数据结构与算法

散列

彭振辉
中山大学人工智能学院

本章大纲

1. 散列表
2. 散列函数构造
3. **冲突处理**
4. **应用实例**

上节回顾：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：

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

  冲突严重：a3、 b2、c1；eat、 tea；

- 2．简单的改进——前3个字符移位法

  改造如下：

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

- 3．好的散列函数——移位法

  涉及关键词的所有n个字符，并且分布得很好：

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

  注：前三个字符的不同组合在实际使用中大概有3000种可能

$$\mathbf{h(key)} = \left(\sum_{\mathbf{i=0}}^{\mathbf{n-1}} \mathbf{key[n-i-1]\times 32^i} \right) \mod \mathbf{TableSize}$$

思考：为什么法2是x27，法3是x32?

上节回顾：1. 线性探测法

- 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,
- 问题1：装填因子 α = 9/13 ≈ 0.69；
- 散列函数为：h(key) = key mod 11。
- 问题2：用线性探测法处理冲突，列出依次插入后的散列表
- 问题3：并估算查找性能。

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

思考：未解决冲突前，散列地址h(key)分别是多少?

上节回顾： 1. 线性探测法

- 成功平均查找长度 (ASLs)
- 不成功平均查找长度 (ASLu)

思考：查找操作时每个散列地址的冲突次数怎么计算的？

散列表： 关键词序列为 {47, 7, 29, 11, 9, 84, 54, 20, 30}

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
查找冲突次数	0	6		0				0	1	0	3	1	3

[分析]

ASLs：查找表中关键词的平均查找比较次数（其冲突次数加1）

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

ASLu：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(0-10)分类

ASLu= （3+2+1+2+1+1+1+9+8+7+6）/ 11 **思考：这个ASLu是怎么算的呢？**

= 41/11 ≈ 3.73

上节回顾：2. 平方探测法

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**

ASL s = （1+1+2+1+1+3+1+4+4）/ 9 = 18/9 = 2

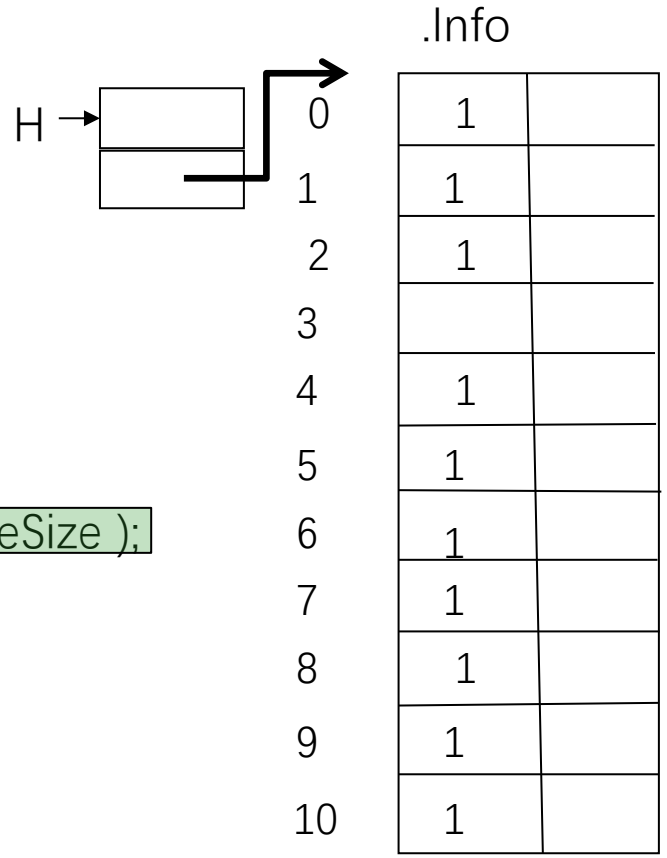
- 即平方探测法以增量序列1^2，-1^2，2^2，-2^2，…，q^2，$-q^2$且$q \leqslant \lfloor TableSize/2 \rfloor$循环试探下一个存储地址。

操作 \ 地址	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	$d_3 = 4$

2. 平方探测法

```
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;
    }
```

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



示例：.Info为1的位置不为空，为0的位置为空

2. 平方探测法

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

d_i	$+1^2$	-1^2	$+2^2$	-2^2	$+3^2$	-3^2 ….
CNum	1	2	3	4	5	6

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

思考1：为什么除以2?

思考2：如果Key不在HashTable中，返回什么？

2. 平方探测法

```
void  Insert( ElementType Key, HashTable H )
{        /* 插入操作 */
          Position Pos;
/* 1*/    Pos = Find( Key, H );      思考：如果Key不在HashTable H中，Pos是什么？
/* 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!! */
        }
    }
```

Legitimate 合法的；
在这里表示非空

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

- 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；
  - 实用最大装填因子一般取 0.5 <= α<= 0.85
- 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）”

4. 再散列

- 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）”

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.19 Hash table with linear probing with input 13, 15, 6, 24

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 5.20 Hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 5.21 Hash table after rehashing

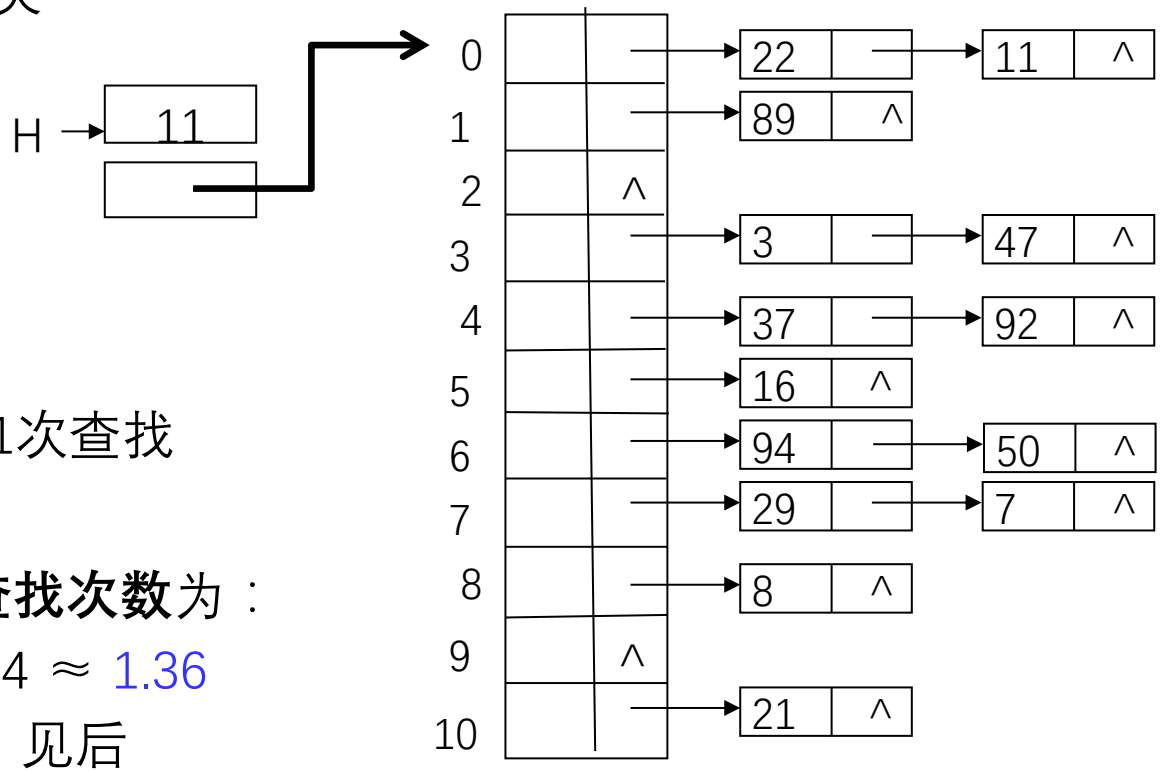
分离链接法（Separate Chaining）

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 mod 11
- 用分离链接法处理冲突

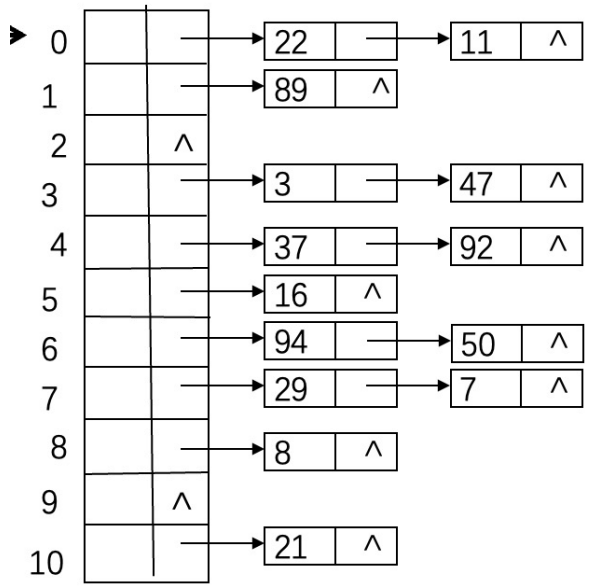
```
struct HashTbl
{
    int TableSize;
    List TheLists;
}H;
```



- 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
  - ASLu估算比较复杂，见后

分离链接法（Separate Chaining）

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



思考：如果要找的Key是47，这个函数的执行过程是什么样的？

```
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;
}
```

4.4 散列表的性能分析

- 平均查找长度（ASL）用来度量散列表**查找效率**。
- 另一方面，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：
  - **散列函数是否均匀；**
  - **处理冲突的方法；**
  - 散列表的**装填因子α**。
- 分析：不同冲突处理方法、装填因子对效率的影响

1. 线性探测法的查找性能

可以证明，线性探测法的期望探测次数满足下列公式：

$$p=\begin{cases}\dfrac{1}{2}\left[1+\dfrac{1}{(1-\alpha)^2}\right] & \text{（对插入和不成功查找而言）}\\ \dfrac{1}{2}\left(1+\dfrac{1}{1-\alpha}\right) & \text{（对成功查找而言）}\end{cases}$$

- 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

- 当α= 0.69时，

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：

$$p = \begin{cases} \dfrac{1}{1-\alpha} & \text{（对插入和不成功查找而言）} \\ -\dfrac{1}{\alpha}\ln(1-\alpha) & \text{（对成功查找而言）} \end{cases}$$

- 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。

- 当α= 0.82时，

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）。

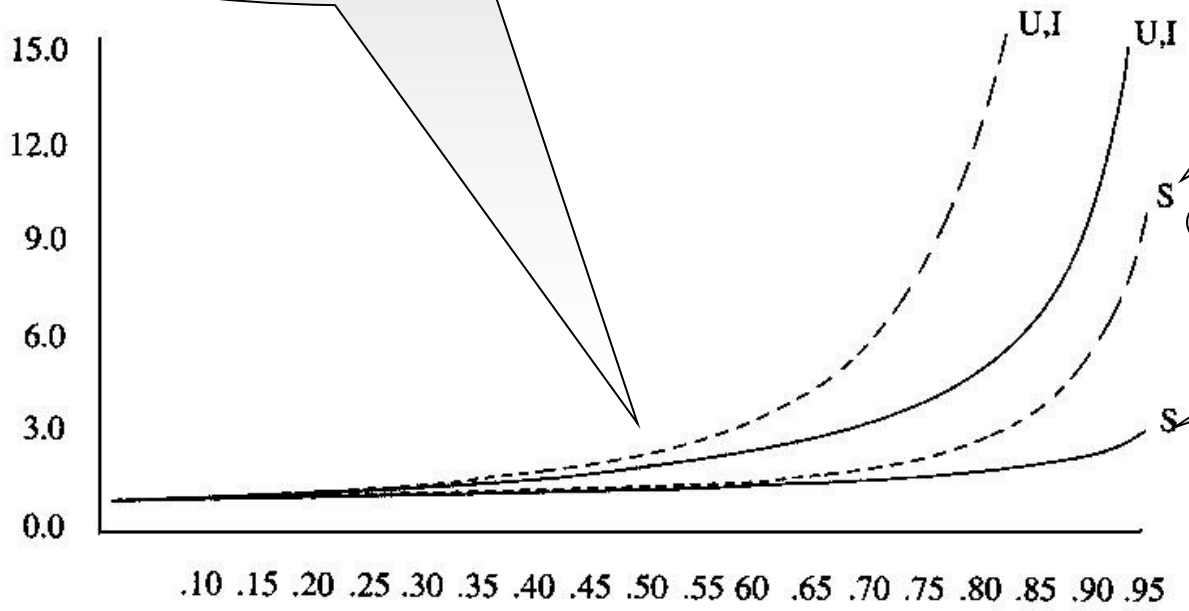
2. 平方探测法和双散列探测的查找性能

期望探测次数与装填因子α的关系

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。



线性探测法（虚线）、**双散列探测法**（实线）

U表示不成功查找，I表示插入，S表示成功查找

3. 分离链表法的查找性能

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。
可以证明：其期望探测次数 p为：

$$p = \begin{cases} \alpha + e^{-\alpha} & \text{（对插入和不成功查找而言）} \\ 1 + \frac{\alpha}{2} & \text{（对成功查找而言）} \end{cases}$$

随着α增大，ASLu增加很慢；而ASLs线性增长。

- 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。

- 前面的例子中14个元素分布在11个单链表中，所以α= 14/11≈1.27，

期望 ASLu = 1.27 + $e^{-1.27}$ ≈ 1.55次
期望 ASLs = 1 + 1.27 / 2 ≈ 1.64次（例中ASLs = 1.36）

3. 散列方法的查找效率

- 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！
- 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。
- 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**。

开放地址法 vs.分离链法

- 开放地址法：
  - 散列表是一个数组，存储效率高，随机查找。
  - 散列表有“聚集”现象，再散列时有“停顿”现象。
- 分离链法：
  - 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**。
  - 关键字删除不需要“懒惰删除”法，从而没有存储“垃圾”。
  - 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。

$$p = \begin{cases} \alpha + e^{-\alpha} & \text{（对插入和不成功查找而言）} \\ 1 + \dfrac{\alpha}{2} & \text{（对成功查找而言）} \end{cases}$$

5.4 应用实例1：文件中单词词频统计

[例] 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] 关键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。

核心问题：

如何设计该单词表的数据结构才可以进行快速地查找和插入？

5.4 应用实例1：文件中单词词频统计

```
      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++;
/* 8 */            InsertAndCount( word, H );
              }
           }
           fclose( fp );
/* 9 */   printf("该文档共出现 %d 个有效单词，", wordcount);
/* 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;
}
```

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。

根据散列表，输出最频繁出现的给定百分比（10%）的单词。

5.4 应用实例2：电话聊天狂人

问题描述：给定大量手机用户通话记录，找出其中通话次数最多的聊天狂人。

输入样例:
4
13005711862 13588625832
13505711862 13088625832
13588625832 18087925832
15005713862 13588625832

输入首先给出正整数N（≤10^5），为通话记录条数。

随后N行，每行给出一条通话记录。简单起见，这里只列出拨出方和接收方的11位数字构成的手机号码，其中以空格分隔。

输出样例：13588625832 3

在一行中给出聊天狂人的手机号码及其通话次数，其间以空格分隔。如果这样的人不唯一，则输出狂人中最小的号码及其通话次数，并且附加给出并列狂人的人数。

13005711862	1
13588625832	3
13505711862	1
13088625832	1
18087925832	1
15005713862	1

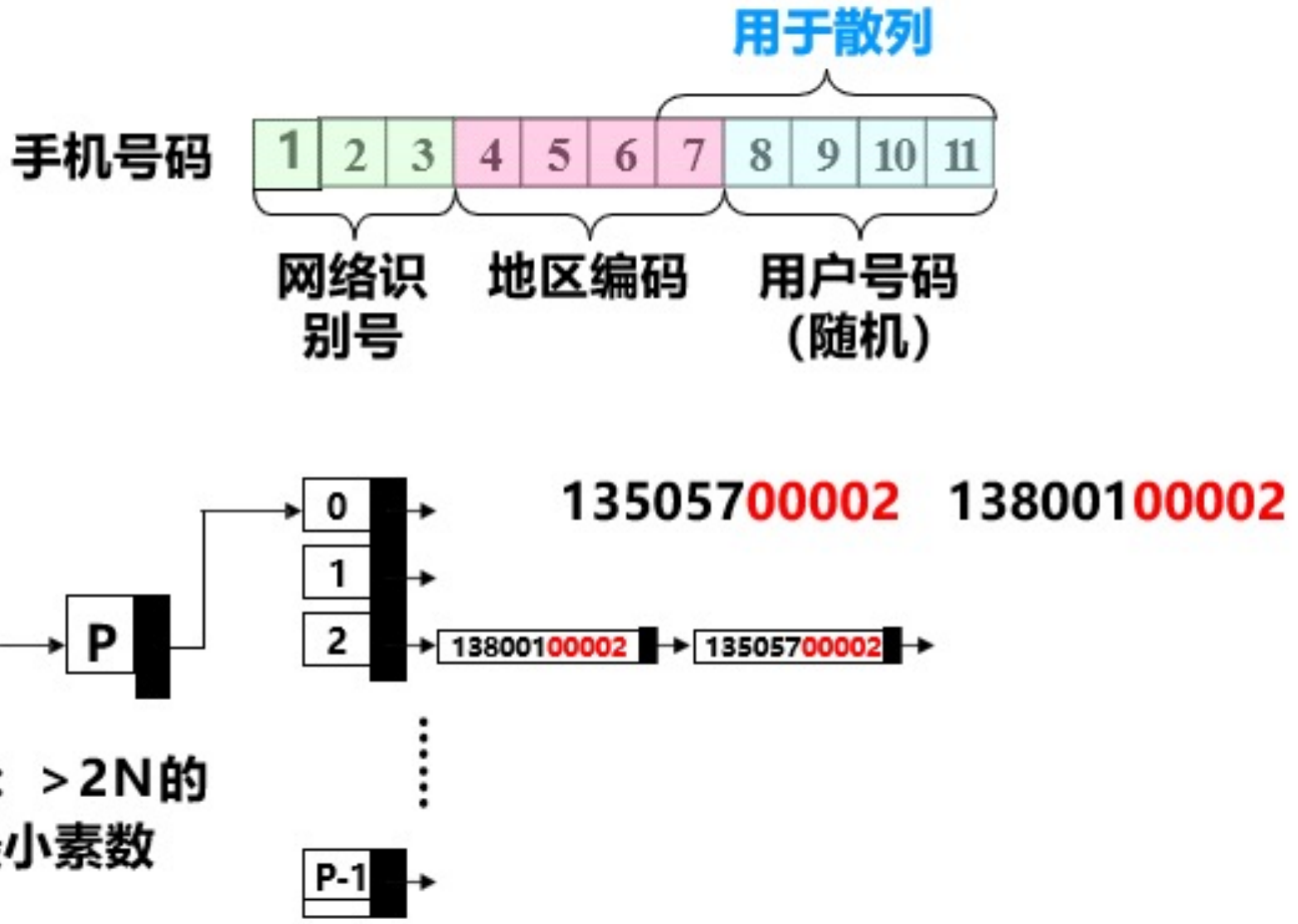
解法1：排序

- 第1步：读入最多$2*10^5$个电话号码，每个号码存为长度为11的字符串
- 第2步：按字符串非递减顺序排序
- 第3步：扫描**有序数组**，累计同号码出现的次数，并且更新最大次数
  - 编程简单快捷
  - **无法拓展解决动态插入的问题**

解法2：直接映射

- 第1步：创建有2*10^{10}个单元的整数数组，**保证每个电话号码对应唯一的单元下标**；数组初始化为0
- 第2步：对读入的每个电话号码，找到以之为下标的单元，数值累计1次
- 第3步：顺序扫描数组，找出累计次数最多的单元
  - 编程简单快捷，动态插入快
  - 下标超过了unsigned long (0 ~ （2的32次方-1)
    - 需要 2*10^{10}*2 bytes ≈ **37GB**
    - 为了 2 * 10^{5}个号码扫描2*10^{10}个单元

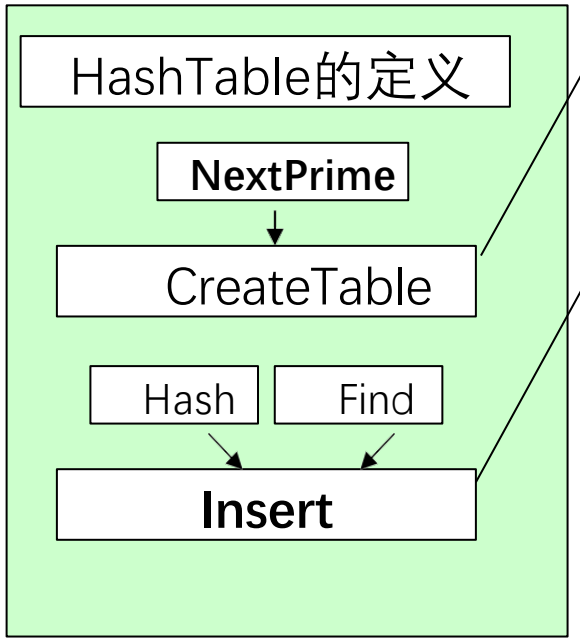
解法3：带智商的散列



N指输入的通话记录有N对

程序框架搭建

```
int main()
{
    创建散列表;
    读入号码插入表
    中; 扫描表输出
    狂人;  return 0;
}
```



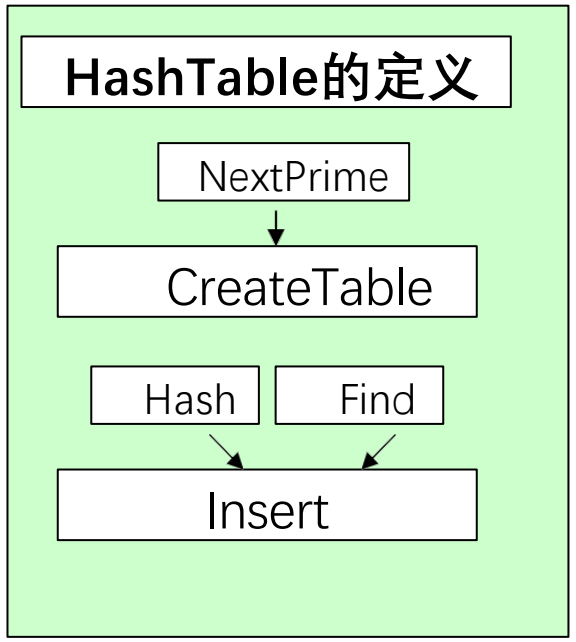

```
int main()
 {   int N, i;
     ElementType Key;
     HashTable H;
     scanf("%d", &N);
     H = CreateTable(N*2); /*创建一个散列表 */
     for (i=0; i<N; i++) {
          scanf("%s", Key); Insert( H, Key );
          scanf("%s", Key); Insert( H, Key );
     }
     ScanAndOutput( H );
     DestroyTable( H );
     return 0;
  }
```

扫描整个散列表

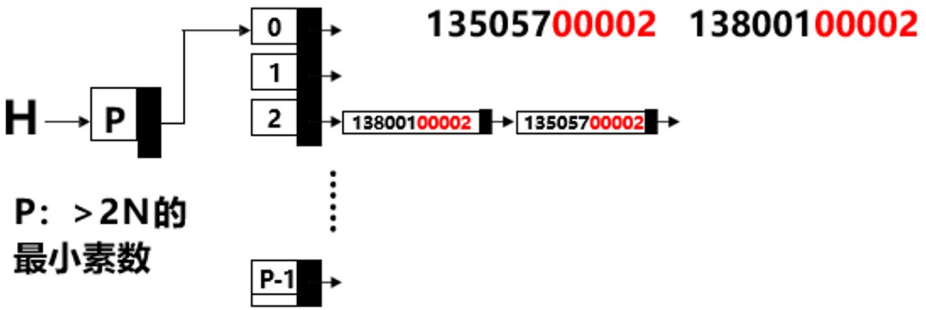
更新最大通话次数；

更新最小号码 + 统计人数；

模块的引用和裁剪

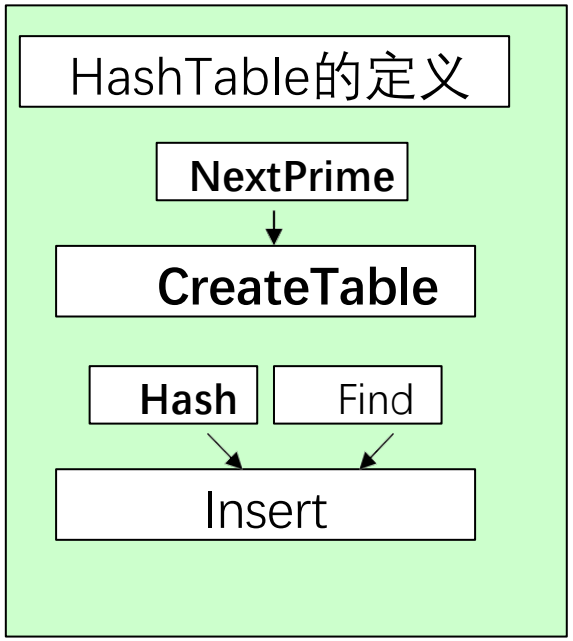



```
#define KEYLENGTH 11 /* 关键词字符串的最大长度 */
                     /* 关键词类型用字符串 */
typedef char ElementType[KEYLENGTH+1];
typedef int Index;          /* 散列地址类型 */
typedef struct LNode *PtrToLNode;  struct LNode {
      ElementType Data;
      PtrToLNode Next;
      int Count;
};
typedef PtrToLNode Position;
typedef PtrToLNode List;
```




```
typedef struct TblNode *HashTable;
 struct TblNode {  /* 散列表结点定义 */
      int TableSize; /* 表的最大长度 */
      List Heads;     /* 指向链表头结点的数组 */
};
```

模块的引用和裁剪



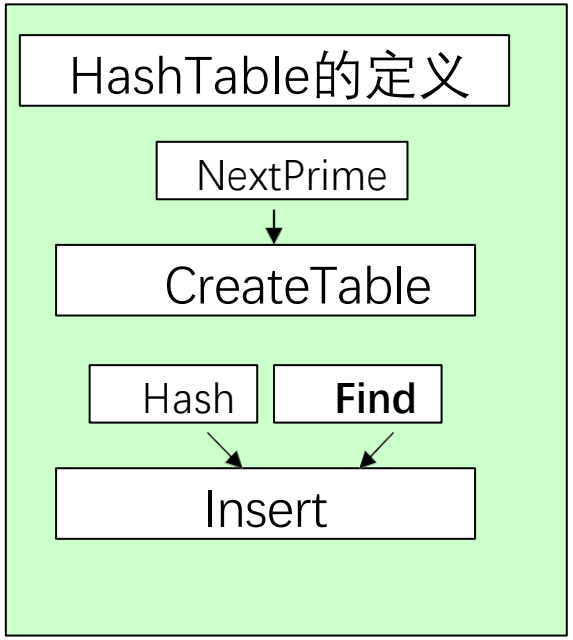

```
#define MAXTABLESIZE 1000000
int NextPrime( int N )
{ /* 返回大于N且不超过MAXTABLESIZE的最小素数 */
   int i, p = (N%2)? N+2 : N+1; /*从大于N的下一个奇数开始 */
   while( p <= MAXTABLESIZE ) {
      for( i=(int)sqrt(p); i>2; i-- )
         if ( !(p%i) ) break; /* p不是素数 */
      if ( i==2 ) break; /* for正常结束，说明p是素数 */
      else  p += 2; /* 否则试探下一个奇数 */
   }
   return p;
}
```

```
int Hash ( int Key, int P )
{ /* 除留余数法法散列函数 */
   return Key%P;
}
```

```
HashTable CreateTable( int TableSize )
{
   HashTable H;  int i;
   H = (HashTable)malloc(sizeof(struct TblNode));
   H->TableSize = NextPrime(TableSize);
   H->Heads = (List)malloc(H->TableSize*sizeof(struct LNode));
   for( i=0; i<H->TableSize; i++ ) {
      H->Heads[i].Data[0] = '\0';
      H->Heads[i].Next = NULL;
      H->Heads[i].Count = 0;
   }
   return H;  }
```

拓展阅读，判断是不是素数的方法：
https://cloud.tencent.com/developer/article/2053474

模块的引用和裁剪



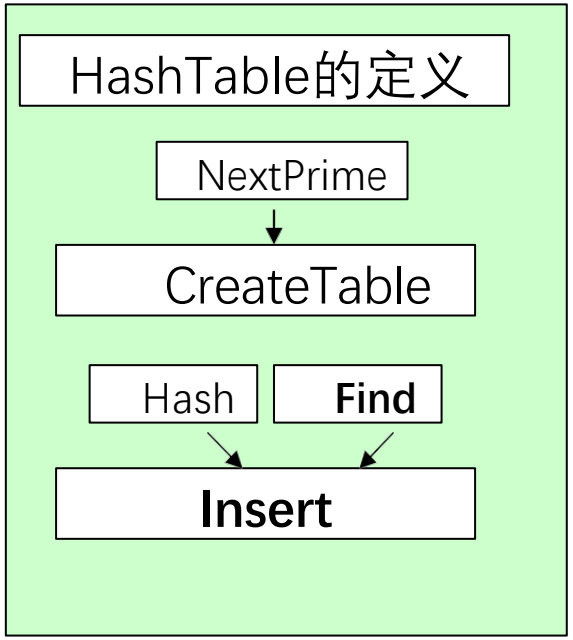

```
Position Find( HashTable H, ElementType Key )
{
    Position P;  Index Pos;
    /* 初始散列位置 */
    Pos = Hash( Key, H->TableSize );
    /* 如果Key是字符串的电话号码，适用于此例*/
    //将电话号码的后5位作为转为数字作为键值
    Pos = Hash(atoi(Key+KEYLENGTH-MAXD), H->TableSize);
    P = H->Heads[Pos].Next; /* 从该链表的第1个结点开始 */
    /* 当未到表尾，并且Key未找到时 */
     while( P && strcmp(P->Data, Key) )
        P = P->Next;

    return P; /* 此时P或者指向找到的结点，或者为NULL */
}
```

注1： MAXD 5 /*参与散列映射计算的字符数*/

注2：atoi (表示ascii to integer)是把字符串转换成整型数的一个函数

模块的引用和裁剪




```
bool Insert( HashTable H, ElementType Key )
{   Position P, NewCell;  Index Pos;
    P = Find( H, Key );
    if ( !P ) { /* 关键词未找到，可以插入 */
        NewCell = (Position)malloc(sizeof(struct LNode));
        strcpy(NewCell->Data, Key);
        /* 初始散列位置 */
        Pos = Hash( Key, H->TableSize );

        NewCell->Count = 1;
        Pos = Hash(atoi(Key+KEYLENGTH-MAXD), H->TableSize);
        /* 将NewCell插入为H->Heads[Pos]链表的第1个结点 */
        NewCell->Next = H->Heads[Pos].Next;
        H->Heads[Pos].Next = NewCell;  return true;
    }
    else { /* 关键词已存在 */

        printf("键值已存在");
        P->Count++;
        return false;
    }
}
```

一个完整代码示例： https://www.cnblogs.com/ranjiewen/p/6872057.html

输出狂人

```
void ScanAndOutput( HashTable H )
{   int i, MaxCnt = PCnt = 0;
    ElementType MinPhone;
    List Ptr;
    MinPhone[0] = '\0';
    for (i=0; i<H->TableSize; i++) { /* 扫描链表 */
        Ptr = H->Heads[i].Next;
        while (Ptr) {
        if (Ptr->Count > MaxCnt) { /* 更新最大通话次数 */
            MaxCnt = Ptr->Count;
            strcpy(MinPhone, Ptr->Data);
            PCnt = 1;
        }
        else if (Ptr->Count == MaxCnt) {
            PCnt ++; /* 狂人计数 */
            if ( strcmp(MinPhone, Ptr->Data)>0 )
                strcpy(MinPhone, Ptr->Data); /* 更新狂人的最小手机号码 */
        }
        Ptr = Ptr->Next;
      }
    }
    printf("%s %d", MinPhone, MaxCnt);
     if ( PCnt > 1 ) printf(" %d", PCnt);  printf("\n");
}
```

扫描整个散列表

更新最大通话次数；

更新最小号码 + 统计人数；