数据结构与算法

第10章　检索

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

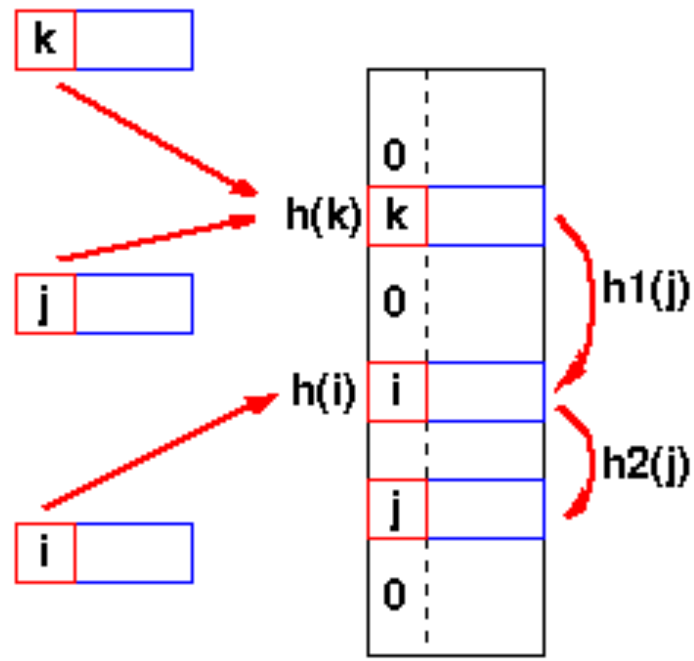
张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

散列技术的两个要素

- 涉及散列的基本问题/首要问题可分成两类
  1. 如何构造（选择）使结点“**分布均匀**”的散列函数?
     - 散列函数需具备怎样的特性（properties）？怎样才能获得或设计一个具有这些特性的散列函数?
  2. 一旦发生**冲突**，用什么方法来**解决**?

碰撞的处理

- **开散列方法**(open hashing，也称为**拉链法**： separate chaining)
  - 把发生冲突的关键码存储在散列表主表之外
- **闭散列方法**(closed hashing，也称为**开地址方法**，open addressing)
  - 把发生冲突的关键码存储在表中另一个槽内



开散列方法

- 当碰撞发生时就拉出一条链，建立一个链式的同义词子表
  - 动态申请同义词的空间，适合于内存操作

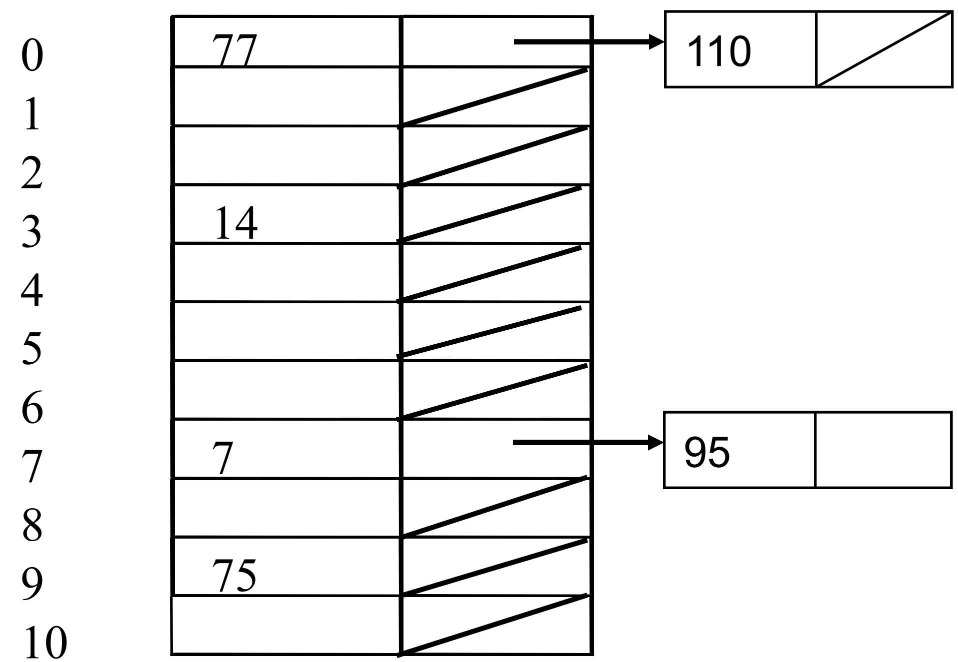
- 拉链法
- 桶式散列

拉链法

- 表中空单元由特殊值标记，例如，-1
  - 或使散列表的内容为指针，空单元则内容为空指针
- 插入同义词时，可以对同义词链排序插入

例： {77 , 7, 110 , 95 , 14 , 75 , 62 }

h(key) = key % 11



拉链法

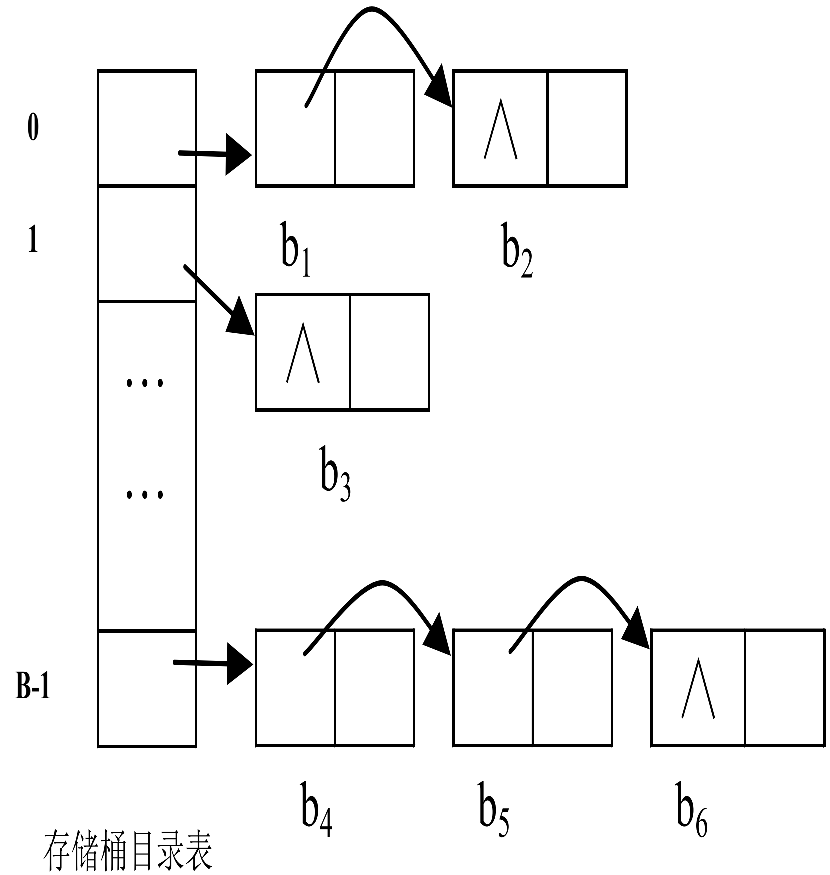
- 给定一个大小为M存储n个记录的表
  - 散列函数（在理想情况下）将把记录在表中M个位置均匀放置，使得每一个链表中平均有n/M个记录
  - M>n 时，散列方法的平均代价为 Θ(1)
- 适用情况
  - 若整个散列表存储在内存中，开散列方法较易实现
  - 开散列方法不太合适于散列表存储在磁盘中的情况
    - 一个同义词表的元素可能存储在不同的磁盘页中，导致检索一个特定关键码时引起多次磁盘访问，从而增加了检索时间
    - 桶式散列

桶式散列

- 适合存储于磁盘的散列表
- 基本思想
  - 把一个文件的记录分为若干存储桶，每个存储桶包含一个或多个页块
  - 一个存储桶内的各页块用指针连接起来，每个页块包含若干记录
  - 散列函数h(K)表示具有关键码值K的记录所在的存储桶号

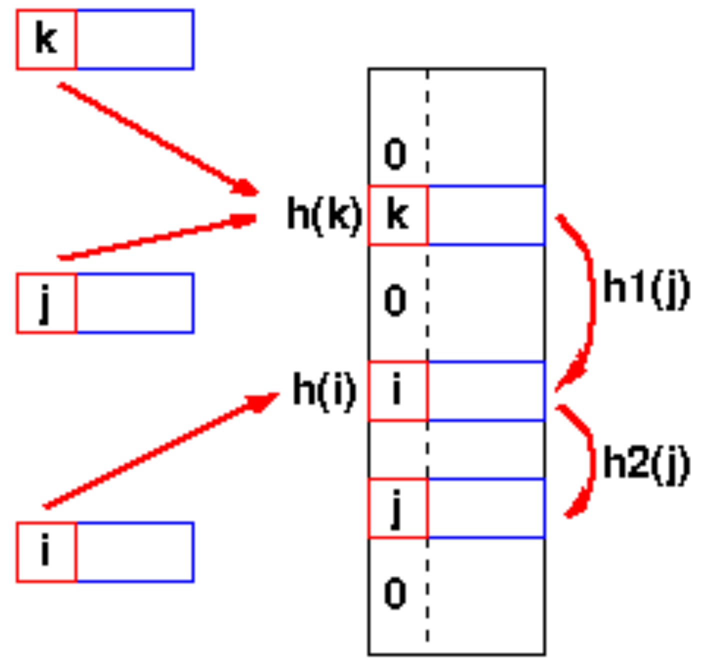
桶式散列文件组织

- 一个具有B个存储桶的散列文件组织
  - 若B很小，存储桶目录表可放在内存
  - 若B较大，要存放好多页块，则存储桶目录表就放到外存上



碰撞的处理

- 开散列方法(open hashing/拉链法separate chaining)
  - 把发生冲突的关键码存储在散列表主表之外
- 闭散列方法(closed hashing/开地址方法，open addressing)
  - 把发生冲突的关键码存储在表中另一个槽内



闭散列方法

- 所有记录均存储在散列表中
  - 每个记录有一个**基位置**，即由散列函数计算出来的地址 h(key)
- 插入一个记录 R 时，若其基位置已被另一记录占据，则发生**碰撞**
  - 需把 R 存储在表中的**其它位置**，由**冲突解决策略**确定此位置

闭散列表冲突解决基本思想

- 当冲突发生时，使用某种方法为关键码 K 生成一个散列地址序列

  d_0， d_1， d_2， … d_i ， …d_{m-1}

  - 其中d_0 = d = h(K) 称为 K 的 基地址
  - 所有d_i(0<i<m)是 后继散列地址

- 探查方法不同，所得到的冲突解决策略也不同

闭散列表冲突解决基本思想

- 当插入记录 *K* 时，若基地址单元已被别的数据元素占用
  - 则按上述地址序列依次探查，将找到的第1个开放的空闲位置d_i作为 K 的存储位置
  - 若所有后继散列地址都非空，说明该闭散列表已满，报告溢出

探查序列

- 基础假设
  - 插入和检索的前提：假定每个关键码的探查序列中至少有一个存储位置是空的，否则会无限循环
- 也可 限制 探查序列的长度

常见的探查方法

- 线性探查
- 二次探查
- 伪随机数序列探查
- 双散列探查法

线性探查

- 基本思想
  - 若记录的基位置存储位置被占用，就在表中下移，直到找到一个空的存储位置
    - 依次探查下述地址单元：d+1，d+2，……，M-1，0，1，……，d-1
  - 用于简单线性探查的**探查函数**

    $$p(K，i) = i$$

- 优点
  - 表中所有存储位置都可以作为插入新记录的候选位置

线性探查示例

- 一组关键码（26，36，41，38，44，15，68，12，06，51，25），散列表长度M = 15，用**线性探查法**解决冲突，构造这组关键码的散列表 （n =11， M = 15）
  - 散列函数采用**除余法**，选 P =13，

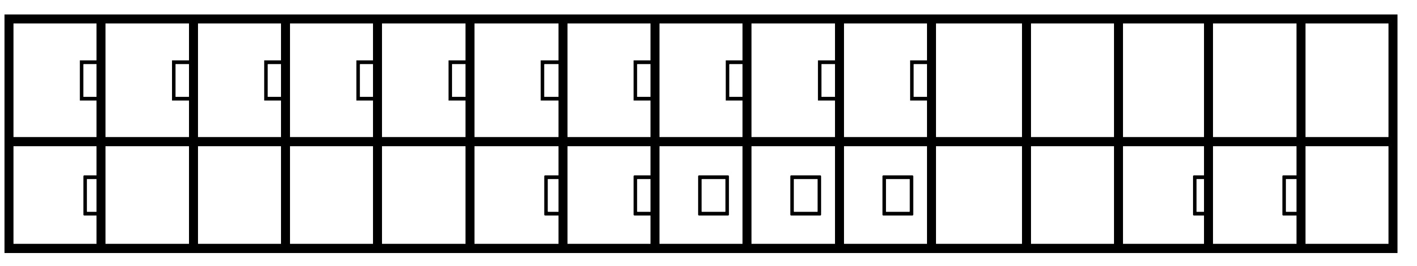
h(key) = key%13

  - 按顺序插入各个结点：

26: h(26) = 0，36: h(36) = 10， 41: h(41) = 2，38: h(38) = 12，44: h(44) = 5

线性探查示例

- 在理想情况下，表中每个空槽都**应有相同机会**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



线性探查的潜在问题

- "聚集"（clustering，也称"堆积"）
  - 散列地址不同的结点，争夺同一后继散列地址
  - 小的聚集可能汇合成大的聚集
  - 导致很长的探查序列

线性探查的改进

- 每次跳过 常数c 个 而非 1 个槽
  - 探查序列中的第 i 个槽是 $(h(k) + i*c) \bmod M$
  - 避免基位置相邻的记录进入同一个探查序列
- 探查函数 $p(k,i) = i*c$
  - 必须使常数 c 与 M 互素

线性探查的改进示例

- 例如，$c = 2$，要插入关键码 k_1 和 k_2，

  $h(k_1) = 3$，$h(k_2) = 5$，

  - k_1的探查序列：3, 5, 7, 9, ...
  - k_2的探查序列：5, 7, 9, ...

- k_1 和 k_2的探查序列还纠缠在一起，引发聚集

二次探查

- 探查序列列依次为：1^2，-1^2，2^2，-2^2，…，即，探查函数

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

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

- 用于二次探查的探查函数

  $p(k, 2i-1) = i^2$

  $p(k, 2i) = -\ i^2$

同义词来回散列在基地址的两侧

二次探查示例

- 例：一个大小M = 13的散列表，对于关键码k_1和k_2有： $h(k_1) = 3$，$h(k_2) = 2$
  - 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，但两个探查序列此后就分开了
  - 偶尔的交错
  - 缺点：不易探查到整个闭散列表的所有位置

伪随机数序列探查

- 探查函数

  $p(k, i) = \text{perm}[i - 1]$

  - perm是一个长度为M-1的数组，包含值从1到M-1的随机序列

避免产生*0* 或*M*，以减少对基地址的无谓探查

```
// 产生n个数的伪随机排列
void permute(int *array, int n) {
    for (int i = 1;  i <= n;  i ++)
        swap(array[i-1], array[Random(i)]);
}
```

伪随机数序列探查示例

- 考虑一个大小为M = 13的表，其中
  perm[0] = 2，perm[1] = 3，perm[2] = 7
- 若两个关键码k_1和k_2，h(k_1)=4，h(k_2)=2
  - k_1的探查序列是4、6、7、11 、...
  - k_2的探查序列是2、4、5、9 、...
- 尽管k_2会把k_1的基位置作为第2个选择来探查，但它们的探查序列就此分开

二级聚集

- 基本聚集
  - 基地址不同的关键码，其探查序列的某些段重叠而形成
  - 伪随机探查和二次探查可消除基本聚集
- 二级聚集 （secondary clustering）
  - 若两个关键码散列到同一个基地址（亦即碰撞），则得到同样的探查序列，由此所产生的聚集
  - 究其原因，探查序列只是基地址的函数，而非关键码值的函数

解决方案：双散列探查法

双散列探查法

- 避免二级聚集
  - 探查序列为关键码值的函数，而非仅为基位置的函数
- 双散列探查法
  - 使用两个散列函数，并将第二个散列函数作为线性探查时探查序列的步长，亦即
    - 双散列函数探查法序列公式:

      $d_i=(d + i * h_2 (key)) \% M$
    - 双散列函数(探查函数）：

      $p(key, i) = i * h_2 (key)$

双散列探查法的基本思想

- 双散列探查法使用两个散列函数 h_1 和 h_2，若在基地址 h_1 (key) = d 发生冲突，再计算 h_2(key)，得到的探查序列为：

  (d+h_2(key)) % M，

  (d+2h_2 (key)) % M，

  (d+3h_2 (key)) % M，

  ……
- h_2(key) 尽量与 M 互素
  - 使发生冲突的同义词地址均匀地分布在整个表中
  - 否则可能造成同义词地址的循环计算

双散列探查法函数的选择

- 方法1：选择M 为一个素数，h_2的值在区间$[1, M-1]$
- 方法2：设置$M = 2^m$，让h_2返回一个 1到2^m 之间的奇数值
- 方法3：若M是素数，$h_1(k) = k \bmod M$
  - $h_2(k) = k \bmod (M-2) + 1$，或
  - $h_2(k) = [k / M] \bmod (M-2) + 1$
- 方法4：若M是任意数，$h_1(k) = k \bmod p$ (p 为小于M的最大素数)
  - $h_2 (k) = k \bmod q + 1$ (q为小于p的最大素数)

双散列探查法的优劣

- 优点： 不易产生“聚集”
  - 探查序列跳跃式散列， 而非顺序式散列
- 缺点：计算量增大
  - 增加一个函数计算时间

思考

- 插入同义词时，如何对同义词链进行组织?
- 双散列函数 h_2 (key) 与 h_1 (key) 有什么关系?

闭散列表的算法实现

闭散列可形成称为字典(dictionary)的数据结构

- 一种特殊的集合，其元素是(关键码，属性值)二元组
  - （同一个字典内）关键码必须互不相同
- 主要操作是依据关键码来插入（存储）和查找（析取）

```
bool hashInsert(const Elem&);
   // insert(key, value)
bool hashSearch(const Key&， Elem&) const;
   // lookup(key)
```

字典的实现方式

- 有序线性表
- 字符树
- 散列方法
  - 散列字典

散列字典ADT

```
// 散列字典的属性
template <class Key， class Elem， class KEComp， class
            EEComp> class hashdict {
private:
   Elem* HT;                      // 散列表
   int M;                         // 散列表大小
   int currcnt;                   // 现有元素数目
   Elem EMPTY;                    // 空槽
   int h(int x) const ;           // 散列函数
   int h(char* x) const ;         // 字符串散列函数
   int p(Key k， int i)           // 探查函数
```

散列字典ADT

```
// 散列字典的方法
public:
    hashdict(int sz， Elem e) {                    // 构造函数
        M=sz;  EMPTY=e;
        currcnt=0;  HT=new Elem[sz];
        for (int i=0; i<M; i++) HT[i]=EMPTY;
    }
    ~hashdict() { delete [] HT; }
    bool hashSearch(const Key&， Elem&) const;
    bool hashInsert(const Elem&);
    Elem hashDelete(const Key& K);
    int size() { return currcnt; }                 // 元素数目
};
```

散列表的插入算法

散列函数h，假设给定的关键码值为*k*

- 若表中基地址对应的空间未被占用，则直接在该地址插入相应记录
- 若基地址中的值与*k* 相等，则报告“散列表中已有此记录”
- 否则，按选定的**冲突处理策略**查找探查序列的下一个地址，如此反复下去
  - 直到某个地址空间未被占用（可以插入）
  - 或者关键码比较相等（不需要插入）为止

插入算法代码

```
// 将数据元素e插入到散列表 HT
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem& e) {
   int home= h(getkey(e));                          // home 存储基位置
   int i=0;
   int pos = home;                                  // 探查序列的初始位置
   while (!EEComp::eq(EMPTY, HT[pos])) {
      if (EEComp::eq(e, HT[pos])) return false;
      i++;
      pos = (home+p(getkey(e), i)) % M;             // 探查
   }
   HT[pos] = e;                                     // 插入元素e
   return true;
}
```

散列表的检索

- 与插入遵循同样的策略
  - 重复插入时的冲突解决过程
    - 须采用与插入时相同的探查序列
  - 找出在基位置没有找到的记录

散列表的检索

- 假设散列函数h，给定的值为k
  - 若表中该地址对应的空间未被占用，则检索失败
  - 否则将该地址中的值与k 比较，若相等则检索成功
  - 否则，按建表时采用的**冲突解决策略**查找探查序列的下一个地址，如此反复下去
    - 关键码比较相等，检索成功
    - 地址空间未被占用，检索失败

散列表检索算法代码

```
template <class Key, class Elem, class KEComp, class EEComp> bool
hashdict<Key, Elem, KEComp, EEComp>::
hashSearch(const Key& K, Elem& e) const {
   int i=0, pos= home= h(K);                               // 初始位置
   while (!EEComp::eq(EMPTY, HT[pos])) {
      if (KEComp::eq(K, HT[pos])) {                         // 找到
         e = HT[pos];
         return true;
      }
      i++;
      pos = (home + p(K, i)) % M;
   }   // while
   return false;
}
```

散列表的删除

- 删除记录的时候，有两点需重点考虑
  (1) 删除一个记录一定不能影响后续的检索
  (2) 释放的存储位置应能为将来的插入所使用

- 由此：
  - 只有开散列方法（分离的同义词子表）可以真正删除
  - 闭散列方法都只能作标记（墓碑），不能真正删除
    - 若真正删除了将使探查序列断裂
      - 检索算法 “直到某个地址空间未被占用（检索失败）”
    - 墓碑标记增加了平均检索长度

删除的潜在问题

0	1	2	3	4	5	6	7	8	9	10	11	12
	K_1	K_2	K_1		K_2	K_2	K_2			K_2		

- 例如，一个长度$M = 13$的散列表，假定关键码k_1和k_2所对应的散列地址分别为：$h(k_1) = 2$，$h(k_2) = 6$；
  - k_1的二次探查序列是2, 3, 1, 6, 11, 11, 6, 5, 12 , ...
  - k_2的二次探查序列是6, 7, 5, 10, 2, 2, 10, 9, 3, ...
- 删除位置 6，用该序列的最后位置 2 的元素替换之，位置2设为空
- 检索k_1的同义词，查不到，而事实上还在位置 3 和 1 上!

墓碑

- 设置一个特殊的标记位，用于记录散列表中的单元状态
  - 单元被占用
  - 空单元
  - 已删除
- 是否可以同等对待空单元、已删除两种状态，用特殊的值标记，以区别于“单元被占用”状态?
  - **不可以！** 须严格区分空单元与已删除单元
- 被删除标记值称为墓碑 (tombstone)
  - 标志一个记录曾经占用这个槽，但现已不再占用了

带墓碑的删除算法

```
template <class Key, class Elem, class KEComp, class EEComp>Elem
hashdict<Key,Elem,KEComp,EEComp>::hashDelete(const Key& K)
{   int i=0,  pos = home= h(K);           // 初始位置
    while (!EEComp::eq(EMPTY, HT[pos]))  {
       if (KEComp::eq(K, HT[pos])){
          temp = HT[pos];
          HT[pos] = TOMB;                 // 设置墓碑
           return temp;                   // 返回目标
       }
       i++;
       pos = (home + p(K, i)) % M;
    }
    return EMPTY;
}
```

带墓碑的插入操作

- 在插入时，如果遇到标志为墓碑的槽，可以把新记录存储在该槽中吗?
  - 避免插入两个相同的关键码
  - 检索过程仍然需要沿着探查序列下去，直到找到一个真正的空位置

带墓碑的插入操作改进

```
template <class Key, class Elem, class KEComp, class EEComp> bool
hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem &e)  {
   int insplace, i = 0, pos = home = h(getkey(e));
   bool tomb_pos = false;
   while (!EEComp::eq(EMPTY, HT[pos])) {
      if (EEComp::eq(e, HT[pos])) return false;
      if (EEComp::eq(TOMB, HT[pos]) &&  !tomb_pos)
      {insplace = pos;  tomb_pos = true;}                 // 第一
      pos = (home + p(getkey(e), ++ i)) % M;
   }
   if (!tomb_pos) insplace=pos;                            // 没有墓碑
     HT[insplace] = e;    return true;
}
```

散列方法的效率分析

- 衡量标准
  - 插入、删除和检索操作所需的记录访问次数
- 散列表的插入和删除操作均 基于检索
  - 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，也需找到尾部才能确定是否有重复记录

影响检索效率的重要因素

- 散列方法预期的代价与负载因子α （= 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的增加，越来越多的记录有可能放到离其基地址更远的位置

散列表算法分析

- 基地址被占用的可能性为 α
- 基地址和探查序列中下一个槽都被占用的可能性

$$\frac{N(N-1)}{M(M-1)}$$

- 发生第 *i* 次冲突的可能性

$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$$

- 若 N 和 M 都很大，则可近似表达为 $(N/M)^i$

散列表算法分析

- 探查次数的期望值 为 1 加上 每个第 i 次($i \ge 1$) 冲突的概率之和，即:

$$1+\sum_{i=1}^{\infty}(N/M)^{i}=1/(1-a)$$

散列表算法分析

- 一次成功检索(或者一次删除)的代价与当时插入的代价相同
- 随着散列表中记录的不断增加，α值也不断增大
  - 根据从0 到α 的当前值的积分可推导出插入操作的平均代价（实质上是所有插入代价的一个平均值）

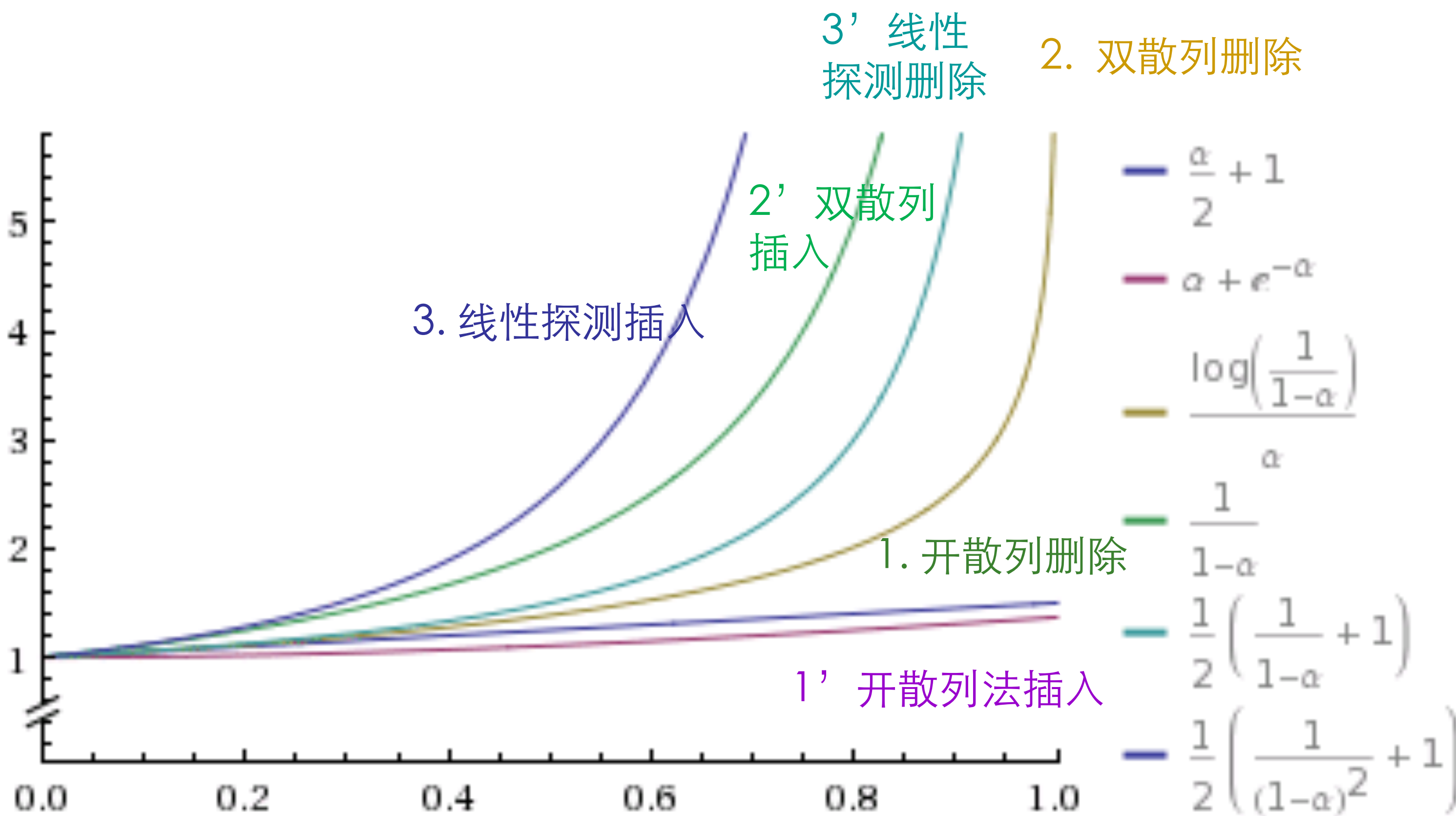
$$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$$

散列表算法分析（表）

编号	冲突解决策略	成功检索（删除）	不成功检索（插入）
1	开散列法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	双散列探查法	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

散列表算法分析（图）

- 几种不同方法解决碰撞时散列表的平均检索长度



散列表算法分析结论

- 散列方法的代价一般接近于访问一个记录的时间，效率非常高，比需要log n次记录访问的二分检索好得多
  - 不依赖于n，只依赖于负载因子α=n/M
  - 随着α 增加，预期的代价会增加
  - α≤0.5时，大部分操作的分析预期代价都小于 2 （也有1.5之说）
- 实际经验表明散列表负载因子的临界值是0.5
  - 大于这个临界值，性能就会急剧下降

散列表算法分析结论

- 散列表的插入和删除操作若很频繁，将降低散列表的检索效率
  - 大量的插入操作，将使负载因子增加
    - 增加同义词子表长度，也即，增加了平均检索长度
  - 大量的删除操作，增加墓碑的数量
    - 导致记录本身到其基地址的平均长度的增加
- 实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重散列
  - 把所有记录重新散列到一个新表中
    - 清除墓碑；最频繁访问的记录放到其基地址

散列的应用

- 检索效率与数据规模无关，平均检索长度1.5，应用广泛
  - 搜索引擎关键词字典
  - 域名服务器域名与IP解析
  - C Shell下可执行程序表
  - 账户/口令
  - 文件压缩
  - 信息加密
  - 字符串模式匹配RB算法

toDo

- 调研除散列以外字典的其他实现方法