数据结构：检索

Data Structure

主讲教师：屈卫兰

Office number: 基地203

tel: 13873195964

查找

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）。

若查找表中存在这样一个记录，则称“查找成功”。查找结果给出整个记录的信息，或指示该记录在查找表中的位置；

否则称“查找不成功”。查找结果给出“空记录”或“空指针”。

顺序检索

- 针对线性表里的所有记录，逐个进行关键码和给定值的比较 。
  - 若某个记录的关键码和给定值比较相等，则检索成功 ；
  - 否则检索失败(找遍了仍找不到)。
- 存储：可以顺序、链接
- 排序要求：无

ST.elem

i i

64	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=64

ST.Length

i i

ST.elem

60	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=60

ST.Length

顺序检索算法

```
template  <class Type>
class Item {
    private:
        Type  key;                          //关键码域
        //…                                 //其它域
    public:
       Item(Type value):key(value) {}
      Type getKey() {return key;}    //取关键码值
      void setKey(Type k){ key=k;} //置关键码
};
vector<Item<Type>*>  dataList;
```

“监视哨”顺序检索算法

- 检索成功返回元素位置，检索失败统一返回0;

```
template  <class Type>  int SeqSearch(vector<Item<Type>*>&
    dataList, int length, Type k) {
      int i=length;
      //将第0个元素设为待检索值
      dataList[0]->setKey (k);     //设监视哨
      while(dataList[i]->getKey()!=k) i--;
      return i;                             //返回元素位置
}
```

顺序查找的时间性能

查找算法的平均查找长度 (Average Search Length)

为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中: n 为表长，P_i 为查找表中第i个记录的概率，

且 $\sum_{i=1}^{n} P_i = 1$,

C_i为找到该记录时，曾和给定值比较过的关键字的个数。

顺序查找的时间性能

对顺序表而言，$C_i = n\text{-}i+1$

$$ASL = nP_1 +(n\text{-}1)P_2 + \quad +2P_{n\text{-}1}+P_n$$

在**等概率**查找的情况下，$P_i = \frac{1}{n}$

顺序表查找的平均查找长度为：

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

顺序查找的时间性能

在不等概率查找的情况下，ASL_{ss} 在

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

时取极小值

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

自组织线性表

根据（估算的）访问频率排列记录.

- 顺序检索

预计的比较时间代价为：

$$\overline{C}_n = 1p_1 + 2p_2 + \ldots + np_n.$$

例(1)

(1) 所有记录的访问频率相同.

$$\overline{C}_n = \sum_{i=1}^{n} i / n = (n+1)/2$$

例(2)

（2）指数频率

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \le i \le n-1 \\ 1/2^{n-1} & \text{if } i = n \end{cases}$$

$$\overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2.$$

Zipf 分布

应用:

- 自然语言的单词使用频率.
- 城市中的人口规模.

$$\overline{C}_n = \sum_{i=1}^{n} i / i \mathrm{H}_n = n / \mathrm{H}_n \approx n / \log_e n.$$

80/20 规则:

- **80% 的访问都是对 20% 的记录进行的.**
- 当频率遵循 80/20 规则, 代价为

$$\overline{C}_n \approx 0.1n.$$

自组织线性表

自组织线性表根据实际的记录访问模式在线性表中修改记录顺序.

自组织线性表使用启发式规则决定如何重新排列线性表.

启发式规则

假设有8条记录，关键码值为A到H，最初以字母顺序排列，按照下面的访问模式： F D F G E G F A D F G E

- 计数统计方法: 保持线性表按照访问频率排序. (类似于缓冲池替代策略中的“最不频繁使用”法.)

  结果：**FGDEABCH**

- 移至前端: 找到一条记录就把它放到线性表的最前面.

  结果：**EGFDABCH**

- 转置:把找到的记录与它在线性表中的前一条记录交换位置.

  结果：**ABFDGECH**

文本压缩示例

发送者和接收者都以同样的方式记录单词在线性表中的位置，线性表根据移至前端规则自组织.

- 如果单词没有出现过，就传送这个单词.
- 否则就传送这个单词在线性表当前的位置.

The car on the left hit the car I left.

The car on 3 left hit 3 5 I 5.

这种压缩方法的思想类似于Ziv-Lempel 编码算法.

二分检索法

- 将任一元素dataList[i] .Key与给定值K比较
  - 三种情况：

（1）Key = K，检索成功，返回dataList[i]

（2）Key > K，若有则一定排在dataList[i]]前

（3）Key < K，若右则一定排在dataList[i]后

- 缩小进一步检索的区间

二分法检索算法

```
template  <class Type> int BinSearch (vector<Item<Type>*>&
    dataList, int length, Type k){
    int low=1, high=length, mid;
    while (low<=high)  {
        mid=(low+high)/2;
        if (k<dataList[mid]->getKey())
                high = mid-1;        //右缩检索区间
        else if (k>dataList[mid]->getKey())
                low = mid+1;         //左缩检索区间
        else return mid;             //成功返回位置
     }
    return 0; //检索失败，返回0
}
```

关键码18 low=1 high=9

1	2	3	4	5	6	7	8	9
15	17	18	22	35	51	60	88	93

low **mid** **high**

第一次：l=1, h=9, mid=5; array[5]=35>18

第二次：l=1, h=4, mid=2; array[2]=17<18

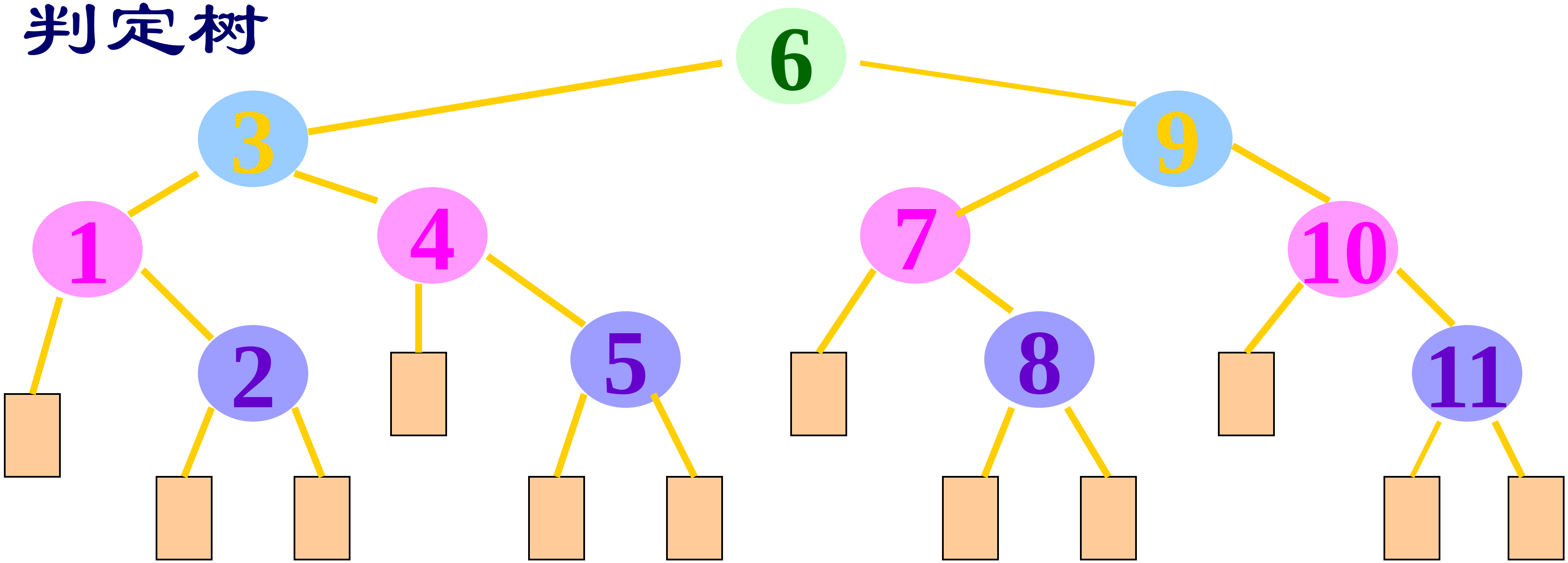
第三次：l=3, h=4, mid=3; array[3]=18＝18

折半查找的平均查找长度

假设：n=11

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4

判定树



折半查找的平均查找长度

一般情况下，表长为n的折半查找的判定树的深度和含有n个结点的完全二叉树的深度相同。

假设 $n=2^h-1$ 并且查找概率相等,则

$$ASL_{bs} = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left[\sum_{j=1}^{h} j \times 2^{j-1}\right] = \frac{n+1}{n}\log_2(n+1)$$

在n>50时，可得近似结果

$$ASL_{bs} \approx \log_2(n+1)-1$$

哈希表

以上讨论的表示查找表的各种结构的共同特点：记录在表中的位置和它的关键字之间不存在一个确定的关系，

查找的过程为给定值依次和关键字集合中各个关键字进行比较，查找的效率取决于和给定值进行比较的关键字个数。用这类方法表示的查找表，其平均查找长度都不为零。

不同的表示方法，其差别仅在于：关键字和给定值进行比较的顺序不同。

哈希表

对于频繁使用的查找表，希望ASL=1。

只有一个办法：预先知道所查关键字在表中的位置，即，要求：记录在表中位置和其关键字之间存在一种确定的关系。

例如：为每年招收的1000名新生建立一张查找表，其关键字为学号，其值的范围为xx000-xx999（前两位为年份）。

若以下标为000 ~ 999 的顺序表表示之。

则查找过程可以简单进行：取给定值（学号）的后三位，不需要经过比较便可直接从顺序表中找到待查关键字。

哈希函数

对于动态查找表，

1）表长不确定；

2）在设计查找表时，只知道关键字所属范围，而不知道确切的关键字。

因此在一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以 f(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 f(key) 为哈希函数。

例如：对于如下9个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dei}

设 哈希函数 f(key) =

$\lfloor$(Ord(第一个字母) -Ord('A')+1)/2$\rfloor$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dei		Han		Li		Qian	Sun		Wu	Ye	Zhao

问题:若添加关键字Zhou，怎么办?

能否找到另一个哈希函数?

哈希函数性质

1) 哈希函数是一个映象，即：将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可；

2) 由于哈希函数是一个压缩映象，因此，在一般情况下，很容易产生“冲突”现象，即： key1≠ key2，而f(key1)=f(key2)。

3) 很难找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

哈希表的定义

根据设定的**哈希函数 H(key)** 和所选中的**处理冲突的方法**，将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的**存储位置**，如此构造所得的查找表称之为“**哈希表**”。

构造哈希函数的方法

对**数字**的关键字可有下列构造方法：

1. 直接定址法

2. 数字分析法

3. 平方取中法

4. 折叠法

5. 除留余数法

6. 随机数法

若是**非数字关键字**，则**需先**对其**进行数字化处理**。

直接定址法

哈希函数为关键字的线性函数

$H(key) = key$ 或者

$H(key) = a \times key + b$

此法仅适合于：

地址集合的大小 = = 关键字集合的大小

数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 $(u_1, u_2, \ldots, u_s)$，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。

平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值” 的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于:

关键字中的每一位都有某些数字重复出现频度很高的现象。

折叠法

将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：移位叠加和间界叠加。

此方法适合于:

关键字的数字位数特别多。

除留余数法

设定哈希函数为:

$$H(key) = key \ MOD \ p$$

其中， $p \le m$（表长） 并且

p 应为不大于 m 的素数

或是

不含 20 以下的质因子

为什么要对 p 加限制

例如：

给定一组关键字为：12, 39, 18, 24, 33, 21,

若取 p=9, 则他们对应的哈希函数值将为：

3, 3, 0, 6, 6, 3

可见，若 p 中含质因子 3， 则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

随机数法

设定哈希函数为:

$$H(key) = Random(key)$$

其中，**Random** 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

示例(1)

```
int h(int x)
{
  return(x % 16);
}
```

散列函数的返回值只依赖于关键码的最低四位.

示例(2)

对于字符串: 将字符串所有字母的 ASCII值累加起来，对M取模.

```
int h(char* x) {
  int i, sum;
  for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
  return(sum % M);
}
```

当累加值比 M 大得多时，散列效果很好.

示例(3)

ELF Hash: 在UNIX系统V Release 4的ELF (Executable and Linking Format)文件格式用到.

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

三、处理冲突的方法

“处理冲突” 的实际含义是：

为产生冲突的地址**寻找下一个**哈希地址。

1. 闭散列法

2. 开散列法

处理冲突的方法

为产生冲突的地址 H(key) 求得一个**地址序列**：

$$H_0, H_1, H_2, \ldots, \quad H_s \qquad 1 \le s \le m-1$$

其中：$H_0 = H(key)$

$$H_i = (H(key) + d_i) \ MOD \ m$$

$$i=1, 2, \ldots, s$$

对增量 d_i 的三种取法

- 线性探测再散列

  $d_i = c \times i$ 最简单的情况 $c=1$

- 平方探测再散列

  $d_i = 1^2, -1^2, 2^2, -2^2, \ldots,$

- 随机探测再散列

  d_i 是一组伪随机数列 或者

  $d_i = i \times H_2(key)$ (又称双散列函数探测)

散列表示例

- {30, 40, 47, 42, 50, 17, 63, 12, 6, 62, 27}
- M = 15, h(key) = key % 15
- 在理想情况下，表中的每个空槽都应该有相同的机会接收下一个要插入的记录。
  - 下一条记录放在第11个槽中的概率是2/15
  - 放到第7个槽中的概率是11/15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30 27		47 17	63		50	6				40		42 27		

改进线性探查

- 每次跳过常数*c*个而不是1个槽
  - 探查序列中的第*i*个槽是(h(*K*) + *ic*) mod *M*
  - 基位置相邻的记录就不会进入同一个探查序列了
- 探查函数是p(*K*，*i*) = *i***c*
  - *必须使常数c与M互素*

例：改进线性探查

- 例如，$c = 2$，要插入关键码k_1和k_2，$\mathrm{h}(k_1) = 3$，$\mathrm{h}(k_2) = 5$
- 探查序列
  - k_1的探查序列是3、5、7、9、...
  - k_2的探查序列就是5、7、9、...
- k_1和k_2的探查序列还是纠缠在一起，从而导致了聚集

二次探查

- 探查增量序列依次为：1^2，-1^2，2^2，-2^2，...，即地址公式是

  $d_{2i-1} = (d + i^2) \% M$

  $d_{2i} = (d - i^2) \% M$

- 用于简单线性探查的探查函数是

  $p(K, 2i-1) = i*i$

  $p(K, 2i) = - i*i$

例：二次探查

- 使用一个大小M = 13的表

假定对于关键码k_1和k_2，$h(k_1)=3$，$h(k_2)=2$

- 探查序列
  - k_1的探查序列是3、4、2、7 、...
  - k_2的探查序列是2、3、1、6 、...
- 尽管k_2会把k_1的基位置作为第2个选择来探查，但是这两个关键码的探查序列此后就立即分开了

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 (表长=11)

若采用线性探测再散列处理冲突

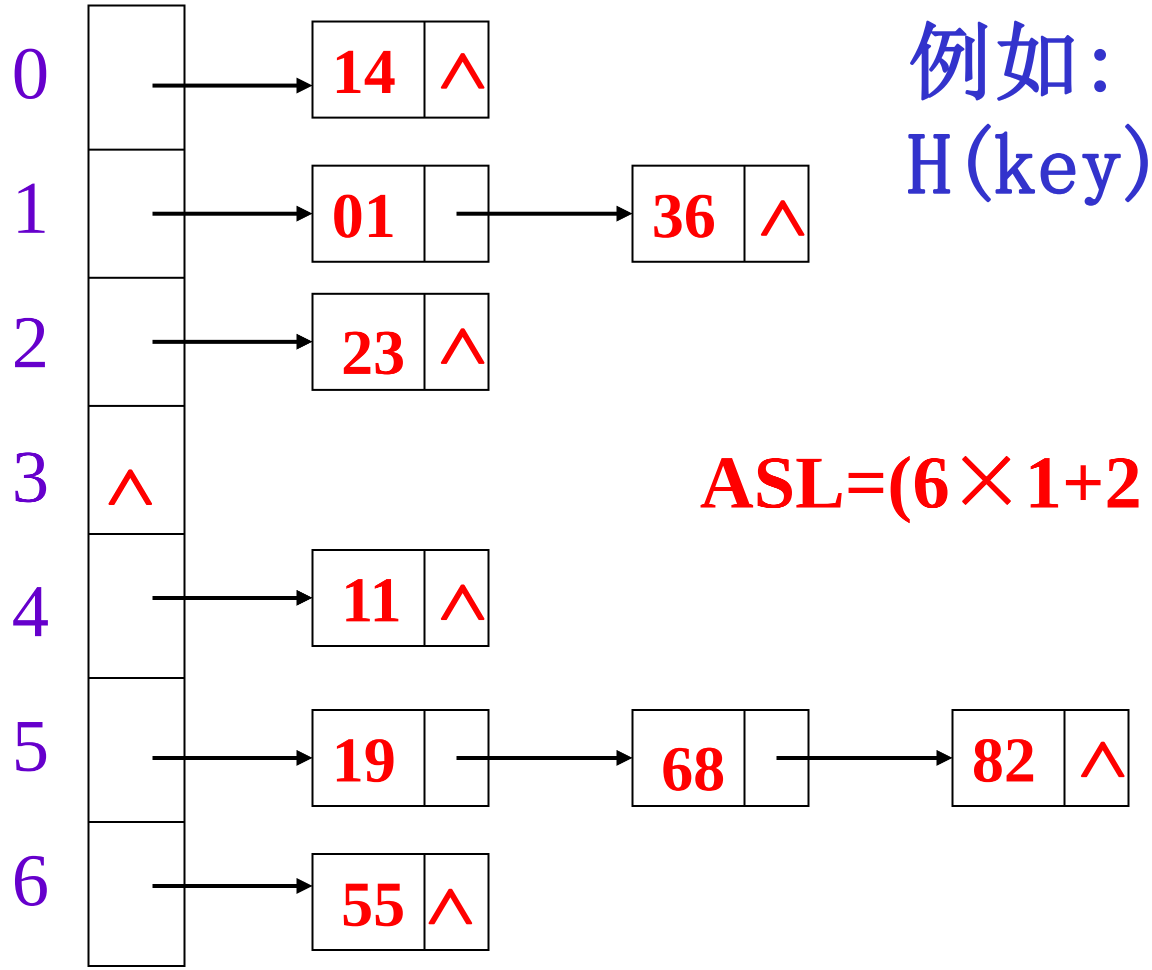
0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

若采用二次探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

2. 开散列法

将所有哈希地址相同的记录都链接在同一链表中。



例如：哈希函数为

H(key)=key MOD 7

ASL=(6×1+2×2+3)/9=13/9

哈希表的查找

查找过程和造表过程一致。假设采用开散列处理冲突，则查找过程为：

对于给定值 K， 计算哈希地址 i = H(K)

若 r[i] = NULL 则查找不成功

若 r[i].key = K 则查找成功

否则 “求下一地址 Hi”， 直至

r[Hi] = NULL (查找不成功)

或 r[Hi].key = K (查找成功) 为止。

Insertion

```
// Insert e into hash table HT
template <typename Key, typename E>
bool hashdict<Key, E>::
hashInsert(const key& e, const E& e) {
  int home;        // Home position for e
  int pos = home = h(k); // Init
  for (int i=1; EMPTYKEY!=( HT[pos].key(); i++) {
    pos = (home + p(k,i)) % M;
    Assert(k!= HT[pos].key(),"Duplicates not allowed");
   }
  Kvpair<Key,E> temp(k,e);
  HT[pos] = temp;      // Insert e
}
```

Search

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
  int home;           // Home position for K
  int pos = home = h(k); // Initial posit
  for (int i = 1; (k!=( HT[pos].key()) &&
          (EMPTYKEY!=( HT[pos]).key()); i++)
    pos = (home + p(k, i)) % M; // Next
  if (k==(HT[pos]).key()) { // Found it
    return( HT[pos]).value();
   else return NULL; // K not in hash table
}
```

散列方法的效率分析

- 衡量标准：插入、删除和检索操作所需要的记录访问次数
- 散列表的插入和删除操作都是基于检索进行的
  - 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检索
    - 对于不考虑删除的情况，是尾部的空槽
    - 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

哈希表查找的分析

决定哈希表查找的ASL的因素：

- 选用的哈希函数；
- 选用的处理冲突的方法；
- 哈希表饱和的程度，装载因子 α =n/m 值的大小（n—记录数，m—表的长度）

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。

影响检索的效率的重要因素

- 散列方法预期的代价与负载因子
  $\alpha = N/M$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槽
- 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

查找成功时有下列结果

线性探测再散列

$$S_{nl} \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$$

随机探测再散列

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

链地址法

$$S_{nc} \approx 1+\frac{\alpha}{2}$$