数据结构与算法 第五次实验

学号：22920212204392 姓名：黄勖

一、 实验目的

1. 了解排序算法的实现方法与原理，实践操作编写折半查找

2. 了解设计二叉排序树的查找算法、插入算法和删除算法的实现方法与原理，在理解的基础上学习代码编写

3. 学会灵活按照哈希表设计的存储方式自由编写存储结构代码，学会相关的处理方式

二、 实验内容

5-1 给定有序整型数组 A[n]和整数 x，试设计一个在 A 中查找 x 的折半查找算法。

设计算法：解释如图所示。

5-1.cpp

```cpp
#include <stdio.h>
#define N 10000
int main()
{
    int a[N],n;
    scanf("%d",&n);
    for (int i=0; i<n; i++)
        scanf("%d",&a[i]);
    int m;
    scanf("%d",&m);
    while (m--)
    {
        int x;
        scanf("%d",&x);
        int low =0, high =n-1,mid;    // 置区间初值
        while (low<=high)
        {
            mid = (low+high)/2 ;
            if (x == a[mid])   break;           // 找到待查记录
            else if (x<a[mid])  high=mid-1;    // 继续在前半区间进行检索
            else  low=mid+1;                    // 继续在后半区间进行检索
        }
        if (low<=high)          // 找到待查记录
            printf("%d在下标位%d的位置\n",x,mid);
        else
            printf("未找到！\n");
    }
    return 0;
}
```

5-2 设二叉排序树采用二叉链表存储结构：

```
typedef struct BiTnode
{
    KeyType key;    //关键字域
    ElemType *otherinfo;    //其它数据项(可以忽略)
    struct BiTnode *Lchild;    //左指针域
    struct BiTnode *Rchild;    //右指针域
} BiTnode, *BiTree;
```

试设计二叉排序树的查找算法、插入算法和删除算法。

二叉排序树特点

1）若左子树不为空，左子树上所有结点的值均小于或等于它的根节点的值

2）若右子树不为空，右子树上所有结点的值均大于或等于它的根节点的值

3）左右子树也分别为二叉排序树

二叉排序算法的复杂度：

时间复杂度：二分查找的思想，查找次数为二叉查找树的高度，若树为平衡二叉树则为 O(logn),否则最坏的情况为右斜树 O(n)

二叉排序算法的缺点是：

二叉树的构建类型多种，不同的二叉树形状会导致查找的性能差异很大，例如普通的二叉树和一棵右斜树。

二叉排序树的查找：

1）查找数据 key，判断 key 是否等于树的根节点数据

2）若待查数据 key 小于根结点数据则递归的在左子树查找

3）若待查数据 key 大于根结点数据则递归的在右子树查找

定义结点结构

```
typedef struct BiTNode
{
    int data;  //结点数据
    struct BiTNode *lchild, *rchild;//左右孩子指针
}BiTNode,*BiTree
```

递归查找二叉排序树 T 中是否存在 key

```
//f 指向 T 的双亲
//p 获得查找到的结点位置
Status SearchBST(BiTree T,int key,BiTree f,BiTree *p)
{
    //查找不成功
     if(!T)//判断当前二叉树是否到叶子结点
      {
         *p=f;//指针 p 指向查找路径上访问的最后一个结点并返回 false
         return False;
      }
    //查找成功
    else if(key==T->data)
     {
        *p=T;
```

```
20.       return True;
21.    }
22.   else if(key<T->data)//待查找元素小于结点数据
23.       return SearchBST(T->lchild, key, T, p);//在左子树继续查找
24.   else
25.       return SearchBST(T->rchild, key, T, p);//在右子树继续查找
26.
```

二叉排序树的插入操作：

1）在二叉排序树找不到待插入的数据 key 则执行 2）步骤

2）待插数据初始化为结点 s，若树为空则直接赋值结点 s 给树

3）待插入数据 key 小于根结点数据则插入为左孩子

4）待插入数据 key 大于根结点数据则插入为右孩子

```
1. Status InsertBST(BiTree *T,int key)
2. {
3.   BiTree p,s;//创建二叉树结点
4.   //在二叉排序树中找不到 key
5.   if(!SearchBST(*T,key,NULL,&p))
6.     {
7.         //s 结点的初始化
8.         s=(BiTree)malloc(sizeof(BiTNode));
9.         s->data=key;
10.           s->lchild=s->rchild=NULL;
11.           //若 p 结点为空
12.           if(!p)
13.                *T=s;
14.           else if (key<p->data)//待插入的值 key 小于 p 结点指向的
    数据
15.                p->lchild=s;//s 插入为左孩子
16.           else//待插入的值 key 大于 p 结点指向的数据
17.               p->rchild=s;//s 插入为右孩子
18.          return True;
19.      }
20.    //树中已有关键字相同的结点，不再插入
21.     else
22.        return False;
23.
24.   }
```

二叉排序树的删除操作：

删除结点的三种情况：

1）删除叶子结点

2）删除的结点只有左或右子树的

3）删除的结点有左右子树

```
1. //删除元素等于 key 的数据结点
```

```
2. Status DeleteBST(BiTree *T,int key)
3. {
4.  //不存在关键字等于key 的数据元素
5.    if(!*T)
6.       return False;
7.   else
8.   {
9.       if(key==(*T)->data)     //找到关键字等于key 的数据元素
10.                return Delete(T);
11.         else if(key<(*T)->data)//待删除的元素key 小于查找到的元素---则在结点的左子树搜索
12.                return DeleteBST(&(*T)->lchild,key);
13.         else                                //待删除的元素key 大于查找到的元素---则在结点的右子树搜索
14.                return DeleteBST(&(*T)->rchild,key);
15.     }
```

```
1. Status Delete(BiTree *p)
2. {
3.     BiTree q,s;
4.    //第一种情况，删除结点只有左子树或右子树
5.     if((*p)->rchild==NULL)//只有左子树
6.      {
7.           q=*p;
8.           *p=(*p)->lchild;
9.           free(q);
10.        }
11.       else if((*p)->lchild==NULL)//只有右子树
12.        {
13.           q=*p;
14.           *p=(*p)->rchild;
15.           free(q);
16.       }
17.     //第二种情况：删除的结点有左子树和右子树
18.     else
19.       {
20.        q=*p;//待删除的结点给临时变量q
21.        s=(*p)->lchild;//待删除结点指向的左子树给临时变量s
22.        while(s->rchild)//左子树s 一直向右找，直到找到待删结点的前驱
23.          {
24.            q=s;
25.            s=s->rchild;
26.          }
```

```
27.       (*p)->data=s->data;//s 指向被删结点的直接前驱，将它的值直接
   赋值给要删除的结点*p
28.
29.       if(q!=*p)//被删结点的直接前驱 p!=被删结点的直接前驱的根结点 q
30.          q->rchild=s->lchild;//根结点 q 的右孩子指针指向被删结点的
   直接前驱的左孩子
31.       else
32.           q->lchild=s->lchild;
33.      free(s);
34.       }
35.
36.    }
```

5-3 哈希表设计。为班级 30 个人的姓氏(单字姓)设计一个哈希表，假设姓氏用汉语拼音表示。要求用除留取余法构造哈希函数，用线性探测再散列法处理冲突，平均查找长度的上限为 2。

算法设计：

1、将姓名表各个名字得 ASCII 码相加求和。

2、创建哈希表，将 ASCII 码取余得 KEY 值，若未发生冲突存入哈希表

3、发生冲突调用冲突函数。进行线性探测。最后存入哈希表。

```
#define HASH_SIZE 50//哈希表的长度
#define Name_SIZE 30//名字表长度
#define R 49//小于哈希表长度的 R
//int i,key;
struct Name {
    char *name;              //姓名
    int ascii;               //对应的 ascii 码和
};
struct hash {
    char *name;              //姓名
    int ascii;               //对应 ASCII 码和
    int s;                   //查找长度
};
Name NameList [Name_SIZE];
hash hashtable [HASH_SIZE];
void init_Namelistlist ( );        //初始化姓名表
void C_hashtable ( ) ;         //创建 hash 表
void collison (int i);           //冲突函数,第 i 个姓名表发生冲突
void print_Namelist ( );
void print_hash ( );             //打印函数
#include<stdio.h>
#include<ctype.h>              //toascii 函数
void init_Namelist( ) {
```

```
    NameList[0].name="zhang";
    NameList[1].name="li";
    NameList[2].name="wang";
    NameList[3].name="huang";
    NameList[4].name="tie";
    NameList[5].name="chen";
    NameList[6].name="xu";
    NameList[7].name="zhou";
    NameList[8].name="tang";
    NameList[9].name="xia";
    NameList[10].name="hong";
    NameList[11].name="sha";
    NameList[12].name="da";
    NameList[13].name="yu";
    NameList[14].name="sao";
    NameList[15].name="yang";
    NameList[16].name="heng";
    NameList[17].name="feng";
    NameList[18].name="fen";
    NameList[19].name="zhi";
    NameList[20].name="lin";
    NameList[21].name="liu";
    NameList[22].name="tan";
    NameList[23].name="gong";
    NameList[24].name="hao";
    NameList[25].name="hua";
    NameList[26].name="shu";
    NameList[27].name="cheng";
    NameList[28].name="hang";
    NameList[29].name="wen";
    int i,j;
    for(i=0; i<Name_SIZE; i++) {
        for(j=0; (*(NameList[i].name+j))!='\0'; j++)
            NameList[i].ascii+=toascii(*(NameList[i].name+j));          //ascii
码求和
        NameList[i].ascii*=3;//扩大值
    }
}
void collison (int i) {
    int key,flag;
    flag=0;                                                              //未探测
至末尾
    key=(NameList[i].ascii)%R;
```

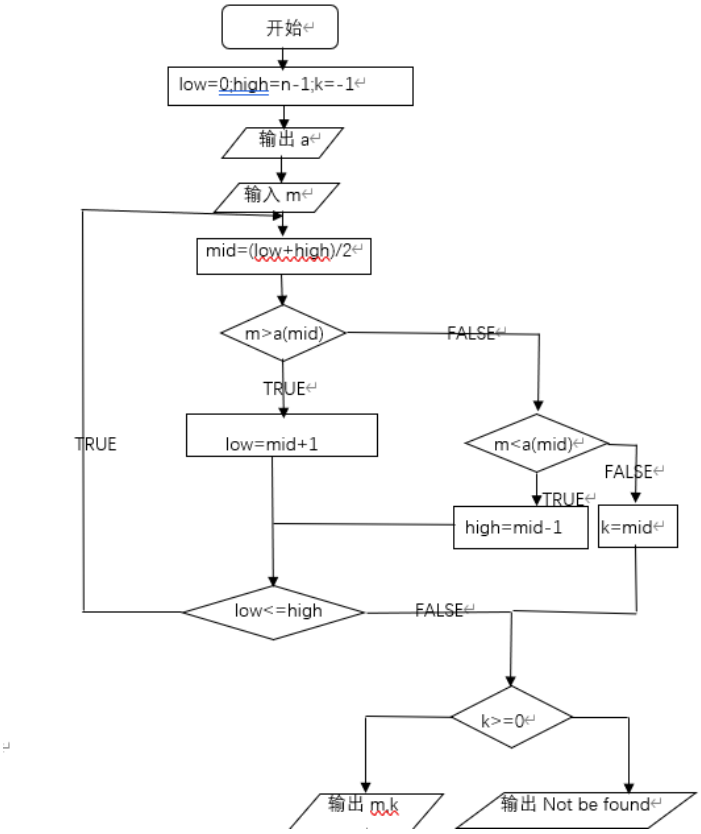
```
    while(hashtable[key].s != 0) {
        key=key+1;
        //   printf("%d",key);                                          //线
性探测每次加 1
        if(key==HASH_SIZE-1) {                                          //探测
至哈希表末端
            key=0;
            flag=1;                                                     //探测至
末尾标识
        }
    }
    if(hashtable[key].s==0) {
        hashtable[key].name=NameList[i].name;
        hashtable[key].ascii=NameList[i].ascii;
        if(flag==0)
            hashtable[key].s= (key-(NameList[i].ascii%R))+1 ;
        else
            hashtable[key].s= (HASH_SIZE-(NameList[i].ascii%R))+key+1;    //查
找                                  次                                  数
//查找次数
    }
}
void C_hashtable() {                                                    //创建哈希函数
    int i,key;
    for(i=0; i<HASH_SIZE; i++) {
        hashtable[i].name="\0";
        hashtable[i].ascii=0;
        hashtable[i].s=0;                                               //初始化哈希表
    }
    for(i=0; i<Name_SIZE; i++) {
        key=(NameList[i].ascii)%R;                                      //除留余数法
        if(hashtable[key].s == 0 ) {                                    //未发生冲突
            hashtable[key].name=NameList[i].name;
            hashtable[key].ascii=NameList[i].ascii;
            hashtable[key].s=1;
        } else                                                          //发生冲突
            collison ( i );                                             //调用冲突函数

    }

}
void show() {
    printf("*****************************Hashtable_creat*************************
*\n");
```

```
    printf("please input \n A:print Namelist \n B:print the hash table \n
C:exit\n");
}
void print_Namelist ( ) {
    int i;
    for(i=0; i<Name_SIZE; i++) {

    printf("number:%d\tname:%s\tascii:%d\n",i,NameList[i].name,NameList[i].a
scii);
    }
}
void print_hash ( ) {
    int i;
    float ASL=0.0;                              //平均查找长度
    for(i=0; i<HASH_SIZE; i++) {

    printf("number:%d\tname:%s\tascii:%d\t%d\n",i,hashtable[i].name,hashta
ble[i].ascii,hashtable[i].s);
        ASL+=hashtable[i].s;
    }
    ASL=ASL/Name_SIZE;
    printf("ASL:%f\n",ASL);
}
int main() {
    char c;
    while(1) {
        show( );
        init_Namelist( );
        C_hashtable( );
        printf ("please in put the order: \n");
        scanf("%c",&c);
        getchar( );
        switch(c) {
            case 'A':
                print_Namelist ( );
                break;
            case 'B' :
                print_hash ( );
                break;
            case 'C' :
                break;
                // default: printf("EROOR\n");break;
        }
    }
```

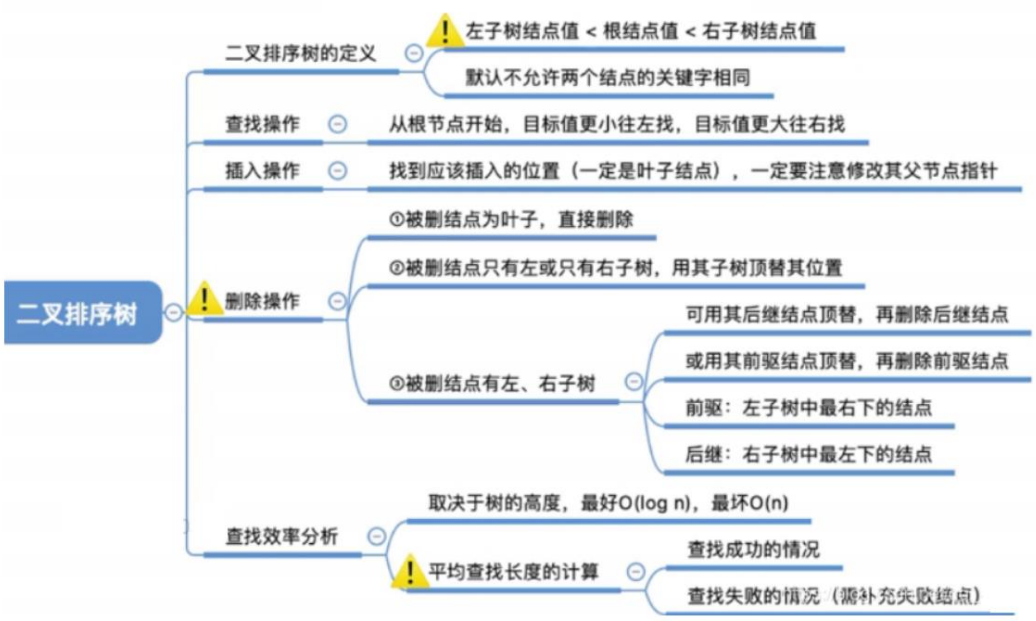
```
    return 0;
}
```

三、 主要算法流程图

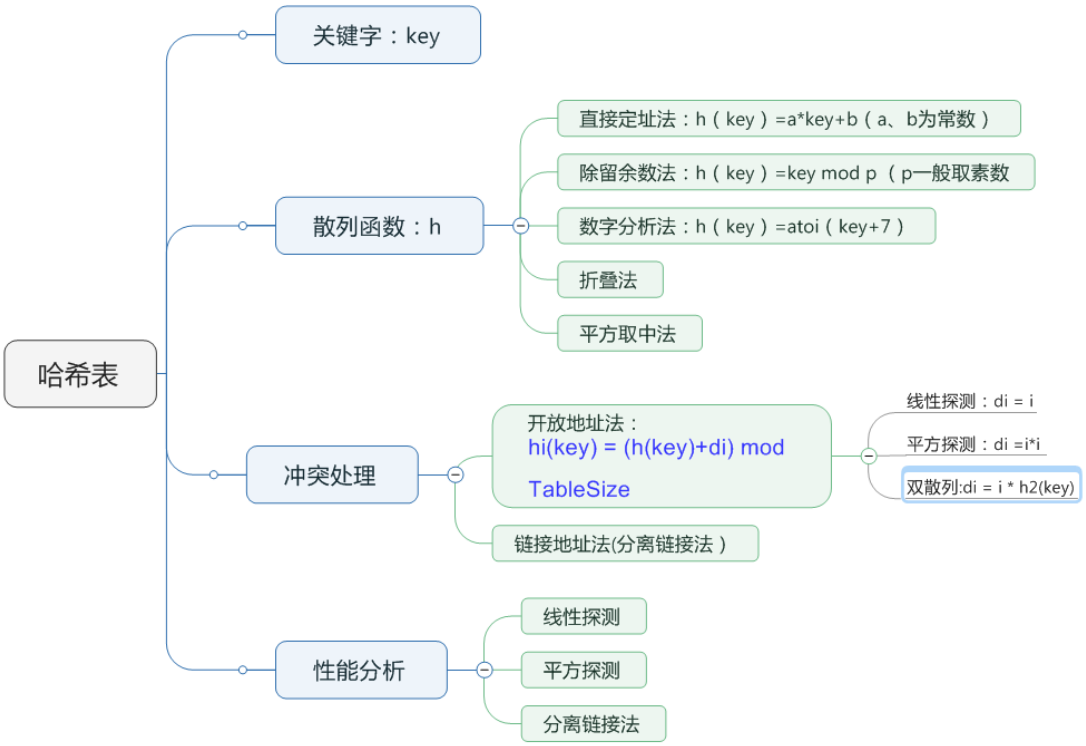
5-1 算法流程

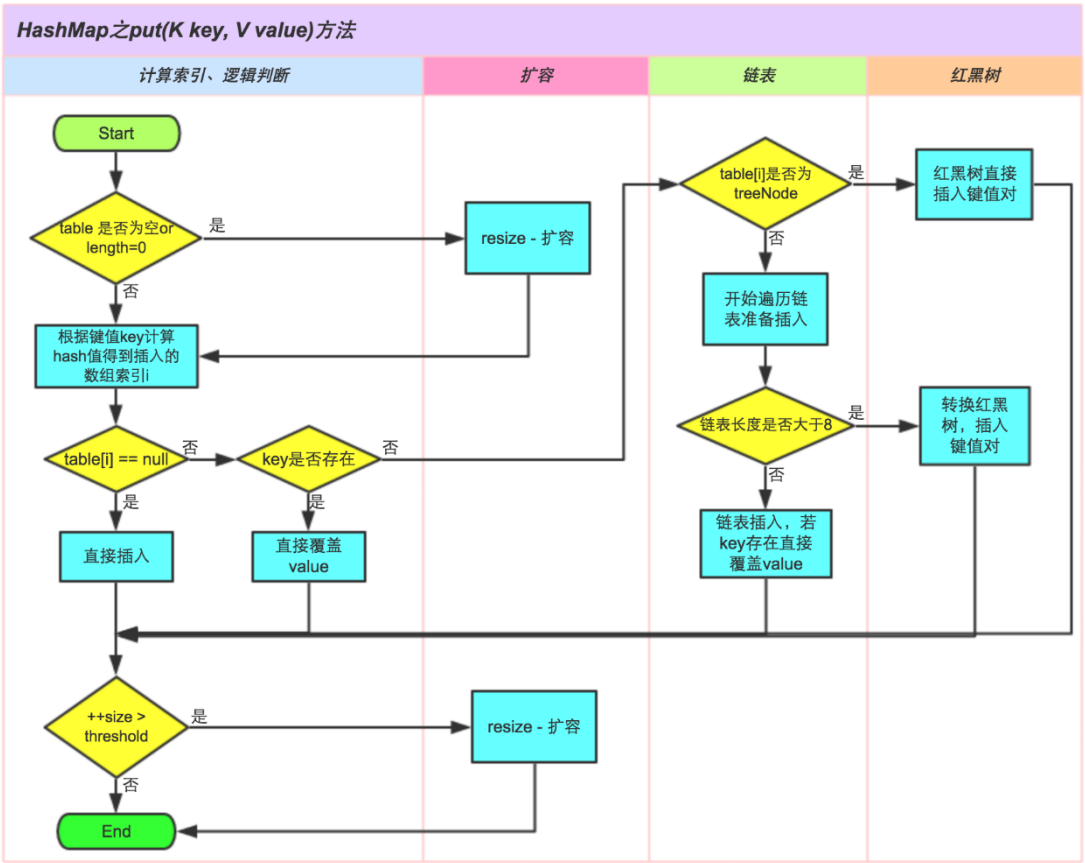


5-2 算法流程



5-3 算法流程

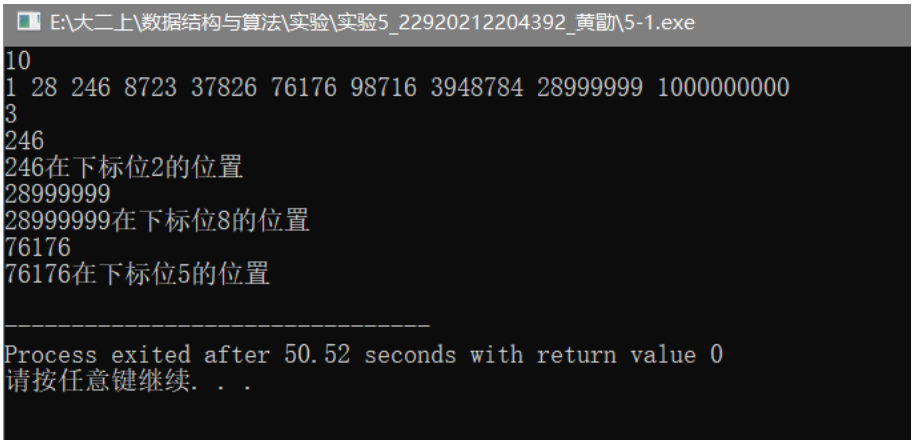




四、 实验结果：

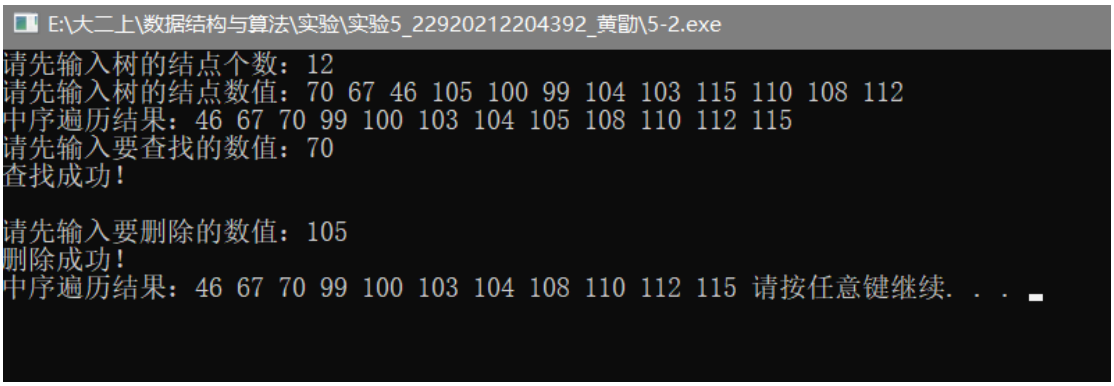
（结合截图说明算法的输入输出）

1、关于 5-1 的输入与输出：



在实际运行中，我创建了如上的数组，并查找到了正确的下标。

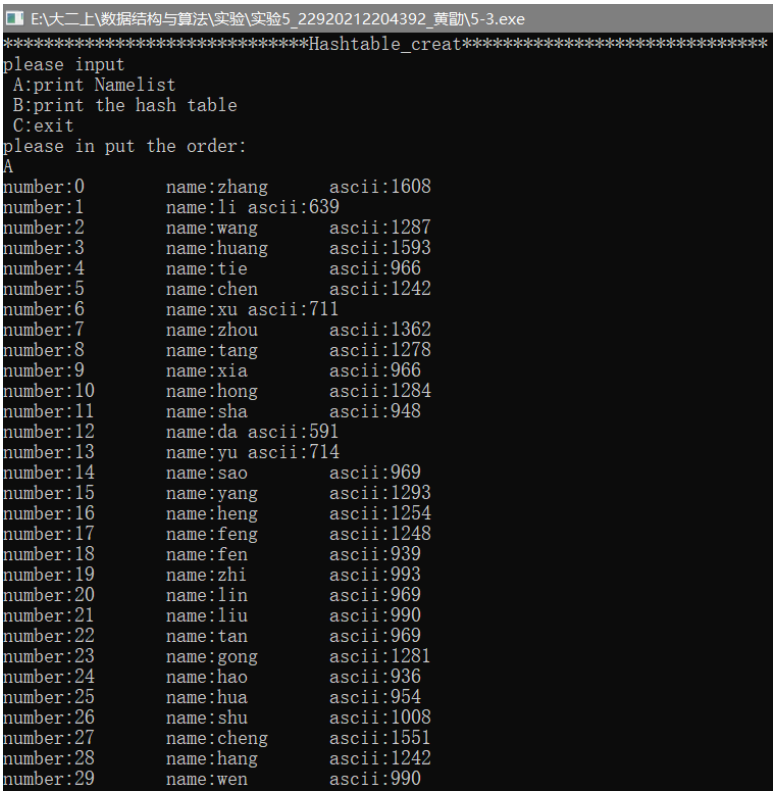
2、关于 5-2 的输入与输出



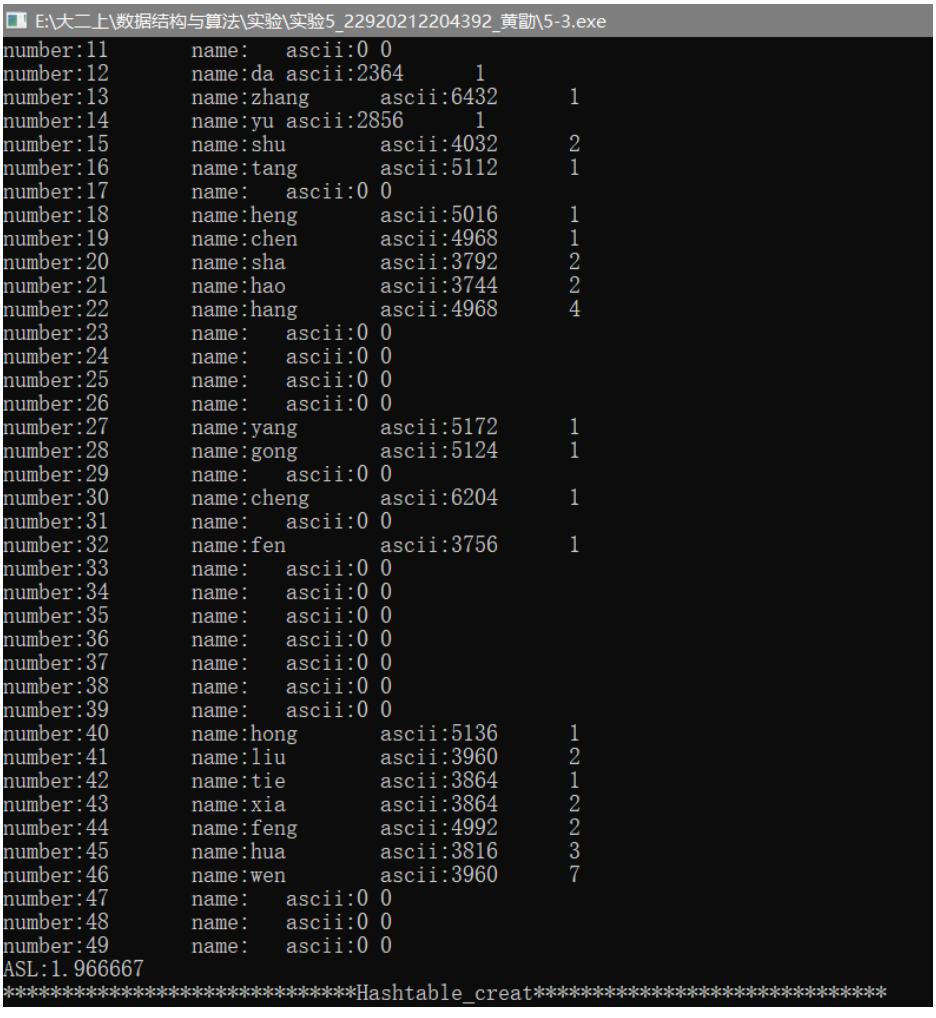
在实际运行中，我利用如上的数列创建了二叉排序树，并找到了正确的数值，实现了删除操作。

3、关于 5-3 的输入与输出

我创建了如下的姓氏表：



最后得到平均查找长度为 1.9667，满足要求!



五、 实验小结（即总结本次实验所得到的经验与启发等）：

在本次实验中，我尝试具体运用了折半查找、二叉排序树与散列表，在实体机的实验中我能够更深刻地理解对这一部分数据结构的执行方式与特点，并且在编写代码的过程中，我通过不断的调试去寻找语句之间的问题和不足，在潜移默化中提高了我的代码编写能力，这是一次完成效果良好的实验！