



**设 计 性 实 验**

课程名称 数据结构

题目名称 平衡二叉树(难度1.3)

学生学院 计算机学院

专业班级 网络工程1602

学 号 3116004982

学生姓名 赵舒宇

指导教师 李杨

2017 年 12 月 19日

**一.绪论**

**1.1 平衡二叉树的定义**

平衡二叉树它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树，且设x为树中的一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则key[y] <= key[x]；如果y是x的右子树的一个结点，则key[y] >= key[x]。

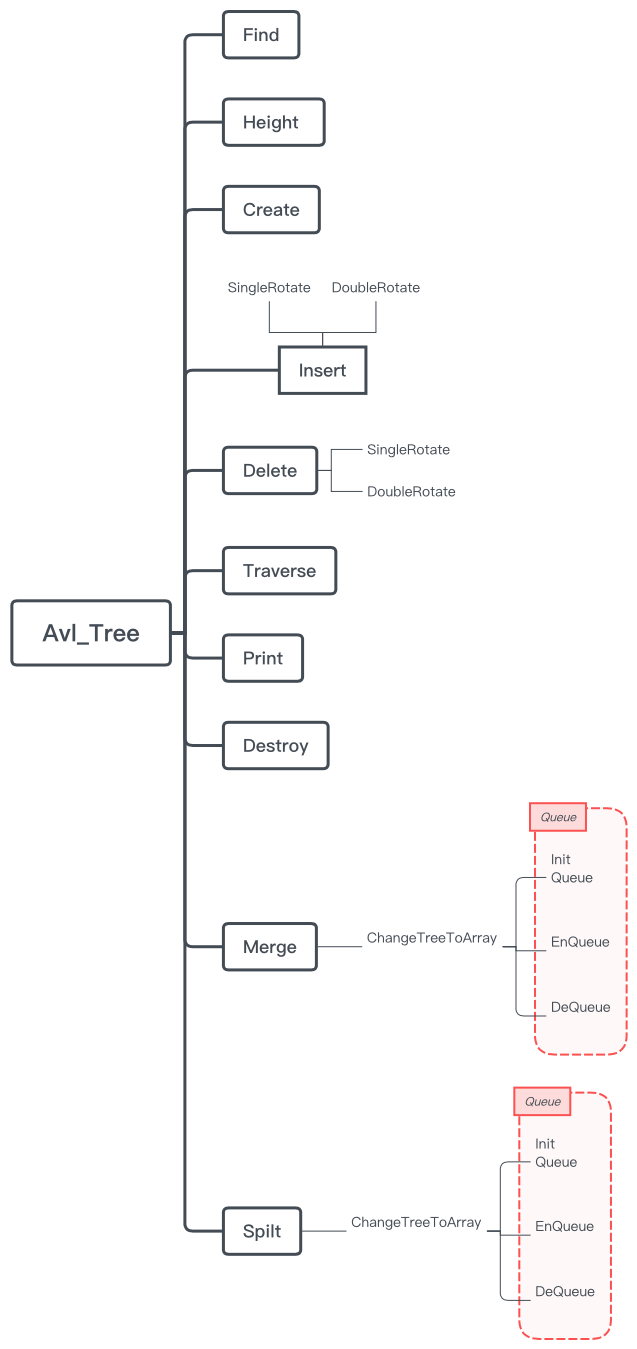
**1.2编程环境与配置**

IDE：Code::Blocks

Language：C

**二.平衡二叉树的具体实现**

**2.1程序结构**

****

**2.2平衡二叉树的抽象数据类型定义**

ADT AvlTree{

**数据对象**：D＝{ ai | ai∈ElemSet, i=1,2,...,n, n≥0 }

**数据关系**：R1＝{ <ai-1, ai>|ai-1, ai∈D, i=2,...,n }

**基本操作**：

AvlTree\_Min(**&**T)

初始条件：平衡二叉树T已存在。

操作结果：寻找并返回二叉树中最小的结点

AvlTree\_Max(**&**T)

初始条件：平衡二叉树T已存在。

操作结果：寻找并返回二叉树中最大的结点

SingleRotate\_Left(**&**T)

初始条件：平衡二叉树T已存在且失衡(LL情况)。

操作结果：旋转使其恢复平衡，并返回恢复平衡后的根节点

SingleRotate\_Right(**&**T)

初始条件：平衡二叉树T已存在且失衡(RR情况)。

操作结果：旋转使其恢复平衡，并返回恢复平衡后的根节点

DoubleRotate\_Left(**&**T)

初始条件：平衡二叉树T已存在且失衡(LR情况)。

操作结果：旋转使其恢复平衡，并返回恢复平衡后的根节点

DoubleRotate\_Left(**&**T)

初始条件：平衡二叉树T已存在且失衡(RL情况)。

操作结果：旋转使其恢复平衡，并返回恢复平衡后的根节点

AvlTree\_Find(x,**&**T)

初始条件：平衡二叉树T已存在

操作结果：寻找并返回二叉树中值为x的结点

AvlTree\_Create(**&**T)

操作结果：创建并返回一个值为x的二叉树结点，其左右孩子均为空。

AvlTree\_Insert(x,**&**T)

初始条件：平衡二叉树T已存在。

操作结果：将值为x的结点插入二叉树中，必要时旋转使其恢复平衡。

返回插入完成后的根节点

AvlTree\_Delete(x,**&**T)

初始条件：平衡二叉树T已存在。

操作结果：将值为x的结点从二叉树中删除，必要时旋转使其恢复平衡。

返回删除完成后的根节点

AvlTree\_PreOrderTraverse (**&**T)

初始条件：平衡二叉树T已存在。

操作结果：先序遍历该二叉树并打印。

AvlTree\_InOrderTraverse (**&**T)

初始条件：平衡二叉树T已存在。

操作结果：中序遍历该二叉树并打印。

AvlTree\_PostOrderTraverse (**&**T)

初始条件：平衡二叉树T已存在。

操作结果：后序遍历该二叉树并打印。

AvlTree\_Print(**&**T，dep)

初始条件：平衡二叉树T已存在。

操作结果：以凹入表形式打印该二叉树。

AvlTree\_Destroy(**&**T，dep)

初始条件：平衡二叉树T已存在。

操作结果：销毁该二叉树。

AvlTree\_Merge(**&**T1，**&**T2)

初始条件：平衡二叉树T1，T2已存在。

操作结果：合并这两个二叉树，返回合并后的二叉树的根节点。

AvlTree\_Spilt(**&**T，x，**&**T1，**&**T2)

初始条件：平衡二叉T已存在。

操作结果：将T分裂，分裂后T2中结点的值均大于x，T1中结点的值均

小于等于x。

} ADT AvlTree

**2.3头文件**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE 0

#define OVERFLOW -2

typedef int Status;

typedef int ElementType; //元素类型为整形类型

/\*储存平衡二叉树结构声明\*/

typedef struct AvlNode{

ElementType element; //数据域

struct AvlNode \*left; //左右孩子指针域

struct AvlNode \*right;

int height; //树高

}AvlNode,\*AvlTree;

/\*存放输入数据的数组结构体\*/

typedef struct ArrayNode{

ElementType element; //存放记录的结点元素值

ArrayNode \*next; //指向下一个结点

}ArrayNode, \*Array;

/\*链队列结构体\*/

typedef struct LQNode{

AvlTree element; //存放遍历时树的指针

struct LQNode \*next; //指向下一个结点

}LQNode, \*QueuePtr;

/\*队列结点结构体\*/

typedef struct{

QueuePtr front; //队头指针

QueuePtr rear; //队尾指针

}LQueue;

/\*初始化链队列\*/

Status LQueue\_Init(LQueue &Q);

/\*链队列进队操作\*/

Status LQueue\_EnQueue(LQueue &Q, AvlTree &T);

/\*链队列出队操作\*/

Status LQueue\_DeQueue(LQueue &Q, AvlTree &T);

/\*求树高\*/

int AvlTree\_Height(AvlTree &T);

/\*比较取最大树高\*/

int MAX(int T1,int T2);

/\*寻找最小结点\*/

AvlTree AvlTree\_Min(AvlTree &T);

/\*寻找最大结点\*/

AvlTree AvlTree\_Max(AvlTree &T);

/\*右旋LL\*/

AvlTree SingleRotate\_Left(AvlTree &T2);

/\*左旋RR\*/

AvlTree SingleRotate\_Right(AvlTree &T2);

/\*先左旋再右旋LR\*/

AvlTree DoubleRotate\_Left(AvlTree &T3);

/\*先右旋再左旋RL\*/

AvlTree DoubleRotate\_Right(AvlTree &T3);

/\*寻找结点\*/

AvlTree AvlTree\_Find(ElementType x,AvlTree &T);

/\*创建结点\*/

AvlTree AvlTree\_Create(ElementType x,AvlNode \*left,AvlNode \*right);

/\*插入结点\*/

AvlTree AvlTree\_Insert(ElementType x,AvlTree &T);

/\*删除结点\*/

AvlTree AvlTree\_Delete(ElementType X,AvlTree &T);

/\*先序遍历\*/

Status AvlTree\_PreOrderTraverse(AvlTree &T);

/\*中序遍历\*/

Status AvlTree\_InOrderTraverse(AvlTree &T);

/\*后序遍历\*/

Status AvlTree\_PostOrderTraverse(AvlTree &T);

/\*销毁\*/

Status AvlTree\_Destroy(AvlTree &T);

/\*创建\*/

AvlTree Create(AvlTree &T);

/\*将树转化为数组\*/

Array ChangeTreeToArray(AvlTree &T)；

/\*合并\*/

AvlTree AvlTree\_Merge(AvlTree &T1,AvlTree &T2);

/\*分裂\*/

Status AvlTree\_Spilt(AvlTree &T,ElementType x,AvlTree &T1,AvlTree &T2);

/\*打印\*/

Status AvlTree\_Print(AvlTree &T, int dep);

/\*测试\*/

Status AvlTree\_Test(AvlTree &T1,AvlTree &T2);

**2.4平衡二叉树具体接口实现**

**2.4.1 AvlTree\_Height函数**

**功能:**求树高

**代码实现:**

int AvlTree\_Height(AvlTree &T){

if(NULL==T)

return 0;

else

return T->height;

}

**2.4.2 MAX函数**

**功能:**比较取最大树高

**代码实现:**

int MAX(int T1,int T2){

return T1>T2 ? T1 : T2;

}

**2.4.3 AvlTree\_Min函数**

**功能:** 寻找最小结点

**代码实现:**

AvlTree AvlTree\_Min(AvlTree &T){

if(NULL==T)

return NULL;

while(T->left!=NULL)

T=T->left; //找到了树最左下角的结点即为最小结点

return T;

}

**2.4.4 AvlTree\_Max函数**

**功能:** 寻找最大结点

**代码实现:**

AvlTree AvlTree\_Max(AvlTree &T){

if(NULL==T)

return NULL;

while(T->right!=NULL)

T=T->right; //找到了树最右下角的结点即为最大结点

return T;

}

**2.4.5 SingleRotate\_Left函数**

**功能:** 右旋操作

**代码实现:**

AvlTree SingleRotate\_Left(AvlTree &T2){

AvlTree T1=(AvlNode\*)malloc(sizeof(AvlNode));

T1=T2->left; //暂存T2的左孩子

T2->left=T1->right; //将T2的左孩子置为其原左孩子的右孩子(如必要)

T1->right=T2; //将T1的右孩子置为修正后的T2

T2->height=MAX(AvlTree\_Height(T2->left),AvlTree\_Height(T2->right))+1;

T1->height=MAX(AvlTree\_Height(T1->left),T2->height)+1; //修正树高

return T1; //此时T1更新为原根节点(T2)

}

**2.4.6 SingleRotate\_Right函数**

**功能:** 左旋操作

**代码实现:**

AvlTree SingleRotate\_Right(AvlTree &T2){

AvlTree T1=(AvlNode\*)malloc(sizeof(AvlNode));

T1=T2->right;

T2->right=T1->left;

T1->left=T2;

T2->height=MAX(AvlTree\_Height(T2->left),AvlTree\_Height(T2->right))+1;

T1->height=MAX(AvlTree\_Height(T1->right),T2->height)+1;

return T1; //LL情况的镜像处理

}

**2.4.7 DoubleRotate\_Left函数**

**功能:** 先左旋再右旋

**代码实现:**

AvlTree DoubleRotate\_Left(AvlTree &T3){

T3->left=SingleRotate\_Right(T3->left); //对T3的左孩子执行左旋操作(此时T3的左孩子变为了原左孩子的右孩子)

return SingleRotate\_Left(T3); //对修正后的T3在执行一次右旋操作(此时根节点变为T3的左孩子，即原来的左孩子的右孩子)

}

**2.4.8 DoubleRotate\_Right函数**

**功能:** 先右旋再左旋

**代码实现:**

AvlTree DoubleRotate\_Right(AvlTree &T3){

T3->right=SingleRotate\_Left(T3->right);

return SingleRotate\_Right(T3); //LR情况的镜像处理

}

**2.4.9 AvlTree\_Find函数**

**功能:** 寻找结点

**代码实现:**

AvlTree AvlTree\_Find(ElementType x,AvlTree &T){

while ((T!=NULL)&&(T->element!=x))

{

if (x<T->element)

T=T->left;

else

T=T->right;

}

return T; //非递归写法

/\*递归写法

if(x==T->element||T==NULL)

return T;

else if(x<T->element)

return AvlTree\_Find(x,T->left);

else

return AvlTree\_Find(x,T->right);\*/

}

**2.4.10 AvlTree\_Create函数**

**功能:** 创建结点

**代码实现:**

AvlTree AvlTree\_Create(ElementType x,AvlNode \*left,AvlNode \*right){

AvlTree p;

p=(AvlNode\*)malloc(sizeof(AvlNode));

if(NULL==p)

return NULL;

p->element=x;

p->left=left;

p->right=right;

p->height=0;

return p;

}

**2.4.11 AvlTree\_Insert函数**

**功能:** 插入结点

**代码实现:**

AvlTree AvlTree\_Insert(ElementType x,AvlTree &T){

if(NULL==T){

T=AvlTree\_Create(x,NULL,NULL);

if(NULL==T){

printf("ERROR\n");

return NULL;

}

}

else if(x<T->element){

T->left=AvlTree\_Insert(x,T->left); //插入至左子树

//失衡调整

if(AvlTree\_Height(T->left)-AvlTree\_Height(T->right)==2){

if(x<T->left->element)

T=SingleRotate\_Left(T); //LL情况,右旋

else

T=DoubleRotate\_Left(T); //LR情况,先左旋再右旋

}

}

else if(x>T->element){

T->right=AvlTree\_Insert(x,T->right); //插入至右子树

//失衡调整

if(AvlTree\_Height(T->right)-AvlTree\_Height(T->left)==2){

if(x>T->right->element)

T=SingleRotate\_Right(T); //RR情况，左旋

else

T=DoubleRotate\_Right(T); //RL情况，先右旋再左旋

}

}

T->height=MAX(AvlTree\_Height(T->right),AvlTree\_Height(T->left))+1; //修正高度

return T;

}

**2.4.12 AvlTree\_Delete函数**

**功能:** 删除结点

**代码实现:**

AvlTree AvlTree\_Delete(ElementType X,AvlTree &T){

if(NULL==T)//空树或待删除结点为空

return NULL;

if(X<T->element){//待删除结点位于左子树

T->left=AvlTree\_Delete(X,T->left);

if(AvlTree\_Height(T->right)-AvlTree\_Height(T->left)==2){

AvlNode \*R=(AvlNode\*)malloc(sizeof(AvlNode));

R=T->right;

if(AvlTree\_Height(R->left) > AvlTree\_Height(R->right))

T=DoubleRotate\_Right(T);

else

T=SingleRotate\_Right(T);

}

}

else if(X>T->element){//待删除结点位于右子树

T->right=AvlTree\_Delete(X,T->right);

if(AvlTree\_Height(T->left)-AvlTree\_Height(T->right)==2){

AvlNode \*L=(AvlNode\*)malloc(sizeof(AvlNode));

L=T->left;

if(AvlTree\_Height(L->right) > AvlTree\_Height(L->left))

T=DoubleRotate\_Left(T);

else

T=DoubleRotate\_Left(T);

}

}

else{ //待删除结点为根节点

if((T->left!=NULL)&&(T->right!=NULL)){

if(AvlTree\_Height(T->left)>AvlTree\_Height(T->right)){

AvlNode \*Max=(AvlNode\*)malloc(sizeof(AvlNode)); //在这种情况下，左子树高于右子树

Max=AvlTree\_Max(T->left); //取左子树中最大的结点作为新的根的替代品

T->element=Max->element; //找到的这个新根依然满足左子树均小于它且右子树均大于他

T->left=AvlTree\_Delete(Max->element,T->left); //新根的左子树为删除替代品后的原左子树

}

else{

AvlNode \*Min=(AvlNode\*)malloc(sizeof(AvlNode)); //这种情况下，右子树高于左子树，镜像操作

Min=AvlTree\_Min(T->right);

T->element=Min->element;

T->right=AvlTree\_Delete(Min->element,T->right);

}

}

else{

AvlNode \*temp=(AvlNode\*)malloc(sizeof(AvlNode)); //在这种情况下，左右子树有一为空

temp=T; //取不为空的那个作为新根

T=T->left ? T->left : T->right;

free(temp);

}

}

if(T)

T->height=MAX(AvlTree\_Height(T->left),AvlTree\_Height(T->right))+1;//修正高度

return T;

}

**2.4.13 AvlTree\_PreOrderTraverse函数**

**功能:** 先序遍历

**代码实现:**

Status AvlTree\_PreOrderTraverse(AvlTree &T){

if(T!=NULL){

printf("%d ",T->element);

AvlTree\_PreOrderTraverse(T->left);

AvlTree\_PreOrderTraverse(T->right);

}

}

**2.4.14 AvlTree\_InOrderTraverse函数**

**功能:** 中序遍历

**代码实现:**

Status AvlTree\_InOrderTraverse(AvlTree &T){

if(T!=NULL){

AvlTree\_PreOrderTraverse(T->left);

printf("%d ",T->element);

AvlTree\_PreOrderTraverse(T->right);

}

}

**2.4.15 AvlTree\_PostOrderTraverse函数**

**功能:** 后序遍历

**代码实现:**

Status AvlTree\_PostOrderTraverse(AvlTree &T){

if(T!=NULL){

AvlTree\_PreOrderTraverse(T->left);

AvlTree\_PreOrderTraverse(T->right);

printf("%d ",T->element);

}

}

**2.4.16 AvlTree\_Destroy函数**

**功能:** 销毁

**代码实现:**

Status AvlTree\_Destroy(AvlTree &T){

if (NULL==T)

return ERROR;

if (T->left != NULL)

AvlTree\_Destroy(T->left);

if (T->right != NULL)

AvlTree\_Destroy(T->right);

free(T);

T=NULL;

}

**2.4.17 Create函数**

**功能:** 创建

**代码实现:**

AvlTree Create(AvlTree &T){

int number,i,j;

printf("Enter how many nodes contain:");

scanf("%d",&number);

printf("Enter every nodes' element\_key.\n");

for(i=1;i<=number;i++){

printf("NO.%d node's element\_key : ",i);

scanf("%d",&j);

T=AvlTree\_Insert(j,T);

}

return T;

printf("Create Successfully.Press any key to return.\n");

getchar();

getchar();

//return OK;

}

**2.4.18 ChangeTreeToArray函数**

**功能:** 将树转化为数组

**代码实现:**

Array ChangeTreeToArray(AvlTree &T){

Status FLAG=TRUE;

ArrayNode \*head=NULL;

ArrayNode \*p=NULL;

ArrayNode \*q=NULL;

if(T==NULL)

printf("The Tree is empty.\n");

else{

LQueue \*Q=(LQueue\*)malloc(sizeof(LQueue));

Q->front=NULL;

Q->rear=NULL;

AvlTree X=T;

p=(ArrayNode\*)malloc(sizeof(ArrayNode));

p->element=X->element;

if(FLAG==TRUE){//数组中为空

head=p;

q=p;

FLAG=FALSE;

}

else{

q->next=p;

q=q->next;

}

LQueue\_EnQueue(\*Q,X); //queue负责遍历整个树，p指针不断向后移，q指针用来指示p的后继

while(LQueue\_DeQueue(\*Q,X)){

if(X->left!=NULL){

p=(ArrayNode\*)malloc(sizeof(ArrayNode));

p->element=X->left->element;

q->next=p;

q=q->next;

LQueue\_EnQueue(\*Q,X->left);

}

if(X->right!=NULL){

p=(ArrayNode\*)malloc(sizeof(ArrayNode));

p->element=X->right->element;

q->next=p;

q=q->next;

LQueue\_EnQueue(\*Q,X->right);

}

}

if(q!=NULL)

q->next=NULL;

}

return head;

}

**2.4.19 AvlTree\_Merge函数**

**功能:** 合并

**代码实现:**

AvlTree AvlTree\_Merge(AvlTree &T1,AvlTree &T2){

ArrayNode \*a=(ArrayNode\*)malloc(sizeof(ArrayNode));

a=ChangeTreeToArray(T2);

while(a!=NULL){

AvlTree\_Insert(a->element,T1);

a=a->next;

}

return T1;

}

**2.4.20 AvlTree\_Spilt函数**

**功能:** 分裂

**代码实现:**

Status AvlTree\_Spilt(AvlTree &T,AvlTree &T1,AvlTree &T2,ElementType x){

ArrayNode \*a=(ArrayNode\*)malloc(sizeof(ArrayNode));

a=ChangeTreeToArray(T);

if(T==NULL)

return FALSE;

else{

while(a!=NULL){

if(a->element<=x){

AvlTree\_Insert(a->element,T1);

a=a->next;

}

else{

AvlTree\_Insert(a->element,T2);

a=a->next;

}

}

}

return TRUE;

}**2.4.21 AvlTree\_Print函数**

**功能:** 打印

**代码实现:**

Status AvlTree\_Print(AvlTree &T, int dep){

int i;

if(T->right)

AvlTree\_Print(T->right,dep+1);

for(i=0;i<dep;i++)

printf(" ");

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

if(T->left)

AvlTree\_Print(T->left,dep+1);

return OK;

}

**2.4.22 AvlTree\_Test函数**

**功能:** 测试

**代码实现:**

Status AvlTree\_Test(AvlTree &T1,AvlTree &T2){

int n,m,k,i,j,o,choose,cho\_m,cho\_s,x;

AvlNode \*TEMP=(AvlNode\*)malloc(sizeof(AvlNode));

AvlTree T3=(AvlNode\*)malloc(sizeof(AvlNode));

T3=NULL;

while(1){

system("cls");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*T1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

if(!T1)

printf("Now T1 is NULL.\n");

else{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Shape\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

AvlTree\_Print(T1,0);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Traverse\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf(" PreOrderTraverse :");

AvlTree\_PreOrderTraverse(T1);

printf("\n InOrderTraverse :");

AvlTree\_InOrderTraverse(T1);

printf("\nPostOrderTraverse :");

AvlTree\_PostOrderTraverse(T1);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*T2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

if(!T2)

printf("Now T2 is NULL.\n");

else{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Shape\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

AvlTree\_Print(T2,0);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Traverse\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf(" PreOrderTraverse :");

AvlTree\_PreOrderTraverse(T2);

printf("\n InOrderTraverse :");

AvlTree\_InOrderTraverse(T2);

printf("\nPostOrderTraverse :");

AvlTree\_PostOrderTraverse(T2);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*T3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

if(!T2)

printf("Now T3 is NULL.\n");

else{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Tree Shape\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

AvlTree\_Print(T3,0);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Operation Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\n T1 1.Create 2.Insert 3.Delete 4.Find 5.Destroy\n");

printf("\n T2 6.Create 7.Insert 8.Delete 9.Find 10.Destroy\n");

printf("\nT1&T2 11:Merge 12:Split\n");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Console Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\nEnter number to choose operation :");

scanf("%d",&choose);

switch(choose){

case 1:

T1=Create(T1);

break;

case 2:{

printf("Enter the element\_key you want to insert :");

scanf("%d",&n);

T1=AvlTree\_Insert(n,T1);

break;

}

case 3:{

printf("Enter the element\_key you want to delete :");

scanf("%d",&m);

T1=AvlTree\_Delete(m,T1);

getchar();

getchar();

break;

}

case 4:{

printf("Enter the element\_key you want to find :");

scanf("%d",&k);

TEMP=T1;

if((AvlTree\_Find(k,T1))!=NULL)

printf("Find %d successfully.\n",k);

else

printf("Find failed.\n");

getchar();

getchar();

T1=TEMP;

break;

}

case 5:{

AvlTree\_Destroy(T1);

printf("Destroy T1 Successfully.\n");

getchar();

getchar();

break;

}

case 6:{

T2=Create(T2);

break;

}

case 7:{

printf("Enter the element\_key you want to insert :");

scanf("%d",&i);

T1=AvlTree\_Insert(i,T2);

break;

}

case 8:{

printf("Enter the element\_key you want to delete :");

scanf("%d",&j);

T2=AvlTree\_Delete(j,T2);

getchar();

getchar();

break;

}

case 9:{

printf("Enter the element\_key you want to find :");

scanf("%d",&o);

TEMP=T2;

if((AvlTree\_Find(o,T2))!=NULL)

printf("Find %d successfully.\n",o);

else

printf("Find failed.\n");

T2=TEMP;

getchar();

getchar();

break;

}

case 10:{

AvlTree\_Destroy(T2);

printf("Destroy T2 Successfully.\n");

getchar();

getchar();

break;

}

case 11:{

printf("Merge T1 to T2(case 1)or T2 to T1(case 2)?\n");

printf("Enter which case you choose :");

scanf("%d",&cho\_m);

if(cho\_m==1){

AvlTree\_Merge(T2,T1);

T1=NULL;

}

if(cho\_m==2){

AvlTree\_Merge(T1,T2);

T2=NULL;

}

printf("Merge Successfully.\n");

getchar();

getchar();

break;

}

case 12:{

printf("Which Tree you want to spilt,T1(case1) or T2(case2)?\n");

printf("Enter which case you choose :");

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

printf("Enter the spilt key :");

scanf("%d",&x);

if(cho\_s==1){

AvlTree\_Spilt(T1,T3,T2,x);

}

if(cho\_s==2){

AvlTree\_Spilt(T2,T1,T3,x);

}

printf("Split Successfully.");

getchar();

getchar();

break;

}

}

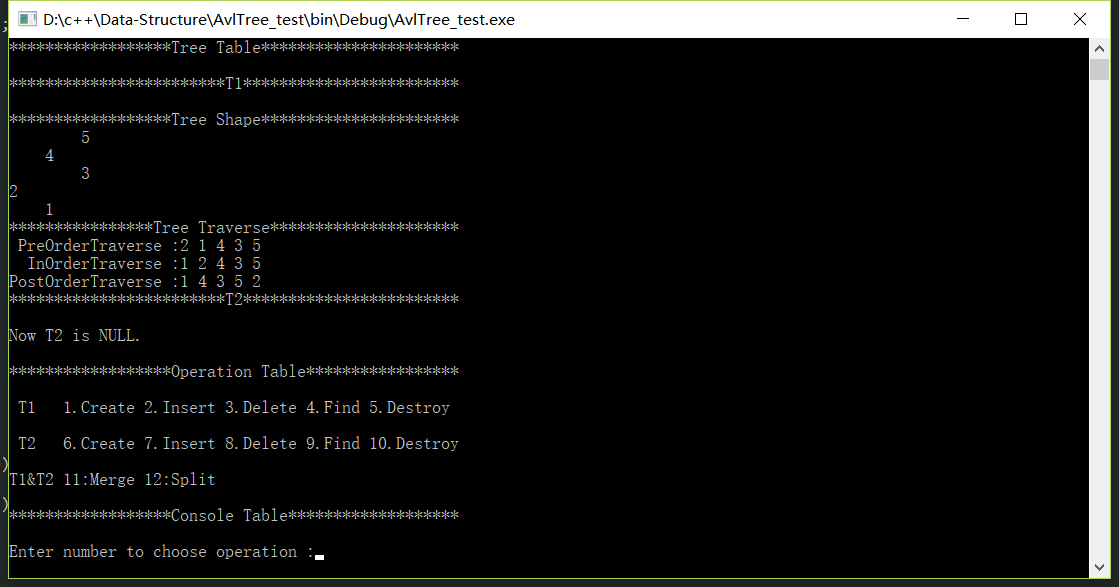
}

}

1. **功能测试**

**3.1插入功能测试/遍历功能测试**

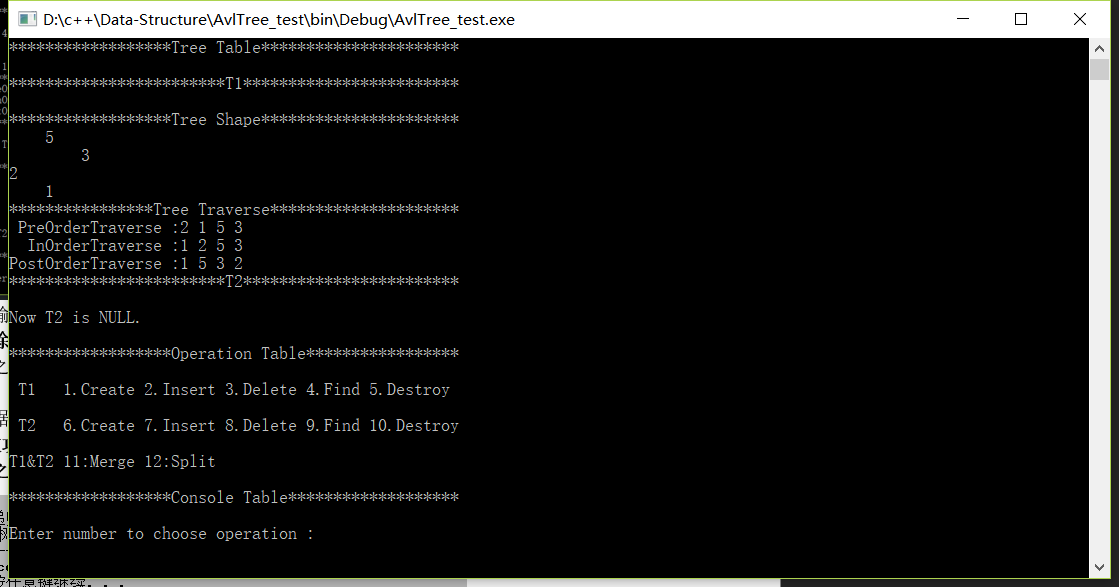
依次插入1-5五个数



由输出可知，插入功能正常并且遍历功能正常

**3.2删除功能测试**

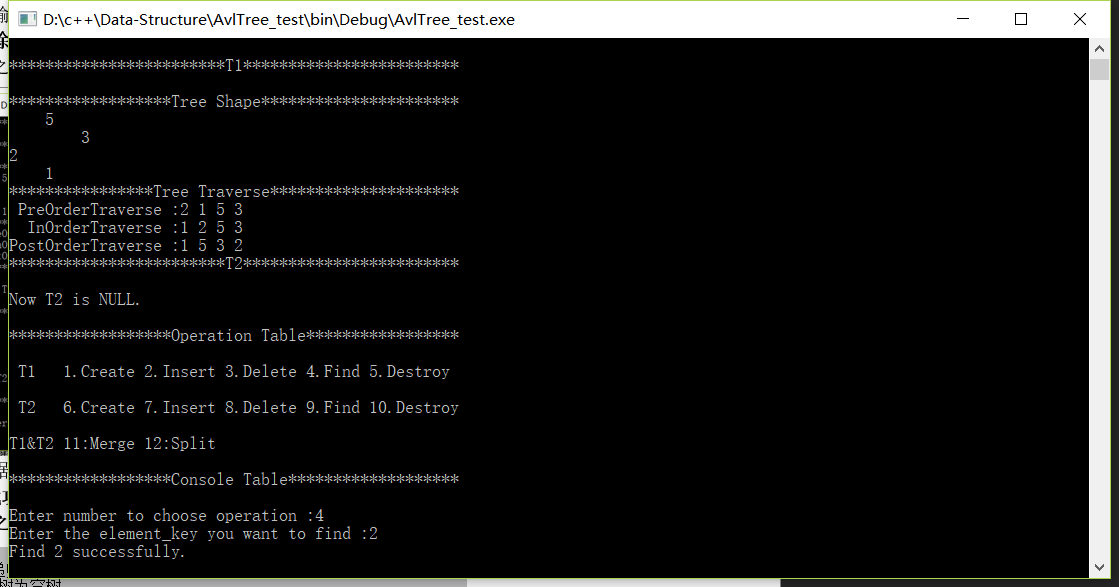
在之前插入后进行删除关键字的功能测试，选取4依次进行删除测试，结果如下：



根据定义和输出，删除功能正常

**3.3查找功能测试**

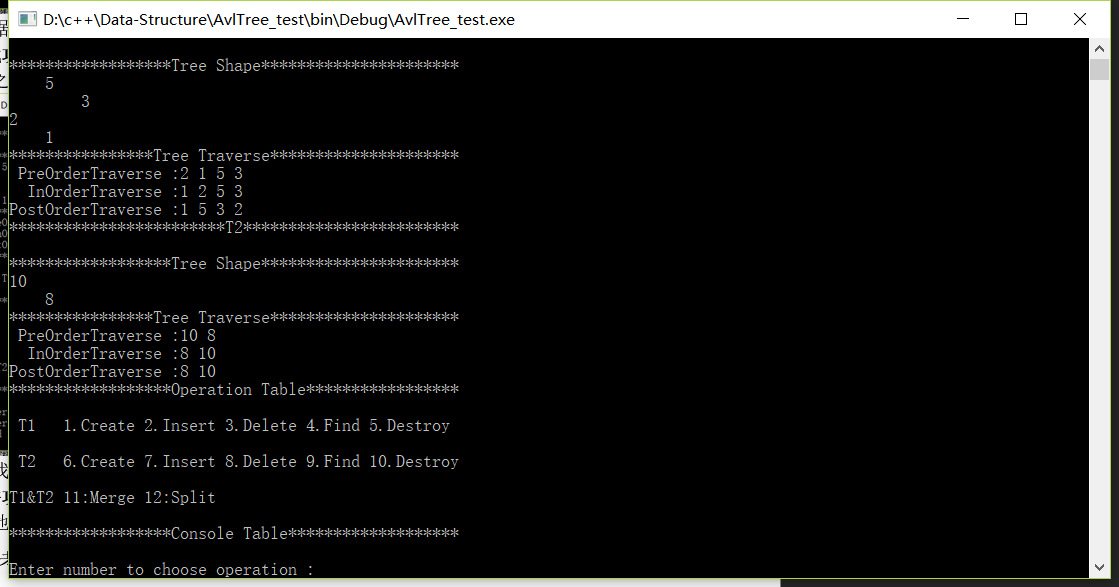
在之前的基础上进行查找测试，查找2，结果如下：



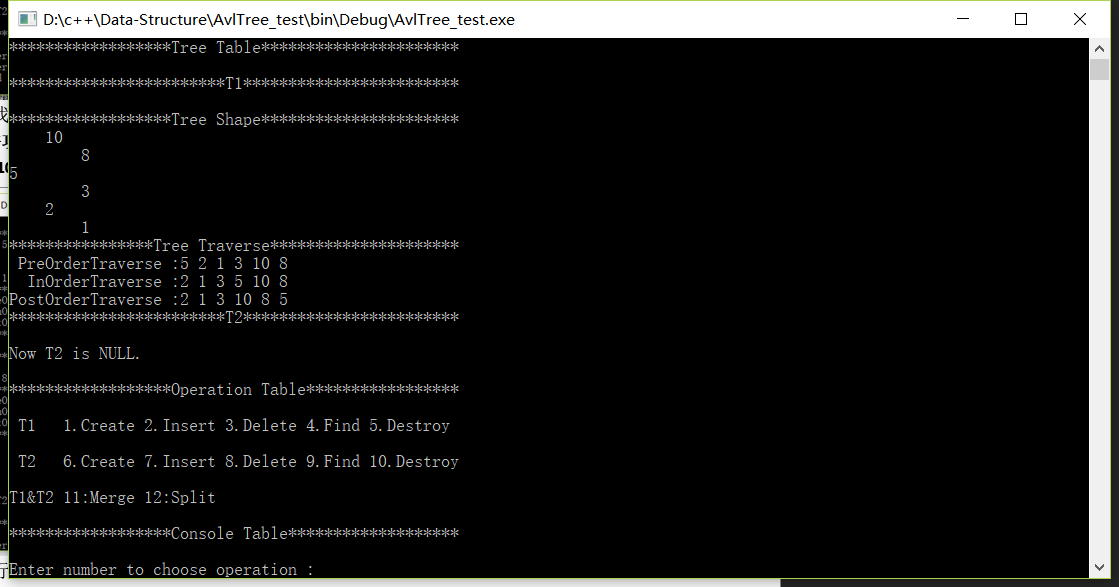
查找功能正常

**3.4合并功能测试**

将10，8插入T2



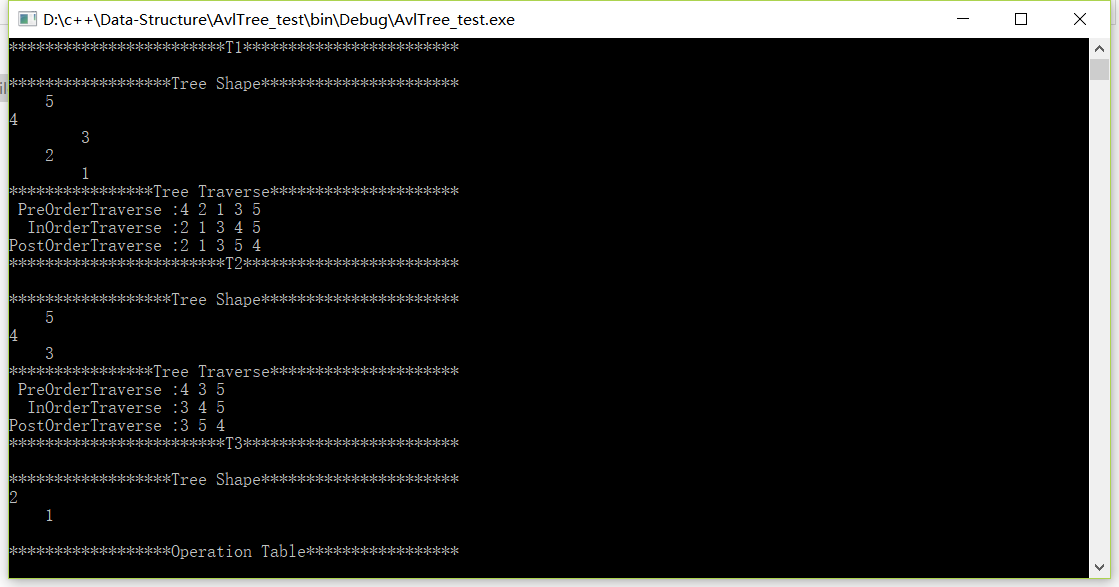
执行合并操作



结果正常，合并功能正确

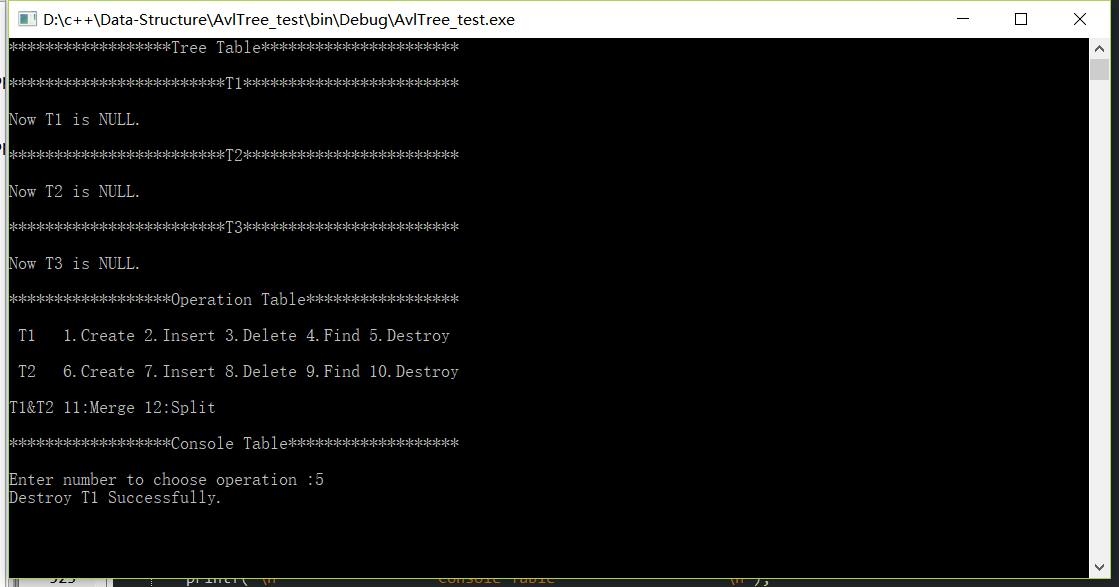
**3.5分裂功能测试**

输入1-5,将分裂界设为3进行分裂



分裂正常

**3.6销毁功能测试**



销毁功能正常

**四.思考与小结**

**4.1错误总结**

1. 递归实现的时候内存分配出错
2. 语法错误
3. 代码健壮性不足

**4.2部分优化**

1. 优化了显示界面
2. 优化了树的表示，从类似于广义表的括号表示法优化为凹入表表示
3. 接口封装完善