实验报告（week-11）

2020111235 马靖淳

一．实验任务

1. 顺序查找
2. 折半查找
3. 二叉排序树
4. 哈希表

二．实验上机时间

三．知识点

1. 顺序查找
   （1） 顺序表的定义类型
   （2） 顺序查找表 有无监视哨
2. 折半查找
   （1） 折半查找的非递归算法
   （2） 折半查找的递归算法
3. 二叉排序树
   （1） 二叉排序树的存储结构
   （2） 二叉排序树的查找算法
   （3） 二叉排序树的插入算法
   （4） 二叉排序树的构造
   （5） 二叉排序树的删除
4. 哈希表
   （1） 哈希表的查找
   （2） 哈希表的插入

四．源代码及结果截屏

1. 顺序查找

a. SequentialSearch.h

```
#ifndef SEQUENTIALSEARCH_H
#define SEQUENTIALSEARCH_H

#include <stdlib.h>
#include "Status.h"

#define MAX_SIZE 20

typedef int KeyType;
typedef struct {
    KeyType key;
    char name;
}ElemType;

typedef struct
{
    ElemType* elem;//数据元素空间基址
```

```
    int length;//表的长度
}SSTable;

Status CreateST(SSTable* ST);
int Search_Seq1(SSTable ST, KeyType key);//无监视哨
int Search_Seq2(SSTable ST, KeyType key);//有监视哨

#endif
```

b. SequentialSearch.c

```
#define _CRT_SECURE_NO_WARNINGS
#include "SequentialSearch.h"
#include "Status.h"
#include <stdio.h>
#include <malloc.h>

Status CreateST(SSTable *ST)
{
    int i, n;
    printf("输入顺序查找表的长度:");
    scanf("%d", &n);
    getchar();
    (*ST).length = n + 1;
    (*ST).elem = (ElemType*)malloc((n+1)*sizeof(ElemType));
    printf("输入每个关键字的信息：\n");
    for (i = 1; i < (*ST).length; i++)
    {
        printf("输入第%d个关键字的信息：", i);
        scanf("%d", &(*ST).elem[i].key);//关键字信息
        getchar();
    }
    return OK;
}//CreateST

int Search_Seq1(SSTable ST, KeyType key)//无监视哨
{
    int i = ST.length;
    while (i > 0 && ST.elem[i].key != key)
        i--;
    return i;
}

int Search_Seq2(SSTable ST, KeyType key)//有监视哨
{
```

```
    int i = ST.length;
    ST.elem[0].key = key;//监视哨
    while (ST.elem[i].key != key)
        i--;
    return i;
}
```

c. test.c

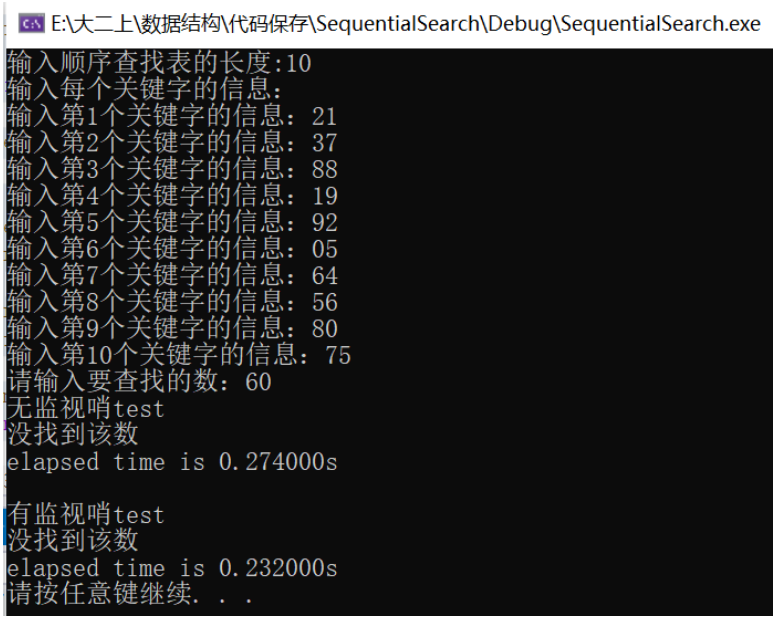
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include "SequentialSearch.h"
#include <time.h>

int main()
{
    SSTable ST;
    CreateST(&ST);
    int search_num;

    clock_t tic, toc;
    int nn = 10000000;

    printf("请输入要查找的数：");
    scanf("%d", &search_num);



    printf("无监视哨test\n");
    tic = clock();
    for (int i = 1; i <= nn; i++)
    {
        Search_Seq1(ST, search_num);
    }
    toc = clock();
    if (Search_Seq1(ST, search_num))
        printf("所查找数位于第%d位\n", Search_Seq1(ST, search_num));
    else
        printf("没找到该数\n");
    printf("elapsed time is %fs\n", (double)(toc - tic) / CLOCKS_PER_SEC);
    printf("\n");

    printf("有监视哨test\n");
    tic = clock();
    for (int i = 1; i <= nn; i++)
```

```
    {
        Search_Seq2(ST, search_num);
    }
    toc = clock();
    if (Search_Seq2(ST, search_num))
        printf("所查找数位于第%d位\n", Search_Seq1(ST, search_num));
    else
        printf("没找到该数\n");
    printf("elapsed time is %fs\n", (double)(toc - tic) / CLOCKS_PER_SEC);

    system("pause");
    return 0;
}
```

能查找到

```
E:\大二上\数据结构\代码保存\SequentialSearch\Debug\SequentialSearch.exe
输入顺序查找表的长度:10
输入每个关键字的信息:
输入第1个关键字的信息: 21
输入第2个关键字的信息: 37
输入第3个关键字的信息: 88
输入第4个关键字的信息: 19
输入第5个关键字的信息: 92
输入第6个关键字的信息: 05
输入第7个关键字的信息: 64
输入第8个关键字的信息: 56
输入第9个关键字的信息: 80
输入第10个关键字的信息: 75
请输入要查找的数: 64
无监视哨test
所查找数位于第7位
elapsed time is 0.200000s

有监视哨test
所查找数位于第7位
elapsed time is 0.185000s
请按任意键继续. . .
```

查找不到



2. 折半查找

a. SequentialSearch.h

b. SequentialSearch.c

```
int Search_Bin1(SSTable ST, KeyType key)//折半查找的非递归算法
{
    //在有序表ST中折半查找其关键字等于key的数据元素
    //若找到，则函数值为该元素在表中的位置，否则为0
    int low, high, mid;
    low = 1;
    high = ST.length;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (ST.elem[mid].key == key)
            return mid;//找到待查元素
        else if (ST.elem[mid].key > key)
            high = mid - 1;//继续在前半区间进行查找
        else
            low = mid + 1;//继续在后半区间进行查找
    }
    return 0;
}//Search_Bin
//时间复杂度O（log n）

int Search_Bin2(SSTable ST, int low, int high, KeyType key)
{
    //递归实现
    int mid = 0;
    if (low > high)
```

```
        return 0;
    else
    {
        mid = (low + high) / 2;
        if (ST.elem[mid].key > key)
            return Search_Bin2(ST, low, mid - 1, key);
        else if (ST.elem[mid].key < key)
            return Search_Bin2(ST, mid + 1, high, key);
        else
            return mid;
    }
}//Search_Bin2
//时间复杂度O（log n)
```

c. test.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include "SequentialSearch.h"
#include <time.h>

int main()
{
    SSTable ST;
    CreateST(&ST);
    int search_num;

    clock_t tic, toc;
    int nn = 10000000;

    printf("请输入要查找的数：");
    scanf("%d", &search_num);


    printf("折半查找非递归test\n");
    tic = clock();
    for (int i = 1; i <= nn; i++)
    {
        Search_Seq1(ST, search_num);
    }
    toc = clock();
    if (Search_Bin1(ST, search_num))
        printf("所查找数位于第%d位\n", Search_Bin1(ST, search_num));
    else
```

```
        printf("没找到该数\n");
    printf("elapsed time is %fs\n", (double)(toc - tic) / CLOCKS_PER_SEC);
    printf("\n");

    printf("折半查找递归test\n");
    tic = clock();
    for (int i = 1; i <= nn; i++)
    {
        Search_Seq1(ST, search_num);
    }
    toc = clock();
    if (Search_Bin2(ST, 1, ST.length-1, search_num))
        printf("所查找数位于第%d位\n", Search_Bin2(ST, 1, ST.length - 1, search_num));
    else
        printf("没找到该数\n");
    printf("elapsed time is %fs\n", (double)(toc - tic) / CLOCKS_PER_SEC);
    printf("\n");


    system("pause");
    return 0;
}
```

- 有序查找表

i	1	2	3	4	5	6	7	8	9	10	11
ST.elem	05	13	19	21	37	56	64	75	80	88	90

```
E:\大二上\数据结构\代码保存\SequentialSearch\Debug\SequentialSearch.exe
输入顺序查找表的长度:11
输入每个关键字的信息:
输入第1个关键字的信息: 05
输入第2个关键字的信息: 13
输入第3个关键字的信息: 19
输入第4个关键字的信息: 21
输入第5个关键字的信息: 37
输入第6个关键字的信息: 56
输入第7个关键字的信息: 64
输入第8个关键字的信息: 75
输入第9个关键字的信息: 80
输入第10个关键字的信息: 88
输入第11个关键字的信息: 90
请输入要查找的数: 21
折半查找非递归test
所查找数位于第4位
elapsed time is 0.227000s

折半查找递归test
所查找数位于第4位
elapsed time is 0.217000s

请按任意键继续. . .
```

3. 二叉排序树

a. BinarySortTree.h

```
#ifndef SEQUENTIALSEARCH_H
#define SEQUENTIALSEARCH_H

#include <stdlib.h>
#include "Status.h"

typedef int KeyType;
typedef char OtherInfoType;
typedef int TElemType;
typedef struct
{
    KeyType key;
    OtherInfoType info;
}ElemType;

typedef struct BiTNode
{
    ElemType data;
    struct BiTNode* lchild, * rchild;
}BiTNode, * BiTree;

void InitBiTree(BiTree* T);
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree* p);
Status InsertBST(BiTree* T, ElemType e);
Status Visit(TElemType e);
```

```
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType));
Status DeleteBST(BiTree* T, KeyType key);
void Delete(BiTree* p);

#endif
```

b. BinarySortTree.c

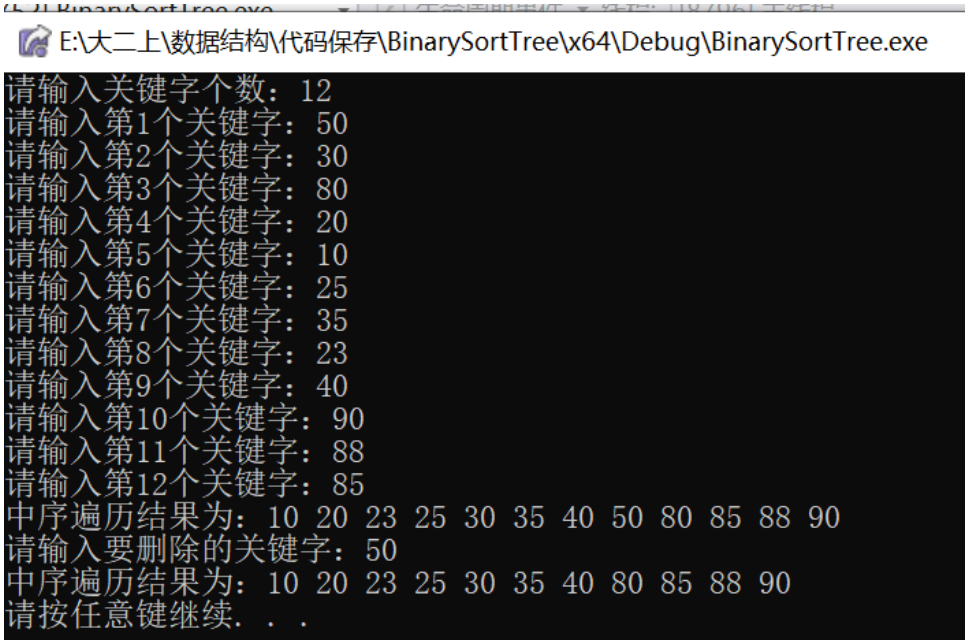
```
#define _CRT_SECURE_NO_WARNINGS
#include "BinarySortTree.h"
#include "Status.h"
#include <stdio.h>
#include <malloc.h>

void InitBiTree(BiTree* T)
{
    *T = NULL;
}

Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree* p)
{
    if (!T)//查找不成功
    {
        *p = f;
        return FALSE;
    }
    else if (key == T->data.key)//查找成功
    {
        *p = T;
        return TRUE;
    }
    else if (key < T->data.key)
        return SearchBST(T->lchild, key, T, &(*p));
    else
        return SearchBST(T->rchild, key, T, &(*p));
}

Status InsertBST(BiTree* T, ElemType e)
{
    BiTree p,s;
    if (!SearchBST(*T, e.key, NULL, &p))
    {
        s=(BiTree)malloc(sizeof(BiTNode));//新节点s
        s->data = e;
        s->lchild = s->rchild = NULL;
```

```
        if (!p)
            *T = s;//插入s为新的根节点
        else if (e.key < p->data.key)
            p->lchild = s;//插入*s为*p的左孩子
        else
            p->rchild = s;//插入*s为*p的右孩子
        return TRUE;
    }
    return FALSE;//树中已有结点，不再插入
}//InsertBST

Status Visit(TElemType e)
{
    printf("%d ", e);
    return OK;
}

Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，中序遍历二叉树T的递归算法
    if (T)
    {
        if (InOrderTraverse(T->lchild, Visit))
            if (Visit(T->data.key))
                if (InOrderTraverse(T->rchild, Visit))
                    return OK;
        return ERROR;
    }
    else
        return OK;
}//InOrderTraverse

Status DeleteBST(BiTree* T, KeyType key)
{
    if (!(*T))
        return FALSE;//不存在关键字等于key的数据元素
    else
    {
        if (key == (*T)->data.key)
        {
            Delete(&(*T));
            return TRUE;
        }
        else if (key < (*T)->data.key)
```

```
            DeleteBST(&((*T)->lchild), key);
        else
            DeleteBST(&((*T)->rchild), key);
    }
}//DeleteBST

void Delete(BiTree* p)
{
    //从二叉排序树中删除结点p，并重接它的左子树或右子树
    BiTree q, s;
    if (!(*p)->rchild)
    {
        //右子树为空树则只需重接它的左子树
        q = (*p);
        (*p) = (*p)->lchild;
        free(q);
    }
    else if (!(*p)->lchild)
    {
        //左子树为空树只需重接它的右子树
        q = (*p);
        (*p) = (*p)->rchild;
        free(q);
    }
    else
    {
        //左右子树都不空
        q = (*p);
        s = (*p)->lchild;
        while (s->rchild)
        {
            //s指向被删结点的前驱
            q = s;
            s = s->rchild;
        }
        (*p)->data = s->data;
        if (q != p)
            q->rchild = s->lchild;
        else
            q->lchild = s->lchild;//重接*q的左子树
        free(s);
    }
}//Delete
```

c. test.c

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include "BinarySortTree.h"

int main()
{
    BiTree T, f, p;
    int num = 0;
    KeyType key;
    ElemType e;

    InitBiTree(&T);
    printf("请输入关键字个数：");
    scanf("%d", &num);
    getchar();
    for (int i = 0; i < num; ++i)
    {
        printf("请输入第%d个关键字：", i + 1);
        scanf("%d", &key);
        getchar();
        e.key = key;
        InsertBST(&T, e);
    }

    //InOrderTraverse test
    printf("中序遍历结果为：");
    InOrderTraverse(T, Visit);
    printf("\n");

    printf("请输入要删除的关键字：");
    scanf("%d", &key);
    getchar();
    DeleteBST(&T, key);

    //InOrderTraverse test
    printf("中序遍历结果为：");
    InOrderTraverse(T, Visit);
    printf("\n");


    system("pause");
    return 0;
}
```



4. 哈希表

a. HashSearch.h

```
#ifndef HASHSEARCH_H
#define HASHSEARCH_H

#include <stdlib.h>
#include "Status.h"

#define N 10

typedef int KeyType;
typedef struct{
    KeyType key;
}ElemType;
typedef struct{
    ElemType* elem;//数据元素存储基址，动态分配数组
    int count;//当前数据元素个数
    int sizeindex;//hashsize[sizeindex]为当前容量
}HashTable;

Status InitHash(HashTable* H);
void DestroyHash(HashTable* H);
void collision(int* p, int d);
Status SearchHash(HashTable H, KeyType K, int* p, int* c);
Status InsertHash(HashTable* H, ElemType e);
void RecreateHashTable(HashTable* H);
void TraverseHash(HashTable H, void(*Vi)(int, ElemType));
void print(int p, ElemType r);
```

```
#endif
```

b. HashSearch.c

```
#define _CRT_SECURE_NO_WARNINGS
#include "HashSearch.h"
#include "Status.h"
#include <stdio.h>
#include <malloc.h>

int m = 0;
int hashsize[] =
{ 11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,12
7,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,223,227,229,233,239,2
41,251,257,263,269,271,277,281,283,293,307,311,313,317,331,337,347,349,353,359,367,373,
379,383,389,397,401,409,419,421,431,433,439,443,449,457,461,463,467,479,487,491,499,503
,509,521,523,541,547,557,563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,64
7,653,659,661,673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,8
09,811,821,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,941,947,
953,967,971,977,983,991,997 };
//哈希表容量递增表，一个合适的素数序列

Status InitHash(HashTable* H)
{
    (*H).count = 0;
    (*H).sizeindex = 0;
    m = hashsize[0];
    (*H).elem = (ElemType*)malloc(m * sizeof(ElemType));
    if (!(*H).elem)
        return OVERFLOW;
    for (int i = 0; i < m; i++)
    {
        (*H).elem[i].key = NULLKEY;
    }
    return OK;
}//初始化哈希表

void DestroyHash(HashTable* H)
{
    free((*H).elem);
    (*H).elem = NULL;
    (*H).count = 0;
    (*H).sizeindex = 0;
}//销毁哈希表
```

```
unsigned Hash(KeyType K)
{
    return K % m;
}

void collision(int* p, int d)
{
    *p = (*p + d) % m;
}//处理冲突函数

Status SearchHash(HashTable H, KeyType K, int* p, int* c)
{
    //在开放定址哈希表H中查找关键码为K的元素，若查找成功，以p指示待查数据元素在表中的位置；
    //否则，以p指示插入位置。c用以计冲突次数，其初置值零，供建表插入时参考
    *p = Hash(K);//求得哈希地址
    while (H.elem[*p].key != NULL && !(K == H.elem[*p].key))
    {
        (*c)++;
        if (*c < m)
            collision(p, *c);//求得下一探查地址p
        else
            break;
    }
    if (K == H.elem[*p].key)
        return SUCCESS;//查找成功，返回待查数据元素位置p
    else
        return UNSUCCESS;//查找不成功
}//SearchHash


Status InsertHash(HashTable* H, ElemType e)
{
    //查找不成功时插入数据元素e到开放定址哈希表H中；若冲突次数过大，则重建哈希表
    int c = 0;
    int p;
    if (SearchHash(*H, e.key, &p, &c))
        return DUPLICATE;//表中已有与e有关键字的元素
    else if (c < hashsize[(*H).sizeindex] / 2)
    {
        //冲突次数c未达到上限（阈值c可调）
        (*H).elem[p] = e;
        ++(*H).count;
        return OK;//插入e
```

```
    }
    else
    {
        RecreateHashTable(H);//重建哈希表
        return UNSUCCESS;
    }
}//InsertHash

void RecreateHashTable(HashTable* H) //重建哈希表
{
    int i = (*H).count;
    int count = (*H).count;
    ElemType* p = (ElemType*)malloc(count * sizeof(ElemType));
    ElemType* elem = (ElemType*)malloc(count * sizeof(ElemType));
    p = elem;
    printf("重建哈希表\n");
    for (i = 0; i < m; i++)
        if (((*H).elem + i)->key != NULLKEY)
            *p++ = *((*H).elem + i);
    (*H).count = 0;
    (*H).sizeindex++;//增大存储容量
    m = hashsize[(*H).sizeindex];
    p = (ElemType*)realloc((*H).elem, m * sizeof(ElemType));
    if (!p)
        exit(0);
    (*H).elem = p;
    for (i = 0; i < m; i++)
        (*H).elem[i].key = NULLKEY;
    for (p = elem; p < elem + count; p++)
        InsertHash(H, *p);
}//RecreateHashTable


void TraverseHash(HashTable H, void(*Vi)(int, ElemType)) //遍历函数
{
    int i;
    printf("哈希地址0~%d\n", m - 1);
    for (i = 0; i < m; i++)
        if (H.elem[i].key != NULLKEY)
            Vi(i, H.elem[i]);
}//TraverseHash

int Find(HashTable H, KeyType K, int* p)
{
```

```
    int c = 0;
    *p = Hash(K);//求得哈希地址
    while (H.elem[*p].key != NULLKEY && !(K == H.elem[*p].key))
    { //该位置中填有记录，并且关键字不相等
        c++;
        if (c < m)
            collision(p, c);//求得下一探查地址p
        else
            return UNSUCCESS; //查找不成功(H.elem[p].key==NULLKEY)
    }
    if (K == H.elem[*p].key)
        return SUCCESS; //查找成功，p返回待查数据元素位置
    else
        return UNSUCCESS; //查找不成功(H.elem[p].key==NULLKEY)
}

void print(int p, ElemType r)//打印函数
{
    printf("%d %d\n", r.key, p);
}
```

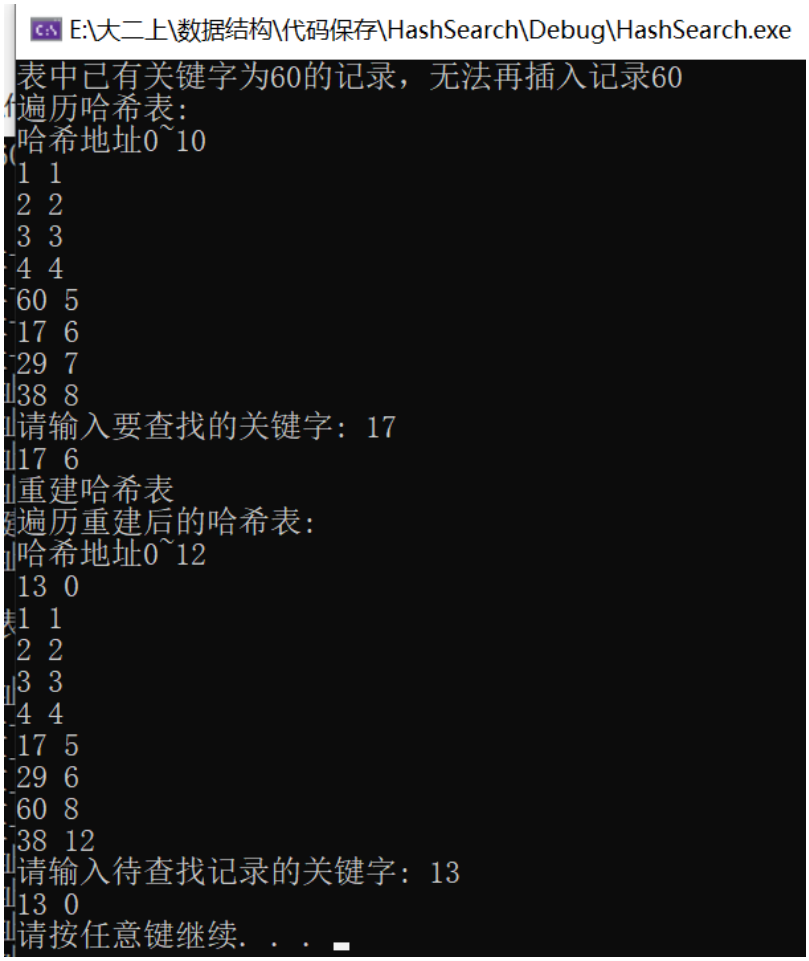
c. Status.h

```
#ifndef     STATUS_H
#define  STATUS_H

#define  TRUE         1
#define  FALSE        0
#define      YES      1
#define      NO           0
#define  OK           1
#define  ERROR        0
#define      SUCCESS  1
#define      UNSUCCESS    0
#define  INFEASIBLE   -1
#define  OVERFLOW     -2
#define      UNDERFLOW    -3
#define      DUPLICATE    -1
#define      NULLKEY      0

/* 状态码类型 */
typedef int Status;
#endif
```

d. test.c

```
#define _CRT_SECURE_NO_WARNINGS
```

```c
#include <stdio.h>
#include "HashSearch.h"


int main()
{
    ElemType r[N] = { 17,60,29,38,1,2,3,4,60,13 };
    HashTable H;
    int i, j, p;
    KeyType K;
    InitHash(&H);
    for (i = 0; i < N - 1; i++)
    {
        //插入前N-1个记录
        j = InsertHash(&H, r[i]);
        if (j == DUPLICATE)
            printf("表中已有关键字为%d的记录，无法再插入记录%d\n", r[i].key, r[i].key);

    }//for
    printf("遍历哈希表:\n");
    TraverseHash(H, print);
    printf("请输入要查找的关键字: ");
    scanf("%d", &K);
    j = Find(H, K, &p);
    if (j == SUCCESS)
        print(p, H.elem[p]);
    else
        printf("没找到该关键字\n");
    j = InsertHash(&H, r[i]);//插入第N个记录
    if (j == 0)//重建哈希表
        j = InsertHash(&H, r[i]);//重建哈希表后重新插入第N个记录
    printf("遍历重建后的哈希表:\n");
    TraverseHash(H, print);
    printf("请输入待查找记录的关键字: ");
    scanf("%d",&K);
    j = Find(H, K, &p);
    if (j == SUCCESS)
        print(p, H.elem[p]);
    else
        printf("没找到\n");
    DestroyHash(&H);
    system("pause");
    return 0;
}
```



五．实验总结

1. 通过比较时间复杂度，发现无论所查数是否在查找表中，有监视哨的算法所需时间都更短（见上面第一条的截图）
2. 在写二叉排序树的时候，本来创建函数 CreateBFT,后来发现每次调用 Insert 函数都会改变二叉排序树的基地址，后改为在 main 函数中实现对二叉排序树的构建
3. 在编写哈希表查找时，发现提供的伪代码有错误，导致后续哈希表插入无法正常实现

   修改为

```
while (H.elem[*p].key != NULL && !(K == H.elem[*p].key))
{
    (*c)++;
    if (*c < m)
        collision(p, *c);//求得下一探查地址p
    else
        break;
}
```

在调断点过程中发现按原来编写后续 c 值无法被改变，就无法继续实现重建哈希表的操作