

散列查找

2019年6月30日 21:16

构造散列表

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

$h(\text{key}) = a \times \text{key} + b$ (a 、 b 为常数)

2. 除留余数法

散列函数为: $h(\text{key}) = \text{key} \bmod p$

例: $h(\text{key}) = \text{key} \% 17$

3. 数字分析法

如果关键词 key 是18位的身份证号码:

h

$h(\text{key}) = (\text{key}[6] - '0') \times 104 + (\text{key}[10] - '0') \times 103 + (\text{key}[14] - '0') \times 102 + (\text{key}[16] - '0') \times 10 + (\text{key}[17] - '0')$

$h(\text{key}) = h_1(\text{key}) \times 10 + 10$	(当 $\text{key}[18] = 'x'$ 时) (当 $\text{key}[18]$ 为 '0'~'9' 时)
或	$= h_1(\text{key}) \times 10 + \text{key}[18] - '0'$

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

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: $h(\text{key}) = \text{atoi}(\text{key} + 7)$ ($\text{char} * \text{key}$)

4. 折叠法

把关键词分割成位数相同的几个部分，然后叠加

如: 56793542

542

793

+ 056

1391

$h(56793542) = 391$

5. 平方取中法

如: 56793542

56793542

x 56793542

3225506412905764

$h(56793542) = 641$

	14	15	16	17	18
	8	0	4	1	9

❖ 字符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：

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

2. 简单的改进——前3个字符移位法

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

3. 好的散列函数——移位法

涉及关键词所有 n 个字符，并且分布得很好：

如何快速计算：

$$h("abcde") = 'a' \times 32^4 + 'b' \times 32^3 + 'c' \times 32^2 + 'd' \times 32 + 'e'$$

```
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;
}
```


❖ 处理冲突的方法

常用处理冲突的思路：

❑ 换个位置：开放地址法


❑ 同一位置的冲突对象组织在一起：链地址法

❖ 开放定址法（Open Addressing）

 若发生了第 i 次冲突，试探的下一个地址将增加 di ，基本公式是：

h

$$h_i(key) = (h(key) + di) \bmod TableSize \quad (1 \leq i < TableSize)$$

 di 决定了不同的解决冲突方案：线性探测、平方探测、双散列。

1. 线性探测法（Linear Probing）

❖ 线性探测法：以增量序列 $1, 2, \dots, (TableSize - 1)$

循环试探下一个存储地址。

聚集现象

散列表查找性能分析

- 成功平均查找长度(ASL_s)
- 不成功平均查找长度 (ASL_u)

散列表:

H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12
key	11	30		47				7	29	9	84	54	20
冲突次数	0	6		0				0	1	0	3	1	3

【分析】

ASL_s: 查找表中关键词的平均查找比较次数 (其冲突次数加1)

$$ASL_s = (1+7+1+1+2+1+4+2+4) / 9 = 23/9 \approx 2.56$$

ASL_u: 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

$$ASL_u = (3+2+1+2+1+1+1+9+8+7+6) / 11 = 41/11 \approx 3.73$$

【例】 将acos、define、float、exp、char、atan、ceil、floor, 顺次存入一张大小为26的散列表中。

H(key)=key[0]-'a', 采用线性探测d_i=i.

acos	atan	char	define	exp	float	ceil	floor		
0	1	2	3	4	5	6	7	8		25

【分析】

ASL_s: 表中关键词的平均查找比较次数

$$ASL_s = (1+1+1+1+1+2+5+3) / 8 = 15/8 \approx 1.87$$

ASL_u: 不在散列表中的关键词的平均查找次数 (不成功)

根据H(key)值分为26种情况: H值为0,1,2,...,25

$$ASL_u = (9+8+7+6+5+4+3+2+1 \times 18) / 26 = 62/26 \approx 2.38$$

2. 平方探测法 (Quadratic Probing) --- 二次探测

❖ 平方探测法: 以增量序列12, -12, 22, -22,, q², -q² 且 $q \leq \lfloor \text{TableSize}/2 \rfloor$ 循环试探下一个存储

2. 平方探测法 (Quadratic Probing)

是否有空间, 平方探测(二次探测)就能找得到?

5	6	7		
0	1	2	3	4

$$h(k) = k \bmod 5$$

插入 11, $h(11)=1$

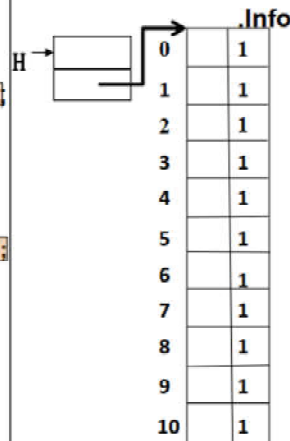
探测序列: $1+1=2$, $1-1=0$, $(1+2^2) \bmod 5=0$, $(1-2^2) \bmod 5=2$,
 $(1+3^2) \bmod 5=0$, $(1-3^2) \bmod 5=2$, $(1+4^2) \bmod 5=2, \dots$

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测到整个散列表空间。


```
HashTable InitializeTable( int TableSize )
```

```
{
    HashTbl H;
    int i;
    if ( TableSize < MinTableSize ){
        Error( "散列表太小" );
        return NULL;
    }
    /* 分配散列表 */
    H = (HashTbl)malloc( sizeof( struct HashTbl ) );
    if ( H == NULL )
        FatalError( "空间溢出!!!" );
    H->TableSize = NextPrime( TableSize );
    /* 分配散列表 Cells */
    H->TheCells=(Cell *)malloc(sizeof( Cell )*H->TableSize);
    if ( H->TheCells == NULL )
        FatalError( "空间溢出!!!" );
    for( i = 0; i < H->TableSize; i++ )
        H->TheCells[ i ].Info = Empty;
    return H;
}
```

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



```
Position Find( ElementType Key, HashTable H ) /*平方探测*/
```

```
{
    Position CurrentPos, NewPos;
    int CNum; /* 记录冲突次数 */
    CNum = 0;
    NewPos = CurrentPos = Hash( Key, H->TableSize );
    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!! */
        if(++CNum % 2){ /* 判断冲突的奇偶次 */
            NewPos = CurrentPos + (CNum+1)/2*(CNum+1)/2;
            while( NewPos >= H->TableSize )
                NewPos -= H->TableSize;
        } else {
            NewPos = CurrentPos - CNum/2 * CNum/2;
            while( NewPos < 0 )
                NewPos += H->TableSize;
        }
    }
    return NewPos;
}
```

d_i	$+1^2$	-1^2	$+2^2$	-2^2	$+3^2$	-3^2
Cnum	1	2	3	4	5	6	


```

void Insert( ElementType Key, HashTable H )
{
    /* 插入操作 */
    Position Pos;
    Pos = Find( Key, H );
    if( H->TheCells[ Pos ].Info != Legitimate ) {
        /* 确认在此插入 */
        H->TheCells[ Pos ].Info = Legitimate;
        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**。

3. 双散列探测法 (Double Hashing)

双散列探测法: di 为 $i \cdot h_2(\text{key})$ ， $h_2(\text{key})$ 是另一个散列函数

探测序列成: $h_2(\text{key})$, $2h_2(\text{key})$, $3h_2(\text{key})$,

对任意的 key , $h_2(\text{key}) \neq 0$!

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

$h_2(\text{key}) = p - (\text{key} \bmod p)$

其中: $p < \text{TableSize}$, p 、 TableSize 都是素数。

4. 再散列 (Rehashing)

□ 当散列表元素太多 (即装填因子 α 太大) 时, 查找效率会下降;

□ 当装填因子过大时, 解决的方法是加倍扩大散列表, 这个过程叫做“**再散列 (Rehashing)**”

实用最大装填因子一般取 $0.5 \leq \alpha \leq 0.85$

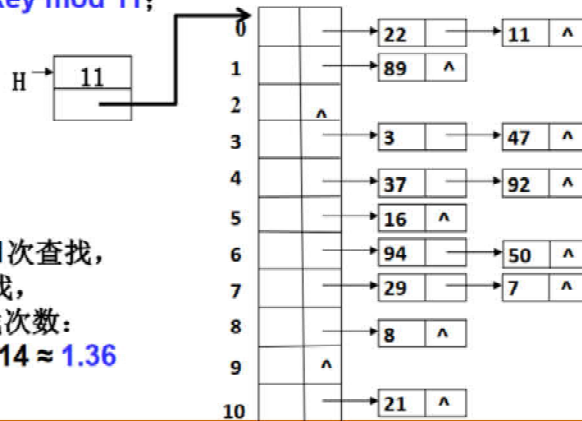
❖ 分离链接法 (Separate Chaining)

分离链接法：将相应位置上冲突的所有关键词存储在**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： $h(key) = key \bmod 11$;
用分离链接法处理冲突。

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

- 表中有9个结点只需1次查找，
- 5个结点需要2次查找，
- 查找成功的平均查找次数：
 $ASL_s = (9 + 5 \times 2) / 14 \approx 1.36$



```
typedef struct ListNode *Position, *List;
struct ListNode {
    ElementType Element;
    Position Next;
};
typedef struct HashTbl *HashTable;
struct HashTbl {
    int TableSize;
    List TheLists;
};
```

```
Position Find( ElementType Key, HashTable H )
{
    Position P;
    int Pos;

    Pos = Hash( Key, H->TableSize ); /*初始散列位置*/
    P = H->TheLists[Pos]. Next;      /*获得链表头*/
    while( P != NULL && strcmp(P->Element, Key) )
        P = P->Next;
    return P;
}
```

