**上机报告**

**学号**：1500011398

**姓名**：秦光辉

**题目描述**：

从一个记录名单的数据文件（list.dat）中读入学生的姓名和学号数据使用不同的方法进行查询。

1. 实现用二分法按照学号检索学生.
2. 选择适当散列函数，根据姓名创建一个散列表示的字典，并实现插入学生信息，通过姓名检索，删除的功能，并给出碰撞等情况统计；

**题目分析**：

本题需要使用两个数据结构，顺序字典和哈希表。二分法检索学生使用顺序字典，散列表示的字典使用哈希表。

考虑到顺序字典实现的功能比较简单，我没有采用链表，而使用了数组。顺序字典要排序，否则无法使用二分法检索。

哈希表需要采用恰当的哈希函数，避免出现过多的碰撞，也需要避免产生太大的内存浪费。通常来说负载因子控制在0.5到1.0为宜。List中一共有80个学生，将哈希表的大小设置为100比较合适。

哈希表同时需要实现插入、检索、删除等功能，给出碰撞统计。这些均可以定义在哈希表类中。

**函数和算法分析**：

顺序字典：

顺序字典对应的类为dict，是一个模板类。其类成员包括一个数组，用来存储字典元素，一个表示字典容量的整数，一个表示当前元素数量的整数。为了使得字典容量尽可能大，没有使用int，使用了size\_t。

element<K, V> \*Dic; //字典元素定义在一个数组中

size\_t Max; //Max是字典的最大容量

size\_t n; //n是字典的当前元素数量

其主要函数包括:

dict<K, V>(size\_t n\_Max = INT\_MAX);

构造函数，默认的容量是一个很大的数。

void sort();

排序函数，负责把数组内的元素排序。排序函数调用了另一个函数quicksort。

dict<K, V> &insert(const K &key, const V &value);

插入函数，插入函数不负责排序，默认把元素存到最后。

dict<K, V> &print();

打印函数，负责打印字典中所有元素。

V operator[](K key);

[ ]接收一个关键码，可以返回对应的值。其搜索方式是二分搜索。二分搜索的方法写在了注释中，此不表述。

另外类中还有default constructor, destructor, copy constructor, assignment function，属于类工具，不详述。

Quicksort是一个快速排序函数，经测试效率比较高，决定使用它排序。

哈希表：

哈希表类名是HashTable，这个类是一个模。M板类。其类成员包括哈希表的容量，哈希表的碰撞统计（一个int存储碰撞次数，一个数组存储碰撞位置），存储元素的指针四个成员。分别为：

int m = 0; //哈希表的容量

int n\_collision = 0; //碰撞次数

element<K, V> \*ele = nullptr; //存储元素的数组

element<K, V> \*ele = nullptr; //存储元素的数组

其关键函数包括：

HashTable(int n\_m);

Constructor，其中n\_m表示哈希表的最大容量。

V operator[](K n\_key);

[ ]中输入Key可以检索到所要的value。每次检索的时候，我们先使用哈希函数寻找当前的哈希值。如果哈希值对应的就是所检索的数据，则立刻返回；否则递增地址，直到检索到为止。

V del(K n\_key);

Delete函数的目的是删除关键码所指的元素，同时需要避免前面发生碰撞的元素检索失败。实现方案如下：

1. 先寻找n\_key对应的伪地址。记录为index。将其delete。（即将其key置空）
2. 用space表示当前空位的伪地址。
3. 向下搜索，找到下一个空位，其伪地址记为next\_space。
4. 接下来沿着space向下搜索，如果发现某个元素的哈希值小于space，则把该元素记录到空位上，讲space定位到该元素位置。
5. 当index递增到next\_space时，结束循环。

上述方案中需要考虑到index在递增过程中是否发生了循环，如果循环的话判断条件需要发生变化，在代码中有注释详解，此不表述。

int h(K n\_key);

这是哈希函数。考虑到所有的人名最少有两位，所以使用了下述哈希函数：

((int)(unsigned char)n\_key[0] + (int)(unsigned char)n\_key[1] + (int)(unsigned char)n\_key[2] + (int)(unsigned char)n\_key[3]) % m;

void insert(K n\_key, V n\_value);

这个是插入算法，和另一个函数linear\_search配合使用，可以插入一个新的元素。如果发生了碰撞，则会被记录下来。如果哈希表已满，则函数会assert退出。

另外类中还有default constructor, destructor, copy constructor, assignment function四个函数，作为类工具，注释在代码中，这里不表述。

**结果与分析**：

**1.数据类型的选择**

本次作业中，顺序字典关键码是学号。我首先想到的是使用int来存储学号，但是这种方法不合理，因为学号格式并不整齐，有的学号第一位是0，如果使用了int，我们就有可能丢掉前面的0。而如果我们使用c++ 库中的string存储学号，不但容易取其中的某一位，确定其长度，而且string自带谓语，可以直接排序，不需要重载比较运算符。所以斟酌之后我选择了string。

**2.排序算法的选择**

在顺序字典中，如果每次插入一个元素就进行一次排序，其时间代价是极高的。这种算法效率相当于进行一次插入排序，考虑之后我放弃了这种算法。

还有一种算法是基数排序法，基数排序法效率是相对较高的，但是也存在一个问题。本次作业中的大部分学号前5位都是10000，如果使用基数排序法，就会造成很大浪费，因为其实前5位是不需要排序的。如果放弃前五位，又会出现有的学号无法排序的情况。无论如何都不合适，所以我放弃了这种算法。

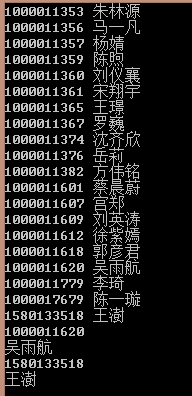
如果使用shell排序，需要确定分组的数目。但是我对shell排序没有任何经验，不好确定分组数目。鉴于我的编程经验不足，我没有采用这种算法。

如果使用交换排序，其效率其实和插入排序差距不大。如果使用交换排序，相比直接插入（每加入一次元素就排序一次），我们不但没有提高效率，而且多调用了一次函数。得不偿失，我没有采用交换排序。

剩下可以采用的排序方法只剩下冒泡排序，快速排序，归并排序。本次数据量较小，三种算法的时间代价相差无几。我在实现过程中使用了快速排序，其效率高于直接插入。剩下的两种算法我还没有测试。如果使用大量学生数据，应该可以测试出三种算法的优劣。当然这也和学号的编码方式有关。我还没有进行这方面的测试。

3.顺序字典输出结果

下面是一些顺序字典的输出结果：



上半部分是字典的全部打印结果，后面是检索（输入学号检索名字）。

**4.哈希函数的取模运算分析**

关于哈希函数取模运算：

在课本中，作者推荐我们把哈希值模一个比容量小的最大质数。真的是这样吗？我测试了一下几个负载因子的情况下不同的模值对应的碰撞次数：

|  |  |  |  |
| --- | --- | --- | --- |
| 容量大小 | 负载因子 | 模值 | 碰撞次数 |
| 82 | 0.9756 | 79 | 43 |
| 82 | 0.9756 | 82 | 36 |
| 100 | 0.8 | 100 | 30 |
| 100 | 0.8 | 97 | 34 |
| 150 | 0.5333 | 150 | 19 |
| 150 | 0.5333 | 149 | 21 |

从上面可以看出，哈希函数的模值取作容量大小时碰撞次数比取作小于容量的最大质数时少。我又做了一组实验：

|  |  |  |  |
| --- | --- | --- | --- |
| 容量大小 | 负载因子 | 模值 | 碰撞次数 |
| 100 | 0.8 | 100 | 30 |
| 100 | 0.8 | 99 | 33 |
| 100 | 0.8 | 98 | 32 |
| 100 | 0.8 | 97 | 34 |
| 100 | 0.8 | 96 | 32 |
| 100 | 0.8 | 95 | 33 |
| 100 | 0.8 | 94 | 32 |
| 100 | 0.8 | 93 | 32 |
| 100 | 0.8 | 92 | 29 |
| 100 | 0.8 | 91 | 34 |
| 100 | 0.8 | 90 | 37 |
| 100 | 0.8 | 89 | 34 |
| 100 | 0.8 | 288 | 33 |
| 100 | 0.8 | 87 | 36 |
| 100 | 0.8 | 86 | 40 |
| 100 | 0.8 | 85 | 37 |
| 100 | 0.8 | 84 | 39 |
| 100 | 0.8 | 83 | 36 |
| 100 | 0.8 | 82 | 35 |
| 100 | 0.8 | 81 | 33 |

从上面两个实验可以看出，模值取容量大小时，碰撞次数最小。所以我在代码实现的时候，使用的模值是容量大小m。

**5.哈希表删除函数的选择**

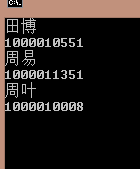
课本上删除算法的实现方式是做一个标记。这种算法难度小，但我认为有一个不好的地方：如果这个字典删除算法使用的几率低，检索算法使用频率高，这种算法无疑会使得更多的元素偏离其哈希值所指的地址。所以我采用了另一种算法：

每次删除的时候，把要删除的元素的key置空，然后向前搜索，遇到的元素哈希值如果小于空位的伪地址，就把该元素移动到空位，把该元素置为空位。一直检索下去，直到遇到下一个空位为止。

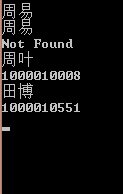
这样我们检索的时候就可以 避免过多的比较，大大提高了检索效率。

**6.哈希表的输出结果**

下面是哈希表的一些输出结果：



这是检索结果。



删除周易之后的检索结果。

**一些可能的改进思路：**

1. 上述讨论的排序算法。
2. 哈希函数的算法，我仅仅是将汉字的各个编码相加，这样的算法并不能够有效避免碰撞。如果能够根据汉字的编码方式单独研发一种哈希函数，应该是最合适的
3. 哈希表的负载因子目前控制在百分之80左右，可以进行测试，观察什么样的负载因子效果最好，即不但节省内存，而且运行速度很快。