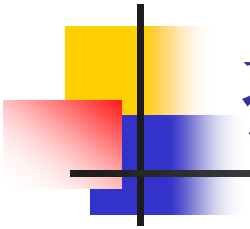
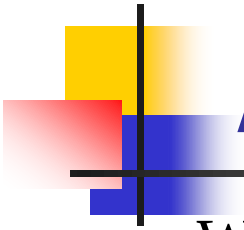


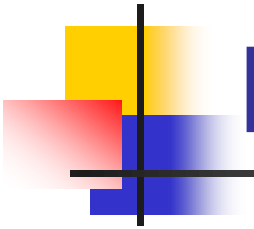
搜索/查找 (Search)

- Search基本概念
- Search算法
  - 静态搜索表
  - 二叉搜索树
  - AVL树
- B树
- B+树
- Hashing
- Search算法的分析



AVL Tree

- With equal search probabilities for keys, both the maximum and average search time will be minimized if the binary search tree is maintained as a complete binary tree at all times.
- However, since the dynamic situation, it is difficult to achieve this without making the time required to insert a key very high.
- It is, however, possible to keep the tree balanced to ensure both average and worst-case retrieval time of O(log n) for a tree with n nodes.



Definition

AVL tree (introduced by Adelson-Velskii and Landis) is a binary search tree that is balanced with respect to the heights of subtrees.

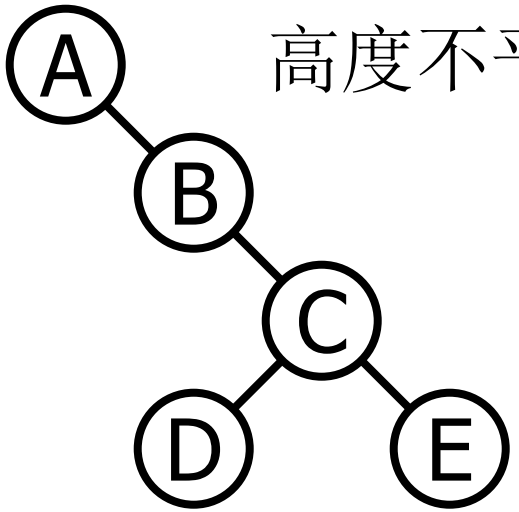
Definition: an empty tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is height-balanced iff

(1) T_L and T_R are height-balanced and

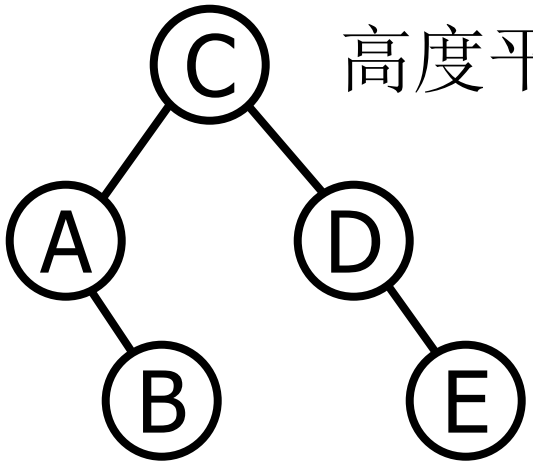
(2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

AVL树 高度平衡的二叉搜索树

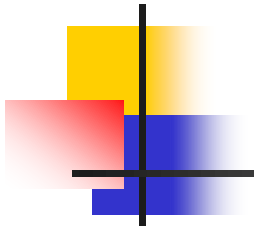
- 一棵 AVL 树或者是空树
- 或者是具有下列性质的二叉搜索树：
  - 它的左子树和右子树都是 AVL 树
  - 且左子树和右子树的高度之差的绝对值不超过1



高度不平衡



高度平衡

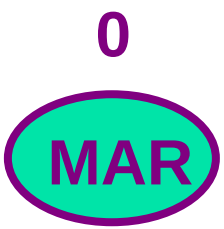


- Definition: The balance factor, BF(T), of a node T in a binary tree is defined to be $h_L - h_R$. For any node T in an AVL tree BF(T)=-1, 0, or 1.

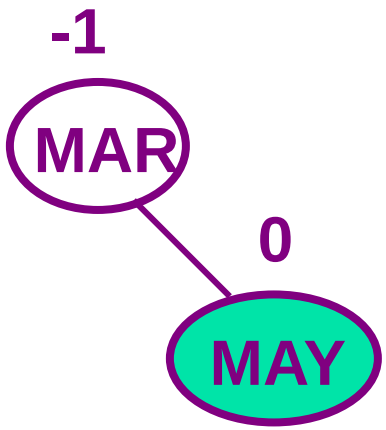
结点的平衡因子bf (balance factor)

- 结点的平衡因子bf = 右子树的高度 - 左子树的高度
- AVL树任一结点平衡因子只能取 –1, 0, 1
- 如果一个结点的平衡因子的绝对值大于1，则这棵二叉搜索树就失去了平衡，不再是AVL树
- 如果一棵有 n 个结点的二叉搜索树是高度平衡的，其高度可保持在$O(\log_2 n)$，平均搜索长度也可保持在$O(\log_2 n)$

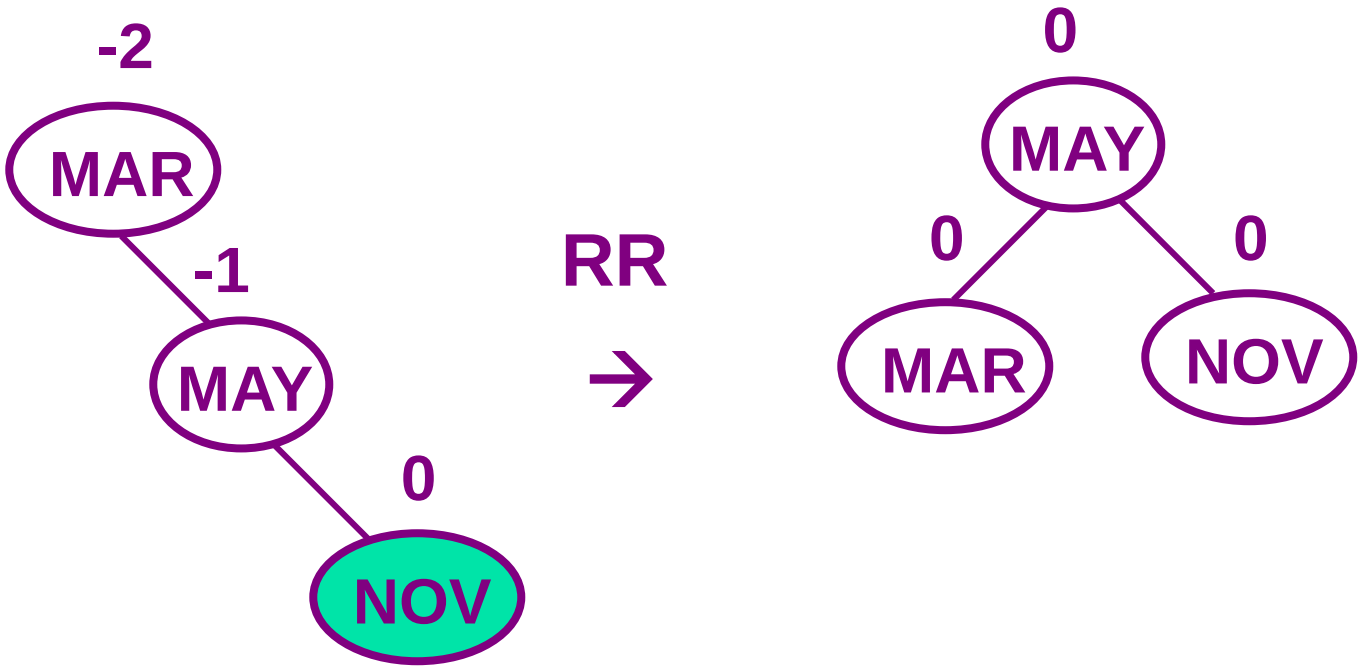
Insert MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB



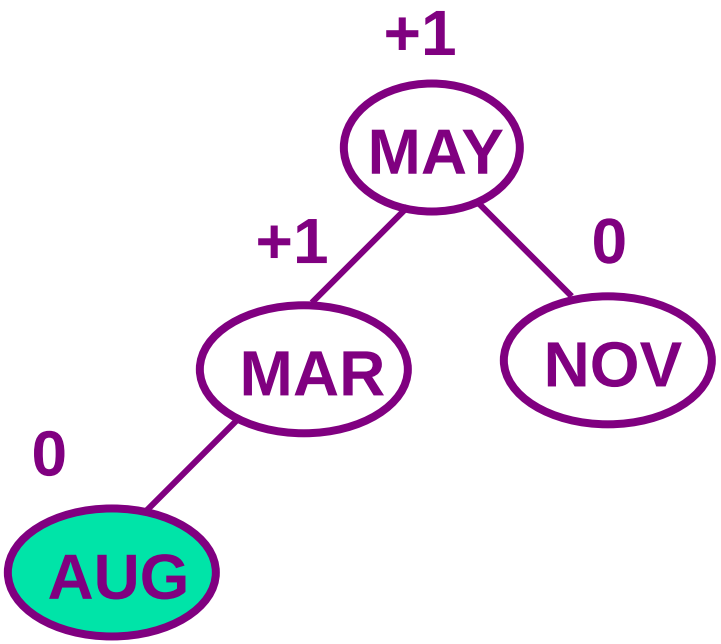
(a) Insert MAR



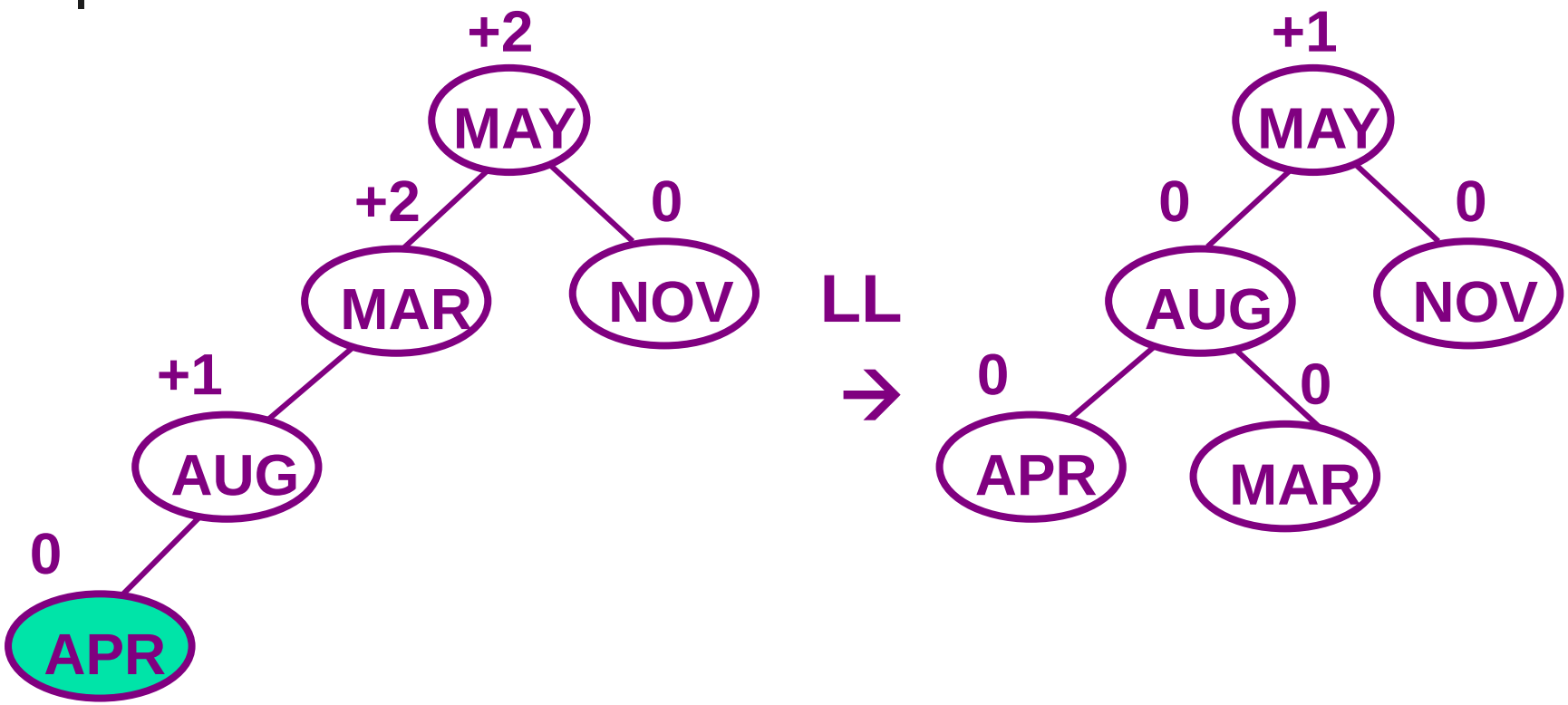
(b) Insert MAY



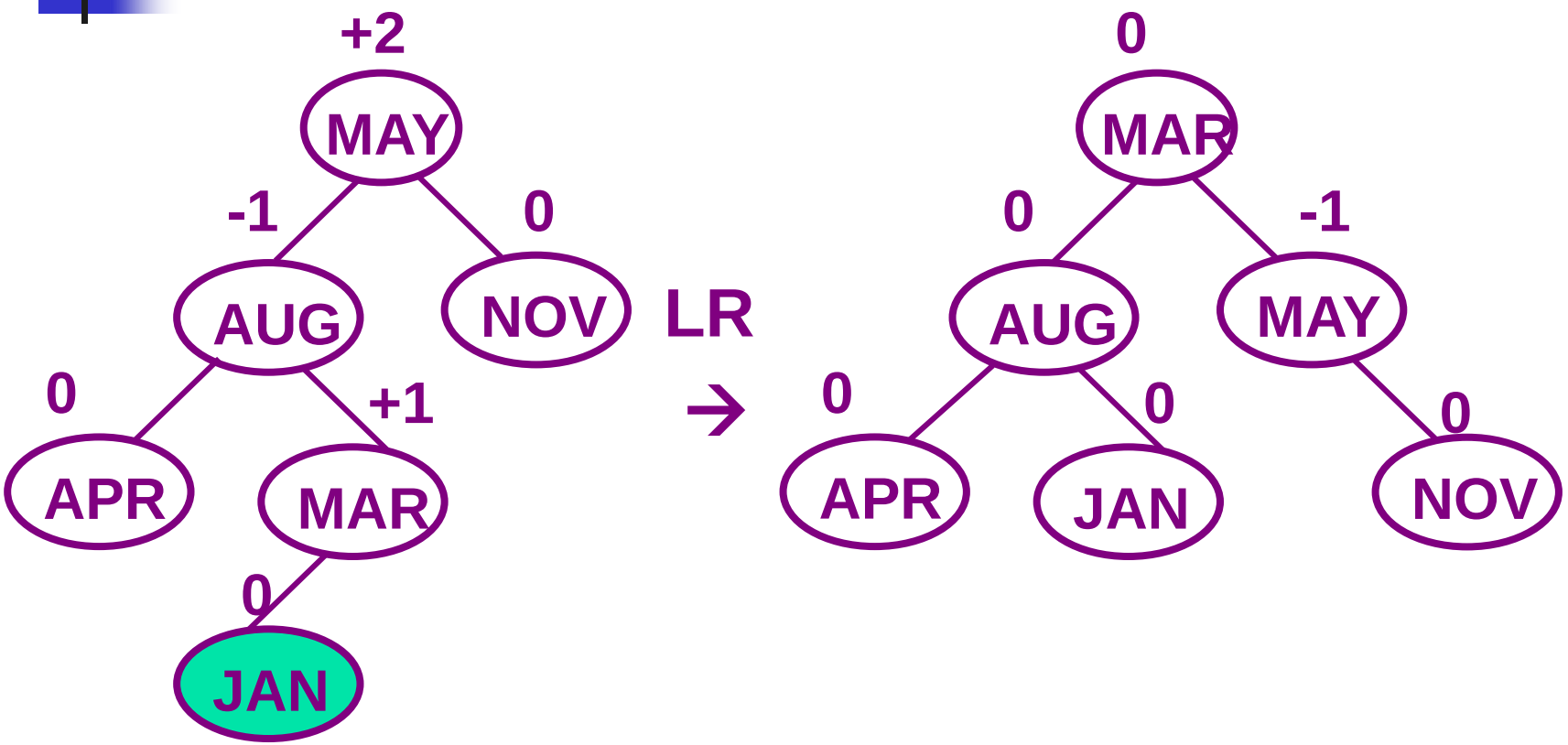
(c) Insert NOV



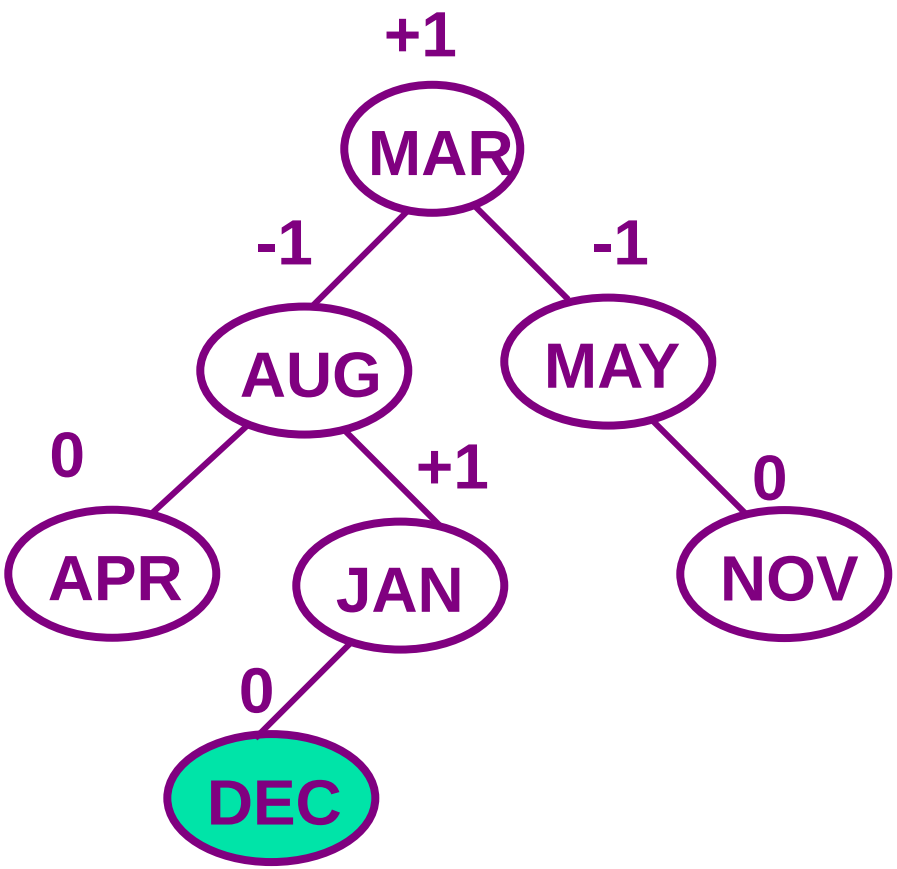
(d) Insert AUG



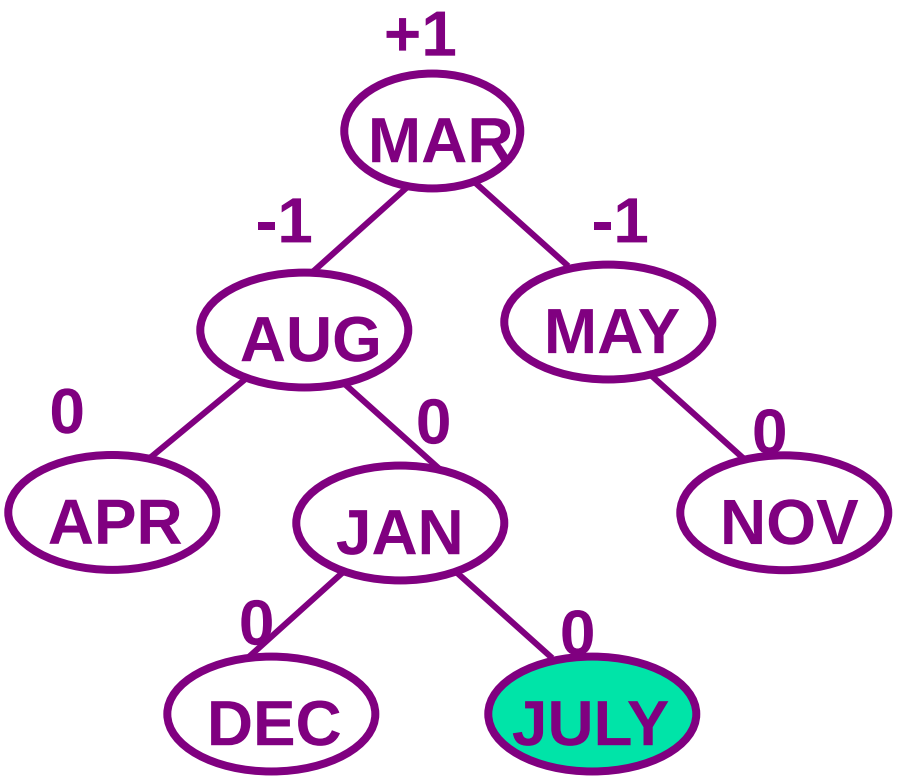
(e) Insert APR


+2
MAY
-1
0
AUG
NOV
0
+1
APR
MAR
0
JAN
LR
0
MAR
0
-1
AUG
MAY
0
0
0
APR
JAN
NOV

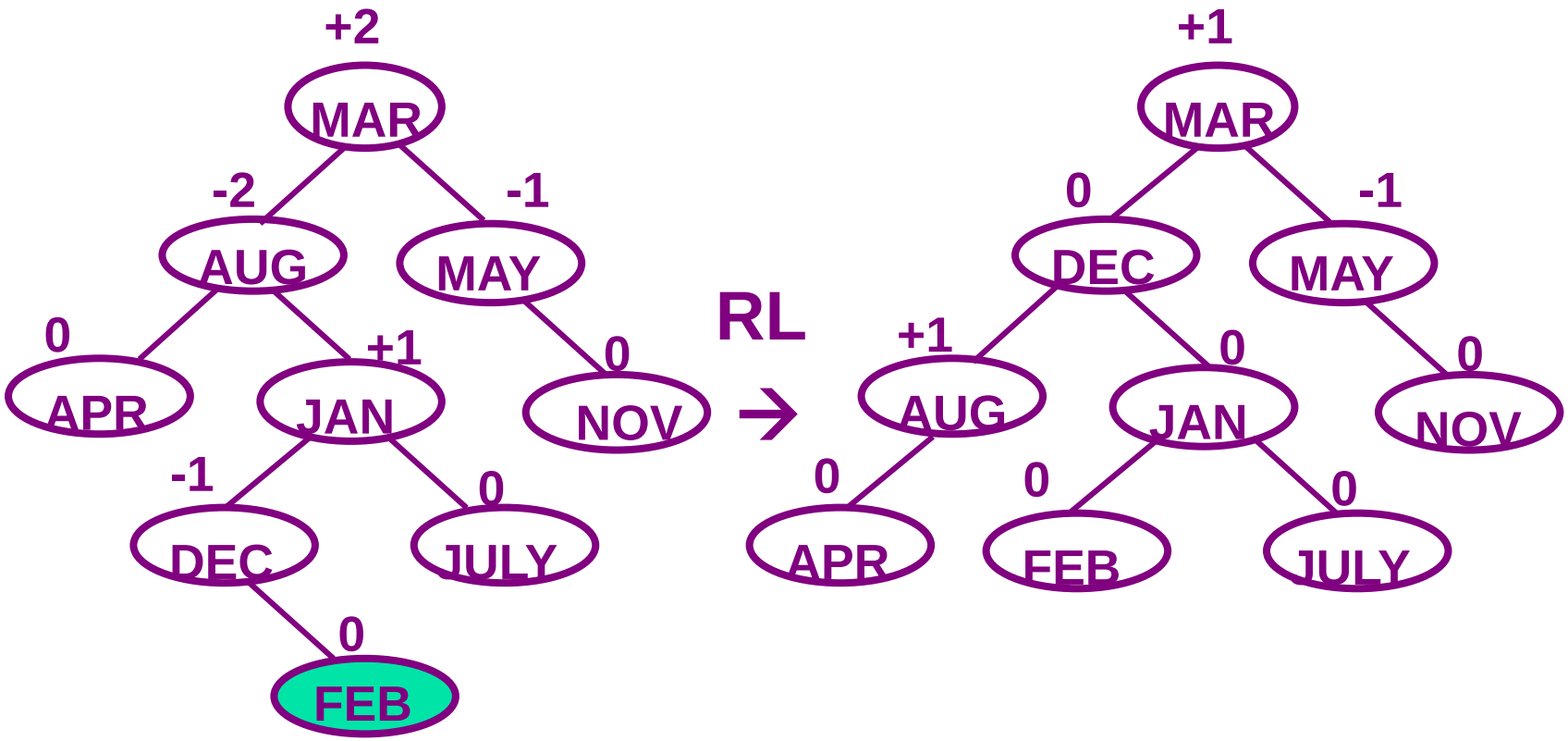
(f) Insert JAN



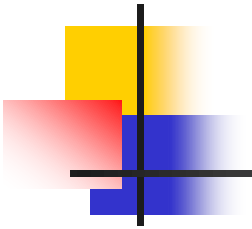
(g) Insert DEC



(h) Insert JULY


+2
MAR
-2
-1
AUG
MAY
0
+1
0
APR
JAN
NOV
-1
0
DEC
JULY
0
FEB
RL
+1
MAR
0
-1
DEC
MAY
+1
0
0
AUG
JAN
NOV
0
0
0
APR
FEB
JULY

(i) Insert FEB



In the proceeding example we saw that the addition of a node to a balanced binary search tree could unbalance it.

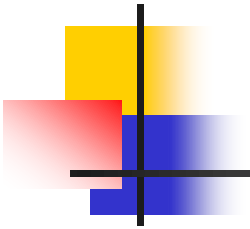
The rebalancing uses essentially 4 kinds of rotations:

LL, RR, LR, and RL.

LL and RR are symmetric, as are LR and RL.

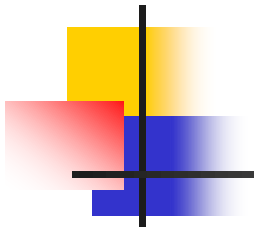
AVL树的类定义

```
#include <iostream.h>
#include “stack.h”
template <class E>
struct AVLNode : public BSTNode<E>
{
    //AVL树结点的类定义
   int bf;
    AVLNode() { left = NULL; right = NULL; bf = 0; }
  AVLNode (E d, AVLNode<E> *l = NULL,
                        AVLNode<E> *r = NULL)
       { data = d;  left = l;  right = r;  bf = 0; }
};
```



```
template <class E>
class AVLTree : public BST<E>
{
  //平衡的二叉搜索树（AVL）类定义
public:
    AVLTree() { root = NULL; }          //构造函数
    AVLTree (E Ref) { RefValue = Ref; root = NULL; }
      //构造函数：构造非空AVL树
```

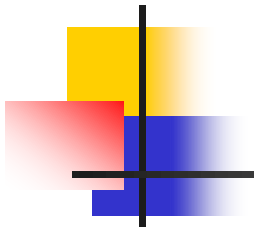
```
int Height() const;   //高度
AVLNode<E>* Search (E x,
                    AVLNode<E> *& par) const;  //搜索
bool Insert (E& e1) { return Insert (root, e1); } //插入
bool Remove (E x, E& e1)
    { return Remove (root, x, e1); }    //删除
friend istream& operator >> (istream& in,
    AVLTree<E>& Tree);        //重载：输入
friend ostream& operator << (ostream& out,
    const AVLTree<E>& Tree);  //重载：输出
```



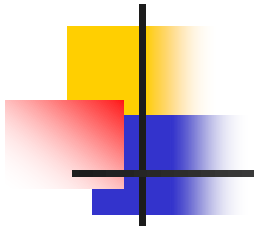
```
protected:
    int Height (AVLNode<E> *ptr) const;
    bool Insert (AVLNode<E>*& ptr, E& e1);
    bool Remove (AVLNode<E>*& ptr, E x, E& e1);
    void RotateL (AVLNode<E>*& ptr); //左单旋
    void RotateR (AVLNode<E>*& ptr); //右单旋
    void RotateLR (AVLNode<E>*& ptr);//先左后右双旋
    void RotateRL (AVLNode<E>*& ptr);//先右后左双旋
};
```

平衡化旋转

- 一棵平衡的二叉搜索树中，插入一个新结点
- AVL 树中相关结点的平衡状态会发生改变，造成不平衡
- 必须调整树的结构，使之平衡化
- 两类平衡化旋转：
  - ✓ 单旋转 (左旋和右旋)
  - ✓ 双旋转 (左平衡和右平衡)



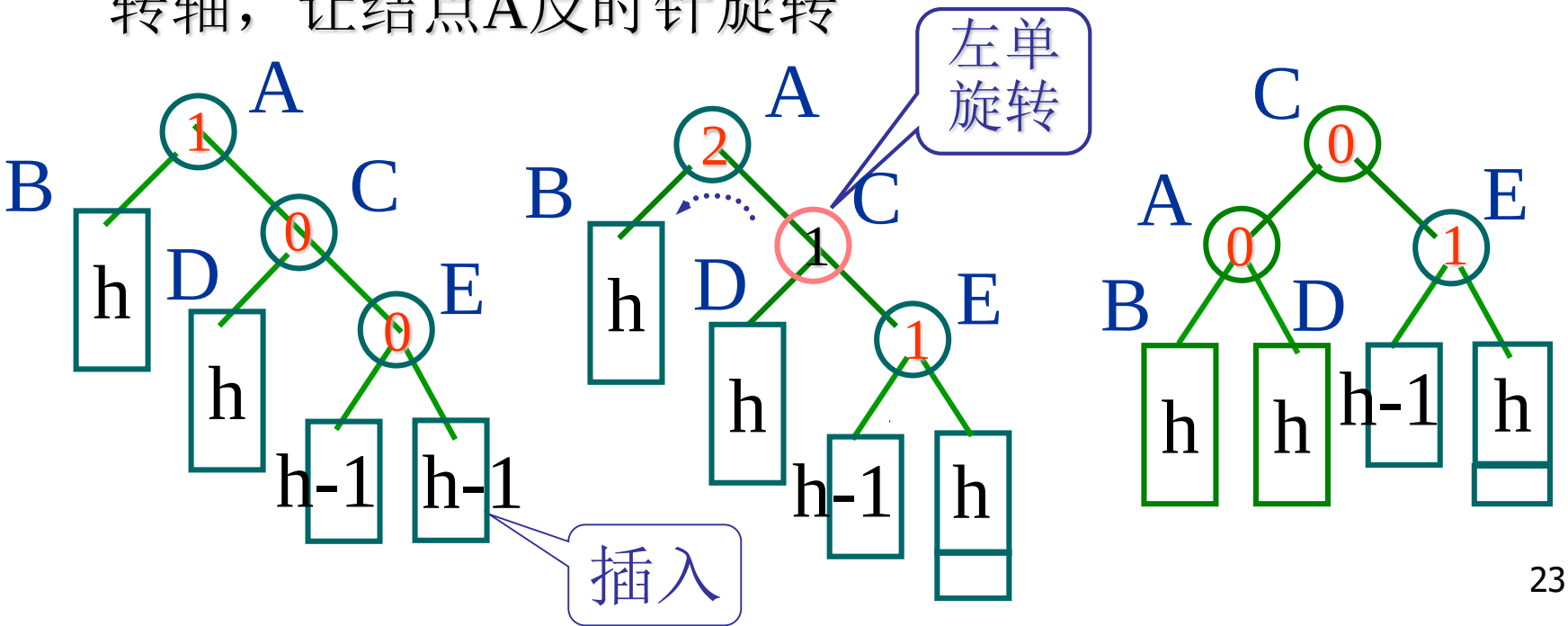
- 插入一 个新结点后
- 从插入位置沿通向根的路径回溯，检查各结点的平衡因子
- 如果在某一结点发现高度不平衡，停止回溯
- 从发生不平衡的结点开始，沿刚才回溯的路径取直接下两层的结点



- 如果这三个结点处于一条直线上，采用单旋转进行平衡化
  - 左单旋转，右单旋转
  - 一个是另一 个的镜像，方向与不平衡的形状相关
- 如果这三个结点处于一条折线上，采用双旋转进行平衡化
  - 先左后右，先右后左

左单旋转 (RotateLeft)

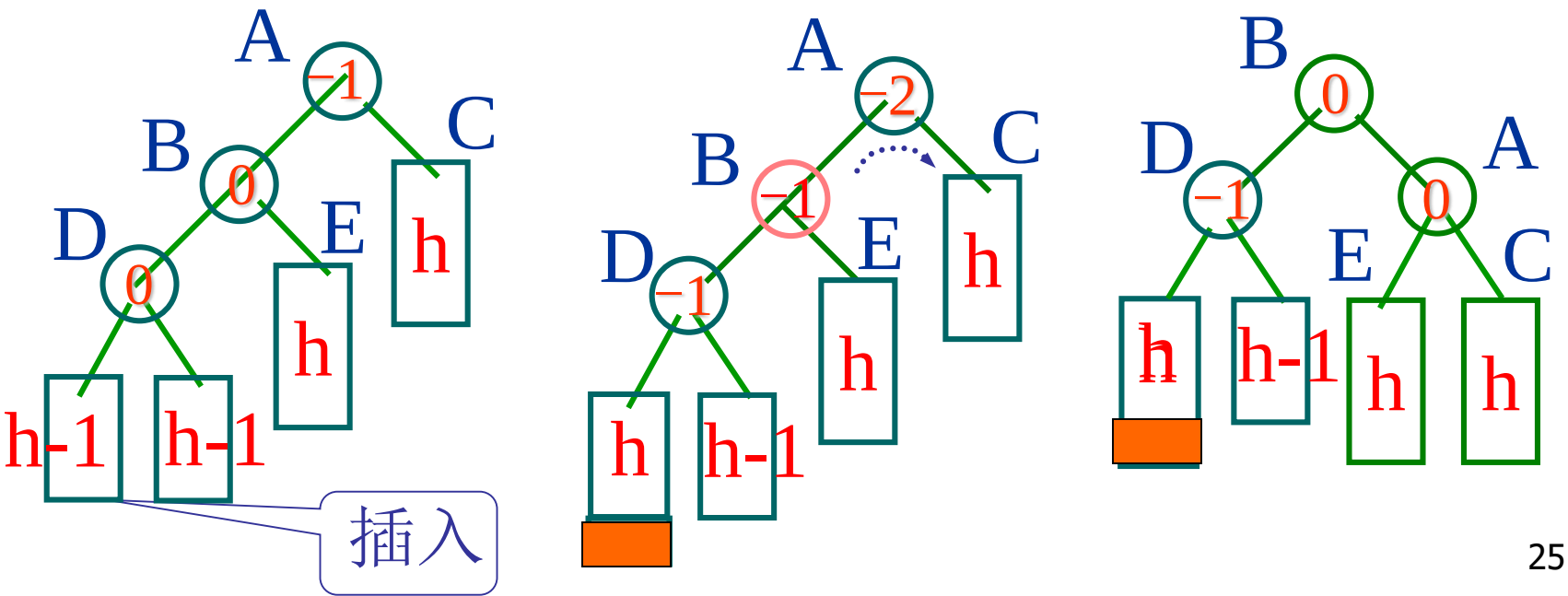
- 在右子树E中插入新结点, 出现不平衡
- 恢复平衡
  - 从A沿插入路径连续取3个结点A、C和E，以结点C为旋转轴，让结点A反时针旋转

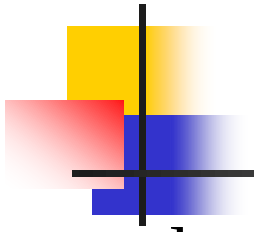


```
template <class E>
void AVLTree<E>::RotateL (AVLNode<E> *& ptr)
{  //右子树比左子树高: 做左单旋转后新根在ptr
   AVLNode<E> *subL = ptr;
                        ptr = subL->right;
                        subL->right = ptr->left;
                        ptr->left = subL;
                        ptr->bf = subL->bf = 0;
}
```

右单旋转 (RotateRight)

- 在左子树D上插入新结点，造成不平衡
- 恢复平衡
  - 从A沿插入路径连续取3 个结点A、B和D，以结点B为旋转轴，将结点A顺时针旋转



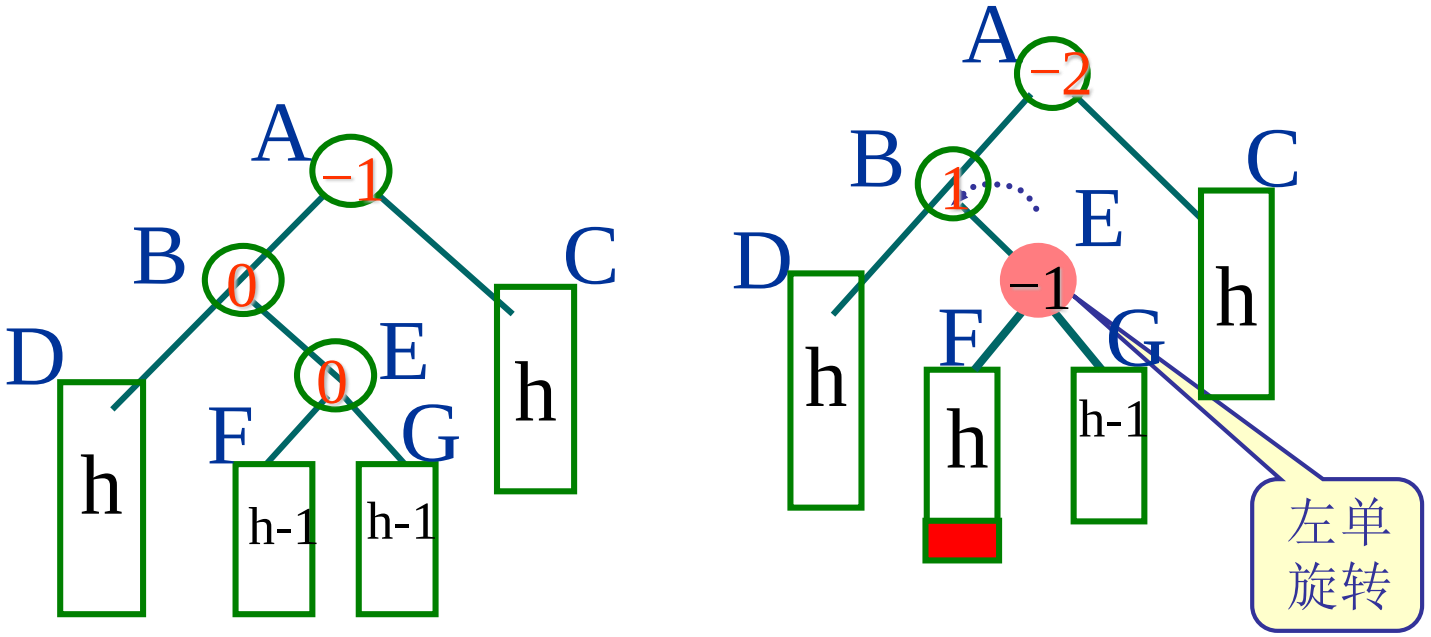


```
template <class E>
void AVLTree<E>::RotateR (AVLNode<E> *& ptr)
{ //左子树比右子树高,旋转后新根在ptr
    AVLNode<E> *subR = ptr;              //要右旋转的结点
                    ptr = subR->left;
                    subR->left = ptr->right; //转移ptr右边负载
                    ptr->right = subR;    //ptr成为新根
                    ptr->bf = subR->bf = 0;
}
```

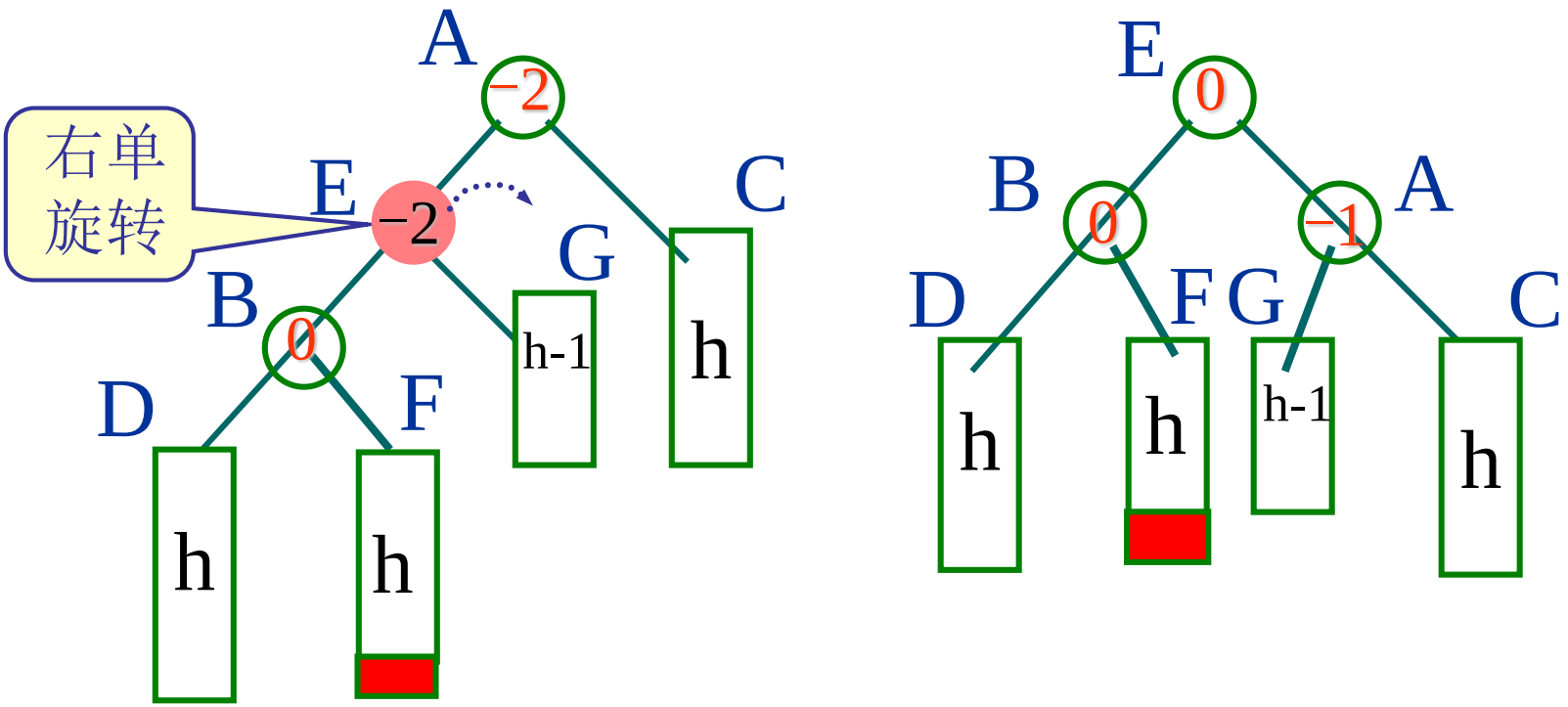
先左后右双旋转 (RotationLeftRight)

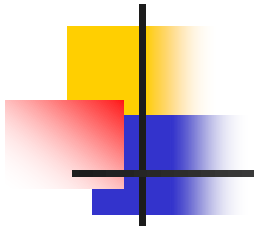
- 在结点A的左子女的右子树中插入新结点

- 以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置



- 再以结点E为旋转轴，将结点A顺时针旋转
- 使之平衡化

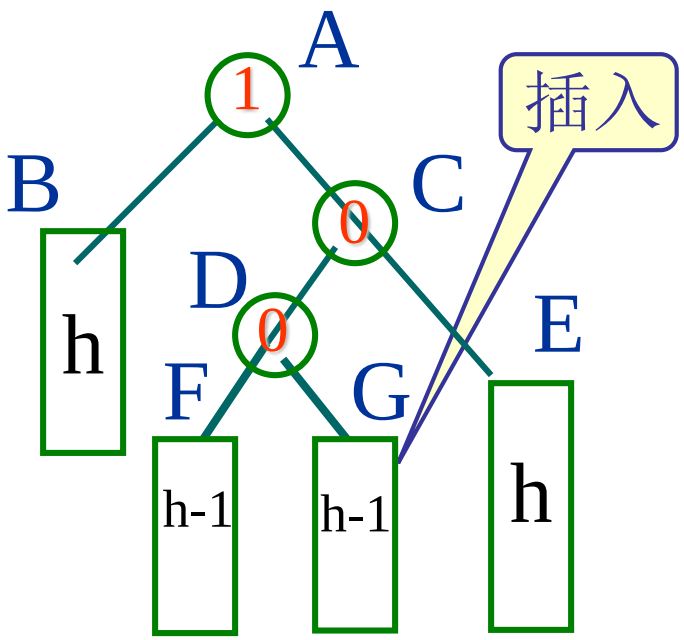


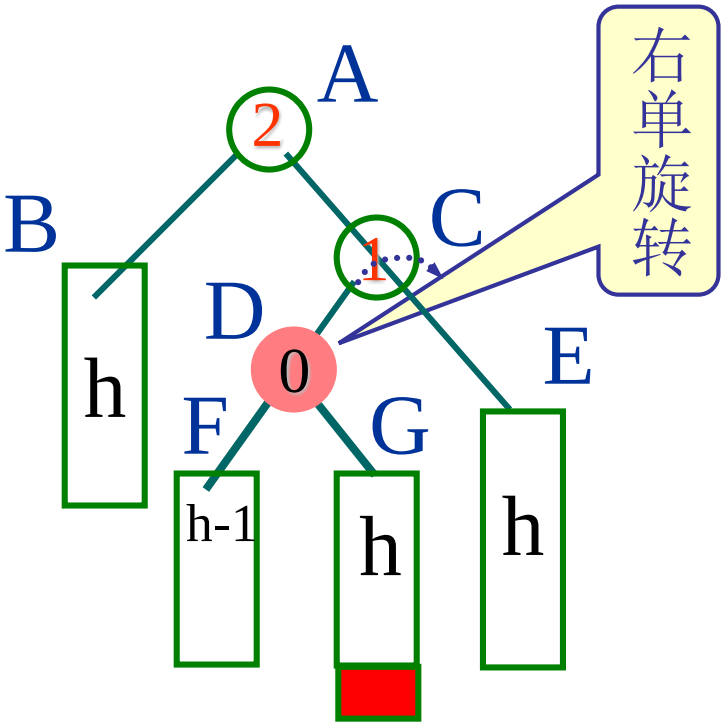


```
template <class E>
void AVLTree<E>::RotateLR (AVLNode<E> *& ptr)
{  AVLNode<E> *subR = ptr;
   AVLNode<E> *subL = subR->left;
                      ptr = subL->right;
                      subL->right = ptr->left;
                      ptr->left = subL;
  if (ptr->bf <= 0)
          subL->bf = 0;
  else  subL->bf = -1;
                     subR->left = ptr->right;
                     ptr->right = subR;
   if (ptr->bf == -1)
          subR->bf = 1;
   else  subR->bf = 0;
          ptr->bf = 0;
}
```

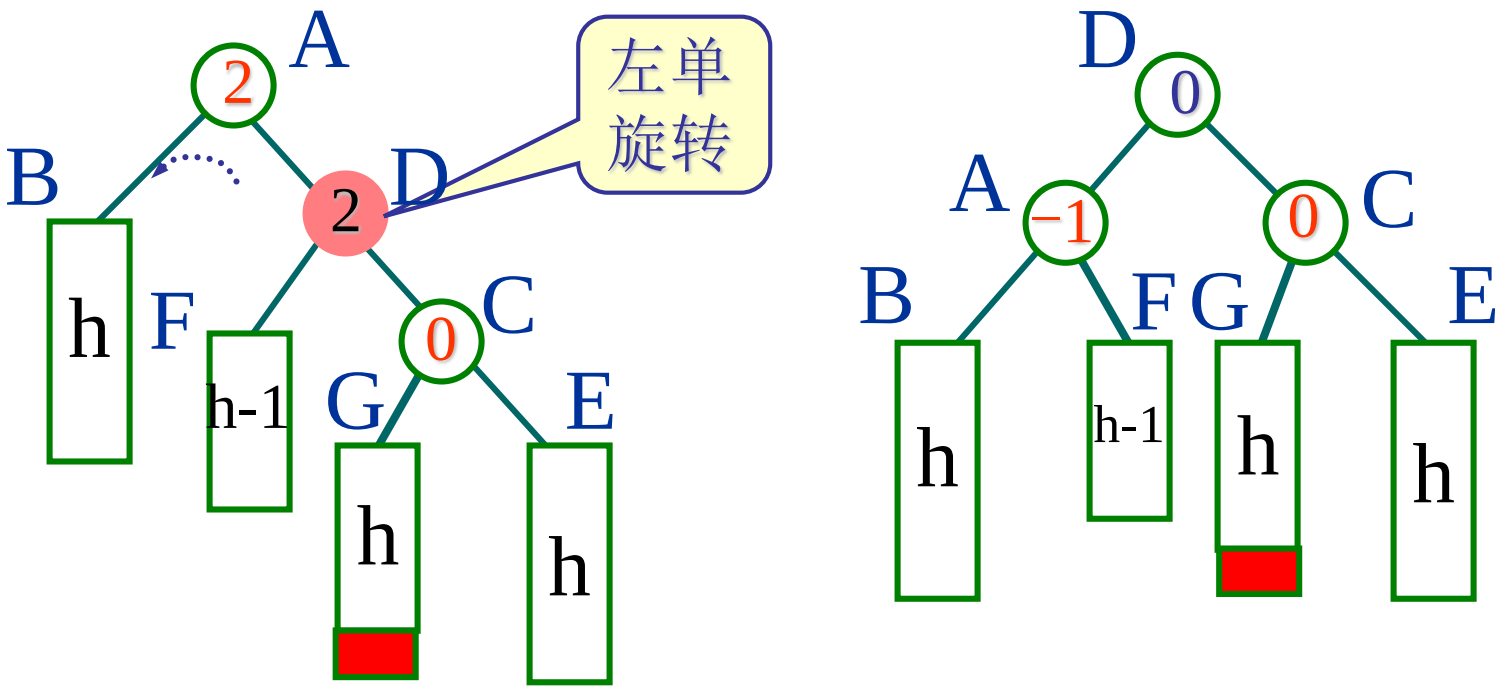
先右后左双旋转 (RotationRightLeft)

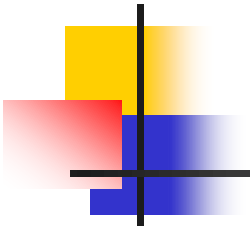
- 在结点A的右子女的左子树中插入新结点
- 以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置





- 再以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。





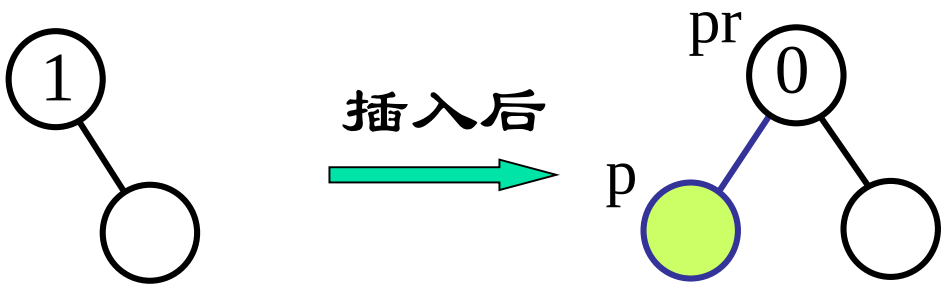
```
template <class E>
void AVLTree<E>::
        RotateRL (AVLNode<E> *& ptr)
{
   AVLNode<E> *subL = ptr;
   AVLNode<E> *subR = subL->right;
            ptr = subR->left;
            subR->left = ptr->right;
            ptr->right = subR;
   if (ptr->bf >= 0) subR->bf = 0;
   else subR->bf = 1;
           subL->right = ptr->left;
           ptr->left = subL;
   if (ptr->bf == 1) subL->bf = -1;
   else subL->bf = 0;
         ptr->bf = 0;
};
```

AVL树的插入算法

- 高度平衡的AVL树中插入一个新结点，
  - 如果树中某个结点的平衡因子的绝对值 |bf| > 1，则出现了不平衡，需要做平衡化处理。
- 算法思路：
  - 需从插入结点沿通向根的路径向上回溯
  - 如果发现有不平衡的结点，需从这个结点出发，使用平衡旋转方法进行平衡化处理

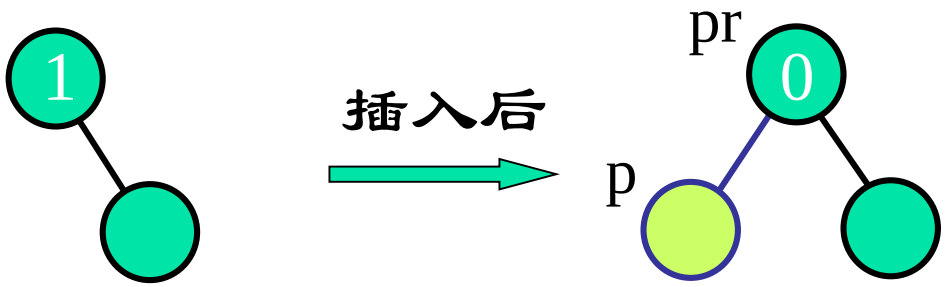
设新结点p的平衡因子为0，其父结点为pr。
插入新结点后，pr的平衡因子值有3种情况：

1. 结点pr的平衡因子bf=0
   - 是在pr的较矮的子树上插入了新结点，
   - 子树的高度不变，
   - 不需做平衡化处理，返回主程序。



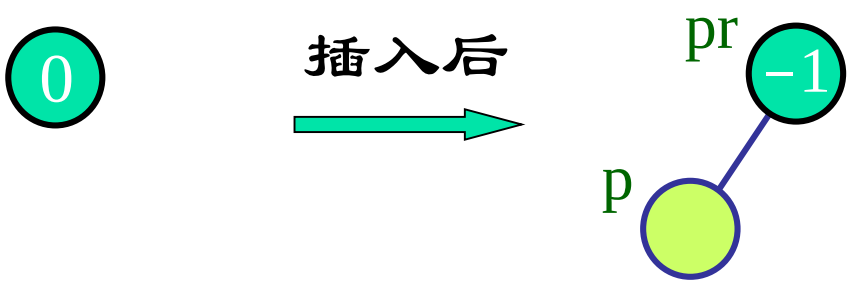
2. 结点pr的平衡因子的绝对值|bf| = 1

- 插入前pr的平衡因子是0，
- 插入新结点后，以pr为根的子树不需平衡化旋转。
- 但该子树高度增加。
- 需从结点pr向根方向回溯，
- 继续检查结点pr双亲(pr = Parent(pr))的平衡状态。



3. 结点pr的平衡因子的绝对值|bf| = 2

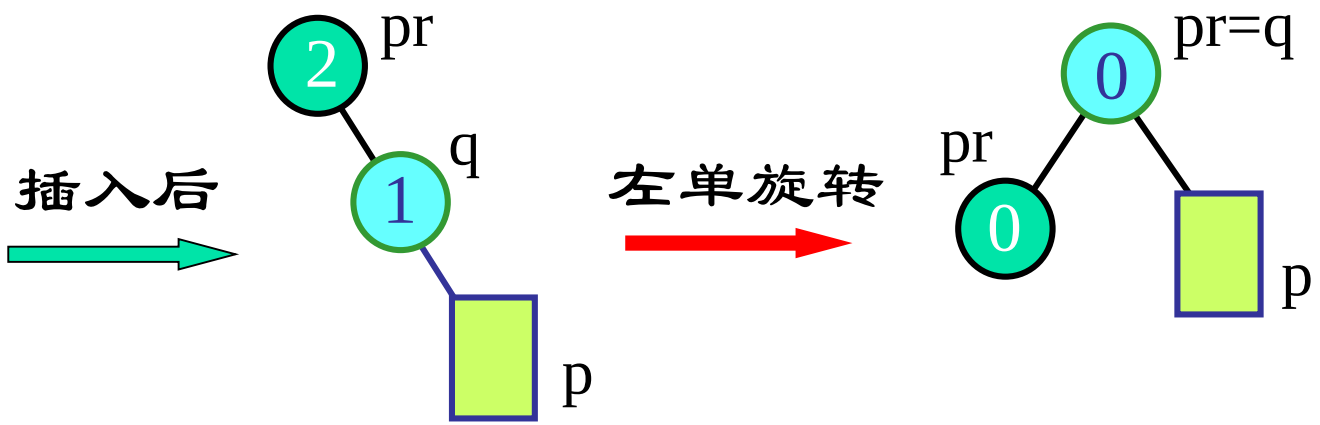
- 新结点在较高的子树上插入，
- 造成了不平衡，需要做平衡化旋转。



- 进一步分2种情况讨论：

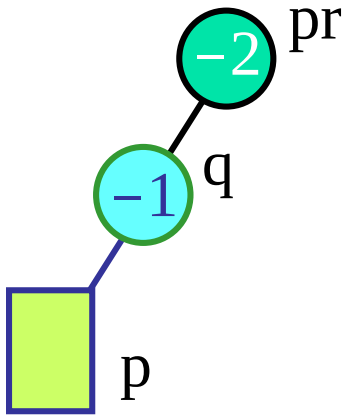
- 若结点pr的bf = 2
  - 右子树高，根据右子女q 的bf，分别处理：

– 若q的bf=1，执行*左单旋转*

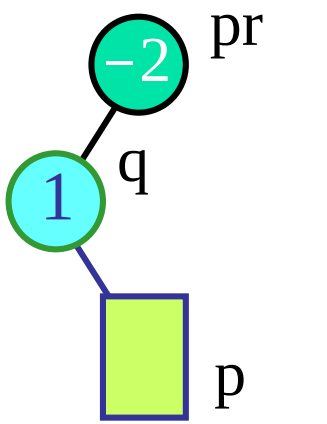


– 若q的bf=–1，执行*先右后左双旋转*

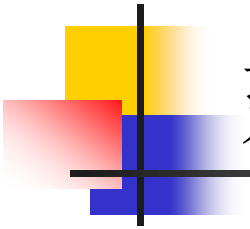
- 若结点pr的bf = –2
  - **左子树高**，根据左子女q 的bf，分别处理：
  - 若q的bf=–1，执行*右单旋转*
  - 若q的bf=1，执行*先左后右双旋转*



右单旋转



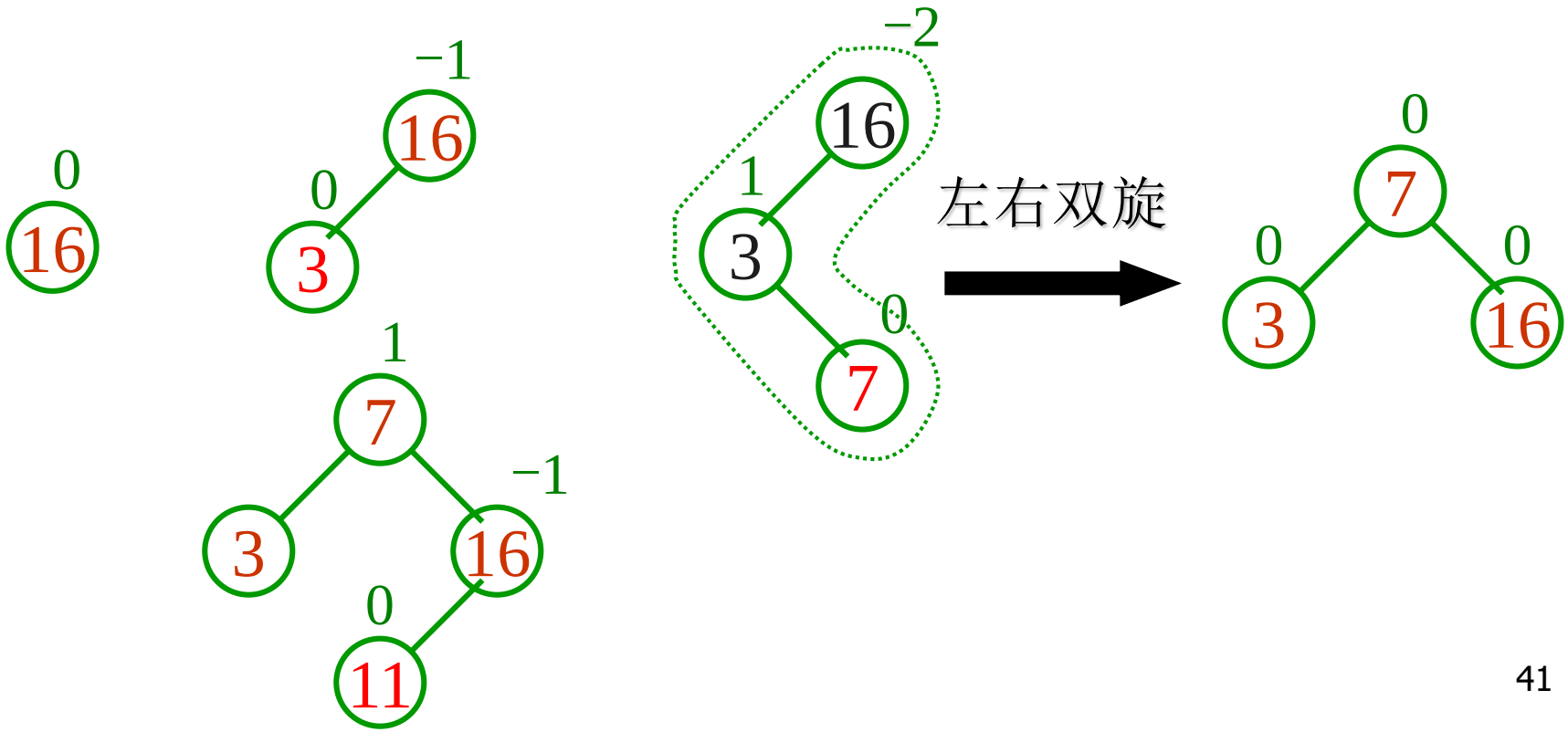
左右双旋转

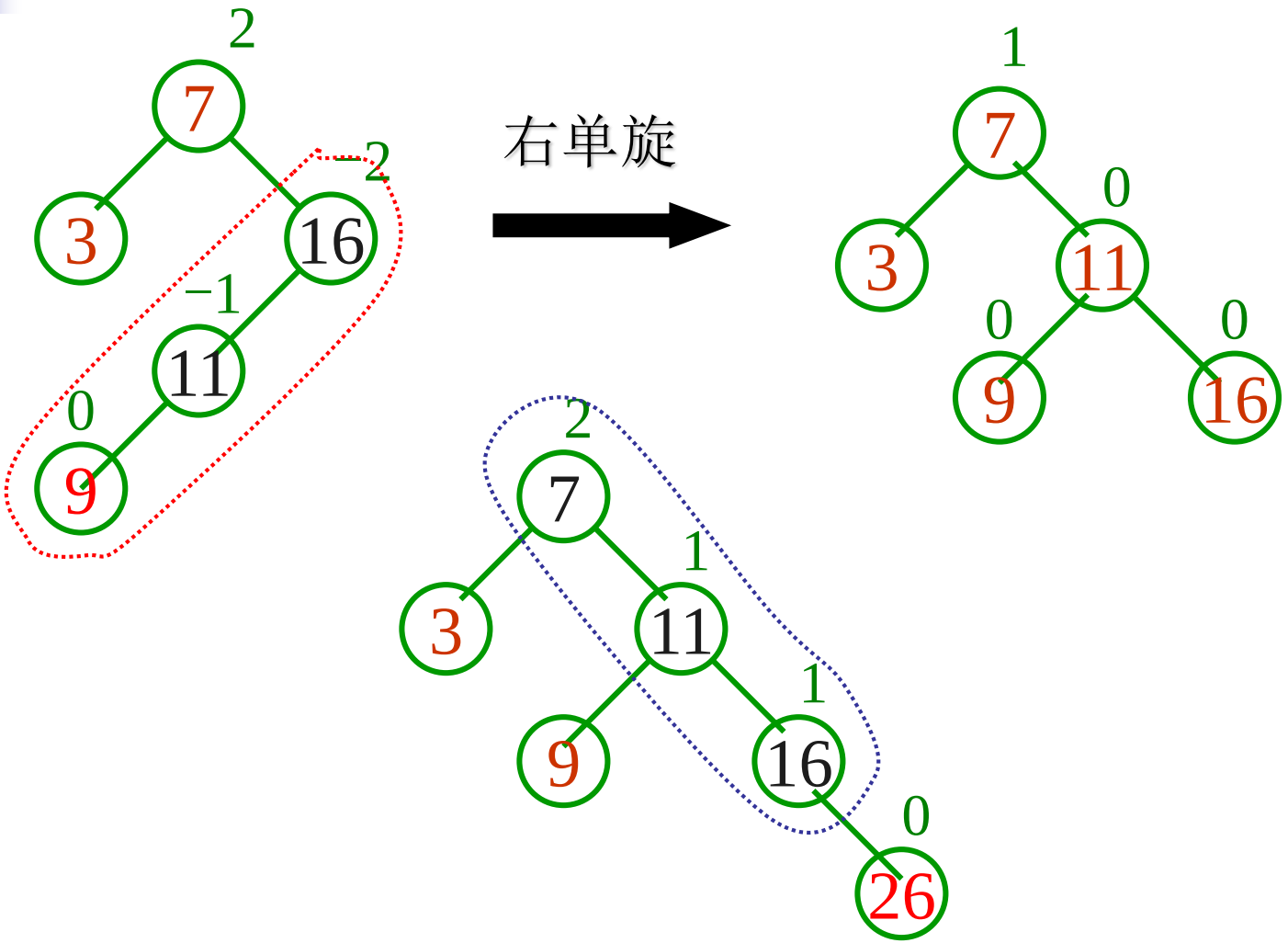


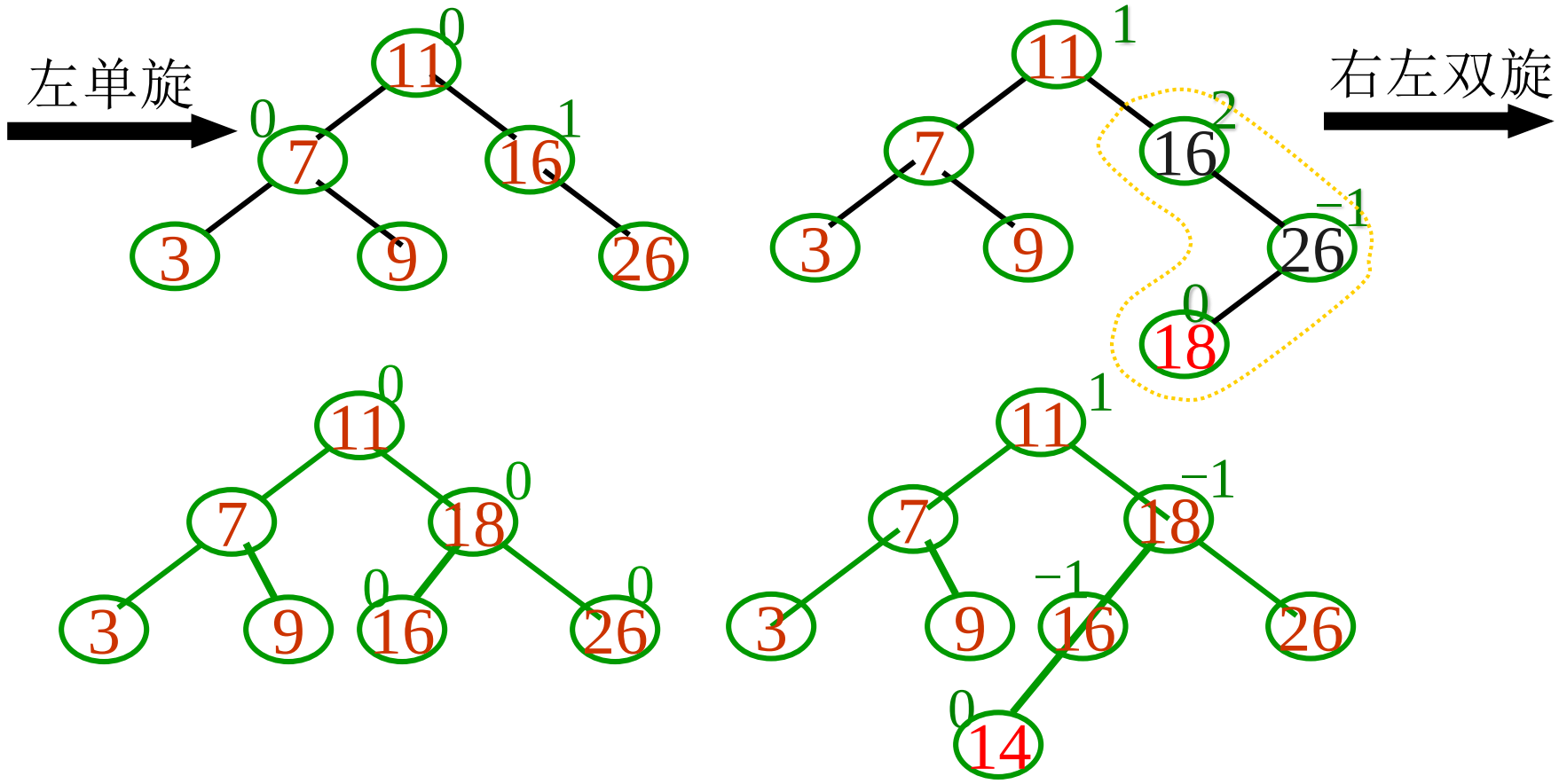
建立AVL树

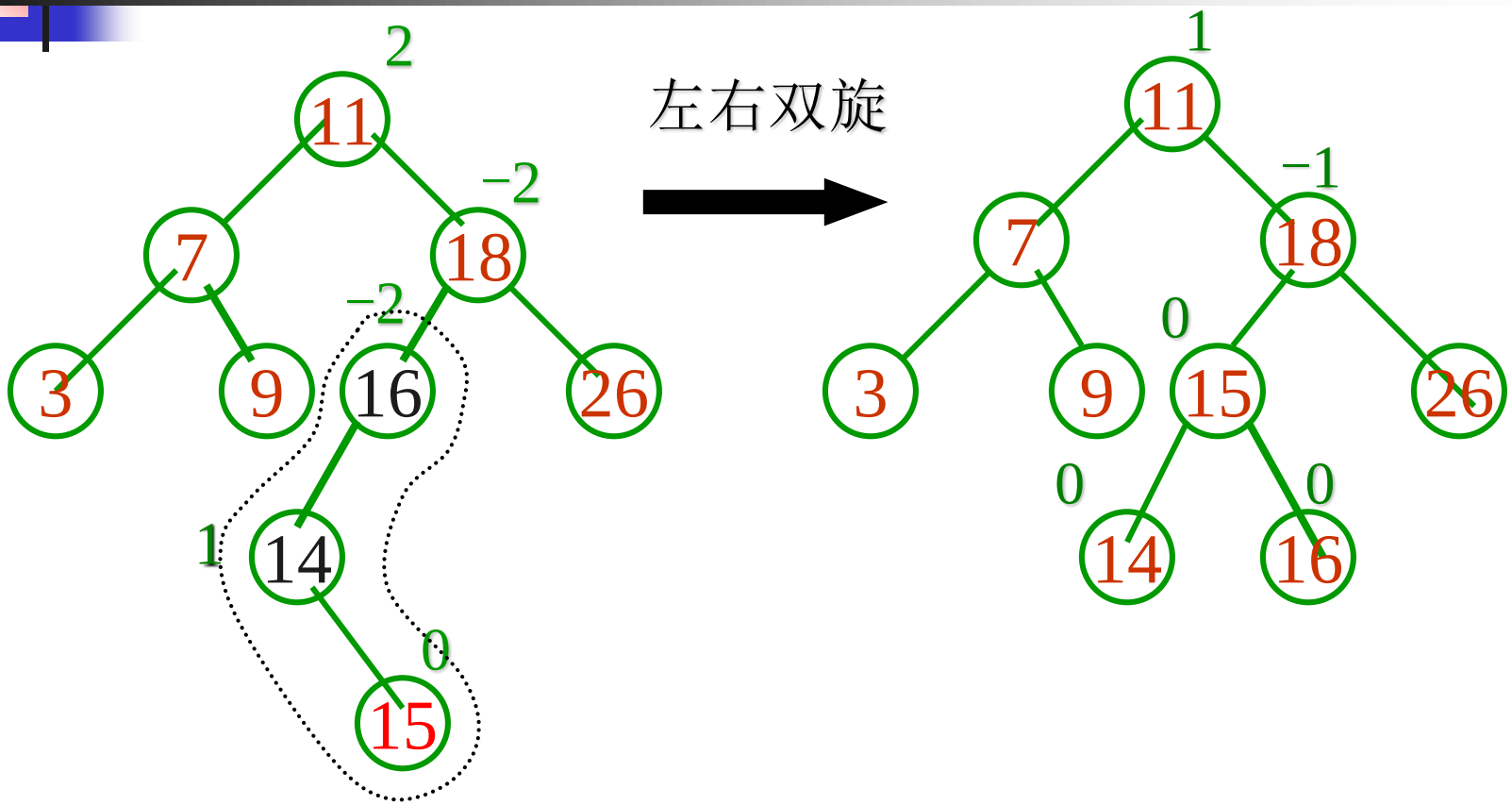
- 利用AVL树的插入算法
- 从一棵空树开始
- 通过输入一系列对象关键码
- 逐步构建

- 例如，输入关键码序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }
  插入和调整过程如下：

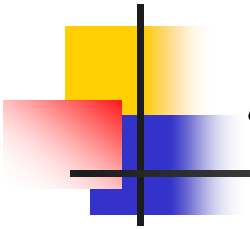



2
7
3
16
−2
−1
11
0
9
右单旋
1
7
3
0
11
0
0
9
16
2
7
3
1
11
9
1
16
0
26


左单旋
右左双旋
11
7
16
3
9
26
18
14

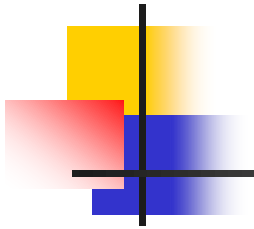


从空树开始建AVL树的过程



AVL树的删除

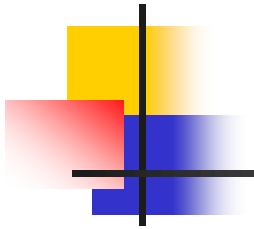
1. 如果结点*x*没有子女
   - *x*双亲原来指向*x*的指针置为NULL。
   - 将原来以结点*x*为根的子树的高度减1。
2. 如果被删结点*x*只有一个子女:
   - 把*x*的双亲中原来指向*x*的指针，改指到这个子女结点；
   - 将结点*x*从树中删去



3. 如果被删结点 x_有两个子女:

- 搜索 x 在中序次序下的直接前驱 y (也可以找直接后继),
- 把结点 y 的内容，传送给结点 x
  - 删除x的问题转移为删除结点 y
- 把结点 y 当作被删结点x
- 结点 y 最多有一个子女
  - 用 1. 2.方法进行删除

- 必须沿结点 x 通向根的路径反向追踪高度的变化对路径上各个结点的影响



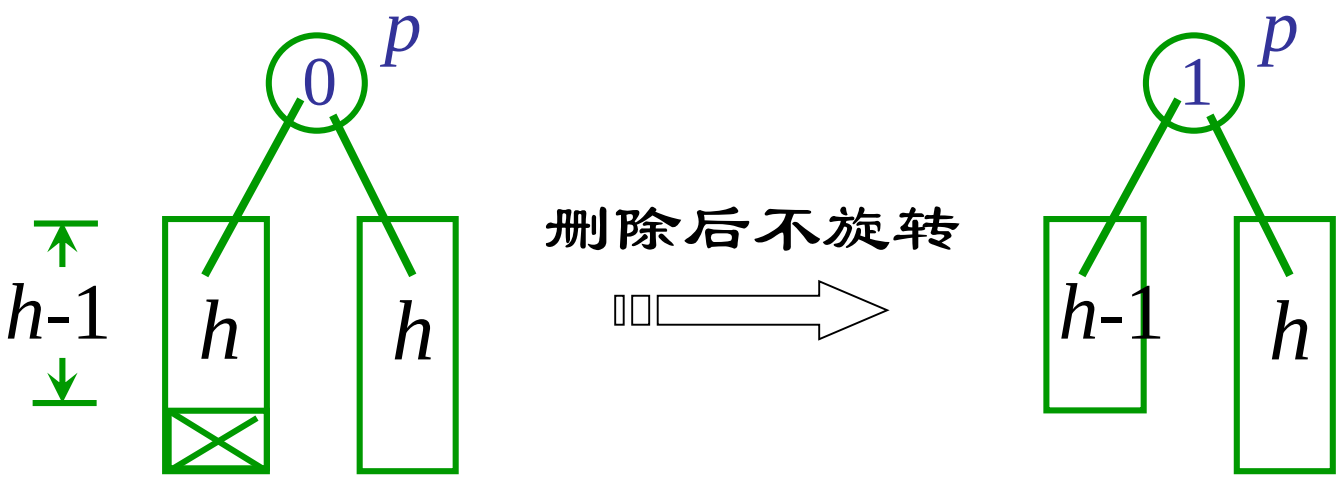
- 1 一个布尔变量shorter（缩短）
  - 表示：子树高度是否被缩短。
- 2 在每个结点上的操作，取决于：
  - shorter的值
  - 结点的bf, 还要依赖子女的bf
- 3 布尔变量shorter的值初始化为True
- 4 对于从*x*的双亲到根的路径上的各个结点*p*
  - 如果，shorter变成False，算法终止；
  - 在 shorter保持为True时，执行下面操作：

❶ 当前结点 p 的bf=0

如果它的左子树或右子树被缩短，

则它的bf改为1或−1，

同时 shorter置为False。

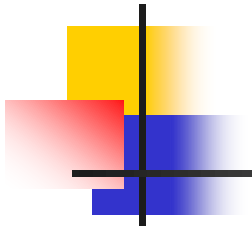


❷ 结点 p 的 bf <>0

且较高的子树被缩短，

则 p 的 bf =0，

同时，shorter置为True。



❸ 结点 *p* 的 bf <>0

且较矮的子树又被缩短，

则在结点 *p* 发生不平衡。

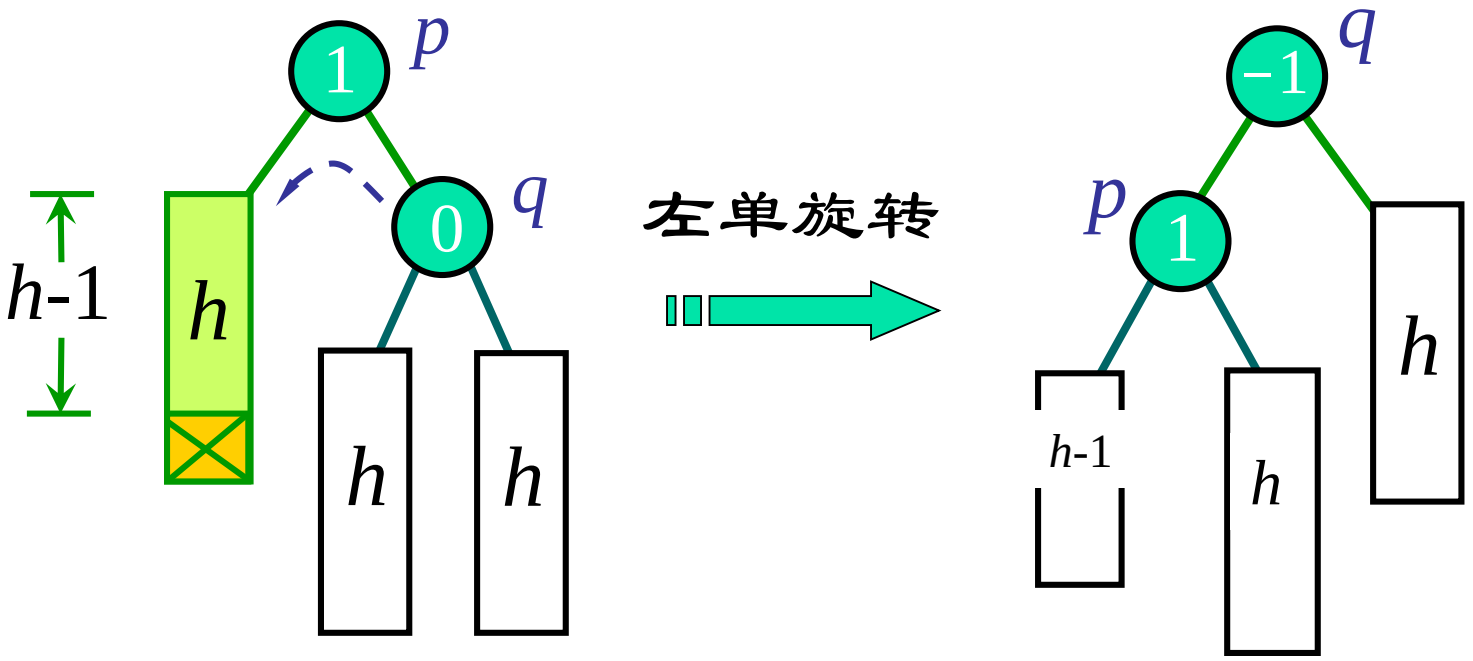
需要进行平衡化旋转来恢复平衡。

- 旋转的方向取决于：是结点 *p* 的哪一棵子树被缩短
- 令 *p* 的较高的子树的根为 *q*（该子树未被缩短）
- 根据 *q* 的 bf，有如下 3 种平衡化操作：

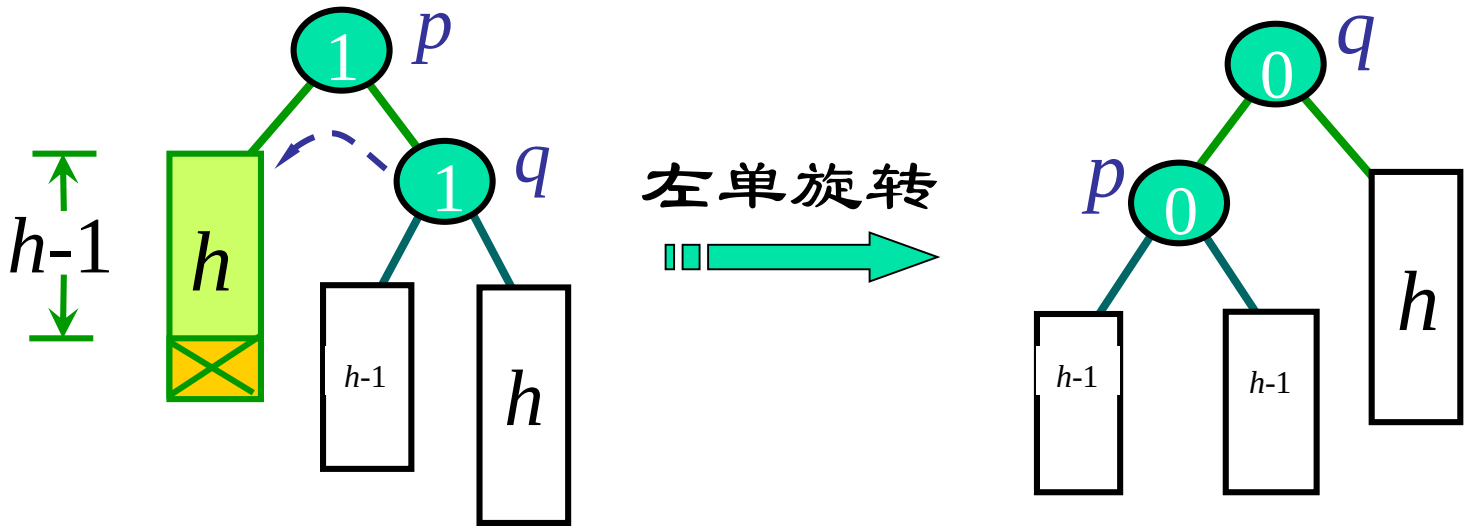
a) 如果 q（较高的子树）的 bf=0

执行一个单旋转来恢复结点 p 的平衡，

置shorter为False。



b) 如果 q 的 bf 与 p 的 bf 相同，
则执行一个单旋转来恢复平衡，
结点 p 和 q 的 bf 均改为0，
置shorter为True。

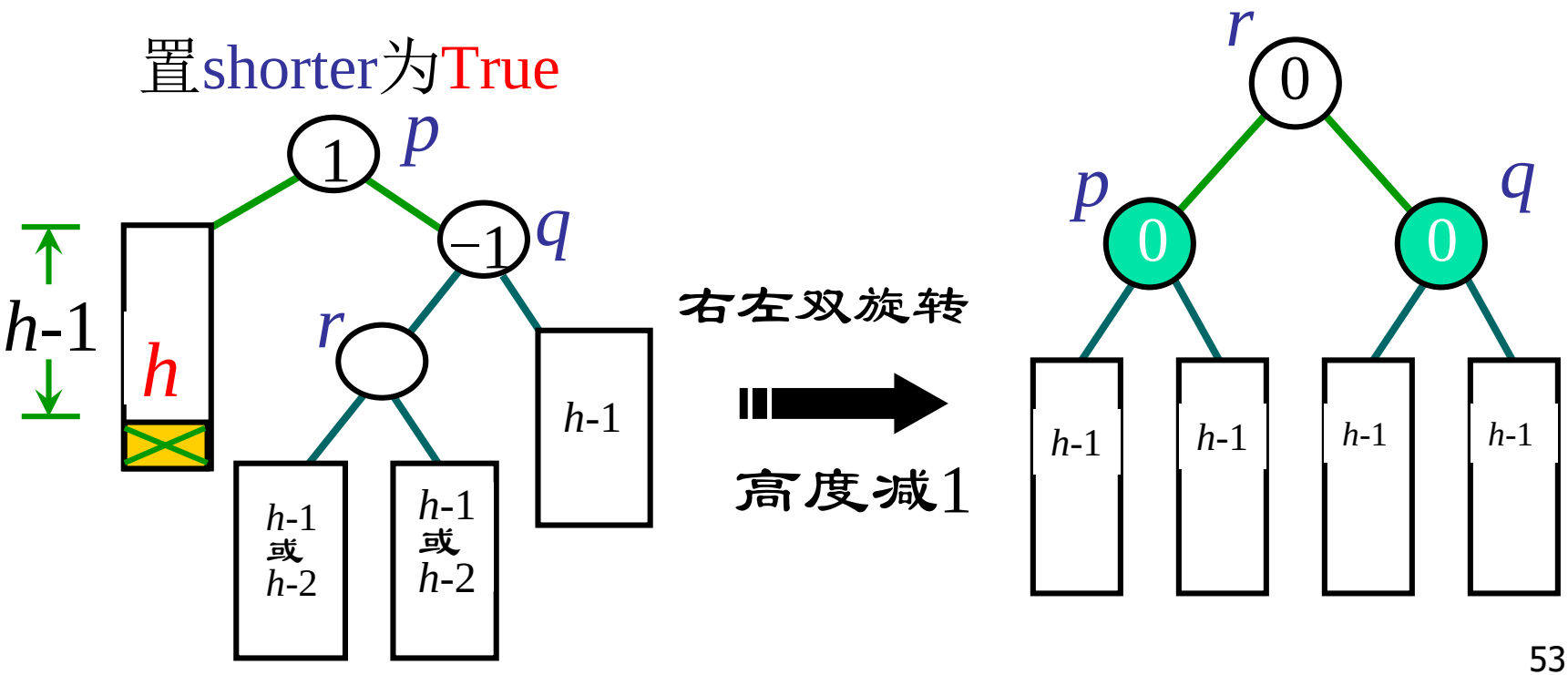


c) 如果 p 与 q 的 bf 相反

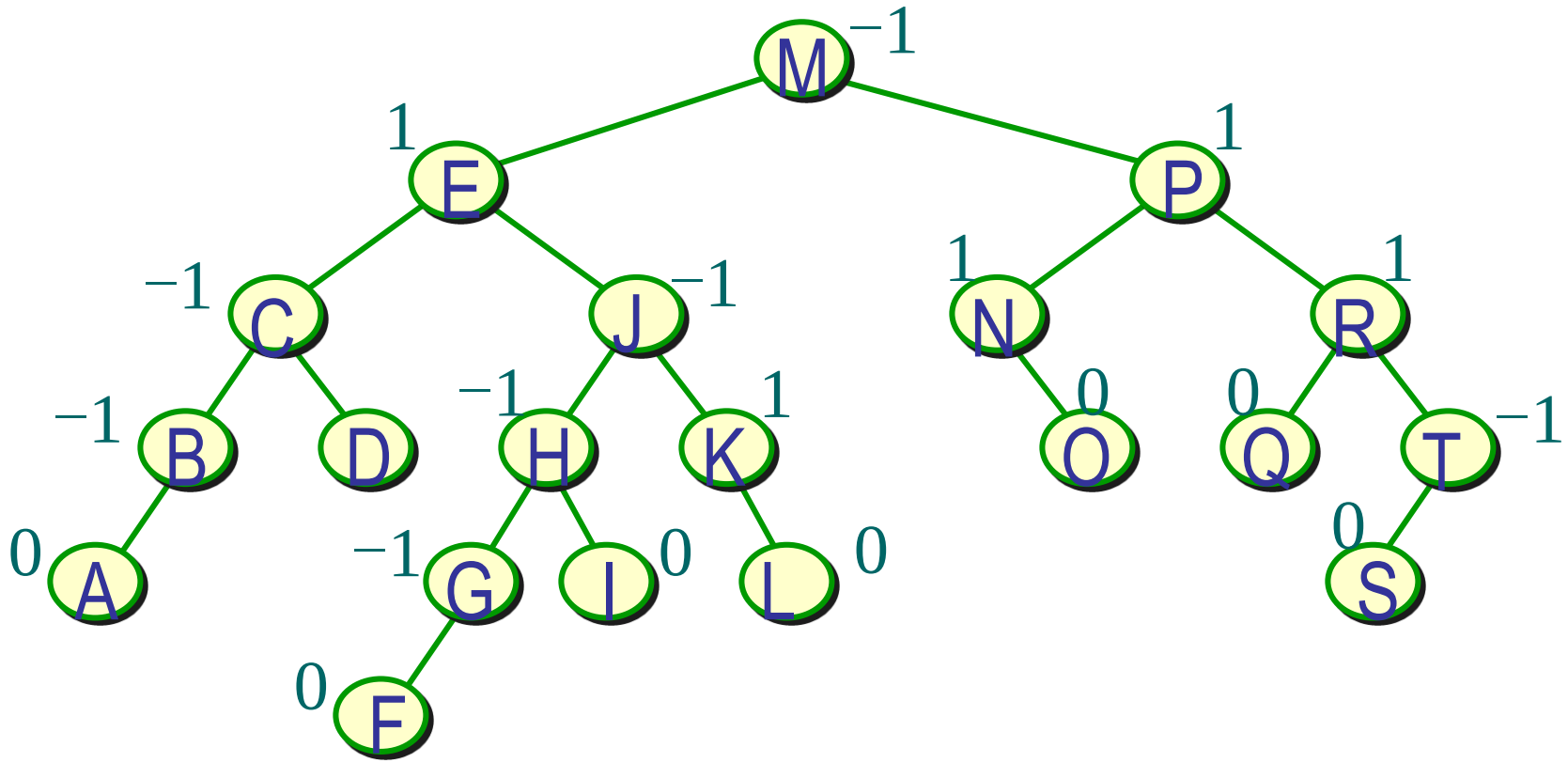
则执行一个双旋转来恢复平衡，先围绕 q 转再围绕 p 转

新根结点的 bf 置为0，其他结点的 bf 相应处理，

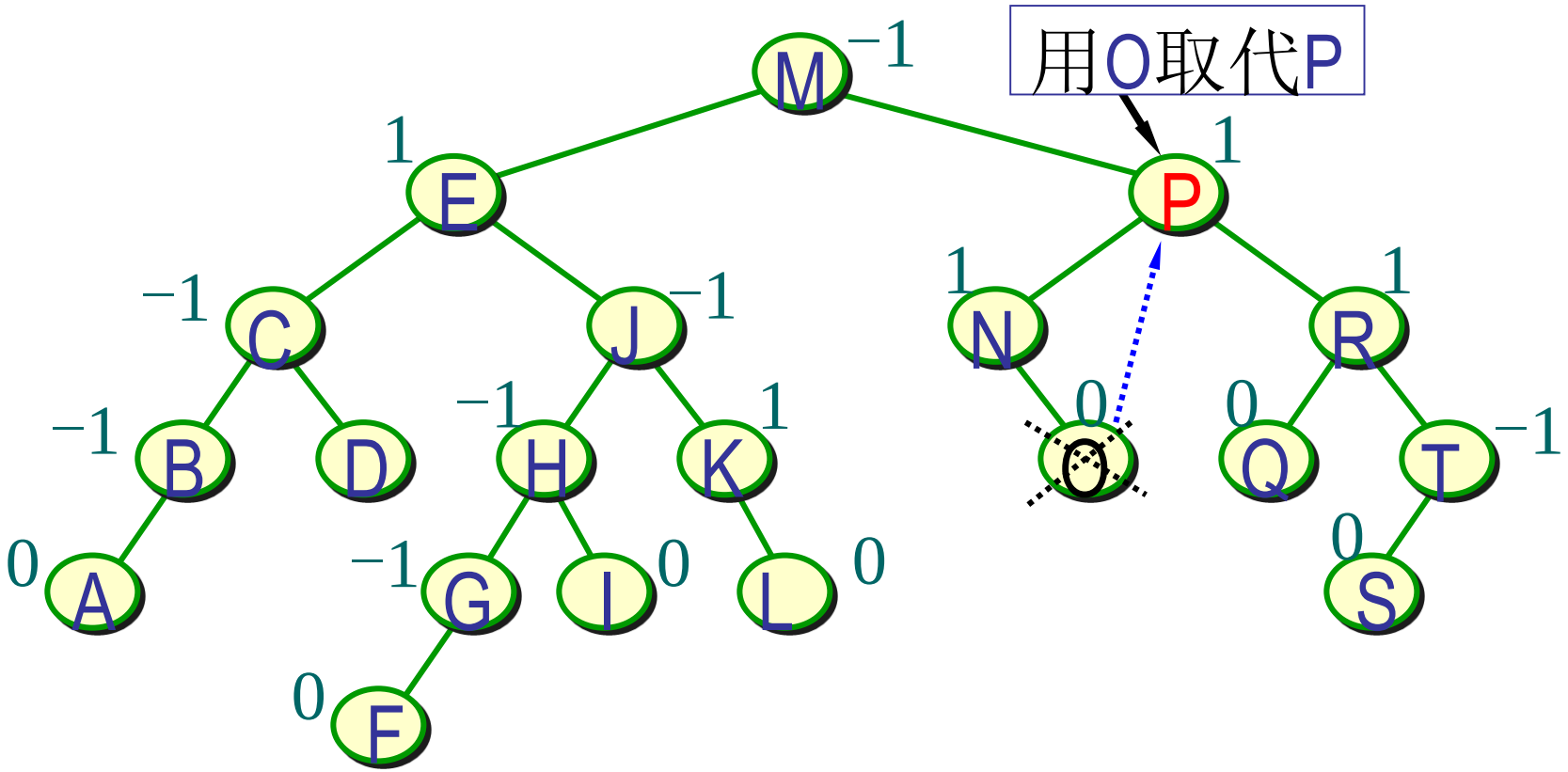
置shorter为True



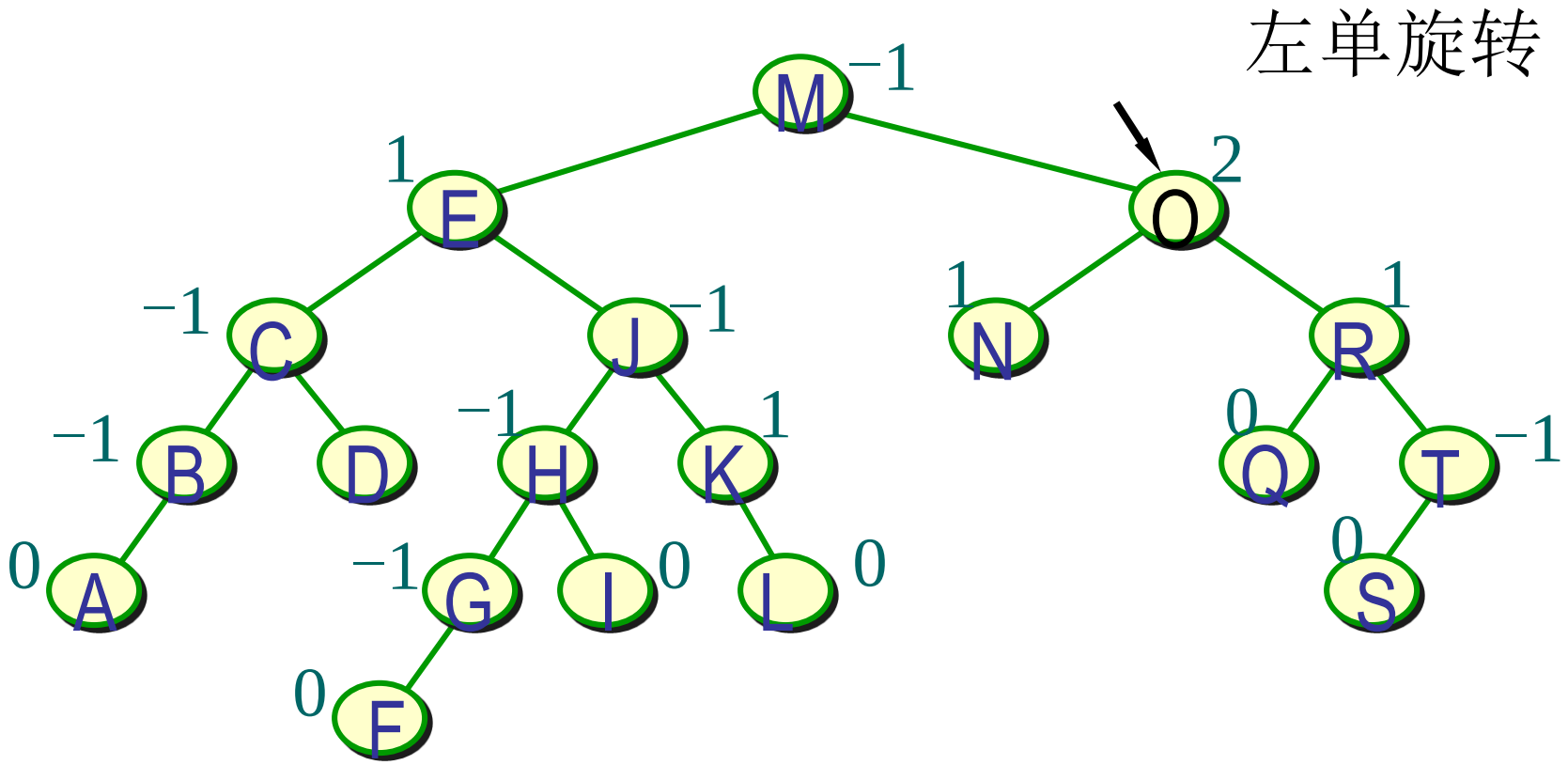
树的初始状态



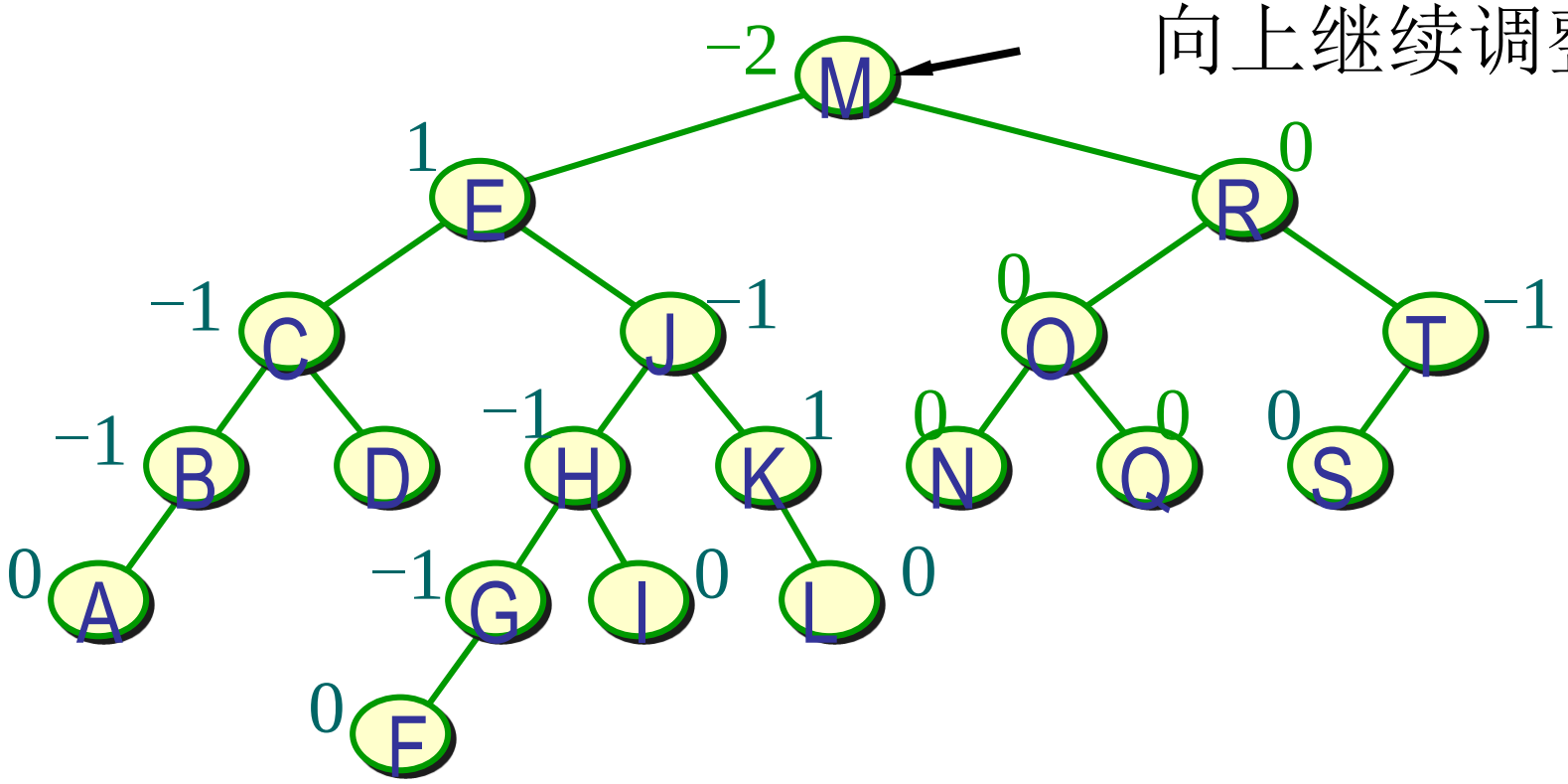
寻找结点P的中序直接前驱O, 用O顶替P, 删除O



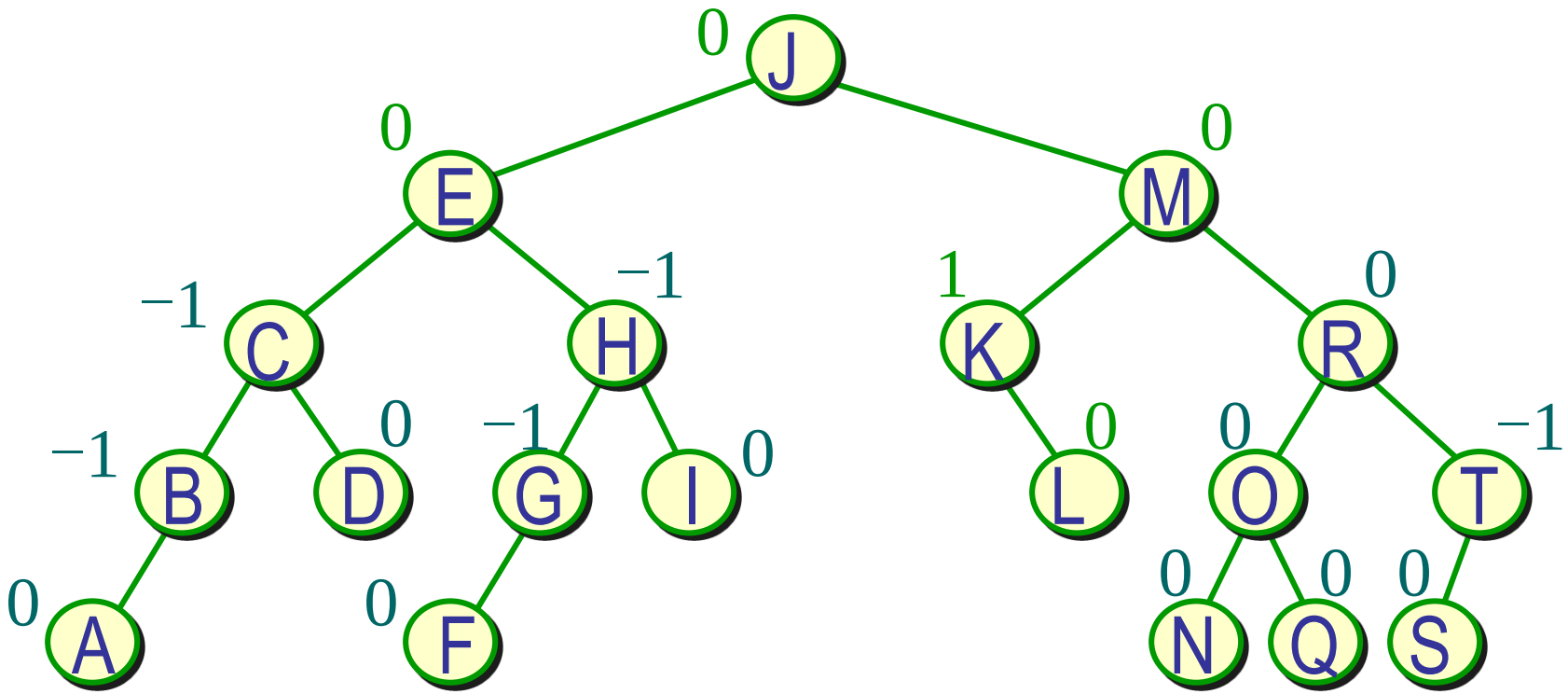
以R为旋转轴做左单旋转, M的子树高度减 1

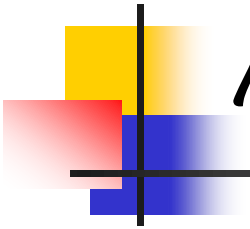


M的子树高度减1，M发生不平衡。M与E的平衡因子反号，做左右双旋转



平衡旋转的结果





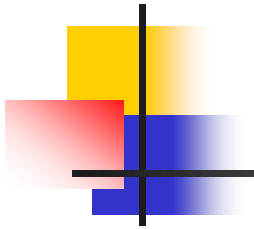
AVL树的高度

- 设在新结点插入前AVL树的高度为h，
  结点个数为n，
  则插入一个新结点的时间是O(h)

- 对于AVL树来说，h多大？

设N_h是高度为h的AVL树的最小结点数

- 根的一棵子树的高度为$h-1$，另一棵子树的高度为$h-2$，这两棵子树也是高度平衡的
- 因此，有：
  - $N_0 = 0$ (空树)
  - $N_1 = 1$ (仅有根结点)
  - $N_h = N_{h-1} + N_{h-2} + 1 , h > 1$



- 有 n 个结点的AVL树的高度不超过

  $1.44*\log_2(n+1)$

- 在AVL树删除一个结点并做平衡化旋转所需时间为 $O(\log_2 n)$