# 四川师范大学 实验报告

学期： 2024 至 2025 第一学期 实验成绩：

课程名字：《程序设计基础——数据结构(C语言版)》 专业： 信息与计算科学

班级： 2023 级 9 班 实验编号： 10

实验项目： 实验十 指导老师： 冯山

姓名： 刘智恒 学号： 2023060522

**一、实验题目**

查找算法实验

**二、实验目的及要求**

1.熟练掌握各种静态查找表方法的基本思想(顺序、折半、索引顺序表等)。

2.熟练掌握二叉排序树的构造方法和元素查找方法。

3.了解和掌握其它相关的查找方法。

**三、实验内容：(类C算法的程序实现，任选其二)**

1.顺序查找的算法实现。(必做)

2.折半查找的算法实现。

3.二叉排序树的构造算法与查找算法实现。

4.哈希表的查找算法实现。

**四、实验准备**

1.计算机设备;

2.程序调试环境的准备，本实验采用**Microsoft Visual Studio**环境;

3.实验内容的算法分析与代码设计与分析准备；

4.实验源程序**Exp\_10(1)**，**Exp\_10(2)，Exp\_10(3)**准备。

5.实验测试用例：

**（1）Exp\_10(1)：**

**（第一组）**

**关键字数目：6**

**具体数据：{12 43 32 55 78 29}**

**待查关键字：78**

**（第二组）**

**关键字数目：5**

**具体数据：{24 56 32 14 79}**

**待查关键字：10**

**（2）Exp\_10(2)：**

**（第一组）**

**关键字数目：8**

**具体数据：{4 6 3 7 2 8 1 5}**

**待查关键字：7**

**（第二组）**

**关键字数目：10**

**具体数据：{12 23 34 45 56 67 78 89 12 23}**

**待查关键字：10**

**（3）Exp\_10(3)：**

**（第一组）**

**关键字数目：10**

**具体数据：{0 11 21 31 41 51 61 71 82 97}**

**待查关键字：以上所有，并查找一个以上序列中没有的。**

**（第二组）**

**关键字数目：10**

**具体数据：{50 81 191 32 109 65 75 97 88 9}**

**待查关键字：以上所有，并查找一个以上序列中没有的。**

**五、实验内容**

**（一）问题分析**

（1）针对实验内容1，**顺序查找**（**Sequential Search**）的过程为：从表中的**最后一个**（或第一个）记录开始，**逐个向前**将每个记录的关键字和给定值进行比较：若某个记录的关键字和给定值比较**相等**，则**查找成功**；反之，若**直至第一个记录**其关键字和给定值比较都**不等**，则表明表中没有所查的记录，此时**查找不成功**。此处运用了“**哨兵**”的方法，即**将给定值放置于表中的第一个位置ST.R[0].key**。

（2）针对实验内容2，**折半查找**（**Binary Search**）的过程为：查找之前**先确定待查记录所在的范围**（**区间**），然后**逐步缩小范围**直到**找到**或**找不到**该记录为止。

本实验中顺序查找和折半查找均采用**顺序存储结构**：

**typedef struct {**

**KeyType key; //key value**

**} ElemType;**

**typedef struct {**

**ElemType \*R; //table: R**

**int length; //length of table R**

**} SSTable;**

**（注：此处将实验内容1和2整合为了Exp\_10(1)）**

（3）针对实验内容3，**二叉排序树**（**Binary Sort Tree**）或是一棵**空树**；或者是一棵具有以下性质的二叉树：若其左子树不为空，则**左子树上所有结点的值均小于它的根节点的值**；若其右子树不为空，则**右子树上所有结点的值均大于它的根节点的值**；它的**左、右子树也分别为二叉树**。

二叉排序树的查找过程为：当二叉排序树**不为空**时，首先将给定值和根结点的关键字比较，若相等，则查找成功；否则将依据给定值和根结点的关键字之间的**大小关系**，分别在**左子树**或**右子树**上继续进行查找。

**（注：实验内容3对应为Exp\_10(2)）**

本实验中二叉排序树的存储结构与二叉树类似，采用**二叉链表**：

**typedef struct BSTNode {**

**ElemType data;**

**struct BSTNode \*lchild;**

**struct BSTNode \*rchild;**

**}BSTNode, \*BSTree;**

（4）针对实验内容4，**哈希表**（**Hash Search**）的查找过程和**哈希表的构造**过程基本一致。给定**K值**，根据造表时设定的**哈希函数**求得**哈希地址**，若表中此位置上没有记录，则查找不成功；否则**比较关键字**：若和给定值相等，则查找成功；否则根据造表时设定的**处理冲突的方法**找“**下一地址**”，直到哈希表中某个位置为“**空**”或则和表中所填记录的关键字等于给定值时为止。

**（注：实验内容4对应为Exp\_10(3)）**

本实验中，哈希表的存储结构采用**顺序存储结构**，结构定义如下：

**typedef struct{**

**KeyType key; //key value**

**}ElemType;**

**typedef struct HashTable{**

**ElemType \*H; //hash table**

**int count; //record key numbers**

**}HashTable;**

**（二）算法描述**

**1. Exp\_10(1)**:

（1）**Status Init\_ST(SSTable \*ST, int n)**;

//本函数用于初始化顺序查找表**ST**：输入表**ST**以及给定的表长**n**，对于**n**进行有效检查，若不合法则输出错误信息并退出；然后对表**ST**进行动态内存分配，将**n+1**个大小的空间估计分配给**ST**（**ST.R[0]作为哨兵位置，用于存储待查记录**）。

（2）**Status Dest\_ST(SSTable \*ST)**;

//本函数用于销毁顺序查找表**ST**：释放表**ST**已分配的内存，并置为**NULL**，表长**length**也置为**0**。

（3）**void Print\_ST(SSTable ST)**;

//本函数用于输出表**ST**的内容：遍历表中的每个元素，打印其关键字。

（4）**int Search\_Seq(SSTable ST, KeyType key)**;

//本函数用于顺序查找表**ST**中是否存在关键字为**key**的元素：输入表**ST**和关键字**key**，首先将**key**赋值给**ST.R[0].key**作为哨兵，然后从表的末尾向前遍历，直到找到**key**或遍历到哨兵位置；如果找到**key**，返回其位置；否则返回**-1**。

（5）**Status Search\_Bin(SSTable ST, KeyType key)**;

//本函数用于二分查找表**ST**中是否存在关键字为**key**的元素：输入表**ST**和关键字**key**，首先初始化查找区间**[low, high]**：当**low <= high**时，执行以下步骤：计算中间位置**mid**；如果**ST.R[mid].key**等于**key**，返回**mid**。如果**key**小于**ST.R[mid].key**，缩小查找区间为左半部分；否则，缩小查找区间为右半部分。如果未找到**key**，返回错误信息。

**2. Exp\_10(2)**:

（1）**Status BST\_Insert(BSTree \*T, ElemType key)**;

//本算法用于向二叉排序树**T**中插入结点：从根节点开始，若新元素小于当前节点值，则往当前节点的左子树方向查找合适插入位置（找到左子节点为空的节点处插入）；若大于则往右子树方向查找合适插入位置（找到右子节点为空的节点处插入）。

（2）**void BST\_Create(BSTree \*T, ElemType a[], int n)**;

//本算法用于输入一组元素以此创建二叉排序树**T**：输入**n**个存储于数组**a[]**的元素，首先初始化二叉排序树的根节点指针为**NULL**；然后遍历输入的元素数组，对于每个元素，创建一个新的二叉树节点，并将元素值赋给节点的数据域，同时初始化其左右子节点指针为 **NULL**；若当前树为空，则将新节点设为根节点；若树不为空，则通过比较新元素与当前节点元素值的大小，将新节点插入到合适的左子树或右子树位置。

（3）**Status BST\_Search(BSTree T, ElemType key)**;

//本算法用于在二叉排序树**T**中查找特定键值**key**对应的节点：从根节点开始，循环比较要查找的键值**key**和当前节点的数据值。如果键值等于当前节点值，则表示找到，返回对应节点的数据值；如果键值小于当前节点值，则往当前节点的左子树继续查找；若大于则往右子树继续查找，直到找到目标节点或者遍历到叶节点仍未找到。

（4）**void InOrder\_Traverse(BSTree T)**;

//本算法用于对二叉排序树**T**进行中序遍历：如果当前节点不为 NULL，则先递归调用自身去遍历当前节点的左子树；然后访问当前节点；再递归调用自身去遍历当前节点的右子树。

**\***（5）**void BST\_Print(BSTree T, int level)**;

//本算法用于以**逆时针旋转90度**的树形结构样式打印二叉排序树，通过**缩进空格体现树的层次结构**：以**递归**方式进行操作，先递归处理当前节点的右子树，传递的层次参数**level**加**1**，这意味着越往深层的右子树，后续打印时缩进会更多。通过循环**输出空格**来实现，**空格数量与level有关**。

（6）**Status BST\_Free(BSTree T)**;

//本算法用于释放二叉排序树所占用的内存空间：首先判断当前节点是否为**NULL**，若不为**NULL**，则先递归调用自身释放当前节点的左子树内存空间；接着再递归调用自身释放当前节点的右子树内存空间；最后释放当前节点自身所占用的内存。

**3. Exp\_10(3)**:

（1）**Status Init\_HT(HashTable \*HT)**;

//本算法用于初始化哈希表：为哈希表分配能够存放**MAXSIZE**个**ElemType**类型元素的内存空间；接着将表示哈希表中元素个数的变量**count**初始化为**0**，通过循环遍历整个哈希表数组，把每个元素的**key**值都设置为**NULLKey**，以此标记这些位置初始状态为空。

（2）**Status Dest\_HT(HashTable \*HT)**;

//本算法用于销毁哈希表：释放哈希表所占用的内存空间，并将相关的指针置为**NULL**，元素个数**count**清零，

（3）**int HashFun(KeyType key)**;

//本算法用于根据给定键值通过哈希函数计算出在哈希表中的地址：简单地对输入的键值**key**进行取余操作，取余的除数为哈希表的最大容量**MAXSIZE**，返回计算得到的余数，其作为该键值对应的哈希地址。

（4）**void InsertHashKey(HashTable \*HT, KeyType key)**;

//本算法用于将给定键值插入到哈希表中合适的位置：首先调用**HashFun**函数计算出键值对应的初始哈希地址（**address**）。接着检查该地址对应的元素键值是否为初始的空值（**NULLKey**），若为空，则可以直接将给定键值存入该位置；若不为空，则表示**发生了冲突**，进入冲突处理环节。在冲突处理环节，采用**线性探测法**，按照步长**d**（**d = 1**）不断更新地址（**address = (address + d) % MAXSIZE**），直到找到一个键值为**NULLKey**的可用位置，然后将给定键值插入到该位置，并将哈希表中元素个数**count**加**1**。

（5）**int SearchKey(HashTable HT, KeyType key)**;

//本算法用于在哈希表中查找给定键值所在的位置：先通过**HashFun**函数计算出要查找键值对应的初始哈希地址（**address**），并将其赋值给临时变量**p**。然后开始循环检查：只要当前位置**p**处的键值不等于要查找的键值，就按照**线性探测法**更新p的值（**p = (p + 1) % MAXSIZE**），同时在更新过程中判断当前位置键值是否为空或者又回到了初始地址（**遍历完一圈都没找到**），若是则说明查找失败，返回**-1**；若（在循环中）找到了键值相等的位置，则返回该位置索引**p**。

（6）**void DisplayHashTable(HashTable HT)**;

//本算法用于显示哈希表中每个位置的键值存储情况：第一个循环依次打印出哈希表的位置索引（从**0**到**MAXSIZE - 1**），第二个循环再次遍历哈希表每个位置，若该位置的键值不为**NULLKey**，则打印出该键值；若键值为**NULLKey**，则打印一个短横线（**-**）来表示该位置为空。

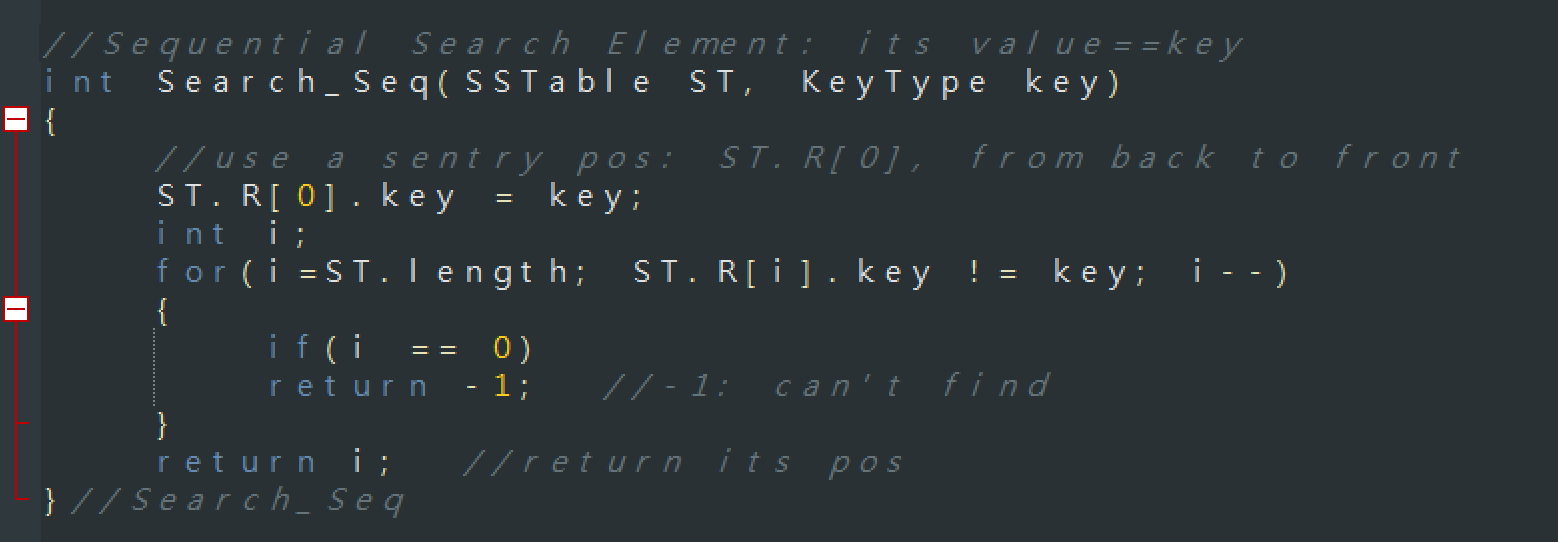
**（三）程序代码**

根据以上对**实验内容**的算法描述，求解的程序代码如下：

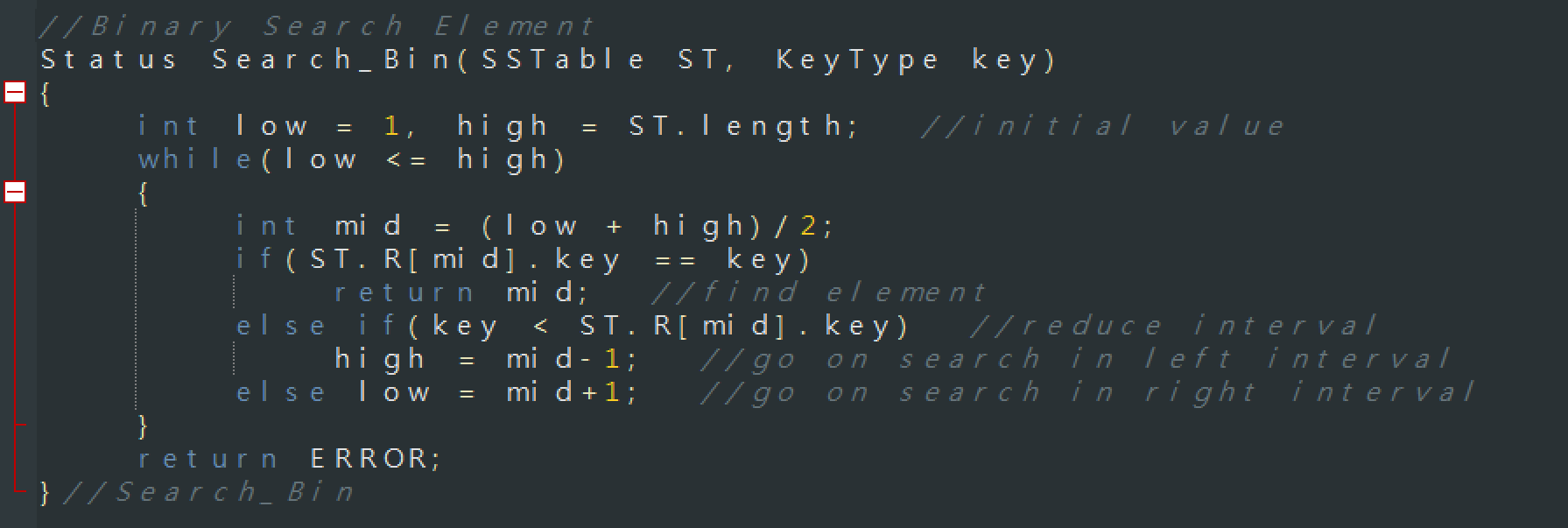


**注：此处只展示主要算法的程序代码图。**

**1. Exp\_10(1)：**

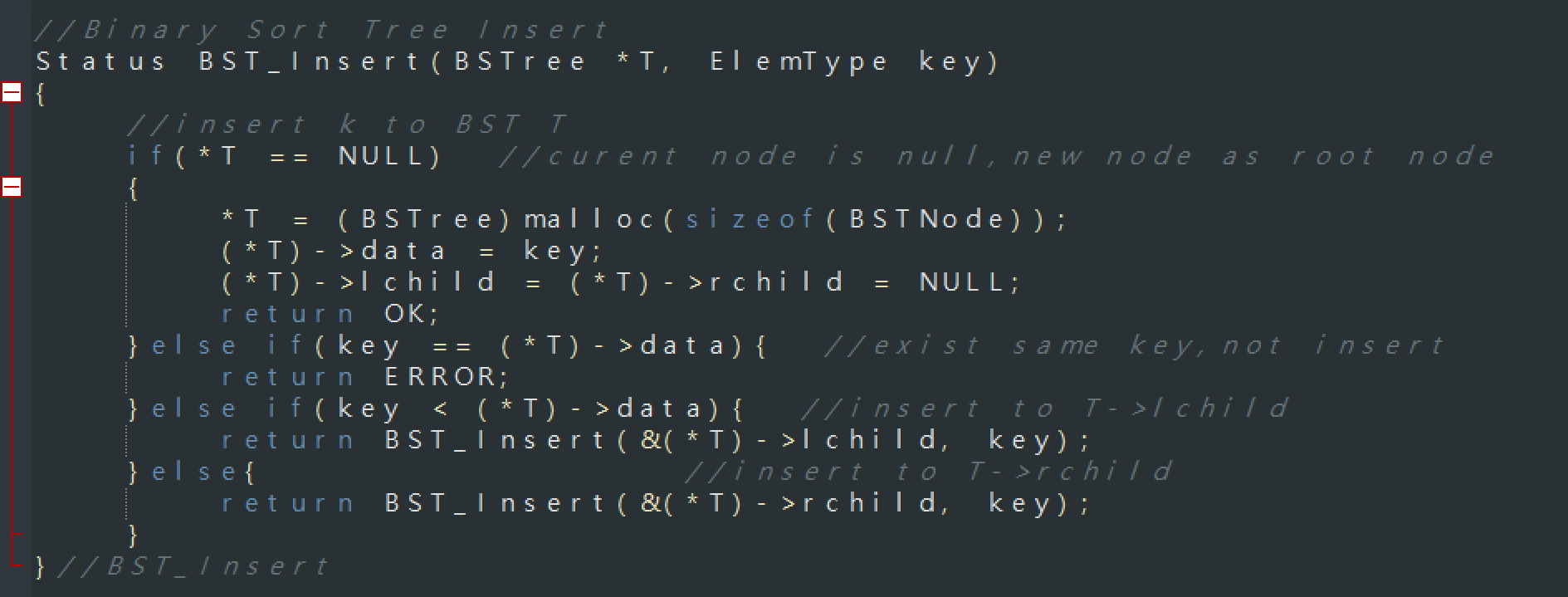


**图5.1顺序查找算法**

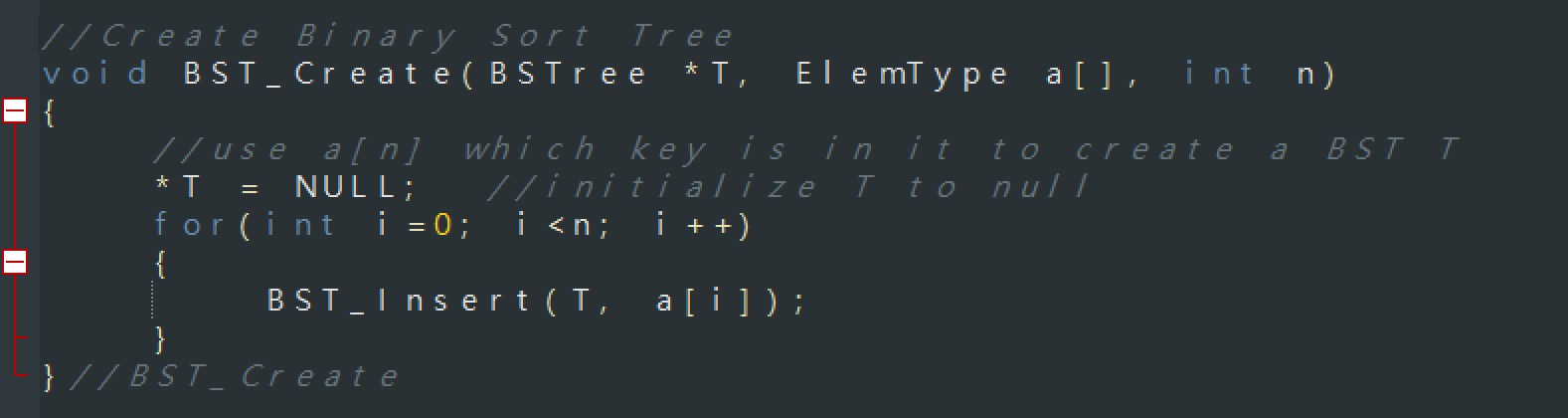


**图5.2折半查找算法**

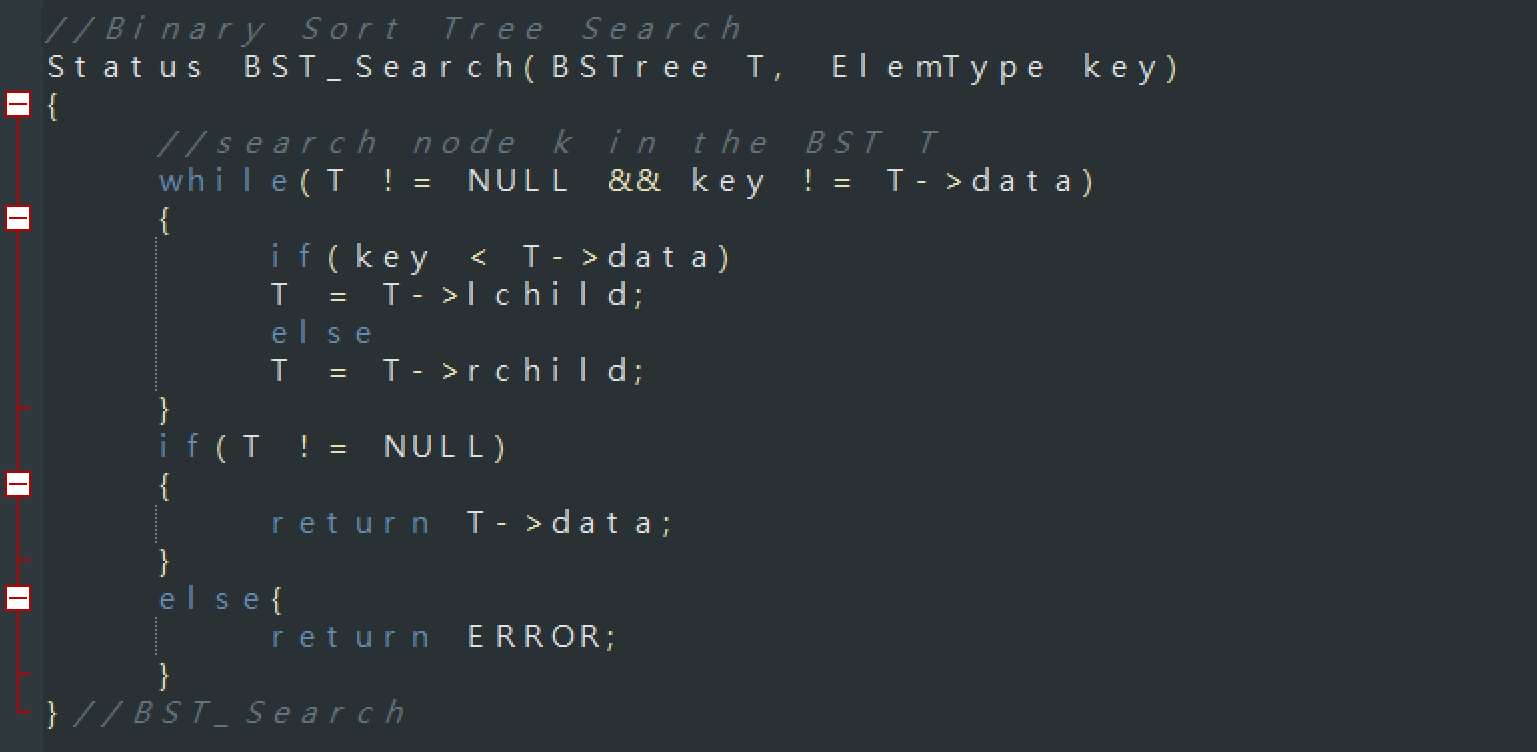
**2. Exp\_10(2)：**



**图5.3二叉排序树插入**



**图5.4创建二叉排序树**

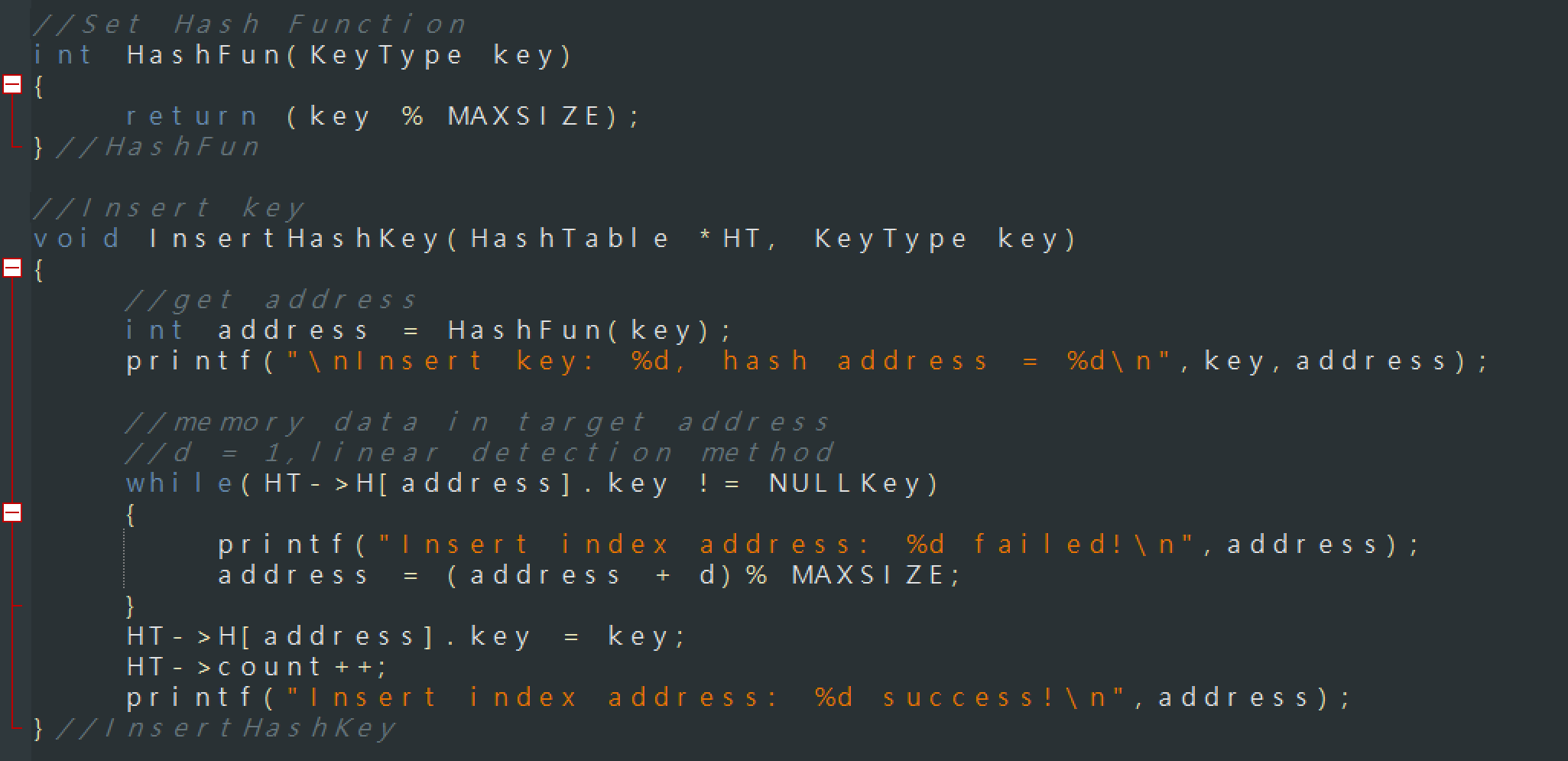
****

**图5.5二叉排序树查找**

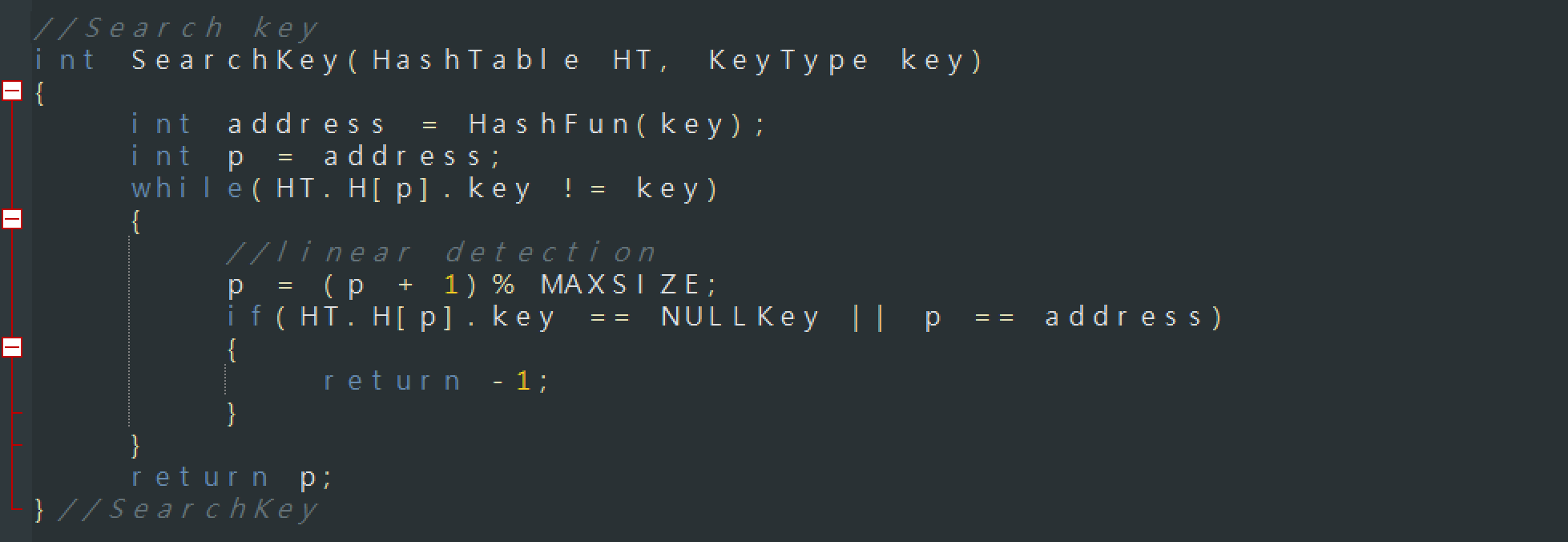


**图5.6打印二叉排序树**

**3. Exp\_10(3)：**

****

**图5.7哈希函数和插入键值**

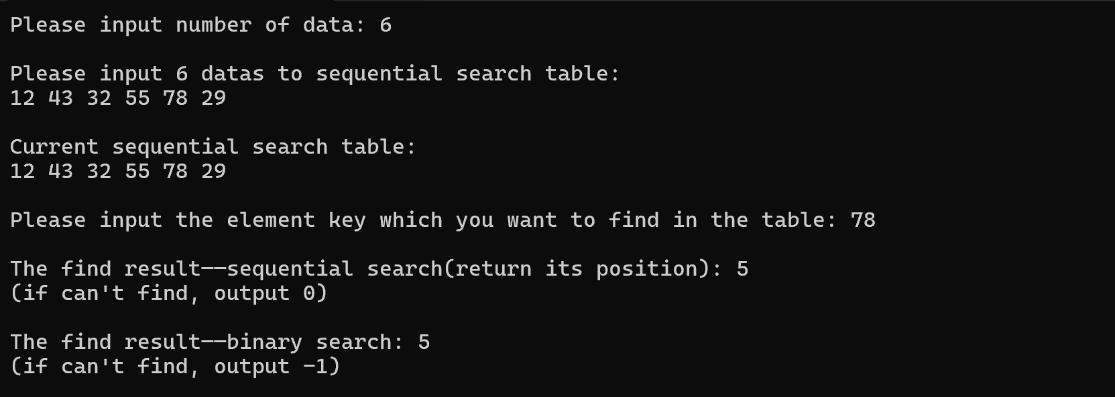
****

**图5.8哈希表查找**

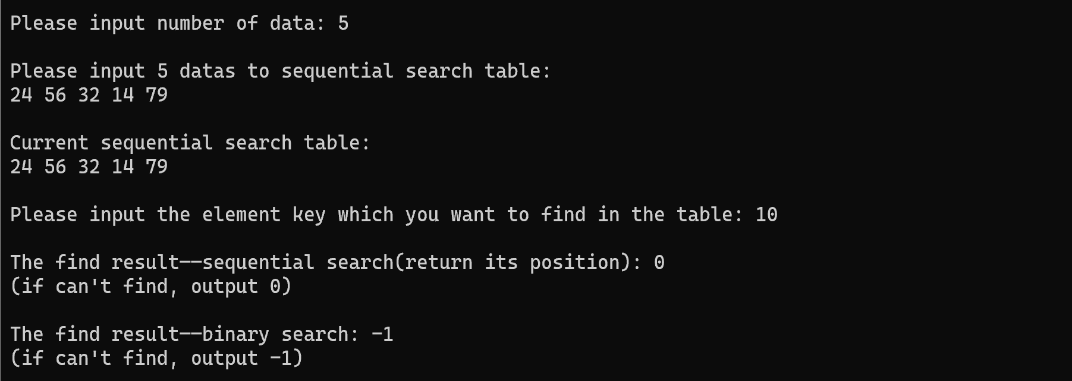
**六、实验结果**

**（一）结果呈现：**

1.根据以上**实验内容1和2**的程序代码**Exp\_10(1)**运行后进行多次测试，以下是根据**实验内容1和2的测试用例**进行测试后的结果示意图：

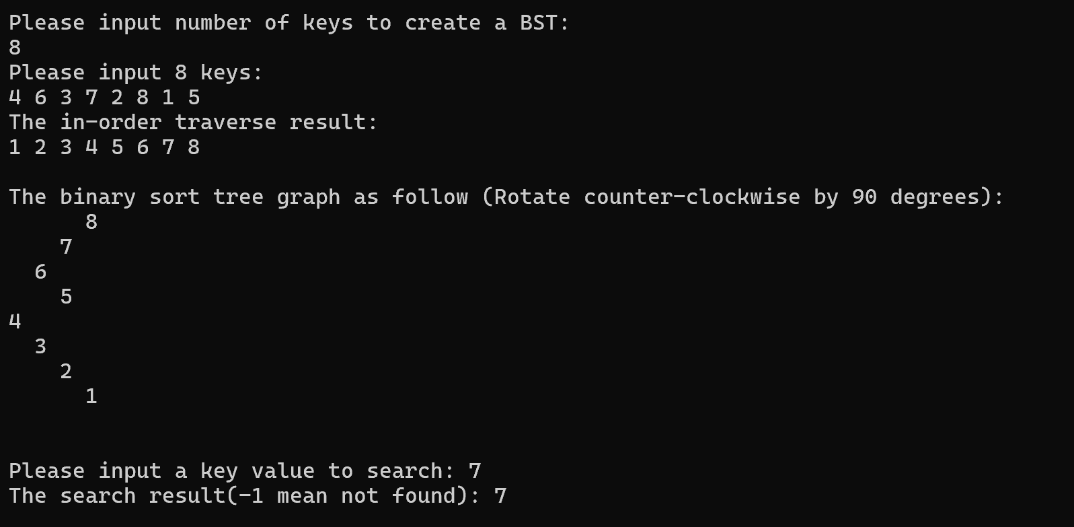


**图6.1 Eep\_10(1)第一组实验测试用例结果**

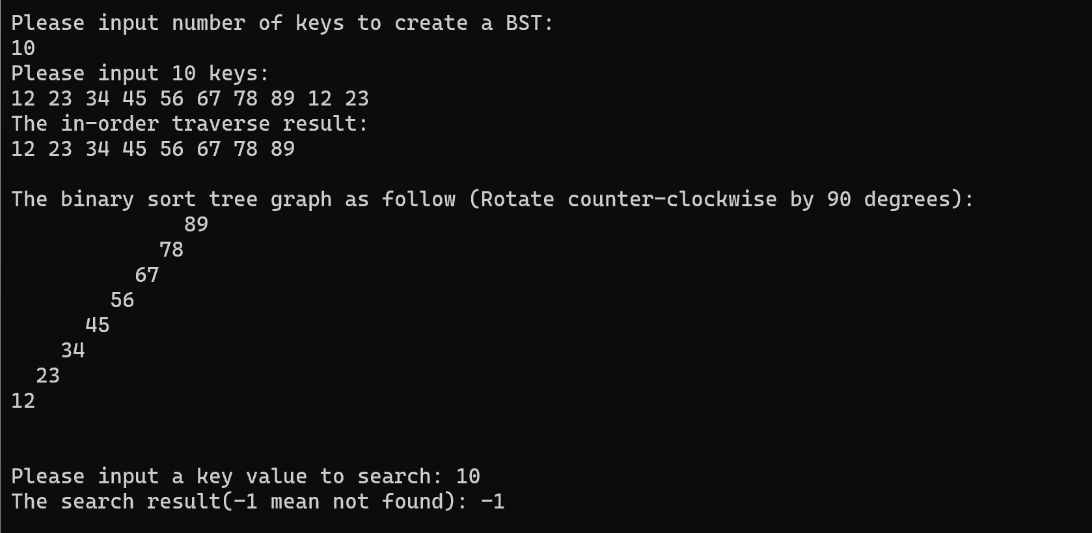
****

**图6.2 Exp\_10(1)第二组实验测试用例结果**

2.根据以上**实验内容3**的程序代码**Exp\_10(2)**运行后进行多次测试，以下是根据**实验内容3的测试用例**进行测试后的结果示意图：

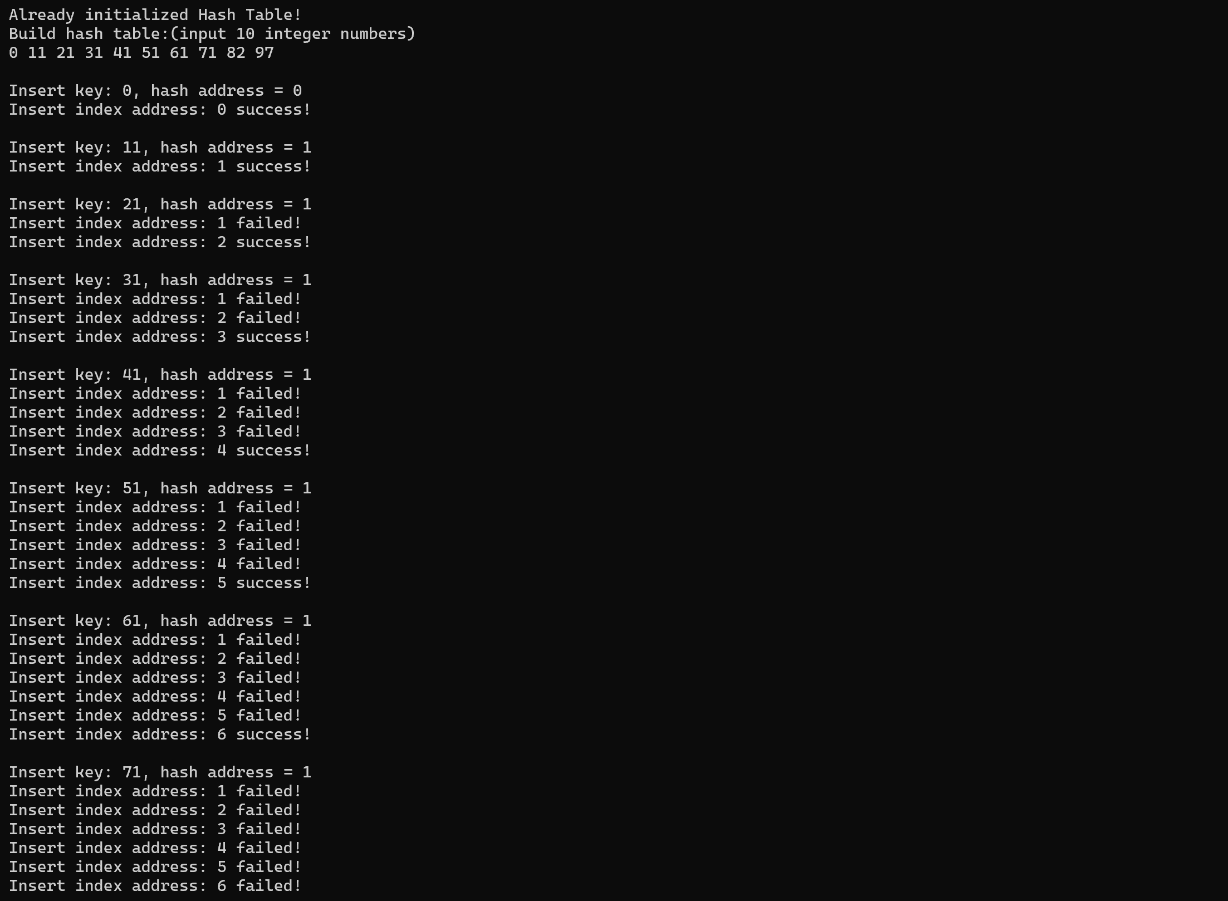
****

**图6.3 Exp\_10(2)第一组实验测试用例结果**

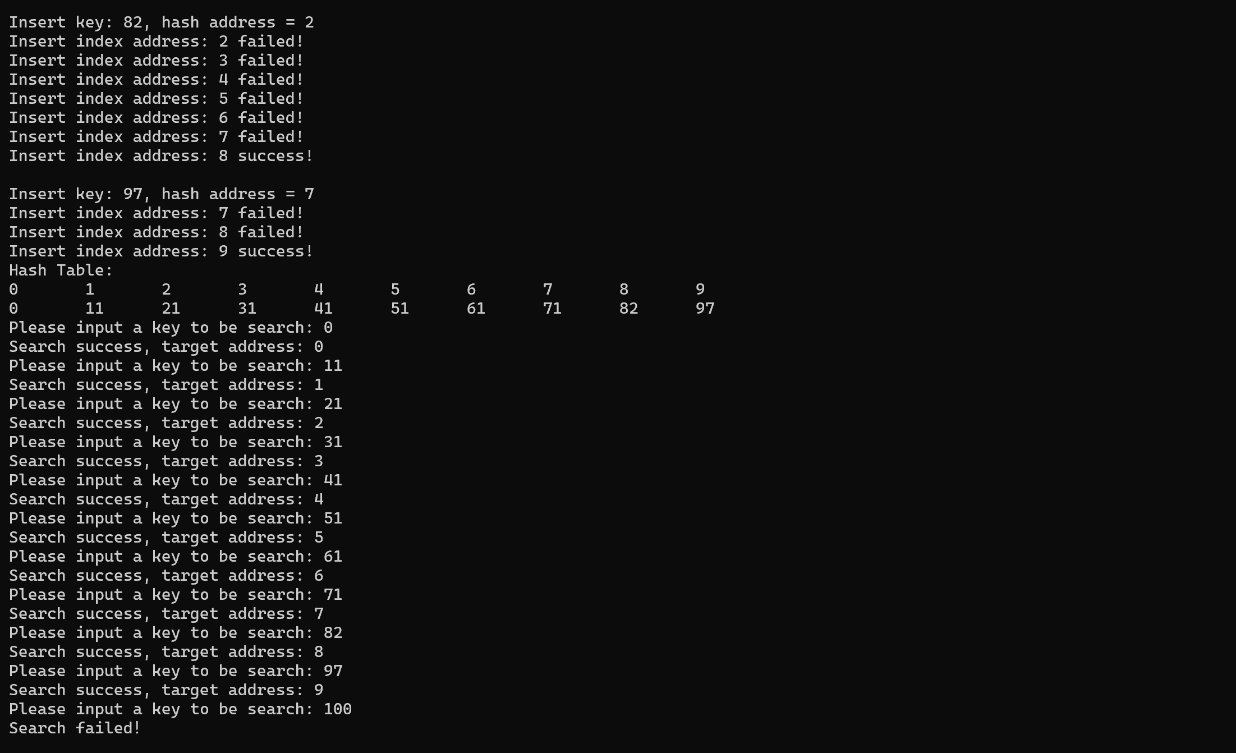
****

**图6.4 Exp\_10(2)第二组实验测试用例结果**

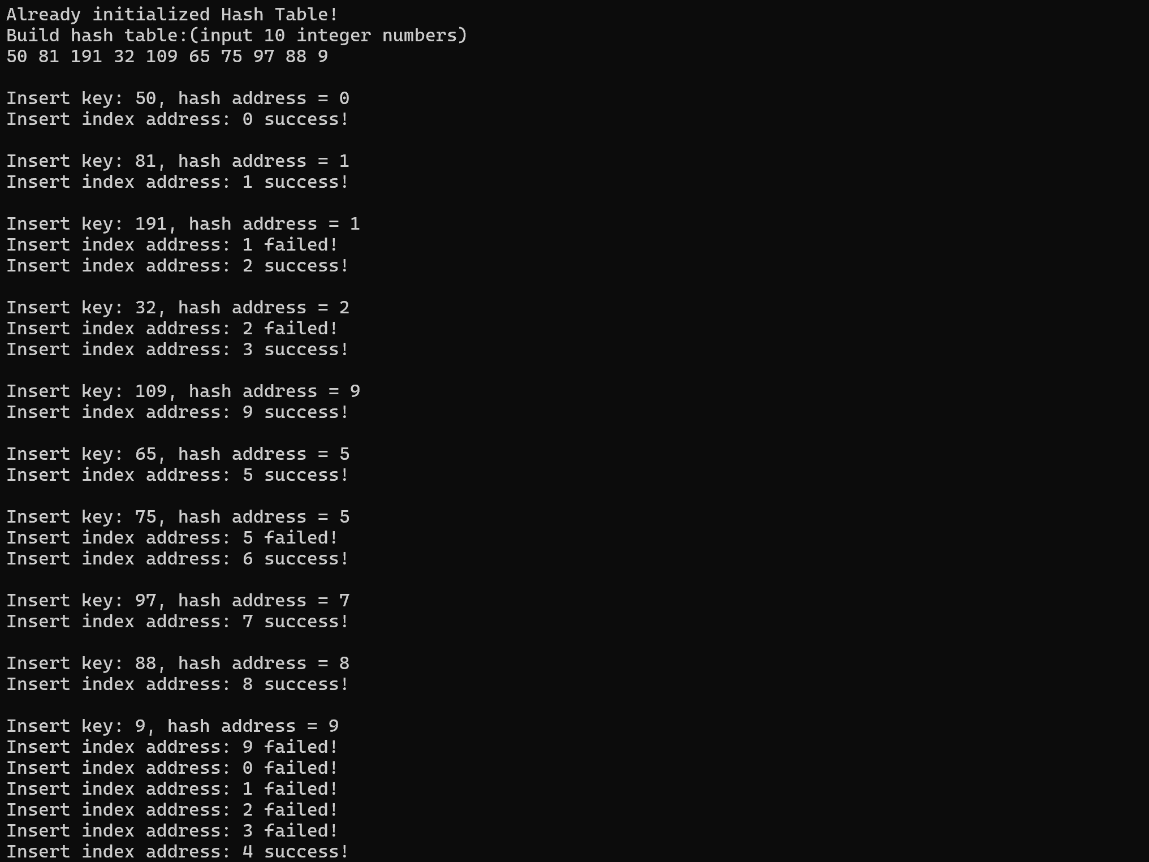
3.根据以上**实验内容4**的程序代码**Exp\_10(3)**运行后进行多次测试，以下是根据**实验内容4的测试用例**进行测试后的结果示意图：

****

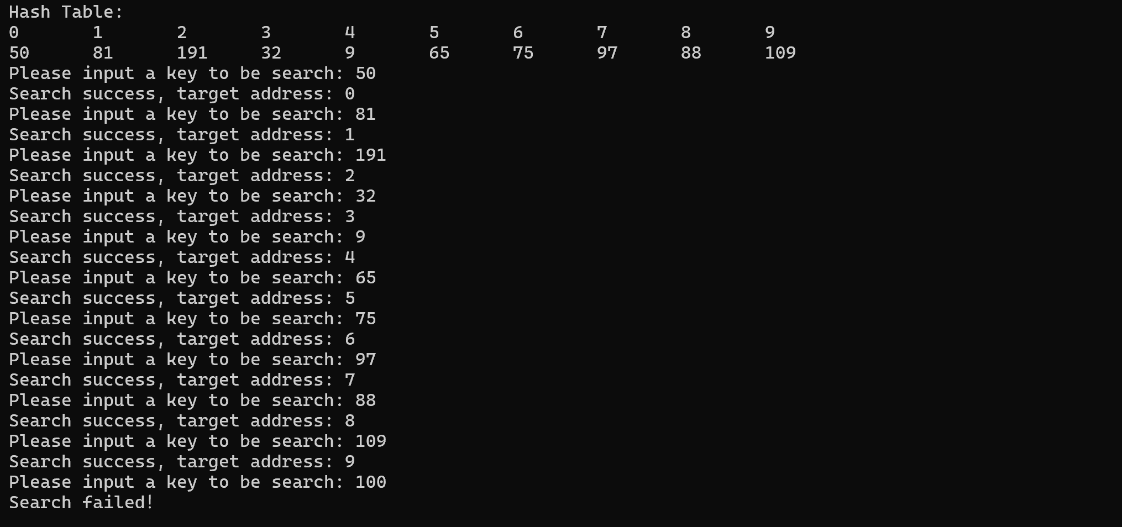
**图6.5 Exp\_10(3)第一组实验测试用例结果**

****

**图6.6 Exp\_10(3)第一组实验测试用例结果**

****

**图6.7 Exp\_10(3)第二组实验测试用例结果**

****

**图6.8 Exp\_10(3)第二组实验测试用例结果**

**（二）结果分析：**

**时间与空间复杂度分析**

**1. Exp\_10(1)：**

**【Search\_Seq】—— 顺序查找**

在最坏情况下，需要遍历整个顺序表才能确定元素是否存在，此时需要比较**n**次，故时间复杂度为：**O(n)**。顺序查找过程中，只使用了有限的几个额外变量（如记录当前位置的变量等），不随**n**的增大而额外占用更多的空间，故空间复杂度为：**O(1)**。

【**Search\_Bin**】**—— 折半查找**

**折半查找要求顺序表是有序的**，每次比较都能将查找区间缩小一半，最多需要比较的次数为以**2**为底**n**的对数向上取整次（或向下取整加**1**次），故时间复杂度为：**O(log2(n))**。和顺序查找类似，折半查找在查找过程中也只是使用了几个额外变量（如记录区间上下界的变量），故空间复杂度为：**O(1)**。

**综上所述：Exp\_10(1)整个时间复杂度为：O(n)，主要受到顺序查找以及表的构建、销毁等于n相关的操作影响；空间复杂度为：O(n)，主要是顺序表存储数据占用的空间随n变化。**

**2. Exp\_10(2)：**

**【BST\_Create】—— 创建二叉排序树**

对于**n**个结点，平均情况下每次插入的时间复杂度接近：**O(logn)**，总共插入**n**个节点，所以创建操作的时间复杂度接近：**O(nlogn)**；最坏情况下树是一条链，每次插入时间复杂度为：**O(n)**，此时整体创建的时间复杂度为：**O(n^2)**。创建过程中主要是存储树的节点空间，最终存储**n**个节点，空间复杂度为：**O(n)**。

【**BST\_Search**】**—— 二叉排序树查找**

每次比较可以将查找范围缩小一半，平均情况下时间复杂度为：**O(logn)**；最坏情况下树是一条链，时间复杂度为：**O(n)**。该函数只使用了几个临时变量，故空间复杂度为：**O(1)**。

**【BST\_Print】—— 打印二叉排序树**

该函数递归地遍历二叉排序树的的每个结点，对于有**n**个结点的二叉树，每个结点都会被访问一次，时间复杂度和结点个数**n**成正比，故时间复杂度为：**O(n)**。递归调用的栈空间取决于树的最大深度，最坏情况下二叉树是一条链（如每个结点只有左子树和右子树），此时树的深度为**n**，空间复杂度为：**O(n)**；最好情况下二叉排序树为完全二叉树的形态，深度为**log2(n)**，平均情况下空间复杂度可近似看作在**O(logn)**到**O(n)**之间。

【**BST\_Free**】—— **释放二叉排序树的内存**

同样也是通过递归的方式释放二叉排序树的每个结点的内存空间，需要遍历树的所有**n**个结点，故时间复杂度为：**O(n)**。同样因为递归调用，最坏情况下空间复杂度为：**O(n)**，平均情况下在**O(logn)**到**O(n)**之间。

**综上所述：Exp\_10(2)整体的时间复杂度为：O(n^2)，平均为：O(nlogn)；空间复杂度为：O(n)**。

**3. Exp\_10(3)：**

【**HashFun**】**—— 哈希函数**

只进行了简单的取余运算（**key % MAXSIZE**），时间复杂度为**O(1)**。没有额外占用空间，空间复杂度为**O(1)**。

【**InsertHashKey**】**—— 插入元素到哈希表**

在理想情况下（**没有冲突**），计算哈希地址并插入元素时间复杂度为**O(1)**。但在最坏情况下，采用**线性探测法**解决冲突，可能需要遍历整个哈希表，时间复杂度会退化为**O(MAXSIZE)**，平均情况下时间复杂度接近**O(1)**，取决于哈希表的**装填因子**等因素。只是修改已分配哈希表空间内的元素值，空间复杂度为**O(1)**。

【**SearchKey**】**—— 在哈希表查找元素**

在理想情况下（**一次就找到**）时间复杂度为**O(1)**，但最坏情况（遍历整个哈希表都没找到或者冲突严重）下时间复杂度为**O(MAXSIZE)**，平均情况下接近**O(1)**。只使用了有限的几个临时变量，空间复杂度为**O(1)**。

**综上所述：Exp\_10(3)整体的时间复杂度主要受插入和查找操作在最坏情况下的影响，可能达到O(n^2)（n为插入元素个数，如果不断冲突需要大量线性探测），但平均情况接近O(n)；空间复杂度主要取决于哈希表本身的空间分配，为O(MAXSIZE)，即O(1)（MAXSIZE为常数）。**

**七、实验总结**

1. **Exp\_10(2)：**打印二叉排序树时是通过递归来打印树的结构，但递归深度过深时，可能导致**栈溢出**，尤其是对于**结点数较多、树的深度较大**的情况。可以考虑使用**队列**实现**层序遍历**以此来打印树。

2. **Exp\_10(3)：**目前代码没有考虑哈希表的**装填因子**，当插入元素过多，装填因子过高时，哈希冲突概率会大大增加。可以添加代码来**监测**装填因子，当达到一定**阈值**（比如0.75）时，对哈希表进行**扩容**操作，并**重新散列**已有元素。

3. **Exp\_10(3**)：目前使用的是**线性探测法**，这种方法在处理冲突时容易出现**聚集**现象，导致后续查找、插入性能变差。可以考虑采用**二次探测法**、**链地址法**等其他冲突解决策略来替代，以此提升效率。