



**数据结构设计性实验报告**

**课程名称 数据结构**

**题目名称 平衡二叉树**

**学生学院 计算机学院**

**专业班级 计算机科学与技术(2)**

**学 号 3122004665**

**学生姓名 王展锐**

**指导教师 苏 庆**

**2023 年 12 月 20 日**

1. **设计任务、要求及所用软件环境或工具；**
2. 设计任务：利用平衡二叉树实现一个动态查找表。

2、设计要求：实现动态查找表的三种基本功能：查找、插入和删除。合并两棵平衡二叉树。把一棵平衡二叉树分裂为两棵平衡二叉树，使得在一棵树中的所有关键字都小于或等于x，另一棵树中的任一关键字都大于x。

3、软件环境及工具：VS 2022

**二、抽象数据类型定义以及各基本操作的简要描述；**

1、抽象数据类型定义：

typedef int ElemType;

typedef int Status;

typedef struct ATNode

{

ElemType key;

int bf; //平衡因子

struct ATNode\* lchild, \* rchild;

}ATNode, \*ATree; //平衡二叉树

2、基本操作：

（1）void CreatAVL(ATree& T, Status& key)

操作结果：创建一棵平衡二叉树

（2）Status Insert(ATree &T,ElemType e, Status& taller)

初始条件：平衡二叉树已存在

操作结果：向平衡二叉树中插入结点

1. void Delete(ATree &T, ElemType p, Status key)

初始条件：平衡二叉树已存在且非空

操作结果：从平衡二叉树中删除结点

1. ATree find(ATree T, ElemType p)

初始条件：平衡二叉树存在

操作结果：在平衡二叉树中查找是否存在p结点

1. void printTree(ATree &T, int num)

初始条件：平衡二叉树存在

操作结果：打印一棵平衡二叉树

（6）void MergeAVL(ATree& T1, ATree& T2)

初始条件：存在两个平衡二叉树

操作结果：合并两棵平衡二叉树为一棵

1. void DivideAVL(ATree& T, ATree& TS, ATree& TB, ElemType temp)

初始条件：平衡二叉树存在

操作结果：分裂一棵平衡二叉树

（8）void destory(ATree& T)

初始条件：平衡二叉树存在

操作结果：销毁一棵平衡二叉树

**三、所选择的存储结构描述及在此存储结构上各基本操作的实现；**

**1、存储结构：**

#include <stdio.h>

#include<stdlib.h>

#define TRUE 1;

#define FALSE 0;

#define OK 1;

#define ERROR 0;

#define OVERFLOW -1;

#define LH 1; // 左子树高

#define EH 0; //两树等高

#define RH -1; //右子树高

typedef int ElemType;

typedef int Status;

typedef struct ATNode

{

ElemType key;

int bf; //平衡因子

struct ATNode\* lchild, \* rchild;

}ATNode, \*ATree; //平衡二叉树

**2、基本操作的实现**

**（1）创建一棵平衡二叉树**

void CreatAVL(ATree& T, Status& key)

{

ElemType temp;

printf("请输入您需要构建的AVL树的结点值：\n");

printf("tips：请按层次输入，输入的结点值必须为非负整数，输入-1结束输入。\n");

printf(" (输入一个数值后可使用回车键换行）\n");

scanf\_s("%d", &temp);

while (temp !=- 1)

{

if (temp < 0)

{

system("cls");

printf("您的输入有误！程序终止！\n");

break;

}

else

{

if (Insert(T, temp, key) == 1) {

scanf\_s("%d", &temp);

}

else

{

system("cls");

printf("您的输入有误！程序终止！\n");

break;

}

}

}

}

1. 失衡调节

//右旋

void R\_Rotate(ATree &T)

{

ATree Lchild;

Lchild = T->lchild; //Lchild指向T结点的左孩子

T->lchild = Lchild->rchild; //Lchild结点的右孩子置为T结点的左孩子

Lchild->rchild = T; //置T结点为Lchild的右孩子

T = Lchild; //T指向新的根结点

}

//左旋

void L\_Rotate(ATree& T)

{

ATree Rchild;

Rchild = T->rchild; //Rchild指向T的右孩子

T->rchild = Rchild->lchild; //Rchild结点的左孩子置为T结点的右孩子

Rchild->lchild = T; //置T结点为Rchild结点的左孩子

T = Rchild; //T指向新的根结点

}

//对树T的左平衡处理

void LBalance(ATree& T)

{

ATree Lc, Rd; //Lc指向T的左孩子 Rd指向Lc的右子树

Lc = T->lchild;

switch (Lc->bf)

{

case 1: //Lc的左树高 LL型 右旋处理

T->bf = EH;

Lc->bf = EH;

R\_Rotate(T);

break;

case 0: //Lc左右等高 T树在插入前已经失衡 LL型 右旋处理

T->bf = LH;

Lc->bf = RH;

R\_Rotate(T);

break;

case -1: //新插入结点在T左孩子的右子树上 LR型 先左旋 再右旋

Rd = Lc->rchild;

switch (Rd->bf) { //修改T及其左孩子的平衡因子

case 1:

T->bf = RH;

Lc->bf = EH;

break;

case 0:

T->bf = EH;

Lc->bf = EH;

break;

case -1:

T->bf = EH;

Lc->bf = LH;

break;

}

Rd->bf = EH;

L\_Rotate(T->lchild); //对T的左孩子做左旋调整

R\_Rotate(T); //对T做右旋调整

break;

}

}

//对树T的右平衡处理

void RBalance(ATree& T)

{

ATree Rc, Ld;

Rc = T->rchild; //Rc指向T的右孩子

switch (Rc->bf)

{

case -1: //RR型 左旋

T->bf = EH;

Rc->bf = EH;

L\_Rotate(T);

break;

case 0:

T->bf = RH; //树T在插入之前就已失衡 T属RR型 左旋

Rc->bf = LH;

L\_Rotate(T);

break;

case 1: //新结点插入在T的右孩子的左子树上 RL型 先右旋再左旋

Ld = Rc->lchild;

switch (Ld->bf) //修改T及其右孩子的左平衡因子

{

case 1:

T->bf = EH;

Rc->bf = RH;

break;

case 0:

T->bf = EH;

Rc->bf = EH;

break;

case -1:

T->bf = RH;

Rc->bf = EH;

break;

}

Ld->bf = EH;

R\_Rotate(T->rchild);

L\_Rotate(T);

break;

}

}

（3）平衡二叉树的插入

//key用于记录由于插入操作是否导致树的高度增加

Status Insert(ATree &T,ElemType e, Status& taller) {

if (e < 0) printf("输入有误！");

else {

if (!T) { //T为空，树长高

T = (ATree)malloc(sizeof(ATNode));

if (!T) return OVERFLOW;

T->key = e;

T->bf = EH;

T->lchild = NULL;

T->rchild = NULL;

taller = TRUE;

}

else if (e == T->key) { //树中已存在和e相等的结点

taller = FALSE;

printf("树中已存在和e相等的结点！");

return FALSE; //未插入

}

else if (e < T->key) { //插入到左子树

if (0== Insert(T->lchild, e, taller)) return FALSE; //未插入

if (taller) {

switch (T->bf) { //检查T的平衡因子

case 1: //原左高，左平衡处理

LBalance(T);

taller = FALSE;

break;

case 0: //原等高，左变高

T->bf = LH;

taller = TRUE;

break;

case -1: //原右高，变等高

T->bf = EH;

taller = FALSE;

break;

}

}

}

else { //插入到右子树

if (0 == Insert(T->rchild, e, taller)) return FALSE;

if (taller) {

switch (T->bf) { //检查平衡因子

case 1: //原本左高，变等高

T->bf = EH;

taller = FALSE;

break;

case 0: //原本等高，变右高

T->bf = RH;

taller = TRUE;

break;

case -1: //原本右高，右平衡处理

RBalance(T);

taller = FALSE;

break;

}

}

}

return TRUE;

}

}

（4）平衡二叉树的结点删除//key用于判断删除是否成功

void Delete(ATree &T, ElemType p, Status key)

{

ATNode \*q, \* s;

q = T;

if (T->lchild == NULL)//被删除结点左子树为空，右子树代替被删除结点

{

T = T->rchild;

free(q);

key = TRUE;

}

else if (T->rchild == NULL)//被删除结点右子树为空，左子树代替被删除结点

{

T = T->lchild;

free(q);

key = TRUE;

}

else {

s = T->lchild;

while (s->rchild != NULL) {//找到左子树中最大的结点作为前驱结点

q = s;

s = s->rchild;

}

ElemType data;

data = s->key;

DelAVL(T, data, key);//将前驱结点的值复制给被删除结点，并将前驱结点删去

T->key = data;

}

}

//AVL的结点删除

Status DelAVL(ATree& T, ElemType p, Status key)

{

if (T == NULL) {

return FALSE;

}

else if (T->key == p) //删除根节点

{

Delete(T, T->key, key);

return TRUE;

}

else if (p < T->key) //删除结点在左子树

{

if (!DelAVL(T->lchild, p, key)) return FALSE;

if (key)

{

switch (T->bf)

{

case 1: //原先左高，变等高

T->bf = EH;

key = TRUE;

break;

case 0: //原先等高，变右高

T->bf = RH;

key = FALSE;

break;

case -1: //原先右高，需右平衡处理

if (!T->rchild || T->rchild->bf == 0) {

key = FALSE;

}

else

key = TRUE;

RBalance(T);

break;

}

}

}

else { //需删除结点在右子树

if (!DelAVL(T->rchild, p, key))

{

return FALSE;

}

if (key)

{

switch (T->bf)

{

case 1: //原左高，若高度降低则需进行左平衡处理

if (!T->lchild || T->lchild->bf == 0)

{

key = FALSE;

}

else

{

key = TRUE;

LBalance(T);

break;

}

case 0: //原等高，变左高

T->bf = LH;

key = FALSE;

break;

case -1: //原右高，变等高

T->bf = EH;

key = TRUE;

break;

}

}

}

}

（5）平衡二叉树的查找

ATree find(ATree T, ElemType p) //查找类型与ATree相同

{

if (T == NULL)

return NULL;

if (T->key == p)

return T;

if (p < T->key) //查找值在T的左子树

return find(T->lchild, p);

return find(T->rchild, p); //查找值在T的右子树

}

（6）打印平衡二叉树

void printTree(ATree &T, int num)

{

if (T == NULL)

return;

printTree(T->rchild, num + 1);

for (int i = 0; i < num; i++)

printf(" ");

printf("%d\n", T->key);

printTree(T->lchild, num + 1);

}

（7）销毁平衡二叉树

void destory(ATree& T)

{

if (T) {

destory(T->lchild);

destory(T->rchild);

free(T);

}

}

（8）构造一棵平衡二叉树

void CreatAVL(ATree& T, Status& key)

{

ElemType temp;

printf("请输入您需要构建的AVL树的结点值：\n");

printf("tips：请按层次输入，输入的结点值必须为非负整数，输入-1结束输入。\n");

printf(" (输入一个数值后可使用回车键换行）\n");

scanf\_s("%d", &temp);

while (temp !=- 1)

{

if (temp < 0)

{

system("cls");

printf("您的输入有误！程序终止！\n");

break;

}

else

{

if (Insert(T, temp, key) == 1) {

scanf\_s("%d", &temp);

}

else

{

system("cls");

printf("您的输入有误！程序终止！\n");

break;

}

//system("cls");

}

}

//printf("构建成功!\n");

}

（9）合并两棵二叉平衡树

void MergeAVL(ATree& T1, ATree& T2)

{

if (T2 == NULL)

return;

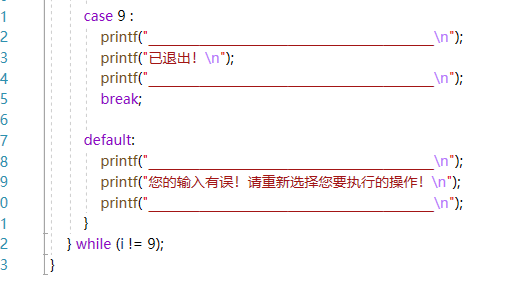
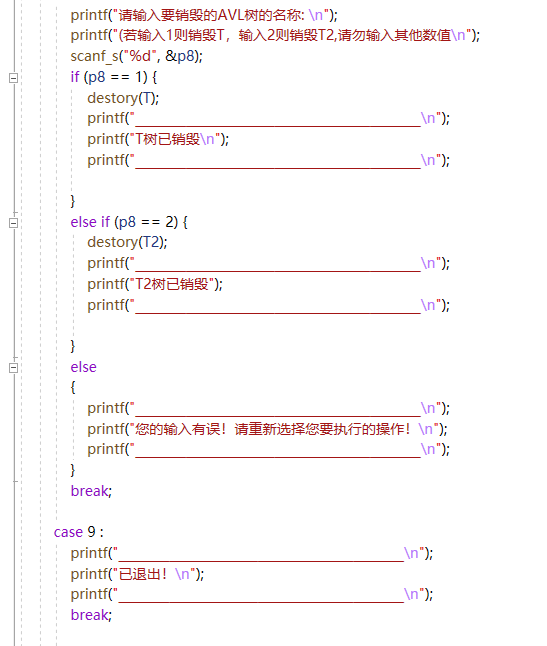
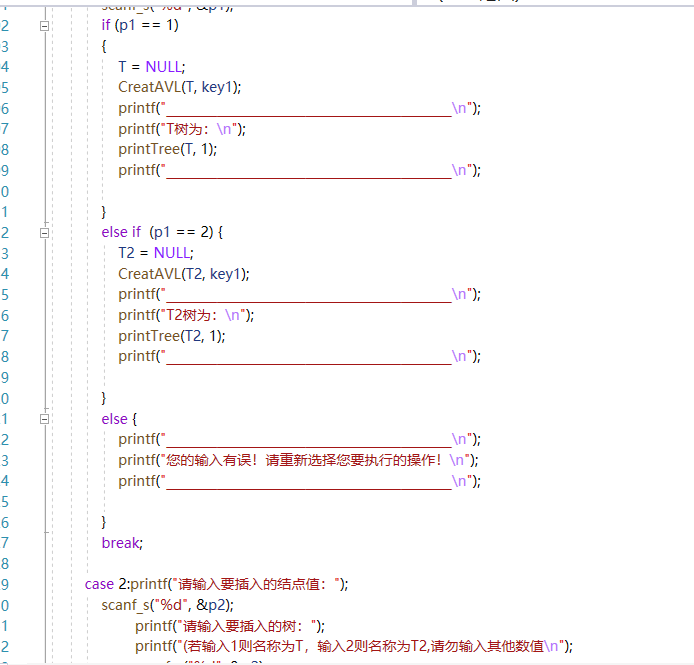
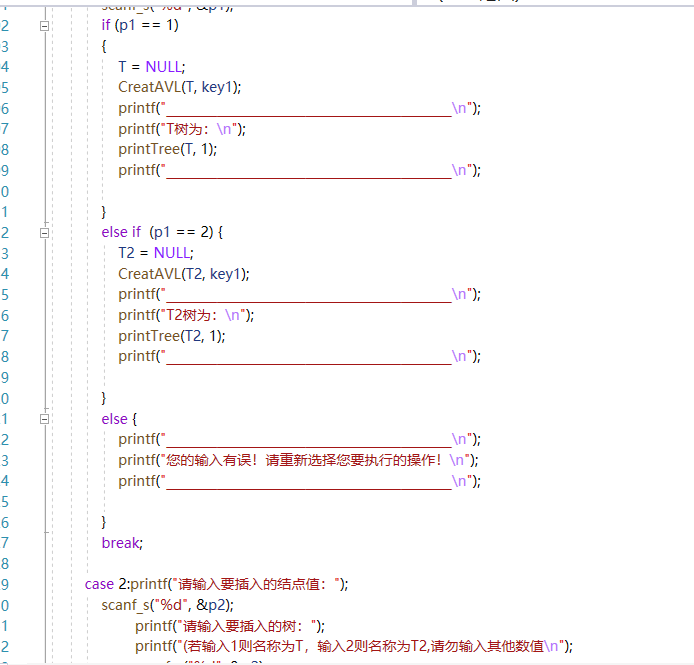
Status key = FALSE;

MergeAVL(T1, T2->lchild);

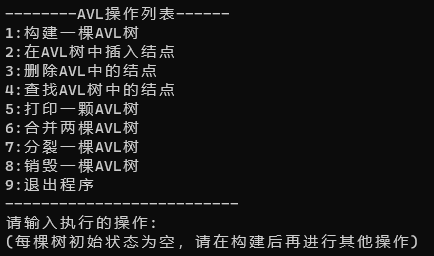
Insert(T1, T2->key, key);

MergeAVL(T1, T2->rchild);

}

1. **程序清单（计算机打印），输入的数据及各基本操作的测试结果；**
2. 程序清单：****
3. **测试**

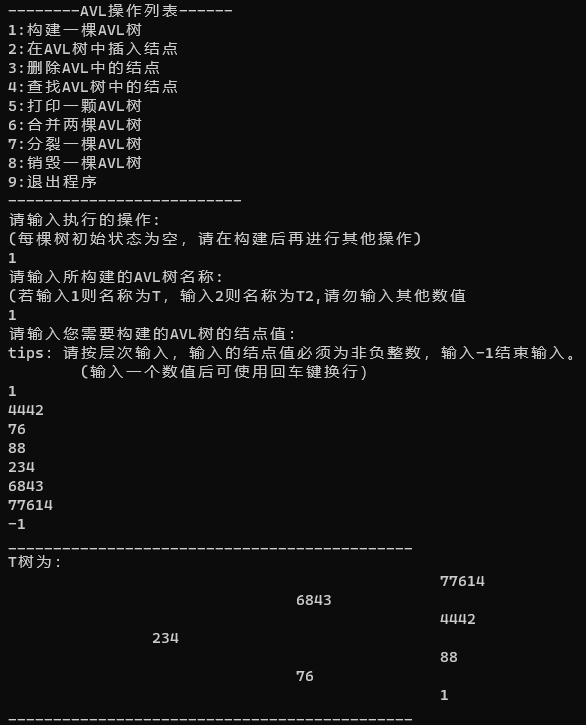
**主界面**

****

创建平衡二叉树并打印

选择操作1后进入创建流程

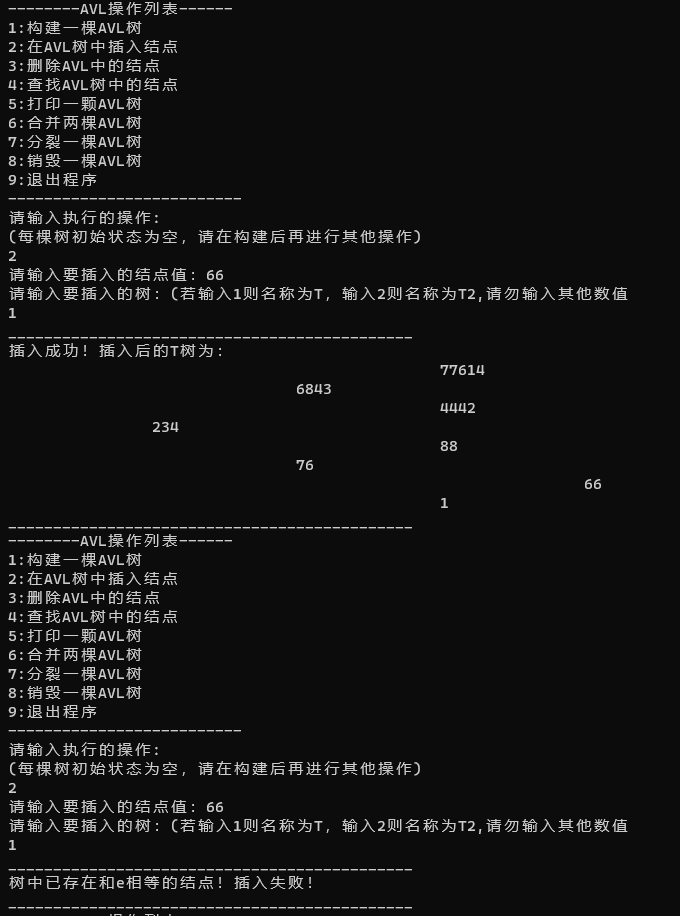
输入1创建树T，输入2创建树T2，输入-1结束构建



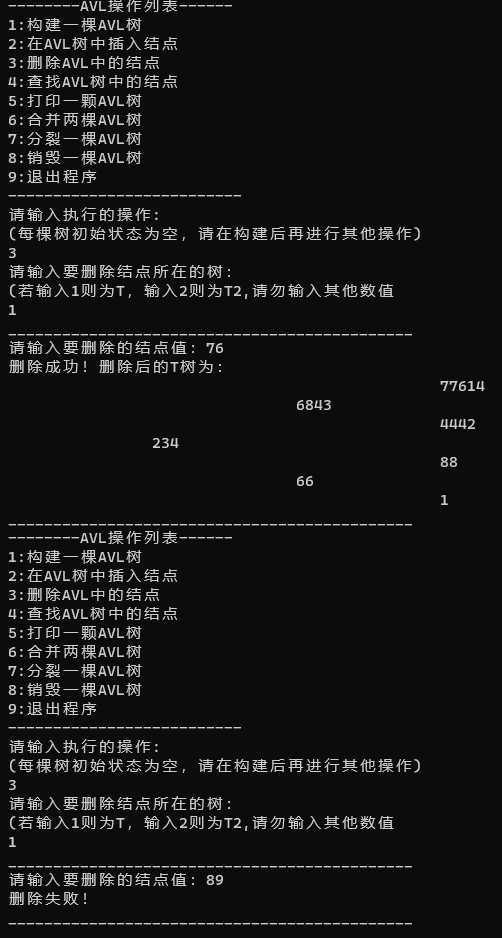
平衡二叉树的插入操作

选择操作2进入插入流程

输入插入结点值，若已存在则会退出。输入之后选择插入的树的名称。

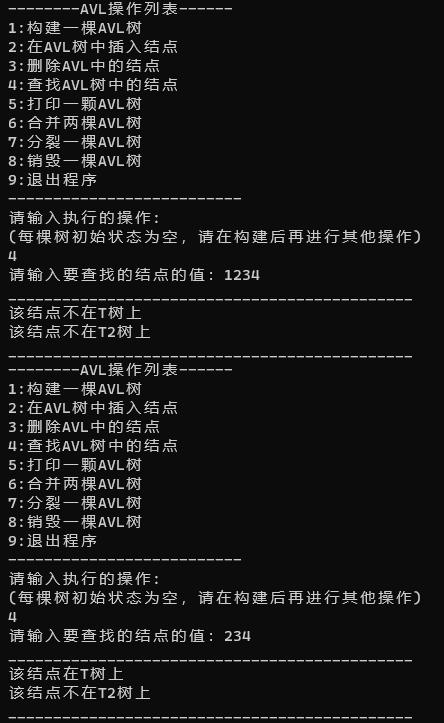


平衡二叉树的删除操作

选择操作3进入删除操作选择删除结点所在的树。输入删除结点值，若不存在则删除失败，退出。

平衡二叉树的查找操作

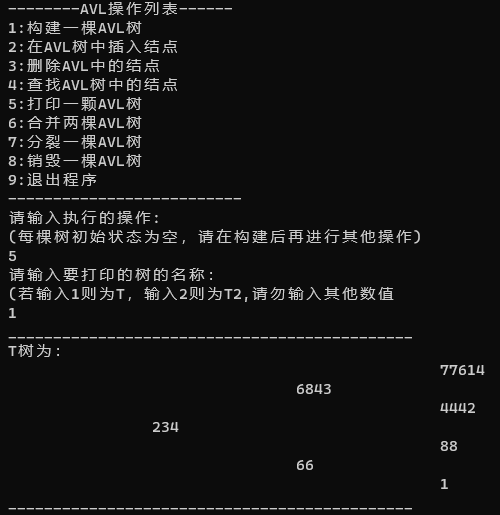
选择操作4进入查找操作



打印平衡二叉树

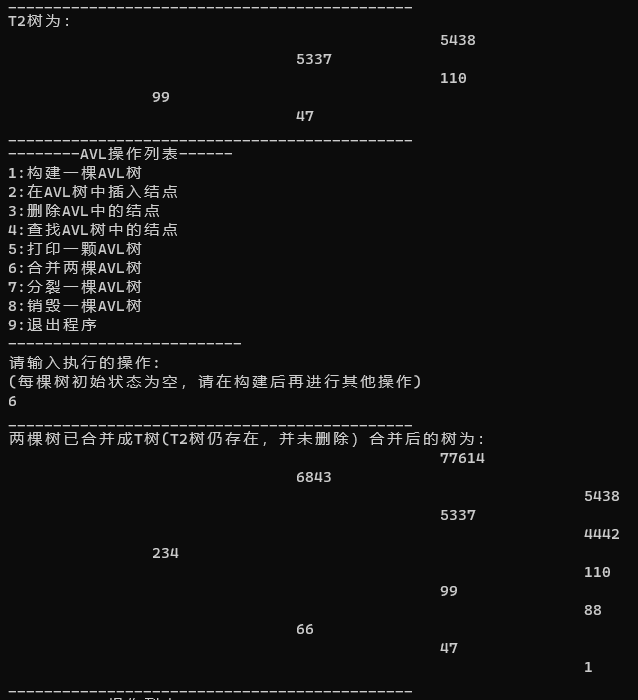
选择操作5进行打印

选择打印哪棵平衡二叉树



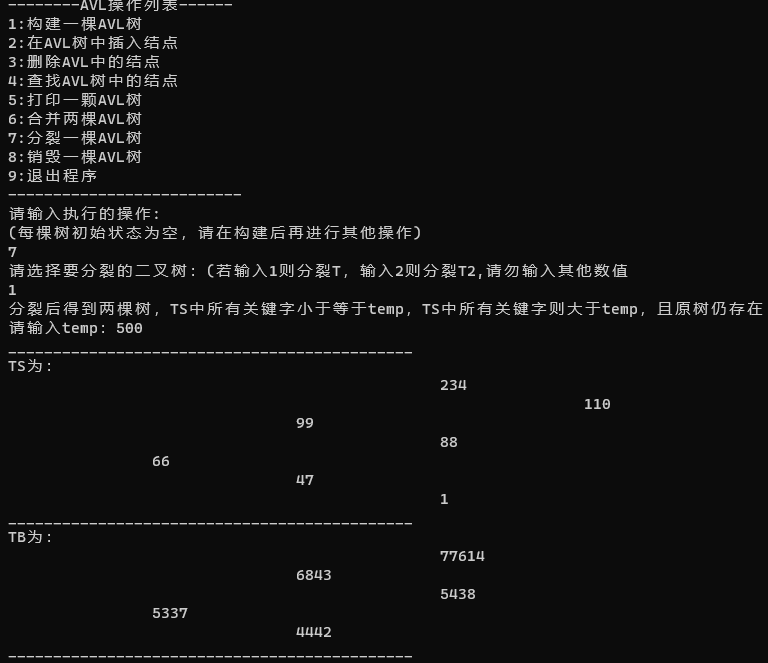
平衡二叉树的合并

选择操作6进行合并，两棵树合并为T



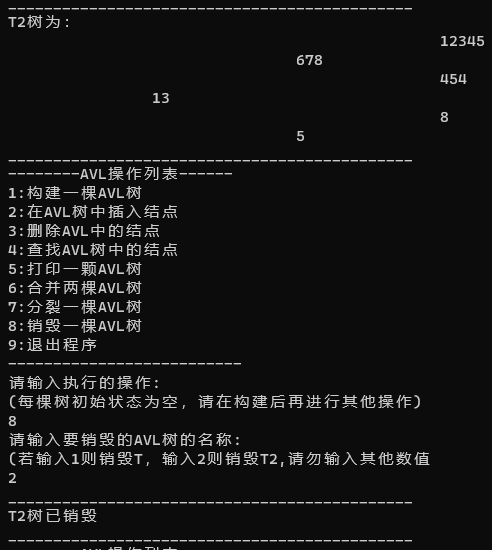
平衡二叉树的分裂

选择操作7进行分裂，选择分裂哪棵平衡二叉树，输入temp值作为分界值，得到一棵所有结点小于等于temp的平衡二叉树TS，一棵所有结点大于temp值的平衡二叉树TB



销毁平衡二叉树

输入操作8进行销毁，选择销毁哪棵树



**五、实验总结和体会。**

1、平衡二叉树的本质上是一种特殊的二叉搜索树。它的左右子树之间的高度差不超过1，从而使得查找的效率提高。因此构建二叉树时，应尽量使得这棵树变“肥”，这便是设计平衡二叉树的初衷。

2、平衡二叉树在增加和删除的过程中会造成高度的改变，有可能会导致失衡。这时候便需要通过旋转重新调整。

3、在设计算法的过程中，多次使用了递归的思想，使用递归可以更加方便的调整平衡二叉树，增强程序的可读性，方便调试，但也导致了时间和空间的消耗较大。

4、编写代码的过程中一定要注意养成良好的编程风格。代码需要规范，这不仅方便代码的阅读，对代码的维护也是增益多多。在编程的过程中，需要在合适的地方作注释，提高阅读效率。

5、写出来的程序一定要人性化，同时还需有防止错误输入的bug出现，提高程序的健壮性。尽管在编写过程中未采用easyx库等制作图形界面，也应制作出清晰易懂，方便操作的菜单窗口。