



数据结构：检索

Data Structure

主讲教师：杨晓波

E-mail: 248133074@qq.com



检索（查找，Search）

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）。

查找结果

■ 查找成功

- 若查找表中存在这样一个记录，则称“**查找成功**”。
- 给出整个记录的信息，或指示该记录在查找表中的位置；

■ 查找不成功

- 若查找表中不存在这样一个记录，则称“**查找不成功**”。
- 给出“空记录”或“空指针”。

检索的形式化定义

假设 k_1, k_2, \dots, k_n 是互不相同的**关键码值**，有一个包含 n 条记录的集合 C ，形式如下：

$(k_1, I_1), (k_2, I_2), \dots, (k_n, I_n)$ 其中 I_j 是与关键码值 k_j 相关联的信息， $1 \leq j \leq n$ 。

给定某个关键码值 K

检索问题就是在 C 中定位记录 (k_j, I_j) 使得 $k_j=K$ 。

检索就是定位关键码值 $k_j=K$ 的系统化方法

检索的分类

- 检索按照是否精确匹配可分为
 - 精确匹配查询
 - 检索关键码值与某个特定值匹配的记录
 - 如：手机银行查询
 - 范围查询
 - 检索关键码值在某个指定值范围内的所有记录
 - 如：期中、期末成绩分段统计。

调查问卷

使用了上面两种查询中哪种查询？或两种都使用过？
请举例说明

检索算法的分类

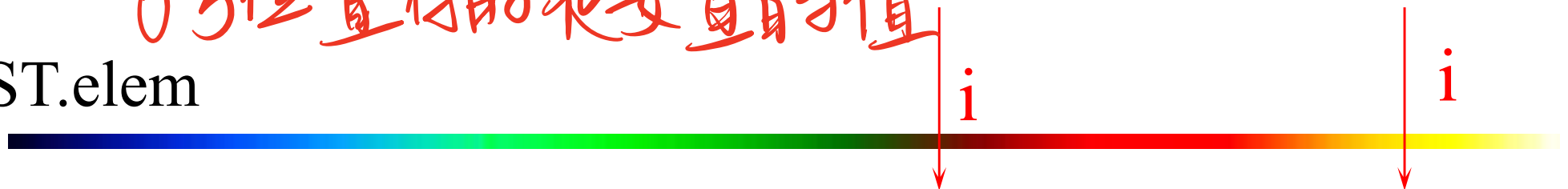
- 检索算法可分为三类
 - 顺序表和线性表方法
 - 顺序查找，复杂度 $O(n)$
 - 对已排序的线性表，可用
 - 二分法检索，复杂度 $O(\log n)$
 - 树索引方法
 - 如：B+树、BST、AVL树、键树（Trie树）
 - 散列 (hash)
 - 根据关键码值计算哈希函数，直接访问的方法
 - 理想情况复杂度 $O(1)$

顺序检索

- 针对线性表里的所有记录，逐个进行关键码和给定值的**比较**。
 - 若某个记录的关键码和给定值比较相等，则检索成功；
 - 否则检索失败(找遍了仍找不到)。
- 存储：可以顺序、链接
- 排序要求：无

0号位置存的是要查的值

ST.elem



64	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=64

ST.Length

这两种情况分别比较了多少次?

ST.elem

60	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=60

ST.Length

空间换时间(少比较)

顺序检索算法

```
template <class Type>
class Item {
    private:
        Type key;           //关键码域
        //...               //其它域
    public:
        Item(Type value):key(value) {}
        Type getKey() {return key;} //取关键码值
        void setKey(Type k){ key=k;} //置关键码
};
vector<Item<Type>*> dataList; 查找表
```


“监视哨” 顺序检索算法

- 检索成功返回元素位置，检索失败统一返回0；

```
template <class Type> int SeqSearch(vector<Item<Type>*>&
dataList, int length, Type k) {
```

//在长度为length的datalist中查找关键码k

```
int i=length;
```

//将第0个元素设为待检索值

```
dataList[0]->setKey(k); //设监视哨
```

```
while(dataList[i]->getKey()!=k) i--;
```

```
return i;
```

```
}
```

从后往前找

如未查到

顺序查找的时间性能

查找算法的**平均查找长度** (Average Search Length)

为确定记录在查找表中的位置，需和给定值**进行比较**的**关键字个数的期望值**

$$ASL = \sum_{i=1}^n P_i C_i$$

其中： n 为表长， P_i 为查找表中第 i 个记录的概率，

且 $\sum_{i=1}^n P_i = 1$,

C_i 为找到该记录时，曾和给定值比较过的**关键字的个数**。

顺序查找的时间性能

对顺序表而言, $C_i = n - i + 1$

$$ASL = nP_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n$$

在等概率查找的情况下, $P_i = \frac{1}{n}$

顺序表查找的平均查找长度为:

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

顺序查找的时间性能

在不等概率查找的情况下, ASL_{ss} 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。

本章要点

■ 基于线性结构的查找

■ 顺序查找

■ 二分检索法

■ 自组织线性表

■ 基于树结构的查找

■ 基于计算的查找

二分检索法的应用背景

- 有些应用场景中数据建立后更新不是很频繁，大量的操作是查找。这样可以先花点时间做个数据的预处理，将所有记录按照关键码的键值进行排序。
- 对于已排序的表，可以采用比顺序查找更高效一些的方法进行查找。
- 比如大家很熟悉的猜数字的游戏，要同学们猜1~100之间的一个数字，你们是怎么猜的呢？最多需要猜多少次呢？


二分检索法

- 应用前提：已排序的表
- 将任一元素 $\text{dataList}[i].\text{Key}$ 与给定值 K 比较
 - 三种情况：
 - (1) $\text{Key} = K$ ，检索成功，返回 $\text{dataList}[i]$
 - (2) $\text{Key} > K$ ，若有则一定排在 $\text{dataList}[i]$ 前
 - (3) $\text{Key} < K$ ，若有则一定排在 $\text{dataList}[i]$ 后
 - 经典场景中 $i = \text{mid} = (\text{low} + \text{high}) / 2$ 向下取整
- 缩小进一步检索的区间

二分法检索算法

```
template <class Type> int BinSearch (vector<Item<Type>*>&
    dataList, int length, Type k){
    int low=1, high=length, mid;
    while (low<=high) {
        mid=(low+high)/2;
        if (k<dataList[mid]->getKey())
            high = mid-1;        //右缩检索区间
        else if (k>dataList[mid]->getKey())
            low = mid+1;        //左缩检索区间
        else return mid;        //成功返回位置
    }
    return 0; //检索失败，返回0
}
```


关键码18 low=1 high=9



1	2	3	4	5	6	7	8	9
15	17	18	22	35	51	60	88	93

low mid high

第一次: $l=1, h=9, mid=5; array[5]=35 > 18$

第二次: $l=1, h=4, mid=2; array[2]=17 < 18$

第三次: $l=3, h=4, mid=3; array[3]=18 = 18$

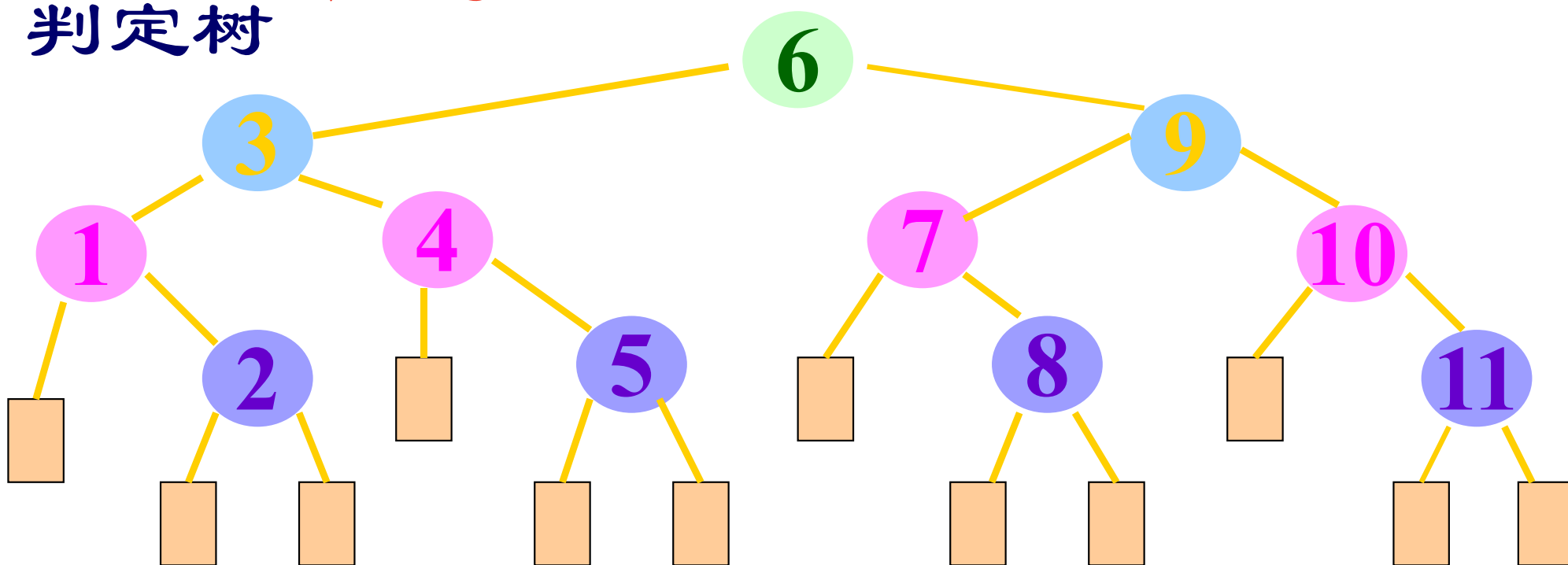
折半查找的判定树和平均查找长度

假设: $n=11$

i	1	2	3	4	5	6	7	8	9	10	11
C_i	3	4	2	3	4	1	3	4	2	3	4

C_i 为比较次数

判定树



折半查找的平均查找长度

一般情况下，表长为n的折半查找的判定树的深度和含有n个结点的完全二叉树的深度相同。

假设 $n=2^h-1$ 并且查找概率相等,则

$$ASL_{bs} = \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left[\sum_{j=1}^h j \times 2^{j-1} \right] = \frac{n+1}{n} \log_2 (n+1) - 1$$

在 $n>50$ 时，可得近似结果

$$\underline{ASL}_{bs} = \log_2 (n+1) - 1 \approx O(\log n)$$

平均查找长度

本章要点

■ 基于线性结构的查找

- 顺序查找

- 二分检索法

- 自组织线性表

■ 基于树结构的查找

■ 基于计算的查找

自组织线性表

- 线性表的另一种组织方法
 - 根据（估算的）访问频率排列记录，一般先放置请求频率最高的记录，然后是请求频率次高的记录，依次类推.
- 自组织线性表可从第一个位置开始顺序检索
预计的比较时间代价为：（ p_i 为第 i 个记录被请求的概率）

$$\overline{C}_n = 1p_1 + 2p_2 + \dots + np_n.$$

自组织线性表的比较次数 例(1)

(1) 所有记录的访问频率相同.

$$\overline{C}_n = \sum_{i=1}^n i / n = (n + 1) / 2$$

例 (2)

(2) 指数频率

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \leq i \leq n-1 \\ 1/2^{n-1} & \text{if } i = n \end{cases}$$

$$\overline{C}_n \approx \sum_{i=1}^n (i/2^i) \approx 2.$$

Zipf 分布

应用:

- 自然语言的单词使用频率.
- 城市中的人口规模.

$$\overline{C}_n = \sum_{i=1}^n i / i H_n = n / H_n \approx n / \log_e n.$$

80/20 规则:

- 80% 的访问都是对 20% 的记录进行的.
- 当频率遵循 80/20 规则, 代价为

$$\overline{C}_n \approx 0.1n.$$

自组织线性表

自组织线性表根据实际的记录访问模式在线性表中修改记录顺序.

自组织线性表使用启发式规则决定如何重新排列线性表.

- 计数统计法
- 移至前端
- 转置

启发式规则——计数统计法

【一起做】假设有8条记录，关键码值为A到H，最初以字母顺序排列，按照下面的访问模式：**F D F G E G F A D F G E**

- **计数统计方法**：保持线性表按照访问频率排序。(类似于缓冲池替代策略中的“最不频繁使用”法.)

- 初始

记录表：

A	B	C	D	E	F	G	H
---	---	---	---	---	---	---	---

- 计数数组

1	0	0	2	2	4	3	0
---	---	---	---	---	---	---	---

- 记录表中按访问频率排序后为：**FGDEABCH**

启发式规则——移至前端

【一起做】假设有8条记录，关键码值为A到H，最初以字母顺序排列，按照下面的访问模式：**F D F G E G F A D F G E**

- **移至前端**：找到一条记录就把它放到线性表的最前面。

启发式规则-移至前端举例

初始记录序列为: ABCDEFGH

访问模式: F D F G E G F A D F G E

结果: **EGFDABCH**

访问记录	记录序列
F	FABCDEGH
D	DFABCEGH
F	FDABCEGH
G	GFDABCEH
E	EGFDABCH
G	GEFDABCH
F	FGEDABCH
A	AFGEDBCH
D	DAFGEBCH
F	FDAGEBCH
G	GFDAEBCH
E	EGFDABCH

启发式规则——转置

【一起做】假设有8条记录，关键码值为A到H，最初以字母顺序排列，按照下面的访问模式： **F D F G E G F A D F G E**

- **转置**:把找到的记录与它在线性表中的前一条记录交换位置。(碎步移)

每次往前移一步

启发式规则-转置举例

初始记录序列为: ABCDEFGH

访问模式: F D F G E G F A D F G E

结果: **ABFDGECH**

访问记录	记录序列
F	ABCD FE GH
D	ABDC FE GH
F	ABDF CE GH
G	ABDFC GE H
E	ABDFCE GH
G	ABDFC GE H
F	ABFDC GE H
A	ABFDC GE H
D	ABDFC GE H
F	ABFDC GE H
G	ABFDG CE H
E	ABFDGE CH

启发式规则-转置举例

初始记录序列为: **ABCDEFGH**

访问模式: **F D F G E G F A D F G E**

结果: **ABFDGECH**

访问记录	记录序列
F	ABCD F EGH
D	ABDC F EGH
F	ABD F CEGH
G	ABD F CGEH
E	ABD F CEGH
G	ABD F CGEH
F	AB F DCGEH
A	AB F DCGEH
D	ABD F CGEH
F	AB F DCGEH
G	AB F DGCEH
E	AB F DGECH

自组织线性表应用举例

——文本压缩示例

发送者和接收者都以同样的方式记录单词在线性表中的位置，线性表根据**移至前端**规则自组织。

- ⑩ 如果单词没有出现过，就传送这个单词。
- ⑩ 否则就传送这个单词在线性表当前的位置。

原文：

The car on the left hit the car I left.

压缩编码报文：

The car on 3 left hit 3 5 I 5.

这种**压缩**方法的思想类似于Ziv-Lempel 编码算法。

自组织线性表应用举例

——文本压缩示例

The car on the left hit the car I left.

发送方/接收方线性表

(1) **The**

(2) **car** The

(3) **on** car The

(4) The on car 发送**3**

(5) **left** The on car

(6) **hit** left The on car

(7) The hit left on car 发送**3**

自组织线性表应用举例

——文本压缩示例

The car on the left hit the car I left.

发送方/接收方线性表

(7) **The hit left on car** 发送**3**

已发送报文**The car on 3 left hit 3**

(8) **car The hit left on**发送**5**

(9) **I car The hit left on**

(10) **left I car The hit on**发送**5**

发送的报文内容如下：

The car on 3 left hit 3 5 I 5.

本章要点

■ 基于线性结构的查找

- 顺序查找

- 二分检索法

- 自组织线性表

- 集合的检索

■ 基于树结构的查找

■ 基于计算的查找

集合的检索

■ 动机

- 确定某个值是否某集合的元素

■ 位向量或者位图（bitmap）

- 在关键码值范围有限的情况下，存储一个位数组，为每个可能的元素分配一个比特位位置
- 若元素包含在实际集合中，则对应位为1，否则为0
- 利用布尔操作和位操作实现查询



例：确定0到15之间的某个值是不是素数

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

确定某个值是不是素数，只需要简单地检查对应的位。



举例 (2)

例：确定数值0到15集合之间的奇素数

0011010100010100 & 0101010101010101

素数bit向量

奇数bit向量

构建两个bit向量，用向量的差（或位运算）
运算完成计算



举例 (3)

文档检索 (document retrieval)

例：从一组文档中挑选出包含某些选定关键字的文档

对于每一个关键字，标识一个位向量，每个文档一个bit位。
如果想知道哪些文档包含某三个关键字，就把相应的三个位向量进行AND操作。位置上值为1的位就对应所需要的文档

对于每一个文档，标识一个位向量（称为签名文件），每个关键字一个bit位。

可以通过对签名的操作找到带有所需要关键字组合的文档



本章要点

■ 基于线性结构的查找

- 顺序查找
- 二分检索法
- 自组织线性表
- 集合的检索
- 分块查找

■ 基于树结构的查找

■ 基于计算的查找

分块查找 (block search)

动机:

大规模集合的顺序查找效率低

二分查找不适用于顺序文件的查找

分块查找 (跳跃查找jump search) 基本思想:

把一个大的线性表分解成若干块, 每块中的元素可以任意存放, 但块与块之间必须排序。

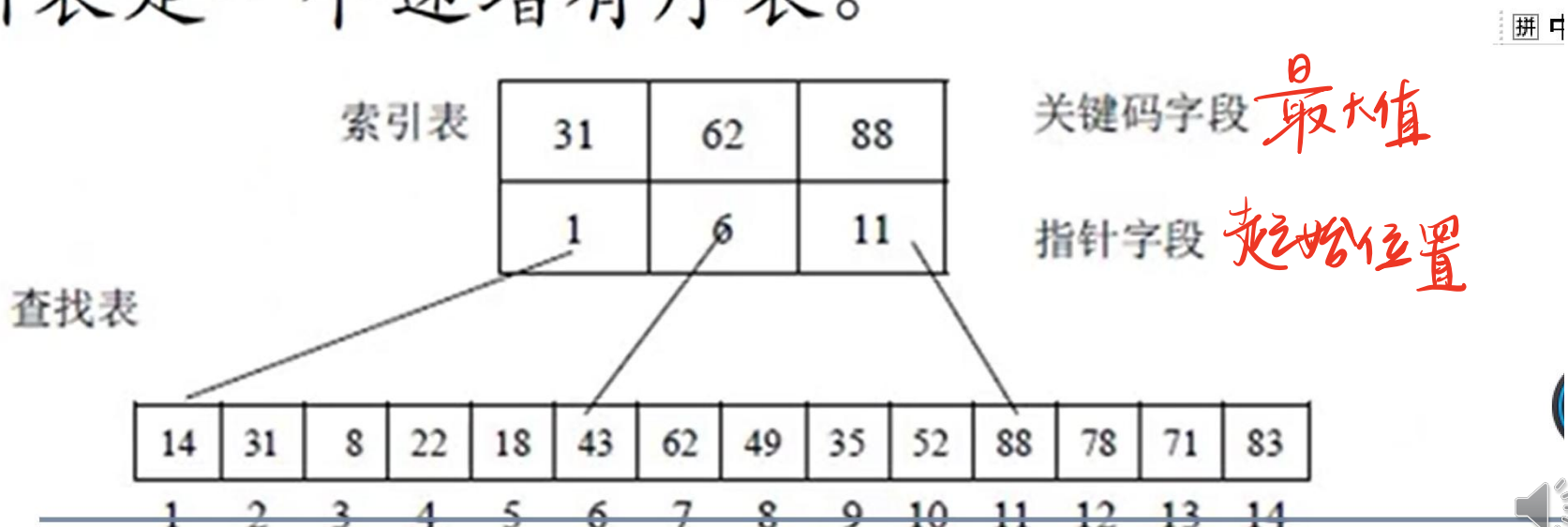
然后, 构建一个有序的索引表

查找时, 先通过索引表查找合适的块, 然后在该块中进行查找



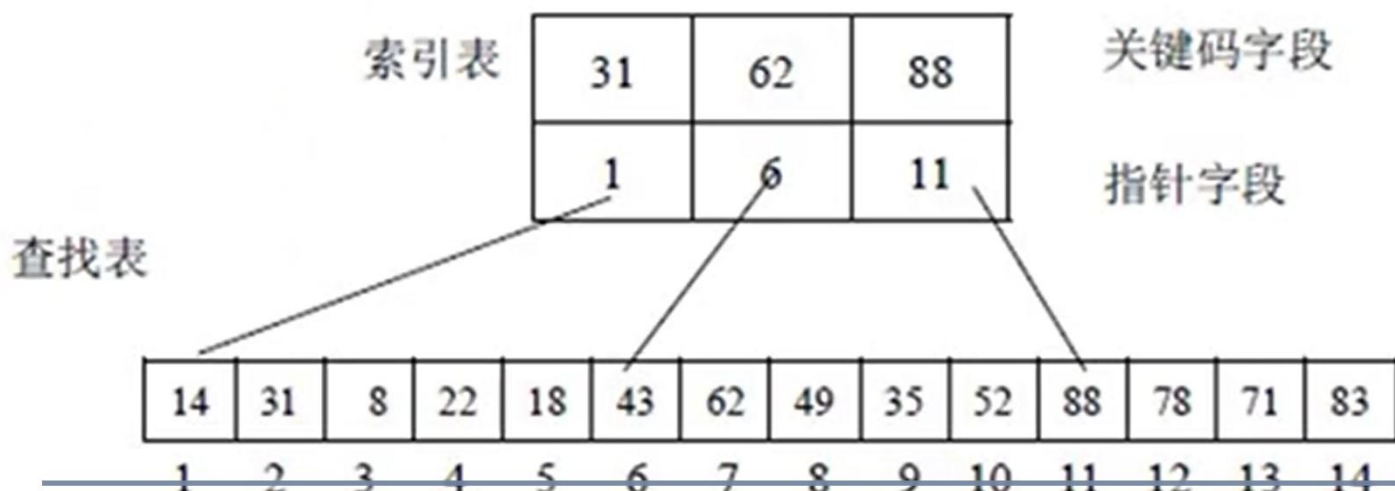
分块查找的存储结构

- 表R分为b块，每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字，即表是“分块有序”的。
- 抽取各块中的最大关键字及其起始位置构成一个索引表，由于表R是分块有序的，所以索引表是一个递增有序表。



分块查找的存储结构

- 表R分为b块，每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字，即表是“分块有序”的。
- 抽取各块中的最大关键字及其起始位置构成一个索引表，由于表R是分块有序的，所以索引表是一个递增有序表。



分块查找的基本过程

1) 首先查找索引表

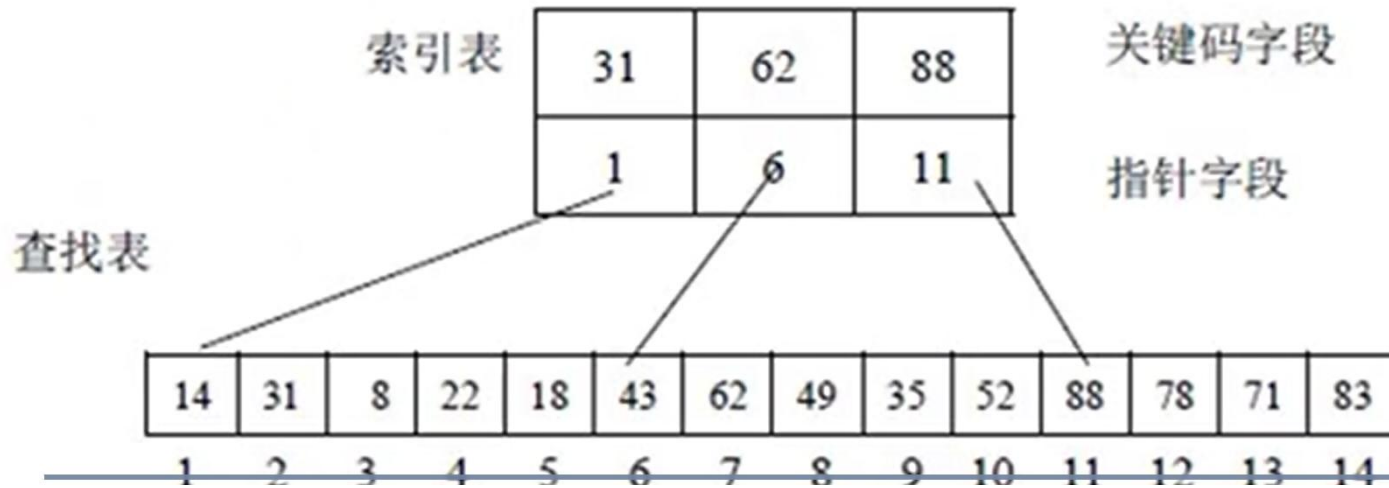
索引表是有序表，可采用二分查找或顺序查找，以确定待查的结点在哪一块。

2) 然后在已确定的块中进行顺序查找

由于块内无序，只能用顺序查找。



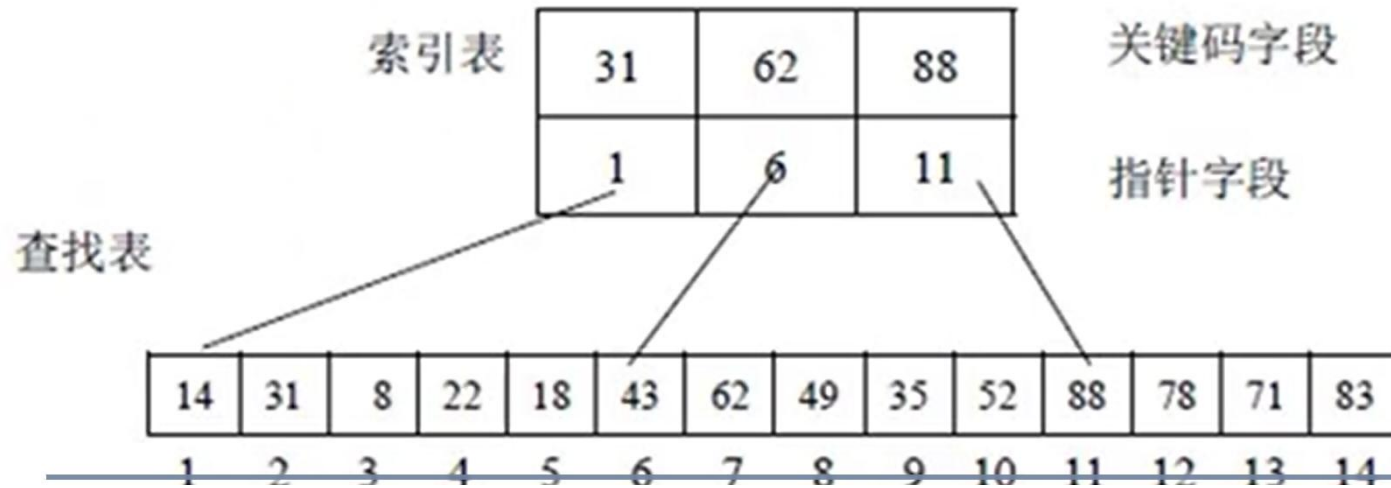
分块查找举例



查找52



分块查找举例



查找63



分块（跳跃）查找的性能分析

时间开销：

查找的时间开销包括索引表的查找和块内查找

设分块查找中将长为 n 的表分成均等的 b 个块，
每块 s 个元素，则 $b = (n / s)$ 上取整

如果索引表中采用顺序查找，则

$$ASL = (n/s + s) / 2 + 1$$

如果索引表中采用折半查找，则

$$ASL = \log_2(n/s + 1) + s/2$$

空间开销：

需要存储索引的辅助数组



分块查找的特点

- 在表中插入或删除一个记录时，只要找到该记录所属的块，就在该块内进行插入和删除运算。
- 因块内记录的存放是任意的，所以插入或删除比较容易，无须移动大量记录。
- 分块查找的主要代价是增加一个辅助数组的存储空间和将初始表分块排序的运算。



本章要点

- 基于线性结构的查找
- 基于树结构的查找
- 基于计算的查找
 - 哈希表（散列表）

哈希表

前面所学的基于线性结构和基于树结构的查找法中查找表的共同**结构特点**：**记录在表中的位置**和它的**关键字之间不存在一个确定的关系**，

查找的过程为给定值依次和关键字集合中各个关键字进行**比较**，**查找的效率**取决于和给定值**进行比较**的关键字个数。用这类方法表示的查找表，**其平均查找长度都不为零**。

不同的表示方法，其差别仅在于：关键字和给定值进行比较的顺序不同。

哈希表

对于频繁使用的查找表，希望 $ASL=1$ 。

只有一个办法：预先知道所查关键字在表中的位置，即，要求：记录在表中位置和其关键字之间存在一种确定的关系。

例如：为每年招收的1000名新生建立一张查找表，其关键字为学号，其值的范围为xx000-xx999（前两位为年份）。

哈希表

例如：为每年招收的1000名新生建立一张查找表，其关键字为学号，其值的范围为xx000-xx999（前两位为年份）。

解决方案：

若以下标为000 ~ 999 的顺序表表示之。

则查找过程可以简单进行：取给定值（学号）的后三位，不需要经过比较便可直接从顺序表中找到待查关键字。

哈希函数

对于动态查找表，

- 1) 表长不确定；
- 2) 在设计查找表时，只知道关键字所属范围，而不知道确切的关键字。

因此在一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以 $f(key)$ 作为关键字为 key 的记录在表中的位置，通常称这个函数 $f(key)$ 为哈希函数。

例如：对于如下9个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dei}

设 哈希函数 $f(\text{key}) =$

$$\lfloor (\text{Ord}(\text{第一个字母}) - \text{Ord}('A') + 1) / 2 \rfloor$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dei		Han		Li		Qian	Sun		Wu	Ye	Zhao

问题：若添加关键字Zhou，怎么办？

能否找到另一个哈希函数？

哈希函数性质

- 1) 哈希函数是一个**映象**，即：将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可；
- 2) 由于哈希函数是一个**压缩映象**，因此，在一般情况下，很容易产生“**冲突**”现象，即： $key1 \neq key2$ ，而 $f(key1) = f(key2)$ 。
- 3) **很难**找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

哈希表的定义

根据设定的**哈希函数 $H(\text{key})$** 和所选中的**处理冲突的方法**，将一组关键字映射到一个有限的、地址连续的地址集（区间）上，并以关键字在地址集中的“象”作为相应记录在表中的**存储位置**，如此构造所得的查找表称之为“**哈希表**”。

构造哈希函数的方法

对数字的关键字可有下列构造方法：

1. 直接定址法

4. 折叠法

2. 数字分析法

5. 除留余数法

3. 平方取中法

6. 随机数法

若是非数字关键字，则需先对其进行数字化处理。

直接定址法

哈希函数为关键字的线性函数

$$H(\text{key}) = \text{key} \quad \text{或者}$$

$$H(\text{key}) = a \times \text{key} + b$$

此法仅适合于：

地址集合的大小 == 关键字集合的大小

数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 (u_1, u_2, \dots, u_s) ，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。

平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于：

关键字中的每一位都有某些数字重复出现频度很高的现象。

折叠法

将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：移位叠加和间界叠加。

此方法适合于：

关键字的数字位数特别多。

除留余数法

设定哈希函数为:

$$H(\text{key}) = \text{key} \text{ MOD } p$$

其中, $p \leq m$ (表长) 并且

p 应为不大于 m 的素数

或是

不含 20 以下的质因子

为什么要对 p 加限制

例如：

给定一组关键字为：12, 39, 18, 24, 33, 21，
若取 $p=9$ ，则他们对应的哈希函数值将为：
3, 3, 0, 6, 6, 3

可见，若 p 中含质因子 3，则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

随机数法

设定哈希函数为：

$$H(\text{key}) = \text{Random}(\text{key})$$

其中，**Random** 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

哈希函数示例(1)

```
int h(int x)
{
    return(x % 16);
}
```

散列函数的返回值只依赖于关键码的最低四位.

哈希函数示例(2)

对于字符串：将字符串所有字母的 ASCII 值累加起来，对 M 取模。

```
int h(char* x) {  
    int i, sum;  
    for (sum=0, i=0; x[i] != '\0'; i++)  
        sum += (int) x[i];  
    return(sum % M);  
}
```

这样设计出来的哈希函数的问题是 x_{12} 和 x_{21} 的哈希函数值会一样

当累加值比 M 大得多时，散列效果很好。

哈希函数示例(3)

ELF Hash: 在UNIX系统V Release 4的ELF (Executable and Linking Format)文件格式用到.

```
int ELFhash(char* key) {  
    unsigned long h = 0;  
    while(*key) {  
        h = (h << 4) + *key++;//使得不同位置的字符权值不同  
        unsigned long g = h & 0xF0000000L;  
        if (g) h ^= g >> 24;  
        h &= ~g;  
    }  
    return h % M;  
}
```

三、处理冲突的方法

“处理冲突”的实际含义是：

为产生冲突的地址寻找下一个哈希地址。

1. 闭散列法（开地址方法）

把冲突记录存储在表中另一个槽中。

“内部问题内部解决”

2. 开散列法（单链方法）

把冲突记录存储在表外。

“内部问题外部解决”

闭散列方法

- 把所有记录直接存储在散列表中
- 每条记录 i 有一个基位置 $h(k_i)$ ，即根据散列函数算出来的槽
- 当要插入一条记录 R 时，如计算出来的基位置已经被另一条记录占据，则根据冲突解决策略来决定把 R 存储在表中的其他槽内；
- 检索方法也相似，先根据关键字值计算基位置，当未找到时再根据冲突解决策略在表中其他相应槽中来寻找。

处理冲突的方法——桶式哈希

- 把散列表分成多个桶，每个桶有若干槽
- 插入记录时计算哈希函数确定记录所在的桶，把记录插入该桶的第一个空闲槽
- 若某个桶全部被占满，则把该记录存储在表后具有无限容量的溢出桶中（该溢出桶为所有桶公用）



散列表	
0	1000
	9530
1	
2	9877
	2007
3	3013
4	9879

溢出桶	
	1057

余数为几插入几号桶

$$H(K) = K \bmod 5$$



桶式哈希的查询

- 当检索某条记录时，首先计算哈希函数确定记录所在的桶，然后在该桶中检索记录。
- 若在该桶中找到记录，则返回成功；
- 若在该桶中没有找到记录，且该桶未满，则返回未查找到该记录；
- 若在该桶中没有找到记录，但该桶已满，则还需检索溢出桶，直到找到记录或者溢出桶的所有记录都已被检索为止。



经典的处理冲突的方法——使用探查序列

为产生冲突的地址 $H(\text{key})$ 求得一个地址序列：

$$H_0, H_1, H_2, \dots, H_s \quad 1 \leq s \leq m-1$$

其中： $H_0 = H(\text{key})$

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

$$i=1, 2, \dots, s$$

冲突次数

对增量 d_i 的三种取法

- 线性探测再散列

$d_i = c \times i$ 最简单的情况 默认值 $c=1$

- 平方探测再散列

$$d_i = 1^2, -1^2, 2^2, -2^2, \dots,$$

- 随机探测再散列

d_i 是一组伪随机数列 或者

$d_i = i \times H_2(\text{key})$ (又称双散列函数探测)

散列表示例

- {30, 40, 47, 42, 50, 17, 63, 12, 6, 62, 27}
- $M = 15$, $h(\text{key}) = \text{key} \% 15$, 若采用线性探查法来解决冲突
- 在理想情况下, 表中的每个空槽都应该有相同的机会接收下一个要插入的记录。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30		47			50					40		42		

比较次数为1

散列表示例

- {30, 40, 47, 42, 50, 17, 63, 12, 6, 62, 27}
- $M = 15$, $h(\text{key}) = \text{key} \% 15$, 若采用线性探查法来解决冲突
- 在理想情况下, 表中的每个空槽都应该有相同的机会接收下一个要插入的记录。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30					50					40				

改进线性探查

- 每次跳过常数 c 个而不是1个槽
 - 探查序列中的第 i 个槽是 $(h(K) + ic) \bmod M$
 - 基位置相邻的记录就不会进入同一个探查序列了
- 探查函数是 $p(K, i) = i * c$
 - 必须使常数 c 与 M 互素

例：改进线性探查

- 例如， $c = 2$ ，要插入关键码 k_1 和 k_2 ， $h(k_1) = 3$ ， $h(k_2) = 5$
- 探查序列
 - k_1 的探查序列是3、5、7、9、...
 - k_2 的探查序列就是5、7、9、...
- k_1 和 k_2 的探查序列还是纠缠在一起，从而导致了聚集

二次探查

- 探查增量序列依次为: $1^2, -1^2, 2^2, -2^2, \dots$, 即地址公式是

$$d_{2i-1} = (d + i^2) \% M$$

$$d_{2i} = (d - i^2) \% M$$

- 探查函数是

$$p(K, 2i-1) = i*i$$

$$p(K, 2i) = -i*i$$

例：二次探查

- 使用一个大小 $M = 13$ 的表
假定对于关键码 k_1 和 k_2 , $h(k_1)=3$, $h(k_2)=2$
- 探查序列
 - k_1 的探查序列是3、4、2、7、...
 - k_2 的探查序列是2、3、1、6、...
- 尽管 k_2 会把 k_1 的基位置作为第2个选择来探查, 但是这两个关键码的探查序列此后就立即分开了

例如：关键字集合

顺序插入

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 $H(\text{key}) = \text{key} \bmod 11$ (表长=11)

若采用线性探测再散列处理冲突 每次+1

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1	比较次数	

例如：关键字集合

$\{ 19, 01, 23, 14, 55, 68, 11, 82, 36 \}$

设定哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$ (表长=11)

若采用线性探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

若采用二次探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

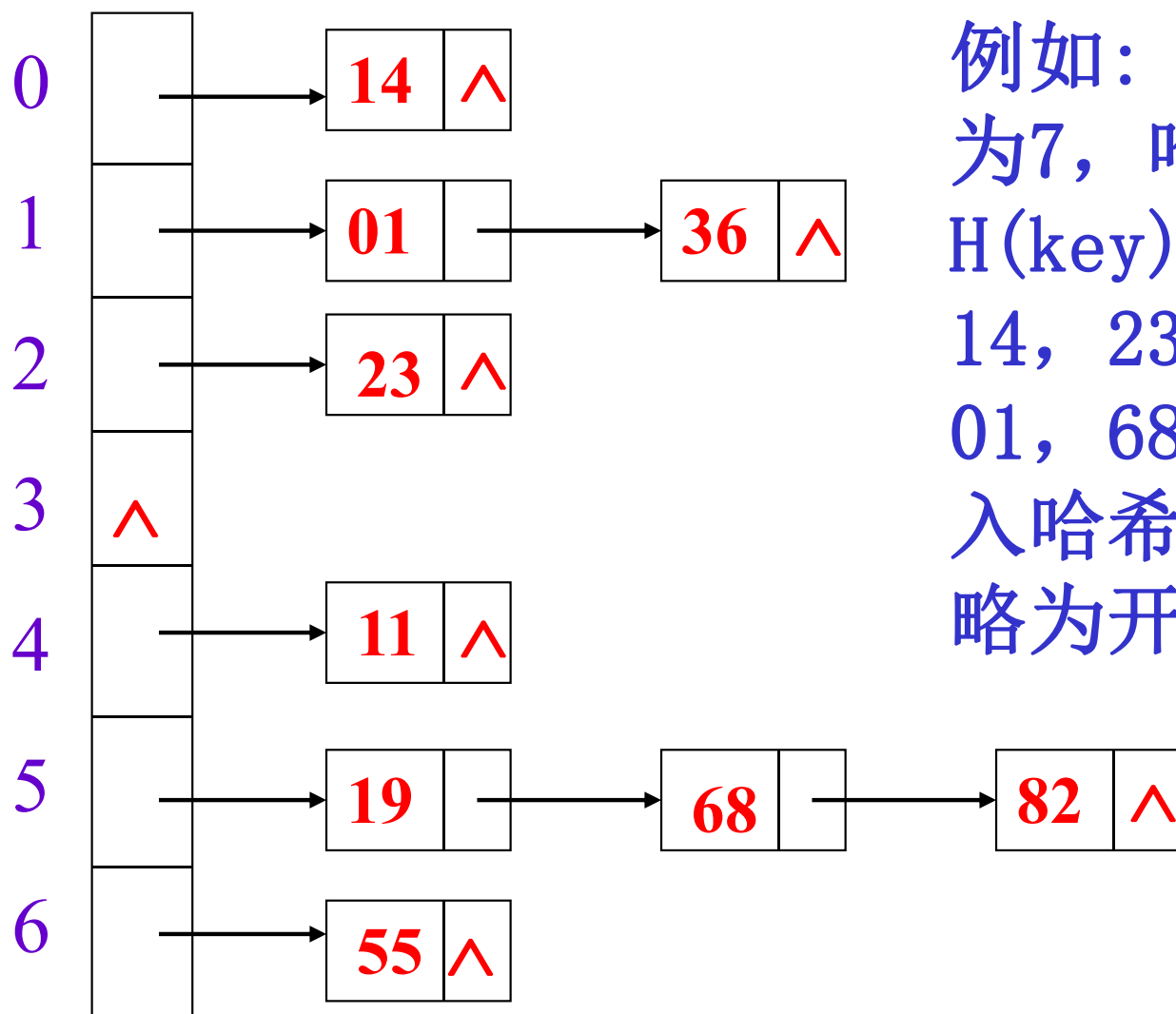
2. 开散列法

将所有哈希地址相同的记录都链接在同一链表中。

例如：已知哈希表大小为7，哈希函数为 $H(\text{key}) = \text{key} \text{ MOD } 7$ ，把14, 23, 19, 55, 11, 01, 68, 36, 82依次插入哈希表，冲突解决策略为开散列方法

将所有哈希地址相同的记录
都链接在同一链表中。

2. 开散列法



例如：已知哈希表大小为7，哈希函数为 $H(\text{key}) = \text{key} \text{ MOD } 7$ ，把 14, 23, 19, 55, 11, 01, 68, 36, 82 依次插入哈希表，冲突解决策略为开散列方法

哈希表的查找

查找过程和造表过程一致。假设采用开散列处理冲突，则查找过程为：

对于给定值 K ，计算哈希地址 $i = H(K)$

若 $r[i] = \text{NULL}$ 则查找不成功

若 $r[i].\text{key} = K$ 则查找成功

否则 “求下一地址 H_i ”，直至

$r[H_i] = \text{NULL}$ (查找不成功)

或 $r[H_i].\text{key} = K$ (查找成功) 为止。

闭散列的插入伪代码

```
// Insert e into hash table HT
template <typename Key, typename E>
bool hashdict<Key, E>::hashInsert(const key& k, const E& e) {
    int home;    // e的基地址
    int pos = home = h(k); // Init
    for (int i=1; EMPTYKEY!= (HT[pos]).key(); i++) {
        pos = (home + p(k,i)) % M;
        Assert(k!= HT[pos].key(), "Duplicates not allowed");
    }
    Kvpair<Key,E> temp(k,e);
    HT[pos] = temp;    // Insert e
}
```

关链码

不允许重复值

闭散列的查找伪代码

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
    int home;          // Home position for K
    int pos = home = h(k); // Initial posit
    for (int i = 1; (k!=( HT[pos].key()) &&
        (EMPTYKEY!=( HT[pos]).key())); i++)
        pos = (home + p(k, i)) % M; // Next
    if (k==(HT[pos]).key()) { // Found it
        return( HT[pos]).value();
    }
    else return NULL; // K not in hash table
}
```

散列方法的效率分析

- 衡量标准：插入、删除和检索操作所需要的记录访问次数
- 散列表的插入和删除操作都是基于检索进行的
 - 删除：必须先找到该记录
 - 插入：必须找到探查序列的尾部，即对这条记录进行一次不成功的检索
 - 对于不考虑删除的情况，是尾部的空槽
 - 对于考虑删除的情况，也要找到尾部，才能确定是否有重复记录

哈希表查找的分析

决定哈希表查找的ASL的因素：

- 选用的**哈希函数**；
- 选用的**处理冲突的方法**；
- 哈希表饱和的程度，**装载因子** $\alpha = n/m$ 值的**大小**
(n —记录数， m —表的长度)

一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论ASL时，可以不考虑它的因素。

因此，哈希表的ASL是**处理冲突方法**和**装载因子**的函数。

哈希表查找的分析

决定哈希表查找的ASL的因素：

- 选用的**哈希函数**；
- 选用的**处理冲突的方法**；
- 哈希表饱和的程度，**装载因子** $\alpha = n/m$ 值的**大小**
(n —记录数， m —表的长度)

一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论ASL时，可以不考虑它的因素。

因此，哈希表的ASL是**处理冲突方法**和**装载因子**的函数。

影响检索的效率的重要因素

■ 散列方法预期的代价与负载因子

$\alpha = N/M$ 有关

- α 较小时，散列表比较空，所插入的记录比较容易插入到其空闲的基地址
 - α 较大时，插入记录很可能要靠冲突解决策略来寻找探查序列中合适的另一个槽
- 随着 α 增加，越来越多的记录有可能放到离其基地址更远的地方

查找成功时有下列结果

线性探测再散列

$$S_{nl} \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

随机探测再散列

$$S_{nr} \approx -\frac{1}{\alpha} \ln(1 - \alpha)$$

链地址法

$$S_{nc} \approx 1 + \frac{\alpha}{2}$$

散列查找小结

在实际应用中，如果需要与检索相关的查找，插入和删除操作的时间复杂度为 $O(1)$ ，散列可以提供平均时间复杂度为 $O(1)$ 的实现方法，尽管在最差情况下散列的时间复杂性仍然是 $O(n)$ 。

散列表的优势就是速度，对于大规模的记录，如果大小不变，经过精心设计散列函数和散列表，可以实现高效的查找

散列方法适合精确查找，不适合范围查找

散列表的记录均匀分布是难点，散列函数的高效性是难点
散列的查找效率是分摊分析，是概率，对一次查找来说，其性能无法保证

