哈希 用数学方法实现快速查找
——领略数学之美

数 据 结 构

第六、七、十章 搜索篇 [5]　　散列或哈希

任课老师：郭艳

数据结构课程组

计算机学院

中国地质大学（武汉）2020年秋

hash——散列、哈希

hash，译为散列、哈希（音译）、杂凑
n.用旧料拼凑成的东西;混乱;一团糟大杂烩;v.把……弄糟（乱）.

哈希思想是计算机科学里的伟大发明，是将任意长度的源数据映射到有限长度（固定）的输出数据，该输出就是散列值。这种转换是一种压缩映射，即散列值的空间通常小于输入空间。输出和源数据的各个部分可能有关系。

哈希已被我们广泛应用：
1、散列表：可以提高数据的查询效率;
2、提高存储空间的利用率;
3、做数字签名来保障数据传递的安全性。

哈希方法的性质：
1、确定性：如果同一个映射方法，两个散列值是不相同的，那么这两个散列值的原始输入也是不相同的。
2、冲突：如果两个散列值相同，两个输入值可能是不相同的。
3、输入一些数据计算出散列值，然后部分改变输入值，一个具有强混淆特性的散列函数会产生一个完全不同的散列值。
4、原像不可逆性：无法通过散列值倒推源数据是什么。
5、典型的散列函数都有无限定义域（比如任意长度的字节字符串）和有限的值域（比如固定长度的比特串）。
6、在某些情况下，散列函数可以设计成具有相同大小的定义域和值域间的一一对应

上堂课要点回顾

- B 树
  - m路搜索树
  - B树 定义
  - B树 的查找　　$h \le 1+\log_{\lceil m/2 \rceil}((N+1)/2)=O(\log_{\lceil m/2 \rceil}(N))$
  - B树 的插入
  - B树 的删除

第 十五 次 课

阅读：

殷人昆，第279-293页

习题：

作业15

6.3 字典（Dictionary）

- 经常需要判定某个元素是否在给定的集合中，并且需要经常对这个集合进行查找、插入和删除操作，这个集合的抽象数据类型叫做字典。
- 字典是一些元素的集合，每个元素有一个称作关键码（key）的域，不同元素的关键码互不相同。
- 在讨论字典抽象数据类型时，把字典定义为<名字-属性>对的集合。根据问题的不同，可以为名字和属性赋予不同的含义。一般，名字是唯一的，用关键码对应名字。
  - 比如有份成绩表数据，语文：79，数学：80，英语：92，这组数据看上去像两个列表，但这两个列表的元素之间有一定的关联关系。如果单纯使用两个列表来保存这组数据，则无法记录两组数据之间的关联关系。其中，名字是课程，属性是课程的对应分数
  - 在图书馆检索目录中，名字是书名，属性是索书号及作者等信息
  - 在计算机活动文件表中，名字是文件名，属性是文件地址、大小等信息。

- 一般来说，有关字典的操作有如下几种：
  - 确定一个指定的名字是包含否在字典中；
  - 搜索出该名字的属性；
  - 修改该名字的属性；
  - 插入一个新的名字及其属性；
  - 删除一个名字及其属性。
- 用文件记录（record）或表格的表项（entry）来表示单个元素时，用（关键码key，记录或表项位置指针adr）构成搜索某一指定记录或表项的索引项

字典的抽象数据类型

```
const int DefaultSize = 26;
template <class Name, class Attribute>
class Dictionary {
//对象: 一组<名字-属性>对, 其中, 名字是唯一的
public:
    Dictionary (int size = DefaultSize);      //构造函数
    bool Member (Name name);      //判name是否在字典中
    Attribute *Search (Name name);
        //在字典中搜索关键码与name匹配的表项
    void Insert (Name name, Attribute attr);
      //若name在字典中, 则修改相应<name, Attr>对
      //的attr项; 否则插入<name, Attr>到字典中
    void Remove (Name name);
      //若name在字典中, 则在字典中删除相应的<name, Attr>对
};
```

字典的实现

- **维护有序性**
  - 静态顺序表、有序顺序表、BST、AVL树、2-3树、 红黑树
- **不考虑有序性，换取查找、插入、删除的高效**
  - **哈希**（哈希是python字典实现方案）

(1)闭散列： $\alpha<0.5$时O(1)，当$\alpha=0.5$需进行扩容时O(n)；
开散列： $\alpha<1.5$时O(1)，当$\alpha=1.5$需进行扩容时O(n)

搜索结构	Member()	Search()	Insert()	Remove()	条件	简易性
staticList	O(N)	O(N)	O(N)	O(N)		简单
sortedList	$O(\log_2 N)$	$O(\log_2 N)$	O(N)	O(N)	有序	简单
BST	O(h)	O(h)	O(h)	O(h)	h：树的深度.随机情况 $h=\log_2 N$,退化情况h=N	复杂
AVL Tree	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$		复杂
2-3 Tree	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$		复杂
RB Tree	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$		复杂
HashTable	$O(\alpha)$	$O(\alpha)$	$O(\alpha)$	$O(\alpha)$	各项被均匀散列在表中;$\alpha=n/m$;(1)	简单、直观

哈希方法实现快速查找

- 哈希表是集合Set和映射Map最普遍的实现方式
  - 实际应用高效
  - 各项不能比较也能使用
  - 实现经常很简单

6.5 散列

- 基于关键字比较的查找（?x<y）
  - 顺序查找：==， !=　　（代价O(n)）
  - 折半法、树型：>，== ，<（代价$O(\log_2 n)$, $n=10^9$, 则约为30）
  - 当问题规模n很大时，上述查找的时间效率可能使用户无法忍受。若关键字不可比较，方法不适用
- 最理想的情况是：
  - 不需要把待查记录的关键字与查找表的某些记录进行逐个比较。而是根据待查记录的关键字值，直接找到该记录的存储地址。

    例如，给定一待查找记录R，根据查找表的起始存储地址、以及待查找记录R的关键字值，而直接计算出记录R的存储地址来，所花费的时间与查找表的规模n无关，查找代价是O(1)。
- 受此启发，计算机科学家发明了散列方法
  - 散列是一种重要的存储方法
  - 散列也是一种常见的查找方法

散列的基本思想

- 一个确定的函数关系H（称为散列函数，或哈希函数），y=H（x）
  - 以结点的关键字key做为自变量
  - 以函数值H(key)作为结点的存储地址
- 需要预先建立散列表(或称哈希表)。
  - 将一组无限长度的关键字映射到一个有限长度的连续的地址集（区间）上，并以关键字在地址集中的“象” 作为记录在表中的存储位置，这种表便称为散列表，这一映象过程称为散列造表或散列
  - 建立散列表时就是根据这个散列函数计算文件中每个记录的存储位置
  - 在散列表查找时也是根据这个散列函数计算待查记录的存储位置
  - 通常散列表的存储空间是一个一维数组，散列地址是数组的下标

散列函数H的举例1

例1：已知一长度为14的线性表关键字集合S = { and, begin, do, end, for, go, if, repeat, then, until, while, else, array, when }。

设查找表中每个关键字表示为：key[8];

设查找表采用散列法表示为：char ht[26][8];

散列函数H(key)的值取为关键字key中的第一个字母在字母表{a, b, c, ..., z}中的序号，即：

H(key)=key[0] – ‘a’

```
int H (char *key)
{   return(key[0]–‘a’);   }
```

散列地址	关键字
0	and (array)
1	begin
2	
3	do
4	end (else)
5	for
6	go
7	
8	if
9	
10	
11	
12	

冲突

冲突

散列地址	关键字
13	
14	
15	
16	
17	repeat
18	
19	then
20	until
21	
22	while with
23	
24	
25	
26	

冲突

散列函数H的举例2

例2：在集合S = { and, begin, do, end, for, go, if, repeat, then, until, while, else, array, when }

修改散列函数H(key)：值为key中首尾字母在字母表中序号的平均值，即：

```
int H (char *key)
{    int  r= 0;//r是关键字key字符串尾部字母的下标
     while ((r<8) && (key[r]!='\0'))  r++;
     return((key[0] + key[r-1] – 2*'a') /2 );
}
```

散列地址	关键字
0	
1	and
2	
3	end
4	else
5	
6	if
7	begin
8	do
9	
10	go
11	for
12	array

无冲突现象！

散列地址	关键字
13	while
14	
15	with
16	until
17	then
18	
19	repeat
20	
21	
22	
23	
24	
25	
26	

6.5.1 Hash表的基本概念

- **散列函数：把关键字值key映射到散列表存储位置的函数，通常用H来表示。**

$$Address = H(key)$$

  - 若对于关键字集合中的任一个关键字，经散列函数映象到地址集合中任何一个地址的概率是相等的，则称此类散列函数为均匀散列函数(Uniform Hash function)

- **装载因子α（load factor）:表明表装满的程度**

$$\alpha = n/m$$

  - n为填入散列表中的记录数；m为散列表的空间大小。

- **冲突**
  - 某个散列函数key1≠key2，而H(key1)=H(key2)。对于不相等的关键字计算出了相同的散列地址，称为“发生了冲突”。在实际应用中，不产生冲突的散列函数极少存在。
  - 冲突不可避免
  - 根据散列函数H(key)和处理冲突的方法，所得的存储位置称散列地址。均匀散列函数因散列地址随机从而减少聚积，减少冲突。

分析散列y=H(x)冲突的概率

若有m个y值，n个x，则

1、n个x中至少两个x映射到同一个y值的概率是

$1-(P(m,n)/m^n)$。

2、若至少两个x映射到同一个y值发生碰撞大于50%概率，则$n=1.2m^{0.5}$。

——以上数学事实十分反直觉。散列值个数为2^N的哈希表可能发生碰撞的测试次数不是 2^N 次而是只有 $2^{N/2}$ 次。

例、n个人中至少两个人的生日相同的概率是多少？

m=365，若n=20，则生日碰撞的概率是47.14%，

若n=60，则生日碰撞的概率是99.41%。

例、不少于多少人中至少两个人生日相同的概率会大于50%？

n=22.9，约23人。

组合数学的抽屉原理（鸽巢原理）

桌上有十个苹果，要把这十个苹果放到九个抽屉里，无论怎样放，我们会发现至少会有一个抽屉里面放不少于两个苹果。这一现象就是我们所说的“抽屉原理”。 抽屉原理的一般含义为：“如果每个抽屉代表一个集合，每一个苹果就可以代表一个元素，假如有n+1个元素放到n个集合中去，其中必定有一个集合里至少有两个元素。” 。

在散列中，n个x压缩到m个y中，n>m。因为整数溢出，所以y的个数m有限。根据抽屉原理，必定两个x会被散列到同一个y散列值，即必定会发生冲突。
底线：冲突是不可避免的。
因此散列技术必须考虑冲突的解决方案。

散列技术的首要问题

- 采用散列技术时需要考虑两个首要问题：
  - 如何选择使记录“分布均匀”的散列函数？
  - 一旦发生冲突，用什么方法来解决？
- 还需考虑散列表本身的组织方法

6.5.2 Hash函数的构造方法 P280

- **Hash函数的选取原则：**

① 函数的值域必须在Hash表长的范围内

② 计算尽可能简单

③ 尽可能随机性要好：指Hash函数应当尽可能均匀地把关键字映射到整个地址区间中（尽可能避免冲突）

- **Hash函数的选取需要考虑的因素：**

① 关键字的长度

② 散列表的大小（表长m）

③ 关键字的分布情况

④ 记录的查找频率

- **整数类型关键字的常用Hash函数构造方法有：**

1、除留余数法	2、直接定址法	3、数字分析法
4、平方取中法	5、分段叠加法	

1. 除留余数法

H (key) =key % p

(p≤m，p往往取Hash表的长度m)

例：m=1000，p=1000，将下面4个关键字用1000去除

关键字值	% 1000
10052501	501
10052502	502
01110525	525
02110525	525

Hash函数值一样，显然不好

p的选择：

- 一般地说，如果 p 的约数越多，那么冲突的几率就越大
  - 简单证明：假设 p 是一个有较多约数的数，同时在数据中存在 q 满足 gcd(p,q)=d >1 ，即有 p=a*d , q=b*d, 则有

    q mod p= q - p* [q div p] =q - p*[b div a] . ①
    - 其中 [b div a] 的取值范围是不会超过 [0，b] 的正整数。也就是说， [b div a] 的值只有 **b+1** 种可能，而 p 是一个预先确定的数。因此 ① 式的值就只有 b+1 种可能了。
    - 这样，虽然mod 运算之后的余数仍然在 **[0，p-1]** 内，但是它的取值仅限于 ① 可能取到的那些值。也就是说余数的分布变得不均匀了。容易看出， p 的约数越多，发生这种余数分布不均匀的情况就越频繁，冲突的几率越高。
- 素数的约数是最少的，因此p我们选用素数
  - 素数是我们的得力助手

p的选择*：

理论分析和实践结果均证明，p应取小于或等于m的(最大)素数。

例：　m=1000，H(key)=key % 997

10052501	747
10052502	748
01110525	864
02110525	873

- **优点**：函数值依赖于自变量key的所有位，而不仅仅是最右边几个低位，增大了均匀分布的可能性。
- **潜在缺点**：连续的关键字映射成连续的散列值，虽然能保证连续的关键字不发生冲突，但也意味着要占据连续的数组单元，可能导致程序性能的降低。

2. 直接定址法

（1）取关键字值作为它的Hash地址：

H(key)=key

（2）取关键字值的某个线性函数：

H(key)=a*key+b　　(a, b为常数)

例1：从1→100岁年龄的人口数字统计表，以年龄作为关键字，地址值取关键字自身。

H(key)= key

例2．统计从1901年开始的各年份的人口总数，以年份为关键字，地址值 H(key)= key+(−1900)

地址	01	02	03	…	
年份	1901	1902	1903	…	
人数	1000	2000	3000	…	

3. 数字分析法

- 设有 n 个 m 位数，每一位可能有 d 种不同的符号
- 这 d 种不同的符号在各位上出现的频率不一定相同
  - 可能在某些位上分布均匀些，每种符号出现的几率均等；
  - 在某些位上分布不均匀，只有某几种符号经常出现。
- 可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。
- 计算各位数字中符号分布的均匀度 λ_k 的公式：

$$\lambda_k = \sum_{i=1}^{d} (\alpha_i^k - n/d)^2$$

  - 其中，α_i^k 表示第 i 个符号在第 k 位上出现的次数，
  - n/d 表示各种符号在 n 个数中均匀出现的期望值。
  - 计算出的 λ_k 值越小，表明在该位 (第k 位)各种符号分布得越均匀。

k1=	9	9	2	1	4	8
k2=	9	9	1	2	6	9
k3=	9	9	0	5	2	7
k4=	9	9	1	6	3	0
k5=	9	9	1	8	0	5
k6=	9	9	1	5	5	8
k7=	9	9	2	0	4	7
k8=	9	9	0	0	0	1
	①	②	③	④	⑤	⑥

①位，$\lambda_1 = 57.60$　$H(k_1)=148$

②位，$\lambda_2 = 57.60$　$H(k_2)=269$

③位，$\lambda_3 = 17.60$　$H(k_3)=527$

④位，$\lambda_4 = 5.60$　$H(k_4)=630$

⑤位，$\lambda_5 = 5.60$　$H(k_5)=805$

⑥位，$\lambda_6 = 5.60$　$H(k_6)=558$

$H(k_7)=047$

$H(k_8)=001$

- 若散列表地址范围有 3 位数字, 取各关键字的④⑤⑥位做为记录的散列地址。
- 数字分析法仅适用于事先明确知道表中所有关键字每一位数值的分布情况。
- 如果换一个关键字集合，选择哪几位数据需要重新决定。

3. 数字分析法（续）

- ①位，仅9出现8次，
  - $\lambda_1 = (8-8/10)^2 \times 1 + (0-8/10)^2 \times 9 = 57.6$
- ②位，仅9出现8次，
  - $\lambda_2 = (8-8/10)^2 \times 1 + (0-8/10)^2 \times 9 = 57.6$
- ③位，0和2各出现两次，1出现4次
  - $\lambda_3 = (2-8/10)^2 \times 2 + (4-8/10)^2 \times 1 + (0-8/10)^2 \times 7 = 17.6$
- ④位，0和5各出现两次，1、2、6、8各出现1次
- ⑤位，0和4各出现两次，2、3、5、6各出现1次
- ⑥位，7和8各出现两次，0、1、5、9各出现1次
  - $\lambda_4 = \lambda_5 = \lambda_6 = (2-8/10)^2 \times 2 + (1-8/10)^2 \times 4 + (0-8/10)^2 \times 4 = 5.6$

4. 平方取中法

先通过求关键字的平方值来扩大差别，再取其中的几位或其组合作为散列地址。具体取几位，由Hash表的表长决定。

例：若表长m=1000，则可取中间三位作为散列地址：

key_i	key_i^2	$H(key_i)$
0100	0010000	**010**
1100	1210000	**210**
1200	1440000	**440**
1160	1345600	**345**

5. 折叠法（或分段迭加法）

- 关键字所含的位数很多，采用平方取中法计算太复杂。
- 折叠法
  - 将关键字分割成位数相同的几部分(最后一部分位数可不同)，
  - 然后取这几部分的叠加和（舍去进位）作为散列地址。

例：如果一本书的编号为8-243-834-7，表长有三位数

（1）移位叠加：把各部分的最后一位对齐相加

key=82438347

347

438

+） 82

867 H (key)=867

（2）**间界叠加**：沿分割界来回折叠，然后对齐相加

key=82438347

347

834

+） 82

1263 H (key)=263

散列函数的应用

- 在实际应用中应根据关键字的特点，选用适当的散列函数。
- 有人曾用“轮盘赌”的统计分析方法对它们进行了模拟分析，结论是平方取中法最接近于“随机化”。
- 若关键字不是整数而是字符串时，可以把每个字符串转换成整数，再应用平方取中法或除留余数法。

```
//字符串Hash函数1
int H(char* key, int tableSize )
{    int i, sum;
     for (sum=0, i=0; key[i] != '\0'; i++)
               sum += (int) key[i];
 //例如 char c='D' 等价于 char c=68
     return(sum % tableSize);
}
```

字符串hash函数示例

源输入	源输入格式	源输入表示	源输入每位转换为整数	基数	哈希函数	示例	C++可表示的最大整数值及其下一个值	因为整数溢出，示例实际的散列值
参考：十进制数		d1d2...dn		10	$(d_1d_2...d_n)_{10}=(d_1)*10^{n-1}+(d_2)*10^{n-2}+...+(d_n)*10^0=D_{10}$	$2789_{10}=2*10^3+7*10^2+8*10^1+9*10^0=2789_{10}$	2, 147, 483, 647; – 2, 147, 483, 648	
小写英文字符的串A	ascii码	a1a2...an	asciitoint()（简写成A2I）	27	$(a_1a_2...a_n)_{27}=A2I(a_1)*27^{n-1}+A2I(a_2)*27^{n-2}+...+A2I(a_n)*27^0=A_{10}$	$cat_{27}=3*27^2+1*27^1+20*27^0=2234_{10}$		
可输出的字符的串A	ascii码	a1a2...an	asciitoint()（简写成A2I）	126	$(a_1a_2...a_n)_{126}=A2I(a_1)*126^{n-1}+A2I(a_2)*126^{n-2}+...+A2I(a_n)*126^0=A_{10}$	$omens_{126}=28,196,917,171$		– 1, 867, 853, 901
支持汉字字符的串U	unicode码	u1u2...un	unicodetoint()（简写成U2I）	40959	$(u_1u_2...u_n)_{40959}=U2I(u_1)*40959^{n-1}+U2I(u_2)*40959^{n-2}+...+U2I(u_n)*40959^0=U_{10}$	守门员$_{40959}$=23432*40959^2+38376*40959^1+21592*40959^0=39, 312, 024, 869, 368		189, 209, 080

因为整数溢出，所以

- 基数一般选择一个小的素数
- 哈希函数使用求模运算限制内存空间

哈希函数常使用 霍纳法则(Horner's Rule)

霍纳方法简化了朴素多项式的求值。在中国叫秦九韶算法。

问题：求多项式$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$在$x_0$处的值。

霍纳方法是：$P_n(x_0)=((\ldots(\ldots(((a_n x_0+a_{n-1})x_0+a_{n-2})x_0+ a_{n-3})x_0+\ldots)x_0+\ldots+ a_1)x_0+ a_0)$

霍纳方法是采用最少的乘法运算策略，将一元n次多项式的求值问题转化为n个一次式的算法，其大大简化了朴素多项式的求值过程，原本是n(n＋1)/2次乘法运算和n次加法的计算过程，简化为n次乘法运算和n次加法运算的计算过程。即使在现代，利用计算机解决多项式的求值问题时，霍纳规则依然是最优的算法规则。

一个好的字符串Hash函数

```
int H (char *key, int tableSize )
{  int hashVal = 0;
   while( *key != '\0' )
        hashVal = 37* hashVal + *key++;
        // *key是第key位的ASCII码值
        //数学的应用1：霍纳法则
        //数学的应用2：使用一个小素数37作为基数
        //由于整数溢出，hashVal可能是个负值
   hashVal %= tableSize;
   if (hashVal <0)
        hashVal += tableSize;
        //将散列值约束在一个固定长度内：0...tableSize-1
   return hashVal;
}
```

ELFhash字符串散列函数

- 用于UNIX系统V4.0“可执行链接格式”(Executable and Linking Format，即ELF)

```
int H(char* key, int tableSize)
{   unsigned long hashVal = 0;
    while(*key)
    {       hashVal = (hashVal << 4) + *key++;
            unsigned long g = hashVal & 0xF0000000L;
            if (g) hashVal ^= g >> 24;
            hashVal &= ~g;
    }
    return hashVal % tableSize;
}
```

- 长字符串和短字符串都很有效
- 字符串中每个字符都有同样的作用
- 对于散列表中的位置不可能产生不平均的分布

冲突的解决方法

1.闭散列方法 (closed hashing, 或开放定址法 open addressing)

把所有记录直接存储在散列表中。把发生冲突的关键字存储在表中另一个位置

- 线性探查再散列
- 二次探查再散列

2. 开散列方法(open hashing，链地址法 separate chaining)

把发生冲突的关键字存储在散列表主表之外

3. 双Hash法

H_0=H(key)

p= *ReHash*(*key*)；*p*是小于*m*且与*m*互质的整数

$H_i = (H_{i-1} + p) \% m$； i=1, 2, … , n

6.5.3 闭散列法——找“下一个”空位

- 把所有记录直接存储在散列表中
- 每个记录关键字有一个基位置，即由散列函数计算出来的地址H(key)
- 如果要插入一个记录R，而另一个记录已经占据了R的基位置(发生冲突)
  - 那么就把R存储在表中的其它地址内（由冲突解决策略确定是哪个地址）
  - 当冲突发生时，使用某种方法为R的关键字key生成一个散列地址序列$d_0, d_1, d_2, \ldots, d_i, \ldots d_{m-1}$。
    - 令d(i)是探查函数
    - 其中d_0=H（key）称为key的基地址。
    - 第i次冲突散列地址$d_i = (d_{i-1} + D) \% m$或$d_i = (d_0 + D_i) \% m$, i=1, 2, …, m-1，其中m是Hash表表长。
  - 按上述地址序列依次探查，将找到的第一个开放的空闲位置d_i作为key的存储位置。
  - 若所有后继散列地址都不空闲，说明该闭散列表已满，报告溢出。

(1)线性探查再散列

$$D = i$$

```
int D(int i)      /*线性探查函数，i表示探查次数，
   返回距离基地址的偏移量*/
{  return i;}
```

优点：表中所有的存储位置都可以作为插入新记录的候选位置。

缺点：不同的关键字的探查序列可能重合，导致很长的探查序列，争夺同一后继散列地址。

例1：已知一组关键字为（9，11，2，16，3，1），散列表长度为 8，用线性探查法解决冲突构造这组关键字的散列。

- 已知n=6, m=8；选p=7，则H(key)=key%7按顺序插入各个记录：

R_i	key_i	$d_0=H(key_i)$
R_1	9	2
R_2	11	4
R_3	2	2 冲突，$d_1=(2+1)\%7=3$。
R_4	16	2 冲突，$d_1=(2+1)\%7=3$，$d_2=(2+2)\%7=4$，$d_3=5$。
R_5	3	3 冲突，$d_1=(3+1)\%7=4$，$d_2=5$，$d_3=6$。
R_6	1	1 完毕！

n=6
m=8
H(key)=key%7
冲突解决：
线性探查法

构造的Hash表ht[8]

0	1	2	3	4	5	6	7
	1	9	2	11	16	3	

线性探查散列表查找算法的性能分析

例1：

H(key)=key%7

ht[8]

0	1	2	3	4	5	6	7
	1	9	2	11	16	3	

关键字	9	11	2	16	3	1
直接散列地址	2	4	3	2	3	1
查找成功探查次数	1	1	2	4	4	1

利用公式 $ASL=\sum_{i=1}^{n}P_iC_i$ ，设查找每个记录的概率相等，即P_i=1/6

则查找成功的$ASL_{success}=\frac{1}{6}\sum_{i=1}^{6}C_i=\frac{1}{6}(1\times3+2\times1+4\times2)=\frac{13}{6}$

下标	0	1	2	3	4	5	6
查找失败探查次数	1	7	6	5	4	3	2

因为H(key)=key%7,所以H(key)=[0..6]，

查找不成功的$ASL_{failure}$=(1+7+6+5+4+3+2)/7=28/7=4

(2) 二次探查再散列

$$D = 1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \ldots\ldots$$

```
int D(int i)    /*二次探查， i表示探查次数,
   返回距离基地址的偏移量*/
{  if (i%2==0)
       return -(i*i/4);
   else
       return (i+1)*(i+1)/4;
}
```

如前例1

R_3：$d_1=(2+1^2)\%\ 7=3$，即R_3存贮在第3个单元。

R_4：$d_1=(2+1^2)\%\ 7=3$，冲突，$d_2=(2-1^2)\%\ 7=1$，R_4存贮在第1个单元。

R_5：$d_1=(3+1^2)\%\ 7=4$，冲突，$d_2=(3-1^2)\%\ 7=2$，冲突，

$d_3=(3+2^2)\%\ 7=0$，R_5存贮在第0单元。

R_6：$d_1=(1+1^2)\%\ 7=2$，冲突，$d_2=(1-1^2)\%\ 7=0$，冲突，

$d_3=(1+2^2)\%\ 7=5$，R_6存贮在第5单元。

ht[8]:

0	1	2	3	4	5	6	7
3	16	9	2	11	1	3	

例2：使用一个大小m = 13的表,假定对于关键字key_1和key_2，

$H(key_1)=3$，$H(key_2)=2$。

- key_1的探查序列是3、4、2、7 、...
- key_2的探查序列是2、3、1、6 、...

优点：尽管key_2会把key_1的基位置作为第2个选择来探查，但这两个关键字的探查序列此后就立即分开了。

(3) 双散列法

二次聚集：如果两个关键码散列到同一个基地址，还是得到同样的探查序列，所产生的聚集称为二次聚集
原因探查序列只是基地址的函数，而不是原来关键码值的函数

双Hash法：有两个Hash函数，探查序列不仅是基地址的函数，也是原来关键码值的函数

$H_0 = Hash(key)$

$p = ReHash(key)$；p是小于m且与m互质的整数

$H_i = (H_{i-1} + p) \% m$； i=1, 2, … , n

双散列的优点：不易产生“聚集”
缺点：计算量增大

- 按一定的距离, 跳跃式地寻找“下一个”桶, 减少了“堆积”的机会。需要两个散列函数
  - 第一个散列函数 *Hash*() ：按表项的关键码 *key* 计算表项所在的桶号 $H_0 = Hash(key)$
  - 一旦冲突，利用第二个散列函数 *ReHash*() 计算该表项到达“下一个”桶的移位量。它的取值与 *key* 的值有关, 要求它的取值应是小于地址空间大小 TableSize, 且与 TableSize 互质的正整数。
- 若设表的长度为 *m* = TableSize，则在表中寻找“下一个”桶的公式为

  $H_0 = Hash(key)$

  $p = ReHash(key)$；p是小于m且与m互质的整数

  $H_i = (H_{i-1} + p) \% m$； i=1, 2, … , n

例：给出一组表项关键码{ 22, 41, 53, 46, 30, 13, 01, 67 }。散列函数为：$Hash(x)=x \% 11$

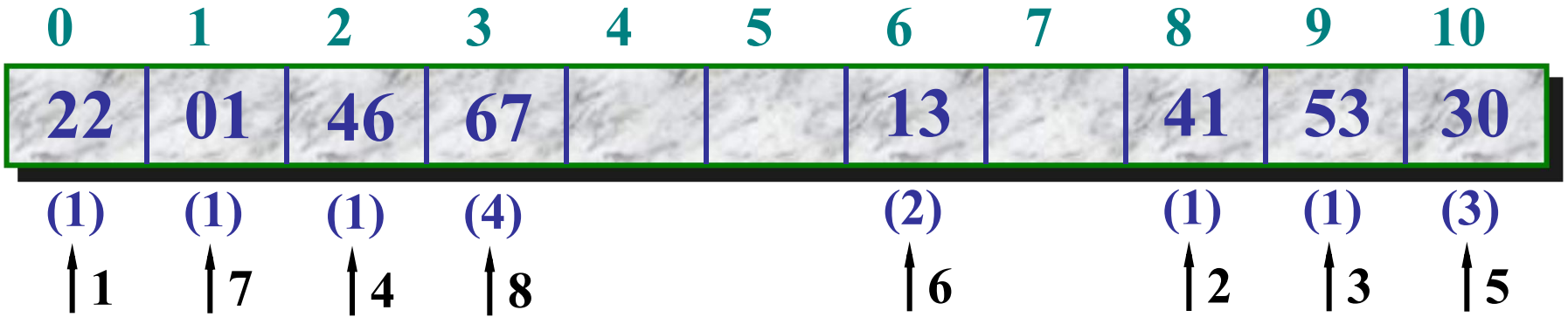
- 散列表HT[0..10]，$ReHash(x) = x \% 10 + 1$。
- $H_i = (H_{i-1} + x \% 10 + 1) \% 11, \quad i = 1, 2, \ldots$
- $H_0(22) = 0$ $H_0(41) = 8$ $H_0(53) = 9$ $H_0(46) = 2$ $H_0(30) = 8$ 冲突 $p = 1$ $H_1 = 8+1 = 9$ 冲突 $H_2 = 9+1 = 10$

  $H_0(13) = 2$ 冲突 $p = 4$ $H_1 = 2+4 = 6$

  $H_0(01) = 1$

  $H_0(67) = 1$ 冲突 $p = 8$ $H_1 = 1+8 = 9$ 冲突

  $H_2 = 9+8 = 6$ 冲突 $H_3 = 6+8 = 3$



闭散列表的算法实现

1、查找算法

- 查找时也要像插入时一样遵循同样的策略
  - 重复冲突解决过程
  - 采用的探查序列也相同
  - 找出在基位置没有找到的记录
- 查找算法：假设给定的值为key，
  - 1、根据所设定的散列函数H，计算出散列地址H (key)，
  - 若表中该地址对应的空间未被占用（空单元），则查找失败；
  - 否则将该地址中的值与key比较，若相等则查找成功；
  - 否则，按建表时设定的处理冲突方法查找探查序列的下一个地址，如此反复下去，直到
    - 某个地址空间未被占用（空单元），则查找失败；
    - 或者关键字比较相等，则查找成功

    为止；
  - 若所有后继散列地址都比较完，关键字比较不相等，说明表满且查找失败。

2、插入算法

- 假设给定的值为key，根据所设定的散列函数H，计算出散列地址H (key)
  - 若表中该地址对应的空间未被占用，则把待插入记录填入该地址空间
  - 如果该地址中的值与key相等，则不插入，报告“散列表中已有此记录”
  - 否则，按设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
    - 直到某个地址空间未被占用，插入在该地址空间
    - 或者关键字比较相等为止（报告有重复记录，不插入）
    - 若所有后继散列地址都不空闲，说明该闭散列表已满，不能插入，报告溢出。

3、删除算法

- 闭散列删除记录的时候，有两点需要重点考虑：
  - 删除一个记录一定不能影响后面的查找；
  - 删除后释放的存储位置应该能够为将来的插入使用（不想让散列表中的位置由于删除而永远不可用）
- 开散列方法可以真正删除
- 但是 闭散列方法不可以真正删除，只能做标记

闭散列法可以真正删除吗?

- 方案一、把被删除位置的探查序列之后其它记录逐步前移? ——不可取
  - 因为一个单元可能处于不只一个探查序列中。
  - 这可能会把其它同义词表中的记录给挪动了，而造成其它混乱。
- 方案二、把该探查序列中最后的记录填入到刚刚被删除的单元? ——不可取
  - 同上述原因。
  - 另外查找该探查序列中最后的记录需要额外的查找时间。
- 闭散列方法不能真正删除，只能作标记(“碑”)。
  - 若真正删除了，探查序列将断掉。
    - 查找算法：“直到某个地址空间未被占用(查找失败)”
  - 标记“碑”增加了平均查找长度

真正删除导致断线索的示例

- m = 15
- H(key) = key%13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	25	41	15	68	44	6				36		38	12	51

- 删除41和查找15
  - 关键字41和15的基地址都是第2个单元，15被线性探查放到第3个单元
  - 如果从表中真正删除41，对15的查找必须仍然探查第2个单元，断线索

引入“碑”

- 设置一个特殊的标记位，来记录散列表中的单元状态
  - 空单元(“Empty”)
  - 非空单元，单元被占用（“Active”）
  - 非空单元，该单元已被删除（“Deleted”，碑）
- 是否可以把“空单元”、“已被删除”这两种状态，用特殊的值标记，以区别于“单元被占用”状态？
  - 不可以！因为空单元是查找失败的判断依据，必须区别空单元。
- 被删除标记称为碑(tombstone)
  - 标志一个记录曾经占用这个单元，但是现在已经不再占用了

“碑”对操作的影响

- 查找操作可以不修改
  - 查找时，如果遇到碑，查找过程会顺着探查序列继续进行。
  - 可以把碑看成是不等于任何关键字的特殊值，对于算法没有影响。
- 加入了删除操作后，闭散列的插入操作需要进行修改。

修改后的插入算法

- 假设给定的值为key，根据所设定的散列函数H，计算出散列地址H (key)
  - 若表中该地址对应的空间~~未被占用~~状态!=Active，则把待插入记录填入该地址空间
  - 如果该地址中的值与key相等，则不插入，报告“散列表中已有此记录”
  - 否则，按设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
    - 直到某个地址空间~~未被占用~~状态!=Active，插入该空间
    - 或者关键字比较相等为止（报告有重复记录，不插入）
    - 若所有后继散列地址都不空闲，说明该闭散列表已满，不能插入，报告溢出。

用线性探查法组织的散列表的类定义P284 程序6. 22

```
const int DefaultSize = 100;
enum KindOfStatus {Active, Empty, Deleted};
   //状态分类 (活动/空/被删除)
template <class E, class K>
class HashTable {                    //散列表类定义
private:
    int divitor;       //除留余数法散列函数的除数
    int CurrentSize, TableSize;//当前数及最大数
    E *ht;                          //散列表存储数组
    KindOfStatus *info; //状态数组
    int FindPos (K k1) const;    //搜索
    void Resize();   //扩容
```

```
//重载函数.....
public:
    HashTable (const int d, int sz = DefaultSize);//构造函数
    ～HashTable() { delete []ht;  delete []info; }//析构函数
    HashTable<E, K>& operator =
            (const HashTable<E, K>& ht2);   //表赋值
   bool Search (K k1, E& e1) const;          //搜索k1
    bool Insert (const E& e1);               //插入e1
    bool Remove (const E& e1);               //删除e1
    void makeEmpty ();                       //置表空
};
```

【线性探查散列表的构造算法】

```
template<class E, class K>                          //构造函数
HashTable<E, K>::HashTable (int d, int sz) {
    divitor = d;                                    //除数
    TableSize = sz;                                 //最大表长
    CurrentSize = 0;                                //当前表长
    ht = new E[TableSize];                          //表存储空间
    info = new KindOfstatus[TableSize];//空间状态
    for (int i = 0; i < TableSize; i++)
        info[i] = Empty;
};
```

【线性探查散列表的清空】P286 程序6.24

```
template <class E, class K>
void HashTable<E,K>::makeEmpty(){
        for (int i = 0; i < TableSize; i++){
                info[i] = Empty;     //只需将info表清空
        }
        CurrentSize = 0;
}
```

【线性探查散列表的搜索算法】P285 程序6.23(1)

```
template <class E, class K>
int HashTable<E, K>::FindPos (K k1) const {
//搜索在一个散列表（除留余数法和线性探测再散列）中
//关键码与k1匹配的元素，搜索成功，则函数返回该元素的
//位置，否则返回插入点（如果有足够的空间）
    int i = k1 % divitor;                    //计算初始散列单元
    int j = i;                               //j是检测下一空单元下标
    do {
        if (info[j] == Empty ||              //查找失败
           (info[j] == Active && ht[j] == k1))//查找成功
               return j;   //查找失败 或 成功
        j = (j+1) % TableSize;      //在循环表中找下一个空单元
    } while (j != i);       //转一圈
    return j;               //转一圈回到开始点j==i, 表已满, 失败
};
```

```
//P285 程序6.23（2）
bool HashTable<E, K>::Search (K k1, E& e1) {
//使用线性探查法在散列表ht(每个桶容纳一个元素)
//中搜索k1，查找成功通过e1返回查找到的元素
    int i = FindPos (k1);                    //搜索
    if (info[i] == Active && ht[i].key == k1)
   {    e1 = ht[i];    //通过第二个参数返回找到的记录
        return true;    }
   return false;
};
```

【线性探查散列表的插入操作】P286 程序6.24（3）

```
template <class E, class K>
bool HashTable<E, K>::Insert (const E& e1) {
//在ht表中搜索k1。若找到则不再插入, 若未找到,
//并查找返回位置的标志是Empty或Deleted, x插入
 int i = FindPos (k1);
//用散列函数计算单元号，成功返回单元号,
//失败返回探测到的第一个空单元
    if (info[i] != Active) {     //该单元为空或被删,存放新元素
        ht[i] = e1;   info[i] = Active;
        CurrentSize++;   return true;}
    if (info[i] == Active && ht[i] == e1)
        cout << "表中已有此元素，不能插入！\n";
    else  //info[i] == Active && ht[i] != e1
        cout << "表已满，不能插入！\n";
    return false;
}
```

【线性探查散列表的删除操作】P286 程序6.24（4）

```
template <class E, class K>
bool HashTable<E, K>::Remove (K k1, E& e1) {
//在ht表中删除元素key, 并在引用参数e1中得到它
    int i = FindPos (k1);
    if (info[i] == Active &&ht[i].key==k1) {
            //找到要删元素, 且是活动元素
        info[i] = Deleted;  CurrentSize--;    e1=ht[i];
            //做逻辑删除标志, 并不真正物理删除
        return true;
    }
    else return false;
};
```

- 可以证明，当表的长度TableSize为质数且表的装载因子 α（α=n/m，表明表的装满程度）不超过0.5 时，新的表项e一定能够插入，而且任何一个位置不会被探查两次。
  - 在搜索时可以不考虑表满情况。
  - 在插入时必须确保表的装填因子α不超过0.5。如果超出必须将表长度扩充一倍，必须将原来的元素重新散列到新表。

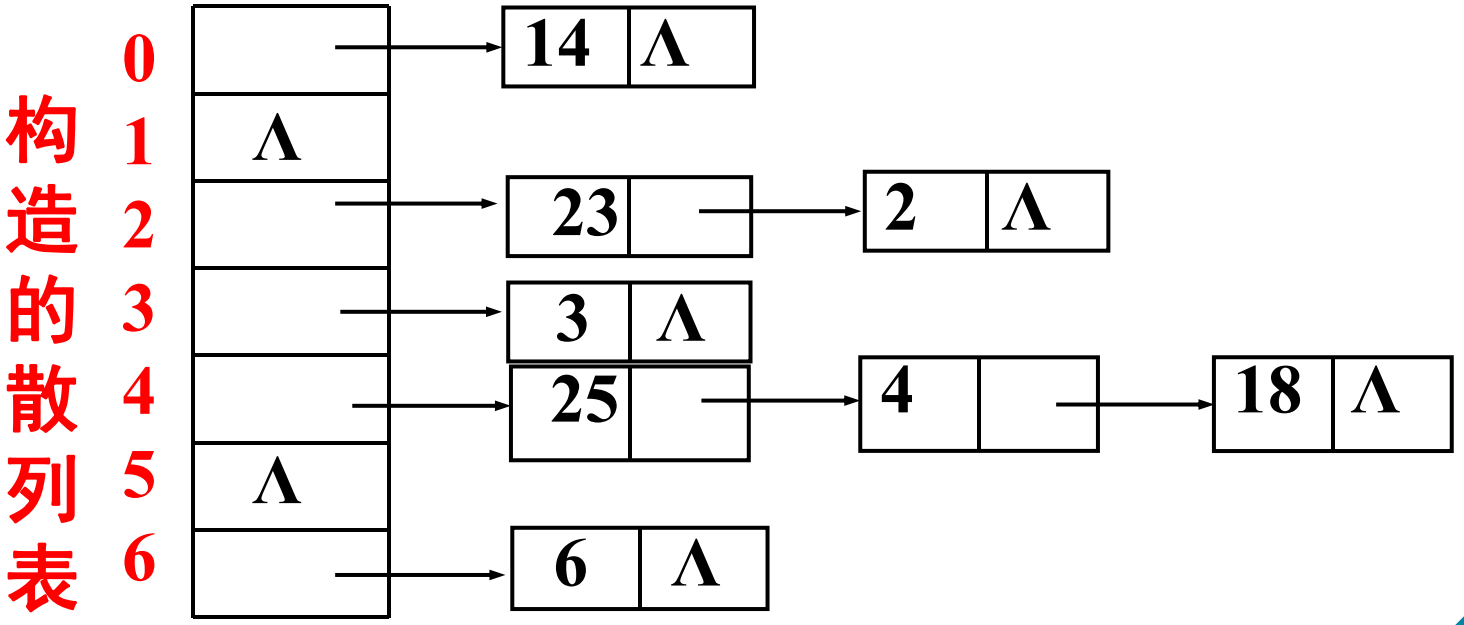
6.5.4. 开散列法

基本思想：为每一个Hash地址建立一个链表，凡散列地址为i的记录都插入到第i个链表中。第i个链表中的所有项的散列值都是i。

例：一组关键字，n=8

	key_1	key_2	key_3	key_4	key_5	key_6	key_7	key_8
key	2	18	4	6	25	23	14	3
H	2	4	4	2	4	2	0	3

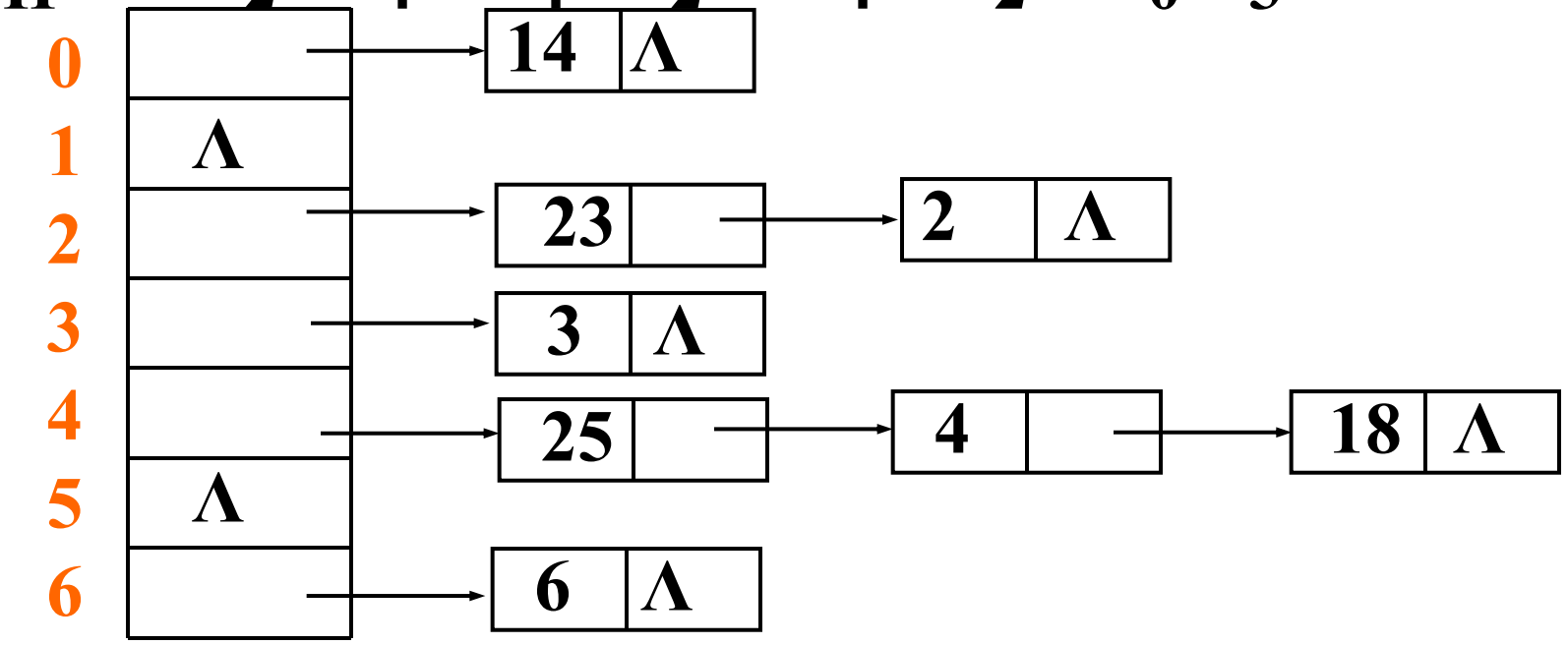
设散列表为ht[7]，散列函数$H(key_i)=key_i \% 7$。



查找算法性能分析

例2：一组关键字：$H(key_i)=key_i \% 7$， 散列表ht[7]

	key_1	key_2	key_3	key_4	key_5	key_6	key_7	key_8
key	2	18	4	6	25	23	14	3
H	2	4	4	2	4	2	0	3



$ASL_{success}=(1*5+2*2+3*1)/8=12/8=1.5$

$ASL_{failure}=(2+1+3+2+4+1+2)/7=15/7$

使用开散列法的散列表类定义P292程序6. 27

```
#include <assert.h>
const int defaultSize = 100;
template <class E, class K>
struct ChainNode {
                        //各桶中同义词子表的链结点定义
    E data;                              //元素
    ChainNode<E, K> *link;    //链指针
};
```

```
template <class E, class K>
class HashTable {     //散列表(表头指针向量)定义
public:
    HashTable (int d, int sz = defaultSize);
                                        //散列表的构造函数
    ～HashTable() { delete [] ht; }          //析构函数
    bool Search (K k1, E& e1);               //搜索
    bool Insert (K k1, E& e1);               //插入
    bool Remove (K k1, E& e1);          //删除
private:
    int divisor;                        //除数（必须是质数）
    int TableSize;      //容量(桶的个数)，无需定义CurrentSize
    ChainNode<E, K> **ht;    //散列表定义
    ChainNode<E, K> *FindPos (K k1);         //查找
    void Resize()  //扩容
};
```

用开散列法定义的散列表的操作

```
template <class E, class K>                    //构造函数
HashTable<E, K>::HashTable (int d, int sz) {
    divisor = d;
    TableSize = sz;
    ht = new ChainNode<E, K>*[sz];//创建头结点数组
    assert (ht != NULL);        //判断存储分配成功否
};
```

```
template <class E, class K>
ChainNode<E, K> *HashTable<E, K>::FindPos(K k1)
{//在散列表ht中搜索关键码为k1的元素。函数返回
//一个指向散列表中某位置的指针
    int j = k1 % divisor;                    //计算散列地址
    ChainNode<E, K> *p = ht[j];
   //遍历指针 p 扫描第j链的同义词子表
    while (p != NULL && p->data != k1) p = p->link;
    return p;                                //返回
};
```

- 其他如插入、删除操作可参照单链表的插入、删除等算法来实现。

开散列解决冲突的哈希查找性能分析

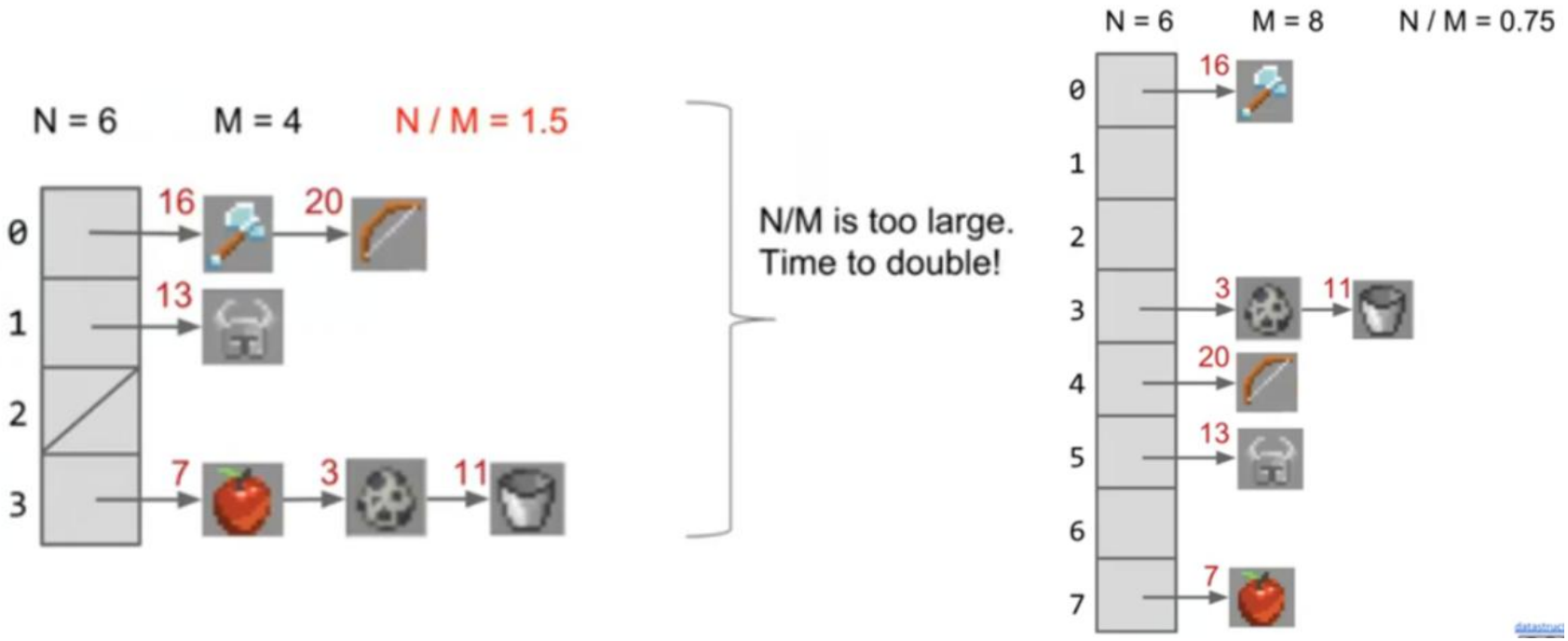
开散列法解决冲突的哈希方法的时间复杂度是O(q)，其中q是最长链表的长度

最好情况：
如果所有项均匀散列到各桶，每个链的长度是n/m，则散列的查找效率是 O（n/m）

最坏情况：
如果各项散列在同一个桶中，最长的链长度是n，则它的查找效率是O(n)

改善方法：扩容。当n/m≥1.5时双倍扩容。扩容时必须重新散列各项。

n动态增加，扩容令m也动态增加，只要m=O(n)，则O(n/m)=O(1)。如果哈希函数可以均匀散列各项，双倍空间扩容使得大多数情况下（当n/m<1.5）链表的平均长度是个常量，即确保插入的大多数情况下性能是O(1)。但是，当n/m=1.5时扩容时刻的性能是O(N)。



散列方法的效率分析

- 衡量标准：插入、删除和查找操作所需要的记录访问次数或判断碑的次数
- 散列表的插入和删除操作都是基于查找进行的
  - 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查找
    - 对于不考虑删除的情况，是尾部的空单元
    - 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
- 查找算法分析
  - 平均查找长度ASL
    - 成功的查找
    - 不成功的查找

装载因子α= n/m

- 散列方法预期的代价与装载因子α= 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单元。
- 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。

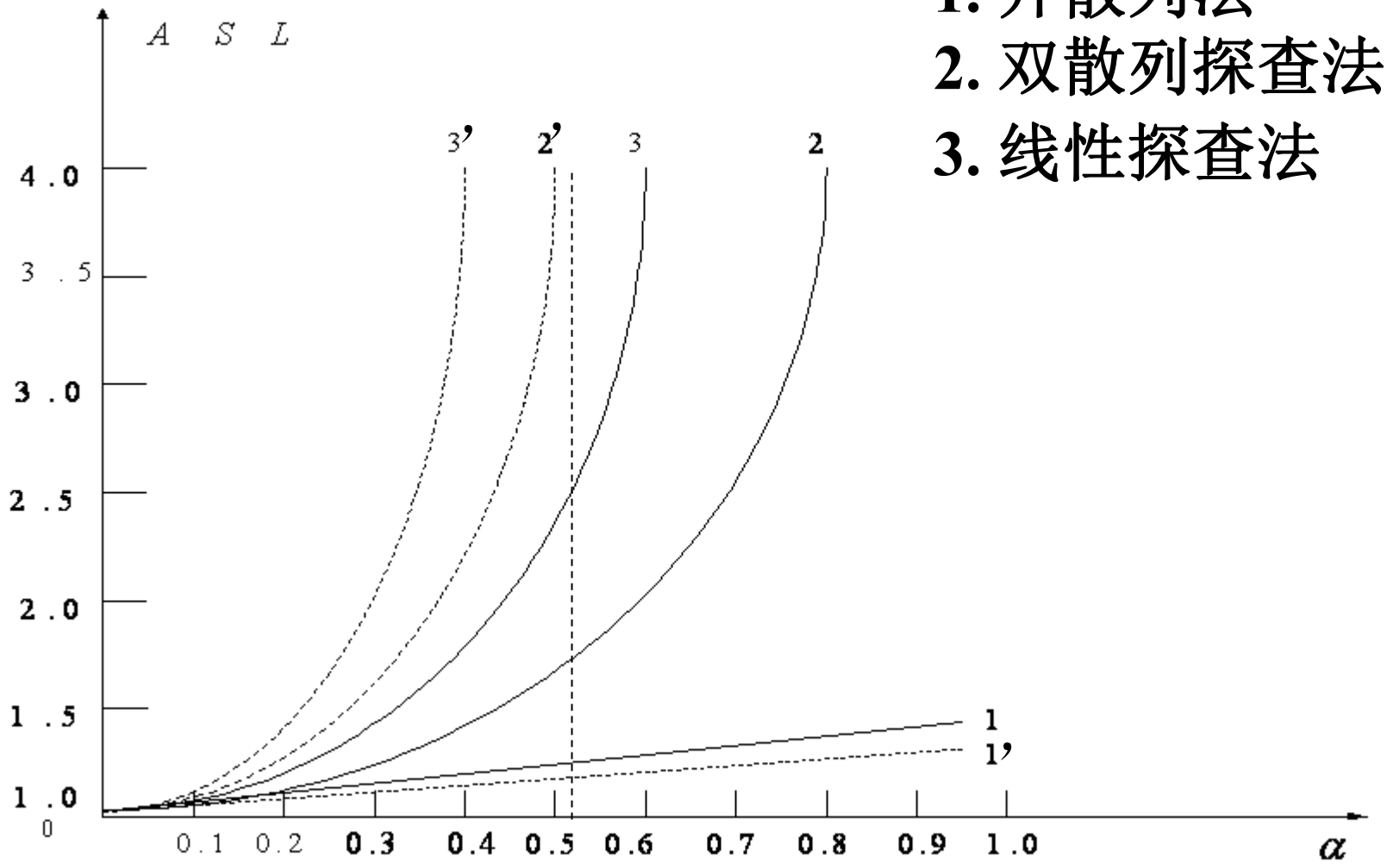
ASL与装载因子α间的关系

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。可以证明有下列结果：

	查找成功的平均查找长度	查找不成功的平均查找长度
线性探查	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
开散列法	$1+\frac{a}{2}$	$\alpha+e^{-\alpha}$

1. 开散列法
2. 双散列探查法
3. 线性探查法



- 用几种不同方法解决冲突时散列表的平均查找长度。实线(1,2,3)显示的是查找成功（或删除）的时间代价ASL，虚线(1’,2’,3’)显示的是查找失败（或插入）的时间代价ASL。

散列方法的效率分析（续）

- 散列方法的代价一般接近于访问一个记录的时间，效率非常高，比需要log *n*次记录访问的二分查找好得多。
  - 不依赖于n，只依赖于装载因子$\alpha=n/m$。
  - 随着α增加，预期的代价也会增加。
  - 分析：α≤0.5时，大部分操作的预期代价都小于2。
- 实际经验也表明散列表装载因子α的临界值是0.5（将近半满）
  - 大于这个临界值，性能就会急剧下降。

散列方法的效率分析（续）

- 散列表的插入和删除操作如果很频繁，将降低散列表的查找效率。
  - 大量的插入操作，将使得装载因子增加；
    - 从而增加了同义词子表的长度，也就是增加了平均查找长度。
  - 大量的删除操作，也将增加碑的数量；
    - 这将增加记录本身到其基地址的平均长度。
  - 实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列，从而提高效率。
    - 把所有记录重新插入到一个新的表中。
      - 清除碑
      - 把最频繁访问的记录放到其基地址。

散列查找小结

- 把数据重新组织到一个散列表中
- 根据关键字的值来确定表中每个记录的位置
- 散列函数的选择
- 冲突策略（探查序列）
- 散列查找的效率
  - ASL不随表目数量的增加而增加；
  - ASL随装载因子α的增大而增加；
  - 如果安排得好，平均查找长度可以小于2。
- 散列法是一种很受欢迎的高效查找方法
- 缺点
  - 不适合进行范围查询
  - 一般也不允许出现重复关键字
- 当散列方法不适合于基于磁盘的应用程序时，我们可以选择B树方法。

Summary

课堂小结

静态搜索结构

- 散列与散列表
- 散列函数
- 处理冲突

搜索算法分析

- 平均搜索长度ASL

Exercises

作业 15

概念题： P295

1、6.9

2、6.10

程序设计：

3、（自选）分别完成使用以下哈希方法实现字典的抽象数据类型的头文件，使用word<单词,词性+释义>类(操作符重载)作为元素类型，完成动态查找的测试程序源文件。

(1)、用线性探测再散列解决冲突，具有插入时若n/m≥0.75则双倍扩容功能（ReSize()方法）。

(2)、用开散列解决冲突，具有插入时若n/m≥1.5则双倍扩容功能（ReSize()方法）。