**《算法与数据结构》**

**实 验 指 导 手 册**

# 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： 实验六 查找**

**实验完成人：**

**姓名：**\_\_\_王宇涵\_\_\_\_\_**学号：**\_\_\_\_2020211730\_\_\_\_**成绩：**\_\_\_\_\_\_\_\_

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

**日 期： 2021 年 12 月 12 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**

**必做内容**

**必做内容**

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。

**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. **实验过程和实验结果**

**1.二叉排序树**

**1.1问题分析**

1）二叉排序树要么是空[二叉树](http://c.biancheng.net/view/3384.html" \t "http://c.biancheng.net/view/_blank)，要么具有如下特点：

1.二叉排序树中，如果其根结点有左子树，那么左子树上所有结点的值都小于根结点的值；

2.二叉排序树中，如果其根结点有右子树，那么右子树上所有结点的值都大小根结点的值；

3.二叉排序树的左右子树也要求都是二叉排序树；

2）二叉排序树中查找某关键字时首先将被查找值同树的根结点进行比较，会有 3 种不同的结果：

1. 相等成功；
2. 根结点的关键字值较大，则说明该关键字可能存在其左子树中；
3. 根结点的关键字值较小，则说明该关键字可能存在其右子树中；

**1.2设计方案**

建立二叉排序树。按照用户需要的二叉排序树，构建二叉排序树，按照树形结构打印输出二叉排序树，对二叉排序树进行遍历操作，对二叉排序树进行查找，包括成功和不成功两种情况，并给出查找长度，对二叉排序树进行插入操作。

1）创建树的结点,队列节点

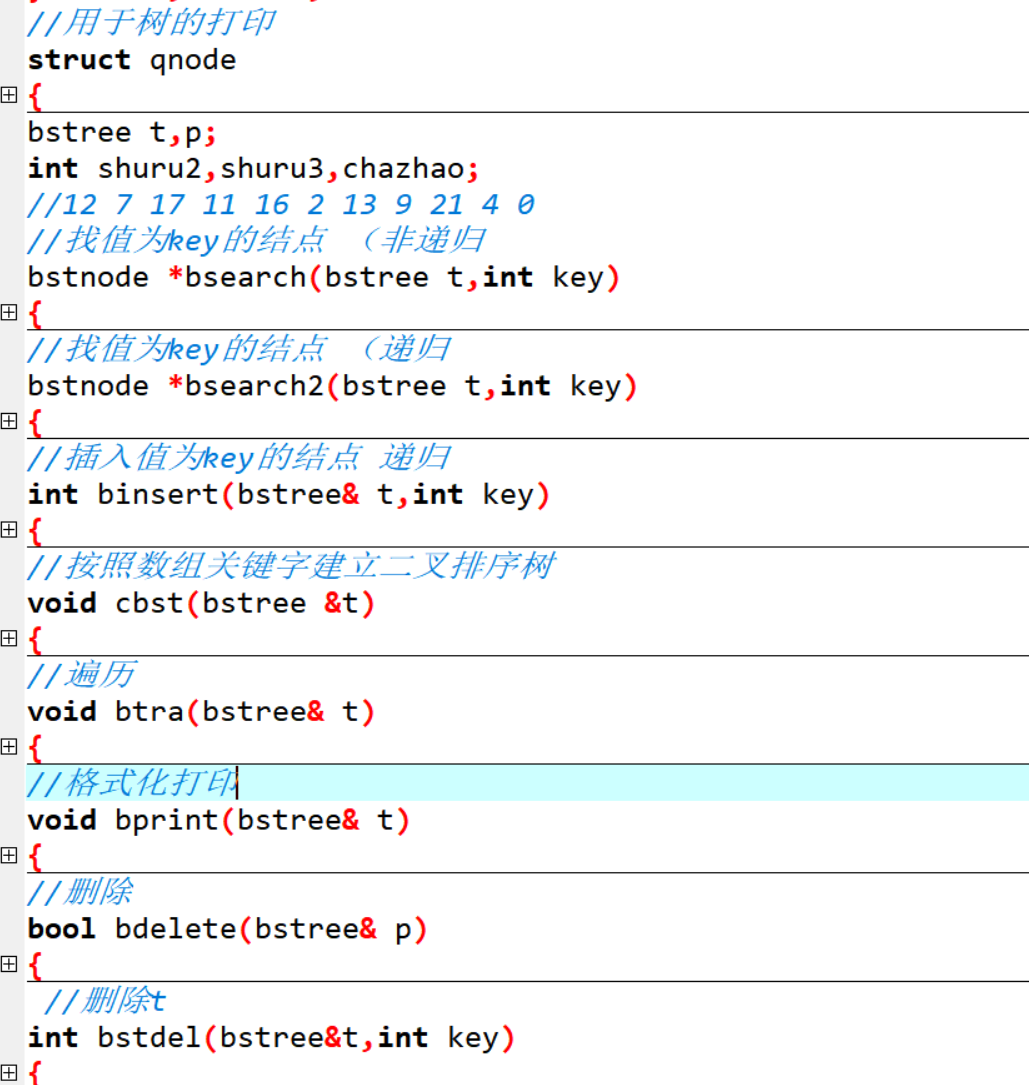
2）创建查找函数bsearch（）

3）创建插入函数binsert（）用于创建

4）创建二叉排序树sbst（）

5）遍历btra（）

6）格式化打印bprint()

****

**1.3算法**

1. 创建树的结点，队列节点

//二叉排序树结点

typedef struct bstnode

{

int key;//结点数据元素

struct bstnode \*lchild,\*rchild;//左右孩子指针

}bstnode,\*bstree;

//用于树的打印

struct qnode

{

bstree data[MAXSIZE];//存放树节点

int deep[MAXSIZE];//结点所在层

int locate[MAXSIZE];//打印节点位置

int front,rear;//头指针，尾指针

}q;

1. 创建查找函数bsearch（）

//找值为key的结点 （非递归

bstnode \*bsearch(bstree t,int key)

{

while(t!=NULL&&key!=t->key)

{

if(key<t->key)//<在左子树找

{

t=t->lchild;

}

else//>在右子树找

{

t=t->rchild;

}

chazhao++;

}

//失败返回NULL

return t;

}

//找值为key的结点 （递归

bstnode \*bsearch2(bstree t,int key)

{

if(t==NULL)//失败返回NULL

return NULL;

if(key==t->key)//成功

return t;

else if(key<t->key)//<在左子树找

{

return bsearch(t->lchild,key);

}

else//>在右子树找

{

return bsearch(t->rchild,key);

}

return t;

}

1. 创建插入函数binsert（）

//插入值为key的结点 递归

int binsert(bstree& t,int key)

{

if(t==NULL)//失败

{

t=(bstree)malloc(sizeof(bstnode));

t->key=key;

t->lchild=t->rchild=NULL;

return 1; //插入成功

}

if(key==t->key)//存在相同，插入失败

return 0;

else if(key<t->key)//<在左子树找

{

binsert(t->lchild,key);

}

else//>在右子树找

{

binsert(t->rchild,key);

}

}

1. 创建二叉排序树sbst（）

//按照数组关键字建立二叉排序树

void cbst(bstree &t)

{

t=NULL;//初始t空

int shuru;

cin>>shuru;

while(shuru!=0)//依次将每个关键字插入二叉排序树

{

binsert(t,shuru);

cin>>shuru;

}

}

1. 遍历btra（）

//遍历

void btra(bstree& t)

{

if(t==NULL)//数为空

{

//cout<<"空树"<<endl;

return;

}

if(t)

{

btra(t->lchild);//遍历左孩子

cout<<t->key<<" ";

btra(t->rchild);//遍历右孩子

}

}

1. 格式化打印bprint()

//格式化打印

void bprint(bstree& t)

{

int j=1,k=0;//层，层内位置

q.front=0,q.rear=1;//初始化

q.data[0]=t;//将根节点相关信息加入队列

q.deep[0]=1;

q.locate[0]=30;

cout<<"\t"<<endl;

while(q.front!=q.rear)

{

//int n=q.rear-q.front;

//cout<<" n"<<n<<endl;

bstree temp=q.data[q.front];

int i=q.deep[q.front];//当前层，层内位置

int nloc=q.locate[q.front];

//cout<<" "<<endl;

if(j<i)//进新层换行

{

cout<<endl<<endl<<endl;

//cout<<endl;

j++;

k=0;

//while(k<(nloc))//制层内位置

{

//cout<<" ";

//k++;

}

}

while(k<(nloc+5))//控制层内位置

{

cout<<" ";

k++;

}

cout<<temp->key;

//cout<<":"<<i;

if(temp->lchild)//有左孩子 ，入队

{

q.data[q.rear]=temp->lchild;//将相关信息加入队列

q.deep[q.rear]=i+1;

q.locate[q.rear]=(int)(nloc-pow(2,(5-i)));

//cout<<" "<<q.locate[q.rear];

q.rear++;

}

if(temp->rchild)//有右孩子 ，入队

{

q.data[q.rear]=temp->rchild;//将相关信息加入队列

q.deep[q.rear]=i+1;

q.locate[q.rear]=(int)(nloc+pow(2,(5-i)));

//cout<<" "<<q.locate[q.rear];

q.rear++;

}

q.front++;//出列

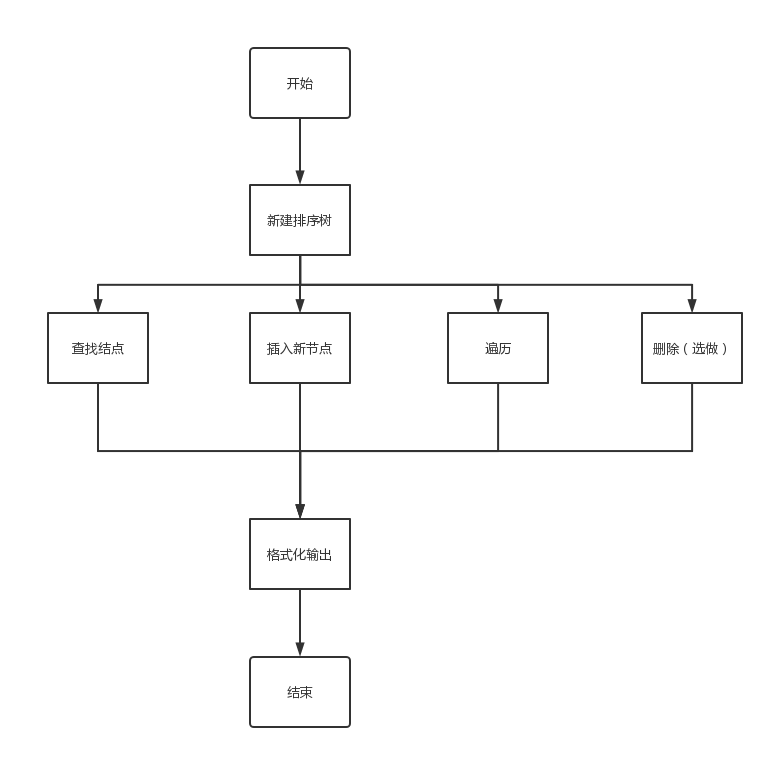
//cout<<"fr"<<front<<endl;

}

cout<<endl;

}

**1.4设计图**



**1.5程序**

**#include<cstdio>**

**#include<cstdlib>**

**#include<cstring>**

**#include<iostream>**

**#include<cmath>**

**#define MAXSIZE 10086**

**using namespace std;**

**//二叉排序树结点**

**typedef struct bstnode**

**{**

**int key;//结点数据元素**

**struct bstnode \*lchild,\*rchild;//左右孩子指针**

**}bstnode,\*bstree;**

**//用于树的打印**

**struct qnode**

**{**

**bstree data[MAXSIZE];//存放树节点**

**int deep[MAXSIZE];//结点所在层**

**int locate[MAXSIZE];//打印节点位置**

**int front,rear;//头指针，尾指针**

**}q;**

**bstree t,p;**

**int shuru2,shuru3,chazhao;**

**//12 7 17 11 16 2 13 9 21 4 0**

**//找值为key的结点 （非递归**

**bstnode \*bsearch(bstree t,int key)**

**{**

**while(t!=NULL&&key!=t->key)**

**{**

**if(key<t->key)//<在左子树找**

**{**

**t=t->lchild;**

**}**

**else//>在右子树找**

**{**

**t=t->rchild;**

**}**

**chazhao++;**

**}**

**//失败返回NULL**

**return t;**

**}**

**//找值为key的结点 （递归**

**bstnode \*bsearch2(bstree t,int key)**

**{**

**if(t==NULL)//失败返回NULL**

**return NULL;**

**if(key==t->key)//成功**

**return t;**

**else if(key<t->key)//<在左子树找**

**{**

**return bsearch(t->lchild,key);**

**}**

**else//>在右子树找**

**{**

**return bsearch(t->rchild,key);**

**}**

**return t;**

**}**

**//插入值为key的结点 递归**

**int binsert(bstree& t,int key)**

**{**

**if(t==NULL)//失败**

**{**

**t=(bstree)malloc(sizeof(bstnode));**

**t->key=key;**

**t->lchild=t->rchild=NULL;**

**return 1; //插入成功**

**}**

**if(key==t->key)//存在相同，插入失败**

**return 0;**

**else if(key<t->key)//<在左子树找**

**{**

**binsert(t->lchild,key);**

**}**

**else//>在右子树找**

**{**

**binsert(t->rchild,key);**

**}**

**}**

**//按照数组关键字建立二叉排序树**

**void cbst(bstree &t)**

**{**

**t=NULL;//初始t空**

**int shuru;**

**cin>>shuru;**

**while(shuru!=0)//依次将每个关键字插入二叉排序树**

**{**

**binsert(t,shuru);**

**cin>>shuru;**

**}**

**}**

**//遍历**

**void btra(bstree& t)**

**{**

**if(t==NULL)//数为空**

**{**

**//cout<<"空树"<<endl;**

**return;**

**}**

**if(t)**

**{**

**btra(t->lchild);//遍历左孩子**

**cout<<t->key<<" ";**

**btra(t->rchild);//遍历右孩子**

**}**

**}**

**//格式化打印**

**void bprint(bstree& t)**

**{**

**int j=1,k=0;//层，层内位置**

**q.front=0,q.rear=1;//初始化**

**q.data[0]=t;//将根节点相关信息加入队列**

**q.deep[0]=1;**

**q.locate[0]=30;**

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

**while(q.front!=q.rear)**

**{**

**//int n=q.rear-q.front;**

**//cout<<" n"<<n<<endl;**

**bstree temp=q.data[q.front];**

**int i=q.deep[q.front];//当前层，层内位置**

**int nloc=q.locate[q.front];**

**//cout<<" "<<endl;**

**if(j<i)//进新层换行**

**{**

**cout<<endl<<endl<<endl;**

**//cout<<endl;**

**j++;**

**k=0;**

**//while(k<(nloc))//制层内位置**

**{**

**//cout<<" ";**

**//k++;**

**}**

**}**

**while(k<(nloc+5))//控制层内位置**

**{**

**cout<<" ";**

**k++;**

**}**

**cout<<temp->key;**

**//cout<<":"<<i;**

**if(temp->lchild)//有左孩子 ，入队**

**{**

**q.data[q.rear]=temp->lchild;//将相关信息加入队列**

**q.deep[q.rear]=i+1;**

**q.locate[q.rear]=(int)(nloc-pow(2,(5-i)));**

**//cout<<" "<<q.locate[q.rear];**

**q.rear++;**

**}**

**if(temp->rchild)//有右孩子 ，入队**

**{**

**q.data[q.rear]=temp->rchild;//将相关信息加入队列**

**q.deep[q.rear]=i+1;**

**q.locate[q.rear]=(int)(nloc+pow(2,(5-i)));**

**//cout<<" "<<q.locate[q.rear];**

**q.rear++;**

**}**

**q.front++;//出列**

**//cout<<"fr"<<front<<endl;**

**}**

**cout<<endl;**

**}**

**//删除**

**bool bdelete(bstree& p)**

**{**

**bstree q1,f,s;**

**f=NULL;**

**//查找key**

**//cout<<p->key<<endl;**

**// 删除p**

**if (!p->rchild&&!p->lchild)//叶子节点**

**{**

**//cout<<0<<endl;**

**p=NULL;**

**//free(p);**

**}**

**else if (!p->rchild)//只有左孩子，左孩子代替**

**{**

**q1=p;**

**p=p->lchild;**

**//q1=p->lchild;**

**//p->key=q1->key;**

**free(q1);**

**}**

**else if (!p->lchild)//只有右孩子，右孩子代替**

**{**

**q1=p;**

**p=p->rchild;//只需用结点 p 的右子树根结点代替结点 p 即可**

**free(q1);**

**}**

**else//左右孩子均存在**

**{**

**q1=p;**

**s=p->lchild;//p左孩子**

**while(s->rchild)//s为中序遍历前驱**

**{**

**q1=s;**

**s=s->rchild;**

**}**

**p->key = s->key;**

**//判断结点 p 的左子树 s 是否有右子树，分为两种情况讨论**

**if( q1 != p )**

**q1->rchild = s->lchild; //若有，则在删除直接前驱结点的同时，令前驱的左孩子结点改为 q 指向结点的孩子结点**

**else**

**q1->lchild = s->lchild; //否则，直接将左子树上移即可**

**free(s);**

**}**

**return 1;**

**}**

**//删除t**

**int bstdel(bstree&t,int key)**

**{**

**if(t==NULL)//不存在关键字等于key的数据元素**

**return 0;**

**if(key==t->key)//成功**

**{**

**bdelete(t);**

**return 1;**

**}**

**else if(key<t->key)//<在左子树找**

**{**

**return bstdel(t->lchild,key);**

**}**

**else//>在右子树找**

**{**

**return bstdel(t->rchild,key);**

**}**

**return 1;**

**}**

**int main()**

**{**

**cout<<"创建排序二叉树："<<endl;**

**cbst(t);**

**cout<<"遍历二叉树："<<endl;**

**btra(t);**

**cout<<endl;**

**cout<<"打印二叉树："<<endl;**

**bprint(t);**

**cout<<endl;**

**cout<<"查找二叉树（输入你要查找的key）："<<endl;**

**cin>>shuru2;**

**p=bsearch(t,shuru2);**

**if(p)**

**{**

**cout<<"该节点存在"<<endl;**

**cout<<"次数："<<chazhao+1<<endl;**

**}**

**else**

**{**

**cout<<"该节点不存在"<<endl;**

**cout<<"次数："<<chazhao+1<<endl;**

**}**

**cout<<"删除二叉树（输入你要查找的key）："<<endl;**

**cin>>shuru3;**

**int panduan=0;**

**panduan=bstdel(t,shuru3);**

**if(panduan)**

**{**

**cout<<"删除成功"<<endl;**

**}**

**else**

**{**

**cout<<"删除失败"<<endl;**

**}**

**cout<<"遍历二叉树："<<endl;**

**btra(t);**

**cout<<endl;**

**cout<<"打印二叉树："<<endl;**

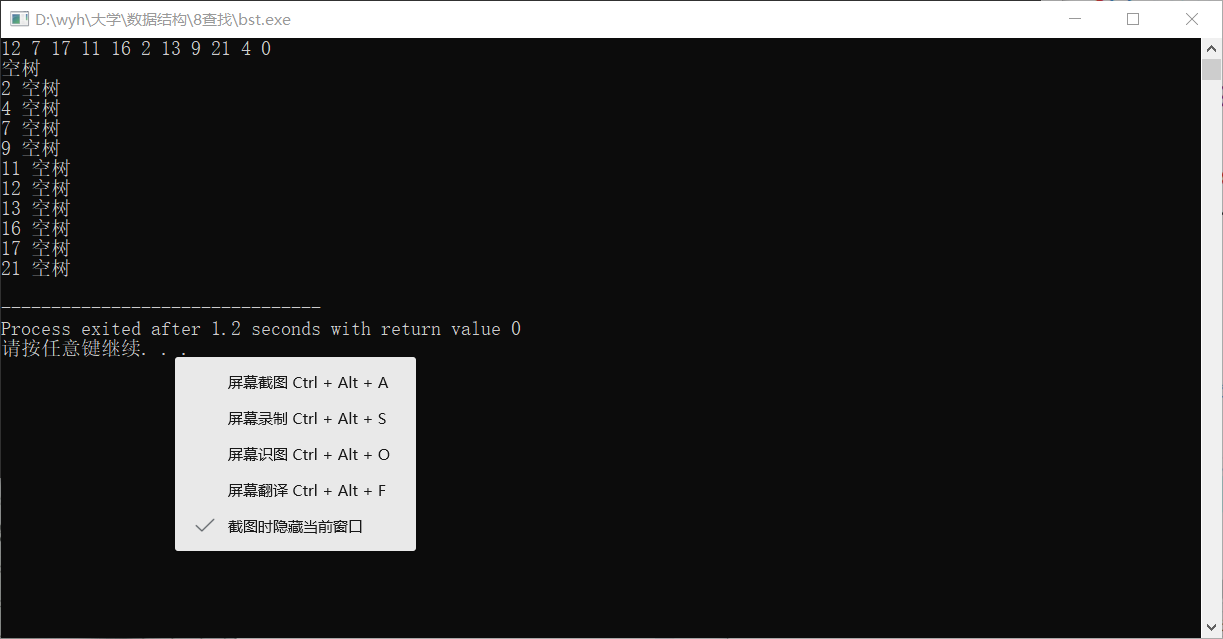
**bprint(t);**

**cout<<endl;**

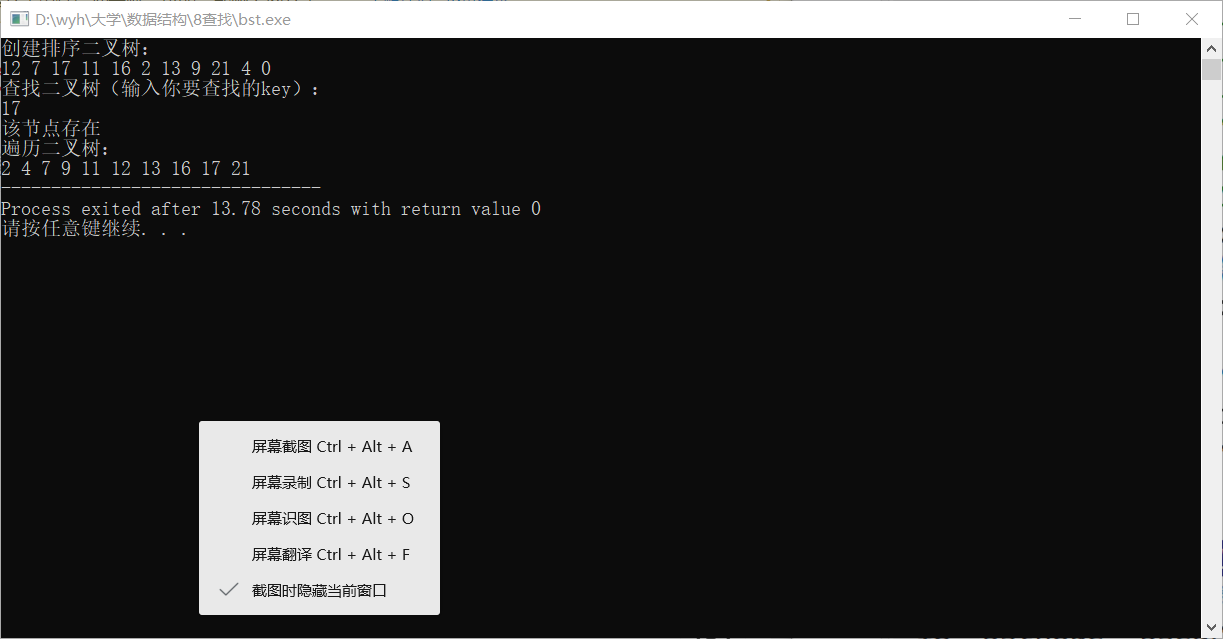
**return 0;**

**}**

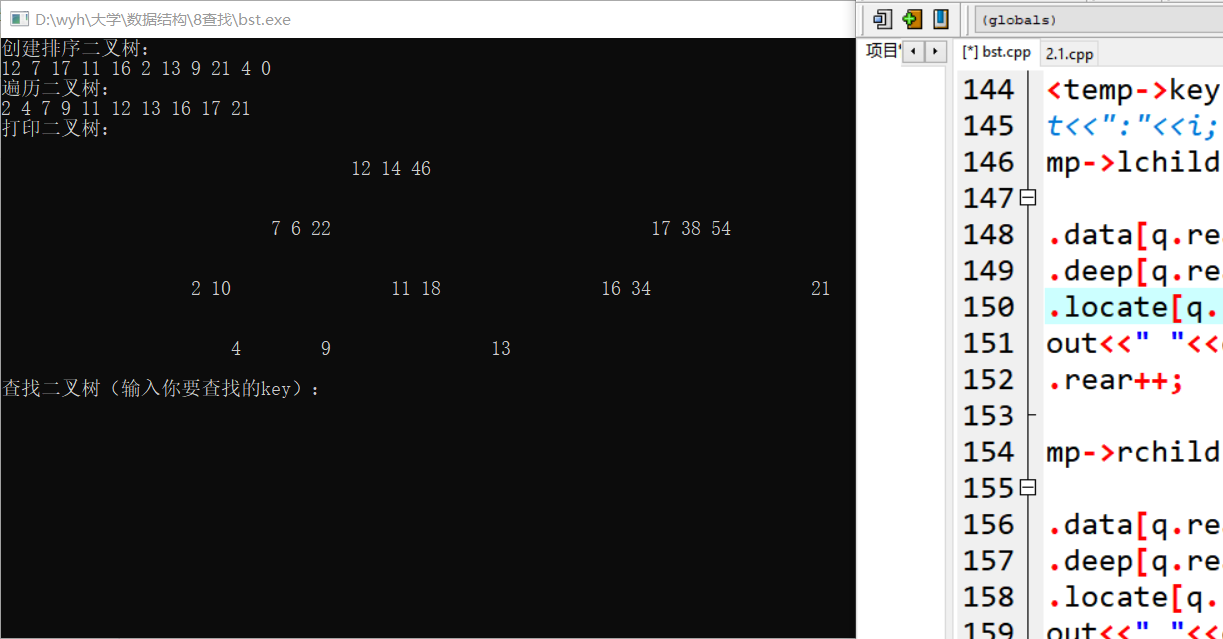
**1.6调试过程截图**

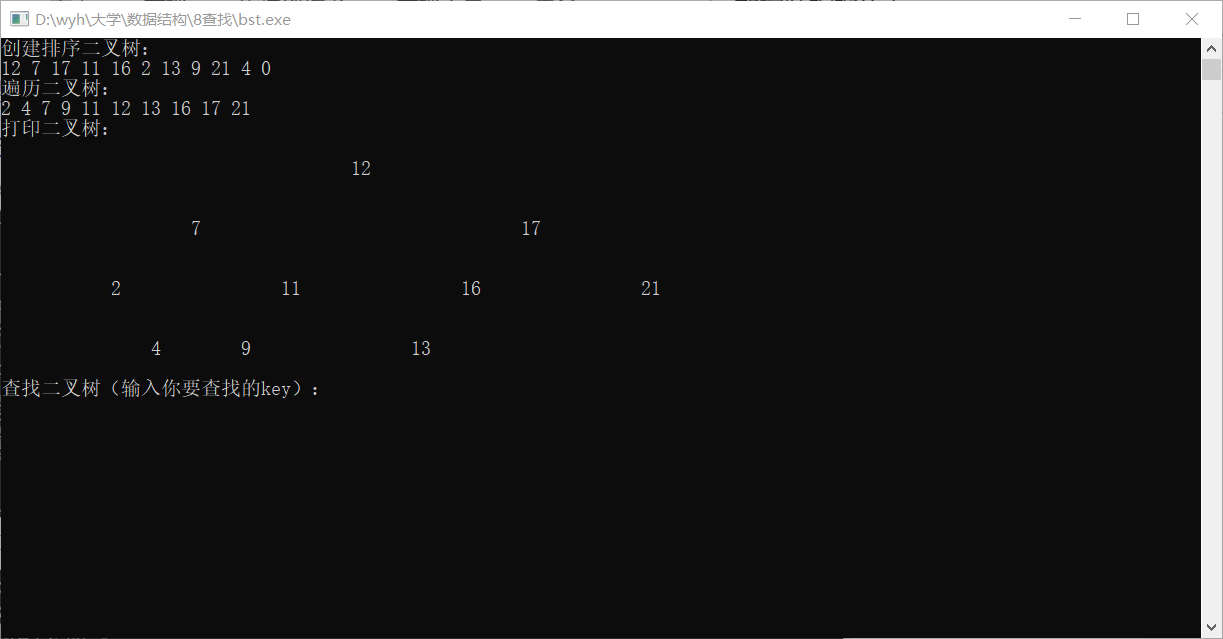


数据未传入，显示空树



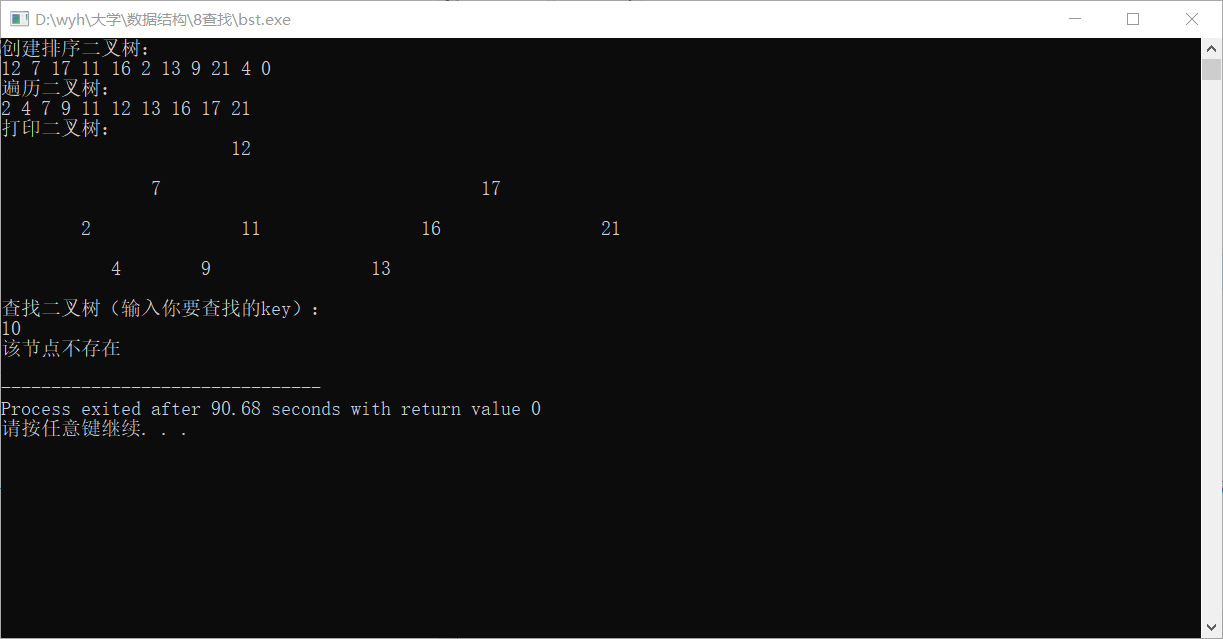
查找测试



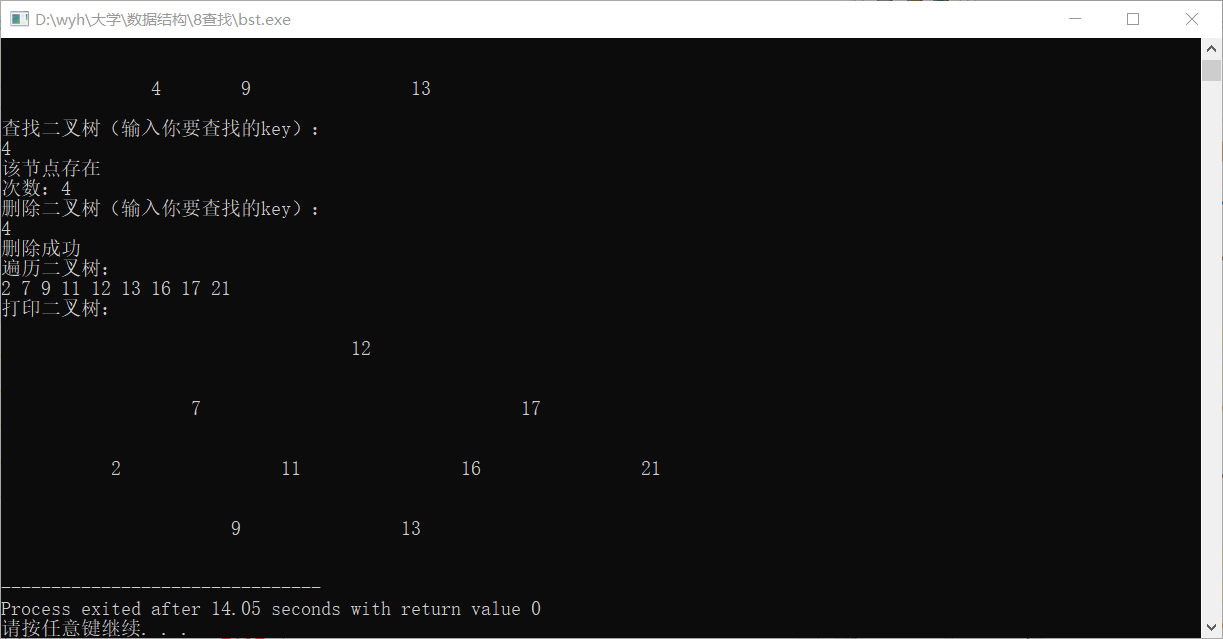
****

输出格式化打印子节点的位置

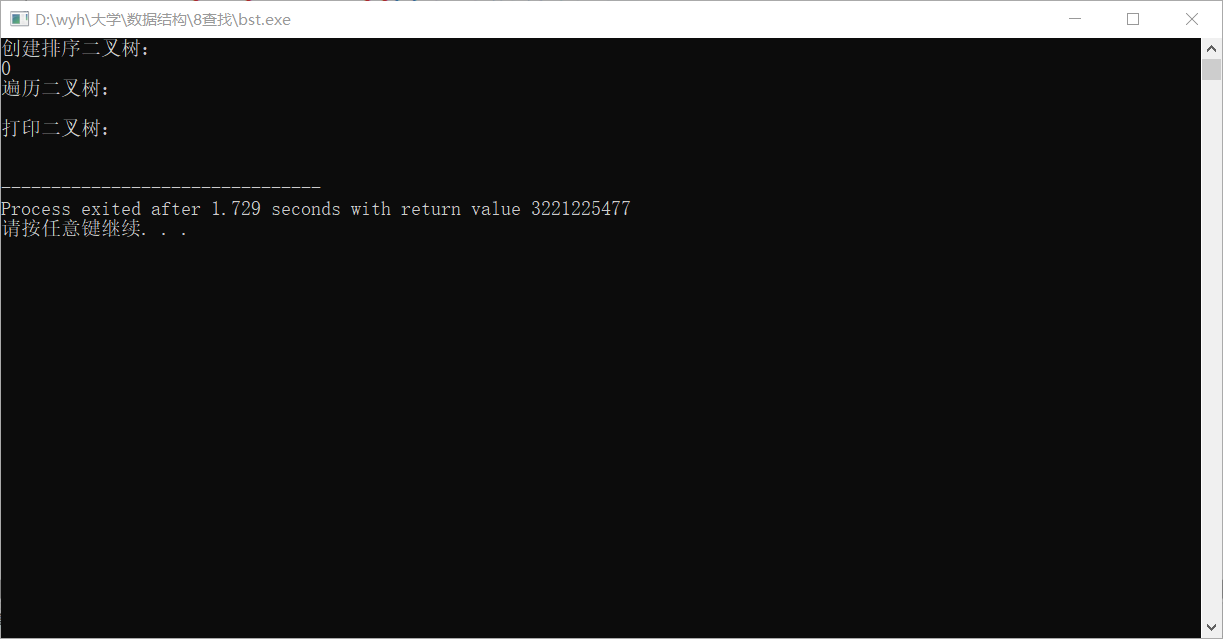
**1.7结果截图**

****

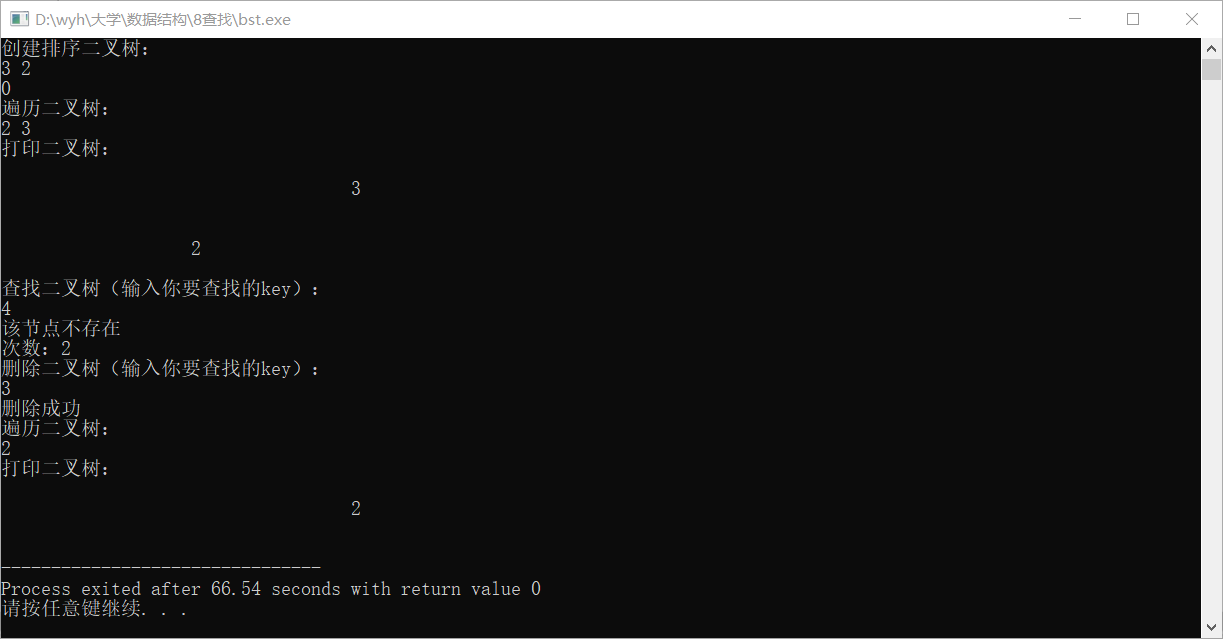
**删除后**

****

**删除**

****

**空树**

****

**查找不存在的节点**

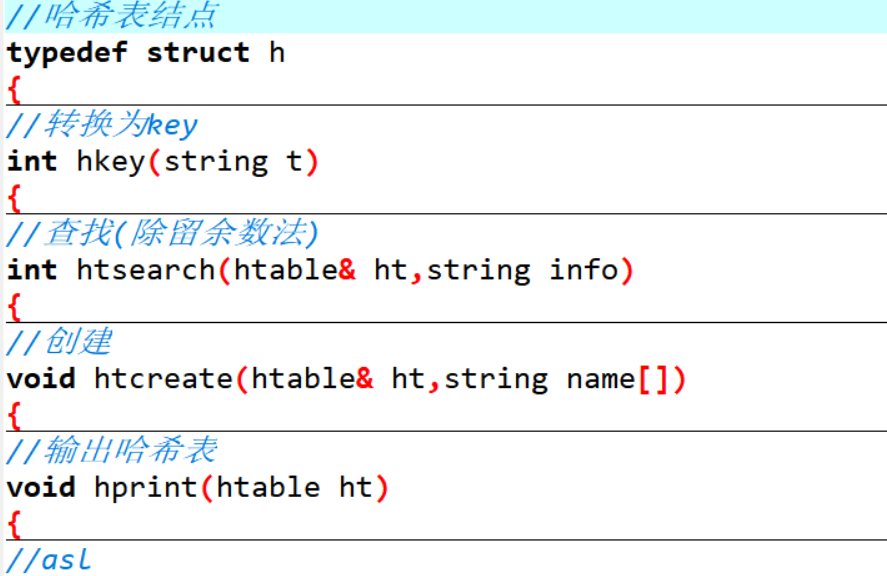
**2.哈希表设计**

**2.1问题分析**

哈希表可以通过关键字直接找到数据的存储位置，不需要进行任何的比较.哈希地址只是表示在查找表中的存储位置，而不是实际的物理存储位置。f（）是一个函数，通过这个函数可以快速求出该关键字对应的的数据的哈希地址。哈希函数用除留余数法，若已知整个哈希表的最大长度 m，可以取一个不大于 m 的数 p，然后对该关键字 key 做取余运算，即：H（key）= key % p。

**2.2设计方案**

1. **哈希表节点**
2. **转化为key的函数**
3. **查找函数**
4. **创建函数**
5. **输出函数**
6. **求asl函数**

****

**2.3算法**

1. **哈希表节点**

//哈希表结点

typedef struct h

{

int key;

string hinfo;

}htable[40];

1. **转化为key的函数**

//转换为key

int hkey(string t)

{

int key1=0;

//将字符串的各个字符所对应的ASCII码相加，所得的整数做为哈希表的关键字

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

{

key1+=(int)t[i];

// cout<<key1<<endl;

}

return key1;

}

1. **查找函数**

//查找(除留余数法)

int htsearch(htable& ht,string info)

{

int a=hkey(info);

int h0=a%37;//求其在表中的位子

count=1;

if(ht[h0].hinfo==info) //成功

{

return h0;

}

else

{

for(int i=1;i<100;i++)

{

int hi=(h0+i)%40;//线性探测

count++;

if(ht[hi].hinfo==info)

{

return hi;

}

else if(ht[hi].key==-1)

{

break;

}

}

}

return -1;

}

1. **创建函数**

//创建

void htcreate(htable& ht,string name[])

{

//memset(ht[])

for(int i=0;i<40;i++)

{

ht[i].key=-1;

}

for(int i=0;i<30;i++)

{

//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37

int a=hkey(name[i]);

int h0=a%37;

if(ht[h0].key==-1)//该位没有值

{

ht[h0].hinfo=name[i];

ht[h0].key=h0;

}

else//冲突探测

{

for(int j=1;j<100;j++)

{

int hj=(h0+j)%40;

if(ht[hj].key==-1)

{

ht[hj].hinfo=name[i];

ht[hj].key=hj;

break;

}

}

}

}

}

1. **输出函数**

//输出哈希表

void hprint(htable ht)

{

cout<<"id 哈 希 值 人 名"<<endl;

for(int i=0;i<30;i++)

{

printf("%-4d\t%8d\t",i,ht[i].key);

cout<<ht[i].hinfo<<endl;

}

}

1. **求asl函数**

//asl

double asl(htable& ht,string name[])

{

double re=0;

for(int i=0;i<30;i++)

{

htsearch(ht,name[i]);

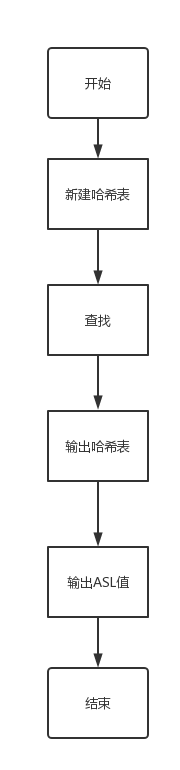
re+=count;

}

return re/30.0;

}

**2.4设计图**

****

**2.5程序**

**#include<cstdio>**

**#include<cstdlib>**

**#include<cstring>**

**#include<iostream>**

**#include<cmath>**

**#define MAXSIZE 10086**

**using namespace std;**

**int hlist[40];**

**int count=0;//计算asl**

**//哈希表结点**

**typedef struct h**

**{**

**int key;**

**string hinfo;**

**}htable[40];**

**//转换为key**

**int hkey(string t)**

**{**

**int key1=0;**

**//将字符串的各个字符所对应的ASCII码相加，所得的整数做为哈希表的关键字**

**for(int i=0;i<t.length();i++)**

**{**

**key1+=(int)t[i];**

**// cout<<key1<<endl;**

**}**

**return key1;**

**}**

**//查找(除留余数法)**

**int htsearch(htable& ht,string info)**

**{**

**int a=hkey(info);**

**int h0=a%37;//求其在表中的位子**

**count=1;**

**//cout<<info<<":"<<h0<<endl;**

**if(ht[h0].hinfo==info) //成功**

**{**

**return h0;**

**}**

**else**

**{**

**for(int i=1;i<100;i++)**

**{**

**int hi=(h0+i)%40;//线性探测**

**count++;**

**if(ht[hi].hinfo==info)**

**{**

**return hi;**

**}**

**else if(ht[hi].key==-1)**

**{**

**break;**

**}**

**}**

**}**

**return -1;**

**}**

**//创建**

**void htcreate(htable& ht,string name[])**

**{**

**//memset(ht[])**

**for(int i=0;i<40;i++)**

**{**

**ht[i].key=-1;**

**}**

**for(int i=0;i<30;i++)**

**{**

**//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37**

**int a=hkey(name[i]);**

**int h0=a%37;**

**if(ht[h0].key==-1)//该位没有值**

**{**

**ht[h0].hinfo=name[i];**

**ht[h0].key=h0;**

**}**

**else//冲突探测**

**{**

**for(int j=1;j<100;j++)**

**{**

**int hj=(h0+j)%40;**

**if(ht[hj].key==-1)**

**{**

**ht[hj].hinfo=name[i];**

**ht[hj].key=hj;**

**break;**

**}**

**}**

**}**

**}**

**}**

**//输出哈希表**

**void hprint(htable ht)**

**{**

**cout<<"id 哈 希 值 人 名"<<endl;**

**for(int i=0;i<30;i++)**

**{**

**printf("%-4d\t%8d\t",i,ht[i].key);**

**cout<<ht[i].hinfo<<endl;**

**}**

**}**

**//asl**

**double asl(htable& ht,string name[])**

**{**

**double re=0;**

**for(int i=0;i<30;i++)**

**{**

**htsearch(ht,name[i]);**

**cout<<name[i]<<":";**

**printf("%-10d\n",count);**

**re+=count;**

**}**

**return re/30.0;**

**}**

**int main()**

**{**

**/\* string name[30]={**

**"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"**

**};\*/**

**string name[30]={**

**"wangyuhan","dukefei","sutianyu","shihanpeng","xiedonglin",**

**"sunliangliang","pengsihao","hanrui","hourunning","xiekai",**

**"zhaoran","jiahaochen","zhangaiqi","yaoliunian","lijiahe",**

**"luxintong","zhouying","zhanglongwei","hanyu","zixinyuan",**

**"xutianhang","heyujie","wangxiaochen","linanxin","zhangshuning",**

**"yanhaolong","zhangmingxin","majiaji","liutianyi","mayuchen"**

**};**

**cout<<"线性探测再散列法处理冲突"<<endl;**

**htable ht;**

**string shuru;**

**htcreate(ht,name);**

**cin>>shuru;//输入要查找的名字**

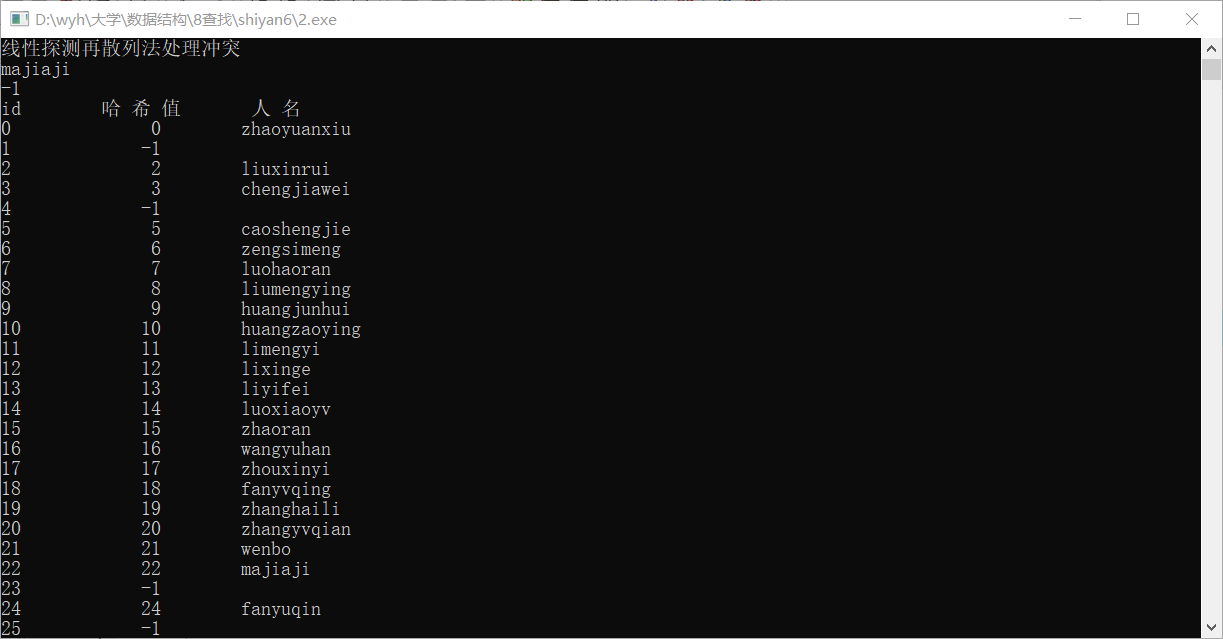
**cout<<htsearch(ht,shuru)<<endl;**

**hprint(ht);**

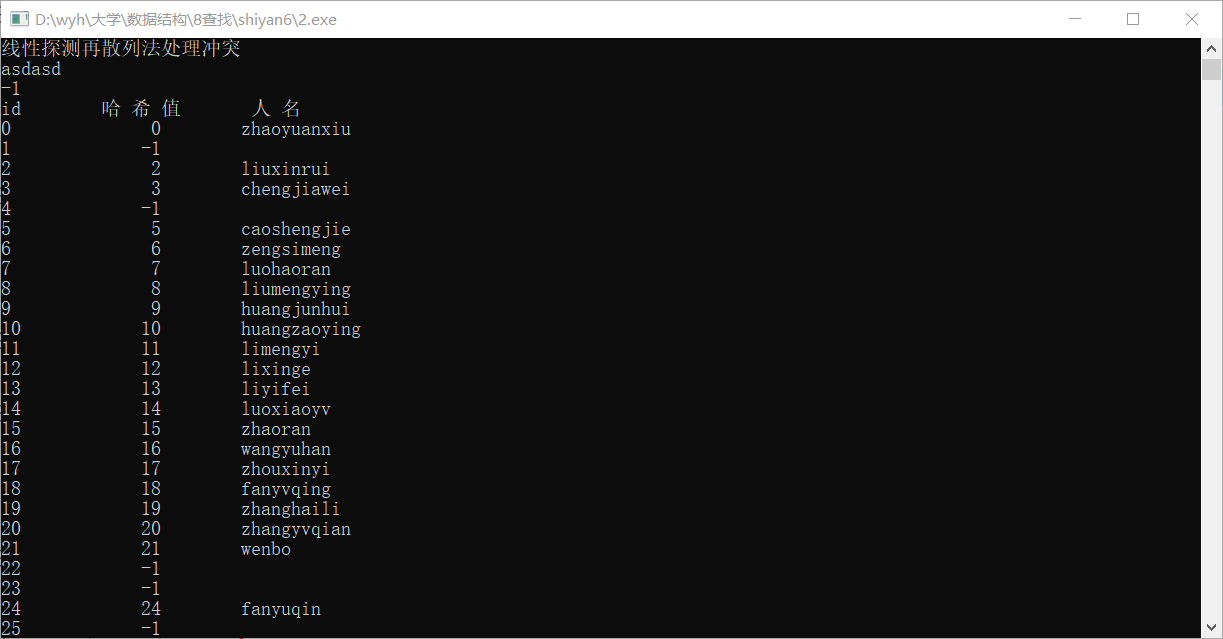
**cout<<"asl:"<<asl(ht,name)<<endl;**

**return 0;**

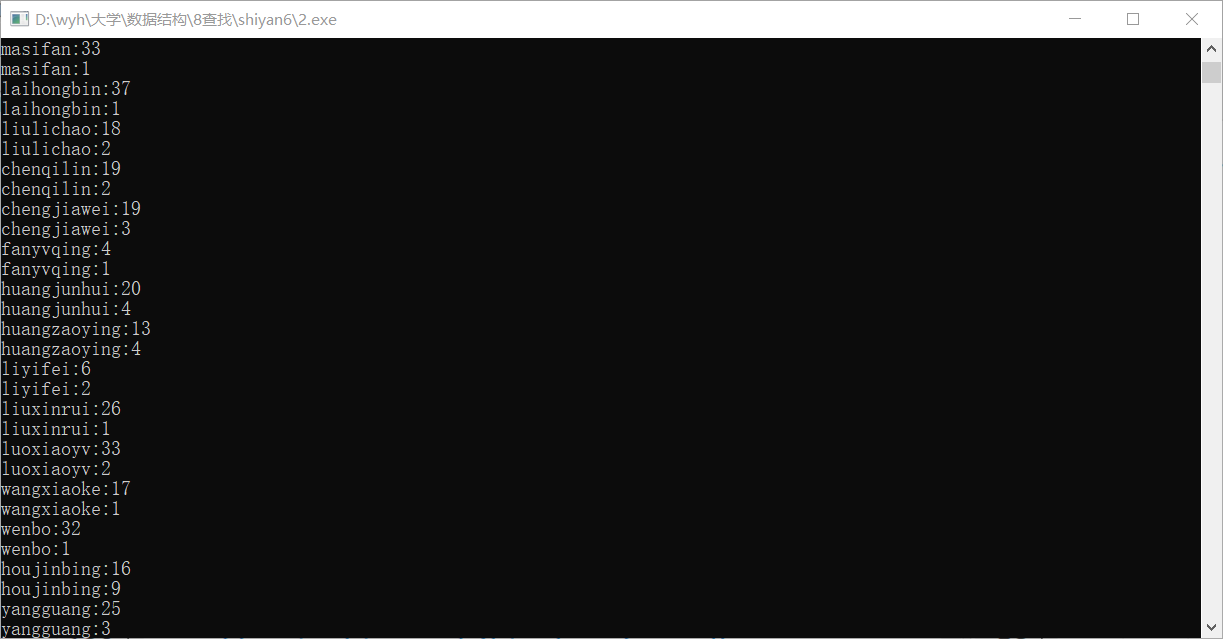
**}2.6调试过程截图**

****

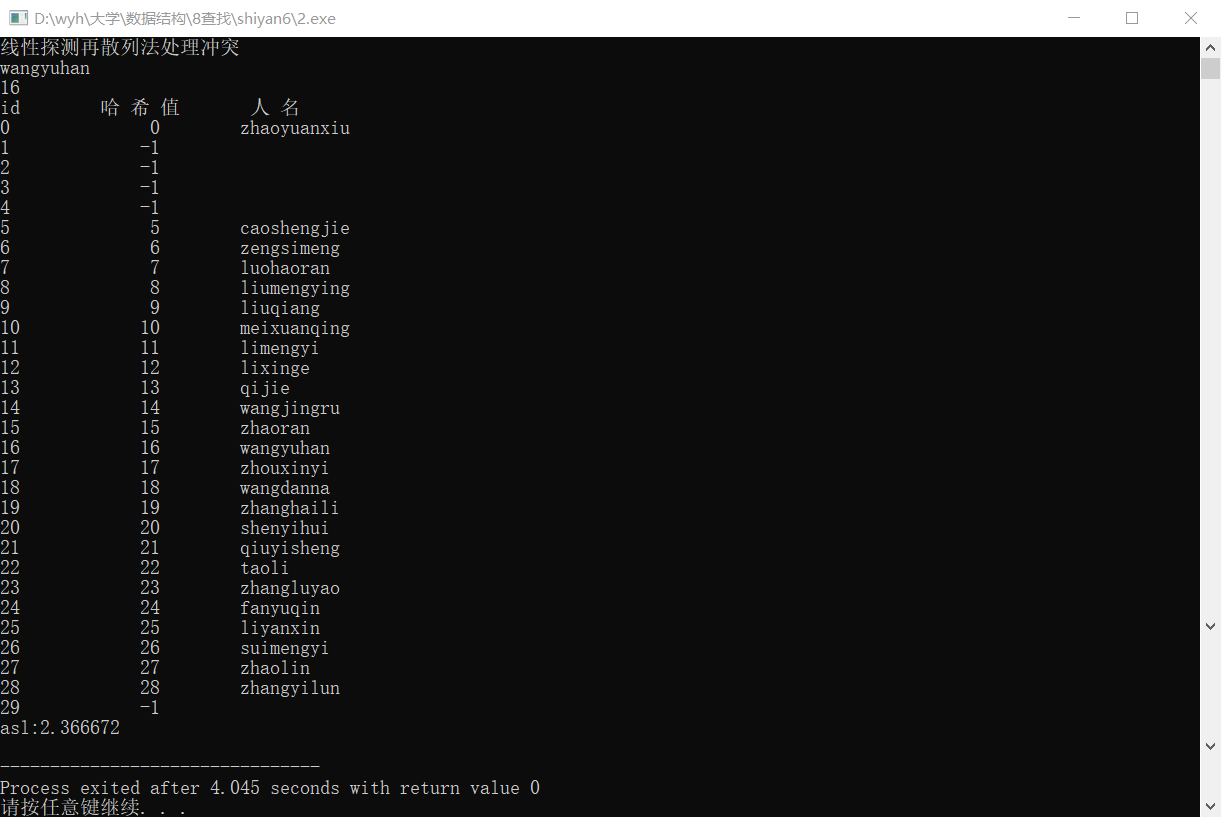
**查找错误，发现是m值输错了**

****

**测试不同m影响，输出-1表示查找不到**

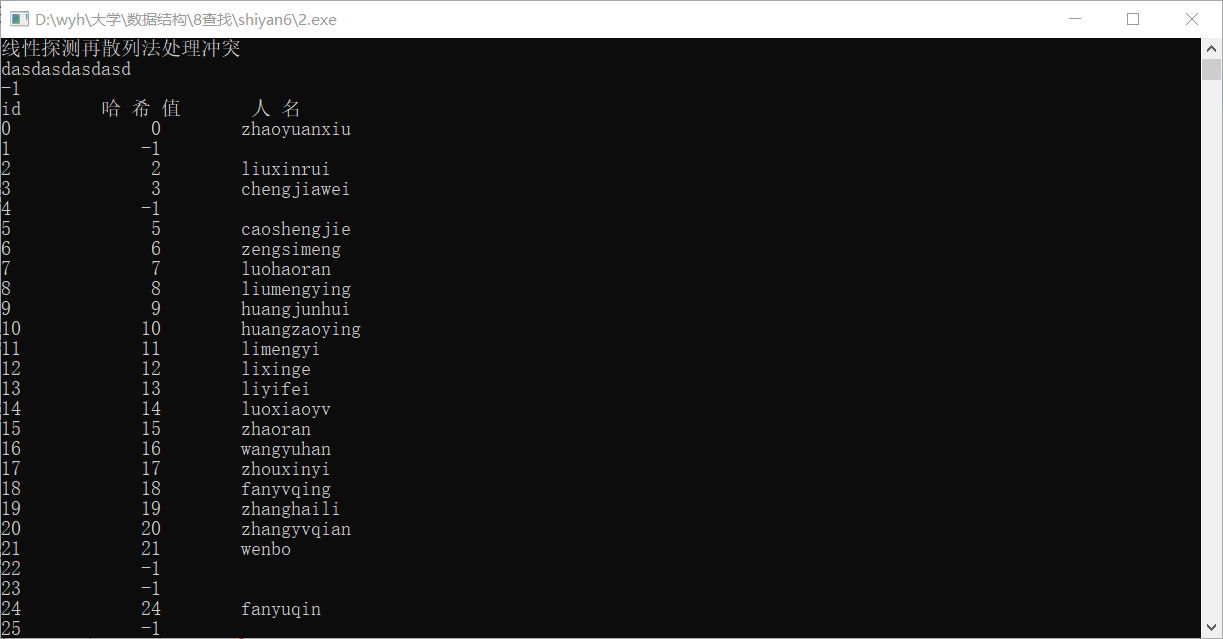
****

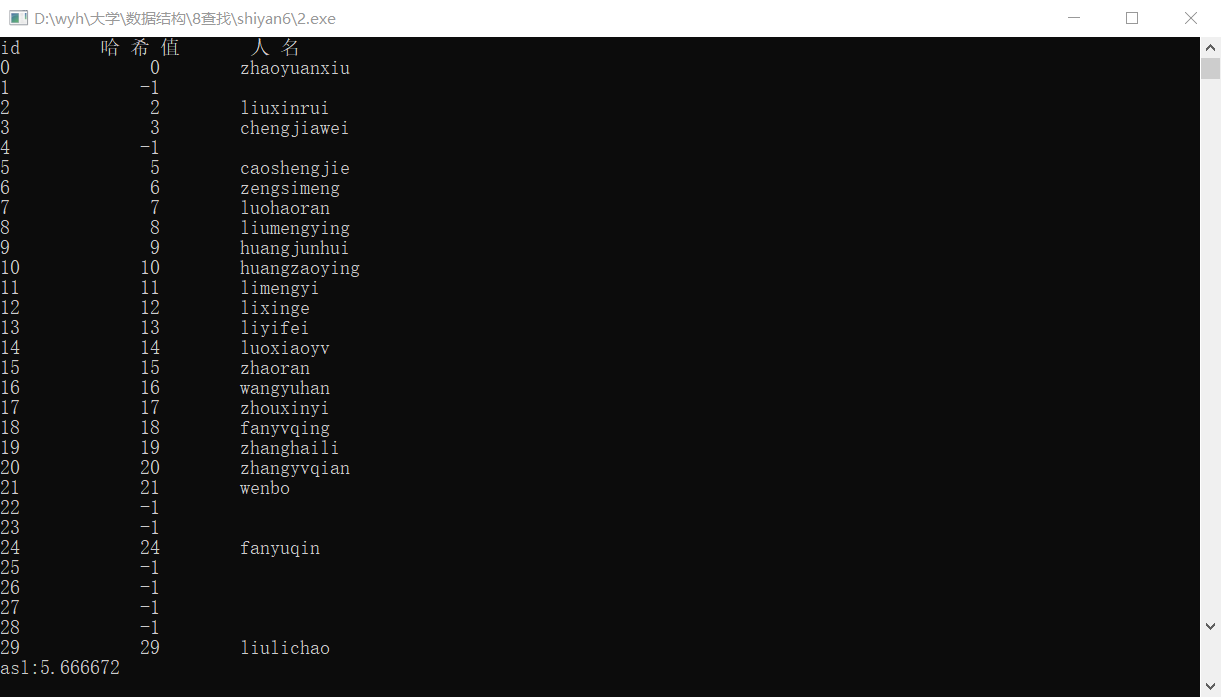
**查看各个key和查找结果**

****

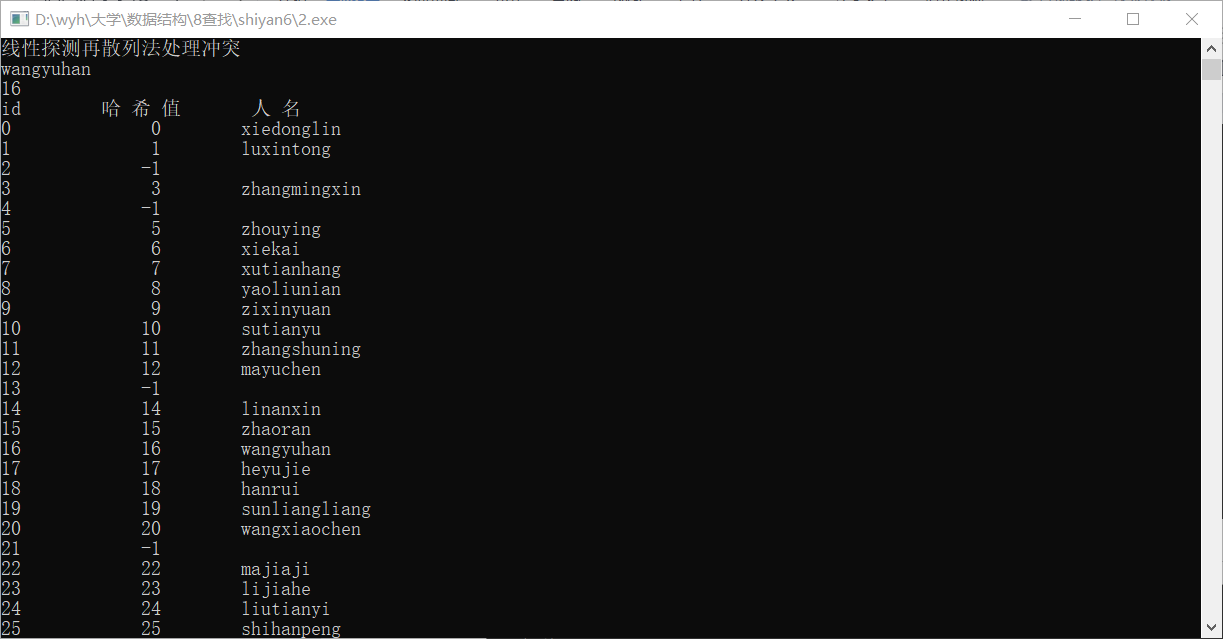
**查找wangyuyhan输出位置为16正确**

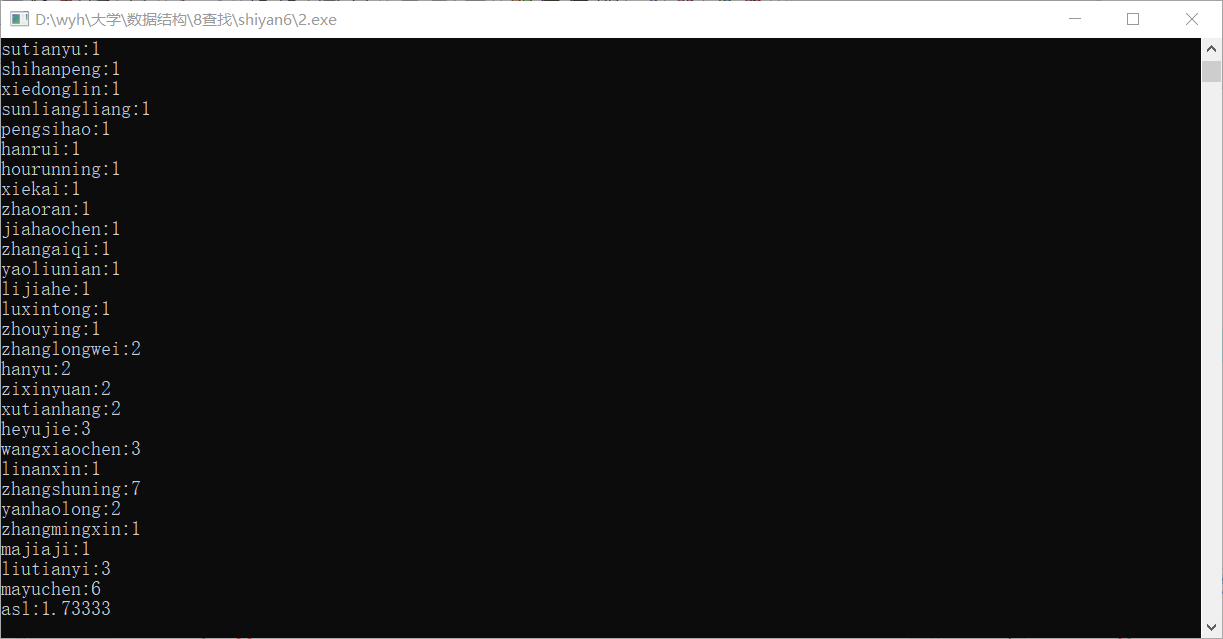
**-1表示在表中无值**

****

****

**2.7运行结果截图**

****

****

**//更改后asl<2**

**该组测试数据的平均成功查找长度为1.73。**

**3选做**

* + - * 1. 实现二叉排序树的插入、删除操作。

//插入值为key的结点 递归

int binsert(bstree& t,int key)

{

if(t==NULL)//失败

{

t=(bstree)malloc(sizeof(bstnode));

t->key=key;

t->lchild=t->rchild=NULL;

return 1; //插入成功

}

if(key==t->key)//存在相同，插入失败

return 0;

else if(key<t->key)//<在左子树找

{

binsert(t->lchild,key);

}

else//>在右子树找

{

binsert(t->rchild,key);

}

}

//删除

bool bdelete(bstree& p)

{

bstree q1,f,s;

f=NULL;

//查找key

//cout<<p->key<<endl;

// 删除p

if (!p->rchild&&!p->lchild)//叶子节点

{

//cout<<0<<endl;

p=NULL;

//free(p);

}

else if (!p->rchild)//只有左孩子，左孩子代替

{

q1=p;

p=p->lchild;

//q1=p->lchild;

//p->key=q1->key;

free(q1);

}

else if (!p->lchild)//只有右孩子，右孩子代替

{

q1=p;

p=p->rchild;//只需用结点 p 的右子树根结点代替结点 p 即可

free(q1);

}

else//左右孩子均存在

{

q1=p;

s=p->lchild;//p左孩子

while(s->rchild)//s为中序遍历前驱

{

q1=s;

s=s->rchild;

}

p->key = s->key;

//判断结点 p 的左子树 s 是否有右子树，分为两种情况讨论

if( q1 != p )

q1->rchild = s->lchild; //若有，则在删除直接前驱结点的同时，令前驱的左孩子结点改为 q 指向结点的孩子结点

else

q1->lchild = s->lchild; //否则，直接将左子树上移即可

free(s);

}

return 1;

}

//删除t

int bstdel(bstree&t,int key)

{

if(t==NULL)//不存在关键字等于key的数据元素

return 0;

if(key==t->key)//成功

{

bdelete(t);

return 1;

}

else if(key<t->key)//<在左子树找

{

return bstdel(t->lchild,key);

}

else//>在右子树找

{

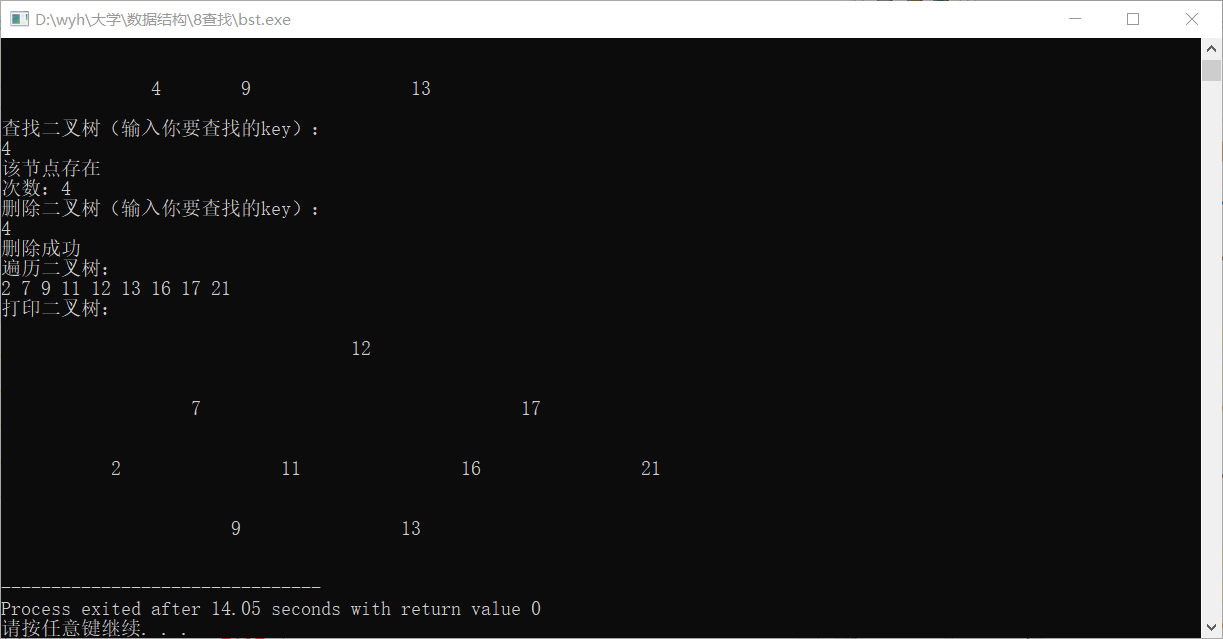
return bstdel(t->rchild,key);

}

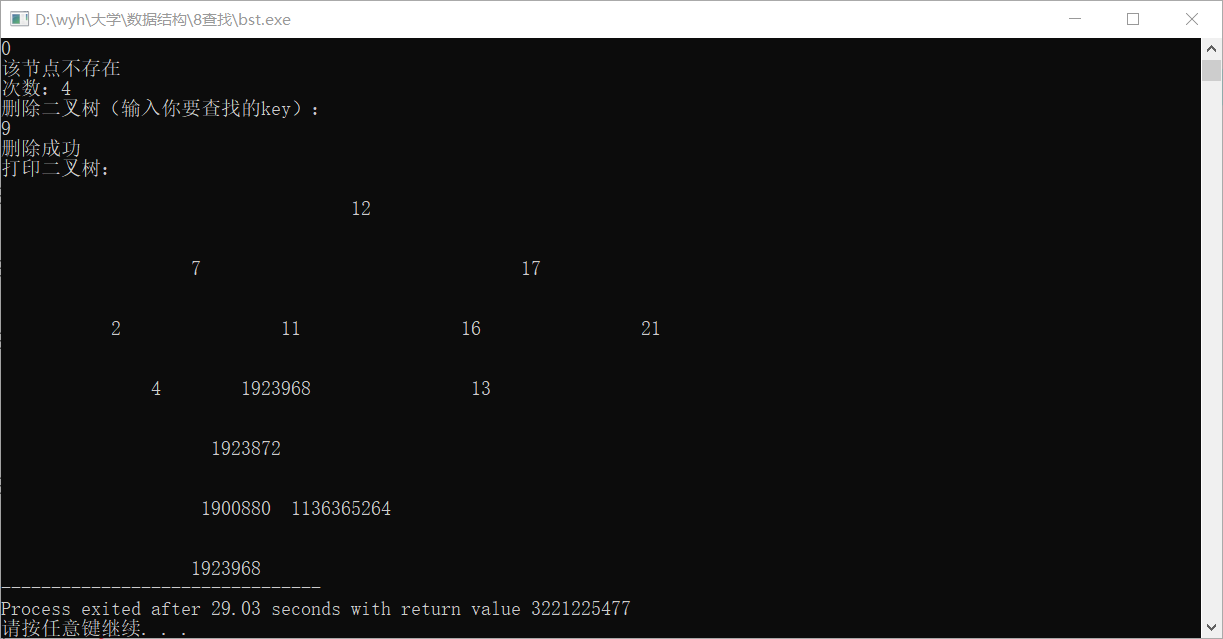
return 1;

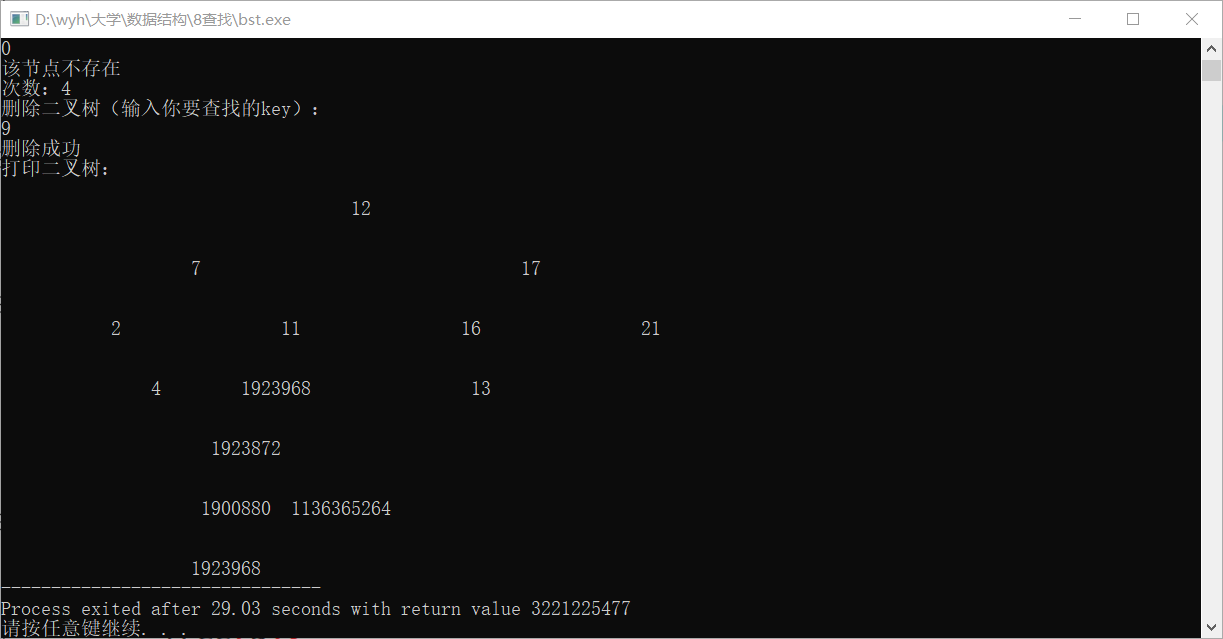
}

**结果如图：**

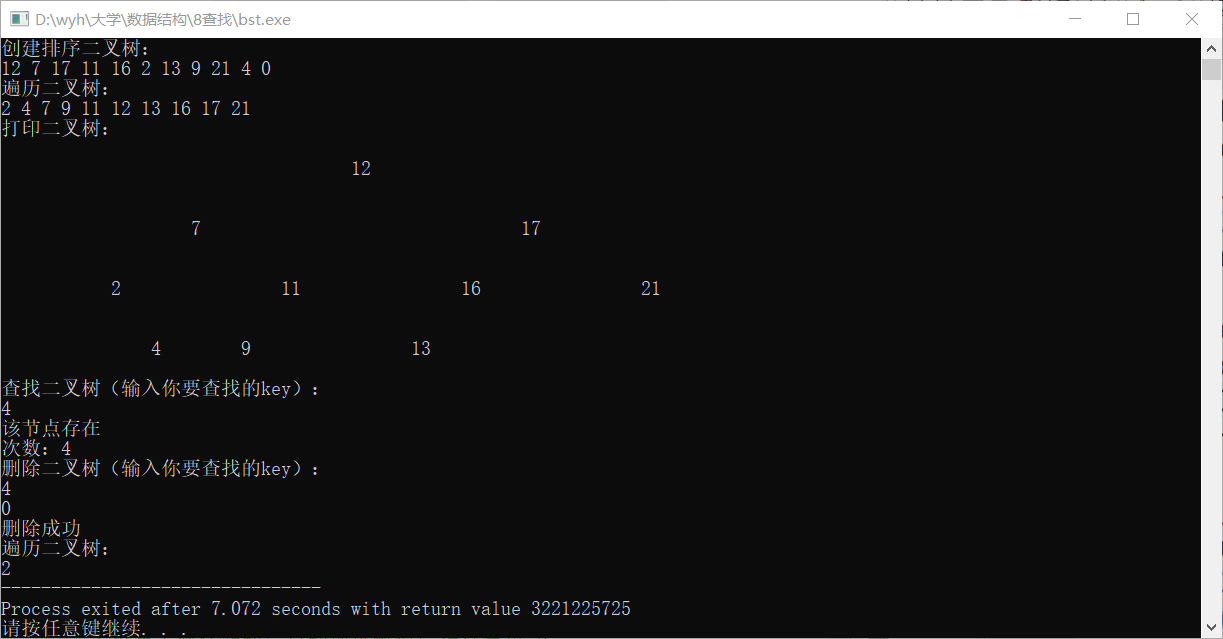


**调试过程：**

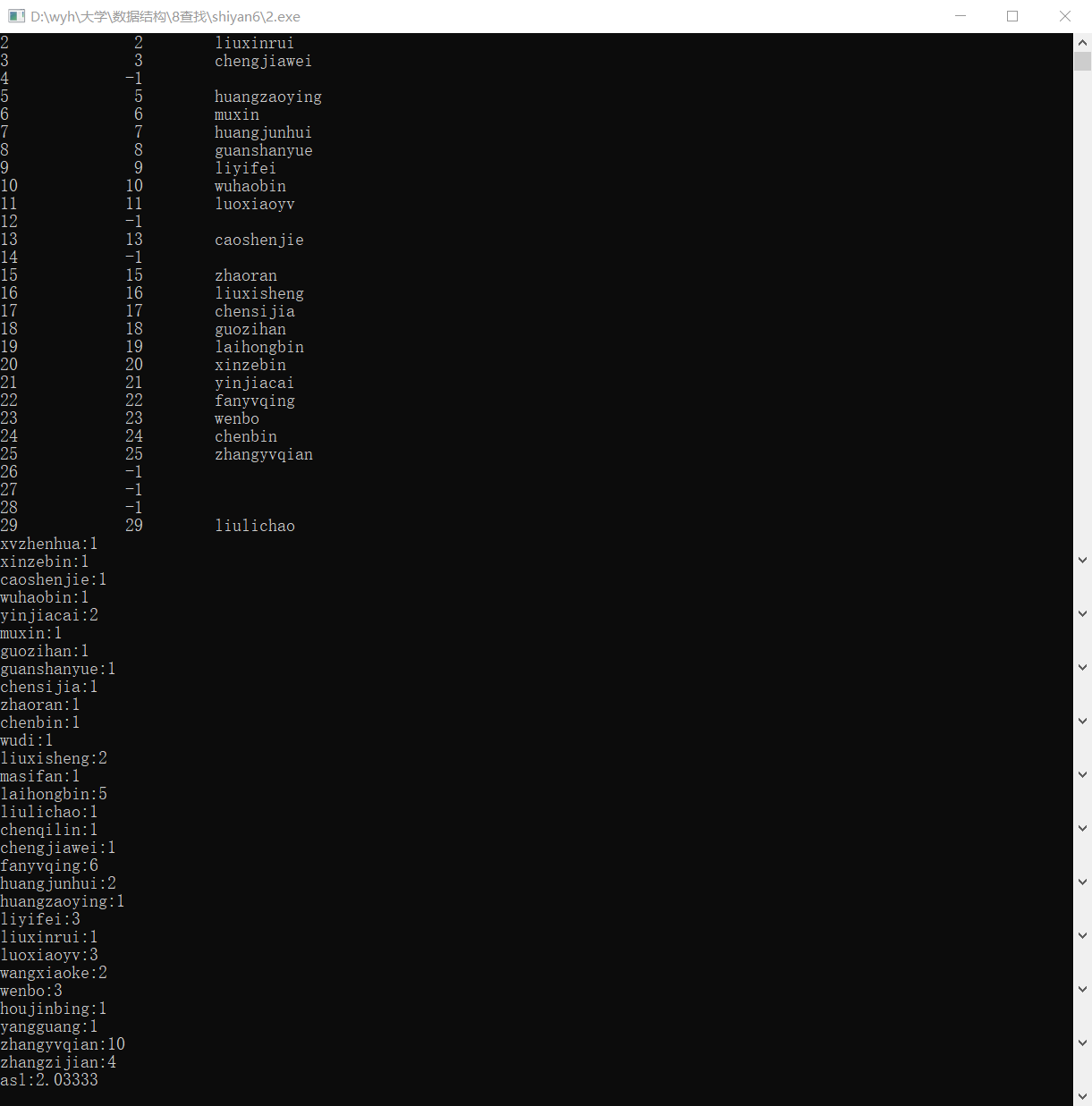




**删除叶子节点失败**



* + - * 1. 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）。
        2. 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。
        3. 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。



**有查找次数可知测试样例不同结果不同**

**为了降低平均成功查找长度，应该使哈希值尽可能分散的分布在整个哈希表中，而适当的降低装填因子可以提高哈希表的查找效率，用空间换时间。**

1. **实验心得**

通过这次实验我对查找表中二叉排序树和哈希表有了更深的了解。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。

二叉排序树在中序遍历得到的是从小到大序列。存储方式与树的实验相同，才有左右子树的指针+key值进行存储。二叉排序树本身是动态查找表的一种表示形式，有时会在查找过程中插入或者删除表中元素，当因为查找失败而需要插入数据元素时，该数据元素的插入位置一定位于二叉排序树的叶子结点，并且一定是查找失败时访问的最后一个结点的左孩子或者右孩子。在查找过程中，如果在使用二叉排序树表示的动态查找表中删除某个数据元素时，需要在成功删除该结点的同时，依旧使这棵树为二叉排序树。  
常用的哈希函数的构造方法有 6 种：直接定址法、数字分析法、平方取中法、折叠法、除留余数法和随机数法。ASL（Average Search Length），即平均查找长度，在查找运算中，由于所费时间在关键字的比较上，所以把平均需要和待查找值比较的关键字次数成为平均查找长度。

同时，我也明白了只有多实践，才能更好的理解掌握数据结构的精髓。从而避免基础语法错误，让代码变得更简洁高效。如此才能准确高效的解决问题。在今后的编程过程中要更注重代码整体设计的提纲记录，用更多的注释，让自己的思路更清晰。