上节复习

- 二叉排序树，又称二叉查找树
  - 左孩子小于父亲，右孩子大于父亲
  - 中序遍历可以输出有序序列
- 二叉排序树的查找
  - 通过比较当前结点的数值大小，沿二叉树的左右孩子往下搜索
  - 掌握查找次数的计算
- 二叉排序树的插入：
  - 创建新结点，定位到查找不成功时路径上访问的最后一个结点
  - 修改最后结点的左孩子或右孩子指针，指向新结点
- 二叉排序树的删除：无孩子，一个孩子，两个孩子
  - 无孩子，直接删除，父亲结点的孩子指针为空
  - 有一个孩子，父亲结点的孩子指针指向删除结点的孩子
  - 有两个孩子，找出左子树的最大右孩子结点，用右孩子替换，然后删除原来的右孩子

第九章 查找

9.1 静态查找表

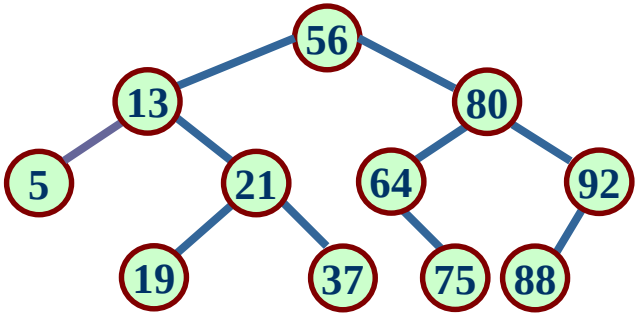
9.2 动态查找表

9.3 哈希表

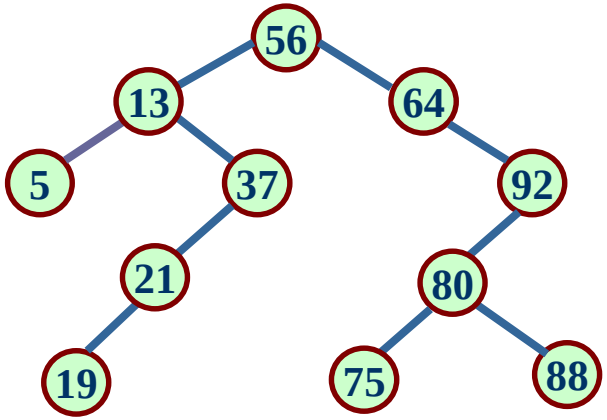
9.2 动态查找表

二. 平衡二叉树

- 平衡二叉树（Balanced Binary Search Tree或Height-Balanced Binary Search Tree）是二叉排序(查找)树的另一种形式，平衡二叉树又称AVL树(Adelsen-Velskii and Landis)。
  - 特点：树中每个结点的左、右子树的深度之差的绝对值最多为1，即$|h_L-h_R|\leqslant 1$



AVL树

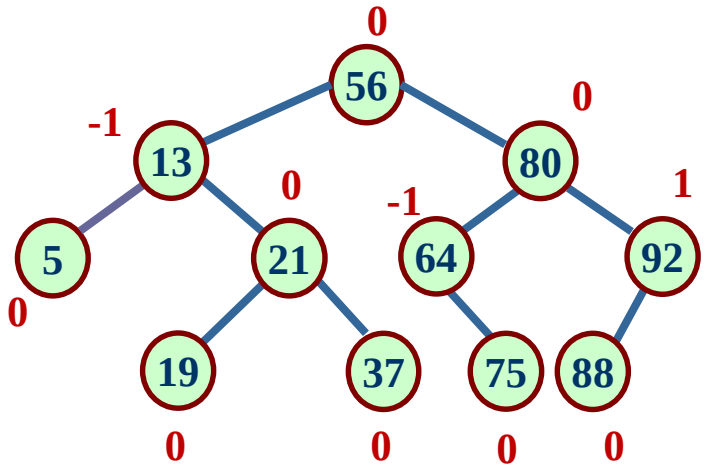


非AVL树

9.2 动态查找表

二. 平衡二叉树

- 平衡因子
  - 每个结点附加一个数字，给出该结点左子树的高度减去右子树的高度所得的高度差，这个数字即为结点的平衡因子*BF*(Balance Factor)
  - AVL树任一结点平衡因子BF只能取值 -1，0和1



9.2 动态查找表

二. 平衡二叉树

- 平衡化旋转
  - 如果在一棵平衡的二叉查找树中插入一个新结点，造成了不平衡，此时必须调整树的结构，使之平衡化。
- 平衡化处理(旋转)有两类：
  - 单向旋转（单向右旋 和 单向左旋）
  - 双向旋转（先左后右旋转 和 先右后左旋转）

9.2 动态查找表

二. 平衡二叉树

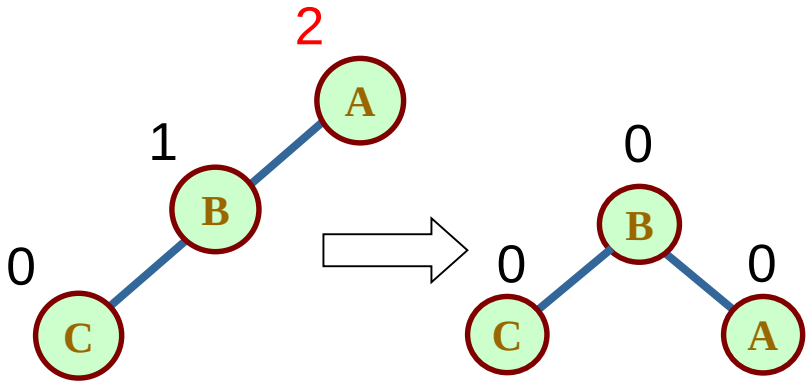
平衡化旋转

- 每插入一个新结点时，AVL树中相关结点的平衡状态会发生改变
- 在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子
- 如果在某一结点发现不平衡，停止回溯，从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。对这三个结点进行平衡化处理
- 平衡化处理包括：
  - 单向右旋
  - 单向左旋
  - 先左后右旋转
  - 先右后左旋转

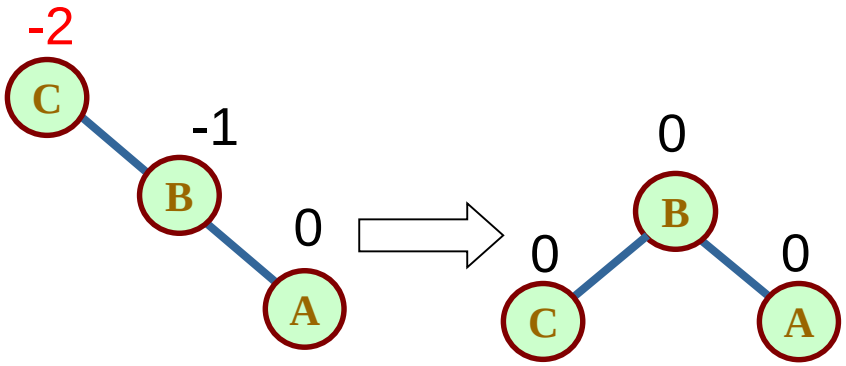
9.2 动态查找表

二. 平衡二叉树

- 平衡化单向旋转
  - 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。对这三个结点进行平衡化处理
  - 如果这三个结点处于一条直线上（“/”LL型或“\”RR型），则采用单向旋转进行平衡化单向旋转，分为：单向右旋（“/” LL型）和单向左旋（“\” RR型）



单向右旋

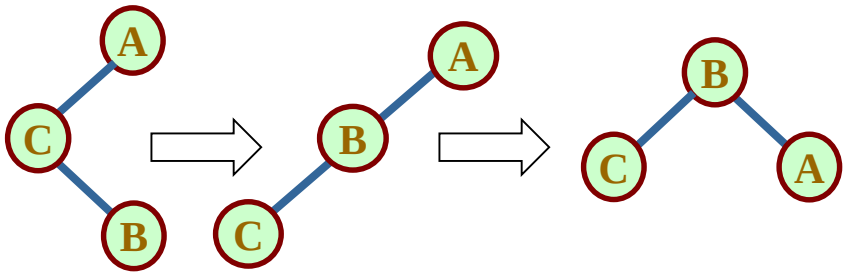


单向左旋

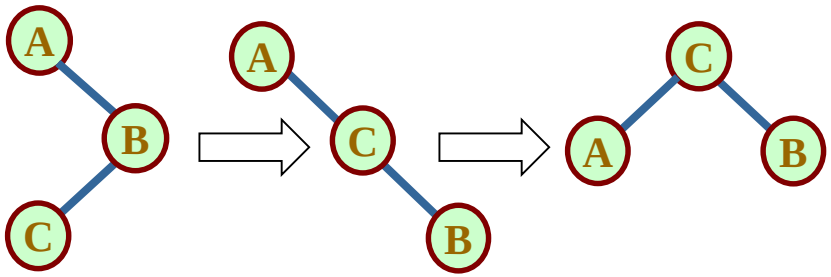
9.2 动态查找表

二. 平衡二叉树

- 平衡化双向旋转
  - 如果这三个结点处于一条折线上（“<”LR型或“>”RL型），则采用双向旋转进行平衡化，分为：先左后右（“<” LR型）和先右后左（“>” RL型）



先左后右旋转

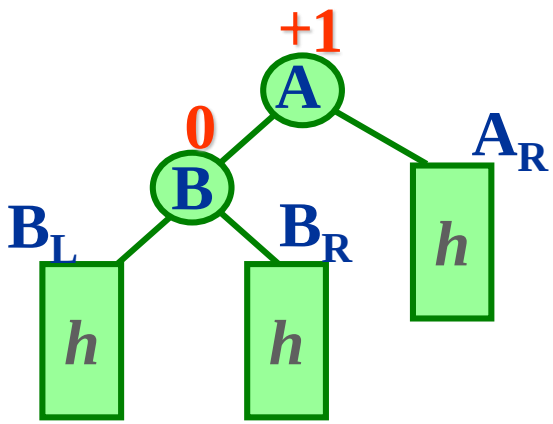


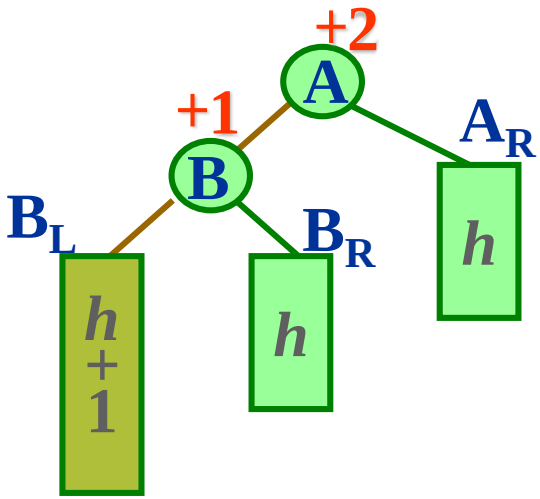
先右后左旋转

9.2 动态查找表

二. 平衡二叉树

- 单向右旋(LL型)平衡处理
  - 在结点A的左孩子结点B的左子树B_L上插入新结点，该子树高度增1，导致结点A的平衡因子变成+2，造成不平衡，则需进行一次右旋转操作，进行平衡化。（A为失去平衡的最小子树根结点）

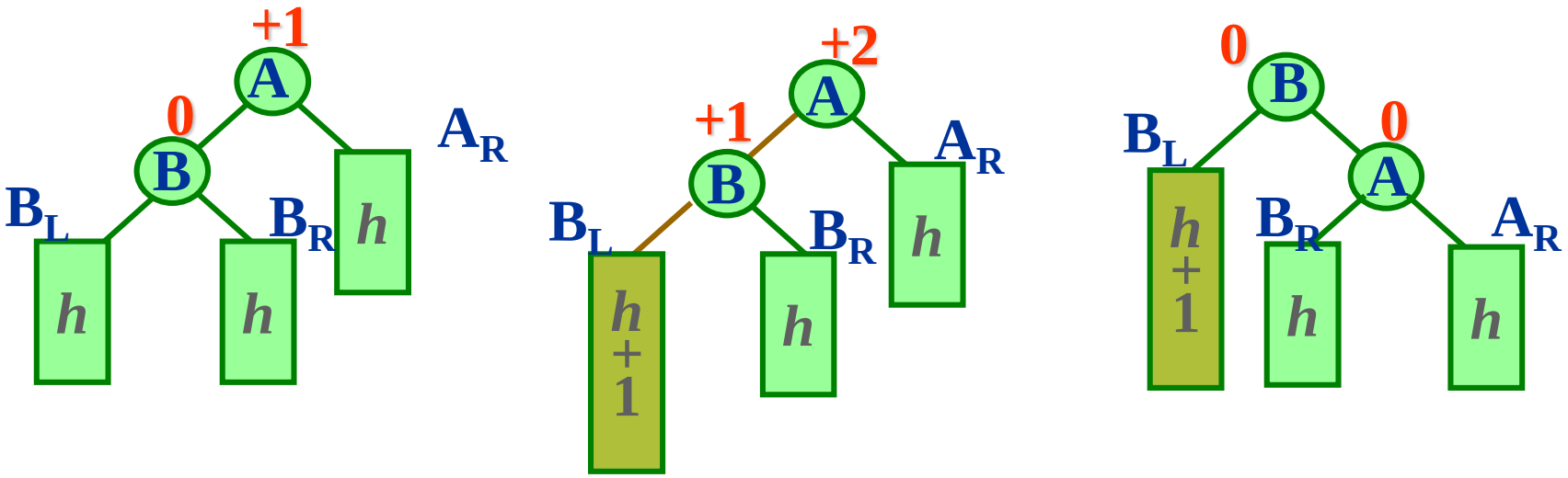




9.2 动态查找表

二. 平衡二叉树

- 单向右旋的具体操作
  - 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和B_L（“/”LL型）
  - 以结点B为旋转轴，将结点A顺时针(右)旋转。

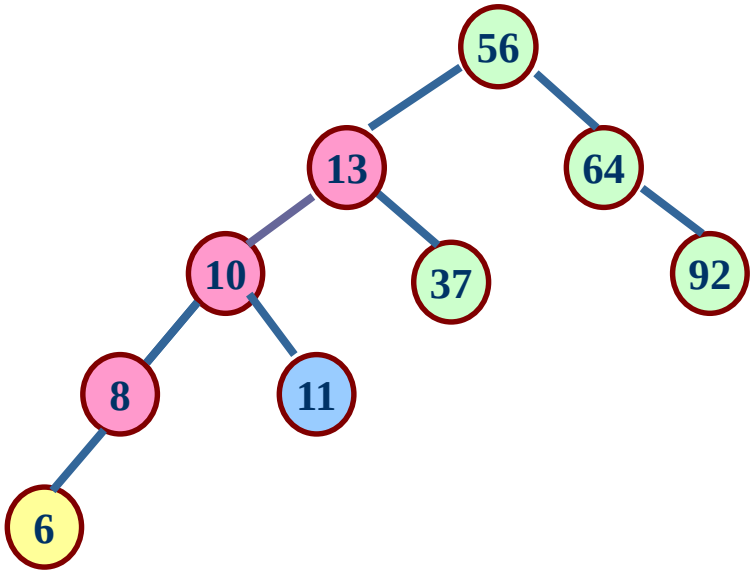


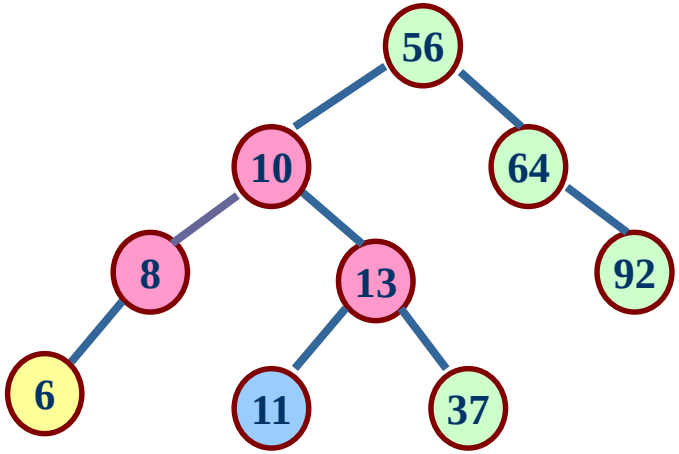
  - 实际操作，A变为B的右孩子，原B右孩子变为A左孩子

9.2 动态查找表

二. 平衡二叉树

- 举例，已知AVL树如下图，插入新数据6，求插入后的树



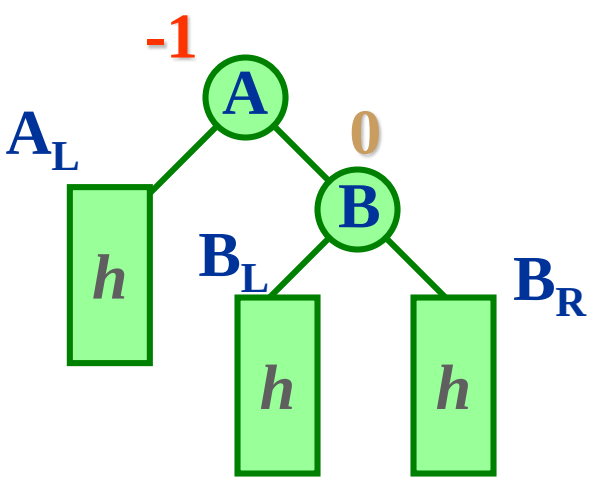


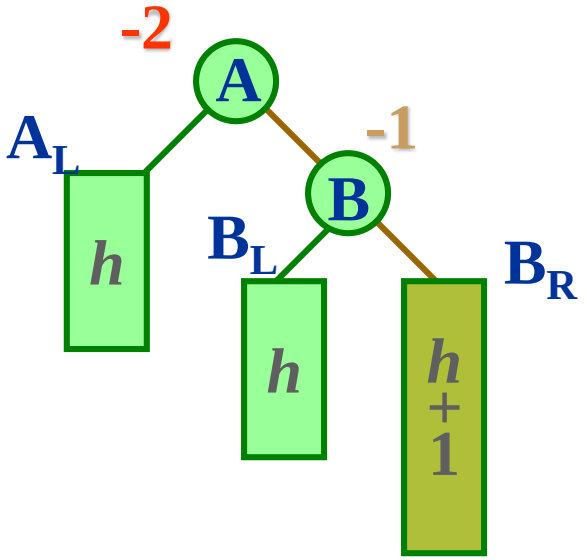
9.2 动态查找表

二. 平衡二叉树

■ 单向左旋(RR型)平衡处理

- ❑ 在结点A的右孩子结点B的右子树B_R中插入新结点，该子树高度增1，导致结点A的平衡因子变成-2，造成不平衡，则需进行一次向左旋转操作，进行平衡化。（A为失去平衡的最小子树根结点）

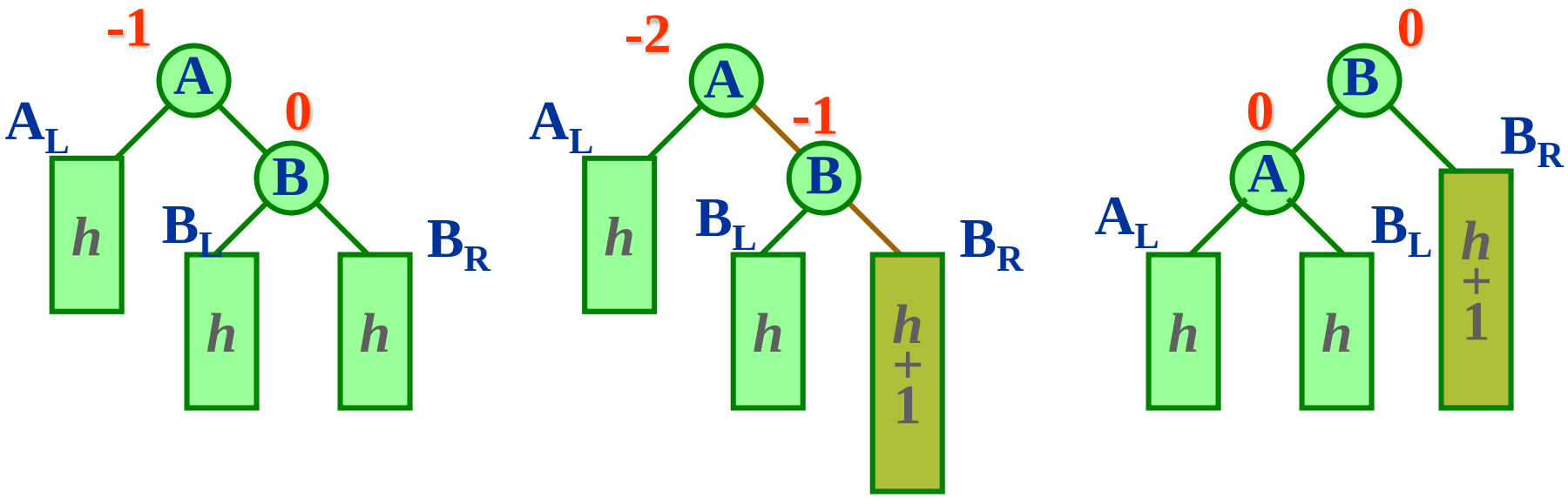




9.2 动态查找表

二. 平衡二叉树

- 单向左旋
  - 沿插入路径检查三个结点A、B和B_R（“\”RR型）
  - 以结点B为旋转轴，让结点A反时针（左）旋转

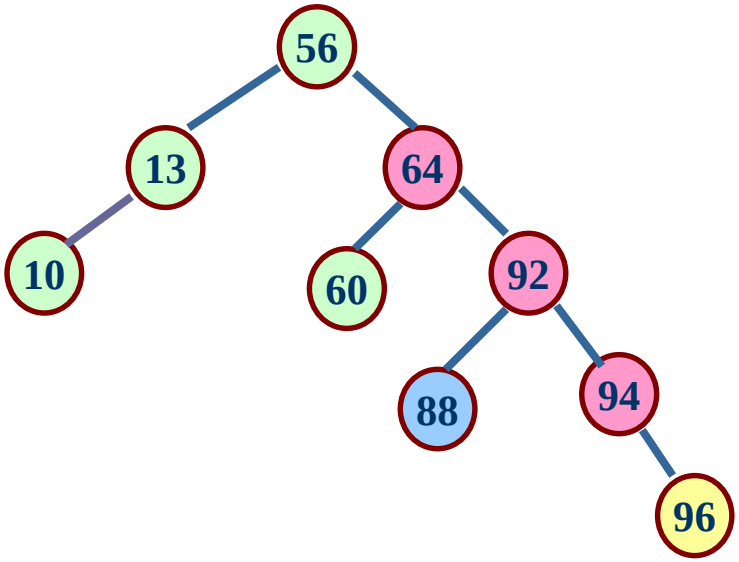


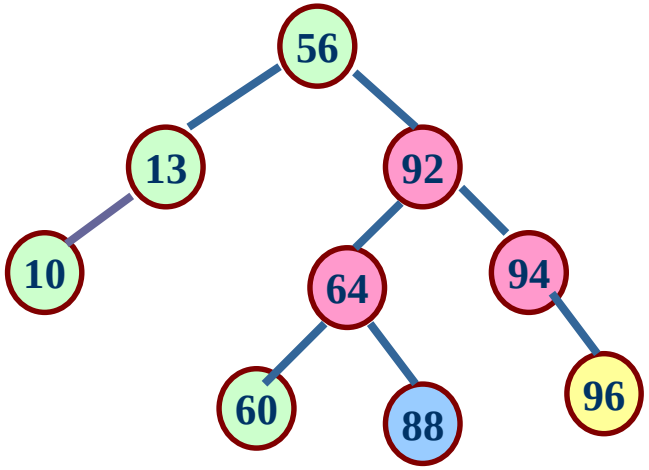
  - 实际操作，A变为B的左孩子，原B左孩子变为A右孩子

9.2 动态查找表

二. 平衡二叉树

- 举例，已知AVL树如下图，插入新数据96，求插入后的树

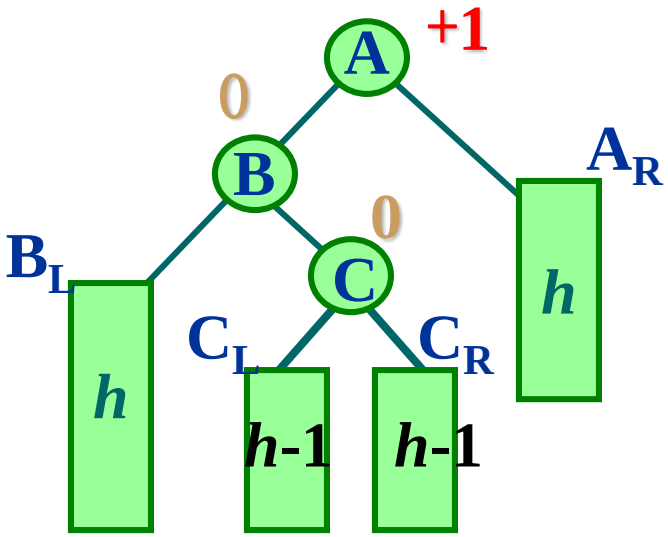


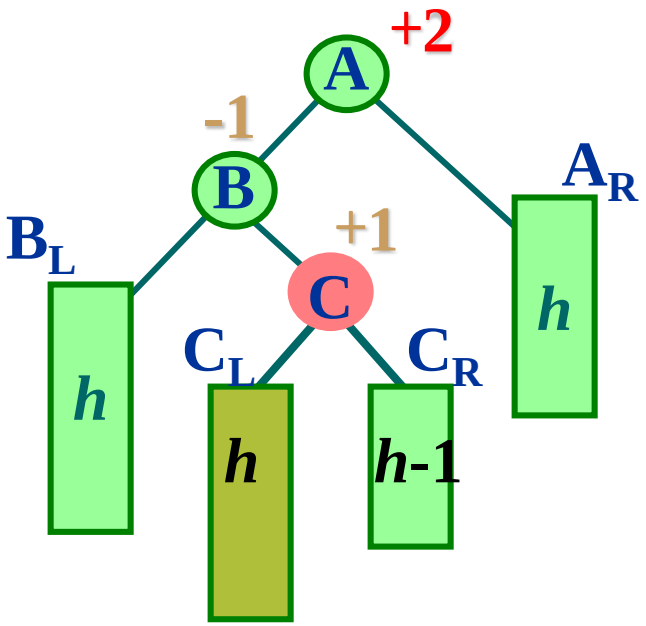


9.2 动态查找表

二. 平衡二叉树

- 先左后右双向旋转(LR型)平衡处理
  - 在结点A的左孩子结点B的右子树根结点C的子树C_L或C_R中插入新结点，结点A的子树失去平衡，则需进行先左旋后右旋两次旋转操作。（A为失去平衡的最小子树根结点）



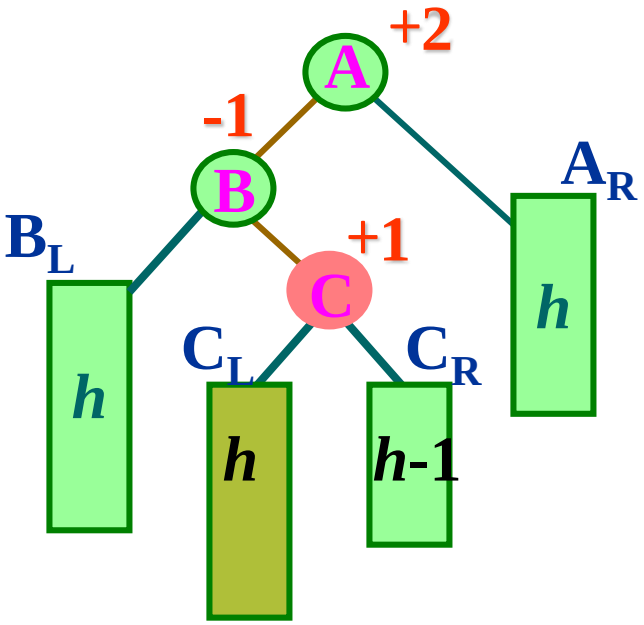


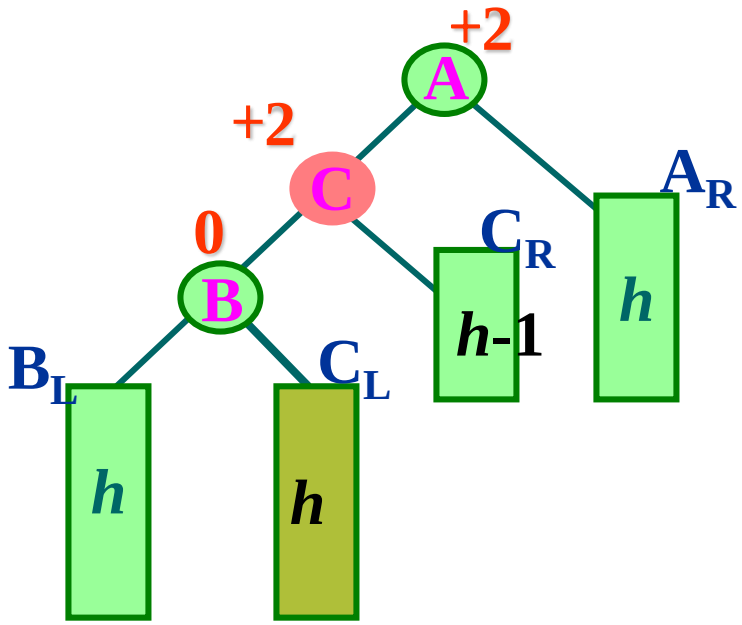
9.2 动态查找表

二. 平衡二叉树

- 先左后右双向旋转
  - 从结点A起沿插入路径选取3个结点A、B和C（“<” LR型）

步骤 1 ：先以结点B为旋转轴，做单向左旋





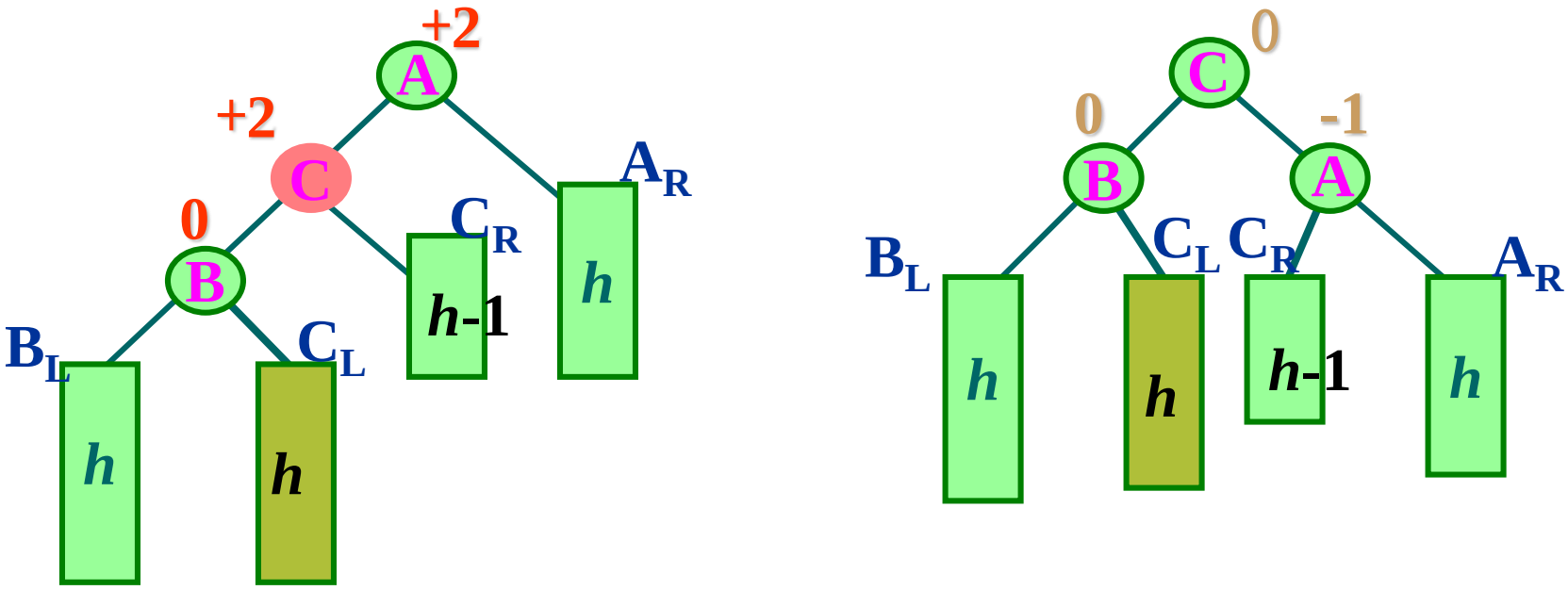
- A左孩子变为C(C替换B)，B变为C左孩子，原C左孩子变为B右孩子

9.2 动态查找表

二. 平衡二叉树

- 先左后右双向旋转

步骤 2：再以结点C为旋转轴，做单向右旋

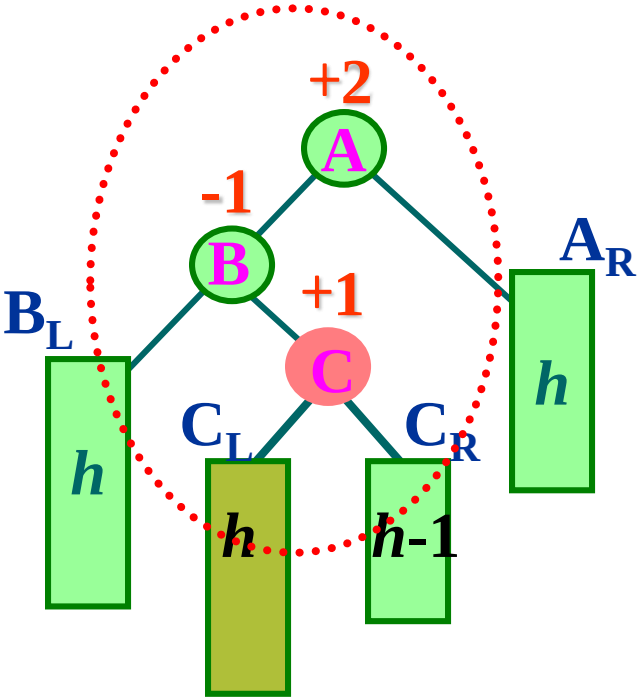


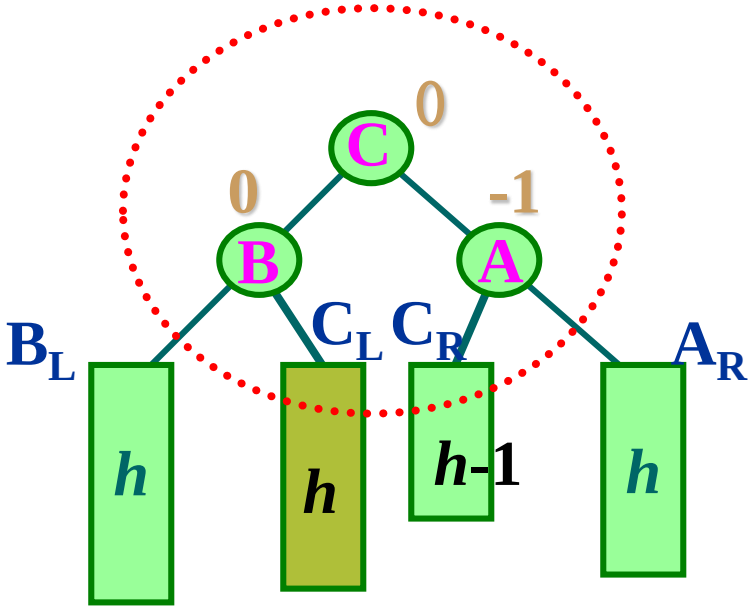
- A变为C右孩子，原C右孩子变为A左孩子

9.2 动态查找表

二. 平衡二叉树

- 先左后右双向旋转，全部操作
  - B变为C左孩子，A变为C右孩子
  - C左孩子变为B右孩子，C右孩子变为A左孩子

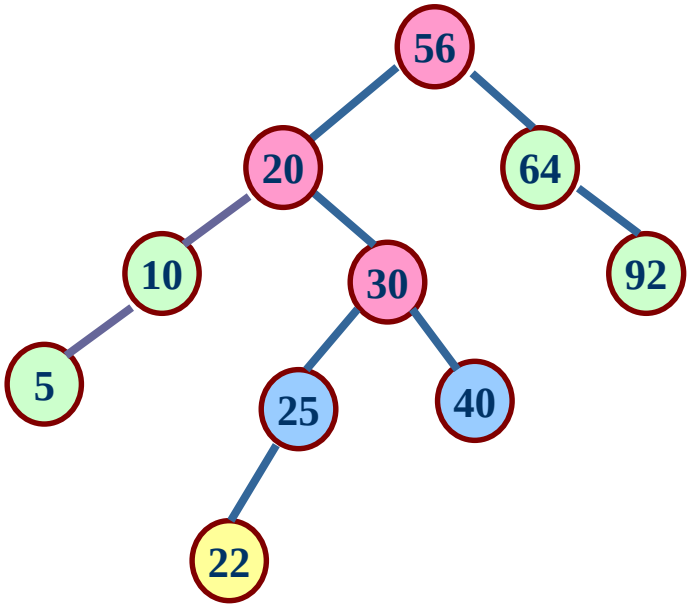


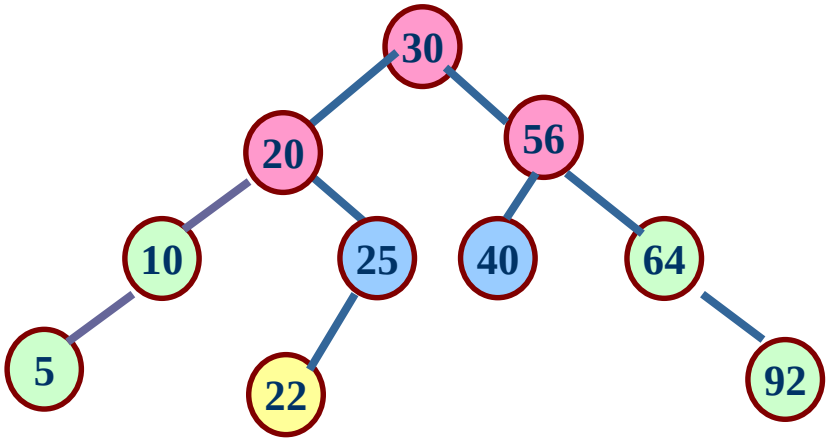


9.2 动态查找表

二. 平衡二叉树

- 举例，已知AVL树如下图，插入新数据22，求插入后的树

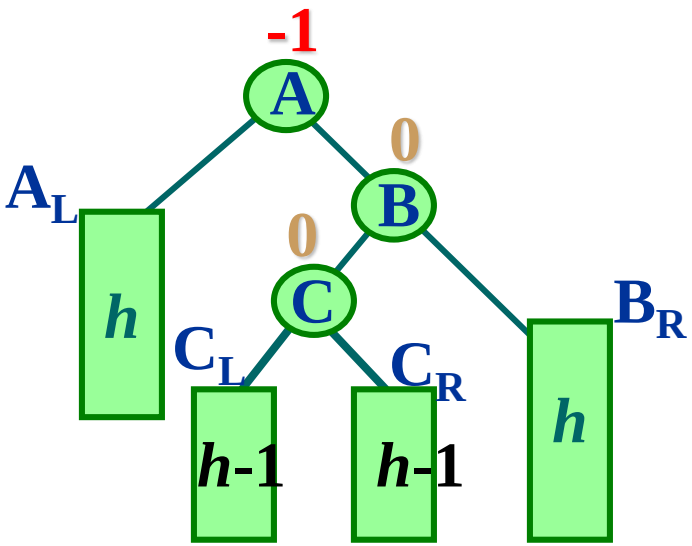


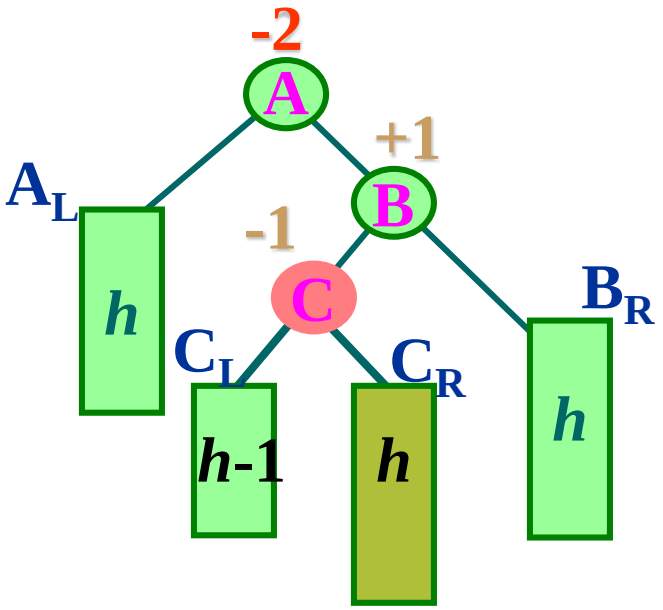


9.2 动态查找表

二. 平衡二叉树

- 先右后左双向旋转（RL型）平衡处理
  - 在结点A的右孩子结点B的左子树根结点C的子树C_L或C_R中插入新结点，结点A的子树失去平衡，则需进行先右旋后左旋两次旋转操作。（A为失去平衡的最小子树根结点）

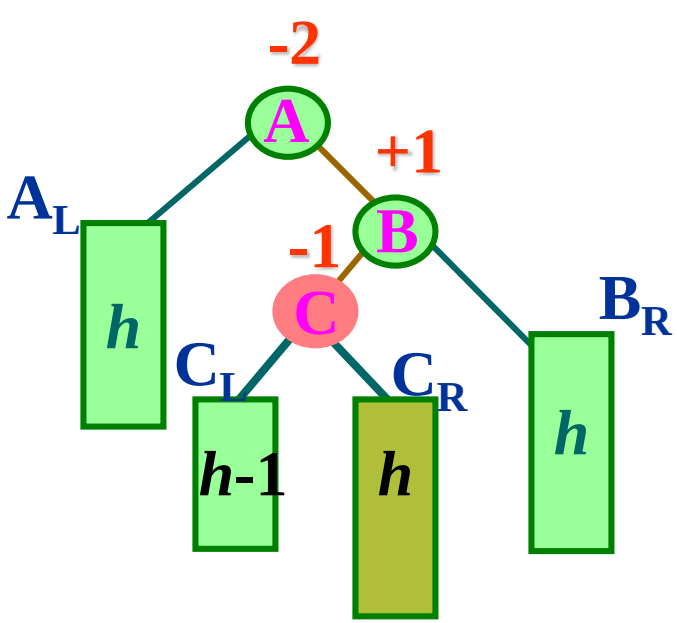


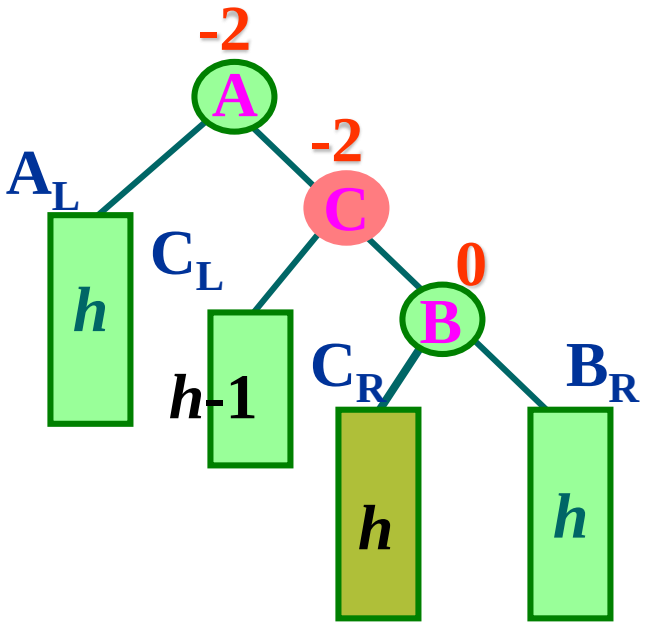


9.2 动态查找表

二. 平衡二叉树

- 先右后左双向旋转
  - 从结点A起沿插入路径选取3个结点A、C和D(“>”RL型)
  - 以结点B为旋转轴，作单向右旋



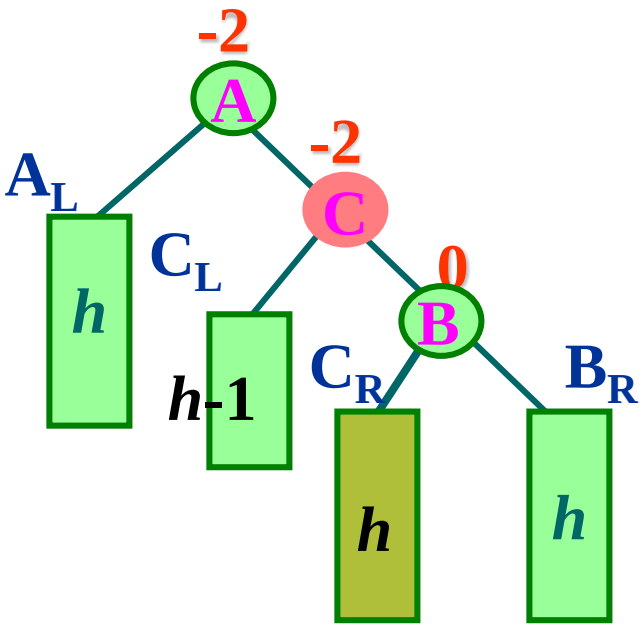


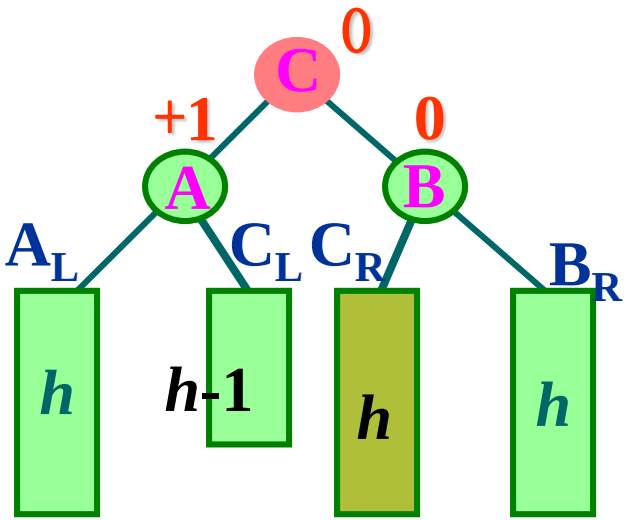
  - A右孩子变为C(C替换B)，B变为C右孩子，原C右孩子变为B左孩子

9.2 动态查找表

二. 平衡二叉树

- 先右后左双向旋转
  - 再以结点C为旋转轴，作单向左旋



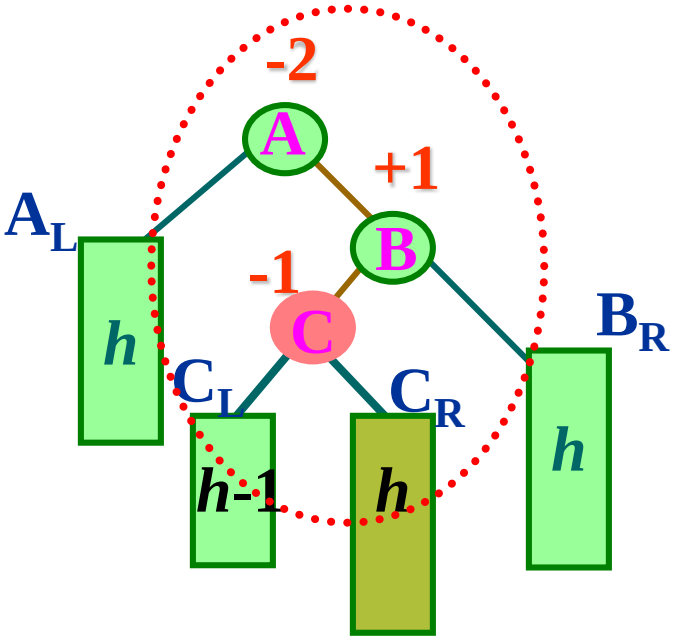


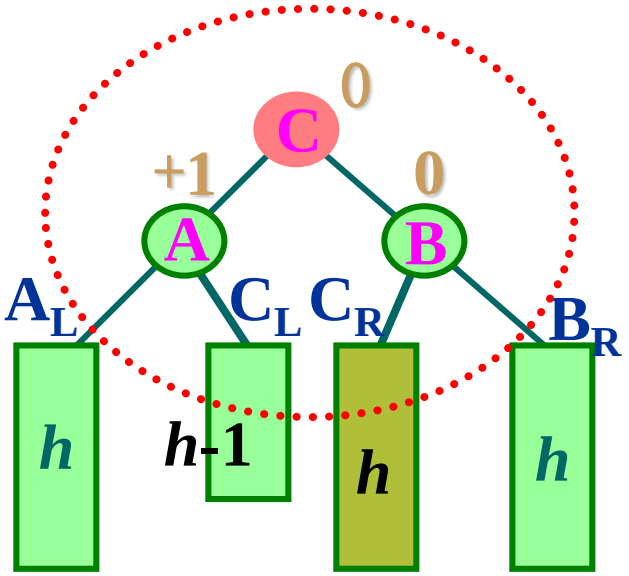
  - A变为C左孩子，原C左孩子变为A右孩子

9.2 动态查找表

二. 平衡二叉树

- 先右后左双向旋转，最终操作
  - A变为C左孩子，B变为C右孩子
  - C左孩子变为A右孩子，C右孩子变为B左孩子

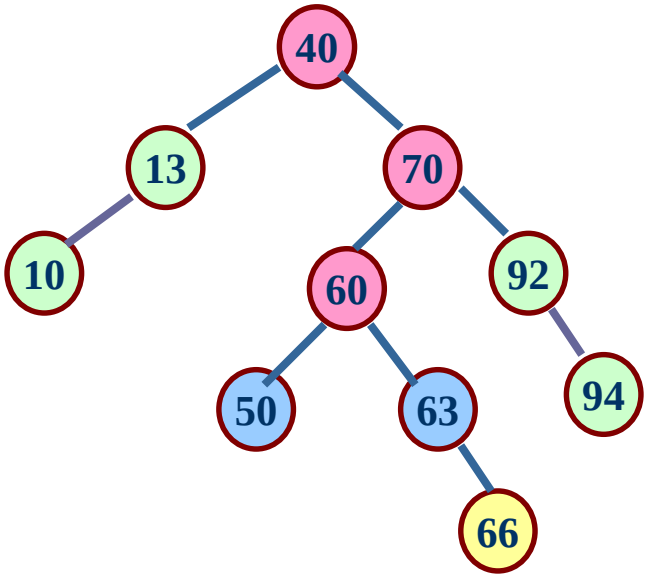


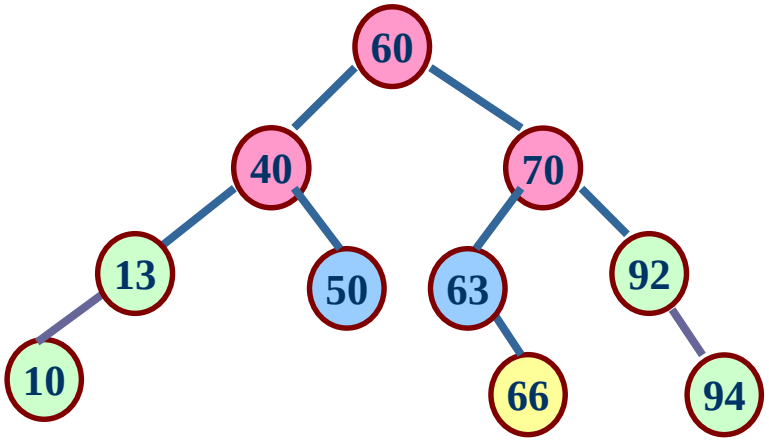


9.2 动态查找表

二. 平衡二叉树

- 举例，已知AVL树如下图，插入新数据66，求插入后的树





9.2 动态查找表

二. 平衡二叉树

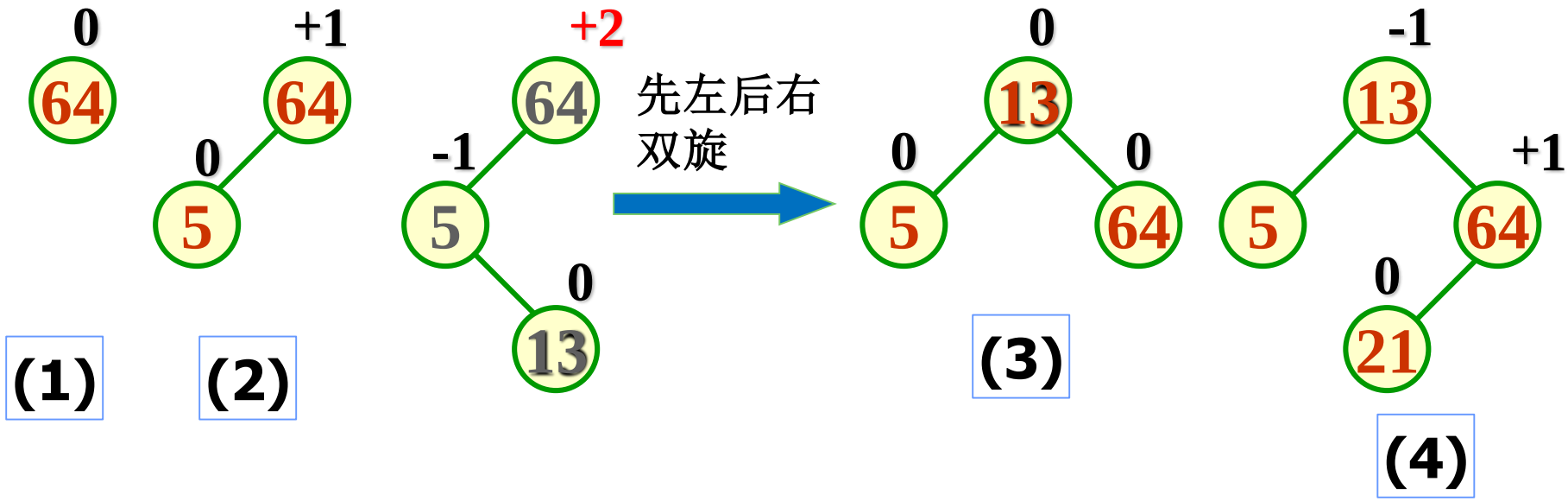
平衡二叉树的删除

- 如果被删结点A没有孩子，则直接删除之，并作平衡化处理
- 如果被删结点A最多只有一个孩子，那么将结点A从树中删去，并将其双亲指向它的指针指向它的唯一的孩子，并作平衡化处理
- 如果被删结点A有两个子女，则用该结点的直接前驱S替代被删结点，然后对直接前驱S作删除处理(S只有一个孩子或没有孩子)
- 和二叉排序树的删除一样，增加了平衡化处理

9.2 动态查找表

二. 平衡二叉树

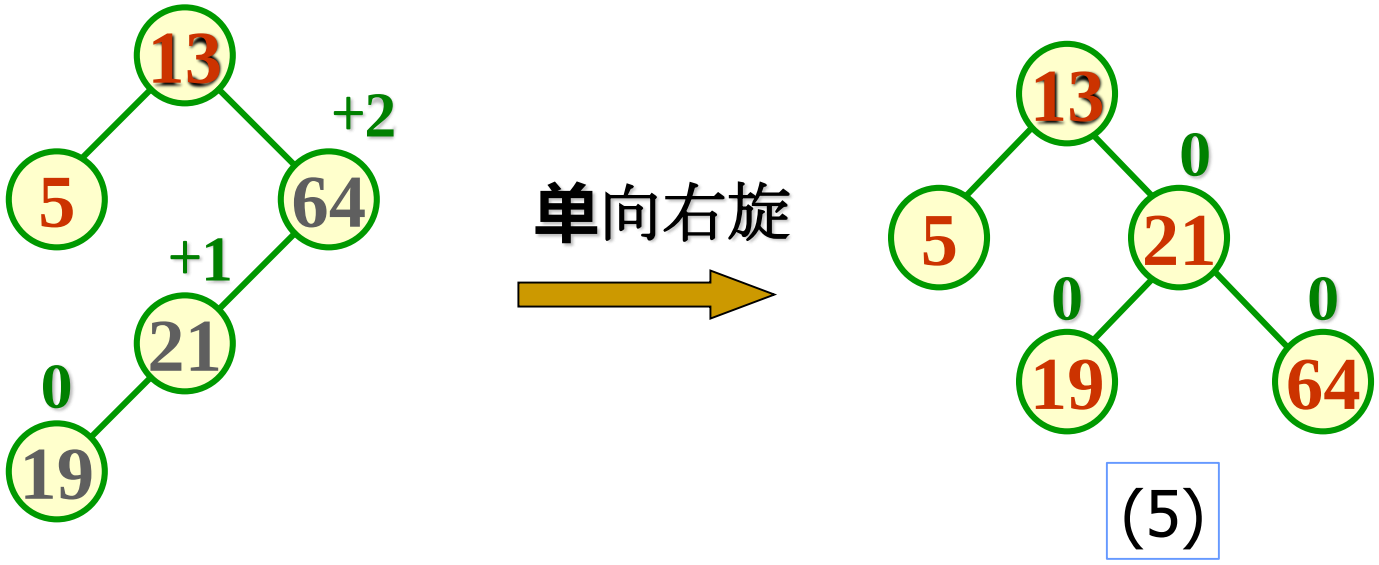
- 综合举例
  - 画出在初始为空的AVL树中依次插入64, 5, 13, 21, 19, 80, 75, 37, 56的生长过程，并在有旋转时说出旋转的类型。



9.2 动态查找表

二. 平衡二叉树

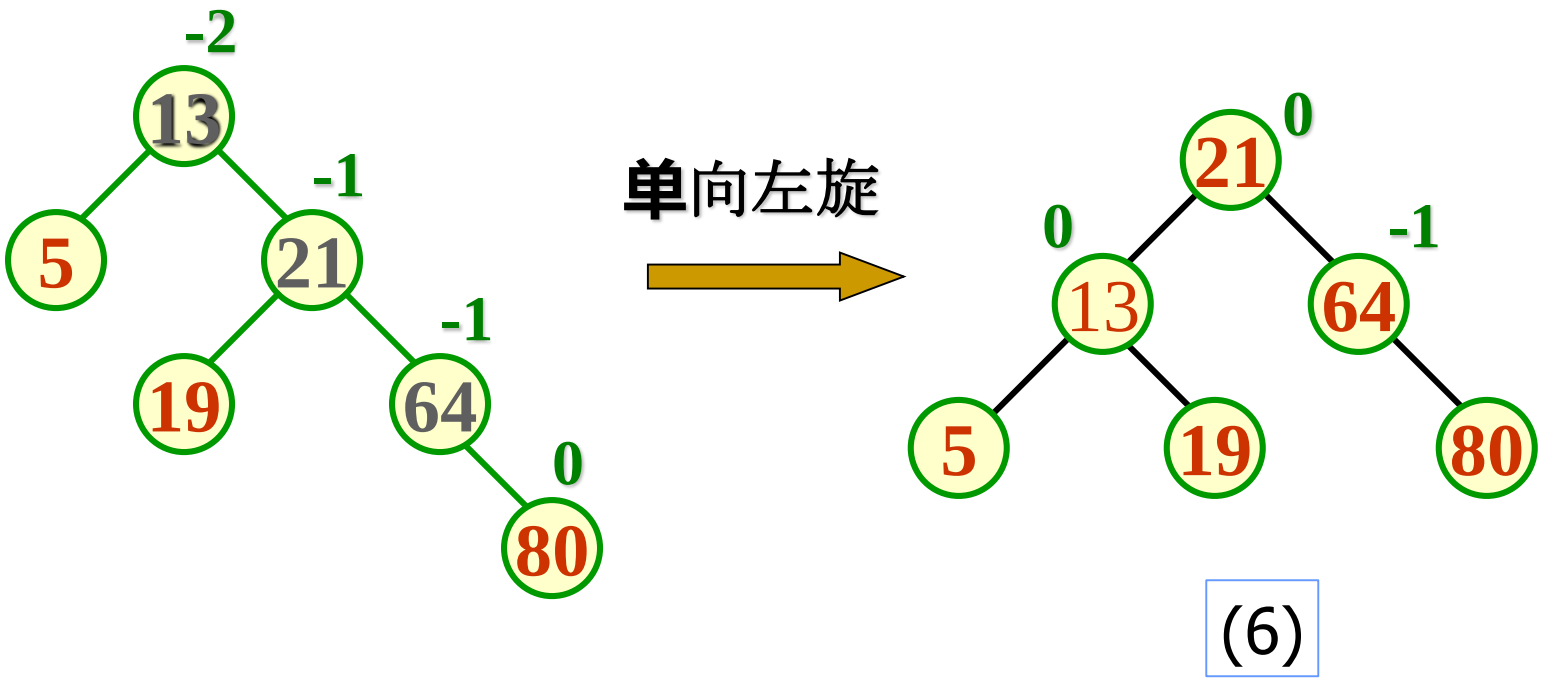
- 举例
  - 在完成*64, 5, 13, 21*后，继续插入19, 80, 75, 37, 56时该树的生长过程，



9.2 动态查找表

二. 平衡二叉树

- 举例
  - 在完成*64, 5, 13, 21, 19*后，继续插入80, 75, 37, 56时该树的生长过程，

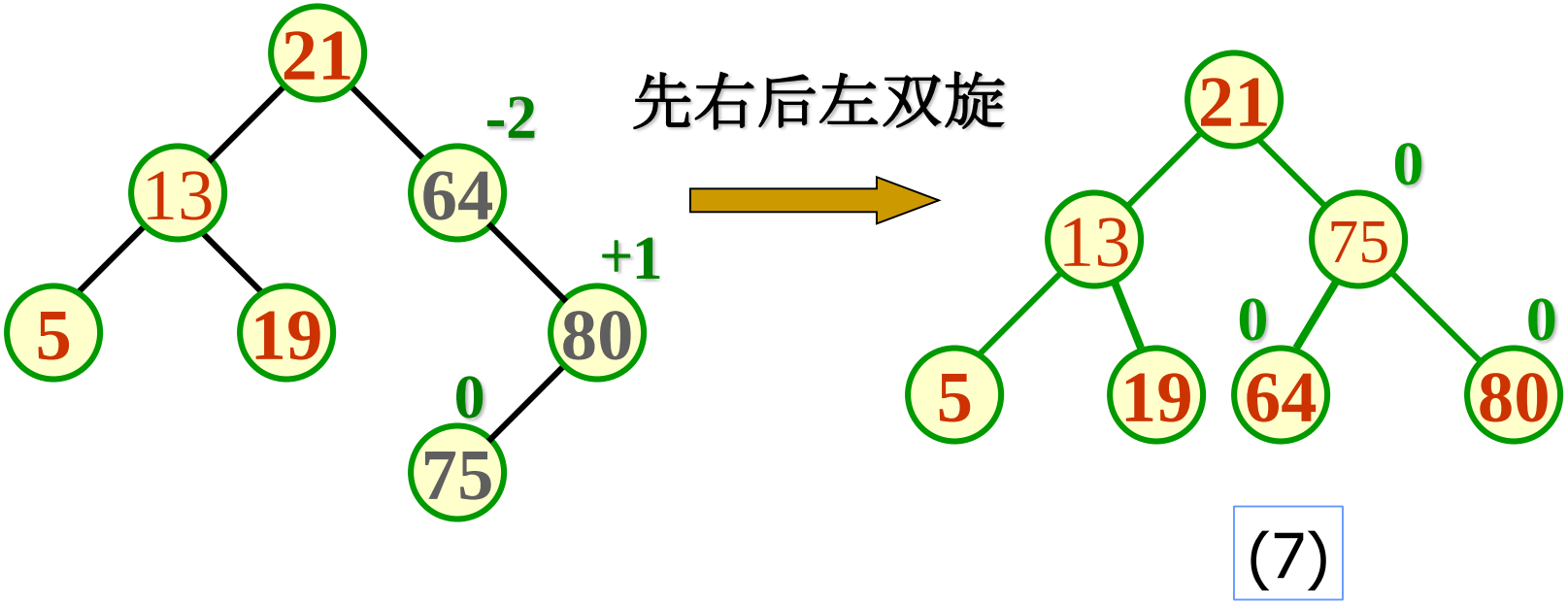


(6)

9.2 动态查找表

二. 平衡二叉树

- 举例
  - 在完成*64, 5, 13, 21, 19, 80*后，继续插入75, 37, 56时该树的生长过程，

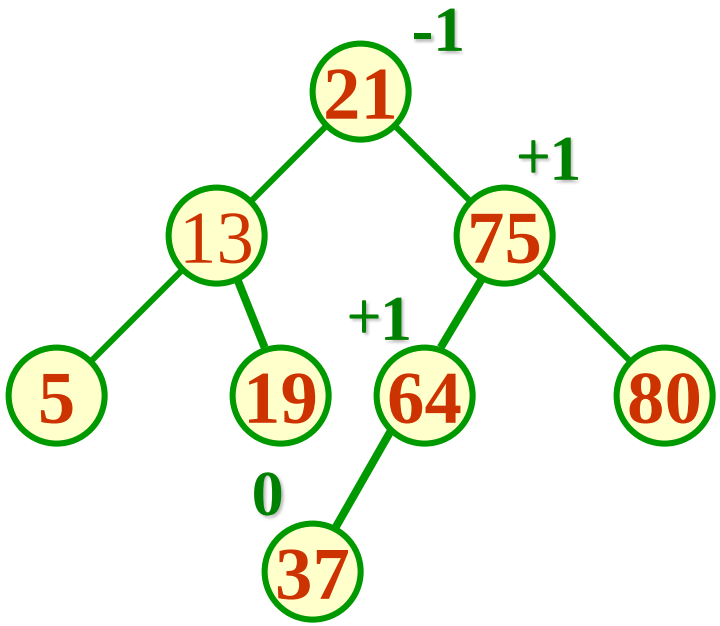


(7)

9.2 动态查找表

二. 平衡二叉树

- 举例
  - 在完成*64, 5, 13, 21, 19, 80, 70*后，继续插入37, 56时该树的生长过程，

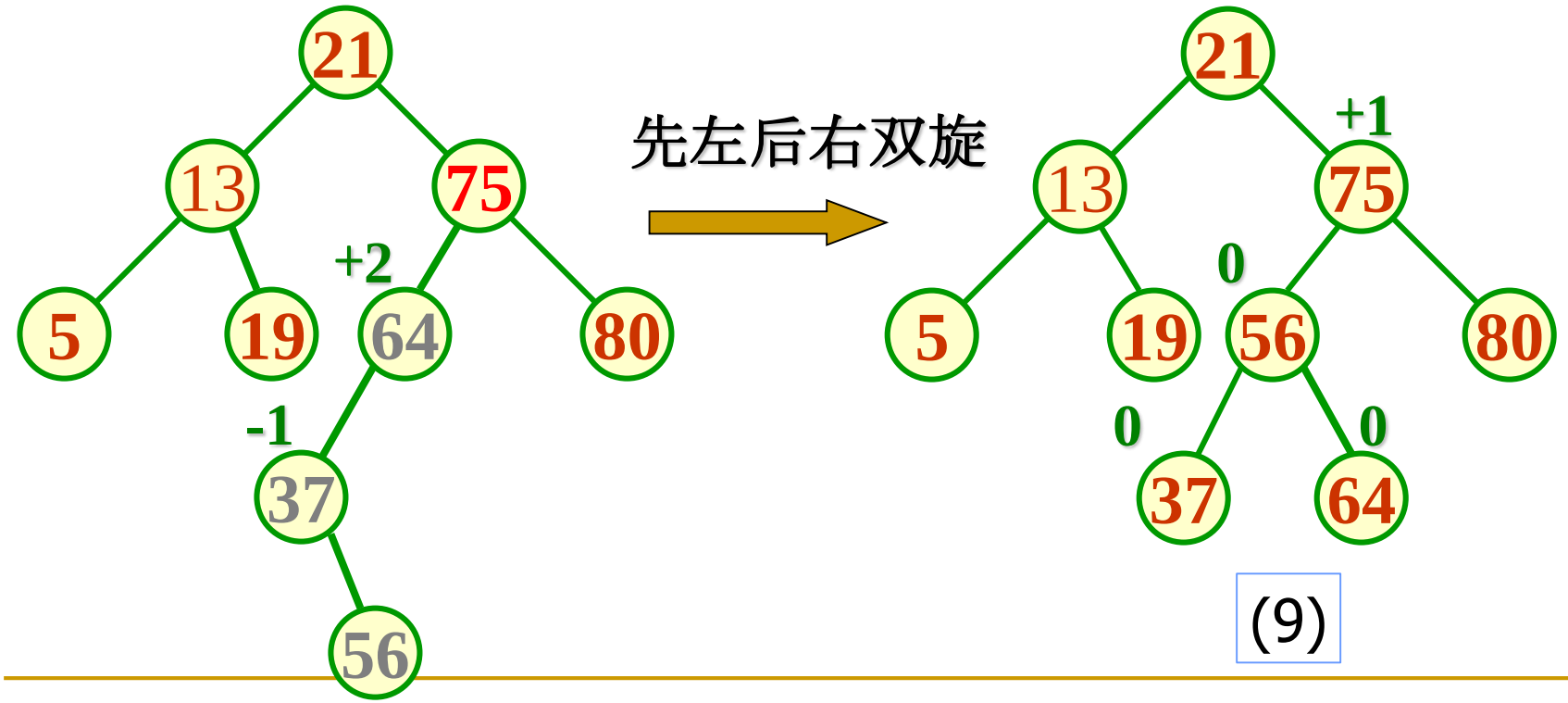


(8)

9.2 动态查找表

二. 平衡二叉树

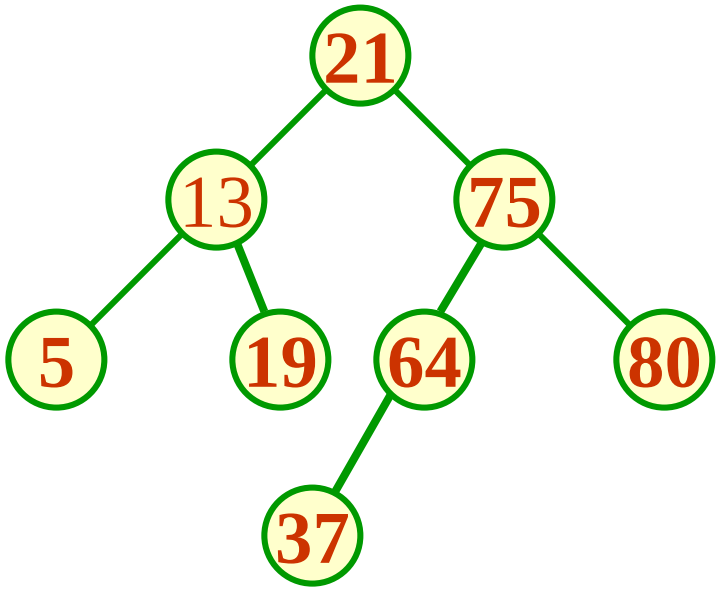
- 举例
  - 在完成*64, 5, 13, 21, 19, 80, 70, 37*后，继续插入56时该树的生长过程，



9.2 动态查找表

二. 平衡二叉树

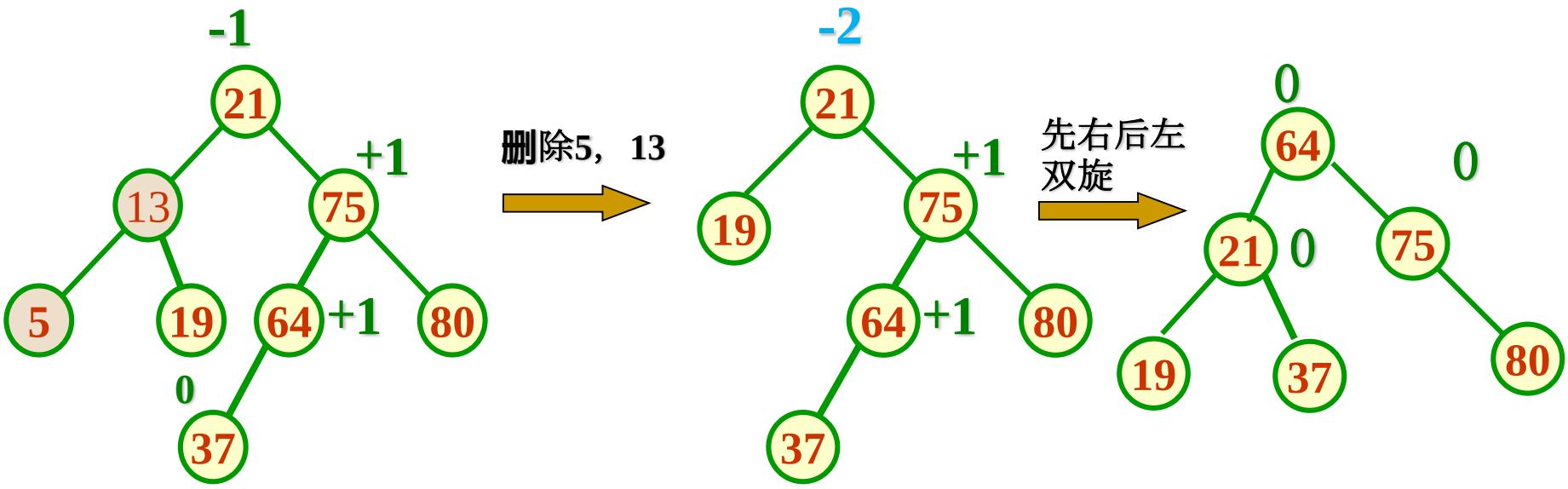
- 平衡二叉树删除举例
  - 独立删除5、64、37，无需平衡化处理
  - 独立删除80，单向右旋处理



9.2 动态查找表

二. 平衡二叉树

- 平衡二叉树删除举例
  - 连续删除5和13，以21-75-64做先右后左双旋



9.2 动态查找表

二. 平衡二叉树

- 平衡二叉排序树的时间复杂度

平衡二叉排序树使得二叉树排序树的结构更好，从而提高了查找操作的速度。但是使得插入和删除操作复杂化，从而降低了插入和删除操作的速度。因此，平衡二叉树适合于二叉排序树一经建立就很少进行插入和删除操作，而主要是进行查找操作的应用场合中。

由于平衡二叉树在查找过程中和给定值进行比较的关键字个数不超过树的深度，因此，在平衡二叉树上进行查找的时间复杂度为**O(logn)**。

练习

一．在初始为空的平衡二叉树中依次插入33、88、66、22、11、15、44、40

- 请画出最终的平衡二叉树，并对其中的平衡化处理做简单说明
- 若删除结点11、88，请画出删除后的平衡二叉树

练习

- 画出在初始为空的AVL树中依次插入2、1、3、5、8、4、7、6时该树的生长全过程，并在有“旋转”时说出“旋转”的类型。

上节复习

- 平衡二叉树，又称AVL树，树中每个结点的左、右子树深度之差的绝对值不大于1，即$|h_L-h_R|\leqslant 1$
  - 平衡因子，左子树的高度减去右子树的高度所得的高度差
- 平衡化处理，又称平衡化旋转
  - 从不平衡的结点往下两层，共三个结点从上往下ABC
  - 单向右旋，三结点成左斜直线，做右旋处理
    - A变为B的右孩子，原B右孩子变A左孩子
  - 单向左旋，三结点成右斜直线，做左旋处理
    - A变为B的左孩子，原B左孩子变A右孩子
  - 先左后右旋转，三结点成左折型，先左旋再右旋处理
    - B变为C左孩子，A变为C右孩子
    - C左孩子变为B右孩子，C右孩子变为A左孩子
  - 先右后左旋转，三结点成右折型，先右旋再左旋处理
    - A变为C左孩子，B变为C右孩子
    - C左孩子变为A右孩子，C右孩子变为B左孩子
- 平衡二叉树的插入/删除，先插入/删除，检查平衡因子，平衡化处理

9.2 动态查找表

三. B-树和B+树

- 前面所讨论的查找算法都是在内存中进行的，它们适用于较小的文件，而对于较大的、存放在外存储器上的文件，就涉及到内、外存之间频繁的数据交换。
- 平衡二叉树的每个结点只能有两个孩子，它自身也只能存储一个元素。在元素非常多的时候，就使得树的高度非常大，对于大规模的文件，在查找效率上非常低。
- 因此，需要打破每一个结点只存储一个元素的限制。

9.2 动态查找表

三. B-树和B+树

- **R.Bayer和E.Mc Creight在1972年提出了一种多路平衡查找树，称为B-树（又称：B树、B_树），其变型体是B+树。**
- 是一种在外存文件系统中常用的动态索引技术

9.2 动态查找表

三. B-树和B+树

- **m阶B-树：** 一棵度为m的B-树，其定义是：
  - 或者是空树，或者是满足以下性质的m叉树：

(1) 根结点或者是叶子，或者至少有两棵子树，至多有m棵子树；

(2) 除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，至多有m棵子树

(3) 每个结点应包含如下信息：

$(n, A_0, K_1, A_1, K_2, A_2, \cdots, K_n, A_n)$

n个关键字，n+1个子树指针，

(4) 所有叶子结点都出现在同一层次，且不带任何信息（可以把叶子看作外部结点或查找不成功的结点。实际上叶子不存在，指向叶子的指针也为空）。

9.2 动态查找表

三. B-树和B+树

- **B-树的每个结点应包含如下信息：**

  $(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

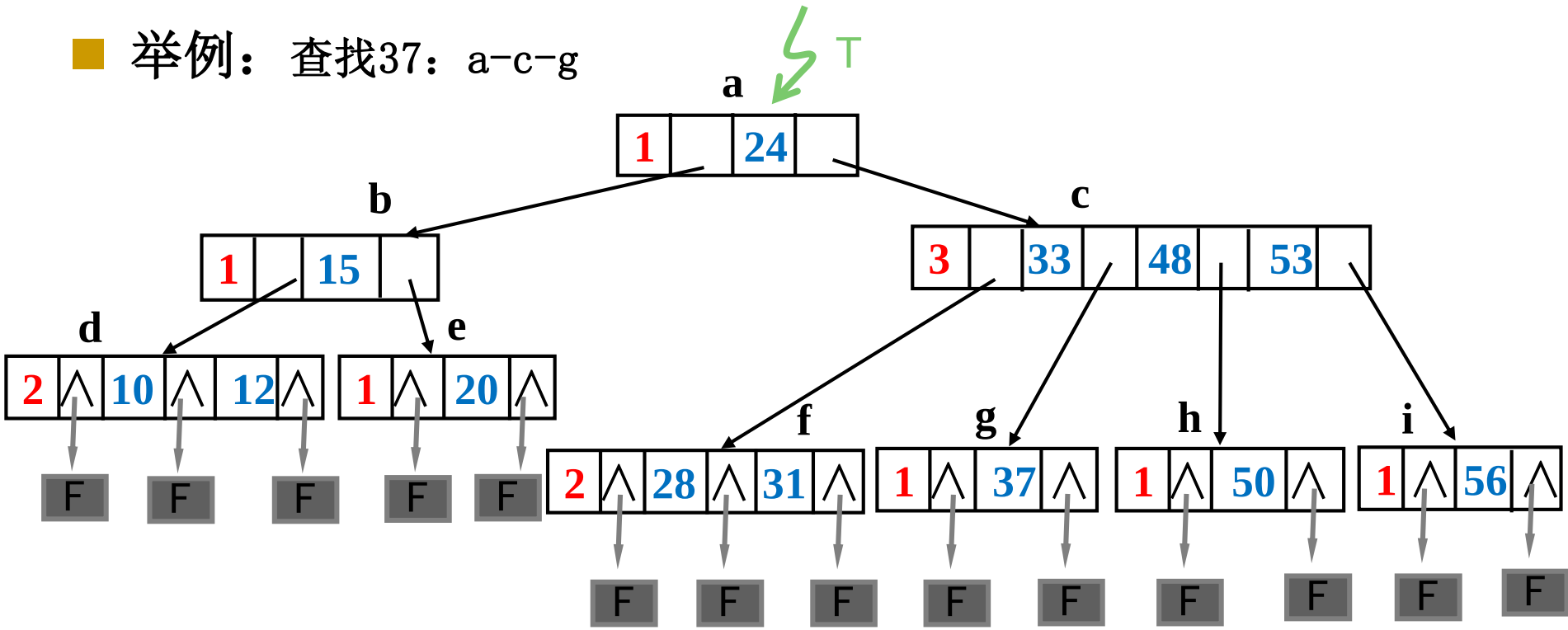
  - **n是结点中关键字的个数，且 $\lceil m/2 \rceil - 1 \le n \le m-1$（或n+1为子树个数）**
  - **K_i(1≤i≤n)是关键字，且 $K_i < K_{i+1}$ (1≤i≤n-1)；**
  - **A_i(i=0，1，…，n)为指向孩子结点的指针，A_{i-1}所指向的子树中所有结点的关键字都小于K_i，A_i所指向的子树中所有结点的关键字都大于K_i；**

9.2 动态查找表

三. B-树和B+树

- **B-树的查找**
  - 顺指针查找结点和在结点中查找关键字交替进行

- 举例：查找37：a-c-g



※若m阶B_树中有N个关键字，则叶子结点即查找不成功的结点为N+1个。

9.2 动态查找表

三. B-树和B+树

- **B-树的查找过程和BST的查找相似**

① 从树的根结点T开始，在T所指向的结点的关键字向量key[1…keynum]中查找给定值K(用折半查找)：

若key[i]=K(1≤i≤keynum)，则查找成功，返回结点及关键字位置；否则，转 ② ；

② 将K与向量key[1…keynum]中的各个分量的值进行比较，以选定查找的子树：

- 若K<key[1]， T = T->ptr[0]；
- 若key[i]<K<key[i+1](i=1, 2, …keynum-1)， T = T->ptr[i]；
- 若K>key[keynum]， T = T->ptr[keynum]；

转①，直到T是叶子结点且未找到相等的关键字，则查找失败。

9.2 动态查找表

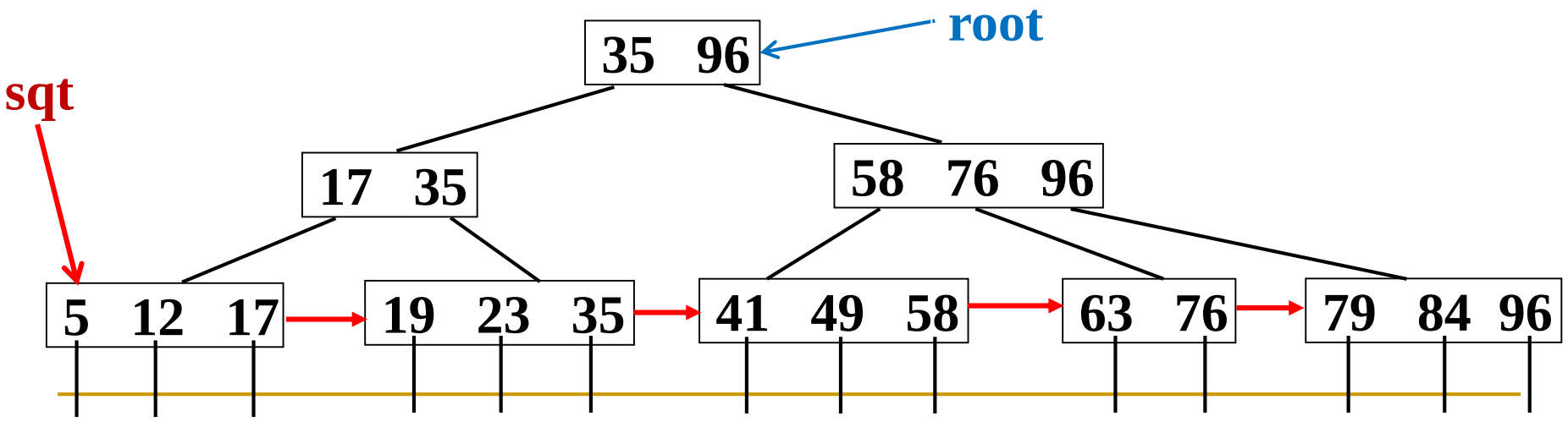
三. B-树和B+树

- **B+树与B-树的主要差异是：**

① 若一个结点有n棵子树，则必含有n个关键字；

② 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；在叶子结点上删除关键字的情况

③ 所有的非终端结点可以看成是索引的部分，非终端结点中只含有其子树的根结点中的最大(或最小)关键字。



9.2 动态查找表

三. B-树和B+树

B+树的查找

- 与B-树相比，对B+树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。在B+树上进行随机查找、插入、删除的过程基本上和B-树类似。
- 在B+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录)， 即无论查找成功与否，都走了一条从根结点到叶子结点的路径。

9.2 动态查找表

三. B-树和B+树

B+树的应用

- 文件系统和数据库系统中常用到B/B+ 树

B/B+ 树通过对每个结点存储个数的扩展，使得对连续的数据能够进行较快的定位和访问，能够有效减少查找时间，提高存储的空间局部性，从而减少IO操作，广泛用于文件系统及数据库中，如：

- Windows：HPFS文件系统
- Mac：HFS，HFS+文件系统
- Linux：ResiserFS，XFS，Ext3FS，JFS文件系统
- 数据库：ORACLE，MYSQL，SQLSERVER等中