河南工業大學

课　程　设　计

课程设计名称：数据结构课程设计

题　　目：散列表的设计与实现

专 业 班 级：软件 1601

学 生 姓 名：高天

学　　号：201616030213

指 导 教 师：谭玉波

课程设计时间：2018.12.24-2019.1.4

软件工程 专业课程设计任务书

学生姓名	高天	专业班级	软件 1601	学号	201616030213
题 目	散列表的设计与实现				
课题性质	A．工程设计		课题来源	D．自拟课题	
指导教师	谭玉波		同组姓名	无	
主要内容	综合应用所学知识，设计完成一个散列表实现的电话号码查找系统。本系统拟实现以下功能： 【基本要求】 1) 设每个记录有下列数据项：电话号码、用户名、地址； 2) 从键盘输入各记录，分别以电话号码和用户名为关键字建立散列表； 3) 采用一定的方法解决冲突； 4) 查找并显示给定电话号码的记录； 5) 查找并显示给定用户名的记录。 【进一步完成内容】 1) 系统功能的完善； 2) 设计不同的散列函数，比较冲突率； 3) 在散列函数确定的前提下，尝试各种不同类型处理冲突的方法，考察平均查找长度的变化。				
任务要求	综合运用和融合所学理论知识，提高分析和解决实际问题的能力，使用C/C++语言设计一个散列表实现的通讯录系统。 完成课程设计报告，报告中对关键部分给出图表说明。要求格式规范，工作量饱满。				
参考文献	[1] 严蔚敏, 吴伟民. 数据结构（C 语言版）[M]. 北京:清华大学出版社, 2012. [2] 陈越, 何钦铭等. 数据结构（第 2 版）[M]. 北京:高等教育出版社,2016 [3] Thomas H.Cormen 等. Introduction to Algorithms, third edition [M]. 北京:机械工业出版社, 2012. [4] https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html [OL].				
审查意见	**指导教师签字：** **教研室主任签字：** **2018 年 12 月 3 日**				

目 录

1 需求分析

本系统为电话号码查找系统，本系统最频繁的操作为查询功能，查询速度的快慢对此系统有至关重要的影响，因此应该选择合适的数据结构来进行设计。散列表可以实现 O(1)的快速查找，用 Hash 数据结构作为底层存储结构较为合适。本系统应首先实现 Hash 表的基本结构和操作，在此基础上构建电话号码查找系统。电话号码查找系统包括若干数据项：电话号码、用户名、地址，可以键盘输入或文件批量导入记录，既可以使用电话号码作为索引建立 Hash 表，也可以使用姓名作为索引建立 Hash 表，并通过电话号码和姓名进行查找记录。更进一步，在设计 Hash 数据结构时，可设计不同的 Hash 函数及采用不同的冲突解决算法，来比较性能的差异。具体功能如下：

1.1 Hash 表设计

设计 Hash 表的 ADT，设计 Hash 表的存储结构以及基本操作，设计不同的 Hash 函数对字符串进行散列，设计不同的冲突解决策略，如链地址法、线性探测法等。Hash 表的结构由 key-value 组成，基本操作有添加元素、查找元素、删除元素、遍历元素、建表、销毁表等，并且设计可以计算平均查找长度（ASL）的函数，以此来比较不同的 Hash 函数使用相同的冲突解决算法，以及相同的 Hash 函数使用不同的冲突解决算法的优劣。为方便 Hash 表的使用，Hash 表可自动扩容。

1.2 添加与导入记录

系统可采用两种方式导入信息：手动添加与文件批量导入。手动添加方式用户通过交互性界面一步一步输入待添加记录的电话号码、姓名、地址，可连续进行添加。文件批量导入方式，预先将记录以指定格式写入文件，系统再从指定文件中批量导入。

1.3 查询记录

查询记录功能为此系统主要功能，可采用两种查询方式：按电话号码查询与按姓名查询。如果是按照电话号码建立索引，按电话号码查询将迅速完成，如果是按照姓名建立索引，按姓名查询将迅速完成。

1.4 不同 Hash 函数比较

设计多种 Hash 函数，使用同一冲突解决方法比较其 ASL，研究不同 Hash 函数对于冲突率的影响。

1.5 不同冲突解决方法比较

设计多种冲突解决方法，使用同一 Hash 函数比较其 ASL，研究不同的冲突解决方法对于 ASL 的影响。

2 概要设计

2.1 数据类型的定义

1) 使用链地址法的 Hash ADT 设计

```
#define MAXCAPACITY 1 << 30
#define LOADFACTOR 0.75f
#define MAXKEYLEN 255

typedef AddList ElemType;
struct Node
{
    char key[MAXKEYLEN];
```

```
    ElemType value;
    Node *next;
};

struct HashTable
{
    int capacity;
    int size;
    Node *table;
};

int NextPrime(int n);
void InitHashTable(HashTable &hash_table, int init_capacity);
unsigned int SumHash(const char *key, int table_size);
unsigned int ShiftHash(const char *key, int table_size);
unsigned int ELFHash(const char *key, int table_size);

bool Put(HashTable &hash_table, const char *key, ElemType value, ElemType
&old_value, unsigned int (*Hash)(const char *key, int table_size));

bool Get(HashTable &hash_table, const char *key,  ElemType &value,
unsigned int (*Hash)(const char *key, int table_size));

bool Remove(HashTable &hash_table, const char *key, ElemType &value,
unsigned int (*Hash)(const char *key, int table_size));

void DelHashTable(HashTable &hash_table);
void TraverseHashTable(HashTable &hash_table, void (*visit)(ElemType v));
double GetASL(HashTable &hash_table, unsigned int (*Hash)(const char *key,
int table_size));
```

2) 使用线性探测法的 Hash　ADT 设计

```
typedef AddList ArrElemType;
// 散列单元状态类型，分别对应：有合法元素、空单元、有已删除元素
typedef enum { Legitimate, Empty, Deleted } NodeType;

struct HashNode {
    char key[MAXKEYLEN];
    ArrElemType value;
    NodeType type;
};

struct ArrayHashTable {
    int size;
    int capacity;
    HashNode *table;
};

void InitArrayHashTable(ArrayHashTable &hash_table, int init_capacity);
void DelArrayHashTable(ArrayHashTable &hash_table);
bool IsFull(ArrayHashTable &hash_table);
bool LinearDelete(ArrayHashTable &hash_table, const char *key);

bool LinearGet(ArrayHashTable &hash_table, const char *key, ArrElemType
&value);

bool LinearGetNum(ArrayHashTable &hash_table, const char *key,
ArrElemType &value, int &num);

int LinearPut(ArrayHashTable &hash_table, const char *key, ArrElemType
value, ArrElemType &old_value);
```

3) 电话查找系统数据类型定义

```
struct AddList {
    char phone_num[MAXPHONENUM];
    char name[MAXNAME];
    char address[MAXADDRESS];

    AddList() {}
    AddList(const char *phone_num, const char *name, const char *address)
    {
        strcpy(this->phone_num, phone_num);
        strcpy(this->name, name);
        strcpy(this->address, address);
    }
};
```

2.2 功能模块结构图

根据需求分析，为了满足用户的功能需求，将系统划分为以下几个模块：Hash 表模块、查询记录模块、添加记录模块、导入记录模块、Hash 函数比较模块、冲突解决办法比较模块。其中 Hash 表模块包括链表实现的 Hash 表子模块和线性探测法实现的 Hash 表子模块，查询记录模块包括电话号码查询子模块和姓名查询子模块。

图 1 模块结构图

2.3 Hash 模块概要设计

1) 链地址法 Hash 表设计

链地址法 Hash 表使用链表来解决冲突，Hash 表维护一定长度的链表数组，对于 Hash 值相同的元素，将其插入 Hash 值对应的数组上的链表中。结构如下图所示，为长度为 7 的 Hash 表：

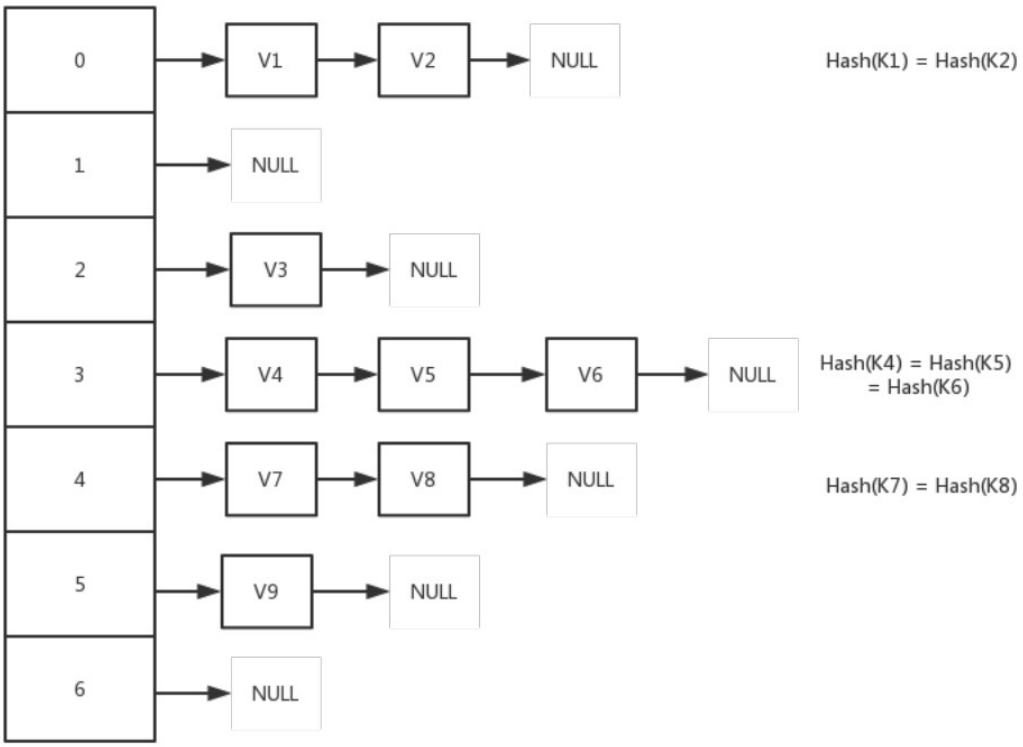

图 2 链地址法 Hash 表结构图

2) 线性探测法 Hash 表设计

使用数组保存元素，插入元素时，先算出 Hash 值，将其放入 Hash 值所对应的位置，如果有冲突，使用线性探测法，地址公式为：

Hi = (Hash(key) + i) % size　　i = 1, 2, 3, 4…… size-1

其中 size 为 Hash 数组的总大小，结构图如下：

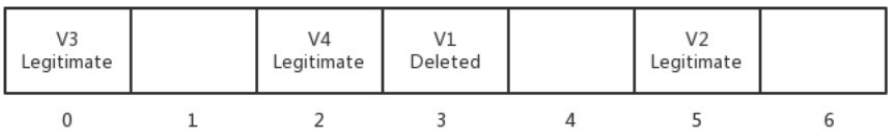

图 3 线性探测法 Hash 表结构图

3 运行环境

硬件环境：

cpu：Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz × 4

内存：8G

软件环境：

OS：Ubuntu 18.04.1 LTS 64 位

Linux 内核：Linux version 4.4.0-17134-Microsoft

4 开发工具和编程语言

编辑软件：

Visual Studio Code 1.30.1

编译软件：

GNU G++ version 7.3.0

构建工具：

GNU Make 4.1

编程语言：

C++语言

5 详细设计

在概要设计的基础上，对每个模块进行详细设计。下面分别列出各个模块的详细设计。

5.1 Hash 函数设计

Hash 表的基本原理为：给出一个 Key，调用 Hash 函数，算出该元素所在的位置。Hash 函数的设计有多种方案，这里给出三种对字符串进行散列的函数。

1) 字符串各位相加

基本思想为将字符串的各位值累加，再求余得到最终位置，代码如下：

```
unsigned int SumHash(const char *key, int table_size) {
    unsigned int h = 0;

    while (*key)
        h += *key++;

    return h % table_size;
}
```

2) 字符串循环移位相加

公式为：

$$h(key) = \left(\sum_{i=0}^{n-1} key\,[n-i-1] \times 32^{i}\right) \%table\_size$$

此散列方法涉及到所有的 n 个字符，且散列分布较好，代码如下：

```
unsigned int ShiftHash(const char *key, int table_size) {
    unsigned int h = 0;
    while (*key)
        h = (h << 5) + *key++;
    return h % table_size;
}
```

3) ELF Hash 方法

该散列方法同样为对字符的 ASCII 编码值进行计算，ELFhash 函数能够比较均匀地把字符串分布在散列表中，代码如下：

```
unsigned int ELFHash(const char *key, int table_size) {
    unsigned long h = 0;
    unsigned long x = 0;
```

```
        while (*key) {
            h = (h << 4) + (*key++);
            if ((x = h & 0xF0000000L) != 0) {
                h ^= (x >> 24);
                h &= ~x;
            }
        }
        return h % table_size;
    }
```

5.2 链地址法 Hash 设计

1) Hash 表初始化

通过初始化函数可实现对 Hash 表的初始化，用户可自定义初始 Hash 表容量大小，初始化函数保证哈希表容量为一个素数，这样可使得散列均匀。int NextPrime(int n) 函数返回大于 n 的下一个素数。流程图及代码如下所示：

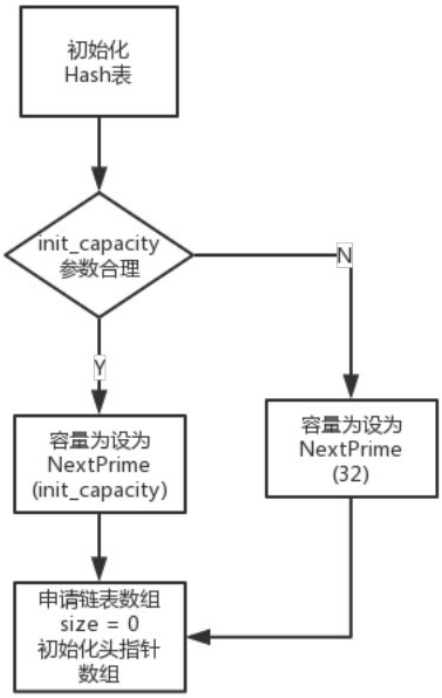

图 4 初始化链地址法 Hash 表流程图

```
void InitHashTable(HashTable &hash_table, int init_capacity) {
    if (init_capacity > 0 && init_capacity < MAXCAPACITY)
        hash_table.capacity = NextPrime(init_capacity);
    else
        hash_table.capacity = NextPrime(32);
    hash_table.table = new Node[hash_table.capacity];
    hash_table.size = 0;

    for (int i = 0; i < hash_table.capacity; i++) {
        hash_table.table[i].next = NULL;
    }
}
```

2) Hash 表扩容机制

当 Hash 表实际存储的元素个数即将等于 Hash 表容量*装填因子时，对 Hash 表进行扩容，使其容量变为当前容量的 2 倍，再取素数。流程图及代码如下图所示：

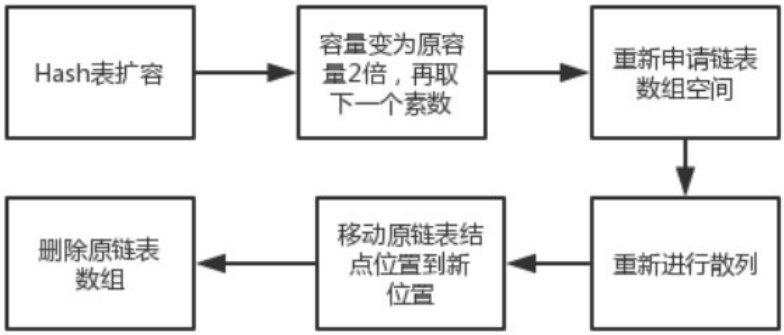

图 5 链地址法 Hash 表扩容机制流程图

```
void ReCapacity(HashTable &hash_table, unsigned int (*Hash)(const char
    *key, int table_size)) {
    const int ori_capacity = hash_table.capacity;
```

```
        hash_table.capacity = NextPrime(hash_table.capacity * 2);
        Node *ori_table = hash_table.table;
        hash_table.table = new Node[hash_table.capacity];
        for (int i = 0; i < hash_table.capacity; i++) {
            hash_table.table[i].next = NULL;
        }

        for (int i = 0; i < ori_capacity; i++) {
            Node *p = ori_table[i].next, *q;
            while (p) {
                q = p;
                p = p->next;
                int new_pos = Hash(q->key, hash_table.capacity);
                q->next = hash_table.table[new_pos].next;
                hash_table.table[new_pos].next = q;
            }
        }

        delete[] ori_table;
    }
```

3) Hash 表 Put 操作

向 Hash 表添加元素时，调用 Put 操作。若添加新元素后使得元素个数等于哈希表容量*装填因子，进行扩容。将以 key 为键的元素 value 放入 Hash 表，返回是否有旧元素，有返回 true，使用 value 进行替换，否则返回 false，old_value 为旧元素。流程图及代码如下图所示：

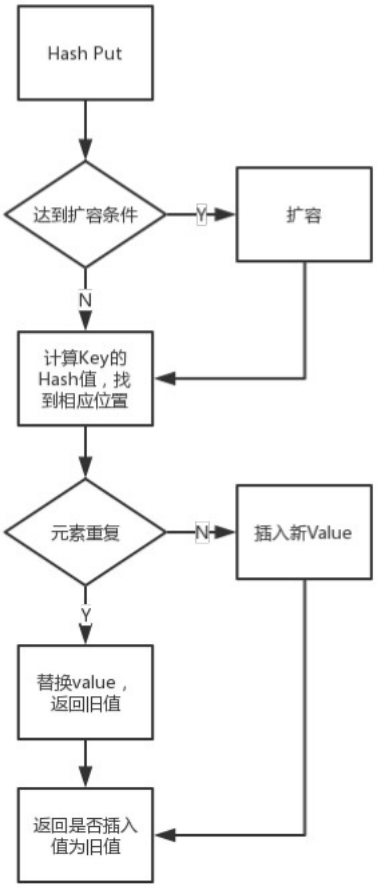

图 6 链地址法 Hash 表 Put 操作流程图

```
bool Put(HashTable &hash_table, const char *key, ElemType value, ElemType
    &old_value, unsigned int (*Hash)(const char *key, int table_size)) {
    if (hash_table.size + 1 >= hash_table.capacity * LOADFACTOR)
        ReCapacity(hash_table, Hash);

    unsigned int pos = Hash(key, hash_table.capacity);

    Node *p = hash_table.table[pos].next;
    while (p) {
        if (!strcmp(p->key, key)) {
            old_value = p->value;
            p->value = value;
```

```
            return true;
        }
        p = p->next;
    }

    Node *node = new Node;
    strcpy(node->key, key);
    node->value = value;
    node->next = hash_table.table[pos].next;
    hash_table.table[pos].next = node;
    hash_table.size++;

    return false;
}
```

4) Hash 表 Get 操作

获取 Hash 表为 key 的元素时，调用 Get 元素，返回是否存在该元素。流程图和代码如下：

图 7 链地址法 Hash 表 Get 操作流程图

```
bool Get(HashTable &hash_table, const char *key,    ElemType &value,
    unsigned int (*Hash)(const char *key, int table_size)) {
    int pos = Hash(key, hash_table.capacity);
```

```
    Node *p = hash_table.table[pos].next;
    while (p) {
        if (!strcmp(key, p->key)) {
            value = p->value;
            return true;
        }
        p = p->next;
    }

    return false;
}
```

5) Hash 表 Remove 操作

当删除 Hash 表中为 Key 的元素时，调用 Remove 函数。代码如下：

```
bool Remove(HashTable &hash_table, const char *key, ElemType &value,
    unsigned int (*Hash)(const char *key, int table_size)) {
    int pos = Hash(key, hash_table.capacity);

    Node *p = &hash_table.table[pos];
    while (p ->next) {
        if (!strcmp(p->next->key, key)) {
            value = p->next->value;
            Node *t = p->next;
            p->next = t->next;
            delete t;
            return true;
        }
        p = p->next;
    }
```

```
        return false;
    }
```

6) Hash 表销毁操作

不需要此 Hash 表时，需要释放空间，调用 DelHashTable 函数，代码如下：

```
    void DelHashTable(HashTable &hash_table) {
        for (int i = 0; i < hash_table.capacity; i++) {
            Node *p = hash_table.table[i].next;
            while (p) {
                Node *t = p;
                p = p->next;
                delete t;
            }
        }

        delete[] hash_table.table;
    }
```

7) Hash 表遍历操作

遍历 Hash 表时，调用 TraverseHashTable 函数，需指定访问函数 Visit，从而实现对 Hash 表内每个元素的访问。代码如下：

```
    void TraverseHashTable(HashTable &hash_table,
                           void (*visit)(ElemType v)) {
        for (int i = 0; i < hash_table.capacity; i++) {
            Node *p = hash_table.table[i].next;
            while (p) {
                visit(p->value);
                p = p->next;
```

```
            }
        }
    }
```

5.3 线性探测法 Hash 表设计

1) Hash 表初始化

线性探测法实现的 Hash 表使用数组存储数据，初始化需要申请数组空间，并为每一元素标记 Empty。代码如下：

```
void InitArrayHashTable(ArrayHashTable &hash_table, int init_capacity) {
    if (init_capacity > 0 && init_capacity < MAXCAPACITY)
        hash_table.capacity = NextPrime(init_capacity);
    else
        hash_table.capacity = NextPrime(32);
    hash_table.table = new HashNode[hash_table.capacity];
    hash_table.size = 0;

    for (int i = 0; i < hash_table.capacity; i++)
        hash_table.table[i].type = Empty;
}
```

2) Hash 表 Put 操作

Put 方法使用线性探测解决冲突，如果算出的 Hash 位置已经有元素或者被删除，则位置不断+1 寻找下一个 Empty 位置，代码如下：

```
//返回值：哈希表已满：-1    存在 key 相同元素，替换：1    新元素插入哈希表：0
int LinearPut(ArrayHashTable &hash_table, const char *key,
                    ArrElemType   value, ArrElemType &old_value) {
    if (IsFull(hash_table)) return -1;
    unsigned int curpos, newpos;
```

```
    curpos = newpos = ShiftHash(key, hash_table.capacity);

    while (!(hash_table.table[newpos].type == Empty
            || (!strcmp(hash_table.table[newpos].key, key) &&
            hash_table.table[newpos].type == Legitimate))) {
        newpos = (newpos + 1) % hash_table.capacity;
    }

    if (hash_table.table[newpos].type != Empty) {
        old_value = hash_table.table[newpos].value;
        hash_table.table[newpos].value = value;
        return 1;
    }

    strcpy(hash_table.table[newpos].key, key);
    hash_table.table[newpos].type = Legitimate;
    hash_table.table[newpos].value = value;
    hash_table.size++;
    return 0;
}
```

3) Hash 表 Get 操作

Get 方法通过算出的 Hash 位置不断寻找 Legitimate 元素，代码如下：

```
bool LinearGet(ArrayHashTable &hash_table, const char *key,
                                        ArrElemType &value) {
    unsigned int curpos, newpos;
    curpos = newpos = ShiftHash(key, hash_table.capacity);

    int cnt = 1;
    while (!(hash_table.table[newpos].type == Empty
```

```
        || (strcmp(hash_table.table[newpos].key, key) == 0    &&
        hash_table.table[newpos].type == Legitimate))) {
            newpos = (newpos + 1) % hash_table.capacity;
            if (++cnt == hash_table.capacity) return false;
        }

        if (hash_table.table[newpos].type == Empty)
            return false;

        value = hash_table.table[newpos].value;
        return true;
    }
```

4) Hash 表 Delete 操作

Delete 方法用于删除为 Key 的元素，添加 Deleted 标记，防止之后查找和插入时出现错误。代码如下：

```
    bool LinearDelete(ArrayHashTable &hash_table, const char *key) {
        unsigned int curpos, newpos;
        curpos = newpos = ShiftHash(key, hash_table.capacity);

        int cnt = 1;
        while (!(hash_table.table[newpos].type == Empty ||
                (strcmp(hash_table.table[newpos].key, key)    == 0 &&
                hash_table.table[newpos].type == Legitimate))) {
            newpos = (newpos + 1) % hash_table.capacity;
            if (++cnt == hash_table.capacity) return false;
        }

        if (hash_table.table[newpos].type == Empty)
            return false;
```

```
        hash_table.table[newpos].type = Deleted;

        return true;
    }
```

5.4 电话号码查询系统设计

系统首先选择建立索引方式，以电话号码为 Key 建立 Hash 表或以姓名为 Key 建立 Hash 表。包括以下函数，以下记录表示一条（电话号码、姓名、地址）的记录：

添加记录：

bool PutRecord(HashTable &hash_tabe, const char *key, AddList add, AddList &old_add, unsigned int (*Hash)(const char *key, int table_size) = ShiftHash)

获取记录：

bool GetRecord(HashTable &hash_table, const char *key, AddList &add, unsigned int (*Hash)(const char *key, int table_size) = ShiftHash)

移除记录：

bool RemoveRecord(HashTable &hash_table, const char *key, AddList &add, unsigned int (*Hash)(const char *key, int table_size) = ShiftHash)

输出记录：

void PrintRecord(AddList add)

添加记录界面：

void AddRecordForm()

查询记录界面：

void QueryRecordForm()

按电话号码查找记录：

void QueryRecordByPhoneNum()

按姓名查找记录：

void QueryRecordByName()

显示所有记录：

void DisplayAllRecord()

从文件导入记录：

void ImportRecordsFromFile()

随机生成记录：

void GenerateRandomRecords(int total_num, AddList Records[])

哈希函数对比：

void HashFuncCompare()

冲突解决策略对比：

void ConflictMethodCompare()

主界面：

void MainForm()

Hash 类型选择：

void ChooseHashType()

6 运行结果

以下为系统运行示例：

图 8 选择建立的 Hash 类型

图 9 系统主界面

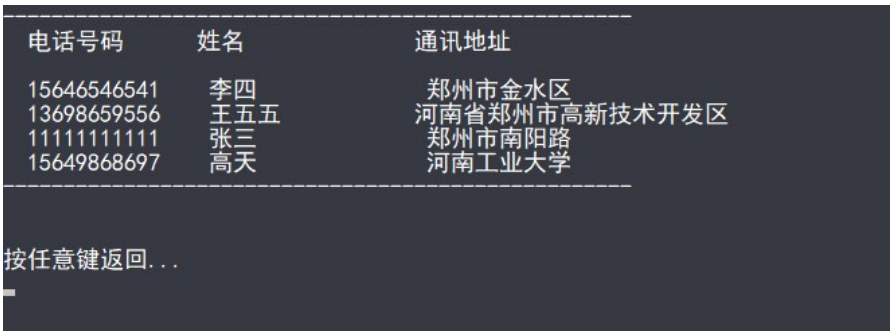

图 10 批量导入记录后查看全部记录

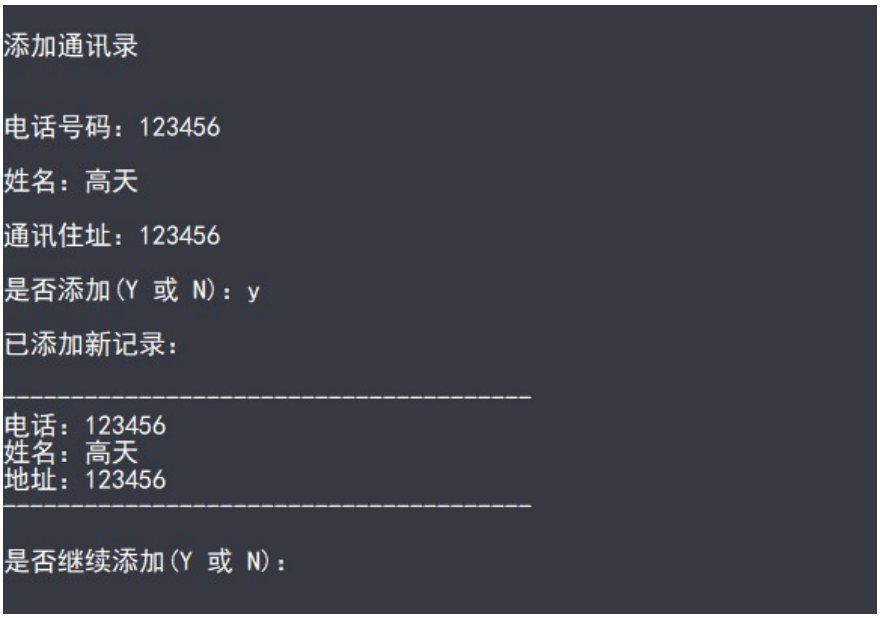

图 11 手动添加一条新纪录

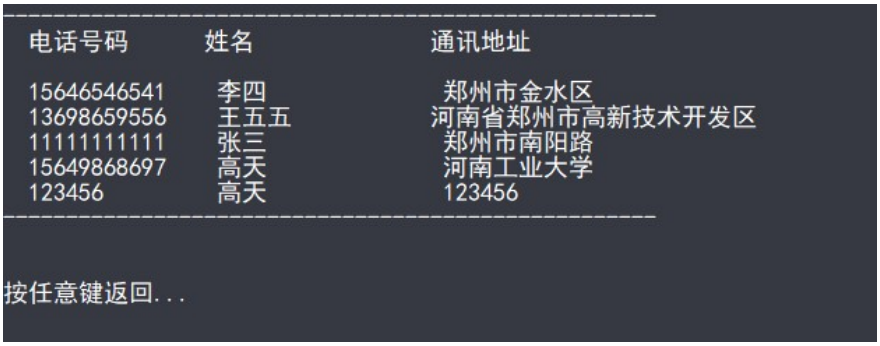

图 12 再次查看全部记录

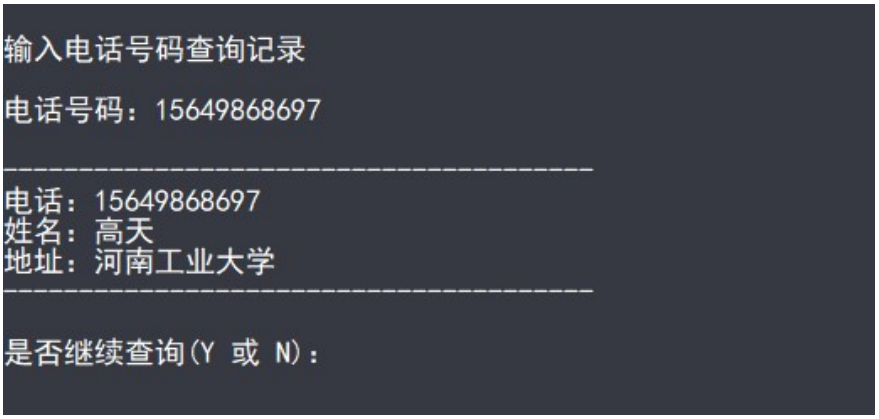

图 13 按电话号码查询记录

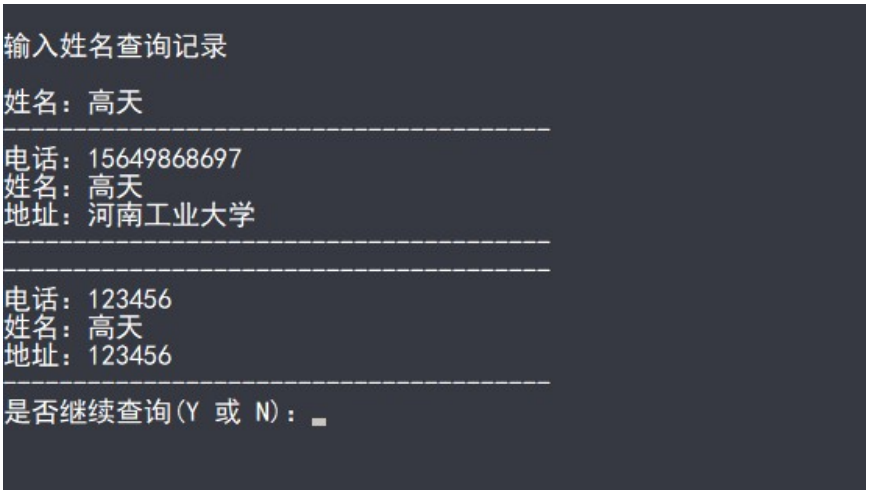

图 14 按姓名查询记录

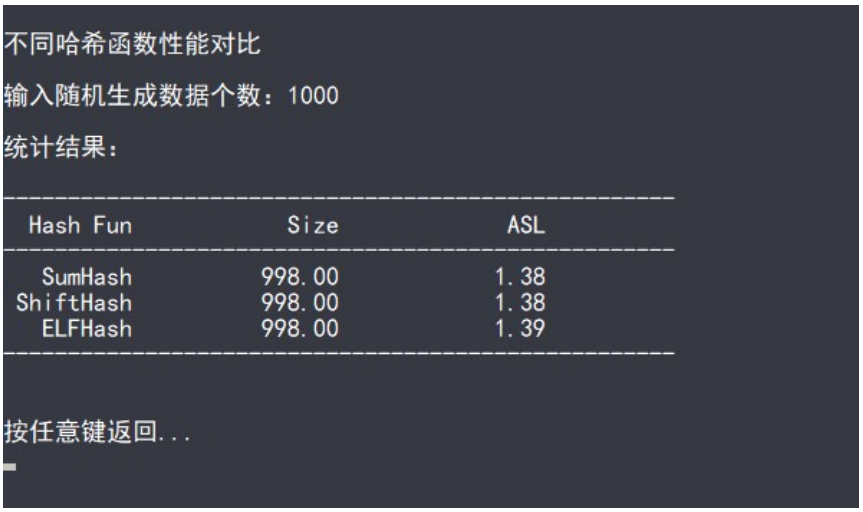

图 15 不同 Hash 函数比较 ASL

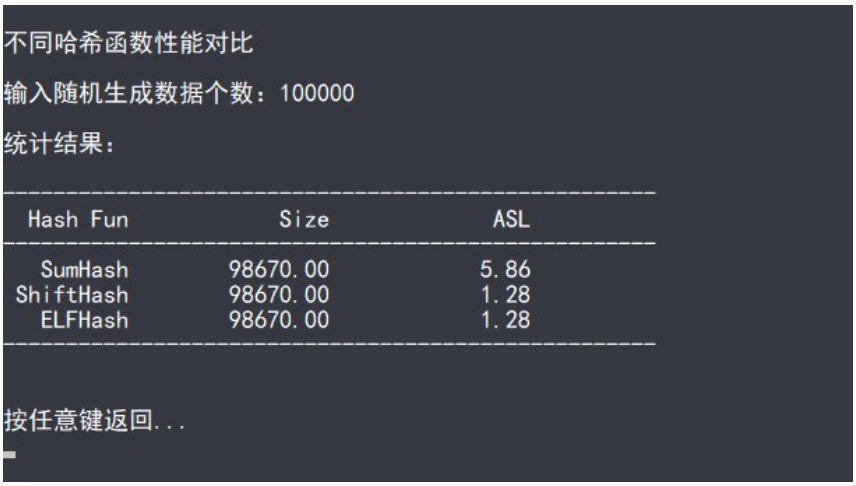

图 16 不同 Hash 函数比较 ASL（大数据）

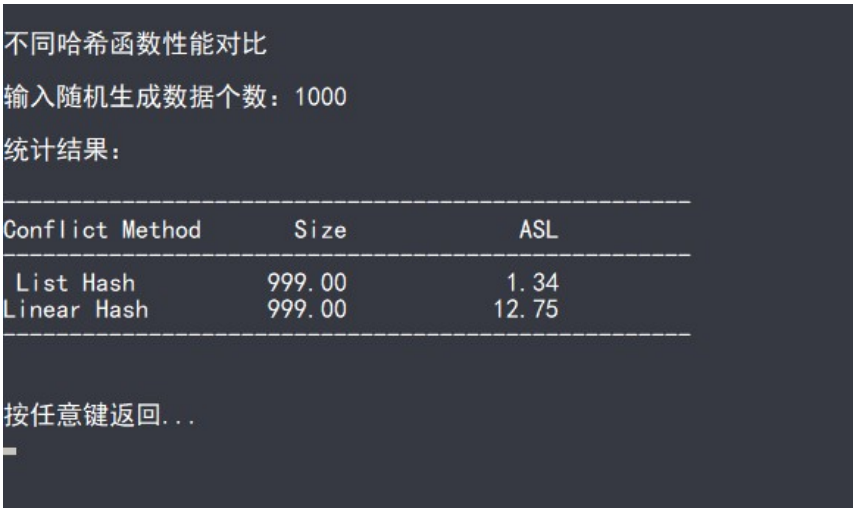

图 17 不同冲突解决策略比较 ASL

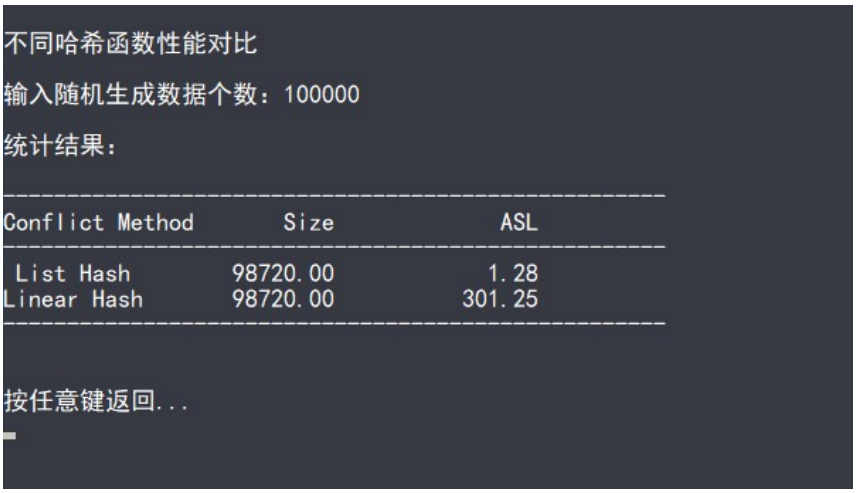

图 18 不同冲突解决策略比较 ASL（大数据）

7 调试分析

在测试大量数据的 Hash 函数比较时，Linux 操作系统会报 Segment Fault，如先随机生成数据测试，之后再次测试就会出现错误。经过仔细分析，发现链表操作存在问题，在进行扩容操作时，对于新创建的链表数组忘记将 next 初始化为 NULL。可见对于链表的相关操作需要格外细心，指针申请使用完应释放。下图 78-80 行为忘记初始化的代码：

```
72  void ReCapacity(HashTable &hash_table, unsigned int (*Hash)(const char *key, int table_size)) {
73      const int ori_capacity = hash_table.capacity;
74
75      hash_table.capacity = NextPrime(hash_table.capacity * 2);
76      Node *ori_table = hash_table.table;
77      hash_table.table = new Node[hash_table.capacity];
78      for (int i = 0; i < hash_table.capacity; i++) {
79          hash_table.table[i].next = NULL;
80      }
81
```

图 19 出现问题的代码段（78-80 行）

8 心得体会

本次课程设计我选择了 Hash 表的设计与实现这个题目，因为在 C++、Java 等语言中经常使用这种数据结构，故对于 Hash 表有一定了解，而且 Hash 的思想用处极为广泛，如数据库、数据加密等，所以希望自己可以实现其基本结构和操作，并分别以链表和数组的方式来实现。

编写 Hash 算法的过程并不很容易，我参考了 JDK 中 HashMap 中的源码实现，了解到其基本的设计思想，并将其应用到自己的 Hash 算法中。在自己的链表实现 Hash 中，使用了 HashMap 所使用的链表来解决冲突的方法，即相同 Hash 值的元素被放在同一个链表中，查找时转变为线性查找；同时应用了 HashMap 中的扩容思想，当 Hash 表中装填因子达到某一阈值时，对 Hash 表进行自动扩容，此方式可增强 Hash 表的易用度和性能。在编写完成进行大容量数据测试时，发现了链表式 Hash 的 Bug，经过仔细查看代码，发现是在扩容时忘记将新申请的空间的每个 next 域初始化，以后在编写链表的程序时要格外细心。

对于数组的 Hash 算法，最开始以为就是简单的数组查找或添加，找不到直接线性探测就可以了，但是当删除元素时，如果简单的进行删除操作就会出现问题，下次操作可能无法找到正确的位置，因为之后插入元素的位置依赖与之前插入元素的位置。所以对于数组的 Hash 算法需要涉及标记，用于标记某个元素是否有效，是否被删除，某个位置是否为空。加入标记后，对 Hash 表的 Get、Put、Delete 操作也变得复杂起来。

实现过 Hash 表之后，开始实现电话号码查询系统，由于主要是查询操作，所以使用 Hash 表作为数据结构非常合适。在实现过 Hash 表之后，以此为基本数据结构再实现系统比较容易。

在比较模块，我对均使用链表式 Hash 但采用不同 Hash 函数，以及均使用移位 Hash 但一个采用链表式 Hash，另一个采用数组线性探测的 Hash 进行了比较测试，通过 ASL 发现，对于自动扩容的链表式 Hash 优势较为明显，当然如果要获得较好的散列性能，需要选择合适的 Hash 函数。

总之，通过这次的数据结构课程设计，让我对数据结构有了更进一步的理解，自己编写算法，不仅收获了知识，而且收获了编写算法程序的快乐。数据结构的学习应该是一个长期实践的过程，之后我要继续努力学习和实践。

9 参考文献

[1] 严蔚敏, 吴伟民. 数据结构（C 语言版）[M]. 北京:清华大学出版社, 2012.

[2] 陈越, 何钦铭等. 数据结构（第 2 版）[M]. 北京:高等教育出版社, 2016.

[3] Thomas H.Cormen 等. Introduction to Algorithms, third edition [M]. 北京:机械工业出版社, 2012.

[4] https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html [OL].