



第七章 搜索结构

- 7.1 静态搜索结构
- 7.2 二叉搜索树
- 7.3 AVL树



第七章 搜索结构

- 7.1 静态搜索结构
- 7.2 二叉搜索树
- 7.3 AVL树



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：
 - 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。
 - 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。



搜索的概念

- 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。



两种搜索表

- 实施搜索时有两种不同的环境。
 - ◆ 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。
— 动态搜索表



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。



数据表与搜索表的类定义

```
#include <iostream.h>
#include <assert.h>
const int defaultSize = 100;
template <class E, class K>
class dataList; //数据表类的前视定义

template <class E, class K >
class dataNode { //数据表中结点类的定义
friend class dataList<E, K>;
//声明其友元类为dataList

public:
```



```
dataNode (const K x) : key(x) {}      //构造函数
K getKey() const { return key; }      //读取关键码
void setKey (K x) { key = x; }        //修改关键码

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
public:
```



```
dataList (int sz = defaultSize)
: ArraySize(sz), CuurentSize(0) {
    Element = new dataNode<E, K>[sz];
    assert (Element != NULL);
}

dataList (dataList<E, K>& R); //复制构造函数
virtual ~dataList() { delete []Element; } //析构函数
virtual int Length() { return CurrentSize; } //求表的长度
virtual K getKey (int i) const {
    //提取第 i ( 1开始 ) 元素值
```



```
assert (i > 0 || i <= CurrentSize);
return Element[i-1].key;
}

virtual void setKey (K x, int i) {
    //修改第 i ( 1开始 ) 元素值
    assert (i > 0 || i <= CurrentSize);
    Element[i-1].key = x;
}

virtual int SeqSearch (const K x) const;      //搜索
virtual bool Insert (E& e1);                  //插入
virtual bool Remove (K x, E& e1);            //删除
```



```
friend ostream& operator << (ostream& out,
    const dataList<E, K>& OutList);          //输出
friend istream& operator >> (istream& in,
    dataList<E, K>& InList);                  //输入
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
};
```



```
template <class E, class K >
bool dataList<E, K>::Insert (E& e1) {
//在dataList的尾部插入新元素, 若插入失败函数返
//回false, 否则返回true.
    if (CurrentSize == ArraySize) return false;
    Element[CurrentSize] = e1;           //插入在尾端
    CurrentSize++;
    return true;
};
```



```
template <class E, class K>
bool dataList<E, K>::Remove (K x, E& e1) {
    //在dataList中删除关键码为x的元素, 通过e1返回。
    //用尾元素填补被删除元素。
    if (CurrentSize == 0) return false;
    for (int i = 0; i < CurrentSize &&
        Element[i].key != x; i++); //在表中顺序寻找
    if (i == CurrentSize) return false; //未找到
    e1 = Element[i].other; //找到,保存被删元素的值
    Element[i] = Element[CurrentSize - 1]; //填补
    CurrentSize--;
    return true;
};
```



顺序搜索 (Sequential Search)

- 顺序搜索主要用于在线性表中搜索。
- 设若表中有 **CurrentSize** 个元素，则顺序搜索从表的先端开始，顺序用各元素的关键码与给定值 x 进行比较
- 若找到与其值相等的元素，则搜索成功，给出该元素在表中的位置。
- 若整个表都已检测完仍未找到关键码与 x 相等的元素，则搜索失败。给出失败信息。



- 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**。



使用监视哨的顺序搜索算法

```
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;
};
```

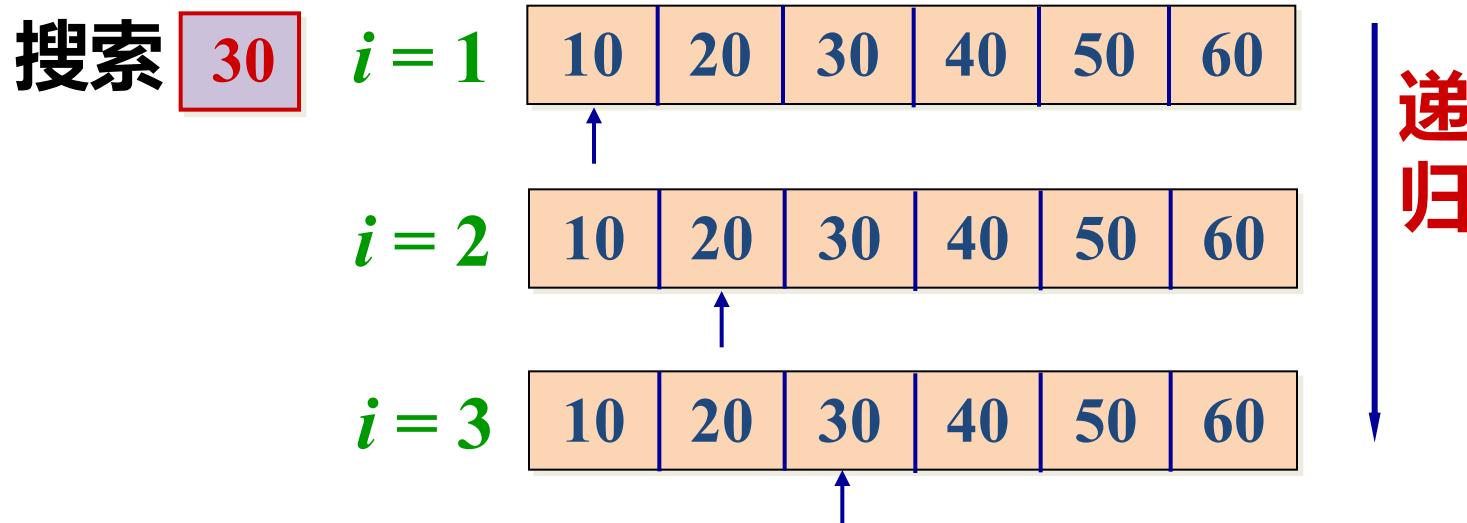


```
const int Size = 10;
main () {
    dataList<int> L1 (Size);          //定义int型搜索表L1
    int Target; int Loc;
    cin >> L1; cout << L1;//输入L1
    cout << "Search for a integer : ";
    cin >> Target;           //输入要搜索的数据
    if ( (Loc = L1.Seqsearch(Target)) <=
        L1.Length() )
        cout << "找到待查元素位置在 : " << Loc+1
        << endl;           //搜索成功
    else cout << " 没有找到待查元素\n";
                                //搜索不成功
};
```



顺序搜索的递归算法

- 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 ($1 \leq i \leq n$) 个位置，则算法递归深度达 i ($1 \sim i$)。





顺序搜索的递归算法

```
template <class E, class K>
```

```
int dataList<E, K>::
```

```
SeqSearch (const K x, int loc) const {
```

```
//在数据表 Element[1..n] 中搜索其关键码与给定值  
//匹配的对象, 函数返回其表中位置。参数 loc 是在  
//表中开始搜索位置
```

```
    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
```

```
};
```



顺序搜索的平均搜索长度

- 设数据表中有 n 个元素，搜索第 i 个元素的概率为 p_i ，搜索到第 i 个元素所需比较次数为 c_i ，则搜索成功的平均搜索长度：

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot c_i. \quad (\sum_{i=0}^{n-1} p_i = 1)$$

- 在顺序搜索并设置“监视哨”情形：

$c_i = i + 1, i = 0, 1, \dots, n-1$ ，因此

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



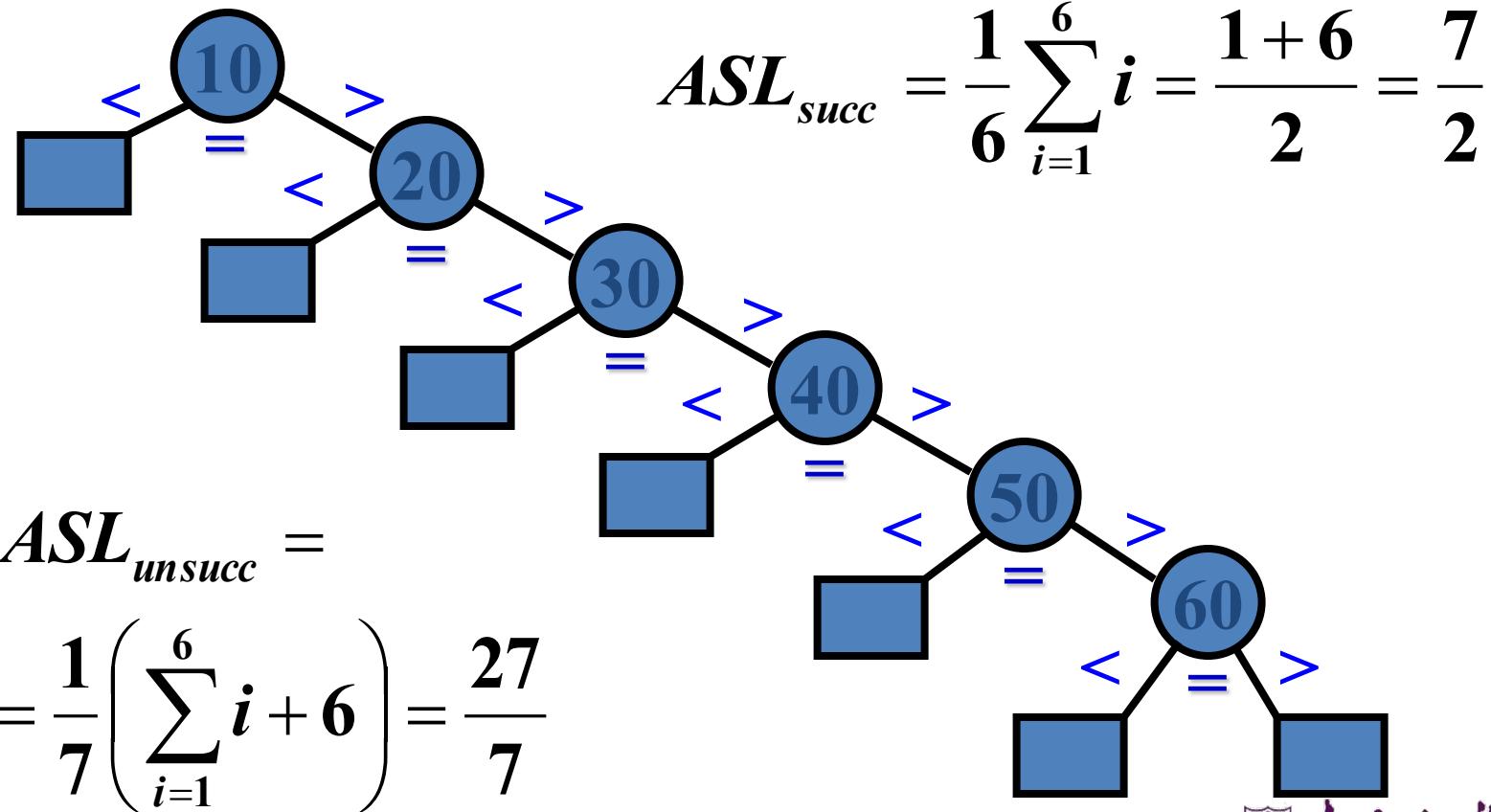
- 一般表中各个元素的搜索概率不同，如果按搜索概率的高低排列表中的元素，从有序顺序表的情况可知，能够得到好的平均搜索长度。
- 在等概率情形， $p_i = 1/n, i = 1, 2, \dots, n$ 。搜索成功的平均搜索长度为：

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

- 在搜索不成功情形， $ASL_{unsucc} = n+1$ 。



- 例如，有序顺序表 **(10, 20, 30, 40, 50, 60)** 的顺序搜索的分析（使用判定树）





- 假定表中所有失败位置的搜索概率相同，则搜索不成功的平均搜索长度：

$$ASL_{unsucc} = \frac{1}{n+1} \left(\sum_{i=1}^n i + n \right)$$

- 时间代价为 $O(n)$ 。
- 为了加速搜索，在有序顺序表的情形，可以采用折半搜索，它也称二分搜索，时间代价可减到 $O(\log_2 n)$ 。

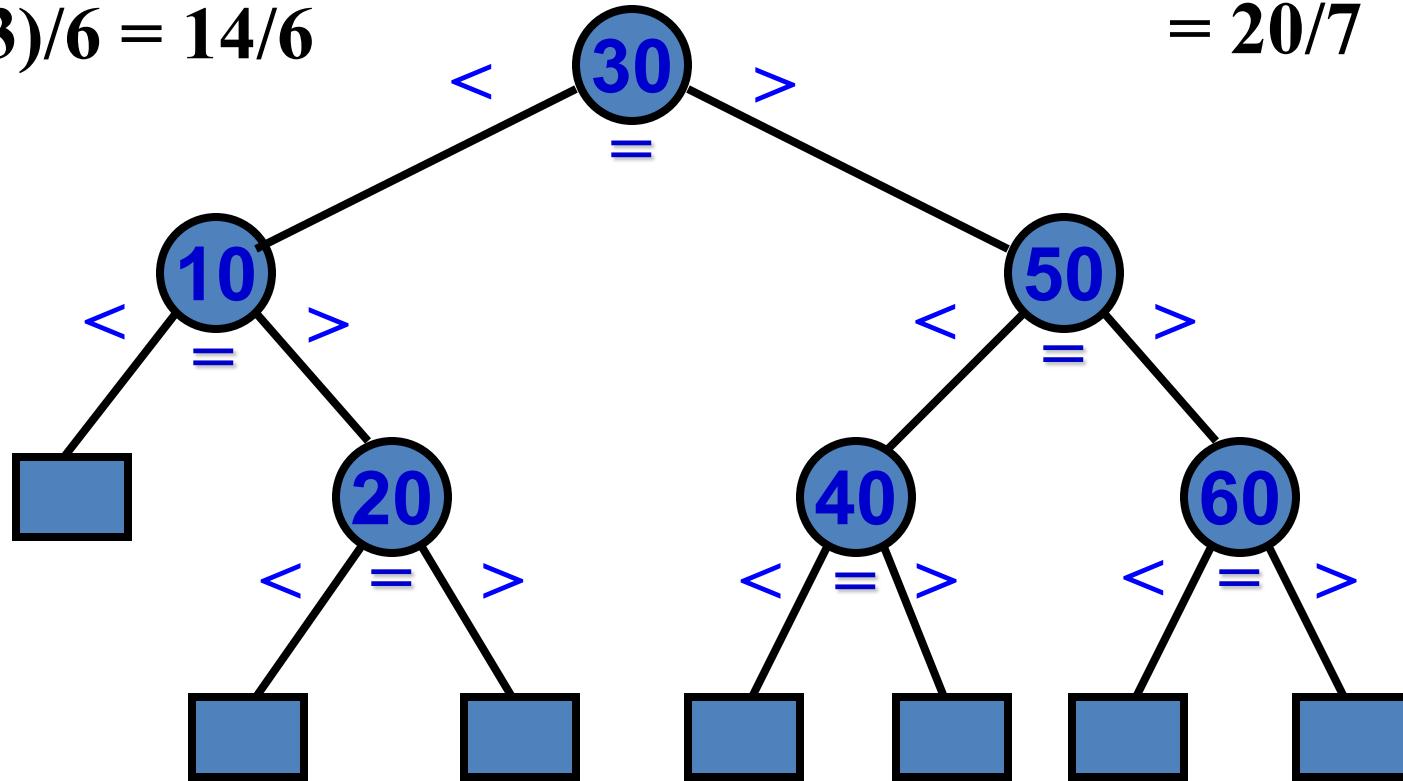


有序顺序表的折半搜索的判定树

(10, 20, 30, 40, 50, 60)

$$ASL_{succ} = (1+2*2+3*3)/6 = 14/6$$

$$ASL_{unsucc} = (2*1+3*6)/7 = 20/7$$





第七章 搜索结构

- 7.1 静态搜索结构
- 7.2 二叉搜索树
- 7.3 AVL树



二叉搜索树 (Binary Search Tree)

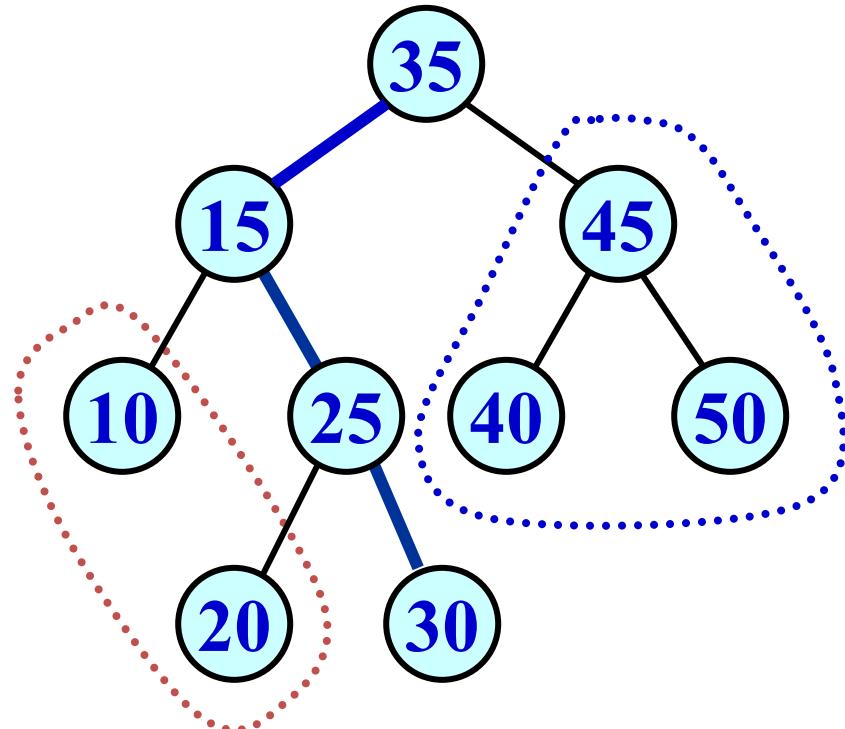
定义 二叉搜索树或者是一棵空树，或者是具有下列性质的二叉树：

- 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。



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；
- 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。





- 如果对一棵二叉搜索树进行中序遍历，可以按从小到大的顺序，将各结点关键码排列起来，所以也称二叉搜索树为二叉排序树。

二叉搜索树的类定义

```
#include <iostream.h>
#include <stdlib.h>
template <class E, class K>
struct BSTNode {           //二叉树结点类
    E data;                //数据域
    BSTNode<E, K> *left, *right; //左子女和右子女
```



```
BSTNode() { left = NULL; right = NULL; }
```

//构造函数

```
BSTNode (const E d, BSTNode<E, K> *L = NULL,  
         BSTNode<E, K> *R = NULL)
```

```
{ data = d; left = L; right = R; }
```

//构造函数

```
~BSTNode() {} //析构函数
```

```
void setData (E d) { data = d; } //修改
```

```
E getData() { return data; } //提取
```

```
bool operator < (const E& x) //重载：判小于  
{ return data.key < x.key; }
```



```
bool operator > (const E& x)           //重载：判大于
    { return data.key > x.key; }

bool operator == (const E& x)           //重载：判等于
    { return data.key == x.key; }

};

template <class E, class K>
class BST {                           //二叉搜索树类定义
public:
    BST() { root = NULL; }             //构造函数
    BST(K value);                   //构造函数
    ~BST() {};                      //析构函数
```



```
bool Search (const K x) const      //搜索
{ return Search(x,root) != NULL; }

BST<E, K>& operator = (const BST<E, K>& R);
                           //重载 : 赋值

void makeEmpty()           //置空
{ makeEmpty (root); root = NULL; }

void PrintTree() const { PrintTree (root); } //输出

E Min() { return Min(root)->data; }        //求最小

E Max() { return Max(root)->data; }        //求最大

bool Insert (const E& e1)          //插入新元素
{ return Insert(e1, root); }
```



bool Remove (const K x) { return Remove(x, root);}
//删除含x的结点

private:

BSTNode<E, K> *root; //根指针
K RefValue; //输入停止标志
BSTNode<E, K> * //递归 : 搜索
 Search (const K x, BSTNode<E, K> *ptr);
void makeEmpty (BSTNode<E, K> *& ptr);
 //递归 : 置空
void PrintTree (BSTNode<E, K> *ptr) const;
 //递归 : 打印
BSTNode<E, K> * //递归 : 复制
 Copy (const BSTNode<E, K> *ptr);



```
BSTNode<E, K>* Min (BSTNode<E, K>* ptr);
    //递归 : 求最小

BSTNode<E, K>* Max (BSTNode<E, K>* ptr);
    //递归 : 求最大

bool Insert (const E& e1, BSTNode<E, K>*& ptr);
    //递归 : 插入

bool Remove (const K x, BSTNode<E, K>*& ptr);
    //递归 : 删除

};
```

- 二叉搜索树的类定义用二叉链表作为它的存储表示，许多操作的实现与二叉树类似。

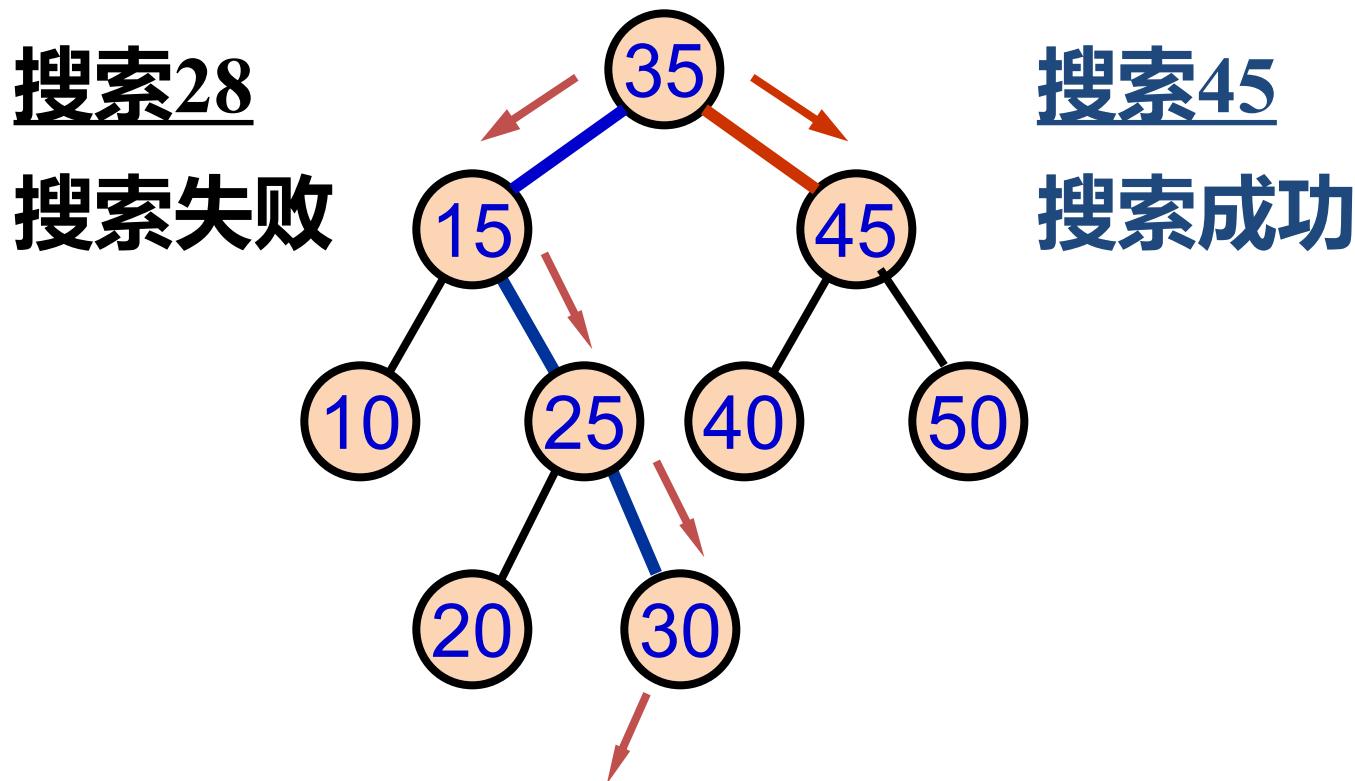


二叉搜索树的搜索算法

- 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。



- ✓ 若给定值小于根结点的关键码，则继续递归搜索根结点的左子树；
- ✓ 否则。递归搜索根结点的右子树。





```
template<class E, class K>
BSTNode<E, K>* BST<E, K>::
Search (const K x, BSTNode<E, K> *ptr) {
    //私有递归函数：在以ptr为根的二叉搜索树中搜
    //索含x的结点。若找到，则函数返回该结点的
    //地址，否则函数返回NULL值。
    if (ptr == NULL) return NULL;
    else if (x < ptr->data.key) return Search(x, ptr-
        >left);
    else if (x > ptr->data.key) return Search(x, ptr-
        >right);
    else return ptr;                                //搜索成功
};
```



template<class E, class K>

BSTNode<E, K>* BST<E, K>::

Search (**const K** x, BSTNode<E, K> *ptr) {

//**非递归函数**：作为对比，在当前以ptr为根的二叉搜索树
//中搜索含x的结点。若找到，则函数返回该结点的地址
//否则函数返回NULL值。

if (ptr == NULL) **return** NULL;

BSTNode<E, K>* temp = ptr;

while (temp != NULL) {

if (x == temp->data.key) **return** temp;

if (x < temp->data.key) temp = temp->left;



```
    else temp = temp->right;  
}  
  
return NULL;  
};
```

- 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。



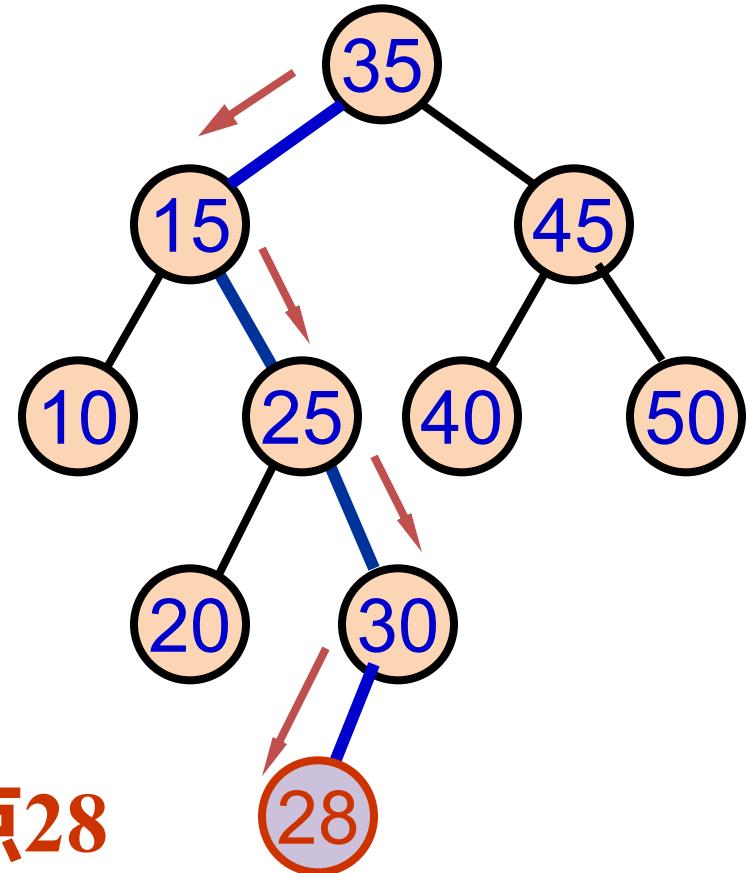
二叉搜索树的插入算法

- 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。



二叉搜索树的插入

- 每次结点的插入，都要从根结点出发搜索插入位置，然后把新结点作为叶结点插入。





二叉搜索树的插入算法

```
template <class E, class K>
bool BST<E, K>::Insert (const E& e1,
    BSTNode<E, K> *& ptr) {
//私有函数：在以ptr为根的二叉搜索树中插入值为
//e1的结点。若在树中已有含e1的结点则不插入
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.key) Insert (e1, ptr->left);
    //左子树插入

else if (e1 > ptr->data.key) Insert (e1, ptr->right);
    //右子树插入

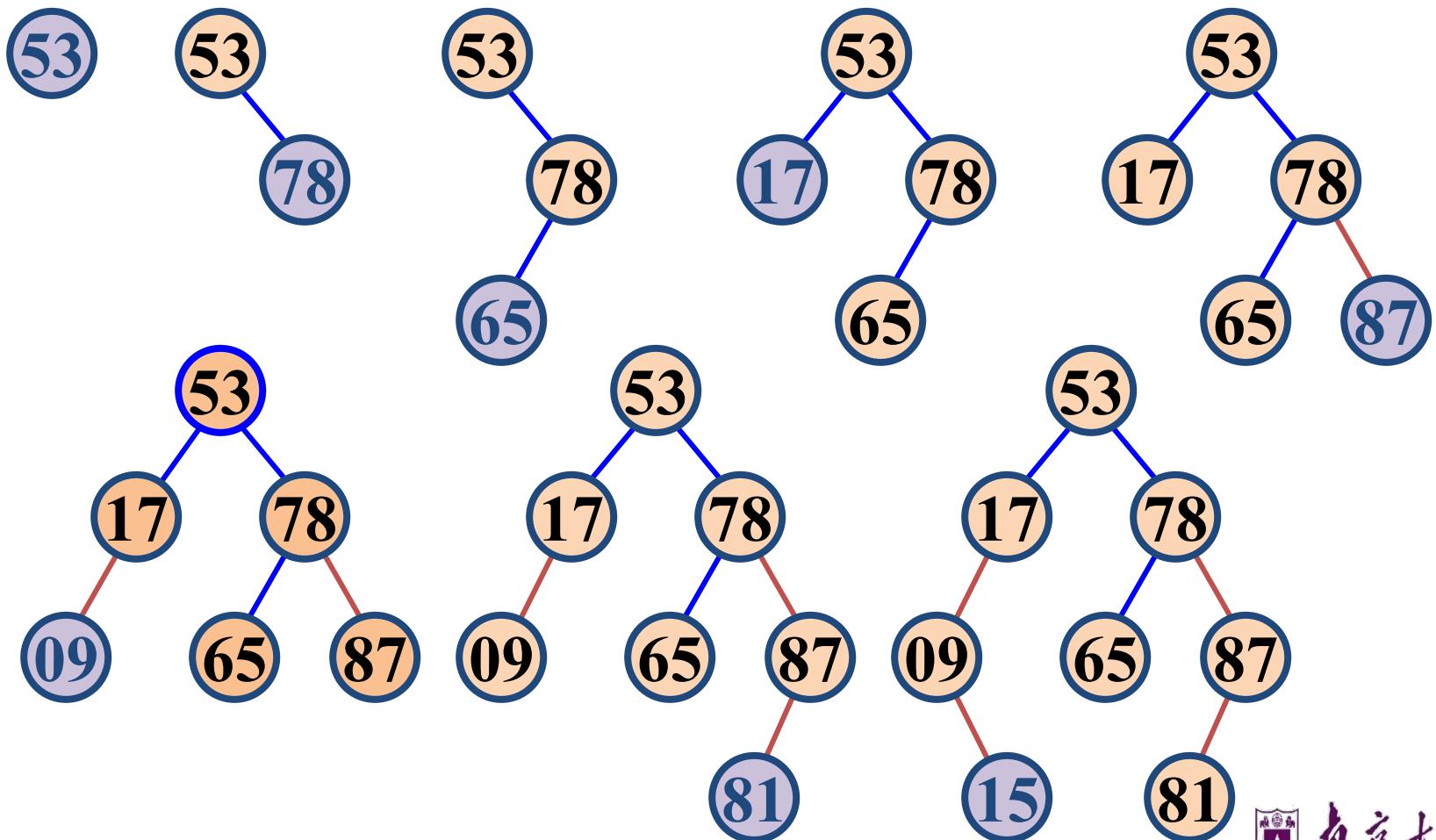
else return false;           //x已在树中,不再插入

};
```

- 注意参数表中引用型指针参数ptr的使用。
- 利用二叉搜索树的插入算法，可以很方便地建立二叉搜索树。



输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }





```
template <class E, class K>
BST<E, K>::BST (K value) {
//输入一个元素序列, 建立一棵二叉搜索树
    E x;
    root = NULL; RefValue = value //置空树
    cin >> x; //输入数据
    while ( x.key != RefValue) {
        //RefValue是一个输入结束标志
        Insert (x, root); cin >> x;//插入 , 输入数据
    }
};
```



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。
 - ✓ 删除叶结点，只需将其父结点指向它的指针清零，再释放它即可。
 - ✓ 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。

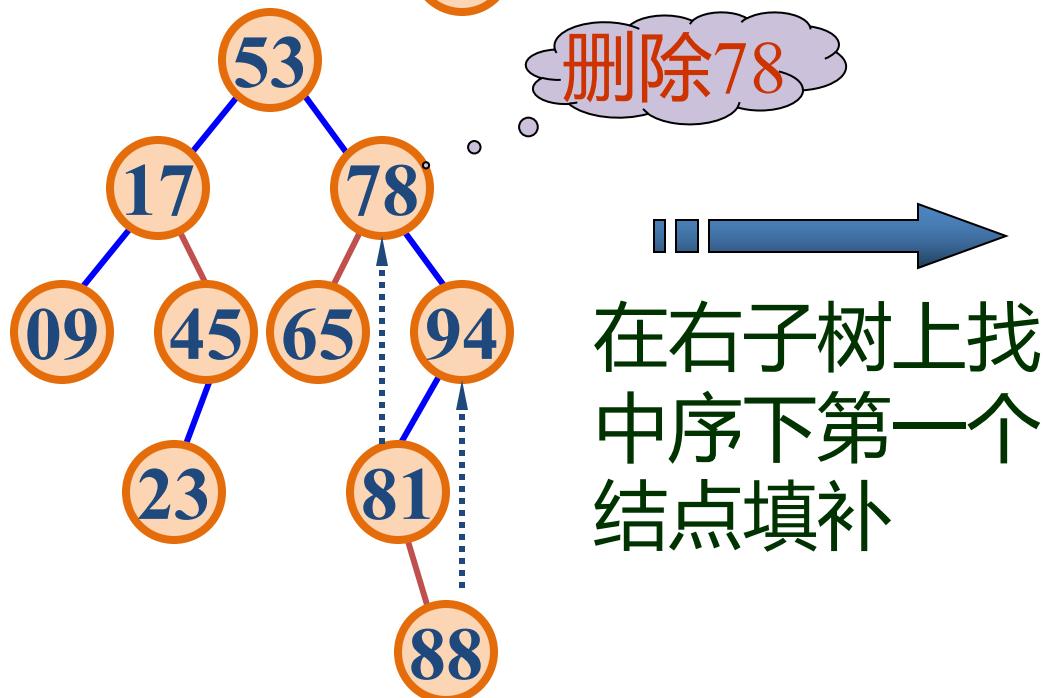
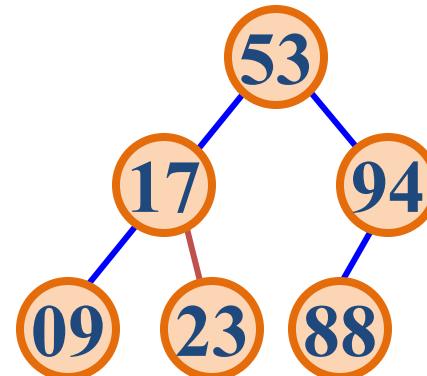


- ✓ 被删结点左子树为空，可以拿它的右子女结点顶替它的位置，再释放它。
- ✓ 被删结点左、右子树都不为空，可以在它的右子树中寻找中序下的第一个结点（关键码最小），用它的值填补到被删结点中，再来处理这个结点的删除问题。

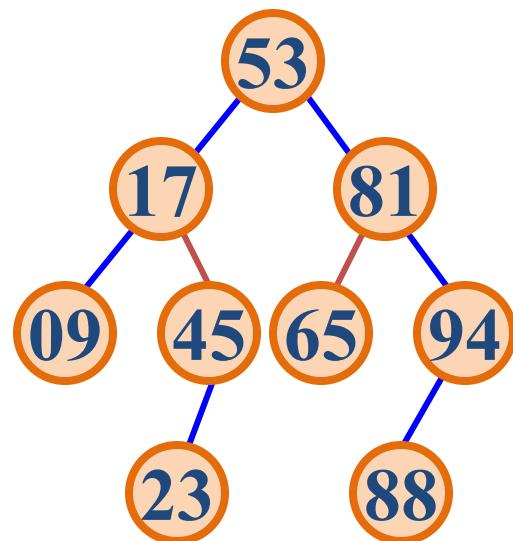




左子树空, 用右子女顶替
Left subtree is empty, replace with right children



在右子树上找中序下第一个结点填补
Find the first node in the right subtree for in-order successor replacement





二叉搜索树的删除算法

```
template <class E, class K>
bool BST<E, K>::Remove (const K x,
    BstNode<E, K> *& ptr) {
    //在以 ptr 为根的二叉搜索树中删除含 x 的结点
    BstNode<E, K> *temp;
    if (ptr != NULL) {
        if (x < ptr->data.key) Remove (x, ptr->left);
            //在左子树中执行删除
        else if (x > ptr->data.key) 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.key = temp->data.key;
    //用该结点数据代替根结点数据
    Remove (ptr->data.key, ptr->right); //已经移到其他位置，删除掉原来位置上的
}
else { //ptr指示关键码为x的结点最多有一个子女
```



```
temp = ptr;  
    if (ptr->left == NULL) ptr = ptr->right;  
    else ptr = ptr->left;  
    delete temp;  
    return true;  
}  
}  
return false;  
};
```

- 注意在删除算法参数表引用型指针参数的使用。



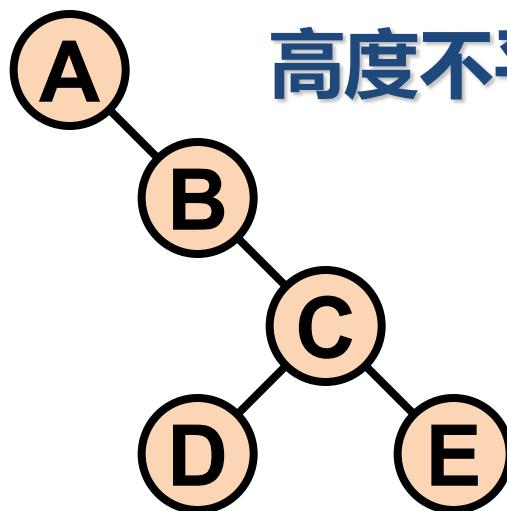
第七章 搜索结构

- 7.1 静态搜索结构
- 7.2 二叉搜索树
- 7.3 AVL树

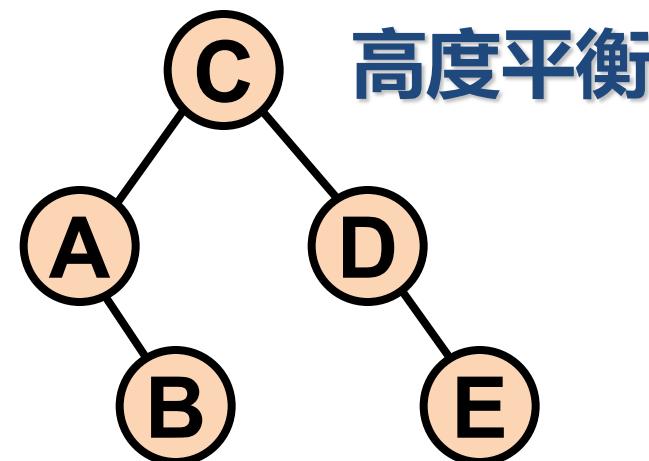


AVL树 高度平衡的二叉搜索树

- **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 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 $O(\log_2 n)$ 。



AVL树的类定义

```
#include <iostream.h>
#include "stack.h"
template <class E, class K>
struct AVLNode : public BSTNode<E, K> {
    //AVL树结点的类定义
    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& e1) { return Insert (root, e1); } //插入
bool Remove (K x, E& e1)
    { return Remove (root, x, e1); } //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& e1);
bool Remove (AVLNode<E, K>*& ptr, K x, E& e1);
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
};
```

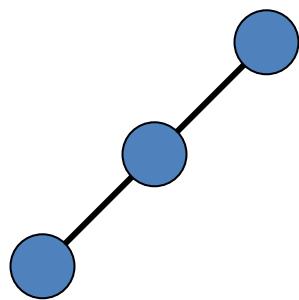


平衡化旋转

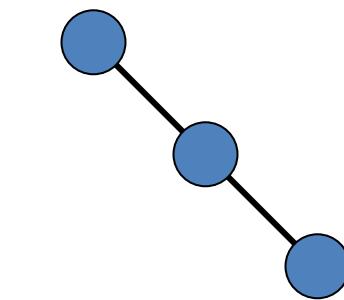
- 如果在一棵平衡的二叉搜索树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 平衡化旋转有两类：
 - ✓ 单旋转 (左旋和右旋)
 - ✓ 双旋转 (左平衡和右平衡)
- 每插入一个新结点时，AVL 树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。



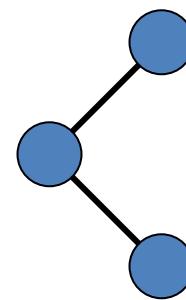
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。 单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。 双旋转分为先左后右和先右后左两类。



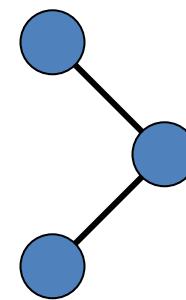
右单旋转



左单旋转



左右双旋转

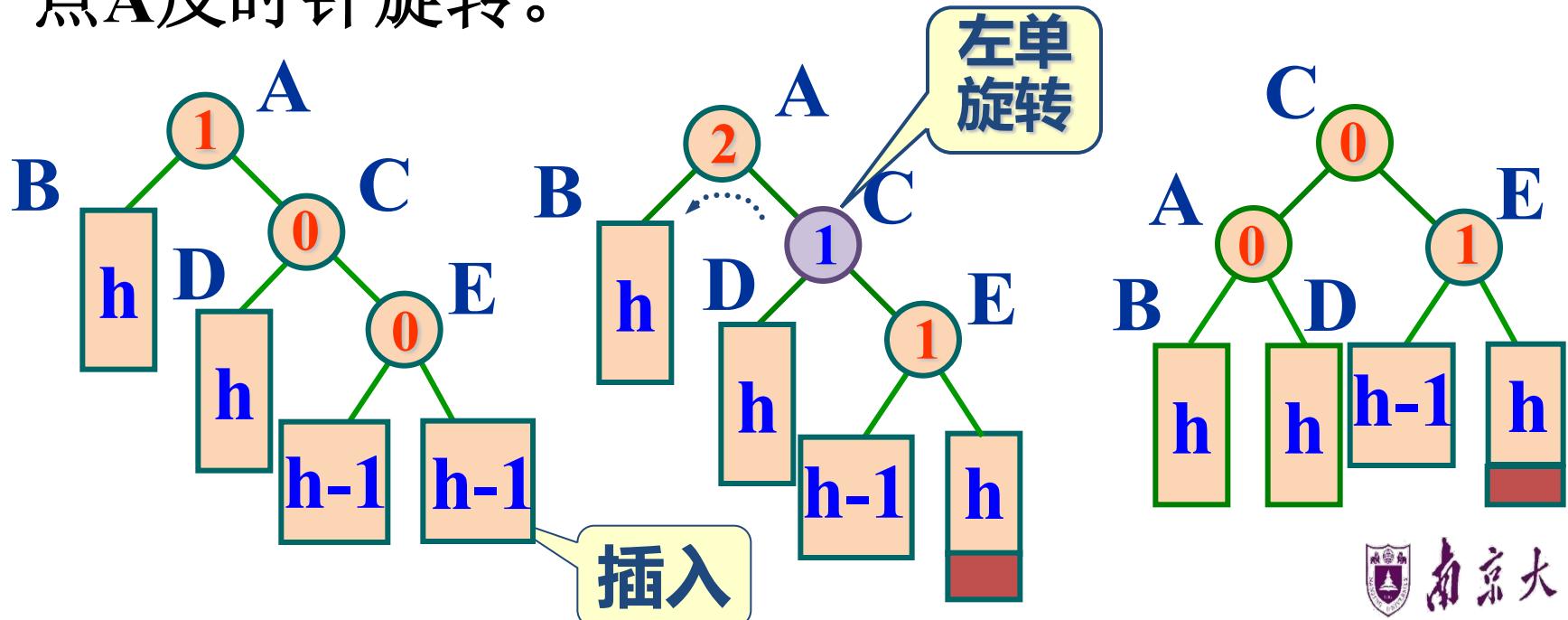


右左双旋转



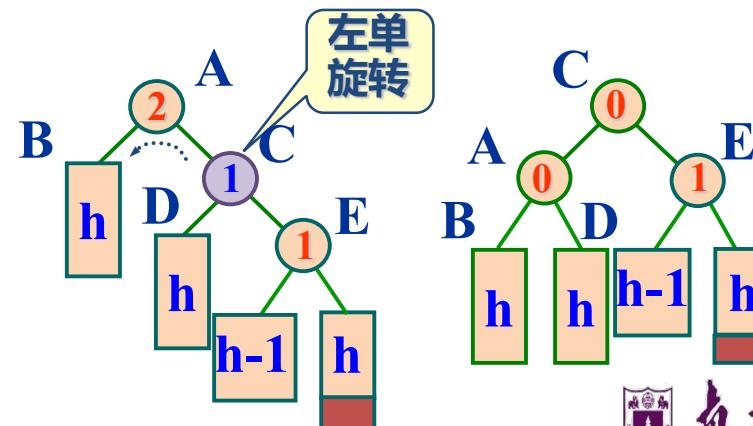
左单旋转 (Rotate Left)

在结点A的右子女的右子树E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，出现不平衡。为使树恢复平衡，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。





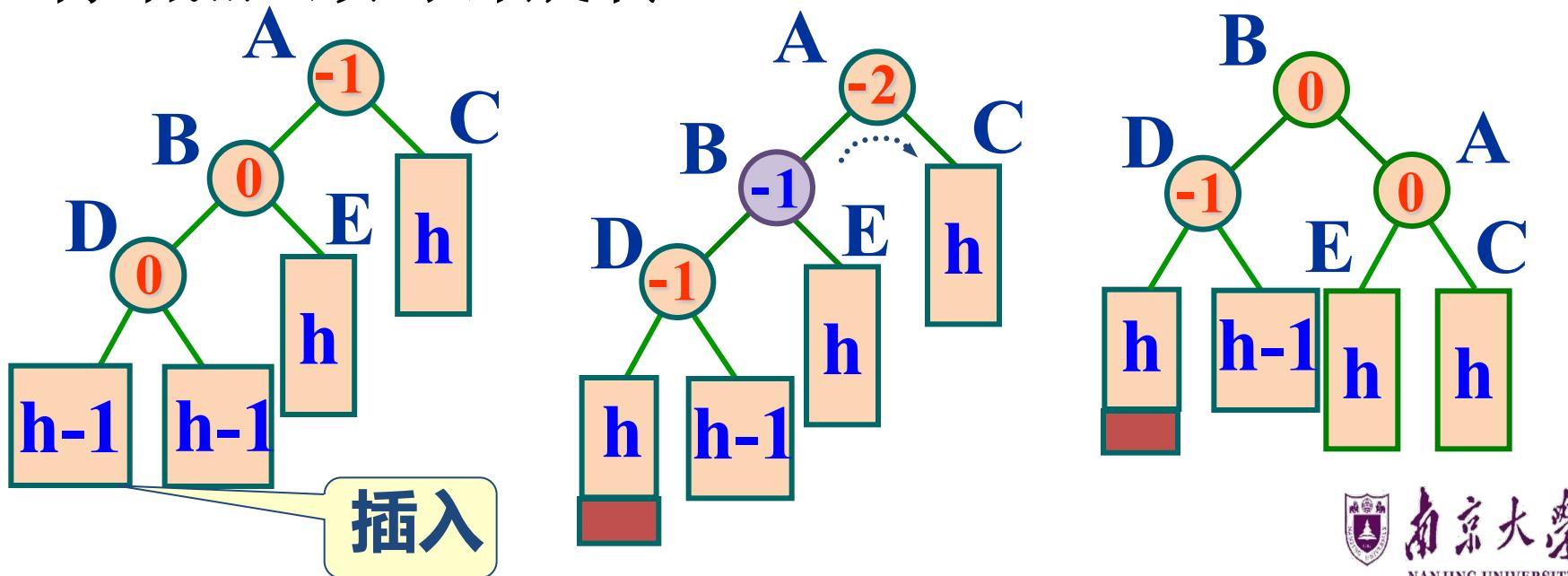
```
template <class E, class K>
void AVLTree<E, K>:: RotateL (AVLNode<E, K> *& ptr) {
//右子树比左子树高: 做左单旋转后新根在ptr
    AVLNode<E, K> *subL = ptr;      // A结点
    ptr = subL->right;              // C结点上升 , 替代A
    subL->right = ptr->left;        // D成为A的右结点
    ptr->left = subL;               // A成为C的左结点
    ptr->bf = subL->bf = 0;
};
```





右单旋转 (Rotate Right)

- 在结点A的左子女的左子树D上插入新结点使其高度增1导致结点A的平衡因子增到-2，造成不平衡。为使树恢复平衡，从A沿插入路径插入路径连续取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。



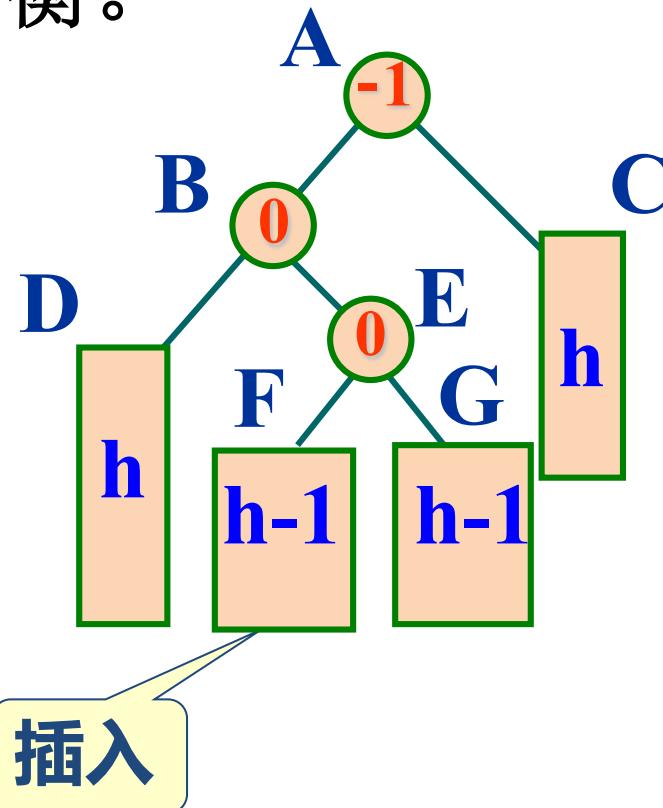


```
template <class E, class K>
void AVLTree<E, K>::
    RotateR (AVLNode<E, K> *& ptr) {
    //左子树比右子树高, 旋转后新根在ptr
        AVLNode<E, K> *subR = ptr; //要右旋转的结点
        ptr = subR->left;
        subR->left = ptr->right; //转移ptr右边负载
        ptr->right = subR; //ptr成为新根
        ptr->bf = subR->bf = 0;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



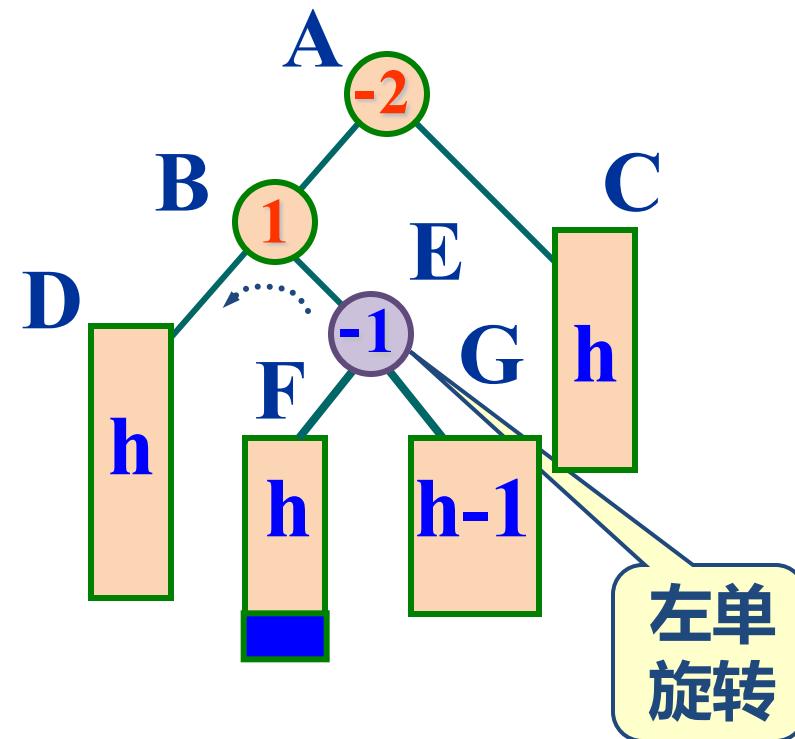
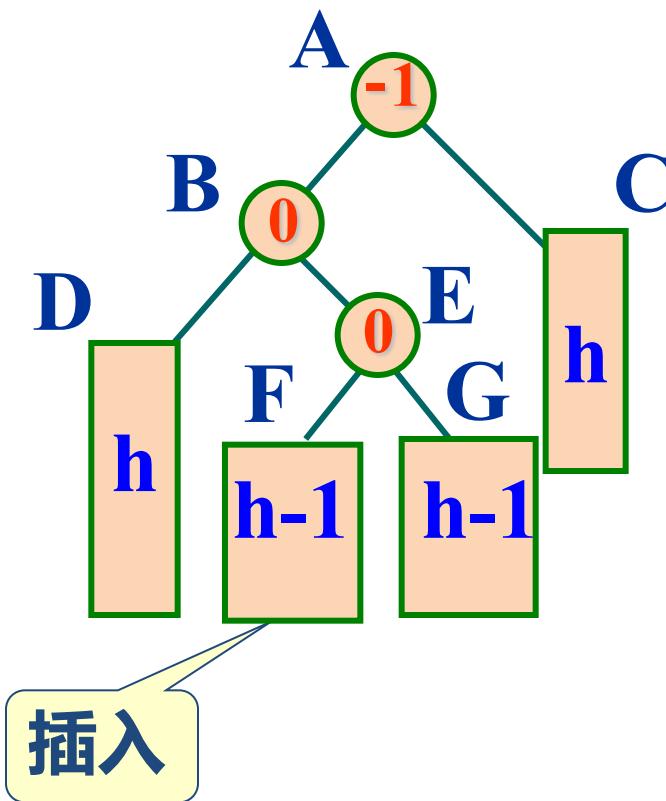
先左后右双旋转 (RotationLeftRight)

- 在结点A的左子女的右子树中插入新结点，该子树高度增1导致结点A的平衡因子变为-2，造成不平衡。



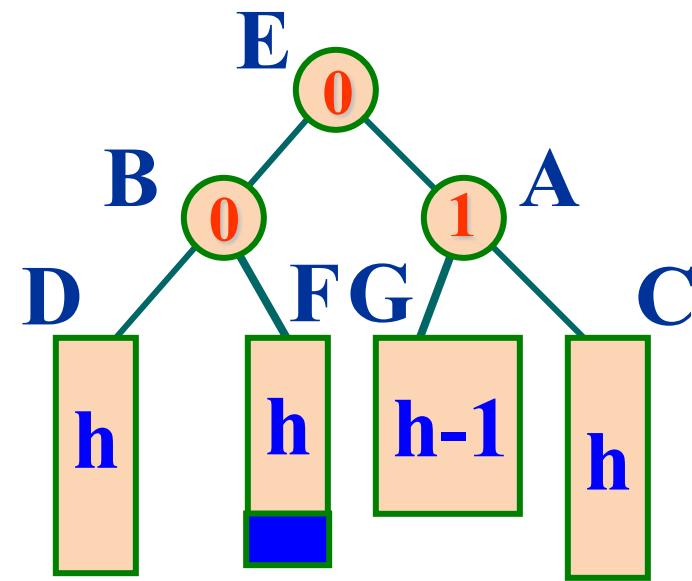
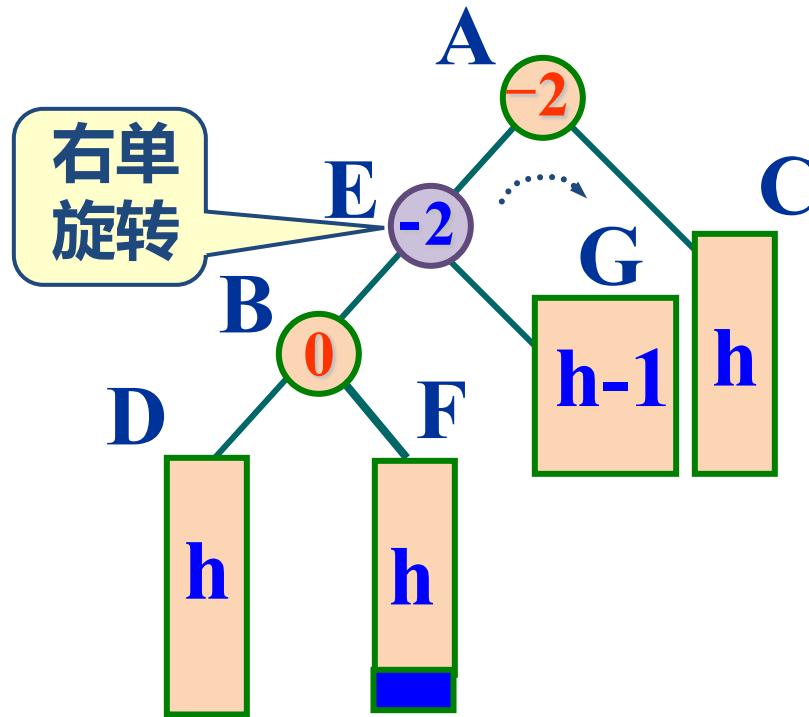


- 以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置。





- 再以结点E为旋转轴，将结点A顺时针旋转。
使之平衡化。



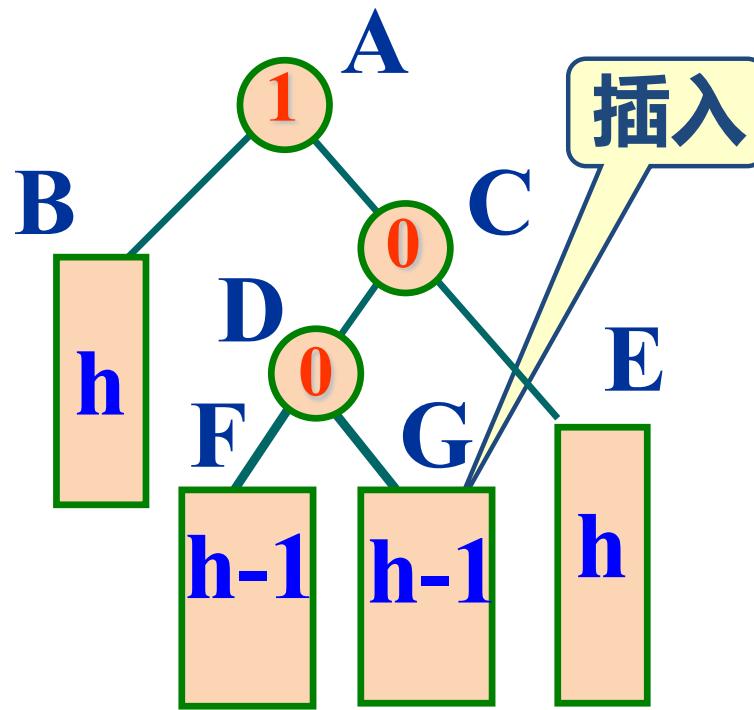


```
template <class E, class K>
void AVLTree<E, K>:: RotateLR (AVLNode<E, K> *& ptr) {
    AVLNode<E, K> *subR = ptr; //A结点
    AVLNode<E, K> *subL = subR->left; //B结点
    ptr = subL->right; //E结点上升
    subL->right = ptr->left; //E的左孩子成为B的右孩子
    ptr->left = subL; //B结点成为E的左孩子
    if (ptr->bf <= 0) subL->bf = 0;
    else subL->bf = -1;
    subR->left = ptr->right; //E的右孩子成为B的左孩子
    ptr->right = subR; //A结点成为E的右孩子
    if (ptr->bf == -1) subR->bf = 1;
    else subR->bf = 0;
    ptr->bf = 0;
};
```



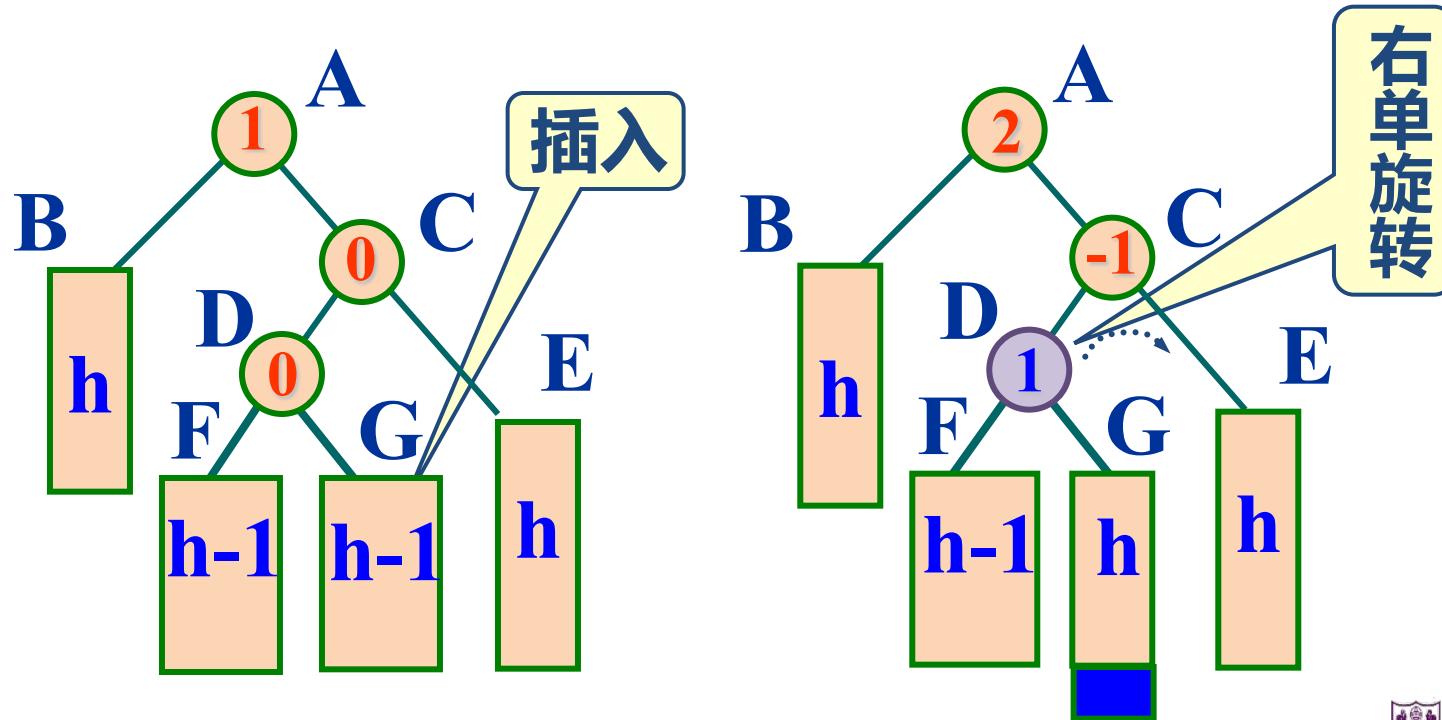
先右后左双旋转 (RotationRightLeft)

- 在结点A的右子女的左子树中插入新结点，该子树高度增1。结点A的平衡因子变为2，发生了不平衡。



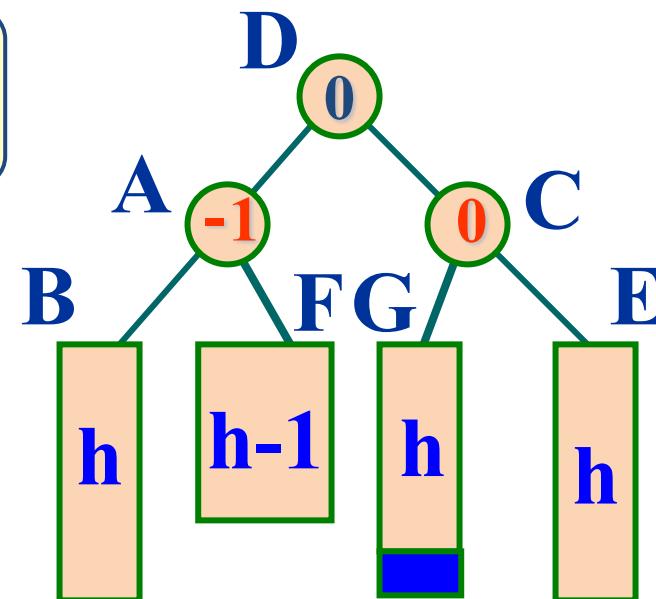
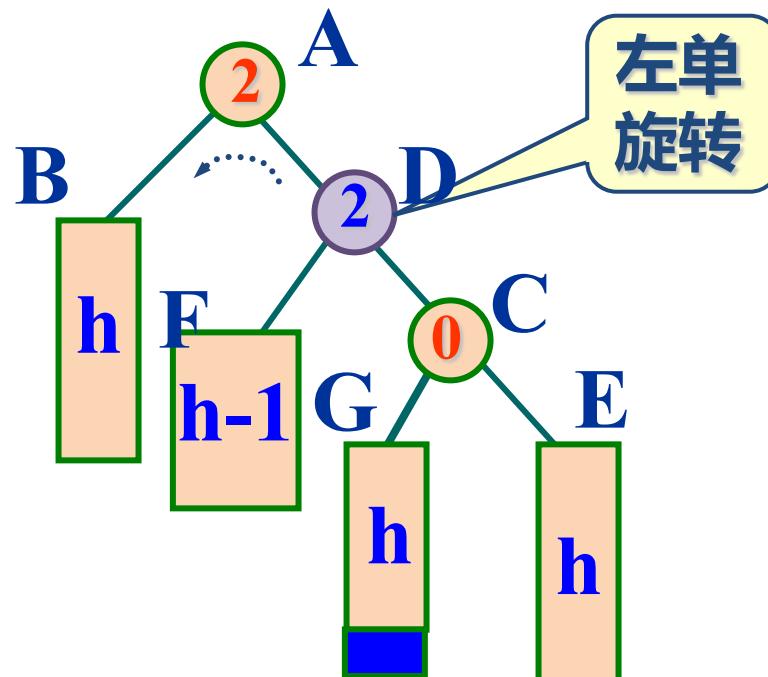


- 首先以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。





- 再以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。





```
template <class E, class K>
void AVLTree<E, K>::
RotateRL (AVLNode<E, K> *& ptr) {
    AVLNode<E, K> *subL = ptr;
    AVLNode<E, K> *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;
};
```

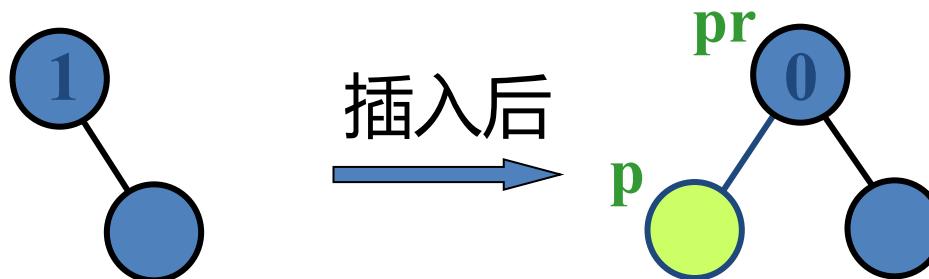


AVL树的插入

- 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。



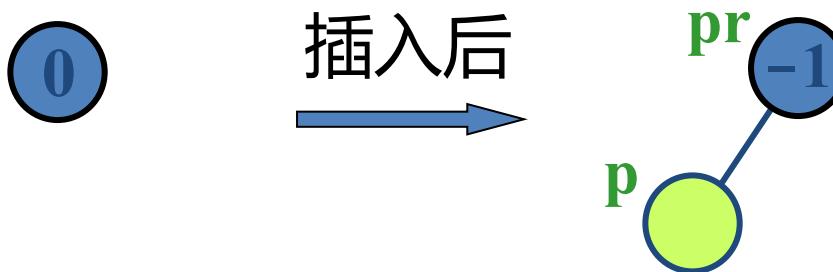
- 设新结点p的平衡因子为0，其父结点为pr。插入新结点后pr的平衡因子值有三种情况：
 - 结点pr的平衡因子为0。说明刚才是在pr的较矮的子树上插入了新结点，此时不需做平衡化处理，返回主程序。子树的高度不变。



- 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($pr = \text{Parent}(pr)$)的平衡状态。

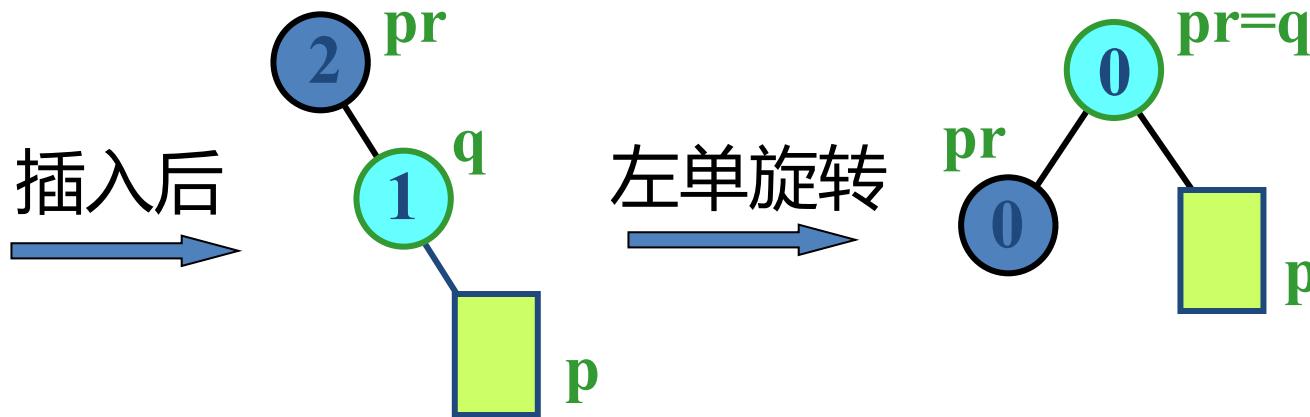


3. 结点pr的平衡因子的绝对值 $|bf| = 2$ 。说明新结点在较高的子树上插入，造成了不平衡，需要做平衡化旋转。此时可进一步分2种情况讨论：

- ① 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：



—若 q 的bf为1， 执行左单旋转。

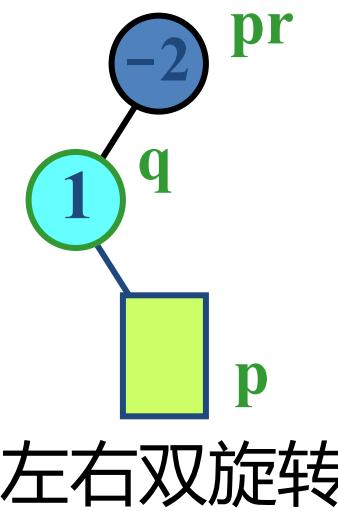
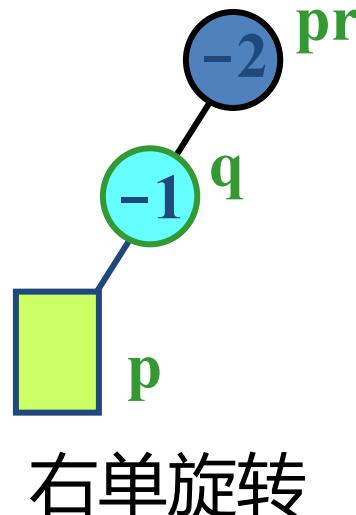


—若 q 的bf为-1， 执行先右后左双旋转。



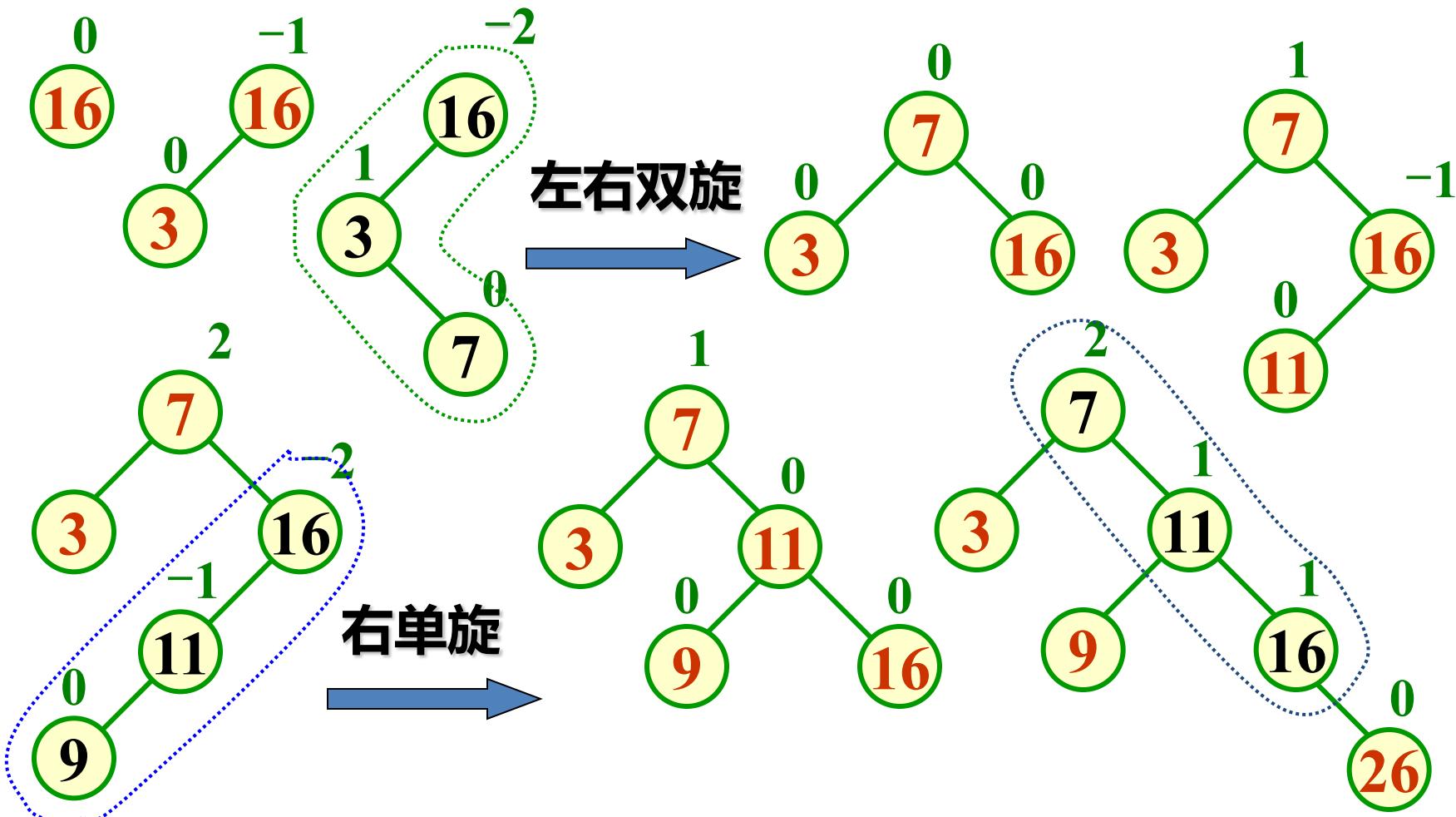


- ② 若结点pr的 $bf = -2$, 说明左子树高, 结合其左子女q的 bf 分别处理:
- 若q的 bf 为-1, 执行右单旋转;
 - 若q的 bf 为1, 执行先左后右双旋转。



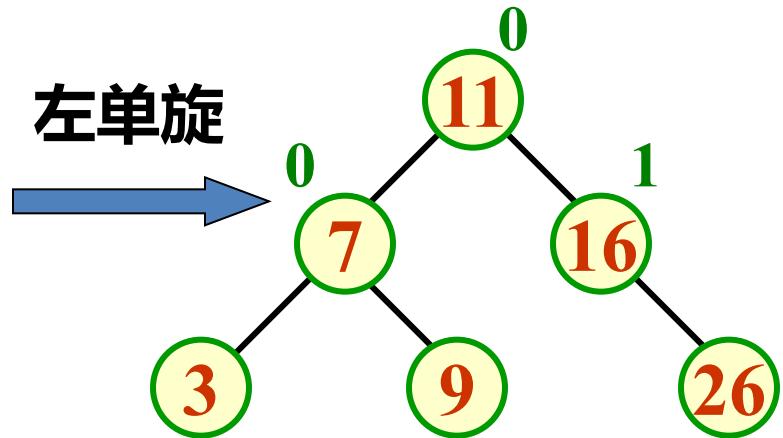
- 下面举例说明在AVL树上的插入过程。

- 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。

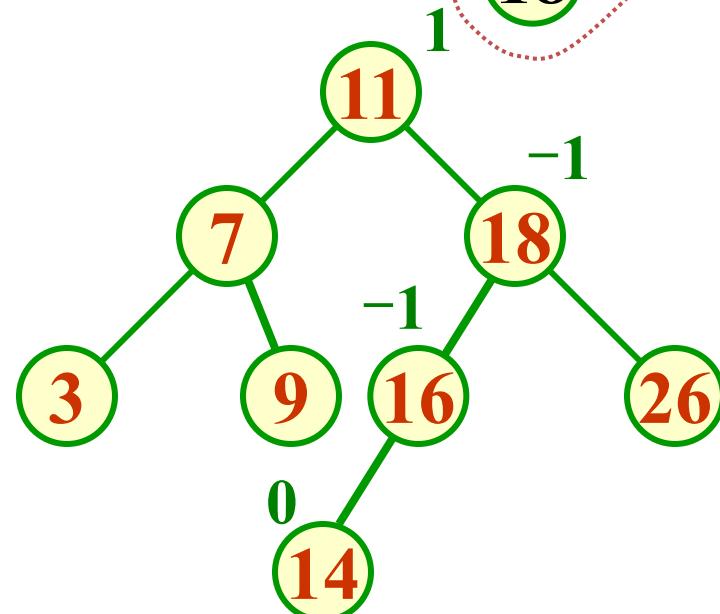
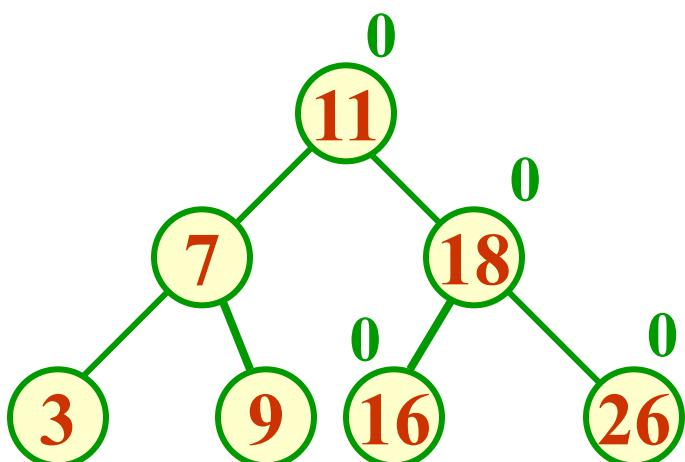
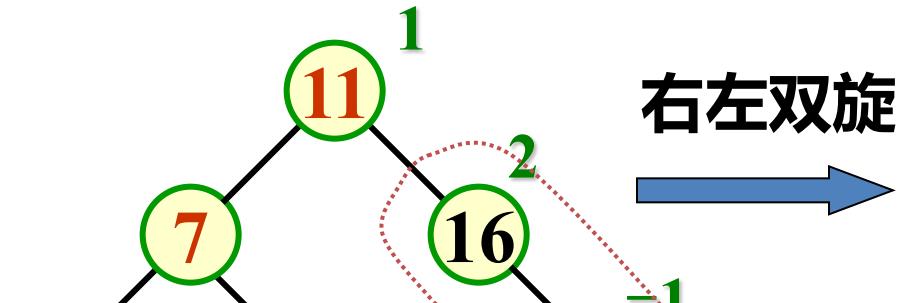


关键码 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }

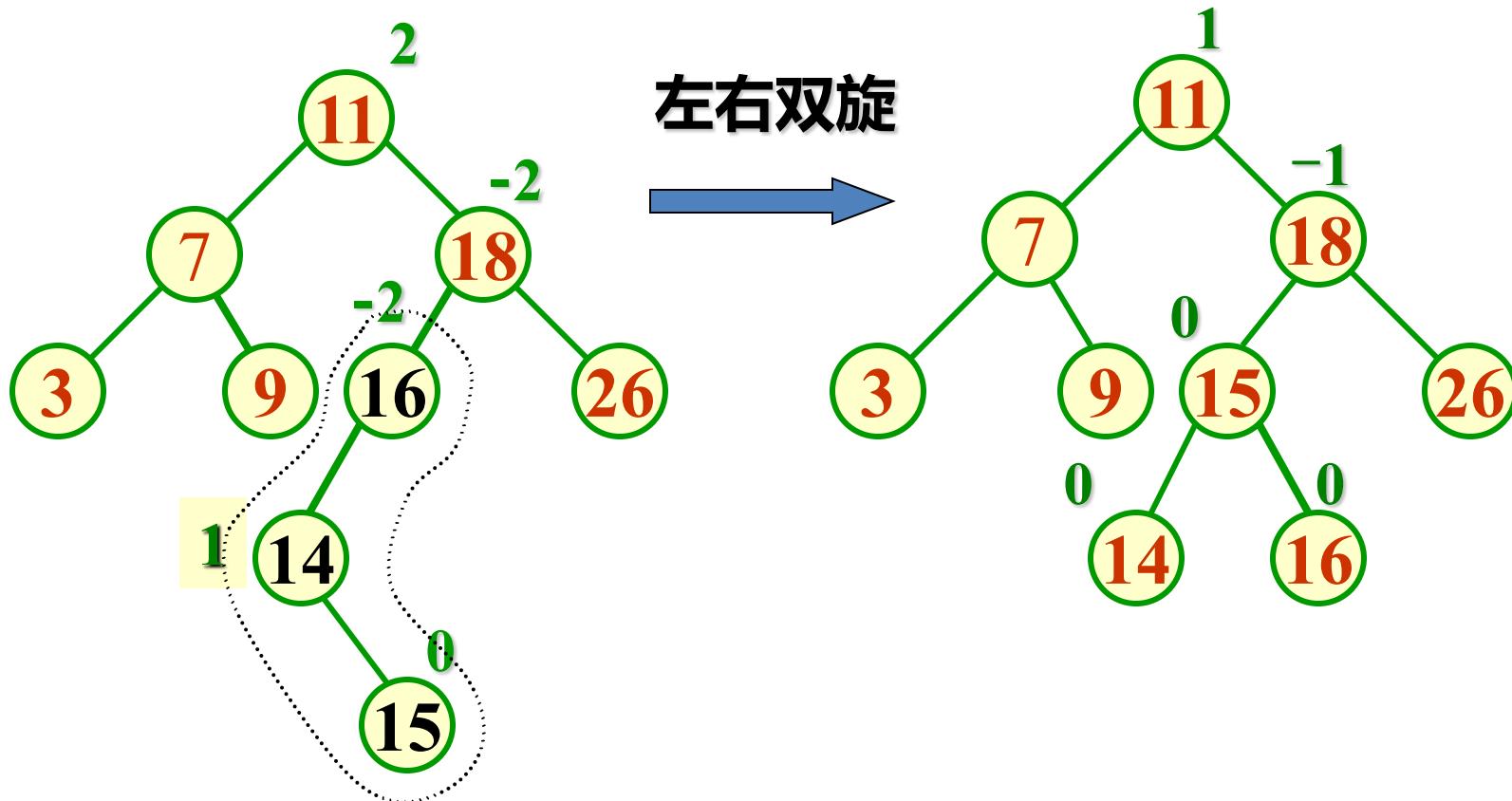
左单旋



右左双旋



关键码 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }



从空树开始的建树过程



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。

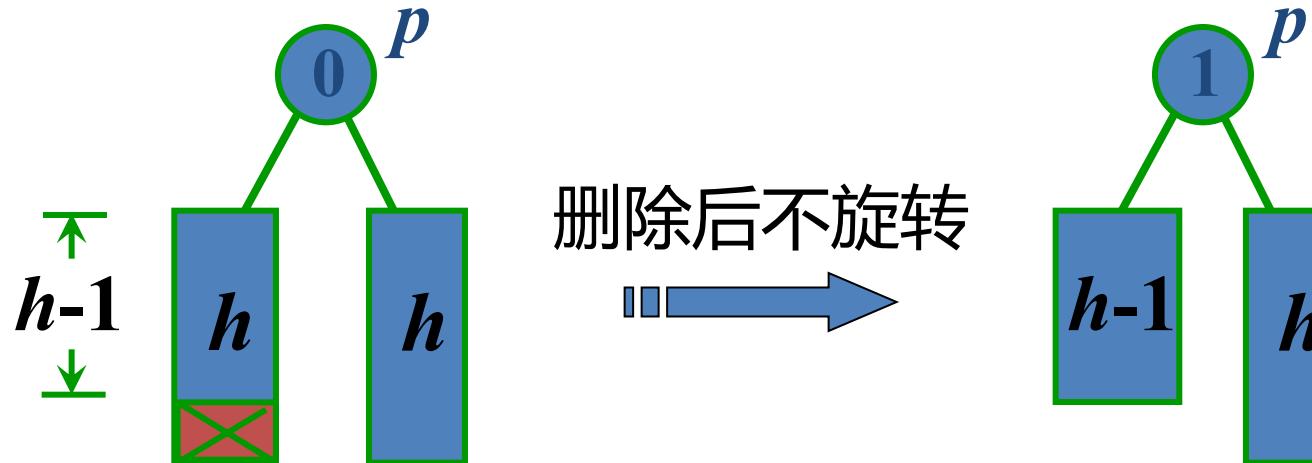


2. 如果被删结点 x 有两个子女:

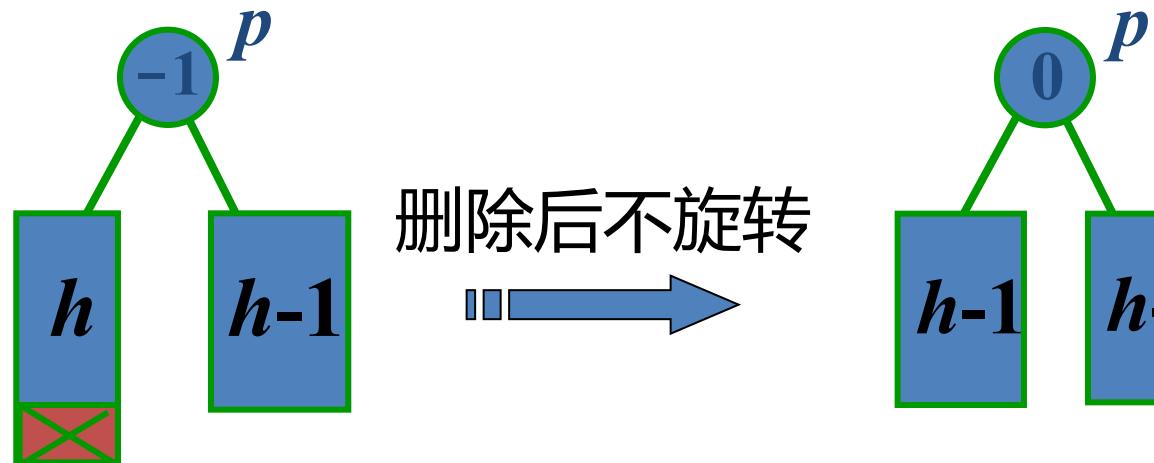
- 搜索 x 在中序次序下的直接前驱 y (同样可以找直接后继)。
- 把结点 y 的内容传送给结点 x , 现在问题转移到删除结点 y 。把结点 y 当作被删结点 x 。
- 因为结点 y 最多有一个子女, 可以简单地用 1. 给出的方法进行删除。
- 必须沿结点 x 通向根的路径反向追踪高度的变化对路径上各个结点的影响。



- . 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。
 - 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**。



- 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。

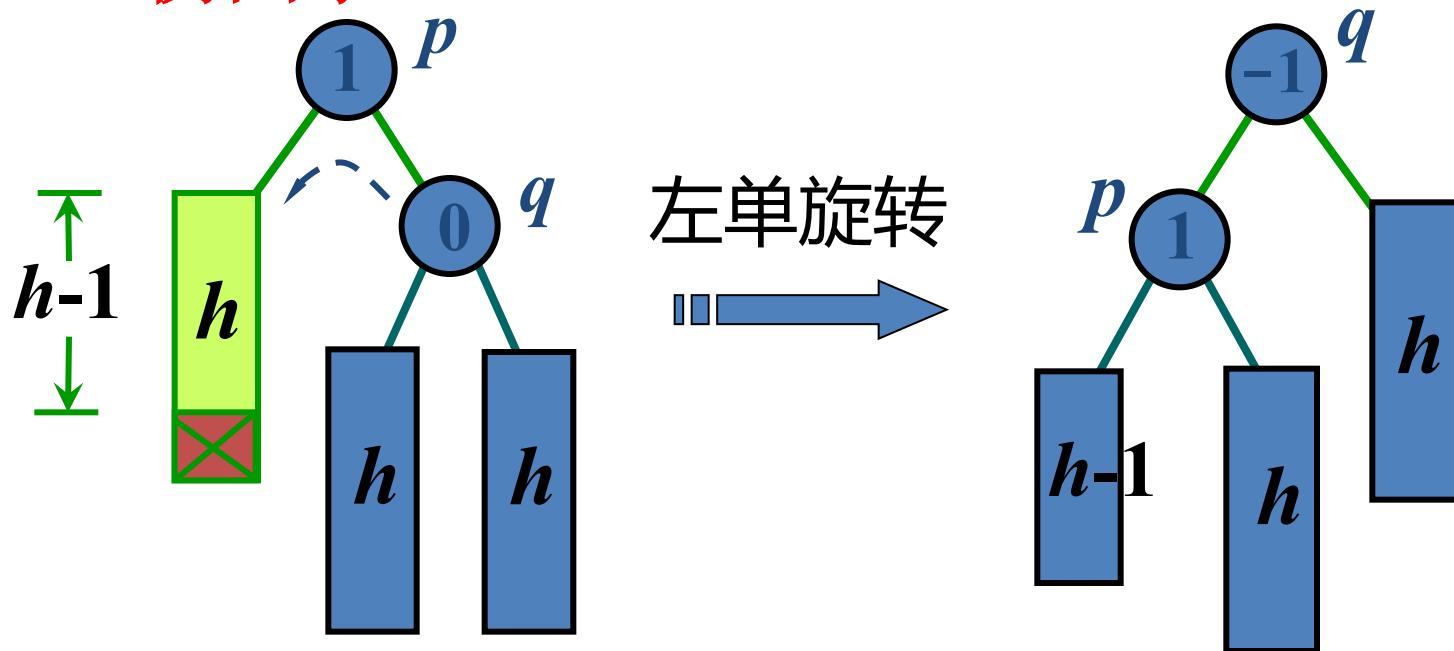




-
- ③ 结点 p 的 bf 不为 0，且较矮的子树又被缩短。则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。
 - 令 p 的较高的子树的根为 q （该子树未被缩短），根据 q 的 bf，有如下 3 种平衡化操作。
 - 旋转的方向取决于是结点 p 的哪一棵子树被缩短。

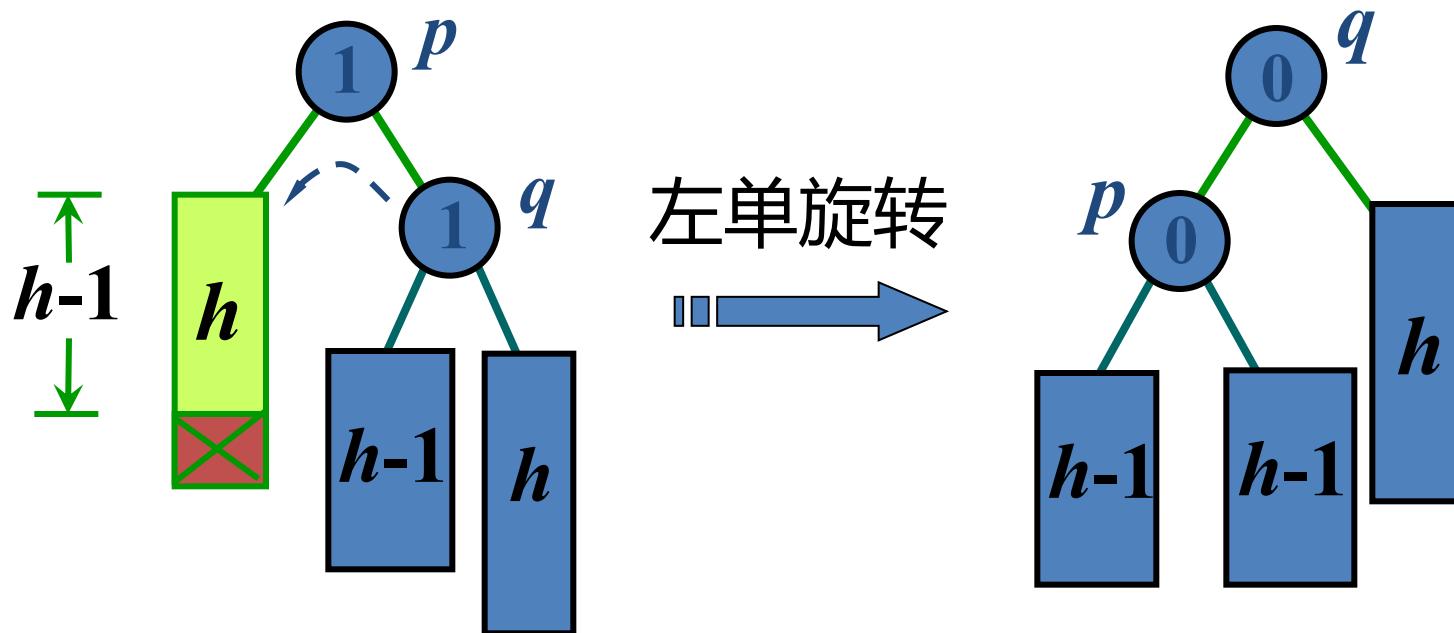


a) 如果 q (较高的子树) 的 bf 为0, 执行一个单旋转来恢复结点 p 的平衡, 置 shorter为False。无需检查上层结点的平衡因子。



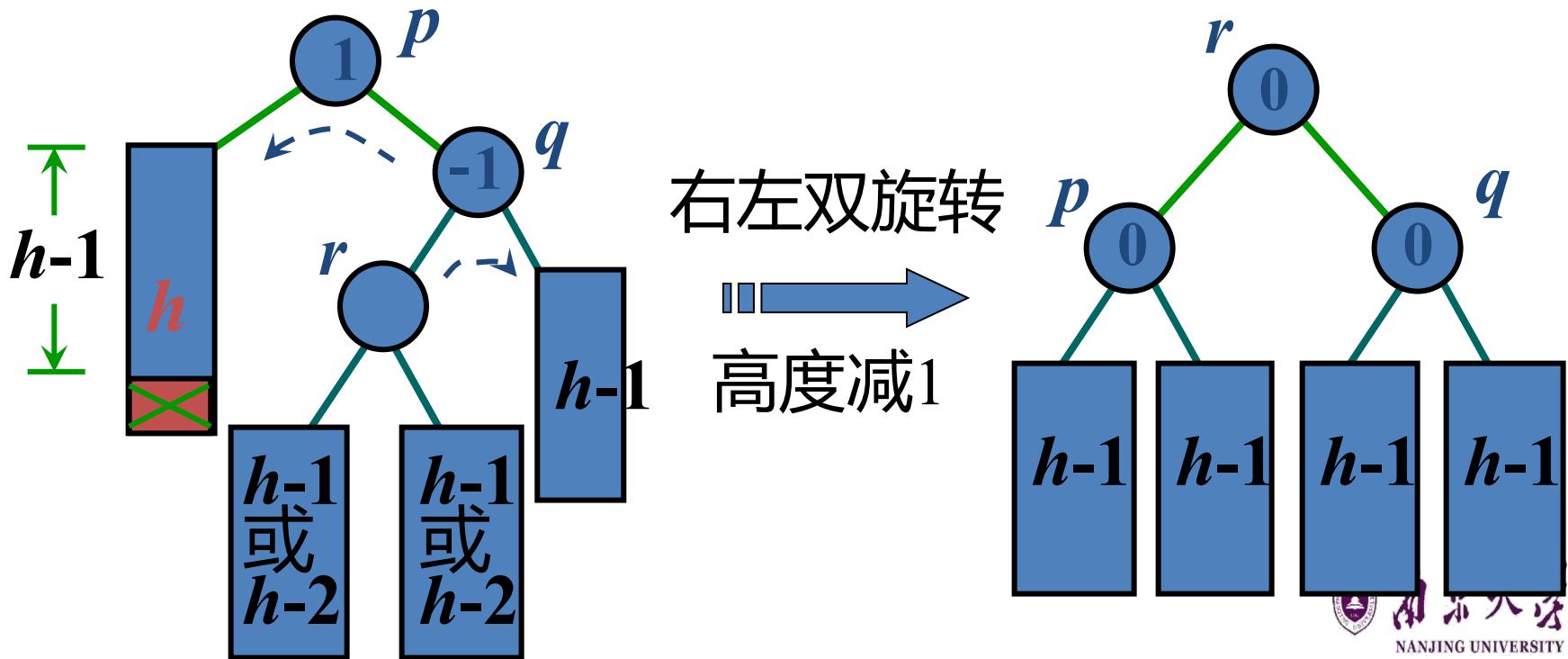


b) 如果 q 的 bf 与 p 的 bf 相同，则执行一个单旋转来恢复平衡，结点 p 和 q 的 bf 均改为0，同时置shorter为True。还要继续检查上层结点的平衡因子。



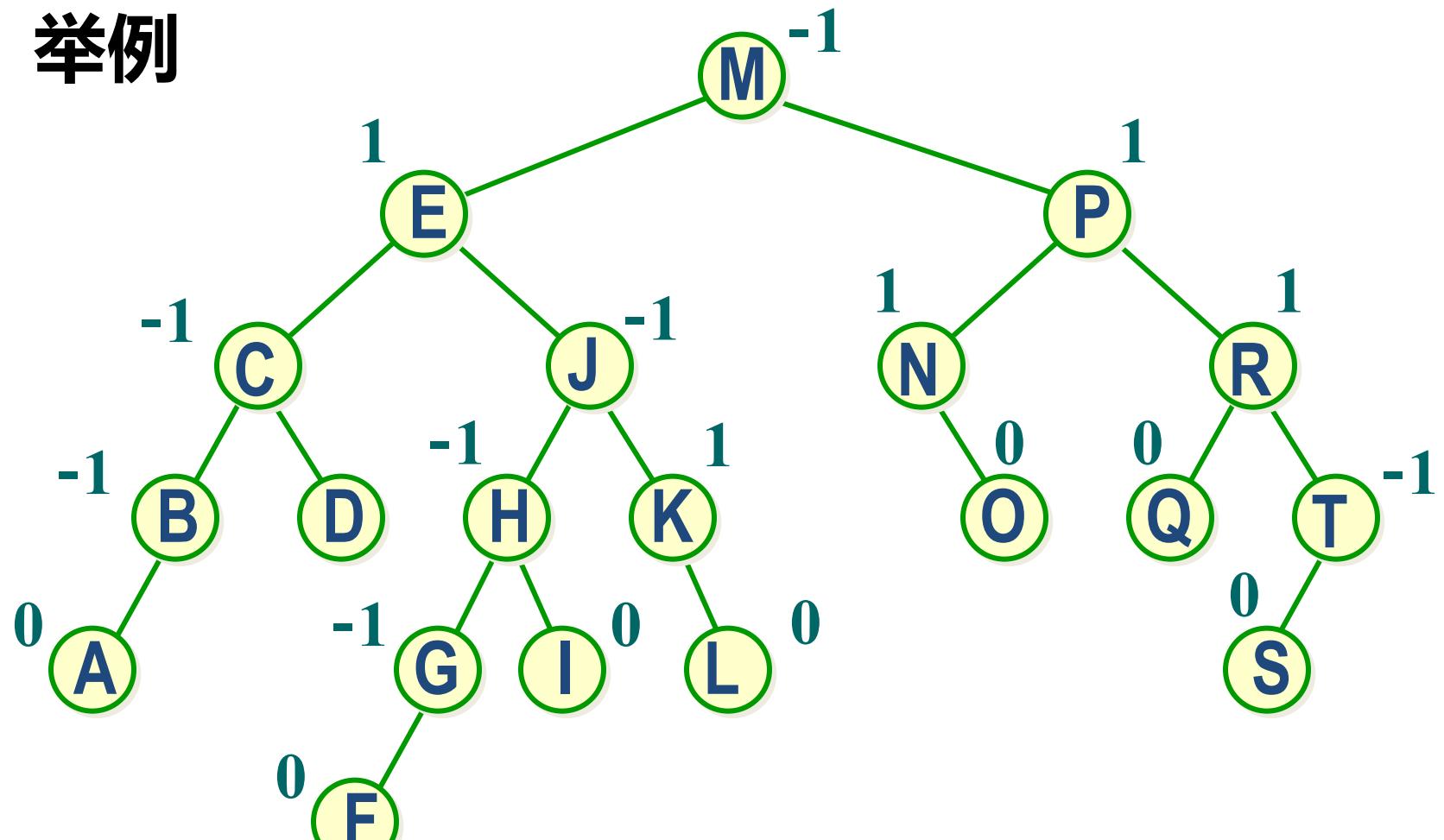


c) 如果 p 与 q 的 bf 相反，则执行一个双旋转来恢复平衡。先围绕 q 转再围绕 p 转。新根结点的 bf 置为0，其他结点的 bf 相应处理，同时置shorter为True。还要继续检查上层结点的平衡因子。





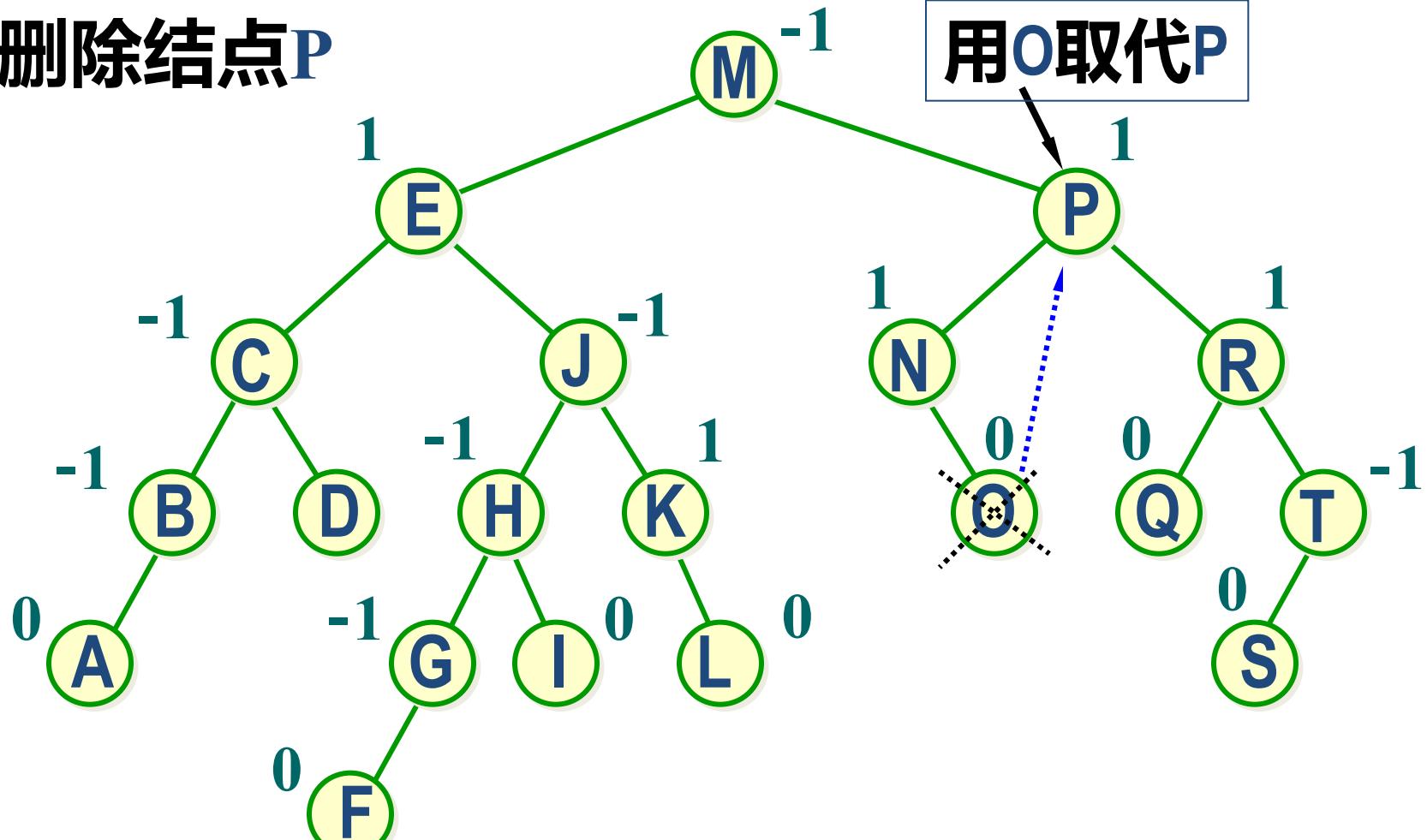
举例



树的初始状态



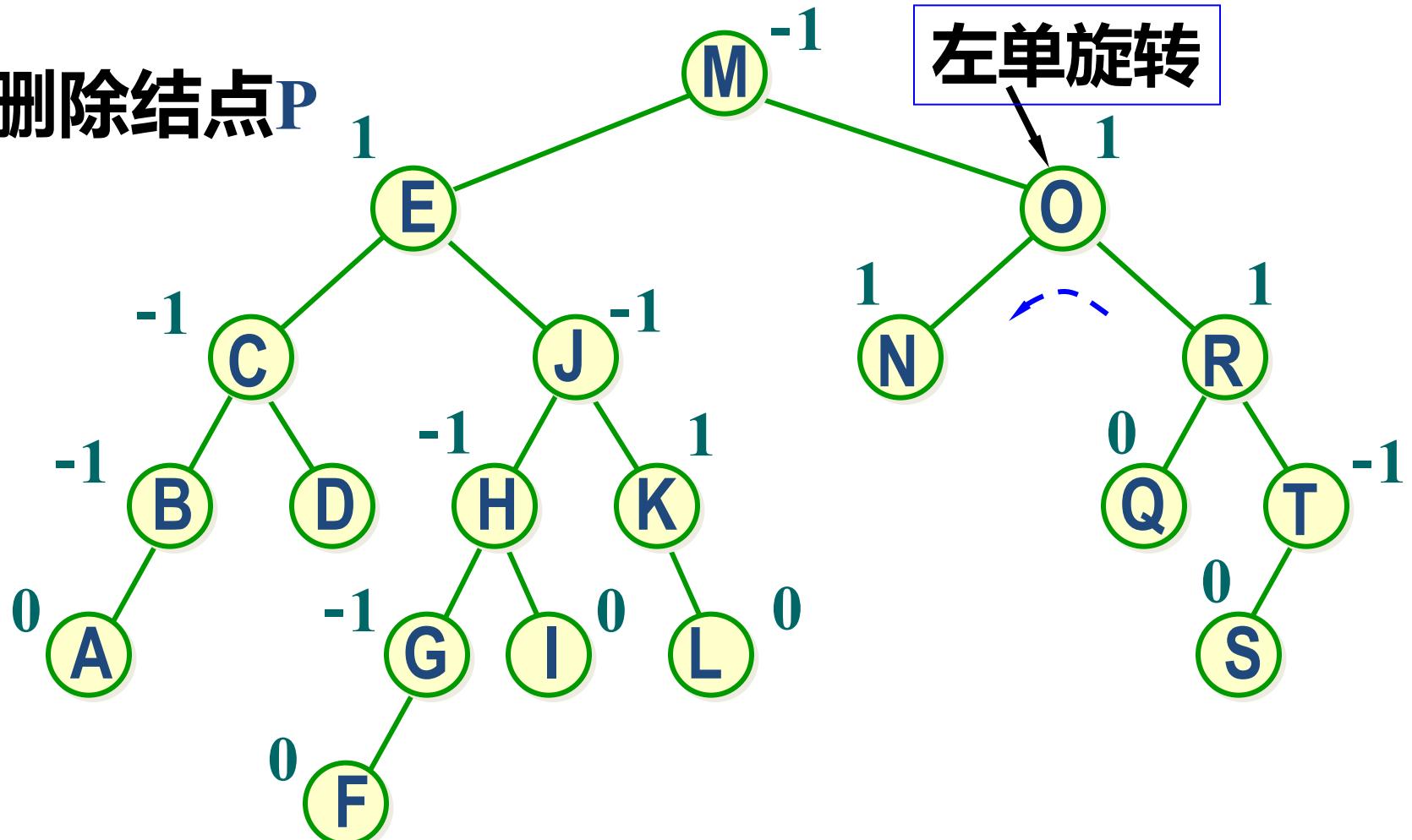
删除结点P



寻找结点P的中序直接前驱O, 用O顶替P, 删除O。



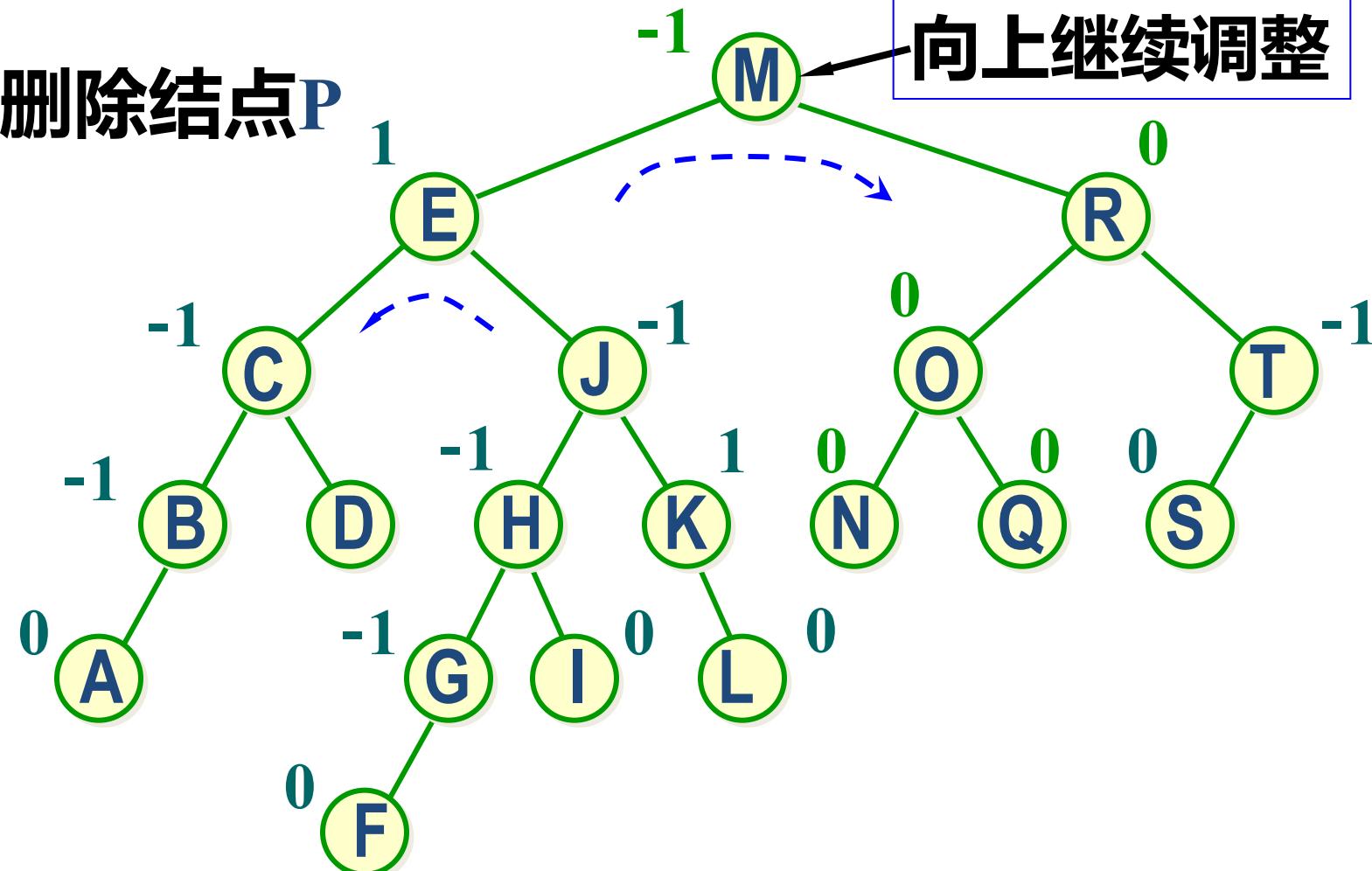
删除结点P



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。



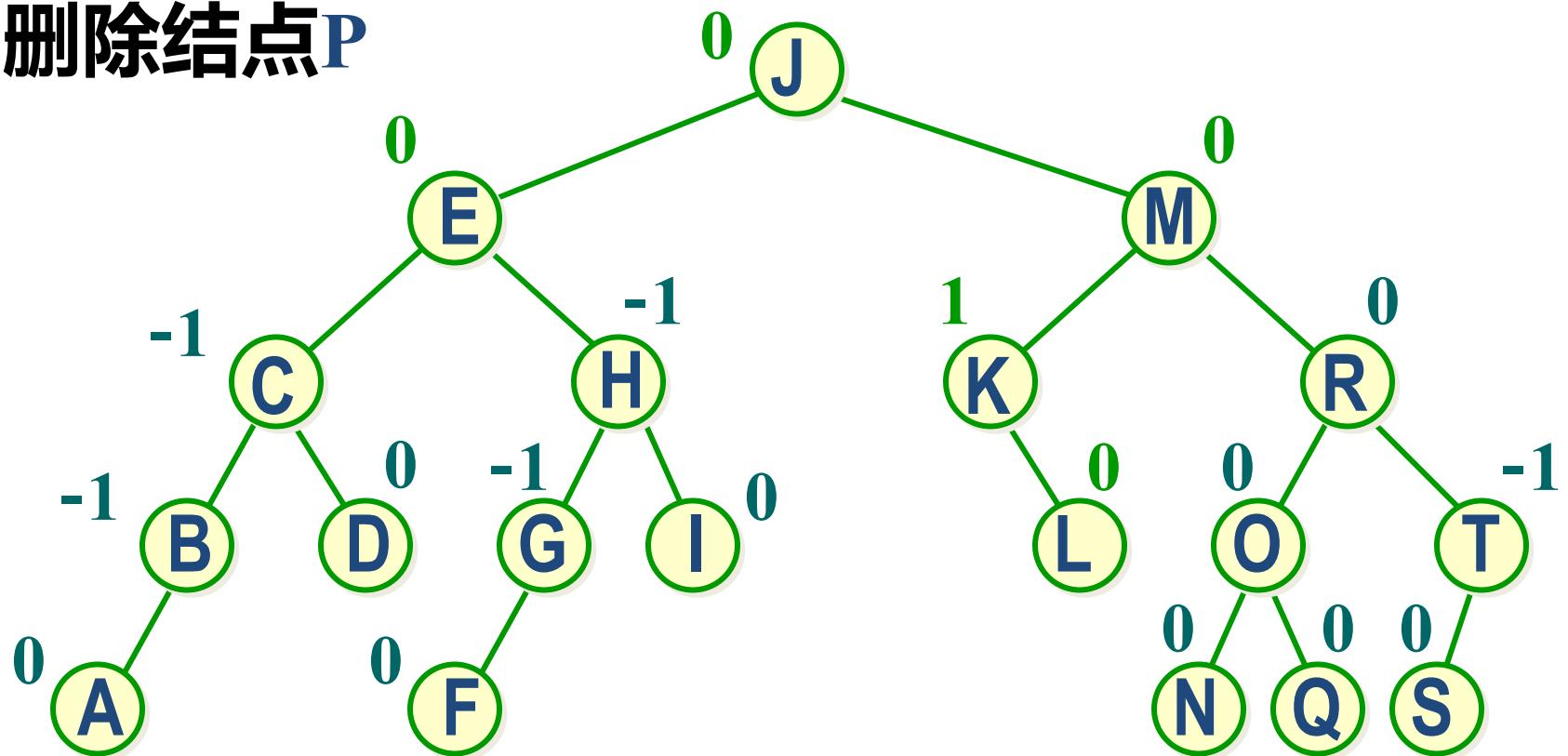
删除结点P



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

- 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)
 - ✓ $N_h = N_{h-1} + N_{h-2} + 1, h > 1$



- ◆ 可以证明,对于 $h \geq 0$, 有 $N_h = F_{h+2} - 1$ 成立。

$$F_h \approx \left(\frac{1+\sqrt{5}}{2}\right)^h / \sqrt{5}, \text{ 则有 } N_h \approx \left(\frac{1+\sqrt{5}}{2}\right)^{h+2} / \sqrt{5} - 1$$

- ◆ 有 n 个结点的AVL树的高度不超过

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

- ◆ 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。



作业

(1) 第 328 页, 4.13

明相关结点平衡因子的变化。

13. 图 7.40 是一棵 AVL 树, 画出从树中删除 22、3、10、9 后树的形态和旋转的类型。要
求以被删关键码的中序下的直接前驱替补该被删关键码。

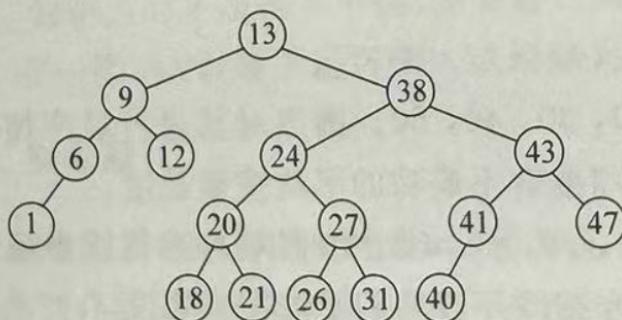


图 7.39 第 12 题的图

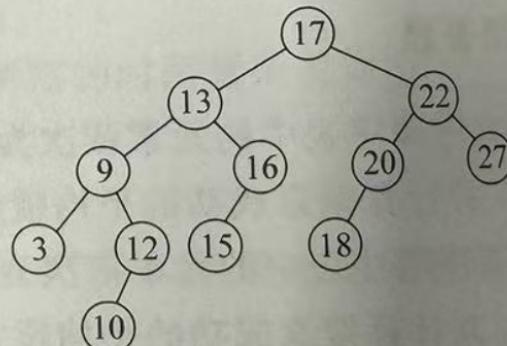


图 7.40 第 13 题的图



作业

(2) 第 327 页, 4.8

看，又。此即题是否正确？取小元素和最大元素一定是叶结点吗？

8. 将{55, 31, 11, 37, 46, 73, 63, 2, 7}中的关键码依次插入初始为空的二叉搜索树中,画出所得到的树 T 。然后画出删除 37 之后的二叉搜索树 T' 。若再将 37 插入 T' 中得到的二叉搜索树 T'' 是否与 T' 相同?



作业

(3) 第 325 页, 2.5 和第 325 页, 2.8

4. 使用折半搜索算法在有 n 个元素的有序表中查找一个元素，其搜索长度为 _____。

5. 从有序顺序表 {12, 18, 30, 43, 56, 78, 82, 95} 中折半搜索元素 56 时，其搜索长度为 _____。

6. 假定对长度 $n = 50$ 的有序表进行折半搜索，则对应的二叉判定树中最下一层的结点数为 _____ 个。

7. 从一棵二叉搜索树中搜索一个元素时，若给定值小于根结点的值，则需要向继续搜索。

8. 从一棵二叉搜索树中搜索一个元素时，若给定值大于根结点的值，则需要向继续搜索。

9. 在一棵二叉搜索树中插入一个新元素时，若该新元素的值小于根结点的值，则应把插入结点作为左子结点；若该新元素的值大于根结点的值，则应把插入结点作为右子结点。则应把插入结点作为右子结点。

