

# 第七章

## 查找

## 本章的主要内容:

7.1 查找的基本概念

7.2 基于线性表的查找

7.3 树表查找

7.4 散列表及其查找

# 7.1 查找的基本概念

**1、查找：**所谓查找就是在具有相同类型的数据元素（或记录）构成的集合中找出满足给定条件的数据元素（或记录）。

**查找的结果：**若在查找集合中找到了与给定值相匹配的记录，则称查找成功；否则，称查找失败。

例如：一个职工档案表

职工号	姓名	性别	年龄	参加工作
19900019	王刚	男	51	1990年7月
19950052	张亮	男	48	1995年7月
20000103	刘楠	女	45	2000年9月
20050147	齐梅	女	40	2005年9月
20100205	王刚	男	35	2010年9月

给定一个职工号，如果表中存在，则可以查到相应的职工信息，即查找成功；否则，查找失败。

给定一个姓名，可能找到多个同名的职工信息。

**2、关键码：**可以标识一个数据元素（或记录）的某个数据项。

**主关键码：**可以**唯一地**标识一个数据元素（或记录）的关键码。如：职工号

**次关键码：**不能唯一地标识一个数据元素（或记录）的关键码。如：姓名

职工号	姓名	性别	年龄	参加工作
19900019	王刚	男	51	1990年7月
19950052	张亮	男	48	1995年7月
20000103	刘楠	女	45	2000年9月
20050147	齐梅	女	40	2005年9月
20100205	王刚	男	35	2010年9月

# 那么，如何进行查找呢？

由于查找集合中的数据元素之间不存在明显的组织规律，因此不便于查找。

通常，为了提高查找的效率，需要在查找集合中的元素之间人为地附加某种确定的关系，换句话说，用另外一种结构来表示查找集合。

因此，查找的方法取决于查找集合的结构。

例如：

- 预排序

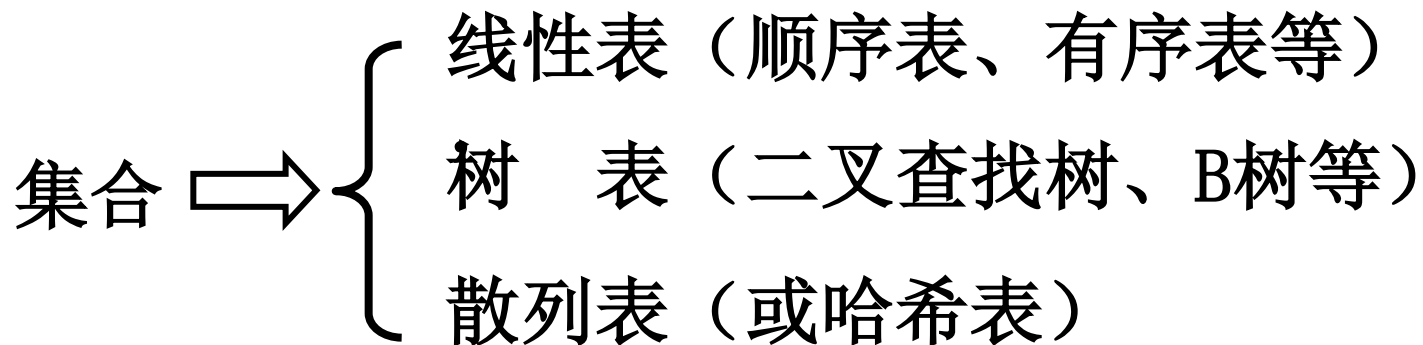
- 排序算法本身比较费时
- 只是预处理（在查找之前已经完成）

- 建立索引

- 查找时充分利用辅助索引信息
- 牺牲一定的空间
- 从而提高查找效率

3、查找集合的结构（也叫查找结构或查找表）：面向查找操作的数据结构，即基于查找的数据结构。

主要包括：





## 4、查找算法的性能

通过关键码的比较次数来度量查找算法的时间性能。

① 关键码的比较次数与哪些因素有关呢？

(1) 算法

(2) 问题规模

(3) 待查关键码在查找集合中的位置

② 同一查找集合、同一查找算法，关键码的比较次数与哪些因素有关呢？

查找算法的时间复杂度是问题规模 $n$ 和待查关键码在查找集合中的位置 $k$ 的函数，记为 $T(n, k)$ 。

通常以“**关键码比较次数的平均值**”作为衡量查找算法好坏的依据。

为此，将查找算法进行的关键码的比较次数的数学期望值定义为**平均查找长度**。计算公式为：

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

其中： $n$ ：问题规模，查找集合中的记录个数；

$p_i$ ：查找第 $i$ 个记录的概率；

$c_i$ ：查找第 $i$ 个记录所需的关键码的比较次数。

**结论：** $c_i$ 取决于算法； $p_i$ 与算法无关，取决于具体应用。  
如果 $p_i$ 是已知的，则**平均查找长度只是问题规模的函数**。

- 假设线性表为 (a, b, c)，查找a、b、c的概率分别为0.4、0.1、0.5，则
  - 顺序查找算法的**平均查找长度**为 $0.4 \times 1 + 0.1 \times 2 + 0.5 \times 3 = 2.1$
  - 即平均需要2.1次给定值与表中关键码值的比较才能找到待查元素
- 但是，如果**调整**一下顺序 (c, a, b)，则
  - 顺序查找算法的**平均查找长度**为 $0.5 \times 1 + 0.4 \times 2 + 0.1 \times 3 = 1.6 < 2.1$
- 智能输入法举例

## 7.2 基于线性表的查找

### 一、顺序查找 (Sequential Search)

**基本思想：**从线性表的一端向另一端逐个将关键码与给定值进行比较，若相等，则查找成功，给出该记录在表中的位置；若整个表检测完仍未找到与给定值相等的关键码，则查找失败，给出失败信息。

例：查找 $k=35$

0	1	2	3	4	5	6	7	8	9
	10	15	24	6	12	35	40	98	55
						$\uparrow_i$	$\uparrow_i$	$\uparrow_i$	$\uparrow_i$

## 顺序查找算法:

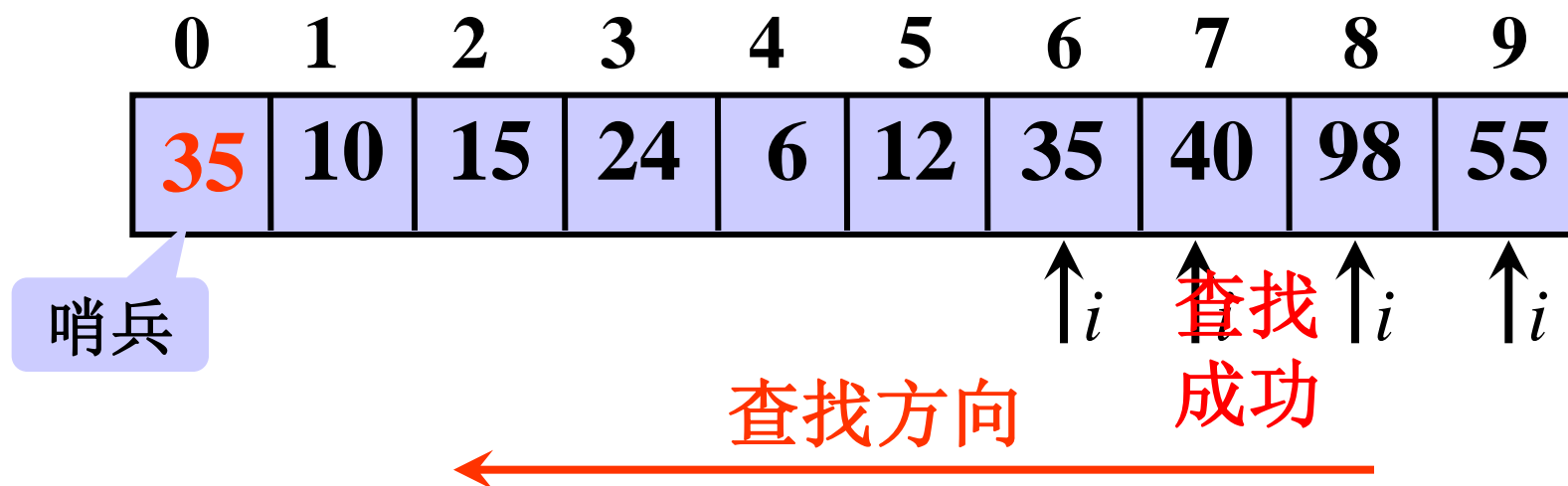
```
int SeqSearch1 (int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    int i=n;
    while (i>0 && r[i]!=k)    //判断是否越界
        i--;
    return i;
}
```

② 这个算法可否改进呢?

# 改进的顺序查找

**基本思想：**设置“哨兵”。哨兵就是待查值，将它放在查找方向的尽头处，免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度

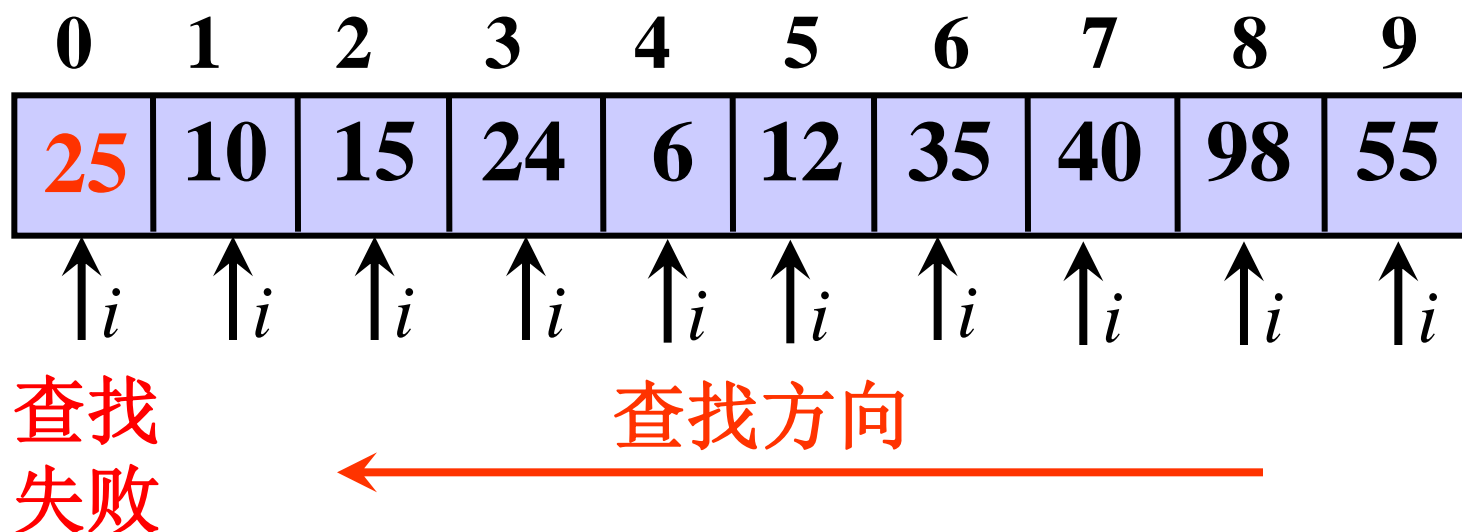
例：查找 $k=35$



## 改进的顺序查找

**基本思想：**设置“哨兵”。哨兵就是待查值，将它放在查找方向的尽头处，免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度

### 例：查找 $k=25$



## 改进的顺序查找算法:

```
int SeqSearch2(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    r[0]=k;           //设置哨兵
    int i=n;
    while (r[i]!=k) //（免去了“i>0”的判断）
        i--;
    return i;
}
```

实践表明：这个改进在表长较大时，进行一次顺序查找的平均时间几乎减少一半。请同学们验证一下！



# 顺序查找性能分析

- 查找成功

假设查找每个关键码是等概率的,  $P_i = 1/n$

$$\begin{aligned} ASL &= \sum_{i=1}^n p_i c_i \\ &= 1/n \sum_{i=1}^n (n - i + 1) \\ &= (n+1)/2 \end{aligned}$$

## 顺序查找性能分析（续1）

### ●查找失败

假设查找失败时都需要比较 $n+1$ 次（设置了一个监视哨），并假设查找成功的概率为 $p$ ，查找失败的概率为 $q=(1-p)$ ，则平均查找长度为

$$\begin{aligned} \text{ASL} &= p \cdot \frac{n+1}{2} + q \cdot (n+1) \\ &= p \cdot \frac{n+1}{2} + (1-p)(n+1) \\ &= (n+1)(1-p/2) \end{aligned}$$

即，  $(n+1)/2 \leq \text{ASL} \leq (n+1)$

## 顺序查找性能分析（续2）

在不等概率查找的情况下， $ASL$ 在  
 $P_n \geq P_{n-1} \geq \dots \geq P_2 \geq P_1$   
时取极小值。

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。或者在每个元素中附设一个访问频度域，并使表中的记录始终保持按访问频度非递减的次序排列，以提高查找效率。

考虑：查找概率的线性调整在中文输入法中的应用。

## 结论:

顺序查找算法的**平均查找长度较大**，特别是当待查找集合中元素较多时，**查找效率较低 $O(n)$** 。

但是，算法简单而且使用面广。

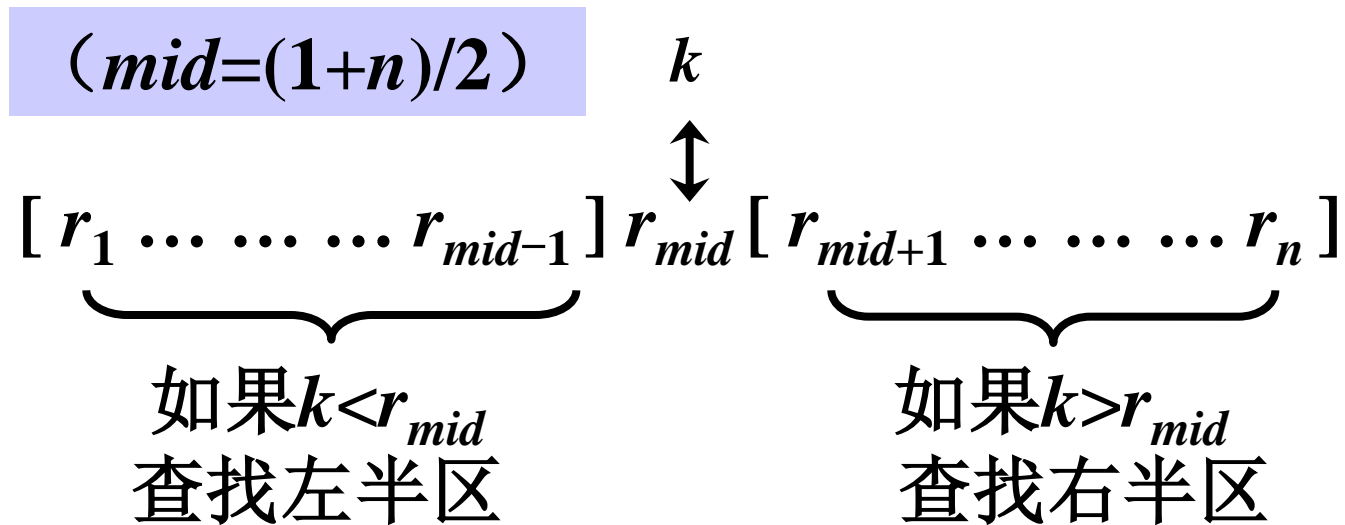
- 对表中记录的**存储没有任何要求**，顺序存储和链式存储均可；**插入元素可以直接加在表尾**。
- 对表中记录的**有序性也没有要求**，无论记录是否按**关键码有序**均可。

## 二、折半查找(Binary Search)

### 基本思想：

在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

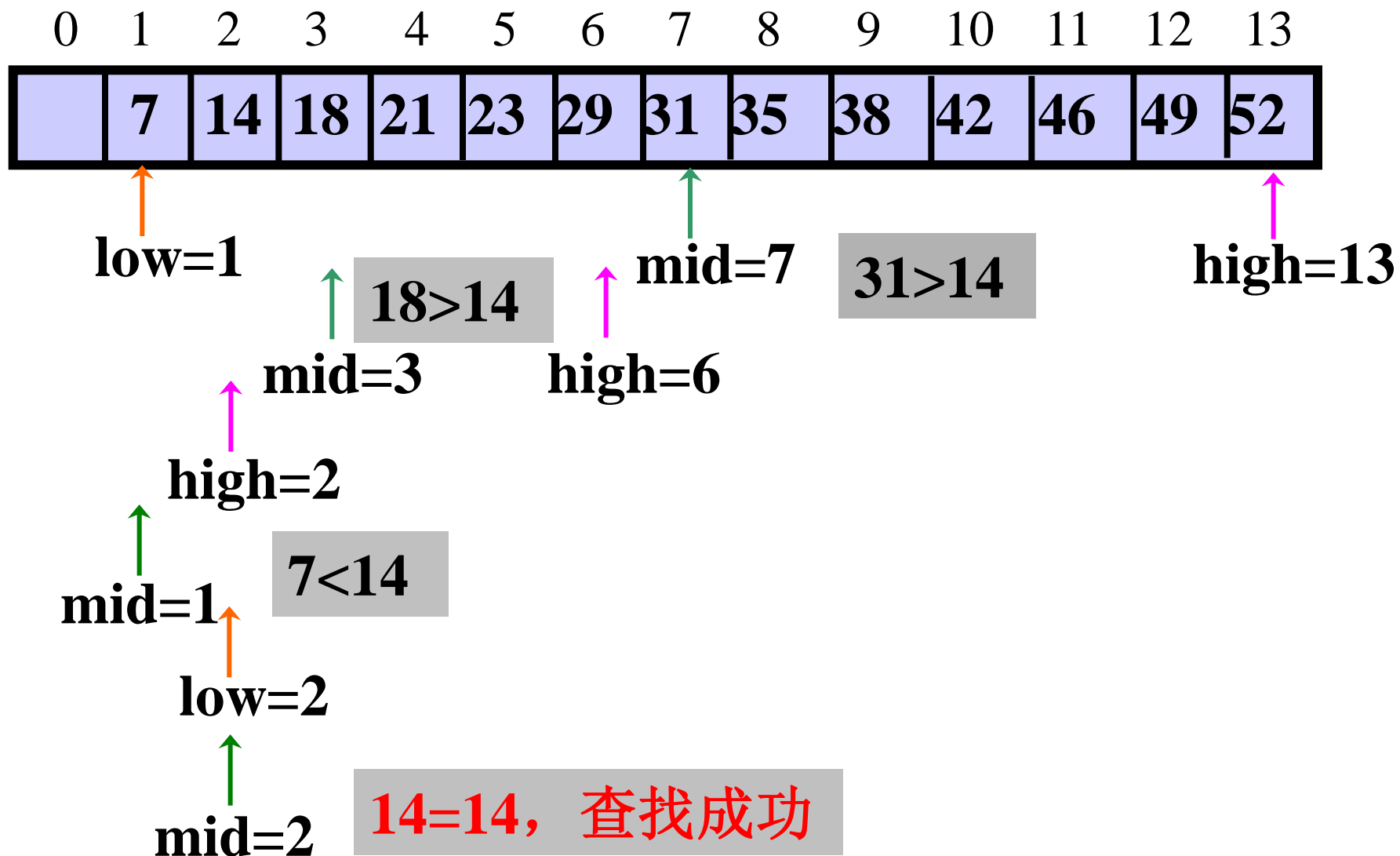
# 折半查找的基本思想图示:



## 折半查找使用条件:

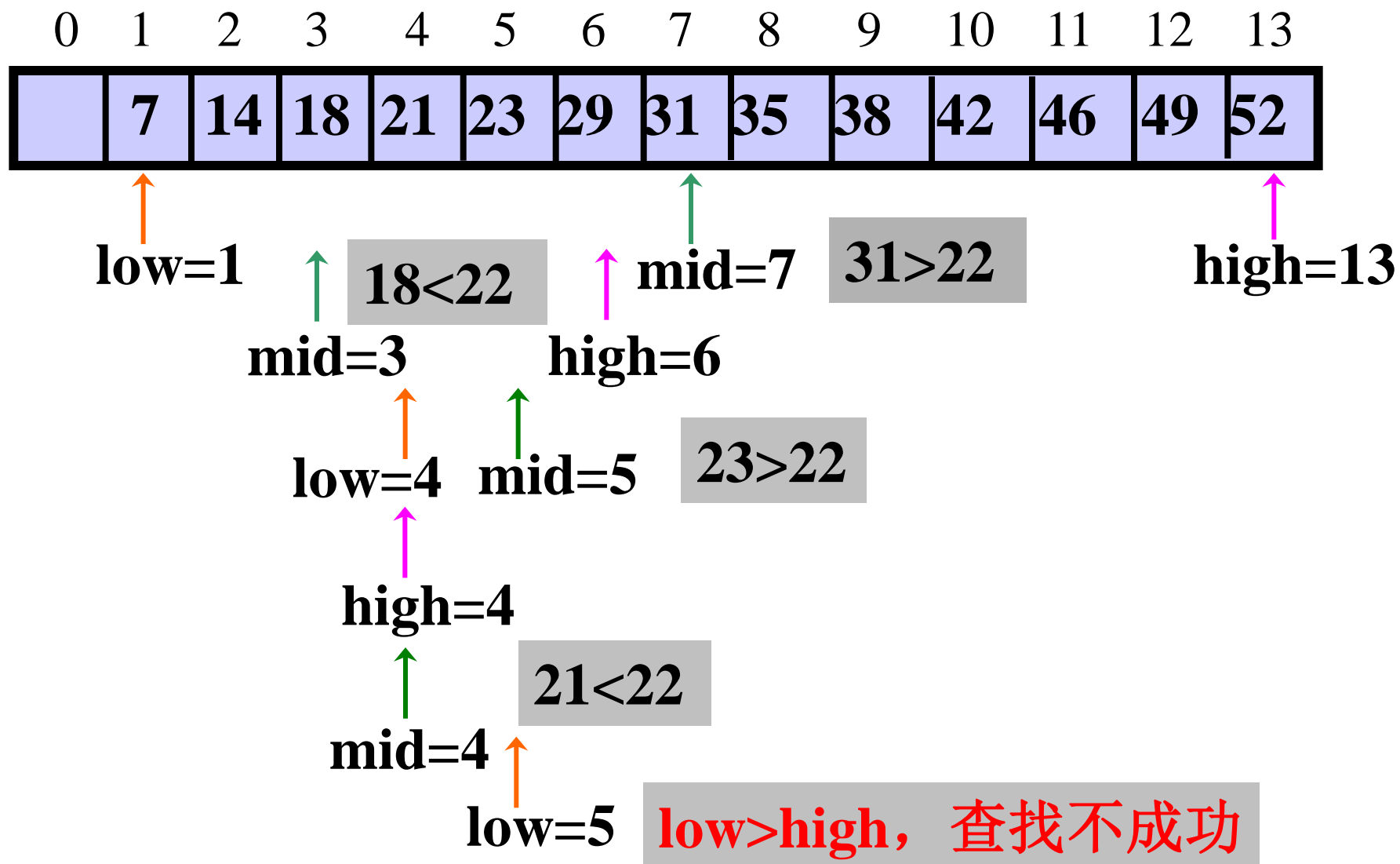
- 线性表中的记录必须按关键码**有序**;
- 必须采用**顺序**存储（对线性链表无法进行折半查找）。

例：查找值为14的记录的过程：





例：查找值为22的记录的过程：



## 折半查找算法描述:

```
int BinSearch1(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    int low=1, high=n;           //设置查找区间
    while (low<=high)           //当区间存在时
    {
        int mid=(low+high)/2;
        if (k<r[mid]) high=mid-1; //左半区
        else if (k>r[mid]) low=mid+1; //右半区
        else return mid;         //查找成功, 返回元素序号
    }
    return 0;                   //查找失败, 返回0
}
```



依据折半查找的执行过程, 可否实现相应的递归算法?

## 折半查找递归实现:

```
int BinSearch2(int r[ ], int low, int high, int k)
//数组r[1] ~ r[n]存放查找集合
{
    if (low > high) return 0;    //递归的边界条件
    else {
        mid = (low + high) / 2;
        if (k < r[mid]) return BinSearch2(r, low, mid - 1, k);
            //查找在左半区进行
        else
            if (k > r[mid]) return BinSearch2(r, mid + 1, high, k);
                //查找在右半区进行
            else return mid;
    }
}
```

# 折半查找性能分析

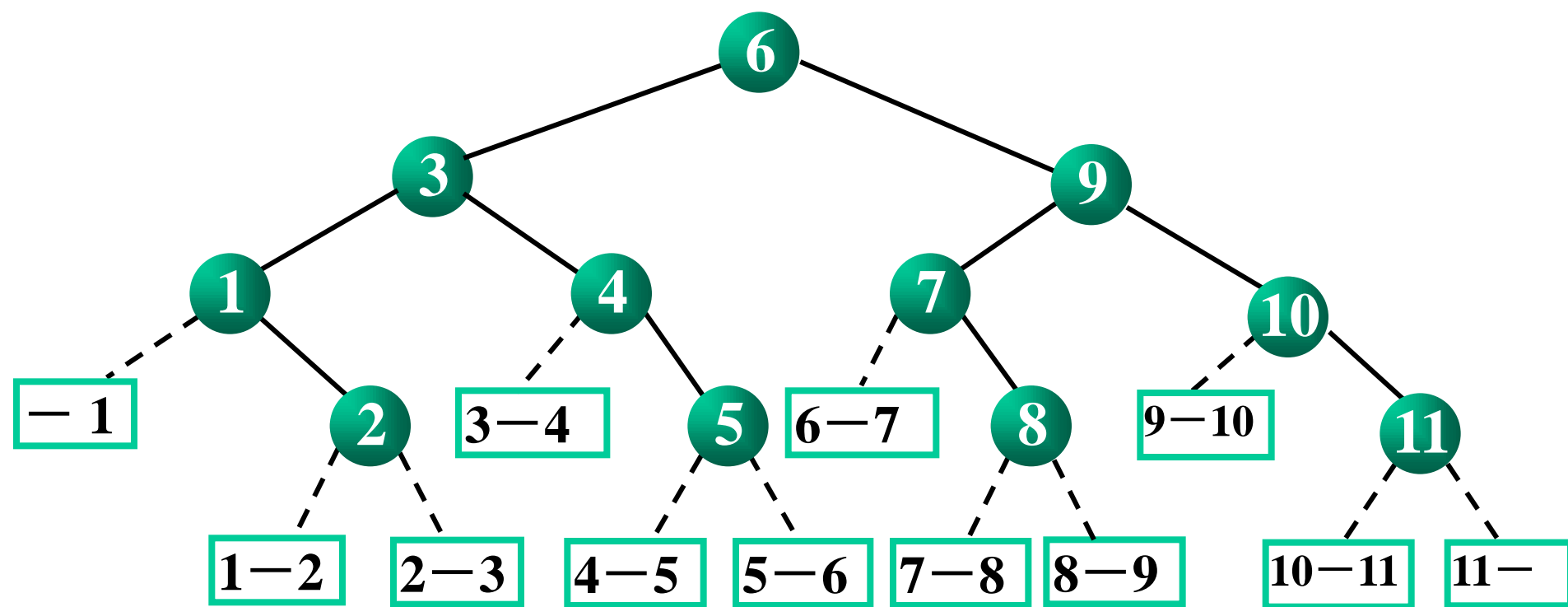
折半查找的过程可以用二叉树来描述，树中的每个结点对应有序表中的一个记录，结点的值为该记录在表中的位置。通常称这个描述折半查找过程的二叉树为折半查找判定树，简称判定树。

# 判定树的构造方法

- (1) 当 $n=0$ 时，折半查找判定树为空；
- (2) 当 $n>0$ 时，折半查找判定树的根结点是有序表中序号为 $\text{mid}=(n+1)/2$ 的记录，根结点的左子树是与有序表 $r[1] \sim r[\text{mid}-1]$ 相对应的折半查找判定树，根结点的右子树是与 $r[\text{mid}+1] \sim r[n]$ 相对应的折半查找判定树。

例如，构造一个具有11结点的判定树：

# 判定树的构造方法（续）

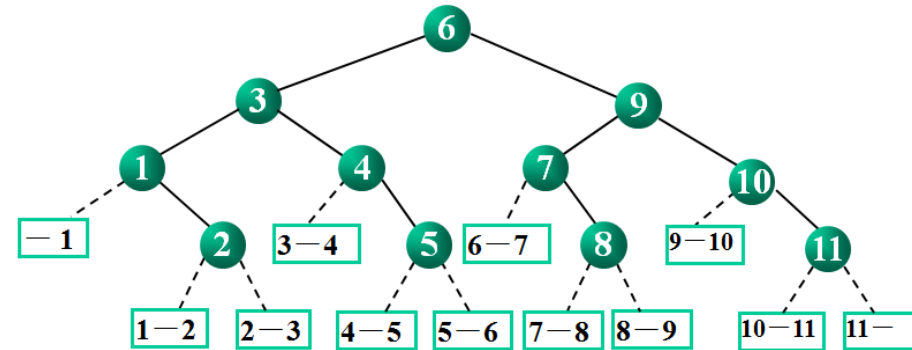


● 内部结点对应查找成功的情况

□ 外部结点对应查找不成功的情况

具有11个结点的判定树

# 折半查找的性能分析




**查找成功：**在表中查找任一记录的过程，即是折半查找判定树中从根结点到该记录结点的路径，和给定值的比较次数等于该记录结点在树中的层数。所以，

$$ASL=(1*1+2*2+4*3+4*4)/11=3$$

**查找不成功：**查找失败的过程就是走了一条从根结点到外部结点的路径，和给定值进行的关键码的比较次数等于该路径上内部结点的个数。所以，

$$ASL=(4*3+8*4)/12=3.67$$

# 折半查找的性能分析（续）

一般地，具有 $n$ 个结点的折半查找判定树的深度为 $\lfloor \log_2 n \rfloor + 1$ 。 

假设表中每个结点的查找概率相同，则**平均查找长度**（在假设表中结点数 $n=2^k-1$ 的情况下）为

$$\begin{aligned} ASL &= \sum_{i=1}^n p_i c_i = \frac{1}{n} \sum_{j=1}^k j \times 2^{j-1} \\ &= \frac{1}{n} (1 \times 2^0 + 2 \times 2^1 + \dots + k \times 2^{k-1}) \\ &= \frac{n+1}{n} \log_2(n+1) - 1 \end{aligned}$$

当 $n$ 较大时可得：

$$ASL \approx \log_2(n+1) - 1$$

所以，折半查找的平均时间复杂度为

$$O(\log_2 n) < O(n)$$



## 7.3 树表查找

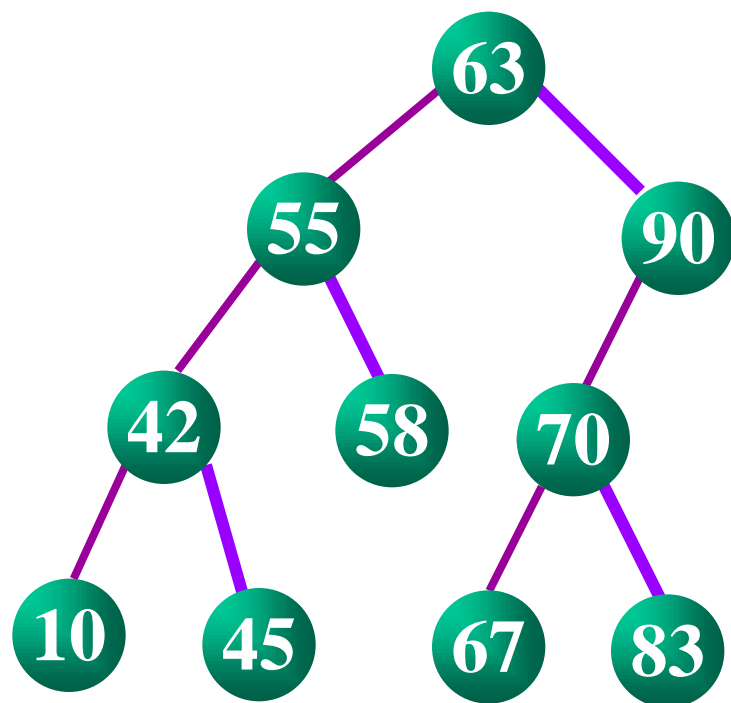
### 二叉排序树(binary sort tree)

**二叉排序树**（也称**二叉查找树**）：或者是一棵空的二叉树，或者是具有下列性质的二叉树：

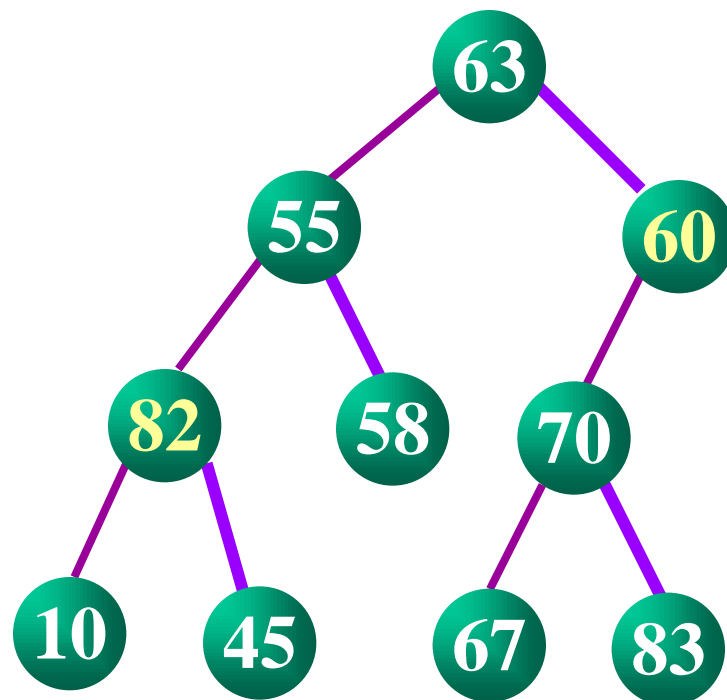
- (1) 若它的左子树不空，则左子树上所有结点的值均小于根结点的值；
- (2) 若它的右子树不空，则右子树上所有结点的值均大于根结点的值；
- (3) 它的左右子树也都是二叉排序树。

二叉排序树的定义采用的是递归方法。

## 二叉排序树示例:



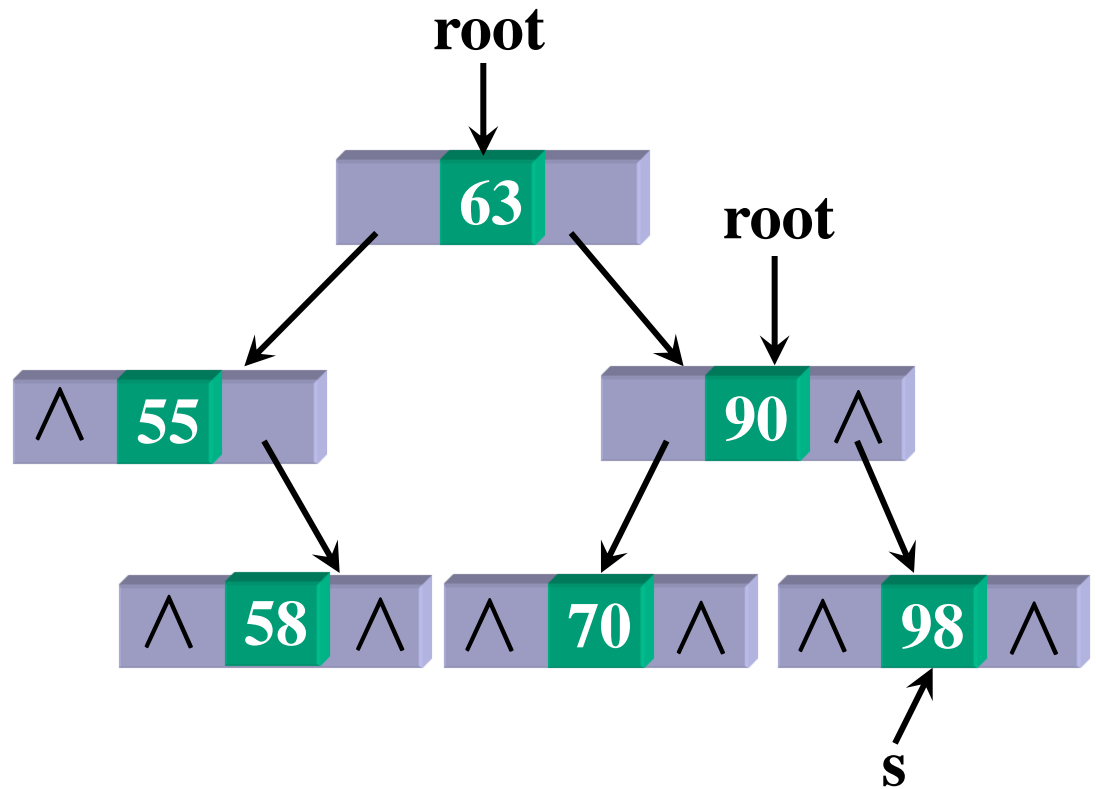
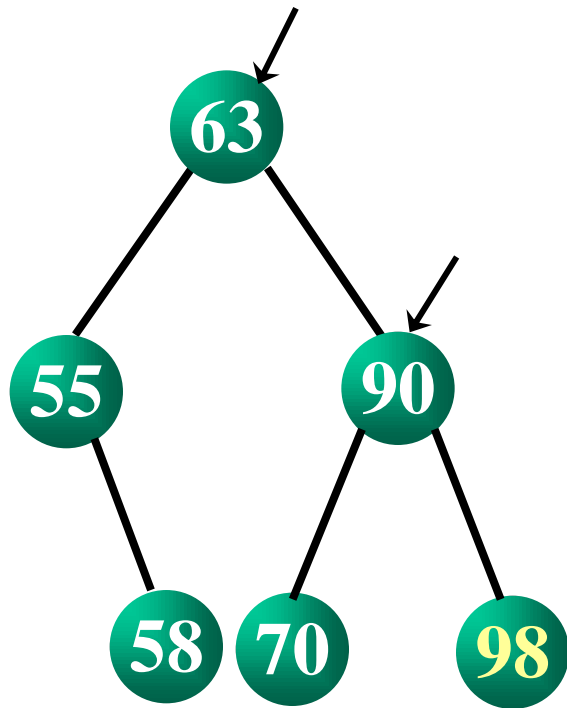
二叉排序树



非二叉排序树

中序遍历二叉排序树可以得到一个按关键码有序的序列

例：插入值为98的结点



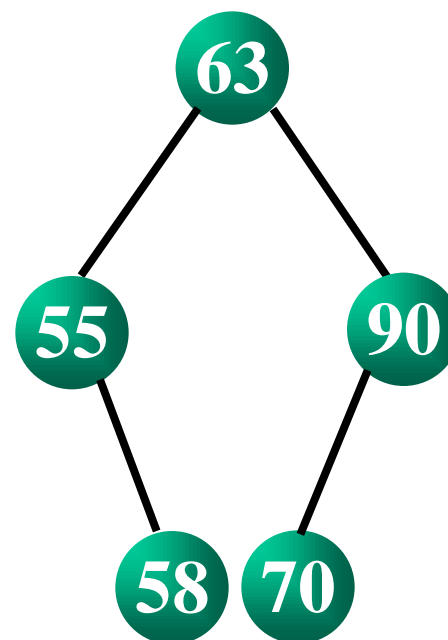
## 二叉排序树的构造过程:

从空的二叉排序树开始，依次插入一个个结点。

例：关键码集合为

**{63, 90, 70, 55, 58}**,

二叉排序树的构造过程为:



## 小结:

- 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列;
- 每次插入的新结点都是二叉排序树上新的叶子结点;
- 找到插入位置后, 不必移动其它结点, 仅需修改某个结点的指针;
- 在左子树/右子树的查找过程与在整棵树上查找过程相同;
- 新插入的结点没有破坏原有结点之间的关系。

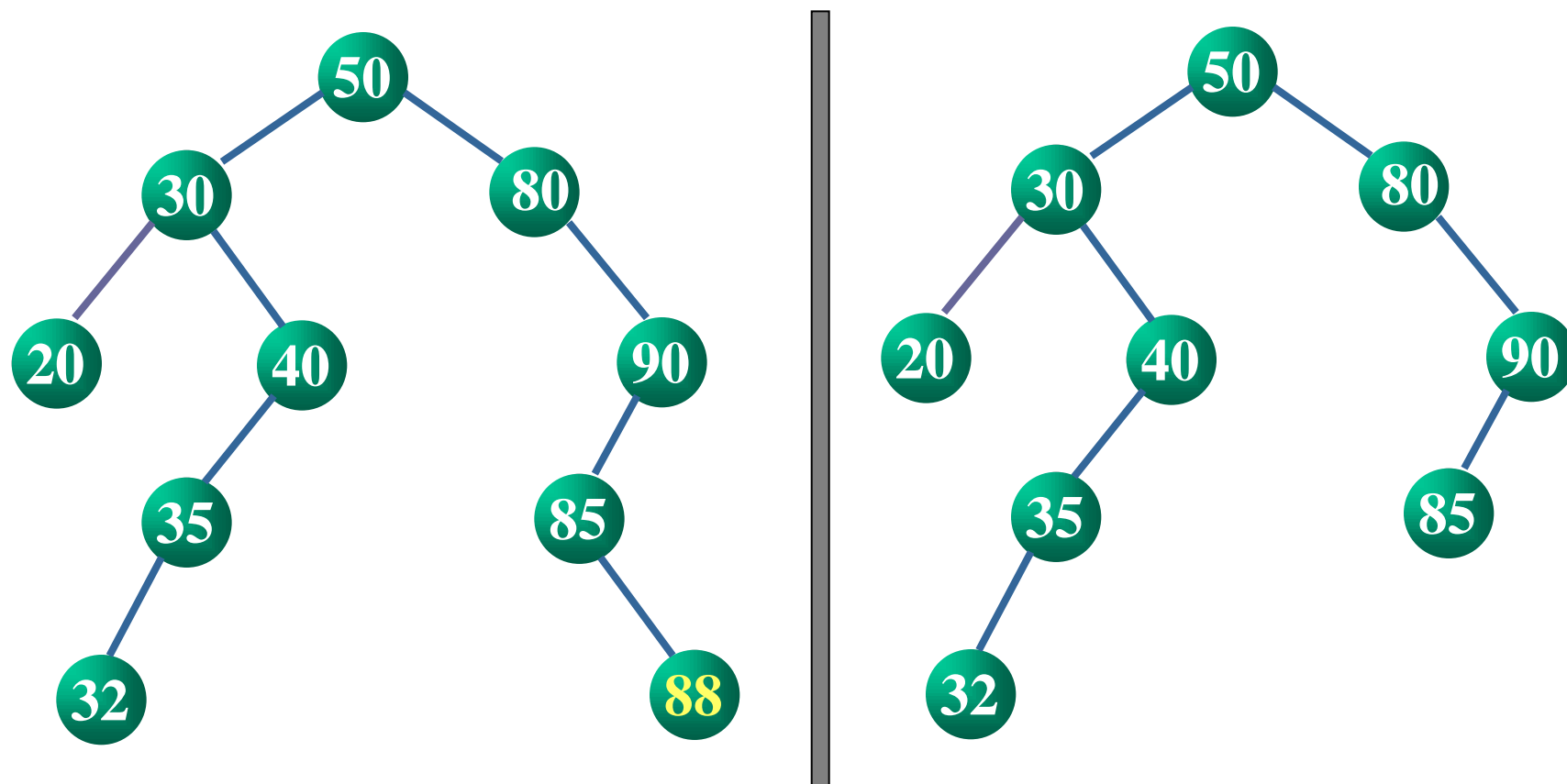
## 二叉排序树的删除操作

在二叉排序树上删除某个结点之后，仍然保持二叉排序树的特性。

分三种情况讨论：

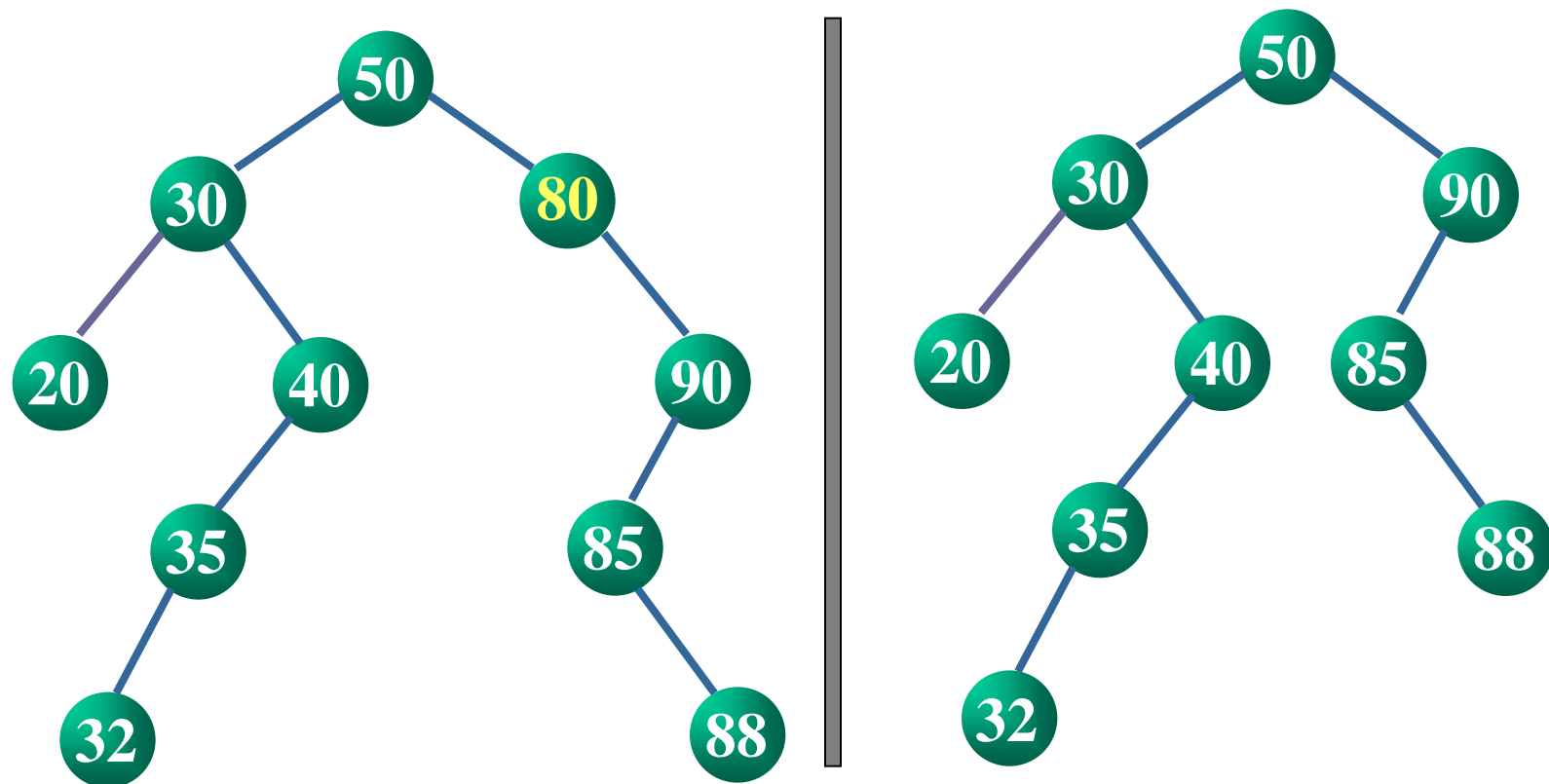
- 被删除的结点是叶子
- 被删除的结点只有左子树或者只有右子树
- 被删除的结点既有左子树，也有右子树

## 情况1——被删除的结点是叶子结点



**操作：** 将双亲结点中相应指针域的值改为空。

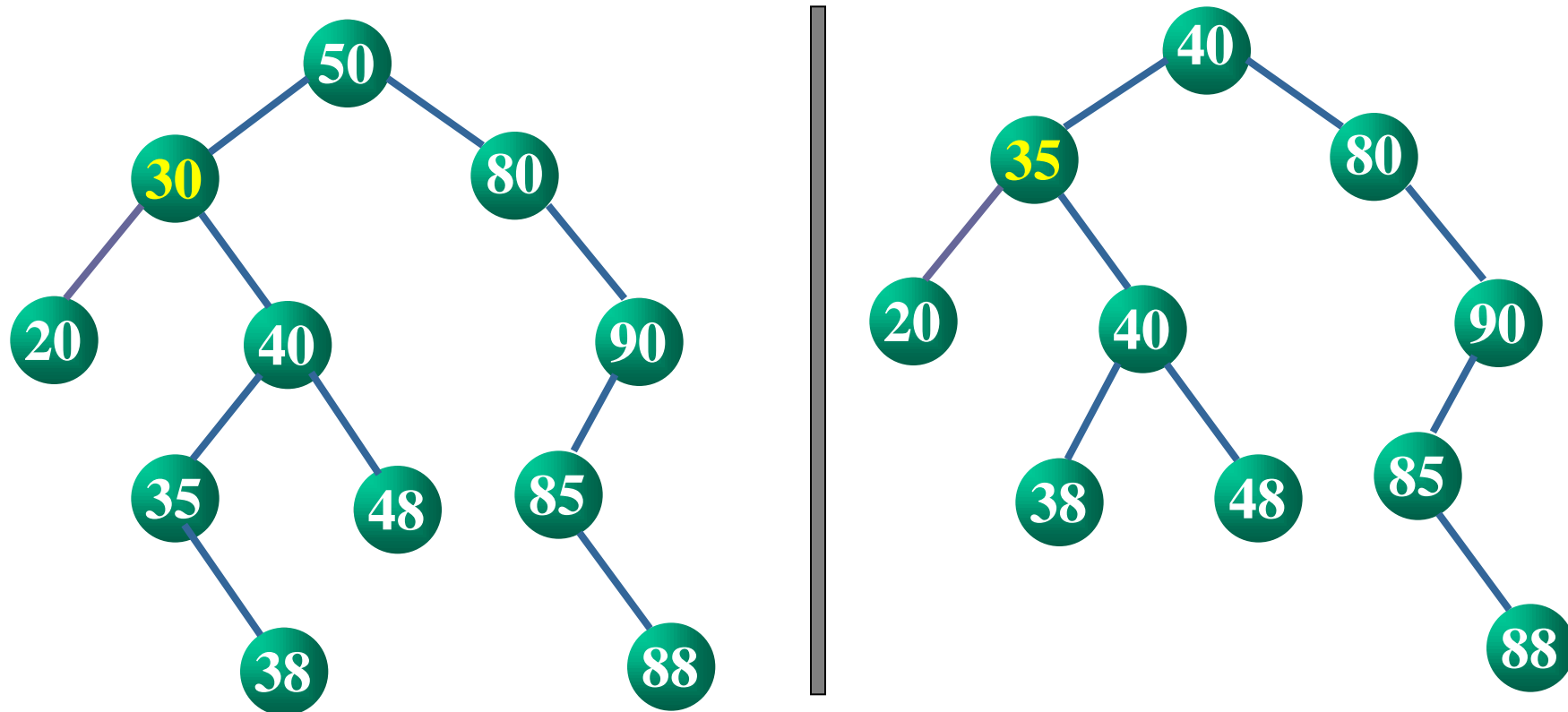
## 情况2——被删除的结点只有左子树或者只有右子树



**操作：**将双亲结点的相应指针域的值指向被删除结点的左子树（或右子树）。

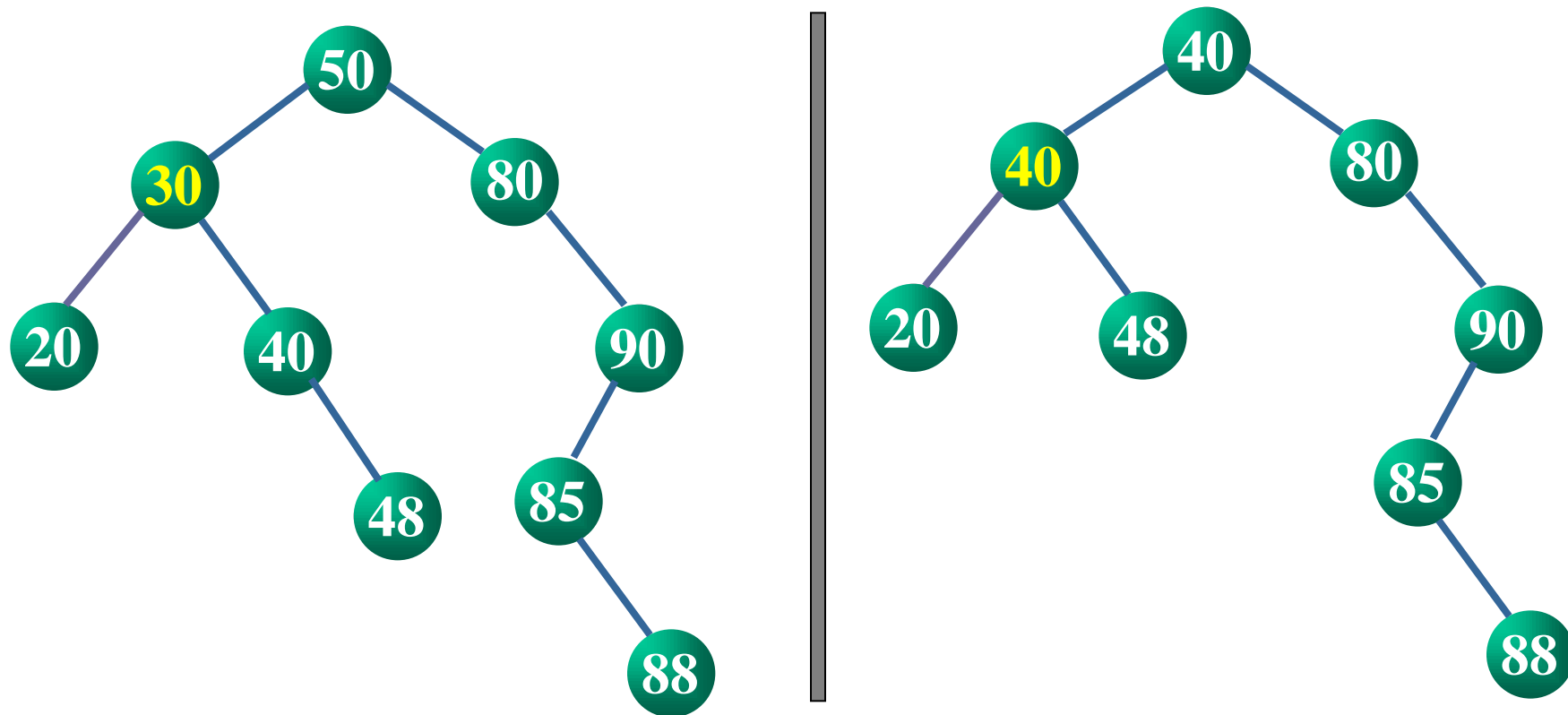


# 情况3——被删除的结点既有左子树也有右子树



**操作：** 以其后继（右子树中的最左下结点）  
替代之，然后再删除该后继结点。

## 情况3——被删除的结点既有左子树也有右子树



**特殊情况：**被删结点的右孩子就是其右子树中的最小结点

## 二叉排序树的查找操作

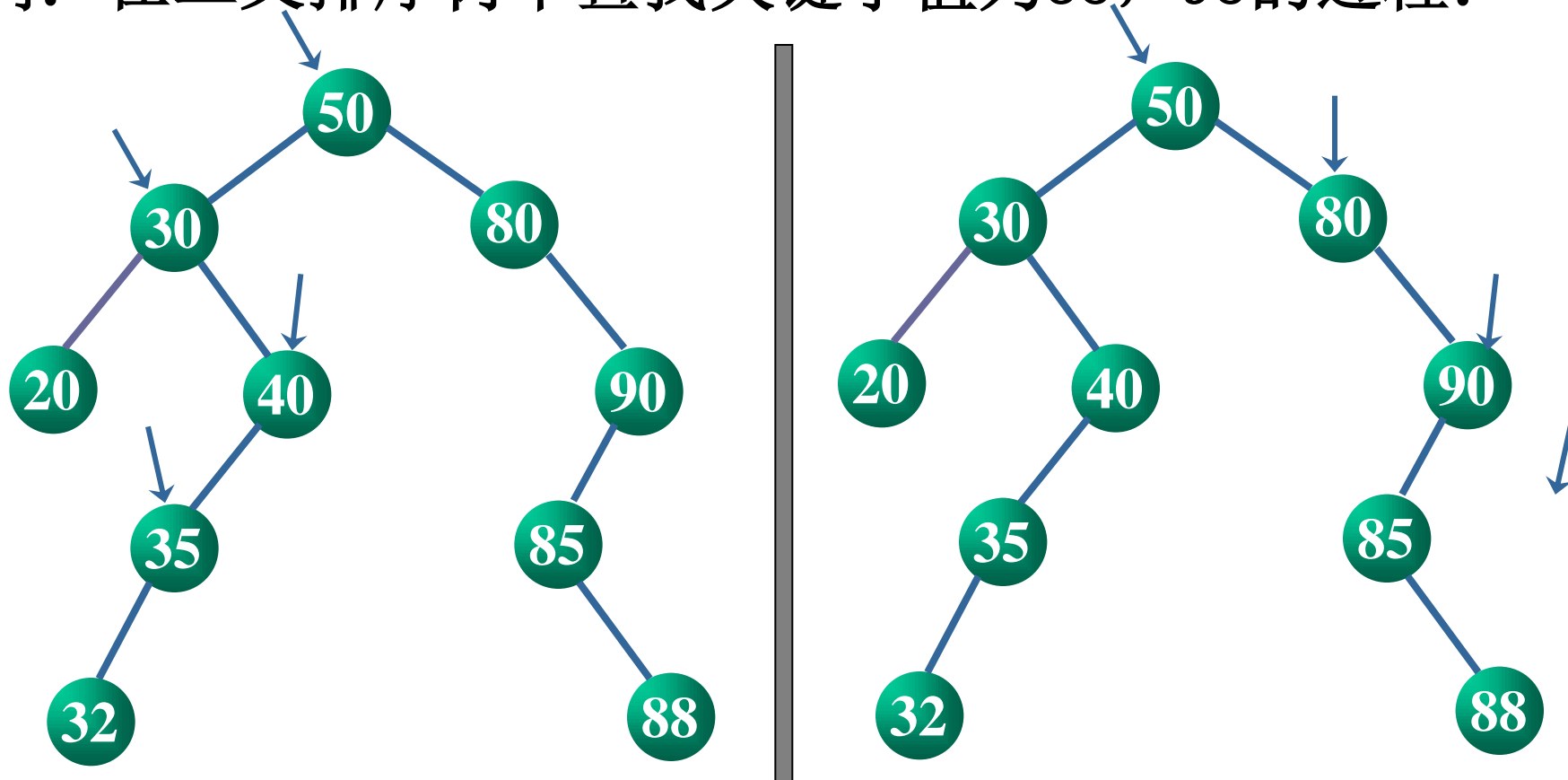
在二叉排序树中查找给定值 $k$ 的过程是：

- (1) 若 $root$ 是空树，则查找失败；
- (2) 若 $k = root \rightarrow data$ ，则查找成功；否则
- (3) 若 $k < root \rightarrow data$ ，则在 $root$ 的左子树上查找；否则
- (4) 在 $root$ 的右子树上查找。

上述过程一直持续到 $k$ 被找到或者待查找的子树为空，如果待查找的子树为空，则查找失败。

## 二叉排序树的查找

例：在二叉排序树中查找关键字值为35，95的过程：



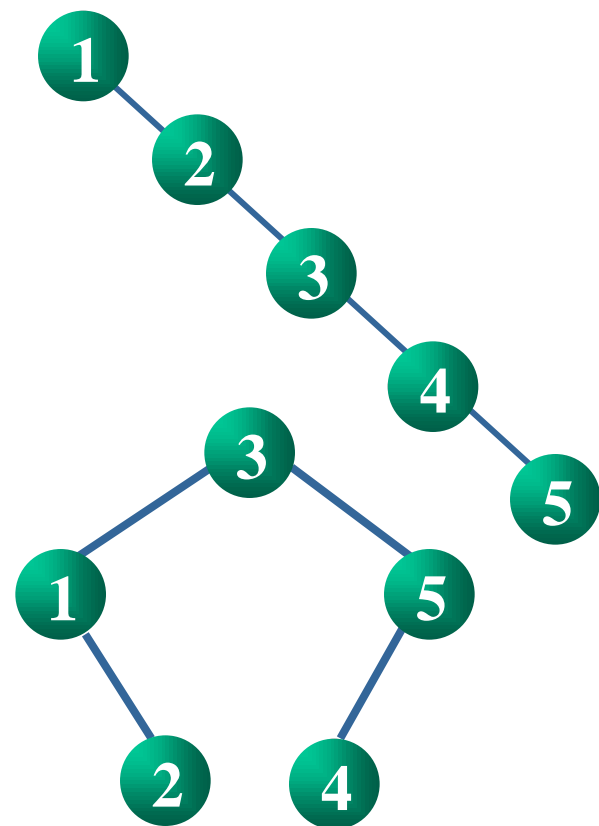
## 二叉排序树的查找性能分析

由序列{1, 2, 3, 4, 5}得到二叉排序树:

$$ASL = (1+2+3+4+5) / 5 = 3$$

由序列{3, 1, 2, 5, 4}得到二叉排序树:

$$ASL = (1+2+3+2+3) / 5 = 2.2$$



二叉排序树的查找性能取决于二叉排序树的形状，  
在 $O(\log_2 n)$ 和 $O(n)$ 之间。

## 平衡二叉树 (balance binary tree)

### 平衡二叉树的定义：

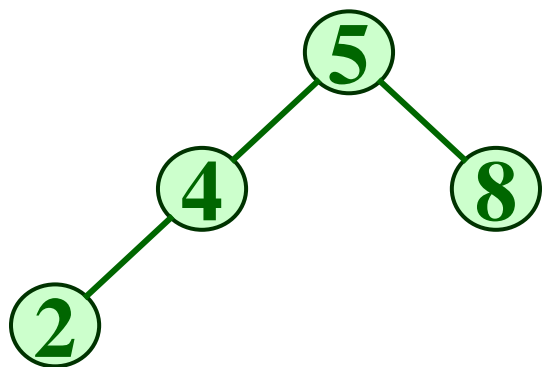
或者是一棵空的二叉排序树，或者是具有下列性质的二叉排序树：

- (1) 根结点的左子树和右子树的深度最多相差1；
- (2) 根结点的左子树和右子树也都是平衡二叉树。

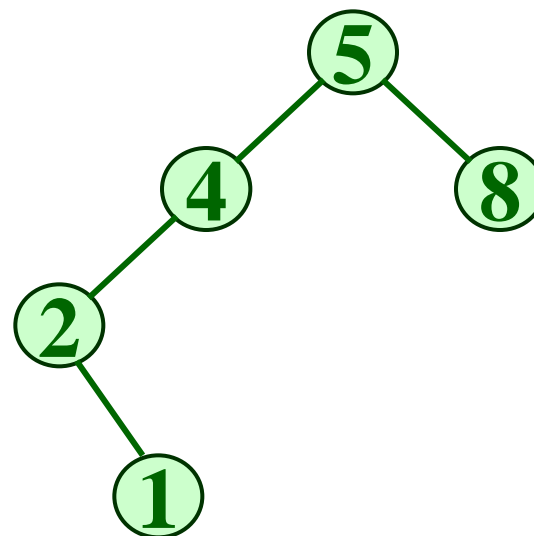
### 1、平衡因子(balance factor):

结点的平衡因子是该结点的左子树的深度与右子树的深度之差。

结点的平衡因子 =  $H_L - H_R$



是平衡树

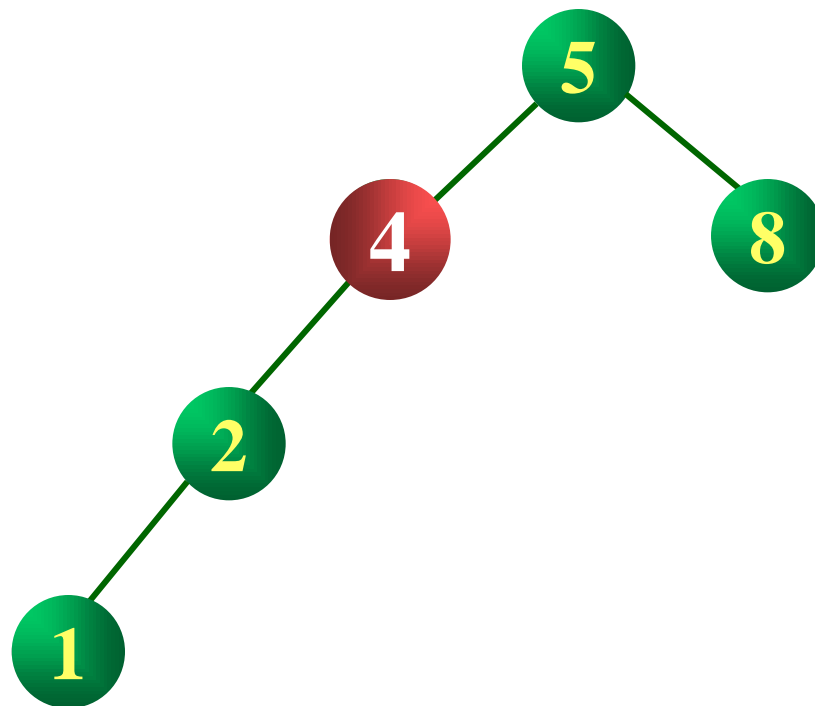


非平衡树

在平衡树中，结点的平衡因子可以是1，0，-1。

## 2、最小不平衡子树：

在平衡二叉树的构造过程中，以距离插入结点最近的、且平衡因子的绝对值大于1的结点为根的子树。



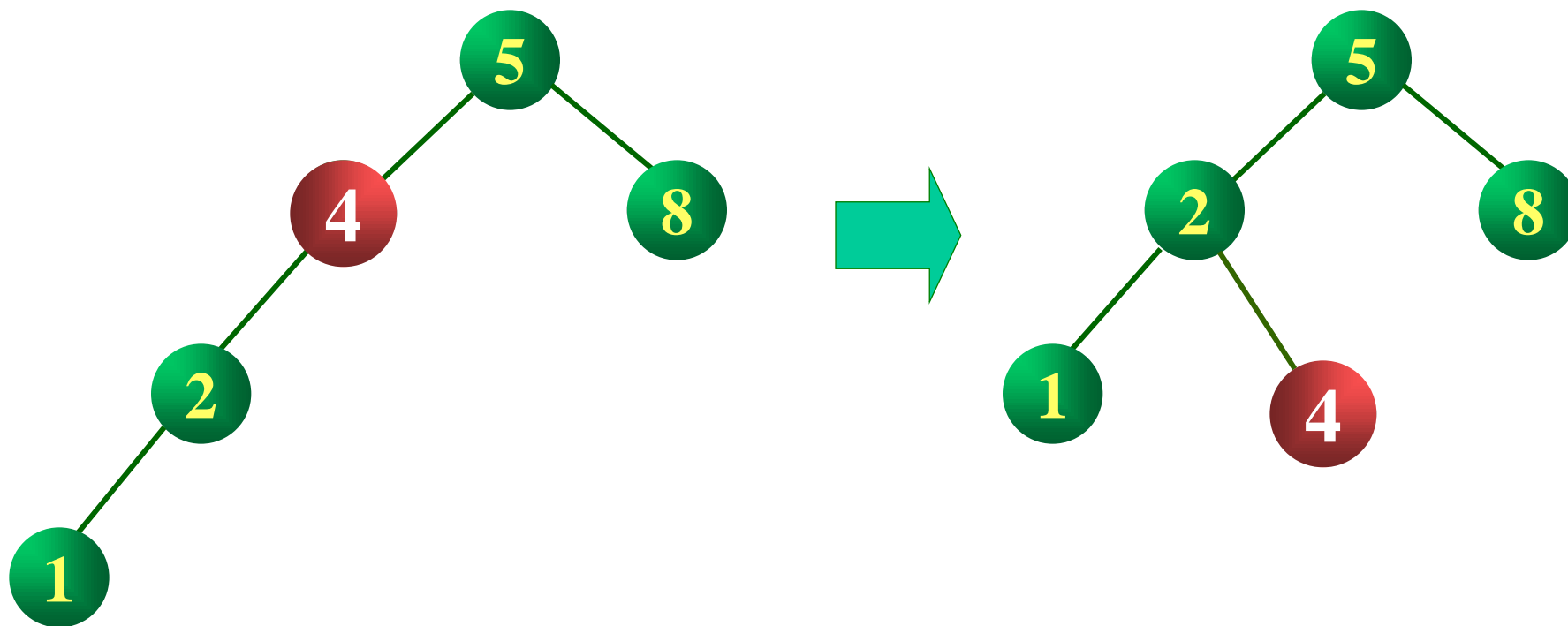


# 平衡二叉树的构造过程

## 基本思想：

在构造二叉排序树的过程中，每插入一个结点时，首先检查是否因插入而破坏了树的平衡性，若是，则找出**最小不平衡子树**，**在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。**

例如：



一般情况下，设结点A为最小不平衡子树的根结点，对该子树进行平衡化调整归纳起来有四种情况：

LL型、RR型、LR型、RL型。

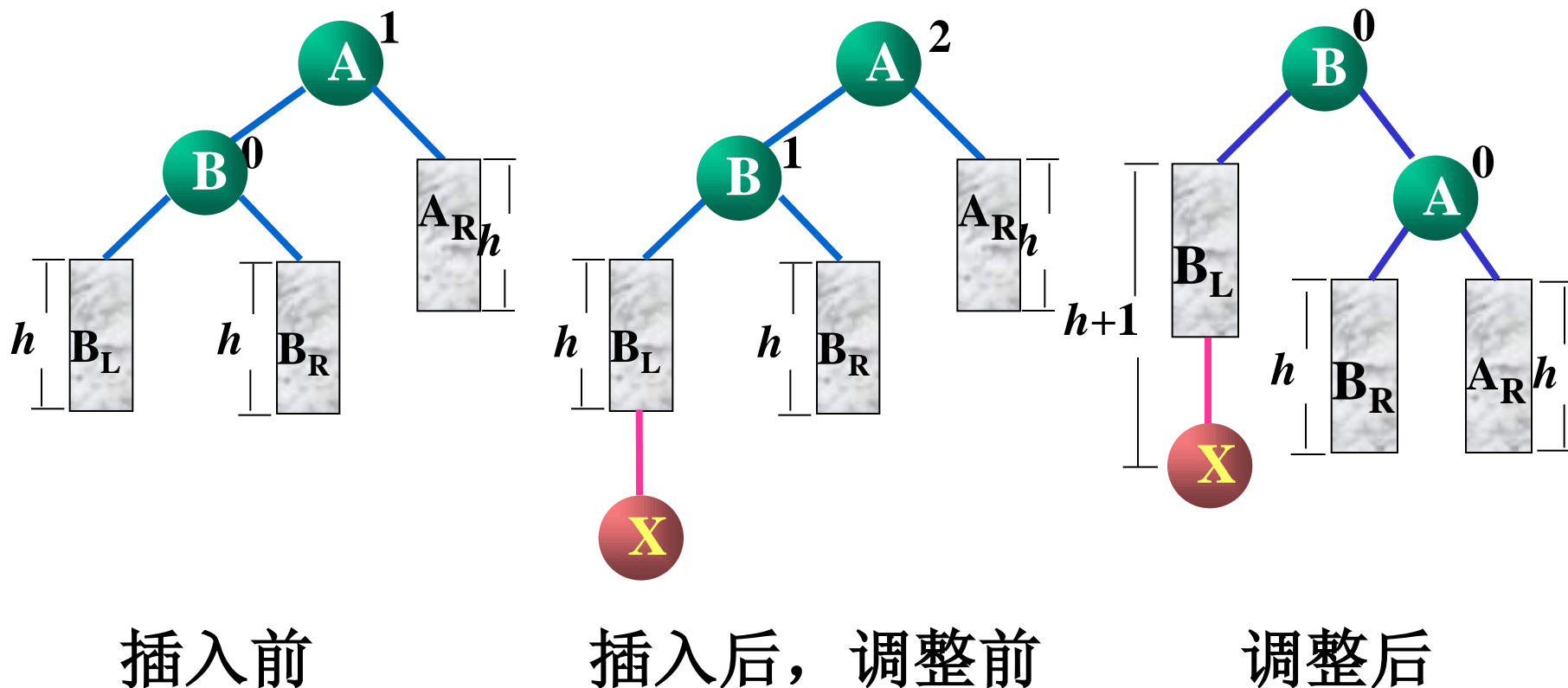
**LL型：** 新插入的结点是插在结点**A**的左孩子的左子树上

**RR型：** 新插入的结点是插在结点**A**的右孩子的右子树上

**LR型：** 新插入的结点是插在结点**A**的左孩子的右子树上

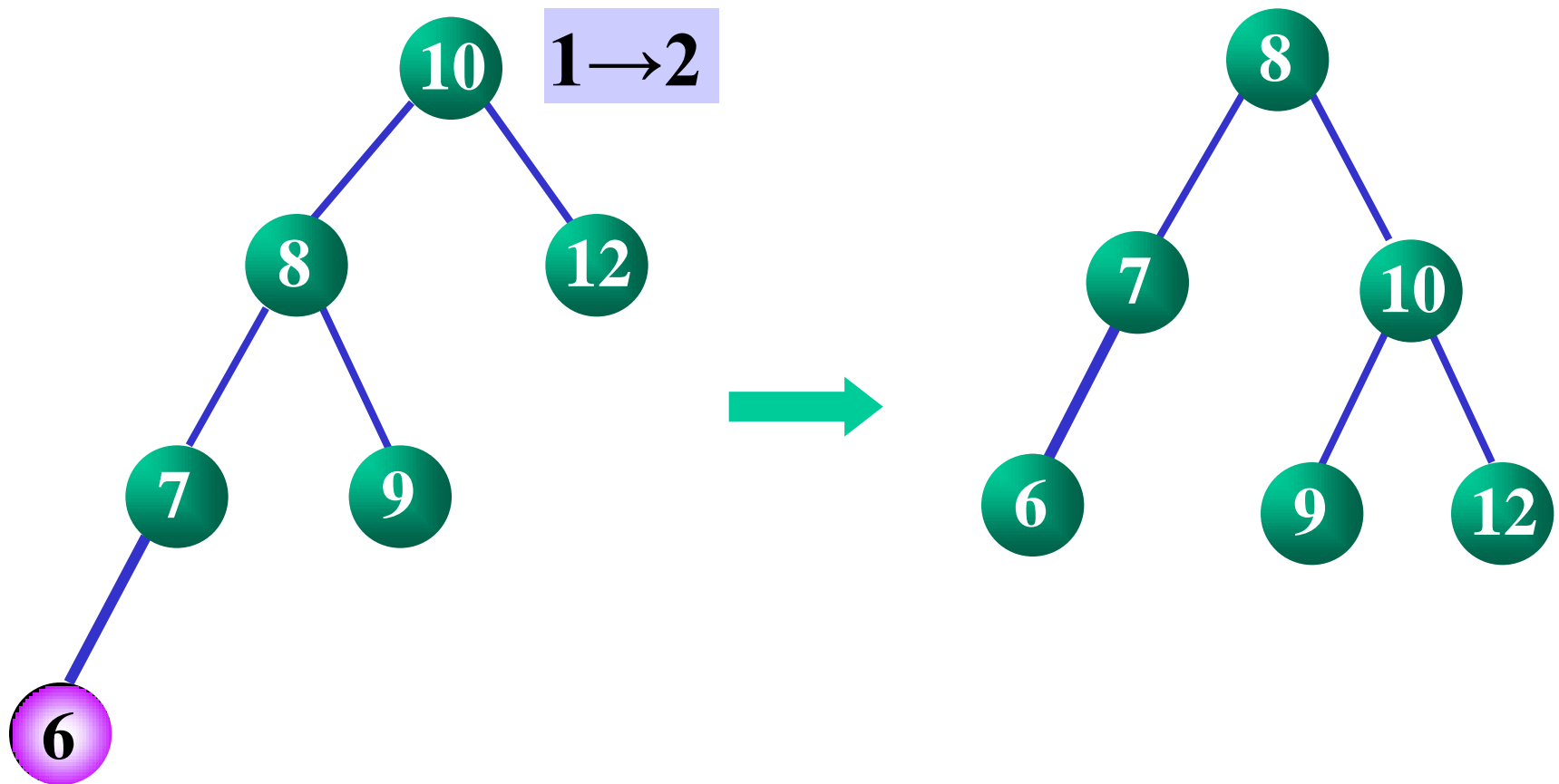
**RL型：** 新插入的结点是插在结点**A**的右孩子的左子树上

# 平衡二叉树——LL型

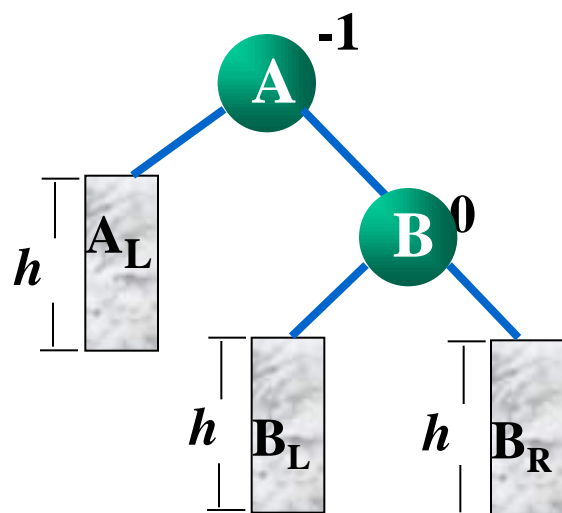


旋转：扁担原理；冲突：旋转优先

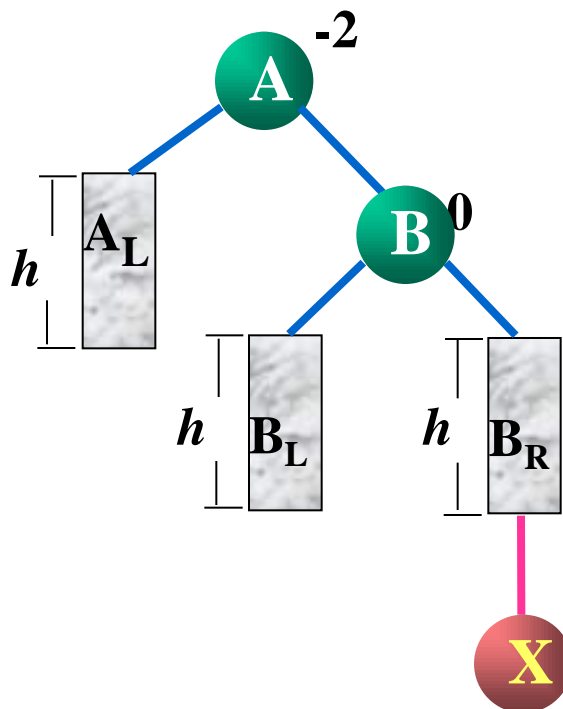
例:LL型



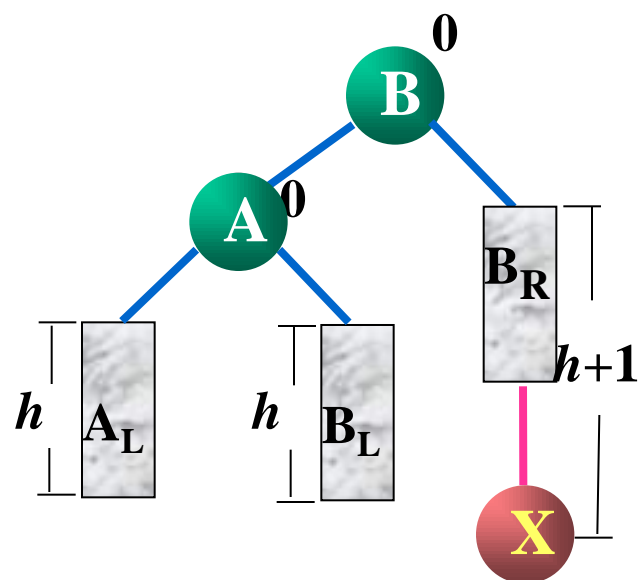
## 平衡二叉树——RR型



插入前

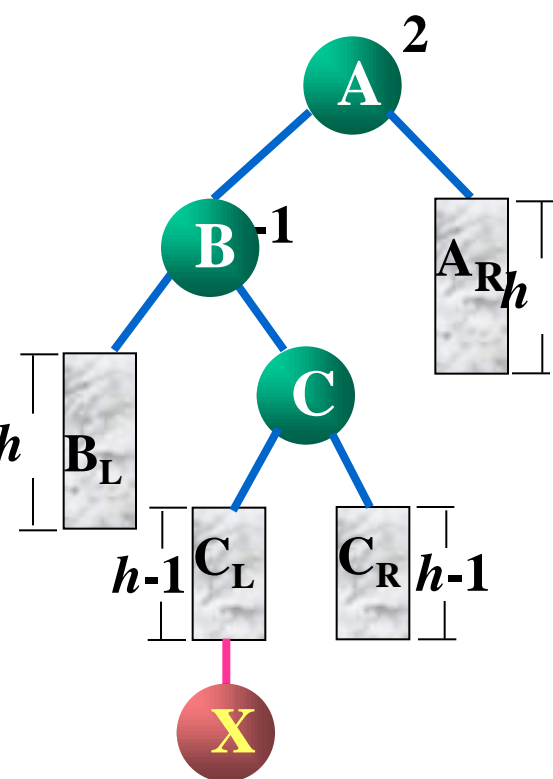


插入后，调整前

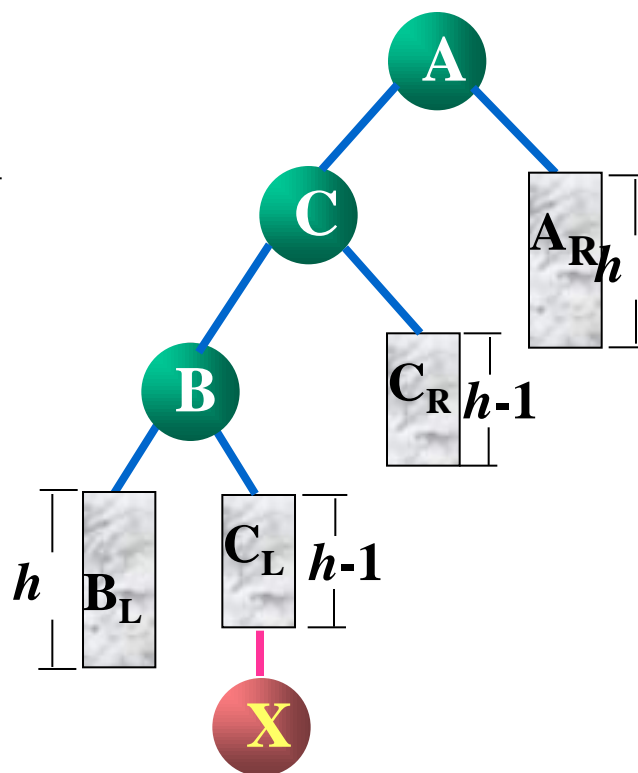


调整后

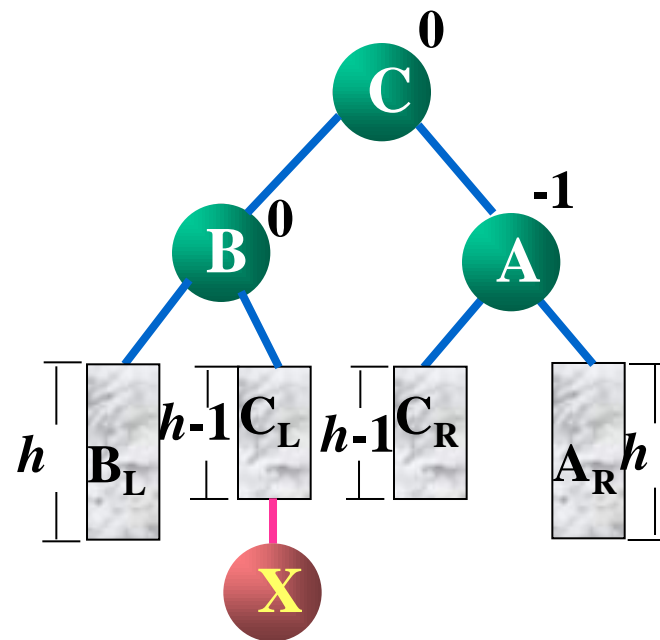
## 平衡二叉树——LR型



插入后，调整前

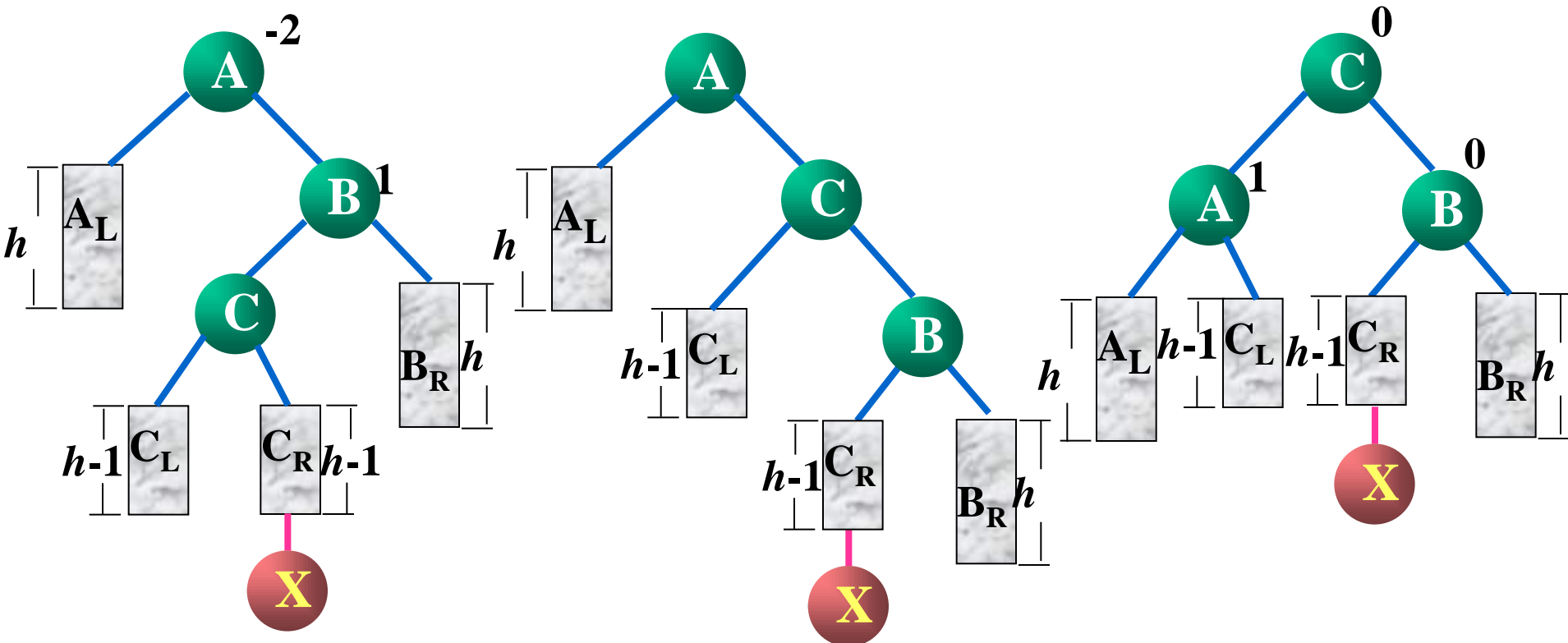


先逆时针旋转



再逆顺时针旋转

## 平衡二叉树——RL型



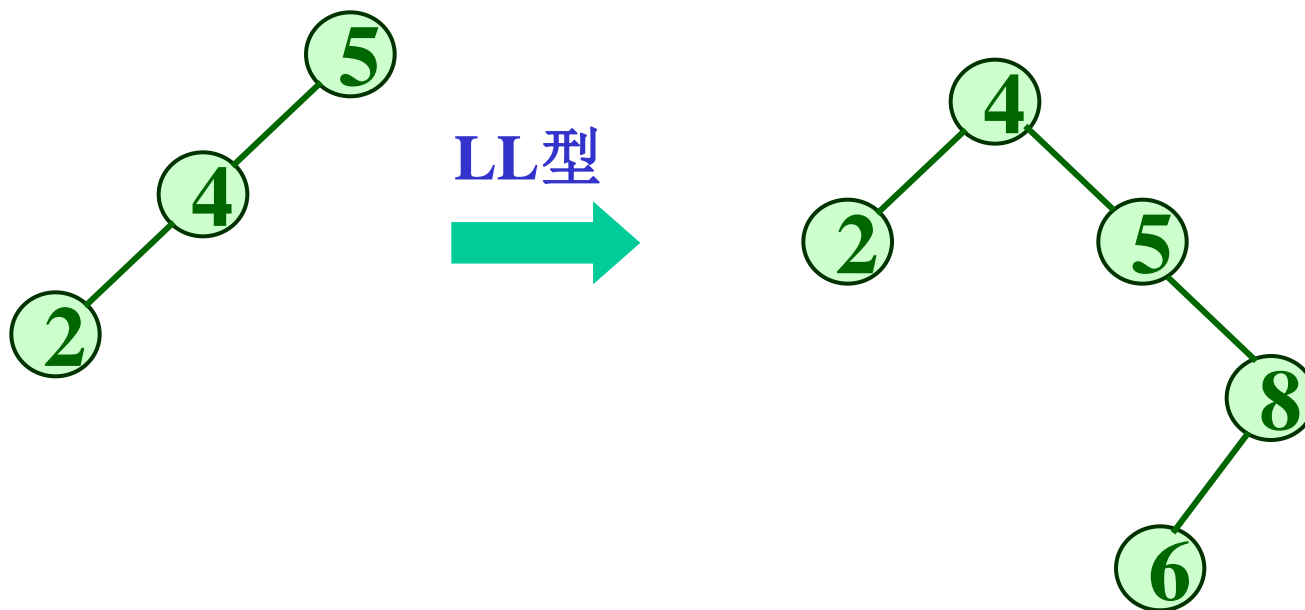
插入后，调整前

先顺时针旋转

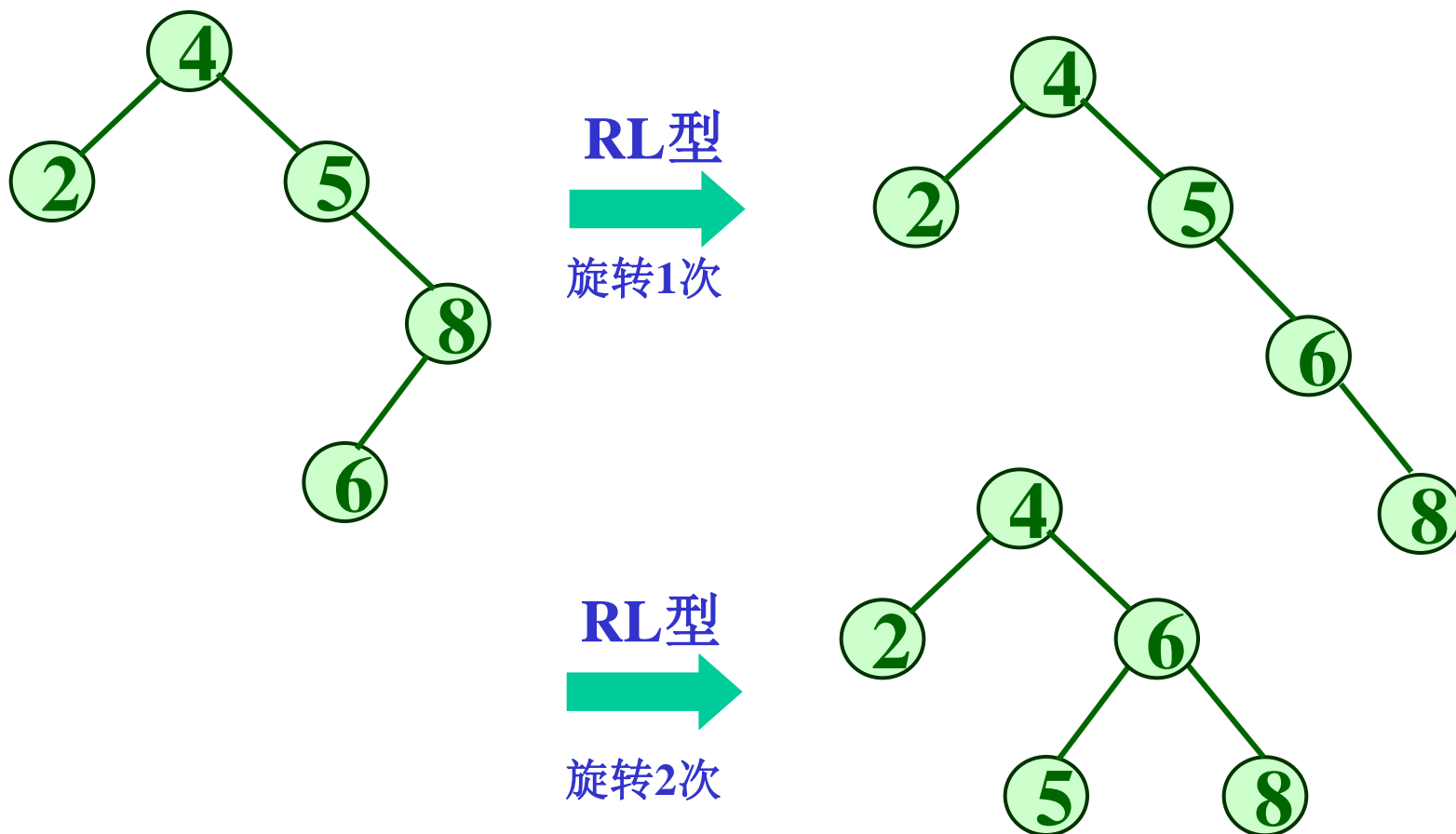
再逆时针旋转



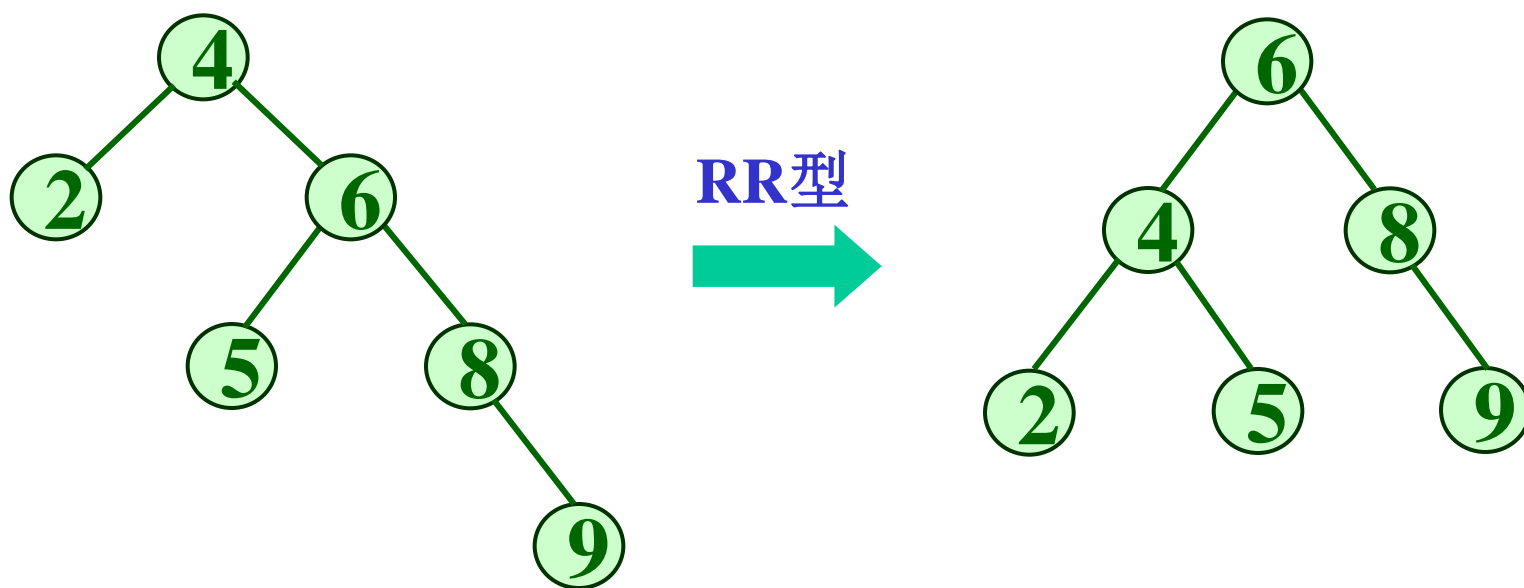
课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



课堂练习：设有关键码序列{5, 4, 2, 8, 6, 9}，构造平衡树



## 7.4 散列表及其查找

① 查找操作要完成什么任务？

待查值 $k$   $\longrightarrow$  确定 $k$ 在存储结构中的位置

② 我们学过哪些查找技术？这些查找技术的共性？

顺序查找、折半查找、二叉排序树查找等。

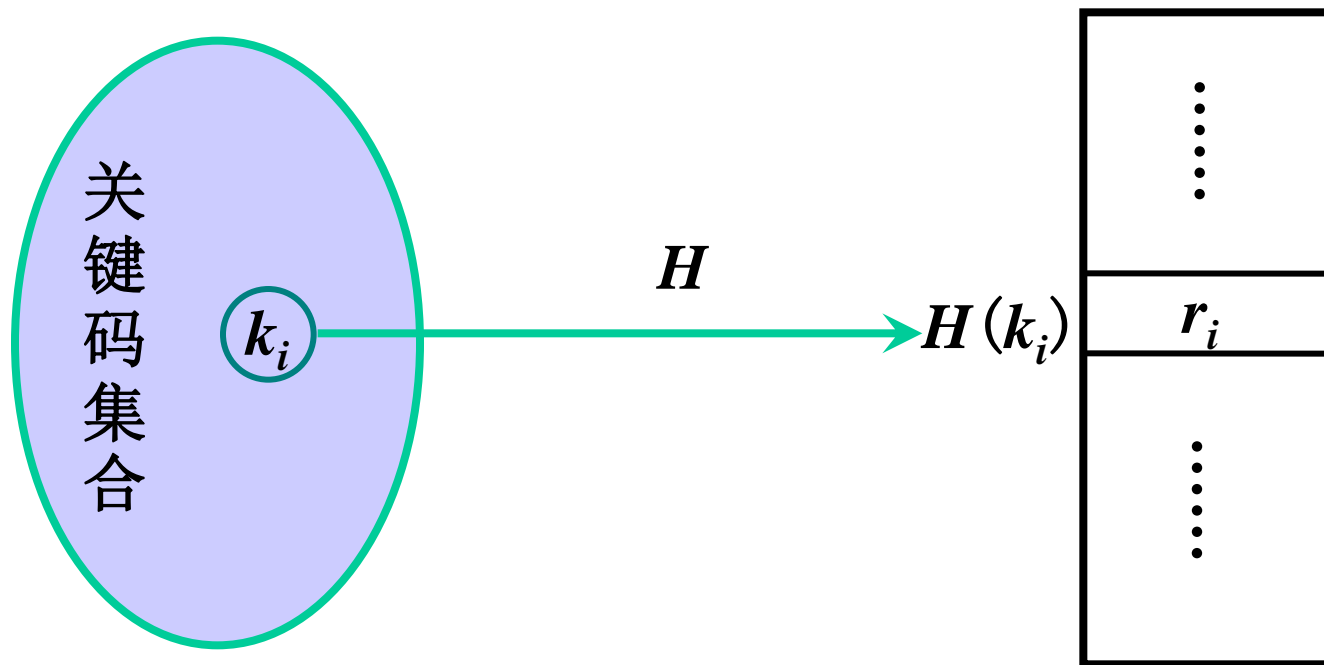
这些查找技术都是通过一系列的给定值与关键码的比较，查找效率依赖于查找过程中进行的给定值与关键码的比较次数。

③ 能否不用比较，通过关键码直接确定存储位置？

在存储位置和关键码之间建立一个确定的对应关系。

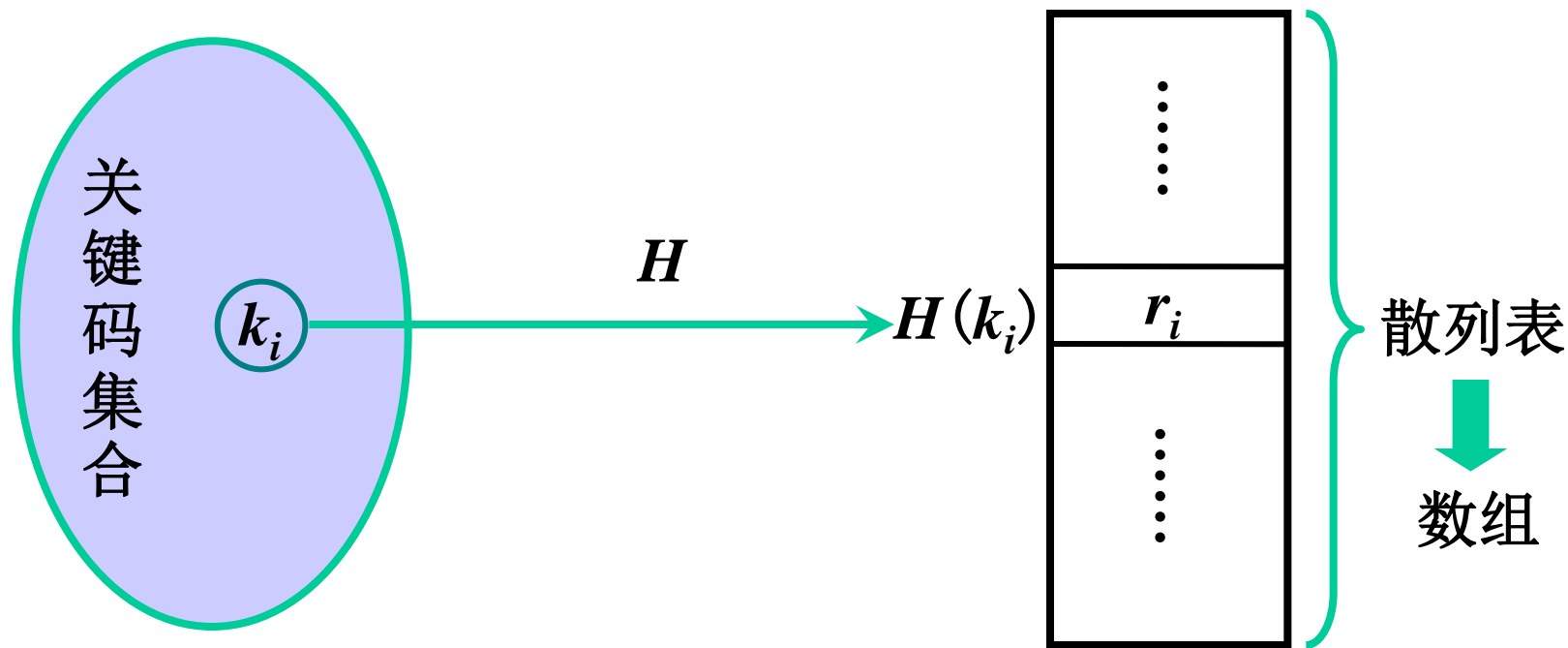
## 散列(hashing)的基本思想

在记录的存储地址和它的关键码之间建立一个确定的对应关系。这样，不经过比较，一次读取就能得到所查元素的查找方法。



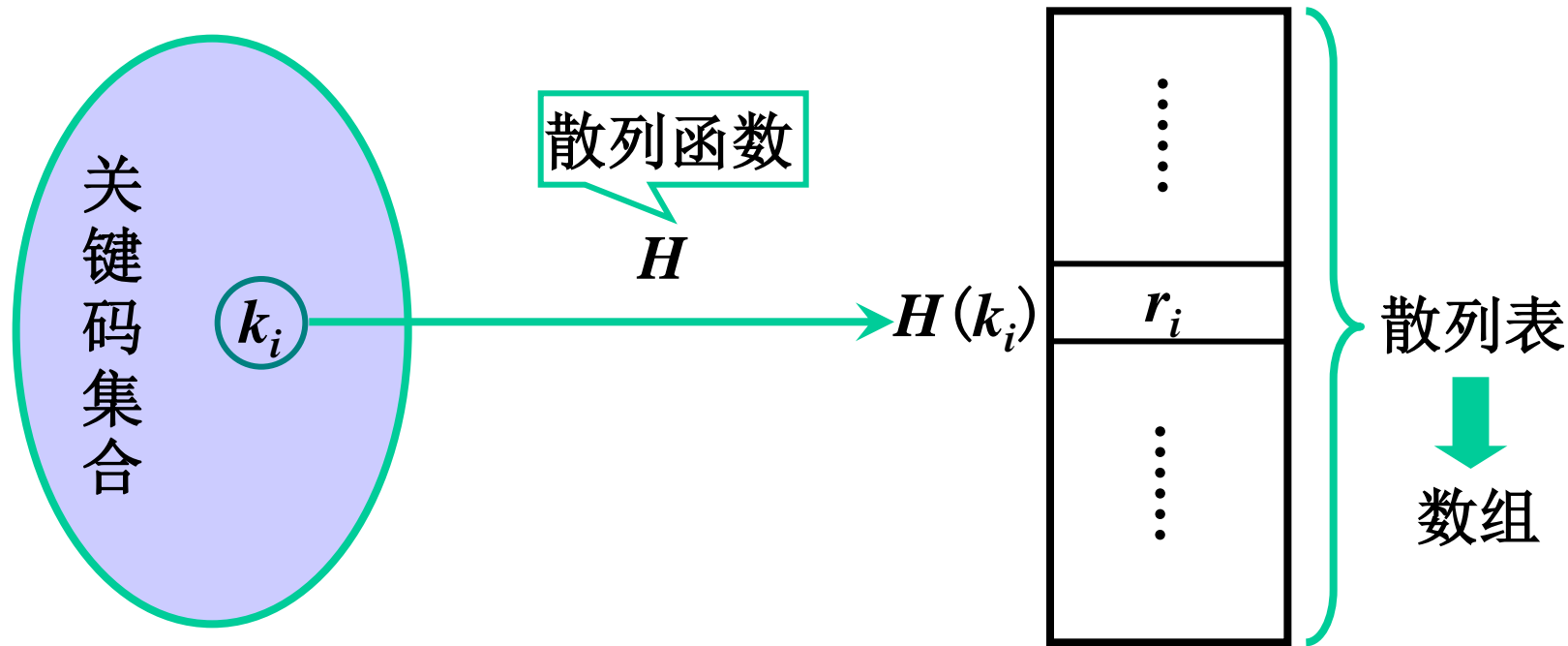
# 散列表的定义

采用散列技术将记录存储在一块连续的存储空间中，这块连续的存储空间称为散列表。



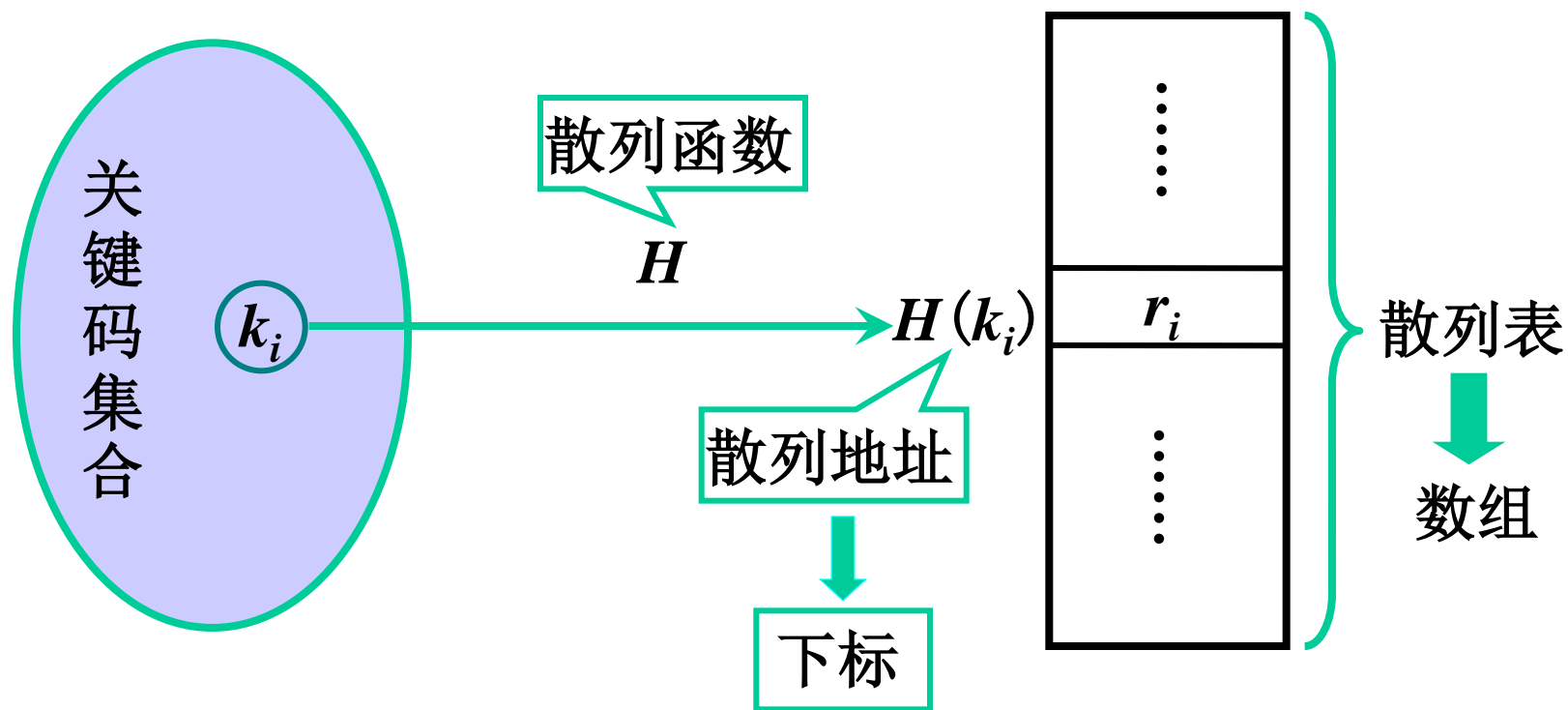
# 散列函数

将关键码映射为散列表中适当存储位置的函数。



# 散列地址

由散列函数所得的存储位置址。





# 思考题

① 散列技术仅仅是一种查找技术吗？

散列既是一种查找技术，也是一种存储技术。

② 散列是一种完整的存储结构吗？

散列只是通过记录的关键码定位该记录，没有完整地表达记录之间的逻辑关系，所以，散列主要是面向查找的存储结构。

## 思考题（续）

① 散列技术适合于哪种类型的查找？

散列技术一般不适用于允许多个记录有同样关键码的情况。散列方法也不适用于范围查找，换言之，在散列表中，我们不可能找到最大或最小关键码的记录，也不可能找到在某一范围内的记录。

散列技术最适合回答的问题是：如果有的话，哪个记录的关键码等于待查值。

# 散列技术的关键问题

(1) **散列函数的设计**。如何设计一个简单、均匀、存储利用率高的散列函数。

(2) **冲突的处理**。如何采取合适的处理冲突方法来解决冲突。这是因为：这种转换是一种压缩映射，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。

# 散列函数的设计问题

- (1) 散列函数不应该有很大的计算量，否则会降低查找效率。
- (2) 函数值要尽量均匀散布在地址空间，这样才能保证存储空间的有效利用并减少冲突。

因此，设计散列函数一般应遵循以下原则：

- (1) 计算简单。
- (2) 函数值即散列地址分布均匀。

## 1、散列函数——直接定址法

散列函数是关键码的线性函数，即：

$$H(key) = a \times key + b \quad (a, b \text{ 为常数})$$

例：关键码集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为 $H(key) = key/10$ ，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90



适用情况？

事先知道关键码，关键码集合不是很大且连续性较好。

## 2、散列函数——除留余数法

散列函数为：

$$H(key) = key \bmod p$$

① 如何选取合适的  $p$ ，产生较少同义词？

若  $p$  选得不好，则容易产生同义词。

例：  $p = 21 = 3 \times 7$

关键码	14	21	28	35	42	49	56
散列地址	14	0	7	14	14	7	14

一般情况下，选 $p$ 为小于或等于表长（最好接近表长）的最小素数。

① 适用情况？

除留余数法是一种最简单、也是最常用的构造散列函数的方法，并且**不要求事先知道关键码的分布**。

### 3、散列函数——数字分析法

根据关键码在各个位上的分布情况，**选取分布比较均匀的若干位组成散列地址。**

例：关键码为8位十进制数，散列地址为2位十进制数

①	②	③	④	⑤	⑥	⑦	⑧
8	1	3	4	6	<u>5</u>	<u>3</u>	2
8	1	3	7	2	<u>2</u>	<u>4</u>	2
8	1	3	8	7	<u>4</u>	<u>2</u>	2
8	1	3	0	1	<u>3</u>	<u>6</u>	7
8	1	3	2	2	<u>8</u>	<u>1</u>	7
8	1	3	3	8	<u>9</u>	<u>6</u>	7



① 适用情况:

适合于事先知道关键码的分布且关键码中有若干位分布较均匀的情况。

## 4、散列函数——平方取中法

对关键码平方后，按散列表大小，取中间的若干位作为散列地址（平方后截取）。

例：散列地址为2位，则关键码1234的散列地址为：

$$(1234)^2 = 1522756$$

② 适用情况:

事先不知道关键码的分布且关键码的位数不是很大。  
有些编译器对标识符的管理就是采用这种方法。

## 5、散列函数——折叠法

将关键码从左到右分割成位数相等的几部分，将这几部分叠加求和，取后几位作为散列地址。

例：设关键码为2 5 3 4 6 3 5 8 7 0 5，散列地址为三位。

$$\begin{array}{r}
 253 \\
 463 \\
 587 \\
 + 05 \\
 \hline
 1308
 \end{array}$$

移位叠加

$$\begin{array}{r}
 253 \\
 364 \\
 587 \\
 + 50 \\
 \hline
 1254
 \end{array}$$

间界叠加

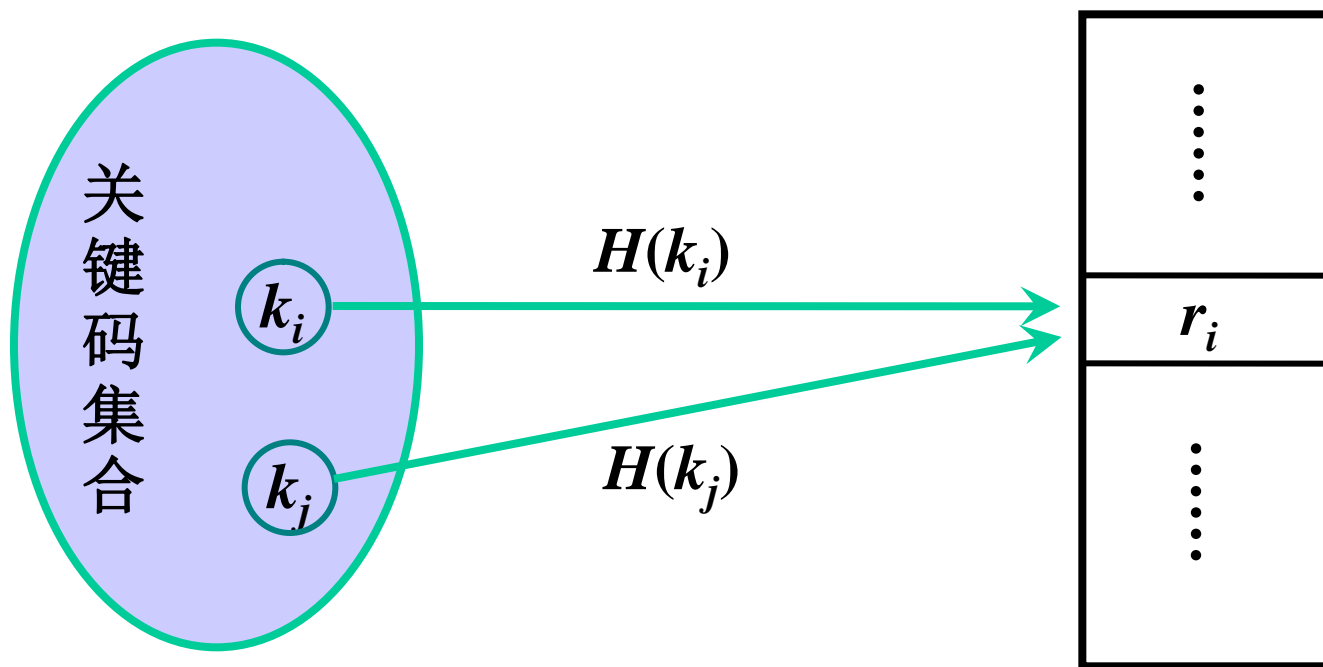


适用情况:

关键码位数很多，事先不知道关键码的分布。

# 冲突问题

**冲突(collision):** 对于两个不同关键码 $k_i \neq k_j$ , 有 $H(k_i) = H(k_j)$ , 即两个不同的记录需要存放在**同一个存储位置**,  $k_i$ 和 $k_j$ 相对于 $H$ 称做**同义词(synonym)**。



## 处理冲突的方法——

### 1、开放定址法 (open addressing)

**基本思想：**由关键码得到的散列地址，一旦产生了冲突，就去寻找下一个空的散列地址，并将记录存入。

用开放定址法处理冲突得到的散列表叫**闭散列表**。

① 如何寻找下一个空的散列地址？

- (1) 线性探测法
- (2) 二次探测法
- (3) 随机探测法

## (1)线性探测法

当发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。

对于键值 $key$ ，设 $H(key)=d$ ，闭散列表的长度为 $m$ ，则发生冲突时，寻找下一个散列地址的公式为：

$$H_i = (H(key) + d_i) \% m \quad (d_i = 1, 2, \dots, m-1)$$

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为 $H(key) = key \bmod 11$ ，用线性探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22		47	92	16	3	7	29	8
22			3	3	3		29	8	

**堆积现象：**在处理冲突的过程中出现的非同义词之间对同一个散列地址争夺的现象。

## 基于线性探测的散列查找算法——伪代码描述

1. 计算散列地址 $j$ ;
2. 若 $ht[j]=k$ ，则查找成功，返回记录在散列表中的下标；否则
3. 若 $ht[j]$ 为空或将散列表探测一遍，则查找失败，转4；否则， $j$ 指向下一单元，转2；
4. 若整个散列表探测一遍，则表满，抛出溢出异常；否则，将待查值插入。



## 基于线性探测的散列查找算法——C++描述

```
int HashSearch1 (int ht[ ], int m, int k)
{
    j=H(k);
    if (ht[j]==k) return j; //没有发生冲突，比较一次查找成功
    i=(j+1) % m;
    while (ht[i]!=Empty && i!=j)
    {
        if (ht[i]==k) return i; //发生冲突，比较若干次查找成功
        i=(i+1) % m; //向后探测一个位置
    }
    if (i==j) throw "溢出";
    else ht[i]=k; //查找不成功时插入
}
```

## (2)二次探测法

当发生冲突时，寻找下一个散列地址的公式为：

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

$$(d_i = 1^2, -1^2, 2^2, -2^2, \dots, q^2, -q^2 \text{ 且 } q \leq m/2)$$

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为  $H(key) = key \bmod 11$ ，用二次探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22	3	47	92	16		7	29	8
22			3	3			29	8	

### (3)随机探测法

当发生冲突时，下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

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

( $d_i$ 是一个随机数列， $i=1, 2, \dots, m-1$ )

## 处理冲突的方法——

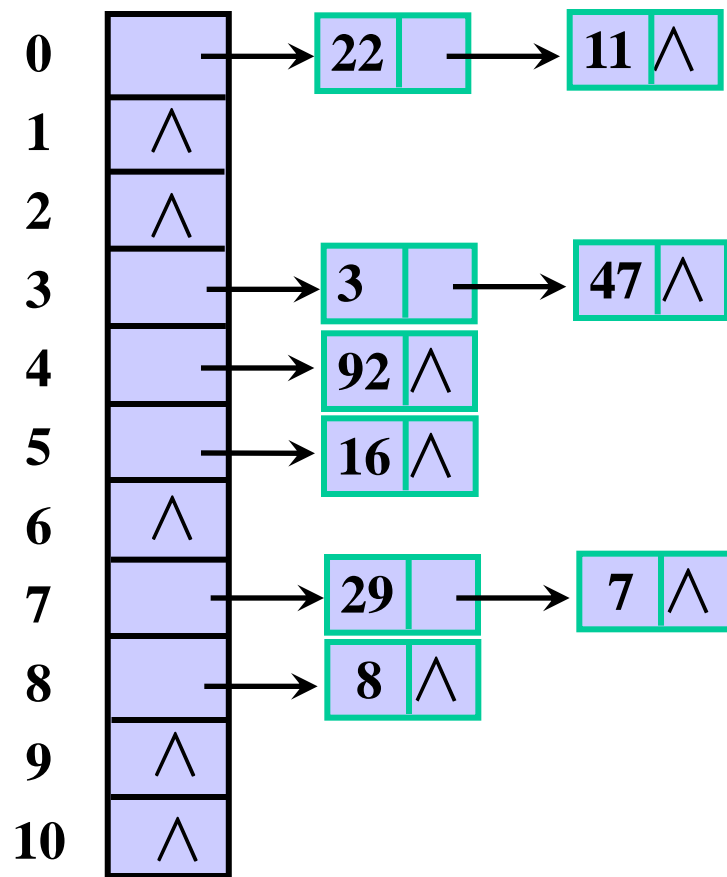
### 2、拉链法（链地址法）(chaining)

**基本思想：**将所有散列地址相同的记录，即所有同义词的记录存储在一个单链表中（称为同义词子表），在散列表中存储的是所有同义词子表的头指针。

用拉链法处理冲突构造的散列表叫做**开散列表**。

设 $n$ 个记录存储在长度为 $m$ 的散列表中，则同义词子表的平均长度为 $n / m$ 。

例：关键码集合 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为  $H(key) = key \bmod 11$ ，用拉链法处理冲突，构造的开散列表为：



## 基于拉链法的散列查找算法——伪代码描述

1. 计算散列地址 $j$ ;
2. 在第 $j$ 个同义词子表中顺序查找;
3. 若查找成功, 则返回结点的地址; 否则, 将待查记录插在第 $j$ 个同义词子表的表头。

## 基于拉链法的散列查找算法——C++描述

```
Node *HashSearch2 (Node *ht[ ], int m, int k)
{
    j=H(k);
    p=ht[j];
    while (p && p->data!=k)
        p=p->next;
    if (p->data==k) return p;
    else {
        q=new Node; q->data=k;
        q->next= ht[j];
        ht[j]=q;
    }
}
```



## 处理冲突的方法——

### 3、公共溢出区(common overflow area)

**基本思想：**散列表包含**基本表**和**溢出表**两部分（通常溢出表和基本表的大小相同），**将发生冲突的记录存储在溢出表中**。查找时，对给定值通过散列函数计算散列地址，先与**基本表**的相应单元进行比较，若相等，则查找成功；否则，再到**溢出表**中进行顺序查找。

例：关键码集合 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为  $H(key) = key \bmod 11$ ，用公共溢出区法处理冲突，构造的散列表为：

0	11
1	
2	
3	47
4	92
5	16
6	
7	7
8	8
9	
10	

基本表

0	29
1	22
2	3
3	
4	
5	
6	
7	
8	
9	
10	

溢出表

## 散列查找的性能分析

由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。因此，对散列表的查找效率仍用平均查找长度。

例如，对线性探测法中的例子有：

$$ASL = (5 \times 1 + 3 \times 2 + 1 \times 4) / 9 = 15 / 9$$

而对拉链法中的例子有：

$$ASL = (6 \times 1 + 3 \times 2) / 9 = 12 / 9$$

在查找过程中，关键码的比较次数取决于产生冲突的概率。

影响冲突产生的因素有：

- (1) 散列函数是否均匀
- (2) 处理冲突的方法
- (3) 散列表的装载因子：

$\alpha$  = 表中填入的记录数 / 表的长度

## 几种不同处理冲突方法的平均查找长度

处理冲突方法 \ ASL	查找成功时	查找不成功时
线性探测法	$\frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$	$\frac{1}{2} \left( 1 + \frac{1}{1-\alpha^2} \right)$
二次探测法	$-\frac{1}{\alpha} \ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1 + \frac{\alpha}{2}$	$\alpha + e^{-\alpha}$

结论：散列表的平均查找长度是装填因子的函数，而不是查找集合中记录个数的函数。

# 开散列表与闭散列表的比较

	堆积现象	结构开销	插入/删除	查找效率	估计容量
开散列表	不产生	有	效率高	效率高	不需要
闭散列表	产生	没有	效率低	效率低	需要

# 本章作业

习题7 (P217) :

4(2)(3)(4)(5)(6)(7)(8), 5(1)(2)(4)