Chapter7 搜索结构

7.1 静态搜索表

7.1.1 搜索的概念

所谓搜索，就是**在数据集合中寻找满足某种条件的数据对象。**

搜索的结果通常有两种可能：

（1）搜索成功，即找到满足条件的数据对象。这时，作为结果，可报告该对象在结构中 的位置, 还可给出该对象中的具体信息。

（2）搜索不成功，或搜索失败。作为结果，应报告一些信息, 如失败标志、位置等。

通常称**用于搜索的数据集合为搜索结构**，它是由同一数据类型的对象(或记录)组成。

在每个对象中有若干属性，其中有一个属性，**其值可唯一地标识这个对象**。称为**关键码**。使用基于关键码的搜索，搜索结果应是唯一的。但在实际应用时，搜索条件是多方面的，可以使用基于属性的搜索方法，但搜索结果可能不唯一。

实施搜索时的环境。

静态环境，搜索（存储）结构在插入和删除等操作的前后不发生改变。——静态搜索表

动态环境，为保持较高的搜索效率, 搜索（存储）结构在执行插入和删除等操作的前后将自动进行调整，结构可能发生变化

7.1.2 静态搜索表

在静态搜索表中，数据元素存放于数组中，**利用数组元素的下标作为数据元素的存放地址**。搜索算法根据给定值 k，在数组中进行搜索。直到找到 k 在数组中的存放位置或可确定在数组中找不到 k 为止。

```
class dataNode{
private:
     K key;                           //关键码域
     E other;                         //其他域（视问题而定）
}

template <class E, class K >
class dataList {                     //数据表类定义
protected:
     dataNode<E, K> *Element;        //数据表存储数组
     int ArraySize, CurrentSize;     //数组最大长度和当前长度
};

```

7.1.3 顺序搜索（Sequential Search)

一般的顺序搜索算法在第二章已经讨论过，本章介绍一种使用“监视哨”的顺序搜索方法。

设在数据表 dataList 中顺序搜索关键码与 给定值 x 相等的数据元素，要求数据元素在表中从下标 0 开始存放, 下标为 CurrentSize 的元素作为控制搜索过程自动结束的“监视哨”使用。

若搜索成功，则函数返回该元素在表中序号 Location（比下标大 1）, 若搜索失败，则函数返回 CurrentSize+1。

```
template <class E, class K>
int dataList<E, K>::SeqSearch (const K x) const {
     Element[CurrentSize].key = x;
     int i = 0;                       //将x设置为监视哨
     while (Element[i].key != x) i++;   //从前向后顺序搜索
     return i+1;
};
```

节省了判断 i < currentsize 的开销

在等概率情形，$p_i = 1/n, i = 1, 2, \ldots, n$。搜索成功的平均搜索程度为：

$$ASL_{succ} = \sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

在搜索不成功情形，$ASL_{unsucc} = n+1$。

7.1.4 顺序搜索的递归算法

v采用递归方法搜索值为 x 的元素，每递归一层就向待查元素逼近一个位置，直到到达该元素。假设待查元素在第 i（1 ≤ i ≤ n）个位置，则算法递归深度达 i（1 ~ i）。

```
if (loc > CurrentSize) return 0;                //搜索失败
else if (Element[loc-1].key == x) return loc;   //搜索成功
else return SeqSearch (x, loc+1);               //递归搜索
```

7.1.5 基于有序顺序表的折半搜索

设n个对象存放在一个有序顺序表中，并按其关键码从小到大排好了序。

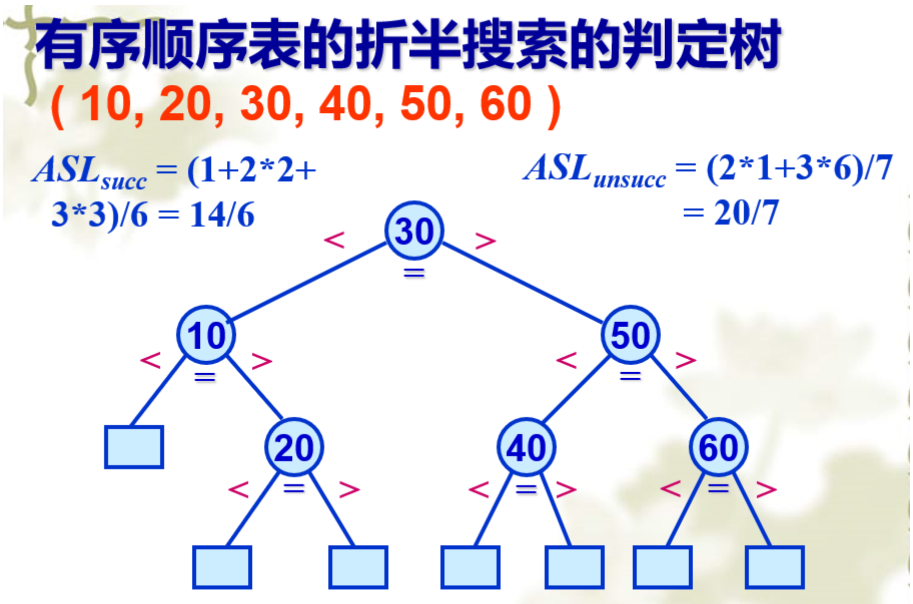
折半搜索时, 先求位于搜索区间正中的对象的下标mid，用其关键码与给定值x比较:

Element[mid].key == x，搜索成功；

Element[mid].key > x，把搜索区间缩小到表的前半部分，继续折半搜索；

Element[mid].key < x，把搜索区间缩小到表的后半部分，继续折半搜索。

如果搜索区间已缩小到一个对象，仍未找到想要搜索的对象，则搜索失败。



7.2 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（如果非空）上所有结点的关键码都**小于**根结点的关键码。

右子树（如果非空）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。

注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键码不一定小于路径上的结点的关键码。

```
class BST{
private:
    BSTNode<E, K> *root; //根指针
    K RefValue;          //输入停止标志
}
```

7.2.1 二叉搜索树的搜索算法

在二叉搜索树上进行搜索，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。

假设想要在二叉搜索树中搜索关键码为 x 的元素，搜索过程从根结点开始。

如果根指针为NULL，则搜索不成功；否则用给定值 x 与根结点的关键码进行比较:

若给定值等于根结点关键码，则搜索成功，返回搜索成功信息并报告搜索到结点地址

```
Search (const K x, BSTNode<E, K> *ptr) {
    if (ptr == NULL) return NULL;
    else if (x < ptr->data) return Search(x, ptr->left);
    else if (x > ptr->data) return Search(x, ptr->right);
    else return ptr;
};
```

搜索过程是从根结点开始，沿某条路径自上而下逐层比较判等的过程。

搜索成功，搜索指针将停留在树上某个结点；搜索不成功，搜索指针将走到树上某个结点的空子树。

设树的高度为h，最多比较次数不超过h

7.2.2 二叉搜索树的插入算法

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。

如果搜索成功，说明树中已经有这个元素，不再插入；

如果搜索不成功，说明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。

```
template <class E, class K>
bool BST<E, K>::Insert (const E& e1, BSTNode<E, K> *& ptr) {
    if (ptr == NULL) {                    //新结点作为叶结点插入
        ptr = new BstNode<E, K>(e1);      //创建新结点
        if (ptr == NULL)
            { cerr << "Out of space" << endl;  exit(1); }
        return true;
    }
    else if (e1 < ptr->data) Insert (e1, ptr->left);     //左子树插入
    else if (e1 > ptr->data) Insert (e1, ptr->right);    //右子树插入
    else return false;                                   //x已在树中,不再插入
};
```

7.2.3 二叉搜索树的删除算法

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。

（1）**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。

（2）**被删结点右子树为空**，可以拿它的左子女结点顶替它的位置，再释放它。

（3）**被删结点左子树为空**，可以拿它的右子女结点顶替它的位置，再释放它。

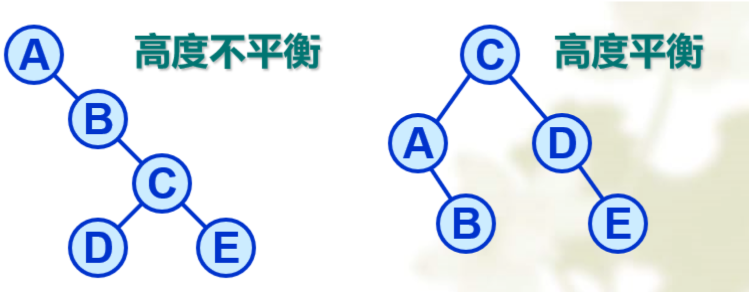
（4）**被删结点左、右子树都不为空**，可以在它的右子树中寻找中序下的第一个结点(关键码最小),用它的值填补到被删结点中，再来处理这个结点的删除问题**（递归处理）**。

```
bool BST<E, K>::Remove (const K x, BstNode<E, K> *& ptr) {
    BstNode<E, K> *temp;
    if (ptr != NULL) {
        if (x < ptr->data) Remove (x, ptr->left);       //在左子树中执行删除
        else if (x > ptr->data) Remove (x, ptr->right); //在右子树中执行删除
        else if (ptr->left != NULL && ptr->right != NULL)
        { //ptr指示关键码为x的结点，它有两个子女
             temp = ptr->right;       //到右子树搜寻中序下第一个结点
             while (temp->left != NULL)
                 temp = temp->left;
             ptr->data = temp->data; //用该结点数据代替根结点数据
           Remove (ptr->data, ptr->right);//递归删除右子树中节点
        }
        else {  //ptr指示关键码为x的结点有一个/无子女
            temp = ptr;
            if (ptr->left == NULL) ptr = ptr->right;
            else ptr = ptr->left;
            delete temp;
            return true;
        }
    }
    return false;
};
```

7.3 AVL树（二叉平衡树）

7.3.1 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



结点的平衡因子 bf (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bf。

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($log_2 n$)

```
struct AVLNode : public BSTNode<E, K> {
    int bf;
    AVLNode() { left = NULL; right = NULL; bf = 0; }
    AVLNode (E d, AVLNode<E, K> *l = NULL, AVLNode<E, K> *r = NULL)
    { data = d;  left = l;  right = r;  bf = 0; }
};
```

```
template <class E, class K>
class AVLTree : public BST<E, K> {
//平衡的二叉搜索树（AVL）类定义
public:
    AVLTree() { root = NULL; }      //构造函数
    AVLTree (K Ref) { RefValue = Ref; root = NULL; }//构造函数：构造非空AVL树
    int Height() const;                          //高度
    AVLNode<E, K>* Search (K x,
    AVLNode<E, K> *& par) const;                 //搜索
    bool Insert (E& el) { return Insert (root, el); } //插入
    bool Remove (K x, E& el)
         { return Remove (root, x, el); }            //删除
    friend istream& operator >> (istream& in,
         AVLTree<E, K>& Tree);            //重载：输入
    friend ostream& operator << (ostream& out,
         const AVLTree<E, K>& Tree);      //重载：输出
protected:
    int Height (AVLNode<E, K> *ptr) const;
    bool Insert (AVLNode<E, K>*& ptr, E& el);
    bool Remove (AVLNode<E, K>*& ptr, K x, E& el);
    void RotateL (AVLNode<E, K>*& ptr);        //左单旋
    void RotateR (AVLNode<E, K>*& ptr);        //右单旋
    void RotateLR (AVLNode<E, K>*& ptr);       //先左后右双旋
    void RotateRL (AVLNode<E, K>*& ptr);       //先右后左双旋
};
```

7.3.2 平衡化旋转

如果在一棵平衡的二叉搜索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。

平衡化旋转有两类：

（1）单旋转 (左旋和右旋)

（2）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

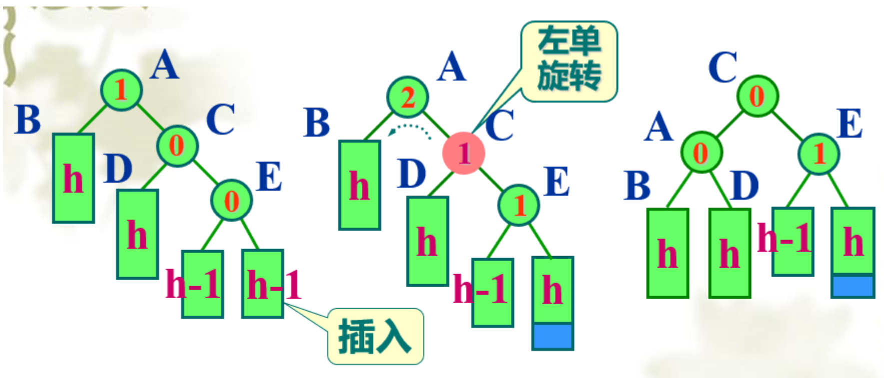
从插入位置沿通向根的路径回溯，检查各结点的平衡因子

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。单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。

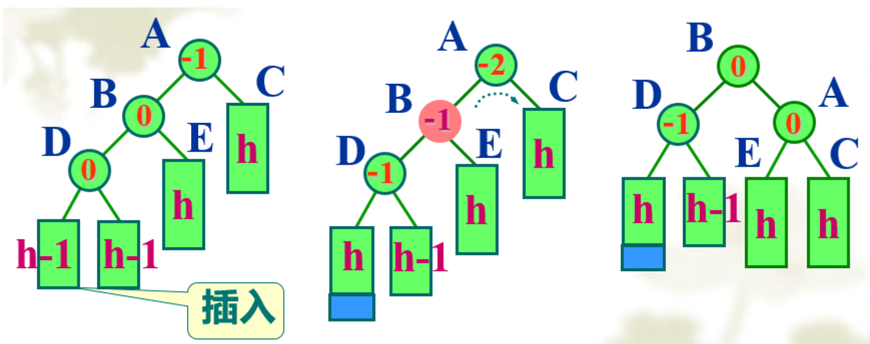
如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。

左单旋转



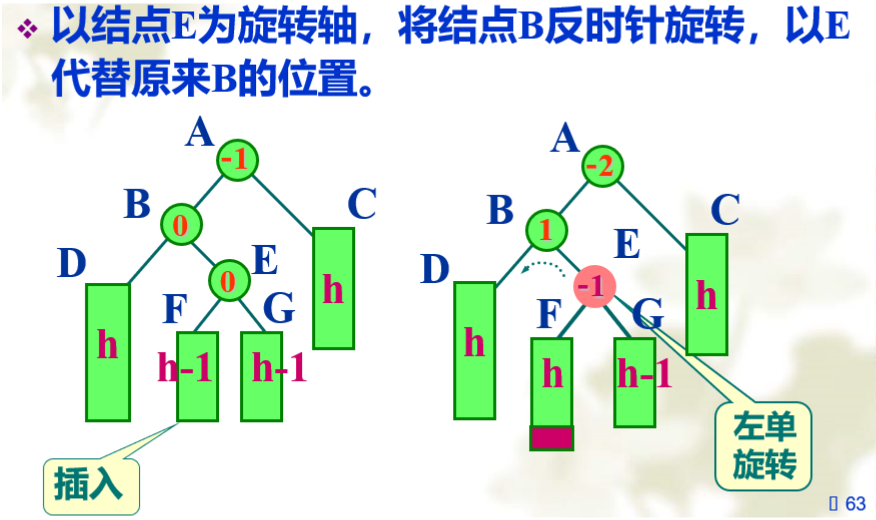

```
RotateL (AVLNode<E, K> *& ptr) {
//右子树比左子树高: 做左单旋转后新根在ptr
    AVLNode<E, K> *subL = ptr;
    ptr = subL->right;
    subL->right = ptr->left;
    ptr->left = subL;
    ptr->bf = subL->bf = 0;
};
```

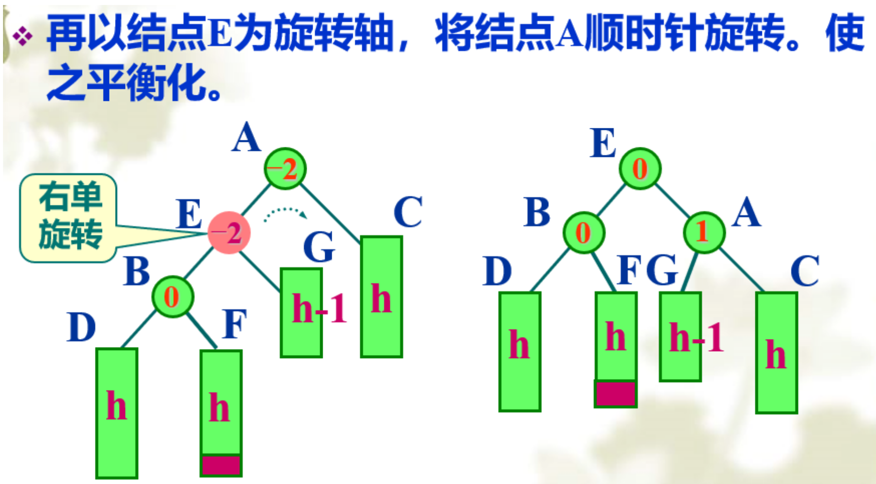
右单旋转



```
RotateR (AVLNode<E, K> *& ptr) {
//左子树比右子树高，旋转后新根在ptr
    AVLNode<E, K> *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)



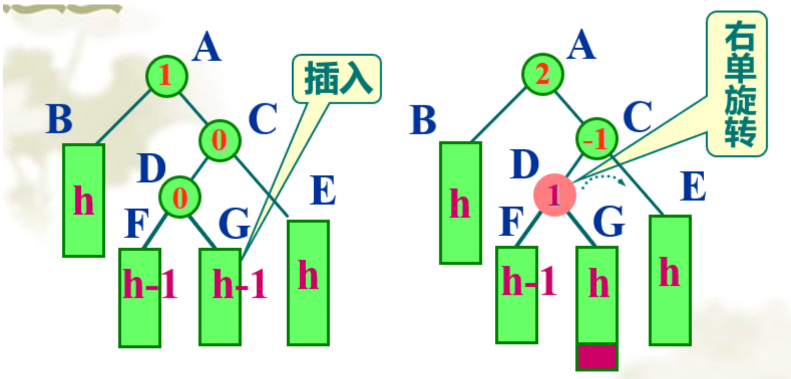



```
void AVLTree<E, K>::RotateLR (AVLNode<E, K> *& ptr) {
    AVLNode<E, K> *subR = ptr;
    AVLNode<E, K> *subL = subR->left;
    ptr = subL->right;
    subL->right = ptr->left;
    ptr->left = subL;
    if (ptr->bf <= 0) subL->bf = 0;
    else subL->bf = -1;
    subR->left = ptr->right;
    ptr->right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

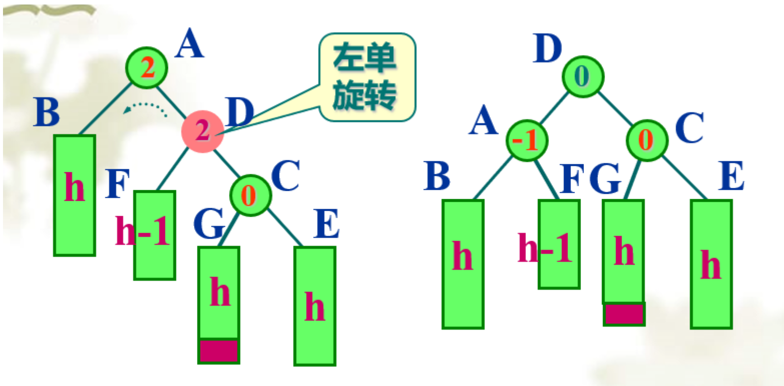
平衡因子计算：用Height()函数，右子树-左子树高度。

先右后左双旋转 (RotationRightLeft)

首先以结点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。



7.3.3 AVL树的插入

旋转是插入的子操作

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 |bf| > 1**，则出现了不平衡，需要做平衡化处理。

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，逐步建立AVL树。

在插入新结点后，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的平衡因子值有三种情况：

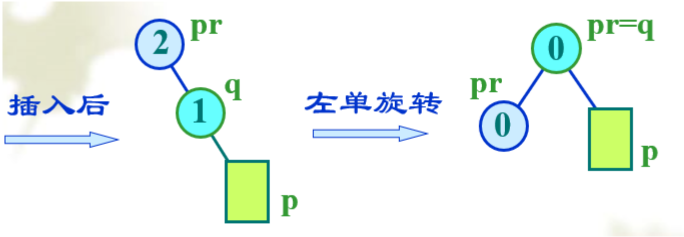
1. 结点pr的平衡因子为0。说明刚才是在pr的较矮的子树上插入了新结点，此时不需做平衡化处理，返回主程序。子树的高度不变。

2. 结点pr的平衡因子的绝对值|bf| = 1。说明插入前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，还需从结点pr向根方向回溯，继续考查结点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。说明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。此时可进一步分2种情况讨论：

- 若结点pr的bf = 2，说明右子树高，结合其右子女q的bf分别处理：

  （1）若q的bf为1，执行左单旋转。



（2）若q的bf为-1，执行先右后左双旋转

- 若结点pr的bf = -2，说明左子树高，结合其左子女q 的bf分别处理：

  （1）若q的bf为-1，执行右单旋转；

  （2）若q的bf为1，执行先左后右双旋转。

看PPT第75-77页实例！

7.3.4 AVL树的删除

1.如果被删结点x最多只有一个子女，可做简单删除:

—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 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。

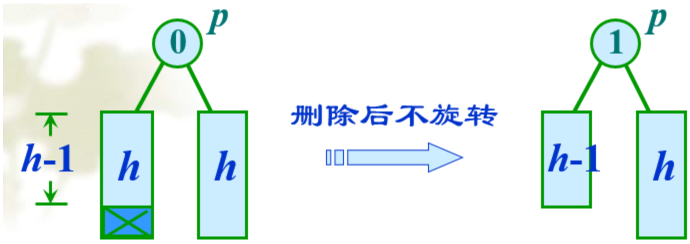
—因为结点 y 最多有一个子女，可以简单地用 1. 给出的方法进行删除。

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。

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

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**

情况1：当前结点 p 的bf为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。

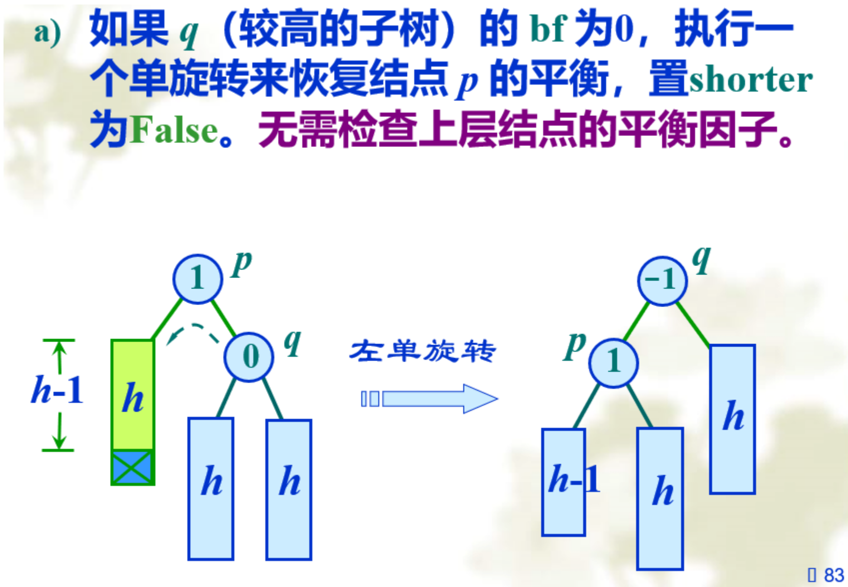


情况2：结点 p 的 bf 不为0且较高的子树被缩短。则 p 的 bf 改为0，同时shorter置为True。

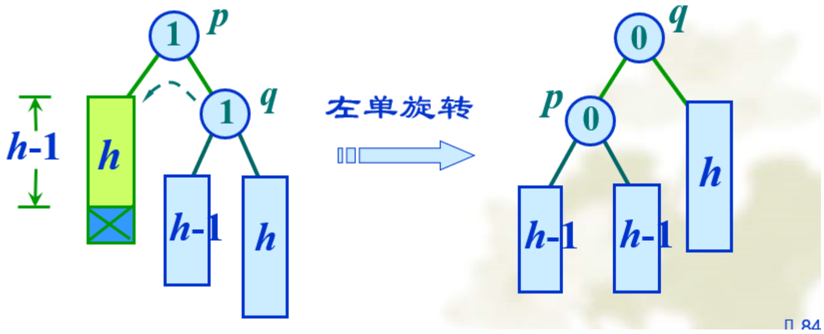
情况3：结点 p 的 bf 不为0，且较矮的子树又被缩短。则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。

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。

旋转的方向取决于是结点 p 的哪一棵子树被缩短。



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，同时置shorter为True。还要继续检查上层结点的平衡因子。



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，同时置shorter为True。