Chapter 5 Search

5.1 Concept of Search

- 查找：就是在数据集合中寻找满足某种条件的数据对象。
- 查找表：是由同一类型的数据元素(或记录)组成的数据集合。

数据元素之间的逻辑关系：没有内在的逻辑约束，是数据集合。

- 关键字：数据元素中某个数据项的值，用以标识一个数据元素。
- 主关键字：可唯一地标识一个数据元素的关键字。
- 次关键字：用以识别若干记录的关键字。
- 使用基于主关键字的查找，查找结果应是唯一的。

- 查找的结果通常有两种可能：
  - 查找成功，即找到满足条件的数据对象。
  - 查找不成功，或查找失败。作为结果，报告一些信息，如失败标志、失败位置等。

■ 查找算法设计与分析

数据元素之间的逻辑关系：没有内在的逻辑约束，是数据集合。

- ✓ 数据的组织 — 数据的存储设计
- ✓ 查找算法设计 — 数据的查找算法
- ✓ 算法性能分析

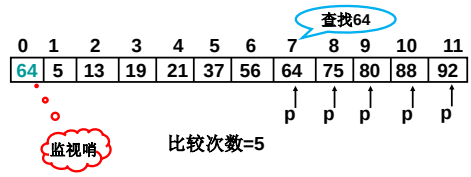
- 衡量一个查找算法的时间效率的标准是：在查找过程中关键字的平均比较次数或平均读写磁盘次数(只适合于外部查找)，这个标准也称为平均查找长度*ASL*(*Average Search Length*)，通常它是查找结构中对象总数 n 或文件结构中物理块总数 n 的函数。
- 另外衡量一个查找算法还要考虑算法所需要的存储量和算法的复杂性等问题。

5.2 Search in Sequence list

- 存储结构：连续设计方式、链接设计方式

查找过程：从表中最后一个元素开始，顺序用各元素的关键字与给定值x进行比较，若找到与其值相等的元素，则查找成功，给出该元素在表中的位置；否则，若直到第一个记录仍未找到关键字与x相等的对象，则查找失败。

顺序查找示例




```
int  Seq_Search(SSTable  ST , KeyType key)
  {  int p ;
     ST. elem[0].key=key ;   /* 设置监视哨兵,失败返回0 */
     for (p=ST.length; !EQ(ST. elem[p].key, key); p--)
     return(p) ;
  }
```

比较次数：

查找第n个元素：　1

……

查找第i个元素：　$n-i+1$

查找第1个元素：　n

查找失败：　$n+1$

顺序查找的平均查找长度

设查找第 i 个元素的概率为 p_i，查找到第 i 个元素所需比较次数为 c_i，则查找成功的平均查找长度:

$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot c_i . \qquad (\sum_{i=1}^{n} p_i = 1)$$

在顺序查找情形，$c_i = n-i+1,\ i = 1, \ldots, n$，因此

$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot (n - i + 1)$$

在等概率情形，$p_i = 1/n,\ i = 0, 1, \ldots, n-1$。

$$ASL_{succ} = \sum_{i=1}^{n} \frac{1}{n}(n - i + 1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

失败长度：n+1

顺序查找的时间复杂性　$O(n)$

插入、删除的时间复杂性　O(1)

5.3 折半查找(Binary Search)

折半查找又称为二分查找，是一种效率较高的查找方法。

前提条件：查找表中的所有记录是按关键字有序(升序或降序)。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

查找思想

用Low、High和Mid表示待查找区间的下界、上界和中间位置指针，初值为Low=1，High=n。

⑴ 取中间位置Mid：Mid=⌊(Low+High)/2⌋；

⑵ 比较中间位置记录的关键字与给定的K值：

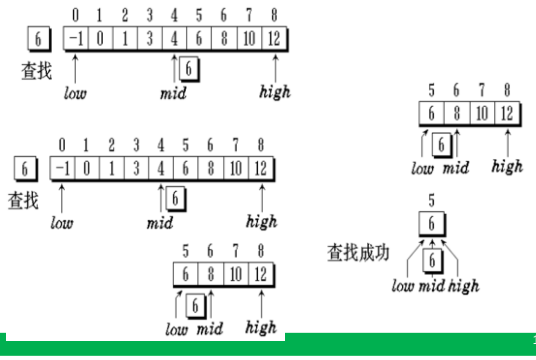
① 相等： 查找成功；

② 大于：待查记录在区间的前半段，修改上界指针： High=Mid-1，转⑴；

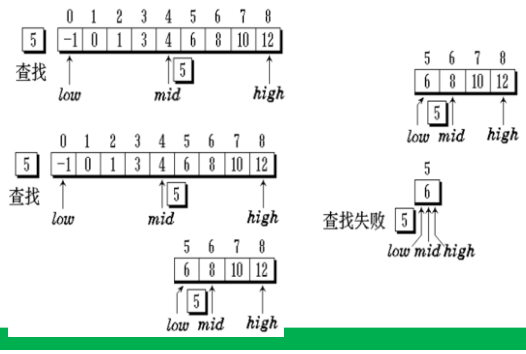
③ 小于：待查记录在区间的后半段，修改下界指针：Low=Mid+1，转⑴；

直到越界(Low>High)，查找失败。

查找成功的例子



查找失败的例子




```
int  Bin_Search(SSTable  ST , KeyType  key)
  {   int  Low=1，High=ST.length, Mid ;
      while (Low<High)
        {    Mid=(Low+High)/2 ;
             if  (EQ(ST. elem[Mid].key, key))
                 return(Mid) ;
             else if (LT(ST. elem[Mid].key, key))
                      Low=Mid+1 ;
                  else   High=Mid-1 ;
        }
      return(0) ;     /*  查找失败  */
  }
```


算法分析

① 查找时每经过一次比较，查找范围就缩小一半，该过程可用一棵二叉树表示：

- 根结点就是第一次进行比较的中间位置的记录；
- 排在中间位置前面的作为左子树的结点；
- 排在中间位置后面的作为右子树的结点；

对各子树来说都是相同的。这样所得到的二叉树称为判定树(Decision Tree)。

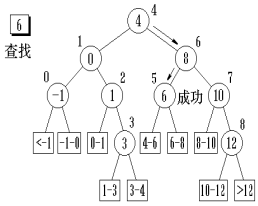
② 将二叉判定树的第$\lfloor \log_2 n \rfloor$+1层上的结点补齐就成为一棵满二叉树，深度不变，$h=\lfloor \log_2(n+1) \rfloor$。

③ 由满二叉树性质知，第i层上的结点数为$2^{i-1}(i\le h)$，设表中每个记录的查找概率相等，即$P_i=1/n$，查找成功时的平均查找长度ASL：

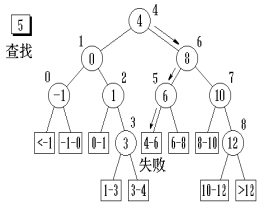
$$ASL=\sum_{i=1}^{n} P_i C_i = \frac{1}{n}\sum_{j=1}^{h} j\times 2^{j-1} = \frac{n+1}{n}\log_2(n+1)-1$$

当n很大 (n>50)时， $ASL\approx \log_2(n+1)-1$。

二叉判定树

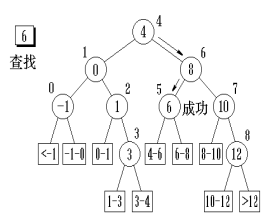


查找成功的情形

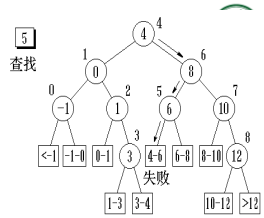


查找不成功的情形

- 若设 $n = 2^h-1$，则描述对分查找的二叉查找树是高度为 h 的满二叉树。$2^h = n+1$, $h = \log_2(n+1)$。



查找成功的情形



查找不成功的情形

$$ASL_{succ} = \sum_{i=1}^{n} p_i c_i$$
$$= \frac{1}{n}\sum_{i=1}^{n} c_i$$
$$= \frac{1}{9}(1+2*2+3*4+4*2)$$

存储方式：顺序存储方式

折半查找的时间复杂性 $O(logn)$

插入、删除的时间复杂性 $O(n)$

5.4 分块查找

分块查找(Blocking Search)又称索引顺序查找，是前面两种查找方法的综合。

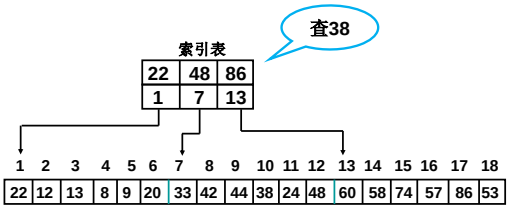
查找表的组织

① 将查找表分成几块。块间有序，即第i+1块的所有记录关键字均大于(或小于)第i块记录关键字；块内无序。

② 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

最大关键字
起始指针

分块查找示例



查找思想

先确定待查记录所在块，再在块内查找(顺序查找)。

算法实现

```
typedef struct IndexType
  { keyType  maxkey ;    /* 块中最大的关键字 */
    int startpos ;    /* 块的起始位置指针 */
  }Index;
```

```
int Block_search(RecType ST[ ] , Index ind[ ] , KeyType key , int n , int b)
/*  在分块索引表中查找关键字为key的记录 */
    /*表长为n ，块数为b */
{   int i=0 , j , k ;
    while ((i<b)&&LT(ind[i].maxkey, key) )  i++  ;
        if (i>b) {  printf("\nNot found");   return(0);  }
    j=ind[i].startpos ;
    while ((j<n)&&LQ(ST[j].key, ind[i].maxkey) )
        {   if ( EQ(ST[j].key, key) )  break ;
            j++ ;
        }    /*  在块内查找  */
    if (j>n||!EQ(ST[j].key, key) )
        {  j=0; printf("\nNot found");  }
    return(j);
}
```


算法分析

设表长为n个记录，均分为b块，每块记录数为s，则$b=\lceil n/s \rceil$。设记录的查找概率相等，每块的查找概率为1/b，块中记录的查找概率为1/s，则平均查找长度ASL：

$$ASL=L_b+L_w=\sum_{j=1}^{b} j+\frac{1}{s}\sum_{i=1}^{s} i=\frac{b+1}{2}+\frac{s+1}{2}$$

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

插入、删除的时间复杂性 ？

O(1)　　*O(n)*

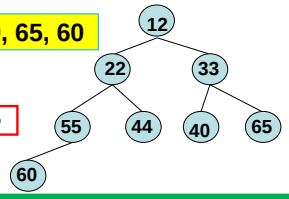
数据不经常变化

5.5 优先队列（priority queue）

- 给定一个具有m个关键字的序列，如果在该序列中，满足：

  对 $i=m/2 \rightarrow 1$的元素，$k_i<=k_{2i}$且$k_i<=k_{2i+1}$

12 , 22, 33, 55, 44, 40, 65, 60



该序列称为优先队列（堆）

5.5.1 priority queue initialization
将初始序列调整成优先队列

- 给定一组关键字， 对 $m/2 \rightarrow 1$的元素依次进行筛选：
  - 若$k_i<=k_{2i}$且$k_i<=k_{2i+1}$，则不换
  - 若$k_i>k_{2i}$(k_{2i+1})且$k_i<=k_{2i+1}$(k_{2i})，则k_i 与k_{2i}（ k_{2i+1} ）交换

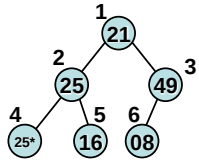
    （大于一个，小于另一个，取其中的一个进行交换
  - 若$k_i>k_{2i}$且$k_i>k_{2i+1}$，则k_i与较小的哪个交换
  - 若k_{2i}($=k_{2i+1}$)< k_i,则k_i与k_{2i}交换（k_{2i}与k_{2i+1}相等）

以上定义和调整算法建立了第一个元素值为最小的优先队列，小顶堆。

建立小顶堆

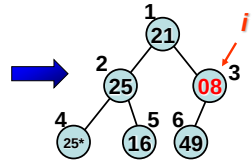
21	25	49	25*	16	08

初始关键字集合



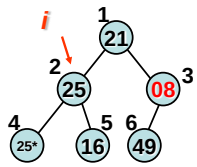
21	25	08	25*	16	49

$i=3$ 时的局部调整



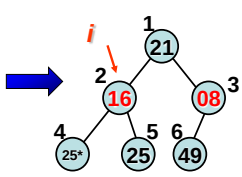
21	25	08	25*	16	49

i = 2 时的局部调整



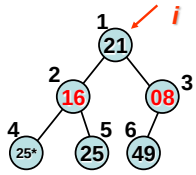
21	16	08	25*	25	49

i = 2 时的局部调整



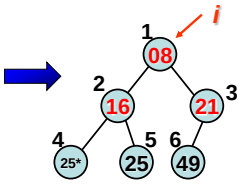
21	16	08	25*	25	49

i = 1 时的局部调整



08	16	21	25*	25	49

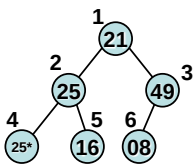
i = 1 时的局部调整



建立大顶堆

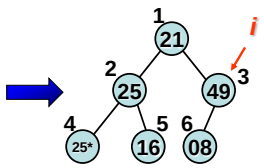
21	25	49	25*	16	08

初始关键字集合



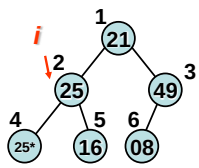
21	25	49	25*	16	08

i = 3 时的局部调整不变



21	25	49	25*	16	08

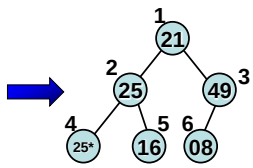
i = 2 时的局部调整



21	25	49	25*	16	08

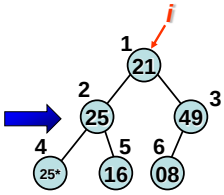
i = 2 时的局部调整

不变

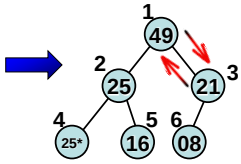


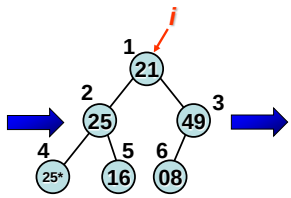
21	25	49	25*	16	08

i = 1 时的局部调整



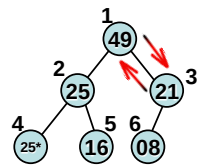
49	25	21	25*	16	08





21	25	49	25*	16	08

i = 1 时的局部调整

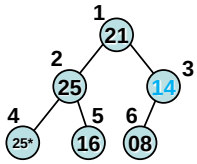


49	25	21	25*	16	08

一个思考问题：建立小顶堆

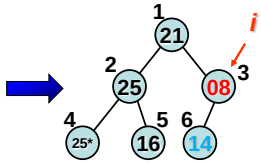
21	25	14	25*	16	08

初始关键字集合



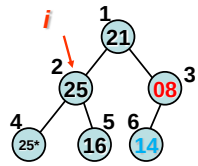
21	25	08	25*	16	14

i = 3 时的局部调整



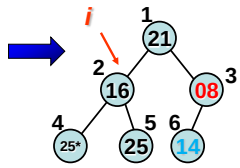
21	25	08	25*	16	14

i = 2 时的局部调整



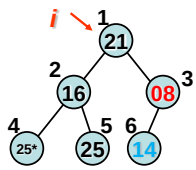
21	16	08	25*	25	14

i = 2 时的局部调整



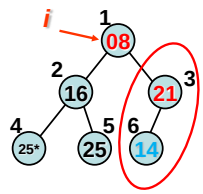
21	16	08	25*	25	14

i = 1 时的局部调整



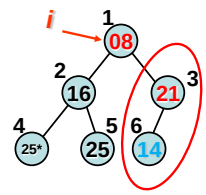
08	16	21	25*	25	14

i = 1 时的局部调整



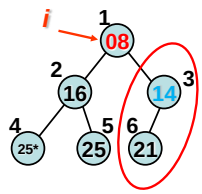
08	16	21	25*	25	14

i = 1 时的局部调整



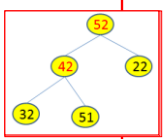
08	16	14	25*	25	21

i = 1 时的局部调整



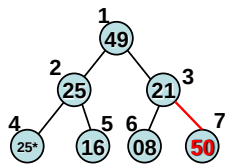
最大堆的向下调整算法

```
void HeapAdjust(HeapType &H,int s,int m)
{ ElemType rc=H.r[s];
  for (int j=2*s;j<=m;j*=2)
    {   if ((j<m) && (H.r[j].key<H.r[j+1].key))
            ++j;
        if (rc.key > H.r[j].key) break;
        H.r[s].key=H.r[j].key;  s=j;
     }
   H.r[s]=rc;
}
```

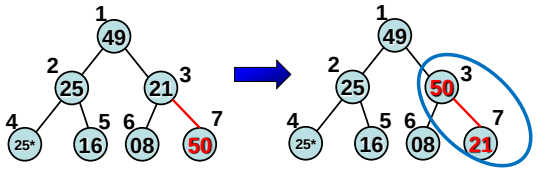



```
for (int i=H.length/2; i>0; --i)
        HeapAdjust(H,i,H.length);
```

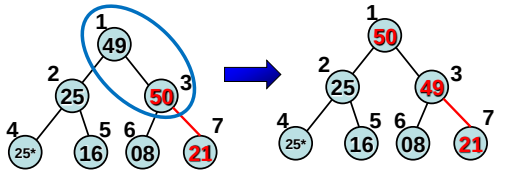

5.5.2 Insert



49	25	21	25*	16	08	50

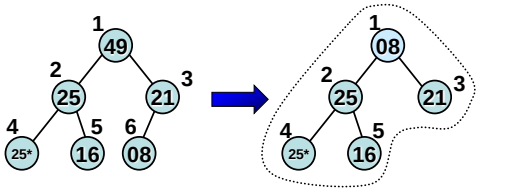


49	25	21	25*	16	08	50



50	25	49	25*	16	08	21

5.5.3 Delete

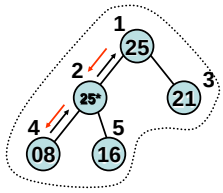


49	25	21	25*	16	08

初始最大堆

08	25	21	25*	16	

将6 号对象调至1号位置



25	25*	21	08	16	

从 1 号到 5号 重新调整为最大堆

5.5.4 性能分析

- ✓ Insert $O(\log n)$
- ✓ Delete $O(1)$
- ✓ Search $O(1)$

存储方式：数组

5.6 树型查找方法

- 二叉查找树
- B树

- ✓ Insert $O(\log n)$
- ✓ Delete $O(\log n)$
- ✓ Search $O(\log n)$

存储方式：链表

5.7 散列（Hashing哈希表）

查找操作要完成什么任务？

- 对于待查找值k，通过比较，确定k在存储结构中的位置

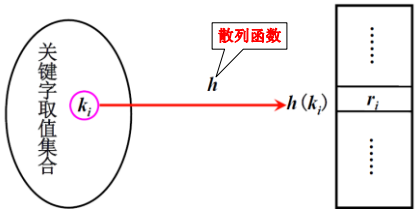
基于关键字比较的查找的时间性能如何？

- 其时间性能为$O(\log n)$~$O(n)$。
- 实际上用判定树可以证明，基于关键字比较的查找的平均和最坏情况下的比较次数的下界是$\log n+O(1)$，即$\Omega(\log n)$
- 要向突破此下界，就不能仅依赖于基于比较来进行查找。

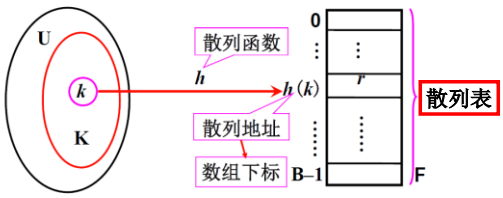
能否不用比较，通过关键字的取值直接确定存储位置？

- 在关键字值和存储位置之间建立一个确定的对应关系

- 散列技术的基本思想
  - 把记录（元素）的存储位置和该记录的关键字的值之间建立一种映射关系。关键字的值在这种映射关系下的像，就是相应记录在表中的存储位置。
  - 散列技术在理想情况下，无需任何比较就可以找到待查的关键字，其查找的期望时间为O(1)。

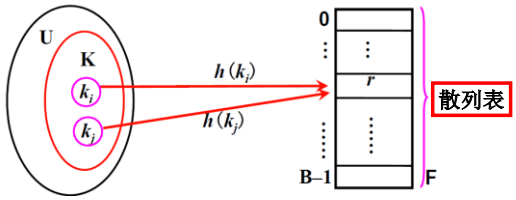


- 散列技术的相关概念
  - 数组 F 称为散列表（Hash表，杂凑表）。数组 F 中的每个单元称为桶(bucket)。
  - 对于任意关键字$k\in U$，函数值$h(k)$称为 k 的散列地址（Hash地址，散列值，存储地址，桶号）



- 散列技术的相关概念
  - 将结点（记录）按其关键字的散列地址存储到散列表中的过程称为散列。
  - 不同的关键字具有相同散列地址的现象称为散列冲突（碰撞）。而发生冲突的两个关键字称为同义词(synonym)。

(collision)



- 散列技术需解决的关键问题：
  - 散列函数的构造。
    - 如何设计一个简单、均匀、存储利用率高的散列函数
  - 冲突的处理
    - 如何采取合适的处理冲突方法来解决冲突。
  - 散列结构上的查找、插入和删除

散列函数的构造

- 散列函数的构造的原则：
  - 计算简单：散列函数不应该有很大的计算量，否则会降低查找效率。
  - 分布均匀：散列函数值即散列地址，要尽量均匀分布在地址空间，这样才能保证存储空间的有效利用并减少冲突。

5.7.1 哈希函数的构造方法

构造散列函数时的几点要求：

- 散列函数的定义域必须包括需要存储的全部关键码，如果散列表允许有m个地址时，其值域必须在 0 到 $m-1$ 之间。
- 散列函数计算出来的地址应能均匀分布在整个地址空间中：若 key是从关键字集合中随机抽取的一个关键字，散列函数应能以同等概率取 0到 $m-1$ 中的每一个值。
- 散列函数应是简单的，能在较短的时间内计算出结果。

1、散列既是查找技术也是存储技术
2、散列只是通过记录关键字的值定位该记录，没有表达记录之间的逻辑关系，所以散列主要是面向查找的存储结构
3、散列适用的场合：通常用于实际出现的关键字数目远小于关键字所有可能的取值的数量
4、散列技术适用的查找类型：不适用于允许多个记录有同样关键字值的情况，也不适用于范围查找，如在散列表中，找最大或最小关键值的记录，也不可能找到在某一范围内的记录

1. 直接定址法

此类函数直接取关键字或关键字的某个**线性函数**值作为散列地址：

Hash（key）＝ $a * \text{key} + b$ ｛a, b为常数｝

- 这类散列函数是一对一的映射，一般不会产生冲突。
  但是，它要求散列地址空间的大小与关键字集合的大小相同。

示例：关键字的取值集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为$h(key)=key/10$，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

适用情况：事先知道关键字的值，关键字取值集合不是很大且连续性 较好。

2. 数字分析法

设有n个d位数，每一位可能有r种不同的符号。这 r 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；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。

对关键字进行分析，取关键字的若干位或组合作为哈希地址。
适用于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

例： 设有80个记录，关键字为8位十进制数，哈希地址为2位十进制数。

①② ③ ④⑤⑥ ⑦⑧

8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

分析： ① 只取8
② 只取1
③ 只取3、4
⑧ 只取2、7、5
④⑤⑥⑦数字分布近乎随机
所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

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，它完全依赖于关键字集合。如果换一个关键字集合，选择哪几位要重新决定。

3. 平方取中法

将关键字平方后取中间几位作为哈希地址。

一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的散列地址也是随机的。

这种方法适于事先不知道关键字的分布情况且关键字的位数不是很大。

记录	key	key^2	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741

4. 折叠法

- 此方法把关键字自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键字的记录的散列地址。
- 有两种叠加方法：
  - 移位法 — 把各部分的最后一位对齐相加；
  - 分界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址。

- 示例：设给定的关键字为 key = 23938587841，若存储空间限定 3 位，则划分结果为每段 3 位. 上述关键字可划分为4段：

239　　385　　878　　41

移位法

```
  2 3 9
  3 8 5
  8 7 8
+)  4 1
-------
  1 5 4 3
```

分界法

```
  2 3 9
  5 8 3
  8 7 8
+)  1 4
-------
  1 7 1 4
```

- 把超出地址位数的最高位删去，仅保留最低的3位，做为可用的散列地址。

5. 除留余数法

- 设散列表中允许的地址数为m,取一个不大于m，但最接近于或等于m的质数p,或选取一个不小于20的质因数的合数作为除数，利用以下公式把关键字转换成散列地址。散列函数为：

  hash (key) = key % p， $p \le m$

- 其中，“%”是整数除法取余的运算，要求这时的质数p不是接近2的幂。
- 实际应用表明，只要选择不能被小于20的质数整除的整数就足够了。

例：有一个关键字 key = 962148，散列表大小 m = 25，即 HT[25]。取质数 p= 23。散列函数 hash (key) = key % p。则散列地址为：

hash (962148) = 962148 % 23 = 12

- 可以按计算出的地址存放记录。需要注意的是，使用上面的散列函数计算出来的地址范围是 0到 22，因此，从23到24这几个散列地址实际上在一开始是不可能用散列函数计算出来的，只可能在处理溢出时达到这些地址。

6. 随机数法

取关键字的随机函数值作哈希地址，即

H(key)=random(key)

当散列表中关键字长度不等时，该方法比较合适。

- 小结：构造Hash函数应注意以下几个问题：
  - 计算Hash函数所需时间 — 计算简单
  - 关键字的长度
  - 散列表的大小
  - 关键字的分布情况
  - 记录的查找频率

  （关键字的长度、散列表的大小、关键字的分布情况、记录的查找频率）— 分布均匀

5.7.2 散列技术

- ✓ 构造散列表
- ✓ 解决散列中的冲突

1、 开放定址法

基本方法：当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。散列地址的计算公式是：

$$H_i(key)=(H(key)+d_i) \ MOD \ m, \ i=1, 2, \ldots, k(k \le m-1)$$

其中：H(key)：哈希函数；m：散列表长度；
d_i： 第i次探测时的增量序列；
H_i(key) ：经第i次探测后得到的散列地址。

(1) 线性探测法

将散列表T[0 …m-1]看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。

增量序列为：d_i=1, 2, 3, …, m-1

设初次发生冲突的地址是h，则依次探测T[h+1]，

T[h+2]…，直到T[m-1]时又循环到表头，再次探测T[0]，

T[1]…，直到T[h-1]。探测过程终止的情况是：

- 探测到的地址为空：表中没有记录。若是查找则失败；若是插入则将记录写入到该地址；
- 探测到的地址有给定的关键字：若是查找则成功；若是插入则失败；
- 直到T[h]：仍未探测到空地址或给定的关键字，散列表满。

例：设散列表长为7，记录关键字组为：15, 14, 28, 26, 56, 23，散列函数：H(key)=key MOD 7，冲突处理采用线性探测法。

H(15)=15 MOD 7=1

H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突 H_1(28)=1 又冲突

H_2(28)=2

H(26)=26 MOD 7=5

H(56)=56 MOD 7=0 冲突 H_1(56)=1 又冲突

H_2(56)=2 又冲突 H_3(56)=3

H(23)=23 MOD 7=2 冲突 H_1(23)=3 又冲突

H_3(23)=4

0	1	2	3	4	5	6
14	15	28	56	23	26	

线性探测法的特点

- 优点：只要散列表未满，总能找到一个不冲突的散列地址；
- 缺点：每个产生冲突的记录被散列到离冲突最近的空地址上，从而又增加了更多的冲突机会(这种现象称为冲突的“聚集”)。

(2) 二次探测法

增量序列为：d_i=1^2,-1^2,2^2,-2^2,3^2,……±k^2 (k≤⌊m/2⌋)

记录关键字组为：15, 14, 28, 26, 56, 23

H(15)=15 MOD 7=1

H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突 H_1(28)=1 又冲突

H_2(28)=4

H(26)=26 MOD 7=5

H(56)=56 MOD 7=0 冲突 H_1(56)=1 又冲突

H_2(56)=0 又冲突 H_3(56)=4 又冲突 H_4(56)=2

H(23)=23 MOD 7=2 冲突 H_1(23)=3

0	1	2	3	4	5	6
14	15	56	23	28	26	

二次探测法的特点

- 优点：探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；
- 缺点：不能保证探测到散列表的所有地址。

(3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间[1，m-1]的随机序列。

例： 表长为11的哈希表中已填有关键字为17，60，29的记录，散列函数为H(key)=key MOD 11 。现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

(1) H(38)=38 MOD 11=5 冲突

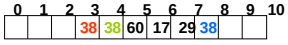
H1=(5+1) MOD 11=6 冲突

H2=(5+2) MOD 11=7 冲突

H3=(5+3) MOD 11=8 不冲突

(2) H(38)=38 MOD 11=5　冲突
H_1=(5+1²) MOD 11=6　冲突
H_2=(5-1²) MOD 11=4　不冲突
(3) H(38)=38 MOD 11=5　冲突

设伪随机数序列为9，则H_1=(5+9) MOD 11=3 不冲突



2 再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止。即：$H_i=RH_i(key)$　i=1, 2, …, k

RH_i：一组不同的哈希函数。第一次发生冲突时，用RH_1计算，第二次发生冲突时，用RH_2计算…依此类推知道得到某个H_i不再冲突为止。

- 优点：不易产生冲突的“聚集”现象；
- 缺点：计算时间增加。

3 链地址法

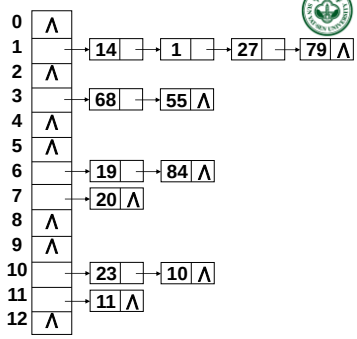
方法：将所有关键字为同义词(散列地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针。

设散列表长为m，定义一个一维指针数组：

RecNode *linkhash[m]，其中RecNode是结点类型，每个分量的初值为空。凡散列地址为k的记录都插入到以linkhash[k]为头指针的链表中，插入位置可以在表头或表尾或按关键字排序插入。

优点：不易产生冲突的“聚集”；删除记录也很简单。

例：已知一组关键字(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)，哈希函数为：H(key)=key MOD 13，用链地址法处理冲突，如右图所示。



用链地址法处理冲突的散列表

4 建立公共溢出区

方法：在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为m，设立基本散列表hashtable[m]，每个分量保存一个记录；溢出表overtable[m]，一旦某个记录的散列地址发生冲突，都填入溢出表中。

例：已知一组关键字(15, 4, 18, 7, 37, 47)，散列表长度为7，哈希函数为：H(key)=key MOD 7，用建立公共溢出区法处理冲突。得到的基本表和溢出表如下：

Hashtable表：

散列地址	0	1	2	3	4	5	6
关键字	7	15	37		4	47	

overtable表：

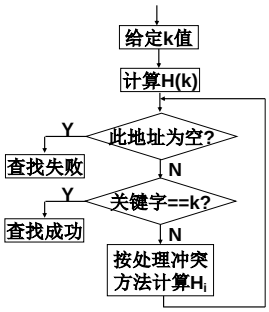
溢出地址	0	1	2	3	4	5	6
关键字	18						

5.7.3 哈希查找过程及分析

1 哈希查找过程

哈希表的主要目的是用于快速查找，且插入和删除操作都要用到查找。由于散列表的特殊组织形式，其查找有特殊的方法。

设散列为HT[0…*m*-1]，散列函数为H(key)，解决冲突的方法为R(*x*, *i*)，则在散列表上查找定值为*K*的记录的过程如图所示。



散列表的查找过程

2 查找算法

```
#define NULLKEY  -1   /*  根据关键字类型定义空标识  */
typedef  struct
   {  KeyType   key ;    /*  关键字域  */
      otherType  otherinfo ;  /*  记录的其它域  */
   }RecType ;
int  Hash_search(RecType HT[], KeyType k, int m)
   /*   查找散列表HT中的关键字K,用开放定址法解决冲突  */
   {  int h, j ;
      h=h(k) ;
      while (j<m && !EQ(HT[h].key, NULLKEY) )
         {  if (EQ(HT[h].key, k) )  return(h) ;
            else h=R(k, ++j) ;
         }
      return(-1) ;
   }
```

```
#define M 15
typedef struct node
   {  KeyType key;
      struct node *link;
   }HNode;
```



```
HNode *hash_search(HNode *t[ ], KeyType k)
  {  HNode *p;   int i;
     i=h(k);
     if (t[i]==NULL)    return(NULL);
     p=t[i];
     while(p!=NULL)
        if (EQ(p->key, k))  return(p);
        else  p=p->link;
     return(NULL);
  }    /*  查找散列表HT中的关键字K,用链地址法解决冲突  */
```


3 哈希查找分析

从哈希查找过程可见：尽管散列表在关键字与记录的存储地址之间建立了直接映象，但由于“冲突”，查找过程仍是一个给定值与关键字进行比较的过程，评价哈希查找效率仍要用ASL。

哈希查找时关键字与给定值比较的次数取决于：

- 哈希函数；
- 处理冲突的方法；
- 哈希表的填满因子α。填满因子α的定义是：

$$\alpha=\frac{\text{表中填入的记录数}}{\text{哈希表长度}}$$

各种散列函数所构造的散列表的ASL如下：

(1) 线性探测法的平均查找长度是：

$$S_{nl成功}\approx\frac{1}{2}\times\left(1+\frac{1}{1-\alpha}\right)$$

$$S_{nl失败}\approx\frac{1}{2}\times\left(1+\frac{1}{(1-\alpha)^2}\right)$$

(2) 二次探测、伪随机探测、再哈希法的平均查找长度是：

$$S_{nl成功}\approx -\frac{1}{\alpha}\times\ln(1-\alpha)$$

$$S_{nl失败}\approx\frac{1}{1-\alpha}$$

(3) 用链地址法解决冲突的平均查找长度是：

$$S_{nl成功}\approx 1+\frac{\alpha}{2}$$

$$S_{nl失败}\approx \alpha+e^{-\alpha}$$

散列表的插入和删除

插入：计算散列地址（解决冲突），插入元素。

删除：查找+？

举 例 一

英文单词系列：
Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec

分别采用线性探测再散列和链地址法解决冲突，构造Hash表，

求成功(失败）的平均查找长度。

H(Key) 为英文单词第一个字母在字母表中的序号/2

表的长度是0..15

线性探测再散列

	Jan	Feb	Mar	Apr	May	June	July	Aug	Sep	Oct	Nov	Dec
序号	10	5	13	1	13	10	10	1	19	15	14	4
地址	5	2	6	0	6	5	5	0	9	7	7	2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Apr	Aug	Feb	Dec		Jan	Mar	May	June	July	Sep	Oct	Nov			
1	2	1	2		1	1	2	4	5	2	5	6			

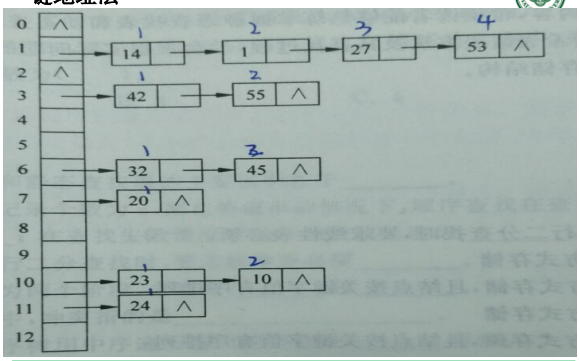
ASL $_{succ}$ =(1+2+1+2+1+1+2+4+5+2+5+6)/12

给定一个单词，可能的位置在0-13，因而检查这些位置的情况

ASL $_{unsucc}$ =(5+4+3+2+1+9+8+7+6+5+4+3+2+1)/14

查找成功时的查找次数等于插入元素时的比较次数（分母为填入表的个数）
第n个位置不成功时的比较次数为第n个位置到第1个没有数据位置的距离（分母为mod的值）

链地址法



举 例 二

19,01,23,14,55,20,84,27,68,11,10,77,24,49

分别采用线性探测再散列和链地址法解决冲突，构造Hash表，

求成功(失败）的平均查找长度。

H(Key)=key % 13

表的长度是0..18

查找成功时的平均查找长度：
ASL = (1*6+2*4+3*1+4*1)/12 = 7/4
查找不成功时的平均查找长度：
ASL = (4+2+2+1+2+1)/13
注意：查找成功时，分母为哈希表元素个数，查找不成功时，分母为mod的值。

5.7.3 散列技术应用实例

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

问题分析：要统计最热门查询，首先就是要统计每个查询串出现的次数，然后根据统计结果，找出前10。

第一步：查询串统计
查询串统计可以选择：

1、直接排序法
对日志里所有查询串都进行排序，然后遍历排好序的查询串，统计每个查询串出现的次数了。
但是题目中有明确要求，那就是内存不能超过1G，一千万条记录，每条记录是255Byte，很显然要占据2.375G内存，这个条件就不满足要求了。可以用外排序中的归并算法，效率为O(nlogn)

排完序之后再对已经有序的查询串文件进行遍历，统计每个查询串出现的次数，再次写入文件中。
综合分析一下，排序的时间复杂度是O(nlogn)，而遍历的时间复杂度是O(n)，因此该算法的总体时间复杂度就是O(n+nlogn)=O（nlogn）。

2、Hash 表

在方法1中，采用了排序的办法来统计每个查询串出现的次数，时间复杂度是nlogn，能否有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，虽然有一千万个查询串，但是由于重复度比较高，因此事实上只有300万的查询串，每个查询串255Byte，因此可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，Hash表绝对是优先的选择，因为Hash表的查询速度非常的快，几乎是O(1)的时间复杂度。

- 算法：维护一个Key为查询串字串，count为该查询串出现次数，每次读取一个查询串，如果该字串不在表中，那么加入该字串，并且将count值设为1；如果该字串在表中，那么将该字串的计数加一即可。最终在O(n)的时间复杂度内完成了对该海量数据的处理。
- 该方法与算法1相比：在时间复杂度上提高了一个数量级，为O(n)，需要读取IO数据文件一次。

91

第二步：找出前10

算法一：普通排序

要注意的是排序算法的时间复杂度是$n\log n$，在本题目中，三百万条记录，用1G内存是可以存下的。

算法二：部分排序

题目要求是求出前10，因此没有必要对所有的查询串都进行排序，只需要维护一个10个大小的数组，初始化放入10个查询串，按照每个查询串的统计次数由大到小排序，然后遍历这300万条记录，每读一条记录就和数组最后一个查询串对比，如果小于这个查询串，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的查询串。最后当所有的数据都遍历完毕之后，那么这个数组中的10个查询串便是我们要找的前10了。

算法的最坏时间复杂度是$n*k$，其中k是指要取得记录多少。

算法三：堆

在算法二中，时间复杂度由$n\log n$优化到$n\log k$。借助堆结构，可以在log量级的时间内查找和调整/移动。算法可以改进为这样，维护一个k(该题目中是10)大小的小根堆，然后遍历300万的查询串，分别和根元素进行对比。采用堆数据结构，算法三，最终的时间复杂度就降到了$n\log k$。

92

先用Hash表统计每个查询串出现的次数，$O(n)$；

然后第二步、采用堆数据结构找出前10，$O(n'\log k)$。

时间复杂度是：$O(n)+O(n'\log k)$。（n为1000万，n'为300万）。

93