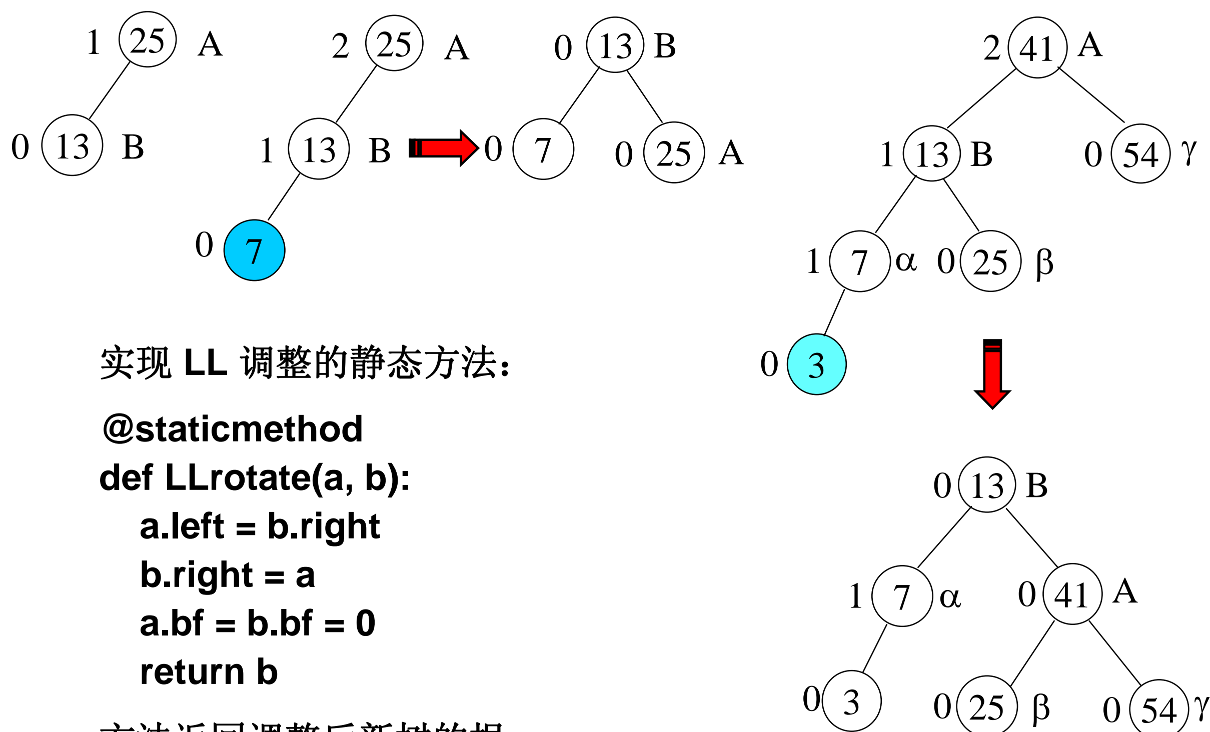
LL 调整

- 原树的左子树高，新结点插入该子树的左边子树（外侧）



- 调整规则：
  - A 的左子树 B 提升为新二叉树的根
  - A（连同其右子树 γ ）向右下旋转作为 B 的右子树
  - B 原来的右子树 β 作为 A 的左子树
- 中序遍历序列不变 (α B β)A(γ)= (α)B(β A γ)，新树与原树同高

LL 调整：实例和函数



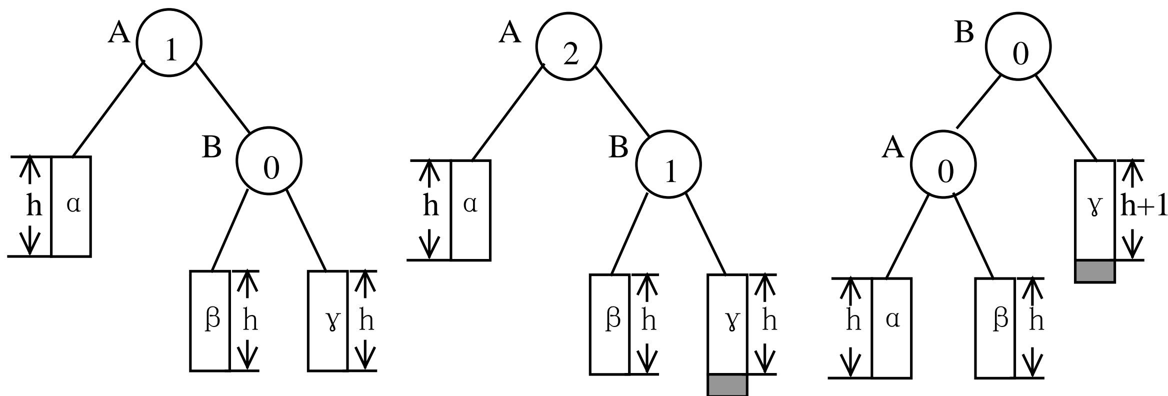
实现 LL 调整的静态方法：

```
@staticmethod
def LLrotate(a, b):
    a.left = b.right
    b.right = a
    a.bf = b.bf = 0
    return b
```

方法返回调整后新树的根

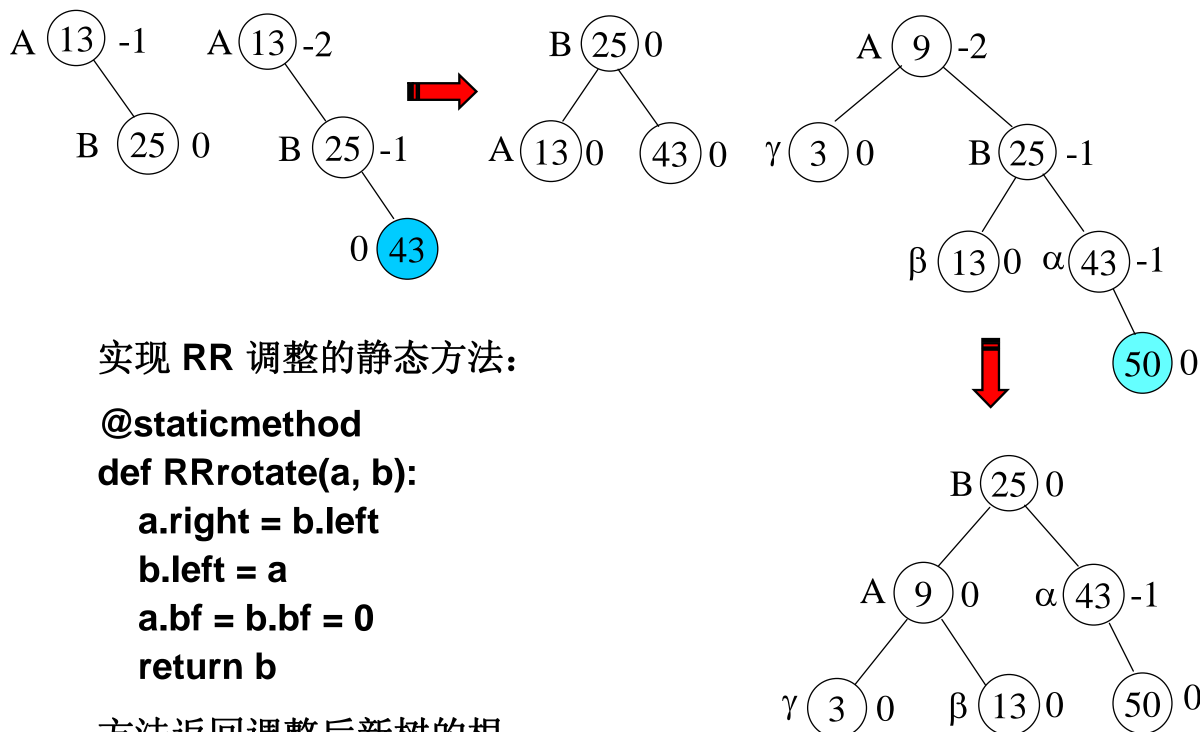
RR 调整（与 LL 对称）

- 原树的右子树高，新结点插入该子树的右边子树（外侧）



- 调整规则：
  - A 的右子树 B 提升为新二叉树的根
  - A（连同其左子树 α ）向左下旋转作为 B 的左子树
  - B 原来的左子树 β 作为 A 的右子树
- 中序遍历序列不变 (α)A(β B γ) = (α A β)B(γ)，新树与原树同高

RR 调整：实例和函数



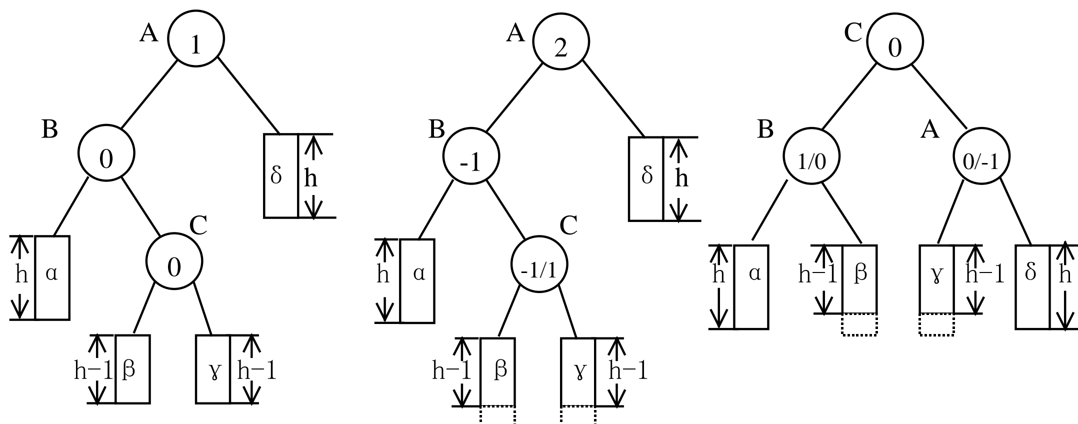
实现 RR 调整的静态方法：

```
@staticmethod
def RRrotate(a, b):
    a.right = b.left
    b.left = a
    a.bf = b.bf = 0
    return b
```

方法返回调整后新树的根

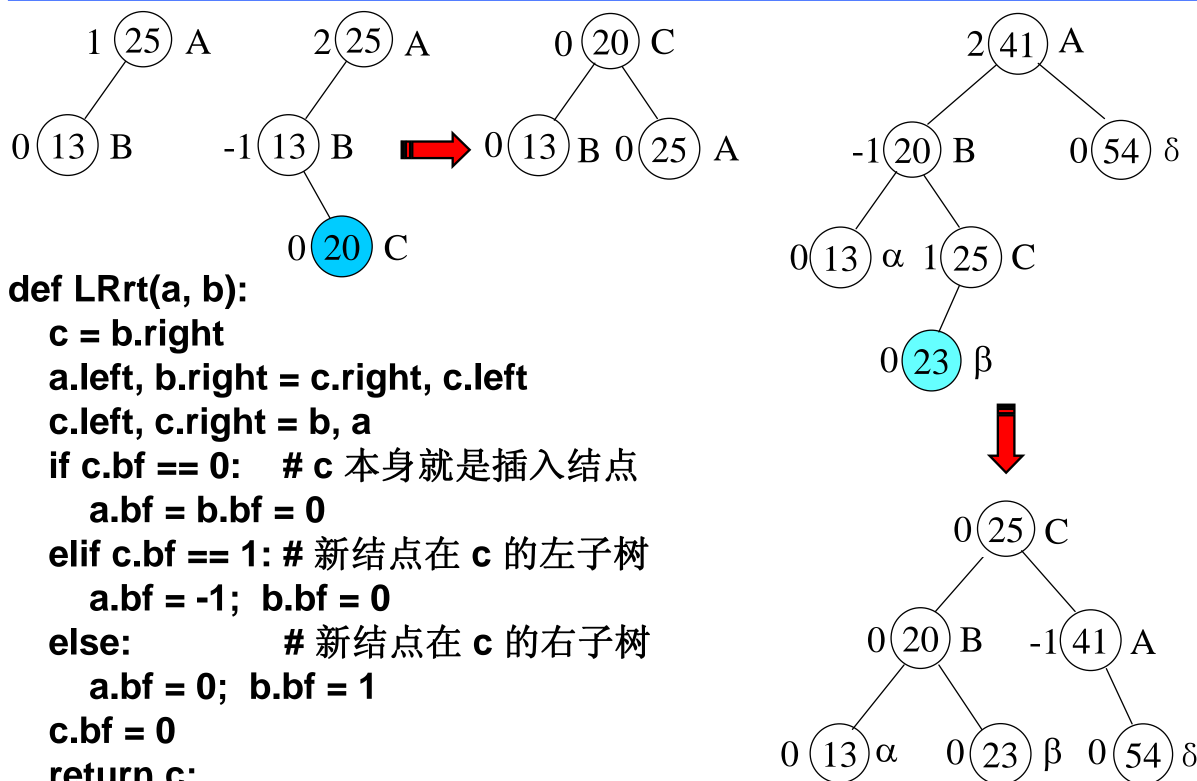
LR 调整

- 原树的左子树高，新结点插入该子树的右边子树（内侧）



- 调整规则：设 **C** 为 **A** 的左子树 **B** 的右子结点，将 **C** 提升为新树的根；**C** 原父结点**B**（连同其左子树 α ）向左下旋转作为 **C** 的左子树，**C** 原左子树 β 作为 **B** 的右子树；**A**（连同其右子树 δ ）向右下旋转作为 **C** 的右子树，**C** 的原右子树 γ 作为 **A** 的左子树
- 显然 **((** α **)B(** β **C** γ **))A(** δ **)= (** α **B** β **)C(** γ **A** δ **)**，且新树与原树同高

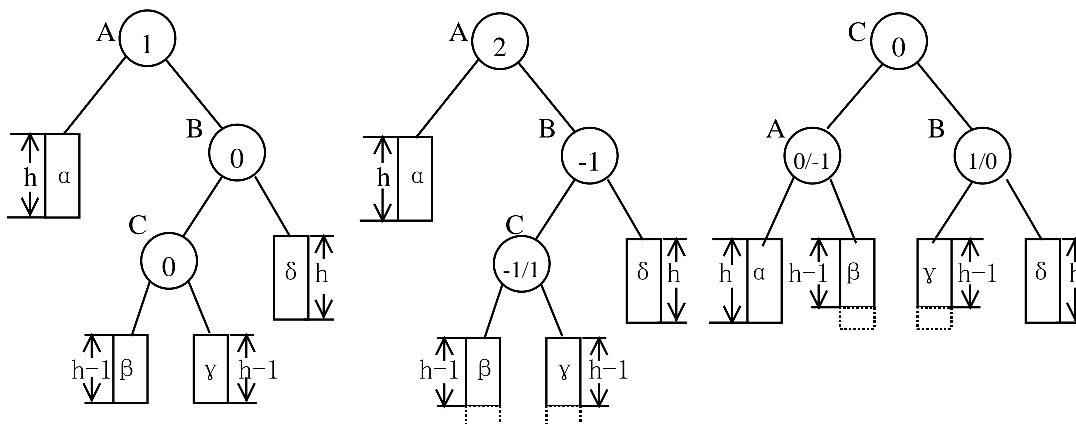
LR 调整：实例和算法




```
def LRrt(a, b):
    c = b.right
    a.left, b.right = c.right, c.left
    c.left, c.right = b, a
    if c.bf == 0:    # c 本身就是插入结点
        a.bf = b.bf = 0
    elif c.bf == 1: # 新结点在 c 的左子树
        a.bf = -1;  b.bf = 0
    else:            # 新结点在 c 的右子树
        a.bf = 0;  b.bf = 1
    c.bf = 0
    return c;
```

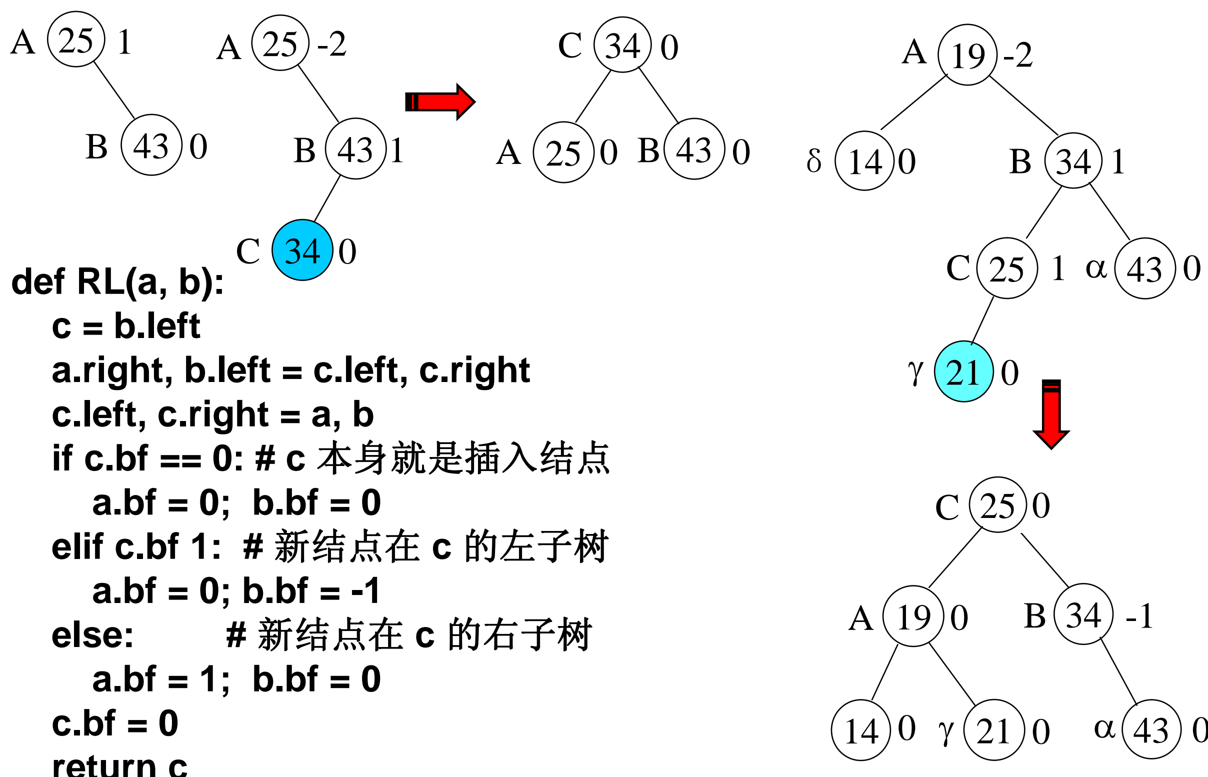
RL 调整

- 原树的右子树高，新结点插入该子树的左边子树（内侧）



- 调整规则：设 **C** 为 **A** 右子树 **B** 的左子结点，将 **C** 提升为新树的根，**C** 的原父结点 **B**（连同其右子树 δ）向右下旋转作为 **C** 的右子树，**C** 原右子树 γ 作为 **B** 的左子树；**A** 连同其左子树 α 向左下旋转作为 **C** 的左子树，**C** 原来的左子树 β 作为 **A** 的右子树
- 显然**(** α **)A((** β **C** γ **)B(** δ **))= (** α **A** β **)C(** γ **B** δ **)**，且新树与原树同高

RL 调整：实例和算法




```
def RL(a, b):
    c = b.left
    a.right, b.left = c.left, c.right
    c.left, c.right = a, b
    if c.bf == 0: # c 本身就是插入结点
        a.bf = 0;  b.bf = 0
    elif c.bf 1:  # 新结点在 c 的左子树
        a.bf = 0; b.bf = -1
    else:         # 新结点在 c 的右子树
        a.bf = 1;  b.bf = 0
    c.bf = 0
    return c
```

AVL 树插入算法

- 失衡的情况只有上面几种，经过局部的旋转调整都可以恢复平衡：
  - 在新结点插入造成失衡后
    - 其所在的最小不平衡子树，经过调整将变成平衡因子为 **0** 的二叉排序树，而且这棵子树的高度与插入前相同
    - 调整并不改变树中数据的排序序列
  - 插入结点后保持平衡，至多只需要调整这时的最小不平衡子树

    由于插入结点并调整后的树与这个位置的原树高度相同，在这棵子树里加入新结点以及其后的可能结构调整，对该子树之外毫无影响，这之外任何结点的平衡因子都不需要修改
  - 总之，这棵子树恢复了平衡，整棵二叉排序树的平衡也就恢复了
- 算法过程中的几个阶段：检索插入位置，实际插入新结点，修改一些结点的平衡因子，失衡后的局部调整，都只是树中的一条路径上进行

  所以：**AVL** 树插入操作的代价为 **O(log n)**

AVL 树插入算法

- 在找新结点插入位置的过程中记录遇到的最小不平衡子树的根：

  用变量 **a** 记录距插入位置最近的平衡因子非 **0** 的结点。不存在这种结点时 **a** 就是树根；如果插入后失衡，**a** 就是失衡位置
- 修改从 **a** 的子结点到新结点的路径中各结点的平衡因子：

  原来对这段结点都有 **BF = 0**；插入后从 **a** 的子结点开始扫描路径上的结点 **p**，若新结点在 **p** 左子树则 **p** 平衡因子改为**1**，否则改为 **-1**
- 检查以 **a** 为根的子树是否失衡，失衡时做调整：

  若 **a.bf == 0**，插入后不会失衡，简单修改平衡因子

  若 **a.bf == 1**且新结点在其左子树则出现失衡：新结点在 **a** 左子结点的左子树时用 **LL** 调整；在 **a** 左子结点的右子树时用 **LR** 调整

  若 **a.bf == –1**且新结点在其右子树则出现失衡：新结点在 **a** 的右子结点的右子树时用 **RR** 调整；在 **a** 右子结点的左子树时用**RL**调整
- 连接好调整后的子树，可能作为根，或作为 **a** 原父结点的子结点

```
def insert(self, key, value):
    a = p = self._root
    if a == None:
        self._root = AVLNode(Assoc(key, value)); return
    pa = q = None # 维持 pa, q 为 a, p 的父结点
    while p is not None: # 确定插入位置及最小非平衡子树
        if key == p.data.key: # key存在，修改关联值
            p.data.value = value; return
        if p.bf != 0: pa, a = q, p # 已知最小非平衡子树
        q = p
        if key < p.data.key: p = p.left
        else: p = p.right
    # q 是插入点的父结点，parent，a 记录最小非平衡子树
    node = AVLNode(Assoc(key, value))
    if key < q.data.key: q.left = node  # 作为左子结点
    else: q.right = node               # 或右子结点
    # 新结点已插入，a 是最小不平衡子树
    if key < a.data.key: # 新结点在 a 的左子树
        p = b = a.left; d = 1
    else:             # 新结点在 a 的右子树
        p = b = a.right; d = -1 # d记录新结点在a哪棵子树
```

```
    # 修改 b 到新结点路上各结点的BF值，b 为 a 的子结点
    while p != node:    # node 一定存在，不用判断 p 空
        if key < p.data.key:   # p 的左子树增高
            p.bf = 1;  p = p.left
        else:              # p的右子树增高
            p.bf = -1;  p = p.right
    if a.bf == 0: a.bf = d; return # a 原 BF 为0，不会失衡
    if a.bf == -d: a.bf = 0; return # 新结点在较低子树里
    # 新结点在较高子树，失衡，必须调整
    if d == 1:      # 新结点在 a 的左子树
        if b.bf == 1: b = self.LL(a, b)  # LL 调整
        else: b = self.LR(a, b)         # LR 调整
    else:           # 新结点在 a 的右子树
        if b.bf == -1: b = self.RR(a, b) # RR 调整
        else: b = self.RL(a, b)         # RL 调整

    if pa == None: self._root = b    # 原 a 为树根
    else:
        if pa.left == a: pa.left = b
        else: pa.right = b
```

AVL 树的删除操作

- 删除算法的基本思想与插入类似，先删除而后调整。几个步骤：
  - 检索需要删除的结点
  - 把删除任意结点的问题变成删除某棵子树的最右结点
  - 实际删除结点（就是二叉排序树的删除）
  - 调整平衡
- 动态操作的时间复杂性：
  - 插入结点的最大时间耗费为树的深度 **O(log n)**，检索插入位置的同时找到了最小不平衡子树。在最小不平衡子树里修改平衡因子的时间都不超过 **O(log n)**。四种调整是 **O(1)** 操作。算法的时间复杂度是 **O(log n)**
  - 删除操作的复杂性也是**O(log n)**
  - 因此 **AVL** 树能较好支持动态字典

支持字典的二叉树结构

- 基于二叉排序树的几种结构的总结：
  - 简单二叉排序树平均检索效率高，但可能出现退化情况
  - 最佳二叉排序树的检索效率高，但如需要做数据的插入或删除操作，操作后维持其最佳性的代价太大，因此不太适用于动态字典
  - 平衡二叉排序树（**AVL** 树）的检索效率与最佳二叉排序树的效率处于同样数量级，优点是维护可以在局部完成
  - 因此，内存的动态字典通常采用平衡二叉排序树或其他与之性质类似的类似结构实现作为存储结构。这种字典的操作效率有确定性的保证（散列表是概率性保证）
  - 但要求数据（或关键码）有可用的序关系
- 人们还提出了一些与性质与 **AVL** 树类似的树型结构（例如红黑树等结构），它们也可以用于支持动态字典的实现