山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：201700130033	姓名： 武学伟	班级： 2017 级 2 班
实验题目：散列表		
实验学时：4	实验日期： 2018.11.18	

实验目的：

1.掌握散列表结构的定义与实现。

2.掌握散列表结构的使用。

软件环境：

Win10home，sublime

1. 实验内容（题目内容，输入要求，输出要求）

1）分别使用线性开型寻址和链表散列解决冲突，创建散列表类；

2）使用散列表设计实现一个字典，实现字典的插入，查询，删除操作。

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

数据结构：

散列表，数组，有序链表。

算法：

1）开型寻址：

首先确定元素的索引值，根据模运算确定桶，若是桶的位置为空或者匹配，则索引就是桶的位置，否则就向后移动，直至为空或者匹配，从而确定某个元素的索引值。

①查询

确定索引值，若索引值对应的位置为空或者不匹配，则说明该元素不再散列表中，返回-1，否则返回索引值。

②插入

确定索引值，若索引所对应的位置为空，直接插入新的元素，
若索引所对应的位置不为空，则更新这个元素，
否则，散列表已满，无法继续插入。

③删除

确定索引值，将其置空（即为删除操作），并标记这个空的位置，与其他空的位置不同的是，这个空的位置是经过删除之后才空的，为了之后向其中添加数据。

然后从被删除元素的下一位开始遍历，直至再次碰到删除元素的位置。

遍历中，若当前的元素不为空时，判断他的当前索引值是否与他应在的索引位置相同，若不同，则说明该元素需要移动（不同的原因是因为之前的元素被删除产生了空位），将其插入到之前被标记的空位，然后将其原本的位置置空，并更新标记的空位，并记录一次移动。

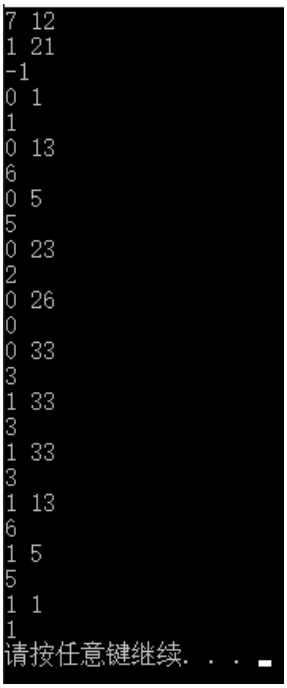
遍历完成之后，输出移动的次数。

2）链表：

建立有序链表类 sortedChain，实现其中的插入，查询，删除操作。然后建立适当容量的元素类型为 sortedChain 数组，再次实现插入，查询，删除操作。

①链表查询
遍历链表，当值匹配时，返回链表元素，否则返回 NULL
②链表插入
定义两个链表元素类指针 current，前驱指针 tp，遍历链表，到插入的位置（位置由 pair 类型的 first 元素决定），若是已经有了元素，则直接更新值即可；若是在链表头插入，直接更改 firstNode 的值；若是在链表中插入，在 current 和 tp 之间插入一个新的值，并使链表的元素数量值加 1。
③链表删除
采用和插入类似的方法直接遍历链表。若是删除头节点，直接更改 firstNode 值，若是在中间删除，使前驱指针 tp 的 next 为 current 的 next，跳过 current，并删除 current，然后将链表元素数量减 1。
若是没有遍历到，说明删除失败，输出删除失败的提示信息。
④散列表查询
调用链表的查询函数，若为空输出 Not Found，否则输出元素的链表长度。
⑤散列表插入
确定桶，确定桶的大小，然后调用链表的插入函数，插入完成后，判断链表的大小是否发生变化，若发生，则改变桶对应链表的元素数量。
⑥散列表删除
直接调用对应桶的删除函数
3. 测试结果（测试输入，测试输出，结果分析）
测试一（开型寻址）：
输入：
7 12
1 21
0 1
0 13
0 5
0 23
0 26
0 33
1 33
1 33
1 13
1 5
1 1
输出：
7 12
1 21
-1
0 1
1
0 13
6
0 5

5
0 23
2
0 26
0
0 33
3
1 33
3
1 33
3
1 13
6
1 5
5
1 1
1
请按任意键继续. . .

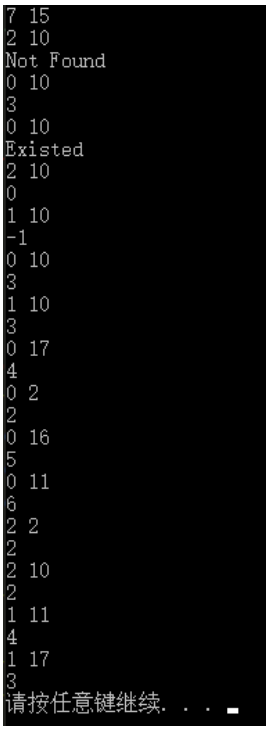


测试二（开型寻址）：

输入：

7 15
2 10
0 10
0 10
2 10
1 10
0 10

```
1 10
0 17
0 2
0 16
0 11
2 2
2 10
1 11
1 17
输出:
7 15
2 10
Not Found
0 10
3
0 10
Existed
2 10
0
1 10
-1
0 10
3
1 10
3
0 17
4
0 2
2
0 16
5
0 11
6
2 2
2
2 10
2
1 11
4
1 17
3
请按任意键继续. . .
```



测试三（链表）：

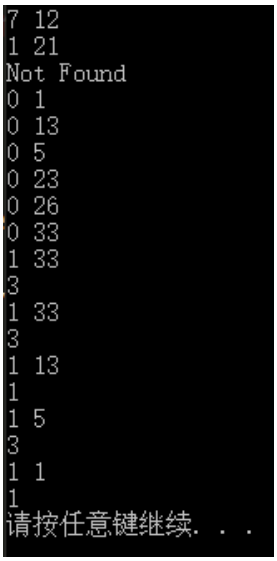
输入：

7 12
1 21
0 1
0 13
0 5
0 23
0 26
0 33
1 33
1 33
1 13
1 5
1 1

输出：

7 12
1 21
Not Found
0 1
0 13
0 5
0 23
0 26
0 33
1 33

3
1 33
3
1 13
1
1 5
3
1 1
1
请按任意键继续. . .



测试四（链表）：

输入：

7 15
2 10
0 10
0 10
2 10
1 10
0 10
1 10
0 17
0 2
0 16
0 11
2 2
2 10
1 11
1 17

输出：

7 15

```
2 10
Delete Failed
0 10
0 10
Existed
2 10
0
1 10
Not Found
0 10
1 10
1
0 17
0 2
0 16
0 11
2 2
1
2 10
1
1 11
1
1 17
1
请按任意键继续. . .
```

```
7 15
2 10
Delete Failed
0 10
0 10
Existed
2 10
0
1 10
Not Found
0 10
1 10
1
0 17
0 2
0 16
0 11
2 2
1
2 10
1
1 11
1
1 17
1
请按任意键继续. . .
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

结果分析：
数据正确。
开型寻址的删除时间复杂度为 $O(n^2)$。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
/*Ex8_1.cpp*/
/*散列表之开型寻址*/
#include <iostream>
#include <string>
using namespace std;

template <class K, class E>
class hashTable
{
    public:
        hashTable(int theDivisor = 11);  //构造函数
        ~hashTable() {delete [] table;}  //析构函数

        bool empty() const {return dSize == 0;}  //判断是否为空
        int size() const {return dSize;}           //返回散列表的元素数量
        void find(const K&) const;                //查找
        void insert(const pair<const K, E>&);     //插入
        void erase(const K&);                     //删除

    private:
        int search(const K&) const;               //根据关键字搜索
        pair<const K, E>** table;
        int dSize;
        int divisor;
};

/*构造函数*/
template <class K, class E>
hashTable<K,E>::hashTable (int theDivisor)
{
    divisor = theDivisor;
    dSize = 0;

    table = new pair<const K, E>* [divisor];
    for (int i=0; i<divisor; i++)
        table[i] = NULL;
}
```

```
/*根据关键字搜索*/
template <class K, class E>
int hashTable<K,E>::search (const K& theKey) const
{
    int home = theKey % divisor;  //设置起始桶
    int i = home;                 //从起始桶开始
    do
    {
        if (table[i] == NULL || table[i]->first == theKey)  //为空或者结果匹配
            return i;           //返回检索索引
        i = (i + 1) % divisor; //否则跳转至下一个元素
    }
    while (i != home);
    return i;   //若再次碰到 j==i 则说明表满
}

/*查找*/
template<class K, class E>
void hashTable<K,E>::find(const K& theKey) const
{
    int index = search(theKey);      //通过查询函数找到 theKey 对应的数组元素下标

    if (table[index] == NULL || table[index]->first != theKey)//判断对应的数组元素是否符合要求
    {//不符合要求就输出-1
        cout << "-1" << endl;
        return;
    }
    cout << index << endl;   //符合要求输出元素下标
    return;
}

/*插入*/
template<class K, class E>
void hashTable<K,E>::insert(const pair<const K, E>& thePair)
{
    int index = search(thePair.first);  //搜索散列表，查找匹配数对

    if (table[index] == NULL)
    {
        table[index] = new pair<const K, E> (thePair);
        dSize++;
```

```
        cout << index << endl;    //输出插入的元素下标
    }
    else
    {
        if (table[index]->first == thePair.first)
        {
            table[index]->second = thePair.second;   //覆写相同关键字的数对
            cout << "Existed" << endl;     //已经存在，输出 Existed
        }
        else
        {
           cout << "It's a full table" << endl;
            string errorFull = "It's a full table";
            throw errorFull;  //表已满
        }
    }
}

/*删除*/
template<class K, class E>
void hashTable<K,E>::erase(const K& theKey)
{
    int index = search(theKey);
    int home = index;   //确定开始的位置
    int count = 0;           //删除过程中移动的元素数量
    if (table[index] == NULL)
    {
        cout << "Not Found" << endl;
        return;
    }
    else
    {
        table[index] = NULL;
        int emptyindex = index;    //现有散列表删除之后的空值索引
        do
        {
            index = (index + 1) % divisor;
            if (table[index] != NULL)    //索引值对应元素不为空
            {
                if(index != search(table[index]->first))    //索引值元素不为应在的位置
                {
                    table[emptyindex]   =   new   pair<const   K,   E>
```

```
(*table[index]);//插入到前一个空值的位置
                table[index] = NULL;  //将转移的值设为空
                emptyindex = index;   //更新空值索引
                count++;              //记录移动元素的数量
            }
        }

    }
    while (index != home);
    cout << count << endl;   //输出删除中移动的元素数量
    return;
}
}

int main()
{
    int D;  //除数
    int m;  //操作数
    int opt;//操作
    cin >> D;
    cin >> m;
    hashTable<int, int> htable(D);
    for (int i=0; i<m; i++)
    {
        cin >> opt;
        switch(opt)
        {
            case 0:
                {
                    pair<int, int> temp;
                    cin >> temp.first;
                    temp.second = temp.first;
                    htable.insert(temp);
                    break;
                }
            case 1:
                {
                    int temp;
                    cin >> temp;
                    htable.find(temp);
                    break;
                }
            case 2:
                {
```

```
                    int temp;
                    cin >> temp;
                    htable.erase(temp);
                    break;
                }
            default:
                break;
        }
    }
}

/*Ex8_2.cpp*/
/*散列表之链表*/
#include <iostream>
using namespace std;

/*散列表节点*/
template <class K, class E>
struct pairNode
{
    pair<const K, E> element;
    pairNode<K,E>* next;

    pairNode(const pair<const K,E>& thePair):element(thePair){}
    pairNode(const pair<const K,E>& thePair, pairNode<K,E>*
theNext):element(thePair)
    {
        next = theNext;
    }
};

/*有序链表*/
template <class K, class E>
class sortedChain
{
    public:
        sortedChain()              //构造函数
        {
            firstNode = NULL;
            dSize = 0;
        }
        ~sortedChain();              //析构函数

        bool empty() const {return dSize == 0;}      //判断是否为空
```

```
        int size() const {return dSize;}            //返回链表元素数
量
        pair<const K,E>* find(const K&) const;       //查找
        void erase(const K&);                        //删除
        void insert(const pair<const K,E>&);         //插入

    private:
        pairNode<K,E>* firstNode;     //指向有序链表首节点的指针
        int dSize;                    //有序链表的元素数量

};

/*析构函数*/
template <class K, class E>
sortedChain<K,E>::~sortedChain()
{
    while (firstNode != NULL)
    {
        pairNode<K,E>* nextNode = firstNode->next;
        delete firstNode;
        firstNode = nextNode;
    }
}

/*查找*/
template <class K, class E>
pair<const K, E>* sortedChain<K,E>::find(const K& theKey) const
{
    pairNode<K,E>* currentNode = firstNode;
    while  (currentNode  !=  NULL  &&  currentNode->element.first  !=
theKey)
        currentNode = currentNode->next;  //遍历链表
    if (currentNode != NULL && currentNode->element.first == theKey)
        return &currentNode->element;     //若匹配，则返回该元素
    return NULL;                          //若未查到，返回 NULL
}

/*插入*/
template <class K, class E>
void sortedChain<K,E>::insert(const pair<const K, E>& thePair)
{
    pairNode<K,E>* currentNode = firstNode;
    pairNode<K,E>* tp = NULL;  //p 的前驱指针
    while  (currentNode  !=  NULL  &&  currentNode->element.first  <
```

```
thePair.first)
    {//遍历链表,插入的位置是 tp 的后面
        tp = currentNode;
        currentNode = currentNode->next;
    }
    if (currentNode != NULL && currentNode->element.first ==
thePair.first)
    {//值匹配，更新值
        currentNode->element.second = thePair.second;
        cout << "Existed" << endl;
        return;
    }
    pairNode<K,E>* newNode = new pairNode<K,E>(thePair, currentNode);
    if (tp == NULL)     //首节点插入
        firstNode = newNode;
    else              //中间节点插入
    {
        tp->next = newNode;
        newNode->next = currentNode;
    }
    dSize++;
}

/*删除*/
template <class K, class E>
void sortedChain<K,E>::erase(const K& theKey)
{
    pairNode<K,E>* currentNode = firstNode;
    pairNode<K,E>* tp = NULL;  //p 的前驱指针
    while (currentNode != NULL && currentNode->element.first <
theKey)
    {//遍历链表进行搜索
        tp = currentNode;
        currentNode = currentNode->next;
    }
    if (currentNode != NULL && currentNode->element.first == theKey)
    {//搜索到预定目标
        if (tp == NULL)  //删除首节点
            firstNode = currentNode->next;
        else
            tp->next = currentNode->next;  //非首节点
        delete currentNode;
        dSize--;
        cout << dSize << endl;
```

```
        return;
    }
    else//删除失败
    {
        cout << "Delete Failed" << endl;
        return;
    }
}

/*链式散列表*/
template <class K, class E>
class chainHash
{
    public:
        chainHash(int theDivisor = 7);              //构造函数
        ~chainHash() {delete []table;}              //析构函数
        bool hempty() const {return dhSize == 0;} //判断是否为空
        int hsize() const {return dhSize;}          //返回桶的数量
        void hfind(const K&) const;                 //查找
        void hinsert(const pair<const K, E>&);      //插入
        void herase(const K&);                      //删除
    private:
        sortedChain<K,E>* table;
        int dhSize;
        int dhvisor;
};

/*构造函数*/
template <class K, class E>
chainHash<K,E>::chainHash(int theDivisor)
{
    dhvisor = theDivisor;
    dhSize = 0;
    table = new sortedChain<K,E> [dhvisor];
}

/*查找*/
template <class K, class E>
void chainHash<K,E>::hfind(const K& theKey) const
{
    if (table[theKey % dhvisor].find(theKey) == NULL)
    {
        cout << "Not Found" << endl;
        return;
```

```
    }
    cout << table[theKey %dhvisor].size() << endl;
}

/*插入*/
template <class K, class E>
void chainHash<K,E>::hinsert(const pair<const K, E>& thePair)
{
    int bucket = thePair.first % dhvisor;      //确定桶
    int bucketSize = table[bucket].size();
    table[bucket].insert(thePair);
    if (table[bucket].size() > bucketSize)
        dhSize++;          //若是插入新的元素，则 dhSize 加一
}

/*删除*/
template <class K, class E>
void chainHash<K,E>::herase(const K& theKey)
{
    table[theKey % dhvisor].erase(theKey);
}

int main()
{
    int D;  //除数
    int m;  //操作数
    int opt;//操作
    cin >> D;
    cin >> m;
    chainHash<int, int> htable(D);
    for (int i=0; i<m; i++)
    {
        cin >> opt;
        switch(opt)
        {
            case 0:
                {
                    pair<int, int> temp;
                    cin >> temp.first;
                    temp.second = temp.first;
                    htable.hinsert(temp);
                    break;
                }
            case 1:
```

```
                {
                    int temp;
                    cin >> temp;
                    htable.hfind(temp);
                    break;
                }
            case 2:
                {
                    int temp;
                    cin >> temp;
                    htable.herase(temp);
                    break;
                }
            default:
                break;
        }
    }
    return 0;
}
```