数据结构

第十七讲 最佳和平衡二叉排序树

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年12月7日

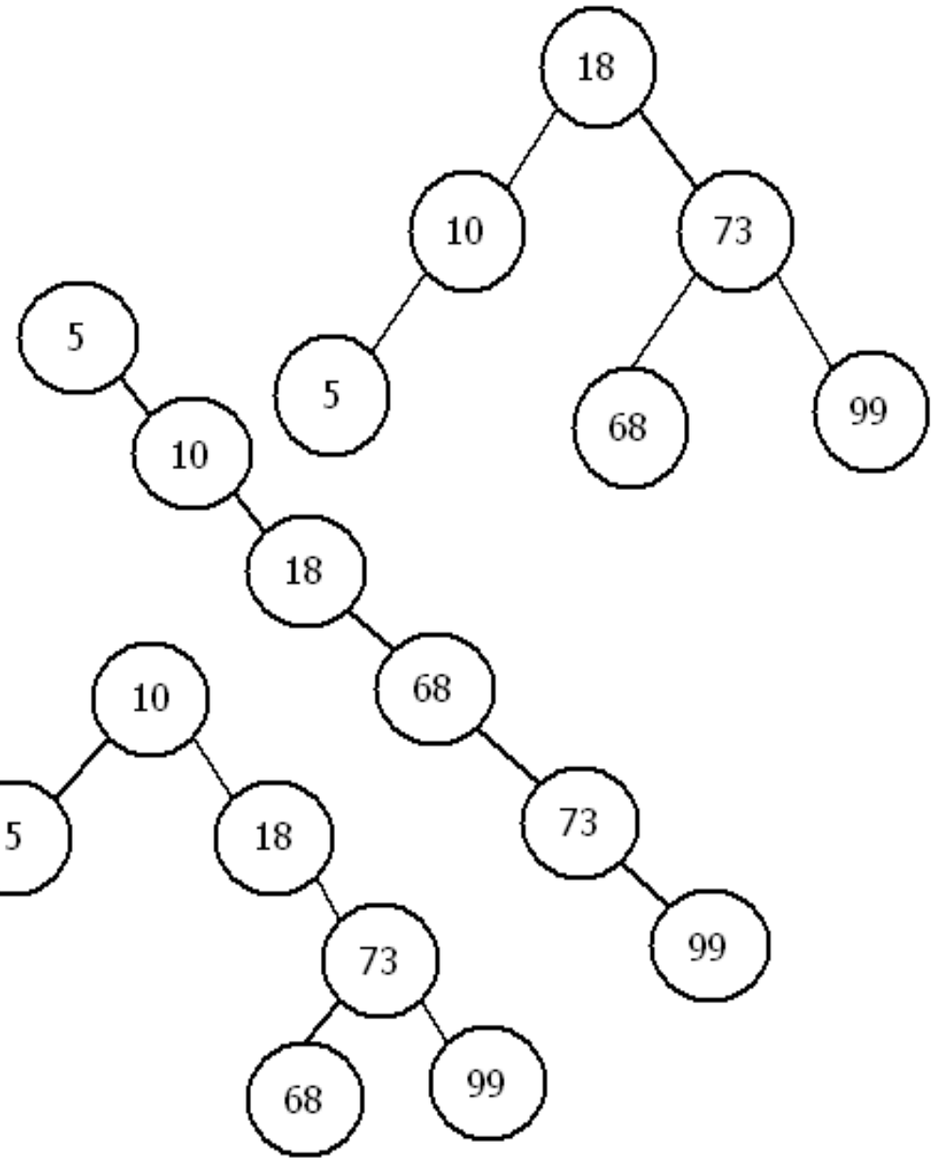
课程内容

- 最佳二叉排序树

- 平衡二叉排序树

最佳二叉排序树

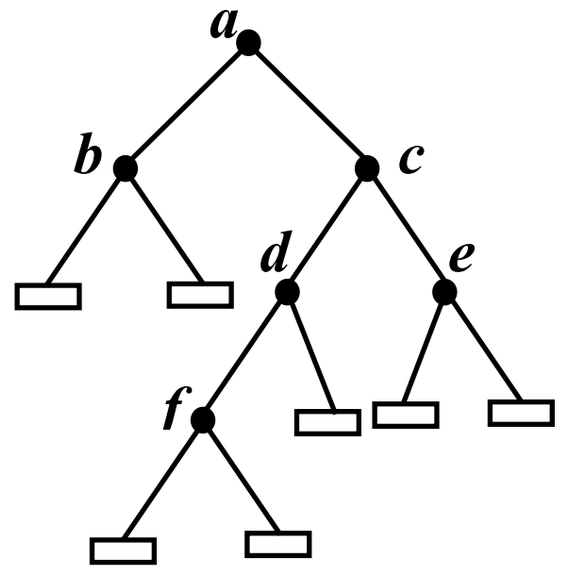
- 对同一关键码集合，不同的元素插入顺序，可能构造$n!$个不同(高度和形态)的二叉排序树!



- 哪种二叉排序树的检索效率最高?

- 具有最小平均比较次数!

扩充的二叉排序树



- 扩充的二叉排序树的对称序周游序列：
  - 从最左下(记号为0)的外部结点开始；
  - 以最右下(记号为n)的外部结点结束；
  - 所有内/外部结点都是交叉排列：
    - 第 i 个内部结点正好位于第 $i-1$ 个外部结点和第 i 个外部结点之间；
    - 外部结点代表位于其相邻的两个内部结点关键码之间的所有不属于当前字典的关键码集合。
- 对检索来说：
  - 如果被检索结点位于二叉树中第 i 层，比较次数为 $i+1$；
  - 如果该结点不存在，只有找到一个外部结点时确定检索失败，比较次数为此外部结点的层数。

平均比较次数

- 在扩充的二叉排序树里，检索一个关键码的平均比较次数为：

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

- 其中l_i是第i个内部结点的层数，l_i'是第i个外部结点的层数，p_i是检索第i个内部结点关键码的频率，q_i是被检索的关键码属于第i个外部结点关键码集合的频率，q_i和p_i也称为结点的权。

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

p_i / w是检索第i个内部结点关键码的概率，q_i / w是被检索的关键码属于第i个外部结点关键码集合的概率。

最佳二叉排序树

- 最佳二叉排序树：$E(n)$最小的二叉排序树。
  - 根结点的关键码确定，左右子树的关键码集合也惟一确定；
  - 左右子树也是最佳二叉排序树。
  - 如何构造最佳二叉排序树?
    - 等概率的检索
    - 不等概率的检索

等概率的检索

- 假设字典的关键码满足

$$key_1 < key_2 < \cdots < key_n$$

- 且检索所有结点的概率都相等，即：

$$\frac{p_1}{w} = \frac{p_2}{w} = \cdots = \frac{p_n}{w} = \frac{q_0}{w} = \frac{q_n}{w} = \cdots = \frac{q_n}{w} = \frac{1}{2n+1}$$

- 则 $E(n) = \frac{1}{2n+1}(IPL + n + EPL) = \frac{2IPL + 3n}{2n+1}$
- 其中***IPL***和***EPL***分别是内部路径长度和外部路径长度：$EPL = IPL + 2 \times n$ 。
- 若内部路径长度最小，则平均比较次数$E(n)$最小！

最小平均比较次数

- 要使得$E(n)$最小，则二叉树需满足:除最下层的叶结点度数均为0外，只有倒数第二层结点的度数可以小于2，其它结点度数必须等于2。这样的二叉排序树IPL为:

$$0+1+1+2+2+2+2+3+\ldots\ldots$$

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

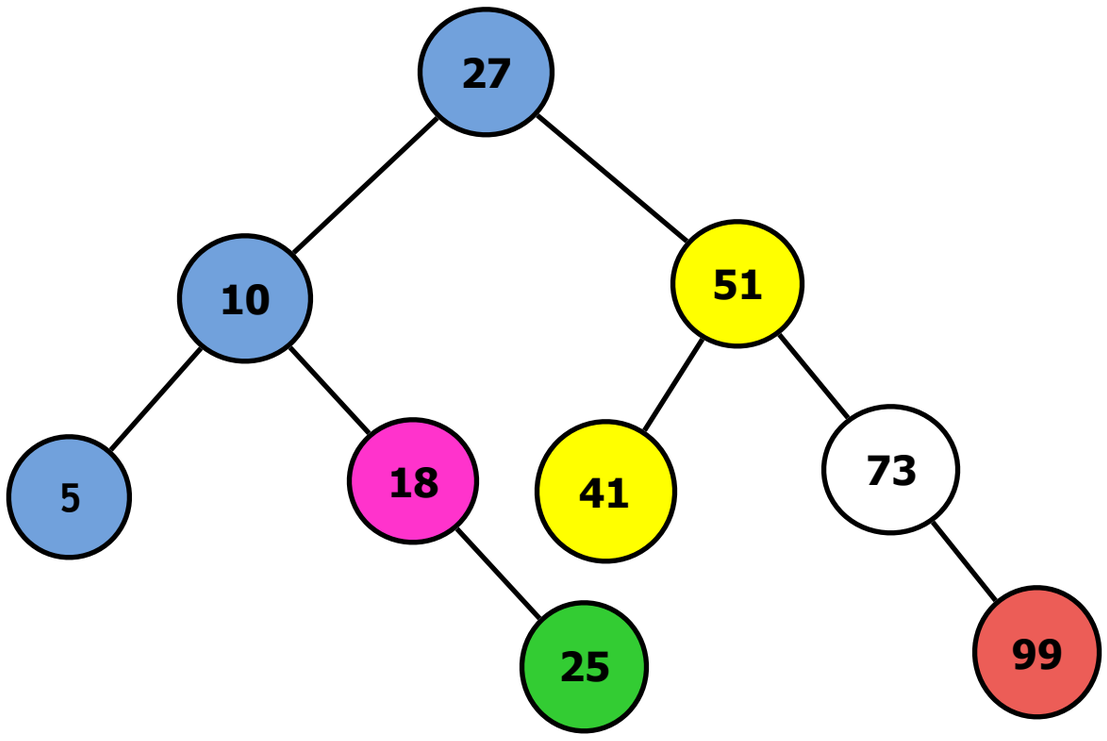
- 平均比较次数$E(n)$为$O(log_2 n)$！

等概率最佳二叉排序树的构造

1. 先将字典元素的关键码排序；
2. 对每个关键码按二分法在排序关键码序列中执行检索，将检索中遇到的还未在二叉排序树中的关键码插入二叉排序树中。

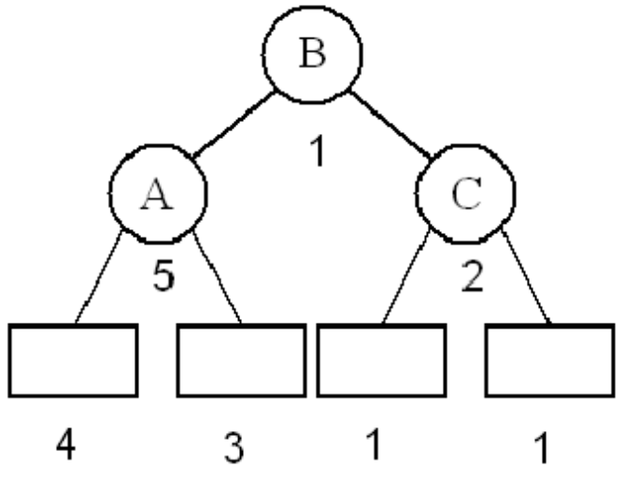
- 使用二叉排序树的插入算法。

- 考虑 K = {5, 10, 18, 25, 27, 41, 51, 73, 99}
- 构造算法(不计排序)的时间代价为$O(nlog_2n)$；
- 检索时间代价为$O(log_2n)$。

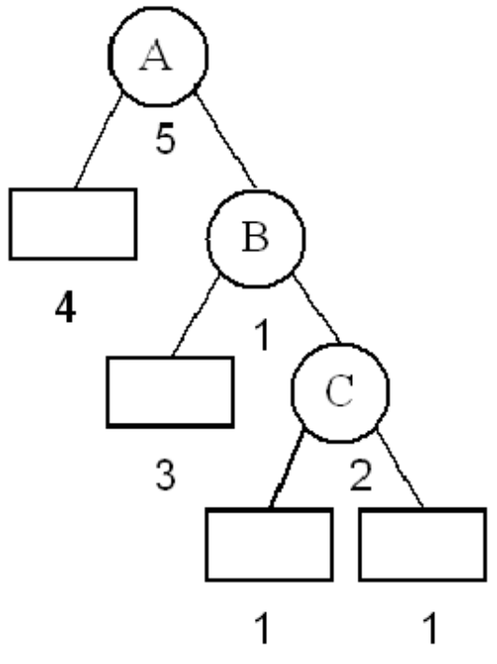


不等概率的检索

- 例子：给定关键码集合$K=\{A, B, C\}$，权集合{5, 1, 2}和{4, 3, 1，1}，构造一个具有三个内部结点，四个外部结点的最佳二叉排序树。
- 采用等概的方法构造的不一定最佳：



E(n)=33/17



E(n)=29/17

不等概率最佳二叉排序树的构造

- 将给定排序的关键码集合 $\{key_1, key_2, \ldots, key_n\}$ 和两个权集合 $\{p_1, \ldots, p_n\}$和$\{q_0, q_1, \ldots, q_n\}$, 其中 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 j 的关键码集合的概率。
- 问题：要求构造一棵最佳二叉排序树，即构造出一棵二叉排序树,使下面的函数达到最小($E(n)$一定也达到最小)：

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

上式称为包含 n 个内部结点和 n+1个外部结点的二叉排序树的花费, 记作 C(0, n)。

- 最佳二叉排序树里的任何子树都最佳。

记法说明

- 设对应 $key_1, key_2, \ldots, key_n$ 的内部结点为$v_1, \ldots, v_n$，外部结点为$e_0, e_1, \ldots, e_n$。
- 用$T(i, j)$代表包含内部结点 $v_{i+1}, \ldots, v_j$ 和外部结点e_i, $e_{i+1}, \ldots, e_j$ 的最佳二叉排序树, 权的总和记为$W(i, j)$，其根记为$R(i, j)$, 花费记为$C(i, j)$。例如：
  - $T(0, 1)$ 表示包含内部结点v_1，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_3, v_4, v_5，以及外部结点 e_2, e_3, e_4, e_5 的最佳二叉排序树。

构造最佳二叉排序树的主要思想

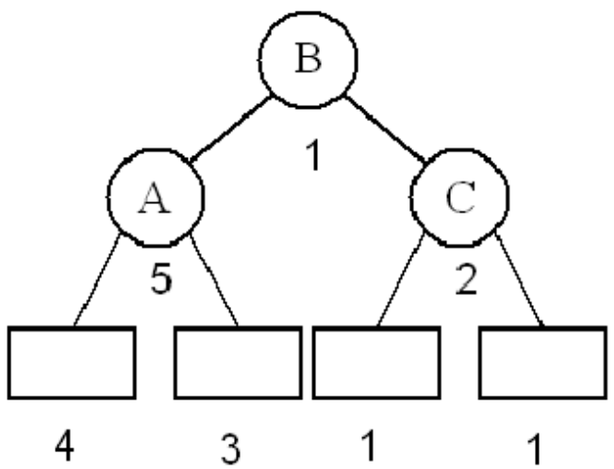
- 从小到大地构造$T(i, j)$：
  - 步骤1：构造包含一个内部结点：

    $T(0, 1), T(1, 2), \ldots, T(n-1, n)$；
  - 步骤2：构造包含二个内部结点：

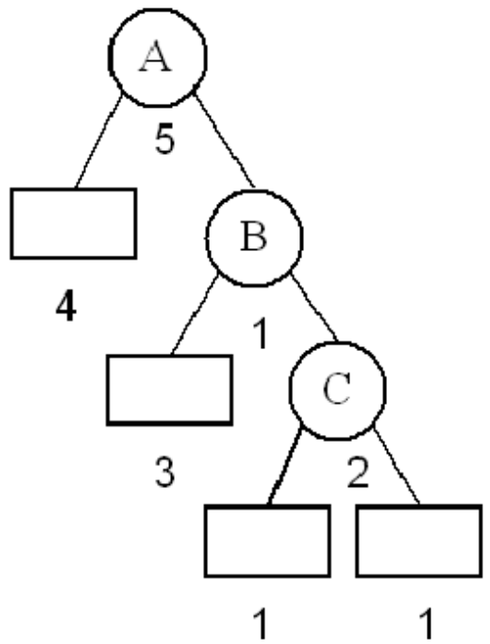
    $T(0, 2), T(1, 3), \ldots, T(n-2, n)$；
  - ……
  - 步骤n：构造$T(0, n)$。

不等概率的检索

- 例子：给定关键码集合$K=\{A, B, C\}$，权集合{5, 1, 2}和{4, 3, 1, 1}，构造一个具有三个内部结点，四个外部结点的最佳二叉排序树。
- 采用等概的方法构造的不一定最佳：



E(n)=33/17



E(n)=29/17

花费的计算

- 在构造 $T(i, j)$ 时,对所有 $i < k \leq j$, $T(i, k-1)$ 和 $T(k, j)$ 都已存在，而且已知相应花费 $C(i, k-1)$ 和 $C(k, j)$。
- 对每个 k, 以 key_k 为根，$T(i, k-1)$ 和 $T(k, j)$ 为左右子树的二叉树的花费为：

$$C_k(i, j) = W(i, j) + C(i, k-1) + C(k, j)$$

(结点增加一层)。

- 从所有可能的 k 中选择使 $C_k(i, j)$ 达到最小的那个 k，与之对应的树就是 $T(i, j)$，其根 $R(i, j) = k$。
- 新树的代价可以由构造它的已知最佳二叉排序树的代价算出：

$$C(i, j) = W(i, j) + \min_{i \leq k \leq j}(C(i, k-1) + C(k, j))。$$

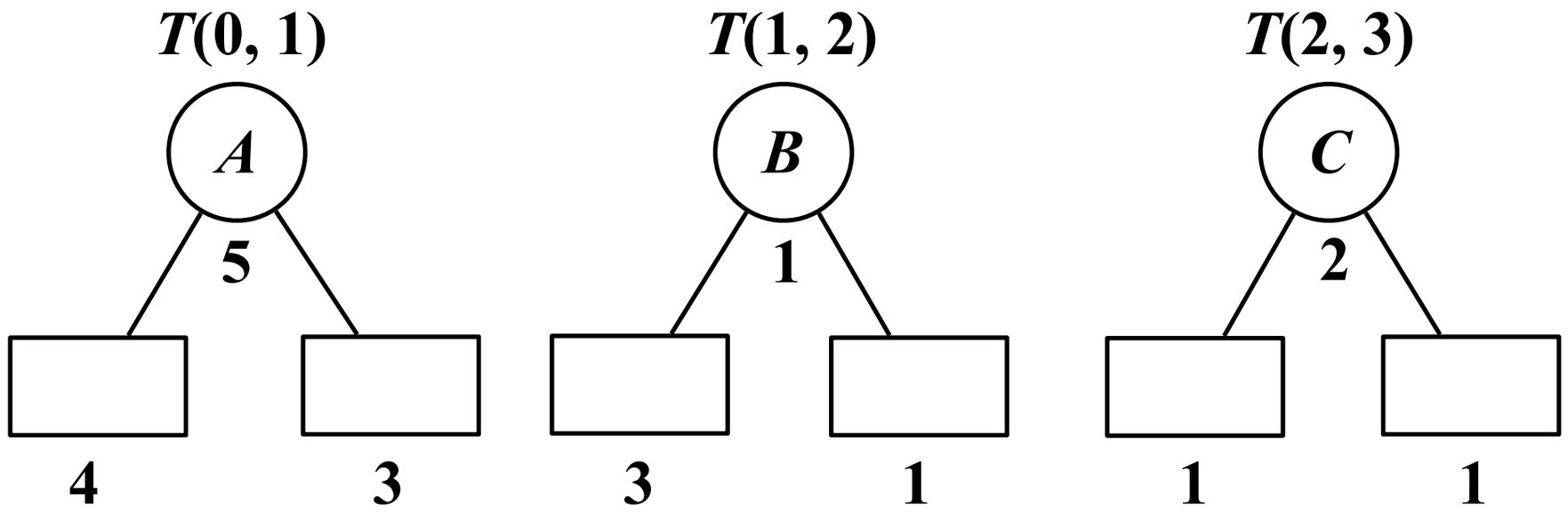
- NOTE：$T(i, i)$ 为不包含任何内部结点，仅由权值为 q_i 的外部结点构成，$C(i, i) = 0$。

- 给定关键码集合$K=\{A, B, C\}$，权集合$\{5, 1, 2\}$和$\{4, 3, 1, 1\}$，构造一个具有三个内部结点，四个外部结点的最佳二叉排序树。

1

用 $T(0, 1)$ 表示只含 e_0, v_1, e_1 的二叉排序树，它最佳(这种二叉排序树只有一棵)，代价是$q_0+p_1+q_1$；

$T(1, 2)$，…，$T(n-1, n)$ 自然也都是最佳二叉排序树(都唯一)，相应的代价 $C(i, i+1)$ 很容易计算。



$C(0, 1)=12$　　$C(1, 2)=5$　　$C(2, 3)=4$

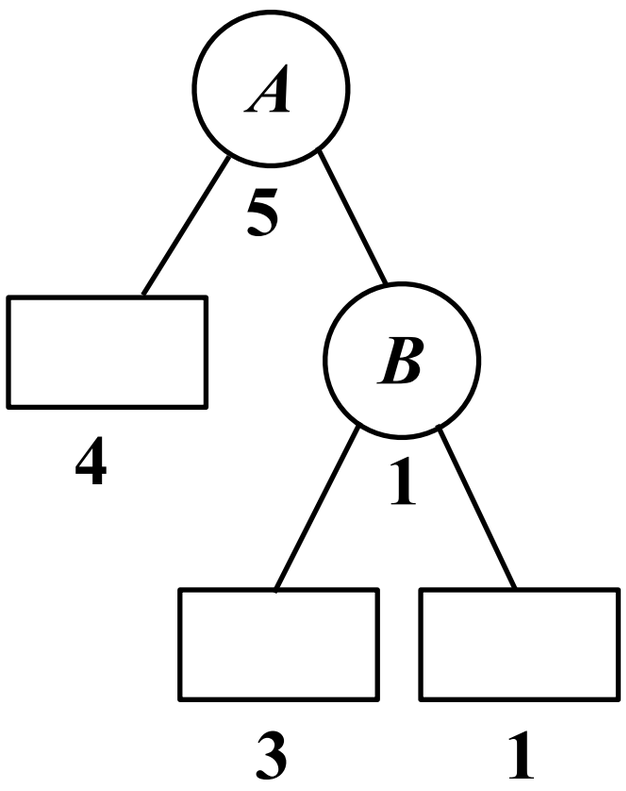
$W(0, 1)=12$　　$W(0, 1)=5$　　$W(2, 3)=4$

2

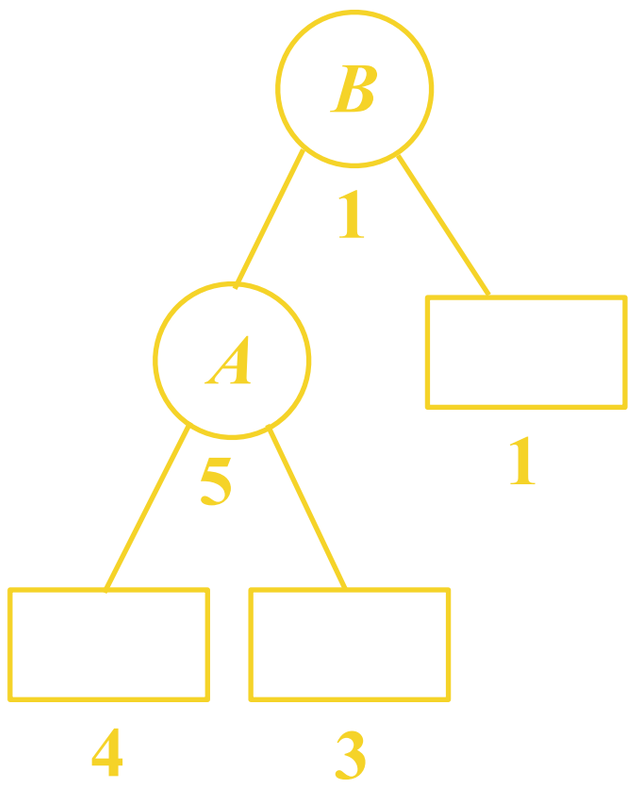
构造 $T(0, 2)$ 和 $T(1, 3)$，它们各包含两个内部结点：

- $T(0, 2)$有两种可能：
  - 以v_1为根，e_0为左子树，$T(1, 2)$ 为右子树
  - 以v_2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，选出它作为$T(0, 2)$

$T(0, 2)$

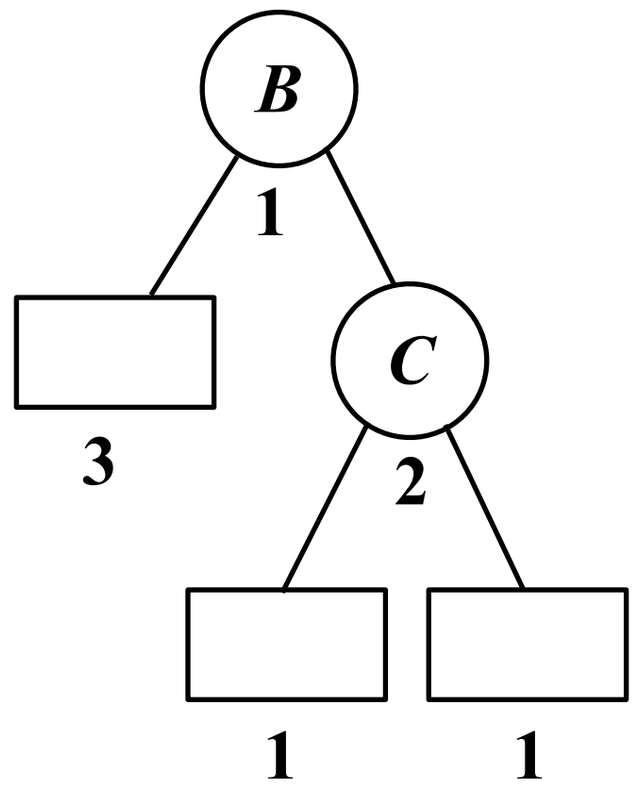


$C(0, 2)=19$



$C=26$

$T(1, 3)$

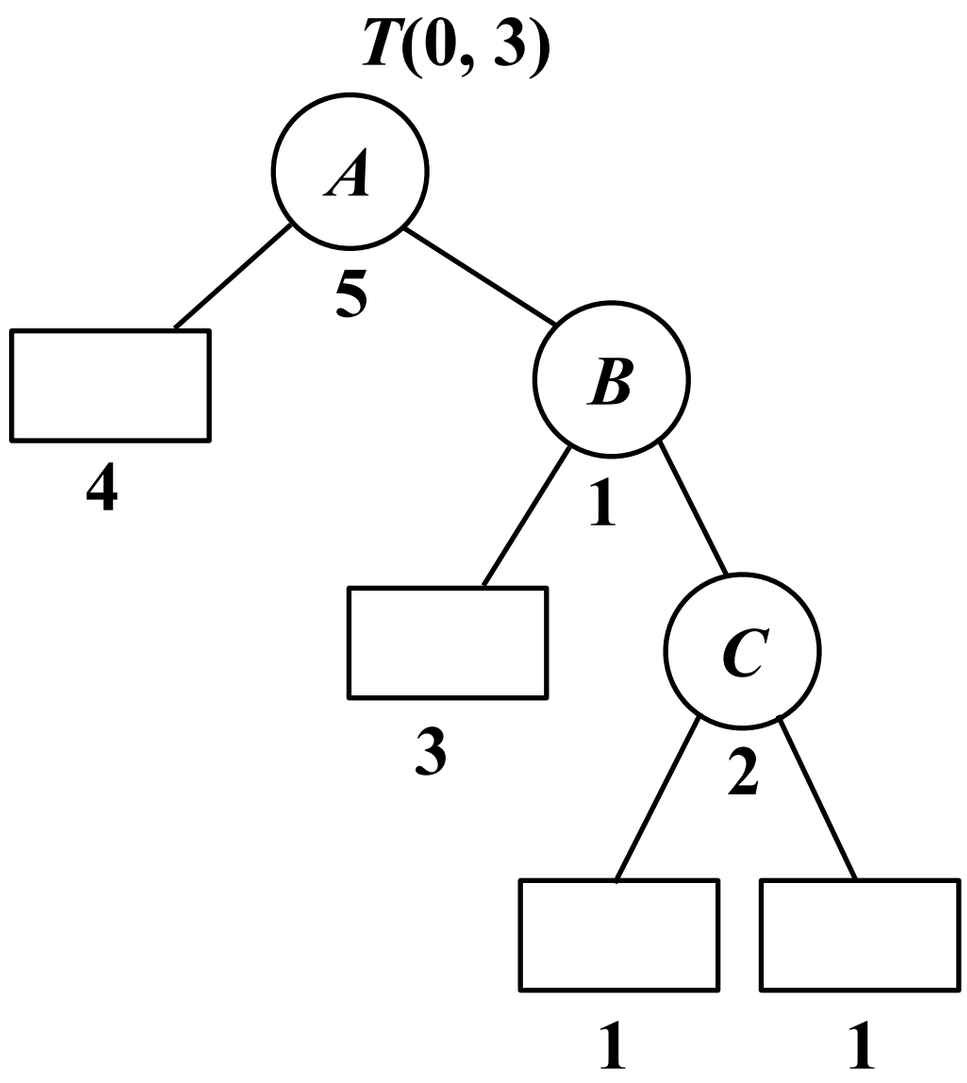


$C(1, 3)=12$

3

构造包含三个内部结点的最佳二叉排序树 $T(0, 3)$，即本问题的最终结果。候选二叉排序树：

- 以v_1为根，左子树为e_0，右子树为$T(1, 3)$
- 以v_2为根，左子树为$T(0, 1)$，右子树为$T(2, 3)$
- 以v_3为根，左子树为$T(0, 2)$，右子树为e_3



数据的存储

- 数组p存放内部结点的权(p[1],…,p[n])；
- 数组q存放外部结点的权(q[0],…,q[n])；
- c[i][j] 保存$T(i, j)$的花费；
- w[i][j] 保存$T(i, j)$的权；
- r[i][j] 保存$T(i, j)$的根。
- 构造好的最佳二叉排序树的总花费存放在 c[0][n]；
- r[0][n] 是根结点的编号。根据 r[0][n] 的值，就可以确定两棵子二叉树的根在那里：
  - 例：假定内部结点为 v_1 到 v_8，r[0][8] 的值是 5，可知
    - 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二叉树的结构。

上面例子的计算结果

P : {5,1,2}
Q: {4,3,1,1}

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	1	0	0
0	0	2	0
0	0	0	3
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	1	1	0
0	0	2	2
0	0	0	3
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	1	1	1
0	0	2	2
0	0	0	3
0	0	0	0

c :

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

$$C(i,j)=W(i,j)+\min_{i\le k\le j}(C(i,k-1)+C(k,j)).$$

算法实现

```
void opticBTree(float p[ ], float q[ ], float *c[ ], float *w[ ], int *r[ ], int n) {
  int k0, i, j, k, m;
  float min;
  for(i=0; i<=n; i++)                                   /* 准备初值 */
     for(j=0; j<=i; j++) { c[i][j]=w[i][j]=0; r[i][j]=0;  }
  for(i=0; i<=n; i++){                                  /* 计算w[i][j] */
     w[i][i]=q[i];
     for(j=i+1; j<=n; j++) w[i][j]=w[i][j-1]+p[j]+q[j];
  }
  for(j=1; j<=n; j++) { c[j-1][j]=w[j-1][j];  r[j-1][j]=j;}      /* 计算T(i,i+1) */
  for(m=2; m<=n; m++)                          /* 计算包含m=2到n个内部结点的最佳二叉排序树 */
     for(i=0; i<=n-m; i++){
        j=i+m; min=MAXVALUE k0=i+1;
        for(k=i+1; k<=j; k++)                  /* 在i<k≤j范围内找出使C(i,k-1)+C(k,j)最小的k*/
          if( (c[i][k-1]+c[k][j])<min) { min=c[i][k-1]+c[k][j]; k0=k; }
        c[i][j]=w[i][j]+min;  r[i][j]=k0;                    /* 计算C[i][j],W[i][j]和R[i][j] */
     }
}
```

算法分析

- 这个算法很典型，采用的算法模式为动态规划；
- 算法的复杂性为 $O(n^3)$，对较大的 n 非常耗时；
- 任意权值的最佳二叉排序树的构造算法更多具有理论价值，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。

平衡二叉排序树

- 最佳二叉排序树: 平均比较次数最小。
- 但是，最佳二叉排序树比较适合静态字典的表示，不适合动态字典的表示。
  - 只能根据元素全面情况静态构造；
  - 结点权值不同时构造过程很耗时；
  - 不能很好支持动态变化：
    - 通常动态变化需要导致重新构造最佳二叉排序树。
- 退而求其次: 容易调整，“较佳”检索效率？

适合动态字典表示的二叉树

- 放弃了最佳，采用某种近似最佳的性质代替最佳概念；
- 实现在动态操作中较易通过局部调整进行维护；
- 提供对各种操作的最坏性能保证。

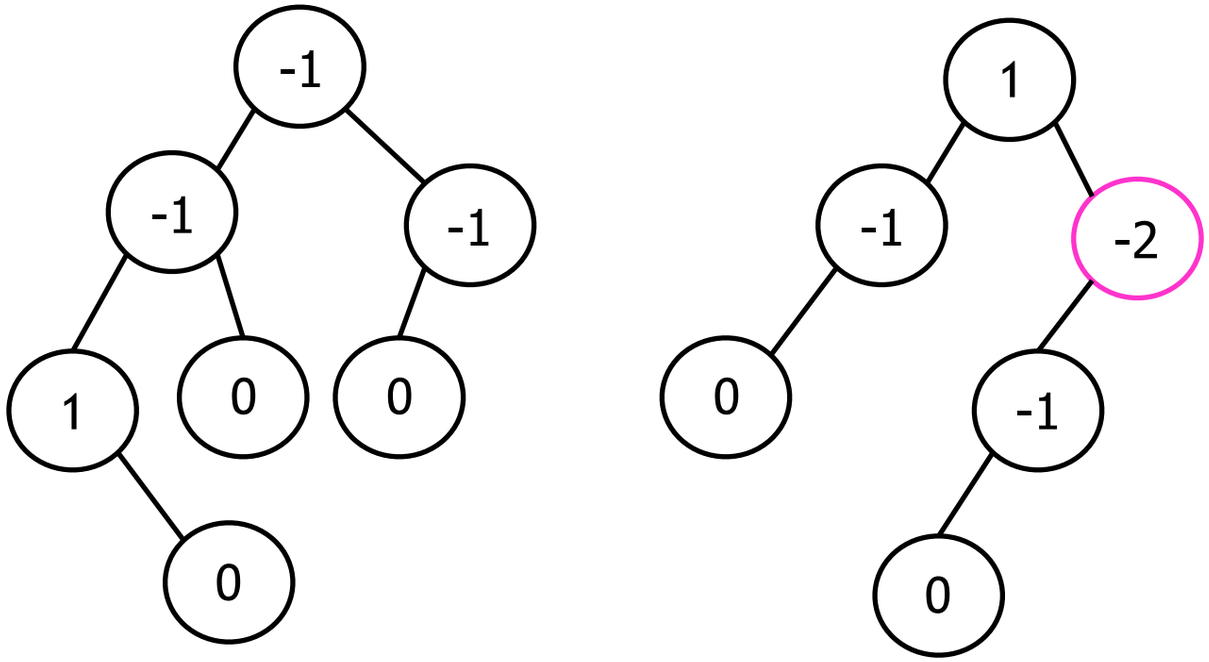
平衡二叉排序树

- 下面的讨论假定所有字典元素的检索概率相等。
- 平衡二叉排序树的直观印象：“结构形态”相对比较好的二叉树。
  - 二叉树中每个结点的左右子树高度差不多；
  - 不会出现特别长的路径。
- 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，而且左右子树的深度之差的绝对值不超过1。
- 平衡二叉排序树又称AVL树，以其提出者（前苏联的G. M. Adel'son-Vel'skii 和 E. M. Landis）的名字命名。

平衡因子

- **平衡因子BF(Balance Factor)：**
  - **结点的右子树与左子树的高度之差称为该结点的平衡因子。**
  - **可能取值只有 -1, 0, 1。**
- **平衡二叉排序树的性质可用各结点的平衡因子刻画。**
- **n个结点平衡二叉排序树的高度小于**

  $c*\log_2 n$ (c为常量)。

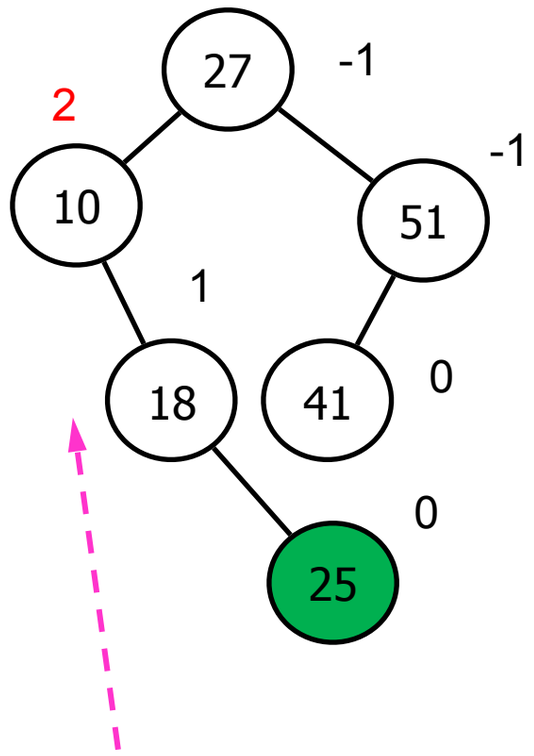


平衡和不平衡二叉排序树

- **问题: 如何调整插入、删除带来的动态变化?**

调整平衡的模式

- 若失去平衡，首先调整最小不平衡子树。
- 最小不平衡子树：离插入结点最近，且根结点的平衡因子绝对值大于1的子树。
- 若调整后最小不平衡子树恢复平衡，,且恢复到插入前的高度，那么整棵树也就恢复平衡。

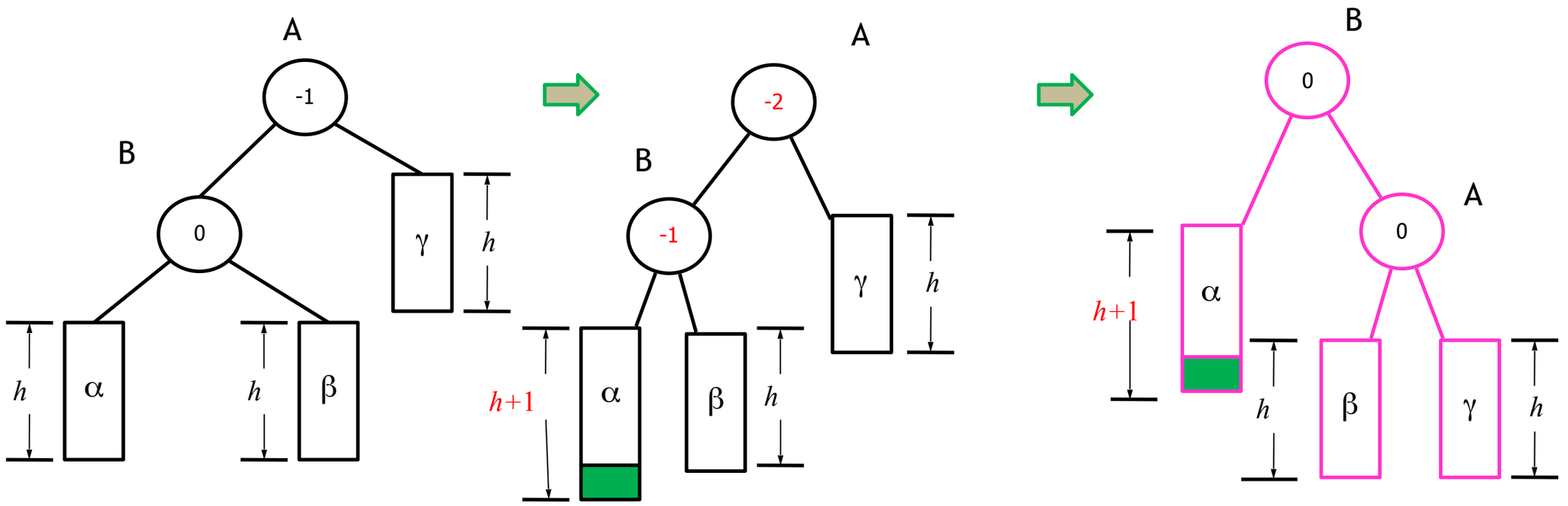


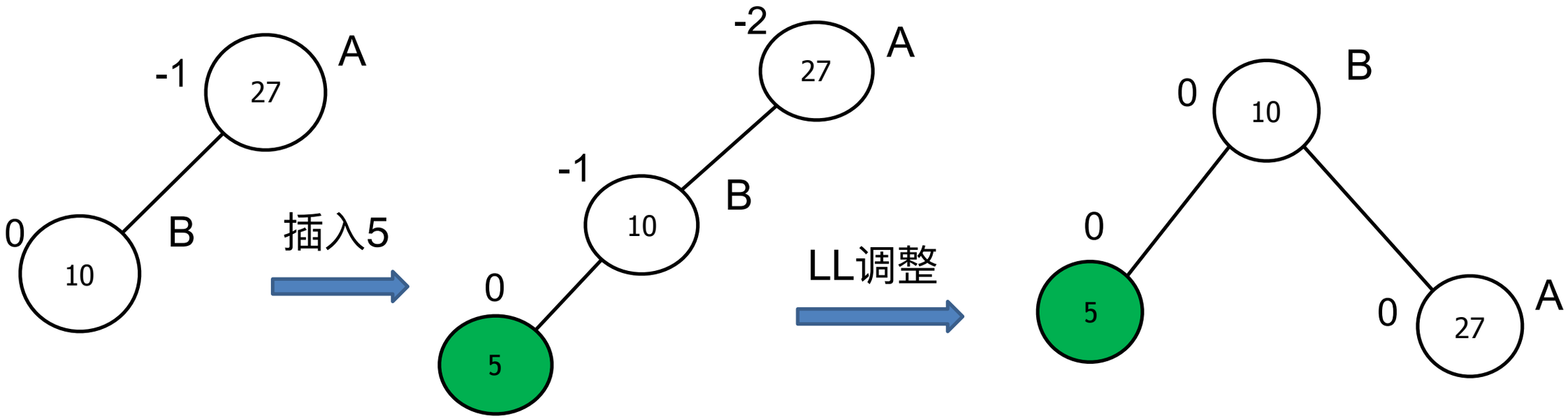
最小不平衡子树

- 设最小不平衡子树的根为A,根据当时的具体情况,调整动作分为四种：
  - 1. LL型调整；2. RR型调整；
  - 3. LR型调整；4. RL型调整.

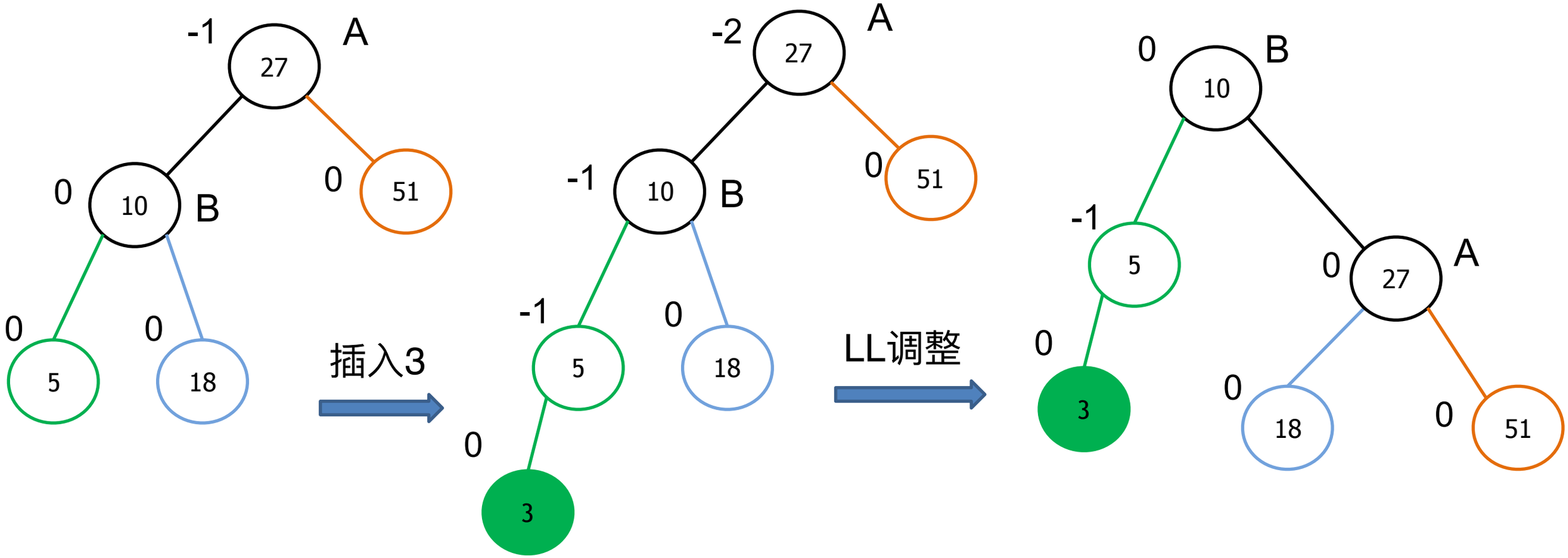
LL型调整：(αBβ)A(γ)=(α)B(βAγ)

- 在A的左子结点(L)的左子树(L)中插入新结点，使A的平衡因子由-1变成-2，打破平衡。
- 调整规则：将A的左子结点提升为新二叉树的根，原来的根A连同其右子树γ向右下旋转成B的右子树；B的原右子树β作为A的左子树。





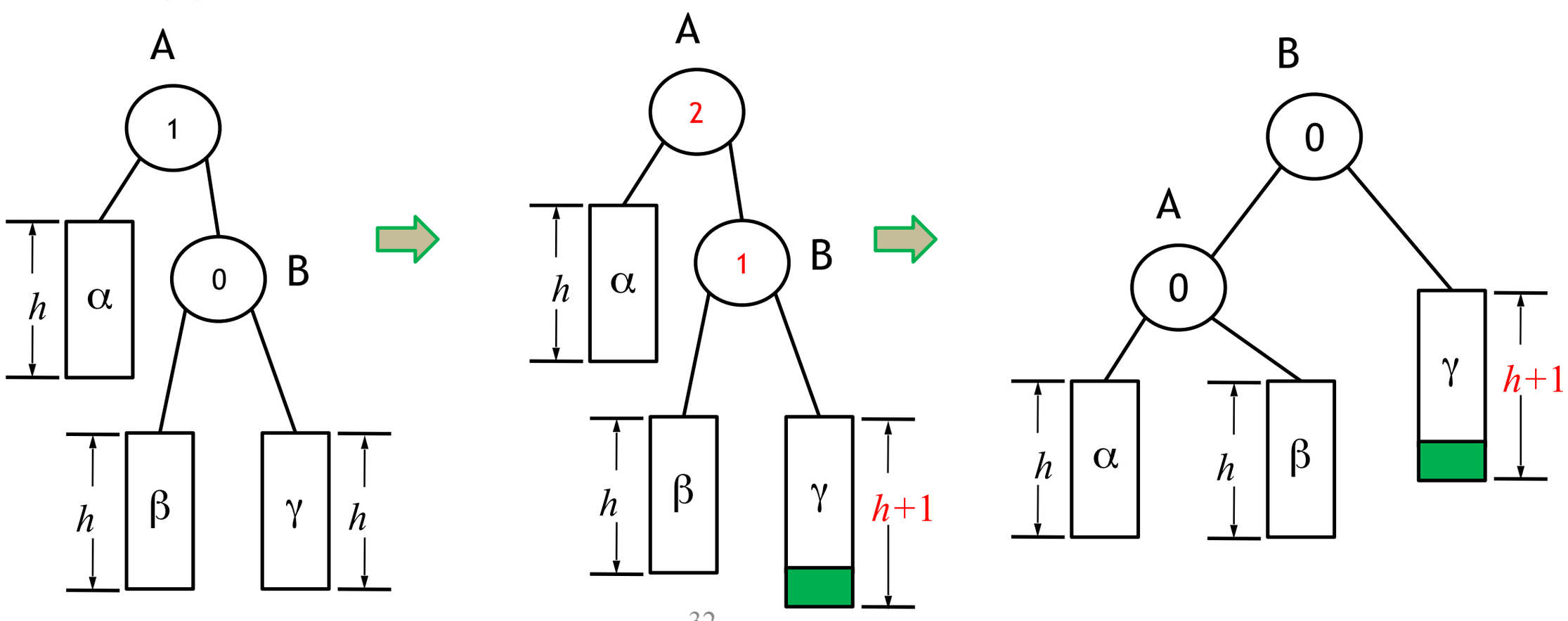
α, β和γ都是空二叉树的LL调整

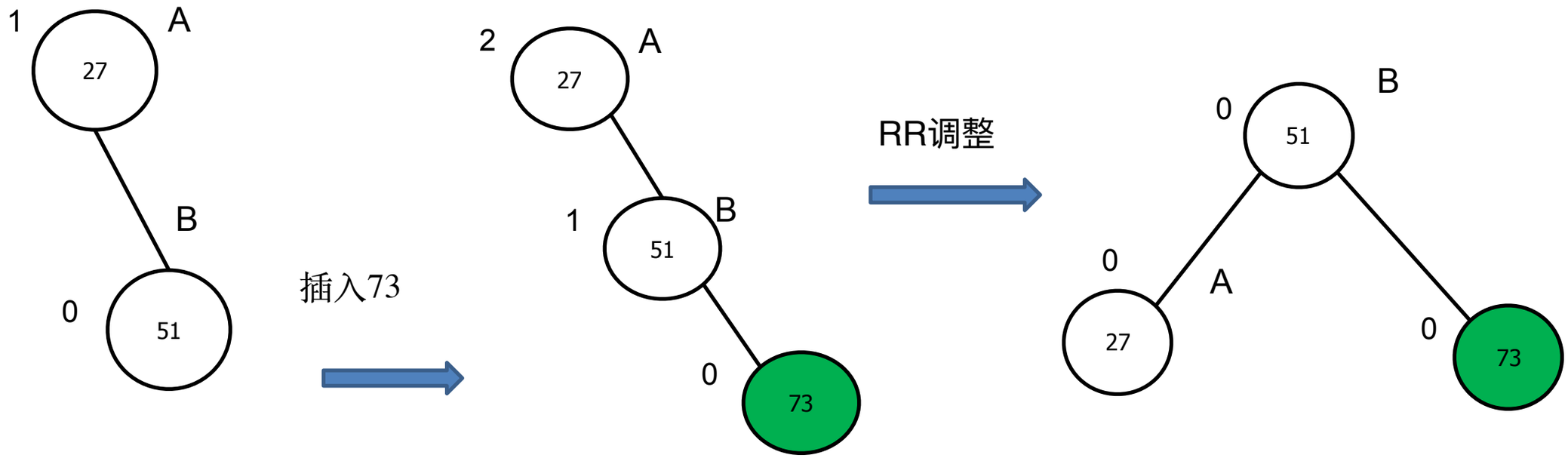


α, β和γ都非空二叉树的LL调整

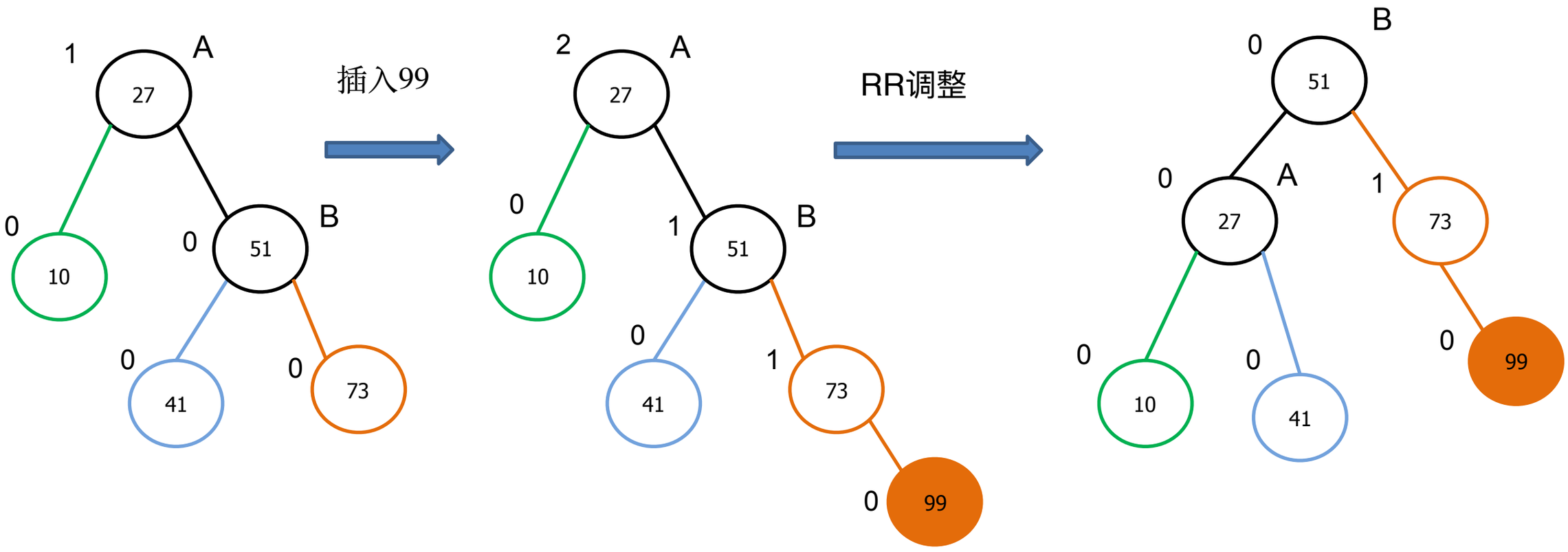
RR型调整:(α)A(βBγ)=(αAβ)B(γ)

- 在A的右子结点(R)的右子树(R)中插入新结点,使A的平衡因子由1变成2, 打破平衡.
- 调整规则: 将A的右子结点提升为新二叉树的根,原来的根A连同其左子树α向左下旋转成B的左子树; B的原左子树β作为A的右子树.





α, β和γ都是空二叉树的RR调整



α, β和γ都非空二叉树的RR调整

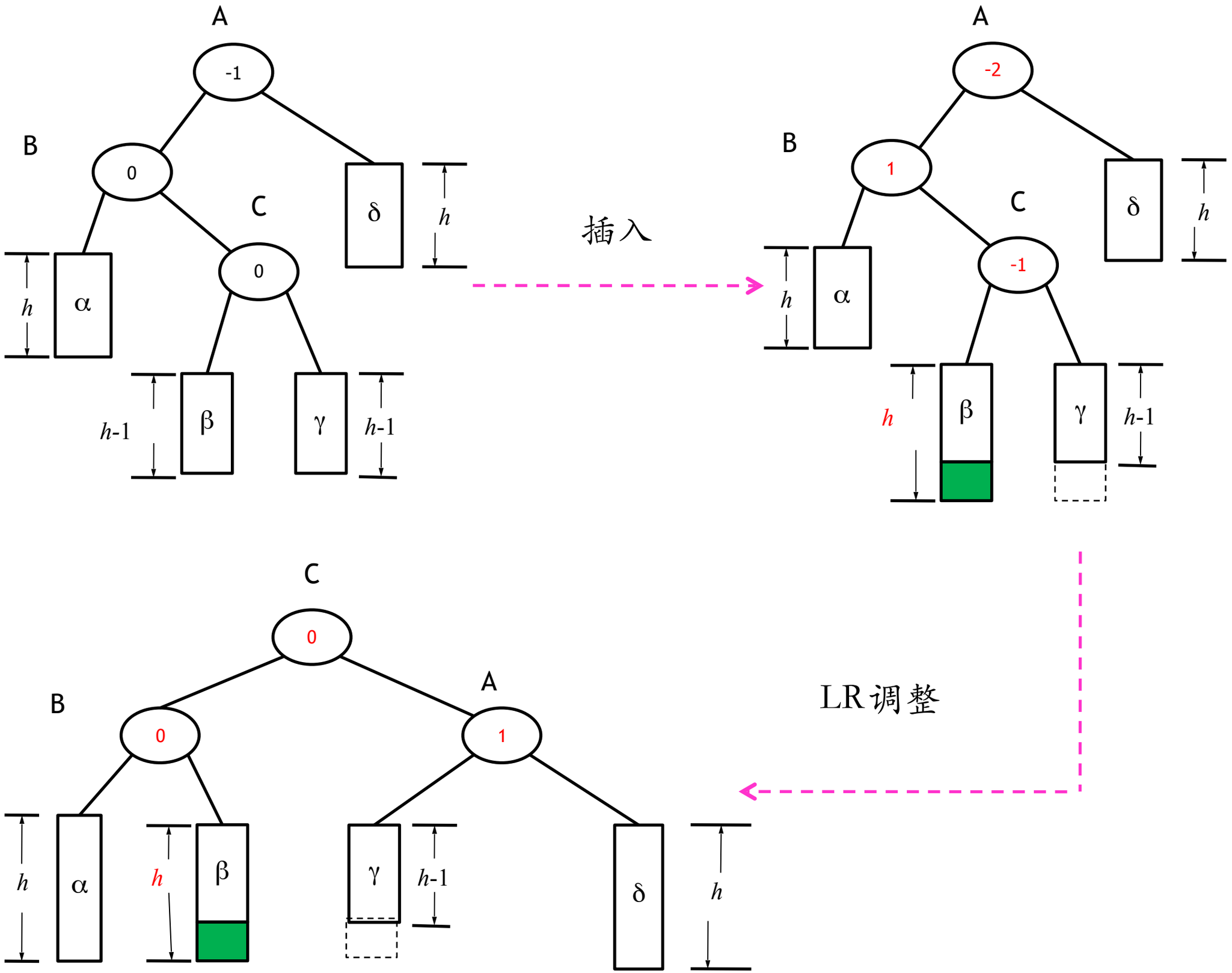
下面操作都返回调整结构后新子树的根

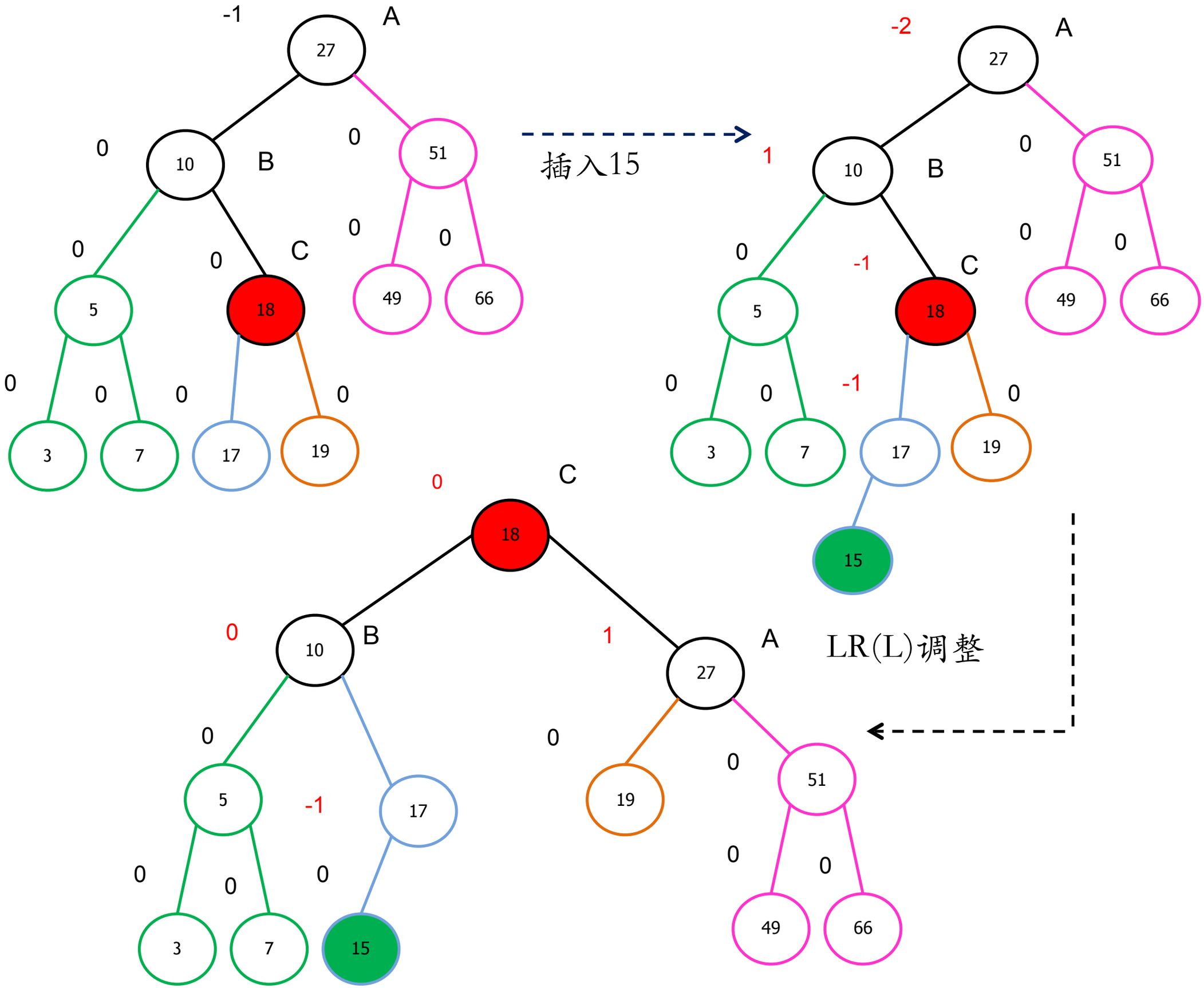
```
PAVLNode lL(PAVLNode a, PAVLNode b) {
    a->llink = b->rlink;
    b->rlink = a;
    a->bf = b->bf = 0;
    return b;                    /* b指向调整后子树的根 */
}

PAVLNode rR(PAVLNode a, PAVLNode b) {
   a->rlink = b->llink;
   b->llink = a;
   a->bf = b->bf = 0;
   return b;
}
```

LR型调整：((α)B(βCγ))A(δ)=(αBβ)C(γAδ)

- 在A的左子结点(L)的右子树(R)中插入新结点,使A的平衡因子由-1变成-2, 打破平衡。
- 调整规则: 设C是A的左子结点的右子结点,
  - 将A的孙子结点C提升为新二叉树的根;
  - 原来C的父结点B连同其左子树α向左下旋转成为新根C的左子树;
  - 原来C的右子树β成为B的右子树;
  - 原根A连同其右子树δ向右下旋转成为新根C的右子树;
  - 原来C的右子树γ成为A的左子树。
- 若α,β,γ,δ全为空树,C就是新插入的结点，记为LR(0);
- 新结点插在C的左(resp. 右)子树中,记为LR(L) (resp. LR(R))。

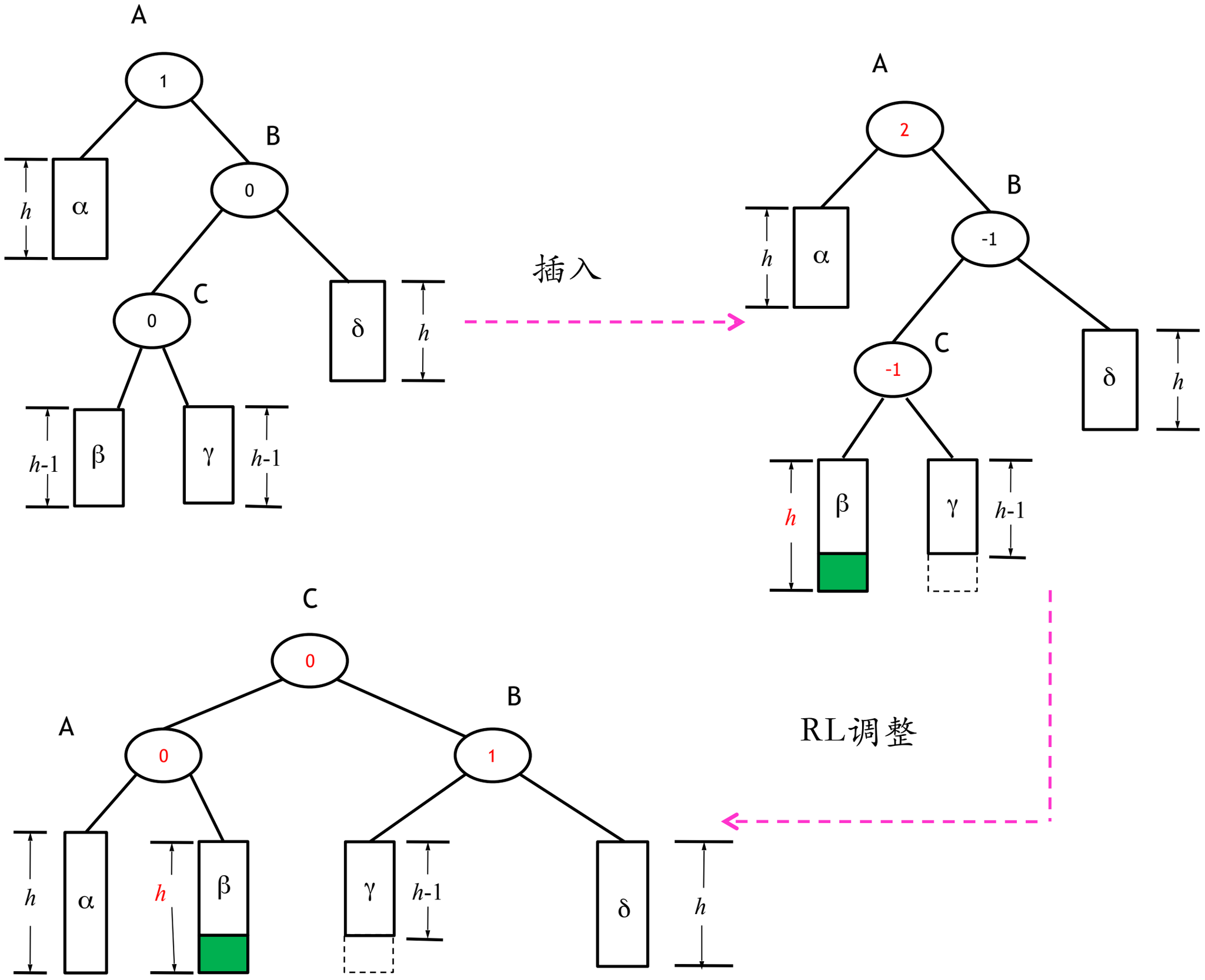

A
-1
B
0
C
0
δ
h
α
h
h-1
β
γ
h-1
插入
A
-2
B
1
C
-1
δ
h
h
α
h
β
γ
h-1
C
0
B
0
A
1
LR调整
h
α
h
β
γ
h-1
δ
h

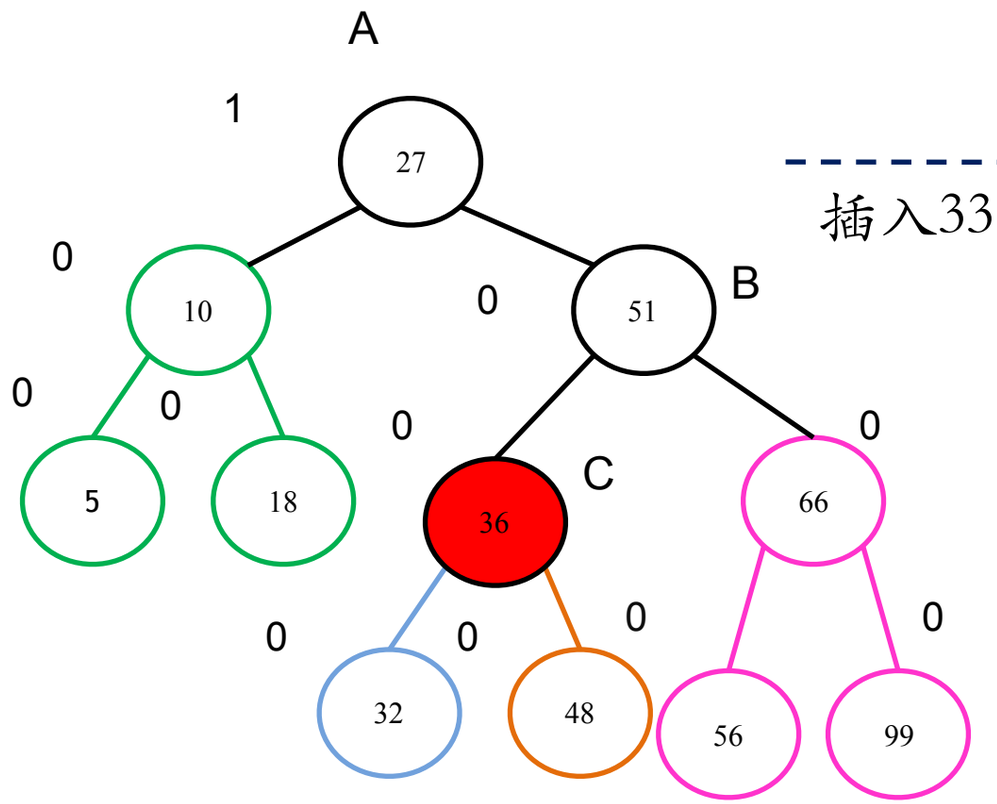

-1
A
27
0
10
B
0
51
插入15
0
0
0
C
0
0
5
18
49
66
0
0
0
0
3
7
17
19
-2
1
-1
-1
15
0
C
18
0
10
B
1
27
A
LR(L)调整
0
0
5
-1
17
19
0
51
0
0
0
3
7
15
49
66

```
PAVLNode IR(PAVLNode a, PAVLNode b) {
    PAVLNode c = b->rlink;
    a->llink = c->rlink; b->rlink = c->llink;
    c->llink = b;  c->rlink = a;
    switch (c->bf) {
    case 0:   /* LR(0)型调整 */
        a->bf = b->bf = 0;  break;
    case -1:   /* 新结点在*c左子树，LR(L)型调整 */
        a->bf = 1;  b->bf = 0;  break;
    case 1:  /* 新结点在*c右子树，LR(R)型调整 */
        a->bf = 0;  b->bf = -1; break;
    }
    c->bf = 0;
    return c;
}
```

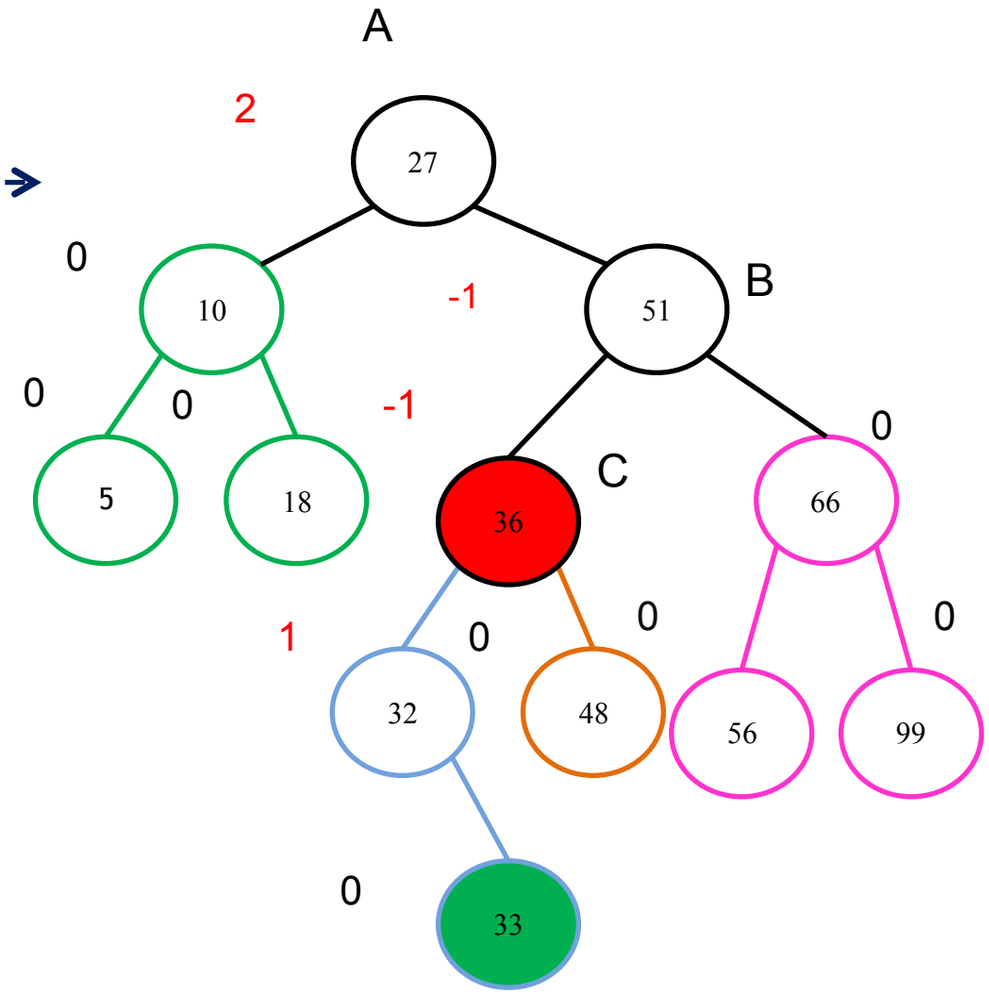
RL型调整：(α)A((βCγ)B(δ))=(αAβ)C(γBδ)

- 在A的右子女(R)的左子树(L)中插入新结点，使A的平衡因子由1变成2，打破平衡。
- 调整规则: 设C是A的右子女的左子女，
  - 将A的孙子结点C提升为新二叉树的根；
  - 原来C的父结点B连同其右子树δ向右下旋转成为新根C的右子树；
  - 原来C的右子树γ成为B的左子树；
  - 原根A连同其左子树α向左下旋转成为新根C的左子树；
  - 原来C的左子树β成为A的右子树。

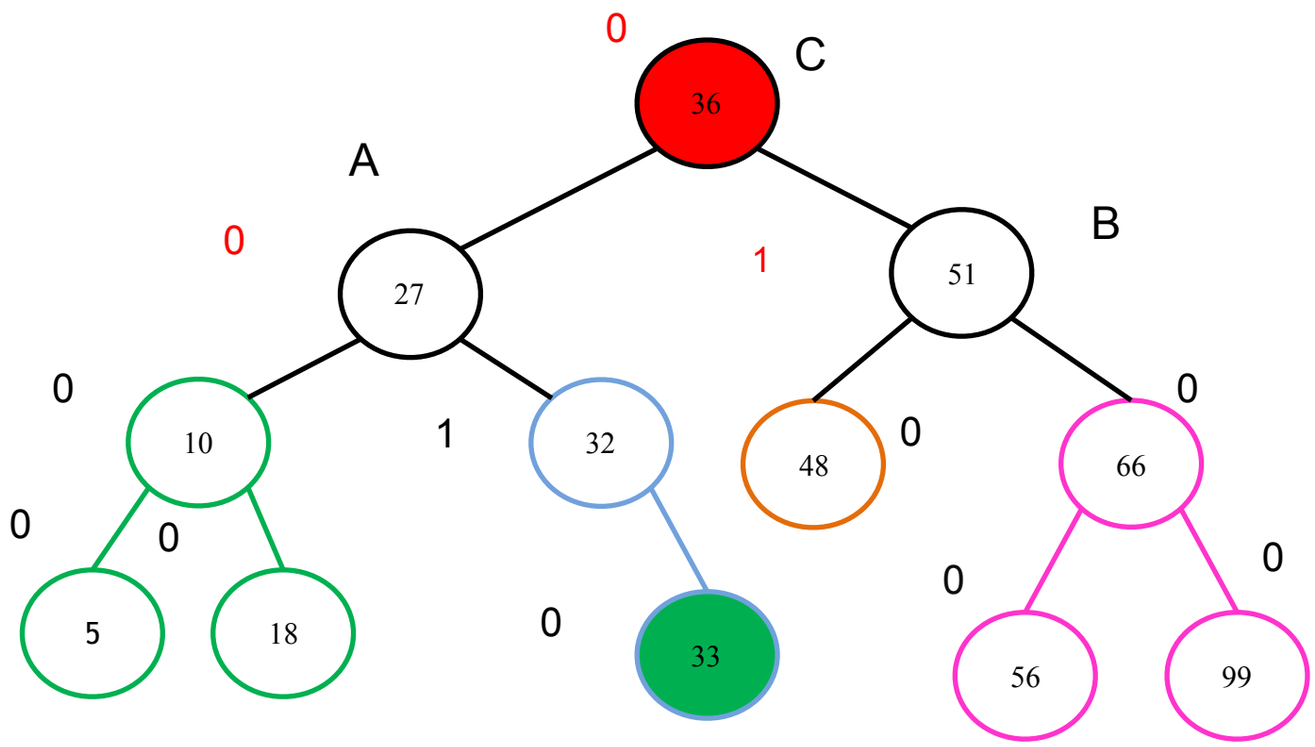

A
1
B
0
C
0
h
α
β
h-1
γ
h-1
δ
h
插入
A
2
B
-1
C
-1
h
α
β
h
γ
h-1
δ
h
RL调整
C
0
A
0
B
1
h
α
h
β
γ
h-1
δ
h


A
1
27
0
10
0
51
B
0
0
0
0
5
18
0
36
C
0
0
66
0
0
32
48
56
99

插入33


A
2
27
0
10
-1
51
B
0
0
-1
0
5
18
C
36
66
1
0
0
0
0
32
48
56
99
0
33

RL调整


0
C
36
A
0
27
1
51
B
0
0
0
0
10
1
32
48
0
66
0
0
0
0
5
18
0
33
0
56
99

```
PAVLNode  rL(PAVLNode a, PAVLNode b) {
    PAVLNode c = b->llink;
    a->rlink = c->llink;  b->llink = c->rlink;
    c->llink = a;  c->rlink = b;
    switch (c->bf) {
    case 0:                    /* *c本身就是插入结点，RL(0)型调整 */
        a->bf = 0;  b->bf = 0;  break;
    case -1:                   /* 插在*c的左子树中，RL(L)型调整 */
        a->bf = 0;  b->bf = 1;  break;
    case 1:                    /* 插在*c的右子树中，RL(R)型调整 */
        a->bf =- 1;  b->bf = 0;  break;
    }
    c->bf = 0;
    return c;
}
```

调整平衡的模式——小结

- 上面的几种情况在经过平衡旋转处理后：
  - 原来的最小不平衡子树变成了平衡二叉排序树，
  - 其高度与插入之前这棵子树的高度相同。
- 当平衡二叉排序树因插入结点而失衡时，仅需对最小不平衡子树进行旋转处理，就可以恢复整棵树的平衡。
- 调整完全是局部的，可以高效实现。

AVL树的实现

- 采用llink-rlink法表示，在结点中增加一个字段存储结点的平衡因子。

```
struct  AVLNode;
typedef  struct  AVLNode  * PAVLNode;
struct AVLNode {
    KeyType   key;                  /* 结点的关键码*/
    int   bf;                       /* 结点的平衡因子*/
    PAVLNode  llink,  rlink;        /* 左、右指针 */
};
typedef  struct   AVLNode  *AVLTree;
typedef   AVLTree  *PAVLTree;
```

- 在AVL树中的检索算法与一般二叉排序树中的检索算法一样。

AVL树的插入运算

- 在AVL树中插入结点的算法框架与一般的二叉排序树的插入算法类似, 但要考虑插入后调整平衡:

1. 新结点插入后， 若二叉树失去了平衡,则要找到最小不平衡子树:

   · 令结点a为离插入位置最近的平衡因子不为0的结点，不存在这种结点时a为树根；

   · 若插入后AVL树失衡，则a就是最小不平衡子树的根；

   · parent_a为a的父结点。

AVL树的插入运算

2. 新结点插入后,要修改从a的子结点到新结点的路径中各结点的平衡因子：

- 插入前这段路径里的结点的平衡因子均为0；
- 插入后从a的子结点开始扫描路径上的结点p，若新结点插入p左子树，则p平衡因子变为-1；否则变为1。

AVL树的插入运算

3. 检查以a为根的子树是否失衡：

- **若a平衡因子为0，则插入后不会失衡,修改平衡因子；**
- **若a平衡因子为-1且新结点插入a的左子树则出现失衡：**
  - **若新结点插入a的左子结点的左子树则采用LL型调整；**
  - **若插入到a的左子结点的右子树则采用LR型调整；**
- **若a平衡因子为1且新结点插入a的右子树则出现失衡：**
  - **若新结点插入a的右子结点的右子树则采用RR型调整；**
  - **若插入到a的右子结点的左子树则采用RL型调整。**

创建以Key为关键码的AVL树结点算法

```
PAVLNode createNode(KeyType key) {
    PAVLNode node  = (PAVLNode)malloc(sizeof(AVLNode));
    if (node != NULL) {
        node->key = key;
        node->bf = 0;
        node->llink = node->rlink = NULL;
    }
    return node;
}
```

AVL树的插入算法

```
int  avlInsert(PAVLTree ptree, KeyType key) {
    PAVLNode  node_a,  node_b, parent_a, p, q,  node;
    int d;
    if(*ptree == NULL) { /* 原树为空 */
        *ptree = createNode(key); return  1;
    }
node_a = p = *ptree;  parent_a = q = NULL;
while ( p != NULL ) {                          /* 确定插入位置及最小不平衡子树 */
     if (key == p->key) return 1;              /* 关键码key存在 */
     if(p->bf != 0) {                          /* 记录最小不平衡子树 *node_a */
          parent_a = q;  node_a = p;
    }
    q = p;
    if(key < p->key) p = p->llink;
    else p = p->rlink;
}
/* *q是插入点的父结点，parent_a 和node_a确定最小不平衡子树 */
```

AVL树的插入算法

```
node = createNode(key);
   if (key < q->key) q->llink = node;   /* 作为*q的左子树 */
   else q->rlink = node;                /* 作为*q的右子树 */
                                        /* 新结点已插入，node_a 指向最小不平衡子树 */
   if (key <node_ a->key) {             /* 新结点在 *node_a 的左子树 */
      p =node_b = node_a->llink; d = -1;
   }
   else {                               /* 新结点在*node_a 的右子树 */
      p =node_b = node_a->rlink; d = 1;
   }                                    /* d 记录了新结点在 *node_a 的哪棵子树 */
                                        /* 修改 *node_b 到新结点路上各结点的BF值*/
   while ( p != node ) {                /* node 一定存在，不用判断 p 空 */
      if (key < p->key) {               /* *p的左子树增高 */
         p->bf = -1;  p = p->llink;
      }
      else {                            /* *p的右子树增高 */
         p->bf = 1;  p = p->rlink;
      }
   }
```

AVL树的插入算法

```
if (node_a->bf == 0) {                  /* *node_a原BF为0, 插入后不会失衡 */
    node_a->bf = d;  return 1;  }
 if (node_a->bf == -d) {                /* 新结点插在较低的子树中 */
    node_a->bf = 0;  return 1; }
                                        /* 新结点插在较高子树中，失衡，需对子树调整 */
 if (d == -1)                           /* 新结点插在*node_a的左子树*/
     if(node_b->bf == -1) node_b = lL(node_a, node_b);     /* LL型调整*/
     else node_b = lR(node_a, node_b);                     /* LR型调整*/
   else                                 /* 新结点插在*node_a的右子树*/
     if(node_b->bf == 1) node_b = rR(node_a, node_b);      /* RR型调整 */
     else node_b = rL(node_a, node_b);                     /* RL 型调整 */
   if (parent_a == NULL) *ptree = node_b;                  /* 原来的 *node_a 为树根 */
   else {
     if (parent_a->llink ==node_a) parent_a->llink =node_b;
     else parent_a->rlink = node_b;
   }
}
```

插入算法分析

- 整个算法的时间复杂度为 $O(log_2 n)$:
  - 插入关键码为key的结点的最大时间耗费为树的深度 $O(log_2 n)$;
  - 算法在检索插入结点的同时找到最小不平衡子树;
  - 对最小不平衡子树中平衡因子的最大调整时间与深度成正比;
  - 四种子树调整时间为常数$O(1)$。

AVL树的删除算法

- 算法基本思想与插入一样，先删除而后调整。其中：
  - 把删除任意结点的问题变成删除某各子树的最右结点
  - 进行删除(就是二叉排序树的删除)
  - 调整平衡。
- AVL树的结点删除操作的复杂性也是$O(log_2 n)$；因此 AVL 树能较好地支持动态字典。

本讲重点

- 最佳二叉排序树适合表示静态字典。
- 构造不等概情况的最佳二叉排序树采用动态规划方法构造。
- 平衡二叉排序树的检索效率与最佳二叉排序树在同一数量级（“次佳”可以接受）。
- 平衡二叉排序树的优点是维护可以在局部完成（$O(log_2 n)$）
  - 优于最佳二叉排序树的$O(n^3)$
- 内存的动态字典通常采用平衡二叉排序树或其他类似结构存储。