

# 第八章 查找



# 本章节目录

[8.1 查找表](#)

[8.2 静态查找表](#)

[8.3 动态查找表](#)

[8.4 哈希表](#)

## 8.1 查找表

- ❖ 查找表由**同一类型**的多个数据元素所构成，其中的每一个数据元素称为**查找对象**，也称为**记录**。
- ❖ **自然情况表**：包括多条记录，每条记录记录一个人的情况（姓名、性别、出生年月、身份证号、联系方式）
- ❖ 每一个数据元素有**若干属性**，能标识该数据元素的属性就称为**关键字**。
- ❖ **主关键字**：其值可以**唯一地**标识数据元素的属性，其他的属性则称为**次关键字**。
- ❖ 查找是指在数据元素集合中查找满足某种条件的数据元素。若按主关键字查找，则查找结果是唯一的，若按次关键字查找，则结果可能不唯一。



查找表种类：

静态查找表：顺序、折半、分块

动态查找表：二叉排序树、平衡二叉树

哈希表（散列表）：

## 8.2 静态查找表

- ❖ **静态查找表**：表一旦建立，就不在对其进行增加、删除和插入操作，表的结构不在变化。
- ❖ 静态查找表主要以**顺序表**作为组织结构，它的类型说明如下：

```
typedef struct
{
    int key;           // 关键字域
    其他属性;
    ...                // 其它域
} rectype
```

## 8.2.1顺序查找

### 1. 基本思想

对于给定的关键字K，从顺序表的第一个元素开始，依次向后与记录的关键字域相比较，如果某个记录的关键字等于K，则查找成功，并给出数据元素在表中的位置；若整个表查找完毕，仍未找到与关键字K相等的记录，则查找失败。

### 2. 算法

```
typedef rectype SeqList[n+1];    // 0号单元用作监视哨

int SearchSeq(SeqList r, int k)
{
    r[0].key=k; i=n;              // 设置监视哨
    while(r[i].key != k) i--;      // 从表尾向前查找
    return i;                     // 找不到为0，找到为在顺序表中的位置
} // SearchSeq
```

struct

```
{ int key;
  其他属性;
  ...
}
```



### 3. 性能分析

和给定值进行比较的次数的“期望值”称为查找算法的**平均查找长度**ASL（Average Search Length）。它是衡量查找算法性能的主要依据。

◆ 对一个含有 $n$ 个数据元素的表，查找成功时有：

$$ASL = \sum_{i=1}^n p_i c_i$$

◆  $c_i$ : 给定值 $k$ 与表中第 $i$ 个元素关键字相等时要比较的次数，需进行 $n-i+1$ 次比较，即 $c_i = n - (i-1)$ 。

◆ 设每个数据元素的查找概率相等，即 $P_i = 1/n$ ，则等概率情况下有：

$$ASL = (n+1) / 2 \quad \rightarrow \quad O(n) \quad \text{线性级}$$

## 8.2.2 折半查找

折半查找又称**二分查找**，它是一种效率较高的查找方法。

**折半查找的条件：**

- ◆ 要求**表有序**，即表中数据元素**按关键字有序排列**；
- ◆ 采用**顺序存储结构**。

### 1. 基本思想

对于**递增有序**的表，首先，将给定的查找关键字 $k$ 与有序表的中间位置上的元素进行比较，若相等，则查找成功。否则，中间元素将线性表分成两部分，前一部分中的元素均小于中间元素，而后一部分中的元素则均大于中间元素。在前、后两部分重复上述过程，直至查找成功或失败。

2, 5, 7, 8, 33, **55**, 76, 89, 121, 167, 564



## 2. 折半查找举例

【例1】在有序表（05, 10, 15, 19, 25, 28, 40, 55, 85）中，查找关键字为55的数据元素。

## 3. 算法

```
int SearchBin1(SeqList r, int k)
{int  mid, low=1,  high=n;
  while(low<=high)
  {mid=(low+high)/2;
    if(k==r[mid].key) return mid;      // 找到待查元素
    else if(k>r[mid].key) low=mid+1;    // 在后半区间查找
    else high=mid-1;                   // 继续在前半区间查找
  }
  return 0; // 查找失败返回0
} // SearchBin1
```

## 4. 性能分析

平均查找长度  $ASL = \log_2(n+1) - 1$  对数级的，优于线性级的

### 8.2.3 分块查找

- ❖ **分块查找**也称**索引顺序查找**，它是顺序查找方法的改进，其目的是通过缩小查找范围来改进顺序查找的性能。
- ❖ 数据分成若干块，块内无序，块间有序
- ❖ **索引表**：每个块内的**最大关键字**+第一个元素的地址

#### 1. 基本思想

- 1) 将待查关键字k与索引表中的关键字进行比较，确定待查记录所在的块。
- 2) 进一步用顺序查找法，在相应块内查找关键字为k的元素。

## 2. 分块查找举例

【例2】将表中的15个记录，按关键字值25，56和89分为三块建立的索引表和查找表。找48

|       |    |    |    |
|-------|----|----|----|
| 最大关键字 | 25 | 56 | 89 |
| 起始地址  | 1  | 6  | 11 |

|    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 10 | 5 | 25 | 20 | 16 | 56 | 35 | 28 | 32 | 48 | 68 | 72 | 62 | 84 | 89 |
|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|

### 3. 性能分析

- ❖ 分块查找由**索引表查找**和**子表查找**两步来完成。均采用**顺序查找**。假设 **$n$** 个数据元素的查找表分为 **$m$** 个子表，且每个子表均为 **$L$** 个元素，则 $L=n/m$ ，则分块查找的平均查找长度为：
- ❖  $ASL = ASL1 + ASL2 = (m+1)/2 + (n/m+1)/2$
- ❖ 其中 $ASL1$ 为索引表的平均查找长度， $ASL2$ 为子表的平均查找长度。

## 8.3 动态查找表

- ❖ 动态查找表的特点是，表结构本身是在查找过程中动态生成的，即对于给定值key，若表中存在其关键字等于key的记录，则查找成功返回，否则插入关键字等于key的记录。

### 8.3.1 二叉排序树

#### 1. 定义

**二叉排序树**或是一棵空二叉树，或是一棵具有下列性质的二叉树：若左子树非空，则左子树上所有结点的值均小于根结点的值；若右子树非空，则右子树上所有结点的值均大于根结点的值；并且其左、右子树均是二叉排序树。

举例：中序遍历

❖ 二叉排序树一般采用**二叉链表**来存储，其类型定义为：

```
typedef struct BSTnode  
  
    { int key;                // 关键字域  
  
        ...                // 其它数据域  
  
    struct BSTnode *lchild, *rchild; // 左、右孩子指针  
  
} BSTNode, *BSTree;
```

## 2. 查找

(1) 基本思想：首先将给定值和根结点的关键字进行比较，若相等，则查找成功；否则，若小于根结点的关键字，则在左子树上查找；若大于根结点的关键字，则在右子树上查找。

返回值?

## (2) 算法 (不考虑插入操作)

```
BSTree SearchBST1(BSTree t, int k)
{
    if( !t || k==t->key)    return(t);
    else
        if(k<t->key)    return(SearchBST1(t->lchild, k));
        else    return(SearchBST1(t->rchild, k));
}
```



### 3.插入

#### (1) 插入原则

已知一个关键字值为 $k$ 的结点，若将其插入到二叉排序树中，只要保证插入后仍满足二叉排序树的定义。

新插入的结点一定是一个**新添加的叶子结点**，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子。

结点插入的步骤如下：

- ①若二叉排序树是空树，则 $k$ 成为二叉排序树的根；
- ②若二叉排序树非空，则将 $k$ 与二叉排序树的根进行比较：  
如果 $k$ 的值等于根结点的值，则停止插入；如果 $k$ 的值小于根结点的值，则将 $k$ 插入左子树；如果 $k$ 的值大于根结点的值，则将 $k$ 插入右子树。

## (2) 递归算法

```
void InsertBST(BSTree t, int k)
```

```
{//若二叉排序树t中无关键字k, 则插入, 否则直接返回
```

```
    if (t->key==k)    return; //已有k, 无需插入
```

```
    if(t==null)        //插入根结点
```

```
    { t=(BSTree)malloc(sizeof(BSTNode));
```

```
        t->key=k;    t->lchild=t->rchild=NULL;
```

```
    }
```

```
    else    if (k<t->key) InsertBST( t->lchild, k);
```

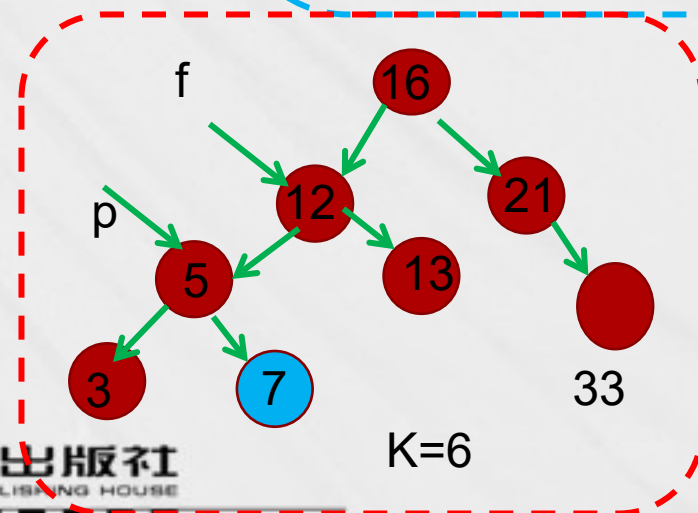
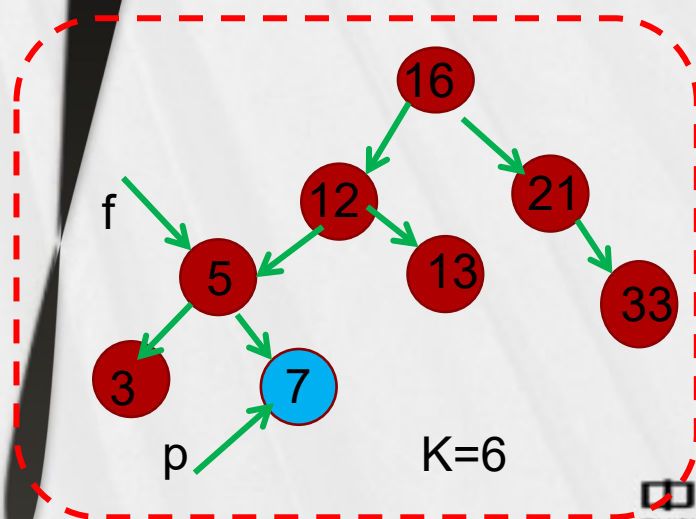
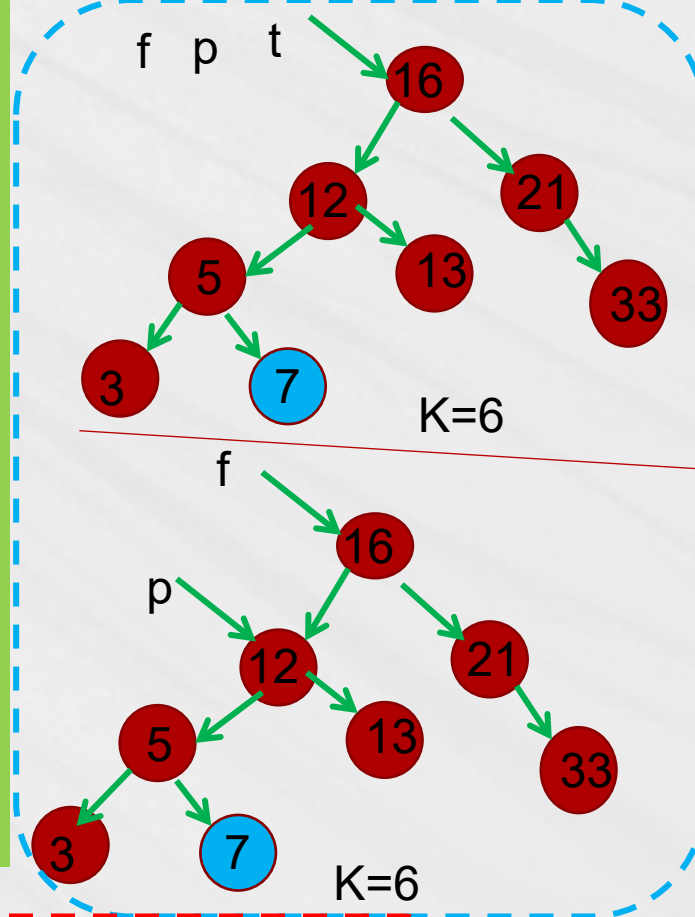
```
        else InsertBST( t->rchild, k);
```

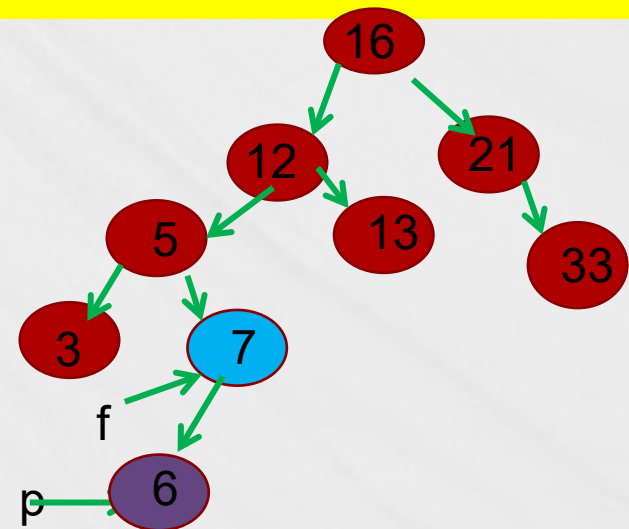
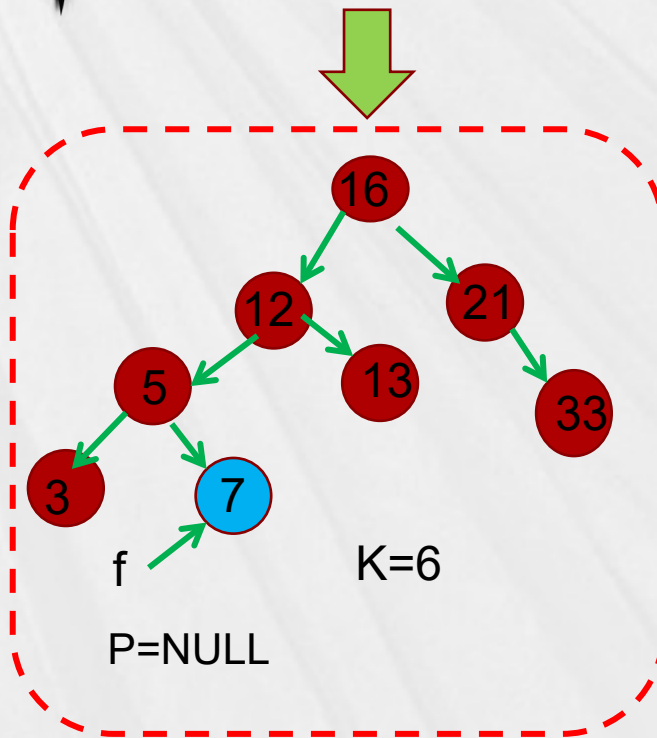
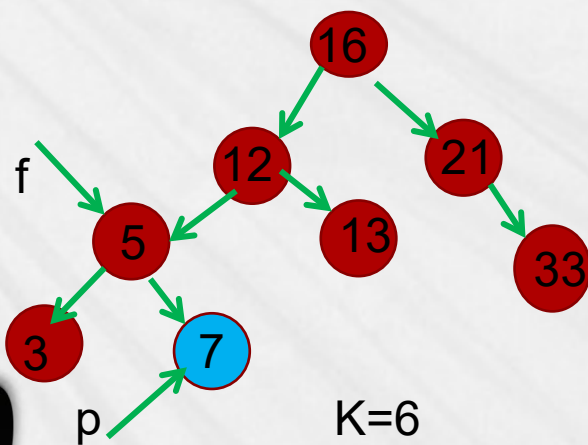
```
}
```

## (2) 非递归算法

```
void InsertBST( BSTree t, int k)
{ BSTree f, p=t;
while(p)
{ if (p->key==k) return; //已有k无需插入

f=p;
p=(k<p->key)?p->lchild : p->rchild;
} //next page
```





```

while(p)
{
    if (p->key==k) return;
    f=p;
    p=(k<p->key)?p->lchild : rchild;
}

```

```

p=(BSTree)malloc(sizeof(BSTNode));
p->key=k;
p->lchild=p->rchild=NULL;

```

```

if(t==NULL) t=p;
else if (k<f->key) f->lchild=p;
else f->rchild=p;
}

```

### (3) 二叉排序树的构造过程

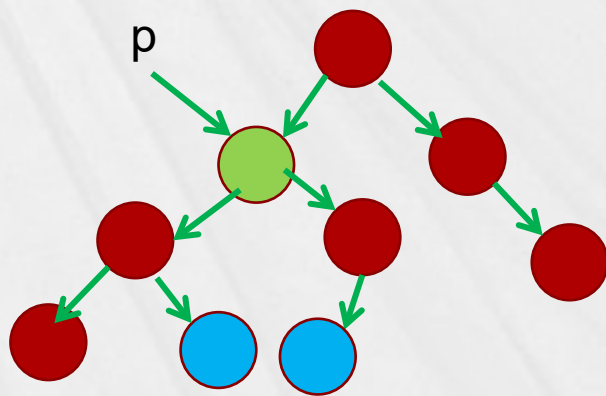
- ❖ 二叉排序树的构造过程是输入一个元素，就调用一次插入算法将其插入到当前生成的二叉排序树中。

【例3】已知有关键字序列为{45, 25, 55, 25, 60, 15, 30, 78}，构造相应的二叉排序树。

## 4.删除

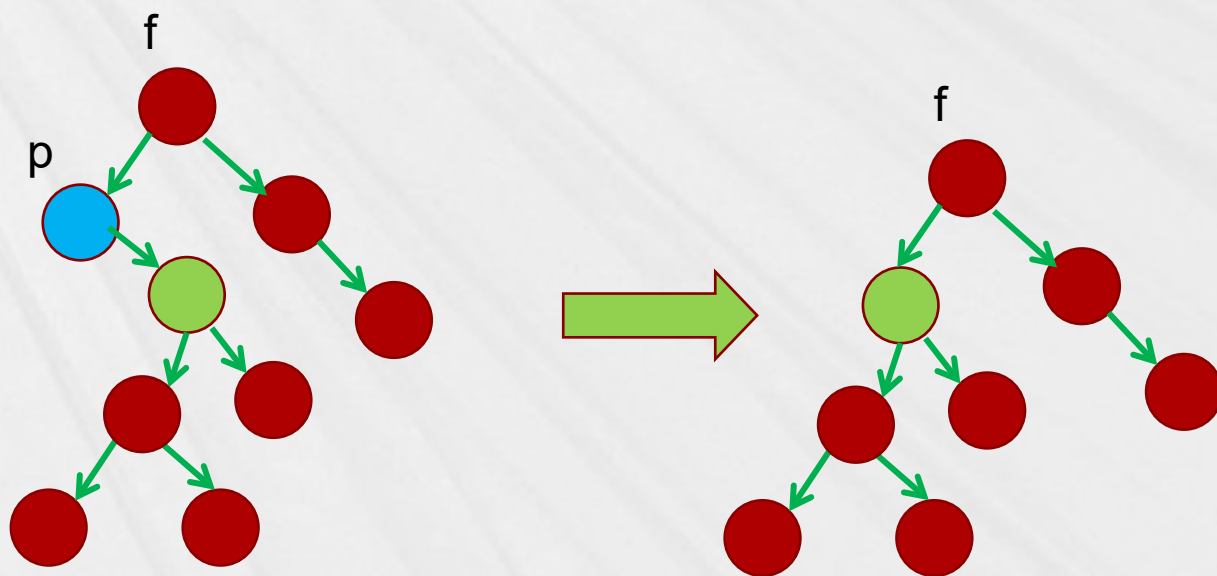
### (1) 删除原则

- ❖ 二叉排序树中删除一个结点，仍要保持是二叉排序树。由于**中序遍历二叉排序树**可以得到结点的有序序列，因此，删除结点后只要由于用结点的直接前驱和直接后继均可代替被删结点的位置，删除算法不唯一。



◆删除过程(分几种情况：有无左子树)：

- ①若被删结点p**无左子树**，则根据p是双亲f的左（或右）子女，令其双亲f的左（或右）指针指向p的右子树，删除p结点；



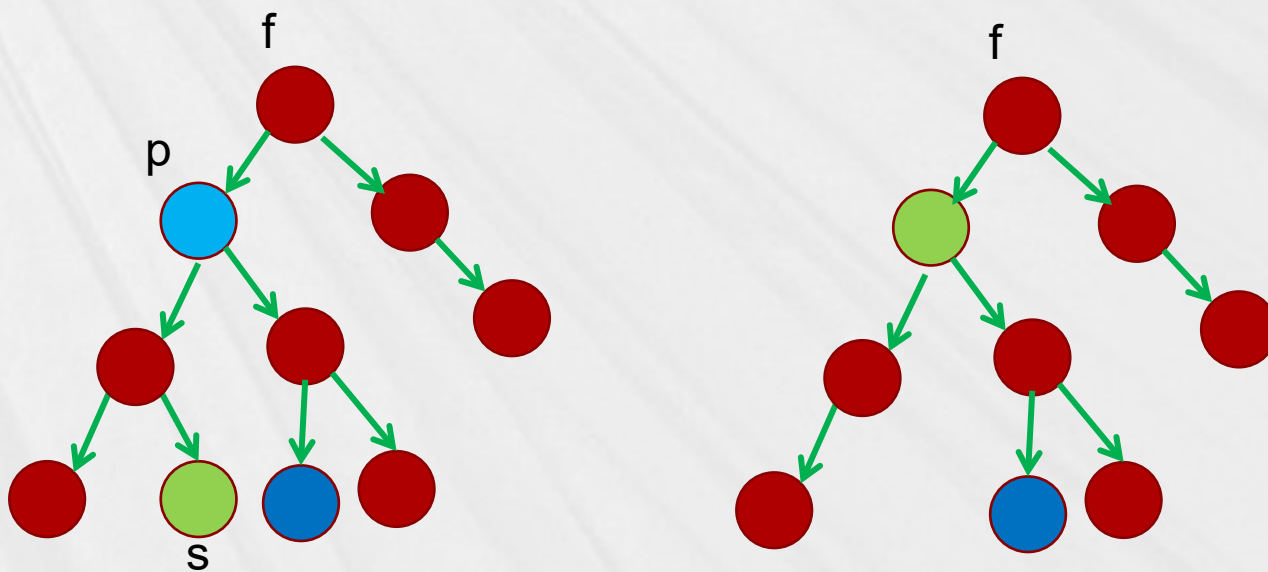
**p无左子树，p是双亲的左孩子**



◆删除过程(分几种情况：有无左子树)：

②若被删结点p有左子树，则用左子树上结点值最大的结点（设为s）替换p结点，并对指向的结点指针进行适当调整。

(2) 删除算法（参见教材）



P有左子树

# 二叉排序树的效率分析

一般情况： 二叉树的形态比较均衡（左右子树高度相近，每次查找范围都缩小一半）

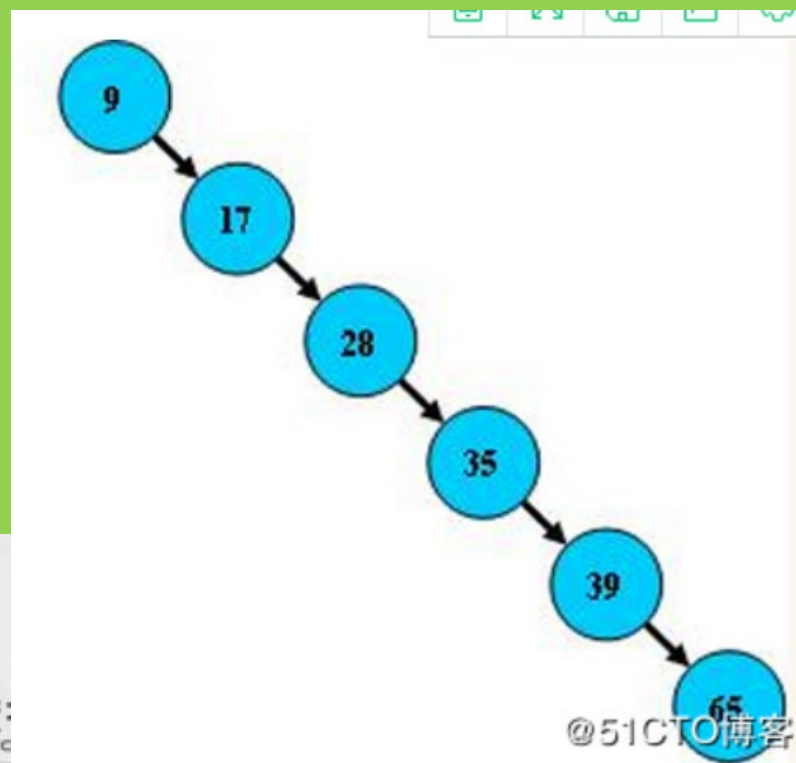
类似折半查找： $\log_2(n+1)$

极端情况： 二叉树极端不平衡（只有左子树，或右子树），时间复杂度将退化成线性

类似顺序查找： $(n+1) / 2$

如何得到一颗形态比较均衡的二叉树？

---》平衡二叉树



## 8.3.2平衡二叉树

### 1.定义

平衡二叉树或者是一棵空二叉树，或者是任意结点的左、右子树高度之差的绝对值不大于1的二叉树。

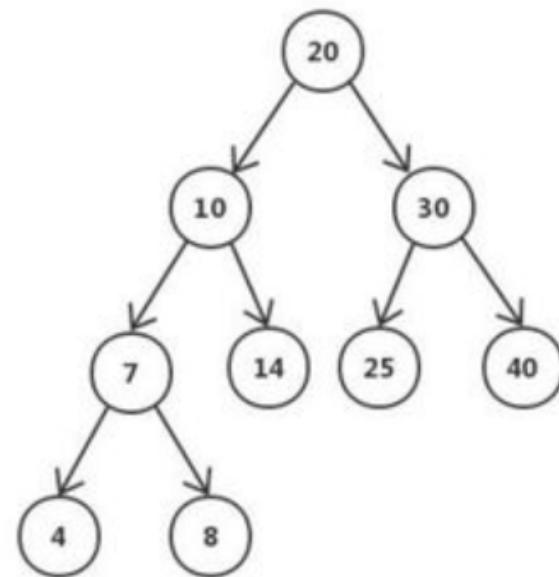
❖ 结点的左、右子树高度差为该结点的平衡因子。

平衡二叉树上所有结点的平衡因子只能是-1、0、1，如果二叉树上一个结点的平衡因子的绝对值大于1，则该二叉树就不是平衡二叉树。

❖ 树高度？

❖ 保证有n个结点的树的高度为 $O(\log n)$

❖ 排序二叉树

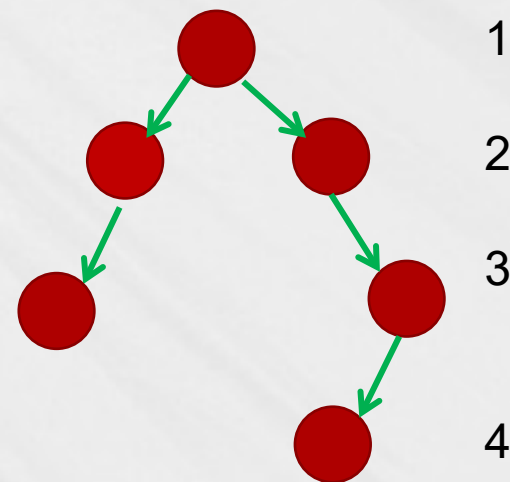
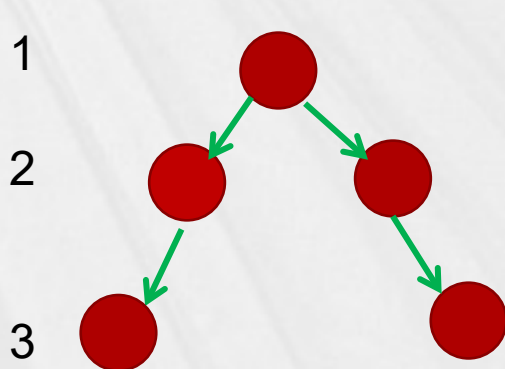


平衡二叉树

## 8.3.2平衡二叉树

### 1.定义

是平衡二叉树吗?

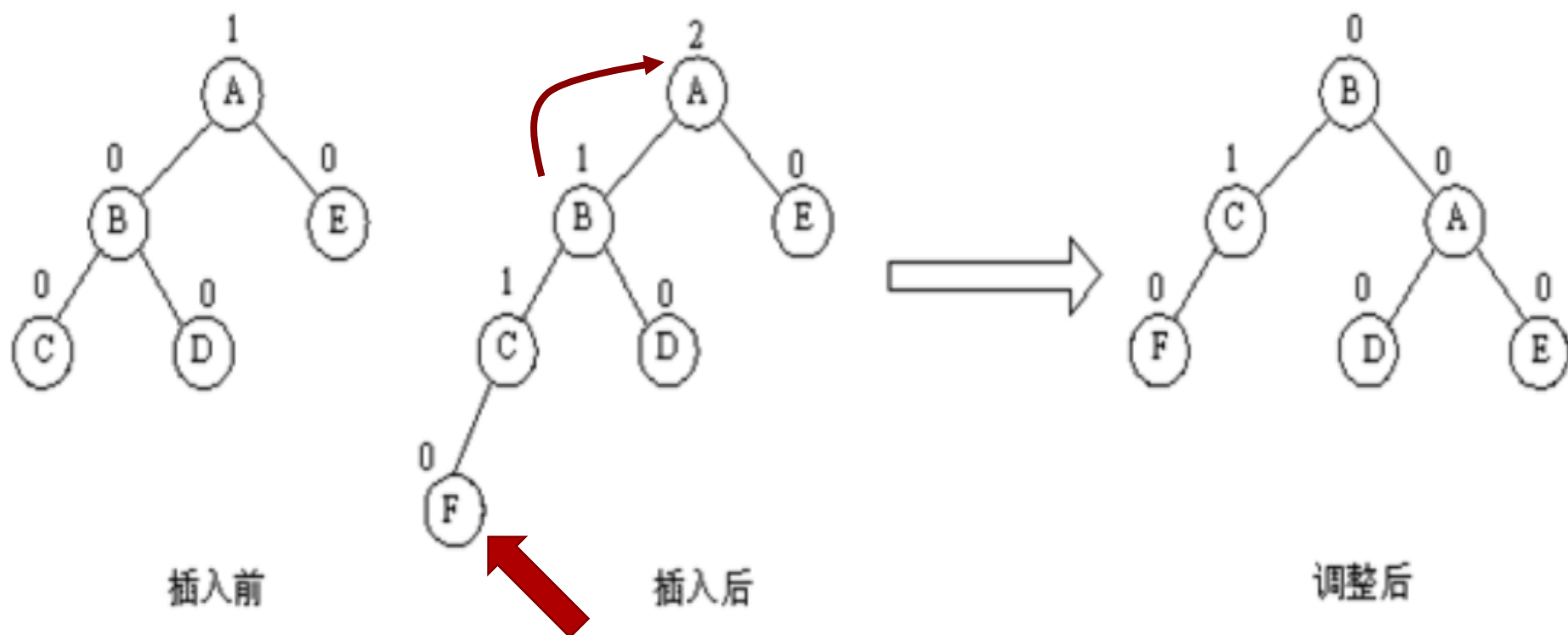


## 8.3.2平衡二叉树

### 2.构造方法

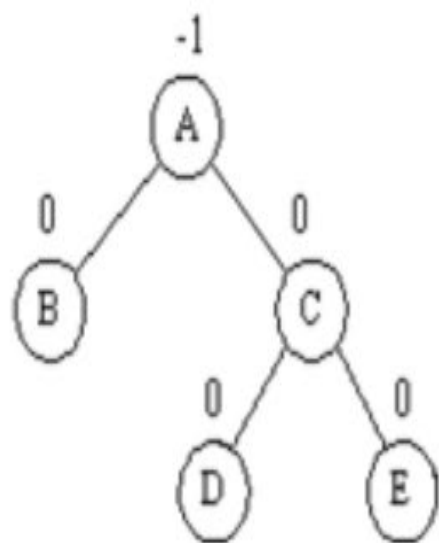
- ❖ 每当插入一个结点，首先检查是否破坏了该二叉树的平衡性，找出其中**最小不平衡子树**，调整成平衡二叉树

(1) LL型：插入位置为左子树的左结点，进行向右旋转

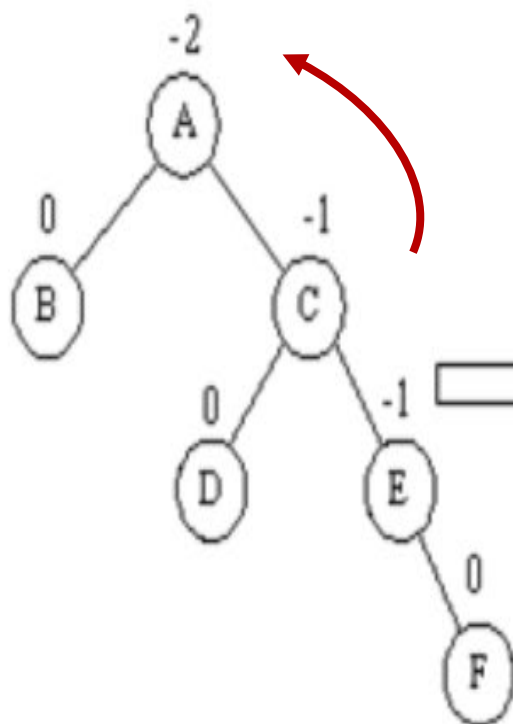


## (2) RR 型

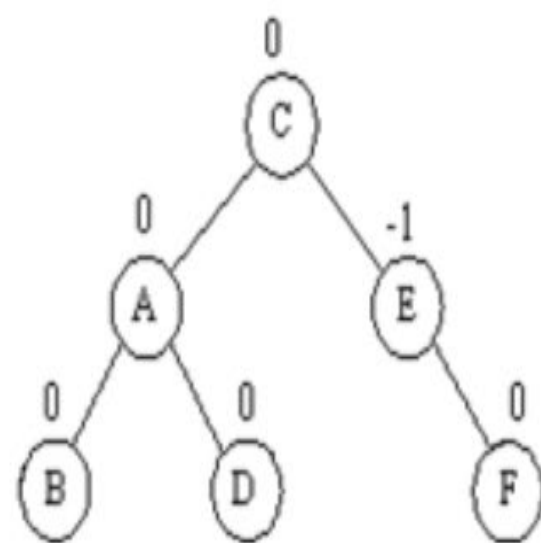
RR 型：插入位置为右子树的右孩子，进行向左旋转



插入前



插入后

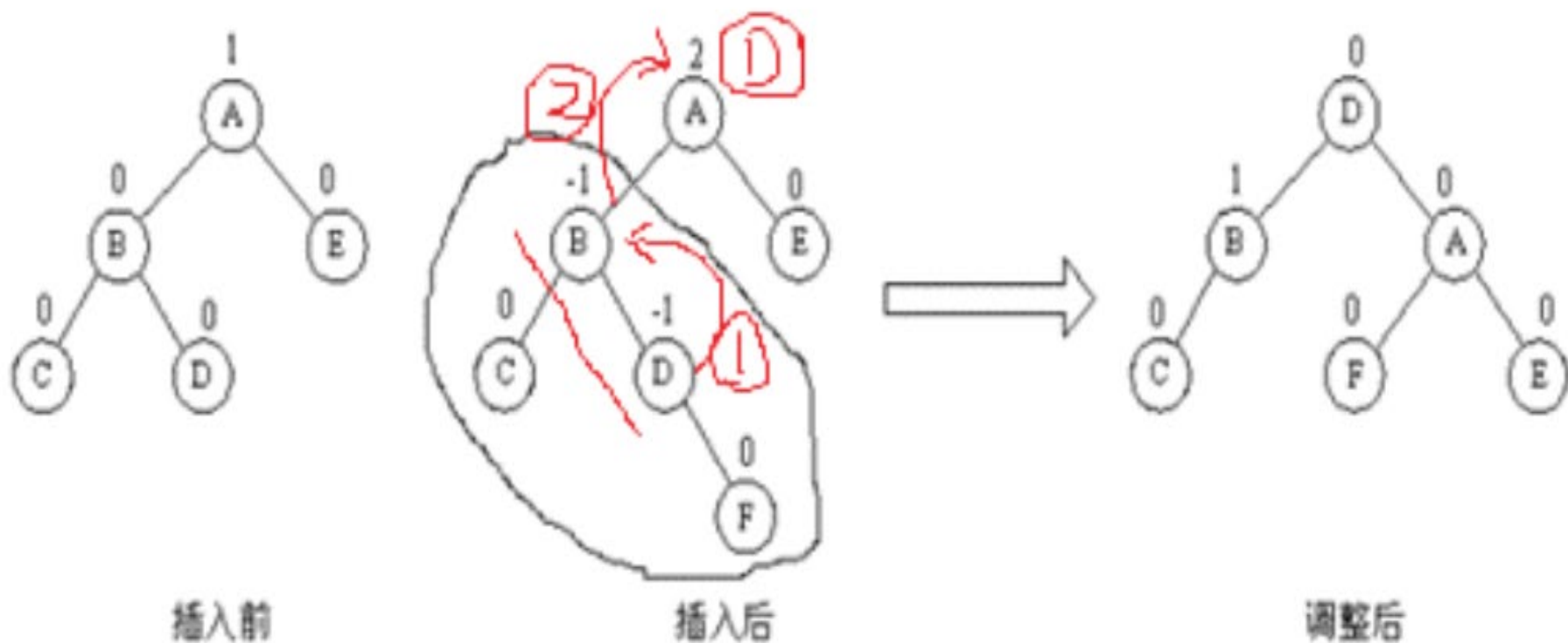


调整后

先向左旋转

### (3) LR 型

LR 型：插入位置为左子树的右孩子，要进行两次旋转，先左旋转，再右旋转；第一次最小不平衡子树的根结点先不动，调整插入结点所在的子树，第二次再调整最小不平衡子树。

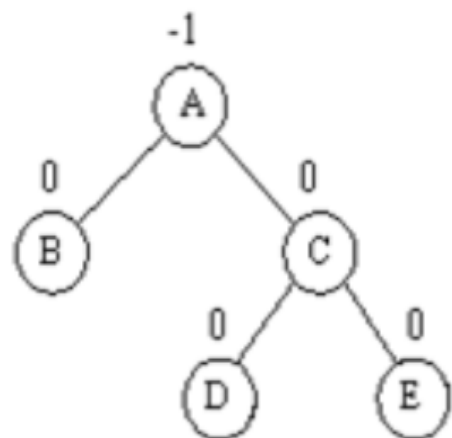


先向左旋转 再右旋

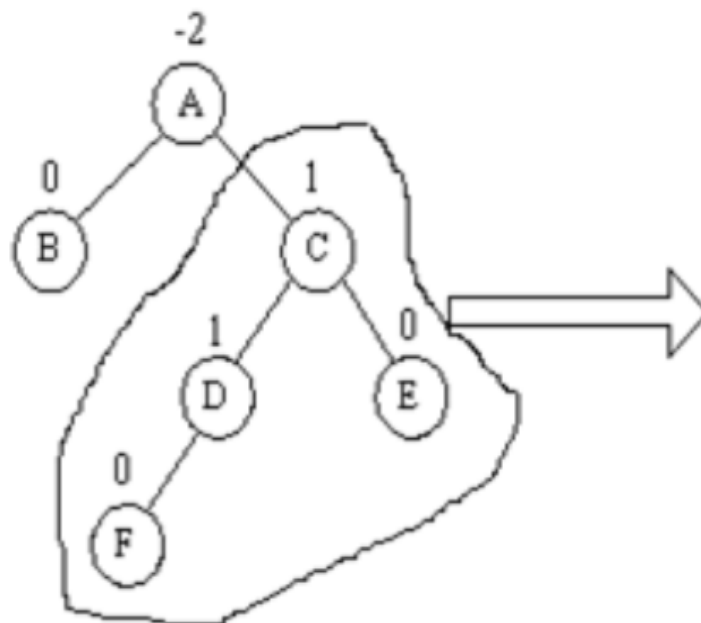


#### (4) RL 型

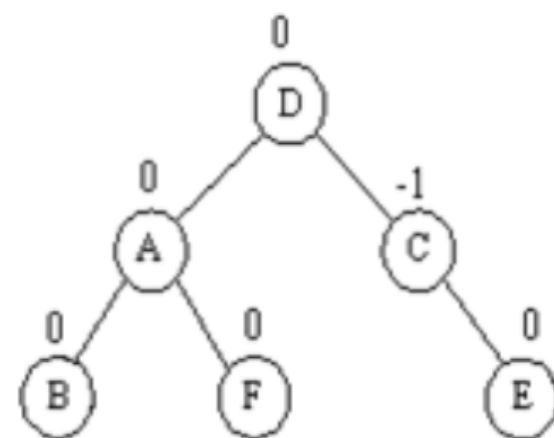
RL 型：插入位置为右子树的左孩子，进行两次调整，先右旋转再左旋转；处理情况与 LR 类似。



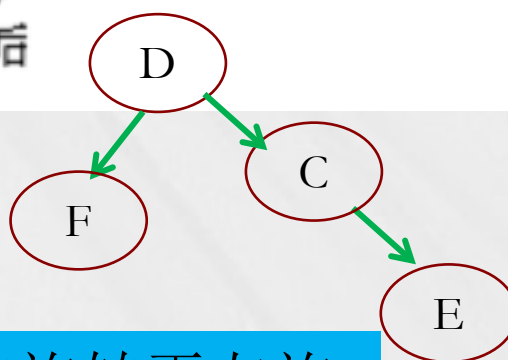
插入前



插入后

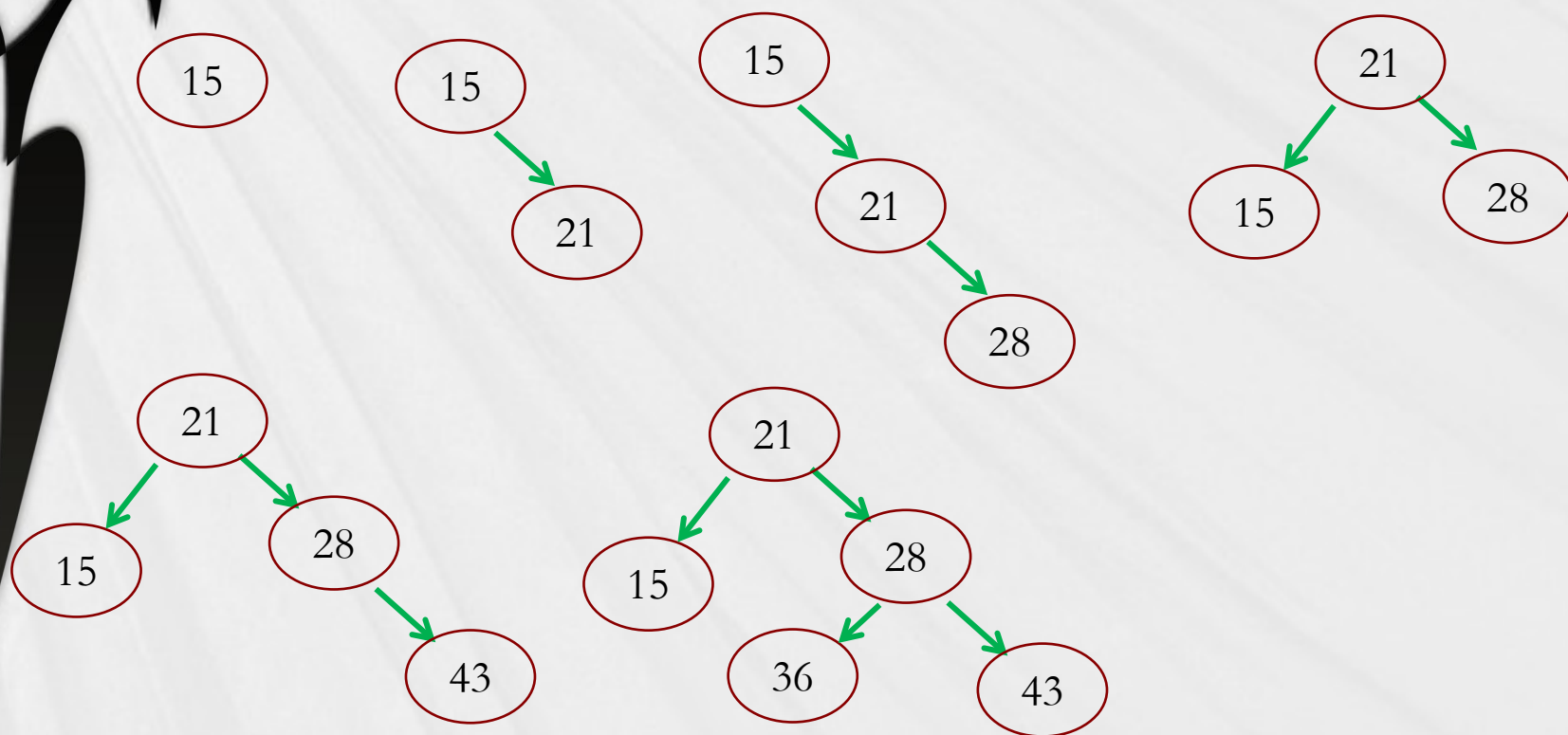


调整后



先向右旋转再左旋

【例4】已知关键字序列为（15，21，28，43，36，8，10），构造一棵平衡二叉树。



## 8.4 哈希表

- ❖ 前面：关键字与存储地址无直接关系
- ❖ 思路：由关键字直接生成数据元素的存储地址
- ❖  $H()$ ：建立存储位置和它的**关键字K**之间的对应关系，使每个关键字和一个唯一的存储位置相对应。
- ❖ 存储： $H(K)$
- ❖ 查找： $H(K)$
- ❖ **哈希函数，哈希地址**
- ❖ 不同的关键字映射到同一个哈希地址上，这种现象称为**冲突**，映射到同一哈希地址上的关键字成为**同义词**。
- ❖  $K1 \neq K2$        $H(K1) = H(K2)$
- ❖ 处理冲突方法

表

|  |
|--|
|  |
|  |
|  |
|  |
|  |

- ❖ 用一组连续的**有限**地址空间存储数据-----**表**
- ❖ 以关键字的哈希值 **$H(\text{key})$** 作为记录在表中的存储位置，这种表称为**哈希表**。
- ❖ 所得的存储位置称为**哈希地址或散列地址**。
- ❖ 这一映像过程称为**哈希造表或散列**。
- ❖ 关键： 哈希函数 $H()$

**表**

|  |
|--|
|  |
|  |
|  |
|  |
|  |

#### 8.4.1 哈希函数的构造方法---**少冲突， 不冲突**

1. 直接定址法
2. 特征位抽取法
3. 平方取中法
4. 折叠法
5. 除留余数法

### 8.4.1 哈希函数的构造方法---少冲突，不冲突

**直接定址法：**  $\text{Hash}(\text{key}) = \text{key}$  or  $a * \text{key} + b$

**特征位抽取法：** 抽取关键字中随机性好的几位作为哈希地址

8 0 4 6 5 3 2 6

8 0 7 2 2 4 2 6

8 0 8 1 3 6 5 7

8 0 1 3 6 7 5 7

a1 a2 a3 a4 a5 a6 a7 a8

### 8.4.1 哈希函数的构造方法---少冲突，不冲突

**平方取中法：**抽取关键字平方的中间几位作为哈希地址

$$\text{key}=471 \quad 471 \times 471 = 221841$$

**折叠法：**抽取关键字中若干位的运算结果作为哈希地址

$$\text{key} = a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8$$

$$\text{Hash}(\text{key}) = a_1 a_2 a_3 + a_4 a_5 a_6 + a_7 a_8$$

**除留余数法：**  $\text{Hash}(\text{key}) = \text{key} \% m$

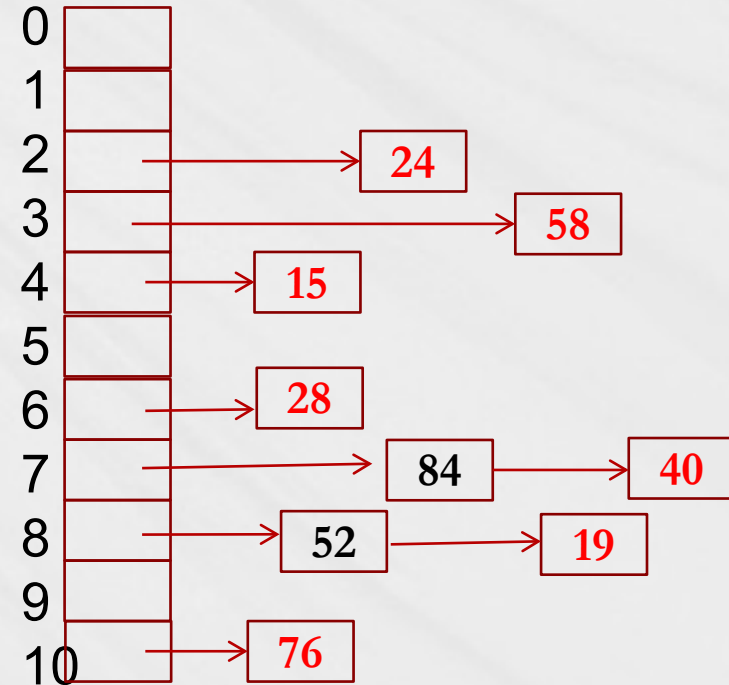
## 8.4.2 哈希冲突的解决方法

表

|  |
|--|
|  |
|  |
|  |
|  |
|  |
|  |

### 1. 闭散列法

### 2. 开散列法（链地址法）



指针  
数组



## 1. 闭散列法

基本思想：所有数据元素存储在**数组（散列表）**中。数组大小固定，可能溢出。

- ❖ 数据元素经散列函数计算出来的地址 **$H(\text{key})$** 称为**基地址**。
- ❖ 若插入元素x，而x的基地址已被某个同义词占用，则根据**某种策略**将x存储在数组的另一个空位置。
- ❖ 寻找“下一个”空位的过程称为**探测**。
- ❖ 探测地址可用如下公式表示（除留取余法）：

$$H_i = (H(\text{key}) + d_i) \% m$$

其中，m为表长， $d_i$ 为增量步长

### ◆ 探测方法

表

|  |
|--|
|  |
|  |
|  |
|  |
|  |
|  |

## (1) 线性探测再散列

- ❖  $d_i$  的取值为 1, 2, 3, ...,  $m-1$  的线性序列。
- ❖ 基本思想：当发生冲突时，从冲突位置的下一个单元 **顺序寻找** 可存放记录的空存储单元，只要找到一个空位，就把元素放入此空位中。

【例5】已知一组关键字为 (15, 19, 40, 24, 28, 58, 76, 52, 84)，散列表长  $m=13$ ，**散列函数**  $H(\text{key}) = \text{key} \% 11$ ，利用线性探测法解决冲突，构造散列表。

(15, 19, 40, 24, 28, 58, 76, 52, 84) % 11

4, 8, 7, 2, 6, 3, 10, 8, 7

(15, 19, 40, 24, 28, 58, 76, 52, 84) %11

4, 8, 7, 2, 6, 3, 10, 8, 7

|    |   |   |   |   |    |   |   |   |   |   |    |    |    |
|----|---|---|---|---|----|---|---|---|---|---|----|----|----|
| 地址 | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|    |   |   |   |   | 15 |   |   |   |   |   |    |    |    |

|    |   |   |   |   |    |   |   |   |    |   |    |    |    |
|----|---|---|---|---|----|---|---|---|----|---|----|----|----|
| 地址 | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8  | 9 | 10 | 11 | 12 |
|    |   |   |   |   | 15 |   |   |   | 19 |   |    |    |    |

|    |   |   |   |   |    |   |   |    |    |   |    |    |    |
|----|---|---|---|---|----|---|---|----|----|---|----|----|----|
| 地址 | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7  | 8  | 9 | 10 | 11 | 12 |
|    |   |   |   |   | 15 |   |   | 40 | 19 |   |    |    |    |

|    |   |   |    |   |    |   |   |    |    |   |    |    |    |
|----|---|---|----|---|----|---|---|----|----|---|----|----|----|
| 地址 | 0 | 1 | 2  | 3 | 4  | 5 | 6 | 7  | 8  | 9 | 10 | 11 | 12 |
|    |   |   | 24 |   | 15 |   |   | 40 | 19 |   |    |    |    |

|    |   |   |    |   |    |   |    |    |    |   |    |    |    |
|----|---|---|----|---|----|---|----|----|----|---|----|----|----|
| 地址 | 0 | 1 | 2  | 3 | 4  | 5 | 6  | 7  | 8  | 9 | 10 | 11 | 12 |
|    |   |   | 24 |   | 15 |   | 28 | 40 | 19 |   |    |    |    |

(15, 19, 40, 24, 28, 58, 76, 52, 84) %11

4, 8, 7, 2, 6, 3, 10, 8, 7

|    |   |   |    |    |    |   |    |    |    |   |    |    |    |
|----|---|---|----|----|----|---|----|----|----|---|----|----|----|
| 地址 | 0 | 1 | 2  | 3  | 4  | 5 | 6  | 7  | 8  | 9 | 10 | 11 | 12 |
|    |   |   | 24 | 58 | 15 |   | 28 | 40 | 19 |   |    |    |    |

|    |   |   |    |    |    |   |    |    |    |   |    |    |    |
|----|---|---|----|----|----|---|----|----|----|---|----|----|----|
| 地址 | 0 | 1 | 2  | 3  | 4  | 5 | 6  | 7  | 8  | 9 | 10 | 11 | 12 |
|    |   |   | 24 | 58 | 15 |   | 28 | 40 | 19 |   | 76 |    |    |

|    |   |   |    |    |    |   |    |    |    |    |    |    |    |
|----|---|---|----|----|----|---|----|----|----|----|----|----|----|
| 地址 | 0 | 1 | 2  | 3  | 4  | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|    |   |   | 24 | 58 | 15 |   | 28 | 40 | 19 | 52 | 76 |    |    |

|    |   |   |    |    |    |   |    |    |    |    |    |    |    |
|----|---|---|----|----|----|---|----|----|----|----|----|----|----|
| 地址 | 0 | 1 | 2  | 3  | 4  | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|    |   |   | 24 | 58 | 15 |   | 28 | 40 | 19 | 52 | 76 | 84 |    |

## (2) 二次探测再散列：加大跳跃步长

❖  $d_i = 12, -12, 22, -22, \dots, k2$  ( $k \leq \lfloor m/2 \rfloor$ )

## (3) 双重散列法

❖ 双重散列法是以关键字的**另一个散列函数值**作为步长增量。设两个散列函数为： $H1()$  和  $H2()$ ，则得到的探测序列为：

$$(H1(key) + H2(key)) \% m, (H1(key) + 2H2(key)) \% m$$

$$(H1(key) + 3H2(key)) \% m, \dots$$

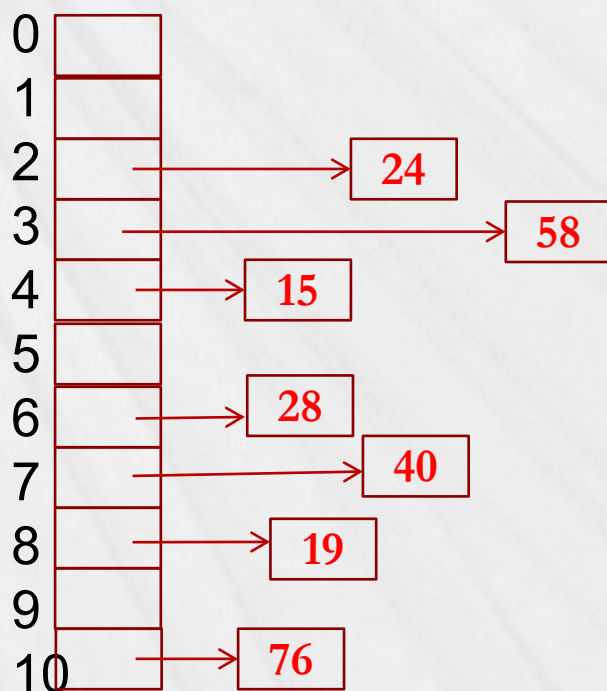
## 2. 开散列法（链地址法）

- ❖ 具有相同哈希函数值的记录都可链到同一链表中；
- ❖ 由于用链表解决冲突，所以这种解决冲突的方法有时称**链地址法**。

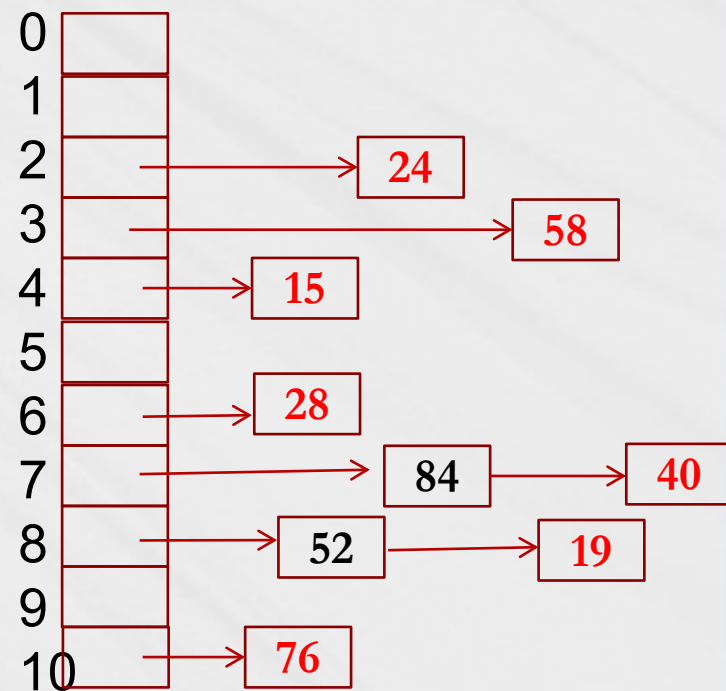
【例6】已知一组关键字为（15，19，40，24，28，58，76，**52，84**），散列函数为 $H(\text{key}) = \text{key} \% 11$ ，利用链地址法解决冲突，构造散列表。

(15, 19, 40, 24, 28, 58, 76, 52, 84) %11

4, 8, 7, 2, 6, 3, 10, 8, 7



指针  
数组



指针  
数组




### 3. 散列表的查找

散列表查找过程和散列表的生成相似：

首先根据选定的哈希函数计算出给定关键字的哈希地址

- (1) 若该地址为空，则查找失败。
- (2) 若该地址不空，且所存关键字的值恰好等于所查关键字，则查找成功。
- (3) 若该地址不空，但所存关键字的值不等于所查关键字，则**按着造表时使用的解决冲突的方法**，继续查找，直到成功或失败。

- 
- ❖ 线性探测法的散列表查找成功时的平均查找长度为：

$$ASL = (1 + 1 / (1 - \alpha)) / 2$$

- ❖ 链地址法的散列表查找成功时的平均查找长度为：

$$ASL = 1 + \alpha / 2$$

- ❖ 其中 $\alpha$ 装填因子， $\alpha$ =表中记录数/散列表长度， $\alpha$ 越小，发生冲突的可能性就越小。 $\alpha$ 越大，发生冲突的可能性就越大，查找时比较次数就越多。

# 本章小结

- ❖ 查找是数据结构中的一种重要操作，为了提高查找的效率，需要专门为查找操作设置数据结构，即查找表。
- ❖ 查找表有静态查找表、动态查找表和散列表。
- ❖ 查找表中的数据元素有若干属性，其中有主关键字和次关键字。
- ❖ 静态查找表上的查找主要有顺序查找、折半查找和分块查找。
- ❖ 动态查找表上的查找方法主要有二叉排序树和平衡二叉树。
- ❖ 哈希表上的查找和哈希造表的过程基本一致。
- ❖ 衡量查找算法性能的主要依据是平均查找长度。和给定值进行比较的关键字次数的“期望值”称为查找算法的平均查找长度。

# 本章习题

1. 假设对有序表 (5, 10, 24, 31, 42, 55, 63, 72, 87, 99) 进行二分查找, 试回答下列问题:
  - ◆ 若查找元素72, 需进行几次比较?
  - ◆ 假定每个元素的查找概率相等, 求查找成功时的平均查找长度。
2. 已知8个元素为 (34, 76, 45, 20, 25, 54, 93, 66), 按照依次插入结点的方法生成一棵二叉排序树。
3. 已知长度为12的表 {Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}。按表中元素的次序依次插入一棵初始状态为空的二叉排序树, 并求在等概率情况下查找成功的平均查找长度。

# 本章习题

4. 设哈希函数 $H(k) = 3k \% 11$ ，哈希地址空间为0到10，对关键字序列（32，13，49，24，38，21，15，12）按下述两种解决冲突的方法构造哈希表：（1）线性探测法（2）链地址法。