# 北京邮电大学软件学院

# 2020-2021学年第1学期实验报告

**课程名称： 算法与数据结构**

**实验名称： 查找**

**实验完成人：**

**姓名：**\_\_赵冉\_\_\_\_\_\_**学号：**\_\_2020211731\_\_\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_贾红娓\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2021 年 12 月 16 日**

1. **实验目的**

本次实验旨在集中对几个专门的问题作较为深入的探讨和理解，不强调对某些特定的编程技术的训练。

1. **实验内容**

**2.1 选做内容**

1. **二叉排序树**

**[问题描述]**

　　从键盘读入一组数据，建立二叉排序树并对其进行查找、遍历、格式化打印等有关操作。

**[基本要求]**

　　建立二叉排序树并对其进行查找，包括成功和不成功两种情况，并给出查找长度。

**[测试数据]**

由学生依据软件工程的测试技术自己确定。注意测试边界数据。

1. **哈希表设计**

**[问题描述]**

　　针对某个集体中人名设计一个哈希表，使得平均查找长度不超过R，并完成相应的建表和查表程序。

**[基本要求]**

　　假设人名为中国人姓名的汉语拼音形式。待填入哈希表的人名共有30个，取平均查找长度的上限为2。哈希函数用除留余数法构造，用线性探测再散列法或链地址法处理冲突。

**[测试数据]**

　　取你周围较熟悉的30个人名。

**2.2 选做内容**

* + - * 1. 实现二叉排序树的插入、删除操作。
        2. 从教科书上介绍的集中哈希函数构造方法中选出适用者并设计几个不同的哈希函数，比较他们的地址冲突率（可以用更大的名字集合作实验）。
        3. 研究必做实验2）的30个人名的特点，努力找一个哈希函数，使得对于不同的拼音名一定不发生地址冲突。
        4. 在哈希函数确定的前提下尝试各种不同处理冲突的方法，考察平均查找长度的变化和造好的哈希表中关键字的聚集性。

1. **实验环境**

VC6.0

1. **实验要求**

**1）**编程实现必做实验内容1）、2）。

* + - 1. 编程实现必做实验内容1）、2）。
      2. 学生可根据自己的能力完成选做内容。

**五、实验过程和实验结果**

**1.二叉排序树**

**(1)问题分析**

二叉排序树是具有若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；它的左、右子树也分别为二叉排序树的性质的树。

**（2）设计方案**

根据题目要求，可以发现实验希望构造一颗二叉排序树，接着完成基本的查找操作和显示查找长度的功能。为了实现这两个功能，分别设计一个Create（）函数和Search（）函数，在前者中完成二叉排序树的构建，在后者中完成特定数据的查找。

同时，在选作内容中，有关于二叉排序树的插入函数要求和删除函数要求。于是又额外设计了Insert（）函数和Delete（）函数，插入操作相对简单，只需要新建一个结点与原本的根结点进行大小比较，从而插入即可；删除函数要稍微复杂一些，需要考虑到被删结点的属性，如果是叶结点，直接删除即可；如果是只有一个子树非空，就有考虑非空子树的重接问题；如果是两个子树都非空，就要作更复杂的进一步操作，完成左右子树的重接。

1. **算法（实现代码）**

#include <iostream>

#define MAXINT 50

#define FALSE 0

#define TRUE 1

using namespace std ;

struct Node{

int num ;

Node \*chl, \*chr ;

};

typedef Node\* BSTree ;

typedef Node node ;

typedef int Status;

**//二叉排序树的插入**

void Insert(int temp, BSTree &n) {

if (n == NULL) {

n = new node() ;

n->chl = n->chr = NULL ;

n->num = temp ;

// cout << n->num << " " ;

}

else if (temp < n->num) {

Insert(temp, n->chl) ;

}

else if (temp > n->num) {

Insert(temp, n->chr) ;

}

}

**//二叉排序树的删除**

int Delete(BSTree &p)

{

**// 从二叉排序树中删除节点p， 并重接它的左或右子树**

BSTree q, s;

if( !p->chl && !p->chr ) **// p为叶子节点**

p = NULL;

else if( !p->chl ) **// 左子树为空，重接右子树**

{

q = p;

p = p->chr;

delete(q);

}

else if( !p->chr ) **// 右子树为空，重接左子树**

{

q = p;

p = p->chl;

delete(q);

}

else  **// 左右子树均不为空**

{

q = p;

s = p->chl;

while(s->chr)  **// 转左，然后向右走到尽头**

{

q = s;

s = s->chr;

}

p->num = s->num ;

if( q != p )

q->chr = s->chl;

else

q->chl= s->chl;

delete(s);

}

return TRUE;

}

Status DeleteBSTree( int key, BSTree &t)

{

**/\* 若二叉排序树T中存在关键字等于key的数据元素时，则删除该数据元素节点**

**\* 并返回TRUE；否则返回FALSE**

**\*/**

if (!t)

return FALSE; **// 不存在关键字等于key的数据元素**

else{

if(key == t->num )

Delete(t);

else if(key < t->num)

return DeleteBSTree(key , t->chl);

else if(key > t->num)

return DeleteBSTree(key, t->chr);

}

}

**//通过中序遍历查看排序二叉树**

void ShowBSTree(BSTree &t) {

if (t != NULL) {

ShowBSTree(t->chl) ;

cout << t->num << " " ;

ShowBSTree(t->chr) ;

}

}

**//在排序二叉树中查找并返回查找到的结点**

BSTree Search(int key, BSTree &t) {

BSTree temp = t ; **//创建临时变量代替原二叉树进行遍历**

int depth = 0 ;

while(temp) {

if(key == temp->num) {

depth ++ ; **//深度+1**

cout << **"查找成功，查找长度为："** << depth << endl ;

return temp;

}

else if(key < temp->num) {

depth ++ ;

temp = temp->chl ;

}

else if(key > temp->num) {

depth++ ;

temp = temp->chr ;

}

}

cout << **"查找失败，排序二叉树中不含此元素。"** << endl ;

cout << endl ;

}

int main() {

int n ;

cout << **"请输入一共有多少个结点: "** ;

cin >> n ;

cout << endl ;

**//创建存储结点值的数组**

int ver[MAXINT] = {0} ;

**//创建二叉排序树并初始化**

BSTree t ;

t = new node() ;

t = NULL ;

**//插入元素**

cout << "**请依次键入数值:**" << endl ;

for (int i = 0 ; i < n ; i ++) {

cin >> ver[i] ;

Insert(ver[i], t) ;

}

cout << endl ;

**//排序二叉树遍历**

cout << "**中序遍历结果为:**" << endl ;

ShowBSTree(t) ;

cout << endl ;

**/\*进行遍历二叉树的查找**

**\*示例使用16和26两个数值**

**\*分别输出结果**

**\*/**

cout << endl ;

cout << "**查找示例:**" << endl ;

cout << "key = 16" << endl ;

Search(16, t) ;

cout << "key = 26" << endl ;

Search(26, t) ;

**/\*进行遍历二叉树的插入**

**\*示例使用23**

**\*分别输出结果**

**\*/**

Insert(23, t) ;

cout << "插入示例:" << endl ;

cout << "key = 23" << endl ;

ShowBSTree(t) ;

cout << endl ;

**/\*进行遍历二叉树的删除**

**\*示例使用9，7**

**\*分别输出结果**

**\*/**

cout << endl ;

cout << "**删除示例**" << endl ;

cout << "key = 9" << endl ;

DeleteBSTree(9, t) ;

ShowBSTree(t) ;

cout << endl ;

cout << "key = 11" << endl ;

DeleteBSTree(11,t) ;

ShowBSTree(t) ;

return 0 ;

}

**（4）设计图**

算法设计图如下：



图1.1设计图

1. **程序**

下述为程序截图：

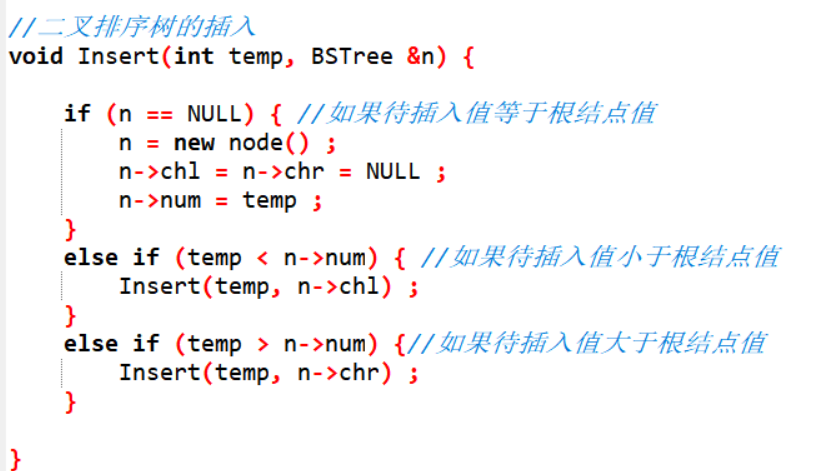


图1.2 插入函数

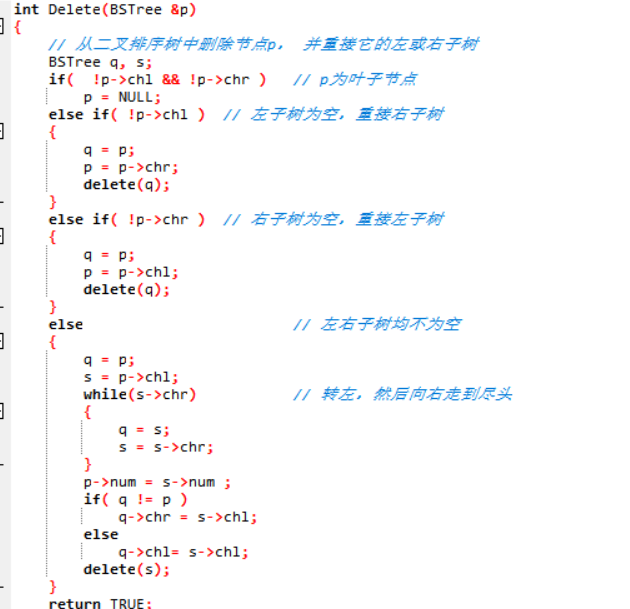


图1.2 删除函数（1）

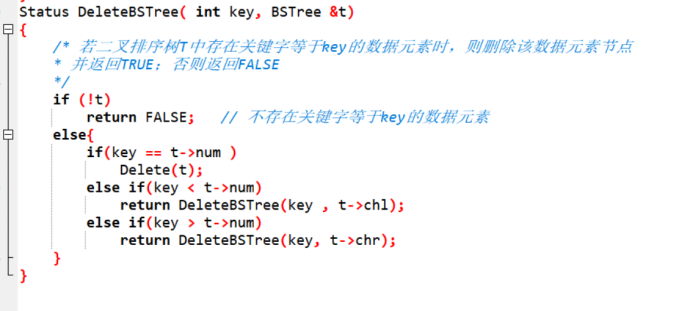


图1.3 删除函数（2）

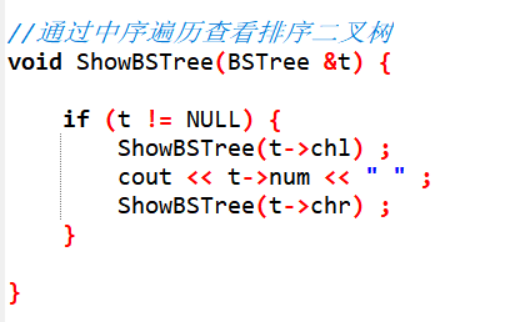


图1.4 打印函数



图1.5 查找函数

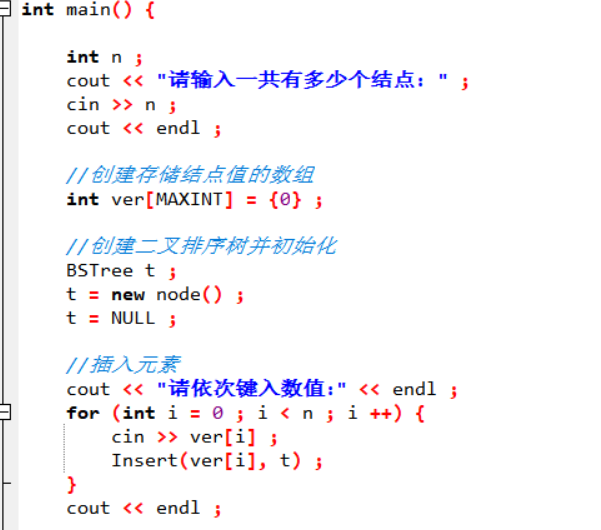


图1.6 主函数（1）

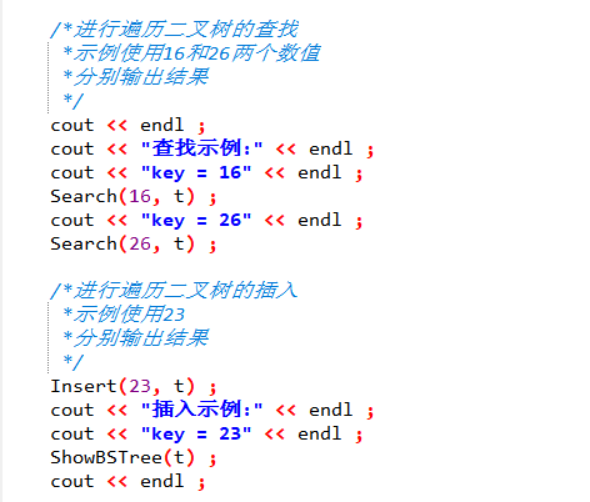


图1.7 主函数（2）

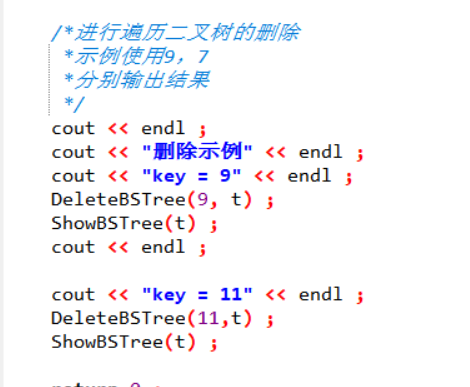


图1.8 主函数（3）

1. **调试过程截图**

调试过程中遇到的问题截图，在主函数中编写代码时，漏写分号。

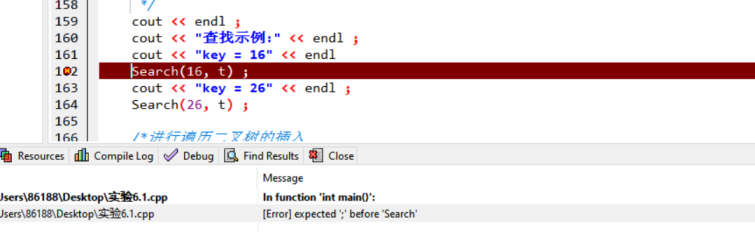


图1.9代码调试截图

1. **运行结果截图**

构建二叉排序树后中序输出。

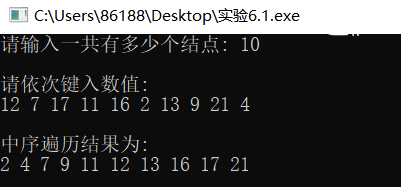


图1.10

查找时预设两个用例，16和26，其中16为结点值，可以查询得到，并且输出其查找长度，而26在结点中不存在，所以显示查找失败。

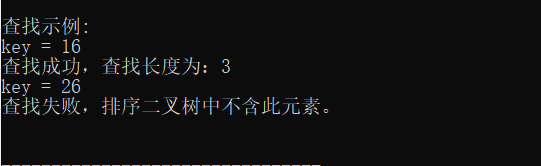


图1.11

删除二叉排序树中的结点，这里选用叶结点9和内部节点11，进行两次的删除操作，同时输出最后的二叉排序树的遍历结果。

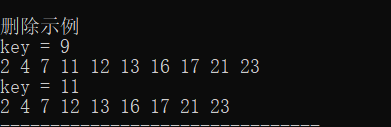


图1.12

插入操作选择在原本的值上插入一个23数值，进行验证，插入后遍历结果输出如下。

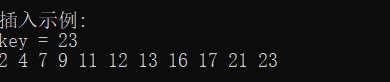


图1.13

**2.哈希表设计**

**(1)问题分析**

实验要求我们创建一个可以查找30个姓名的哈希表。

哈希表是一种建立在哈希函数思想上的表，所谓哈希函数思想，是指在理想情况下，查找操作不希望经过任何比较，一次存取便能得到所查记录，那就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系，使每个关键字和结构中唯一的存储位置相对应。因而在查找时，只要根据这个对应关系f找到给定值K的像f（K），即可得到所需。

不难看出，建立一个哈希表的核心，便是找到这个对应关系，即哈希函数。所以实验的核心问题，其一便是如何将30和字符串姓名与哈希值对应从而得到哈希函数，其二则是解决建立哈希函数后可能会遇到的数据冲突问题。

**（2）设计方案**

在设计哈希函数时，根据题目要求，选用除留余数法。我们将key值拟定为每一个字符串的每个字符的ASCII值相加，然后除去同一个除数，保留其余数作为新的key值。除数的选择，则依赖于哈希表的既定长度。实验建立一个长度为40的哈希表，除数选择最接近小于40的值39，最后验证实验时要保证其ASL值小于2。数据冲突问题，则采用线性探测方法解决。

在完成哈希表的建立后，将其输出打印，接着又完成Search()函数,以满足对哈希表的最基础的访问查找操作。

**（3）算法（实现代码）**

#include <iostream>

#include <string>

#define MAXHASH 40

#define NUM 30 **//哈希表记录30人的姓名**

#define M 39  **//除数**

#define MAXINT 100 **//数据冲突时使用的最大数值**

using namespace std ;

struct HashTable{

int key ;

int search ; **//查找长度**

string name ;

};

int GetKey(string s) {

int num = 0 ;

for (int i = 0 ; i < s.length() ; i ++) {

num += (int)s[i] ;

}

return num ;

}

void Create(HashTable ht[], string name[]) {

**//哈希表初始化**

for (int i = 0 ; i < MAXHASH ; i++) {

ht[i].key = -1 ;

ht[i].search = -1 ;

ht[i].name = "NULL" ;

}

for (int j = 0 ; j < NUM ; j ++) { **//确保哈希表的40个位置都有被遍历到的可能**

**/\*新建一个index，是元素在哈希表中的位置**

**\*index即为每个元素的key值**

**\*哈希函数用除留余数法构造 H（key） = key % 39**

**\*因为哈希表长40，mod数取39**

**\*/**

int index ;**//得到index值**

index = GetKey(name[j])% M ; **//将字符串每个字符的值相加得key**

//cout << j <<" " <<name[j] << " " <<index << endl ;

if(ht[index].key == -1) { **//如果此Index位置上没有其它的数，则赋值**

ht[index].name = name[j] ;

ht[index].key = index ;

ht[index].search = 1 ;

}

else { **//如果发生数据冲突，利用线性探测法**

for (int k = 1 ; k < MAXINT ; k ++) {

int index2 = (index + j) % M ; **//在Index的基础上进一步求得新的Index2**

int search = 1 ;

search ++ ;

if(ht[index2].key == -1) { **//如果Index2上没有其它值**

ht[index2].name = name[j] ;

ht[index2].key = index2 ;

ht[index2].search = search ;

break ;

}

}

}

}

}

void ShowHashTablel(HashTable ht[]) {

cout << "\t\t\t" << "[序号]" << "\t" << "[哈希值]"<< "\t" << "[人名]" << "\t\t" << "[搜索长度]" << endl ;

for (int i = 0 ; i < MAXHASH ; i ++) {

cout << "\t\t\t" << i << "\t" <<ht[i].key<< "\t\t" << ht[i].name << "\t\t" << ht[i].search << endl ;

}

cout << endl ;

}

float GetASL(HashTable ht[]) {

int sum = 0 ;

for(int i = 0 ; i < MAXHASH ; i++) {

if (ht[i].key != -1) {

sum += ht[i].search ;

}

}

return (sum / (float)30) ;

}

int Search(string s, HashTable ht[]) {

**//得到待查找值的index**

int keyIndex = GetKey(s) % M;

if(ht[keyIndex].name == s) {

return keyIndex ; **//找到值**

}

else { **//如果发生数据冲突，进入线性探测**

for (int j = 1 ; j < MAXINT ; j++) {

**//得到新的哈希值**

int keyIndex2 = (keyIndex + j) % M ;

if(ht[keyIndex2].name == s) {

return keyIndex2 ;  **//如果找到，返回**

}

else if(ht[keyIndex2].name == "NULL"){

break ;

}

}

}

return -1 ;

}

int main() {

string name[NUM] = {"xvzhenhua", "xinzebin",

"caoshenjie", "wuhaobin",

"yinjiacai", "muxin",

"guozihan", "guanshanyue",

"chensijia", "zhaoran",

"chenbin", "wudi",

"liuxisheng", "masifan",

"laihongbin", "liulichao",

"chenqilin", "chengjiawei",

"fanyvqing", "huangjunhui",

"huangzaoying", "liyifei",

"liuxinrui", "luoxiaoyv",

"wangxiaoke", "wenbo",

"houjinbing", "yangguang",

"zhangyvqian", "zhangzijian"} ;

**//新建一个哈希表**

HashTable ht[MAXHASH];

Create(ht, name) ;

**//打印出哈希表**

cout << endl ;

cout << "\t\t\t\t\t\t" << "**哈希表**" << endl ;

cout << endl ;

ShowHashTablel(ht) ;

**//求取此哈希表的ASL**

cout << "此表的ASL = "<< GetASL(ht) << endl ;

cout << endl ;

**//查找哈希表实例**

cout << "**查找哈希表示例:**" << endl

<< "**[如返回值为-1，说明表中无此元素]**" << endl ;

cout << "key = zhaoran" << endl ;

cout << "**其哈希值为：**" << Search("zhaoran", ht) << endl;

cout << "key = waixingren" << endl ;

cout << "**其哈希值为：**" << Search("waixingren", ht) ;

return 0 ;

}

**（4）设计图**

算法设计图如下：

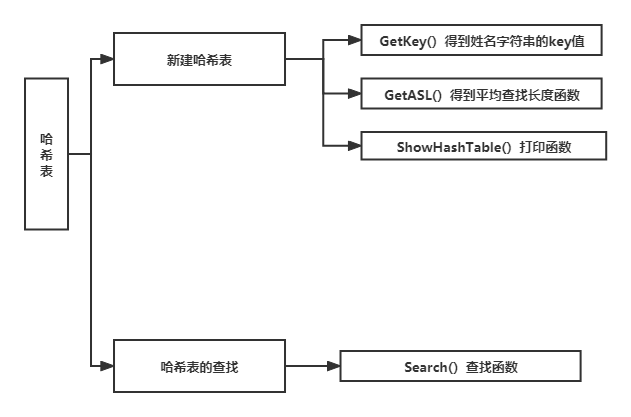


图2.1设计图

**（5）程序**

下述为程序截图：

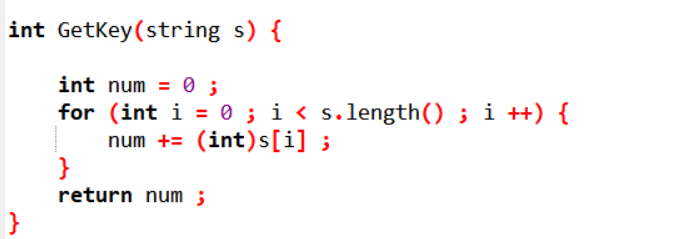


图2.2 GetKey()函数

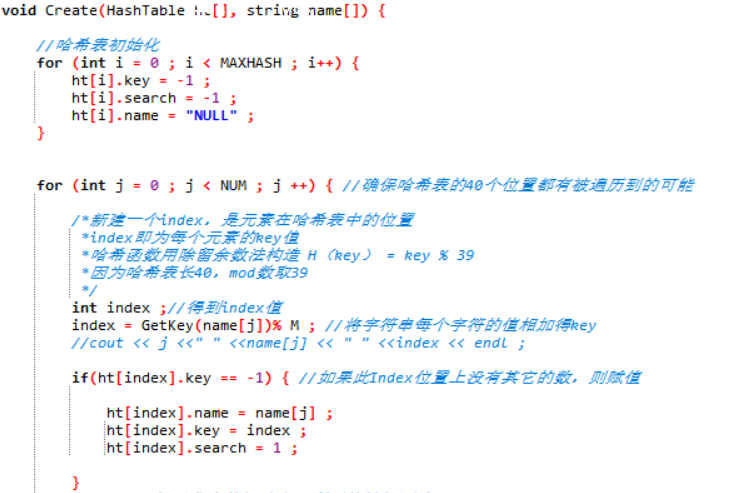


图2.3 Create()函数

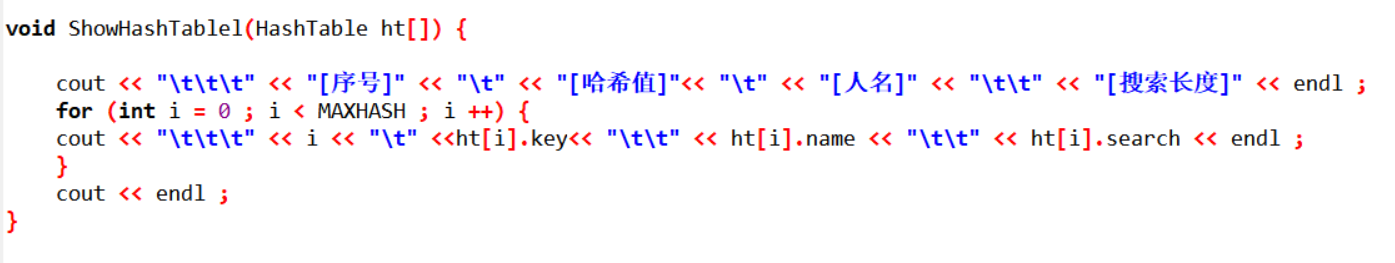


图2.4 打印函数

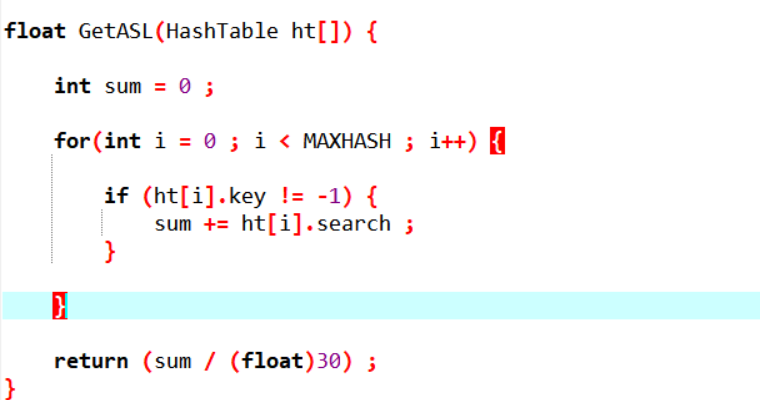


图2.5得到ASL的函数

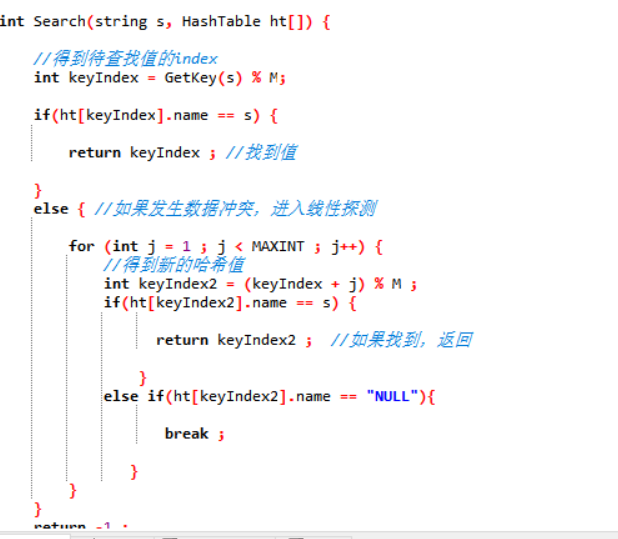


图2.6查找函数

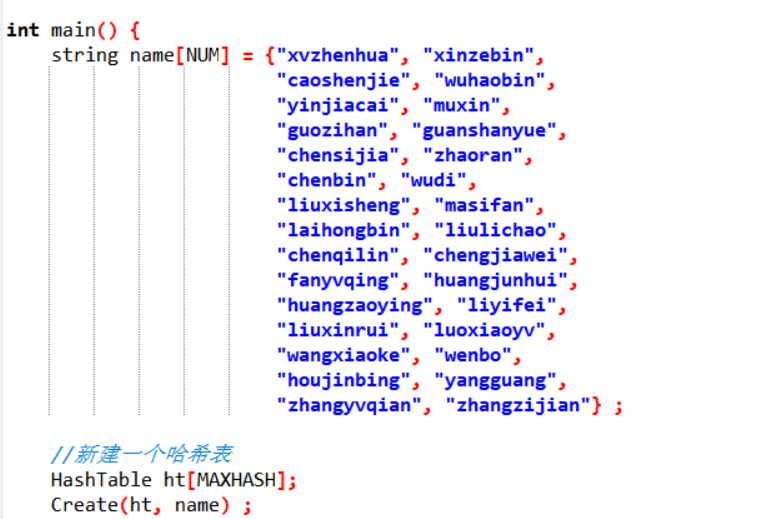


图2.7主函数（1）



图2.8主函数（2）

**（6）调试过程截图**

在第一次调试程序时，发现结果不能正确输出，没有办法查找到确定的字符串。

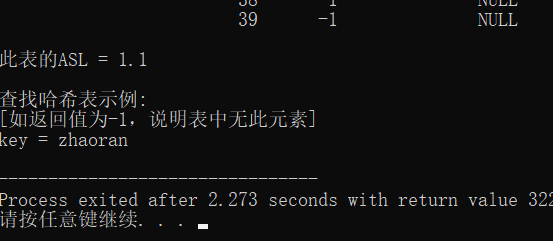


图2.9 代码调试截图（1）

经过对代码的检查后，发现是因为在主函数中，计算给定的值的index值时没有给key值除余数M，导致计算得出的哈希值远远超出哈希表的搜查大小，从而出错。

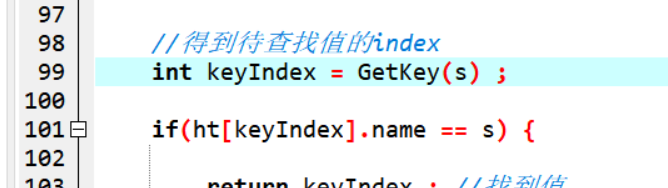


图2.10 代码调试截图（2）

更正后的结果如下，输出如图：

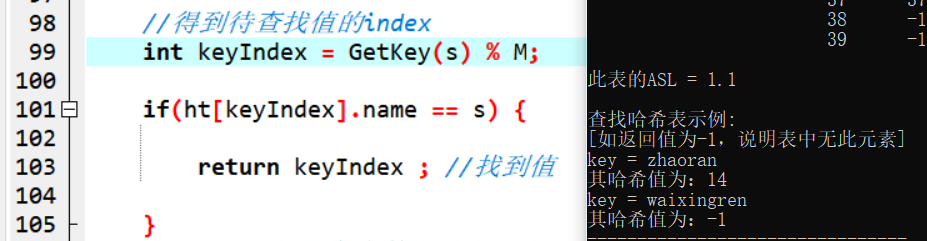


图2.11 代码调试截图（2）

**（7）运行结果截图**

哈希表的输出：

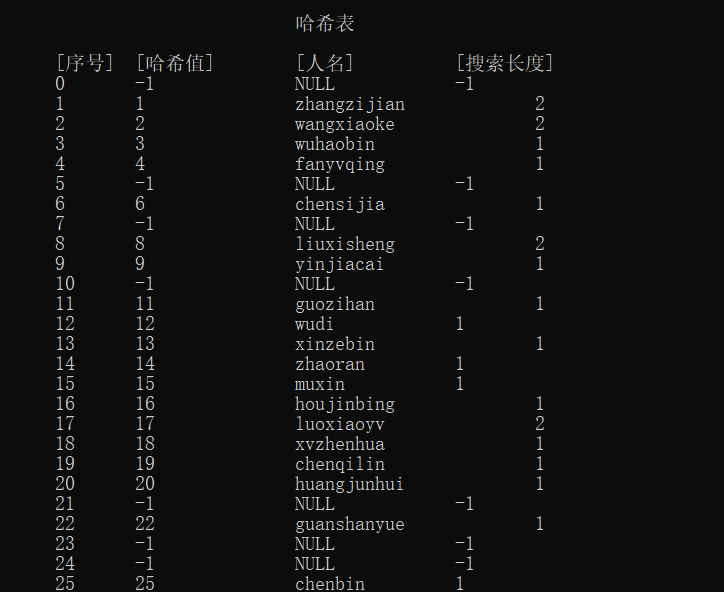


图2.12 哈希表（1）

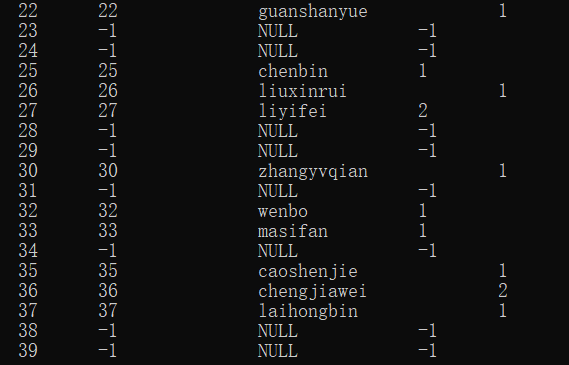


图2.13 哈希表（2）

C:\Users\86188\Desktop\7.3.png7.3

图2.14 ASL值

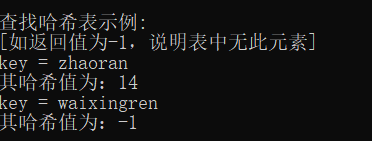


图2.15 哈希表查找结果

**六、实验心得**

此次实验重点考察对于查找表的理解。查找表是由同一类型的数据元素（或记录）构成的集合。由于“集合”中的数据元素之间存在着完全松散的关系，因此查找表是一种非常灵便的数据结构，其在日常生活中，也受到广泛的应用。

通过这次实验，我掌握了二叉排序树和哈希表这两种查找结构的建立和查找的算法。二叉排序树是一种动态树表，其特点是：树的结构通常不是一次生成的，而是在查找过程中，当树中不存在[关键字](https://baike.baidu.com/item/%E5%85%B3%E9%94%AE%E5%AD%97" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)等于给定值的[结点](https://baike.baidu.com/item/%E7%BB%93%E7%82%B9" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)时再进行插入。新插入的结点一定是一个新添加的[叶子结点](https://baike.baidu.com/item/%E5%8F%B6%E5%AD%90%E7%BB%93%E7%82%B9" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。其插入算法是首先执行查找算法，找出被插结点的父亲结点。判断被插结点是其父亲结点的左、右儿子。将被插结点作为[叶子结点](https://baike.baidu.com/item/%E5%8F%B6%E5%AD%90%E7%BB%93%E7%82%B9" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)插入。若[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)为空。则首先单独生成根结点。

哈希表是根据key而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做哈希函数（散列函数），存放记录的数组称做哈希表（散列表）。通过这次实验，我对哈希表的本质有了更加深入的理解。

这次实验我不仅对二叉排序树和哈希表有了更深的理解，同时还学习了二者的建立和查找功能的算法，同时也对查找表是实际应用中大量使用的特点有了更加深刻的体会。