

# 第七章 字典与检索

---

张史梁

slzhang.jdl@pku.edu.cn

# 何为字典与检索

---

- 存储的所有数据——字典
- 重在检索
- 如何提高检索的效率？

# 内容提要

---

- 基本概念
- 字典的顺序表表示
- 字典的散列表示 => 散列函数
- 字典的树型表示 => 二叉排序树、B树、B+树

# 基本概念

---

- 检索是计算机中使用最频繁的任务之一
- 组织与检索信息是大多数计算机应用程序的核心
- 检索表现为：
  - 在一定数据结构（如字典）中查找满足某种条件的结点，即确定一个具有某特定值的元素集合

# 字典

---

- 字典(Dictionary)是一种特殊的集合，其主要的操作为字典中元素的检索。
- 字典是元素的有穷集合，其中每个元素由两部分组成，分别称为元素的“**关键码**”(key)和“**属性**”(attribute)。
  - 例：英汉字典中，每个词条是一个元素，词条中的英文单词可看作是该元素的关键码，词条中对该英文单词的解释可看作是元素的属性。
  - 字典中的两个元素能够根据其**关键码进行比较**，对字典元素的存取、检索也是以关键码为依据进行的。称关键码和属性存在**关联**(Association)。

# 字典的种类

---

- 为了便于字典的维护，有时也要考虑在字典中插入和删除元素的操作
- 静态(static)字典：字典一经建立就基本固定不变，主要的操作就是字典元素的检索
  - 为静态字典选择存储方法主要需考虑检索效率、检索运算的简单性
- 动态(dynamic)字典：经常需要改动的字典
  - 对于动态字典，存储方法的选择不仅要考虑检索效率，还要考虑字典元素的插入、删除运算是否简便。

# 检索

---

- 检索：给定一个key，在字典中找出关键码等于key的元素
  - 如果找到，则检索成功
  - 否则检索失败
- 两类检索
  - 精确匹配查询（exact-matching query）：检索关键码值匹配某个特定值的记录
  - 范围查询（range query）：检索关键码值在某个指定值范围内的所有记录

# 检索算法的分类

---

- 根据被检索数据的组织方式不同，检索算法可分为：
  - 线性表方法=> 顺序检索、二分法检索
  - 散列法（根据关键码值直接访问法）
  - 树索引法=> 二叉排序树、字符树、B树等
  - 基于属性的检索=> 倒排表



# 检索算法的效率

---

- 衡量一个检索算法效率的主要标准是检索过程中和关键码的平均比较次数，即**平均检索长度ASL** (Average Search Length)，定义为：

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

- $n$ 是字典中元素的个数。
- $p_i$ 是查找第 $i$ 个元素的概率。若不特别声明，一般认为每个元素的检索概率相等，即 $p_i=1/n$
- $c_i$ 是找到第 $i$ 个元素的比较次数。
- 好坏标准还考虑算法的空间开销，以及算法是否易于理解等因素。

# 本章中的假定

---

□ 在本章的讨论中，假设

- 字典元素类型相同

- 关键码为数值类型（例如正整数），

因此，可以将字典元素按关键码排序

# 内容提要

---

- 基本概念
- 字典的顺序表表示
  - 顺序检索
  - 二分检索
  - 分块检索
- 字典的散列表示 => 散列函数
- 字典的树型表示 => 二叉排序树、B树、B+树

# 字典的顺序表示定义

---

//为了描述简单，将KeyType and DataType 定义为int类型

```
typedef int      KeyType;
```

```
typedef int      DataType;
```

```
typedef struct {
```

```
    KeyType key;           /* 字典元素的关键码字段*/
```

```
    DataType other;       /* 字典元素的属性字段*/
```

```
}DicElement;
```

```
typedef struct {
```

```
    DicElement element[MAXNUM];
```

```
    int n;                /* 为字典中元素的个数 */
```

```
} SeqDictionary;
```

# 顺序检索算法

---

## □ 算法结束时返回检索成功或失败信息

- 若检索成功，则参数position指向找到的元素位置
- 否则，position指向应插入元素的位置

## □ `int SeqSearch( SeqDictionary *pdic, KeyType key, int *pos)`

*/\*在字典中顺序检索关键码为key的元素\*/*

```
{ int i;
  for ( i = 0; i < pdic->n; i++)          /* 从头开始 */
  {
    if (pdic->element[i].key == key) /* 成功 */
    { *pos = i;      return TRUE;    }
  }
  *pos = i;  return FALSE;           /* 失败 */
}
```

# 顺序检索的平均检索长度

---

- 若找到的是第 $i$ 个元素，则比较次数为 $c_i=i$ 。因此，

$$ASL=1 \times P_1+2 \times P_2+\dots+n \times P_n$$

- 假设每个元素的检索概率相等，即 $P_i=1/n$ ，则平均检索长度为：

$$ASL = \sum_{i=1}^n p_i c_i = \sum_{i=1}^n i / n = (n+1) / 2$$

- 因此，成功检索的平均比较次数约为字典长度的一半；若字典中不存在关键码为key的元素，则需进行 $n$ 次比较。
- 总之，顺序检索的平均检索时间为 $ASL=O(n)$

# 顺序检索的特点

---

## □ 顺序检索优点：

算法简单，适应面广，无论字典中元素是否有序均可使用，插入效率很高。

## □ 顺序检索缺点：

平均检索长度较大，特别是当 $n$ 很大时，检索效率较低。

# 顺序检索的改进

---

□ 在不等概率情况下，

- 当 $P_1 \geq P_2 \dots \geq P_n$ 时，顺序检索需要使ASL最小，应该保持概率最大的元素在最前面，概率最小的元素在最后面。
- 通常，无法预先知道各个元素的查找概率
- 解决的办法是为用检索成功次数代替查找概率，当检索元素成功时，其检索成功次数加1，保持检索成功次数最大的元素在前面，检索成功次数最小的元素在最后面。

针对上述方法的改进：使用堆来组织字典



# 二分法检索

---

- 二分法检索（折半检索）要求字典元素已按关键码排序。
- 基本思想：
  - 首先将字典中间位置上元素的关键码和给定值key比较，如果相等，则检索成功；
  - 否则，若大于key，则在字典前半部分中继续进行二分法检索；否则，在字典后半部分中继续进行二分法检索。
- 折半检索的实质是逐步缩小查找区间的查找方法。

# 二分法检索

- 分别用low和high表示当前查找区间的下界和上界
- 例如：在下列待检索字典中检索关键码为25和78的元素

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99
05,	10,	18,	25,	27	↑	41,	51,	68,	73,	99
		↑	25,	27				↑	73,	99
			↑					↑	99	
			成功						失败	

# 二分法检索

```
① int binarySearch(SeqDictionary * pdic, KeyType key, int *position)
② {
③     int low, mid, high;
④     low=0; high=pdic->n-1;
⑤     while( _____ )
⑥     {
⑦         mid= _____ ; /* 当前检索的中间位置 */
⑧         if (pdic->element[mid].key==key) /* 检索成功 */
⑨         {
⑩             *position=mid;    return(1);
⑪         }
⑫         else if ( _____ ) high=mid-1;
⑬         else low=mid+1; /* 要检索的元素在右半区 */
⑭     }
⑮     *position=low;
⑯     return(0); /* 检索失败 */
⑰ }
```

# 二分法检索

---

```
① int binarySearch(SeqDictionary * pdic, KeyType key, int *position)
② {
③     int low, mid, high;
④     low=0; high=pdic->n-1;
⑤     while(low<=high)
⑥     {
⑦         mid=(low+high)/2;          /* 当前检索的中间位置 */
⑧         if (pdic->element[mid].key==key) /* 检索成功 */
⑨         {
⑩             *position=mid;    return(1);
⑪         }
⑫         else if (pdic->element[mid].key>key) high=mid-1;
⑬         else      low=mid+1;    /* 要检索的元素在右半区 */
⑭     }
⑮     *position=low;
⑯     return(0);    /* 检索失败 */
⑰ }
```

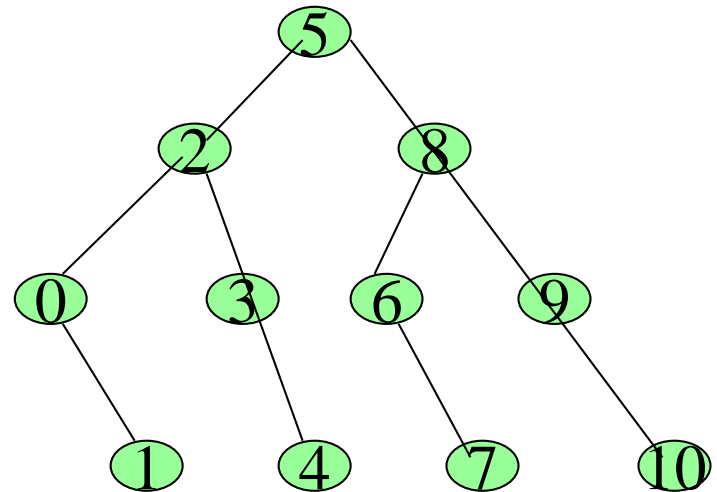
# 二分检索算法的时间复杂性分析

- 每比较一次缩小一半的查找区间。查找过程可用二叉树来描述。树中结点数字表示结点在有序表中的位置，通常称这个描述查找过程的二叉树为判定树。

0    1    2    3    4    5    6    7    8    9    10

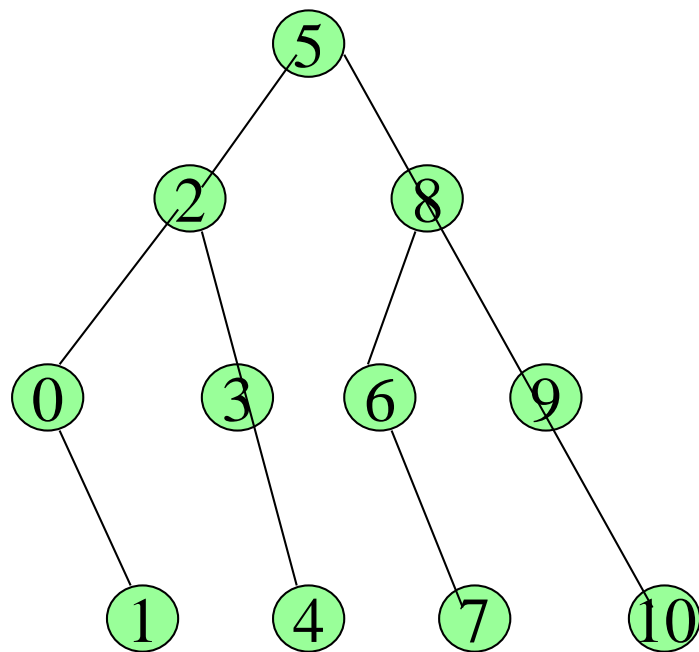
05, 10, 18, 25, 27, 32, 41, 51, 68, 73, 99

右图为11个记录的判定树。如要检索index为3、9的记录需要3次比较；检索1、4、7、10的记录需要4次比较



# 二分检索算法的时间复杂性分析

- ❑ 折半检索的过程恰好是在判定树中走了一条从根到检索结点的路径
- ❑ 比较的关键码个数恰为该结点在二叉树中的层数加1。因此，成功折半检索关键码比较次数不会超过树的深度加1



折半检索的ASL等于多少？

# 二分检索算法的时间复杂性分析

---

假定记录数 $n=2^h-1$ ，即 $h=\log_2(n+1)$ ，则描述折半检索的判定树是一棵深度为 $h-1$ 的满二叉树。在该满二叉判定树中，有：

层数为0的结点个数为 $2^0$ ， 比较1次；

层数为1的结点个数为 $2^1$ ， 比较2次；

.....

层数为 $k$ 的结点个数为 $2^k$ ， 比较 $k+1$ 次；

.....

层数为 $h-1$ 的结点个数为 $2^{h-1}$ ， 比较 $h$ 次；

等概率检索下， 检索成功时：

$$\begin{aligned}
ASL &= \sum_{i=1}^n p_i c_i \\
&= \frac{1}{n} \sum_{i=1}^h i \times 2^{i-1} \\
&= \frac{1}{n} \times (1 + 2 + 2^2 + \cdots + 2^{h-1} \\
&\quad + 2 + 2^2 + \cdots + 2^{h-1} \\
&\quad + 2^2 + \cdots + 2^{h-1} \\
&\quad + 2^{h-1}) \\
&= \frac{1}{n} \times \sum_{i=1}^h \sum_{m=i}^h 2^{m-1} \\
&= \frac{1}{n} \times \sum_{i=1}^h (2^h - 2^{i-1})
\end{aligned}$$

$$\begin{aligned}
&= \frac{1}{n} \times (h \times 2^h - \sum_{i=1}^h 2^{i-1}) \\
&= \frac{1}{n} \times (h \times 2^h - 2^h + 1) \\
&= \frac{1}{n} \times (h \times (n+1) - (n+1) + 1) \\
&= \frac{1}{n} \times (h \times (n+1) - n) \\
&= \frac{n+1}{n} \times h - 1 \\
&= \frac{n+1}{n} \log_2(n+1) - 1
\end{aligned}$$

- 当n很大时, 如n>100, 则得到:  
 $ASL \approx \log_2(n+1) - 1$



# 二分法检索特点

---

## □ 二分法检索的优点

- 折半检索的效率要高于顺序检索。如 $n=1000$ ，顺序 $ASL=500$ ，而折半 $ASL \approx 9$ 。
- 比较次数少，检索速度快。

## □ 二分法检索缺点

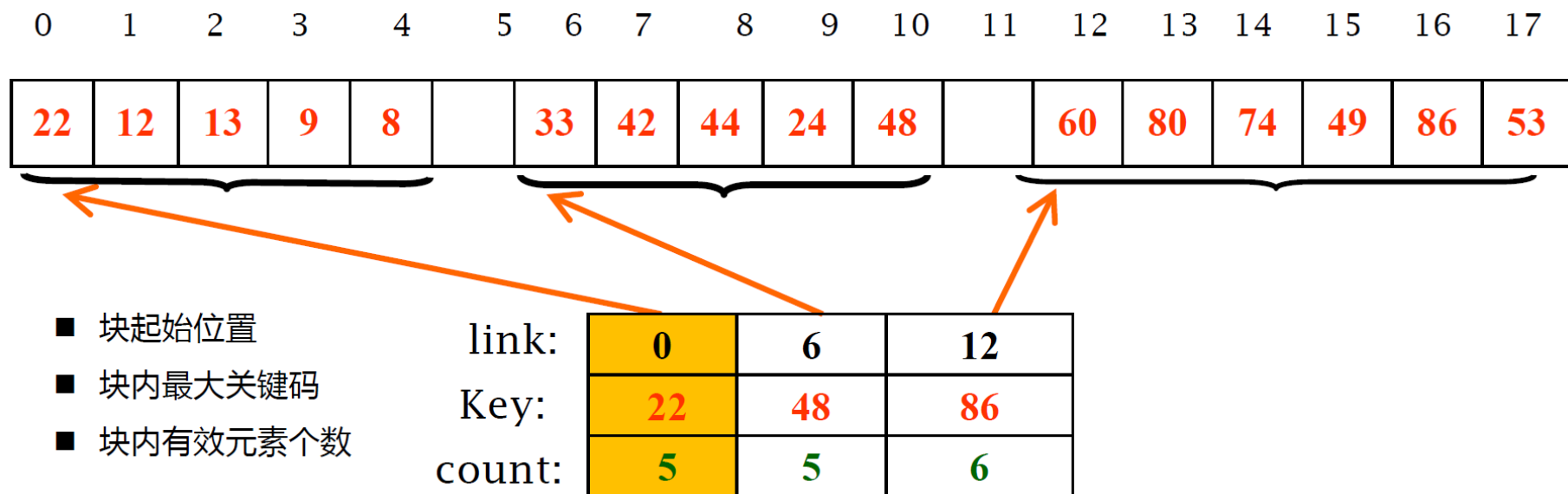
- 要将字典按关键码排序，且只适用于顺序存储结构。
- 插入移动元素较多
- 通过给定值与关键码的比较才能确定记录位置，检索效率与检索过程中进行的比较次数有关。

# 分块检索思想

---

- “按块有序” 设线性表中共有 $n$  个数据元素，将表分成 $b$  块
  - 前一块最大关键码必须小于后一块最小关键码
  - 每一块中的关键码不一定有序
  
- 顺序与二分法的折衷
  - 既有较快的检索
  - 又有较灵活的更改

# 分块检索—顺序索引结构



# 分块检索性能分析

---

- 分块检索为**两级检索**：先在索引表中确定待查元素所在的块， $ASL_b$
- 然后在块内检索待查的元素， $ASL_w$

$$\begin{aligned} ASL &= ASL_b + ASL_w \\ &\approx \log_2(b+1) - 1 + (s+1)/2 \\ &\approx \log_2(1+n/s) + s/2 \end{aligned}$$

# 分块检索性能分析

- 假设在索引表中用顺序检索，在块内也用顺序检索

$$ASL_b = \frac{b+1}{2}$$

$$ASL_w = \frac{s+1}{2}$$

$$\begin{aligned} ASL &= \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1 \\ &= \frac{n+s^2}{2s} + 1 \end{aligned}$$

- 当  $s = \sqrt{n}$  时，ASL 取最小值

$$ASL = \sqrt{n} + 1 \approx \sqrt{n}$$

# 分块检索性能分析

---

- 当 $n=10,000$  时顺序检索5,000 次
- 二分法检索14 次
- 分块检索100 次


# 分块检索优缺点

## □ 优点：

- 插入、删除相对较易
- 没有大量记录移动

## □ 缺点：

- 增加一个辅助数组的存储空间
- 初始线性表分块排序
- 当大量插入/删除时，或结点分布不均匀时，速度下降



link:	0	6	12
Key:	22	48	86
count:	5	5	6

# 内容提要

---

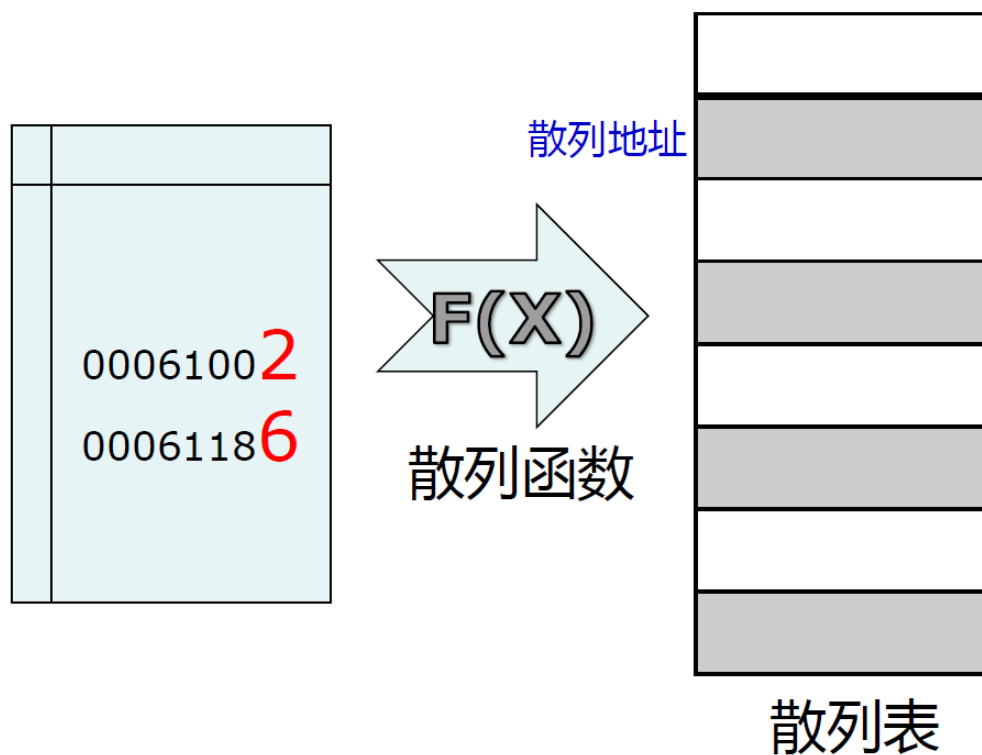
- 基本概念
- 字典的顺序表表示
- 字典的散列表示
  - 散列表
  - 散列函数
  - 存储表示与碰撞的处理
- 字典的索引和树型表示



# 散列存储简介

## □ 组织数据的方式——存储结构

- 顺序存储
- 链式存储
- 散列存储
- 索引存储



# 散列与散列表

---

## □ 散列法(hashing)/杂凑法的基本思想：

- 直接根据记录的内容得到其存储位置
- 设一个字典元素，若其关键码为key，按一个确定的散列函数 $h$ 计算 $h(\text{key})$ ，并把 $h(\text{key})$ 作为该元素的存储地址，即散列地址。
- 检索时，仍需要根据 $h(\text{key})$ 得到关键码所在元素的存储地址。

## □ 散列表/Hash表：用散列法表示的字典。

# 散列表例子

- 假设要建立一张全国**30**个地区的各民族人口统计表，每个地区为一个记录，记录的各数据项为：

编号	地区名	总人口	汉族	回族	...
1	BEIJING				
2	TIANJIN				
...	...				

用一维数组C[30]来存放这张表，其中C[i]是编号为i的地区的人口情况。编号i为关键字，唯一确定了记录的存储位置。相当于 $h(key)=key$ 。

# 散列表例子

- 假设以地区名作关键字，地区名以汉语拼音的字符表示，
  - 取关键字中第一个字母在字母表中的序号作为散列函数。如 BEIJING 的散列数值为 2；
  - 取关键字的第一个和最后一个字母在字母表中的序号之和为散列函数。判别这个和值，若比 30（表长）大，则减去 30。

key	BEIJING 北京	TIANJIN 天津	HEBEI 河北	SHAXNXI 山西	SHANGHAI 上海	SHANDONG 山东	HENAN 河南	SICHUAN 四川
h1(key)	02	20	08	19	19	19	08	19
h2(key)	09	04	17	28	28	26	22	03

# 散列相关概念

- ❑ **碰撞 (conflict)** : 如果两个不相等的关键词key1和key2, 用散列函数h得到相同的散列地址  $h(\text{key1}) = h(\text{key2})$ , 这种现象称为碰撞。
- ❑ 发生碰撞的两个(或多个)关键词称为**同义词**.
- ❑  $h(\text{key})$ 的值域所对应的地址空间称为**基本区域**.
- ❑ 发生碰撞时, 同义词可以存放在基本区域中未被占用的单元, 也可以放到基本区域以外另开辟的区域 (称**溢出区**)

$$\text{负载因子}(\alpha) = \frac{\text{散列表中结点数目}}{\text{基本区域能容纳的结点数}}$$

- ❑ 当  $\alpha > 1$  时碰撞不可避免。

# 散列表的构造和检索

---

- 散列表的构造和检索中，需要解决的主要问题包括：
- 定义散列表和散列函数
  - 散列函数：散列表如何检索？
  - 检索效率如何？检索效率与哪些因素有关？
- 解决碰撞
  - 对于任意的散列函数，都可能出现“碰撞”现象
  - 碰撞发生时如何处理？

# 常用的散列函数

---

- 直接定址法
- 数字分析法
- 平方取中法
- 折叠法
- 除留余数法
- 基数转换法

## 散列函数的选择标准

- 使得任何关键字的散列函数值落在表长的范围内。
- 对任一个关键码，经散列函数映象到散列表任何一个地址的概率是相等的（关键字的散列地址均匀分布在整个地址空间，从而减少碰撞）；
- 散列函数尽可能简单，提高计算速度

# 直接定址法

- 取**关键字**或关键字的某个线性函数值为散列地址
  - $h(\text{key}) = a * \text{key} + b$  (a和b为常数)
  - 适用于**关键码分布基本连续**情况。若关键码分布不连续，容易造成空单元，存储空间浪费。

1~100岁人口数字统计表 (a=1,b=0, key=年龄)

地址	01	02	03	...	25	26	27	...	100
年龄	1	2	3	...	25	26	27	...	100
人数	3000	2000	5000	...	1050	...	...	...	...



# 直接定址法

- 取关键字或关键字的某个线性函数值为散列地址
  - $h(\text{key}) = a * \text{key} + b$  (a和b为常数)
  - 适用于关键码分布基本连续情况。若关键码分布不连续，容易造成空单元，存储空间浪费。

解放后人口调查

地址	01	02	03	...	22
年份	1949	1950	1951	...	1970
人数					15000

# 数字分析法

---

- 关键码数位比存储区的地址码位数多，这时可以对各位进行分析，丢掉分布不均匀的位而留下均匀的位作为地址。
  - 例如：(395003, 395010, 395012, 395085, 395097),
  - 前四位相同，后两位随机分布，故可以取后两位构成散列函数，得到的散列地址为（03，10，12，85，97）
  
- 通常用于已知记录关键码，并且关键码各位分布已经知道的情况，适合于静态的字典。

# 平方取中法

---

- 先求出关键码的平方，然后取中间若干位构成散列函数。
  - 例如：  $\text{key} = 4731$ ，  $\text{key}^2 = 22382361$ ，如地址码为3位，则可以取382为散列地址，即  $h(4731) = 382$
- 这是一种较常用的构造散列函数的方法，一个数在平方后的中间几位数和每一位都相关，由此使随机分布的关键字得到的散列地址也是随机的。
- 取的位数由表长决定。

# 折叠法

- 如果关键码的位数多于地址位数，且关键码中每一位分布均匀时，此时可以将关键码分成若干部分
- 其中一部分的长度等于地址位数。各部分相加，舍弃进位，最后的和作为散列地址。
- 如：key = 0582422241，地址位数为4。

$$\begin{array}{r} 2241 \\ 8242 \\ +) \quad 05 \\ \hline [1]0488 \end{array}$$

移位相加

$$\begin{array}{r} 2241 \\ 2428 \\ +) \quad 05 \\ \hline 4674 \end{array}$$

间界叠加

# 除留余数法

---

□  $h(\text{key}) = \text{key} \% p$

- $p$ 的选择非常重要，通常选小于等于散列表长度 $m$ 的符合一定要求的最大数（比如要求是素数）。
- 如：

m=	8	16	32	64	128	256	512	1024
p=	7	13	31	61	127	251	503	1019

- 除留余数法计算简单，许多情况下效果较好，是一种常用的散列函数。

# 基数转换法

---

□ 将关键码首先看作是另一进制的表示，然后再转换为原来的进制数，用数字分析法取若干位作为散列地址。一般转换基数大于原基数，且最好二者互素。

■ 例如：  $\text{key} = (236075)_{10}$ ，把它看作13进制数 $(236075)_{13}$ ，

■ 再转换为10进制，地址位数为4：

$$(236075)_{13} = 2 \times 13^5 + 3 \times 13^4 + 6 \times 13^3 + 7 \times 13^2 + 5 = (841547)_{10}$$

■ 选择2~5，得到散列地址 $h(236075) = 4154$

# 选取散列函数的因素

---

## □ 考虑的因素：

- ① 计算散列函数所需时间
- ② 关键字的长度
- ③ 散列表的大小
- ④ 关键字的分布情况
- ⑤ 记录的查找频率

□ 无论选择何种散列函数，碰撞都可能发生。换句话说讲，合适的散列函数可以减少碰撞发生的几率，但不能保证不发生碰撞。

□ 碰撞发生时如何处理？

# 存储表示与碰撞的处理

---

## □ 开地址法（探查法）

- 线性探查法
- 双散列函数法

## □ 拉链法



# 开地址法

---

## □ 基本思想：

- 当碰撞发生时，用某种方法在基本区域内形成一个探查序列，沿着探查序列逐个单元查找，直到找到该元素或碰到一个未被占用的地址为止
- 若插入元素，则碰到空的地址单元就存放要插入的同义词。若检索元素，则碰到空的地址单元说明表中没有待查的元素

## □ 形成探查序列的方法有多种，下面介绍两种探查方法：

- 线性探查法
- 双散列函数法

# 线性探查法

---

- 将基本存储区看作一个循环表。
  - 若在地址为 $d$  ( $d=h(\text{key})$ ) 的单元发生碰撞, 则依次探查下述地址单元:  $d+1, d+2, \dots, m-1, 0, 1, \dots, d-1$  ( $m$ 为基本存储区的长度), 直到找到一个空单元 或 查找到关键码为 $\text{key}$ 的元素为止。
  - 如果从单元 $d$ 开始探查, 查找一遍后, 又回到地址 $d$ , 则表示基本存储区已经溢出。

# 示例

---

设散列表用数组element表示，关键码集合

$K = \{18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25\}$ ,

用线性探查法解决碰撞：

①  $m=13$ ，散列函数为 $h(key)=key\%13$ ；

$h(18)=5$ ,  $h(73)=8$ ,  $h(10)=10$ ,  $h(05)=5$ ,  $h(68)=3$ ,

$h(99)=8$ ,  $h(27)=1$ ,  $h(41)=2$ ,  $h(51)=12$ ,  $h(32)=6$ ,  $h(25)=12$

② 计算散列地址 $d=key\%13$ ，若地址未被占用，则插入新结点；否则进行线性探查。

# 示例(续)

$h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3,$

$h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12$

- 插入18, 73, 10时, 地址都未被占用, 可以直接存放;
- 插入05时, 其地址与18的地址发生碰撞, 进行线性探查, 由于 `element[6]` 为空单元, 可以插入;
- 68的地址为3, `element[3]` 是空单元, 直接插入;
- 99的地址为8, 与73的地址相冲突, 线性探查后放入 `element[9]`;
- 27, 41, 51的地址都未被占用, 可以直接存放;
- 32的地址为6, `element[6]` 已经被5占用, 线性探查后存入 `element[7]`;
- 25的地址为12, 与51发生冲突, 线性探查后存入 `element[0]`

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	25	27	41	68		18	5	32	73	99	10		51

# 散列表的运算

---

## □ 散列表表示的字典定义：

```
typedef struct
```

```
{
```

```
    DicElement element[REGION_LEN];
```

```
    int m; /*m=REGION_LEN,为基本区域长度*/
```

```
}HashDictionary;
```

## □ 散列表的检索算法

## □ 散列表的插入算法

```

int linearSearch(HashDictionary * phash, KeyType key, int *position)
{
    /* 检索给定key，成功返回key位置，失败则返回插入位置 */
    int d, inc;
    d=h(key); /* d为散列地址，散列函数为h(key) */
    for(inc=0; inc<phash->m; inc++)
    {
        if(phash->element[d].key==key)
        {
            *position=d; /* 检索成功 */
            return(TRUE);
        }
        else if(phash->element[d].key==NULL)
        {
            *position=d; /* 检索失败，找到插入位置 */
            return(FALSE);
        }
        d=(d+1)%phash->m;
    }
    *position=-1; /* 散列表溢出 */
    return(FALSE);
}

```

*/\*用线性探查法解决碰撞\*/*

```
int linearInsert(HashDictionary *phash, KeyType key)
{
    int position;
    if(linearSearch(phash, key, &position) ==TRUE )
        /* 散列表中已有关键码为key 的结点 */
        printf("Find\n");
    else if(position!=-1)
        phash->element[position].key=key;
        /* 插入结点,忽略对value字段的赋值 */
    else return(FALSE); /* 散列表溢出 */
    return(TRUE);
}
```

# 堆积

---

- 线性探测容易出现不同的关键码争夺同一个散列地址问题，这种现象称为“**二次聚集[堆积]**”。
  - 即**不是同义词的结点**处在同一个探查序列中，增加了探查序列的长度；
  - 在上例中， $h(32)=6$ ， $h(5)=5$ ，32和5不是同义词，但是在解决05与18的碰撞时，05已经存入`element[6]`，使得在插入32时，32与05不冲突的**两个非同义词之间发生了碰撞**
- 为了减少堆积的产生，可以改进线性探查方法，使探查**顺序跳跃式地散列在表中**。



# 存储表示与碰撞的处理

---

## □ 开地址法（探查法）

- 线性探查法
- 双散列函数法

## □ 拉链法

# 双散列函数法

---

- 双散列函数法中选用两个散列函数 $h_1$ 和 $h_2$ 
  - $h_1$ 以关键码为自变量，产生一个0到 $m-1$ 之间的数作为地址。
  - $h_2$ 以关键码为自变量，产生一个1到 $m-1$ 之间的并和 $m$ 互素的数作为对地址的补偿。
    - 例如， $h_1(\text{key})=\text{key}\%m$ ， $h_2(\text{key})=\text{key}\%(m-2)+1$ 。
    - 如果 $d=h_1(\text{key})$ 发生碰撞，则再计算 $h_2(\text{key})$ ，
    - 取 $(d+h_2(\text{key}))\%m$ ， $(d+2h_2(\text{key}))\%m$ ，... 直到找到未被占用的地址插入同义词为止。

# 示例

---

- 已知关键码集合 $K=\{18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25\}$ ，用双散列函数法解决碰撞
  - $m=13$ ， $h1$ 、 $h2$ 函数分别为：  
 $h1(key)=key\%13$ ，  
 $h2(key)=key\%(m-2)+1=key\%11+1$
  - 计算地址 $d=h1(key)$ ，若地址未被占用，则插入新结点；否则用 $h2$ 函数计算地址补偿，直到找到未被占用的地址。

$h1(18)=5$ ,  $h1(73)=8$ ,  $h1(10)=10$ ,  $h1(05)=5$ ,  $h1(68)=3$ ,  
 $h1(99)=8$ ,  $h1(27)=1$ ,  $h1(41)=2$ ,  $h1(51)=12$ ,  $h1(32)=6$ ,  $h1(25)=12$

# 示例-续

---

- 05与18冲突，用h2函数对地址进行补偿，  
 $h2(05)=5\%11+1=6$ ，得到散列地址为  
 $(d+6)\%m=(5+6)\%13=11$ ，是空地址单元可以直接插入
- 同样可以得到99的插入地址为9，25的地址为7。
- 存放后的散列表为：

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	32	25	73	99	10	5	51

# 开地址法特点

---

## □ 逻辑删除处理：

- 用开地址法构造散列表，删除结点时，不能简单地将要删除的结点空间置为空，因为各种开地址法中，**空地址单元是检索失败的依据，删除一个结点会影响其它结点的检索，因此**只能在被删除结点上做标记，不能真正删除。
- 这样，做了许多删除后，形式上散列表是满的，但实际上却很空，浪费了存储空间。

## □ 改进的方法：在插入结点时利用做了标记的地址空间。

# 存储表示与碰撞的处理

---

## □ 开地址法

- 线性探查法
- 双散列函数法

## □ 拉链法

# 拉链法

---

- 设基本区域长度为 $m$ ，碰撞发生时建立一个链表，若 $n$ 个关键码映象到基本区域的 $m$ 个存储单元上，则最多可以建立 $m$ 个链表。
- 例如：
  - 已知关键码集合 $K=\{18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25\}$ ，用拉链法解决碰撞。
  - 设散列函数为 $h(\text{key})=\text{key}\%13$ ，插入新结点时，将结点插入到链表中

□  $h(18)=5$ ,  $h(73)=8$ ,  $h(10)=10$ ,  $h(05)=5$ ,  $h(68)=3$ ,  $h(99)=8$ ,  
 $h(27)=1$ ,  $h(41)=2$ ,  $h(51)=12$ ,  $h(32)=6$ ,  $h(25)=12$

□ 得到的散列表为：

地址	关键码	拉链		
0				
1	27	∧		
2	41	∧		
3	68	∧		
4				
5	18	→ <table><tr><td>5</td><td>∧</td></tr></table>	5	∧
5	∧			
6	32			
7				
8	73	→ <table><tr><td>99</td><td>∧</td></tr></table>	99	∧
99	∧			
9				
10	10	∧		
11				
12	51	→ <table><tr><td>12</td><td>∧</td></tr></table>	12	∧
12	∧			

散列表的存储表示  
需要另外定义



# 拉链法特点

---

## □ 优点：

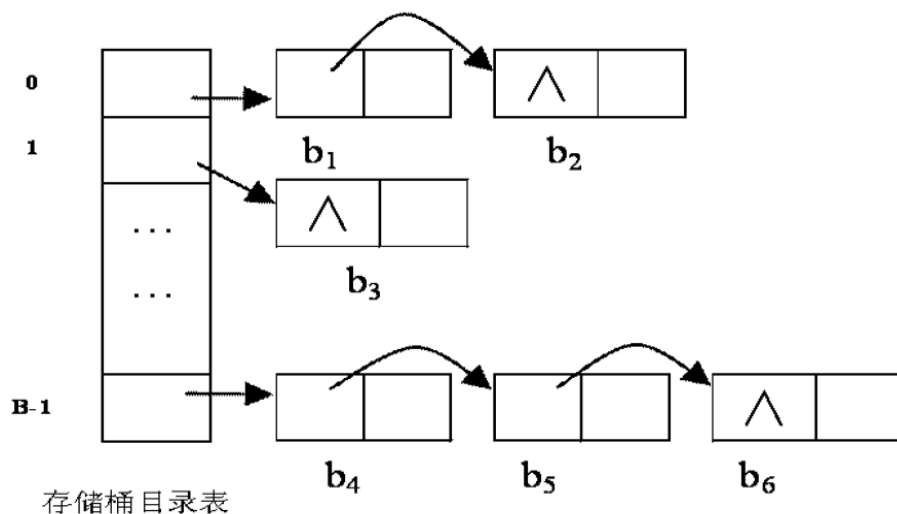
- 设 $n$ 个关键码通过散列函数对应到 $m$ 个单元中，每个同义词子表的平均长度为 $n/m$ ，检索的速度较快；
- 由于各链表上的结点空间是动态申请的，因此更适应于构造表前无法确定表长的情况；
- 拉链法不会造成堆积现象，结点的删除操作较方便。

## □ 缺点：

- 指针的开销；负载因子越大，开地址法所需的检索长度越长

# 桶散列

- 有时把存放散列函数值相同的记录（同义词）的结构称为**桶**，则整个散列结构由一个**指向桶的指针数组**和**若干个桶**组成，称为**桶散列**。
- 下图表示了一个具有B个存储桶的散列文件组织
  - 如果B很小，存储桶目录表可放在内存
  - 如果B较大，要存放好多页块，则存储桶目录表就放到外存上



# 总结-散列表的建立分析

---

- 对于给定关键码值key，计算 $H(key)$ ，得到散列地址。
- 如果该散列地址对应的单元为空，表明没有冲突，直接将记录插入；
- 否则如果该散列地址对应的单元非空，表明该单元已经被前面的记录占用，出现碰撞问题。
  - 如果采用开放地址法解决碰撞，则按照一定的序列寻找下一个空单元；
  - 如果是采用拉练法解决碰撞，则在碰撞地址对应的同义词链表中（桶）完成插入。

# 总结-散列表的检索分析

---

- 对于散列表的检索，如果该散列地址对应的单元为空，表明检索失败；否则，
  - 如果散列表是采用开放地址法解决碰撞建立的，按照散列表建立时寻找下一个单元的序列（建立时的规则）寻找下一个地址，如此反复进行，直到关键码比较相等（检索成功）或找到空单元（检索失败）为止；
  - 如果散列表是采用拉链法解决碰撞建立的，那么沿着该非空指针进入同义词单链表，在该链表中进行检索。如果找到某个结点的存储的记录的关键码与给定值key相同，检索成功；否则，检索失败。

# 检索效率分析

---

## □ 检索效率

- 虽然散列表在关键码与记录的存储位置之间建立了直接映象，但由于“碰撞”的产生，使得散列表的检索过程仍然是一个给定值和记录关键码进行比较的过程。
- 因此，仍需要以平均查找长度衡量散列表检索效率

## □ 检索过程中，关键码的比较次数取决于下列三个因素：

- 散列函数
- 处理碰撞的方法
- 负载因子

# 影响因素

---

- 散列函数的“好坏”首先影响出现冲突的频率。对于同一组随机的关键码，“均匀分布”的散列函数产生碰撞的可能性相同，则可不考虑它对ASL的影响。
- 碰撞处理方法：通常拉链法的ASL要小于开放地址法。线性探测容易产生“二次聚集[堆积]”问题，而拉链法不会出现这种情况。
- 负载因子：负载因子越小，发生碰撞的可能性越小；反之，负载因子越大，表中已填入的记录多，再填记录时，发生碰撞的可能性就越大，检索时比较次数就越多。