数据结构实验报告

实验成绩：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 专业班级 |  |
| 指导教师 | 张先宜 | 实验时间 | 6月9日 | 实验地点 | 计算机楼 |

**实验名称： 查找实验**

1. **实验目标**

(1) 掌握顺序表的查找方法，尤其是二分查找方法。

(2) 掌握二叉排序树的建立及查找。

查找是软件设计中的最常用的运算，查找所涉及到的表结构的不同决定了查找的方法及其性能。二分查找是顺序表的查找中的最重要的方法，应能充分理解其实现方法和有关性能，并能借助其判定树结构来加深理解。二叉排序树结构在实验时具有一定的难度，可结合二叉树的有关内容和方法来实现。

1. **实验要求**
2. 查找表的定义和算法实现放入库文件，如“search.h”中；
3. 查找的测试数据用文本文件方式给出，例如测试数据名为ser.dat；
4. 可多次连续测试。

1. **实验内容**

编写算法实现下列问题的求解。

(1) 对下列数据表，分别采用二分查找算法实现查找，给出查找过程依次所比较的元素（的下标），并以二分查找的判定树来解场释。

第一组测试数据：

数据表为 (1,2,3,4,6,7,8,9,10,11,12,13,17,18,19,20,24,25,26,30,35,40,45,50,,100)

查找的元素分别为： 2，8，20, 30，50，5，15，33，110

第二组数据：

数据表为 (2,3,5,7,8,10,12,15,18,20,22,25,30,35,40,45,50,55,60, 80,100)

查找的元素分别为： 22，8，80，3，100，1，13，120

(2) 设计出在二叉排序树中插入结点的算法，在此基础上实现构建二叉排序树的算法。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35

(3) 设计算法在二叉排序树中查找指定值的结点。

测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别查找下列元素： 150，70，160，190，10，55，175

(4) 设计算法在二叉排序树中删除特定值的结点。

测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别删除下列元素：30，150，100

(5) 设计算法，对给定的二叉排序树，求出在等概论情况下的平均查找长度。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35

(6) 对给定的二叉树，假设其中各结点的值均不相同，设计算法以判断该二叉树是否是二叉排序树。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

70，100，60，80，150，120，50，160，30，40，170，180，175，35

1. **数据结构设计**

typedef struct lBNode

{

int data;

struct lBNode \*lChild, \*rChild;

}BiNode;

1. **算法设计**

一二三均为书上算法

<4>先遍历二叉树找到数据结点，若该结点无左右子树，则直接删除；若只有左子树或者只有右子树，就把唯一的子树接在数据结点位置；若左右子树都存在，就把左子树中最大值接在数据结点位置，并把最大值结点的左子树接在最大值结点的原结点位置，如果最大值结点就为数据结点的左子树，把整个左子树接在原数据结点上。

void deleteBiNode(BiNode \*&T, int x) //4.二叉树删除

{

if(T)

{

if(T->data == x)

{

if(T->lChild == NULL && T->rChild == NULL)

{

delete T;

T = NULL;

return;

}

else if(T->lChild == NULL && T->rChild)

{

BiNode \*p;

p = T;

T = T->rChild;

delete p;

}

else if(T->lChild && T->rChild == NULL)

{

BiNode \*p;

p = T;

T = T->lChild;

delete p;

}

else

{

BiNode \*sf = T, \*s = T->lChild;

while(s->rChild)

{

sf = s;

s = s->rChild;

}

T->data = s->data;

if(sf == T)

{

sf->lChild = s->lChild;

}

else

{

sf->rChild = s->lChild;

}

delete s;

}

}

deleteBiNode(T->lChild, x);

deleteBiNode(T->rChild, x);

}

else return;

}

<5>void CalcASL(BiNode\* root, double& sEi, int& sCi, int level)

{

if (root == NULL) {

sCi = 0; // 当前节点为空时，Ci设为0

return;

}

// 中序遍历左子树

int lCi = 0, rCi = 0, lsCi = 0, rsCi = 0;

double lsEi = 0, rsEi = 0;

CalcASL(root->lChild, lsEi, lsCi, level + 1);

// 中序遍历当前节点：对于二叉排序树来说，当前节点对应的关键字有一个单独的查找次数，因此 Ci 为 1

double cur\_Ei = level; // 当前节点对应的查找长度为 level

sEi += cur\_Ei;

sCi += 1;

// 中序遍历右子树

CalcASL(root->rChild, rsEi, rsCi, level + 1);

// 计算当前节点的 Ci 和 Ei

lCi = lsCi + 1; // 左子树中的关键字在查找当前节点时都要向右查找（包括当前节点的一个单独的查找次数），因此 Ci 为左子树中关键字的总数加上1

rCi = rsCi + 1; // 右子树中的关键字在查找当前节点都要向左查找（包括当前节点的一个单独的查找次数），因此 Ci 为右子树中关键字的总数加上1

if (lCi == 0) {

lCi = 1;

} // 如果左子树中没有关键字，则左子树中的关键字在查找当前节点时不需要进行查找，因此 Ci 应该为1

cur\_Ei += (lsCi + rsCi) \* cur\_Ei; // 当前节点对应的查找长度等于左子树和右子树中所有关键字的 Ci 之和再乘以 level

cur\_Ei /= (lCi + rCi); // 当前节点对应的查找长度除以总的查找次数，得到当前节点的 Ei

sEi += lsEi + rsEi + cur\_Ei; // 累加每个关键字的 Ei

sCi += lCi + rCi; // 累加每个关键字被

}

<6>判断一个二叉树是否为二叉排序树，需要满足以下条件：

左子树中所有结点的值均小于根节点的值；右子树中所有结点的值均大于根节点的值；左子树和右子树都必须是二叉排序树。

isBST函数用于判断当前节点是否满足二叉排序树的条件，并递归判断左右子树是否也满足二叉排序树的条件。isBinarySearchTree函数则是用于判断整棵二叉树是否为二叉排序树。

其中，INT\_MIN和INT\_MAX分别为整型变量的最小值和最大值，用于限制根节点的值的范围。

bool isBST(BiNode\* root, int min, int max) {

if (root == NULL) {

return true;

}

if (root->data < min || root->data > max) {

return false;

}

return isBST(root->lChild, min, root->data - 1) &&

isBST(root->rChild, root->data + 1, max);

}

bool isBinarySearchTree(BiNode\* root) {

return isBST(root, INT\_MIN, INT\_MAX);

}

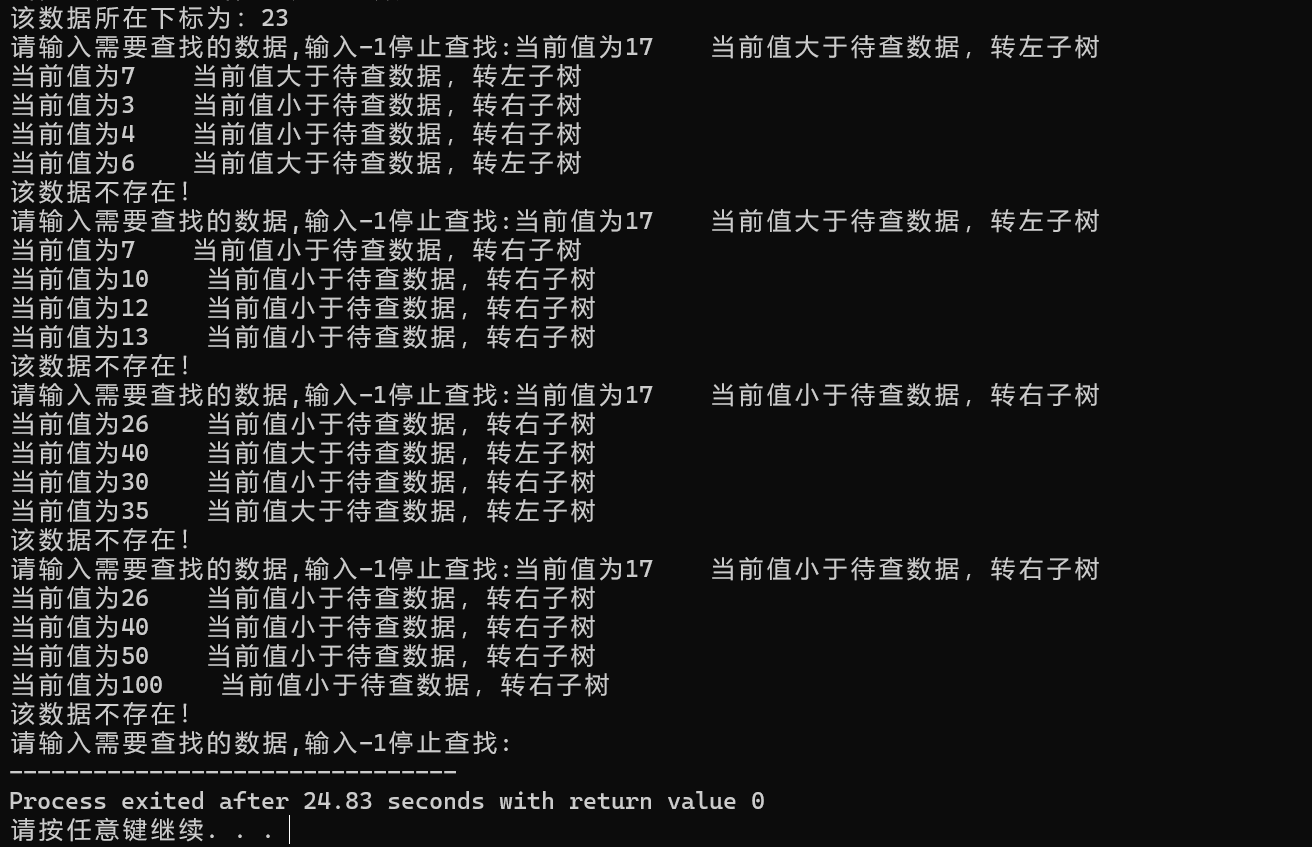
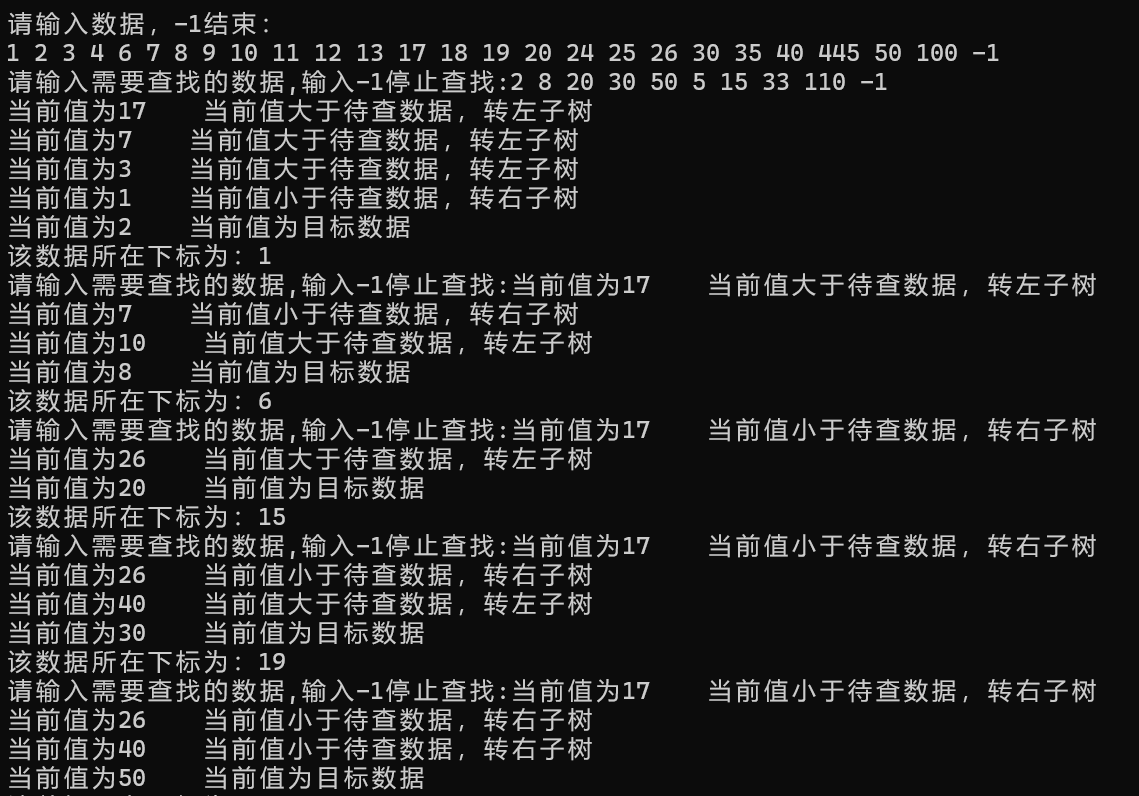
1. **运行和测试**

(1) 对下列数据表，分别采用二分查找算法实现查找，给出查找过程依次所比较的元素（的下标），并以二分查找的判定树来解场释。

第一组测试数据：

数据表为 (1,2,3,4,6,7,8,9,10,11,12,13,17,18,19,20,24,25,26,30,35,40,45,50,,100)

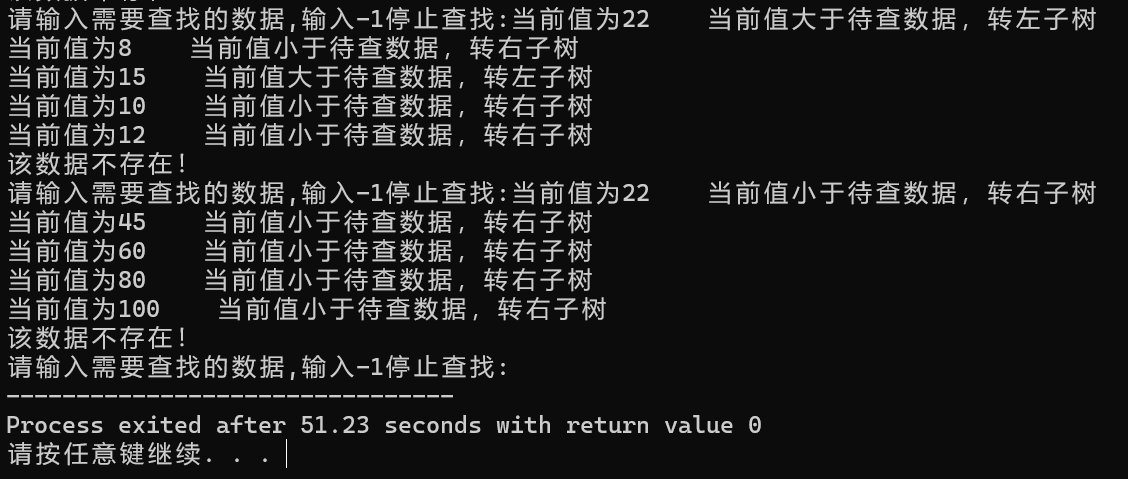
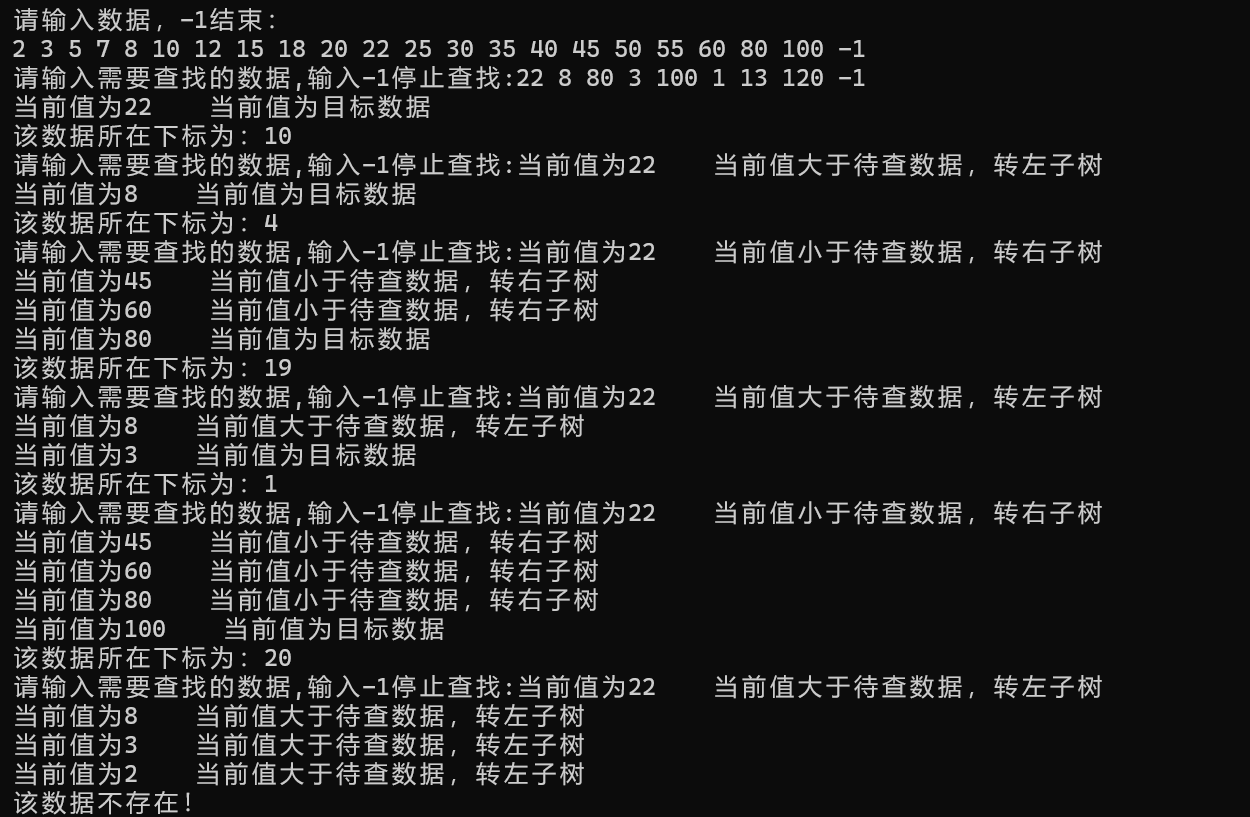
查找的元素分别为： 2，8，20, 30，50，5，15，33，110



第二组数据：

数据表为 (2,3,5,7,8,10,12,15,18,20,22,25,30,35,40,45,50,55,60, 80,100)

查找的元素分别为： 22，8，80，3，100，1，13，120

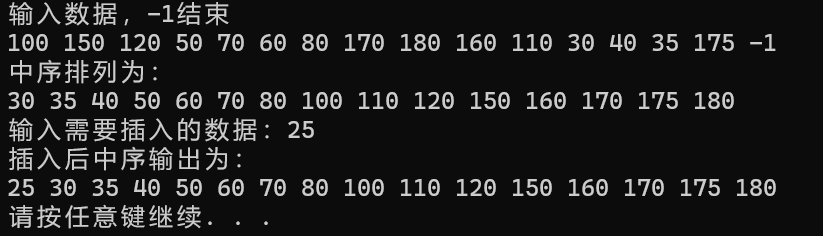


(2) 设计出在二叉排序树中插入结点的算法，在此基础上实现构建二叉排序树的算法。

测试数据：构建二叉排序树的输入序列如下：

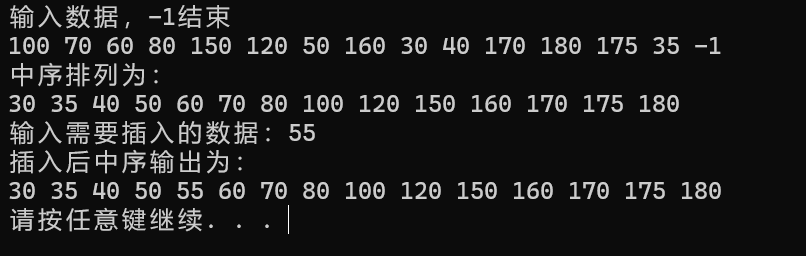
第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175



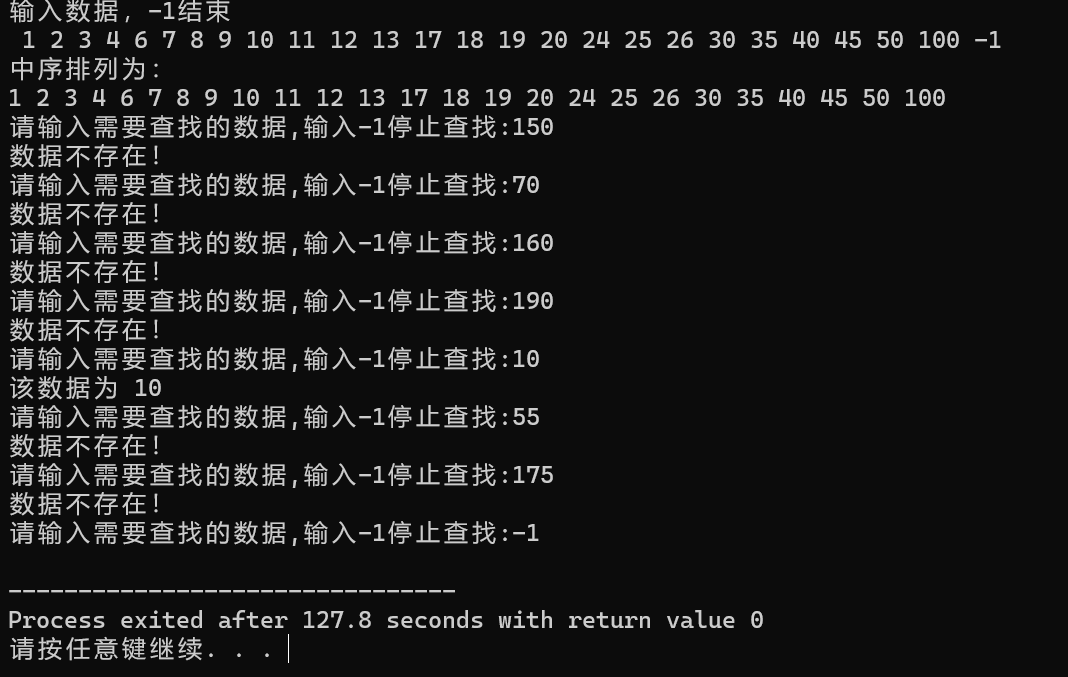
第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35



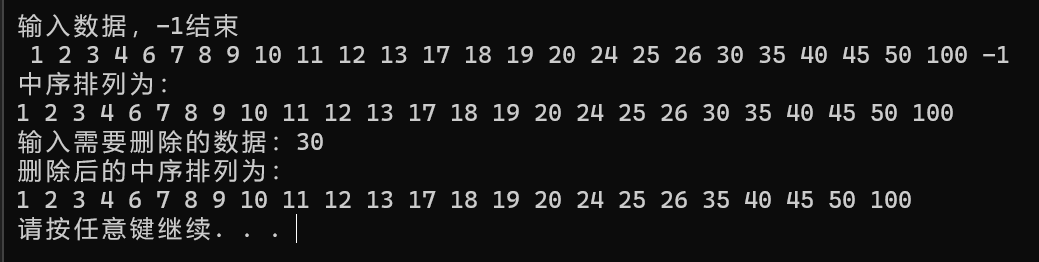
(3) 设计算法在二叉排序树中查找指定值的结点。

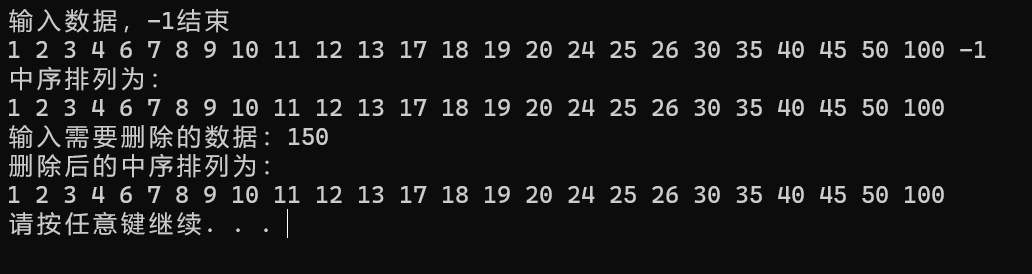
测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别查找下列元素： 150，70，160，190，10，55，175

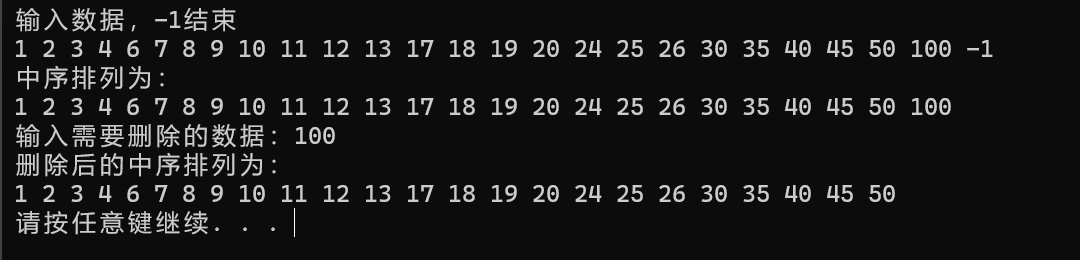


(4) 设计算法在二叉排序树中删除特定值的结点。

测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别删除下列元素：30，150，100





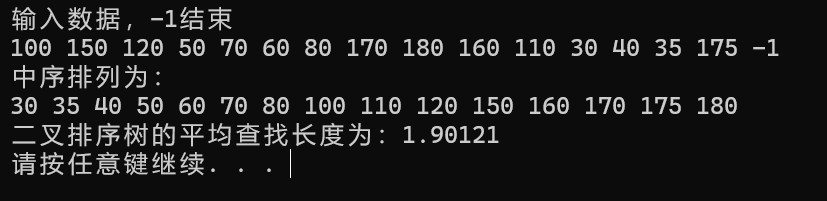


(5) 设计算法，对给定的二叉排序树，求出在等概论情况下的平均查找长度。

测试数据：构建二叉排序树的输入序列如下：

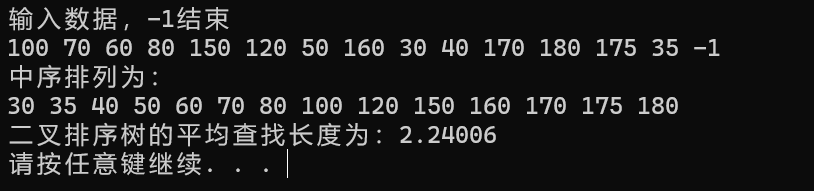
第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175



第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35

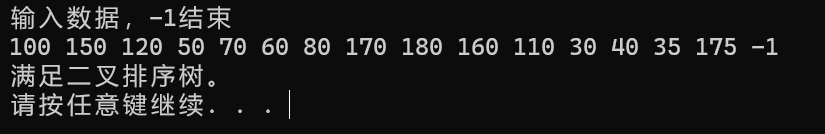


(6) 对给定的二叉树，假设其中各结点的值均不相同，设计算法以判断该二叉树是否是二叉排序树。

测试数据：构建二叉排序树的输入序列如下：

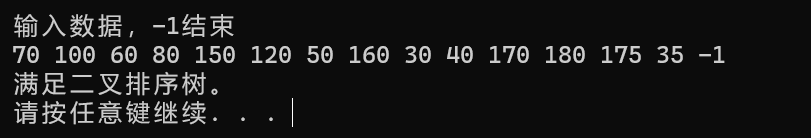
第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175



第二组数据：

70，100，60，80，150，120，50，160，30，40，170，180，175，35



1. **总结、心得和建议**

查找是软件设计中的最常用的运算，查找所涉及到的表结构的不同决定了查找的方法及其性能。二分查找是顺序表的查找中的最重要的方法。查找在现实生活中应用极为广泛，小至几十数据，用各种方法找出所要的数据所花时间差不了多少，但对于大至上万千万的数据，选取更为有效的方法来提高效率就显得非常有必要。

在递增有序表中采用二分查找法查找的速度是比较快的，但也存在问题，若要在其中插入或者删除元素时，需要移动元素以保持其有序性，为此可以采用动态链表结构。排序二叉树的查找与二分查找法相同，且在录入数据的时候就将数据排序好，对后序的查找提供了很大帮助。

本次实验课的学习，让我对简单的查找有了更深的理解，不是停留在顺序表，而是提升到了树表的层次