第九章 查找

- 9.1 静态查找表
- 9.2 动态查找表
- 9.3 哈希表

查找(Search)的概念

- 查找　就是在数据集合中寻找满足某种条件的数据元素。
  - 查找成功，即找到满足条件的数据元素。查找结果可为该元素在结构中的位置。
  - 查找不成功，即找不到满足条件的数据元素。查找结果为一些指示信息，如失败标志、失败位置等。
- 关键字：元素中可标识该元素的属性。使用基于主关键字的查找，查找结果应是唯一的。
- 静态查找表：数据结构在执行查找操作的前后不发生改变的查找表。
- 动态查找表：数据结构在执行查找操作时，可同时进行插入和删除等操作（结构可能发生变化）的查找表。

9.1 静态查找表

- 在静态查找表中，数据元素存放于线性表中。查找算法根据给定值x，在线性表中进行查找。直到找到x在线性表中的存放位置或可确定在线性表中找不到x为止。
- 静态查找表的类型定义：

```
typedef struct
{ ElemType *elem;
    int length;
}SSTable;
```

9.1 静态查找表

9.1.1 顺序查找

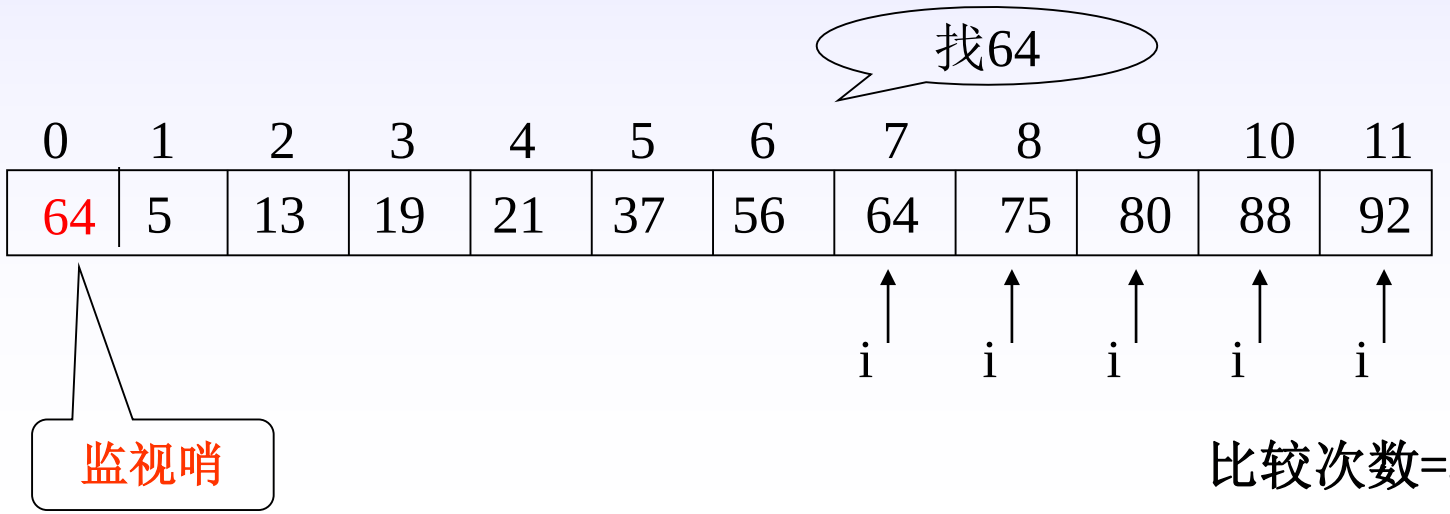
- 所谓顺序查找，又称线性查找，主要用于在线性结构中进行查找。
- 设若表中有*n*个元素，则顺序查找从表的一端开始，顺序用各元素的关键字与给定值*x*进行比较，直到找到与其值相等的元素，则查找成功，给出该元素在表中的位置。
- 若整个表都已检测完仍未找到关键字与*x*相等的元素，则查找失败。给出失败信息。

9.1 静态查找表

- 顺序查找算法：

```
int Search_Seq(SSTable ST, KeyType key)
// 查找成功，返回非0；否则返回0。
{  ST.elem[0].key=key;   //0号单元为监视哨
    for(i=ST.length; !EQ(ST.elem[i].key, key); i--);
    return i;
}
```

例



9.1 静态查找表

顺序查找的平均查找长度

- 衡量一个查找算法的时间效率的标准是：在查找过程中关键字的平均比较次数，这个标准也称为平均查找长度*ASL*(*Average Search Length*)，通常它是查找元素总数*n*的函数。设查找第i个元素的概率为pi，查找到第i个元素所需比较次数为ci，则查找成功的平均查找长度：

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

- 在顺序查找情形，$c_i = n-i+1,\ i = 1, 2, \ldots, n$，因此

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

- 在等概率情形，$p_i = 1/n,\ i = 1, 2, \ldots, n$。

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

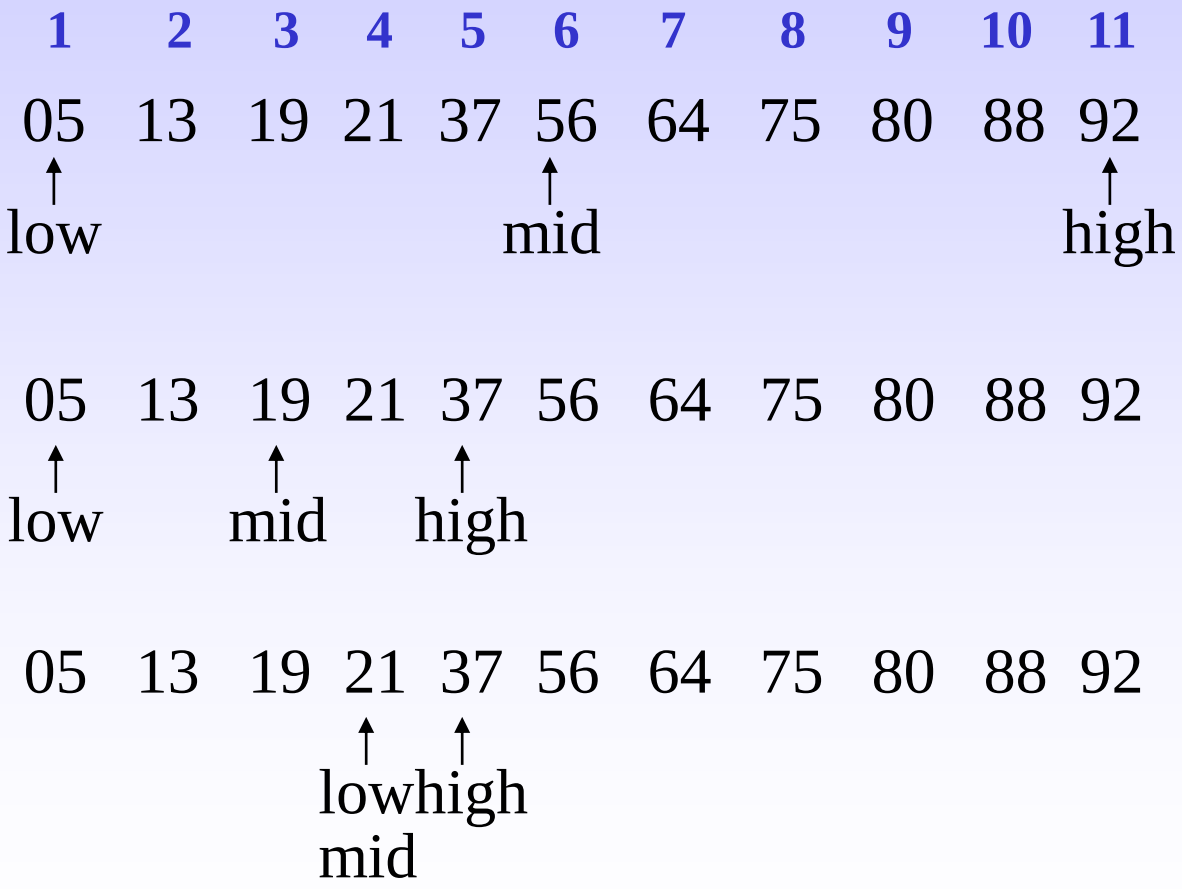
9.1 静态查找表

9.1.2 折半查找

- 设*n*个元素存放在一个有序顺序表中，并按其关键字从小到大排好了序。
- 采用折半查找时，先求位于查找区间正中的元素的下标*mid*，用其关键字与给定值*x*比较:
  - *elem*[*mid*].key = *x*，查找成功；
  - *elem*[*mid*].key > *x*，把查找区间缩小到表的前半部分，再继续进行折半查找；
  - *elem*[*mid*].key < *x*，把查找区间缩小到表的后半部分，再继续进行折半查找。
- 每比较一次，查找区间缩小一半。如果查找区间已缩小到一个元素，仍未找到想要查找的元素，则查找失败。

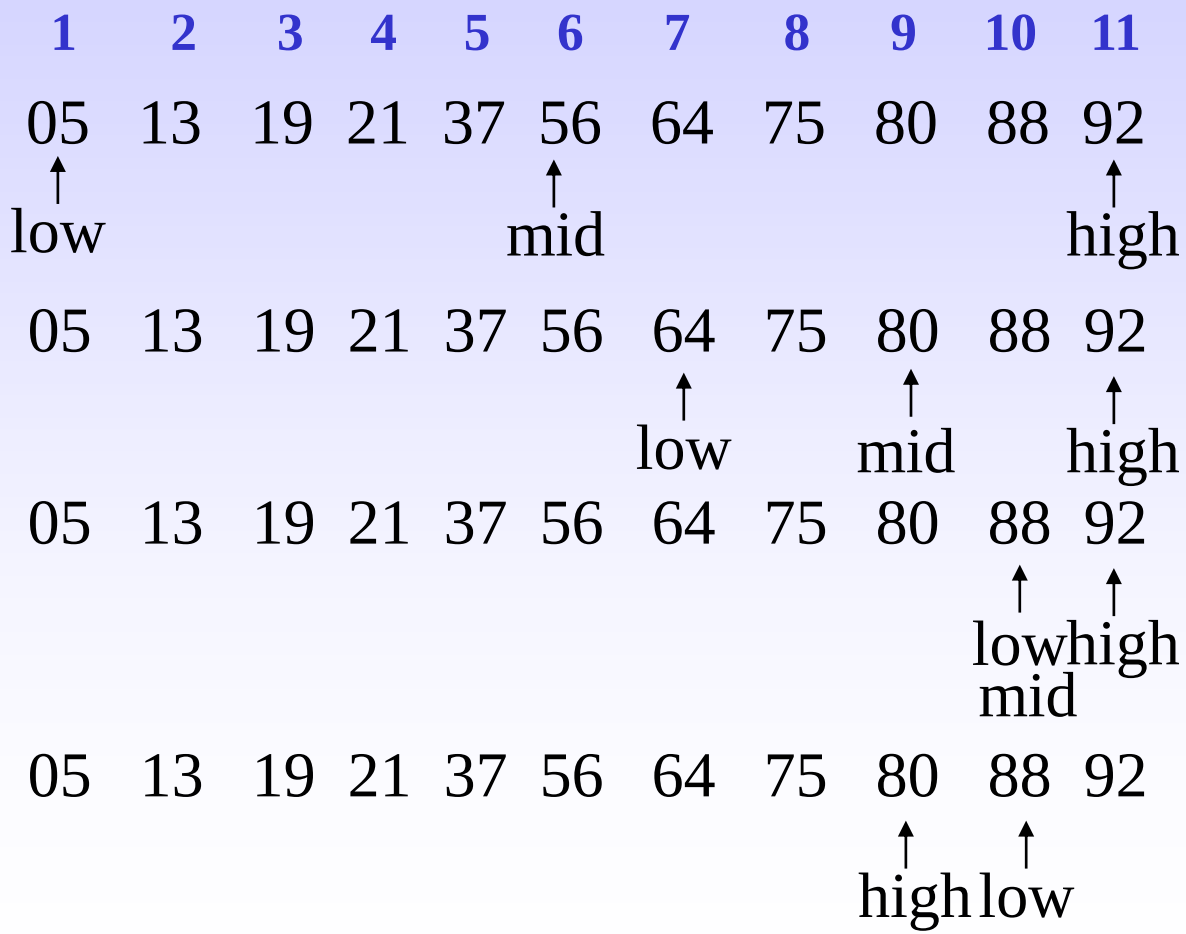
9.1 静态查找表

例 Key=21的查找过程：



9.1 静态查找表

例 Key=85的查找过程：



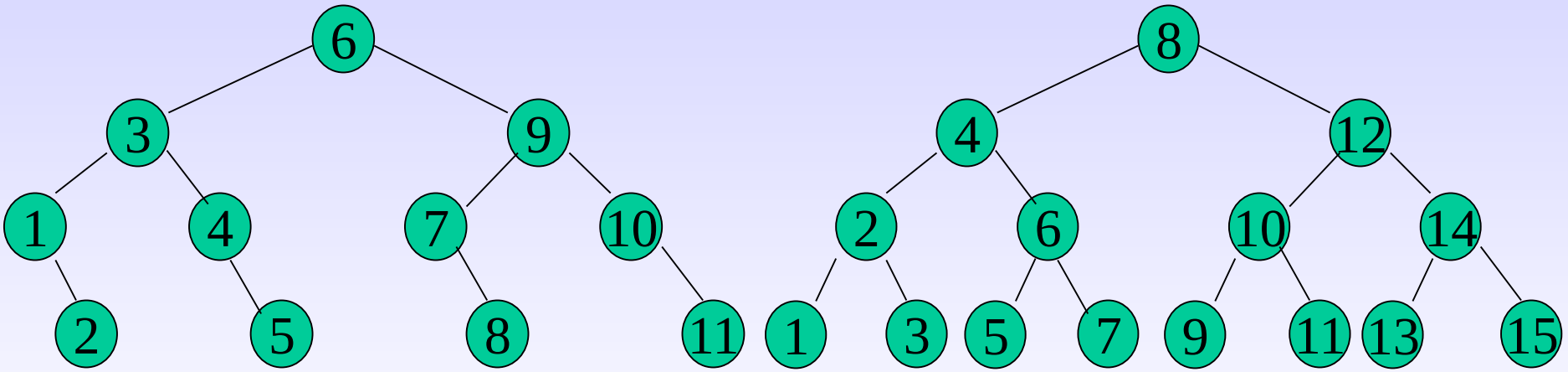
9.1 静态查找表

折半查找的算法：

```
int Search_Bin(SSTable ST, KeyType key)
{  low=1;  high=ST.length;
   while( low<=high )
   {   mid= (low+high)/2;
       if  EQ(ST.elem[mid].key, key)
                return mid;
       else if LT(key, ST.elem[mid].key)
                high=mid-1;
       else   low=mid+1;
   }
   return 0;
}
```

9.1 静态查找表

- 从有序表构造出的二叉查找树(判定树)



- 若设 $n = 2^h-1$，则描述折半查找的二叉查找树是深度为 h 的满二叉树。$2^h = n+1$, $h = \log_2(n+1)$。

9.1 静态查找表

- 第1层结点有1个，查找第1层结点要比较1次；
- 第2层结点有2个，查找第2层结点要比较2次；
- …；
- 第 i $(1 \le i < h)$ 层结点有 2^{i-1} 个，查找第 i 层结点要比较 i次，…。
- 假定每个结点的查找概率相等，即$pi = 1/n$，则查找成功的平均查找长度为：

9.1 静态查找表

$$ASL_{succ}=\sum_{i=1}^{n}P_i\cdot C_i=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\sum_{j=1}^{h}j\cdot 2^{j-1}$$

由 $\sum_{j=1}^{h}j\cdot 2^{j-1}=1+2\cdot 2+3\cdot 2^2+\ldots+h\cdot 2^{h-1}$ (1)

及 $2\cdot\sum_{j=1}^{h}j\cdot 2^{j-1}=2+2\cdot 2^2+3\cdot 2^3\ldots+h\cdot 2^h$ (2)

(2)−(1)得 $\sum_{j=1}^{h}j\cdot 2^{j-1}=h\cdot 2^h-(1+2+2^2+\ldots+2^{h-1})$

$$=h\cdot 2^h-2^h+1$$

$$ASL_{succ}=\frac{1}{n}((h-1)\times 2^h+1)=\frac{1}{n}((n+1)\times(\log_2(n+1)-1)+1)$$

$$=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)-1$$

9.1 静态查找表

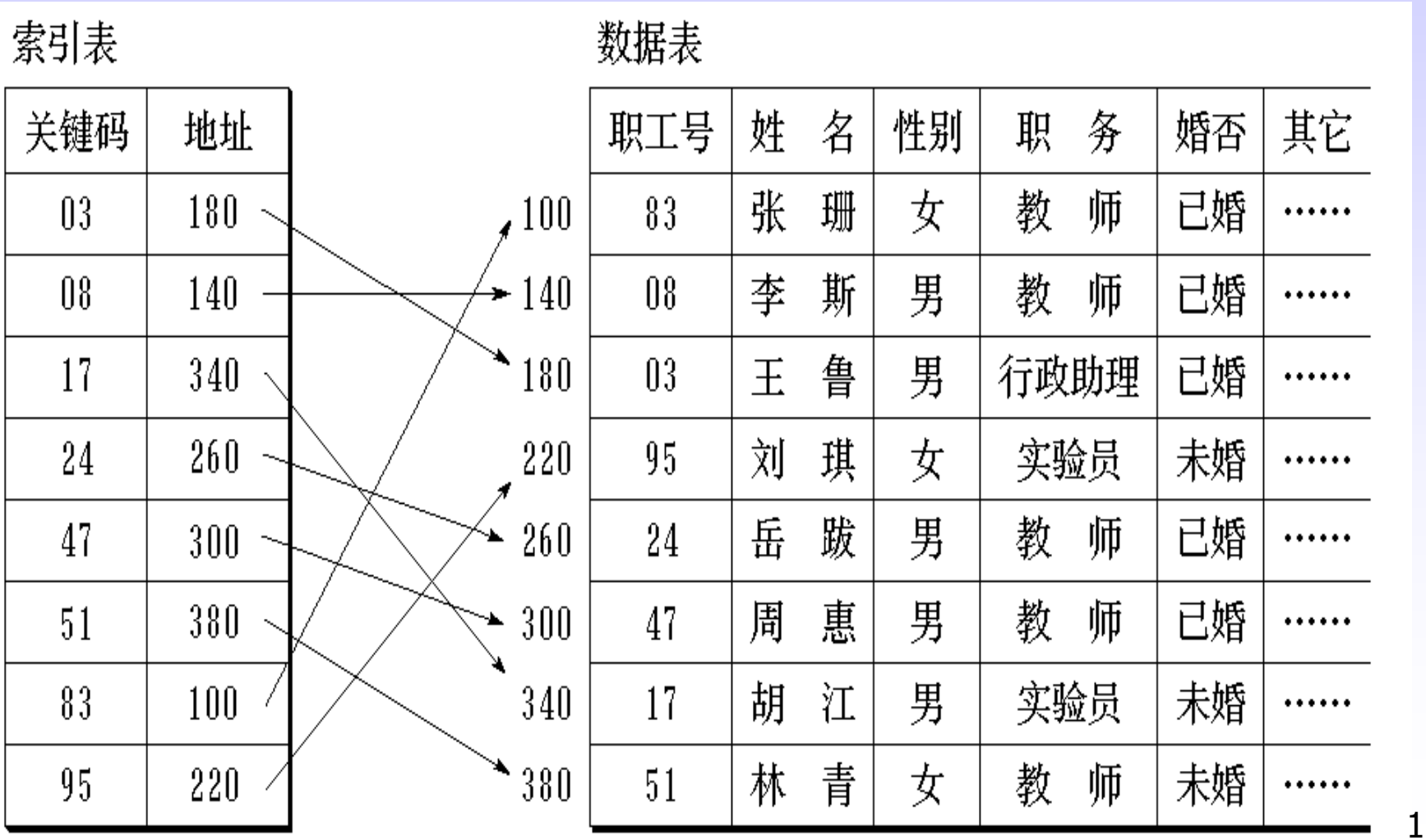
9.1.3 索引查找

当数据元素个数 n 很大时，如果用无序表形式的静态查找结构存储，采用顺序查找，则查找效率较低。如果采用有序表存储形式的静态查找结构，则插入新记录进行排序，时间开销也很可观。这时可采用索引方法来实现存储和查找。

- 稠密索引：一个索引项对应数据表中一个元素的索引结构。当元素在外存中按加入顺序存放而不是按关键字有序存放时必须采用稠密索引结构，这时的索引结构叫做索引非顺序结构。

9.1 静态查找表

示例：有一个存放职工信息的数据表，每一个职工对象有近 1k 字节的信息。



9.1 静态查找表

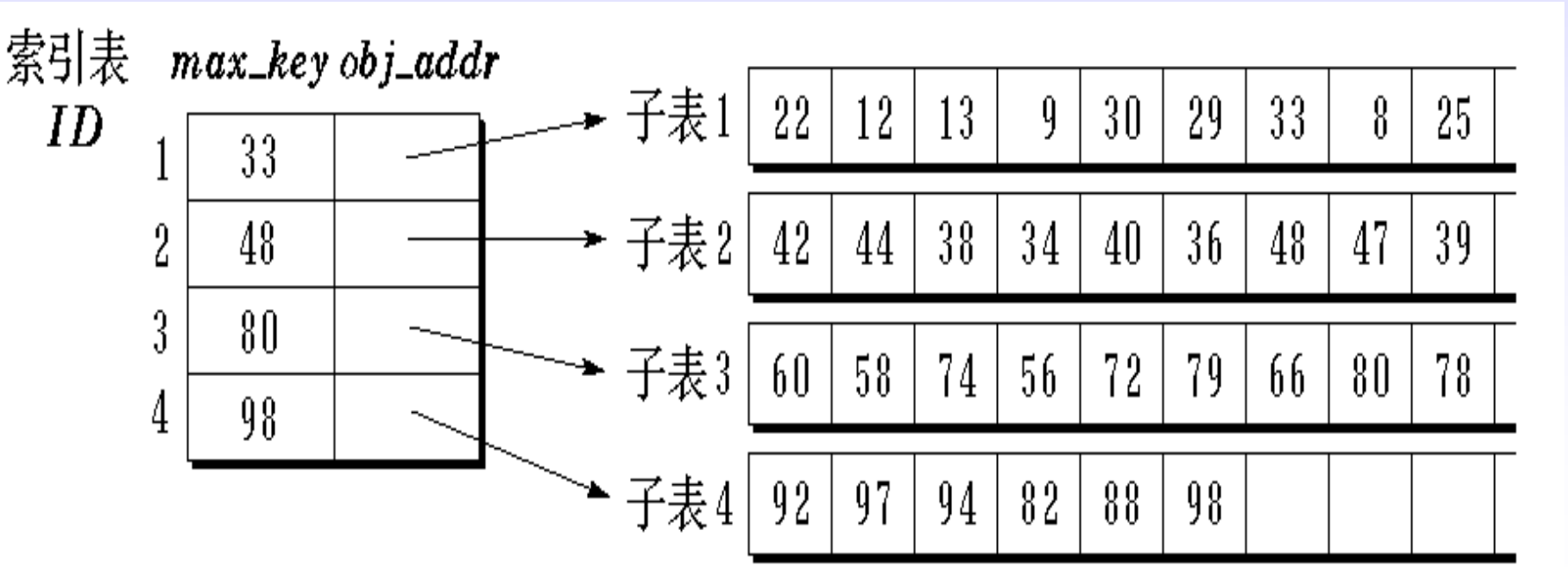
- 假设内存工作区仅能容纳 64k 字节的数据，在某一时刻内存最多可容纳 64 个元素以供查找。
- 如果元素总数有 1024 个，不可能把所有元素的数据一次都读入内存。无论是顺序查找或对分查找，都需要多次读取外存记录。
- 如果在索引表中每一个索引项占4个字节, 每个索引项索引一个职工对象，则 1024 个索引项需要 4k 字节，在内存中可以容纳所有的索引项。
- 这样只需从外存中把索引表读入内存，经过查找索引后确定了职工对象的存储地址，再经过 1 次读取元素操作就可以完成查找。

9.1 静态查找表

- 稀疏索引：把所有 n 个元素分为 b 个子表(块)存放，一个索引项对应数据表中一组元素(一个子表)。
- 在子表中，所有元素可能按关键字有序地存放，也可能无序地存放。但所有这些子表必须分块有序，后一个子表中所有元素的关键字均大于前一个子表中所有元素的关键字。它们都存放在数据区中。另外建立一个索引表。
- 各个索引项在索引表中的序号与各个子表的块号有一一对应的关系（第 i 个索引项是第 i 个子表的索引项，$i = 0, 1, \ldots, n-1$），这样的索引结构叫做索引顺序结构。

9.1 静态查找表

索引表由若干 个索引项组成，第 *i* 个索引项记录了子表 *i* 中最大关键字***max_key***以及该子表在数据区中的起始位置***obj_addr***。

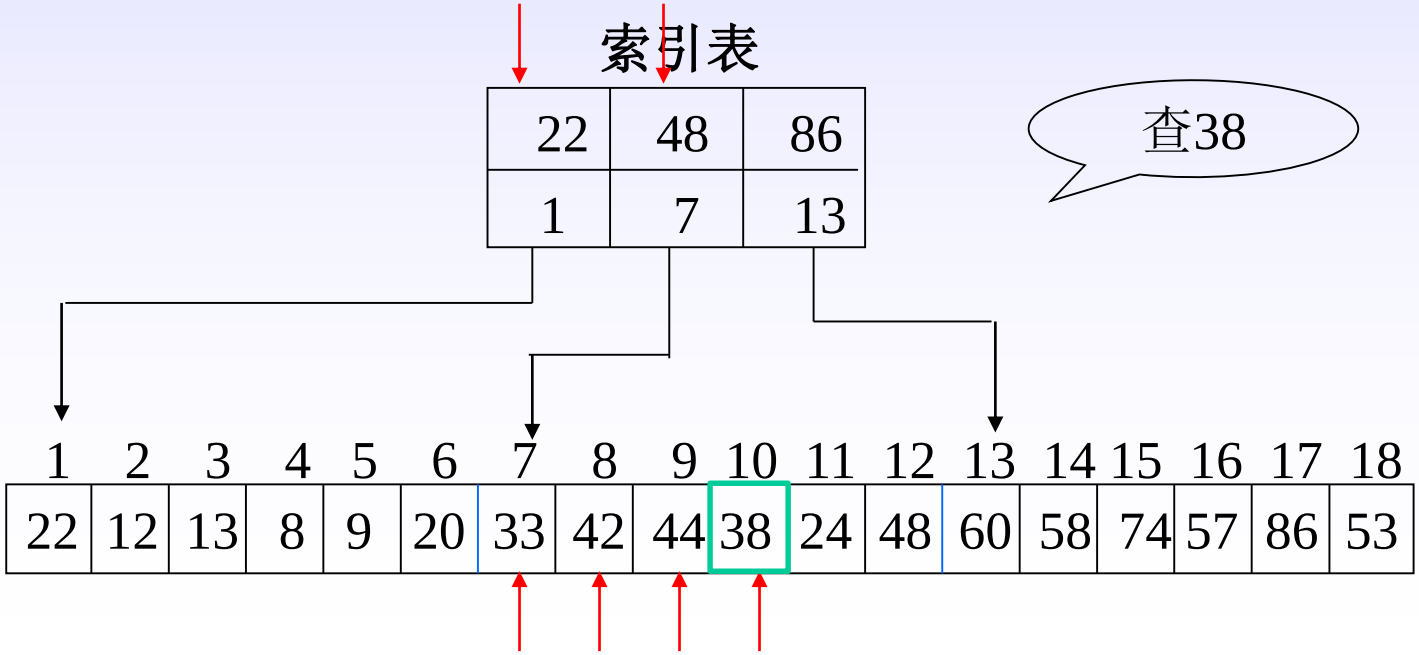


9.1 静态查找表

- 对索引顺序结构进行查找时，一般分为两级查找：

① 先在索引表 *ID* 中查找给定值 *K*，确定满足 *ID*[*i*−1].*max_key* < *K* ≤ *ID*[*i*].*max_key* 的 *i* 值，即待查元素可能在的子表的序号。

② 然后再在第 *i* 个子表中按给定值查找要求的元素。



9.1 静态查找表

- 索引表是按*max_key*有序的，且长度也不大，可以折半查找，也可以顺序查找。
- 各子表内各个元素如果也按元素关键字有序，可以采用折半查找或顺序查找；如果不是按元素关键字有序，只能顺序查找。
- 索引顺序查找的查找成功时的平均查找长度

$$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$$

- 其中，ASL_{Index} 是在索引表中查找子表位置的平均查找长度，$ASL_{SubList}$ 是在子表内查找元素位置的查找成功的平均查找长度。

9.1 静态查找表

- 设把长度为 n 的表分成均等的 b 个子表，每个子表 s 个元素，则 $b = \lceil n/s \rceil$。又设表中每个元素的查找概率相等，则每个子表的查找概率为$1/b$，子表内各元素的查找概率为 $1/s$。
- 若对索引表和子表都用顺序查找，则索引顺序查找的查找成功时的平均查找长度为 $ASL_{IndexSeq} = (b+1)/2+(s+1)/2 = (b+s)/2 +1$
- 利用数学方法可以导出，当 $s = \sqrt{n}$ 时，$ASL_{IndexSeq}$取极小值 $\sqrt{n}+1$。这个值比顺序查找强，但比折半查找差。
- 若采用折半查找确定元素所在的子表，则查找成功时的平均查找长度为 $ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$

  $\approx \log_2 (b+1)-1 + (s+1)/2 \approx \log_2(1+n/s) + s/2$

9.1 静态查找表

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序、无序表	有序表	分块有序表
存储结构	顺序表、链表	顺序表	顺序表、链表

9.2 动态查找表

9.2.1 二叉排序树和平衡二叉树

一、二叉排序树及其查找过程

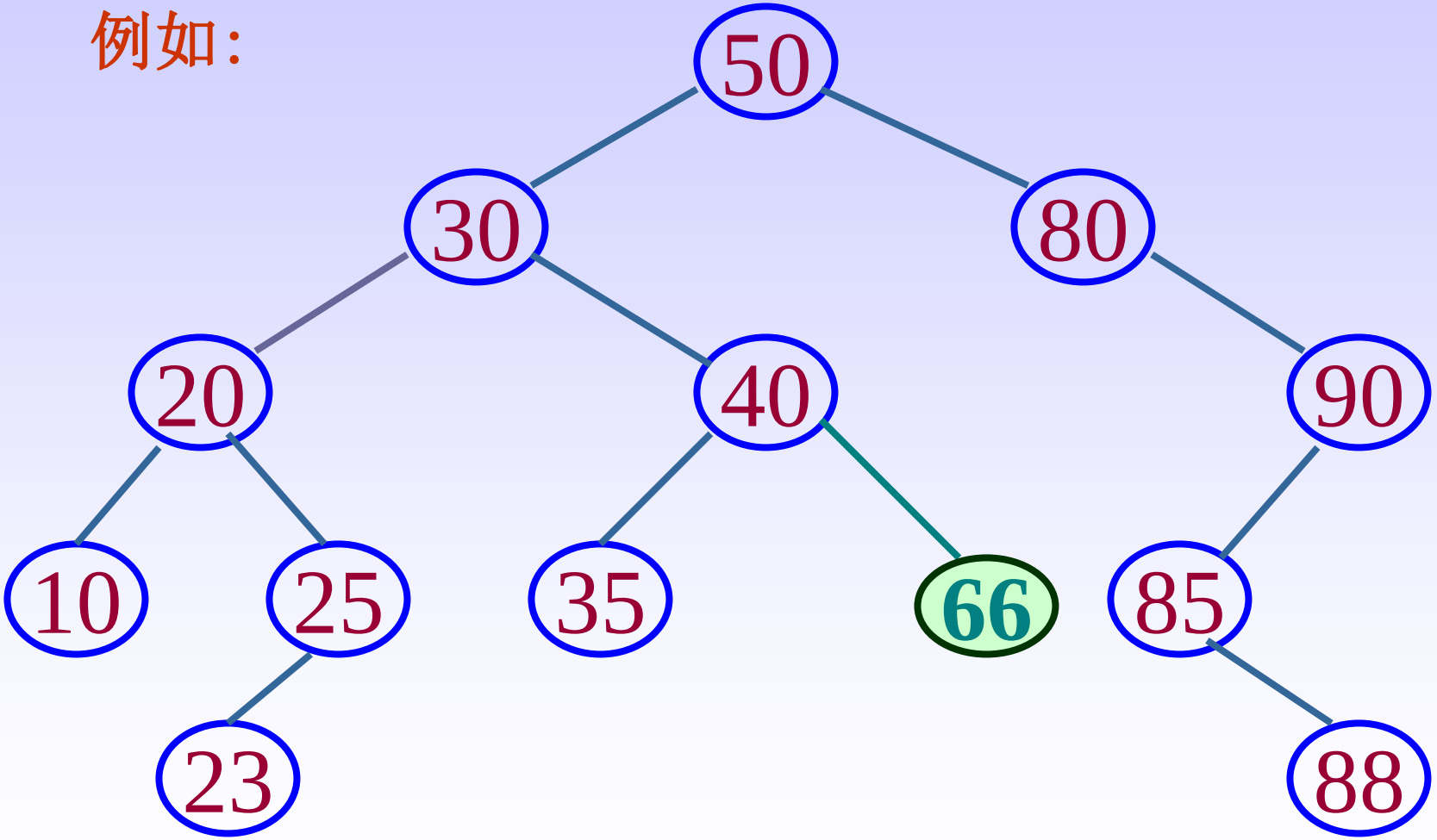
定义： 二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

- 左子树(如果存在)上所有结点的关键字都小于根结点的关键字。
- 右子树(如果存在)上所有结点的关键字都大于根结点的关键字。
- 左子树和右子树也是二叉排序树。

特点： 如果对一棵二叉排序树进行中序遍历，可以按关键字从小到大的顺序，将各结点排列起来，所以称为二叉排序树。

9.2 动态查找表

例如:



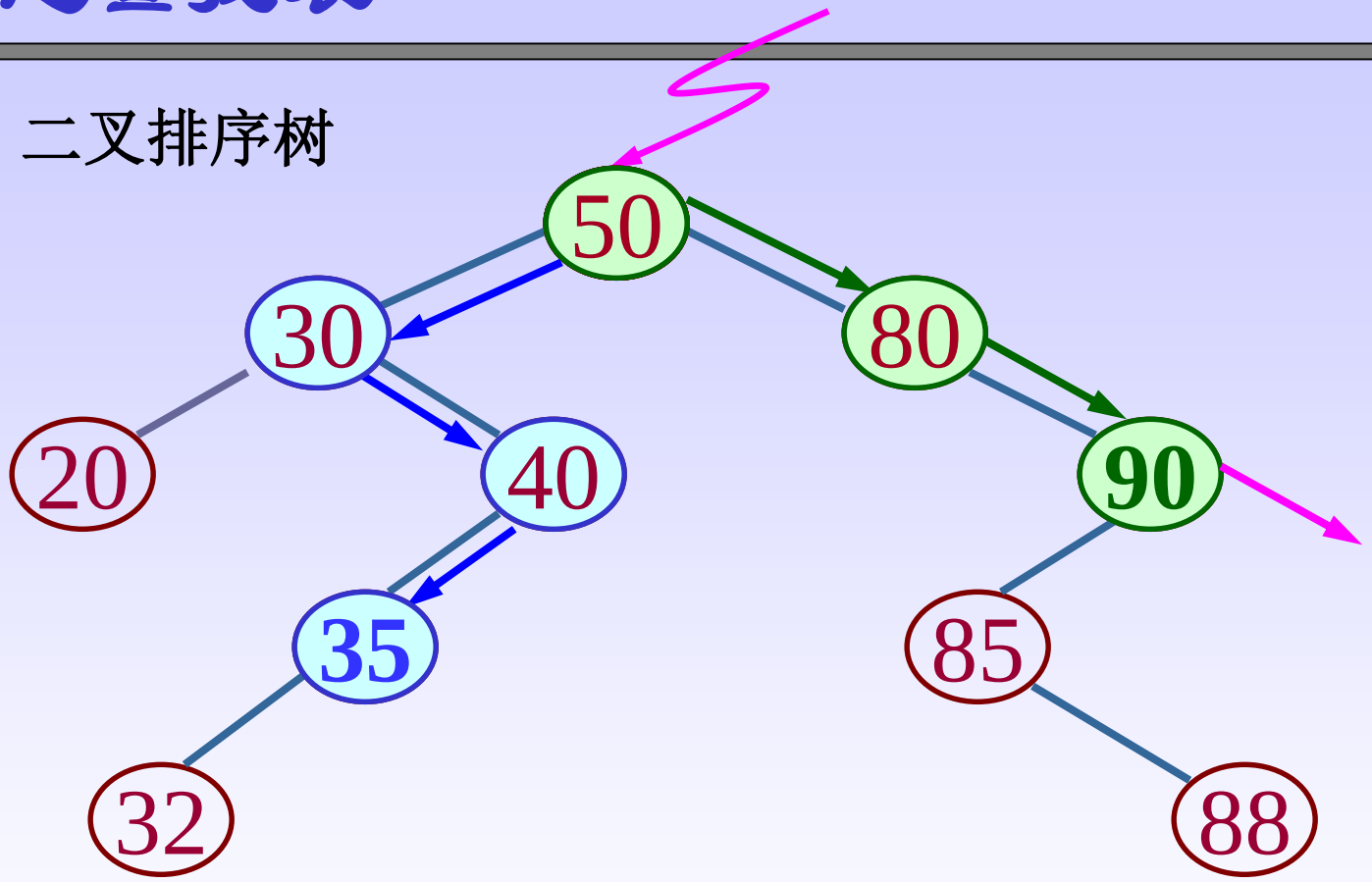
不 是二叉排序树。

9.2 动态查找表

- 在二叉排序树上进行查找，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，也可以是一个非递归的过程。
- 假设想要在二叉排序树中查找关键字为***x***的元素，查找过程从根结点开始。如果根指针为***NULL***，则查找不成功；否则用给定值***x***与根结点的关键字进行比较：
  - 如果给定值等于根结点的关键字，则查找成功。
  - 如果给定值小于根结点的关键字，则继续递归查找根结点的左子树；
  - 否则，递归查找根结点的右子树。

9.2 动态查找表

例如： 二叉排序树



查找关键字

== 50,　　35,　　90,　　95

9.2 动态查找表

递归查找算法

```
BiTree SearchBST(BiTree T, KeyType key)   //若失败，返回空指针
{   if(!T)|| EQ(key, T->data.key)  return(T);
    else if(LT(key, T->data.key))  return(SearchBST(T->lchild, key));
    else return(SearchBST(T->rchild, key));
}
```

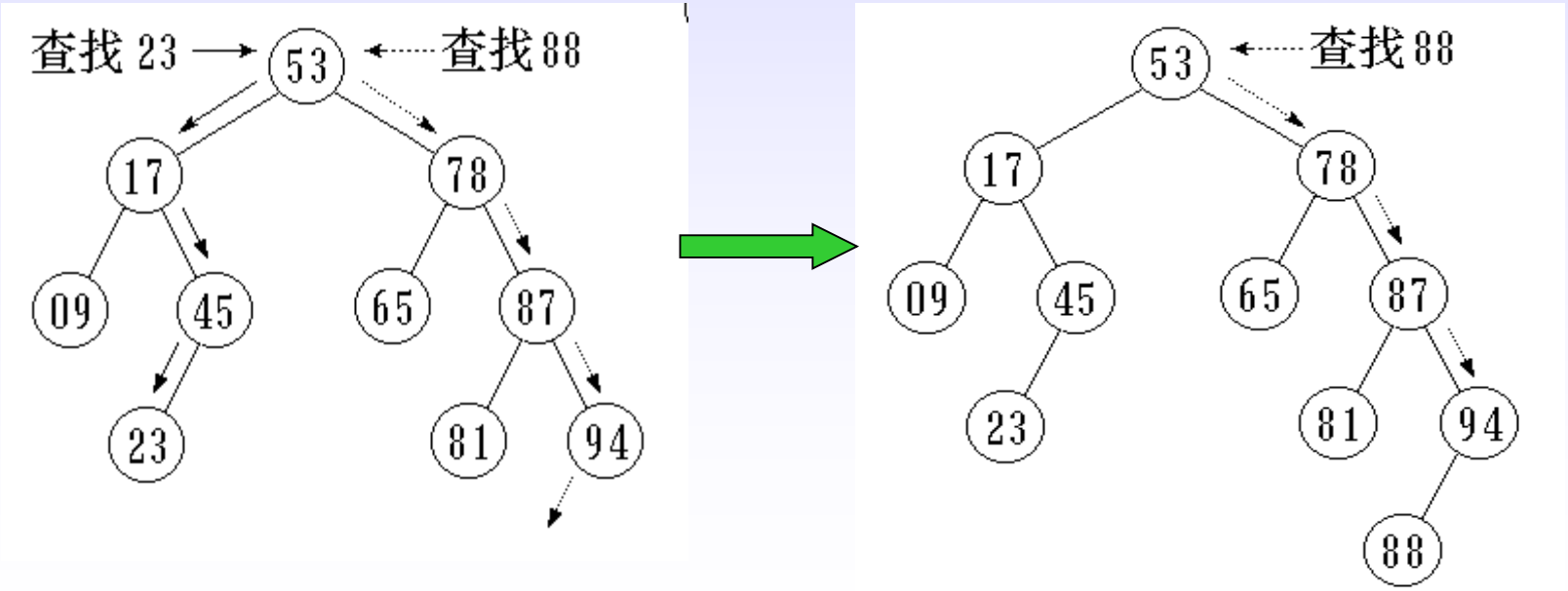
非递归查找算法

```
BiTree SearchBST( BiTree T, KeyType key )
{   p=T;       // 查找成功，返回指向该结点的指针，否则返回NULL。
    while( p && !EQ( key, p->data.key ))
        if( LT( key, p->data.key ))  p = p->lchild;
        else  p = p->rchild;
    return( p );
}
```

9.2 动态查找表

为了向二叉排序树中插入一个新元素，必须先使用查找算法检查这个元素是否在树中已经存在。

- 查找成功：树中已有这个元素，不再插入。
- 查找不成功：树中没有这个元素，把新元素加到查找操作停止的地方。



9.2 动态查找表

修改后的非递归查找算法

```
BiTree SearchBST(BiTree T, KeyType key, BiTree &pre)
{ //查找成功，返回指向要找的结点的指针，pre指向该结点的父结点；
  // 否则返回NULL，pre指向查找路径上的最后一个结点。
    p = T;
    pre = NULL;
    while(p && !EQ(key, p->data.key)
    {  pre = p;
       if(LT(key, p->data.key))  p = p->lchild;
       else  p = p->rchild;
    }
    return( p );
}
```

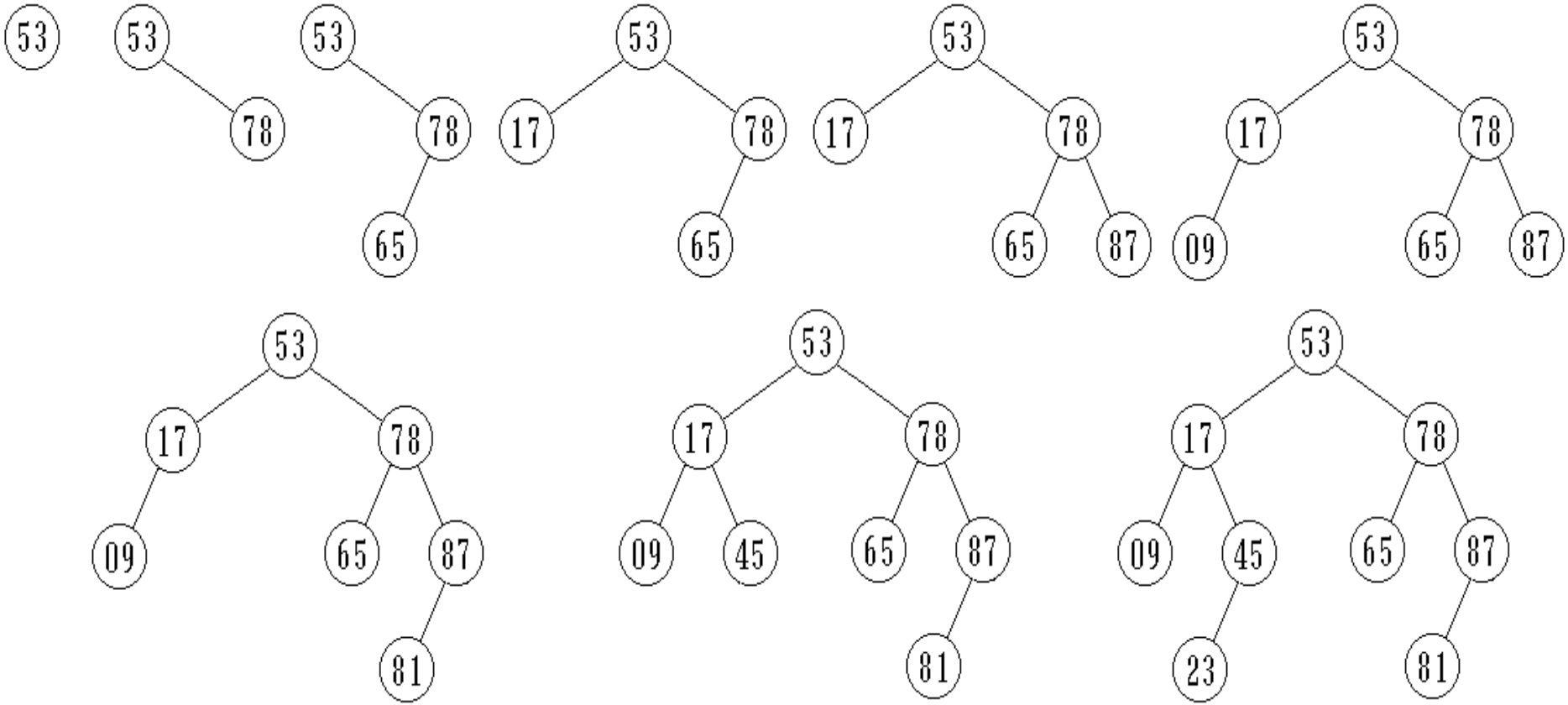
9.2 动态查找表

二叉排序树的插入算法

```
Status InsertBST( BiTree &T, ElemType e )
{  BiTree pre;
   if( !SearchBST( T, e.key, pre ))  // 查找不成功
   {   s = ( BiTNpde * )malloc(sizeof( BiTNode )); // 构造新结点
       s->data = e; s->lchild = s->rchild = NULL;
       if( !pre ) T = s;                    // 原树为空时，新结点为根
       else if( LT( e.key, pre->data.key ) )
                  pre->lchild = s;     // 新结点为左子结点
       else   pre->rchild = s;         // 新结点为右子结点
       return( OK );
   }
   return( FALSE );
}
```

9.2 动态查找表

- 输入数据序列｛ 53，78，65，17，87，09，81，45，23 ｝，建立二叉排序树的过程

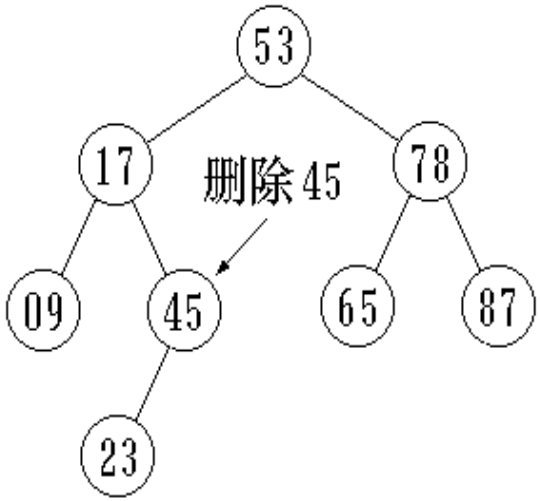


9.2 动态查找表

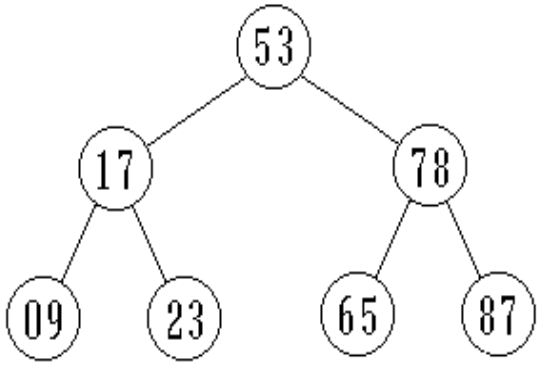
三、二叉排序树的删除

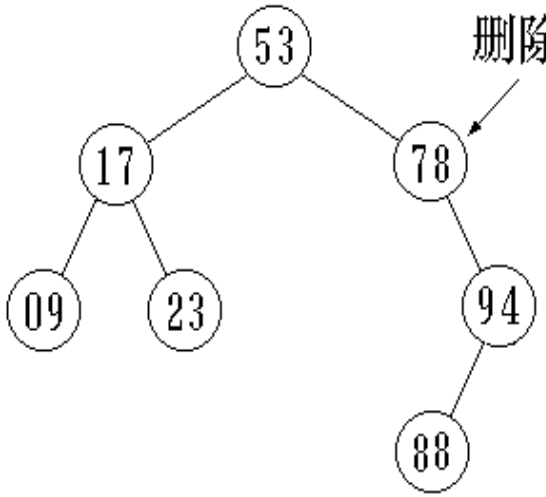
在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会失去。

- 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。
- 被删结点缺左（右）子树，可以拿它的右（左）子结点顶替它的位置，再释放它。
- 被删结点左、右子树都存在，可以在它的右子树中寻找中序遍历下的第一个结点(关键字最小，无左子树)，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。

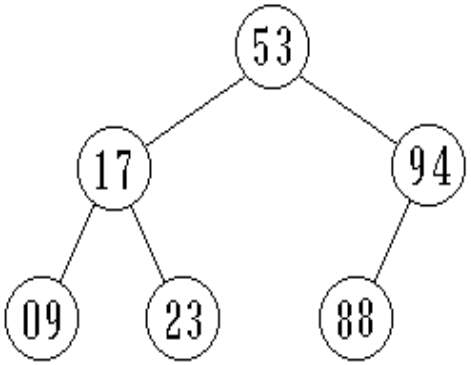


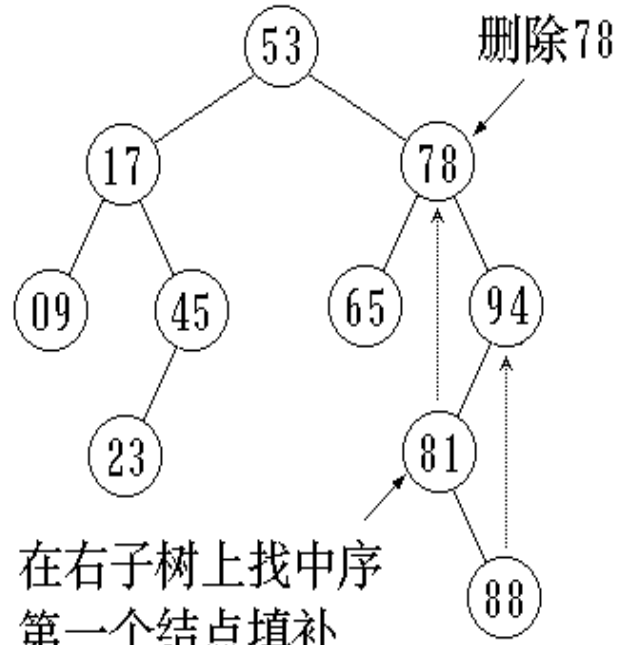
缺右子树用左子女填补



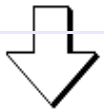


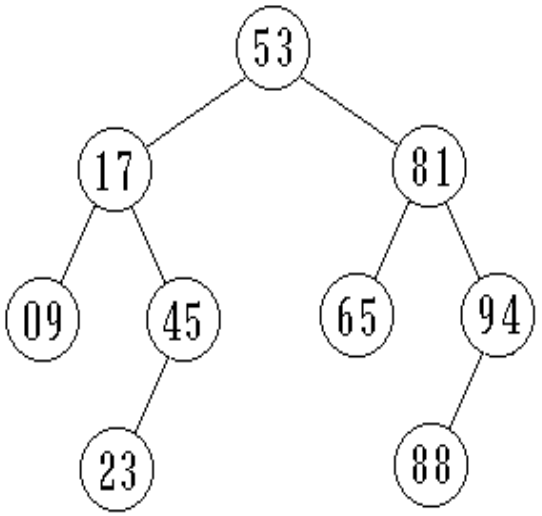
缺左子树用右子女填补





在右子树上找中序
第一个结点填补



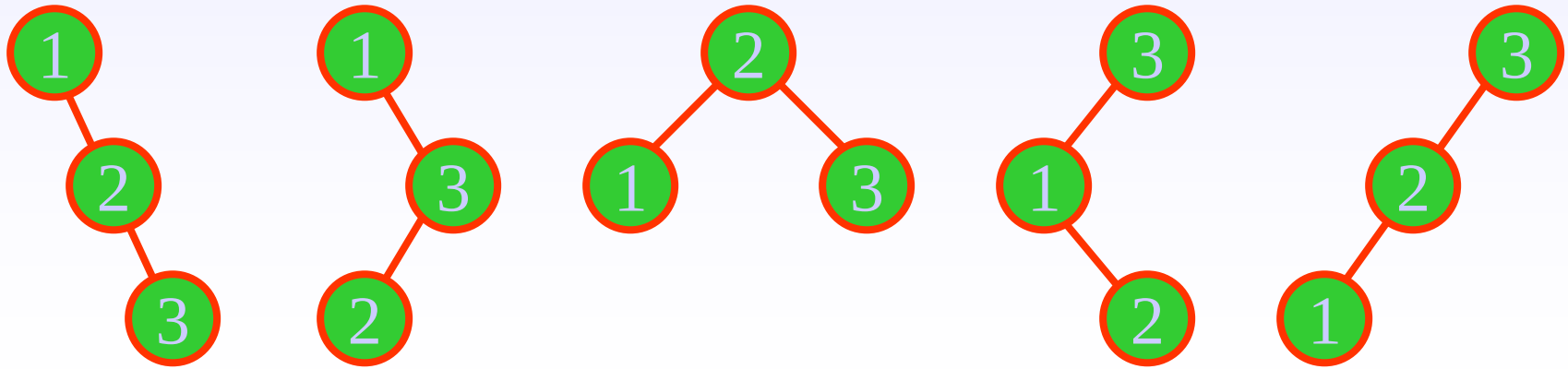


9.2 动态查找表

- BST的平均查找长度与它的形态有关，最好情况下与折半查找的平均查找长度一致，最坏情况下与顺序查找一致。
- 同样3个数据{1,2,3}，输入顺序不同，建立起来的二叉排序树的形态也不同，直接影响到二叉排序树的查找性能。

{2, 1, 3}

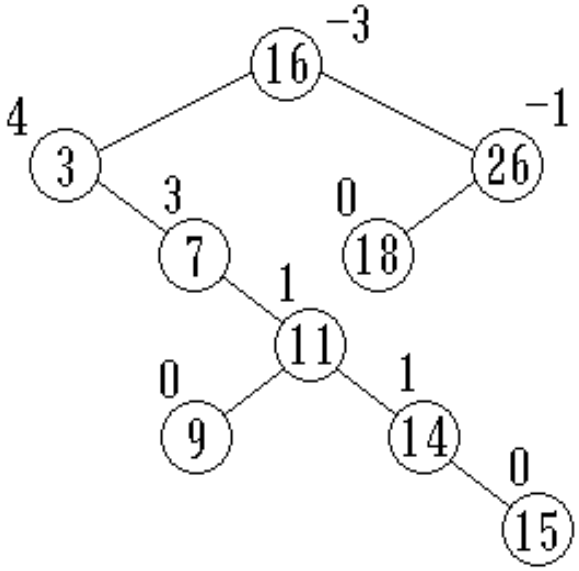
{1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}



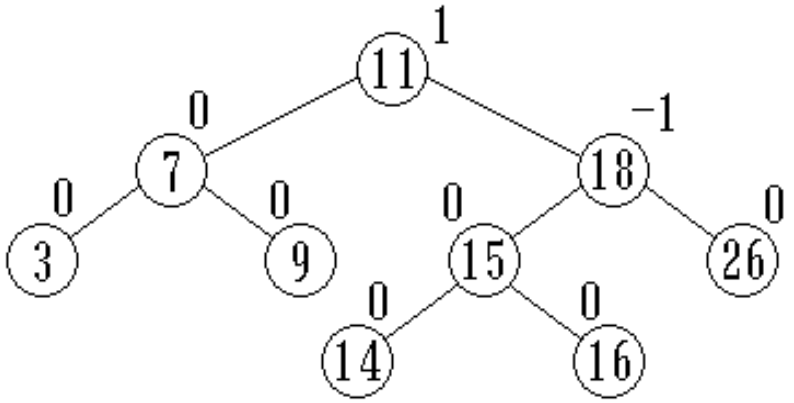
9.2 动态查找表

四、平衡二叉树（AVL树）

1 *AVL*树的定义：一棵***AVL***（**1962**年由**Adelson, Velskli**和**Landis**提出）树或者是空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是***AVL***树，且左子树和右子树的深度之差的绝对值不超过**1**。



深度不平衡的二叉排序树



深度平衡的二叉排序树

9.2 动态查找表

2 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深度减去左子树的深度（或左子树的深度减去右子树的深度）所得的深度差。这个数字即为结点的平衡因子***balance*** 。
- 根据***AVL***树的定义，任一结点的平衡因子只能取 **-1**，**0**和 **1**。
- 如果一个结点的平衡因子的绝对值大于**1**，则这棵二叉排序树就失去了平衡，不再是***AVL***树。
- 如果一棵二叉排序树是深度平衡的，它就成为 ***AVL***树。如果它有 ***n*** 个结点，其深度可保持在$O(\log_2 n)$，平均查找长度也可保持在$O(\log_2 n)$。

9.2 动态查找表

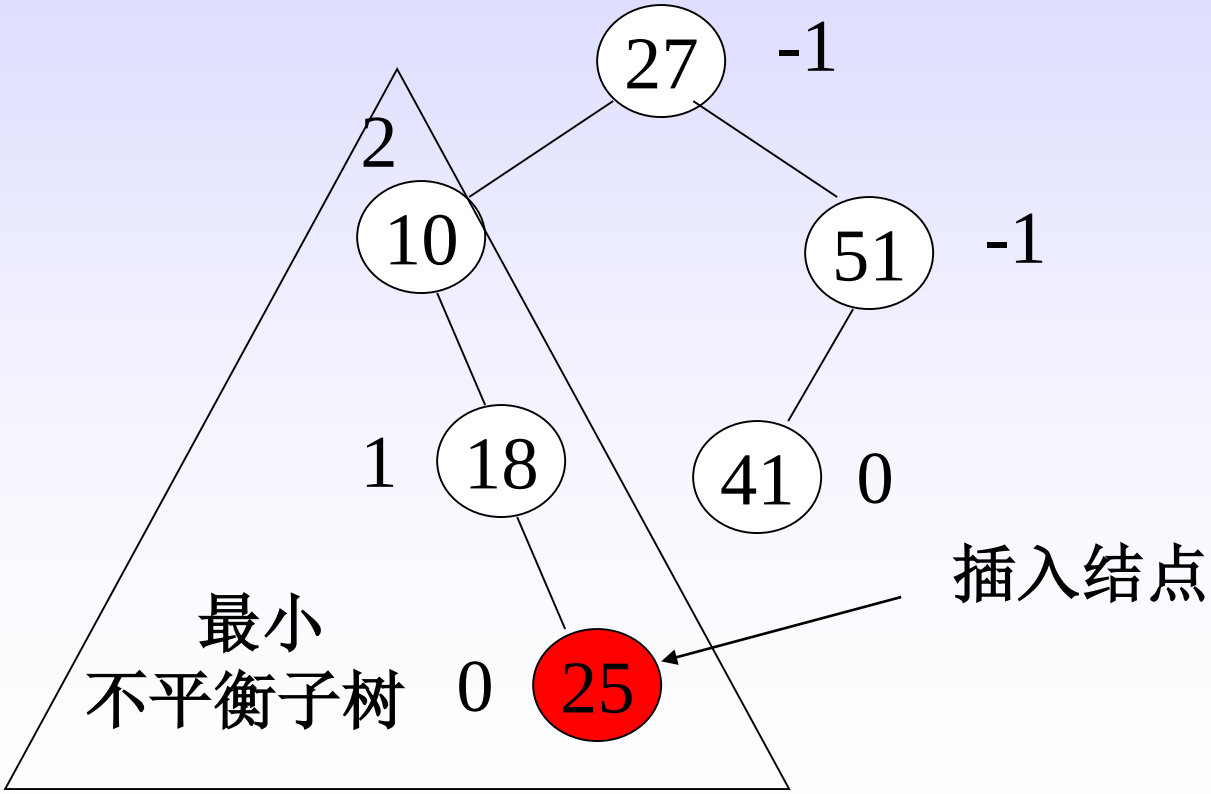
3 平衡化旋转：

- 如果在一棵平衡的二叉排序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。
- 必须保证平衡化后的二叉树依然是一棵二叉排序树。
- 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(左、右子树的深度差)。
- 如果在某一结点发现深度不平衡，停止回溯。
- 从发生不平衡的结点起，调整最小不平衡子树的结构。

9.2 动态查找表

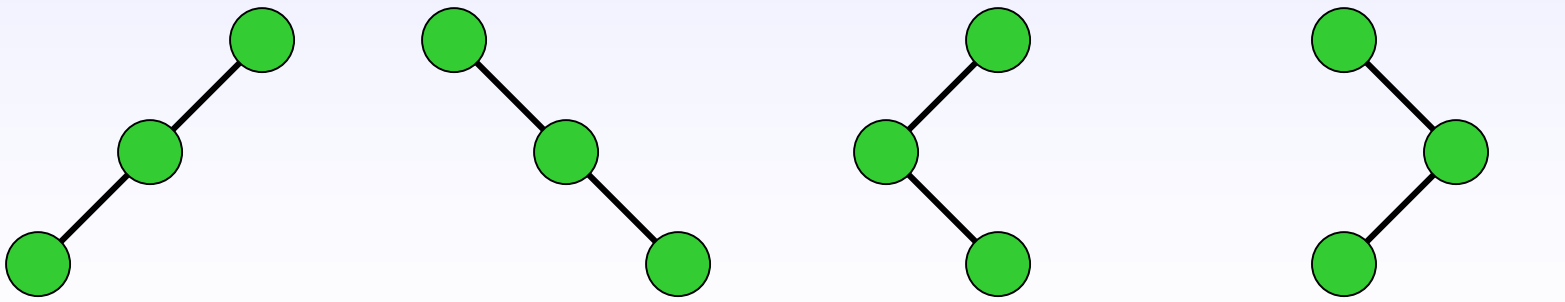
最小不平衡子树：

指离插入结点最近，且以平衡因子绝对值大于1的结点为根的子树。



9.2 动态查找表

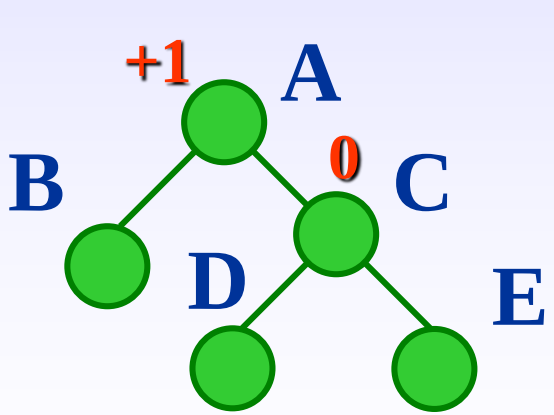
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。
  单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。
  双旋转分为先左后右和先右后左两类。



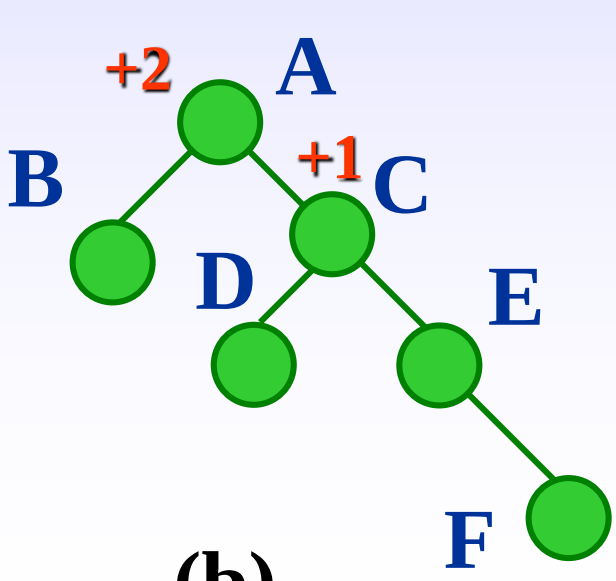
9.2 动态查找表

4 左单旋转 (RR型):

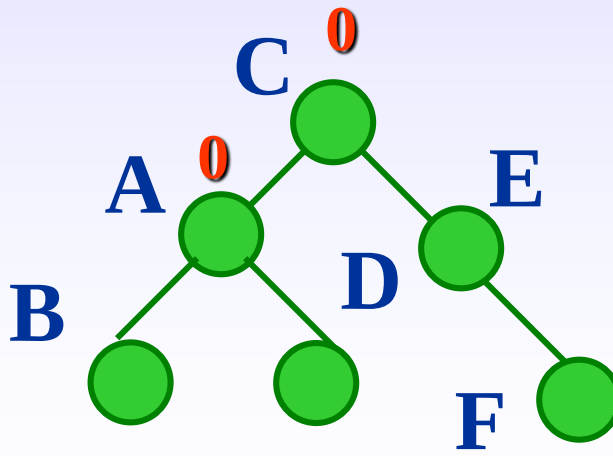
- 如果在子树E中插入一个新结点，该子树深度增1导致结点A的平衡因子变成+2，出现不平衡。
- 沿插入路径检查三个结点A、C和E。它们处于一条方向为“\”的直线上，需要做左单旋转。
- 以结点C为旋转轴，让结点A反时针旋转。



(a)



(b)

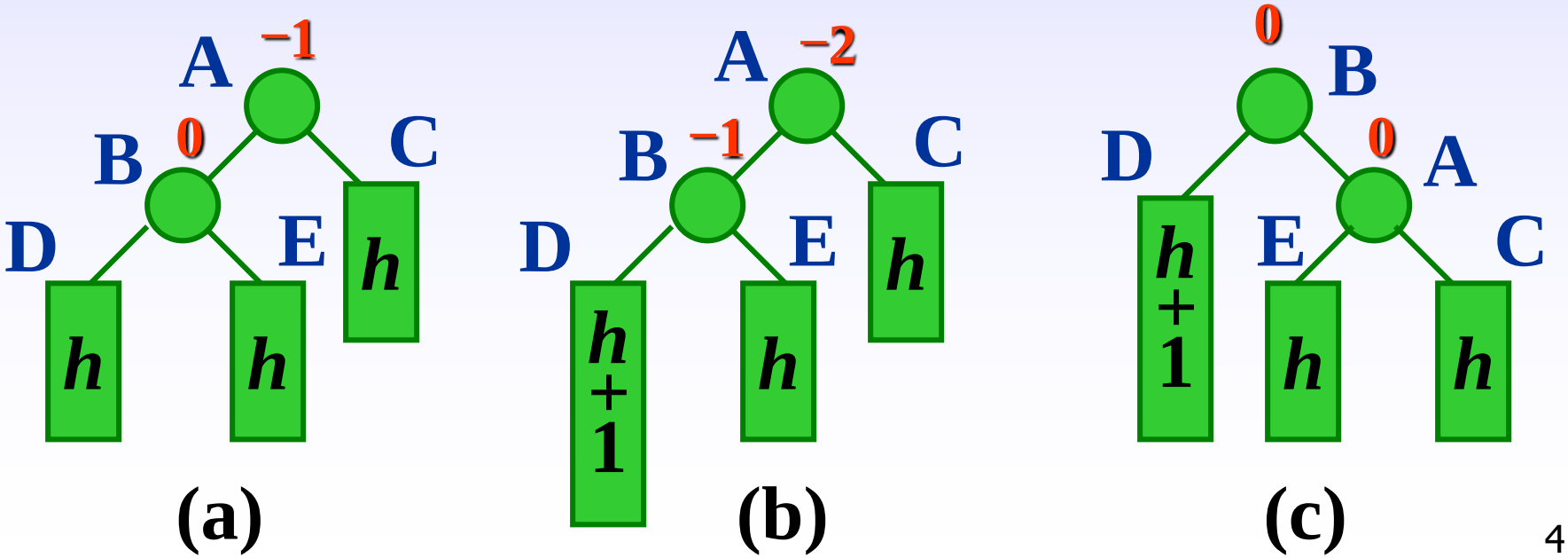


(c)

9.2 动态查找表

5 右单旋转 (LL型):

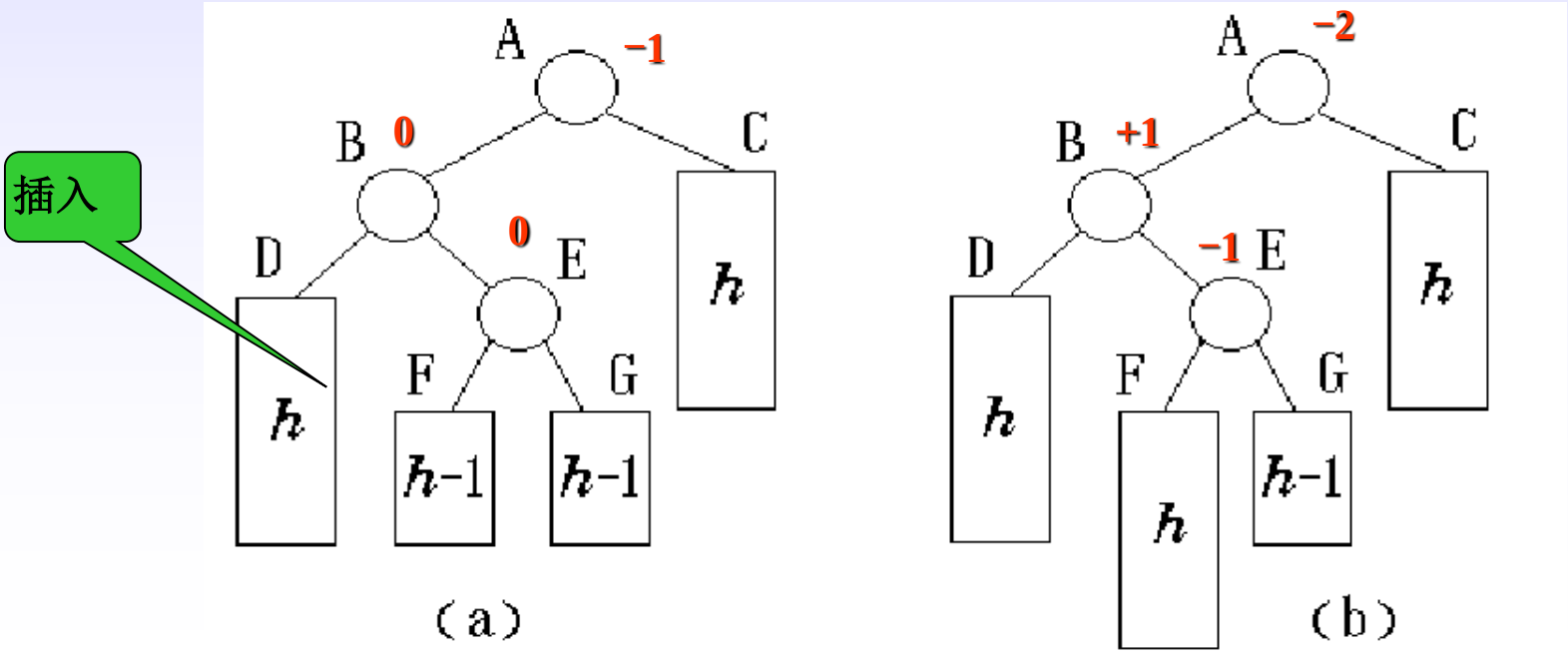
- 在左子树D上插入新结点使其深度增1，导致结点A的平衡因子增到 -2，造成了不平衡。
- 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和D，它们处于一条方向为“/”的直线上，需要做右单旋转。
- 以结点B为旋转轴，将结点A顺时针旋转。



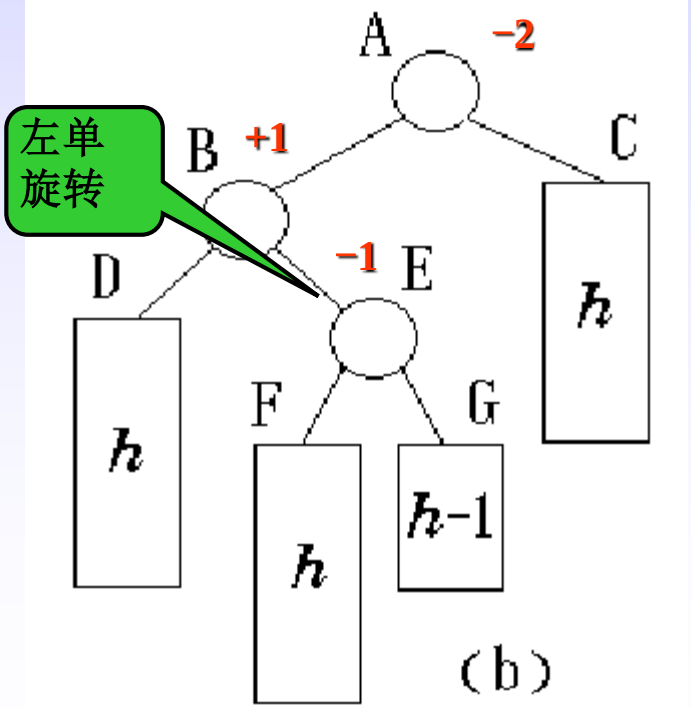
9.2 动态查找表

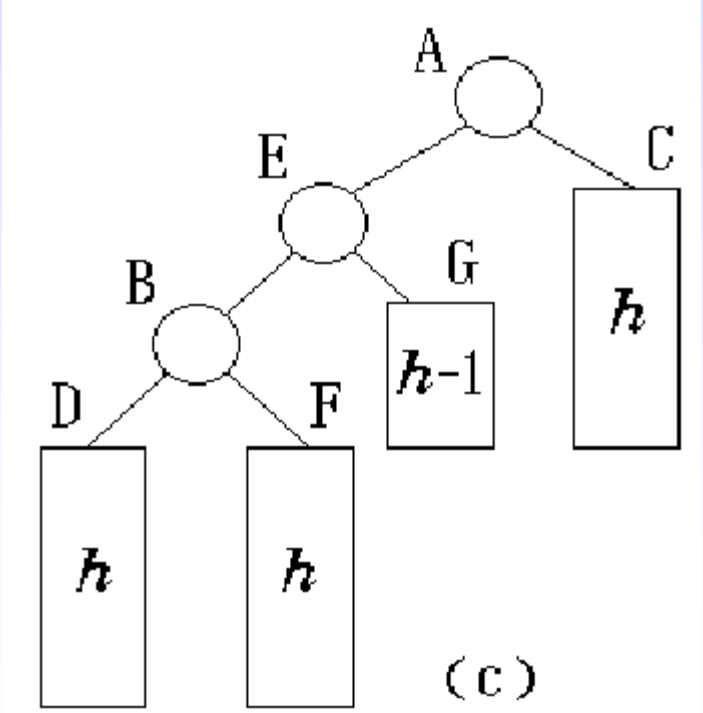
6 先左后右双旋转 (LR型):

- 在下图中子树**F**或**G**中插入新结点，该子树的深度增**1**。结点**A**的平衡因子变为 **-2**，发生了不平衡。
- 从结点**A**起沿插入路径选取**3**个结点**A**、**B**和**E**，它们位于一条形如“〈”的折线上，因此需要进行先左后右的双旋转。

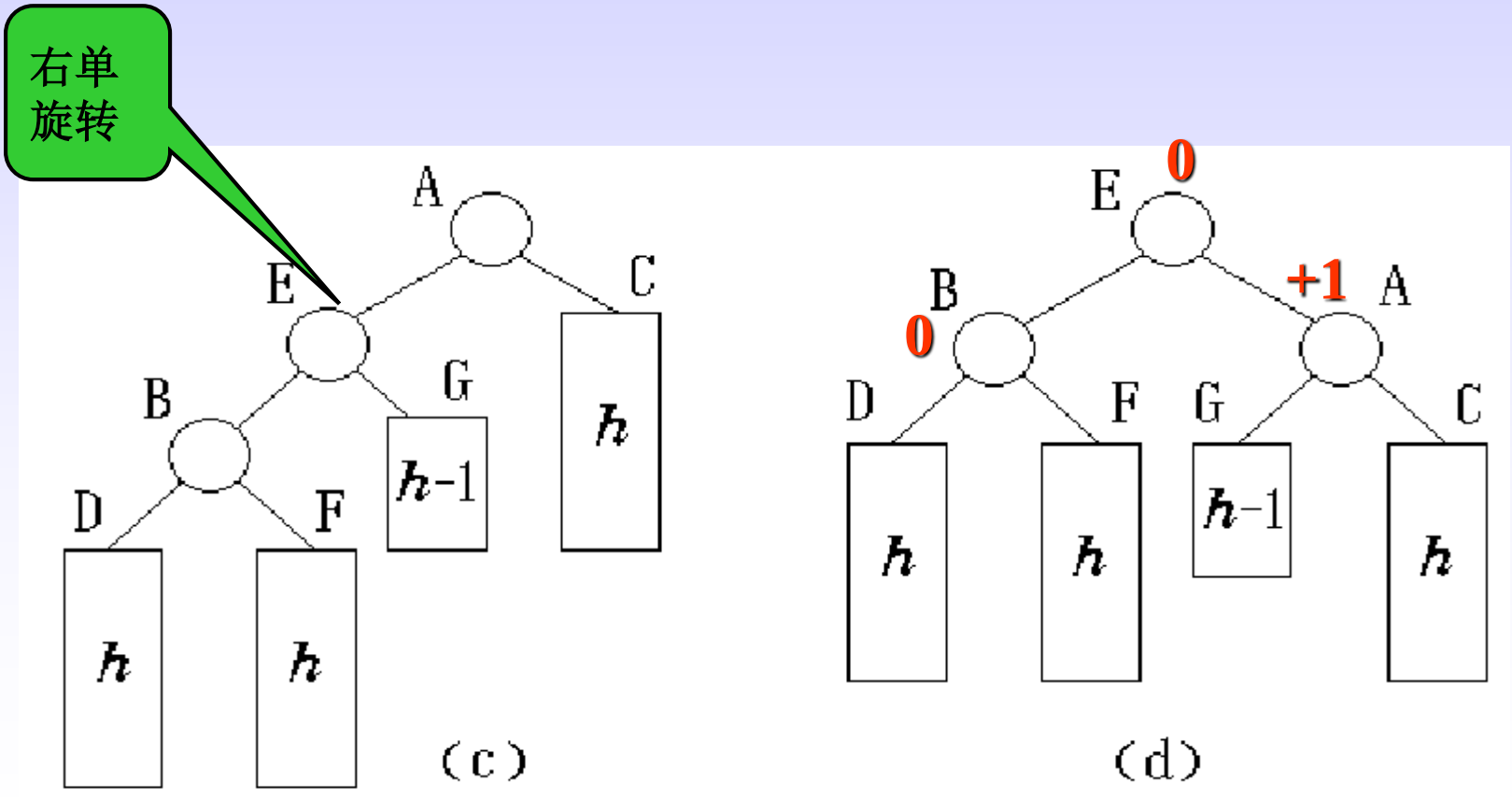


- 首先以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置，做左单旋转。





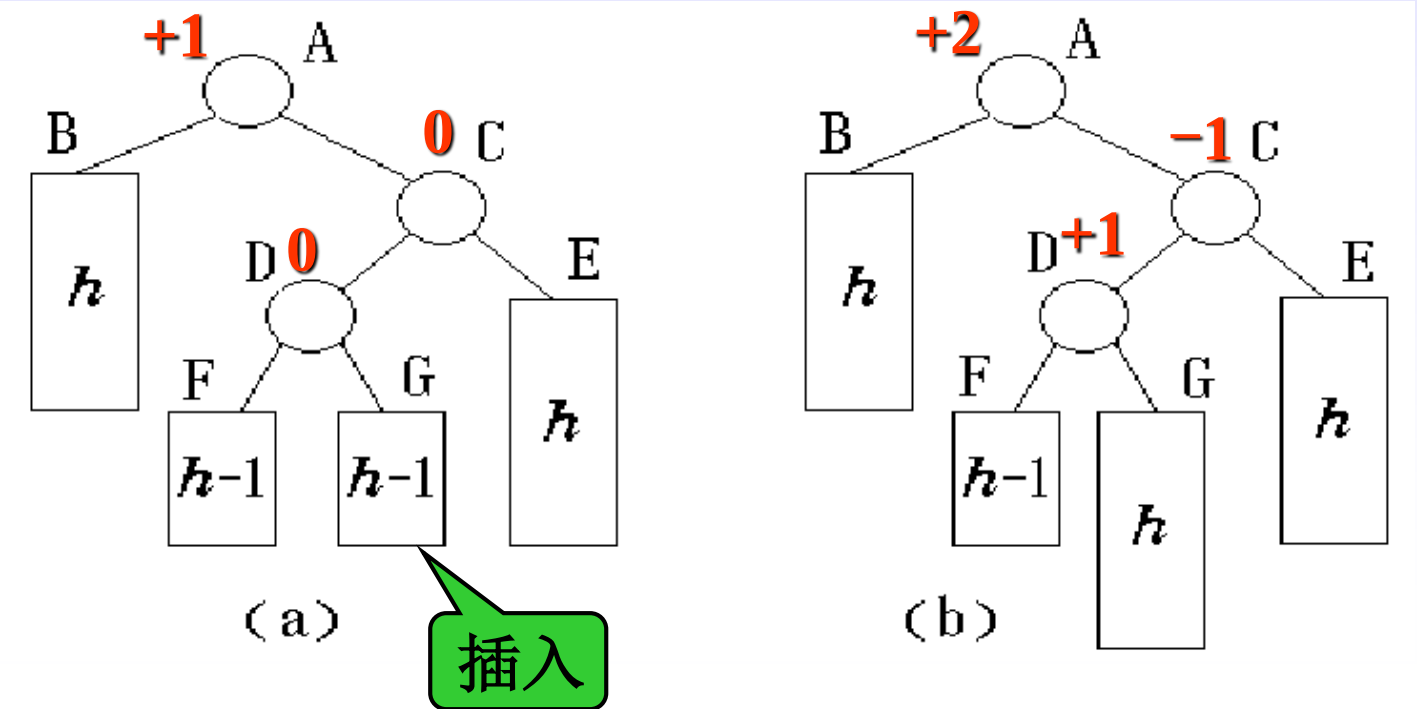
- 再以结点**E**为旋转轴，将结点**A**顺时针旋转，做右单旋转。使之平衡化。



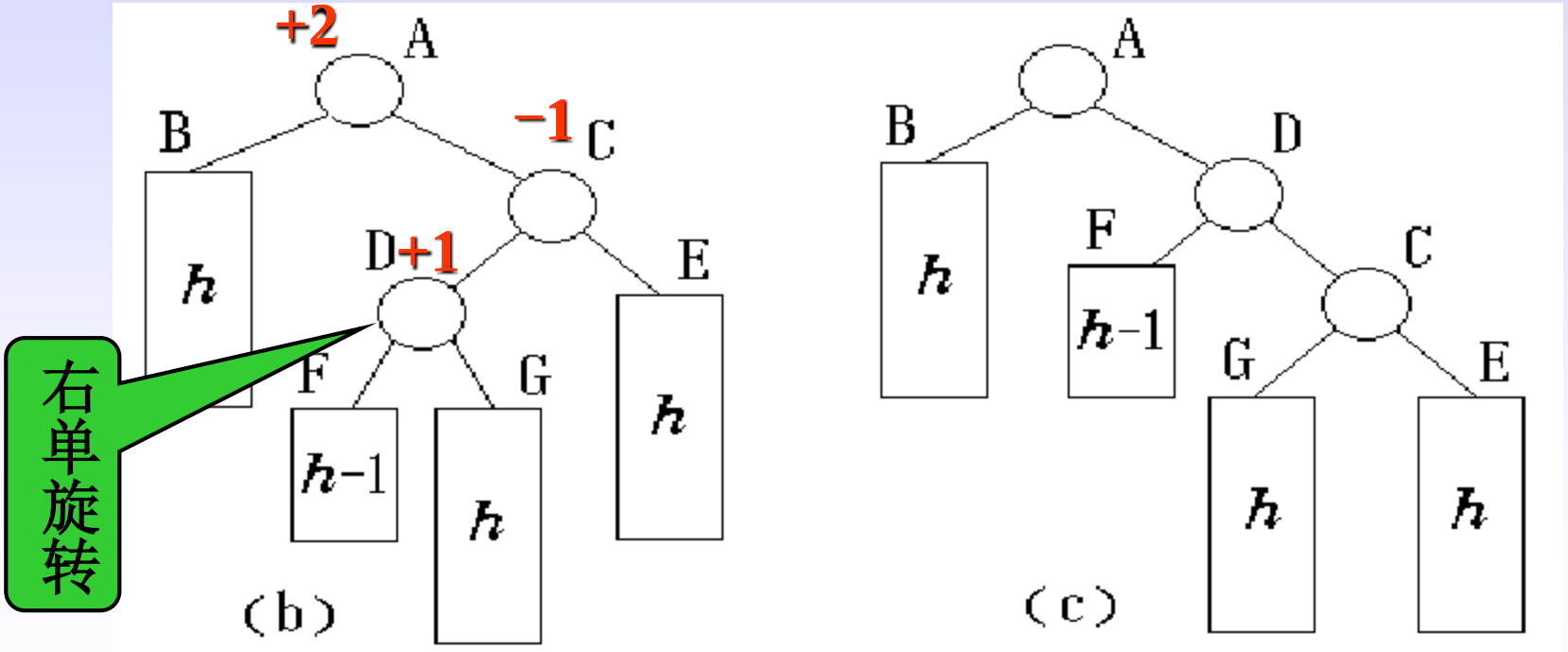
9.2 动态查找表

7 先右后左双旋转 (RL型):

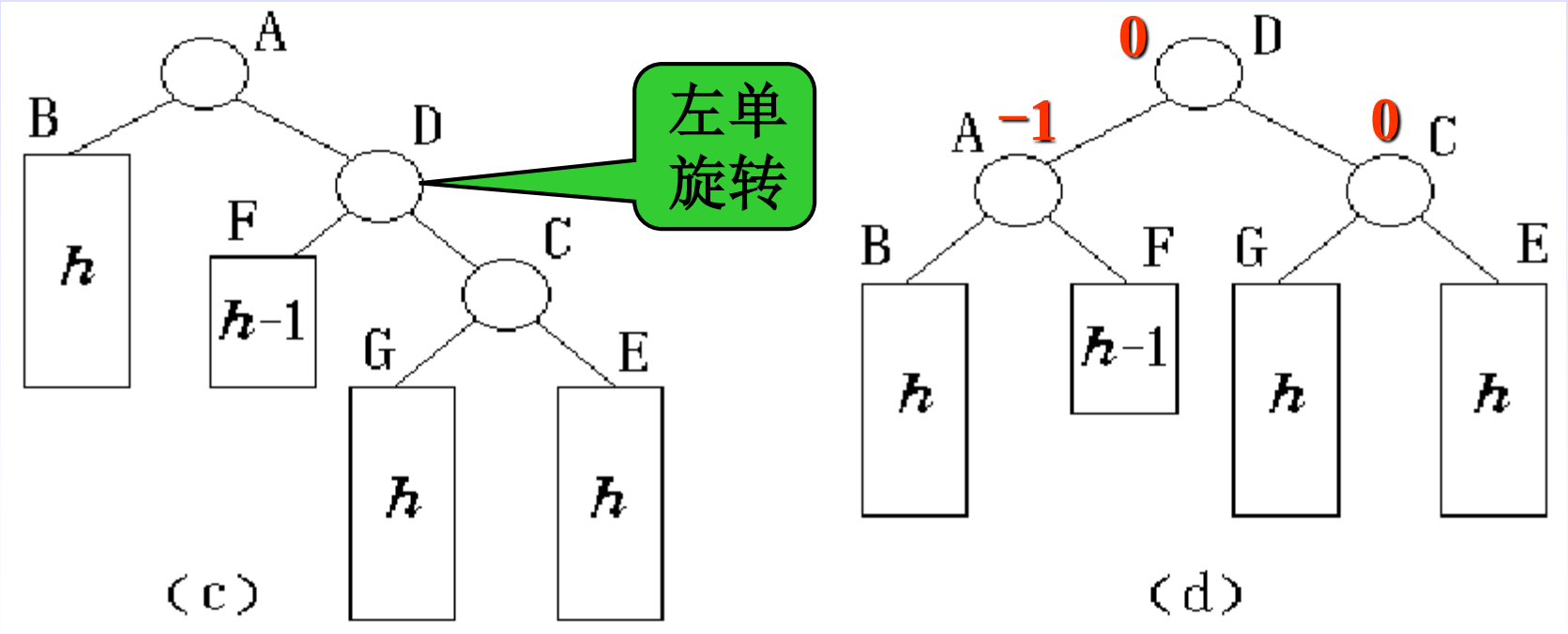
- 在下图中子树F或G中插入新结点，该子树深度增1。结点A的平衡因子变为2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“〉”的折线上，需要进行先右后左的双旋转。



- 首先做右单旋转：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。



- 再做左单旋转：以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。



9.2 动态查找表

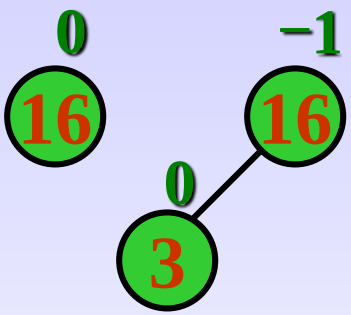
例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵**AVL**树。

9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。

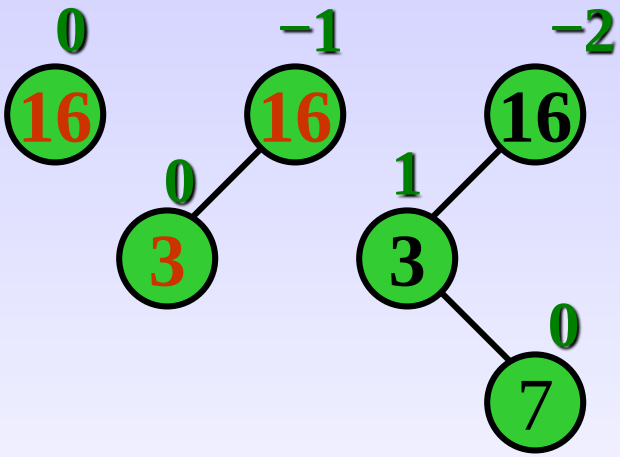
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



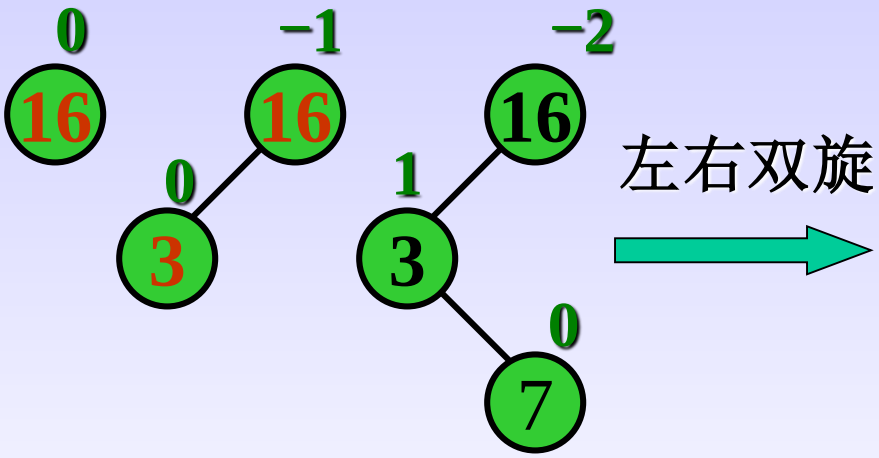
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



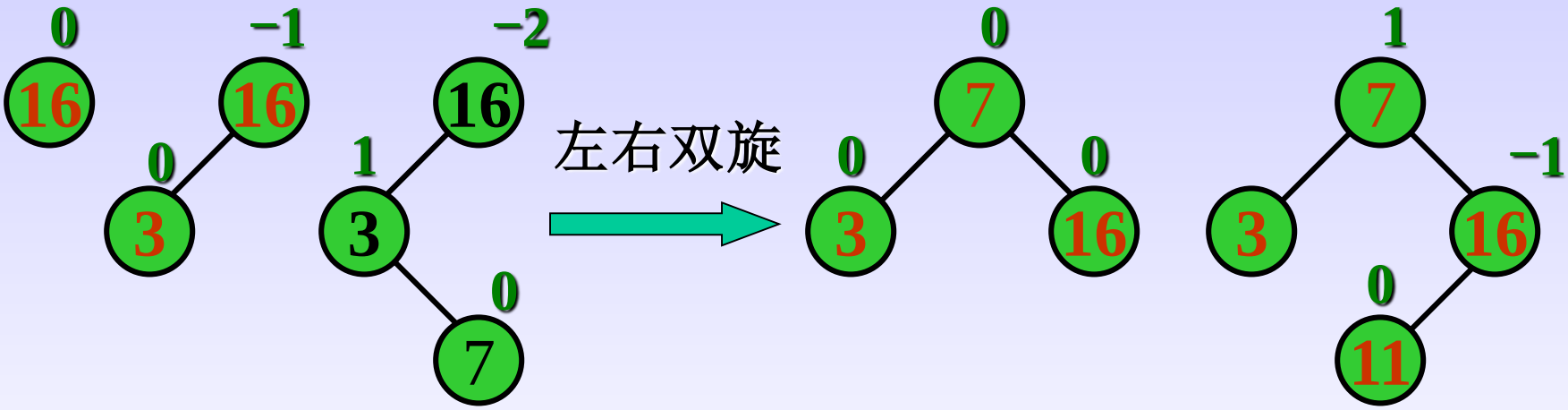
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



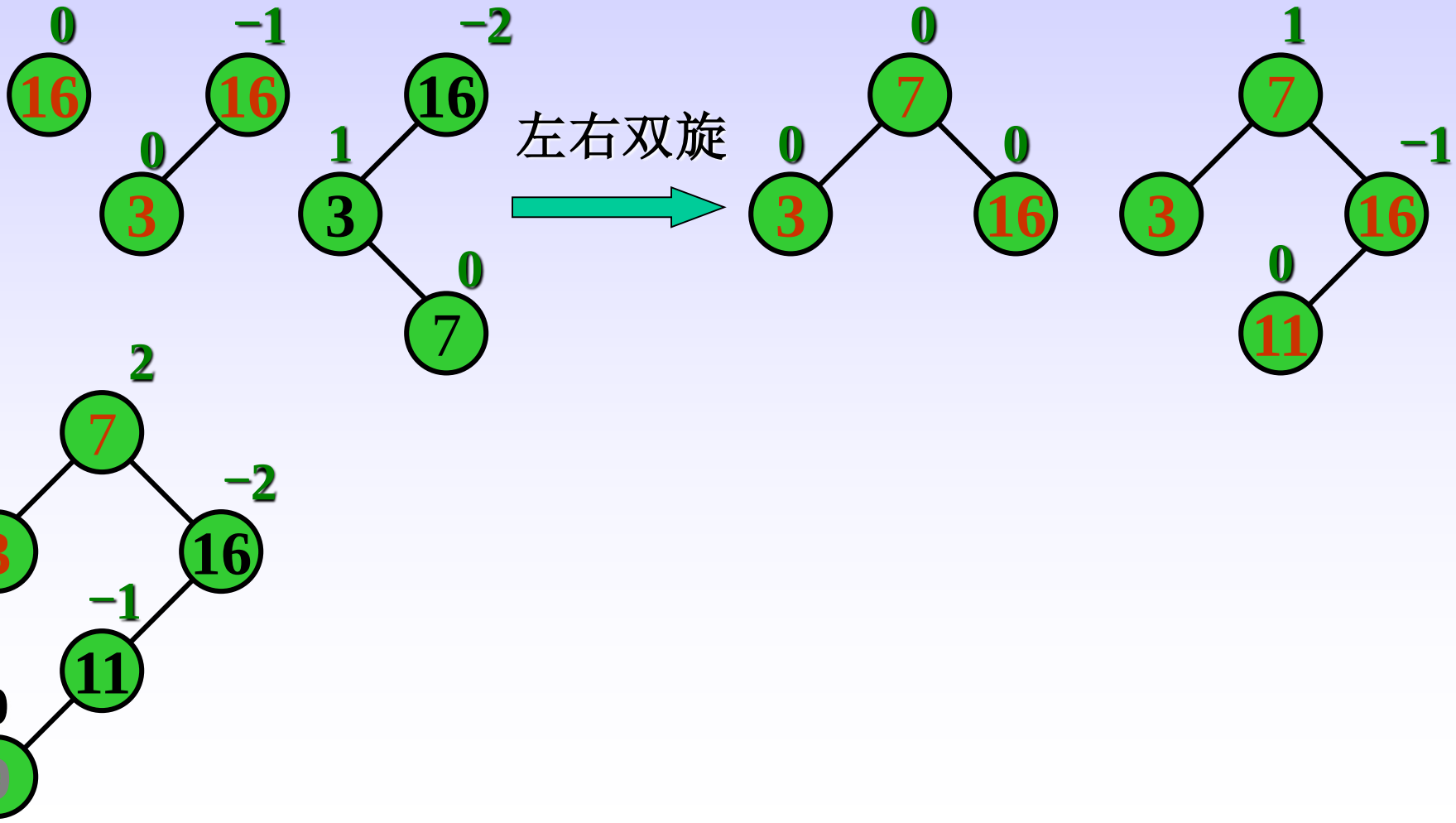
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



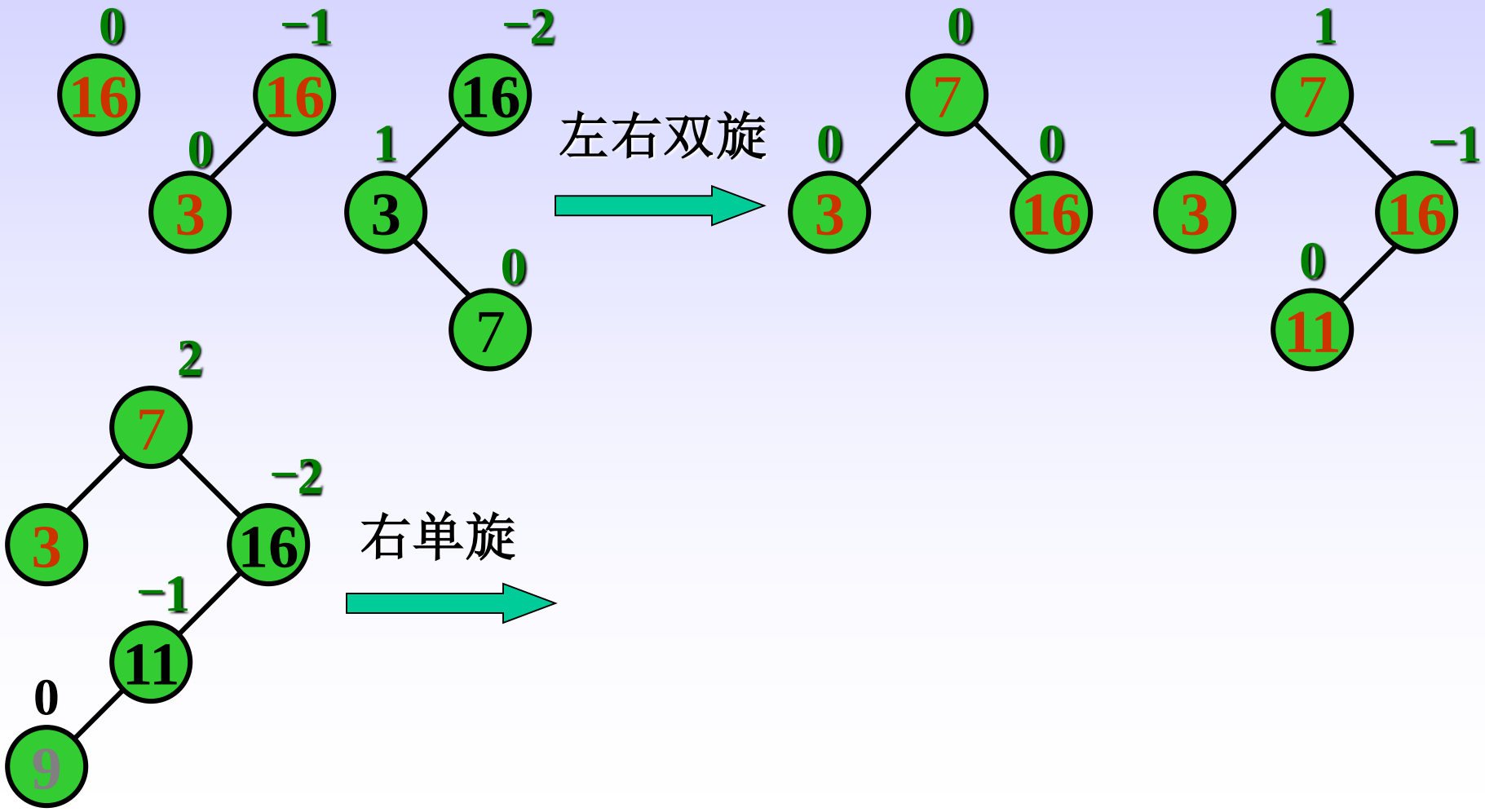
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



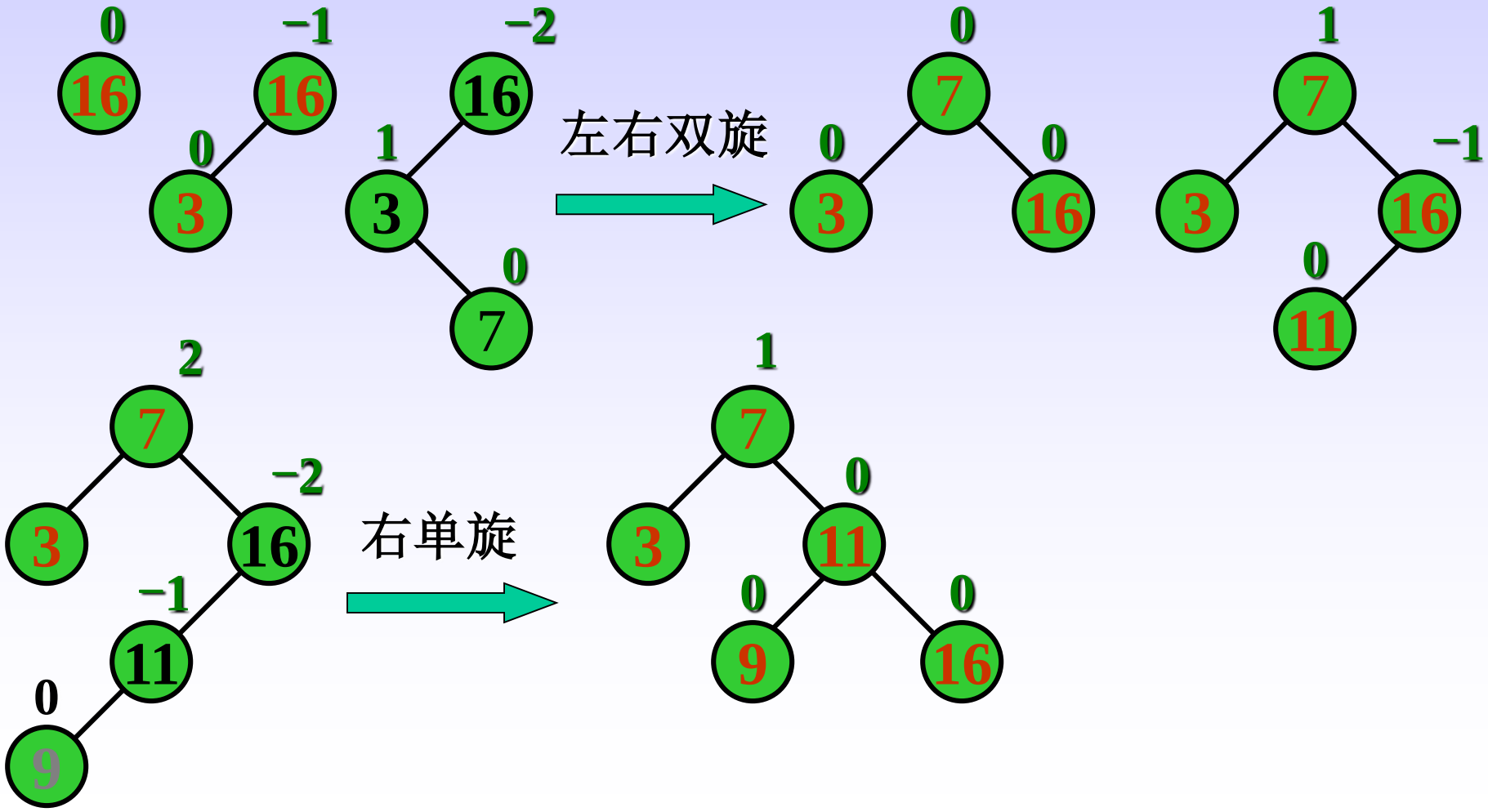
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



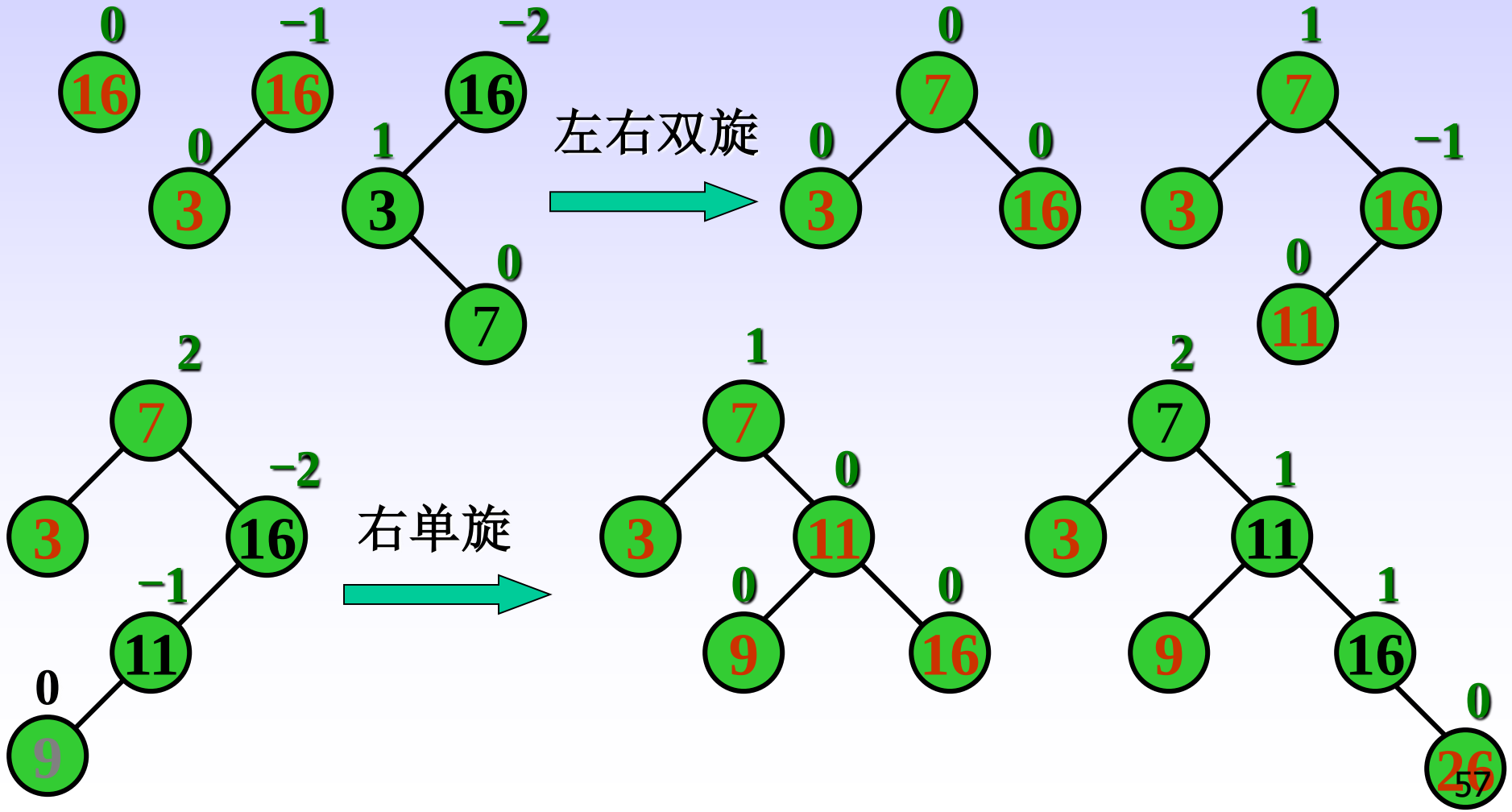
9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



9.2 动态查找表

例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，建立一棵AVL树。



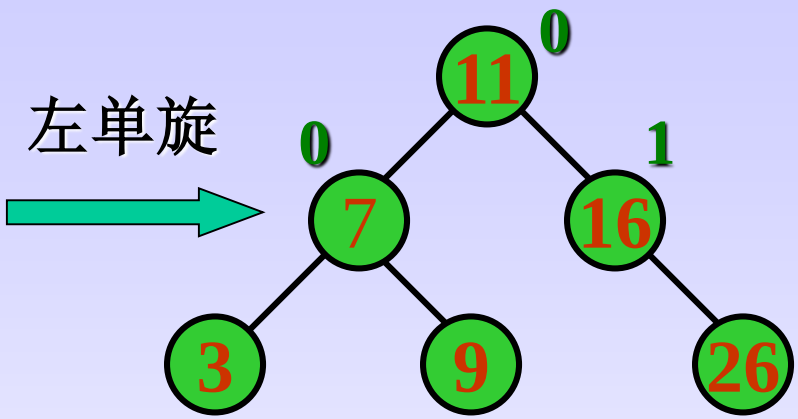
9.2 动态查找表

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

左单旋

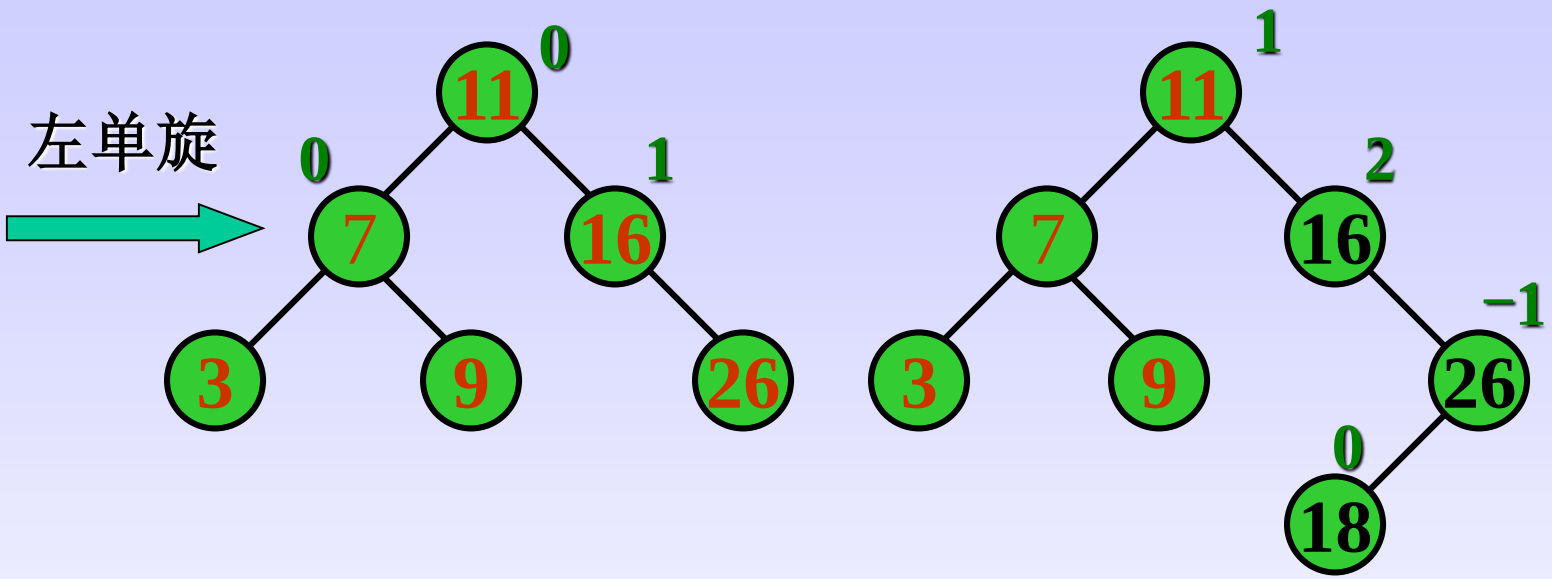
9.2 动态查找表

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



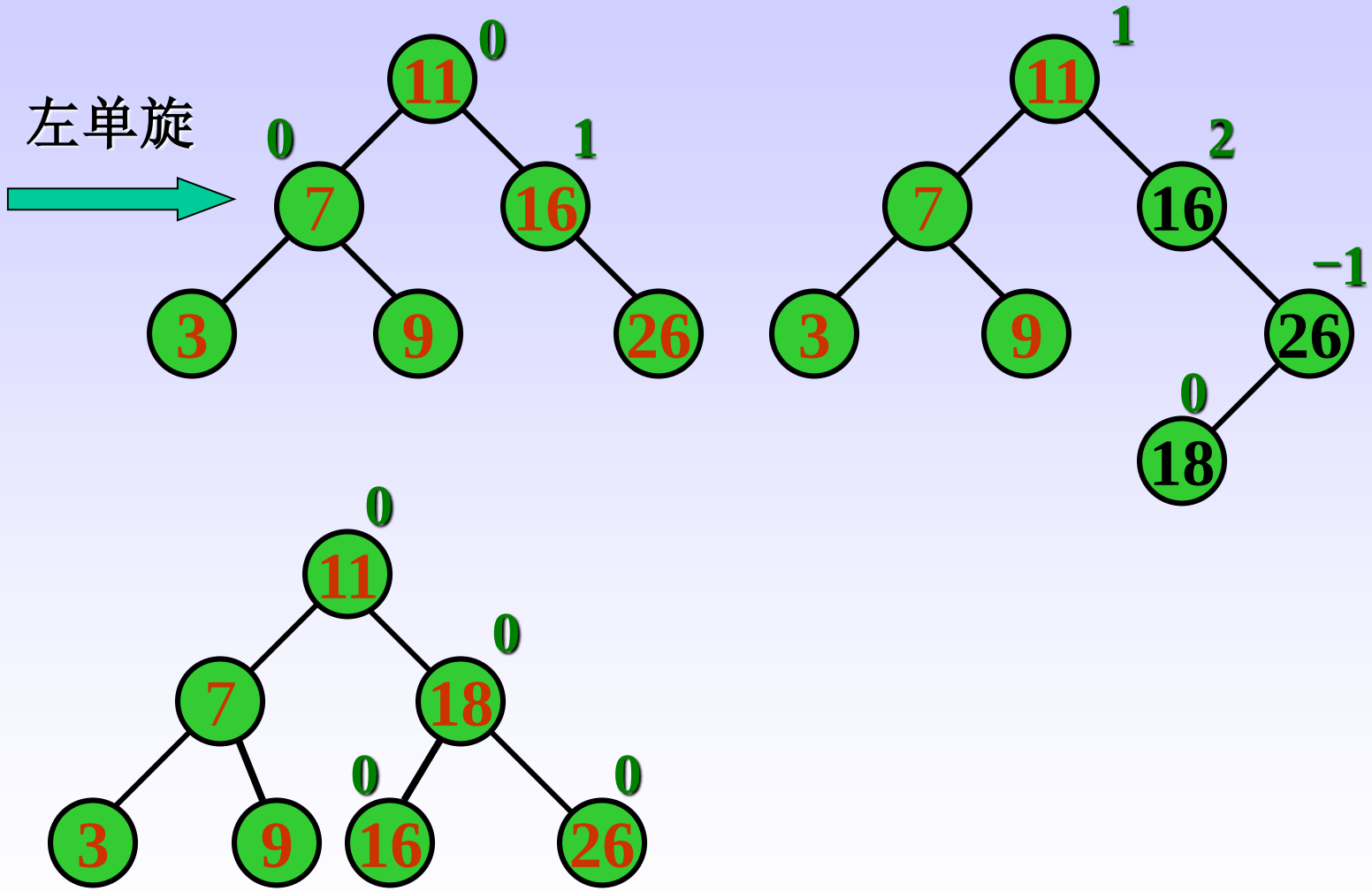
9.2 动态查找表

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



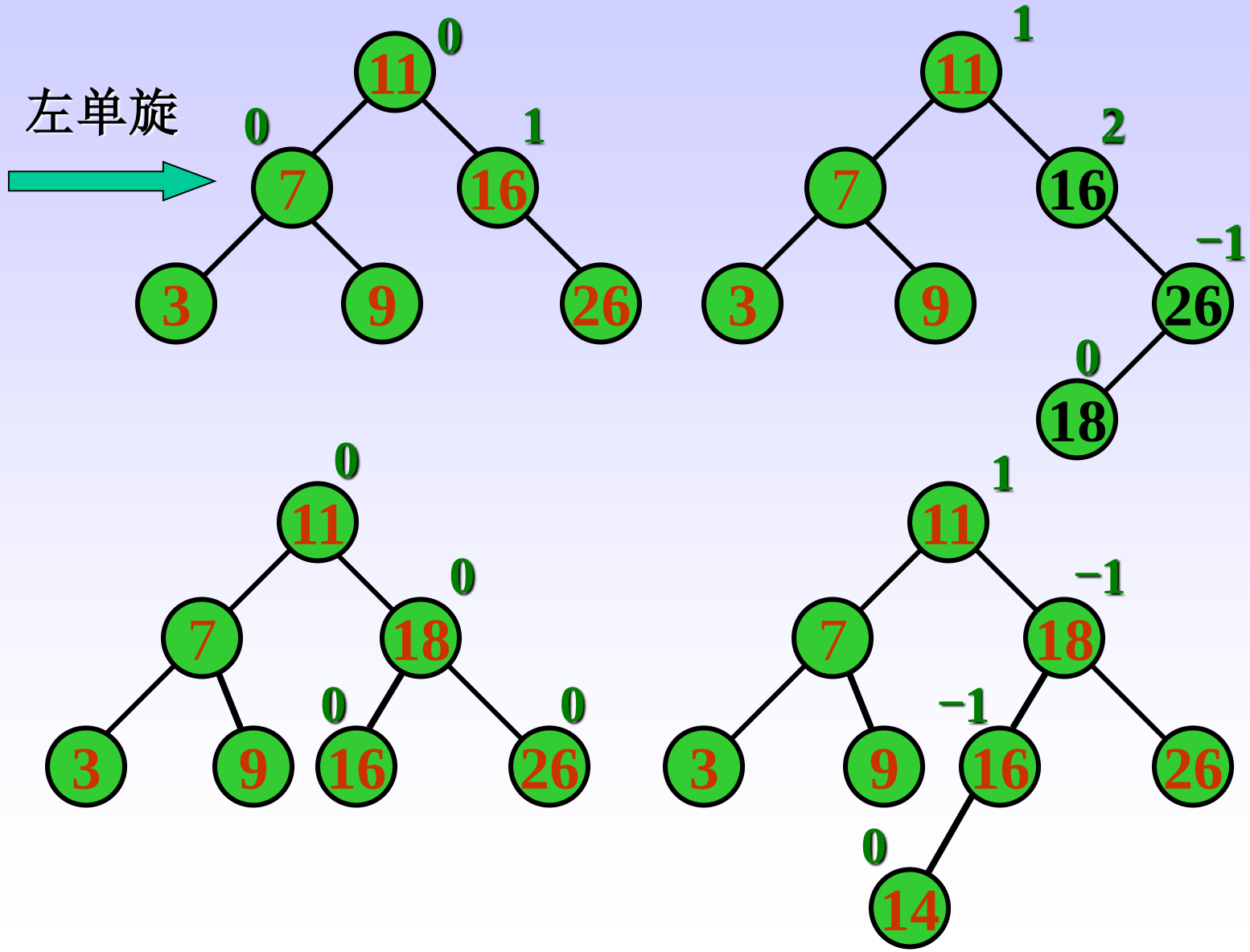
9.2 动态查找表

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



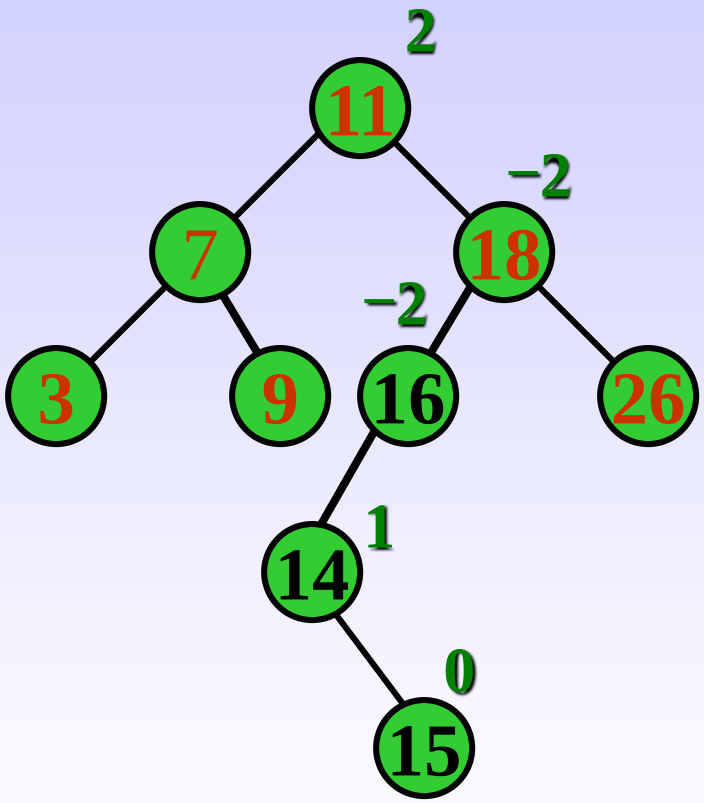
9.2 动态查找表

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



9.2 动态查找表

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



9.2 动态查找表

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