



**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（5）班**

**学 号 3219004950**

**学生姓名 吴泳诗**

**指导教师 曾孜**

**2020年12月 20 日**

**报告：**

报告内容：□详细 □完整 □基本完整 □不完整

设计方案：□非常合理 □合理 □基本合理 □较差

算法实现：□全部实现 □基本实现 □部分实现 □实现较差

测试样例：□完备 □基本完备 □基本完备 □不完备

文档格式：□规范 □比较规范 □基本规范 □不规范

**Anyview成绩及上机表现：**

□优秀 □良好 □中等 □及格 □不及格

**总评成绩：**

□优秀 □良好 □中等 □及格 □不及格

1. **题目：平衡二叉树**

[问题描述]

利用平衡二叉树实现一个动态查找表。

[基本要求]

实现动态查找表的三种基本功能：查找、插入和删除。

[实现提示]

1. 初始，平衡二叉树为空树，操作界面给出查找、插入和删除三种操作供选择。每种操作均要提示输入关键字。每次插入或删除一个结点后，应更新平衡二叉树的显示.
2. 平衡二叉树的显示可采用如6.3题要求的凹入表形式，也可以采用图形界面画出树形。
3. 教科书已给出查找和插入算法，本题重点在于对删除算法的设计和实现。假设要删除关键字为x的结点。如果x不在叶子结点上，则用它左子树中的最大值或右子树中的最小值取代x。如此反复取代，直到删除动作传递到某个叶子结点。删除叶子结点时，若需要进行平衡变换，可采用插入的平衡变换的反变换（如，左子树变矮对应于右子树长高）。

[基本操作]

1. **左旋处理**

**void** L\_Rotate(**BBSTree** &p)

初始条件：p不为空。

操作结果：对右高（最小失衡子树的根节点平衡因子为-2）的子树进行左旋调整。

1. **右旋处理**

**void** R\_Rotate(**BBSTree** &p)

初始条件：p不为空。

操作结果：对左高（最小失衡子树的根节点平衡因子为2）的子树进行右旋调整。

1. **左平衡处理**

**void** Left\_Balance(**BBSTree** &T)

初始条件：T不为空。

操作结果：对T进行左平衡处理。（处理LL型和LR型）

1. **右平衡处理**

**void** Right\_Balance(**BBSTree** &T)

初始条件：T不为空。

操作结果：对T进行右平衡处理。（处理RR型和RL型）

1. **插入结点**

**Status** Insert\_AVL(**BBSTree** &T, **RcdType** e, **Status** &taller)

操作结果：查询插入结点是否已经存在于目标树，没有则插入。

返回值：插入成功返回1，不成功返回-1。

1. **创建平衡二叉树**

**BBSTree** Create\_BBSTree()

操作结果：根据用户输入的结点顺序逐个插入，创建一棵平衡二叉树。

返回值：创建完成的平衡二叉树。

1. **求平衡二叉树深度**

**int** Get\_Depth(**BBSTree** T)

操作结果：求平衡二叉树的深度。

返回值：平衡二叉树的深度，如果该树为空则返回0。

1. **判断平衡二叉树是否不为空**

**Status** Is\_Not\_Null(**BBSTree** T)

操作结果：判断平衡二叉树是否不为空。

返回值：平衡二叉树为空则返回-1，不为空则返回1。

1. **搜索结点**

**BBSTree** Serach\_Tree(**BBSTree** T, **RcdType** e)

操作结果：搜索结点。

返回值：返回空表示搜索不到该结点；

如果搜索到该结点则返回以该结点为根的子树。

1. **删除结点**

**Status** Delete\_Tree(**BBSTree** &T, **RcdType** e, **Status** &shorter)

初始条件：T不为空。

操作结果：删除结点。

返回值：删除成功返回1，不成功返回-1。

1. **递归先序遍历**

**void** PreOrder\_RecTraverse(**BBSTree** T)

操作结果：先序遍历平衡二叉树并依次打印结点值。

1. **递归中序遍历**

**void** InOrder\_RecTraverse(**BBSTree** T)

操作结果：中序遍历平衡二叉树并依次打印结点值。

1. **递归后序遍历**

**void** LastOrder\_RecTraverse(**BBSTree** T)

操作结果：后序遍历平衡二叉树并依次打印结点值。

1. **凹入表形式打印平衡二叉树**

**void** Print(**BBSTree** T)

操作结果：打印分割线，并调用*Print\_AVL*方法打印平衡二叉树（传入层数参数为1）

**void** Print\_AVL(**BBSTree** &T,**int** i)

操作结果：按照先序遍历访问平衡二叉树并打印，按照传入的层数参数（i）判断打印该结点前缩进的程度。其中，如果某一结点的左右孩子之一为空，则以x显式地打印出来。

1. **合并平衡二叉树**

**void** Merge\_AVL(**BBSTree** &T1, **BBSTree** &T2)

操作结果：判断T1和T2谁的深度更大，并把深度较大的那一棵树作为第一个参数（被插入的树）传入真正的合并平衡二叉树方法*Real\_Merge*。

**void** Real\_Merge(**BBSTree** &T, **BBSTree** T2)

初始条件：T2不为空。

操作结果：把T2的所有结点插入到T中，如果T中已存在该结点则不插入。

1. **分裂平衡二叉树**

**void** Spilt\_AVL(**BBSTree** T, **RcdType** e, **BBSTree** &T1, **BBSTree** &T2)

初始条件：T不为空。

操作结果：T被分裂为两棵树T1和T2，

其中T1中所有结点均小于等于e，

T2中所有结点均大于e。

**2．基本定义**

1. 公用头文件:

#**include** <stdio.h>

#**include** <stdlib.h>

1. 返回类型常量

**#define** TRUE 1

**#define** FALSE 0

**#define** OK 1

**#define** ERROR 0

**#define** SUCCESS 1

**typedef** int Status;

1. 平衡因子常量

**#define** LH +1 //左子树比右子树高，简称左高

**#define** EH 0 //左右子树等高，简称等高

**#define** RH -1 //右子树比左子树高，简称右高

1. B树存储结构

**typedef** int RcdType; //本次实验中定义数据类型为int型

//平衡二叉树结构体定义

**typedef** **struct** BBSTNode

{

**RcdType** data; //存储的数据

**int** bf; //平衡因子

**BBSTNode** \*lchild, \*rchild; //左右孩子结点

} **BBSTNode**, \***BBSTree**; //平衡二叉树的结点及指针类型

1. **算法设计**
2. **左旋处理**

**void** L\_Rotate(**BBSTree** &p)

{

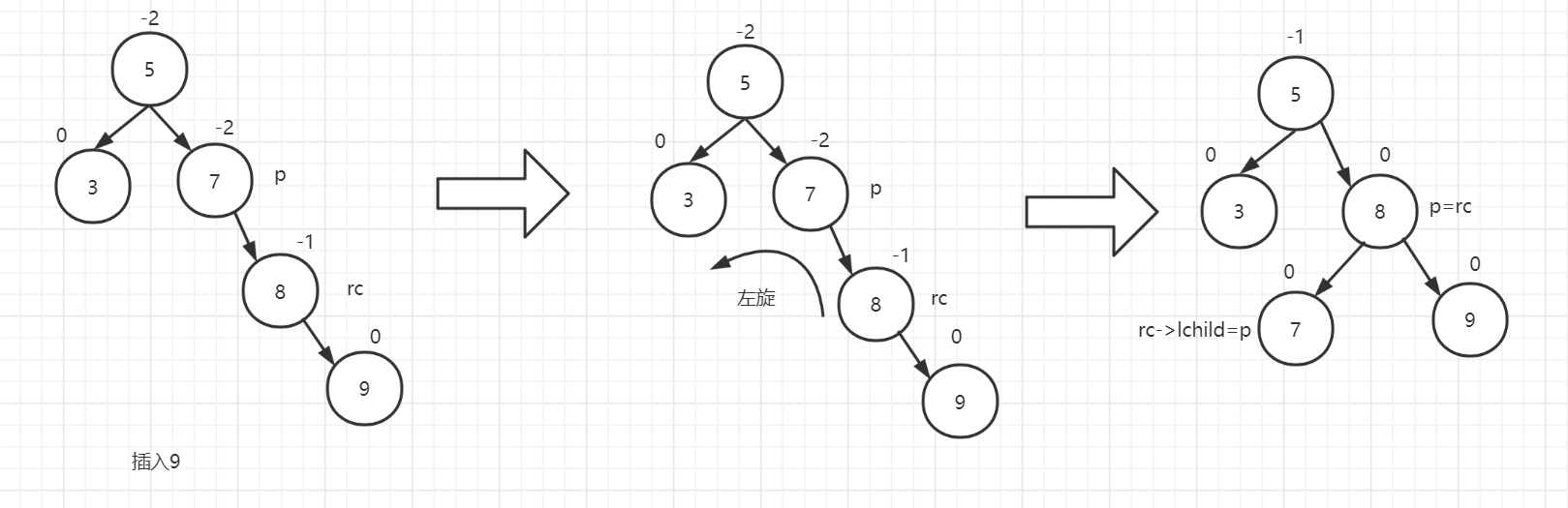
**BBSTree** rc = p->rchild;

p->rchild = rc->lchild;

rc->lchild = p;

p = rc;

}



1. **右旋处理**

**void** R\_Rotate(**BBSTree** &p)

{

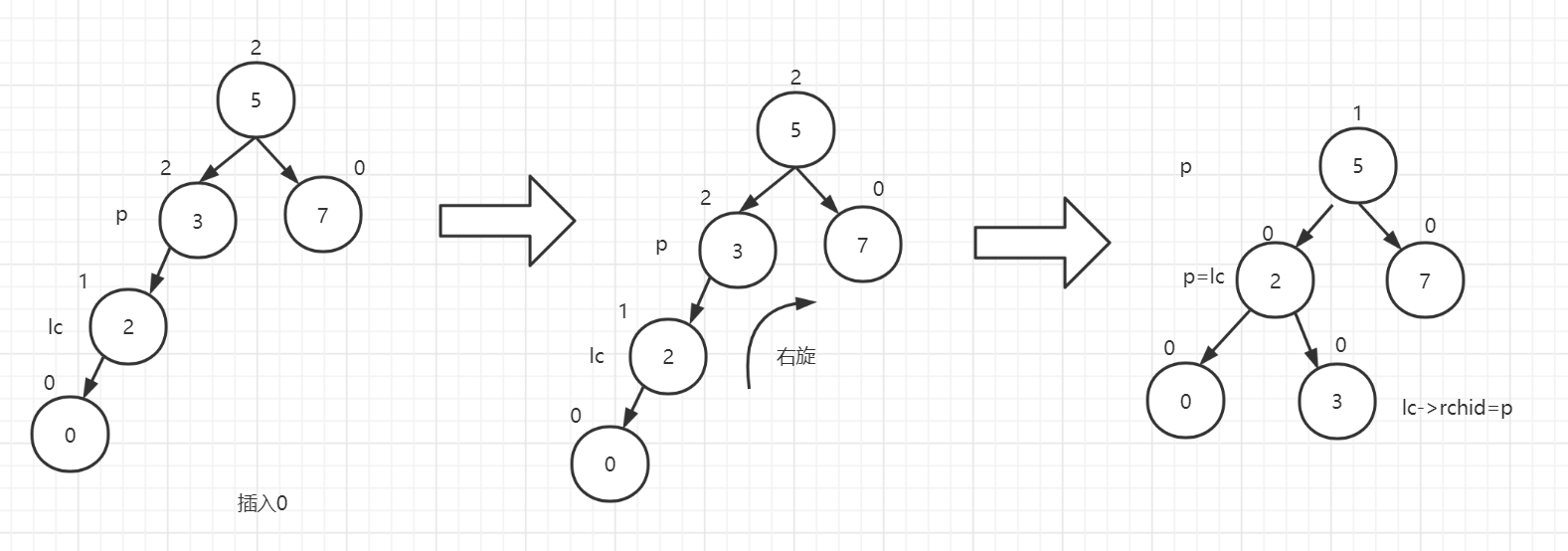
**BBSTree** lc = p->lchild;

p->lchild = lc->rchild;

lc->rchild = p;

p = lc;

}



1. **左平衡处理**

**void** Left\_Balance(**BBSTree** &T)

{

**BBSTree** lc, rd;

lc = T->lchild;

switch (lc->bf)

{

//LL型，右旋调整

case **LH**:

T->bf = lc->bf = **EH**;

R\_Rotate(T);

break;

//LR型 ，双旋处理

case **RH**:

rd = lc->rchild;

//修改T及其左孩子的平衡因子

switch (rd->bf)

{

case **LH**:

T->bf = **RH**;

lc->bf = **EH**;

break;

case **EH**:

T->bf = lc->bf = **EH**;

break;

case **RH**:

T->bf = **EH**;

lc->bf = **LH**;

break;

}

rd->bf = **EH**;

//对T的左子树作左旋调整

L\_Rotate(T->lchild);

//对T作右旋调整

R\_Rotate(T);

break;

}

}

1. **右平衡处理**

**void** Right\_Balance(**BBSTree** &T)

{

**BBSTree** rc, ld;

rc = T->rchild;

switch (rc->bf)

{

//RR型，左旋处理

case **RH**:

T->bf = rc->bf = **EH**;

L\_Rotate(T);

break;

//RL型，双旋处理

case **LH**:

ld = rc->lchild;

switch (ld->bf)

{

case **RH**:

T->bf = **LH**;

rc->bf = **EH**;

break;

case **EH**:

T->bf = rc->bf = **EH**;

break;

case **LH**:

T->bf = **EH**;

rc->bf = **RH**;

break;

}

ld->bf = **EH**;

//对T的右孩子作右旋调整

R\_Rotate(T->rchild);

//对T作左旋调整

L\_Rotate(T);

}

}

1. **插入结点**

**Status** Insert\_AVL(**BBSTree** &T, **RcdType** e, **Status** &taller)

{

//实现对e插入到二叉树的操作

if(!T) {

T = (**BBSTree**)malloc(sizeof(**BBSTNode**));

T->data = e;

T->bf = **EH**;

T->lchild = **NULL**;

T->rchild = **NULL**;

taller = **TRUE**;

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

taller = **FALSE**;

return **FALSE**; //未插入

} else if(e < T->data) { //插入到左子树

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

if(**TRUE** == taller) {

switch(T->bf) { //检查T的平衡因子

//原左高，左平衡处理

case **LH**:

Left\_Balance(T);

taller = **FALSE**;

break;

//原等高，左变高

case **EH**:

T->bf = **LH**;

taller = **TRUE**;

break;

//原右高，变等高

case **RH**:

T->bf = **EH**;

taller = **FALSE**;

break;

}

}

} else {

if(**FALSE** == Insert\_AVL(T->rchild, e, taller)) return **FALSE**;

if(**TRUE** == taller) {

switch(T->bf) {

//原本左高，变等高

case **LH**:

T->bf = **EH**;

taller = **FALSE**;

break;

//原本等高，变右高

case **EH**:

T->bf = **RH**;

taller = **TRUE**;

break;

//原本右高，右平衡处理

case **RH**:

Right\_Balance(T);

taller = **FALSE**;

break;

}

}

}

return **TRUE**;

}

1. **创建平衡二叉树**

**BBSTree** Create\_BBSTree()

{

**BBSTree** T = **NULL**;

**Status** taller = **TRUE**;

**RcdType** \*a;

**int** n;

printf("请输入插入元素的个数：");

scanf("%d", &n);

a = (**RcdType**\*)malloc(sizeof(**int**)\*n);

printf("请输入要插入的元素：");

for(**int** i = 0; i<n; i++){

scanf("%d", &a[i]);

}

for(**int** j = 0; j<n; j++){

Insert\_AVL(T, a[j], taller);

}

return T;

}

1. **求平衡二叉树深度**

**int** Get\_Depth(**BBSTree** T)

{

**int** depthLeft, depthRight;

if (!T)

return 0;

else

{

depthLeft = Get\_Depth(T->lchild);

depthRight = Get\_Depth(T->rchild);

return 1 + (depthLeft > depthRight ? depthLeft : depthRight);

}

}

1. **判断平衡二叉树是否不为空**

**Status** Is\_Not\_Null(**BBSTree** T)

{

if (!T)

return **FALSE**;

return **TRUE**;

}

1. **搜索结点并返回以其为根的子树**

**BBSTree** Serach\_Tree(**BBSTree** T, **RcdType** e)

{

if (!T)

return **NULL**;

else if (e == T->data)

return T;

else if (e < T->data)

return Serach\_Tree(T->lchild, e);

else

return Serach\_Tree(T->rchild, e);

}

1. **删除结点**

**Status** Delete\_Tree(**BBSTree** &T, **RcdType** e, **Status** &shorter)

{

static **int** tag = 0;

if(!T) {

return **FALSE**;

} else if(e == T->data) {

**BBSTNode** \*q = **NULL**;

//如果该结点只有一个孩子，则将自子树取代该结点

if(!T->lchild) {

q = T;

T = T->rchild;

free(q);

shorter = **TRUE**;

} else if(!T->rchild) {

q = T;

T = T->lchild;

free(q);

shorter = **TRUE**;

}

//如果被删结点有两个孩子，则找到结点的前驱结点的前驱结点，并将前驱结点的值赋给该结点，然后删除前驱结点

else{

q = T->lchild;

while(q->rchild) {

q = q->rchild;

}

T->data = q->data;

if(T->lchild->data == q->data) {

tag = 1;

}

Delete\_Tree(T->lchild, q->data, shorter);

if(tag == 1) {

**BBSTree** r = T->rchild;

if(**NULL** == r) T->bf = 0;

else {

switch(r->bf) {

case **EH**:

T->bf = **RH**; break;

default: Right\_Balance(T); break;

}

}

}

}

} else if(e < T->data) {

if(!Delete\_Tree(T->lchild, e, shorter)) {

return **FALSE**;

}

//删除完后，调整结点的平衡因子

if(shorter && (tag == 0)) {

switch(T->bf) {

case **LH**:

T->bf = **EH**;

shorter = **TRUE**;

break;

case **EH**:

T->bf = **RH**;

shorter = **FALSE**;

break;

//如果本来右子树较高，删除后不平衡，进行右平衡操作

case **RH**:

Right\_Balance(T); //右平衡处理

if(T->rchild->bf == **EH**)

shorter = **FALSE**;

else

shorter = **TRUE**;

break;

}

}

} else if(e > T->data) {

if(!Delete\_Tree(T->rchild, e, shorter)) {

return **FALSE**;

}

//删除完后，调整平衡因子

if(shorter && (tag == 0)) {

switch(T->bf) {

case **LH**:

Left\_Balance(T); //左平衡处理

if(T->lchild->bf == **EH**)

shorter = **FALSE**;

else

shorter = **TRUE**;

break;

case **EH**:

T->bf = **LH**;

shorter = **TRUE**;

break;

}

}

if(tag == 1) {

**int** depthLeft = Get\_Depth(T->lchild);

**int** depthRight = Get\_Depth(T->rchild);

T->bf = depthLeft - depthRight;

}

}

return **OK**;

}

1. **递归先序遍历**

**void** PreOrder\_RecTraverse(**BBSTree** T)

{

if (!T) return;

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

PreOrder\_RecTraverse(T->lchild);

PreOrder\_RecTraverse(T->rchild);

}

1. **递归中序遍历**

**void** InOrder\_RecTraverse(**BBSTree** T)

{

if (!T) return;

if (T->lchild)

InOrder\_RecTraverse(T->lchild);

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

if (T->rchild)

InOrder\_RecTraverse(T->rchild);

}

1. **递归后序遍历**

**void** LastOrder\_RecTraverse(**BBSTree** T)

{

if (!T) return;

if (T->lchild)

LastOrder\_RecTraverse(T->lchild);

if (T->rchild)

LastOrder\_RecTraverse(T->rchild);

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

}

1. **凹入表形式打印平衡二叉树**

//显示平衡二叉树，i表示当前层次

**void** Print\_AVL(**BBSTree** &T,**int** i)

{

//以凹入表形式打印（先序遍历）

for(**int** x = 0;x <= i;x++) {

printf(" ");

}

if (!T) {

printf("x \n");

} else {

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

if(T->lchild || T->rchild) {

Print\_AVL(T->lchild,i+1);

Print\_AVL(T->rchild,i+1);

}

}

}

**void** Print(**BBSTree** T)

{

printf("==============================\n");

Print\_AVL(T,1);

printf("==============================\n");

}

1. **合并平衡二叉树**

//真正的合并方法，T为合并后的树

**void** Real\_Merge(**BBSTree** &T, **BBSTree** T2) {

**Status** taller = **FALSE**;

if (T2) {

Real\_Merge(T, T2->lchild);

Insert\_AVL(T, T2->data, taller);

Real\_Merge(T, T2->rchild);

}

}

//合并平衡二叉树（此方法只是判断了应该合并到哪一棵树，真正的合并方法是Real\_Merge）

**void** Merge\_AVL(**BBSTree** &T1, **BBSTree** &T2) {

//把深度较小的树合并到深度较大的树

if(Get\_Depth(T1) >= Get\_Depth(T2)) {

//合并到T1

Real\_Merge(T1,T2);

Print(T1);

} else {

//合并到T2

Real\_Merge(T2,T1);

Print(T2);

}

}

1. **分裂平衡二叉树**

//分裂平衡二叉树（T1结点均小于/等于e，T2均大于e）

**void** Spilt\_AVL(**BBSTree** T, **RcdType** e, **BBSTree** &T1, **BBSTree** &T2) {

**Status** taller = **FALSE**;

if (T) {

//递归访问左子树

Spilt\_AVL(T->lchild, e, T1, T2);

if(T->data <= e) {

Insert\_AVL(T1, T->data, taller);

} else if (T->data > e) {

Insert\_AVL(T2, T->data, taller);

}

//递归访问右子树

Spilt\_AVL(T->rchild, e, T1, T2);

}

}

1. **程序主方法**

int main()

{

**BBSTree** T = **NULL**;

**int** num = 0;

**RcdType** e;

**Status** taller = **TRUE**;

**Status** shorter = **TRUE**;

printf("----------------------------\n");

printf("-- 数据结构实验 --\n");

printf("-- 平衡二叉树 --\n");

printf("-- 3219004950 --\n");

printf("-- 吴泳诗 --\n");

printf("----------------------------\n\n");

printf("------------------------------\n");

printf("-- 二叉树展示 --\n");

printf("-- 注：如果左右孩子之一为空 --\n");

printf("-- 则以x显式地打印出来 --\n");

printf("-- 但叶子结点不会打印两个x --\n");

printf("------------------------------\n");

while(num != 11){

printf("\n");

printf("1：创建一棵平衡二叉树\n");

printf("2：先序遍历\n");

printf("3：中序遍历\n");

printf("4：后序遍历\n");

printf("5：插入一个结点\n");

printf("6：删除一个结点\n");

printf("7：求平衡二叉树的深度\n");

printf("8: 合并两棵平衡二叉树\n");

printf("9: 分裂二叉树\n");

printf("10: 查找结点\n");

printf("11：退出系统\n");

printf("\n");

printf("请输入对应功能的序号：");

scanf("%d", &num);

switch(num){

case 1:

T = Create\_BBSTree();

if(**NULL** != T) printf("创建成功！\n");

Print(T);

break;

case 2:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("先序遍历结果如下：\n");

PreOrder\_RecTraverse(T);

printf("\n");

break;

}

case 3:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("中序遍历结果如下：\n");

InOrder\_RecTraverse(T);

printf("\n");

break;

}

case 4:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("后序遍历结果如下：\n");

LastOrder\_RecTraverse(T);

printf("\n");

break;

}

case 5:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("请输入要插入的元素：");

scanf("%d", &e);

if(**TRUE** == Insert\_AVL(T, e, taller)){

printf("插入成功！\n");

Print(T);

} else {

printf("插入失败！\n");

}

break;

}

case 6:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("请输入要删除的结点：");

scanf("%d", &e);

if(**TRUE** == Delete\_Tree(T, e, shorter)){

printf("删除成功！\n");

Print(T);

} else {

printf("删除失败，结点不存在！\n");

}

break;

}

case 7:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("平衡二叉树的深度为：");

**int** dep = Get\_Depth(T);

printf("%d", dep);

printf("\n");

break;

}

case 8:

{

printf("请创建两棵平衡二叉树\n");

printf("第一棵树\n");

**BBSTree** T1 = Create\_BBSTree();

if(**NULL** != T1) printf("创建成功！\n");

Print(T1);

printf("第二棵树\n");

**BBSTree** T2 = Create\_BBSTree();

if(**NULL** != T2) printf("创建成功！\n");

Print(T2);

printf("合并后的树\n");

Merge\_AVL(T1,T2);

break;

}

case 9:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("请输入分裂的依据\n");

int x;

scanf("%d",&x);

**BBSTree** T1 = **NULL**;

**BBSTree** T2 = **NULL**;

Spilt\_AVL(T,x,T1,T2);

Print(T1);

Print(T2);

break;

}

case 10:

if(Is\_Not\_Null(T) == **FALSE**) {

printf("请先创建一颗平衡二叉树！\n");

break;

} else {

printf("请输入想要查找的结点\n");

int y;

scanf("%d",&y);

**BBSTree** T1 = Serach\_Tree(T,y);

if(T1) {

printf("查找成功！\n");

Print(T1);

} else {

printf("查找失败!\n");

}

break;

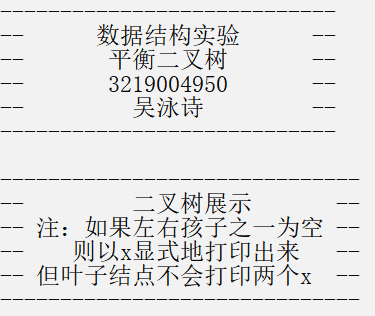
}

}

}

}

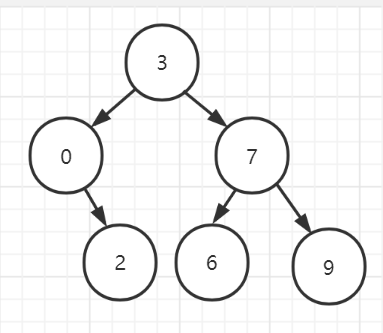
**4．测试**



1. **创建平衡二叉树**

测试数据：9,7,0,3,6,2

预期结果：

3

0

X

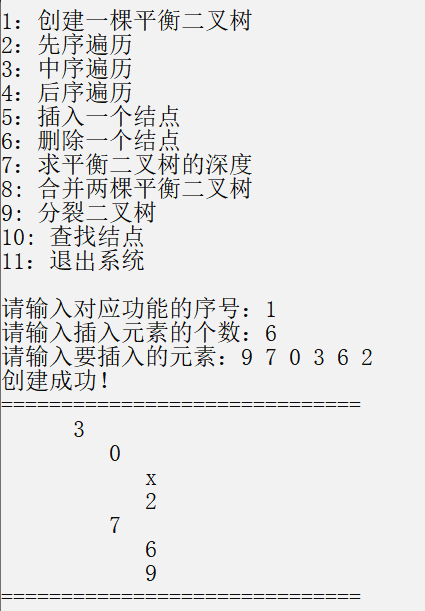
2

7

6

9

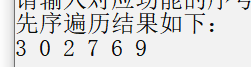
测试结果：



1. **先序遍历**

预期结果：3 0 2 7 6 9

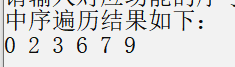
测试结果：



1. **中序遍历**

预期结果：0 2 3 6 7 9

测试结果：



1. **后序遍历**

预期结果：2 0 6 9 7 3

测试结果：

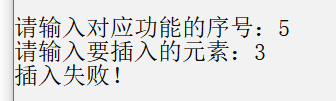


1. **插入结点**

测试数据：在（1）基础上插入3

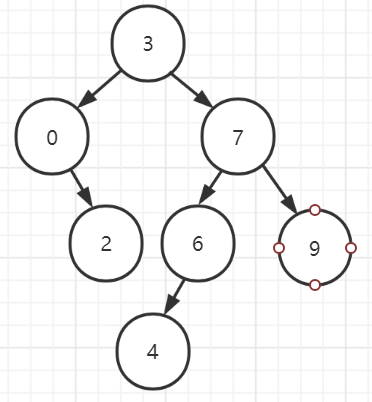
预期结果：插入失败

测试结果：



测试数据：在（1）基础上插入4

预期结果：

3

0

x

2

7

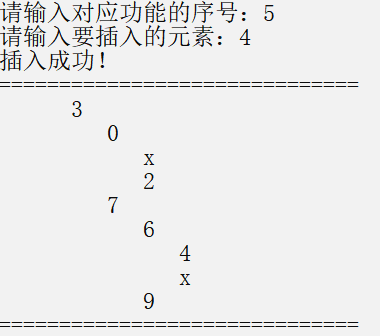
6

4

x

9

测试结果：

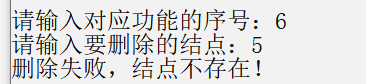


1. **删除结点**

测试数据：在（5）基础上删除5

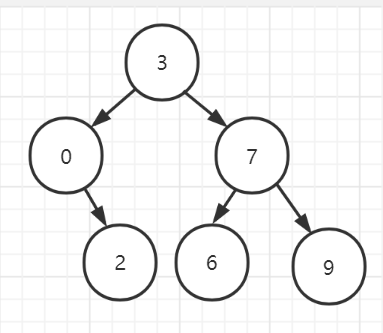
预期结果：删除失败

测试结果：



测试数据：在（5）基础上删除4

预期结果：

3

0

X

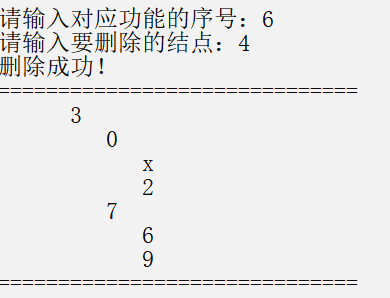
2

7

6

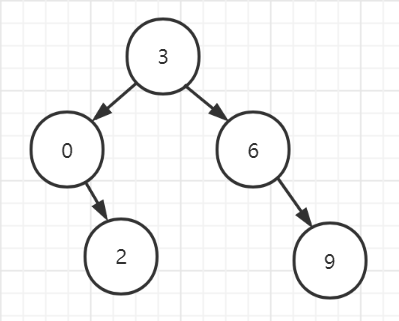
9

测试结果：



测试数据：在（1）基础上删除7

预期结果：

3

0

X

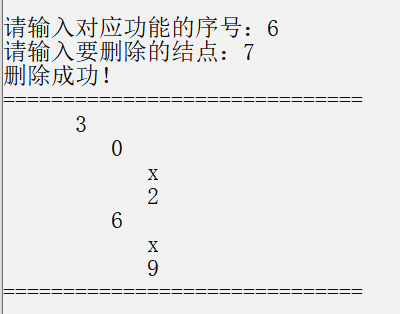
2

6

X

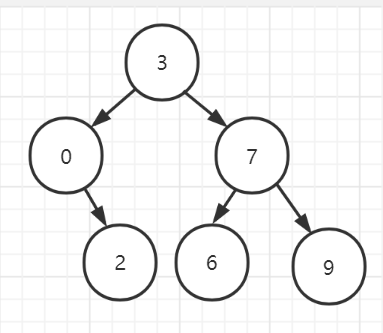
9

测试结果：



1. **求平衡二叉树深度**

测试数据：

3

0

X

2

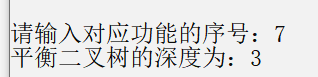
7

6

9

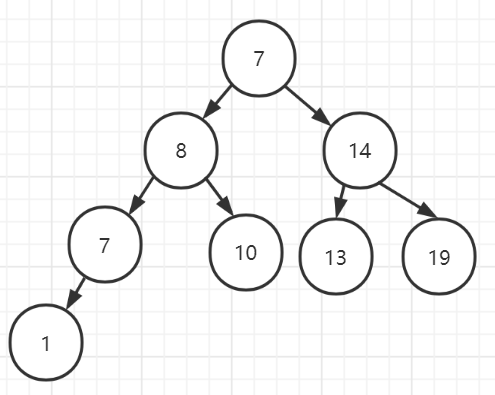
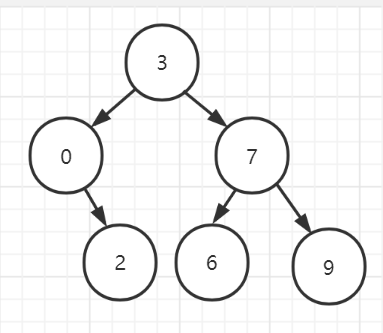
预期结果：3

测试结果：



1. **合并平衡二叉树**

测试数据：9，7,0,3,6,2和7,8,10,11,13,14,1,19



预期结果：

7

1

0

3

2

6

11

9

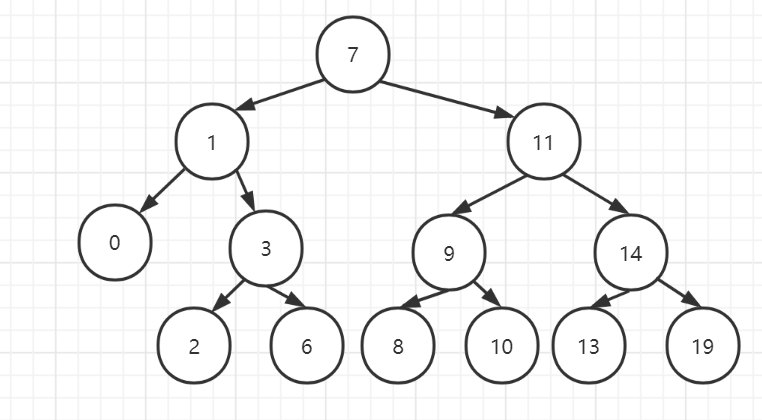
8

10

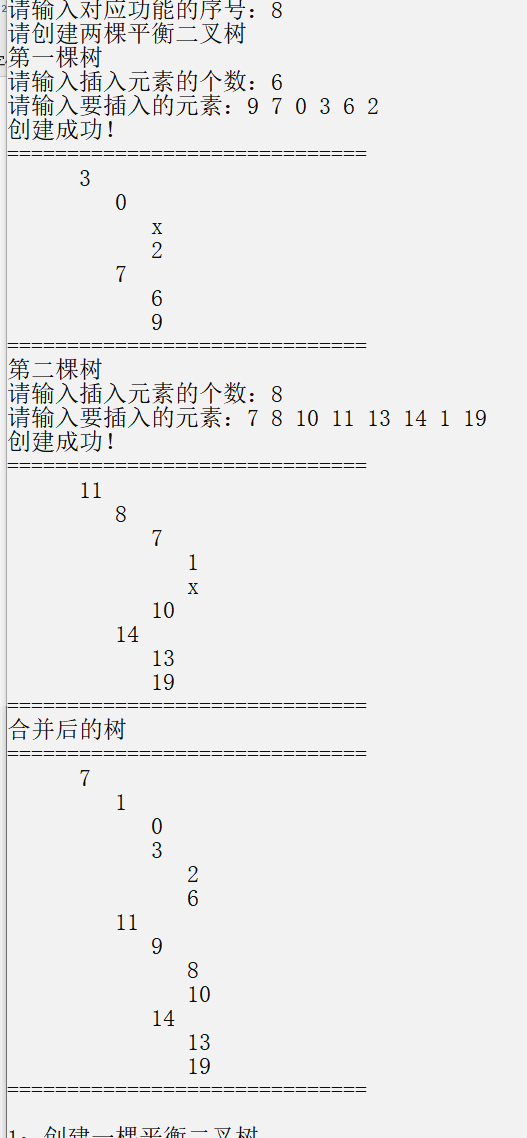
14

13

19



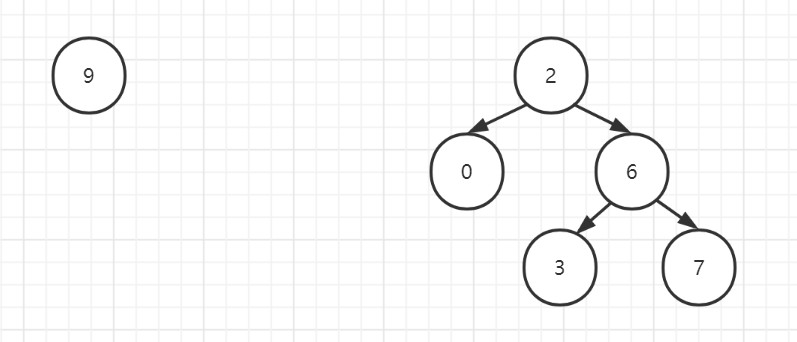
测试结果：



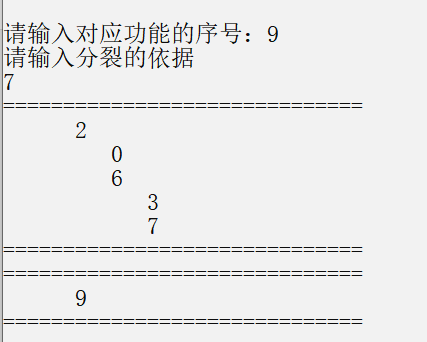
1. **分裂平衡二叉树**

测试数据：在（1）基础上根据7分裂

预期结果：



测试结果：

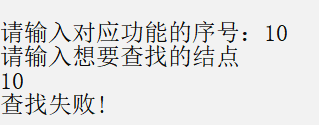


1. **搜索结点**

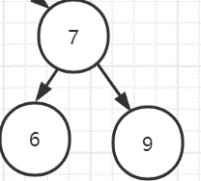
测试数据：在（1）基础上搜索10

预期结果：搜索失败

测试结果：



测试数据：在（1）基础上搜索7

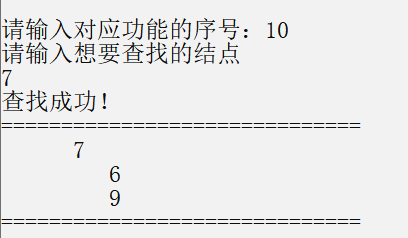
预期结果：

7

6

9

测试结果：



1. **思考与小结**
2. 代码健壮性：

平衡二叉树中的所有结点都是不可重复的，因此考虑到代码的健壮性，在插入操作前要先遍历被插入的树中是否含有待插入结点，避免重复插入。同样的，删除等操作前要先对被操作树进行判空，防止程序崩溃。

1. 时间复杂度：

**查找**：在一个树中查找一个数字，第一次在根节点判断，第二次在第二层节点判断，以此类推，树的高度是多少就会判断多少次。因此树的高度和节点的关系就是以2为底，树的节点总数n的对数。(即查找的时间复杂度为O(logn))。

**插入**：由于在插入前要先走到该结点应该存在的位置并判断是否已经存在在该树中，因此时间复杂度与查找操作的时间复杂度相同（O(logn)）。

**删除：**与插入同理，为O(logn)。

1. 空间复杂度：

需要树形结构，相比顺序存储需要占用更多的空间，但也有链接型数据结构灵活可拓展的优点。因为需要建立排序二叉树，所以空间复杂度为O(n)。