# 3基于二叉链表的二叉树实现

## 3.1 问题描述

**3.1.1 总述**

通过二叉链表作为二叉树的物理结构，创建一个简易的菜单演示系统。该系统定义了初始化二叉树、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树、求二叉树深度、获得结点、插入子树、删除子树和遍历等运算对应的函数，并给出了相应的操作提示，并可选择以文件的形式进行存储和加载。在该系统中，可以通过选择序号来初始化并操作不同的树。

**3.1.2 多树操作与附加功能**

在本程序中，定义了一个结构体SqList，其中包含二叉树数组tree，定义一个SqList L对象来表示多个二叉树的总体，通过L.tree[i]来访问所要操作的树；在操作的过程中可以通过标号21来进行所要操作的树的序号的选择。

附加功能：

1）选择13号功能，使用文件存储二叉树的各个结点的信息，文件中按照前序遍历的方式存储二叉树中每一个结点的整形变量关键字key与字符数组others，存入时最后存入一个表示结束的结点信息，关键字为-1，内容为”jieshu”；

2）选择14号功能，读取已经存在的存储二叉树各结点内容的文件来构建一个二叉树，包括读取每一个结点的关键字key与内容others，当读取到表示结束的结点信息-1，“jieshu”时退出；

3）选择15号功能，用于判断当前进行操作的二叉树是否是一棵平衡二叉树； 4）选择16号功能，可以输入两个结点的关键字key1，key2，来查找这两个关键字所在的结点的最近的公共祖先；

5）选择17号功能，输入两个关键字key1,key2，可以判断这两个关键字所在的结点是否为堂兄弟，即深度相等但双亲不为同一结点的两个结点；

6）选择18号功能，输出当前所操作的二叉树的直径，即树中相距最远的两个结点的距离；

7）选择19号功能，翻转二叉树，即将二叉树的每一个结点的左右孩子交换；

8）选择20号功能，输入一个目标和sum，判断树中是否存在从根开始到叶子结点的路径，使得路径上的所有结点的关键字之和为sum。

9）选择21号功能，得到当前所操作的树的最大路径和。

**3.2 系统设计**

**3.2.1 数据的物理结构**

typedef struct {

KeyType key;

char others[20];

} TElemType; //二叉树结点类型定义

二叉树结点中的数据域包括关键字key和存储内容的字符数组others；

typedef struct BiTNode { //二叉链表结点的定义

TElemType data;

struct BiTNode\* lchild, \* rchild;

} BiTNode, \* BiTree;

二叉链表的每一个结点包括数据域TelemType data，即上面定义内容，然后还要包括指向左孩子的结点指针lchild，右孩子指针rchild;

typedef struct {

BiTree T;//创建二叉树用的指针T

char name[20];//用于保存树的名称

}LElemType;

结构LelemType代表一棵二叉树，它包含指向二叉树根节点的BiTree类型的指针T,以及存储这棵树名称的字符数组name;

typedef struct {

LElemType tree[20];//多个树进行操作

int listsize;

}SqList;

结构SqList表示树的列表，其中包含LelemType类型的数组tree存储多个树，listsize为所容纳的树的容量。

**3.2.2 函数运算**

程序以函数的形式定义了初始化二叉树、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树、求二叉树深度、获得结点、插入子树、删除子树，二叉树的前序，中序，后序，层序遍历，求公共祖先，判断堂兄弟结点和二叉树反转等运算对应的函数，在进行函数功能的实现时都会有对应的输入提示与结果表现。

函数的定义：

1）创建二叉树（CreateBiTree）：通过输入的前序遍历序列构建一棵二叉树，初始条件是一个待输入的包含二叉树结点关键字与内容信息的类型为TelemType的数组definition以及一个空的BiTree类型的结点T；操作结果是创建一个二叉树；

2）清空二叉树（ClearBiTree）：清空二叉树的内容，初始条件是一棵已经构建好的二叉树，允许是空树，操作结果是释放树中每一个结点的空间，根节点置为NULL；

3）二叉树深度（BiTreeDepth）：求二叉树的深度，初始条件是一棵以构建好的二叉树，操作结果是将二叉树的深度作为返回值返回；

4）结点定位（LocateNode）：定位关键字所在的结点输出它包含的内容；初始条件是已构建的二叉树，以及所要查找的结点的关键字key，操作结果是找到结点后返回这个BiTNode\*类型的指针；

5）结点赋值（Assign）：找到所要重新赋值的结点后用新的data域进行替代从而完成赋值；初始条件是已构建的二叉树，待重新赋值的结点的关键字key，以及用于替换旧的data域的新的TElemType型变量value；

6）查找兄弟结点（GetSibling）：输入一个结点的关键字，查找这个结点的兄弟结点；初始条件为已构建的二叉树，关键字key，操作结果是返回这个结点的兄弟结点，无兄弟则返回NULL；

7）插入结点（InsertNode）：将一个新的结点插入到参数控制的对应的位置，初始条件为已构建的树，作为新结点双亲的结点的关键字，表示插入左边还是右边的整型变量LR，新结点的data域—TelemType型的变量；操作结果是返回插入后的树；

8）删除结点（DeleteNode）：删除一个结点，重新布置它的子结点，初始条件为已构建的二叉树，所要删除的结点的关键字key，删除成功返回OK，否则返回ERROR；

9）先序遍历树（PreOrderTraverse）：输出二叉树的结点的内容的先序遍历，成功输出后返回OK；

10）中序遍历树（InOrderTraverse）：输出二叉树的结点的内容的中序遍历，成功输出后返回OK；

11）后序遍历树（PostOrderTraverse）：输出二叉树的结点的内容的后序遍历，成功输出后返回OK；

12）层序遍历树（LevelOrderTraverse）：输出二叉树的结点按层序从上到下，层中从左到右的顺序输出结点中内容，返回OK；

13）保存文件（SaveBiTree）：将二叉树的各个结点的关键字和内容数组都保存入一个文件中，在文件的最后写入一个表示保存结束的结点,文件名就是初始化创建树时输入的树的名称；

14）读取文件（LoadBiTree）：通过读取保存了二叉树各个结点内容的文件来创建一棵二叉树；

15）平衡树判断（IsBalanced）：判断一棵二叉树是否为二叉平衡树，平衡树就是每一个结点的左右子树的深度差绝对值不超过1，若是则返回1，否则返回0；

16）最近公共祖先（CommonAncestor）：输入两个关键字key1,key2，返回两个关键字所在结点的最近的公共祖先；

17）堂兄弟结点判断（IsCousins）：输入两个关键字key1,key2，判断两个关键字所在的结点是否为堂兄弟结点，若是则返回TRUE，否则返回FALSE；

18）树的直径（DiameterOfTree）：得到当前操作的树的直径，即相距最远的两个结点的距离，双亲与他的左右孩子的距离均记为1；

19）翻转二叉树(InvertTree)：将当前所操作的树进行翻转操作，交换所有的左右孩子结点；

20）距离之和（HasPathSum）：输入一个整形常量sum,判断当前操作的二叉树是否有一条从根出发到叶子结点的路径，使得路径上的所有结点的关键字之和为sum；若是则函数返回1，否则返回0。

21）最大路径和（maxPathSum）：路径和为从树中一个结点到另一个结点的路径上所有结点的关键字的和，函数的目的是得到它的最大值。

**3.2.3 输入与输出说明**

进入系统后首先出现功能菜单，首先输入所要操作的线性表的序号，这里允

许在1~20号上操作；菜单会罗列20个选择项，系统会提示首先选择功能20来选择所要操作的树的序号，选择完毕后就可以选择功能对应的序号进行相应操作，倘若不进行所要操作的表的序号选择，则会默认在序号为1的树上进行操作。

系统通过switch-case语句进行对应的功能操作，每一个操作完毕后按回车或其他字符跳出当前操作，继续执行while循环选择功能并实现，在每一次操作结束后系统都会显示操作的结果。对于树中元素操作，可以通过功能9树的前序遍历和功能10树的中序遍历来验证每一个功能实现是否成功。当在系统中输入0后会退出系统。

当要操作另一棵树时，只需在操作所对应的树时先选择功能20，输入所要操作的树的序号，即可转换为对新的树进行操作，原来的树中的数据元素依然会保存。

## 3.3 系统实现

**3.3.1 环境说明**

本系统在Microsoft Visual Studio2019上执行，编译器为MSVC++ 14.2 \_MSC\_VER == 1920.并在其上编译运行。

**3.3.2 预定义变量**

//数据元素类型定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

//全局变量

int tail = 0;//文件操作时作为位置后移的标志

int p = 0;//记录树中所有关键字时用

int k[200];//存储树中所有的关键字

int place = 0;//初始建树时用

int MaxDistance;//用于求二叉树直径

**3.3.3 基于二叉链表的二叉树的实现**

本实验通过结构体SqList来存储多个树的信息，进入系统后选择20号功能来对所要操作的树的序号进行选择，选择完毕后即可进行多种功能的演示。在每一次选择功能进入case语句时都会首先对树是否为空树进行判断，若是则会显示“树不存在”，下述对各个函数算法设计过程的描述过程除创建树外均建立在前提条件满足，即已构建的树上进行操作。

1）status BuildTree(BiTree& T, TElemType definition[])

创建二叉树借助了TelemType类型的数组definition来完成，definition接收输入二叉树的结点data域的前序遍历的序列，即前序遍历时的每一个结点的关键字key和内容others，当接收到大小为-1的关键字时退出输入循环，然后通过CreateBiTree函数来创建二叉树；

在CreateBiTree函数中，先通过辅助函数check1，来检测输入definition的过程中是否会出现结点关键字相同的情况；

辅助函数1：bool check1(TElemType definition[])

定义一个整形数组t，初始化所有元素为0，t[i]代表关键字i出现的次数，对definition数组进行遍历，当遇到key为0时说明是空结点，无需计数，否则判断t[key]是否为1，是1说明此前出现过，即出现关键字重复，给CreateBiTree函数返回false；否则t[key]++；遍历所有关键字后若不返回false则返回true;

CreateBiTree接收到check1的返回值若为false,则返回ERROR;否则执行辅助函数BuildTree来进行递归建立结点；

辅助函数2：status BuildTree(BiTree& T, TElemType definition[])

使用全局变量place表示所遍历到的definition数组的位置，初始化为0，当当前位置的key为-1时表示构建完毕，返回0，否则继续对key的值进行判定，若大小为0，则说明这个结点为空结点，令T为NULL，返回OK；否则当前所要构建的结点是有内容的，先给T申请空间，然后将definition[place]中的key赋值给T的数据域的key，然后通过一个遍历将definition[i]中的others的内容全部复制给T的数据域的others.赋值完毕后place后移，由于是按先序遍历进行输入，所以先调用自身递归遍历左子树T->lchild，左子树建立完毕后place继续后移，调用自身递归建立右子树T->rchild；直到遇到key为-1时退出，二叉树构建完毕，返回OK；

2）status ClearBiTree(BiTree& T)

清空一个二叉树，需要把二叉树所有结点的空间释放，若T为NULL则直接返回OK，若T的左孩子不为空，则递归清空左子树，若T的右孩子不为空，则递归清空右子树，左右子树均清空后，使用free函数释放T指向的结点的空间，并令T指向NULL，清空完毕后返回OK；

3）int BiTreeDepth(BiTree T)

求二叉树的深度，定义整形变量depth表示深度，初始化为0，由于函数的返回值为T作为根节点的树的深度，当T不为空时，定义整形变量lchilddepth，让它为函数BiTreeDepth(T->lchild)的返回值，即通过递归的方式得到左子树的深度，同样的方法定义rchilddepth为右子树的深度，则T的深度为左右子树的深度的较大值再加上1；递归求得深度depth后将其作为返回值返回；

4）BiTNode\* LocateNode(BiTree T, KeyType e)

定位结点，若T为空，则直接返回空结点，否则判断T指向的结点的data域的关键字key是否等于e，若是，则返回T，否则递归地去左子树中判断是否存在关键字大小为e的结点，若有则返回在左子树中找到的结点指针，没找到则递归地去右子树中判断是否存在关键字大小为e的结点，若是则返回右子树中找到的结点指针；

5）status Assign(BiTree& T, KeyType e, TElemType value)

结点赋值，由于赋值操作中，改变了某个结点的关键字和内容属性，所以当关键字改变后倘若与树中原有的关键字集发生重复则是不合理的赋值，并且我们还要求所要改变的结点的关键字e应该是在树中存在的，因此应该首先通过一个辅助函数，这里定义为check2，来判断赋值操作是否合乎规范；

辅助函数1：status check2(BiTree& T, KeyType e, TElemType value)

在辅助函数1中先调用辅助函数2checkkey对赋值条件进行筛选；

辅助函数2：void checkkey(int k[], BiTree& T)

整形数组k用于存储所有的关键字，若T为空直接退出，否则按前序遍历的顺序依次存储结点的data域的关键字；

存储完毕后，首先定义一个标记tag，初始化为0，通过一个for循环，当数组k中存在一个元素与e相等时，说明找到了这个应该被重新赋值的结点，tag赋值为1，退出循环。对tag的值进行判断，当为0,时说明循环完毕未能找到关键字，给Assign函数返回INFEASIBLE；当Assign接收到INFEASIBLE时，直接返回INFEASIBLE；否则判断下一项，是否有关键字重复，采用数组计数，sum数组初始化为0，对数组k进行循环遍历，当遍历到关键字等于e的元素时不是让sum[e]自增，而是让替换后的新的关键字的数量自增，每一次增加数量后判断当前位置的关键字的数量是否超过1，若是则说明更新后出现关键字重复，返回FALSE,随后Assign返回ERROR。

当check2函数返回TRUE时说明满足了结点赋值的条件，此时调用辅助函数Ass来递归地进行赋值；

辅助函数3：void Ass(BiTree& T, KeyType e, TElemType value)

当T为空树时直接返回，当T中的data域的key与e相等时说明找到了需要赋值的结点，由于data的类型为TElemType，所以直接令T中的data等于新的数据域value，然后返回；以上条件都不满足时，递归地在左子树与右子树中进行赋值，调用自身时参数改为为T->lchild与T->rchild即可。

赋值成功后返回Assign函数，函数返回OK；

6）BiTNode\* GetSibling(BiTree T, KeyType e)

得到关键字e所在结点的兄弟结点，当这个结点为根节点时它一定没有兄弟结点，所以首先对根节点中的key进判断，若相等直接返回NULL，表示没有兄弟结点；若T指向空结点则直接返回NULL；否则对T指向的结点的左孩子进行判断，当左孩子存在且左孩子的关键字与e相等时说明找到该节点，那么此时右孩子就是它的兄弟结点，返回了空结点也没关系，主函数会对结点类型进行判断，得到空结点时输出没有兄弟结点，对右孩子的判断也一样，不过符合条件时返回左孩子；

当左孩子不满足条件时，递归地在左子树进行搜寻，当在左子树中能查找到兄弟结点时返回左子树的兄弟结点；右子树的递归遍历亦然。若在递归遍历的过程中没有返回存在的兄弟结点，则函数结尾返回NULL；

7）status InsertNode(BiTree& T, KeyType e, int LR, TElemType c)

插入结点时同样要考虑是否出现关键字重复的情况，首先通过辅助函数check3来检查；

辅助函数1：bool check3(BiTree& T, int e)

当T为空时直接返回true给InsertNode，当T指向的结点的关键字等于e时说明存在结点关键字的重复，返回false，否则递归判断左右子树，返回true的条件是check3(T->lchild,e)和check3(T->rchild,e)同时返回true；

当InsertNode调用check3返回false时直接返回ERROR；否则进行插入操作，首先对特殊情况进行处理，即LR为-1时，此时是把新结点作为根结点，定义一个BiTree型的指针newT,申请空间后将data域赋值为新的TElemType类型c，左孩子指向T，右孩子指向NULL，然后修改根节点T为newT,返回OK；

若LR不为-1，则通过辅助函数Insert来进行插入；

辅助函数2：status Insert(BiTree& T, KeyType e, int LR, TElemType c)

当T为NULL时返回ERROR；当T的数据域的关键字与e相等时说明找到了结点，此时对LR进行判断，当LR为0时，结点应该作为左孩子，这时需要对结点进行分类讨论，当结点的左子树为空时可以直接为左孩子申请空间，然后将待插入的TElemType 类型的c作为左孩子的数据域，然后让新结点的左右孩子置为空；而如果左孩子不为空，按要求需改变T的左孩子为新结点，T原来的左子树作为新结点的右子树，先用BiTree类型的指针记录T的左孩子，然后定义一个BiTree类型的指针newnode，申请空间后将数据域赋值为c，然后让T的左孩子指针指向newnode，而newnode的左孩子置为NULL，右孩子指向先前记录的左孩子node；

当LR为1时则是作为右孩子插入，思路与作为左孩子插入时相同，若没有右孩子则直接插入，否则记录右孩子，新建结点赋值后作为右孩子插入，再将记录了原来的右子树作为新结点的右子树；插入成功返回OK；若T指向的结点中的关键字不等于e，则递归判断在左子树中是否能够插入，若返回OK则给InsertNode返回OK，同样地在递归判断右子树，能够插入则返回OK；遍历整个树结束未能插入则返回ERROR;

8）status DeleteNode(BiTree& T, KeyType e)

删除特定结点，首先判断T是否为空，若是返回ERROR；若T指向的结点的关键字等于e说明找到了要删除的结点，按要求要将被删除结点的右子树移到左子树中的最右边的结点的右孩子指针上，那么我们需要找到找到左子树中的最右边的结点，即从根节点开始向右子树搜索到的第一个右孩子指针为空的结点；借助辅助函数find来完成，它的返回值是所要找的最右边结点的指针；

辅助函数1：BiTree find(BiTree& T)

判断结点的右孩子指针是否为NULL，若是则说明找到了目标结点，直接返回；否则返回在右子树中查找的结果；

由于删除根与删除其他结点的步骤把不同，所以删除根的语句在DeleteNode中完成，递归删除树中其他位置的结点通过辅助函数完成，若是删除根，先标记一下左孩子，使用find函数找到它的最右结点，让最右结点的右子树指向T的右子树完成转移，然后修改根的位置为标记的结点返回OK；

根节点的关键字不为e时递归调用辅助函数来完成删除操作；

辅助函数2：status Delete(BiTree& T, KeyType e)

首先判断T的左孩子是否存在，若存在判断关键字是否为e，若是则说明找到结点，此时应分为四个类别讨论，当待删除结点的左右孩子均为NULL时直接释放内存，并使指针置为NULL；当两个孩子都存在时，需要标记左孩子（之后作为T的新的左孩子），定义一个BiTree类型指针node指向find找到的左子树中的最右边的结点，然后让node的右孩子结点指向T的左孩子的右孩子结点完成转移，然后修改T的左孩子结点为之前标记的左孩子的左孩子结点；当T的左孩子只有左孩子不为空时，直接标记tem指向T的左孩子，然后让tem的左孩子作为T的新的左孩子，释放tem的空间；当T的左孩子只有右孩子不为空时，原理同上，只不过标记右孩子再释放空间而已；

而如果左孩子结点的关键字不为e，由于左子树存在，则可以调用自身，将参数的指针改为T->lchild进行递归搜寻；

左子树搜寻完毕后对右子树进行判断，右子树不为空时，原理同左子树，先对关键字进行判断，若相等依然分四种情况进行删除，关键字不等则在右子树中递归搜寻，成功删除后给DeleteNode返回OK，DeleteNode给主函数返回OK,若Delete返回ERROR则删除失败，DeleteNode返回ERROR；

9）status PreOrderTraverse(BiTree T, void (\*visit)(BiTree))

首先定义一个辅助函数用于访问结点的数据域，它在功能9,10,11均用到；

void visit(BiTree T)

它的功能是打印传入的BiTree类型的指针T的数据域的关键字key和内容others；

使用非递归方式进行前序遍历，定义一个结点指针BiTree型的数组st,作为栈，定义整型变量top指向栈的顶部，初始化为0，当根不为空，根入栈，top自增1，当top不为0时，每次取栈顶元素，非空则用visit函数进行打印内容，由于是先序遍历，要想先输出左子树的内容，则要右子树的结点指针先入栈，对右孩子指针进行判断，非空入栈，top++，再判断左孩子，非空入栈，top++；这样回退时就先取到左子树的结点，满足前序遍历的方式；遍历结束后返回OK；

10）status InOrderTraverse(BiTree T, void (\*visit)(BiTree))

当T为空时直接返回OK，中序遍历则先递归输出左子树；然后访问双亲结点T的内容，最后递归输出右子树的内容；遍历结束后返回OK；

11）status PostOrderTraverse(BiTree T, void (\*visit)(BiTree))

当T为空时直接返回OK，后序遍历先递归输出左子树，然后递归输出右子树，最后打印双亲结点T的内容；遍历结束后返回OK；

12）status LevelOrderTraverse(BiTree T, void (\*visit)(BiTree))

树的层序遍历需要借助队列数据结构，可以用数组来实现，定义入队指针in，出队指针out，用整型变量表示，当in大于out时说明未输出完毕，首先让根节点入队，每一次对out所指向的指针进行访问，访问后将out所指向的结点的左孩子入队，右孩子入队，同时out后移一位，由于树中结点有限，结点指针入队完毕后in会停留，因此out不断后移输出内容，最后必定会和in相等，此时退出循环，层序遍历完毕，返回OK；

13）status SaveBiTree(BiTree T, char FileName[])

保存二叉树结点的信息为文件，首先定义文件指针，倘若为空树直接返回ERROR；由于要借助递归，需要传递文件指针，因此借助辅助函数完成写入；

int write(BiTree T, FILE\* fp)

当T为空结点时，写入的关键字为0，内容为空字符’\0’，返回；否则读入关键字与内容，调用自身写入左子树的内容，然后调用自身写入右子树的内容；

在write函数写入完毕后，为了维护后续读取文件时的连贯性，写入一个表示读取结束的结点，它的关键字记为-1，字符数组others定义为”jieshu”；

14）status LoadBiTree(BiTree& T, char FileName[])

加载含有树的各个结点内容的文件来构建一个二叉树，首先定义文件指针fp，定义整形数组a存储关键字key，定义二维字符数组存储结点的字符串，当读取文件未读取到结束结点时，将读取到的关键字和字符串分别赋给a和b数组，遇到关键字为-1时表示读取结束，退出while循环，借助辅助函数来创建二叉树；

int create(BiTree& T, int a[100], char b[100][20])

函数首先为结点申请空间，全局变量tail访问数组a，当访问到0时说明未空结点，直接置为NULL，tail后移；否则将a[tail]和b[tail]复制给T的数据域，左右孩子结点暂时指向NULL；然后去递归地构建左子树和右子树；

构建完毕后返回，LoadTree返回OK；

15）status IsBalanced(BiTree T)

判断一棵树是否是平衡树，若是空树直接返回true；否则对于任意一个结点而言，它满足的条件是左右子树的深度差不超过1，并且要求IsBalanced(T->lchild)和IsBalanced(T->rchild)都能返回true，即左右孩子结点都满足条件，如此便能通过递归的方式判断二叉平衡树；

16）BiTNode\* CommonAncestor(BiTree T, int key1, int key2)

找到两个结点的最近公共祖先，思路是首先利用LocateNode函数定位两个结点，倘若有空结点说明输入有误，返回NULL；否则借助辅助函数SearchAncestor来得到最近公共祖先；

BiTNode\* SearchAncestor(BiTree T, BiTNode\* p, BiTNode\* q)

当根节点为p或q中任意一个时说明根节点就是最近公共祖先，直接返回T，否则定义BiTNode\*型的指针left表示在左子树中寻找公共祖先的结果，定义BiTNode\*型的指针right来表示在右子树中寻找公共祖先的结果，若left和right均不为空说明两个目标结点分列T的两侧，那么说明T就是离两个结点最近的公共祖先，否则对left和right进行判断，若left不为空而right为空，说明两结点均在左子树中且最近公共祖先为left，若right为空而left不为空，说明两结点均在右子树中且最近公共祖先为right，作为返回值返回；

CommonAncestor函数将SearchAncestor的返回值作为返回值；

17）status IsCousins(BiTree T, int key1, int key2)

判断两个结点是否为堂兄弟结点，即在同一层且双亲不同，思路是先用LocateNode函数定位两个目标结点，若存在空结点直接返回INFEASIBLE，说明输入有误；定义一个BiTNode\*类型的队列q，设置一个标签flag，初始化为2，首先根结点指针T入队，当队列不为空时，令整型变量n记录队列的大小，遍历当前队列中所有元素（即同一层的结点），取队首元素，并弹出，当队首结点的左右结点关键字分别对应key1和key2时说明两个结点为亲兄弟，返回false；

否则左孩子入队，同时判断左孩子的关键字是否为key1和key2的一个，若是说明在这一层找到一个目标结点，flag自减1，对右孩子进行同样的操作，n个结点遍历完后，进行flag值的判断，若为1，说明在这一层中只找到一个目标结点，不满足堂兄弟结点在同一层这个条件，返回false；若为0，说明在这一层找到了两个目标节点，又没有在之前返回FALSE说明不为亲兄弟，那么就是堂兄弟结点，返回TRUE;整棵树遍历完毕未返回值则返回FALSE;

18) int DiameterOfTree(BiTree T)

求树的直径，即最远点的距离，定义整形变量MaxDistance表示二叉树的直径，初始化为1，在更新这个值的过程中需要不断求得结点的左右子树的深度，借助辅助函数完成；

int UpdateDepth(BiTree T)

具体思路其实很简单，除了更新MaxDepth外与求树的深度并无区别；当访问到空结点时返回深度0，定义整型变量L表示左子树的深度，R为右子树的深度，更新MaxDistance就是自身和L+R+1的最大值（左右子树间路径需加上在双亲结点的过渡1个单位长度），它的返回值依然是以T为根节点的子树的深度max(L,R)+1;当函数递归执行完毕时MaxDistance就更新为最大的左右子树的结点的距离之和，但由于累计的是深度和，转化为路径时在根节点处重复计算了一个1，因此DiameterOfTree的返回值为MaxDistance-1;

19) status InvertTree(BiTree T)

翻转二叉树，当T指向NULL时直接返回OK；思路是从上向下翻转，即对于每一个根节点，先交换它的左右孩子，定义BiTNode\*型指针tem指向左孩子结点，然后让左孩子指针指向右孩子，再让右孩子指针指向tem,然后递归地调用自身去翻转以T的左孩子结点为根的树和以T的右孩子结点为根的树，这样执行到最深处就完成了所有结点的翻转；

20）status HasPathSum(BiTree T, int sum)

判断是否有从根开始到叶子结点的路径使得路径上所有关键字和为目标和sum；当T为空时直接返回FALSE；当T的左右孩子结点均为空指针时只需判断T中的关键字是否等于sum，是则返回TRUE否则返回FALSE;

随后的思路是：每走过一个结点，相当于这个结点以下的路径所要得到的目标和为sum减去已经走过的路径和，也就是说调用自身时，若选择走左子树，那么指针参数就是T->lchild，sum修改为sum-T->data.key；右子树思路相同；所以对于当前结点而言，这个函数返回TRUE的条件是左右子树中有一种走法满足返回TRUE即可，也就是HasPathSum(T->lchild, sum - T->data.key) || HasPathSum(T->rchild, sum - T->data.key)；若存在一条路径则返回TRUE否则返回FALSE.

21）int maxPathSum(BiTree T)

求树中最大路径和，即一条路径中所有结点的关键字和最大（不一定经过根节点），定义全局变量MaxNum,初始置为-1，在辅助函数中进行递归调用自身来更新这个值；

int maxGain(BiTree node)

由于要使路径和最大，可以考虑对于每一个结点，当它作为路径的拐点时所能贡献的最大路径和，而这个最大路径和来自于左子树中的最大路径和右子树中的最大路径的和，再加上这个结点的关键字，将三者和用整形变量price记录并与MaxNum比较进行更新，更新为较大者，而每一层遍历返回以该结点为根节点的树的最大路径（单条路径），当结点为空时直接返回0，更新完毕后，MaxNum的值就是树中的最大路径和。

**3.4测试用例**

在命令行中执行生成的exe文件，对每一个函数进行测试。

首先在空树下选择几个函数执行；

空树测试用例表

|  |  |  |
| --- | --- | --- |
| 测试函数 | 预期结果 | 操作结果 |
| ClearBiTree | 显示“二叉树不存在” |  |
| BiTreeDepth | 显示“二叉树不存在” |  |
| LocateNode | 显示“二叉树不存在” |  |
| Assign | 显示“二叉树不存在” |  |
| GetSibling | 显示“二叉树不存在” |  |
| InsertNode | 显示“二叉树不存在” |  |
| DeleteNode | 显示“二叉树不存在” |  |
| PreOrderTraverse | 显示“二叉树不存在” |  |
| InOrderTraverse | 显示“二叉树不存在” |  |
| PostOrderTraverse | 显示“二叉树不存在” |  |
| LevelOrderTraverse | 显示“二叉树不存在” |  |
| SaveBiTree | 显示“二叉树不存在” |  |
| LoadBiTree | 显示“文件加载失败!” |  |
| IsBalanced | 显示“二叉树不存在” |  |
| CommonAncestor | 显示“二叉树不存在” |  |
| IsCousins | 显示“二叉树不存在” |  |
| DiameterOfTree | 显示“二叉树不存在” |  |
| InvertTree | 显示“二叉树不存在” |  |
| HasPathSum | 显示“二叉树不存在” |  |
| maxPathSum | 显示“二叉树不存在” |  |

以下函数的操作与结果验证都在构建树的条件下完成

1)CreateBiTree测试

由于是按前序遍历的方式建树，用例的正确性用中序遍历和后序遍历的结果来验证是否正确；

|  |  |
| --- | --- |
| 测试用例 | 输入 |
| 1 | 1 a 2 b 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 2 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 3 e 0 null 0 null -1 null |
| 3 | 101 陈道明 102 王刚 103 张嘉译 0 null 0 null 104 本山 0 null 0 null 105 冰冰 106 KimJisoo 0 null 0 null 107 德刚 0 null 0 null -1 null |
| 4 | 1 a 2 b 0 null 3 c 4 d 0 null 5 e 0 null 0 null 0 null 0 null -1 null |

结果验证：

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 中序：2,b 1,a 4,d 3,c 5,e  后序：2,b 4,d 5,e 3,c 1,a |  |
| 2 | 关键字不唯一 |  |
| 3 | 中序：103,张嘉译 102,王刚 104,本山 101,陈道明 106,KimJisoo 105,冰冰 107,德刚  后序：103,张嘉译 104,本山 102,王刚 106,KimJisoo 107,德刚 105,冰冰 101,陈道明 |  |
| 4 | 中序：2,b 4,d 5,e 3,c 1,a  后序：5,e 4,d 3,c 2,b 1,a |  |

2）ClearBiTree测试

通过将已建好的树清空后判断是否能执行其他函数的方式验证是否成功清空为空树

|  |  |  |
| --- | --- | --- |
| 测试用例 | 初始树 | 清空后执行的函数名 |
| 1 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null | BiTreeDepth |
| 2 | 1 a 2 b 0 null 3 c 4 d 0 null 5 e 0 null 0 null 0 null 0 null | LevelOrderTraverse |
| 3 | 9 JISOO 0 null 18 LISA 0 null 27 ROSE 0 null 36 JENNIE 0 null 0 null | IsBalanced |

结果验证

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 二叉树清空成功  执行函数时显示二叉树不存在 |  |
| 2 | 二叉树清空成功  执行函数时显示二叉树不存在 |  |
| 3 | 二叉树清空成功  执行函数时显示二叉树不存在 |  |

3)BiTreeDepth

|  |  |
| --- | --- |
| 测试用例 | 构建树 |
| 1 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 2 | 1 abaaba 0 null 0 null -1 null |
| 3 | 1 a 2 b 4 d 8 h 0 null 0 null 0 null 5 e 0 null 0 null 3 c 6 f 0 null 0 null 7 g 0 null 0 null -1 null |
| 4 | 101 p 111 q 131 w 0 null 0 null 0 null 121 r 0 null 141 x  0 null 0 null -1 null |

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 树的深度为3 |  |
| 2 | 树的深度为1 |  |
| 3 | 树的深度为4 |  |
| 4 | 树的深度为3 |  |

4）LocateNode

|  |  |  |
| --- | --- | --- |
| 测试用例 | 构建树 | 查找结点 |
| 1 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null | 3 |
| 2 | 101 陈道明 102 王刚 103 张嘉译 0 null 0 null 104 本山 0 null 0 null 105 冰冰 106 KimJisoo 0 null 0 null 107 德刚 0 null 0 null -1 null | 106 |
| 3 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null | 20 |
| 4 | 5 m 10 n 20 x 0 null 0 null 25 y 0 null 0 null 15 p 0 null 0 null -1 null | 5 |

测试结果

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 3,c |  |
| 2 | 106,KimJisoo |  |
| 3 | 树中无此元素 |  |
| 4 | 5,m |  |

5)Assign

赋值函数在一棵树上进行，构建树的语句为:

1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null

|  |  |  |
| --- | --- | --- |
| 测试用例 | 结点关键字 | 用于替换的数据域 |
| 1 | 10 | Key:100 others:newnode |
| 2 | 3 | Key:20 others:金智秀 |
| 3 | 5 | Key:20 others:张蒋航帆 |
| 4 | 1 | Key:100 others:XXOO |

测试结果

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 无此节点 |  |
| 2 | 赋值成功 |  |
| 3 | 赋值后出现关键字重复 |  |
| 4 | 赋值成功 |  |

6）GetSibling

得到兄弟结点在同一棵树上进行操作：

构建语句为：5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 0 null 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 查找结点关键字 | 理论结果 | 操作结果 |
| 1 | 5 | 无兄弟结点 |  |
| 2 | 25 | 20，n |  |
| 3 | 10 | 15，m |  |
| 4 | 60 | 无兄弟结点 |  |

7）InsertNode

以一棵树为基准在其上进行插入结点的操作

构建树的语句为：5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 0 null 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 结点关键字 | 插入方式LR | 新结点数据域 |
| 1 | 3 | -1 | 50 f |
| 2 | 3 | 0 | 40 L |
| 3 | 15 | 1 | 25 M |
| 4 | 5 | 1 | 55 GG |
| 5 | 10 | 0 | 80 QAQ |

测试结果

用插入后树的前序遍历来检查正确性

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 插入操作成功! |  |
| 2 | 由于查找的结点关键字不存在，应该是插入操作失败 |  |
| 3 | 由于插入后出现关键字不唯一，应该是插入操作失败 |  |
| 4 | 插入操作成功！ |  |
| 5 | 插入操作成功！ |  |

8）DeleteNode

以一棵树为基准在其上进行删除结点的操作

构建树的语句为：5 p 10 q 20 n 30 y 0 null 0 null 0 null 25 x 0 null 0 null 15 m 0 null 0 null -1 null

通过遍历判断操作的正确性

|  |  |  |
| --- | --- | --- |
| 测试用例 | 待删结点关键字 | 理论结果 |
| 1 | 12 | 结点删除失败 |
| 2 | 5 | 前序：10,q 20,n 30,y 25,x 15,m  后序：30,y 20,n 15,m 25,x 10,q |
| 3 | 20 | 前序：10,q 30,y 25,x 15,m  后序：30,y 15,m 25,x 10,q |
| 4 | 25 | 前序：10,q 30,y 15,m  后序：30,y 15,m 10,q |

测试结果

|  |  |
| --- | --- |
| 测试用例 | 操作结果 |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |

9）10）11）对前序遍历，中序遍历，后序遍历的检测在前多次作为检测手段出现，后续也会继续出现，不再单独验证；

12）LevelOrderTraverse

|  |  |
| --- | --- |
| 测试用例 | 建树 |
| 1 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 2 | 5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 0 null 0 null -1 null |
| 3 | 1 abaaba 0 null 0 null -1 null |

测试结果

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 1,a 2,b 3,c 4,d 5,e |  |
| 2 | 5,p 10,q 15,m 20,n 25,x 30,z |  |
| 3 | 1,abaaba |  |

13）SaveBiTree

|  |  |  |
| --- | --- | --- |
| 测试用例 | 选择树，命名文件 | 构建树 |
| 1 |  | 5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 0 null 0 null -1 null |
| 2 |  | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |

测试结果

|  |  |
| --- | --- |
| 测试用例 | 操作结果 |
| 1 |  |
| 2 |  |

14）LoadBiTree

先将树清空，然后在空树上加载在验证13时生成的文件来生成二叉树

|  |  |  |
| --- | --- | --- |
| 测试用例 | 文件名 | 操作结果 |
| 1 | aa |  |
| 2 | lol |  |

15）IsBalanced

|  |  |
| --- | --- |
| 测试用例 | 构建树 |
| 1 | 1 GG 0 null 0 nul -1 null |
| 2 | 1 a 0 null 2 b 0 null 3 c 0 null 0 null -1 null |
| 3 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 4 | 2 k 3 m 4 n 5 p 0 null 0 null 0 null 0 null 6 z 0 null 0 null -1 null |

测试结果

|  |  |  |
| --- | --- | --- |
| 测试用例 | 理论结果 | 操作结果 |
| 1 | 它是一棵二叉平衡树 |  |
| 2 | 它不是一棵二叉平衡树 |  |
| 3 | 它是一棵二叉平衡树 |  |
| 4 | 它不是一棵二叉平衡树 |  |

16) CommonAncestor

在一棵已构建的树上进行查找检测

构建树的语句为：2 k 3 m 4 n 5 p 0 null 0 null 0 null 8 f 0 null 0 null 6 z 0 null 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 两个关键字 | 理论结果 | 操作结果 |
| 1 | 3 9 | 关键字有不存在的结点 |  |
| 2 | 3 5 | 3 m |  |
| 3 | 5 6 | 2 k |  |
| 4 | 5 8 | 3 m |  |

17)IsCousins

在一棵已构建的树上进行检测

构建树的语句为5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 35 k 0 null 0 null 55 g 0 nul 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | 两个关键字 | 理论结果 | 操作结果 |
| 1 | 20 40 | 有关键字不在树中 |  |
| 2 | 20 35 | 是堂兄弟 |  |
| 3 | 10 15 | 不是堂兄弟 |  |
| 4 | 20 30 | 不是堂兄弟 |  |

18）DiameterOfTree

|  |  |
| --- | --- |
| 用例 | 构建树 |
| 1 | 1 a 0 null 0 null -1 null |
| 2 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 3 | 1 a 2 b 4 d 8 h 0 null 0 null 0 null 5 e 0 null 0 null 3 c 6 f 0 null 0 null 7 g 0 null 0 null -1 null |

操作结果

|  |  |  |
| --- | --- | --- |
| 用例 | 理论结果 | 操作结果 |
| 1 | 0 |  |
| 2 | 3 |  |
| 3 | 5 |  |

19）InvertTree

反转二叉树通过前序遍历和中序遍历来验证正确性

|  |  |
| --- | --- |
| 用例 | 构建树 |
| 1 | 1 a 2 b 3 c 0 null 4 d 0 null 0 null 0 null 0 null -1 null |
| 2 | 1 啊啊啊 0 null 0 null -1 null |
| 3 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |

操作结果

|  |  |  |
| --- | --- | --- |
| 用例 | 理论结果 | 操作结果 |
| 1 | 前序：1,a 2,b 3,c 4,d  后序：1,a 2,b 4,d 3,c |  |
| 2 | 前序：1 啊啊啊  中序：1 啊啊啊 |  |
| 3 | 前序：1,a 3,c 5,e 4,d 2,b  中序：5,e 3,c 4,d 1,a 2,b |  |

20）HasPathSum

在一棵已构建的树上进行操作

构建树的语句为: 5 p 10 q 20 n 0 null 0 null 25 x 0 null 30 z 0 null 0 null 15 m 0 null 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 用例 | Sum | 理论结果 | 操作结果 |
| 1 | 70 | 存在这条路径 |  |
| 2 | 5 | 不存在路径 |  |
| 3 | 35 | 存在这条路径 |  |
| 4 | 100 | 不存在路径 |  |

21）maxPathSum

|  |  |
| --- | --- |
| 用例 | 构建树 |
| 1 | 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null |
| 2 | 101 陈道明 102 王刚 103 张嘉译 0 null 0 null 104 赵本山 0 null 0 null 105 李冰冰 106 牛莉 0 null 0 null 107 郭德纲 0 null 0 null -1 null |
| 3 | 1 a 2 b 0 null 3 c 4 d 0 null 5 e 0 null 0 null 0 null 0 null -1 null |

|  |  |  |
| --- | --- | --- |
| 用例 | 理论结果 | 操作结果 |
| 1 | 3+4+5=12 |  |
| 2 | 104+102+101+105+107=519 |  |
| 3 | 1+2+3+4+5=15 |  |

**3.4 实验小结**

我在本次实验上所花的时间比前两个实验的时间加起来还多，由于以前经常刷LeetCode,深知企业面试对树结构的重视，因此在课堂学习和实验课程的操作都投入了更多的时间，与预想的一样，这次实验的问题比前两个实验多的多，而不是像前两个实验一样将EduCoder上的函数进行粘贴就直接能够顺利运行了。主要问题来自几个方面，一个是在这次实验中大量使用了递归函数，但是我发现事实上有的地方递归出口是不正确的，例如Assign函数中我就进行了多次修改，起初我在操作时发现针对赋值函数命令，程序确实正确地执行了，在遍历操作中能看到赋值后的结点信息，但是却会打印操作失败的信息，原因是在复辅助函数Ass中习惯性地在递归地在左右子树中去查找赋值结点后给Assign函数返回了FALSE，当时的想法是遍历结束未返回OK则说明无此结点，事实上这个函数完全不需要返回值，只需要在check2函数中提前分析好赋值操作不能执行的情况，先判断能否赋值再执行Ass即可，这样一定能够完成赋值操作。而在不同错误情况下打印不同信息也更加严谨，也是在发现错误时顺带完善的点，而不是一律返回ERROR，因为在真正的系统中，需要根据错误的不同来思考不同的解决办法，关键字重复与目标结点不存在是完全不一样的。

随后在附加功能上是针对以前在LeetCode上刷题的经验，写了几个经典功能，在完成附加功能时发现了很多巧妙的用递归方法完成的思路，最后也都得到了正确的验证，翻转二叉树更是因Google因此题拒收大牛Howell而出名，所以这次实验虽然耗时非常久，但是我认为非常值得，因为熟练掌握树的结构与操作是十分重要的！