SHANGHAI JIAO TONG UNIVERSITY

哈希表设计

吕艺

517021910745

December 14, 2018

1 需求分析

1. 本演示文件的主要需求建立哈希表。具体背景为对文件读入的 30 个人名建立哈希表，进行必要的查表和建表功能。要实现的具体功能有建立哈希表，通过学生的名字查找哈希表，在哈希表中对指定学生作删除标记，以及彻底删去做了了删除标记的学生。

2. 演示文件无需用户输入，程序自动将哈希表中的学生按名字排序后打印学生的名字

3. 程序执行的命令包括:
a. 根据除留余数法将学生放在哈希表的不同位置，并通过线性探测法解决冲突
b. 按顺序打印哈希表中学生的名字
c. 将学生插入到哈希表中
d. 在哈希表中对指定学生作删除标记
e. 彻底删去做了删除标记的学生

4. 测试数据
演示文件中用到的测试数据为文件读入的 30 个中文人名拼音，长度不超过 20 个字符。

2 概要设计

本演示文件中的哈希表采用的是除留取余的哈希函数，以及线性探测法的冲突解决方法。为了实现这一功能，演示文件中定义了 *student* 结构体，父类 *hashTable*, 以及派生类 *closeHashTable*

1) 结构体 *student*

数据对象：char * name = NULL;　　int state;

基本操作：student()state = 0;

操作条件：*student* 还未被初始化

操作结果：默认构造 *student* 类的对象

student(char * names) name = names; state = 0;

初始条件：*student* 类还未被初始化

操作结果：根据名字初始化 *student* 类的对象

char * getName() const {return name;}

初始条件：*student* 类已经被初始化

操作结果：返回 *student* 类的实例对象的名字

int getState () constreturn state;

操作条件：*student* 类已被初始化

操作结果：返回 *student* 类的实例对象的状态

void changeState(int state1)state = state1;

操作条件：*student* 类已经被初始化

操作结果：返回 *student* 类的实例对象的状态改变为 *state*1

void setName(char* c) name = c;

操作条件：*student* 类被初始化

操作结果：给 *student* 类的对象设置名字

2) 父类 *hashTable*

基本操作：virtual bool find(const Type & x) const= 0;

virtual bool insert(const Type & x) = 0;

char * getName() const {return name;}

virtual bool remove(const Type & x) = 0;

3) 派生类 *closeHashTable*

数据对象：student * array;　　int size;

int(*key)(const Type x);

基本操作：closeHashTable(int length, int(*f)(const Type x));

操作条件：*closeHashTable* 还未被初始化

操作结果：根据哈希函数将学生插入到哈希表的特定位置

bool insert(const Type & x);

初始条件：*closeHashTable* 类已被初始化

操作结果：将该学生插入到指定位置

bool remove(const Type & x);

初始条件：*closeHashTable* 类已经被初始化

操作结果：在哈希表中对指定学生作删除标记

bool find(const Type & x) const;

操作条件：*closeHashTable* 类已被初始化

操作结果：返回指定学生是否在该表中存在的布尔值

void changeState(int state1)state = state1;

操作条件：*student* 类已经被初始化

操作结果：返回 *student* 类的实例对象的状态改变为 *state*1

void rehash();

操作条件：*closeHashTable* 类已被初始化

操作结果：彻底删除哈希表中做了删除标记的学生

2. 本程序包括两个模块：

1) 主函数模块

```
int main() {
    ifstream in;
    in.open("name.txt");
    int (*f)(const student A) = hash1;
    closeHashTable<student> a(30, f);
    char tmp[Num][20];
    for(int i = 0;i < Num; i++){
    in » tmp[i];
    student A = student(tmp[i]);
    a.insert(A);}
    for(int i = 0 ; i< 30 ; i++)
        if (a.array[i].getState() == 1)
        { cout « a.array[i].getName() « endl; }
    return 0; }
```

2) 父类 *hashTable* 单元模块--定义派生类必须实现的功能

3) 派生类 *closeHashTable* 单元模块--具体实现功能

3 详细设计

1）*hashTable* 单元模块

```
template<class Type>
class hashTable{                                    //建立哈希表的父类，规定其在派生类中必须实现的操作
public:
    virtual bool find(const Type & x) const= 0;
    virtual bool insert(const Type & x) = 0;
    virtual bool remove(const Type & x) = 0;
};
```

2）*closeHashTable* 单元模块

```
template<class Type>
class closeHashTable:public hashTable<Type>{
public:
    student * array;                //存放学生的哈希表
    int size;                       //学生数量
    int(*key)(const Type x);        //哈希函数指针
public:
    closeHashTable(int length, int(*f)(const Type x));      //哈希函数的构造函数
    bool insert(const Type & x);        //将学生插入到哈希表的指定位置
    bool remove(const Type & x);        //在哈希表中对指定学生作删除标记
    bool find(const Type & x) const;    //在哈希表中寻找是否有指定学生
    void rehash();                      //彻底删除哈希表中有删除标记的学生
    int getSize(){return size;}         //返回学生个数
};


//初始化哈希表
template<class Type>
closeHashTable<Type>::closeHashTable(int length, int(*f)(const Type x))
{   size = length;
    array = new student[length];
    key = f;
}

template<class Type>
bool closeHashTable<Type>::insert(const Type &x) {
    int initPos,pos;
    initPos = pos = key(x);
    static int num = 0;
    do{

        if(array[pos].getState() != 1)          //当哈希表该位置为空时直接插入该学生，返回插入成功
        {
            array[pos].setName(x.getName());
            array[pos].changeState(1);xu
            num ++;
            return true;
```

```
        }

        //当该学生已在哈希表中，则不重复插入，返回插入成功
        if(array[pos].getState() == 1 && strcmp(array[pos].getName(),x.getName()) == 0)
            return true;

        pos = (pos + 1) % size;

    }while(pos != initPos);          //当该学生没有地方可以放下时，返回插入失败
    return false;
}


template <class Type>
bool closeHashTable<Type>::remove(const Type & x){
    int initpos,pos;
    initpos = pos =key(x);
    do{
        if(array[pos].getState() == 0)    return false;           //当删除的位置节点为空时，返回插入失败
        if(array[pos].getState() == 1 && strcmp(array[pos].getName(),x.getName()) == 0)
        {array[pos].changeState(2);  return true;}       //当需要删除的节点存在且姓名匹配时，对其做上删除
             标记，并返回删除成功
        pos = (pos + 1) % size;
    }while(pos != initpos);          //当该节点已被删除时，寻找下一个节点，若一圈后都没找到，返回删除失败
    return false;
}


template <class Type>
bool closeHashTable<Type>::find(const Type &x) const {
    int initpos,pos;
    initpos = pos = key(x);
    do{
        if(array[pos].getState() == 0) return false;         //当该位置为空时，返回寻找失败
        if(array[pos].getState() == 1 )                      //当该位置不为空且名字匹配时，返回寻找成功
        {
            const student tmp = array[pos];
            if( strcmp(tmp.getName(), x.getName()) == 0)
            return true;
        }
        pos = (pos + 1) % size;          //当该节点已被删除，则查找他的下一个节点
    }while(pos != initpos);              //若一圈后还未找到，则返回查找失败
    return false;
}


template<class Type>
void closeHashTable<Type>::rehash(){
    student * tmp = array;
    array = new student[size];
```

```
    for(int i = 0; i < size; i++)
    {   int state = tmp[i].getState();      //新建哈希表，将没被删除的节点插入到新表中，并用新表代替旧表
        if(state == 1)
            insert(tmp[i]);
    }
    delete tmp;
}
```

3）主函数模块

```
#define Num 30                          //定义同学个数为30

int hash1(const student A)  {           //定义采用除留余数法的哈希函数
    int sum = 0;
    for(int i = 1; i < 5; i++)
        sum += int(A.getName()[i]);
    return sum % NUM ;
}

int main() {
    ifstream in;
    in.open("name.txt");                //读入30个人名
    int (*f)(const student A) = hash1;  //定义指向哈希函数的函数指针
    closeHashTable<student> a(30, f);   //利用默认构造函数新建对象
    char tmp[Num][20];                  //每一行代表一个同学的名字
    for(int i = 0;i < Num; i++)
    {
        in >> tmp[i];
        student A = student(tmp[i]);
        a.insert(A);                    //将同学插入到哈希表的对应位置
    }

    for(int i = 0 ; i< 30 ; i++) {
        if (a.array[i].getState() == 1)         //将未被删除的同学的名字按他们在哈希表中的位置输出
        { cout << a.array[i].getName() << endl; }
    }
        return 0;
}
```

4 调试分析

1. 一开始本演示文件仅设计了 *closeHashTable* 类，将所有操作都直接定义在了这个类中，这导致程序的可移植性较低，为了更好地规范哈希表的基本操作，程序中又设计了 *hashTable* 父类，将 *closeHashTable* 类作为 *hashTable* 类的派生类，规范了哈希表的基本操作。

2. 算法的复杂度分析

1）时间复杂度由于采用数组的形式存储学生的名字，各种操作的算法复杂度比较合理。*closeHashTable* 类的构造函数的时间复杂度是 $O(1)$，函数 $insert$,$remove$,$find$,$rehash$ 的时间复杂度均为 $O(n)$。

closeHashTable 类的初始化函数仅将学生个数，名字矩阵和指向哈希函数的函数指针赋值，因此这个函数的时间复杂度为 $O(1)$。

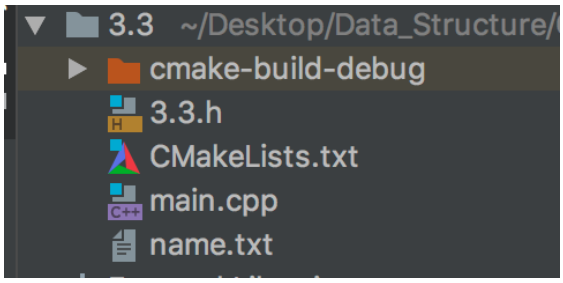
$insert$,$remove$,$find$ 和 $rehash$ 函数的时间复杂度都与元素个数成正比，因此时间复杂度为 $O(n)$。

2）空间复杂度

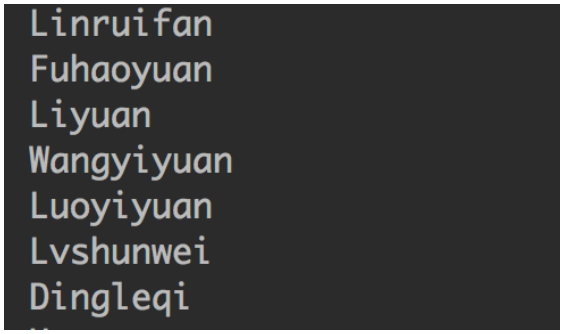
closeHashTable 类模块的空间复杂度与叶节点的个数成正比，即 $O(n)$。主函数模块的复杂度取决于定义主函数作用域中的 *closeHashTable* 类，故空间复杂度也为 $O(n)$。

5 用户手册

1. 本程序以 Jetbrains Clion 2018.2.5, 采用 C++ 11 标准，程序以项目方式组织 (project)，如图 1 所示：



2. 依次点击菜单"Run"->build, 再点击"Run"，程序就执行哈希表的生成，并按哈希表的顺序打印对应学生的名字



6 测试结果

程序开始后读入"name.txt" 中的 30 个学生的名字，并根据哈希函数对他们进行排序并将排序后的学生名字依次打印在屏幕上。

7 附录

源程序文件名清单

main.cpp　　//主函数

hashTable.h　　//哈希表单位模块