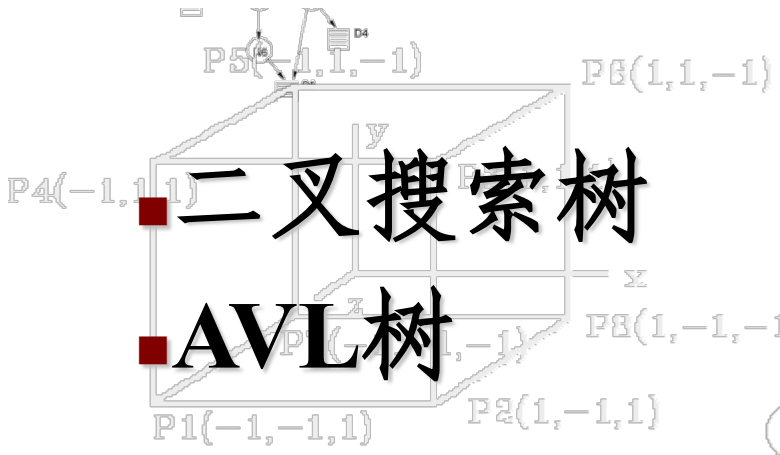
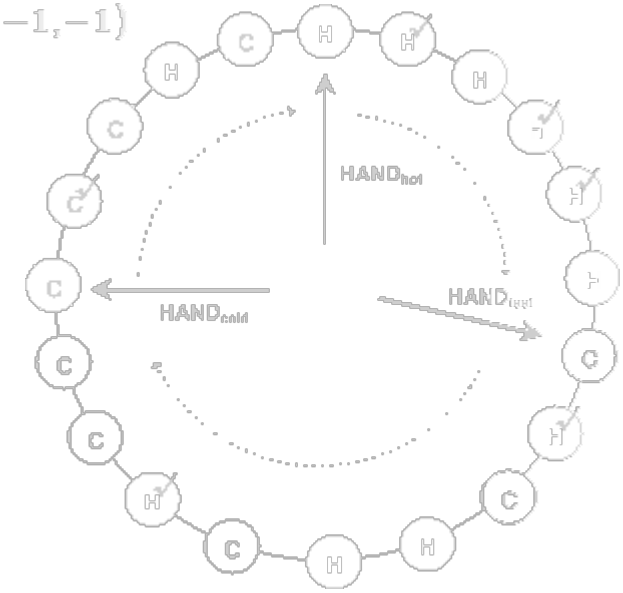
Chapter 1　两个搜索树



- 二叉搜索树
- AVL树



二叉搜索树 (Binary Search Tree)

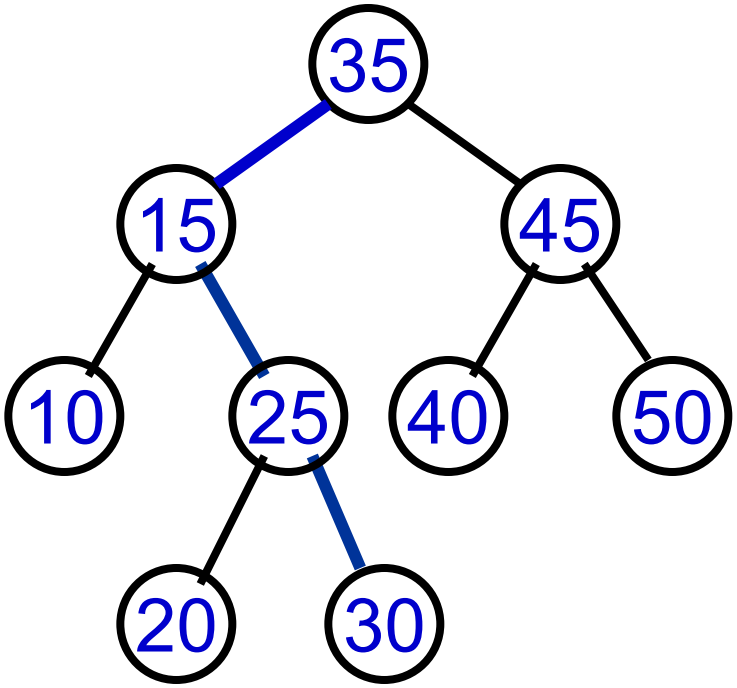
定义

二叉搜索树或者是一棵空树，或者是具有下列性质的二叉树:

- 每个结点都有一个作为搜索依据的关键码(key)，所有结点的关键码互不相同。
- 左子树(如果存在)上所有结点的关键码都小于根结点的关键码。
- 右子树(如果存在)上所有结点的关键码都大于根结点的关键码。
- 左子树和右子树也是二叉搜索树。

二叉搜索树例

- 结点左子树上所有关键码小于结点关键码。
- 右子树上所有关键码大于结点关键码。



如果对一棵二叉搜索树进行中序遍历？

二叉搜索树（二叉排序树）

有三种基本操作：

- 查找
- 插入
- 删除

基本思想：

- 假设在二叉搜索树中搜索关键码为x的元素
  - 如果根指针为NULL，则搜索不成功；
  - 如果x与根结点的关键码进行比较
    - 等于：搜索成功，对应结点就是根节点
    - 小于：二叉搜索树中存在关键码为x的结点，当且仅当其左子树中存在关键码为x的结点
    - 大于：二叉搜索树中存在关键码为x的结点，当且仅当其右子树中存在关键码为x的结点

```
BstNode * Find ( Type x, BstNode * ptr )
{
    //二叉搜索树的递归的搜索算法
    if ( ptr == NULL ) return NULL;  //搜索失败
    else if ( x < ptr->data )          //在左子树搜索
            return Find ( x, ptr->leftChild );
    else if ( x > ptr->data )     //在右子树搜索
            return Find ( x, ptr->rightChild );
    else return ptr;                   //相等,搜索成功
}
```

这个搜索过程是尾递归

```
BstNode * Find ( Type x, BstNode * ptr ) const
{
   //二叉搜索树的迭代的搜索算法
   if ( ptr != NULL ) {
      BstNode * temp = ptr;   //从根搜索
      while ( temp != NULL ) {
            if ( temp->data == x ) return temp;
            if ( temp->data < x )
             temp = temp->rightChild;   //右子树
          else temp = temp->leftChild;  //左子树
      }
   }
   return NULL;              //搜索失败
}
```

二叉搜索树上的搜索（迭代）

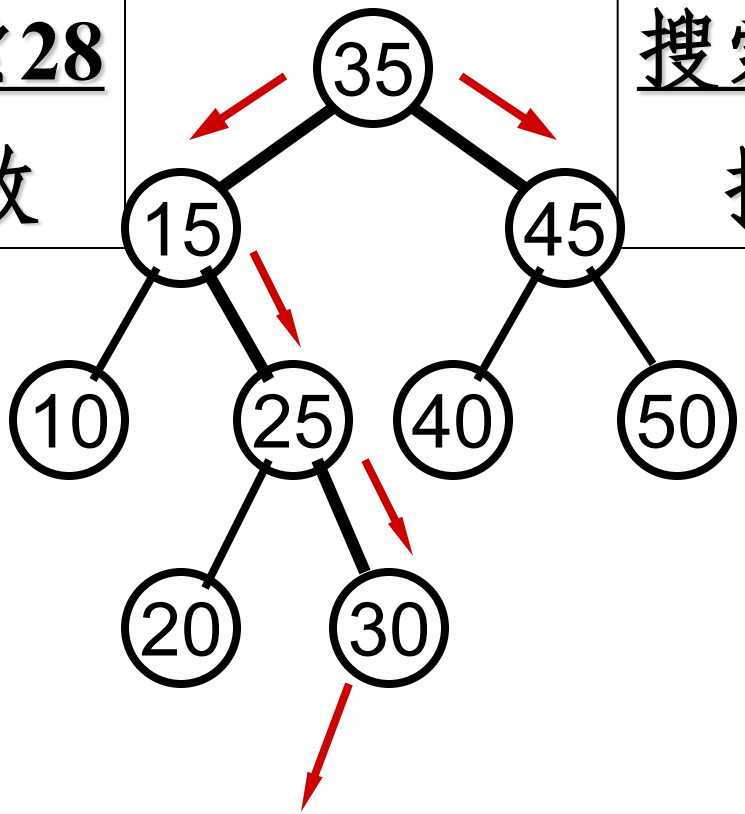
是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。它可以是一个递归的过程。

搜索28

搜索失败

搜索45

搜索成功



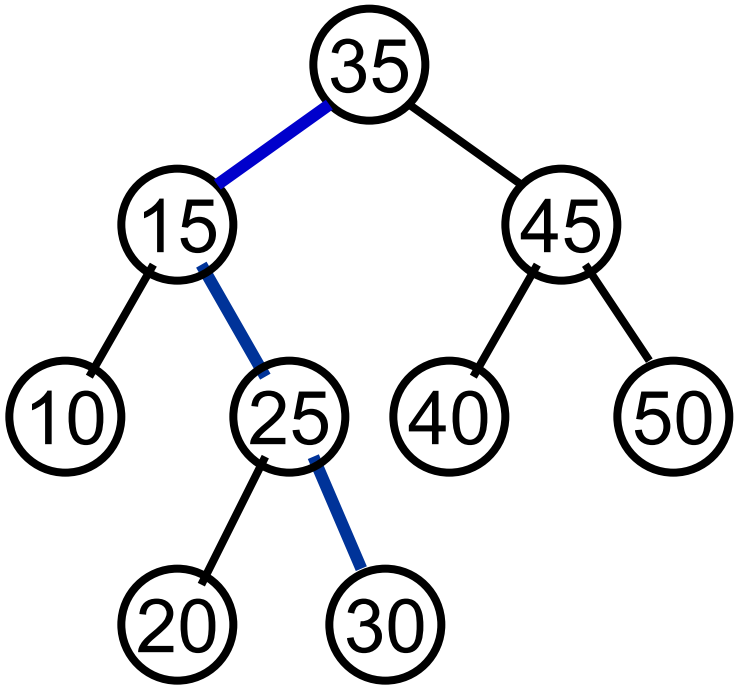
二叉搜索树的递归思路

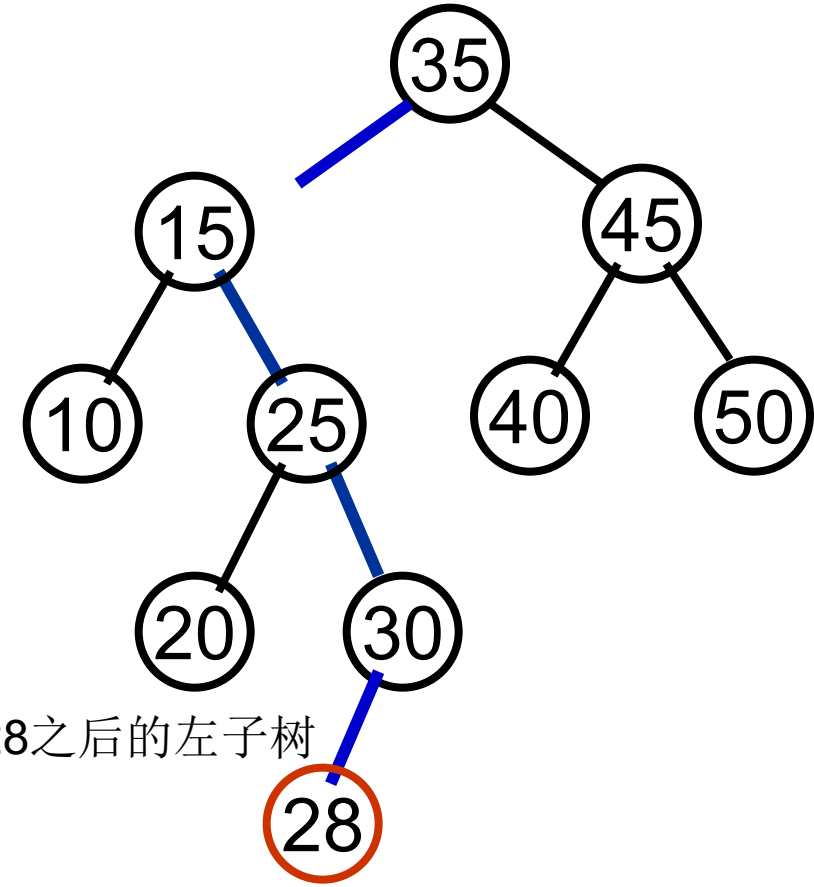
在根结点为root的二叉搜索树中插入一个关键码x

- 如果root等于null，插入后的二叉树是新创建的，关键码为x的单个结点；

- 如果x等于根结点的关键码，不需要插入；
- 如果x小于根结点的关键码，在向左子树中插入，并把插入后得到的结点作为根结点的新二叉树；
- 如果x大于根结点的关键码，在向右子树中插入，并把插入后得到的结点作为根结点的新二叉树；

后三种情况，新树的根结点仍然是原来的根结点





插入28之后的左子树

```
BstNode * Insert ( Type x, BstNode * ptr )
{
    if ( ptr == NULL )
            return new BstNode(x);  //新结点
    if ( x < ptr->data )
        ptr->leftChild = Insert( x, ptr->leftChild );
    else if ( x > ptr->data )
        ptr->rightChild = Insert( x, ptr->rightChild );

    return ptr;
}
```

尾递归的二叉搜索树插入算法

```
Insert ( Type x, BstNode * & ptr) {
    if ( ptr == NULL )
        ptr = new BstNode (x); //创建结点
    else if ( x < ptr->data )          //在左子树插入
        Insert ( x, ptr->leftChild );
    else if ( x > ptr->data )   //在右子树插入
        Insert ( x, ptr->rightChild );
}
```

对于引用参数，实际传递的是ptr的地址，而对ptr的赋值、使用都是根据这个地址解引用而得到的

```
//ptr是根结点指针的地址
Insert ( Type x, BstNode ** ptr) {
    if ( *ptr == NULL )
        *ptr = new BstNode (x); //创建结点
    else if ( x < (*ptr)->data )     //在左子树插入
        Insert ( x, &(*ptr)->leftChild );
    else if ( x > (*ptr)->data )   //在右子树插入
        Insert ( x, &(*ptr)->rightChild );
}
```

```
Insert ( Type x, BstNode ** ptr)
{
      while(true)
      {
            if ( *ptr == NULL )
            {           *ptr = new BstNode (x);
                        return;
            }
            else if ( x < (*ptr)->data )
            {           ptr = &(*ptr)->leftChild;
                        continue;
                        //Insert ( x, &(*ptr)->leftChild );
            }
            else if ( x > (*ptr)->data )
            {           ptr = &(*ptr)->rightChild;
                        continue;
                        //Insert ( x, &(*ptr)->rightChild );
            }
            else
                        return;
      }
}
```

二叉搜索树的插入（迭代）

基本思想:

为了向二叉搜索树中插入一个新元素，必须先检查这个元素是否在树中已经存在。

- 在插入之前，先使用搜索算法在树中检查要插入元素有还是没有。
  - 搜索成功：树中已有这个元素，不再插入。
  - 搜索不成功：树中原来没有关键码等于给定值的结点，把新元素加到搜索操作停止的地方。

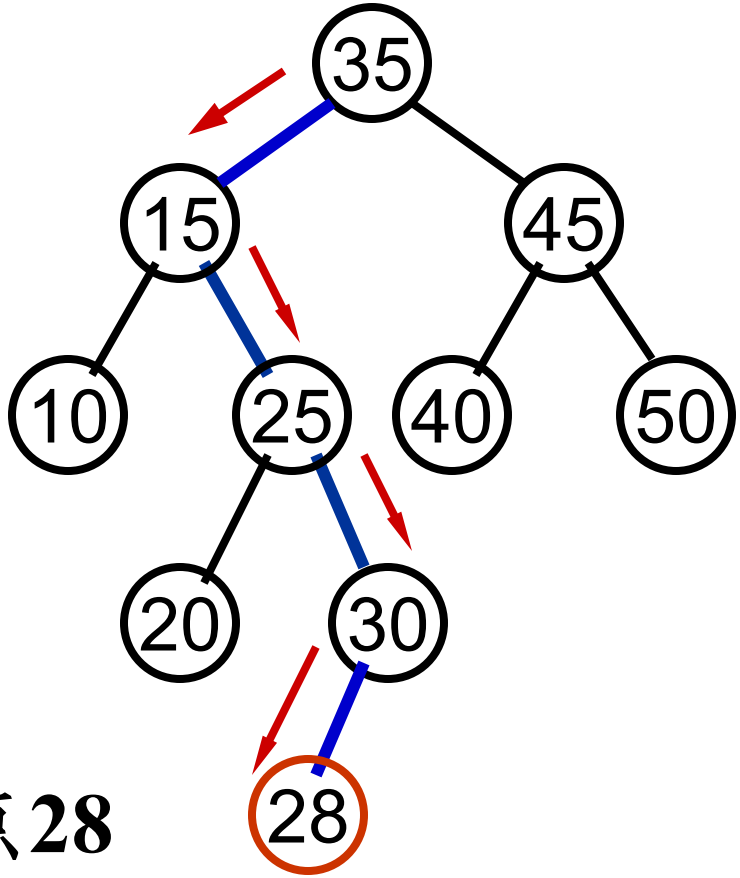
```
Insert ( Type x, BstNode * root)  //root不等于null!!!
{
    parent = root;
    while(true)
    {
        if ( x < parent->data )
        {       if(parent -> leftChild == null)
                    { parent ->leftChild = new BstNode(x); return; }
                parent = parent->leftChild;
        }
        else if ( x > (*ptr)->data )
        {
                if(parent -> right == null)
                    { parent->rightChild = new BstNode(x); return; }
                parent = parent->rightChild;
        }
        else
                return;
    }
}
```

建议和单链表的操作进行比较

这里的parent类似于单链表操作中的prev指针。

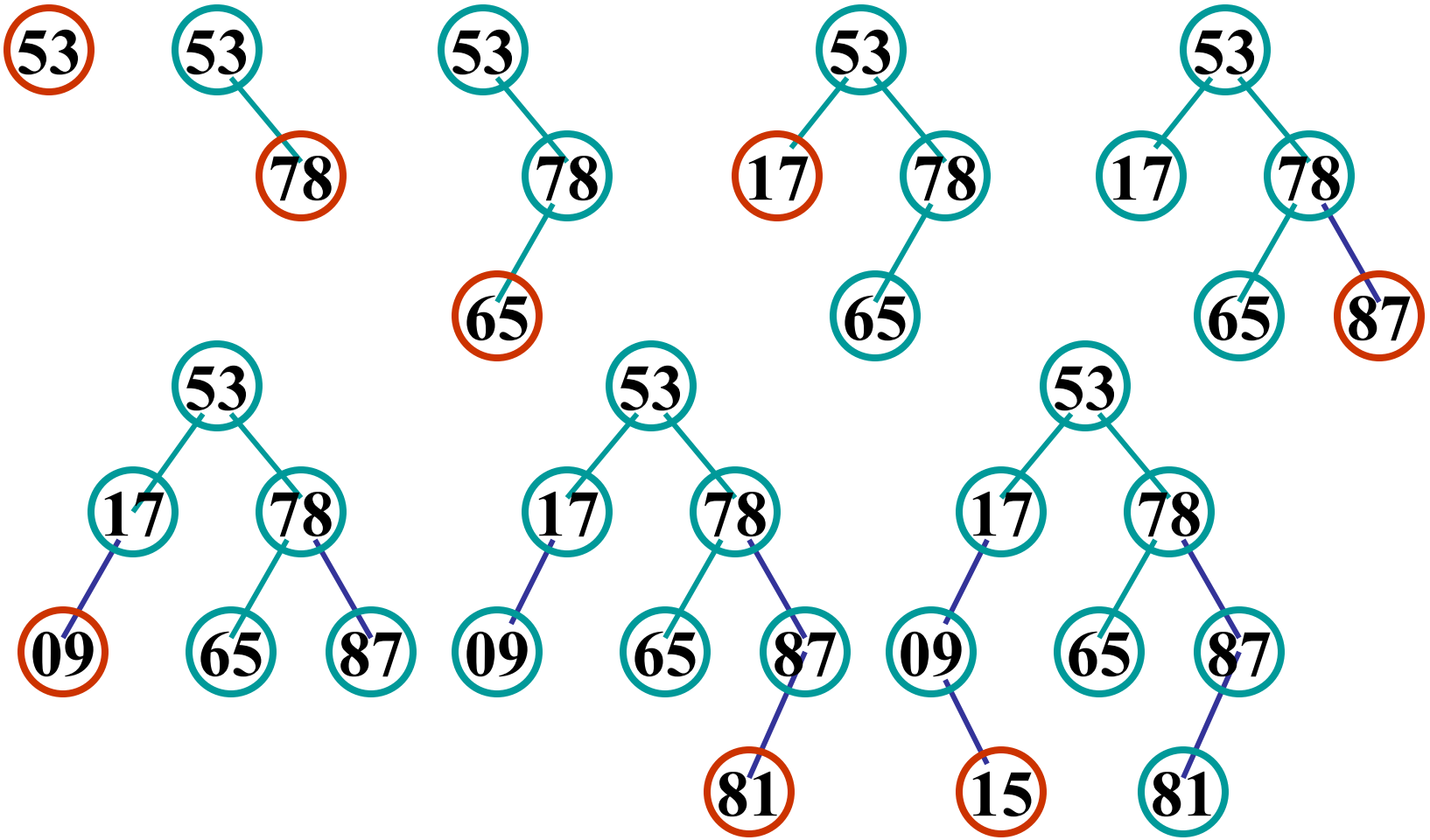
二叉树有时也被称为二叉链表!

二叉搜索树的插入

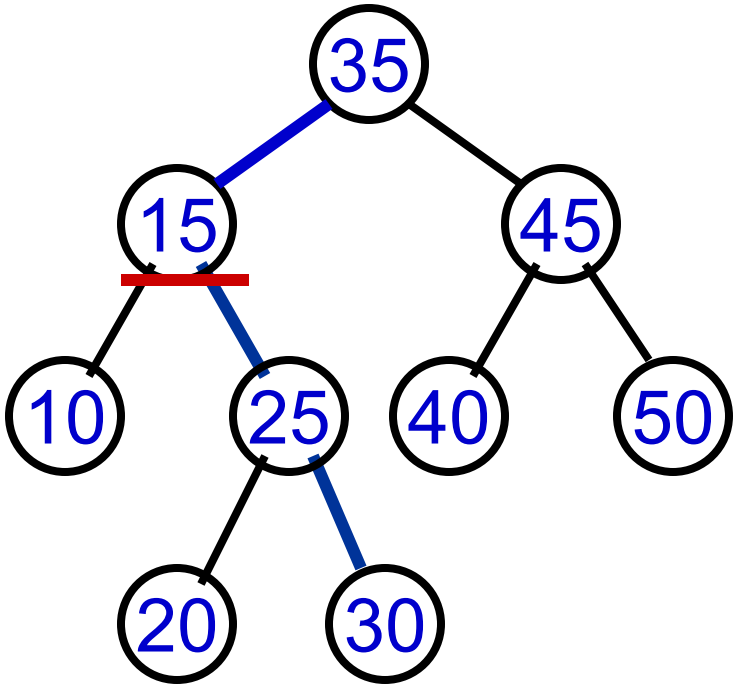


插入新结点28

输入数据{53, 78, 65, 17, 87, 09, 81, 15}，建立二叉搜索树的过程



二叉搜索树的删除



- 在二叉搜索树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉搜索树的性质不会失去。
- 为保证在删除后树的搜索性能不至于降低，还需要防止重新链接后树的高度增加。

基本思想：

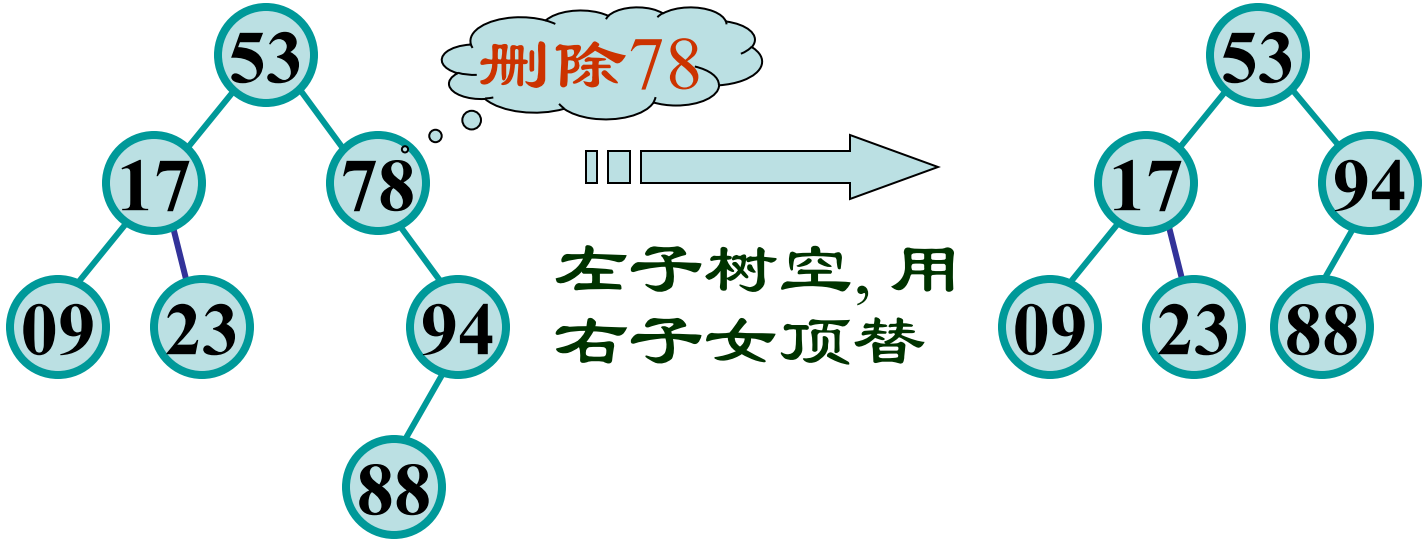
- **首先查找，确定被删除结点是否在二叉搜索树中。**

（删除结点为ptr指针所指，其双亲结点为f结点指针所指，被删除结点的左子树和右子树分别用pl和pr表示。）

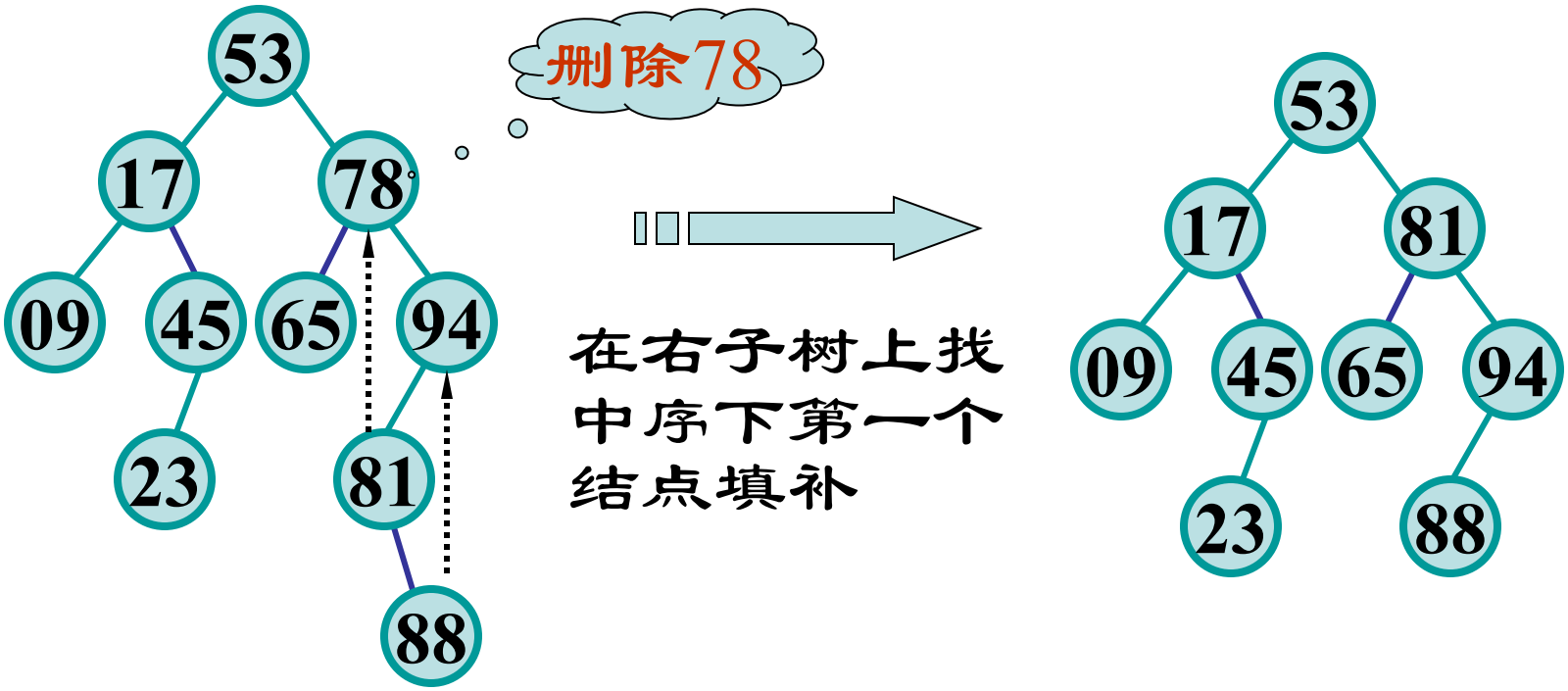
分情况讨论：

- 删除叶结点：
  - 只需将其双亲结点指向它的指针清零，再释放它即可。
- 被删结点右子树或左子树为空：
  - 可以拿它的左子女结点或右子女结点顶替它的位置，再释放它。


53
17
78
09
45
65
87
23
删除45
右子树空，用
左子女顶替
53
17
78
09
23
65
87


53
17
78
09
23
94
88
删除78
左子树空，用
右子女顶替
53
17
94
09
23
88

- 被删结点左、右子树都不为空，？



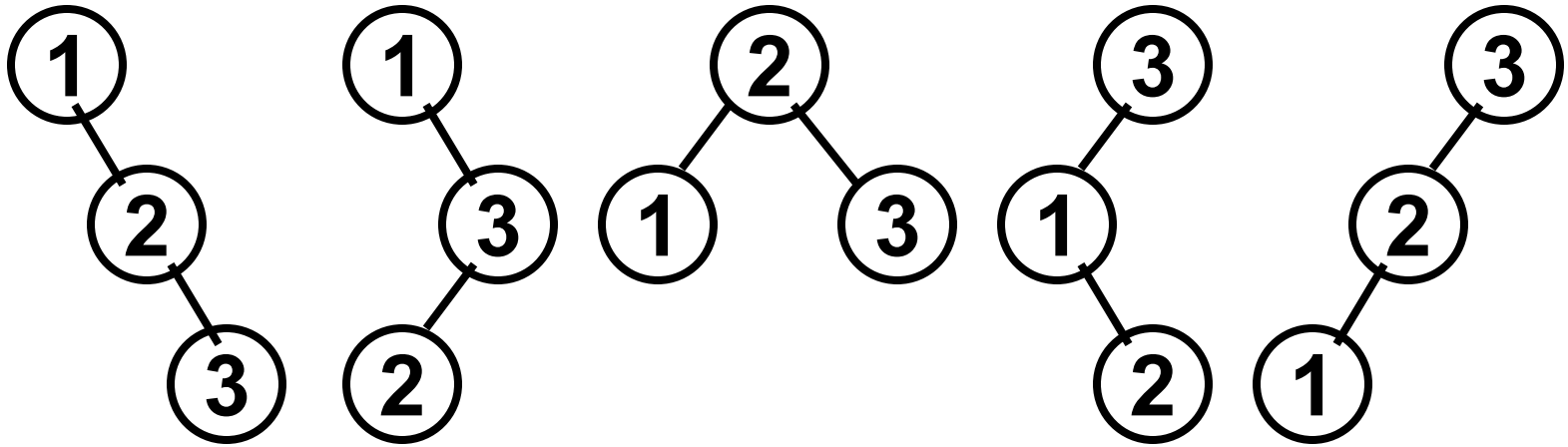
- 可以在它的右子树中寻找中序下的第一个结点(关键码最小)，用它的值填补到被删结点中，再来处理这个结点的删除问题。

二叉搜索树的删除算法

```
Remove (const Type& x, BstNode * & ptr) {
    BstNode * temp;
    if ( ptr == NULL )
            return;
    if ( x < ptr->data )        //在左子树中删除
            Remove ( x, ptr->leftChild );
    else if ( x > ptr->data )   //在右子树中删除
            Remove ( x, ptr->rightChild );
    else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
            temp = Min( ptr->rightChild );
            //找ptr右子树中序下第一个结点
            ptr->data = temp->data;   //填补上
            Remove ( ptr->data, ptr->rightChild );
            //在ptr的右子树中删除temp结点
    }
     else {  // ptr结点只有一个或零个子女
            temp = ptr;
            if ( ptr->leftChild == NULL )
                    ptr = ptr->rightChild;  //只有右子女
            else if ( ptr->rightChild == NULL )
                    ptr = ptr->leftChild;    //只有左子女
            delete temp;
    }
}
```

同样3个数据{1, 2, 3}，输入顺序不同，建立起来的二叉搜索树的形态也不同。

{1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}



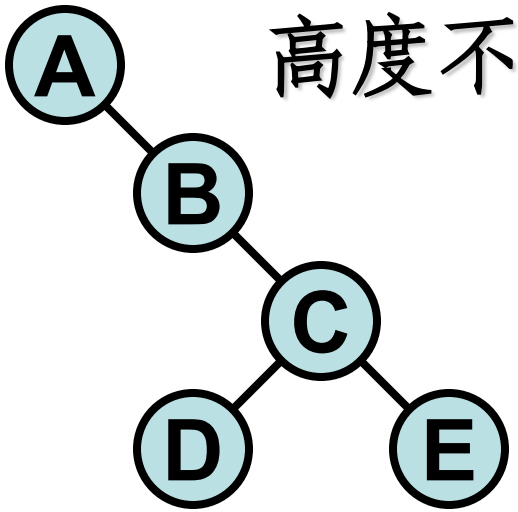
- 如果输入序列选的不好，会建立起一棵单支树，使得二叉搜索树的高度达到最大

平衡处理

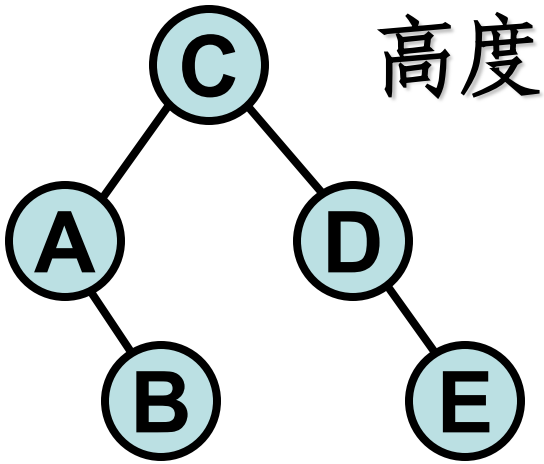
7.5 AVL树

AVL树的定义

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。



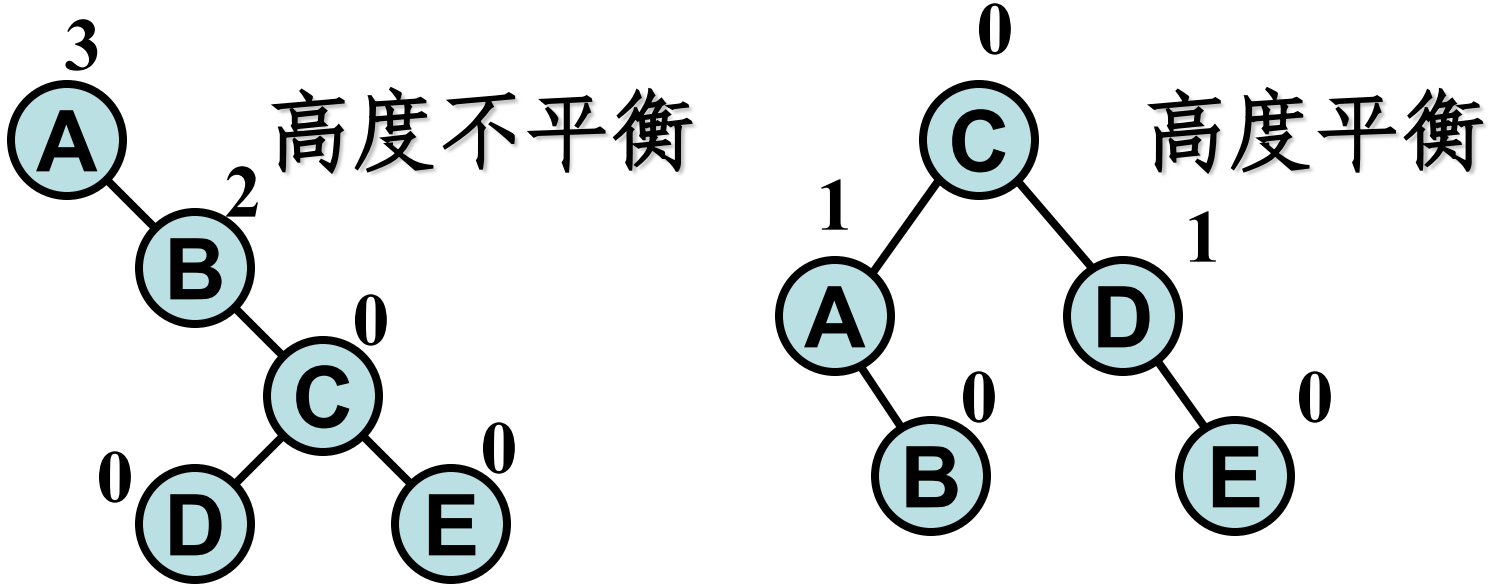
高度不平衡



高度平衡

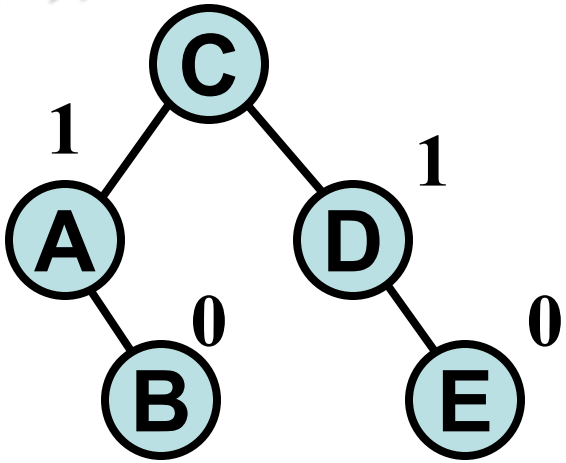
结点的平衡因子balance (balance factor):

- 每个结点附加一个数字,给出该结点右子树的高度减去左子树的高度所得的高度差,这个数字即为结点的平衡因子balance



关于AVL树:

- 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。
- 如果一棵二叉搜索树是高度平衡的, 且有 n 个结点，其高度可保持在$O(\log_2 n)$,平均搜索长度也可保持在$O(\log_2 n)$。



AVL树的类定义

```
template <class Type> class AVLTree {
public:
    struct AVLNode {          //AVL树结点
        Type data;   int balance;
        AVLNode<Type> * left, * right;
        AVLNode ( ) : left (NULL), right (NULL),
            balance (0) { }
        AVLNode ( Type d, AVLNode<Type>
*l = NULL, AVLNode<Type> *r = NULL)
            : data (d), left (l), right (r), balance (0) { }
    };
```

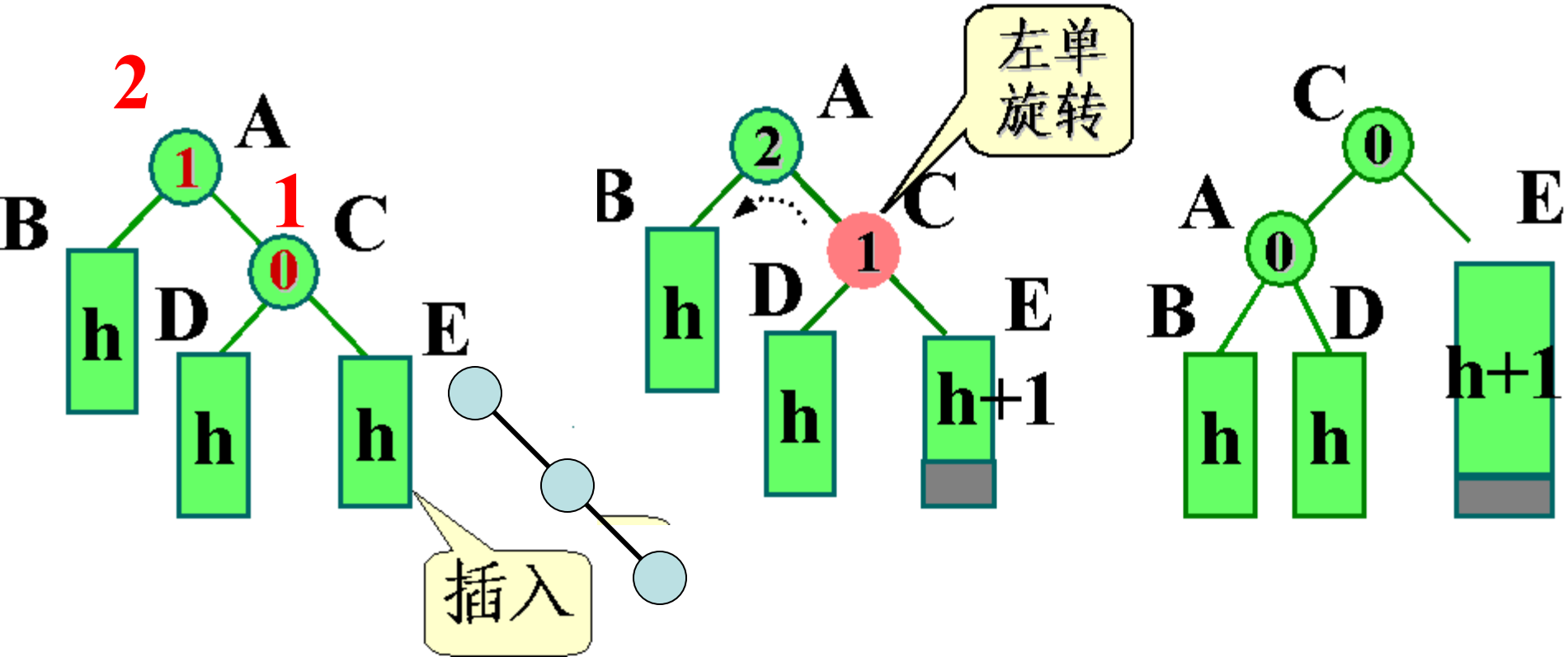
```
protected:
   Type RefValue;
   AVLNode *root;
   int Insert (AVLNode<Type>*& Tree, Type x);
   void RotateLeft (AVLNode<Type> * Tree,
                    AVLNode<Type> *& NewTree);
   void RotateRight (AVLNode<Type> * Tree,
                    AVLNode<Type> *& NewTree);
   void LeftBalance (AVLNode<Type> *& Tree);
   void RightBalance (AVLNode<Type> *&Tree);
   int Height (AVLNode<Type> * t) const;
public:
```

```
    AVLTree ( ) : root (NULL) { }
    AVLNode (Type Ref ) : RefValue (Ref),
          root (NULL) { }
    int Insert (Type x)
       { return Insert (root, x); }
    friend istream& operator >> (istream& in,
        AVLTree<Type>& Tree);
    friend ostream& operator << (ostream& out,
       const  AVLTree<Type>& Tree);
    int Height ( ) const;
}
```

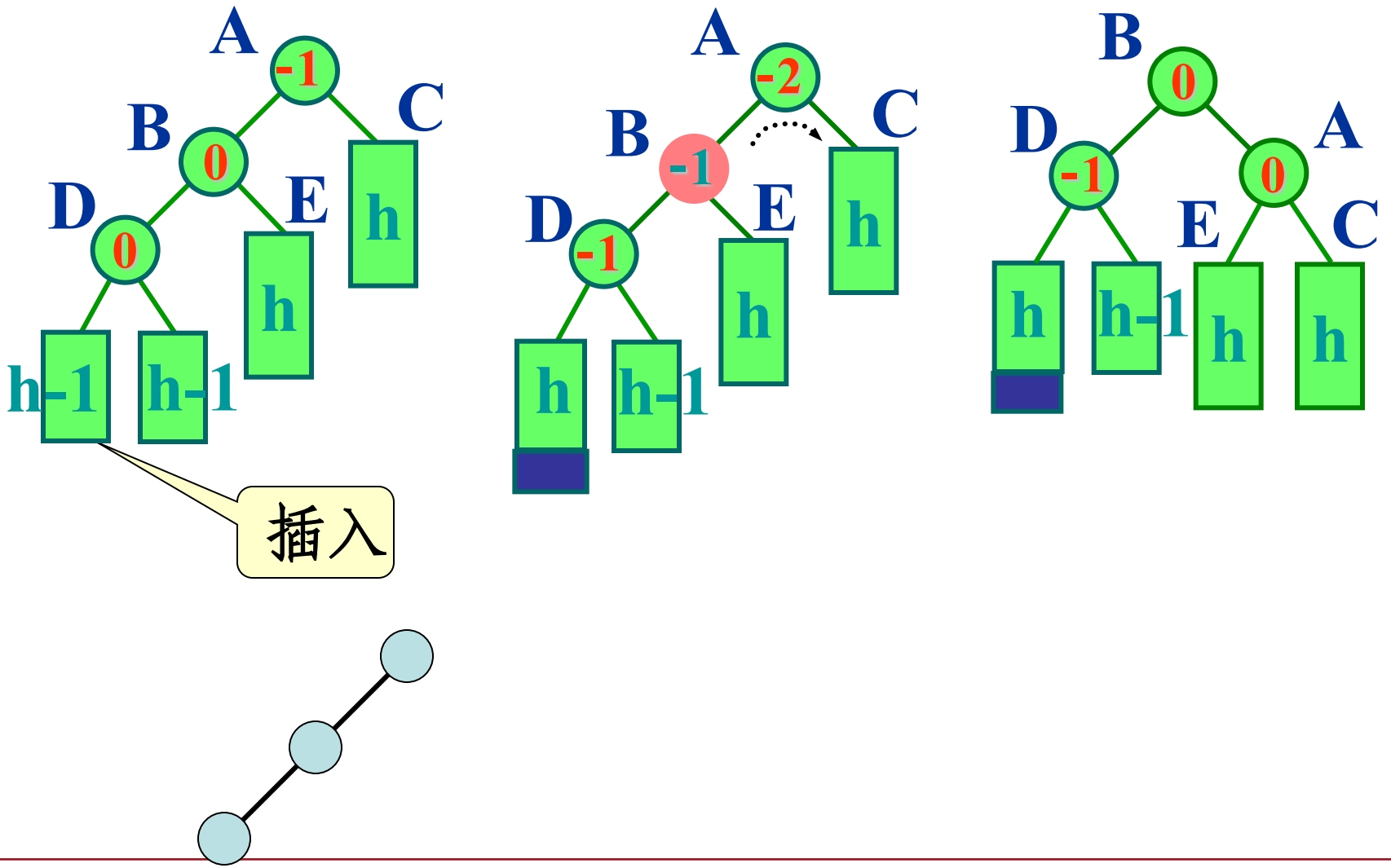
平衡化旋转

- 有两类:
  - 单旋转 (左旋和右旋)
  - 双旋转 (左平衡和右平衡)
- 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。因此，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。

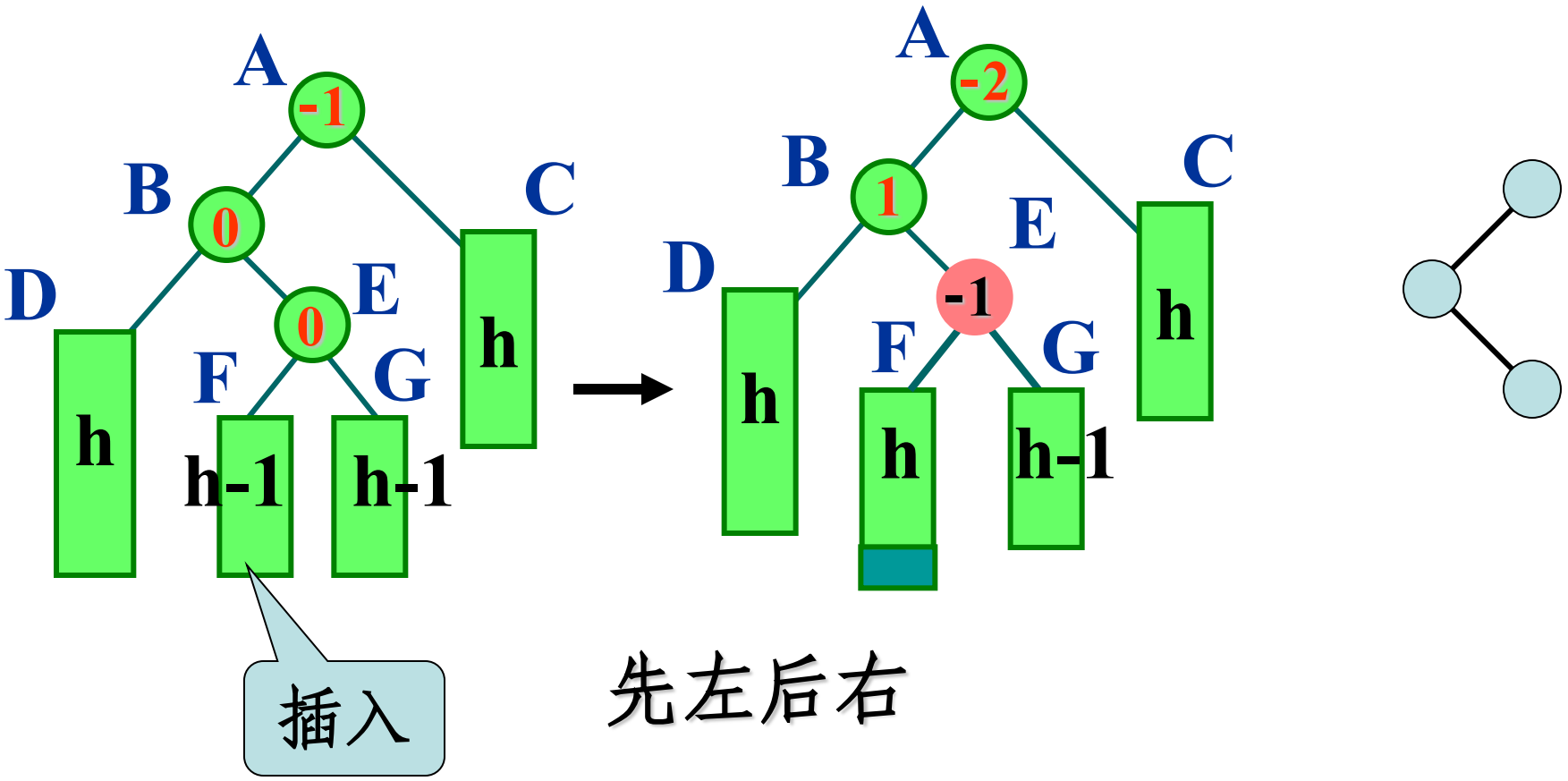
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。

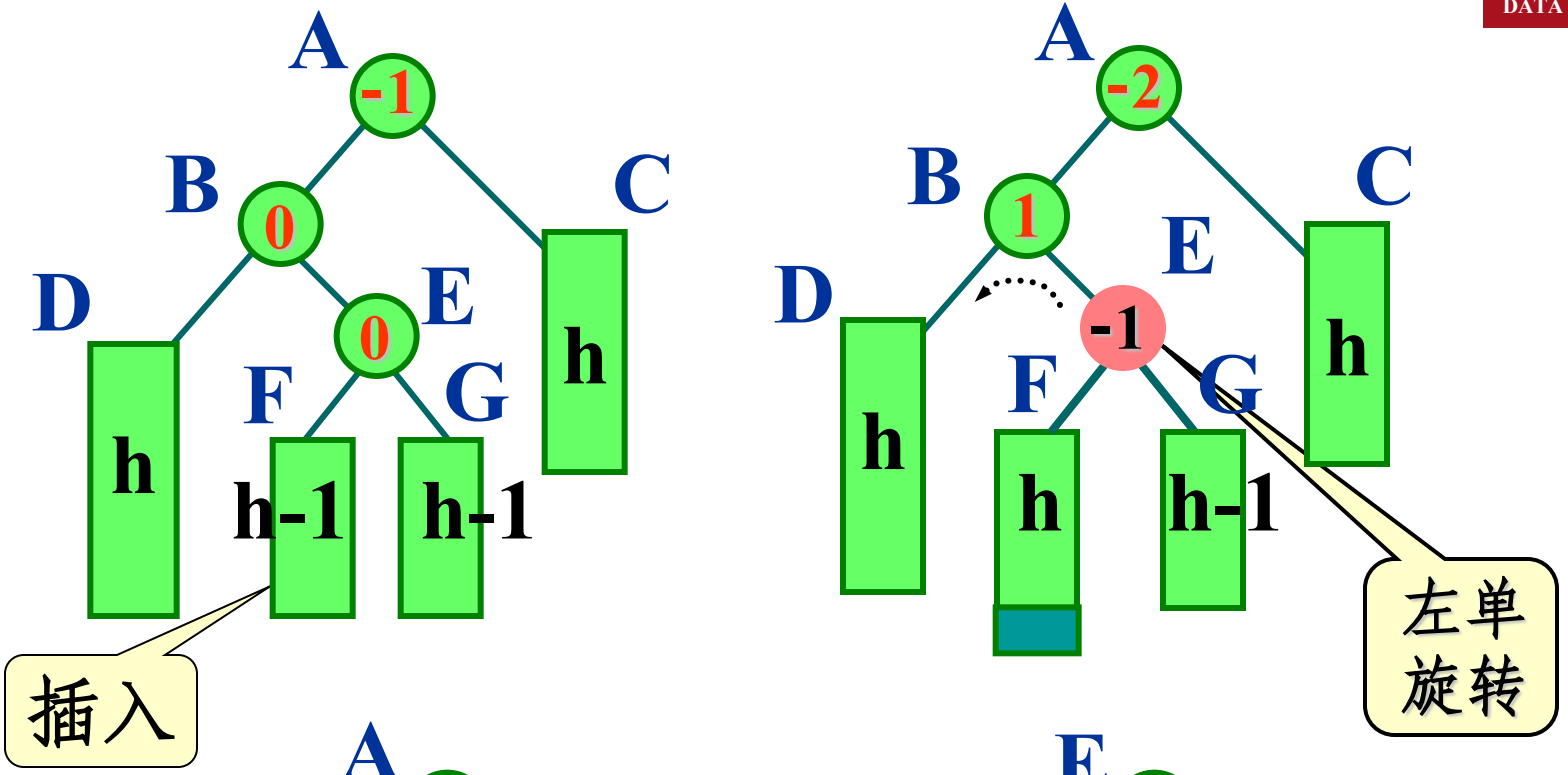


右单旋转 (RotateRight)

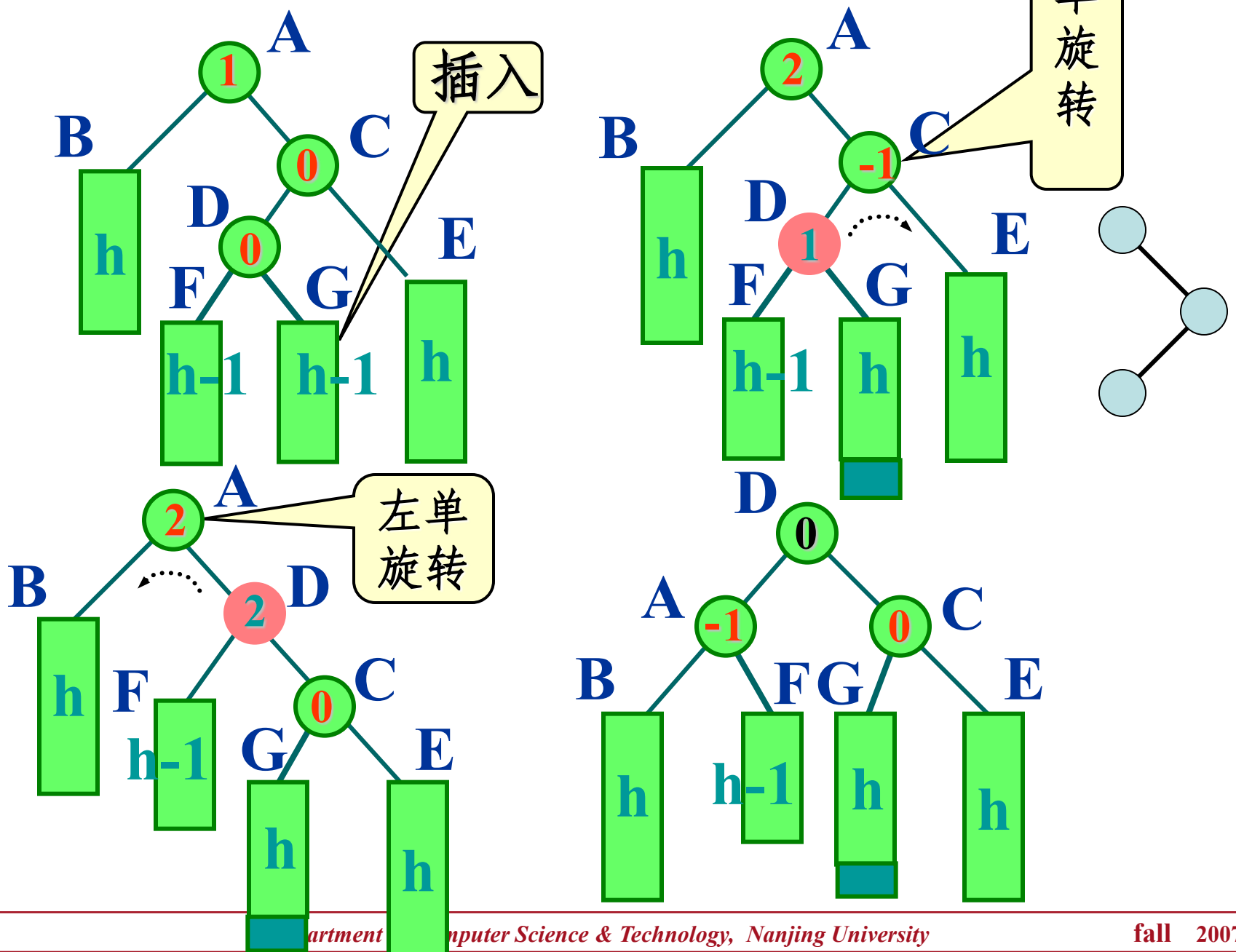


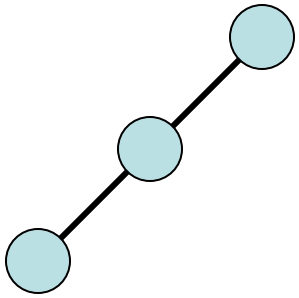
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。



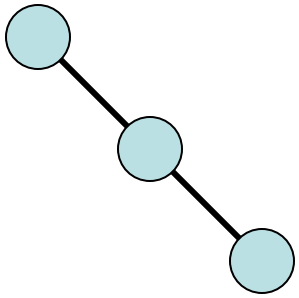

A
-1
B
0
C
D
E
0
h
F
G
h
h-1
h-1
插入
A
-2
B
1
E
C
D
-1
h
F
G
h
h
h-1
左单
旋转
A
-2
右单
旋转
E
-2
C
G
B
0
h-1
h
D
F
h
h
E
0
B
0
A
-1
D
F
G
C
h
h
h-1
h

先右后左

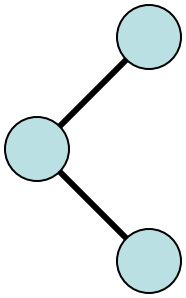

A
1
B
h
0
C
D
0
E
F
G
h-1
h-1
h
插入
A
2
B
h
-1
C
D
1
E
F
G
h-1
h
h
右单旋转
A
2
B
h
D
2
F
h-1
0
C
G
E
h
h
左单旋转
D
0
A
-1
0
C
B
F
G
E
h
h-1
h
h



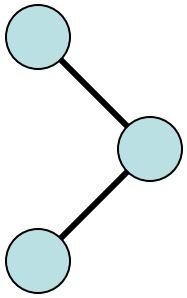
右单旋转



左单旋转



左右双旋转



右左双旋转

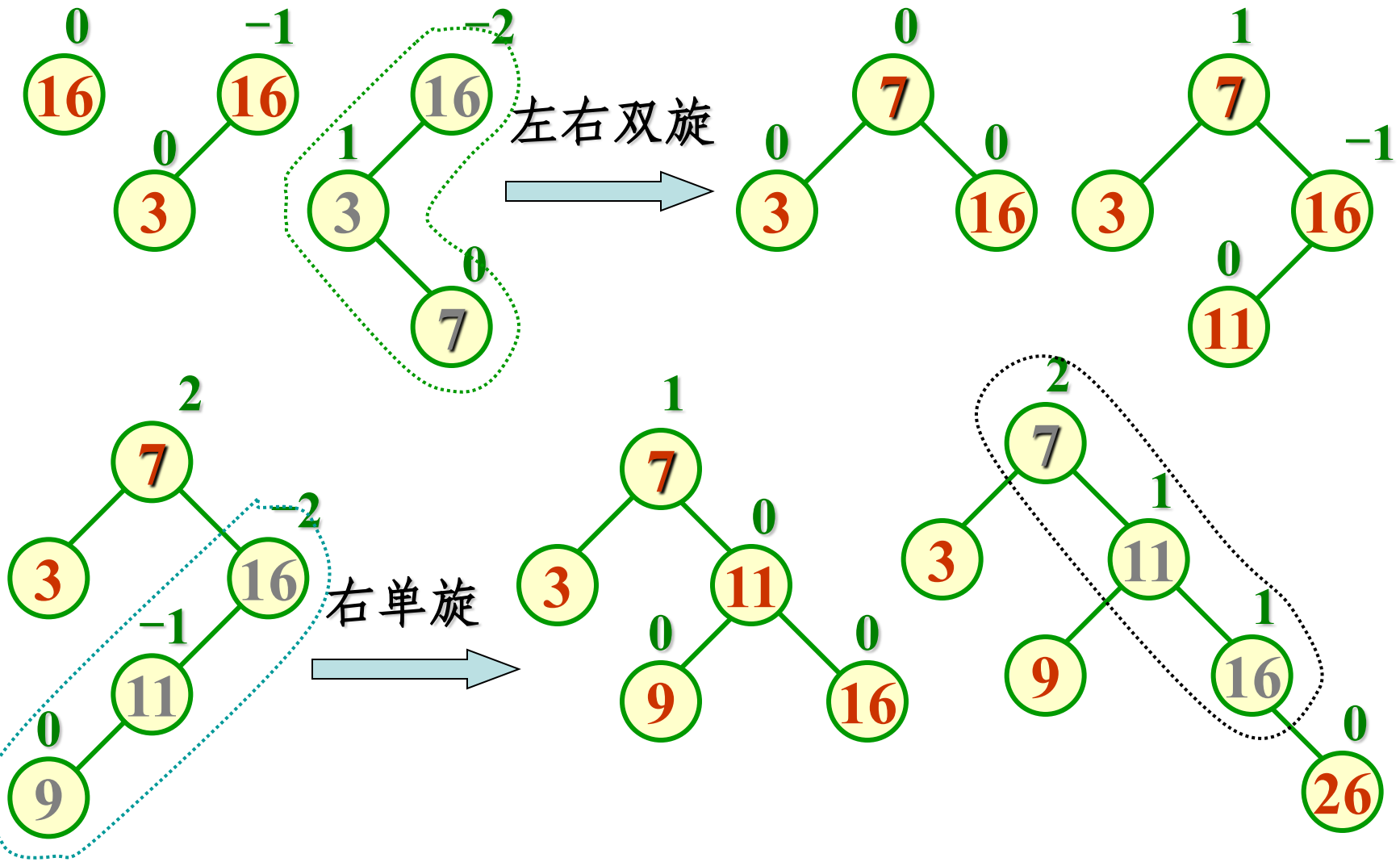
AVL树的插入

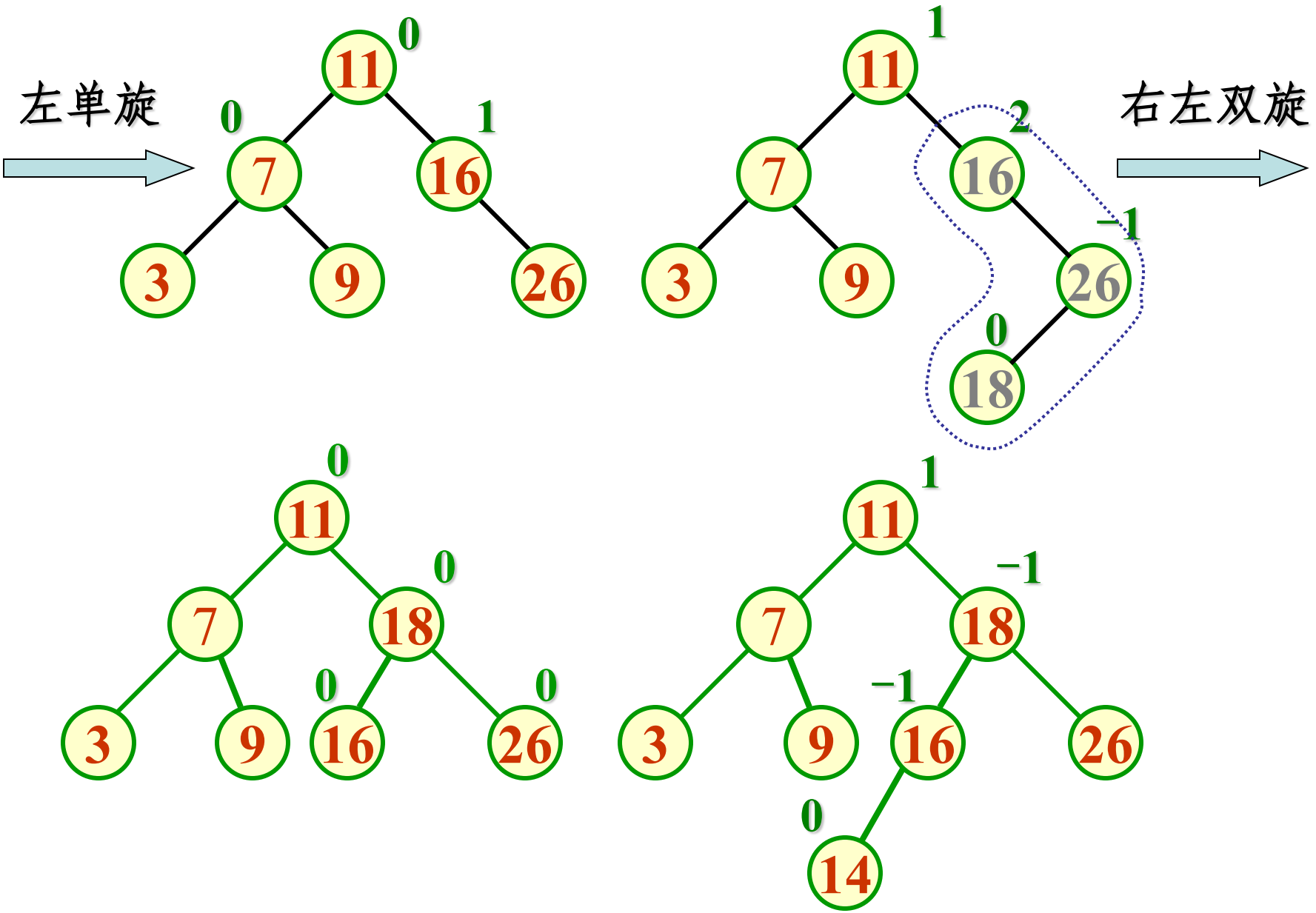
- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 |balance| > 1，则出现了不平衡，需要做平衡化处理。

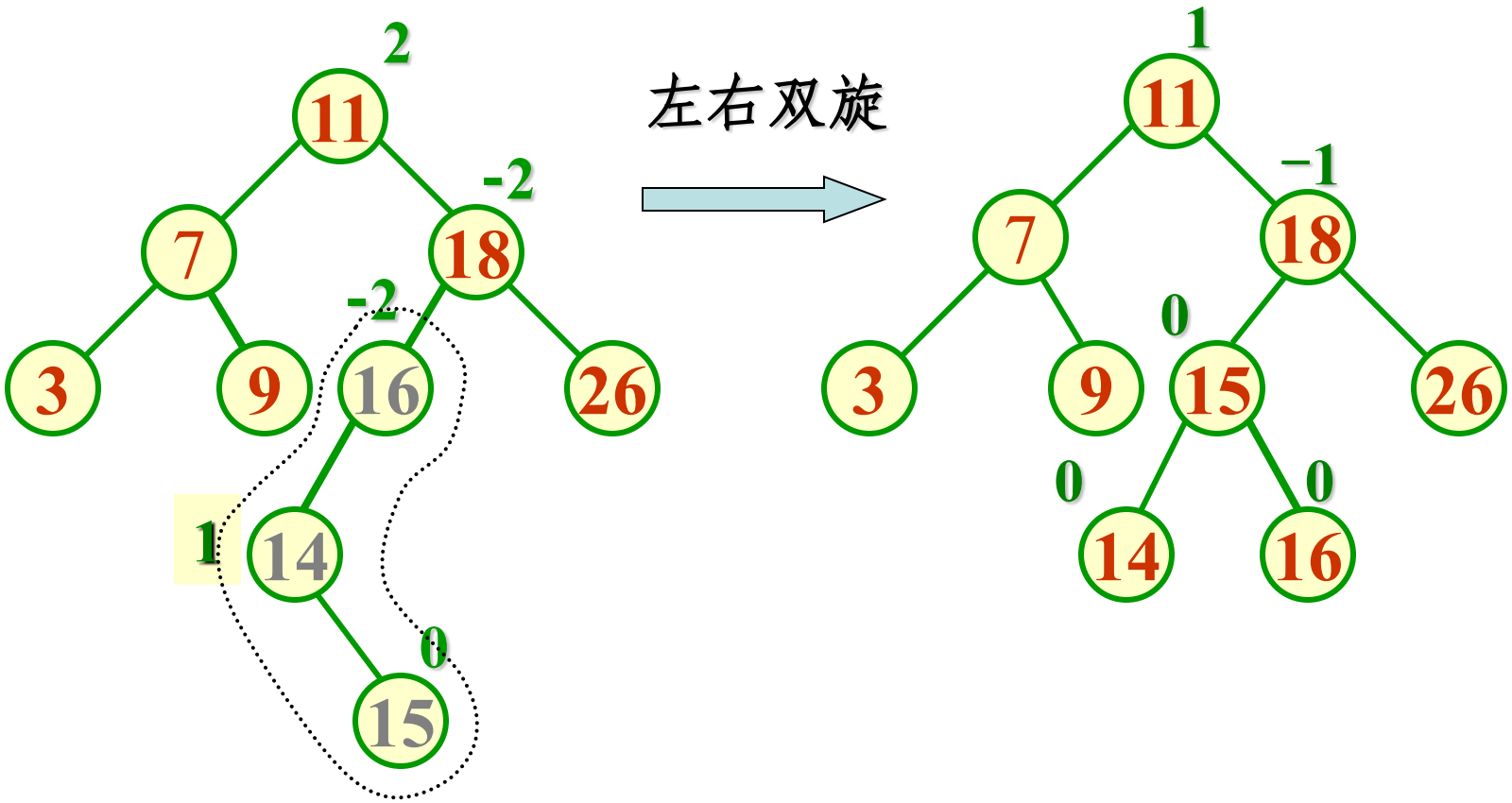
- 在AVL树上定义了重载操作“>>”和“<<”，以及中序遍历的算法。利用这些操作可以执行AVL树的建立和结点数据的输出。

- 算法从一棵空树开始，通过输入一系列对象关键码，逐步建立AVL树。在插入新结点时使用平衡旋转方法进行平衡化处理。

例，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。




左单旋
11
7
16
3
9
26
右左双旋
18
14



从空树开始的建树过程

AVL树的插入算法

- 下面的算法将通过递归方式将新结点作为叶结点插入并逐层修改各结点的平衡因子。
- 在发现不平衡时立即执行相应的平衡化旋转操作，使得树中各结点重新平衡化。
- 在程序中，用变量**success**记载新结点是否存储分配成功，并用它作为函数的返回值。
- 算法从树的根结点开始，递归向下找插入位置。在找到插入位置(空指针)后，为新结点动态分配存储空间，将它作为叶结点插入，并置**success**为**1**，再将**taller**置为**1**，以表明插入成功。在退出递归沿插入路径向上返回时做必要的调整。

```
template <class Type> int AVLTree<Type> ::
Insert ( AVLNode<Type>* &tree, Type x,
        int &taller ) {
//AVL树的插入算法
    int success;
    if ( tree == NULL ) {
       tree = new AVLNode (x);
       success = ( tree != NULL ) ? 1 : 0;
       if ( success ) taller = 1;
    }
    else if ( x < tree->data ) {
       success = Insert ( tree->left, x, taller );
       if ( taller )
```

```
    switch ( tree->balance ) {
      case -1 : LeftBalance ( tree, taller ); break;
      case 0 :  tree->balance = -1; break;
      case 1 :  tree->balance = 0;  taller = 0;
    }
}
else if ( x > tree->data ) {
   success = Insert ( tree->right, x, taller );
   if ( taller )
     switch ( tree->balance ) {
       case -1 : tree->balance = 0;  taller = 0;
                 break;
       case 0 :  tree->balance = 1;  break;
       case 1 :  RightBalance ( tree, taller );
```

```
        }
    }
    return success;
}
```

AVL树的删除

(1) 如果被删结点x最多只有一个子女，那么问题比较简单:

- 将结点x从树中删去。
- 因为结点x最多有一个子女，可以简单地把x的双亲结点中原来指向x的指针改指到这个子女结点；
- 如果结点x没有子女，x双亲结点的相应指针置为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*有两个子女:

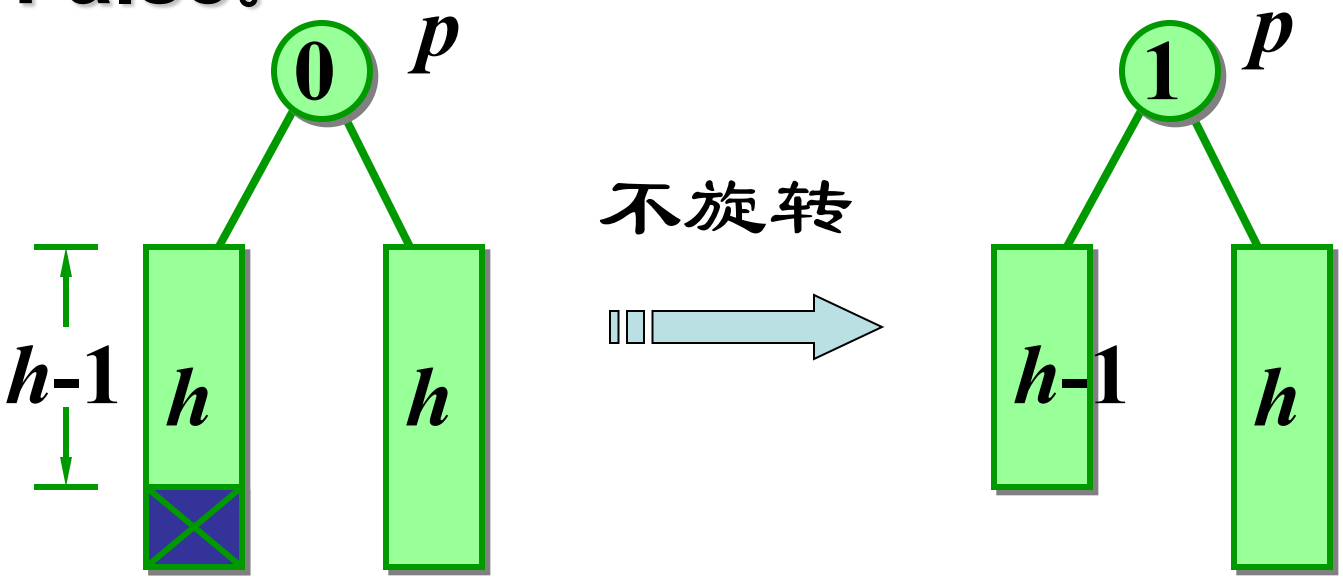
- 搜索 *x* 在中序次序下的直接前驱 *y* (同样可以找直接后继)。
- 把 结点*y* 的内容传送给结点*x*，现在问题转移到删除结点 *y*。把结点*y*当作被删结点*x*。
- 因为结点*y*最多有一个子女，我们可以简单地用(1)给出的方法进行删除。

- 必须沿 *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。

- 用一个布尔变量 **shorter** 来指明子树的高度是否被缩短。在每个结点上要做的操作取决于 **shorter** 的值和结点的 **balance**，有时还要依赖子女的 **balance** 。

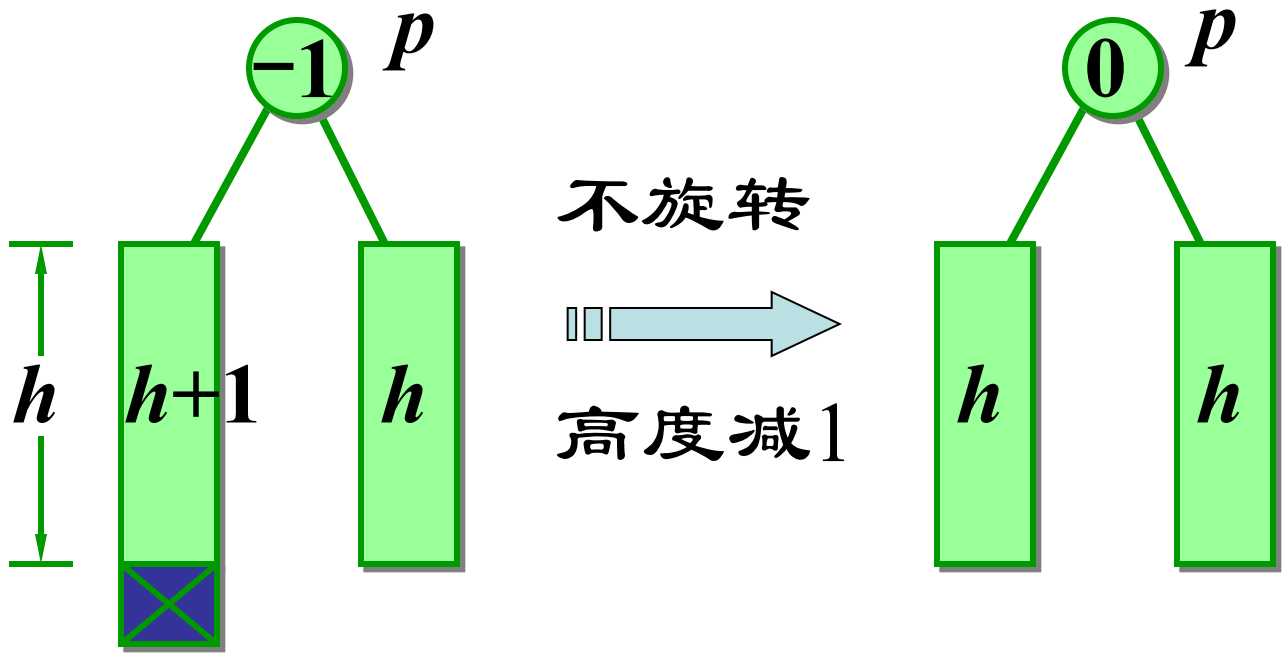
- 布尔变量 **shorter** 的值初始化为**True**。然后对于从 **x** 的双亲到根的路径上的各个结点 **p**，在 **shorter** 保持为 **True** 时执行下面操作。如果 **shorter** 变成**False**，算法终止。

- case 1：当前结点 *p* 的balance为0。如果它的左子树或右子树被缩短，则它的balance改为 1 或-1，同时 shorter 置为 False。



删除一个结点，
高度降低一层

- case 2：结点 *p* 的balance不为0，且较高的子树被缩短，则 *p* 的balance改为0，同时 shorter 置为True。



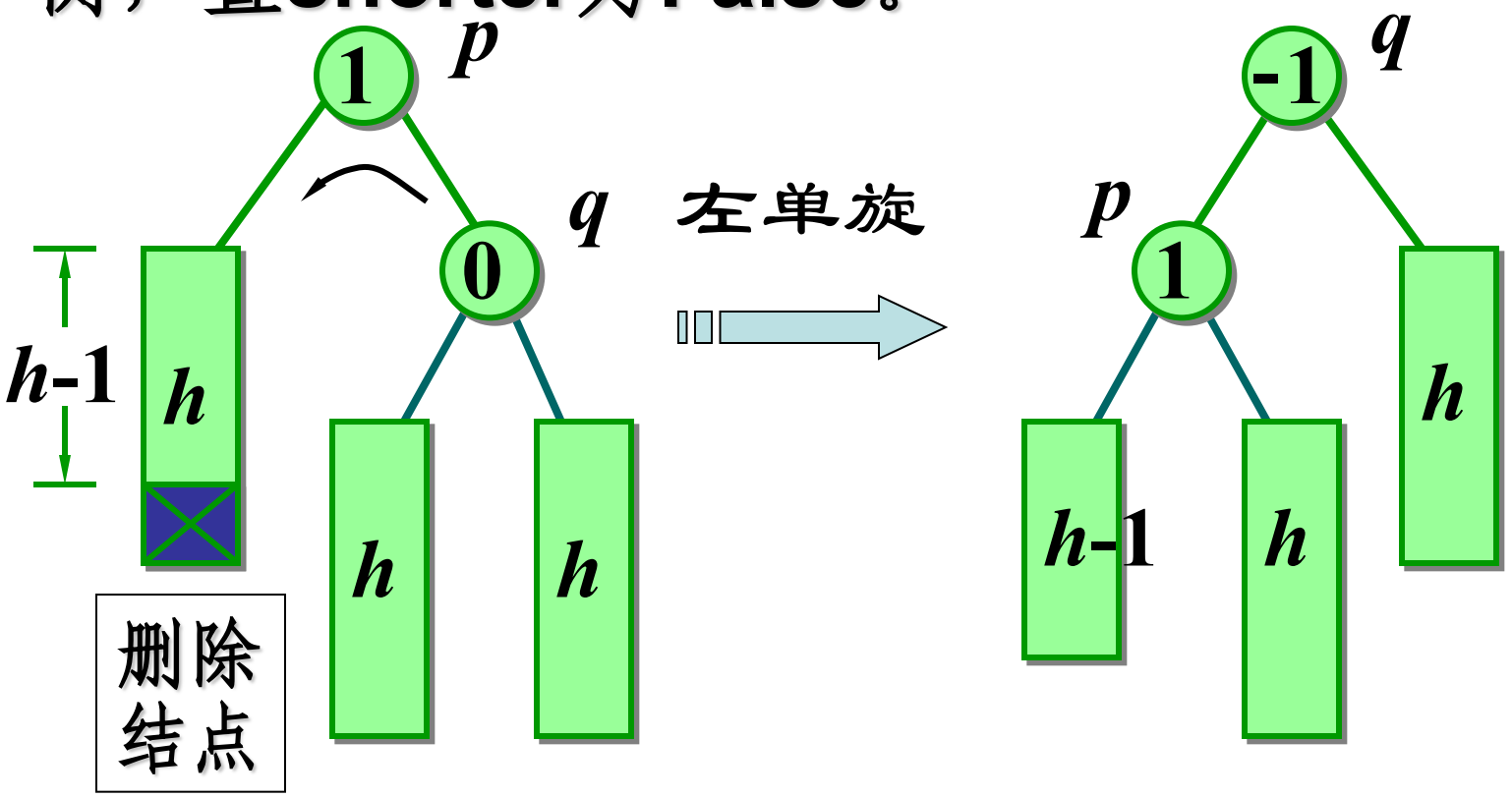
删除一个结点，高度降低一层

- **case 3**：结点 ***p*** 的**balance**不为**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。

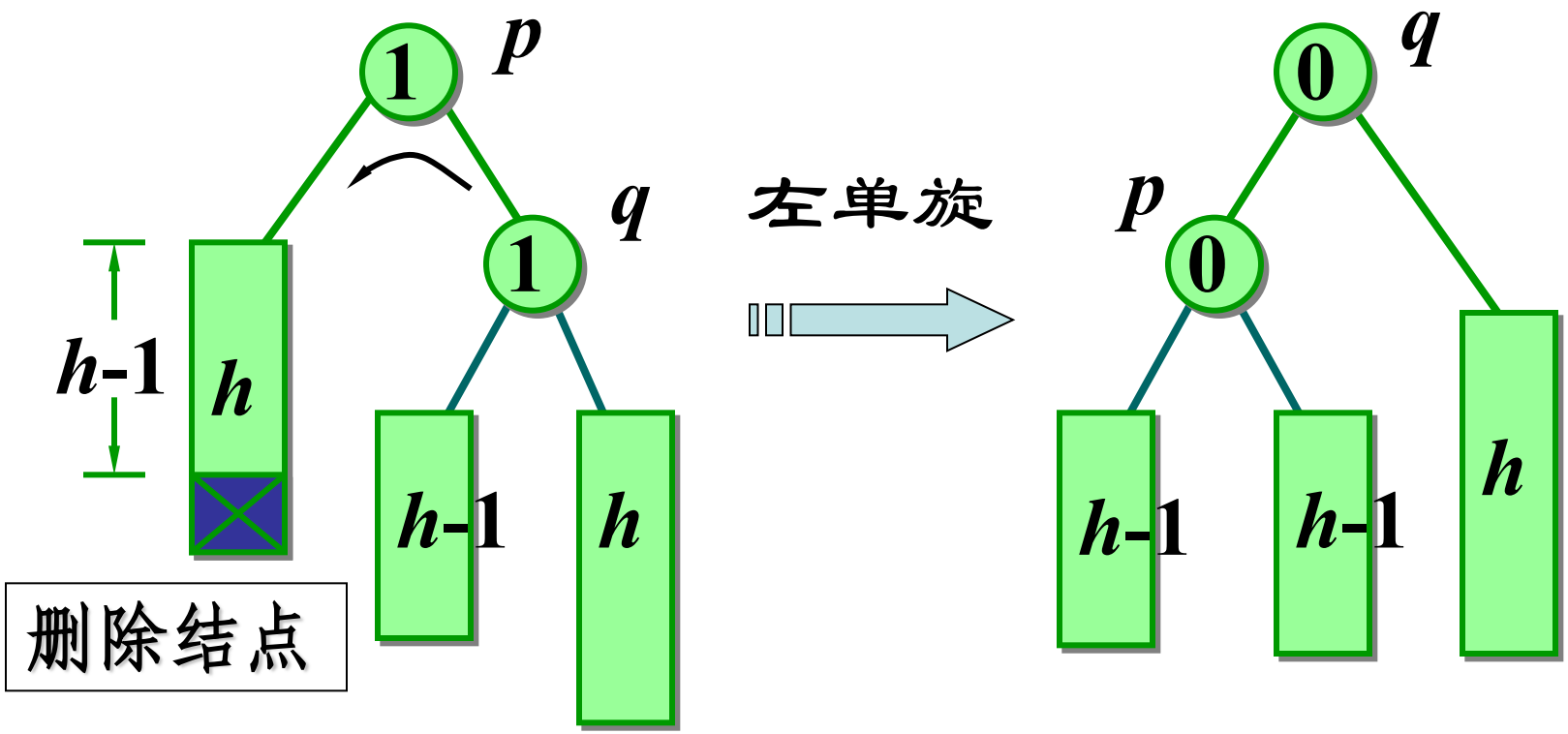
有 3 种平衡化操作：

令 p 的较高的子树的根为 q (该子树未被缩短)，根据 q 的balance

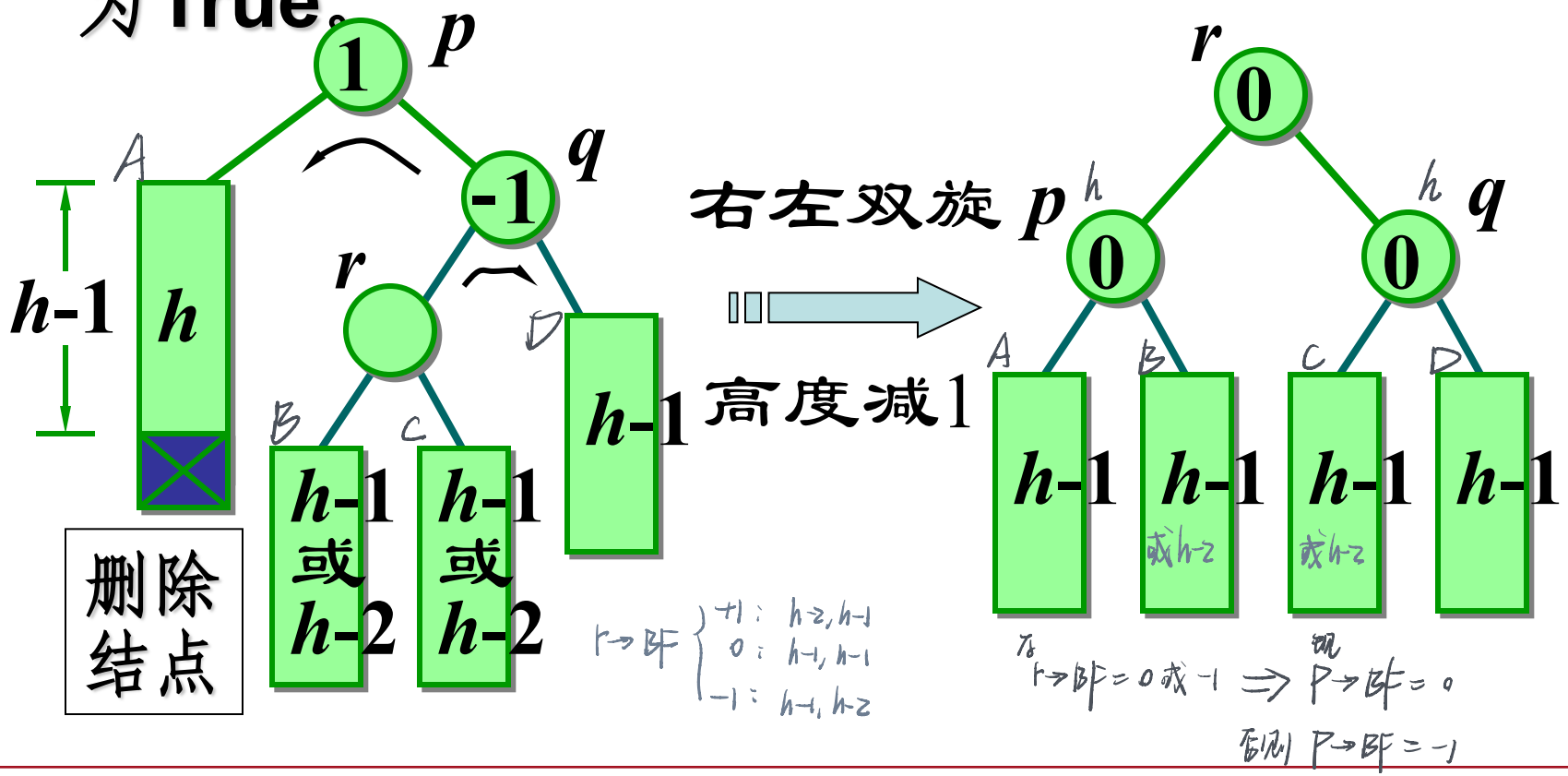
- case 3a：如果 q (较高的子树) 的balance为 0，执行一个单旋转来恢复结点 p 的平衡，置shorter为False。



- case 3b：如果 *q* 的balance与 *p* 的balance相同，则执行一个单旋转来恢复平衡，结点 *p* 和 *q* 的balance均改为0，同时置shorter 为 True。



- **case 3c：如果 *p* 与 *q* 的balance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balance置为0，其他结点的balance相应处理，同时置shorter为True。**

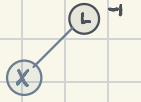


偷懒算法：

原：Node* A = P->left
B = r->left
C = r->right
D = q->right

现：r->left = P
r->right = q
P->left = A
P->right = B
q->left = C
q->right = D

Insert(M, X)
Insert(E, X)
Insert(J, X)
Insert(K, X)
Insert(L, X)
⇒



LeftRebalance(L)
查看，-1

} 无需再返回

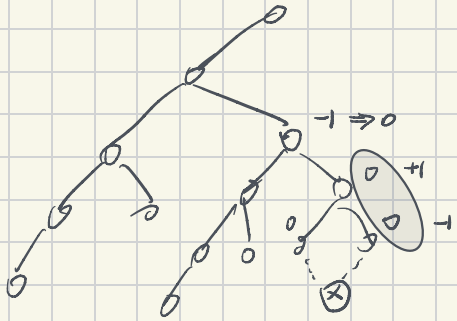
J->BF, J->height, break 结束

RightRebalance(K) Rotate.

current: L	K, right
current: K	J, right
current: J	E, right
current: E	M, left

K > J ? right : left.

进一步指去位置空间



找最后一个BF≠0的节点

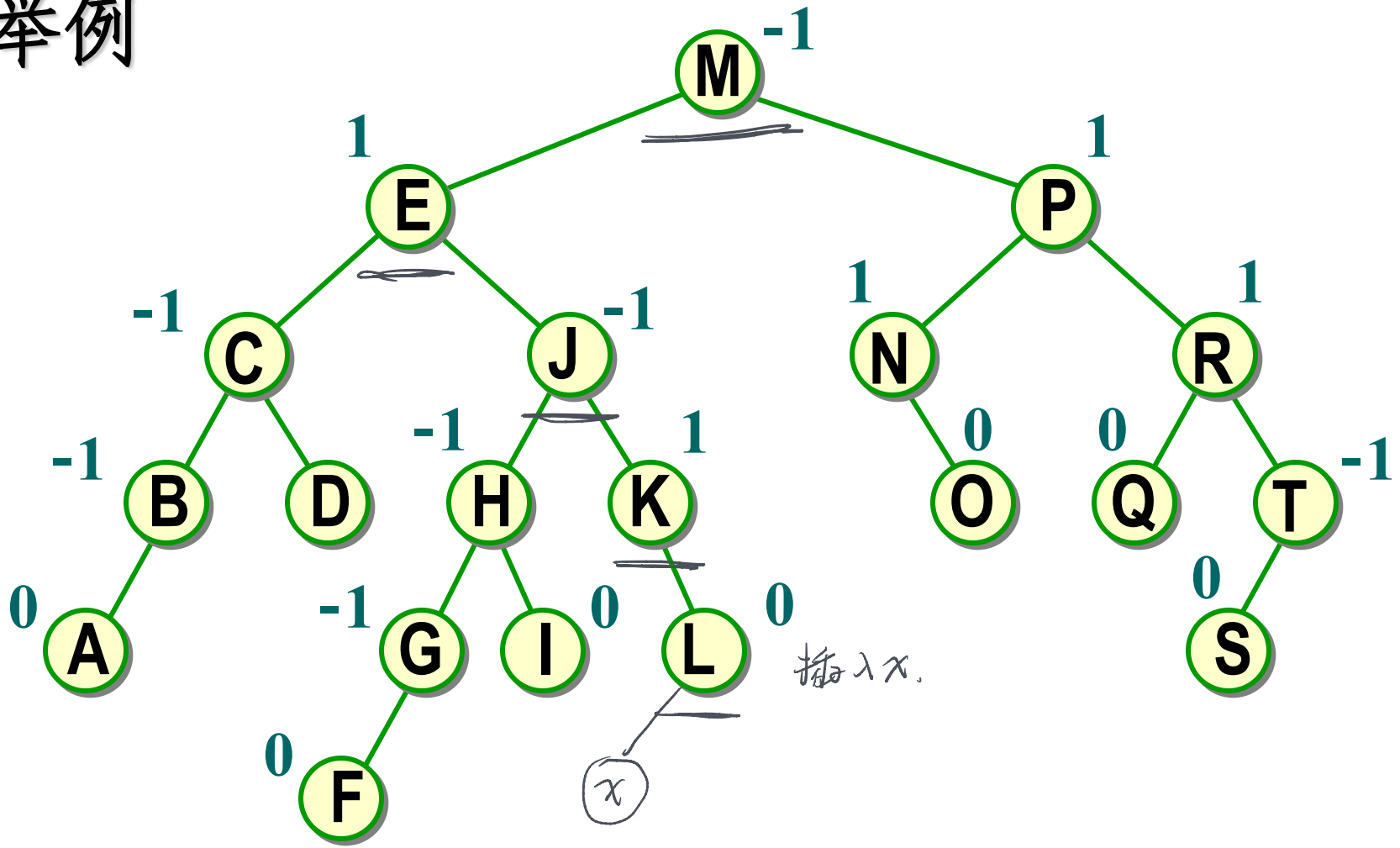
指针

封住BST插入
- ① 搜索插入位置并寻找最后一个BF≠0的节点及其父节点
  - 临时 temp = 所有非零节点
- ② 插入
- ③ 修正最后一个BF≠0节点的BF
- ④ 对最后BF≠0节点旋转（可能是根节点，无需rotate）

左子树增加
- P->BF = -1
- P->BF = 1 且 q->BF ≠ 0 → 右子节点

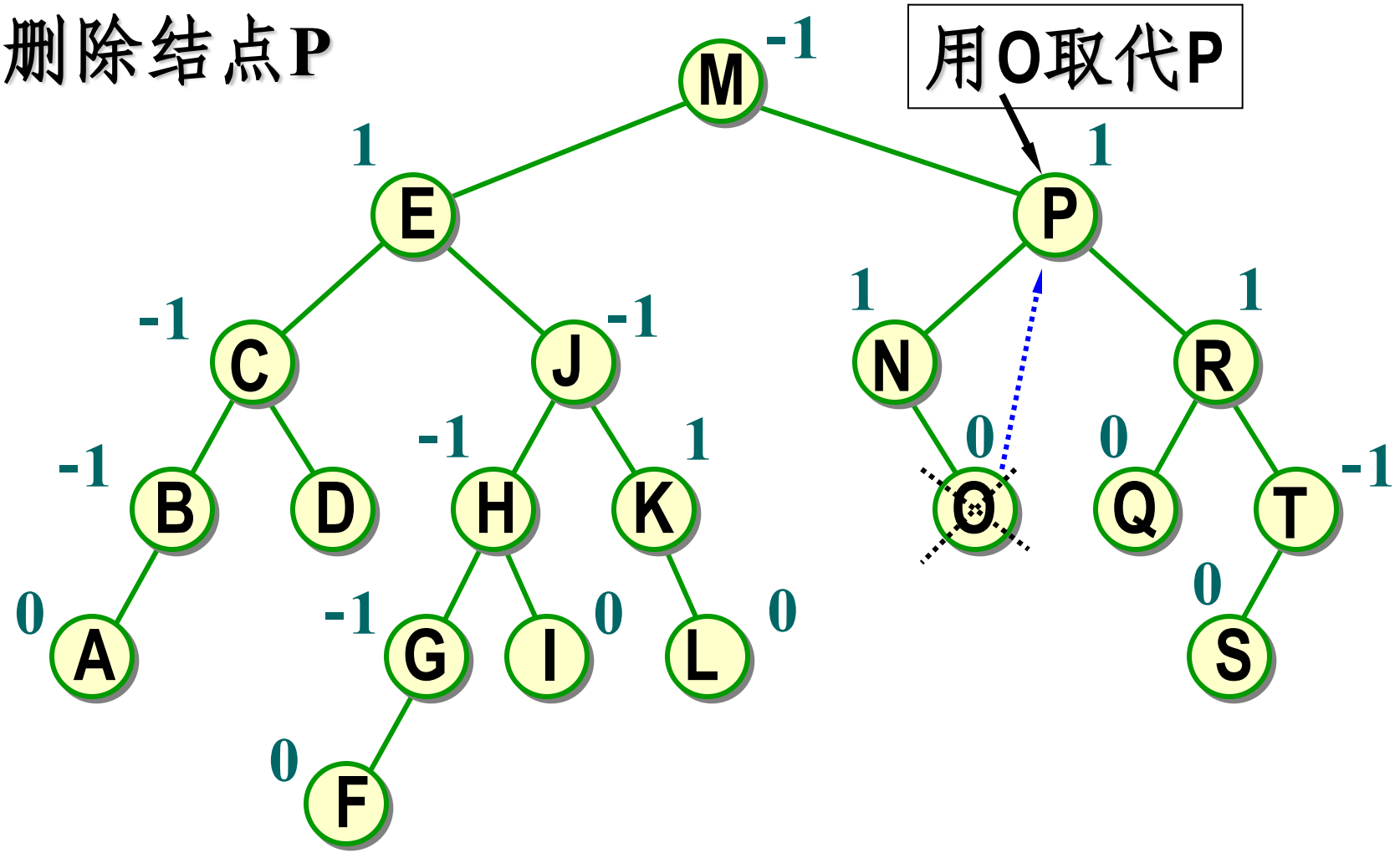
非上述 最后一个不会变化的子树节点

举例



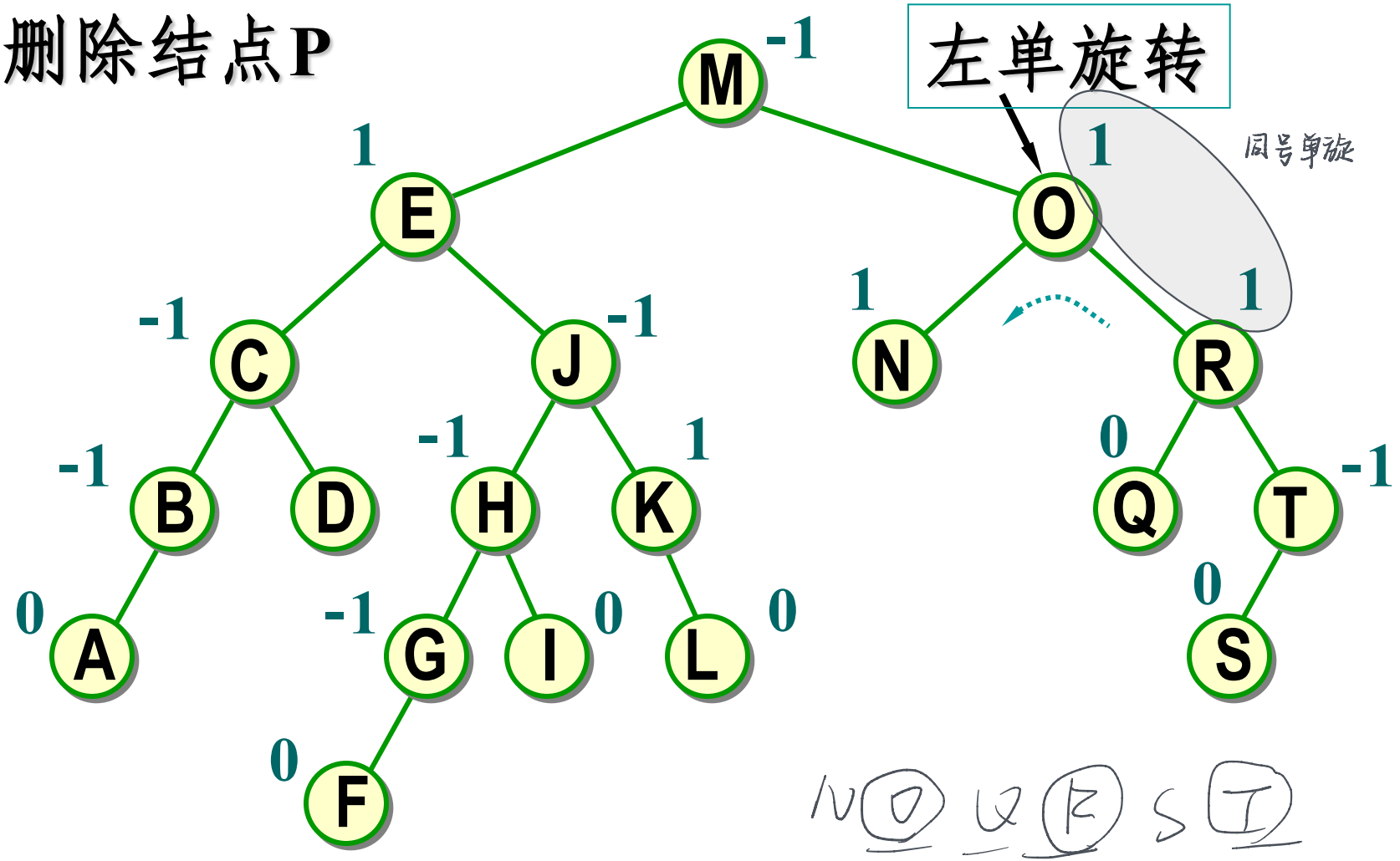
树的初始状态

删除结点P



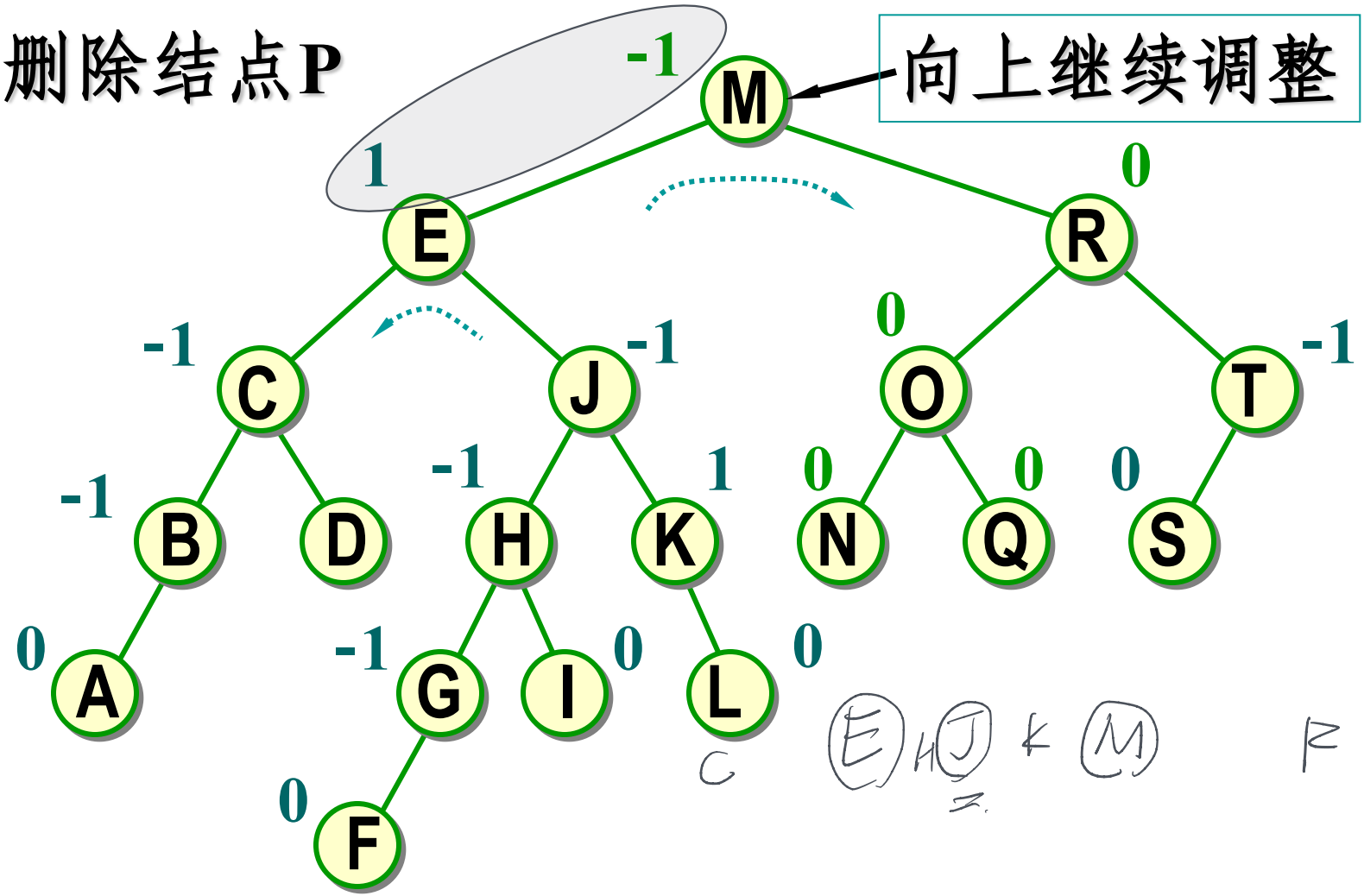
寻找结点P在中序下的直接前驱O, 用O顶替P, 删除O。

删除结点P



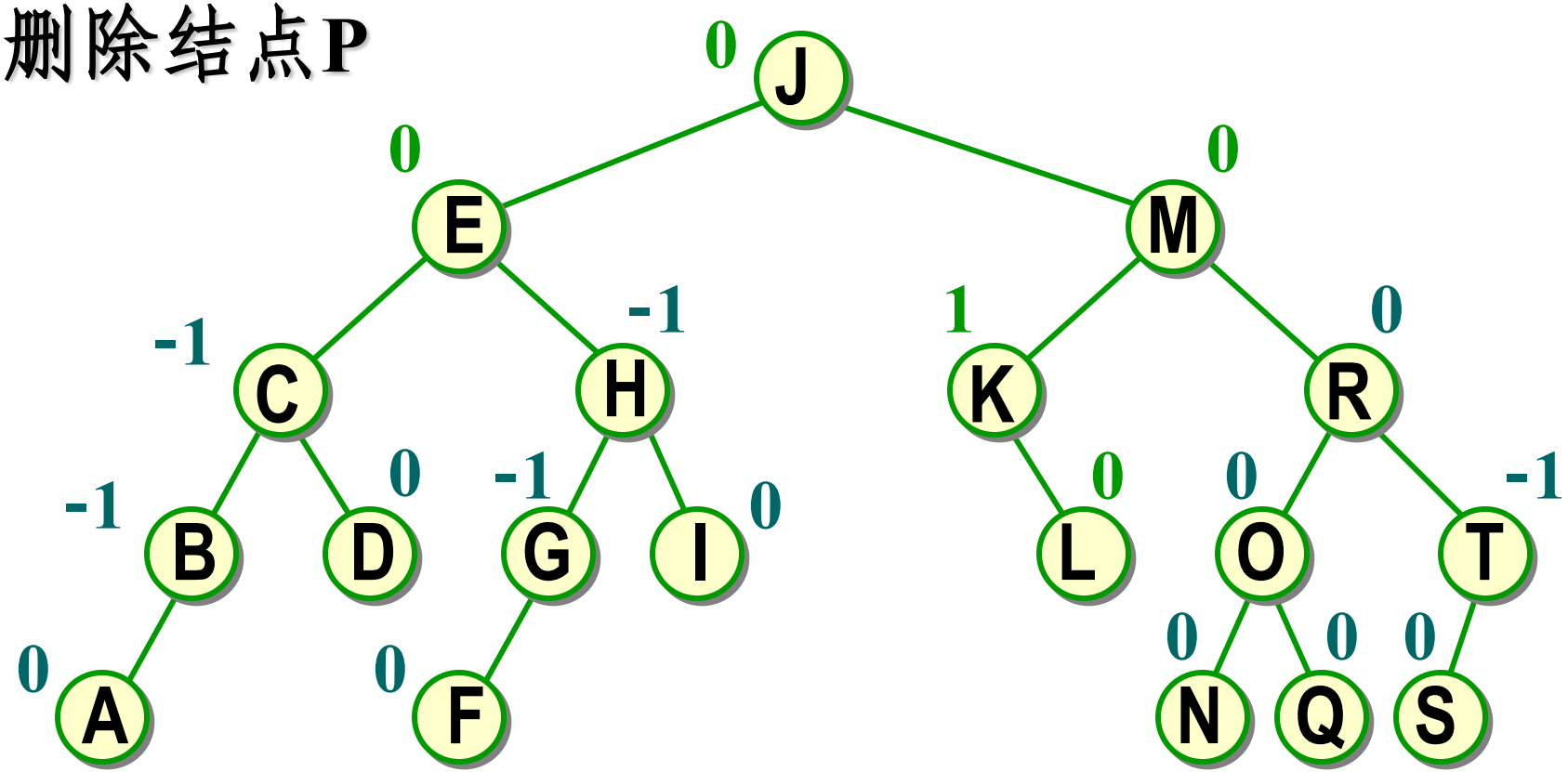
O与R的平衡因子同号，以R为旋转轴做左单旋转，M的子树高度减1。

删除结点P



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。

删除结点P



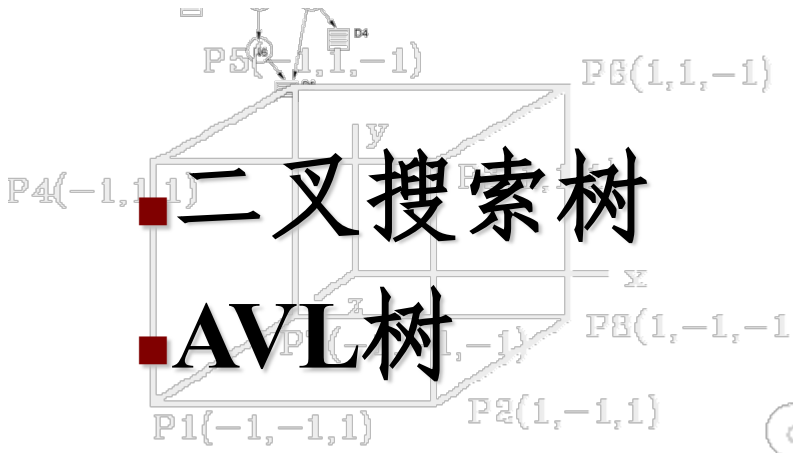
AVL树的高度

- 有 n 个结点的AVL树的高度不超过

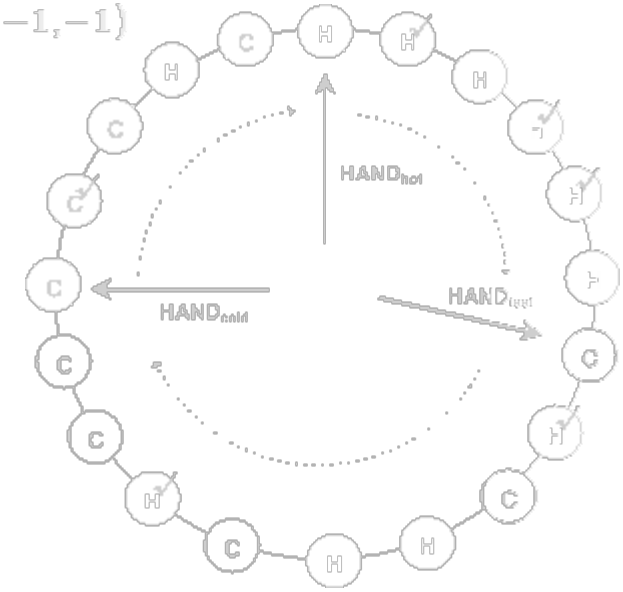
$$1.44*\log_2(n+2)-1.328$$

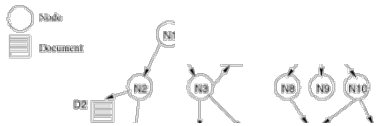
- 在AVL树删除一个结点并做平衡化旋转所需时间为 $O(\log_2 n)$。
- 二叉搜索树适合于组织在内存中的较小的索引(或目录)。对于存放在外存中的较大的文件系统，用二叉搜索树来组织索引不太合适。
- 在文件检索系统中大量使用的是用B-树或B+树做文件索引。

Chapter 1　两个搜索树



- 二叉搜索树
- AVL树

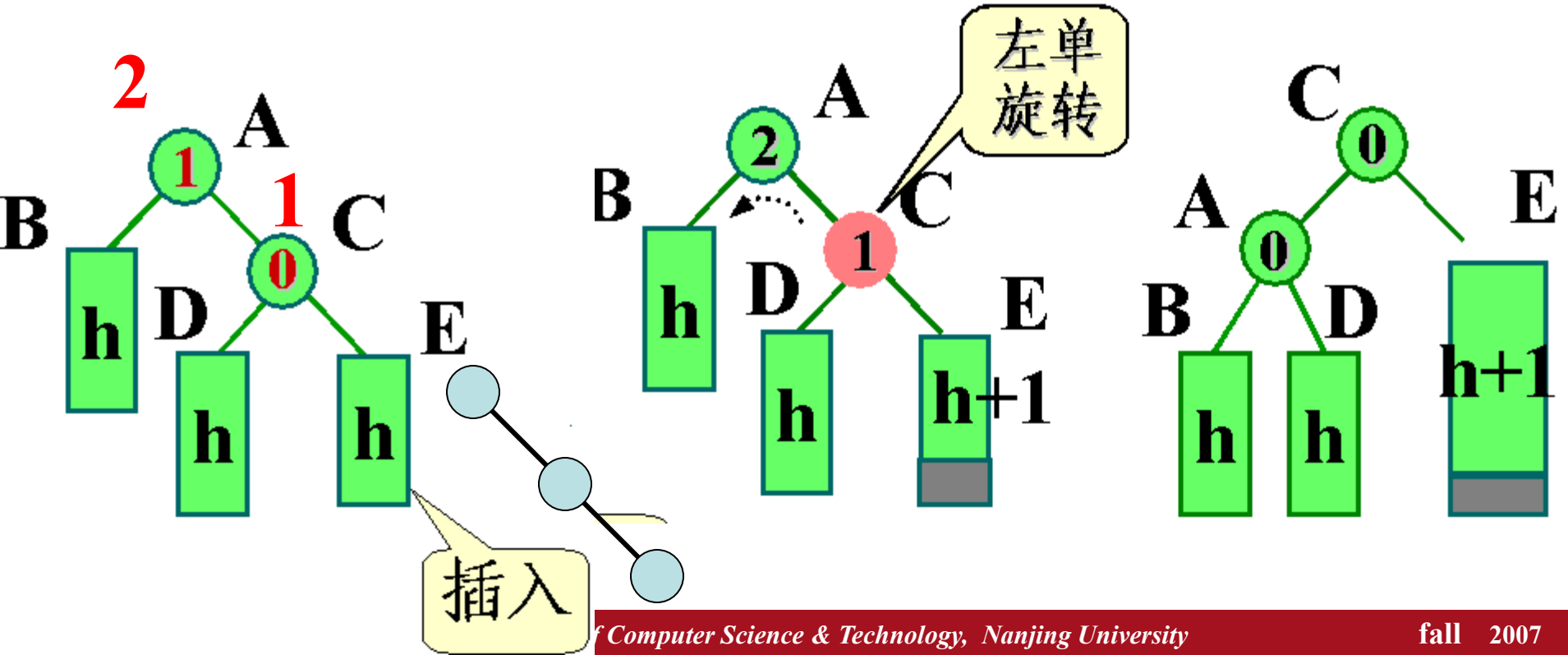




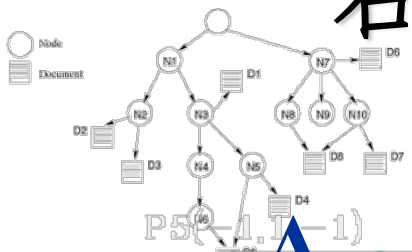
AvI树的平衡化旋转

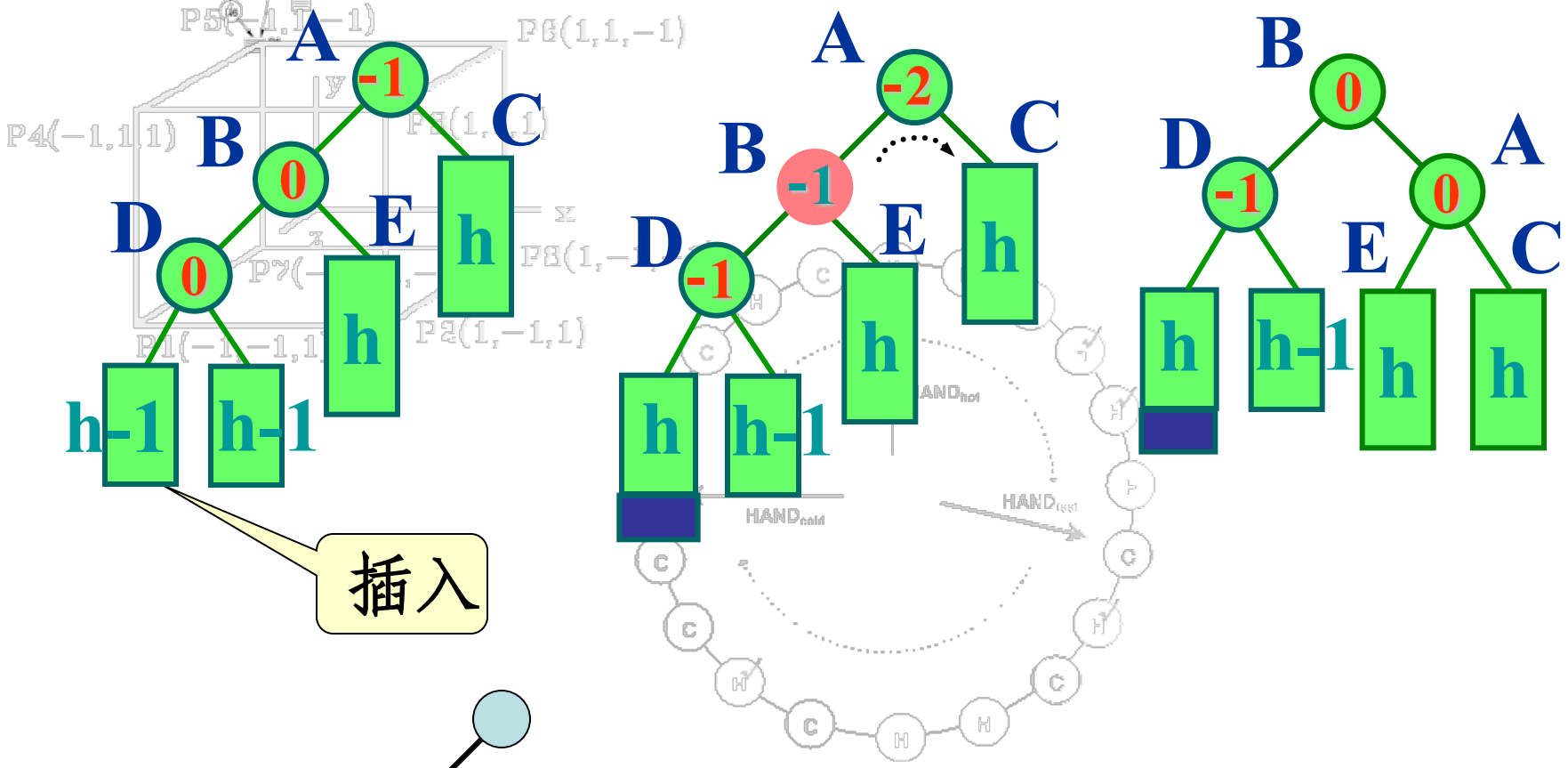
- 有两类:
  - 单旋转 (左旋和右旋)
  - 双旋转 (左平衡和右平衡)
- 每插入一个新结点时, AVL 树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。

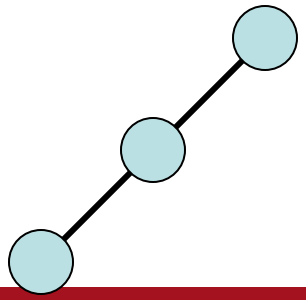
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。



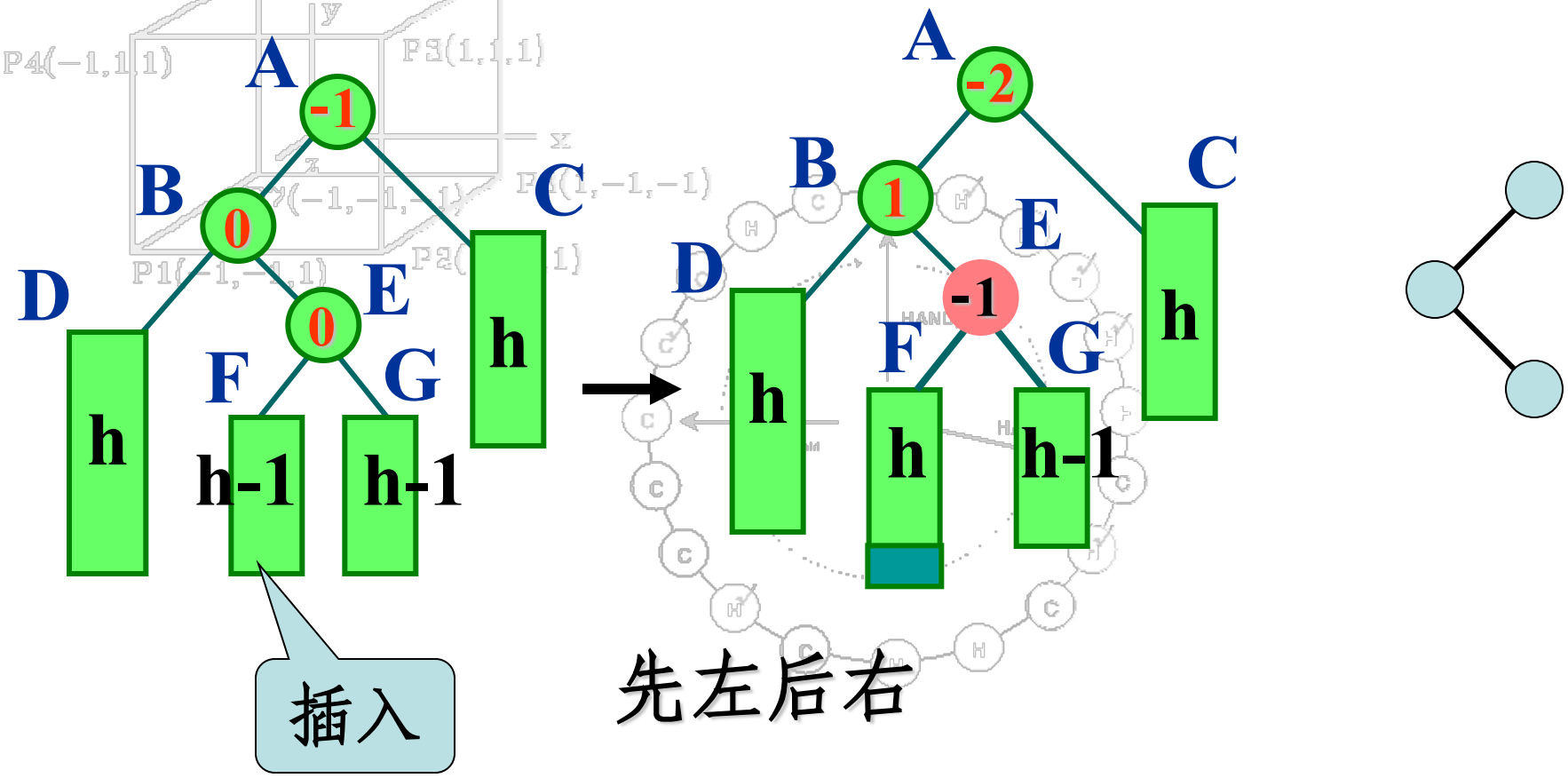
右单旋转 (RotateRight)

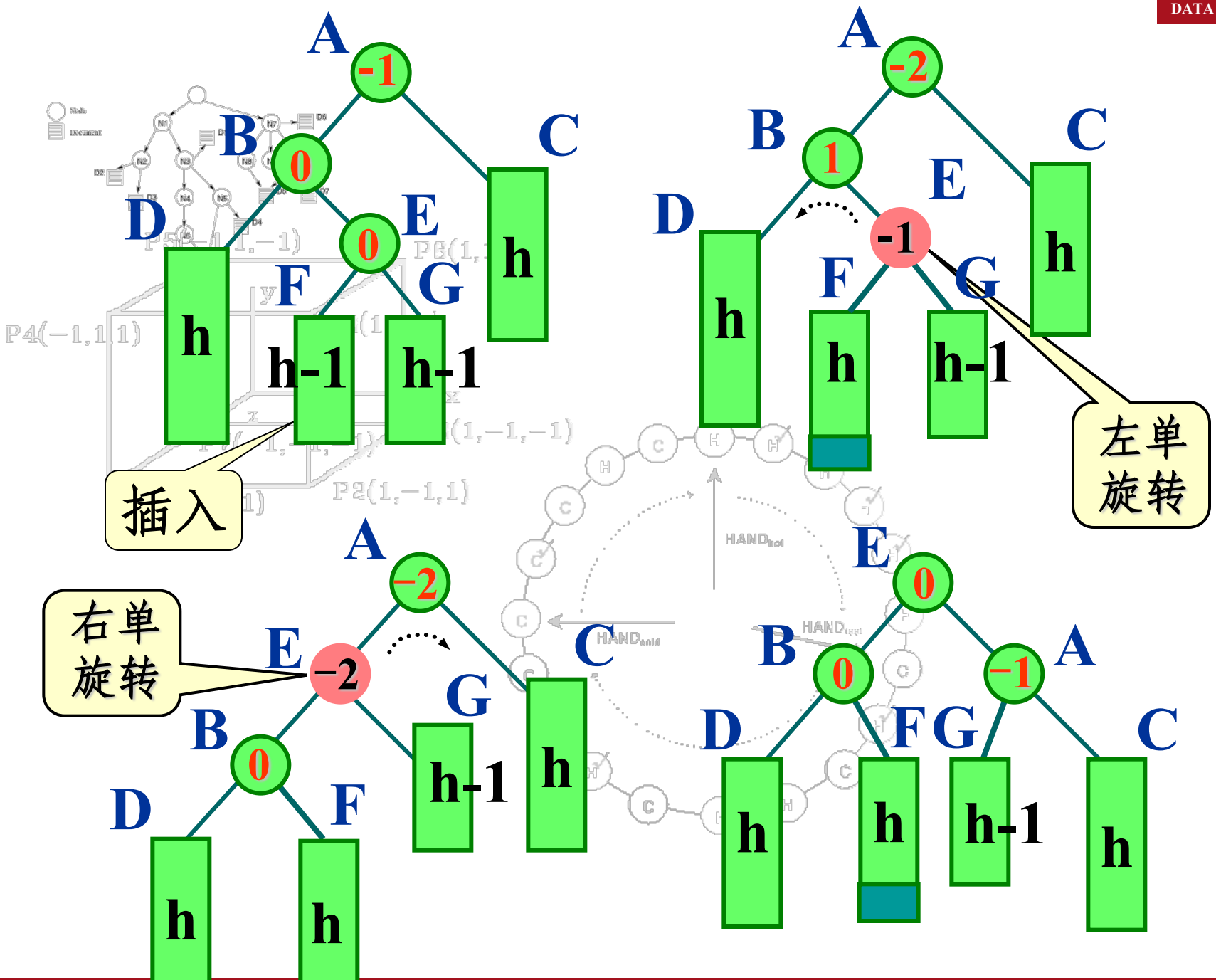




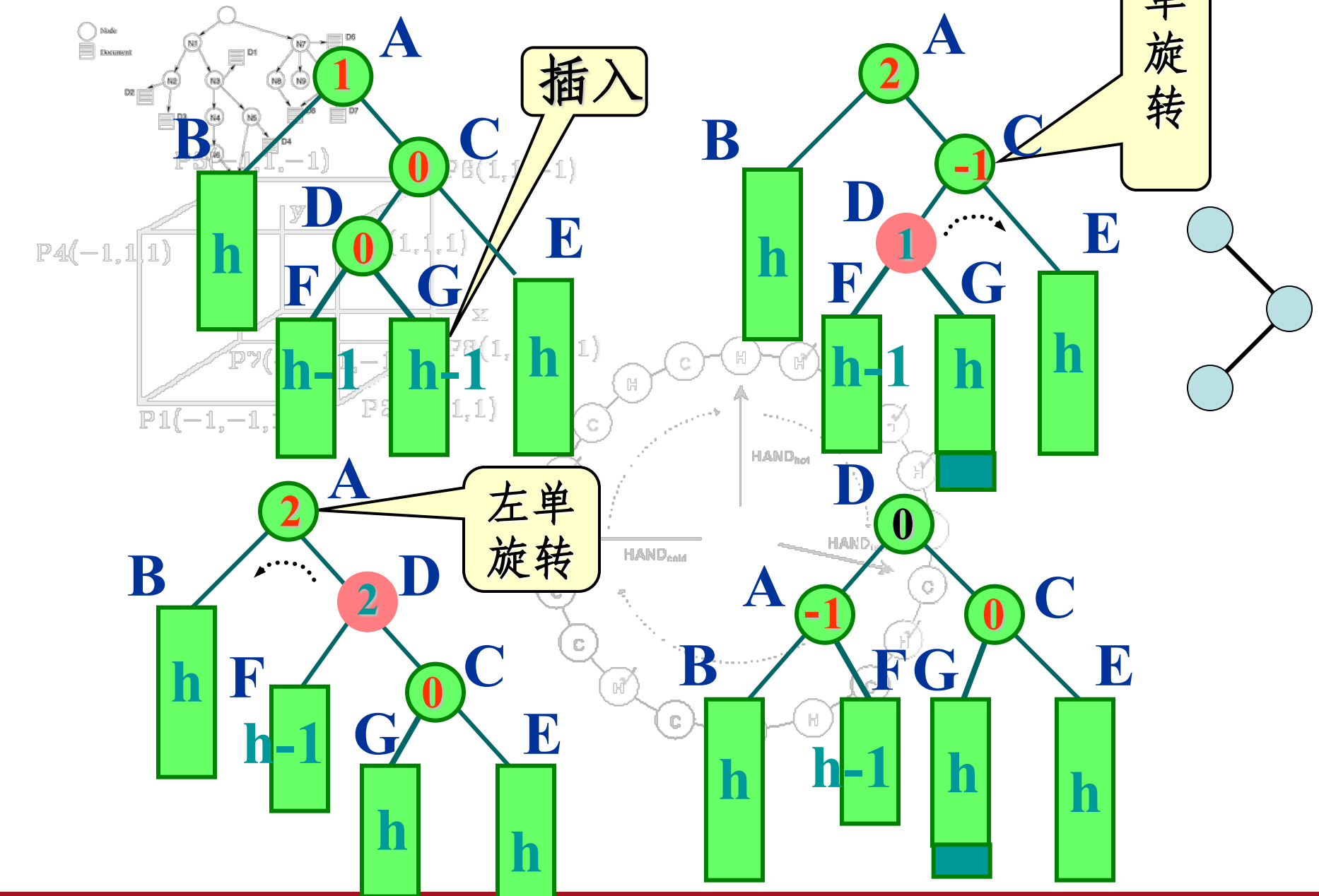


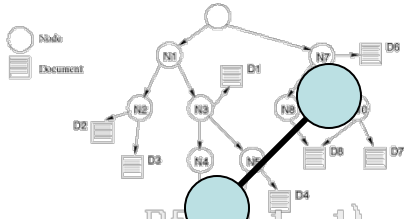
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。

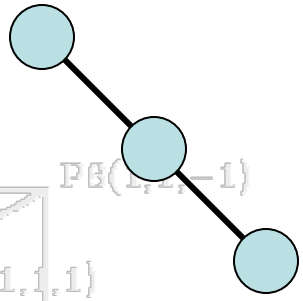



A
-1
B
0
C
D
0
E
h
F
G
h
h-1
h-1
插入
A
-2
B
1
C
E
D
-1
h
F
G
h
h
h-1
左单旋转
右单旋转
A
-2
E
-2
C
B
0
G
D
F
h-1
h
h
h
E
0
B
0
A
-1
D
F
G
C
h
h
h-1
h

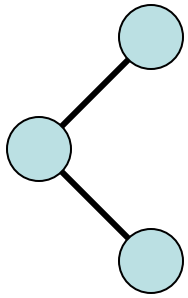
先右后左


A
1
B
h
0
C
D
0
E
F
G
h-1
h-1
h
插入
A
2
B
h
-1
C
D
1
E
F
G
h-1
h
h
右单旋转
A
2
B
h
D
2
F
h-1
C
0
G
E
h
h
左单旋转
D
0
A
-1
B
h
h-1
0
C
F
G
h
h
E
h

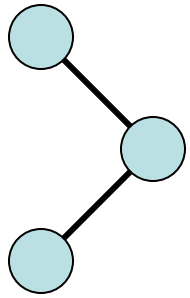




右单旋转

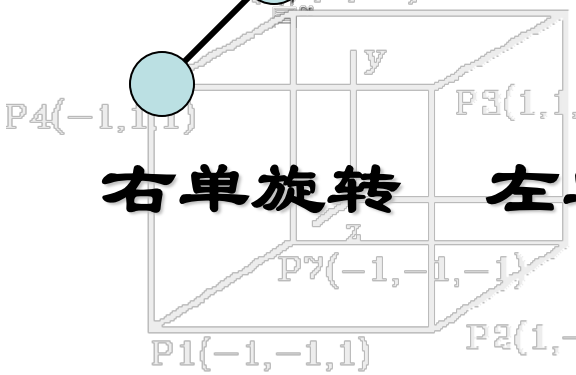


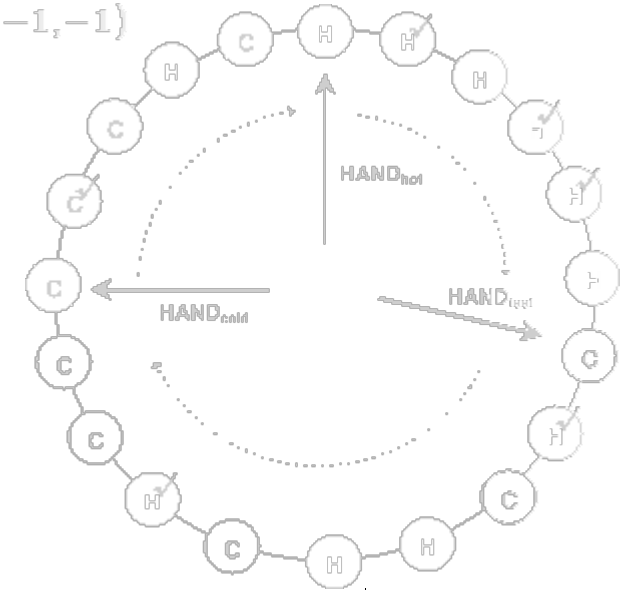
左单旋转



左右双旋转

右左双旋转

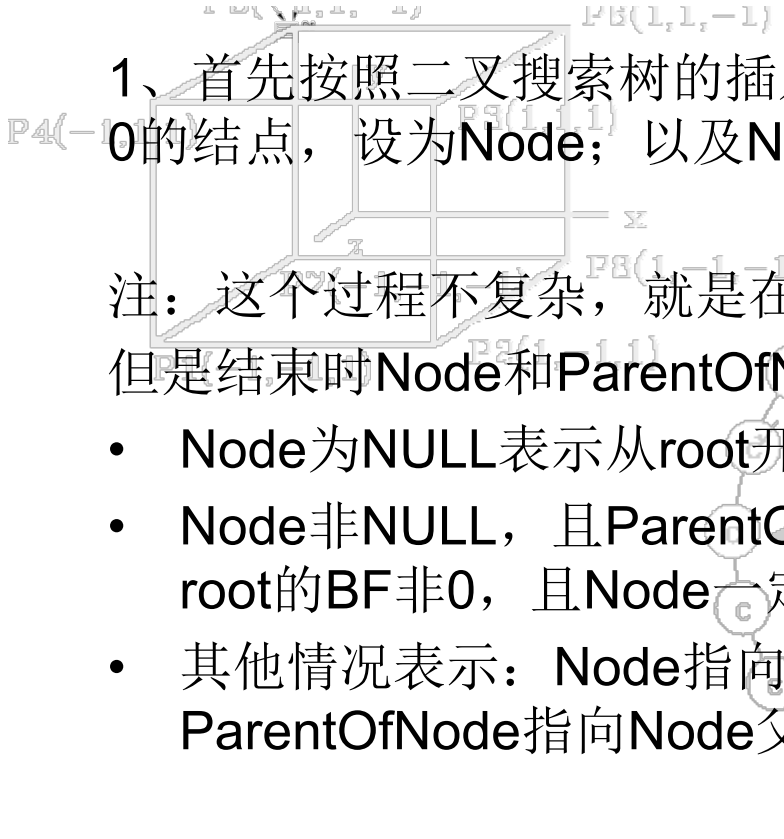




AVL树的插入（无栈）-原理

- 在AVL树中插入一个新结点时
  - 插入过程中最多只有一次旋转（单旋或双旋）
  - 如果一个结点的平衡因子不等于0，那么在它的子树中插入新结点后，（可能经过旋转之后）树的高度不会变化。
  - 当且仅当从某个结点开始的插入路径上的平衡因子都是0，以这个结点为根的子树的高度才会增加，并且不需要旋转，只需要修改平衡因子。
- 因此，插入路径上，（从叶子结点开始）最后一个需要修改平衡因子或选择的结点是第一个平衡因子非0的结点，也就是从根结点开始最后一个平衡因子非0的结点。
- 修改结点Node的平衡因子的规则：
  - Node->BalanceFactor += (x < Node->Data ? -1 : +1)

AVL树的插入（无栈）-算法Step1



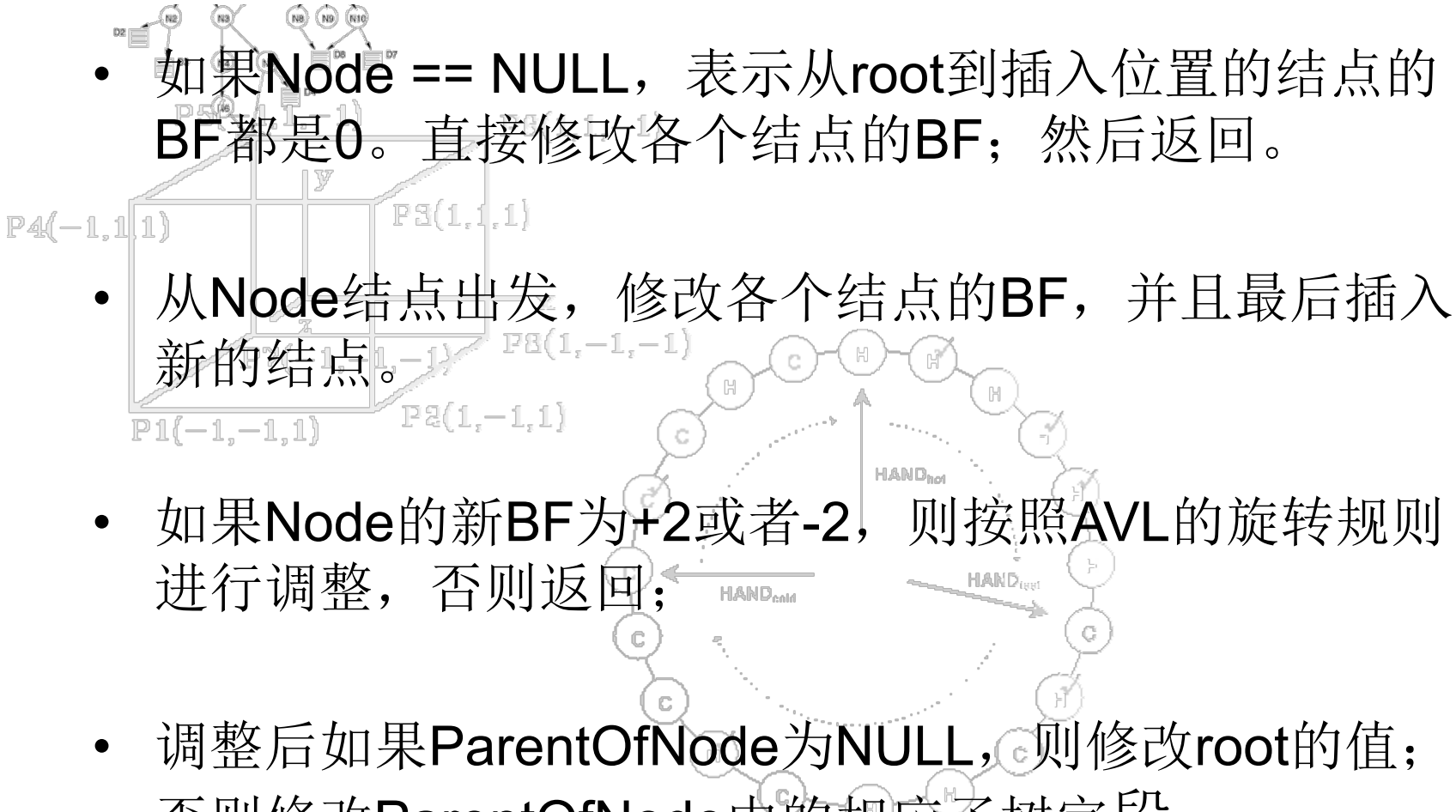
1、首先按照二叉搜索树的插入过程，确定搜索路径中最后一个BF非0的结点，设为Node；以及Node的父亲节点，设为ParentOfNode

注：这个过程不复杂，就是在遍历过程中记录cur和parent的过程。但是结束时Node和ParentOfNode都可能是NULL

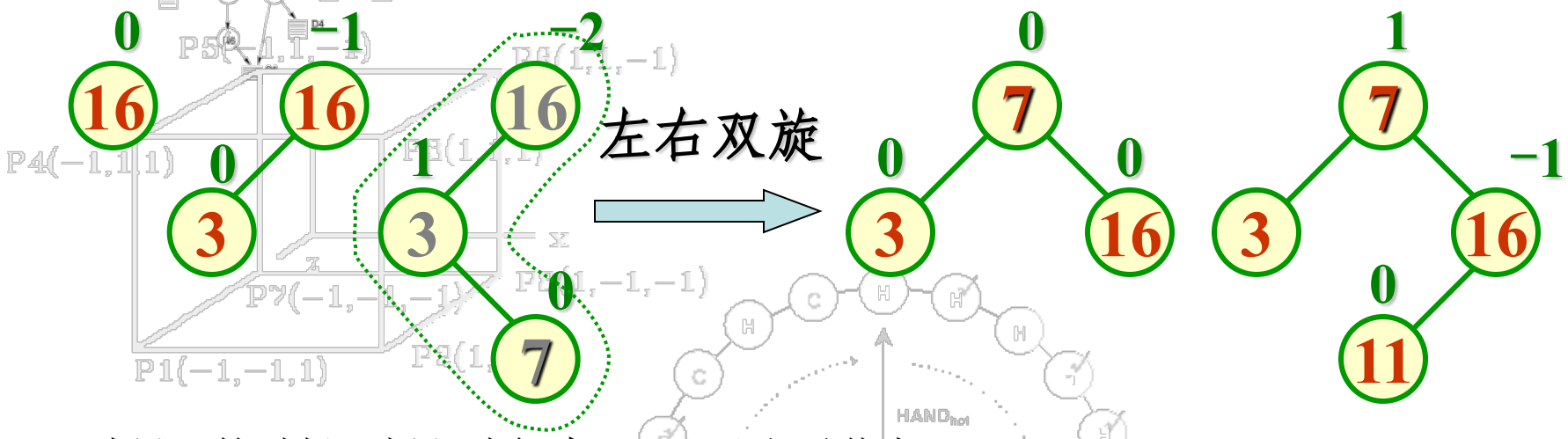
- Node为NULL表示从root开始的搜索路径上的结点的BF都是0；
- Node非NULL，且ParentOfNode为NULL，表示搜索路径上只有root的BF非0，且Node一定等于root。
- 其他情况表示：Node指向路径中最后一个BF非0的结点，而ParentOfNode指向Node父亲结点。

AVL树的插入（无栈）-算法Step2

- 如果Node == NULL，表示从root到插入位置的结点的BF都是0。直接修改各个结点的BF；然后返回。
- 从Node结点出发，修改各个结点的BF，并且最后插入新的结点。
- 如果Node的新BF为+2或者-2，则按照AVL的旋转规则进行调整，否则返回；
- 调整后如果ParentOfNode为NULL，则修改root的值；否则修改ParentOfNode中的相应子树字段。



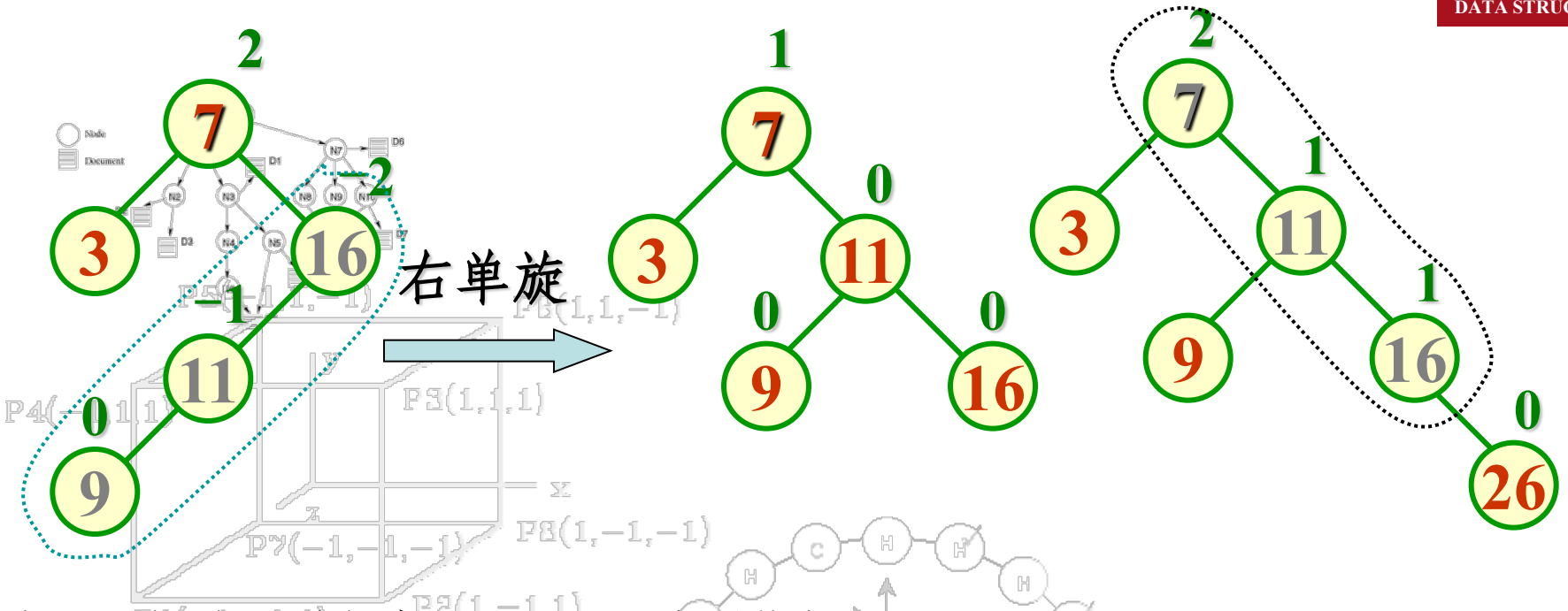
例，输入关键码序列为{16, 3, 7, 11, 9, 26, 18, 14, 15}，插入和调整过程如下。



插入7的时候，插入路径为 16->3->右子节点
从根节点开始，最后一个BF非0的结点为根结点16。

首先从这个结点开始，更新插入路径各个结点的平衡因子（BF）；
然后旋转即得到新的AVL树。

插入11的时候，插入路径为 7→16→左子节点
路径上所有的结点的BF都是0.
因此从根结点开始，更新插入路径上各个结点的平衡因子（BF）
不需要旋转。

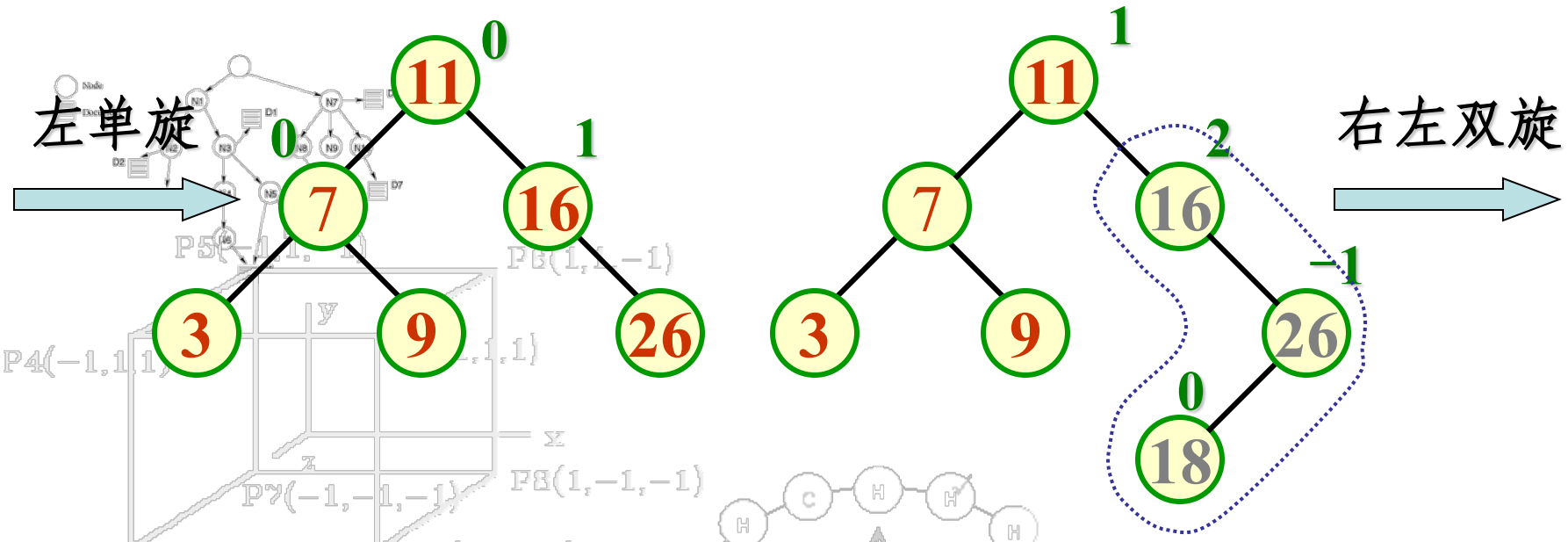


插入9时，插入路径为7→16→11→左子节点，
最后一个BF非0的结点为16（见前一页）。
修改BF后可以得到上图最左边的图。

因为16的BF变为-2，需要进行转置，得到中间的图。（注意此时16的父节点的右子节点改变了）

插入26时，插入路径为7→11→16→右子节点
最后一个BF非0的结点为7（即根结点）
在路径上修改BF后，得到上图最右边的图。

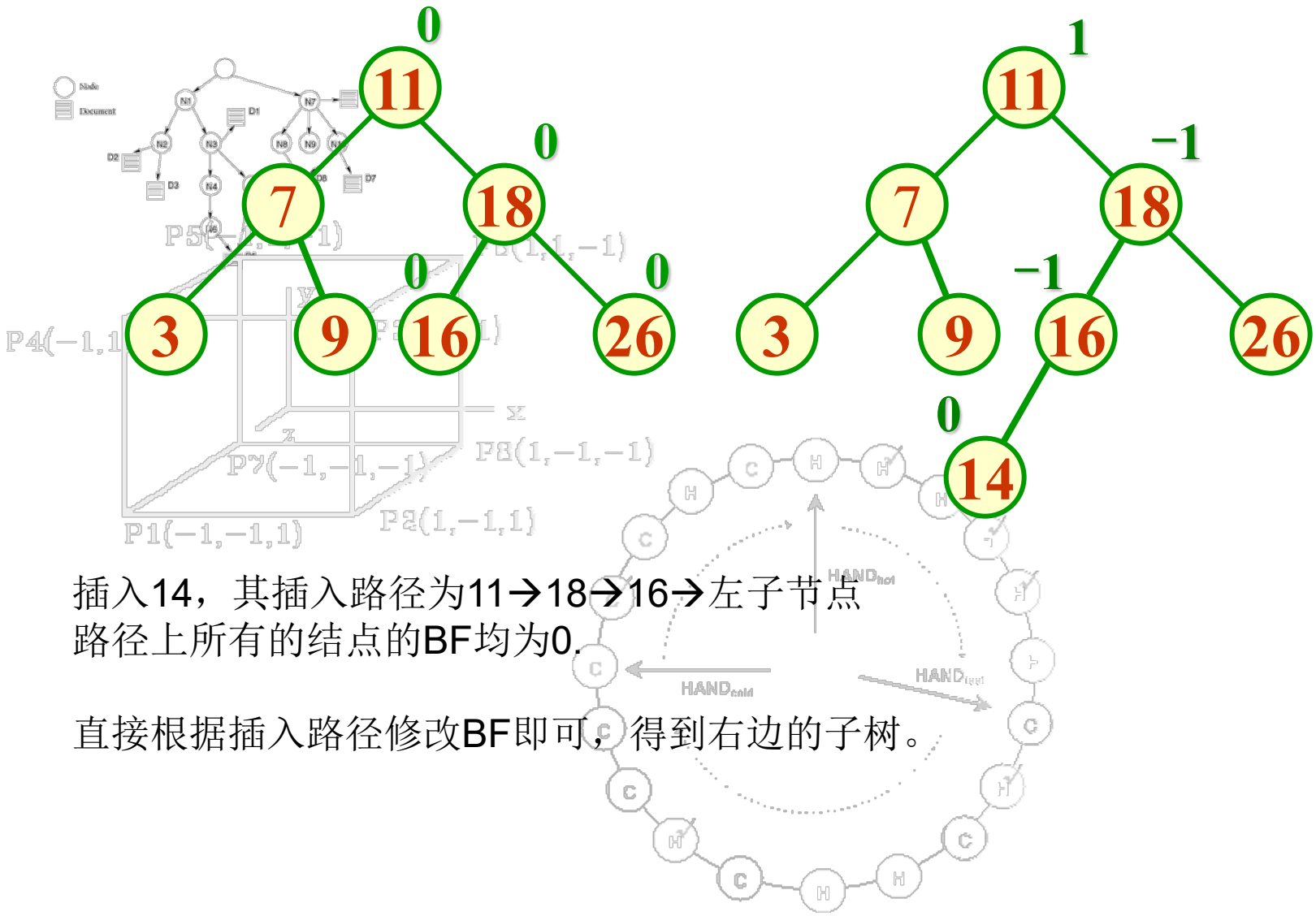
因为7的BF变为+2，所以需要进行旋转，得到下一页的图。



插入18，其插入路径为11→16→26→左子节点
最后一个BF非0的结点为16.

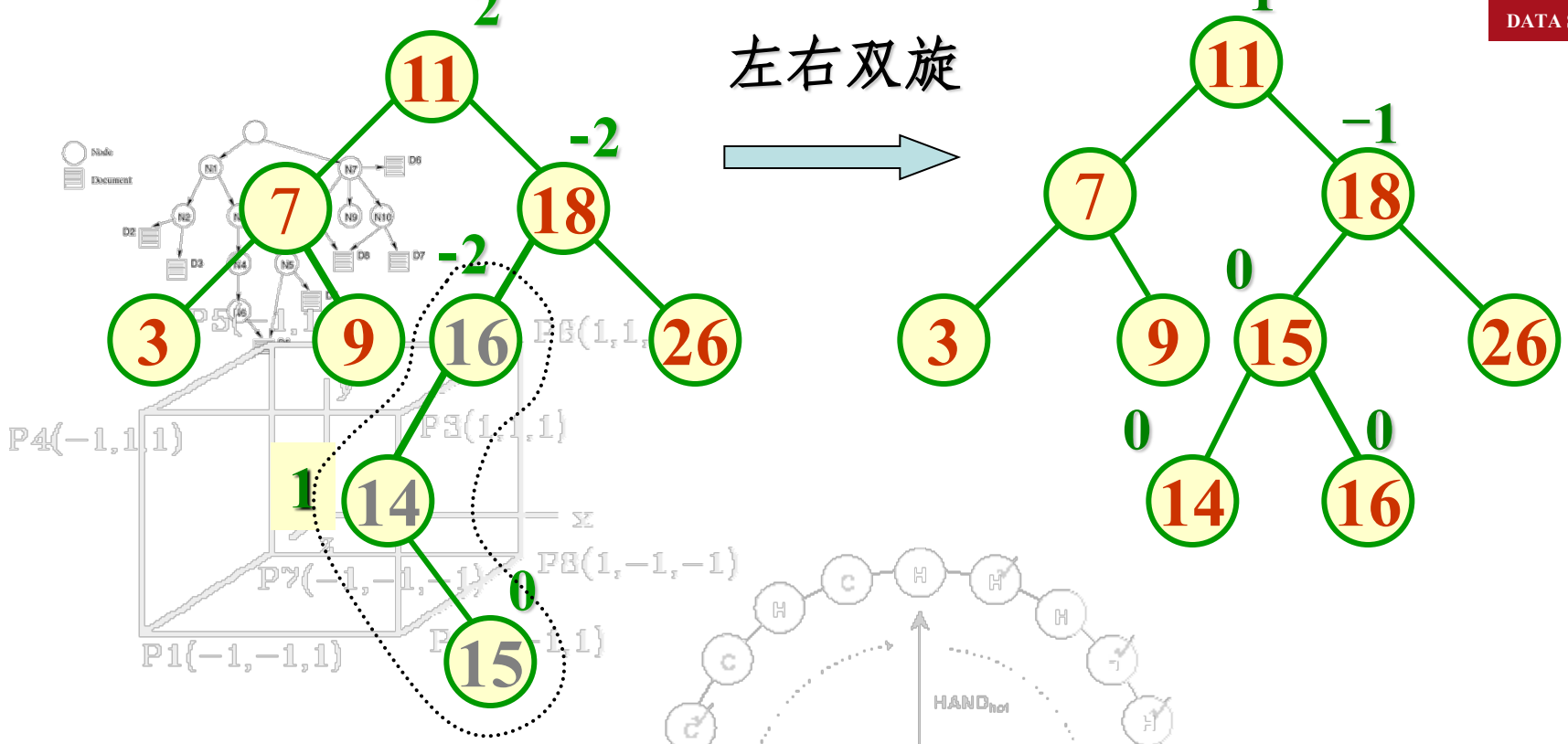
从16开始修改BF后，16上的BF改变为+1，需要进行旋转。

旋转后得到下一页的AVL树。（注意16的父亲结点的右子结点改变了）



插入14，其插入路径为11→18→16→左子节点
路径上所有的结点的BF均为0.

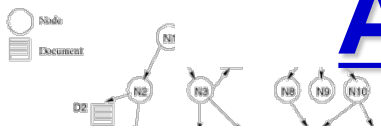
直接根据插入路径修改BF即可，得到右边的子树。



插入15时，插入路径为11→18→16→14→右子节点
最后一个非0结点为16。
按照插入路径修改BF后，得到如上左图所示。

因为16的BF变为-2，进行旋转即可得到右边的AVL树。

AvI树的删除（递归）一



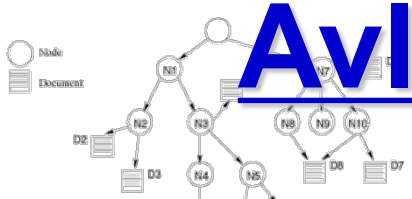
- 因为在AVL树中删除时，如果x所在结点有两个子树，就需要删除其中序后继（或前驱），所以我们在递归函数中加一个boolean类型参数，表示要寻找x，还是寻找其后继进行删除。

- Delete(root, x, bFindNext)
  - bFindNext为true时，寻找的是右子树的最左子节点，而不是x。

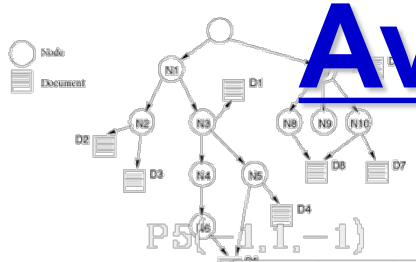
AvI树的删除（递归）二

```
static Node * theNode; //静态变量，用于记住x所在结点
Delete(root, x, bFindNext, int &shorter)
{       if(root == NULL)        return NULL;
        if(bFindNext)
        {       if(root->left == NULL)
                {       theNode->Data = root->Data;
                        shorter = 1; return root->right;    } //没有考虑删除结点
                shorter = 0;
                root->left = Delete(root, x, bFindNext, shorter);
                根据shorter值和root的BalanceFactor进行旋转，并设置shorter值
                return 新的根结点;
        }
        if(x < root->data) //如果x小于根结点的data
        {       root->left = Delete(root->left, x, bFindNext, shorter);
                根据shorter值和root的BalanceFactor进行旋转，并设置shorter值
                return 新的根结点;
        }
        …       //见下一页
}
```

AvI树的删除（递归）三



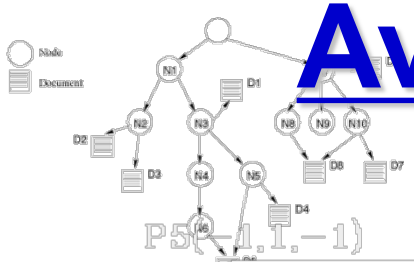
```
static Node * theNode; //静态变量，用于记住x所在结点
Delete(root, x, bFindNext, int &shorter)
{
        …           //接上一页
        else if(x > root->data)
        {           和上一个分支类似处理，对称                    }
        else //x == root->data
        {           if(root->left == NULL || root->right == NULL)
                    {       shorter == 1;
                            return root->left == NULL ? root->right : root->left;
                    }
                    theNode = root;     //这里要记住x所在的结点，以便在结构删除是进行移
动
                    root->right = Delete(root->right, x, true, shorter);
                    根据shorter值和root的BalanceFactor进行旋转，并设置shorter值;
                    return 新的根结点
        }

}
```



AvI树的删除（无栈）
基本思想　三

- 首先，仍然在删除路径中寻找最靠近叶子的不会变矮的子树的根结点NoShorterNode，结构上删除一个结点后，从NoShorterNode开始沿着删除路径开始进行调整。
- 规则要比插入时复杂一些：
  - 结构上被删除的结点，即X所在结点或者X的中序后继结点，一定会变矮。
  - 当Node的左子树变矮时，以Node为根的子树变矮的条件
    - Node->BalanceFactor == -1；//左子树本来较高

    或者
    - Node->BalanceFactor == +1 且 Node->right->BalanceFactor != 0
  - 当Node的右子树变矮时，条件类似（对称即可）
- 调整的过程要复杂一些：需要进行多次调整，且在概念上需要先调整子树，然后再调整父节点（但是我们没有栈）
  - 考虑转化成为尾递归!



AvI树的删除（无栈）基本思想 三

- 从NoShorterNode开始进行删除时，其子节点必然会变矮，可以逐步转化成为尾递归
- 简化后的递归过程：

```
//注意这里已经确定x是可以被删除结点的Key了；
Delete(root, x)
{   if(root应被删除)
        return      root->left == NULL
                  ?     root->right
                  :     root->left;
    if(应该在左子节点删除)
    {    root->left = Delete(root->left, x); //root->left必然变矮！
         进行调整;
         返回新的根结点.
     }
     else //应该删除右子节点
     {    root->right = Delete(root->right, x); //root->left必然变矮
          进行调整;
           返回新的根结点;
     }
}
```

但是，这个仍然不是尾递归！

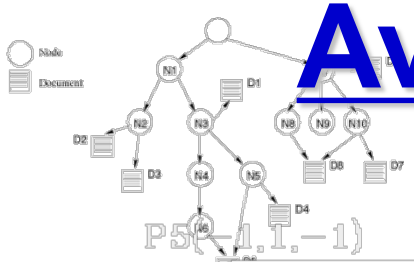
注意：

1、进行旋转调整的时候，不会改变较矮子树（也就是递归返回的子树）的内部结构

2、被删除的子树必然变矮

3、调整后，删除后的root->left/root->right应该存放在什么地方是确定的。

因此，可以考虑先调整，后递归删除子节点，就变成了尾递归！

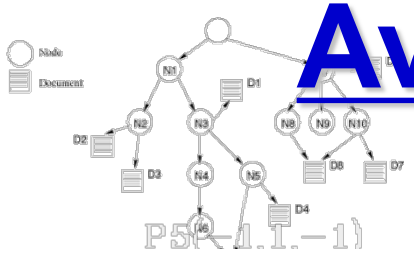


AvI树的删除（无栈）

基本思想　三

```
//toBeDelNode是之前已经确定的，需要在结构上被删除的结点。
//NewRoot是引用参数，用于存放本次Delete之后的新的根结点
Delete(root, x, Node * &NewRoot, toBeDelNode)
{   if(root == toBeDelNode)
            NewRoot =   root->left == NULL
                      ?     root->right
                      :     root->left;
    if(应该在左子节点删除)
    {    在假设root->left变矮的前提下，进行进行调整; //具体的方法见下一页。
         *palceOfOrigLeft = 调整前root->left的新位置;
         NewRoot = 调整后的根结点;
         Delete(*palceOfOrigLeft, x, *palceOfOrigLeft);
    }
     else //应该删除右子节点
     {
         和上面的类似
     }
}
```

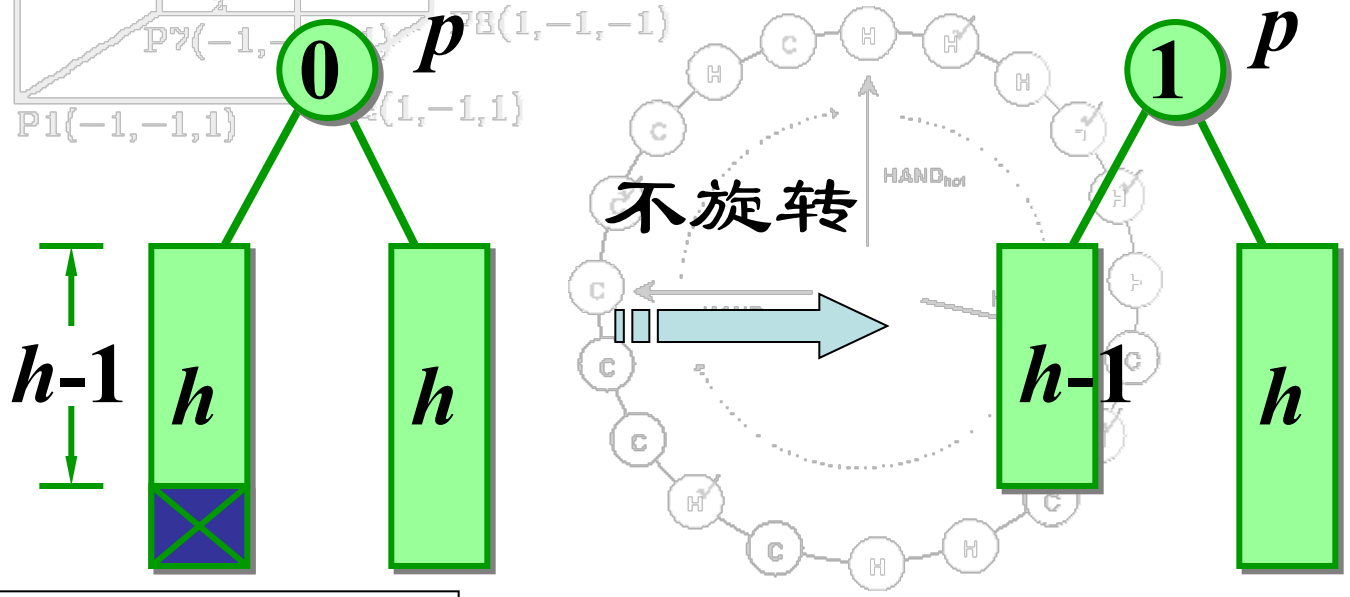
这是尾递归，可以转化成为循环



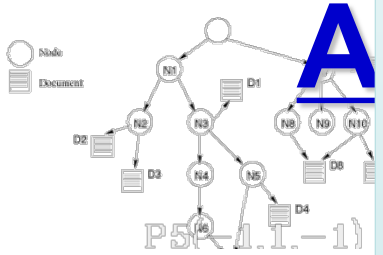
AVl树的删除（无栈）基本思想　四

- case 1

```
NewRoot = root;
root->BalanceFactor ++;
palceOfOrigLeft = &root->left;
```



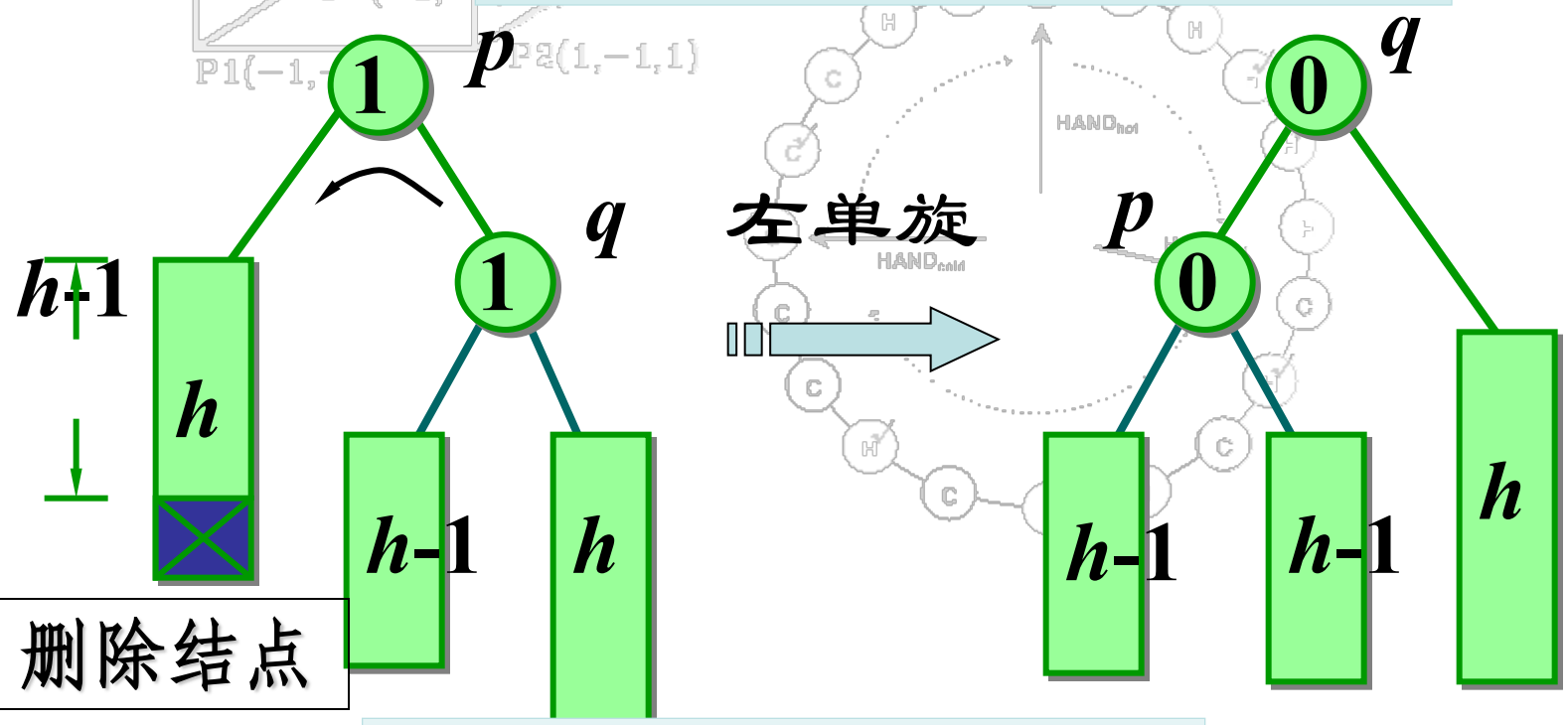
删除一个结点，
高度降低一层



A

- **case 3b：如果** 行一个单旋转
  来恢复平衡，结 rter 为 True。

```
定位：
p = root;
q = root->right;
nLeftRight = q->left;

重组：
NewRoot = q;          //调整后的根结点
q->left = p;
p->right = nLeftRight;
p->BF = q->BF = 0;
//原来的root->left的位置在root->left->left
palceOfOrigLeft = &root->left->left
```



删除结点

case 3a的处理类似

AVL树的删

五

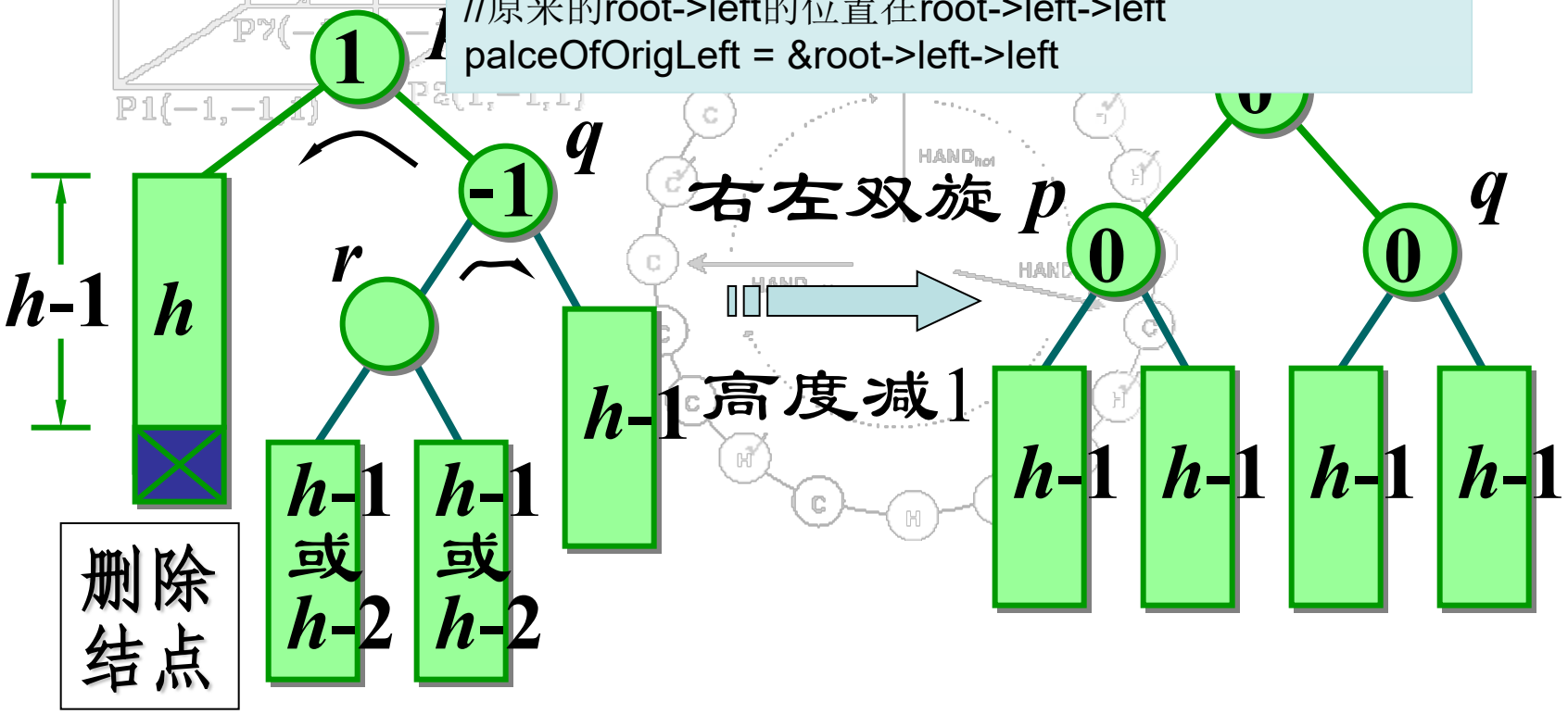
- case 3c

定位：

```
p = root;  q = root->right; r = q->left;
nLR = r->left;   nRL = r->right;
```

重组：

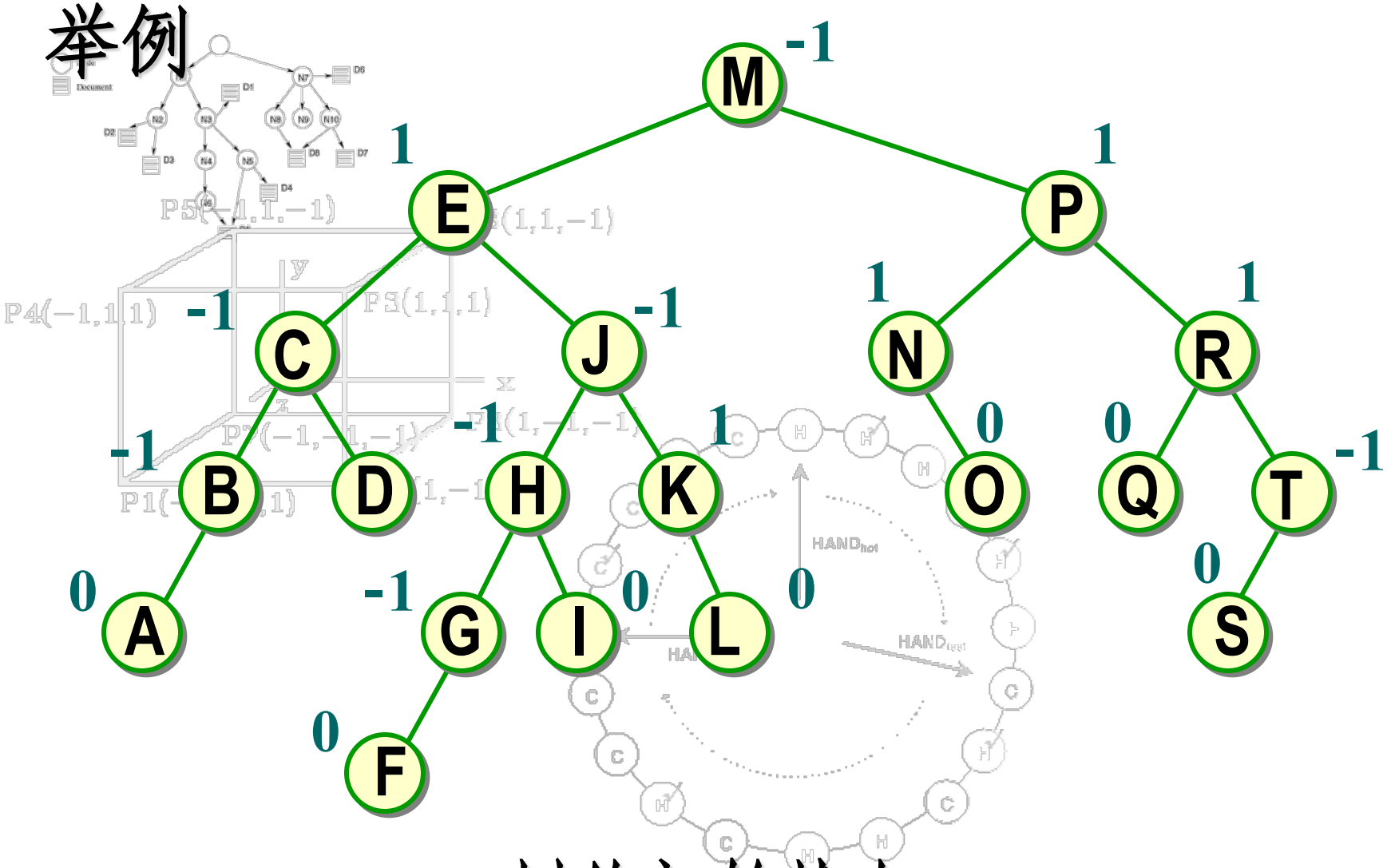
```
NewRoot = r;          //调整后的根结点
r->left = p; r->right = q;
q->right = nLR;       q->left = nRL;
p->BF = r->BF == +1 ? -1 : 0;
q->BF = r->BF == -1 ? +1 : 0;
r->BF = 0;

//原来的root->left的位置在root->left->left
palceOfOrigLeft = &root->left->left
```



AvI树的删除（无栈）算法整体

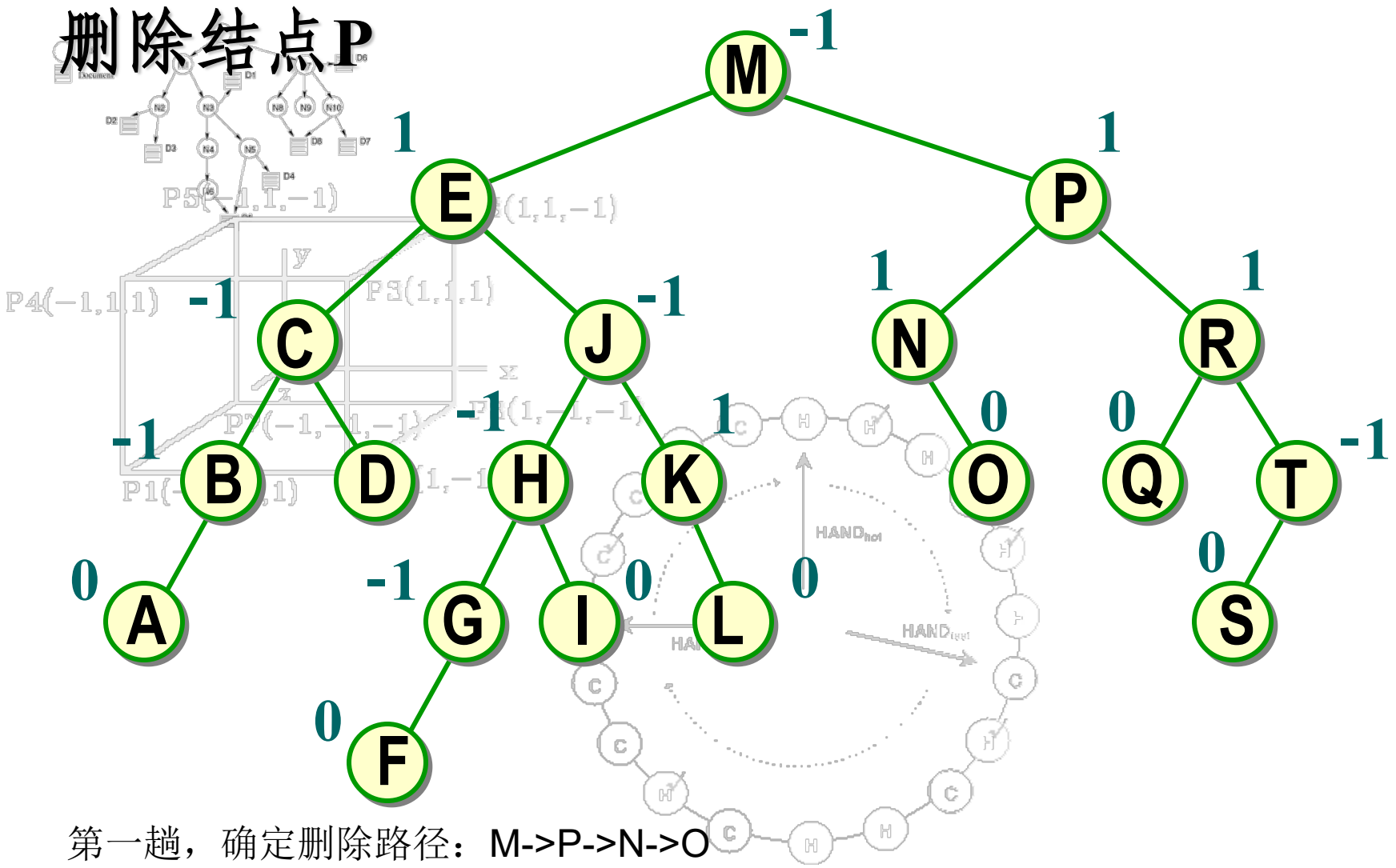
- 第一步：
  - 从根结点开始，沿着删除路径遍历。寻找
    - 最后一个不会变矮的结点lastNonShorterNode（可能为NULL，此时整棵树都会变矮，以及该结点的父节点theParent（可能为NULL）
    - 确定X所在的结点theNode，
    - 将在结构上被删除的结点toBeDelNode
  - 注意：删除路径向左走还是向右走，要考虑多一点。
- 第二步：
  - 从lastNonShorterNode开始（如果为NULL，则从root开始）,按照上面的方式进行调整和删除。必要时需要把toBeDelNode中的Data拷贝到theNode中

举例



树的初始状态

删除结点P

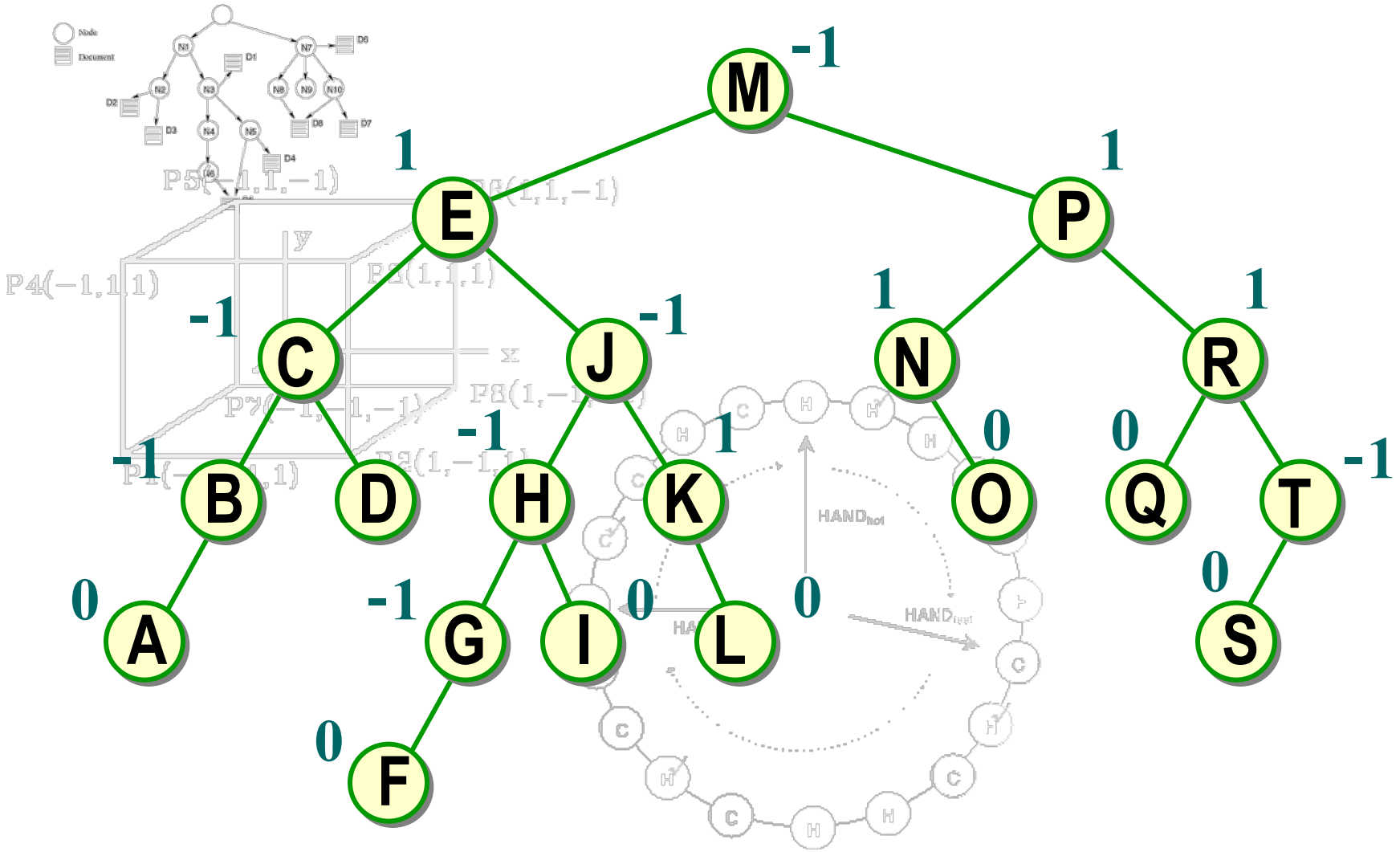


第一趟，确定删除路径：M->P->N->O

1、lastNonShorterNode: NULL,　parent:　NULL

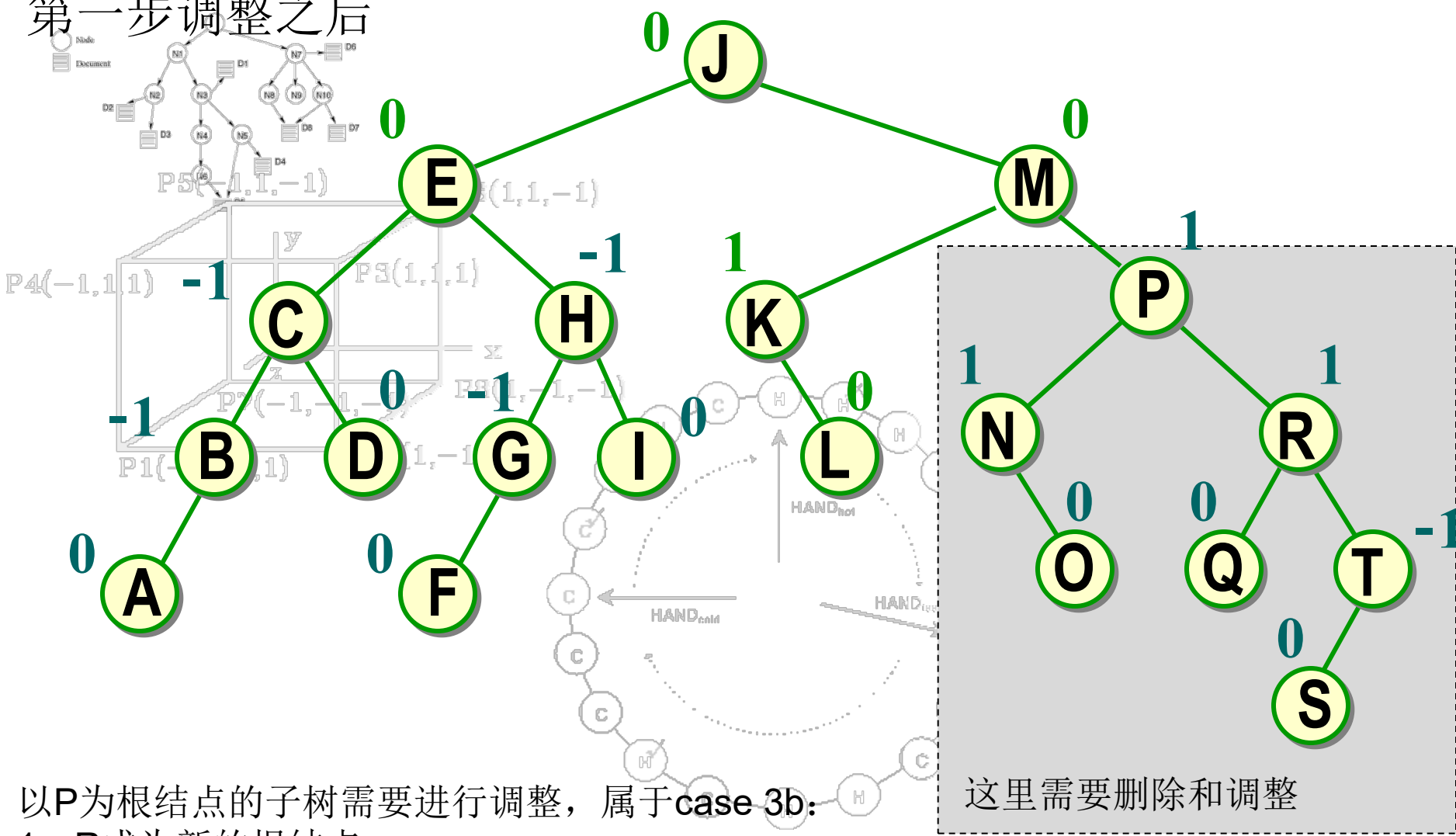
2、theNode:　P　　toBeDelNode:　O

开始调整和删除，属于case 3c的对称形式



J会成为根结点，E,M分别是J的左右子节点，…
而P为根结点的子树会进一步调整后成为M的右子节点!

第一步调整之后

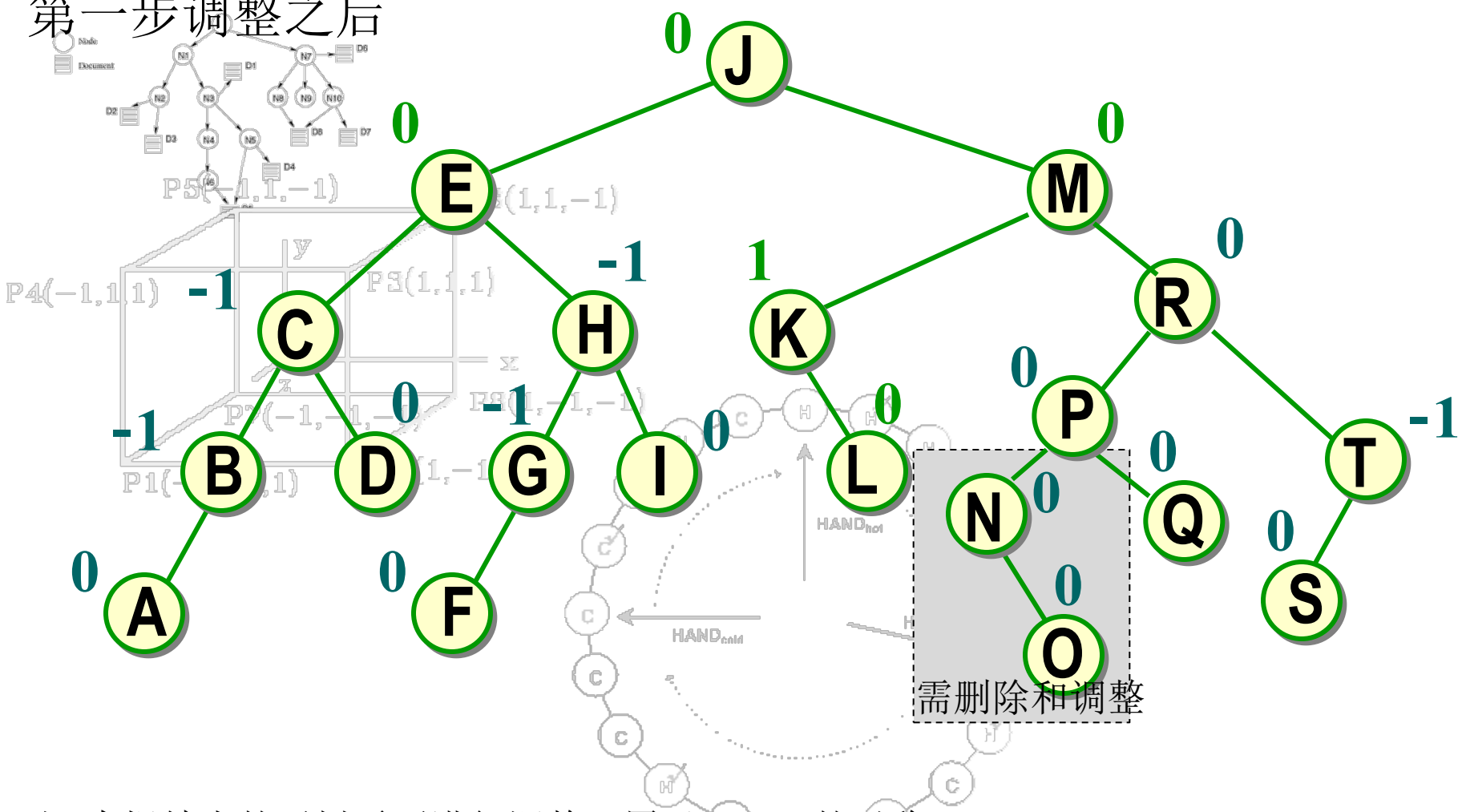


以P为根结点的子树需要进行调整，属于case 3b：

1、R成为新的根结点

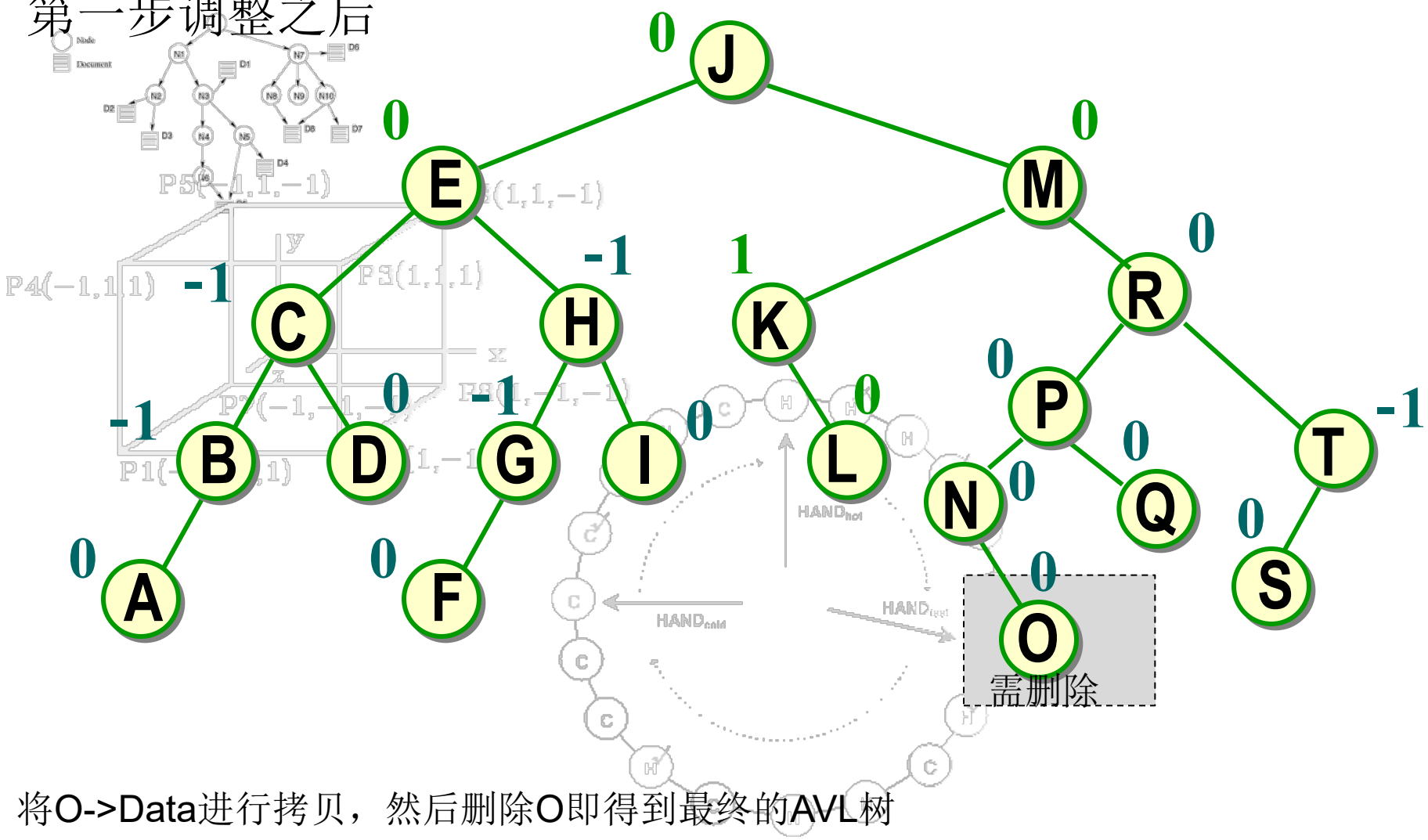
2、R的左右子树分别是P和T

第一步调整之后



以N为根结点的子树需要进行调整，属于case 1的对称情况，即较长的子树被删除后变短的情况。将N的BalanceFactor减去1。以O为根的子树要进一步删除和调整。

第一步调整之后



将O->Data进行拷贝，然后删除O即得到最终的AVL树。

最终结果