

# 第9章 查找

---

## 9.0 基本概念

## 9.1 静态查找表

## 9.2 动态查找表

## 9.3 哈希表

# 回顾

## B-树

B-树是一种平衡的**多路查找树**。一棵m阶的B-树，或为空树，或为满足下列特性的m叉树：

①根节点：叶子节点；或**至少有两棵子树**

②除根之外的所有非终端结点：

$$\left\lceil \frac{m}{2} \right\rceil - 1 \leq \text{关键字个数} \leq m-1 \quad \left\lceil \frac{m}{2} \right\rceil \leq \text{子树} \leq m$$

③非终端节点包含： $(n, A_0, K_1, A_1, K_2, A_2, \dots, K_n, A_n)$

$K_i$ 为关键字，且  $K_{i-1} < K_i$  ( $i=1, \dots, n$ )

④所有的叶子结点都出现在同一层次上，并且不带信息

# 回顾

## B-树

#define M 3 //B-树的阶，根据实际情况需要设定

typedef struct BTNode{

int keynum; //结点中的关键字个数

struct BTNode \*parent; //指向双亲结点

KeyType key[M+1]; //关键字向量，0号单元未用

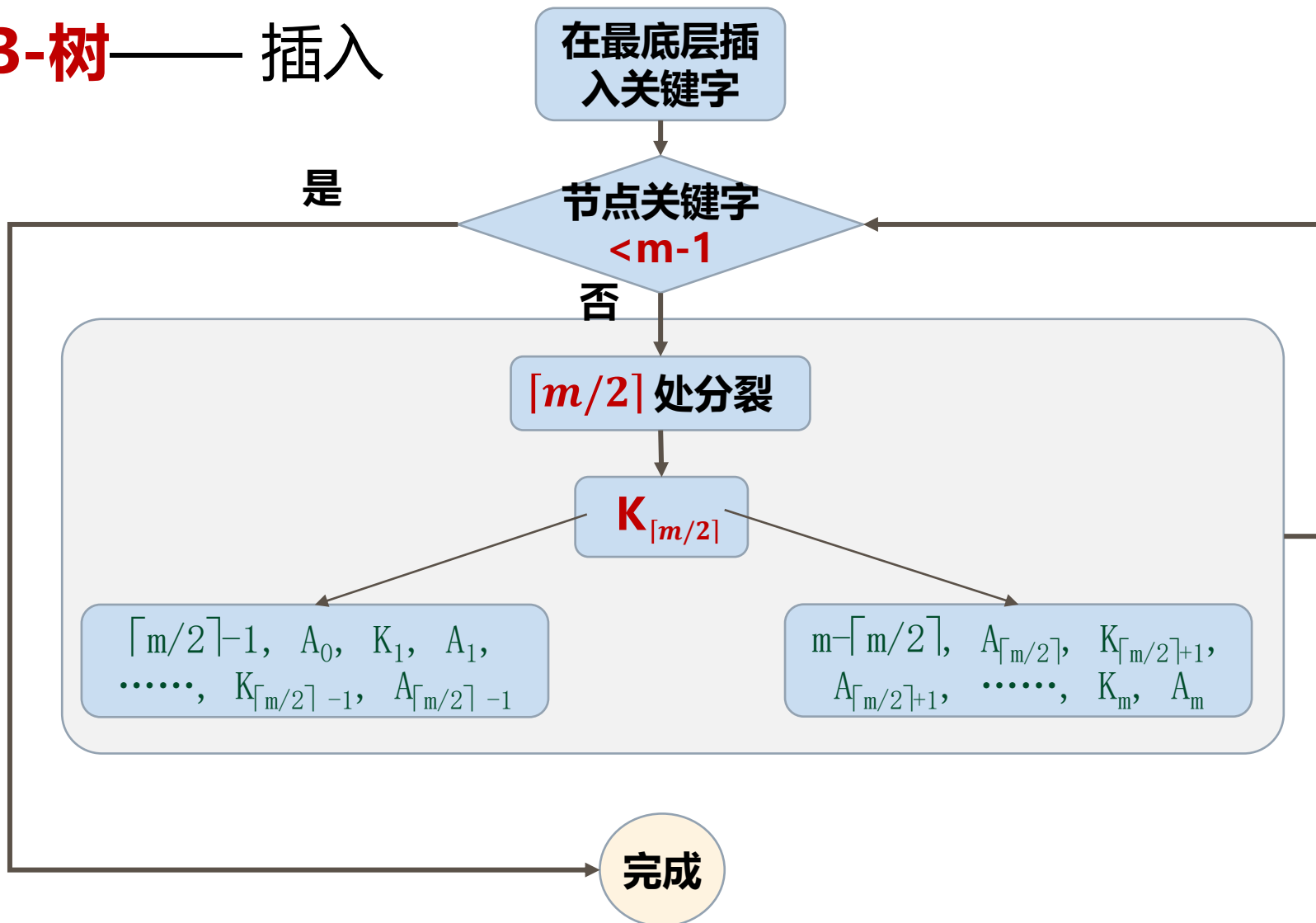
struct BTNode \*ptr[M+1] //子树指针向量

Record \*recptr[M+1]; /\*记录指针向量，0号单元未用\*/

}BTNode, \*Btree;

# 回顾

## B-树——插入



# 回顾

## B-树——删除

①若所删关键字为非终端结点中的 $K_i$ ，则以指针 $A_i$ 所指子树中的最小关键字 $Y$ 替代 $K_i$ ，然后在相应的结点中删去 $Y$ 。

②若所删关键字在**最下层结点**中。有3种情况：

(1)简单删除：删除前该结点中关键字个数  $n \geq \lceil m/2 \rceil$

(2)结点联合调整：若被删关键字所在结点删除前关键字个数  $n = \lceil m/2 \rceil - 1$ ，且该结点相邻的右兄弟（或左兄弟）结点的关键字个数  $n \geq \lceil m/2 \rceil$ ，**向兄弟借**（先右后左：兄弟结点上移一个关键字给双亲，双亲下移一个相邻关键字到当前结点）

(3)结点合并：被删除关键字所在结点和其相邻的兄弟结点中的关键字个数均等于  $\lceil m/2 \rceil - 1$ ，从父结点中取出介于被删除结点和兄弟结点之间的关键字，连同被删结点剩余部分**一起合并到兄弟结点中**

③自底向上操作。

## 9.2.3 B-树和B+树

### 1、B+树定义

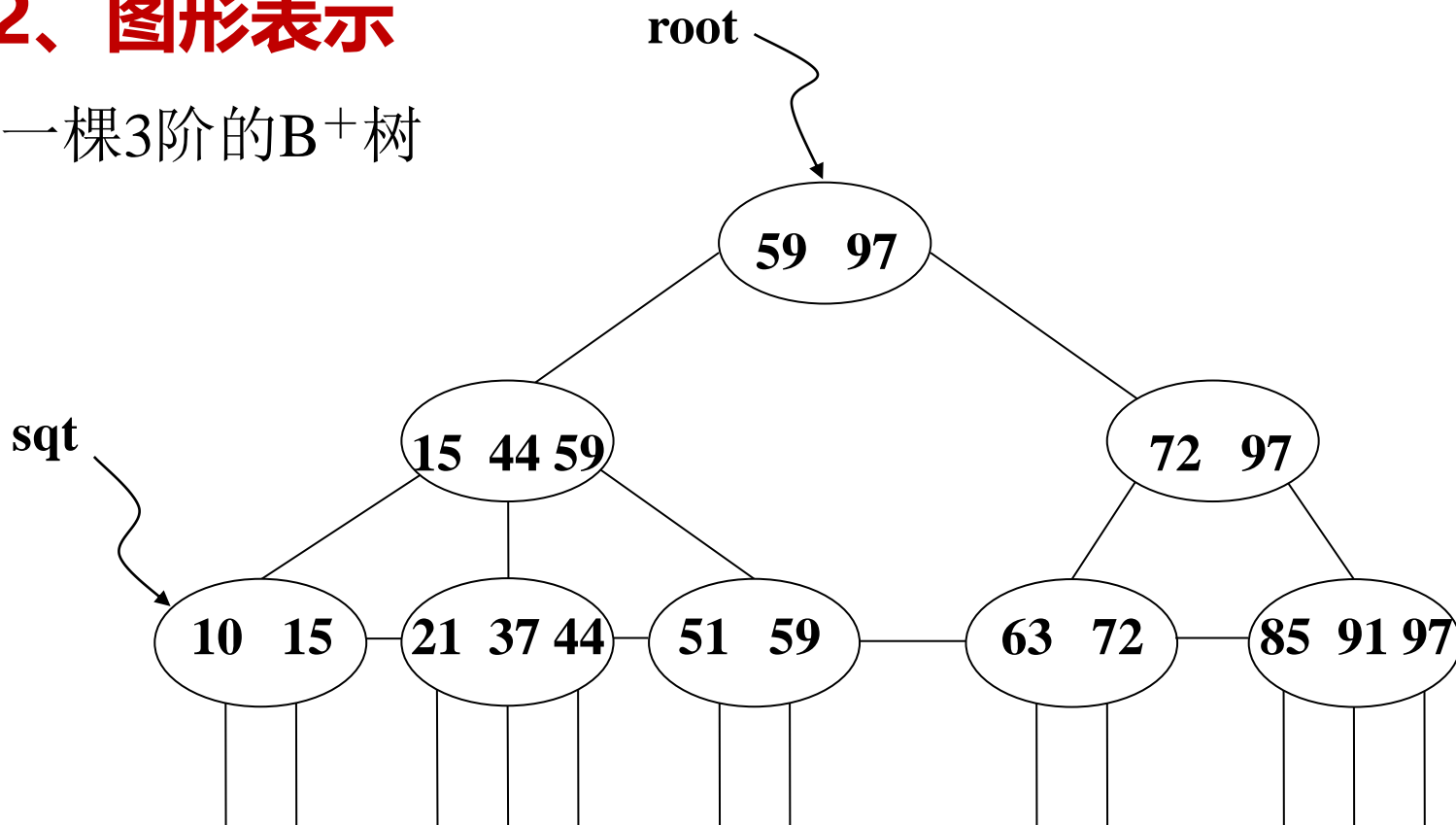
B<sup>+</sup>树是应文件系统所需而出的一种B-树的变型树。一棵m阶的B<sup>+</sup>树和m阶的B-树的差异在于：

- (1) 每个结点最多m棵子树，每个结点中最多含有m个关键字；
- (2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小到大顺序链接。
- (3) 所有的非终端结点可以看成是索引部分，结点仅含其子树中的最大（或最小）关键字。

## 9.2.3 B-树和B+树

### 2、图形表示

一棵3阶的B<sup>+</sup>树



通常在B<sup>+</sup>树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

## 9.2.3 B-树和B+树

### 3、B<sup>+</sup>树的查找

对B<sup>+</sup>树可以进行两种查找运算：

- ①从最小关键字起顺序查找；
- ②从根结点开始，进行随机查找。

在查找时，若非终端结点上的索引值等于给定值，并不终止，而是继续向下直到叶子结点。因此，在B<sup>+</sup>树中，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。其余同B-树的查找类似。



## 9.2.3 B-树和B+树

### 4、B<sup>+</sup>树的插入

B<sup>+</sup>树的插入仅在叶子结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为  $\left\lceil \frac{m+1}{2} \right\rceil$  和  $\left\lfloor \frac{m}{2} \right\rfloor$ 。并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。其余同B-树的插入类似。

## 9.2.3 B-树和B+树

### 4、B<sup>+</sup>树的删除

B<sup>+</sup>树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于 $\lceil \frac{m}{2} \rceil$  时，其和兄弟结点的合并过程亦和B-树类似。

## 9.4 哈希查找表

---

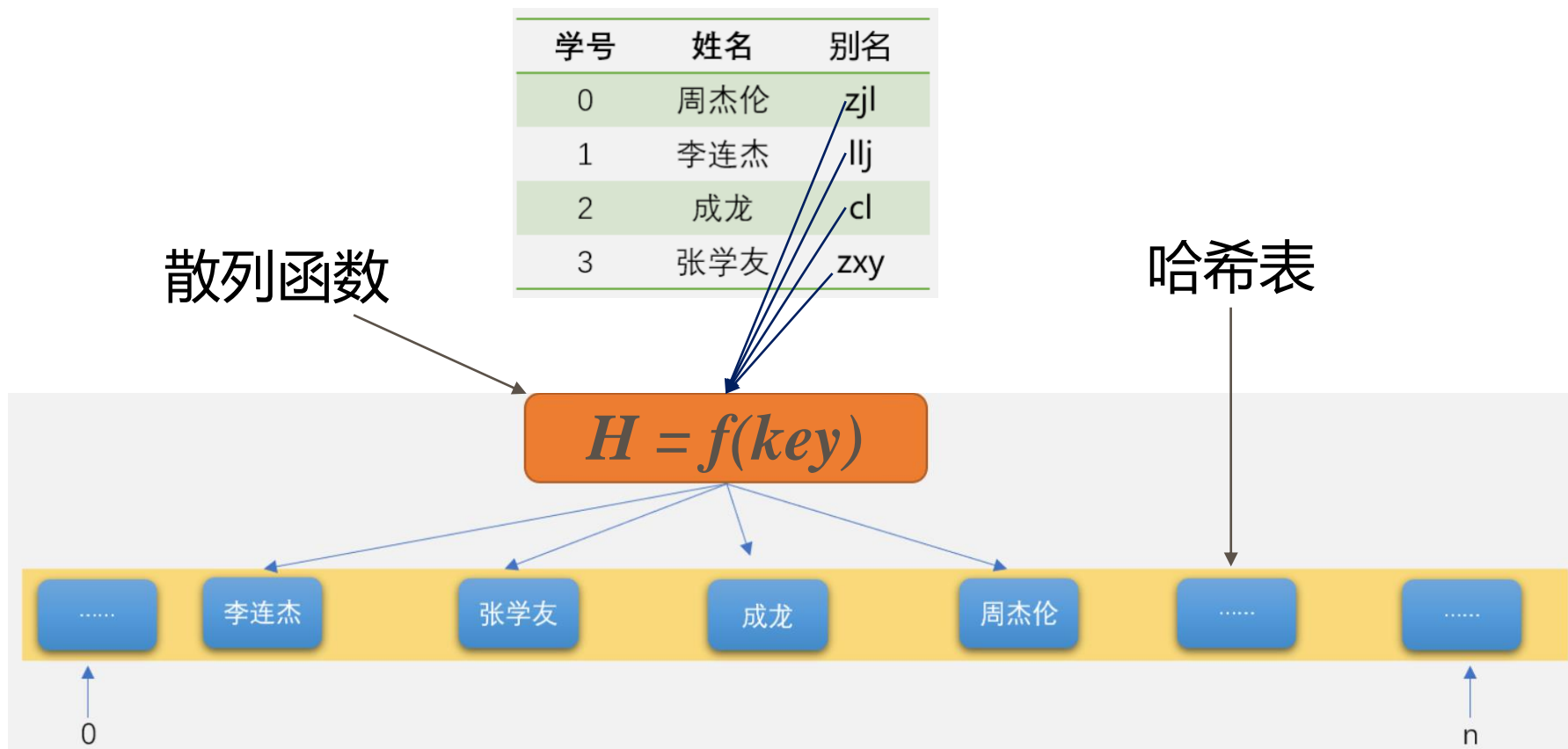
- 一、哈希表的概念
- 二、哈希函数的构造方法
- 三、冲突处理方法
- 四、哈希表的查找及分析

## 9.4.1 哈希表的概念



学生的学号和列表的索引号进行关联，查询某一个学生时，知道了学生的学号也就知道了学生数据存储在列表中的位置，可以认为查询的时间复杂度为  $O(1)$

## 9.4.1 哈希表的概念



优点：查找速度极快（ $O(1)$ ），查找效率与元素个数 $n$ 无关！

## 9.4.1 哈希表的概念

例1：若将学生信息按如下方式存入计算机，如：

将2001011810201的所有信息存入V[01]单元；

将2001011810202的所有信息存入V[02]单元；

.....

将2001011810231的所有信息存入V[31]单元。

欲查找学号为2001011810216的信息，便可直接访问  
V[16]！

## 9.4.1 哈希表的概念

### 例2：

有数据元素序列(14, 23, 39, 9, 25, 11), 若规定每个元素 $k$ 的存储地址 $H(k)=k$ , 请画出存储结构图。

$H(k)$ 称为散列函数

解：根据散列函数 $H(k)=k$ ，可知元素14应当存入地址为14的单元，元素23应当存入地址为23的单元，……，  
对应散列存储表（哈希表）如下：

地址	...	9	...	11	...	14	...	23	24	25	...	39	...
内容		9		11		14		23		25		39	

## 9.4.1 哈希表的概念

讨论：如何进行散列查找？

根据存储时用到的散列函数 $H(k)$ 表达式，迅即可查到结果！

例如，查找 $key=9$ ，则访问 $H(9)=9$ 号地址，若内容为9则成功；

若查不到，应当设法返回一个特殊值，例如空指针或空记录。

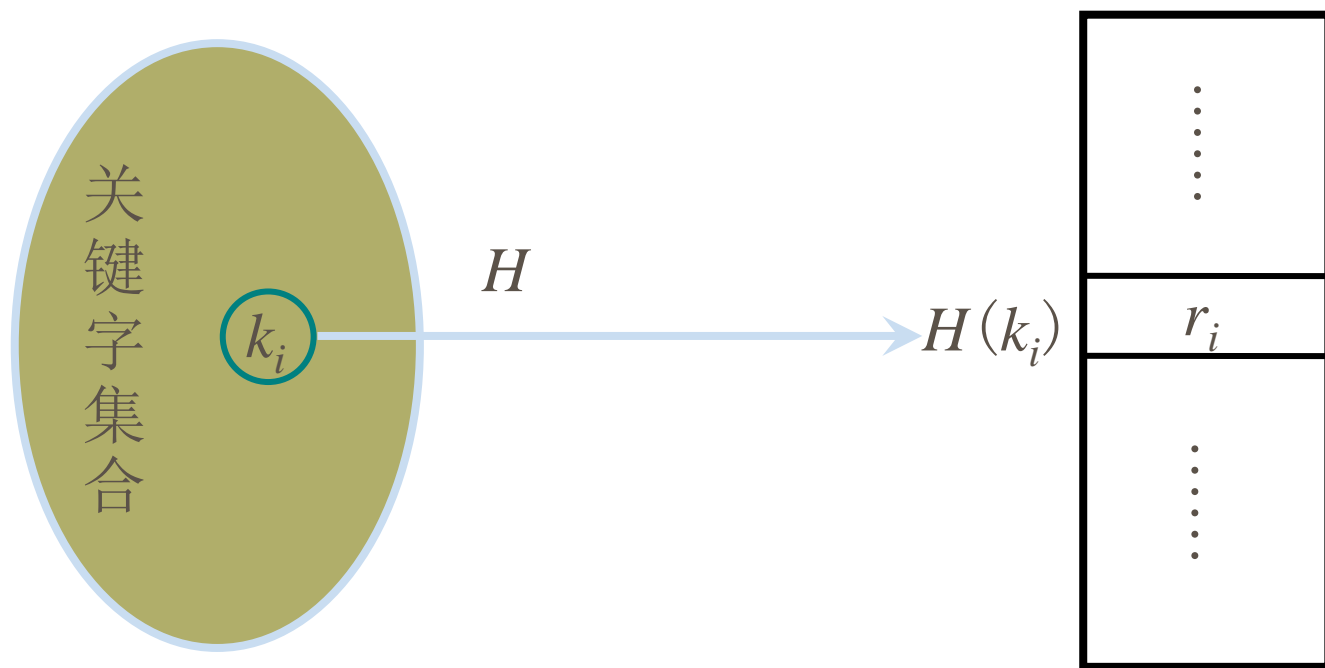
明显缺点：空间效率低

如何解决？



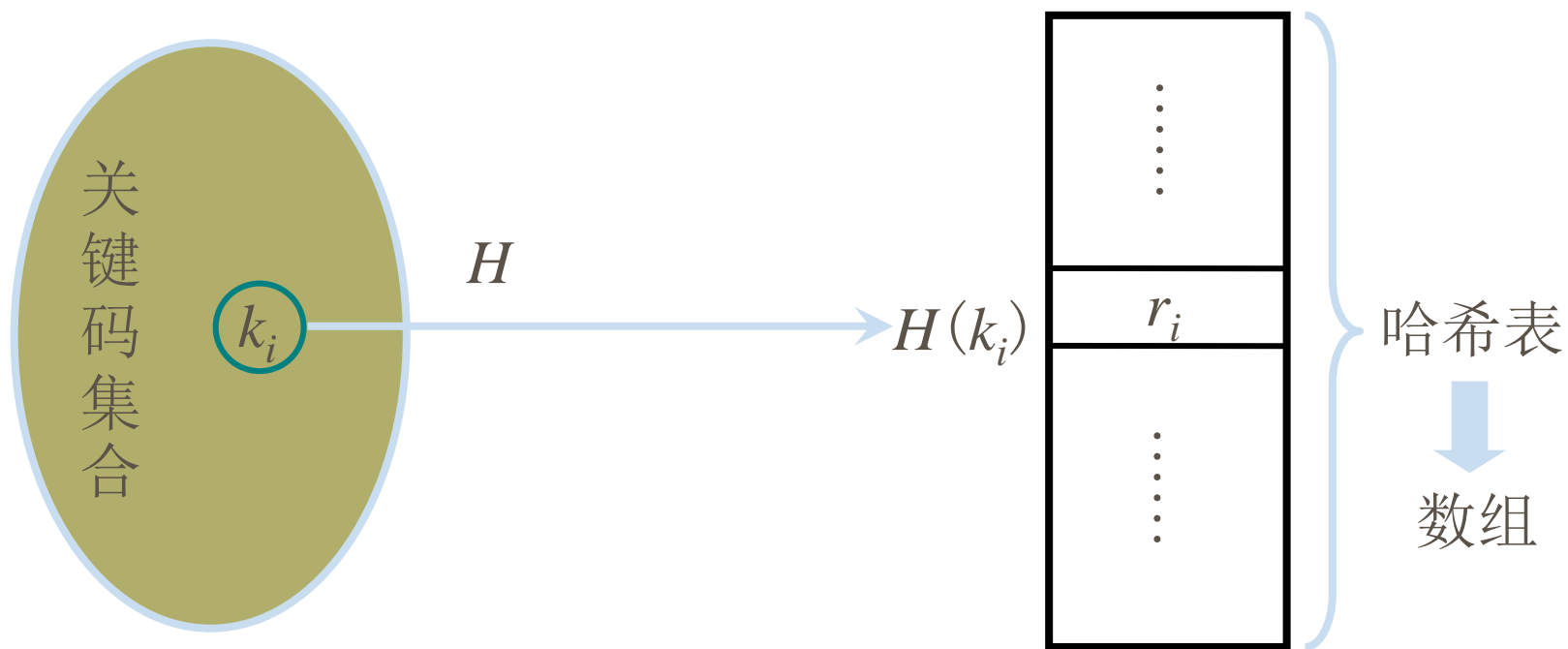
## 9.4.1 哈希表的概念

**哈希的基本思想：**在记录的存储地址和记录的关键字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的记录。



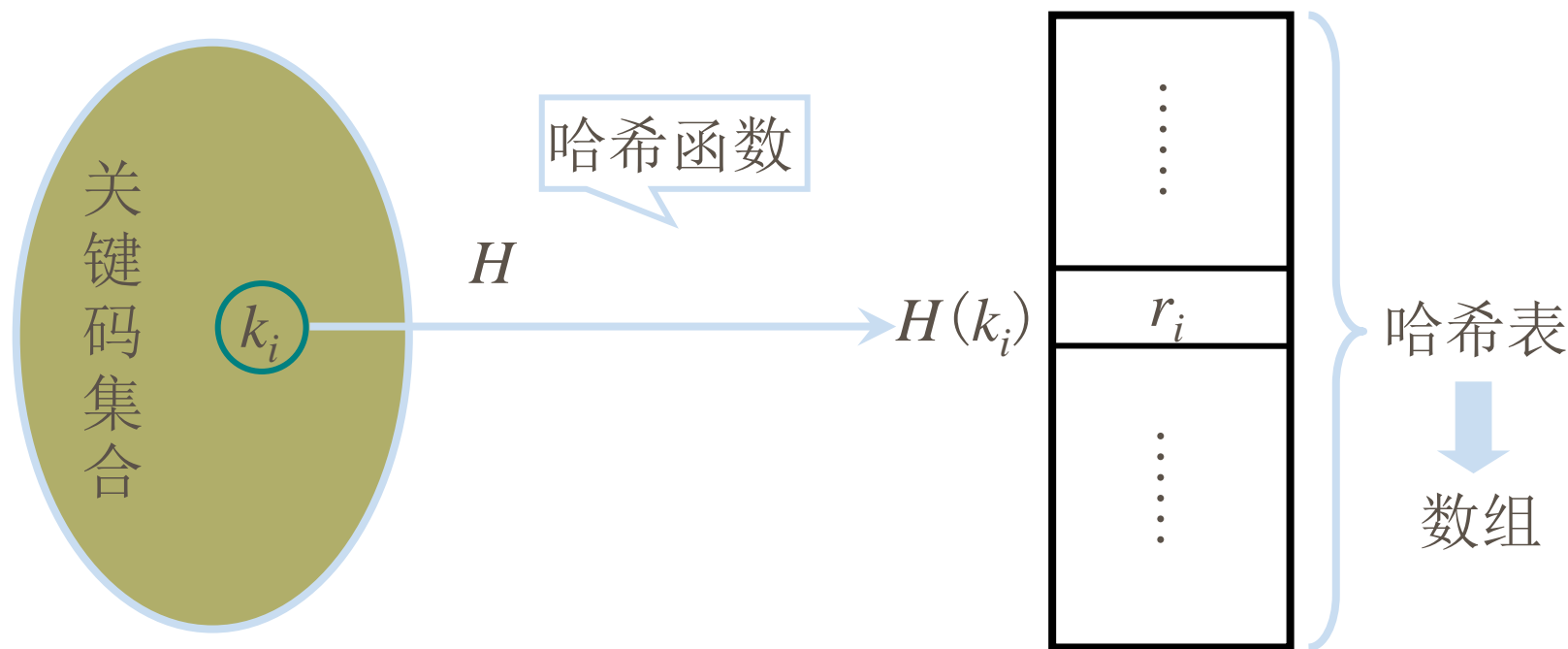
## 9.4.1 哈希表的概念

**哈希表：**采用哈希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哈希表。哈希表的容量 $m \geq$ 记录个数 $n$ 。



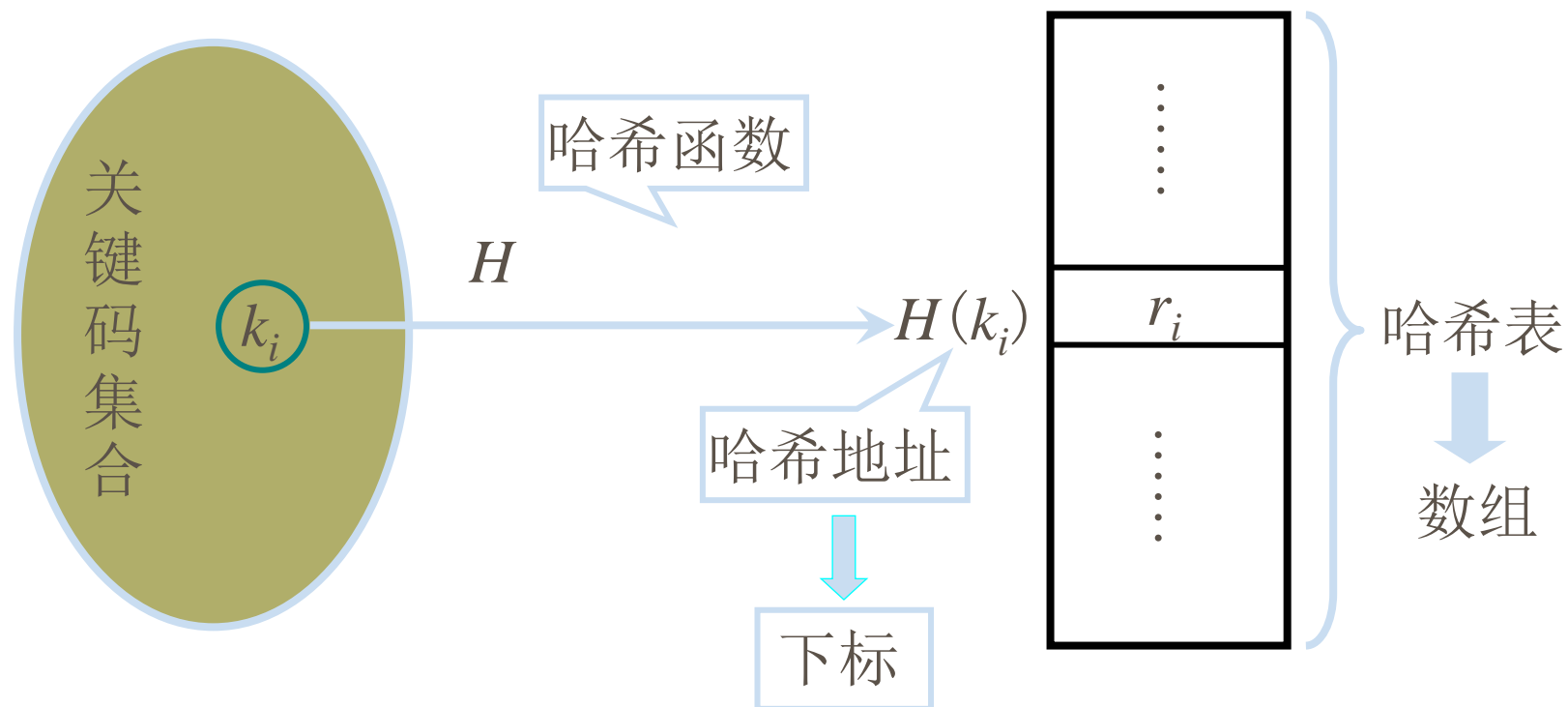
## 9.4.1 哈希表的概念

**哈希函数：** 将关键字映射为哈希表中适当存储位置的函数，也称为散列函数。



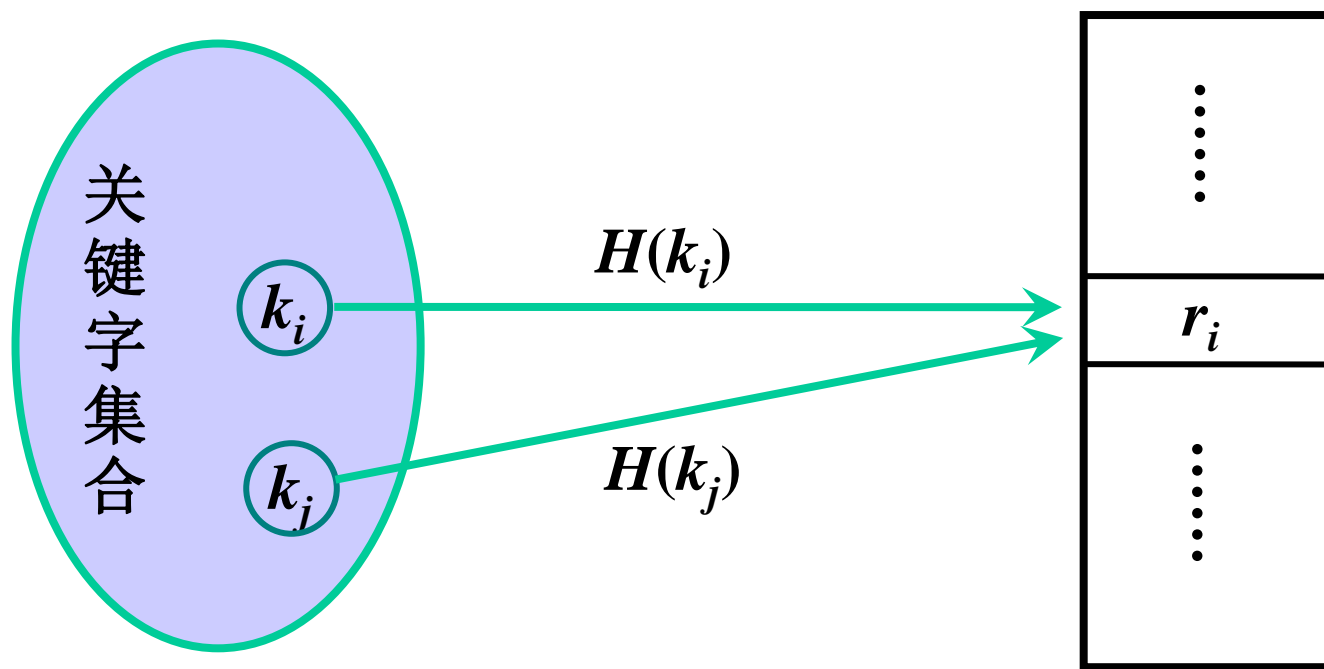
## 9.4.1 哈希表的概念

**哈希地址：** 由哈希函数所得的存储位置值 。



## 9.4.1 哈希表的概念

**冲突：**通常关键码的集合比哈希地址集合大得多，因而经过哈希函数变换后，可能将不同的关键码映射到同一个哈希地址上，这种现象称为冲突。



## 9.4.1 哈希表的概念

### 冲突现象举例：

有6个元素的关键码分别为 (14, 23, 39, 9, 25, 11)  
选取关键码与元素位置间的函数为  $H(k) = k \bmod 7$

通过哈希函数对6个元素建立哈希表：

0	1	2	3	4	5	6
14		23		39		

6个元素用7个  
地址应该足够!

$$H(14) = 14 \% 7 = 0$$

9

25

11

$$H(25) = 25 \% 7 = 4$$

$$H(11) = 11 \% 7 = 4$$

有冲突!

## 9.4.1 哈希表的概念

在哈希查找方法中，冲突是不可能避免的，只能尽可能减少。

所以，哈希方法必须解决以下两个问题：

### 1) 构造好的哈希函数

- (a) 所选函数尽可能简单，以便提高转换速度；
- (b) 所选函数对关键码计算出的地址，应在哈希地址内集中并大致均匀分布，以减少空间浪费。

## 9.4.1 哈希表的概念

---

2) 制定一个好的解决冲突的方案

查找时，如果从哈希函数计算出的地址中查不到关键码，则应当依据解决冲突的规则，有规律地查询其它相关单元。



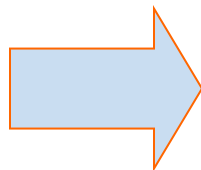
## 9.4.2 哈希函数的构造方法

要求一： $n$ 个数据原仅占用 $n$ 个地址，虽然散列查找是以空间换时间，但仍希望散列的地址空间尽量小。

要求二：无论用什么方法存储，目的都是尽量均匀地存放元素，以避免冲突。

## 9.4.2 哈希函数的构造方法

常用的哈希函数  
构造方法有：



1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 乘余取整法
7. 随机数法

## 9.4.2 哈希函数的构造方法

### 1、直接定址法

$$\text{Hash}(\text{key}) = a \cdot \text{key} + b \quad (a、b \text{ 为常数})$$

优点：以关键码key的某个线性函数值为哈希地址，  
不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

① 适用情况？

事先知道关键字，关键字集合不是很大且连续性较好。

## 9.4.2 哈希函数的构造方法

### 1、直接定址法

例： 关键码集合为{100, 300, 500, 700, 800, 900}， 选取哈希函数为 $\text{Hash}(\text{key})=\text{key}/100$ ,

则存储结构（哈希表）如下：

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

## 9.4.2 哈希函数的构造方法

### 2、数字分析法

- 特点：选用关键字的某几位组合成哈希地址。选用原则应当是：各种符号在该位上出现的频率大致相同。
- 适用于关键字位数比哈希地址位数大，且可能出现的關鍵字事先知道的情况。
- 例如：取18位身份证号码的其中5位做关键字，可定义如下哈希函数： $H(key) = (key[6] - '0') * 10^4 + (key[10] - '0') * 10^3 + (key[14] - '0') * 10^2 + (key[16] - '0') * 10 + key[17] - '0'$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	3	0	1	0	6	1	9	9	0	1	0	0	8	0	4	1	9
省		市		区（县） 下属辖区 编号		（出生）年份				月份		日期		该辖区中的 序号		校验	

## 9.4.2 哈希函数的构造方法

### 2、数字分析法

例：有一组（例如80个）关键码，其样式如下：

3	4	7	0	5	2	4
3	4	9	1	4	8	7
3	4	8	2	6	9	6
3	4	8	5	2	7	0
3	4	8	6	3	0	5
3	4	9	8	0	5	8
3	4	7	9	6	7	1
3	4	7	3	9	1	9

位号：① ② ③ ④ ⑤ ⑥ ⑦

讨论：

① 第1、2位均是“3和4”，第3位也只有“7、8、9”，因此，这几位不能用，余下四位分布较均匀，可作为哈希地址选用。

② 若哈希地址取两位（因元素仅80个），则可取这四位中的任意两位组合成哈希地址，也可以取其中两位与其它两位叠加求和后，取低两位作哈希地址。

## 9.4.2 哈希函数的构造方法

### 3、平方取中法

特点：对关键码平方后，按哈希表大小，取中间的若干位作为哈希地址。

理由：因为中间几位与数据的每一位都相关。

例：2589的平方值为6702921，可以取中间的029为地址。

## 9.4.2 哈希函数的构造方法

### 4、折叠法

特点：将关键码自左到右分成位数相等的几部分（最后一部分位数可以短些），然后将这几部分叠加求和，并按哈希表表长，取后几位作为地址。

适用于：每一位上各符号出现概率大致相同的情况。

法1：移位法——将各部分的最后一位对齐相加。

法2：间界叠加法——从一端向另一端沿分割界来回折叠后，最后一位对齐相加。

例：元素42751896, 用法1:  $\underline{427} + \underline{518} + \underline{96} = 1041$

用法2:  $\underline{427} \ 518 \ \underline{96} \rightarrow 724 + 815 + 69 = 1608$



## 9.4.2 哈希函数的构造方法

### 5、除留余数法

$Hash(key) = key \bmod p$  (p是一个整数)

特点：以关键码除以p的余数作为哈希地址。

关键：如何选取合适的p?

① 适用情况？

除留余数法是一种最简单、也是最常用的构造哈希函数的方法，并且不要求事先知道关键字的分布。

## 9.4.2 哈希函数的构造方法

### 5、除留余数法

例：  $p = 21 = 3 \times 7$

关键字	14	21	28	35	42	49	56
哈希地址	14	0	7	14	0	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**或不包含小于20质因子的**合数**。

## 9.4.2 哈希函数的构造方法

### 6、乘余取整法

$(A * \text{key} \bmod 1)$   
就是取 $A * \text{key}$ 的小数部分

$$\text{Hash}(\text{key}) = \lfloor B * (A * \text{key} \bmod 1) \rfloor$$

(A、B均为常数，且 $0 < A < 1$ ，B为整数)

特点：以关键码key乘以A，取其小数部分，然后再放大B倍并取整，作为哈希地址。

例：欲以学号最后两位作为地址，则哈希函数应为：  
 $H(k) = 100 * (0.01 * k \% 1)$

其实也可以用法5实现： $H(k) = k \% 100$

## 9.4.2 哈希函数的构造方法

---

### 7、随机数法

$Hash(key) = random(key)$  (random为伪随机函数)

适用于：关键字长度不等的情况。造表和查找都很方便。

## 9.4.2 哈希函数的构造方法

小结：构造哈希函数的原则：

- ① 执行速度（即计算哈希函数所需时间）；
- ② 关键字的长度；
- ③ 哈希表的大小；
- ④ 关键字的分布情况；
- ⑤ 查找频率。

## 9.4.3、冲突处理方法

---

常见的冲突处理方法有

1. 开放定址法（开地址法）

2. 链地址法（拉链法）

3. 再哈希法（双哈希函数法）

4. 建立一个公共溢出区

## 9.4.3、冲突处理方法

### 1、开放定址法（开地址法）

设计思路：有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

① 如何寻找下一个空的哈希地址？

- (1) 线性探测法（**要求掌握**）
- (2) 二次探测法（**要求掌握**）
- (3) 随机探测法

用开放定址法处理冲突得到的哈希表叫**闭散列表**。

## 9.4.3、冲突处理方法

### (1) 线性探测法

$$H_i = (\text{Hash}(\text{key}) + d_i) \bmod m \quad (1 \leq i < m)$$

其中：

$\text{Hash}(\text{key})$  为哈希函数

$m$  为哈希表长度

$d_i$  为增量序列  $1, 2, \dots, m-1$ , 且  $d_i = i$

含义：一旦冲突，就找附近（下一个）空地址存入。



**例：** 关键码集为 {47, 7, 29, 11, 16, 92, 22, 8, 3},

设：哈希表表长为 $m=11$ ;

哈希函数为 $\text{Hash}(\text{key})=\text{key} \bmod 11$ ;

拟用线性探测法处理冲突。建哈希表如下：

0	1	2	3	4	5	6	7	8	9	10
11	22		47	92	16	3	7	29	8	

- ① 47、7是由哈希函数得到的没有冲突的哈希地址；
- ②  $\text{Hash}(29)=7$ ，哈希地址有冲突，需寻找下一个空的哈希地址：由 $H_1=(\text{Hash}(29)+1) \bmod 11=8$ ，哈希地址8为空，因此将29存入。
- ③ 22、8、3同样在哈希地址上有冲突，也是由 $H_1$ 找到空的哈希地址的。

其中3 还连续移动了两次（二次聚集）

## 9.4.3、冲突处理方法

### (1) 线性探测法

**优点：**只要哈希表未被填满，保证能找到一个空地址单元存放有冲突的元素；

**缺点：**可能使第 $i$ 个哈希地址的同义词存入第 $i+1$ 个哈希地址，这样本应存入第 $i+1$ 个哈希地址的元素变成了第 $i+2$ 个哈希地址的同义词， $\dots$ ，因此，可能出现很多元素在相邻的哈希地址上“堆积”起来，大大降低了查找效率。

解决方案：采用二次探测法或伪随机探测法。

## 9.4.3、冲突处理方法

### 1、开放定址法（开地址法）

#### (2) 二次探测法

$$H_i = (\text{Hash}(\text{key}) \pm d_i) \bmod m$$

其中：Hash(key)为哈希函数

m为哈希表长度；

$d_i$ 为增量序列  $1^2, -1^2, 2^2, -2^2, \dots, q^2$

## 9.4.3、冲突处理方法

### (2) 二次探测法

仍举上例为 {47, 7, 29, 11, 16, 92, 22, 8, 3} ,  
改用二次探测法处理冲突, 建表如下:

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	

△    ▲                      △    △

只有3这个关键码的冲突处理与上例不同,

$\text{Hash}(3)=3$ , 哈希地址上冲突, 由

$H_1=(\text{Hash}(3)+1^2) \bmod 11=4$ , 仍然冲突;

$H_2=(\text{Hash}(3)-1^2) \bmod 11=2$ , 找到空的哈希地址, 存入。

## 9.4.3、冲突处理方法

---

### (2) 二次探测法

**优点：**探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；

**缺点：**不能保证探测到散列表的所有地址。

## 9.4.3、冲突处理方法

### (3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间 $[1, m-1]$ 的随机序列。

例：表长为11的哈希表中已填有关键字为17, 60, 29的记录，散列函数为 $H(\text{key}) = \text{key} \bmod 11$ 。现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

(1)  $H(38) = 38 \bmod 11 = 5$  冲突

$H_1 = (5+1) \bmod 11 = 6$  冲突

$H_2 = (5+2) \bmod 11 = 7$  冲突

$H_3 = (5+3) \bmod 11 = 8$  不冲突

## 9.4.3、冲突处理方法

### (3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间 $[1, m-1]$ 的随机序列。

(2)  $H(38)=38 \text{ MOD } 11=5$  冲突

$H1=(5+1^2) \text{ MOD } 11=6$  冲突

$H2=(5-1^2) \text{ MOD } 11=4$  不冲突

(3)  $H(38)=38 \text{ MOD } 11=5$  冲突

设伪随机数序列为9, 则 $H1=(5+9) \text{ MOD } 11=3$  不冲突

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

## 9.4.3、冲突处理方法

### 2、再哈希法（双哈希函数法）

$$H_i = RH_i(\text{key}) \quad i=1, 2, \dots, k$$

$RH_i$ 均是不同的哈希函数，当产生冲突时就计算另一个哈希函数，直到冲突不再发生。

优点：不易产生聚集；

缺点：增加了计算时间。



## 9.4.3、冲突处理方法

### 3、链地址法(拉链法)

**基本思想：**将具有相同哈希地址的记录链成一个单链表， $m$ 个哈希地址就设 $m$ 个单链表，然后用一个数组将 $m$ 个单链表的表头指针存储起来，形成一个动态的结构。

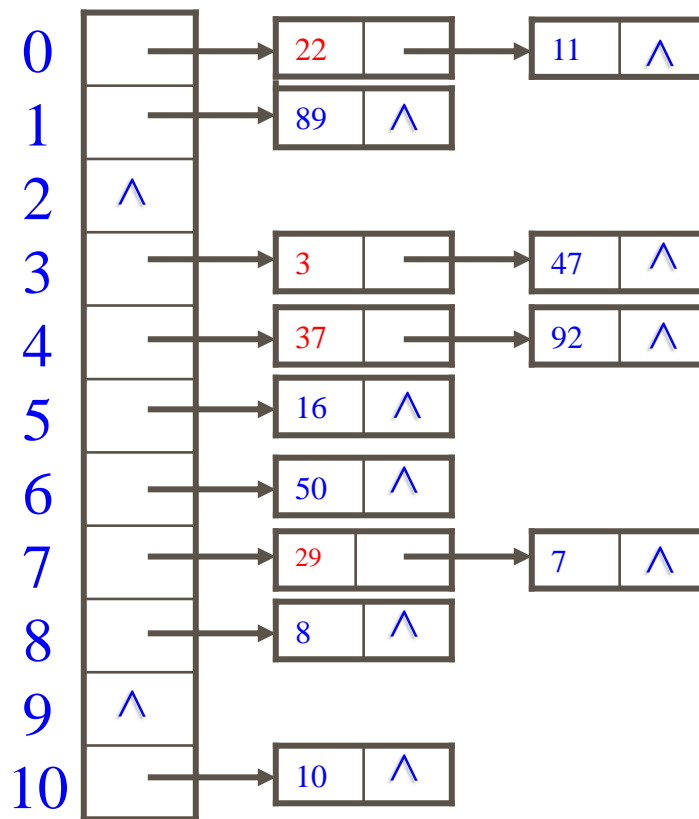
用拉链法处理冲突构造的哈希表叫做**开散列表**。

## 9.4.3、冲突处理方法

### 3、链地址法(拉链法)

例：设{ 47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 10}的哈希函数为： $\text{Hash}(\text{key}) = \text{key} \bmod 11$ ，用拉链法处理冲突，则建表如右图所示。

有冲突的元素可以插在表尾,也可以插在表头。



## 9.4.3、冲突处理方法

---

### 4. 建立一个公共溢出区

**基本思想：**哈希表包含**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，对给定值通过哈希函数计算哈希地址，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。

## 9.4.3、冲突处理方法

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

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

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

## 9.4.3、冲突处理方法

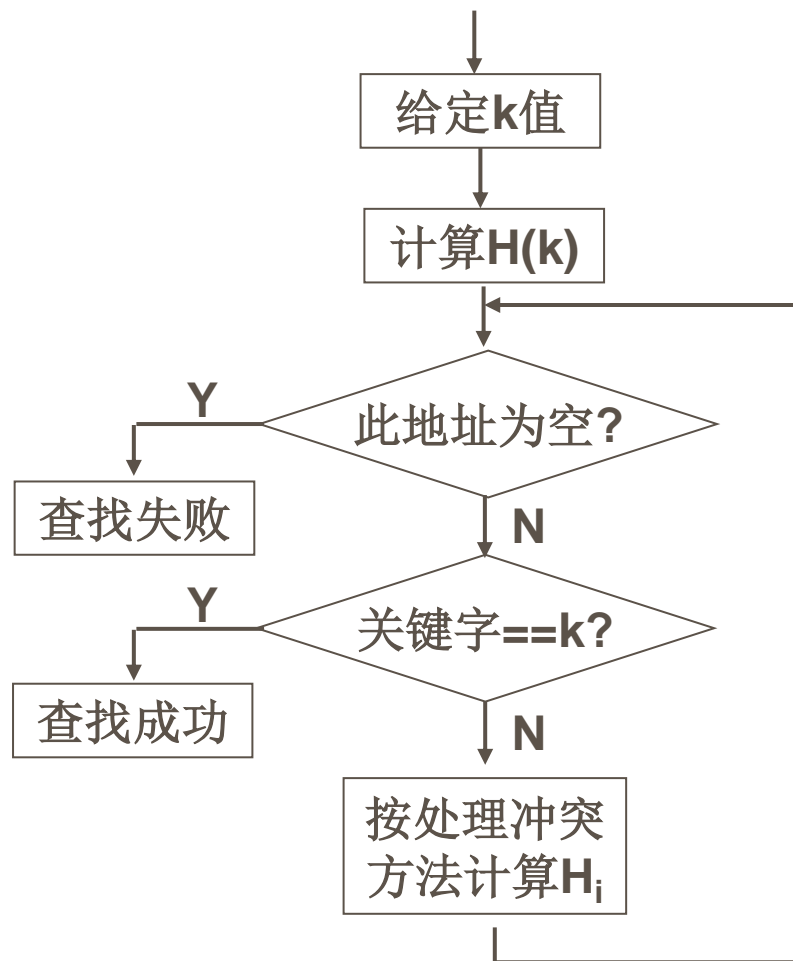
---

- 线性探测只要哈希表有空间，数据一定能够插进去。
- 二次探测，如果表长 $m=4 \times k+3$  ( $k=1,2,\dots$ ) 不成立，哈希表尽管有空间，数据有可能插不进去。
- 拉链法在任何情况下都能将数据插进去。

## 9.4.4 哈希表的查找及分析

### □ 基本思想

给定K值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。



散列表的查找过程

## 9.4.4 哈希表的查找及分析

---

### 开放定址哈希表的存储结构

```
int hashsize[N]={997};
typedef struct{
    char *key;
}ElemType;
typedef struct{
    ElemType *elem; //动态分配数组
    int count; //当前数据元素个数
    int sizeindex; //hashsize[sizeindex]为当前容量
}HashTable;
```

## 9.4.4 哈希表的查找及分析

开放地址哈希查找

```
int searchHash(HashTable h,KeyType key,int &p,int &c){// 查找
    // c用以计冲突次数, 其初值置零,供建表插入时参考
    // p返回的是插入位置
    p = Hash(key); /// 求得哈希地址
    int q = p; //q保存hash值
    while (h.elem[p].key!=NULLKEY && // 该位置中填有记录
        !EQ(key,h.elem[p].key)){ // 并且关键字不相等
        p = q;
        collision(p,++c); //求得下一次探查地址p
        // 冲突次数未达到上限, 继续; 否则退出循环
    }
    if EQ(K, H.elem[p].key) return 1;
    //查找成功, p返回待查数据元素位置
    else return 0; // 查找不成功(H.elem[p].key == NULLKEY)
}
```



## 9.4.4 哈希表的查找及分析

```
int insertHash(HashTable &h, ElemType e){  
    // 查找不成功时插入数据元素 e到开放定址哈希表H中, 并返回 K; 若冲突次数  
    // 过大, 则重建哈希表  
    c=0;  
    if (searchHash(h, e.key, p, c))  
        return -1; // 已存在  
    else if (c < hashsize[h.sizeindex]/2){ // 达到冲突上限  
        h.elem[p]=e; // 插入e  
        ++h.count; return 1;  
    }else{  
        RecreateHashTable(H); // 重建哈希表  
        return 0;  
    }  
}
```

## 9.4.4 哈希表的查找及分析

- 哈希表的查找过程仍然是给定值和关键字进行比较的过程，仍然用平均查找长度衡量哈希表的查找效率
- 查找过程需和关键字比较的次数取决于三个因素：**哈希函数、处理冲突的方法、哈希表的装填因子**。
  - ①  $\alpha$  = 表中填入的记录数 / 哈希表的长度
  - ② 处理冲突方法相同的哈希表，平均查找长度依赖于哈希表的装填因子
  - ③ 一般来说， $\alpha$  越小，发生冲突的可能性越小。

## 9.4.4 哈希表的查找及分析

### □ 哈希查找的特点：

① 优点：插入查找的速度快；

② 缺点：

- a. 通过哈希函数计算哈希地址时，占用一定的计算时间。
- b. 占用的存储空间多。为减少冲突的发生，哈希表的长度应大于记录的长度。
- c. 在哈希表中只能按关键字进行查找。

## 9.4.4 哈希表的查找及分析

明确：散列函数没有“万能”通式，要根据元素集合的特性而分别构造。

1：哈希查找的速度是否为真正的 $O(1)$ ？

答：不是。由于冲突的产生，使得哈希表的查找过程仍然要进行比较，仍然要以平均查找长度ASL来衡量。

## 9.4.4 哈希表的查找及分析

2: “冲突”是不是特别讨厌?

答: 不一定! 正因为有冲突, 使得文件加密后无法破译! (单向散列函数不可逆, 常用于数字签名和间接加密)。

哈希表特点: 源文件稍稍改动, 会导致哈希表变动很大。

典型应用: md5散列算法

## 9.4.4 哈希表的查找及分析

3 散列存储的查找效率到底是多少？

答：ASL与装填因子 $\alpha$ 有关！既不是严格的 $O(1)$ ，也不是 $O(n)$

$$ASL \approx 1 + \frac{\alpha}{2} \quad (\text{拉链法})$$

$$ASL \approx \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad (\text{线性探测法})$$

$$ASL \approx -\frac{1}{\alpha} \ln(1 - \alpha) \quad (\text{随机探测法})$$

应尽量选择一个合适的 $\alpha$ ，以降低ASL的长度

## 9.4.5 哈希应用

### 数字签名（数字手印）

HASH函数，是在信息安全领域有广泛和重要应用的密码算法，它有一种类似于指纹的应用。

在网络安全协议中，哈希函数用来处理电子签名，将冗长的签名文件压缩为一段独特的数字信息，像指纹鉴别身份一样保证原来数字签名文件的合法性和安全性。

## 9.4.5 哈希应用

### 数字签名（数字手印）

SHA-1和MD5都是目前最常用的哈希函数。经过这些算法的处理，原始信息即使只更动一个字母，对应的压缩信息也会变为截然不同的“指纹”，这就保证了经过处理信息的唯一性。为电子商务等提供了数字认证的可能性。



## 9.4.5 哈希应用

md5散列算法——信息-摘要算法（1991年）

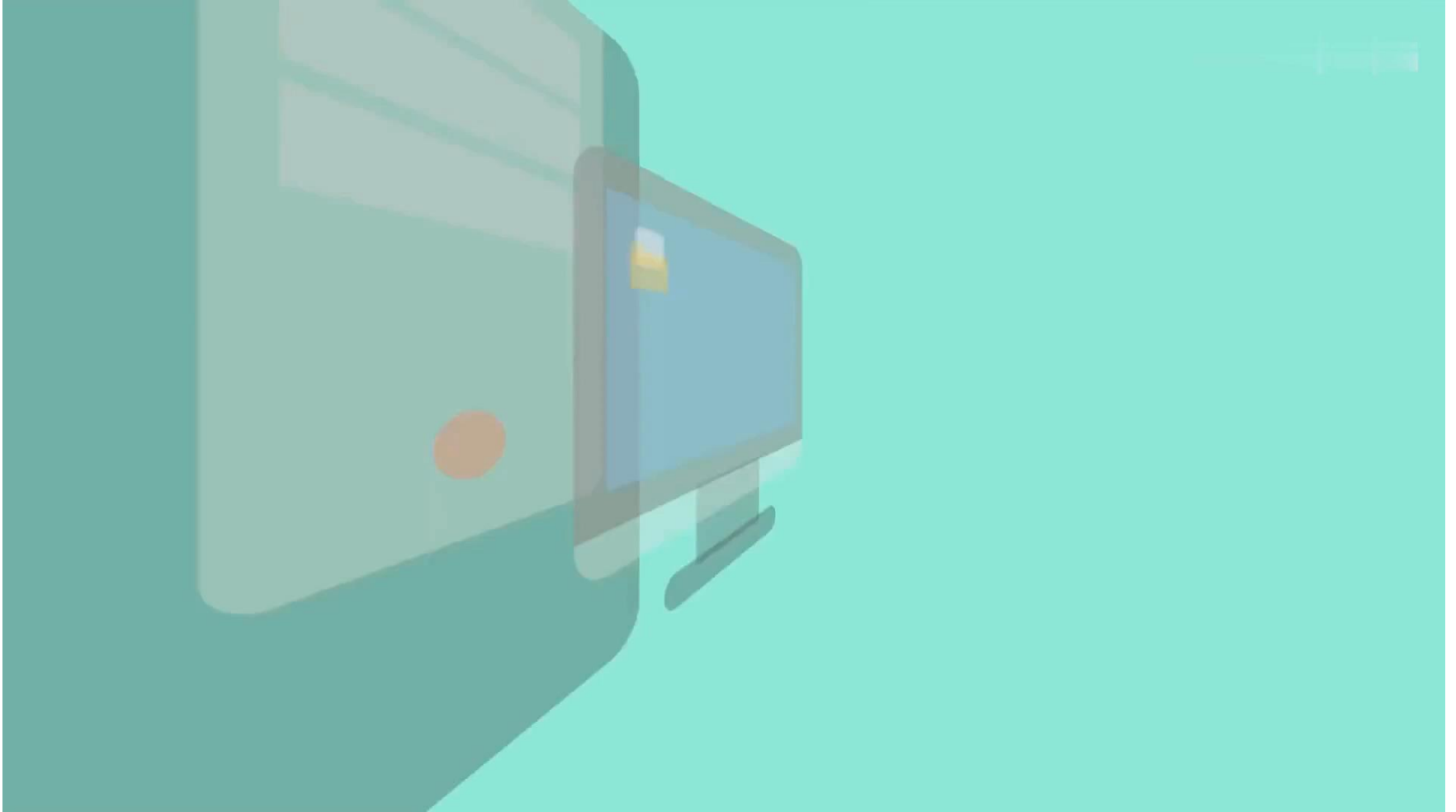
message-digest algorithm) ——用于加解密和数字签名

md5的典型应用是对一段信息（message）产生一个128位的信息摘要（message-digest），以防止被篡改。

md5以512位分组来处理输入的信息，且每一分组又被划分为16个32位子分组，经过了一系列的处理后，算法的输出由四个32位(链接变量参数)分组组成，将这四个32位分组级联后将生成一个128位散列值。

## 9.4.5 哈希应用

---



## 9.4.5 哈希应用

### 例1: md5用于BBS登录时的身份认证

在BBS服务器上，用户的密码都是以md5（或其它类似的算法）经加密后存储在文件系统中。

当用户登录的时候，系统把用户输入的密码计算成md5值，然后再去和预存在服务器中的md5值进行比较，进而确定输入的密码是否正确。

## 9.4.5 哈希应用

---

### 例1: md5用于BBS登录时的身份认证

优点：系统在不知用户密码明码的情况下就可以确定用户登录系统的合法性。这不但可以避免用户的密码被具有系统管理员权限的用户知道，而且还在一定程度上增加了密码被破解的难度。

## 9.4.5 哈希应用

---

### 例2：单纯的数据校验

下载光盘镜像文件时 一般会有一个MD5文件记录校验和，以防止下载600MB的文件出现错误导致软件无法安装，这在Linux/FreeBSD等通过网络发布的光盘安装系统中常用。

## 9.4.5 哈希应用

---

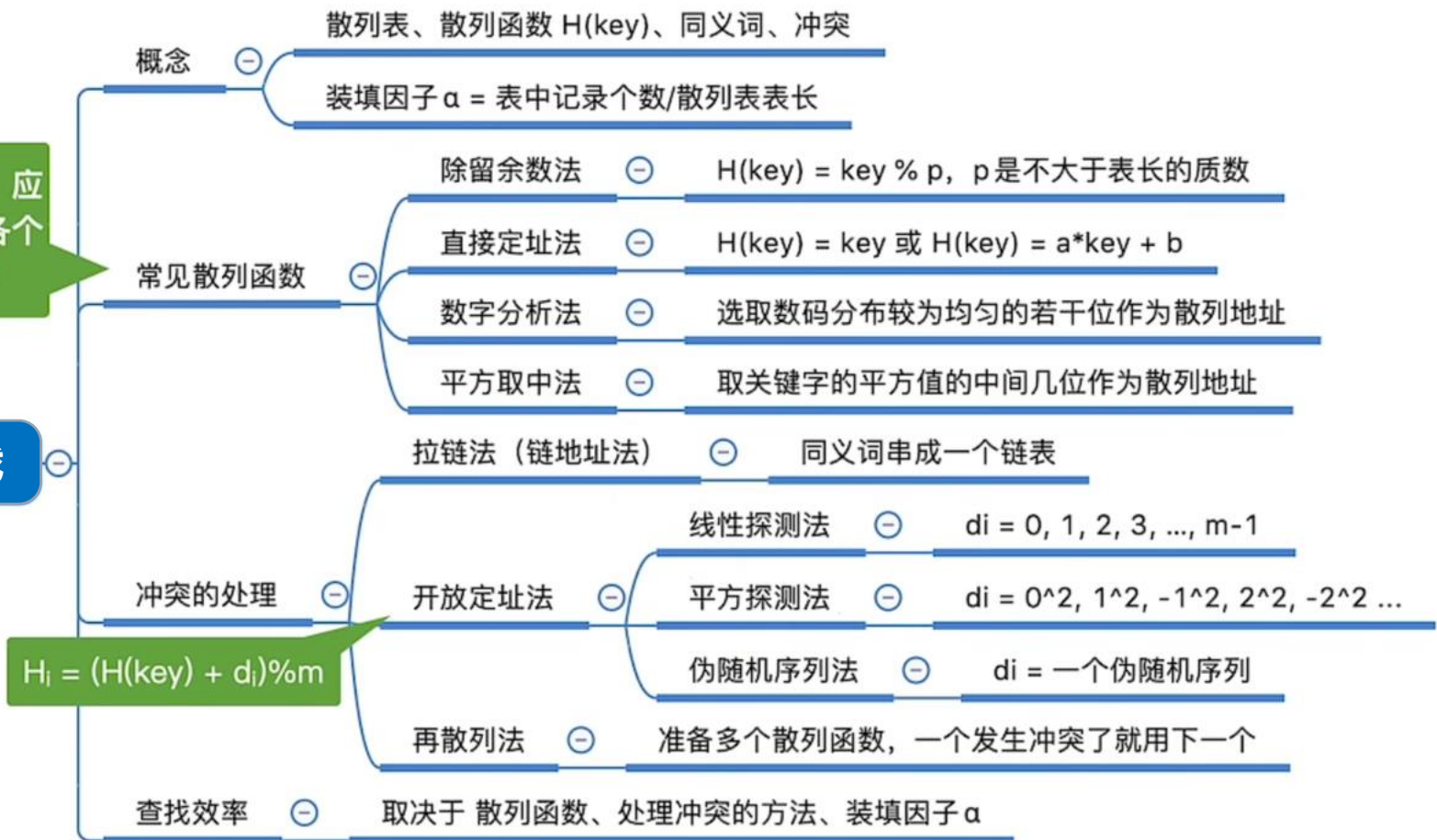
现在使用的重要计算机安全协议，如SSL，PGP都用哈希函数来进行签名。

但是，如果有人一旦找到两个文件可以产生相同的压缩值，就可以伪造签名，给网络安全领域带来巨大隐患。

# 小结

对给定的关键字集合，应尽可能均匀地散列到各个地址，使冲突更少

## 哈希查找



# 总结

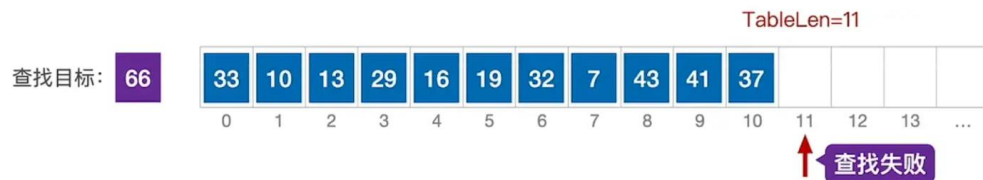
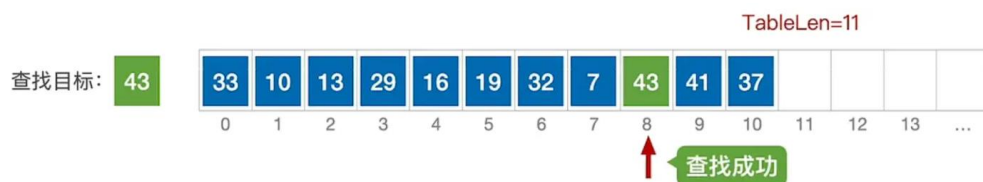




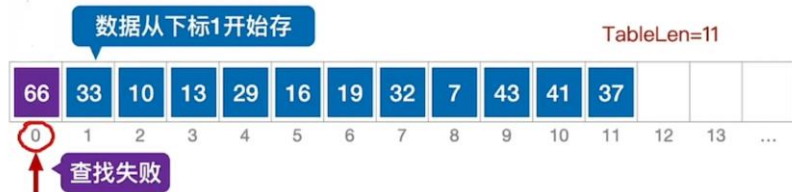
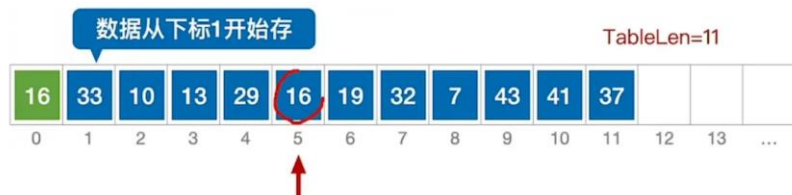
# 总结

## 1、顺序查找

算法思想：从头到脚挨个找（或者反过来也可以）



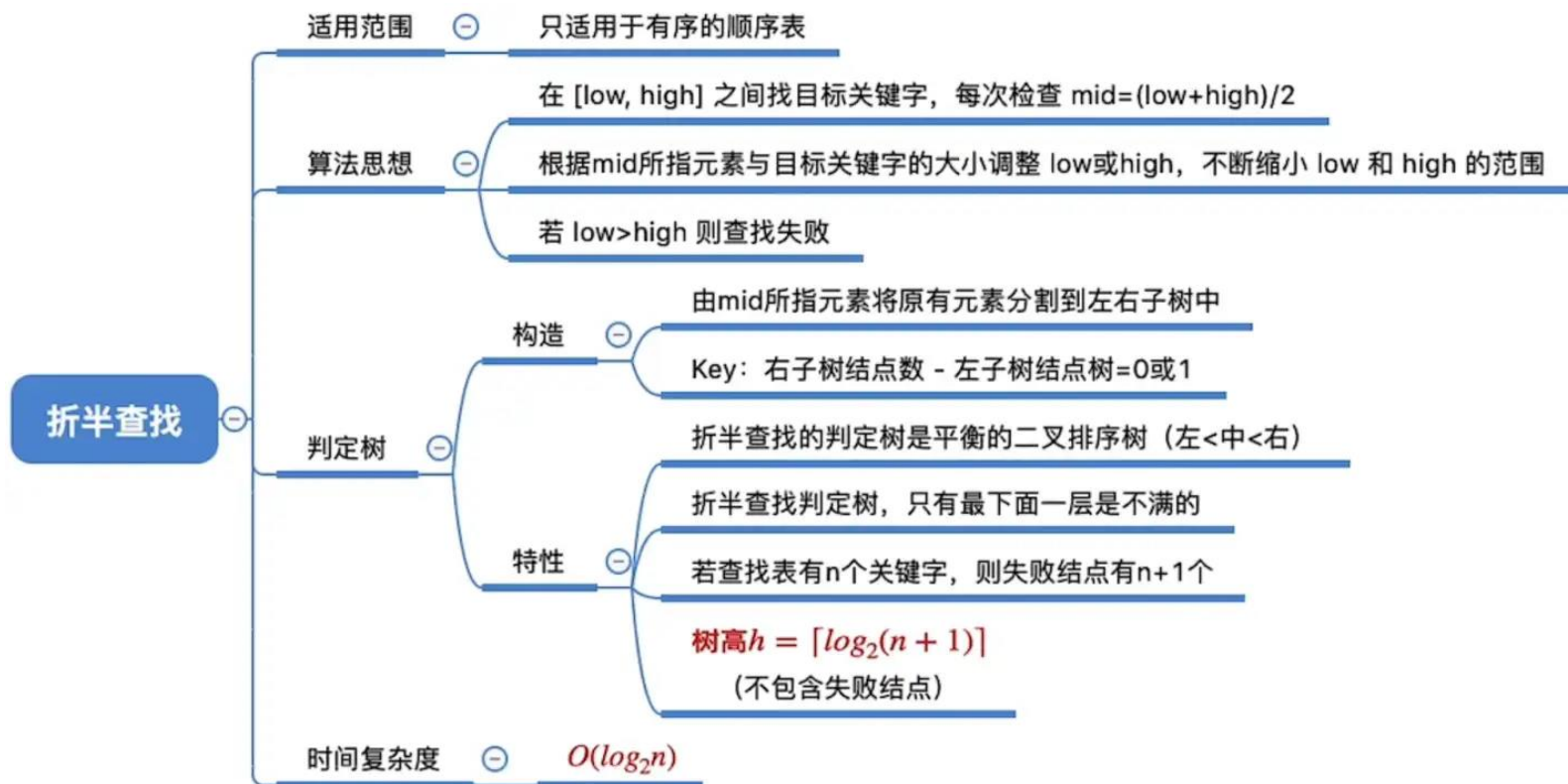
加哨兵



# 总结

## 2、折半查找

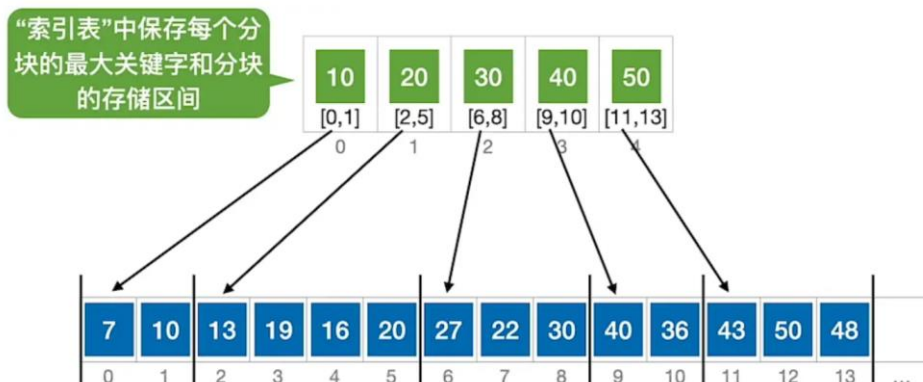
算法思想：将有序数组分成两个部分，找到中间元素，与要查找的关键字进行比较。如果相等，则查找成功；如果关键字比中间元素小，则在左半部分继续查找；如果关键字比中间元素大，则在右半部分继续查找



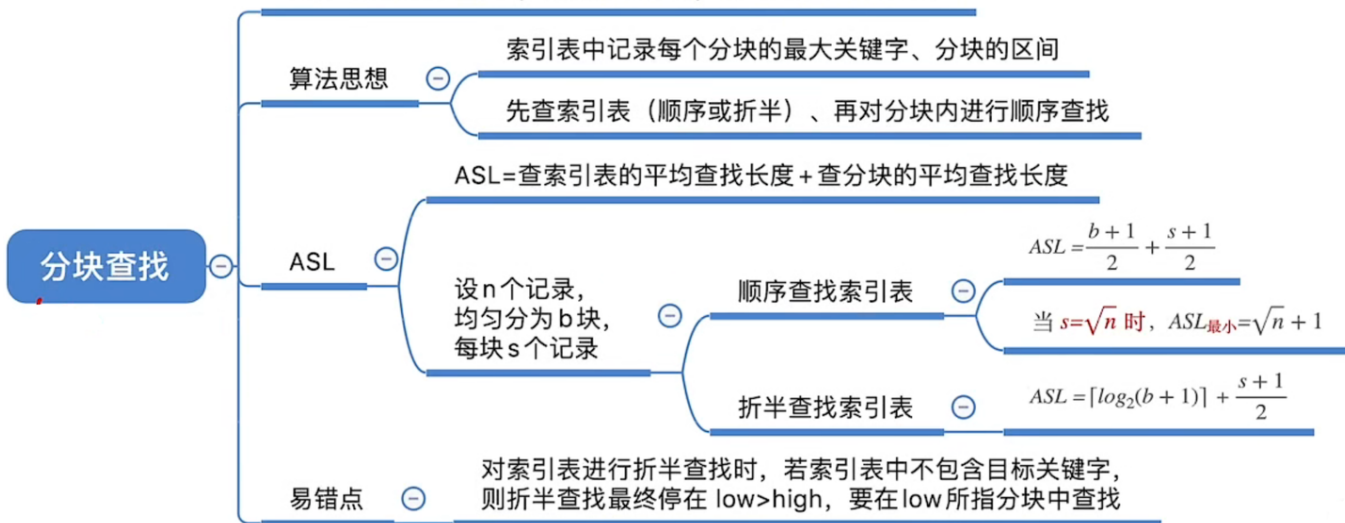
# 总结

## 3、分块查找

算法思想：块内无序、块间有序。块间折半查找，块内顺序查找



又称“索引顺序查找”，数据分块存储，块内无序、块间有序



# 总结

## 4、二叉排序树

或是一棵空树；或者是具有如下性质的非空二叉树：

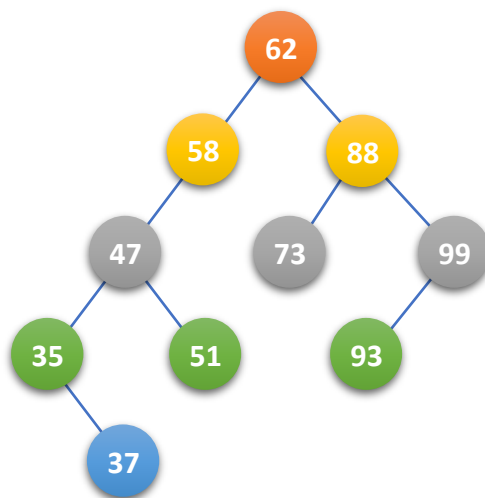
- (1) 左子树的所有结点均小于根的值；
- (2) 右子树的所有结点均大于根的值；
- (3) 它的左右子树也分别为二叉排序树。

- 二叉排序树查找

- 二叉排序树插入

数据插入

62,88,58,47,35,73,51,99,37,93



# 总结

## 4、二叉排序树

或是一棵空树；或者是具有如下性质的非空二叉树：

- (1) 左子树的所有结点均小于根的值；
- (2) 右子树的所有结点均大于根的值；
- (3) 它的左右子树也分别为二叉排序树。

### ● 二叉排序树删除

- ✓ 被删除的结点D是叶子结点：直接从二叉排序树当中移除即可，不影响树的结构
- ✓ 被删除的结点D仅有一个孩子：如果只有左孩子，没有右孩子，那么只需要把要删除结点的左孩子链接到要删除结点的父亲节点，然后删除D结点就好了；如果只有右孩子，没有左孩子，那么只要将要删除结点D的右孩子重接到要删除结点D的父亲结点
- ✓ 被删除结点的左右孩子都存在：1) 令p的左子树为 f的左子树，p的右子树接为s的右子树；2) 直接令s代替p，s的左子树接为p<sub>L</sub>

# 总结

---

## 5、平衡二叉树

或是一棵空树；或者是具有如下性质的非空二叉树：

- (1) 是二叉排序树；
- (2) 左子树和右子树都是平衡二叉树；
- (3) 所有结点的平衡因子只可能是-1,0,和1。

- 平衡二叉树查找
- 平衡二叉树插入
- 平衡二叉树删除

# 总结

---

## 5、平衡二叉树

(1) 首先确定属于何种类型的旋转；

- LL平衡旋转（单右旋）
- RR平衡旋转（单左旋）
- LR平衡旋转（先左后右）
- RL平衡旋转（先右后左）

(2) 确定旋转轴

LL和RR型的旋转轴在沿着失衡路径上，以失去平衡点的**直接后继结点**作为旋转轴；LR和RL的旋转轴是沿着失衡路径上，失去平衡点的**后两层结点**作为旋转轴。

# 总结

---

## 6、B-树

一棵 $m$ 阶的B-树，或为空树，或为满足下列特性的 $m$ 叉树：

- (1) 树中每个结点至多有 $m$ 棵子树， $m-1$ 个关键字；
- (2) 若根结点不是叶子结点，则至少有两棵子树；
- (3) 除根之外的所有非终端结点至少有  $\lceil m/2 \rceil$  棵子树，至少有  $\lceil m/2 \rceil - 1$  个关键字。

- B-树查找
- B-树插入
- B-树删除



# 总结

---

## 7、B+树

一棵 $m$ 阶的B+树，或为空树，或为满足下列特性的 $m$ 叉树：

- (1) 每个结点最多 $m$ 棵子树，每个结点中最多含有 $m$ 个关键字；
- (2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，叶子结点本身依关键字的大小自小而大顺序链接；
- (3) 所有的非终端结点可以看成是索引部分，结点仅含其子树中的最大（或最小）关键字。

- B+树查找
- B+树插入
- B+树删除

# 总结

## 7、B-和+树



# 总结

---

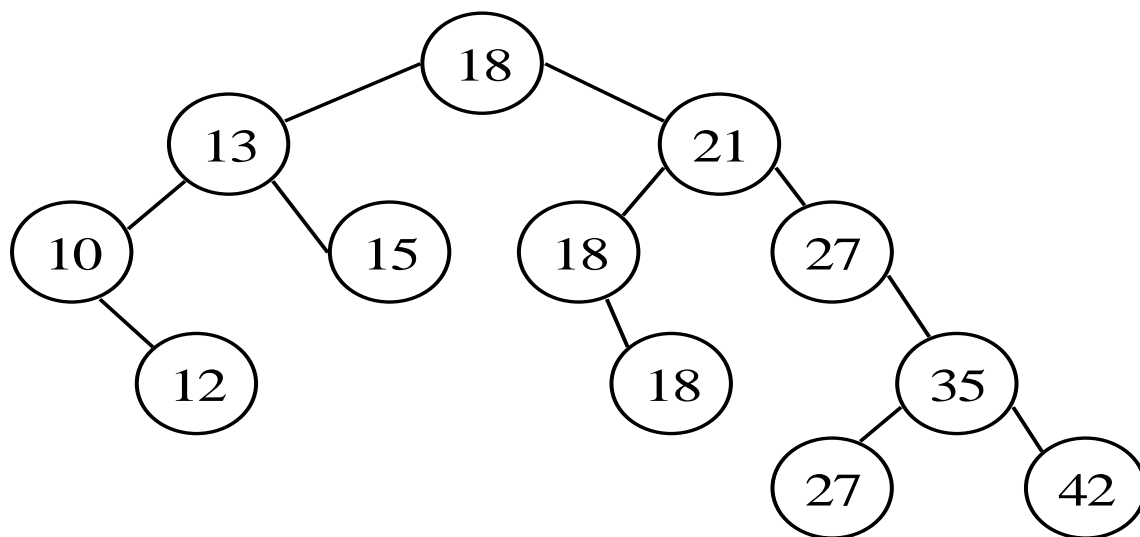
## 8、哈希表

根据关键码和值 (key和value) 直接进行访问的数据结构，通过key和value来映射到集合中的一个位置，这样就可以很快找到集合中的对应元素。

**影响因素：**哈希函数、哈希表的大小、冲突处理方法。

# 总结

例：在二叉排序树的结构中，有些数据元素值可能是相同的，设计一个算法实现按递增有序打印结点的数据域，要求相同的数据元素仅输出一个，算法还应能报出最后被滤掉，而未输出的数据元素个数，对如图所示的二叉排序树，输出为：**10, 12, 13, 15, 18, 21, 27, 35, 42**。滤掉**3**个元素。



# 总结

```
void BSTPrint(BSTree t,int *count) {  
    //递增序输出二叉排序树中结点的值，滤去重复元素  
    if(t) {  
        BSTPrint(t->lchild); //中序遍历左子树  
        if(pre==null) //pre是当前访问结点的前驱，  
            pre=t; //调用本算法时初值为null  
        else if(pre->key==t->key)  
            *count++; // *count记重复元素，初值为0  
        else { //输出元素，前驱后移  
            printf("%4d",t->key); pre=t;  
        }  
        BSTPrint(t->rchild); //中序遍历右子树  
    }  
}
```

**正在答疑**

---