第十一讲：散列查找

11.1 散列表

11.1.1 散列的基本思路

编译处理时，设计变量及属性的管理：

（1）插入：新变量定义。

（2）查找：变量的引用。

编译处理中对变量管理：**动态查找问题**。

利用查找树进行变量管理，由于两个变量名（字符串）比较效率不高。

我们已知的查找方法：

（1）顺序查找，复杂度 O(N)；

（2）二分查找（静态查找），复杂度 O(logN)

（3）二叉搜索树，复杂度为 O(h)，其中 h 为二叉查找树的高度；

（4）平衡二叉树，复杂度为 O(logN)。

如何快速搜索到需要的关键词呢？如果关键词不方便比较该怎么办呢？

我们回顾下查找的本质：已知对象找到位置。有序安排对象：全序、半序。直接“算出”对象位置：**散列**。

散列查找法的两项基本工作：

（1）计算位置：构造散列函数确定关键词存储位置；

（2）解决冲突：应用某种策略解决多个关键词位置相同的问题。

时间复杂度几乎是常量：O(1)，即查找时间与问题规模无关！

11.1.2 什么是散列表

散列表（哈希表）

类型名称:符号表（SymbolTable）

数据对象集：符号表是“名字(Name)-属性(Attribute)”对的集合。

操作集：对于一个符号表 Table∈SymbolTable，一个给定名字 Name∈NameType，属性 Attr∈AttributeType，以及正整数 TableSize，符号表的基本操作主要有：

1、SymbolTable InitializeTable(int TableSize)：创建一个长度为 TableSize 的符号表；

2、Boolean IsIn(SymbolTable Table, NameType Name)：

查找特定的名字 Name 是否在符号表 Table 中；

3、AttributeType Find(SymbolTable Table, NameType Name)：

获取 Table 中指定名字 Name 对应的属性；

4、SymbolTable Modefy(SymbolTable Table, NameType Name, AttributeType Attr)：

将 Table 中指定名字 Name 的属性修改为 Attr；

5、SymbolTable Insert(SymbolTable Table, NameType Name, AttributeType Attr)：

向 Table 中插入一个新名字 Name 及其属性 Attr；

6、 SymbolTable Delete(SymbolTable Table, NameType Name)：

从 Table 中删除一个名字 Name 及其属性。

【例 1：一位数组】有 n = 11 个数据对象的集合，关键词是正整数，分别为 18，23，11，20，2，7，27，30，42，15，34。

如果符号表的大小用 TableSize = 17（通常用一个素数），选取散列函数 h 如下：

h(key) = key mod TableSize （公式 5.1）

其中 mod 是求余运算，相当于 C 语言中的%运算。

用这个散列函数对 11 个数据对象建立查找表（忽略各关键词对应的属性部分，该例的关键词没有冲突）如下：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键词	34	18	2	20			23	7	42		27	11		30		15	

（1）存放：

h(18)=1，h(23)=6，……。如果新插入 35，h(35)=1，该位置有对象，**冲突！**

（2）查找（先考虑在没有冲突情况下）

查找时，对给定关键词 keyi 依然通过公式 5.1 计算出地址，再将 keyi 与该地址单元中关键词比较，若相等，则查找成功。例如：

key=22，h(22)=5，该地址空，不在表中；

key=30，h(30)=13，该地址存放是 30，找到！

【定义】设散列表空间大小为 m，填入表中的元素个数是 n，则称 α = n/m 为散列表的“装填因子（Loading Factor）”。

装填因子 α = 11 / 17 ≈ 0.65。

实用时，常将散列表大小设计使得 α = 0.5 ~ 0.8 为宜。

[例 2：二维数组]将给定的 10 个 C 语言中的关键词(保留字或标准函数名)顺次存入一张散列表。这 10 个关键词为：acos、define、float、exp、char、atan、ceil、floor、clock、ctime。散列表设计为一个二维数组 Table[26][2]，2 列分别代表 2 个槽，如图 1 所示。

图 1

利用一个二维数组，为例 1 中所提到的冲突提供了一个解决方案。可以将相同的两个元素放到多个槽中。

考虑到这些都是单词，我们在设计散列函数的时候可以以首字母为基准进行，比如设计为：

$$h(key) = key[0] - 'a'$$

如图 2 所示，当放到 clock 和 ctime 的时候，发现产生了冲突，如图 2 所示。

图 2

如果没有溢出，则

$$T_{查询} = T_{插入} = T_{删除} = O(1)$$

总结一下：

“散列（Hashing）”的基本思想是：以数据对象的关键字 key 为自变量，通过一个确定的函数关系 h，计算出对应的函数值 h(key)，把这个值解释为数据对象的存储地址，并按此

存放，即“存储位置 = h（key）”。

散列方法中使用的计算函数称为“散列函数”（也称哈希函数），按这个思想构造的表称为“散列表”，所以它也是一种存储方法。

在查找某数据对象时，用同样的方法“存储位置 = h（key）”计算出地址，将 key 与该地址单元中数据对象关键字进行比较，确定查找是否成功。

可能将**不同的关键字**映射到同一个散列地址上，即 $h(key_i) = h(key_j)$（当 $key_i \neq key_j$），这种现象称为“冲突(Collision)”， key_i 和 key_j 称为“**同义词**（synonym）”。

通常关键词的值域（允许取值的范围）远远大于表空间的地址集，所以说，冲突不可能避免，只能尽可能减少。

11.2 散列函数的构造方法

一个“好”的散列函数一般应考虑下列两个因素：

1. **计算简单**，以便提高转换速度；
2. 关键词对应的**地址空间分布均匀**，以尽量减少冲突。

11.2.1 数字关键词的散列函数构造

1.直接定址法：取关键词的某个线性函数值为散列地址，即 h(key)=a×key+b(a、b 为常数)。

2.除留余数法：散列函数为：h(key) = key mod p。为了地址空间分布均匀，一般 p 取**素数**。

3.数字分析法：分析数字关键字在各位上的变化情况，取比较随机的位作为散列地址。

比如：取 11 位手机号码 key 的后 4 位作为地址：散列函数为：h(key) = atoi(key+7)

4.折叠法：把关键词分割成位数相同的几个部分，然后叠加。

例如 56793542，则做以下处理：（542）+（793）+（056）=1391，取 391 作为关键字。

5.平方取中法：例如 56793542，则计算 56793542×56793542=3225506412905764，那么就取中间的 641 作为关键字。

11.2.2 字符串关键词的散列函数构造

1.一个简单的散列函数——ASCII 码加和法

对字符型关键词 key 定义散列函数如下：

$$h(key) = (\Sigma key[i]) \bmod TableSize$$

这种方法也会有**比较严重的冲突**！

2.简单的改进——前 3 个字符位移法

$$h(key)=(key[0] + key[1]\times 27 + key[2]\times 27^2) \bmod TableSize$$

缺点：**仍然冲突：string、 street、strong、structure 等等；空间浪费：$3000/26^3 \approx 30\%$**

3.好的散列函数——移位法

设计关键词所有 n 个字符，并且分布得很好。如图 3 所示的公式。

图 3

在实际应用这个公式的时候，我们可以考虑在第一节的时候计算多项式的那个简便方法。代码如下：

```
Index Hash ( const char *Key, int TableSize )
{
    unsigned int h = 0;        /* 散列函数值，初始化为 0 */
    while ( *Key != '\0')   /* 位移映射 */
        h = ( h << 5 ) + *Key++;
    return   h % TableSize;
}
```

11.3 冲突处理方法

常用处理冲突的思路：

（1）换个位置：开放地址法；

（2）同一位置的冲突对象组织在一起：链地址法。

11.3.1 开放定址法

所谓开放定址法（Open Addressing）就是一旦产生了冲突（该地址已经有其它元素），就按某种规则去**寻找另一空地址**。

若发生了第 i 次冲突，试探的下一个地址将增加 d_i，基本公式是：

$$h_i(key) = (h(key)+d_i) \bmod TableSize \qquad (1 \le i < TableSize)$$

di 决定了不同的解决冲突方案：线性探测（di=i）、平方探测（$d_i=\pm i^2$）、双散列（$d_i=i*h_2(key)$）。

11.3.2 线性探测

即线性探测法以增量序列 1，2，……，（TableSize-1）循环试探下一个存储地址。

【例】设关键词序列为 {47，7，29，11，9，84，54，20，30}，

散列表表长 TableSize=13，

装填因子 α=9/13 ≈0.69；

散列函数为：h(key) =key mod 11。

用线性探测法处理冲突，列出依次插入后的散列表，并估算查找性能。（1 分钟）

关键词 (key)	**47**	**7**	**29**	**11**	**9**	**84**	**54**	**20**	**30**
散列地址 h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

地址 / 操作	0	1	2	3	4	5	6	7	8	9	10	11	12	说 明
插入 47				47										无冲突
插入 7				47				7						无冲突
插入 29				47				7	29					$d_1 = 1$
插入 11	11			47				7	29					无冲突
插入 9	11			47				7	29	9				无冲突
插入 84	11			47				7	29	9	84			$d_3 = 3$
插入 54	11			47				7	29	9	84	54		$d_1 = 1$
插入 20	11			47				7	29	9	84	54	20	$d_3 = 3$
插入 30	11	30		47				7	29	9	84	54	20	$d_6 = 6$

线性探测会产生聚集现象。

1.散列表查找性能分析

（1）成功平均查找长度（ASLs）。假设要查找的关键词一定在散列表中存在。只要对**查找表中的每个关键词的比较次数加起来，除以关键词的个数**，就得到平均每个关键词的查找长度。而**每个关键词的比较次数是其冲突次数加 1**。

ASL s= （1+7+1+1+2+1+4+2+4）/ 9 = 23/9 ≈ 2.56

（2）不成功平均查找长度（ASLu），即不在散列表中的关键词的平均查找次数（不成功）。一般方法：将不在散列表中的关键词分若干类。如根据 H(key)值分类。查找方式举个例子就明白了，假设我们查找 22,22 mod 11=0，那么从 H(key)=0 开始查找，发现此时没有它，往后移一位还是不对，再后一位是空的，故判断其不在表中，故查找次数为 3。然后所有这些找不到的元素的查找次数之和除以其个数就可以得出其 ASLu。

11.3.3 线性探测——字符串的例子

【例】将 acos/define/float/exp/char/atan/ceil/floor 顺次放入一张大小为 26 的散列表中。H(key)=key[0]-'a'，采用线性探测 di=i。

0	1	2	3	4	5	6	7	……
acos	atan	char	define	exp	float	ceil	floor	……

分析：

ASLs： （1+1+1+1+1+2+5+3）/8=15/8≈**1.87**

ASLu：根据 H(key)值分为 26 种情况，H 值为 0,1,2…,25，则

ASLu=(9+8+7+6+5+4+3+2+1*18)/26=62/26≈**2.38**

11.3.4 平方探测法（二次探测）

即平方探测法以增量序列 1^2，-1^2，2^2，-2^2，…，q^2，$-q^2$ 且 $q \leqslant \lfloor TableSize/2 \rfloor$ 循环试探下一

个存储地址。

【例】设关键词序列为 {47，7，29，11，9，84，54，20，30}

（1）散列表表长 TableSize = 11（即满足 4×2+3 形式的素数）

（2）装填因子 α = 9/11 ≈ 0.82

（3）散列函数为：h(key) = key mod 11

（4）用平方探测法处理冲突，列出依次插入后的散列表并估算 ASLs

关键词 key	47	7	29	11	9	84	54	20	30
散列地址 h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	2	0	3	3

地址 / 操作	0	1	2	3	4	5	6	7	8	9	10	说 明
插入 47				47								无冲突
插入 7				47				7				无冲突
插入 29				47				7	29			$d_1 = 1$
插入 11	11			47				7	29			无冲突
插入 9	11			47				7	29	9		无冲突
插入 84	11			47			84	7	29	9		$d_2 = -1$
插入 54	11			47			84	7	29	9	54	无冲突
插入 20	11		20	47			84	7	29	9	54	$d_3 = 4$
插入 30	11	30	20	47			84	7	29	9	54	$d_3 = 4$

ASLs=（1+1+2+1+1+3+1+4+4）/ 9 = 18/9 = 2

是否有空间，平方探测（二次探测）就能找得到？**这不一定！**

但是有定理显示：**如果散列表长度 TableSize 是某个 4k+3（k 是正整数）形式的素数时，平方探测法就可以探查到整个散列表空间。**

11.3.5 平方探测法的实现

```
typedef struct HashTbl *HashTable;
struct HashTbl{
    int TableSize;
    Cell TheCells;
}H ;
```

```
        HashTable    InitializeTable( int TableSize )
        {
                HashTable H;
                int i;
/* 1*/          if ( TableSize < MinTableSize ){
/* 2*/              Error( "散列表太小" );
/* 3*/              return NULL;
                    }
                 /* 分配散列表 */
/* 4*/          H = malloc( sizeof( struct HashTbl ) );
/* 5*/          if ( H == NULL )
/* 6*/              FatalError( "空间溢出!!!" );
/* 7*/          H->TableSize = NextPrime( TableSize );
                 /* 分配散列表 Cells */
/* 8*/          H->TheCells = malloc( sizeof( Cell ) * H->TableSize );
/* 9*/          if( H->TheCells == NULL )
/*10*/              FatalError( "空间溢出!!!" );
/*11*/          for( i = 0; i < H->TableSize; i++ )
/*12*/              H->TheCells[ i ].Info = Empty;
/*13*/          return H;
        }
        Position    Find( ElementType Key, HashTable H )
        {       Position CurrentPos, NewPos;
                int CNum;           /*    记录冲突次数    */
/* 1*/          CNum = 0;
/* 2*/          NewPos = CurrentPos = Hash( Key, H->TableSize );
/* 3*/          while( H->TheCells[ NewPos ].Info != Empty &&
                                H->TheCells[ NewPos ].Element != Key ) {
                                 /* 字符串类型的关键词需要 strcmp 函数!! */
/* 4*/              if(++CNum % 2){ /*   判断冲突的奇偶次   */
/* 5*/                  NewPos = CurrentPos + (CNum+1)/2*(CNum+1)/2;
/* 6*/                  while( NewPos >= H->TableSize )
/* 7*/                      NewPos -= H->TableSize;
                    } else {
/* 8*/                  NewPos = CurrentPos - CNum/2 * CNum/2;
/* 9*/                  while( NewPos < 0 )
/* 10*/                     NewPos += H->TableSize;
                    }
                }
/* 11*/     return NewPos;
        }
void    Insert( ElementType Key, HashTable H )
{               /* 插入操作 */
                Position Pos;
```

```
/* 1*/        Pos = Find( Key, H );
/* 2*/        if( H->TheCells[ Pos ].Info != Legitimate ) {
                    /* 确认在此插入 */
/* 3*/            H->TheCells[ Pos ].Info = Legitimate;
/* 4*/            H->TheCells[ Pos ].Element = Key;
                                /*字符串类型的关键词需要 strcpy 函数!! */
              }
          }
```

在开放地址散列表中，删除操作要很小心。通常只能“懒惰删除”，即需要增加一个“删除标记(Deleted)”，而并不是真正删除它。以便查找时不会“断链”。其空间可以在下次插入时重用。

另外还有两种方法：

3.双散列探测法

d_i 选为 $i*h_2(key)$，其中 $h_2(key)$是另一个散列函数。我们把它叫做双散列探测法。由此，探测序列成了：$h_2(key)$，$2h_2(key)$，$3h_2(key)$，…

对任意的 key，$h_2(key) \neq 0$

探测序列还应该保证所有的散列存储单元都应该能够被探测到。选择以下形式有良好的效果：

$$h_2(key)=p-(key \bmod p)$$

其中：p < TableSize，p、TableSize 都是素数。

4.再散列

当散列表元素太多（即装填因子α太大）时，查找效率会下降；重新计算，再装入散列表。

11.3.6 分离链接法

分离链接法是解决冲突的另一种方法，其做法是将所有关键词为同义词的数据对象通过结点链接存储在同一个单链表中。

【例】 设关键字序列为 47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 94, 21……

散列函数取为：h(key) = key mod 11，用分离链接法处理冲突。

如图 4 所示。

图 4

该表中有 9 个结点只需 1 次查找

5 个结点需要 2 次查找

因此查找成功的平均查找次数为：

ASL s=（9+5*2）/ 14 ≈ 1.36。

代码如下：

```
struct HashTbl
{
    int TableSize;
    List TheLists;
}H;
struct ListNode;
```

```
typedef struct ListNode *Position, *List;
struct HashTbl;
typedef struct HashTbl *HashTable;
struct ListNode
{
    ElementType Element;
    Position Next;
};
Position    Find( ElementType Key, HashTable H )
{
                    Position P;
                    List L;
/* 1*/          L = &( H->TheLists[ Hash( Key, H->TableSize ) ] );
/* 2*/          P = L->Next;
/* 3*/          while( P != NULL && strcmp(P->Element, Key) )
/* 4*/                P = P->Next;
/* 5*/          return P;
}
```

11.4 散列表的性能分析

关键词的比较次数，取决于产生冲突的多少。

影响产生冲突多少有以下三个因素：

（1）散列函数是否均匀；

（2）处理冲突的方法；

（3）散列表的装填因子 α。

1.线性探测发的查找性能

可以证明，线性探测法的期望探测次数满足下列公式：

图 5

当 α=0.5 时，

插入操作和不成功查找的期望 ASLu=0.5*(1+1/(1-0.5)^2)=2.5 次，

成功查找的期望 ASLs=0.5*(1+1/(1-0.5))=1.5 次。

例子的 α=0.69，于是

期望 ASLu=0.5*(1+1/(1-0.69)^2)=5.70 次

期望 ASLs=0.5*(1+1/(1-0.69))=2.11 次（例中 ASLs=2.56）。

2. 平方探测法和双散列探测法的查找性能

可以证明，平方探测法和双散列探测法探测次数 满足下列公式：

图 6

当 α=0.5 时，

插入操作和不成功查找的期望 ASLu=1/(1-0.5)=2 次，

成功查找的期望 ASLs=-1/0.5*ln(1-0.5)≈1.39 次。

例子的 α=0.82，于是

期望 ASLu=1/(1-0.82)≈5.56 次
期望 ASLs=-1/0.5*ln(1-0.5)≈2.09 次（例中 ASLs=2）。

图 7 表示了期望探测次数与装填因子α的关系。

图 7

当装填因子α<0.5 的时候，各种探测法的期望探测次数都不大，也比较接近。

随着 α 的增大，线性探测法的期望探测次数增加较快，不成功查找和插入操作的期望探测次数比成功查找的期望探测次数要大。

合理的的最大装入因子α应该不超过 0.85。

3. 分离链接法的查找性能

所有地址链表的平均长度定义成装填因子α，α有可能超过 1。
不难证明：其期望探测次数 p 为：

图 8

当 α=1 时，

插入操作和不成功查找的期望 ASLu=1+e^{-1}=1.37 次，

成功查找的期望 ASLs=1+1/2=1.5 次。

我们发现：随着 α 增大，ASLu 增加很慢；而 ASLs 线性增长。

总结一下散列查找：

1.选择合适的 h(key)，散列法的查找效率期望是常数 *O(1),* 它几乎与关键字的空间的大小 n 无关!

2.它是以较小的 α 为前提。因此，散列方法是一个以空间换时间的成功范例。

3.散列方法的存储对关键字是随机的，不便于顺序查找关键字，也不适合于范围查找，或最大值最小值查找。

再来总结下开放地址法和分离链接法：

1.开放地址法：

（1）散列表是一个数组，存储效率高，随机查找。

（2）散列表有“聚集”现象，再散列时有“停顿”现象。

（3）关键字的删除只能采用“懒惰删除”法，会导致存储“垃圾”。

2.分离链接法：

（1）关键字删除不需要“懒惰删除”法，从而没有存储“垃圾”。

(2)散列表是顺序存储和链式存储的结合,链表部分的存储效率和查找效率都比较低。

（3）太小的α可能导致空间浪费，大的α又将付出更多的时间代价。不均匀的链表长度导致时间效率的严重下降。

11.5 应用实例：文件中单词词频统计

【问题】给定一个英文文本文件，统计文件中所有单词出现的频率，并输出词频最大的前 10%的单词及其词频。

为简单起见，假设单词字符定义为大小写字母、数字和下划线，其他字符均认为是单词分隔符，不予考虑。

【分析】解决这个问题的最基本的工作、也是大量的工作是不断对新读入的单词在已有单词表中查找，如果已经存在，则将该单词的词频加 1，如果不存在，则插入该单词并记词频为 1。

使用**散列表**设计该单词表的数据结构可以进行快速地查找和插入

【解决方案】代码如下：

```
            int main() {
/* 1 */    int    TableSize = 10000;    /* 散列表的估计大小 */
            int wordcount = 0,    length;
                HashTable H;
                ElementType word;
                FILE *fp;
/* 2 */     char document[30]= "HarryPotter.txt"; /* 要被统计词频的文件名 */
/* 3 */     H = InitializeTable( TableSize ); /* 建立散列表 */
/* 4 */     if(( fp = fopen(document, "r" ))==NULL)    FatalError("无法打开文件!\n" );
                  while( !feof( fp ) ){
/* 5 */              length = GetAWord( fp, word ); /* 从文件中读取一个单词 */
/* 6 */              if(length > 3){            /* 只考虑适当长度的单词 */
/* 7 */                    wordcount++;
/*查找散列表，若存在，则将该单词的词频加 1，若不存在，则插入。*/
/* 8 */                    InsertAndCount( word, H );
                           }
                  }
                  fclose( fp );
/* 9 */       printf("该文档共出现 %d 个有效单词，", wordcount);
/*根据散列表，输出最频繁出现的给定百分比（10%）的单词。*/
/* 10 */     Show( H ， 10.0/100 );   /* 显示词频前 10%的所有单词 */
/* 11 */     DestroyTable( H );             /* 销毁散列表 */
                  return 0;
}
```