实验五——哈希

PB17000002

古宜民

2018.12.27

1. 实验基本要求

输入关键字序列，使用除留余数法构造哈希函数，使用线性探测和拉链法解决冲突，构造哈希表，并计算两个哈希表等概率情况下查找成功和失败的平均查找长度（ASL），给出关键字计算查找次数。

2. 实验实现内容

实现电话号码的哈希存储、查找、插入操作，并可以计算查找成功、失败是的查找长度，可以计算理论和实际的 ASL。

完整代码见：

https://github.com/ustcpetergu/Datastructure/tree/master/Course2018Spring/Hash

3. 实验代码结构和关键代码讲述

程序结构：设立了 Elem 类处理元素、哈希函数和所得关键字；两个哈希类 LinearHash 和 LinkHash 完成哈希表的创建、插入查找等操作。从文件输入随机生成的电话号码并在 main 函数中进行处理分析。

关键代码：

1.随机电话号码：

用 Python 生成。

```
print(random.choice(['139','188','185','136','158','151'])+"".join(random.choice("0123456789") for i in range(8)))
```

2.哈希函数的选取：

选取质数 5003 作为表长，考虑到电话号码的性质，取电话号码中间 4 位与后四位的平

方和取对表长的模作为哈希函数值。

Elem 类：

```
class Elem{
public:
    Elem(){
        number = -1;
    }
    explicit Elem(long long number){
        this->number = number;
    }
    long long GetKey() const{
        return ((number % 10000) * (number % 10000) + (number / 10000 % 10000) * (number / 10000 % 10000)) % PRIME;
    }
    long long number;
};
```

3.哈希表的设计及算法：

线性探测法解决冲突：建立 LinearHash 类，使用一个以 Elem 类型为元素的数组作为存储结构。类的一个主要方法为 int NextAddr(int key0, int cnt) const，即给出在第 cnt 次(前 cnt-1 次查找失败) 对关键字为 key 的元素进行查找时应该去查找的地址，使用线性探测，地址为 key0 + cnt + (key0 + cnt >= HASH_LEN ? -HASH_LEN : 0)。另一个主要方法为 int BaseSearch(Elem elem, bool insert, int& cmp)，即在哈希表中查找 elem 元素，insert 控制是否在未找到时插入，cmp 将返回查找过程中比较的次数。对元素的查找、插入、计算查找长度等操作都可以以 BaseSearch 为基础实现。计算平均查找长度即对每个可能的情况进行查找，再对这些查找长度取加权平均，等概率情况直接取平均值。

BaseSearch 代码：

```
    // 1: found
    // 2: inserted
    // -1: not found
    // -2: table full and cannot insert
    int BaseSearch(Elem elem, bool insert, int& cmp){
        int key0 = elem.GetKey();
```

```
            int key = key0;
            int cnt = 1;
            while(hash[key].number    !=    -1    &&    elem.number    !=
hash[key].number){
                key = NextAddr(key0, ++cnt);
                if(cnt > HASH_LEN){
                    if(insert)
                        return -2;
                    else
                        return -1;
                }
            }
            cmp = cnt;
            if(hash[key].number == elem.number)
                return 1;
            else if(insert){
                hash[key] = elem;
                return 2;
            }
            else
                return -1;
        }
```

包装的查找、插入和比较次数代码:

```
        bool Search(Elem elem){
            int dum = 0;
            int stat = BaseSearch(elem, false, dum);
            return stat > 0;
        }
        bool Insert(Elem elem){
            int dum = 0;
            int stat = BaseSearch(elem, true, dum);
            return stat > 0;
        }
        int CompareTimes(Elem elem){
            int cmp = -1;
            BaseSearch(elem, false, cmp);
            return cmp;
        }
```

两个 ASL 计算的代码：其中，int CompareTimes(int key0)为计算查找关键字 key 直到查找失败的查找长度，失败 ASL 中使用；int CompareTimes(Elem elem)为计算查找元素 elem(成功或失败) 的查找长度，成功 ASL 中使用。查找失败的 ASL 只要对每个哈希表中的每个位置

进行一次计算并取平均即可，查找成功的 ASL 要找出表中所有的（已插入的）元素，分别做一次查找并计算。

```
double AverageCmpTimesFailed(){
    int times = 0;
    for(int i = 0; i < HASH_LEN; i++)
        times += CompareTimes(i);
    return times * 1.0 / HASH_LEN;
}
double AverageCmpTimesSuccess(){
    long long times = 0;
    int elems = 0;
    for(int i = 0; i < HASH_LEN; i++){
        if(hash[i].number != -1) {
            times += CompareTimes(hash[i]);
            elems++;
        }
    }
    if (elems == 0)
        return -1.0;
    return times * 1.0 / elems;
}
```

拉链解决冲突：建立 LinkHash 类，以 Linklist<Elem>** hash;即链表指针的一维数组作为存储结构，查找过程与上 LinearHash 逻辑完全相同，只是改为链表操作。NextAddr 可直接用 p=p->next 代替。

BaseSearch 代码：

```
int BaseSearch(Elem elem, bool insert, int& cmp){
    int key0 = elem.GetKey();
    int cnt = 1;
    Node<Elem>* p;
    p = hash[key0]->head->next;
    while(p && elem.number != p->data.number){
        p = p->next;
        cnt++;
    }
    cmp = cnt;
    if(!p){
        if(insert){
            Status stat = hash[key0]->ListInsert(1, elem);
```

```
                if(stat != OK)
                    return -2;
                return 2;
            }
            else
                return -1;
        }
        else
            return 1;
    }
```

4. 实验结果及分析

进行插入测试并计算理论和实际 ASL，数据如下（表长为 5003）：

元素个数	装填因子	线性探测成功 ASL	线性探测成功理论 ASL	线性探测失败 ASL	线性探测失败理论 ASL	拉链成功 ASL	拉链成功理论 ASL	拉链失败 ASL	拉链失败理论 ASL
1000	0.200	1.109	1.125	1.268	1.281	1.09	1.10	1.017	1.018
3000	0.600	1.631	1.749	3.121	3.619	1.308	1.300	1.152	1.149
5000	1.000	32.637	834.333	2485.05	1.39e6	1.513	1.500	1.371	1.368
9000	1.799	-	-	-	-	1.912	1.899	1.961	1.964

可见拉链处理冲突的 ASL 小于线性探测处理冲突，并且拉链法能够用于元素个数大于哈希表长的情况，且理论 ASL 与实际 ASL 大体符合。进行查找测试，结果均正确。

可改进处：

将文件输入输出和交互式查询结合，增强用户体验；

创建 Hash 父类并继承得 LinearHash、LinkHash 两个子类，充分利用 OOP 思想；

提高程序灵活性，如提供自定义表长、选择多种哈希函数等功能。

5. 实验小结

本次实验实现了有很大理论和实践意义的哈希表程序，并计算对比了理论和实际的 ASL，让我体会到了数学在计算机科学中的重要意义。