



**数据结构设计性实验报告**

**课程名称 数据结构**

**题目名称 二叉排序树**

**学生学院 计算机学院**

**专业班级 22软件工程卓越班**

**学 号 3222004465**

**学生姓名 陈婉瑜**

**指导教师 苏 庆**

**2023 年 12 月 31 日**

### 一、题目

采用整型int为元素类型和二叉链表为存储结构，实现抽象数据类型二叉排序树

ADT BSTree{

**数据对象**D：D是具有相同特性的数据元素的集合

**数据关系**R：若D = Φ,则R = Φ，称BSTree为空二叉排序树

若D ≠ Φ,则R = {H}，H有如下二元关系：

（1）在D中存在唯一的称为根的数据元素root,它在关系H下无前驱；

（2）若D-{root}≠Φ,则存在D-{root}={Dl,Dr},且Dl ∩ D=Φ；

（3）若Dl≠Φ,则Dl中存在唯一的元素xl，<root,xl>∈H,且存在Dl上的关系Hl⊂H；若Dr ≠Φ,则Dr中存在唯一的元素xr，<root,xr>∈H,且存在Dr上的关系Hr⊂H;H={<root,xl>，<root,xr>，Hl,Hr}；

（4）(Dl,{Hl})称为根的左子树；(Dr,{Hr})称为根的右子树；

（5）若左子树不空，则左子树上所有结点的值均小于根节点；

（6）若右子树不空，则右子树上所有结点的值均大于根结点；

（7）左右子树也分别是二叉查找树

**基本操作**P:

InitBSTree(BSTree &T)

操作结果：初始化构造一棵空的二叉排序树T

SearchBST(BiTree T, KeyType key)

操作结果：若二叉排序树T存在值为key的结点，则返回该结点指针，否则返回NULL

InsertBST(BiTree &T, ElemType e)

操作结果：若二叉排序树root不存在值为e的结点，则插入到T

DeleteBST(BiTree & T, KeyType key)

操作结果：若二叉排序树中存在值为key的结点，则删除

Status destroyBST(BSTree \*root);

操作结果：销毁二叉排序树

}ADT BSTree

### 二、存储结构定义

**公用头文件：**

**#include<stdio.h>**

**#include<stdlib.h>**

**#define ElemType int //元素类型为整型**

**#define KeyType int //元素类型为整型**

**typedef enum { FALSE, TRUE } Status;//定义枚举类型判断函数是否正常运行**

**二叉链表存储结构：**

**typedef struct BSTNode**

**{**

**int data;//结点存储数据元素**

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

**} BSTNode, \* BSTree; //二叉排序树**

### 三、算法设计

（**1）二叉排序树T中查找元素key**

**SearchBST(BSTree T, KeyType key)**

**实现思路：**

在二叉排序树中查找目标元素，就是从树根结点出发，先将树根结点和目标元素做比较：若当前结点不存在，则查找失败；若当前结点的值和目标元素相等，则查找成功；若当前结点的值比目标元素大，目标元素只可能位于当前结点的左子树中，继续进入左子树查找；若当前结点的值比目标元素小，目标元素只可能位于当前结点的右子树中，继续进入右子树查找；

**实现代码：**

BiTree SearchBST(BSTree T, KeyType key) {

//如果 T 为空，则查找失败，返回NULL；

if(T==NULL) return NULL;

//或者查找成功，返回指向存有目标元素结点的指针

if (!T || key == T->data) return T;

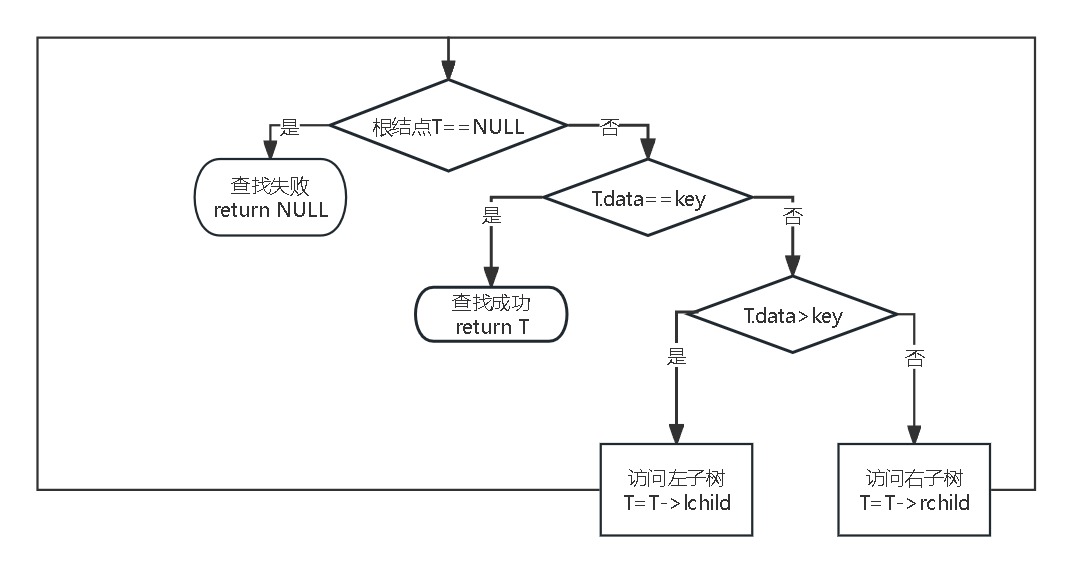
else if (key < T->data) {

//继续去左子树中查找目标元素

return SearchBST(T->lchild, key); }

else {//继续去右子树中查找目标元素

return SearchBST(T->rchild, key); }}



**图1：二叉排序树查找函数SearchBST流程图**

**（2）二叉排序树 T 中插入目标元素 e**

**InsertBST(BSTree\* T, ElemType e)**

**实现思路：**

二叉排序树中各个结点的值都不相等，因此新插入的元素一定是原二叉排序树没有的，否则插入操作会失败。此外插入新元素后，必须保证整棵树还是一棵二叉排序树。因此二叉排序树插入新元素的方法是：在树中查找新元素，最终查找失败时找到的位置，就是放置新元素的位置。

**实现代码:**

//向二叉排序树 T 中插入目标元素 e

Status InsertBST(BiTree\* T, ElemType e) {

//如果本身为空树，则直接添加 e 为根结点

if ((\*T) == NULL)

{

(\*T) = (BiTree)malloc(sizeof(BiTNode));

(\*T)->data = e;

(\*T)->lchild = NULL;

(\*T)->rchild = NULL;

return TRUE;}

//如果找到目标元素，插入失败

else if (e == (\*T)->data)

{

return FALSE; }

//如果 e 小于当前结点的值，表明应该将 e 插入到该结点的左子树中

else if (e < (\*T)->data) {

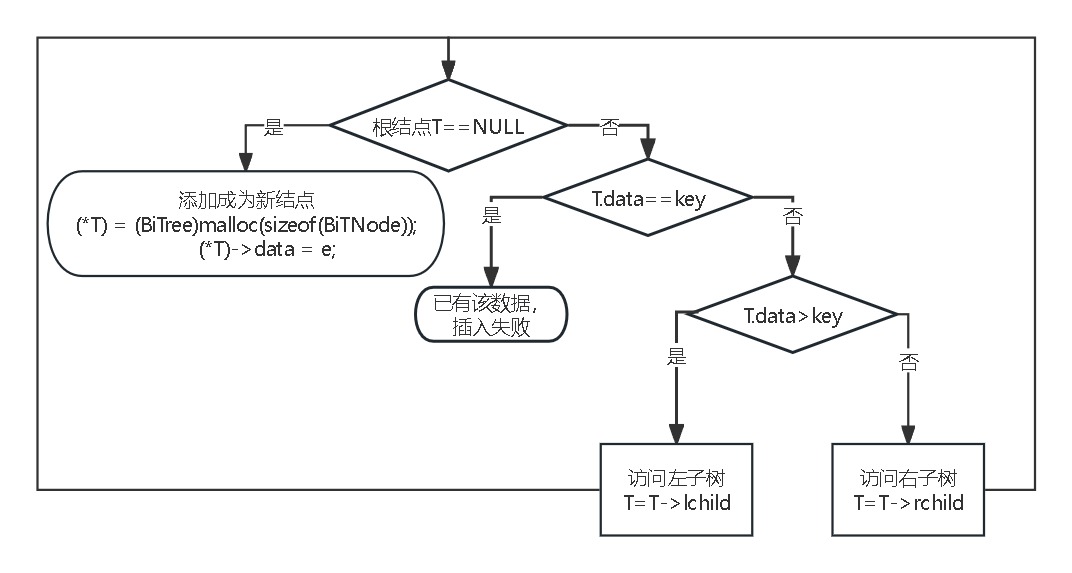
InsertBST(&(\*T)->lchild, e);

}

//如果 e 大于当前结点的值，表明应该将 e 插入到该结点的右子树中

else

{InsertBST(&(\*T)->rchild, e); }}



**图2：二叉排序树插入元素InsertBST流程图**

**（3）删除二叉排序树T中元素key**

**Status DeleteBST(BiTree\* T, int key)**

**实现思路：**

删除二叉排序树中已有的元素，必须确保整棵树还是一棵二叉排序树。假设被删除的元素所在结点是 P，根据结点 P 是否有左、右孩子，可以归结为以下 3 种情况：

1：P 是叶子结点：可以直接摘除

2：P 只有一个孩子（左孩子或右孩子）：若 P 是双亲结点（用 F 表示）的左孩子，直接将 P 的孩子结点作为 F 的左孩子；反之若 P 是 F 的右孩子，直接将 P 的孩子结点作为 F 的右孩子。

3：P 有两个孩子：中序遍历整棵二叉排序树，在中序序列里找到 P 的直接前驱结点 S，将 P 结点修改成 S 结点，然后再将之前的 S 结点从树中摘除。

在二叉排序树中，对于拥有两个孩子的结点，它的直接前驱结点要么是叶子结点，要么是没有右孩子的结点，所以删除直接前驱结点可以套用前面两种情况的实现思路。

在这里先编写了删除节点的函数Status Delete(BiTree\* p)，要删除某个结点必须先查找到该节点再利用删除节点的函数删除该结点，从而实现二叉排序树的删除。

**实现代码：**

Status Delete(BiTree\* p)

{

BiTree q = NULL, s = NULL;

//情况 1，结点 p 本身为叶子结点，右孩子也为 NULL，用 NULL 直接替换 p 结点即可

//情况 2,结点 p 只有一个孩子

if (!(\*p)->lchild) { //左子树为空，只需用结点 p 的右子树根结点代替结点 p 即可；

q = \*p;

\*p = (\*p)->rchild;

free(q);

}

else if (!(\*p)->rchild) {//右子树为空，只需用结点 p 的左子树根结点代替结点 p 即可；

q = \*p;

\*p = (\*p)->lchild;

free(q);

}

//情况 3，结点 p 有两个孩子

else {

q = \*p;

s = (\*p)->lchild;

//遍历，找到结点 p 的直接前驱，最终 s 指向的就是前驱结点，q 指向的是 s 的父结点

while (s->rchild)

{

q = s;

s = s->rchild;

}

//直接改变结点 p 的值

(\*p)->data = s->data;

//删除 s 结点

//如果 q 和 p 结点不同，删除 s 后的 q 将没有右子树，因此将 s 的左子树作为 q 的右子树

if (q != \*p) {

q->rchild = s->lchild;

}

//如果 q 和 p 结点相同，删除 s 后的 q（p） 将没有左子树，因此将 s 的左子树作为 q（p）的左子树

else {

q->lchild = s->lchild;

}

free(s);

}

return true;

}

//删除二叉排序树T中元素key

Status DeleteBST(BiTree\* T, int key)

{

//如果当前二叉排序树不存在，则找不到 key 结点，删除失败

if (!(\*T)) {

return false;

}

else

{

//如果 T 为目标结点，调用 Delete() 删除结点

if (key == (\*T)->data) {

Delete(T);

return true;

}

else if (key < (\*T)->data) {

//进入当前结点的左子树，继续查找目标元素

return DeleteBST(&(\*T)->lchild, key);

}

else {

//进入当前结点的右子树，继续查找目标元素

return DeleteBST(&(\*T)->rchild, key);

}

}

}

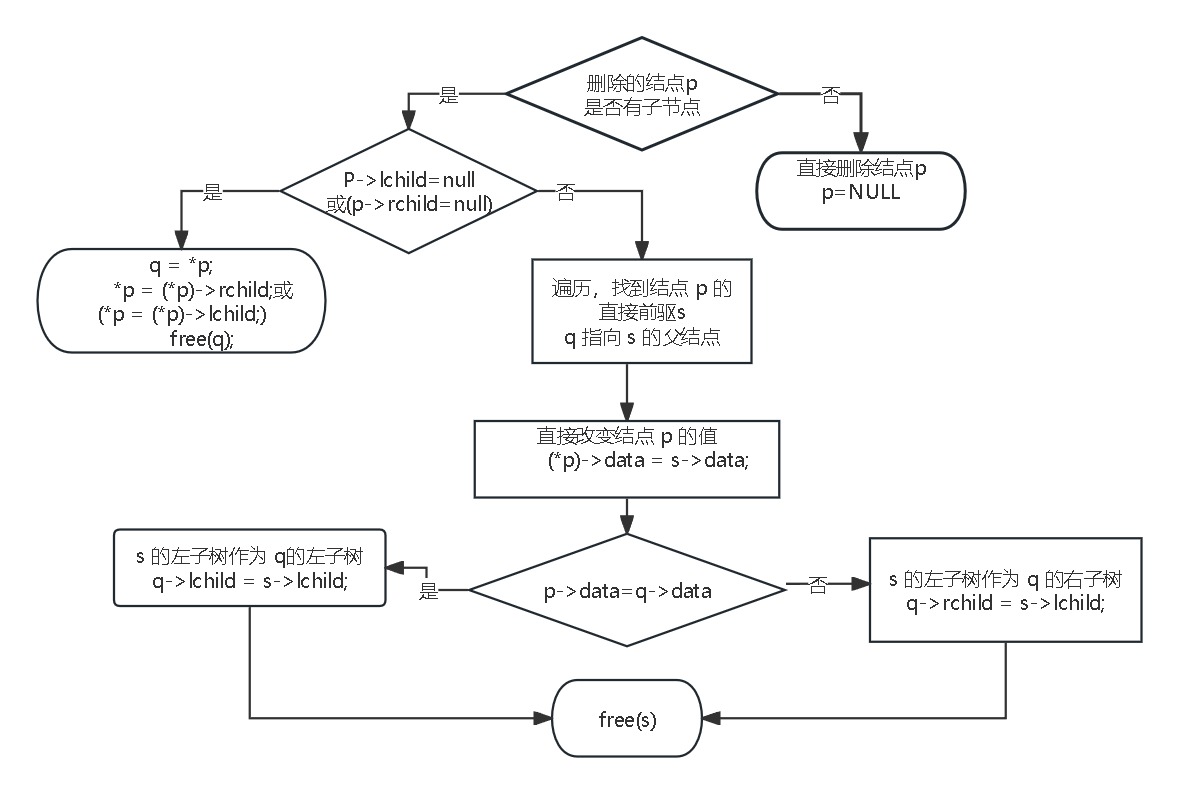


图3：**二叉排序树删除元素函数DeleteBST流程图**

**（4）销毁二叉查找树 Status destroyBST(BSTree\* root)**

**实现思路：**

不能先销毁根结点，不然就找不到左子树和右子树了，所以先释放左子树和右子树再释放根结点，后序遍历的思路。

**实现代码：**

Status destroyBST(BSTree\* root) {

if (\*root != NULL) {

// 递归地销毁左子树和右子树

destroyBST(&((\*root)->lchild));

destroyBST(&((\*root)->rchild));

// 释放当前节点

free(\*root);

\*root = NULL; // 将根指针置为 NULL，避免悬挂指针

return true; // 返回成功状态

}

return false ; // 如果 root 为空，返回错误状态

}

### 四、测试

为便于展示测试结果，会用中序遍历展示二叉排序树，得到的将会是一个有序递增数列：

//中序遍历二叉排序树

void INOrderTraverse(BiTree T) {

if (T) {

INOrderTraverse(T->lchild);//遍历当前结点的左子树

printf("%d ", T->data); //访问当前结点

INOrderTraverse(T->rchild);//遍历当前结点的右子树

}

}

**测试代码：**

int main(){

int op=0;

int f;

BSTree T=NULL;

while(op!=6){

printf("================= 二叉查找树功能 ====================\n");

printf("============[1]初始化构建一棵空二叉查找树============\n");

printf("====================[2]插入元素======================\n");

printf("====================[3]删除元素======================\n");

printf("====================[4]查找元素======================\n");

printf("====================[5]销毁二叉排序树================\n");

printf("====================[6]退出程序======================\n");

printf("请选择要进行的操作：");

scanf("%d",&op);

switch(op){

case 1:

if(InitBSTree(&T)==true) printf("[构建成功]\n\n");

break;

case 2:

printf("请输入要插入的数据 \n");

scanf("%d",&f);

Status p=InsertBST(&T, f);

if(p==false) printf("[ERROR!!!插入失败]\n");

if(p==true){printf("[插入成功]\n");

printf("中序遍历二叉排序树：[");

INOrderTraverse(T);

printf("]\n\n");

break;

case 3:

printf("请输入要删除的数据值：\n");

scanf("%d",&f);

if(DeleteBST(&T, f)){

printf("[删除成功]\n");

}else {

printf("中序遍历二叉排序树：[");

INOrderTraverse(T);

printf("]\n\n");

break;

case 4: printf("请输入要查找的数据：\n");

scanf("%d",&f);

BSTree P=SearchBST(T,f);

if(P!=NULL) printf("[成功查找到数据%d] \n",P->data);

if(P==NULL) printf("[ERROR!!!未查找到该数据]\n");

printf("中序遍历二叉排序树：[");

INOrderTraverse(T);

printf("]\n\n"); break;

case 5:

p=destroyBST(&T);

if(p==true) printf("[销毁成功]\n\n");

break;

case 6: printf("退出程序\n");break;

default: printf("[ERROR!!!输入错误]\n"); break;

}

}

} return 0;

}

测试代码中用while，switch等构造了一个菜单，通过输入不同选项可以运行各个函数测试各种操作。

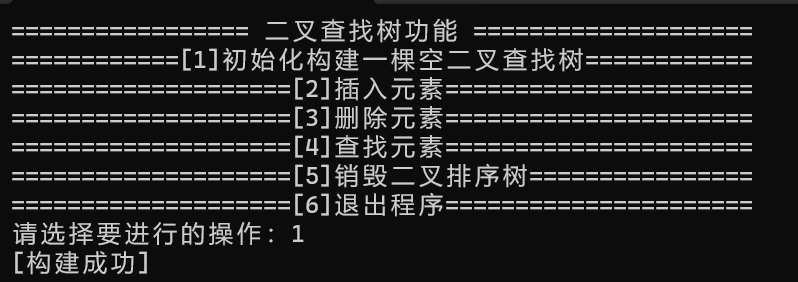
1. **初始化构造一棵空的二叉查找树T函数Status InitBSTree(BSTree\* T);**

测试代码：

case 1: if(InitBSTree(&T)==true)

printf("[构建成功]\n\n"); break;

输入选项1，调用了InitBSTree(BSTree\* T);函数，成功构建了一个空二叉树后显示构建成功



**2、二叉排序树插入元素函数InsertBST(BSTree\* T, ElemType e)**

初始化构造了一棵空树后，输入选项2，测试插入元素功能。

分别依次插入了测试数据：12，34，8，-1，-30，4，100，17，4，100，-30

**一、树本身不存在的数据：**

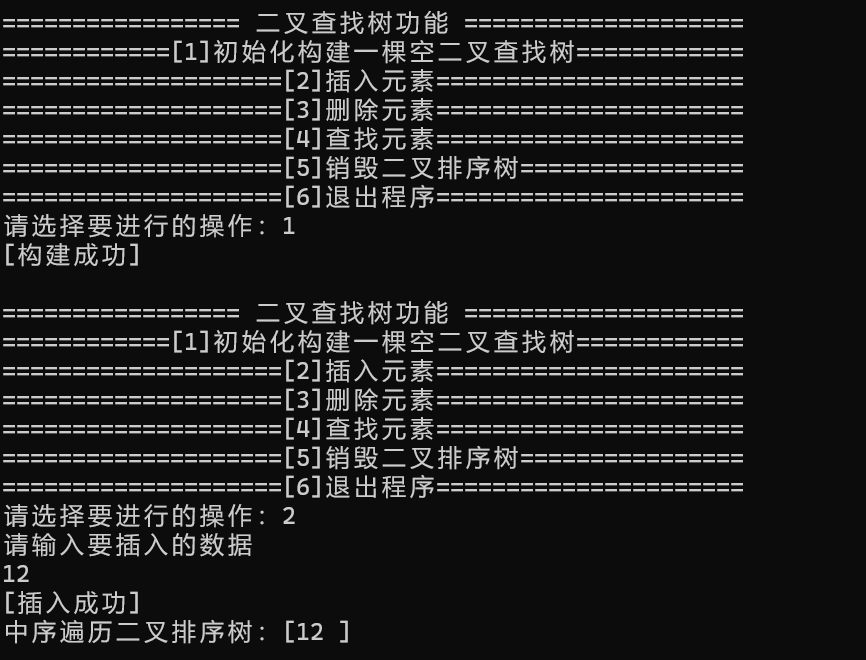
1. 12插入空树
2. 小于或大于当前树中所有元素 34,8，-1，-30,100
3. 处于居中大小的4，17

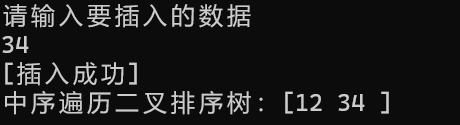
均成功插入二叉排序树中，并且中序遍历结果呈递增排列

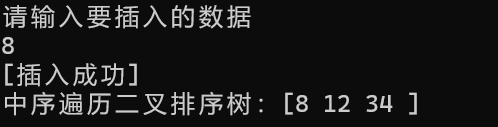
**二、插入重复的已有数据**：

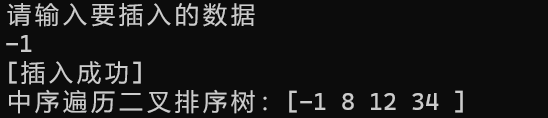
最小的-30，最大的100叶子结点处，居中大小的4。

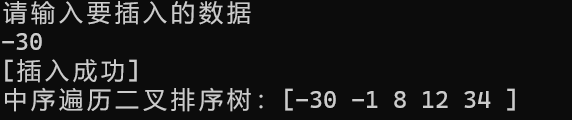
均插入失败，显示[ERROR!!!插入失败,数据已存在]返回中序遍历结果与原本相同

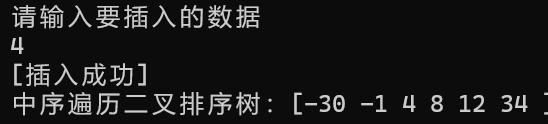


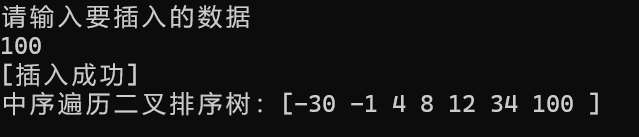


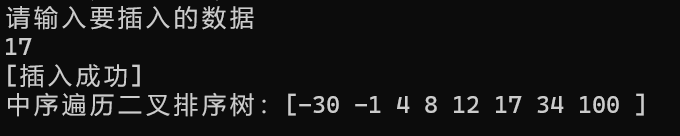


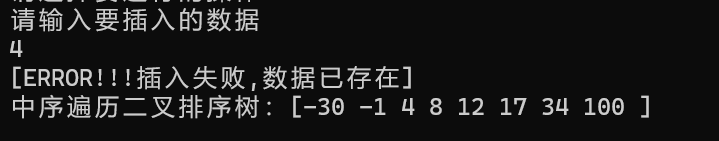


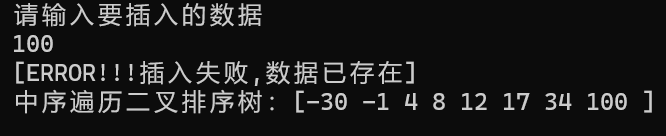


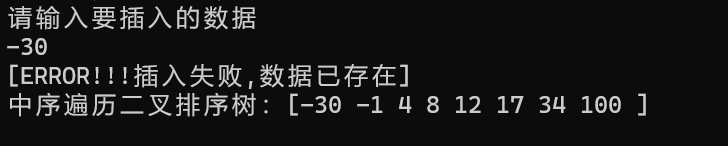












1. **二叉排序树查找元素SearchBST(BSTree T, KeyType key)**

**输入选项4，在存有元素-30 -1 4 8 12 17 34 100的二叉查找树上测试查找功能**

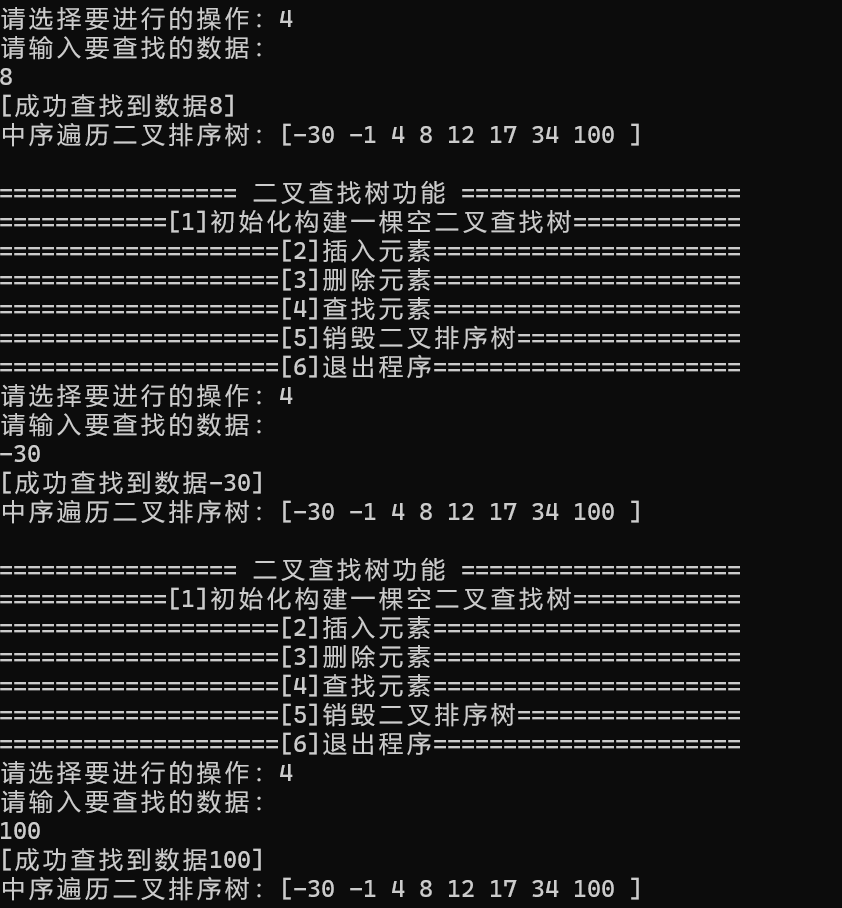
分别依次输入测试数据：8，-30,100,120，-50,18

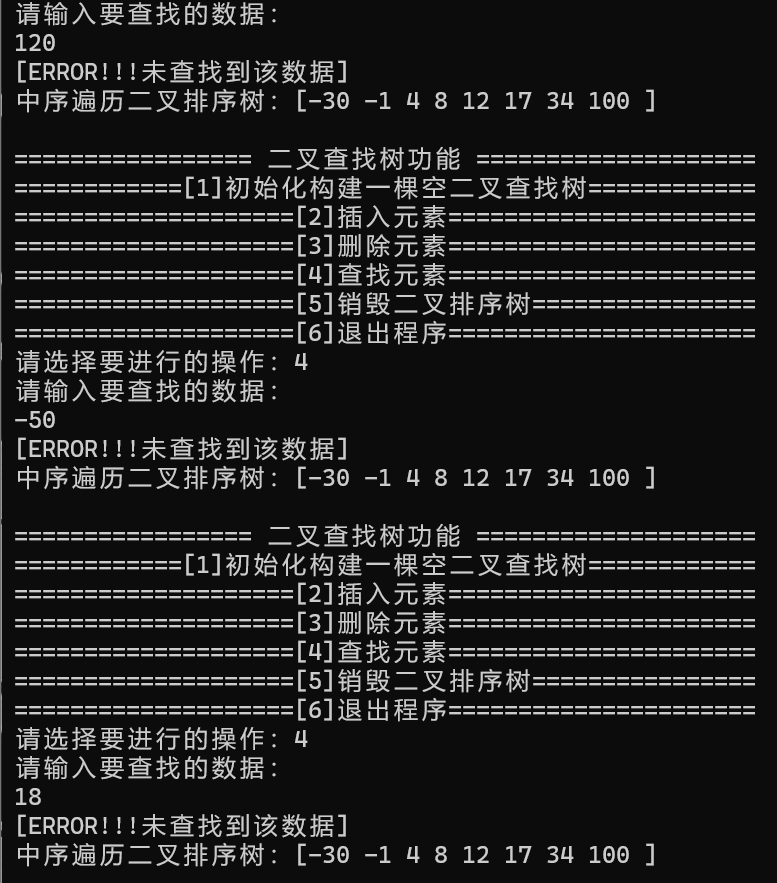
**一、查找树中已存有的数据：**

测试中间位置的含有两个子节点的数据8,叶子结点数据-30和100都成功查找到该数据

**二、查找不存在的数据：**

测试大于所有数据的120，和小于所有数据的-50和位于中间位置但不存在的数据18，都查找失败，显示未查找到该数据





1. **二叉排序树删除结点DeleteBST(BSTree & T, KeyType key)**

输入选项3调用**DeleteBST函数**，在存有数据-30 -1 4 8 12 17 34 100的二叉排序树上测试删除操作

分别依次输入测试数据：100,-1,17，99,1

**一、删除二叉查找已有的存在的数据：**

（1）测试数据100，位于叶子结点，无子节点，删除成功

（2）测试数据-1含有一个子节点，删除成功

（3）测试数据17含有两个子节点，删除成功

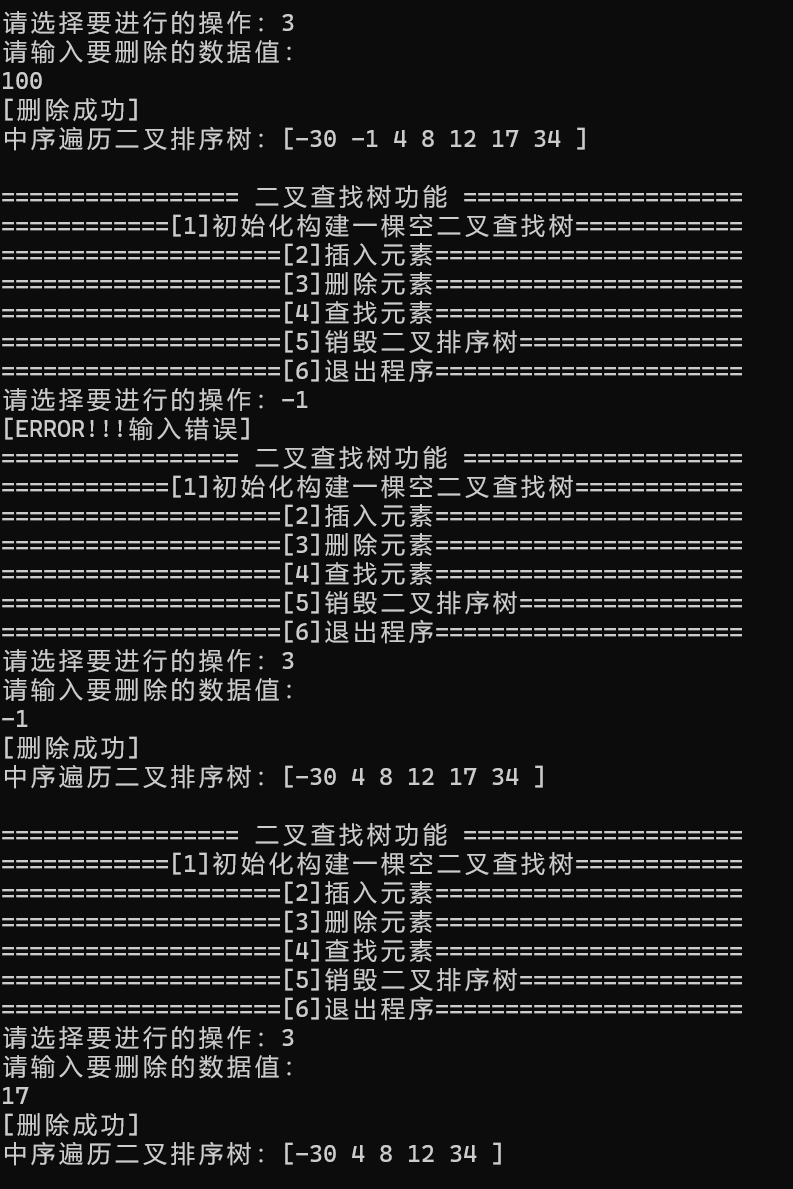
**二、.删除二叉排序树不存在的数据：**

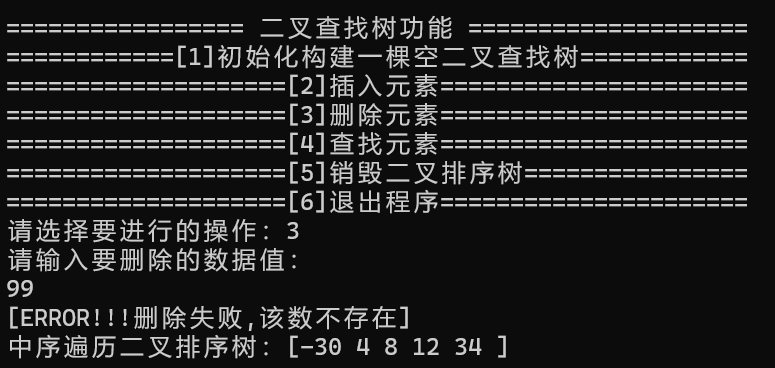
测试数据99，二叉排序树中无法找到，删除失败，显示[ERROR!!!删除失败,该数不存在]

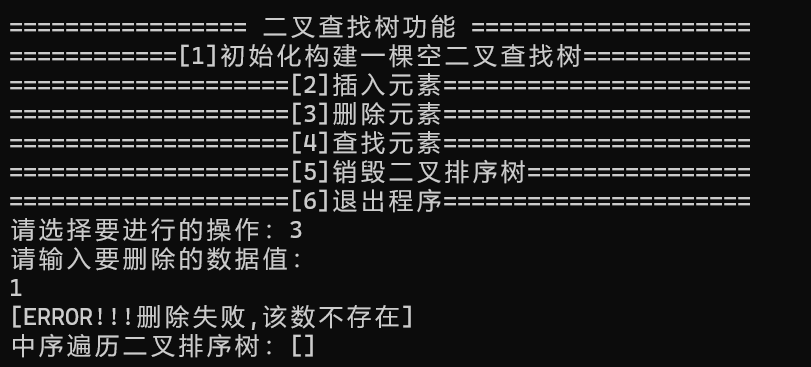
返回中序遍历结果与原本相同

**三、二叉排序树为空时：**

输入数据1，删除失败显示[ERROR!!!删除失败,该数不存在]



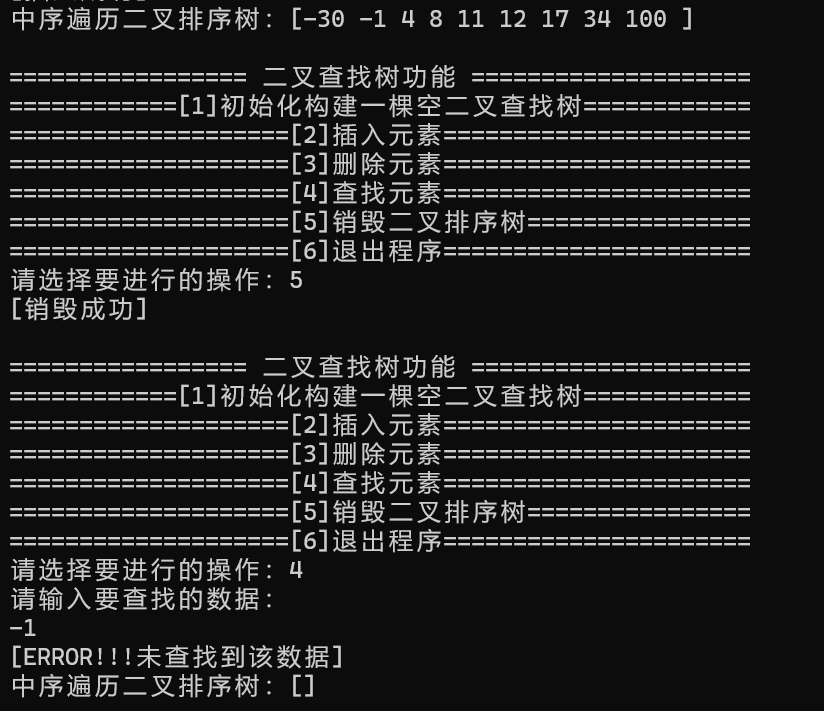




4. **销毁二叉排序树 Status destroyBST（BSTree \*root）**

在已存有数据-30 -1 4 8 11 12 17 34 100的树上，测试销毁二叉排序树操作。

输入选项5，调用destroyBST函数，屏幕显示销毁成功，这时查找原来二叉树存有的数据-1，显示[ERROR!!!未查找到该数据]，中序遍历二叉树为空，无数据显示，销毁成功



### 五、小结

本实验采用整型int为元素类型和二叉链表为存储结构，实现抽象数据类型二叉排序树，完成了二叉排序树的构建，查找，插入，删除，销毁，中序遍历操作，并通过测试验证了其正确性。在实现插入，查找，销毁函数时用到了递归，对递归调用也有了更深一步的了解。

因为二叉排序树左子树上所有结点的值均小于根节点，右子树上所有结点的值均大于根结点的特点，中序遍历二叉排序树便可得到一个有序序列，所以一个无序序列可以通过构造一棵二叉排序树变成一个有序序列。

二叉排序树是一种动态树表，其特点是树的结构通常不是一次生成的，而是在查找的过程中，当树中不存在关键字值等于给定值的结点才再进行插入，并且插入的结点一定是一个新添加的叶结点，且是查找失败时查找路径上访问的最后一个结点的左孩子或者右孩子；

二叉排序树可以高效地进行查找操作，查找效率主要取决于树的高度，查找关键字比较次数不超过树的高度。最好的情况： 当二叉排序树形态比较对称即平衡二叉树，此时与折半查找相似，时间复杂度为O(log2n)；最坏情况：即输入序列为有序数列时，形成的二叉排序树是一棵单支二叉树，此时查找退化成了单链表的查找，树的深度为n，其平均查找长度为(n + 1) / 2，时间复杂度为O(n)。二叉排序树无需移动结点，只需要修改指针指向即可完成插入和删除操作，平均执行时间为O(log2n)。；