1. **题目**

**实现抽象数据类型BBSTNode。**

**基本操作：**

R\_Rotate(BBSTree &p)

初始条件：p为平衡二叉树

操作结果：以p为根节点的二叉排序树进行右旋转

L\_Rotate(BBSTree &p)

初始条件：p为平衡二叉树

操作结果：以p为根节点的二叉排序树进行左旋转

LeftBalance(BBSTree &T)

初始条件：T为平衡二叉树

操作结果：对二叉平衡树进行左平衡处理

RightBalance(BBSTree &T)

初始条件：p为平衡二叉树

操作结果：对二叉平衡树进行右平衡处理

InsertAVL(BBSTree &T, int e, Status &taller)

初始条件：T为平衡二叉树

操作结果：向平衡二叉树T插入数值为e的结点

Depth(BBSTree T)

操作结果：求平衡二叉树的深度

TreeBf(BBSTree T)

初始条件：T为平衡二叉树

操作结果：求平衡二叉树的bf值

DeleteAVL(BBSTree &T, ElemType e, Status &shorter)

初始条件：T为平衡二叉树

操作结果：删除平衡二叉树中数值为e的结点

PrintBBSTree(BBSTree T)

操作结果：打印平衡二叉树

InorderTraverse(BBSTree T)

操作结果：中序遍历平衡二叉树

1. **存储结构定义**

**公用头文件DSO.h:**

#include<stdio.h>

#include<stdlib.h>

#define LH 1

#define EH 0

#define RH -1

#define TRUE 1

#define FALSE 0

typedef int Status;

typedef int ElemType;

//定义树结点

typedef struct BBSTNode

{

ElemType data;

int bf;//平衡因子（balance factor）

struct BBSTNode \*lchild,\*rchild;

} BBSTNode,\*BBSTree;

1. **算法设计**

void R\_Rotate(BBSTree &p) //以p为根节点的二叉排序树进行右旋转

{

BBSTree L;

L=p->lchild;

p->lchild=L->rchild;

L->rchild=p;

p=L;//p指向新的根节点

}

void L\_Rotate(BBSTree &p) //以p为根节点的二叉排序树进行左旋转

{

BBSTree R;

R=p->rchild;

p->rchild=R->lchild;

R->lchild=p;

p=R;

}

void LeftBalance(BBSTree &T) //对二叉平衡树进行左平衡处理

{

BBSTree L,Lr;

L=T->lchild;

switch(L->bf) //检查T的左子树平衡度，并作相应的平衡处理

{

case LH: //新节点插入在T的左孩子的左子树上，做单右旋处理

T->bf=L->bf=EH;

R\_Rotate(T);

break;

case EH: /\*这个地方需要特别注意，在如果程序涉及到删除结点，这个情况就必须考虑\*/

T->bf = EH;

L->bf = RH;

R\_Rotate(T);

break;

case RH: //新插入节点在T的左孩子的右子树上，做双旋处理

Lr=L->rchild;

switch(Lr->bf)

{

case LH:

T->bf=RH;

L->bf=EH;

break;

case EH:

T->bf=L->bf=EH;

break;

case RH:

T->bf=EH;

L->bf=LH;

break;

}

Lr->bf=EH;

L\_Rotate(T->lchild);

R\_Rotate(T);

}

}

void RightBalance(BBSTree &T) //对二叉平衡树进行右平衡处理

{

BBSTree R,Rl;

R=T->rchild;

switch(R->bf)

{

case RH: //新节点插在T的右孩子的右子树上，要做单左旋处理

T->bf=R->bf=EH;

L\_Rotate(T);

break;

case EH: /\*这个地方需要特别注意，在如果程序涉及到删除节点，这个情况就必须考虑\*/

T->bf = RH;

R->bf = LH;

L\_Rotate(T);

break;

case LH: //新节点插在T的右孩子的左子树上，要做双旋处理

Rl=R->lchild;

switch(Rl->bf)

{

case LH:

T->bf=EH;

R->bf=RH;

break;

case EH:

T->bf=R->bf=EH;

break;

case RH:

T->bf=LH;

R->bf=EH;

break;

}

Rl->bf=EH;

R\_Rotate(T->rchild);

L\_Rotate(T);

}

}

//向平衡二叉树T插入数值为e的结点

Status InsertAVL(BBSTree &T, int e, Status &taller)//变量taller反映T长高与否

{

if(NULL == T)

{

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

T->data = e;

T->lchild = T->rchild = NULL;

T->bf = EH;

taller = TRUE;

}

else

{

if(e==T->data)//不插入

{

taller = FALSE;

return FALSE;

}

if(e<T->data)

{

if(FALSE == InsertAVL(T->lchild,e,taller))//未插入

return FALSE;

if(taller) //以插入左子树，且左子树变高

{

switch(T->bf)

{

case LH: //原本左子树比右子树高，需要做左平衡处理

LeftBalance(T);

taller = FALSE;

break;

case EH: //原本左右子树等高，现因左子树增高而树增高

T->bf=LH;

taller = TRUE;

break;

case RH: //原本右子树比左子树高，现在左右子树等高

T->bf=EH;

taller = FALSE;

break;

}

}

}

else

{

//应在T的右子树中搜寻

if(FALSE == InsertAVL(T->rchild,e,taller))

return FALSE;

if(taller) //插入右子树，且右子树长高

{

switch(T->bf)

{

case LH: //原本左子树比右子树高，现在左右子树等高

T->bf = EH;

taller = FALSE;

break;

case EH: //原本左右子树等高，现在右子树变高

T->bf = RH;

taller = TRUE;

break;

case RH: //原本右子树比左子树高，现在需做右平衡处理

RightBalance(T);

taller = FALSE;

break;

}

}

}

}

return TRUE;

}

//求平衡二叉树的深度

int Depth(BBSTree T)

{

if(NULL == T) return 0;

int l,r;

l = Depth(T->lchild);

r = Depth(T->rchild);

return l>r?(l+1):(r+1);

}

int TreeBf(BBSTree T)

{

return Depth(T->lchild)-Depth(T->rchild);

}

//删除平衡二叉树中数值为e的结点

Status DeleteAVL(BBSTree &T, ElemType e, Status &shorter)

{

if(T == NULL)

{

return FALSE;

}

else if(e == T->data) //删除 此处删除算法原理同二叉搜索树

{

BBSTree temp = T;

if(T->lchild == NULL)

{

T = T->rchild;

free(temp);

shorter = TRUE;

}

else if(T->rchild == NULL)

{

T = T->lchild;

free(temp);

shorter = TRUE;

}

else

{

BBSTree p = T->lchild;

while(p->rchild)

{

p = p->rchild;

}

T->data = p->data;

DeleteAVL(T->lchild,p->data,shorter);

if(shorter == TRUE)

{

switch(T->bf)

{

case LH:

T->bf = EH;

shorter = TRUE;

break;

case EH: //子树高度不降低的特殊情况:即左右都有结点，删除一个不影响子树高度

T->bf = RH;

shorter = FALSE;

break;

case RH:

RightBalance(T); //处理右平衡

if(T->rchild->bf == EH) / \*删除时的特殊情况：右孩子结点bf为0，子树高度不变\*/

{

shorter = FALSE;

}

else

{

shorter = TRUE;

}

break;

}

}

}

}

else if(e < T->data) //左子树中继续查找

{

if(!DeleteAVL(T->lchild,e,shorter))//删除失败直接return flase

{

return FALSE;

}

if(shorter) //左子树中结点删除成功,并且子树高度降低

{

switch(T->bf)

{

case LH:

T->bf = EH;

shorter = TRUE;

break;

case EH: /\*子树高度不降低的特殊情况:即左右都有结点，删除一个不影响子树高度\*/

T->bf = RH;

shorter = FALSE;

break;

case RH:

RightBalance(T); //处理右平衡

if(T->rchild->bf == EH)/\*删除时的特殊情况：右孩子结点bf为0，子树高度不变\*/

{

shorter = FALSE;

}

else

{

shorter = TRUE;

}

break;

}

}

}

else //右子树中继续查找

{

if(!DeleteAVL(T->rchild,e,shorter))

{

return FALSE;

}

if(shorter)

{

switch(T->bf)

{

case LH:

LeftBalance(T);

if(T->lchild->bf == EH) /\*删除时的特殊情况：左孩子结点bf为0，子树高度不变\*/

{

shorter = FALSE;

}

else

{

shorter = TRUE;

}

break;

case EH:

T->bf = LH;

shorter = FALSE;

break;

case RH:

T->bf = EH;

shorter = TRUE;

break;

}

}

}

return TRUE;

}

//打印平衡二叉树

void PrintBBSTree(BBSTree T)

{

if(T)

{

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

int i = TreeBf(T);

printf("[%d]",i);

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

{

printf("(");

PrintBBSTree(T->lchild);

printf(",");

PrintBBSTree(T->rchild);

printf(")");

}

}

}

//中序遍历平衡二叉树

void InorderTraverse(BBSTree T)

{

if(NULL == T) return;

InorderTraverse(T->lchild);

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

InorderTraverse(T->rchild);

}

1. **测试**

int main()

{

int i,select,data;

ElemType A[]= {3,2,1,4,5,6,7,10,9,8}; //预先插入一些数据

BBSTree T = NULL;

Status taller = FALSE, shorter = FALSE;

for(i=0; i<10; i++)

{

InsertAVL(T,A[i],taller);

}

printf("现在的平衡树结点值为：\n");

PrintBBSTree(T);

printf("\n");

do

{

printf("输入 1 插入结点\n");

printf("输入 2 删除结点\n");

printf("输入 3 打印平衡二叉树\n");

printf("输入 4 退出程序\n");

printf("输入 5 中序遍历平衡二叉树\n");

printf("请输入你的选择\n");

scanf("%d",&select);

switch(select)

{

case 1:

printf("请输入你要插入的结点值\n");

scanf("%d", &data);

if(FALSE == InsertAVL(T,data,taller))

printf("插入出错，请检查插入的数值是否有误\n");

else printf("插入成功！\n");

break;

case 2:

printf("请输入你要删除的结点值\n");

scanf("%d", &data);

if(FALSE == DeleteAVL(T,data,shorter))

printf("删除出错，请检查删除的数值是否有误\n");

else printf("删除成功!\n");

break;

case 3:

printf("现在平衡二叉树的所有结点为：\n");

PrintBBSTree(T);

printf("\n");

break;

case 4:

break;

case 5:

InorderTraverse(T);

printf("\n");

break;

default:

printf("输入有误，请重新选择！\n");

break;

}

}

while(select!=4); //当输入4时退出程序，否则重新进入循环体

printf("程序已退出，谢谢！\n");

return 0;

}

1. **思考与小结**
2. 在左平衡和右平衡处理时，LR型和RL型的旋转需要特别注意相关结点bf值的变化，最好画出图来判断，避免出错。
3. 涉及删除操作时，左平衡和右平衡都要考虑子结点的EH(等高)情况，这一点在插入操作中无须涉及到，所以应当留意。
4. 测试函数最好有中序遍历打印和从上到下输出打印平衡二叉树的调用，这个便于检测代码是否正确。
5. 平衡二叉树涉及到的情况比较多，容易出现错误，所以测试的数据尽可能多一点。