数据结构与算法（十）

张铭 主讲

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

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

第十章　检索

- 10.1 线性表的检索
- 10.2 集合的检索
- 10.3 散列表的检索
- 总结

散列检索

- 10.3.0 散列中的基本问题
- 10.3.1 散列函数碰撞的处理
- 10.3.2 开散列方法
- 10.3.3 闭散列方法
- 10.3.4 闭散列表的算法实现
- 10.3.5 散列方法的效率分析

闭散列表的算法实现

字典 (dictionary)

- 一种特殊的集合，其元素是(关键码，属性值)二元组。
  - 关键码必须是互不相同的(在同一个字典之内)
- 主要操作是依据关键码来插入和查找
  - **bool hashInsert(const Elem&);**
    // insert(key, value)
  - **bool hashSearch(const Key& , Elem&) const;**
    // lookup(key)

散列字典ADT（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（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
};
```

插入算法

散列函数 h，假设给定的值为 K

- 若表中该地址对应的空间未被占用，则把待插入记录填入该地址
- 如果该地址中的值与 K 相等，则报告“散列表中已有此记录”
- 否则，按设定的处理冲突方法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

散列表插入算法代码

```
// 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;
}
```

检索算法

- 与插入过程类似
  - 采用的探查序列也相同
- 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，按建表时设定的处理冲突方法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
    - 走到探测序列尾部还没找到，检索失败

10.3 散列检索

```
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;
}
```

删除

- 删除记录的时候，有两点需要重点考虑：
  - （1）删除一个记录一定不能影响后面的检索
  - （2）释放的存储位置应该能够为将来插入使用
- 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
  - 若真正删除了探查序列将断掉
    - 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

删除带来的问题

0	1	2	3	4	5	6	7	8	9	10	11	12
	K1	K2	K1		K2	K2	K2			K2		

- 例，长度 $M = 13$ 的散列表，假定关键码 $k1$ 和 $k2$，$h(k1) = 2$，$h(k2) = 6$。
- 二次探查序列
  - $k1$ 的二次探查序列是 2、3、1、6、11、11、6、5、12、...
  - $k2$ 的二次探查序列是 6、7、5、10、2、2、10、9、3、...
- 删除位置 6，用$k2$ 序列的最后位置 2 的元素替换之，位置 2 设为空
- 检索 $k1$ ，查不到（事实上还可能存放在位置 3 和 1 上! ）

墓碑

- 设置一个特殊的标记位，来记录散列表中的单元状态
  - 单元被占用
  - 空单元
  - 已删除
- 被删除标记值称为 墓碑（tombstone）
  - 标志一个记录曾经占用这个槽
  - 但是现在已经不再占用了

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版

```
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 所需要的记录访问次数
- 散列表的插入和删除操作 都基于检索进行
  - 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

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

散列表算法分析（1）

- 基地址被占用的可能性是 α
- 发生第 i 次冲突的可能性是

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

- 如果 N 和 M 都很大，那么可以近似地表达为

$$(N/M)^i$$

- 探查次数的期望值是 1 加上每个第 i 次（$i \geq 1$）冲突的概率之和，即插入代价：

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

散列表算法分析（2）

- 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
- 由于随着散列表中记录的不断增加，α 值也不断增大
- 我们可以根据从 0 到 α 的当前值的积分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)：

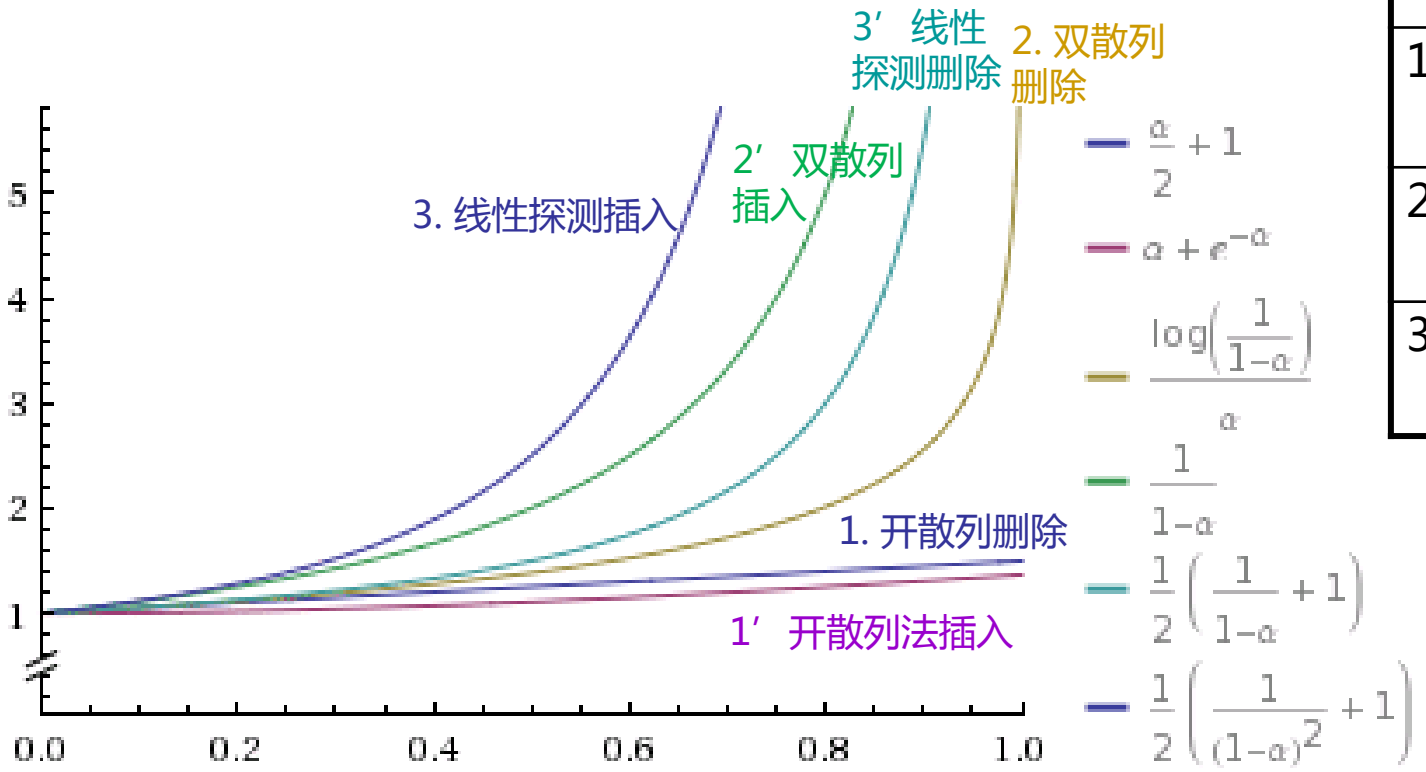
$$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$$

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）

- 用几种不同方法解决碰撞时散列表的平均检索长度



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	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

散列表算法分析结论（1）

- 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 $\alpha=n/M$
  - 随着 α 增加，预期的代价也会增加
  - $\alpha \leq 0.5$ 时，大部分操作的分析预期代价都小于 2（也有人说1.5）
- 实际经验也表明散列表负载因子的临界值是0.5（将近半满）
  - 大于这个临界值，性能就会急剧下降

散列表算法分析结论（2）

- 散列表的插入和删除操作如果很频繁，将降低散列表的检索效率
  - 大量的插入操作，将使得负载因子增加
    - 从而增加了同义词子表的长度，即增加了平均检索长度
  - 大量的删除操作，也将增加墓碑的数量
    - 这将增加记录本身到其基地址的平均长度
- 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新散列
  - 把所有记录重新插入到一个新的表中
    - 清除墓碑
    - 把最频繁访问的记录放到其基地址

思考

- 是否可以把空单元、已删除这两种状态，用特殊的值标记，以区别于“单元被占用”状态？
- 调研除散列以外字典的其他实现方法

数据结构与算法

谢谢聆听

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