# 3基于二叉链表的二叉树实现

## 3.1 实验目的

二叉树是一种树型结构，二叉树的每个结点至多只有两棵子树。二叉树的二叉链表表示指的是数据元素在物理位置上不相邻的二叉树，而是通过附加指针域的方法将各个数据元素联系起来，使各个数据元素之间具有一定的逻辑关系。本实验要求封装一个基于二叉链表结构的二叉树ADT模块，提供二叉树ADT基本的、常见的20种操作，并且为该二叉树ADT提供文件的I/O方法。

## 3.2 模块设计

### 3.2.1 总体架构

根据实验的要求，给出整个模块的总体架构：

模块包括一个二叉树实现文件(.cpp)

模块还包括了一个栈ADT模块和一个队列ADT模块，以实现非递归遍历二叉树和层序遍历二叉树，同时在其他功能上也有对栈ADT模块的引用

模块的结构定义、结构声明、函数声明、函数指针以及相应的别名typedef均在实现文件的开头，函数实现在实现文件的尾部

模块的ADT操作实现被包括在实现文件中部

### 3.2.2 数据结构设计

数据结构的定义

根据实验的要求以及二叉树ADT的定义，给出基于C++语言实现的二叉树ADT的结构定义和别名：

typedef struct RooTNode{

BiTree child;

}RooTNode,\* RooTree;

其中，child指向该二叉树的根结点。同时为了防止指针写法的混乱，将RooTNode \* 定义为RooTree

二叉树ADT的结点BiTNode的定义和别名如下：

typedef struct BiTNode{

ElemType \*elem;

struct BiTNode \* lchild;

struct BiTNode \* rchild;

} BiTNode, \* BiTree;

其中，elem是ElemType类型的指针，用于指向学生的数据，struct BiTNode \* lchild是指向左孩子的指针， struct BiTNode \* rchild是指向右孩子的指针，同时为了防止指针写法的混乱，将BiTNode\*定义为BiTree。

在本次的实验中，采用了如下的结构作为数据元素：

typedef struct{

char id[EVERY];

char name[EVERY];

char phone[EVERY];

} ElemType;

该结构表示一个用户通讯录，其中，id表示的是学生的学号。Name表示的是学生的姓名，phone表示的是学生的联系方式。

### 3.2.3 ADT操作设计

根据实验的要求以及二叉树ADT的定义，该二叉树ADT应包括如下的操作：

InitTree（初始化），DestroyTree（销毁），CreateTree（构造），ClearTree（清空），TreeEmpty（判断树是否为空），TreeDepth（求树的深度），TreeRoot（取得根节点），Value（取得结点的值），Assign（为结点赋值），Parent（取得结点的父结点），LeftChild（取得结点的左孩子），RighitChild（取得结点的右孩子），LeftSibling（取得结点的左兄弟），RightSibling （取得结点的右兄弟），InsertChild（插入子树），DeleteChild（删除子树），PreOrderTraverse（前序遍历），InOrderTraverse（中序遍历），PostOrderTraverse（后序遍历）。

根据相关ADT操作的特性，给出如下基于C++语言的相关定义。

为了ADT操作统一性以及错误处理，在C++语言中给出如下的通用函数原型：

Status MethodName (Typename1 arg1, Typename2 arg2, ...);

其中，Status表示为预定义的枚举类型，用来表明函数执行的结果以及错误的处理，在C++语言中的定义如下：

#define BEF 0 //先序序列输入

#define MID 1 //中序序列输入

#define AFT 2 //后序序列输入

#define FALSE 0

#define OK 1

#define ERROR 0

#define EVERY 20

#define STACK\_INIT\_SIZE 100

#define STACKINCREMENT 10

#define MAXQSIZE 100

#define NOTINIT 0

#define INIT 1

#define CREATE 2

#define ABANDON 3

BEF表示用户选择先序输入，MID表示用户选择中序输入，AFT表示用户选择后序输入，FALSE表示函数执行失败，ERROR表示函数异常退出，EVERY表示id、name和phone最大长度，STACK\_INIT\_SIZE表示为栈分配的空间，STACKINCREMENT表示栈空间不够时重新分配增加的空间，MAXQSIZE表示队列分配的空间，NOTINIT表示树未初始化，INIT表示树已初始化，CREATE表示树已创建，ABANDON表示树已被舍弃。

为了模块的拓展性，可能会用到访问元素并打印全部信息和访问函数只打印id所用到的函数PrintElem、VisitElem，以及它们所用到的访问函数Visit，在C++语言中定义如下：

Status PrintElem(ElemType \*e);

Status VisitElem(ElemType \*e);

基于以上的定义，给出所有ADT操作在C++语言中对应的函数原型：

Status InitBiTree(RooTree \*T);

Status DestroyBiTree(RooTree \*\*T,int moro,int \*where,int \*Isinit,int \*\*how,RooTree \*\*TS,int \*choice,int number);

Status CreateBiTree(BiTree \*T,int definition);

Status ClearBiTree (BiTree \*T);

Status BiTreeEmpty(BiTree T);

int BiTreeDepth(BiTree T);

Status Root(BiTree T);

Status Value(BiTree T,BiTNode e);

Status Assign(BiTree \*T,BiTNode &e,ElemType value);

Status Parent(BiTree T,BiTree e);

Status LeftChild(BiTree T,BiTree e);

Status RightChild(BiTree T,BiTree e);

Status LeftSibling(BiTree T,BiTree e);

Status RightSibling(BiTree T,BiTree e);

Status InsertChildBiTNode (BiTree \*T,BiTree p,int LR);

Status DeleteChild(BiTree \*T,BiTree p,int LR);

Status PreOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e));

Status InOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e));

Status PostOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e));

Status LevelOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e));

### 3.2.4 文件存储设计

根据实验的要求以及二叉树ADT的数据结构，该二叉树ADT应包括如下的文件操作：

WriteToFile(将二叉树写入文件)、ReadFromFile(从文件中读取二叉树)

根据ADT操作设计中的定义，给出文件存储操作在C语言中对应的函数原型：

Status Write(BiTree T,ofstream &outfile);

Status WriteToFile(BiTree T);

Status Read(BiTree \*T,ifstream &infile);

Status ReadFromFile(BiTree \*T);

### 3.2.5 配套演示系统设计

配套演示系统采用文本菜单界面，该系统每次从输入流接受一个数字，并且根据数字的不同执行不同的操作。

## 3.3 模块实现

### 3.3.1 开发环境

开发本系统时的操作环境为Windows 10

编程环境

IDE：Clion

Toolchains：

Environment：MinGW version：5.0

C Compiler：mingw32-gcc

C++ Compiler：mingw32-g++

### 3.3.2 ADT操作实现

InitTree(RooTree \*T)

该操作接受一个RooTree \*指针

该操作通过动态分配空间的方式分配一个RooTNode结构变量，将其指针存入\*T中，设计该二叉树的根结点为空

该二叉树初始化为空树

该操作的时间复杂度O(1)，空间复杂度O(1)。

DestroyBiTree(RooTree \*\*T,int moro,int \*where,int \*Isinit,int \*\*how,RooTree \*\*TS,int \*choice,int number)

该操作接受一个RooTree \*\*指针、一个int moro、一个int\* where、一个int\* Isinit、一个int \*\* how、一个RooTree \*\*指针、一个int\* choice和一个int number

该操作首先检查根结点是否存在，若不存在，则会告知用户该树未创建；否则，首先询问用户是否保存该树于文件中，然后执行用户相应的选择。最后调用递归函数ClearBiTree清空树。若该树为单树，销毁树；若该树为多树中的单树，则删除该单树。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

CreateBiTree(BiTree \*T,int definition)

该操作接受一个BiTree\*指针和一个int definition

该操作首先判断用户选择的输入方式，然后根据用户的选择使用递归的方式依次从控制台读取用户输入的信息，存储到树中

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

ClearBiTree (BiTree \*T)

该操作接受一个BiTree\*指针

该操作和DestroyTree操作相似，不同之处是该操作只是简单的清空树且释放空间，并没有其他的功能

该操作使用递归的方式来释放空间

该操作的时间复杂度为O(n)，空间复杂度为O(logn)

BiTreeEmpty(BiTree T)

该操作接受一个BiTree指针

该操作通过判断根结点是否为空来判断该树是否为空树，若根结点为空，则告知用户该树为空树；否则，告知用户该树不为空

该操作的时间复杂度为O(1),空间复杂度为O(1)。

BiTreeDepth(BiTree T)

该操作接受一个BiTree指针，该函数的返回值为整型

该操作使用递归的方式计算二叉树的深度，并且将二叉树的深度作为返回值返回。递归程序会判断自己的左孩子和右孩子是否非空，对于非空的孩子调用下一次递归计算深度，并且返回更大的返回值+1

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

Root(BiTree T)

该操作接受一个BiTree指针

该操作首先判断根结点是否为空，若为空，则告知用户该树为空树；否则，打印根结点所包含的信息

该操作的时间复杂度为O(1)，空间复杂度为O(1)。

Value(BiTree T,BiTNode e)

该操作接受一个BiTree指针和一个BiTNode结构

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点其id与e中id相同，则打印该结点的所有信息；否则，告知用户结点不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

Assign(BiTree \*T,BiTNode &e,ElemType value)

该操作接受一个BiTree指针、一个BiTNode引用和一个ElemType结构

该操作通过调用栈的方式来非递归寻找与value中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点其id与value中id相同，则使得e指向该结点，打印该结点的所有信息，然后将value的值赋予给该结点，打印改变之后的结点信息；否则，告知用户结点不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

Parent(BiTree T,BiTree e)

该操作接受一个BiTree指针和一个BiTree指针

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先判断e中id是否与根结点中的id相同，若相同，则告知用户结点为根结点不存在双亲；否则，初始化栈，然后非递归的方式先序遍历各个结点，若存在结点的左孩子或者右孩子其id与e中id相同，则打印该结点左孩子或右孩子的所有信息；否则，告知用户结点不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

LeftChild(BiTree T,BiTree e)

该操作接受一个BiTree指针和一个BiTree指针

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点的左孩子其id与e中id相同，则打印该结点左孩子的所有信息；否则，告知用户不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

RightChild(BiTree T,BiTree e)

该操作接受一个BiTree指针和一个BiTree指针

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点的右孩子其id与e中id相同，则打印该结点右孩子的所有信息；否则，告知用户不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

LeftSibling(BiTree T,BiTree e)

该操作接受一个BiTree指针和一个BiTree指针

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点的右孩子其id与e中id相同，则判断该结点的左孩子是否存在，若左孩子存在，则打印左孩子结点的信息，否则告知用户所找结点的左兄弟不存在；若不存在结点的右孩子其id与e中id相同，告知用户不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

RightSibling(BiTree T,BiTree e)

该操作接受一个BiTree指针和一个BiTree指针

该操作通过调用栈的方式来非递归寻找与e中id相同的结点

该操作首先初始化栈，然后非递归的方式先序遍历各个结点，若存在结点的左孩子其id与e中id相同，则判断该结点的右孩子是否存在，若右孩子存在，则打印右孩子结点的信息，否则告知用户所找结点的右兄弟不存在；若不存在结点的右孩子其id与e中id相同，告知用户不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

InsertChildBiTNode (BiTree \*T,BiTree p,int LR)

该操作接受一个BiTree指针、一个BiTree指针和一个int LR

该操作通过调用栈的方式来非递归寻找与p中id相同的结点

该操作首先初始化栈，然后按照用户选择输入的方式来接受用户再次创建的树，然后非递归的方式遍历各个结点，若存在结点其id与p中的id相同，通过用户的选择来执行相应的插入操作；若不存在结点其id与p中的id相同，则告知用户所找结点不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

DeleteChild(BiTree \*T,BiTree p,int LR)

该操作接受一个BiTree指针、一个BiTree指针和一个int LR

该操作通过调用栈的方式来非递归寻找与p中id相同的结点

该操作首先初始化栈，然后以非递归的方式遍历各个结点，若存在结点其id与p中的id相同，通过用户的选择来执行相应的删除操作；若不存在结点其id与p中的id相同，则告知用户所找结点不存在，然后释放栈的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

PreOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e))

该操作接受一个BiTree指针和一个Visit函数指针，按照先序遍历当前二叉树

该操作首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则用递归方式先序遍历整棵二叉树。递归程序接收一个BiTree指针和一个Visit函数指针，对当前结点执行Visit函数，然后依次对于当前结点的左孩子和右孩子调用子递归。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

InOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e))

该操作接受一个BiTree指针和一个Visit函数指针，按照中序遍历当前二叉树

该操作首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则用递归方式中序遍历整棵二叉树。递归程序接收一个BiTree指针和一个Visit函数指针，对当前结点的左孩子调用子递归，执行Visit函数，然后对于当前结点的右孩子调用子递归。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

PostOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e))

该操作接受一个BiTree指针和一个Visit函数指针，按照后序遍历当前二叉树

该操作首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则用递归方式后序遍历整棵二叉树。递归程序接收一个BiTree指针和一个Visit函数指针，对当前结点的左孩子和右孩子调用子递归，最后对当前结点执行Visit函数。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

LevelOrderTraverse(BiTree T,Status(\* Visit)(ElemType \*e))

该操作接受一个BiTree指针和一个Visit函数指针，按照层序遍历当前二叉树

该操作首先判断二叉树是否为空。若不为空，则执行一下操作：

1.初始化一个队列

2.将根结点入队

3.判断队列是否为空，若队列不为空；否则，循环结束。将队列中第一个结点出队，然后打印该结点的信息。判断出队结点的左孩子是否存在，若存在，则将左孩子入队。再判断出队结点的右孩子是否存在，若存在，则将右孩子入队。

最后释放队列的空间

该操作的时间复杂度为O(n)，空间复杂度为O(n)。

先序遍历树的非递归方式：

传入BiTree指针T

1. 访问结点T，并将其入栈

2. 判断T的左孩子是否为NULL，如果为NULL，则访问栈顶结点并且出栈，并且将其右结点设为T，执行1。如果不为NULL，则将T的左孩子设为T。

3. 若T为NULL且栈为空，遍历结束。

### 3.3.3 文件存储实现

Write(BiTree T,ofstream &outfile)

该操作通过递归的方式先序将树信息打印到文件中

1.接受一个BiTree指针T和一个输出流引用outfile。

2.向文件中输入内容时，首先判断该结点是否为空，若该结点为空，则向文件中打印字符串“#“。若该结点不为空，则打印该结点信息，打印id、name和phone时，每打印一次就换行继续打印。然后按照递归的方式先序打印该树的所有结点信息

3.当前树存储完成

该操作的时间复杂度是O(n)，空间复杂度是O(logn)。

WriteToFile(BiTree T)

接受用户输入所创建文件的名称，并且判断文件是否打开成功。若文件打开成功，则执行Write函数，否则，告知用户打开文件失败。

Read(BiTree \*T,ifstream &infile)

该操作通过递归的方式先序从文件中读取信息

1.接受一个BiTree\*指针T和一个输入流引用infile。

2.读取文件内容时，首先读取第一个字符串，若字符串为“#“，则使得当前结点\*T为空(nullptr)。若第一个字符串不为”#“，则动态分配一个结点的空间，将刚刚读取的信息和下面两个字符串依次赋给结点的id、name和phone。然后按照递归方式将文件内容先序构建树。

3.树构建完成

该操作的时间复杂度是O(n)，空间复杂度是O(logn)。

ReadFromFile(BiTree \*T)

接受用户输入所读取文件的名称，并且判断文件是否打开成功。若文件打开成功，则执行Read函数，否则，告知用户打开文件失败。

### 3.3.4 配套演示系统实现

外部函数实现

visit：

该操作接收一个ElemType一级指针，指向特定类型的数据元素，将其输出。

情况1：在该实验的演示系统中，该操作将ElemType数据结构的三个成员变量id、name和phone的值依次打印输出。

情况2：在该实验的演示系统中，该操作将ElemType数据结构的三个成员变量id的值打印输出。

该系统采用简单文本界面，实现的过程相对而言简单——首先输出系统菜单，然后从输入流中读入一个数字，在根据数字所对应的功能执行相应的代码。

## 3.4 模块测试

### 3.4.1 配套演示系统操作介绍

演示系统的界面如下图所示：



该演示系统操作简单，输入对应的数字并按下回车即可，输入1--25表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入0退出系统。

### 3.4.2 测试方案

根据程序要实现的功能，拟定了以下一个测试方案：

1. 初始化一个二叉树。
2. 构建一棵有11个元素的满二叉树。
3. 测试该树是否为空以及该树的深度。
4. 先序以及后序遍历该二叉树。
5. 将该二叉树写入文件。
6. 执行多树功能并创建两个二叉树
7. 创建第一个二叉树，二叉树的元素和结构与单树相同
8. 切换到第二个二叉树，创建树使树为有3个元素的满二叉树
9. 切换到第一个二叉树
10. 打印根结点的值。
11. 将第二课树插入进第一棵树的右子树。
12. 测试该树的深度。
13. 层序遍历该二叉树。
14. 销毁第一棵二叉树。
15. 切换到第二个二叉树
16. 从文件中加载单树于该树中。
17. 中序遍历该二叉树。
18. 打印该树树形
19. 退出程序

### 3.4.3 测试数据

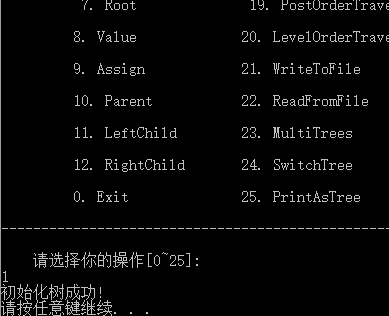
单树：A a 1 B b 2 D d 3 G g 4 # # H h 5 # # E e 6 # # C c 7 K k 8 # # F f 9 I i 10 # J j 11 # # #

多树中的第二颗树：

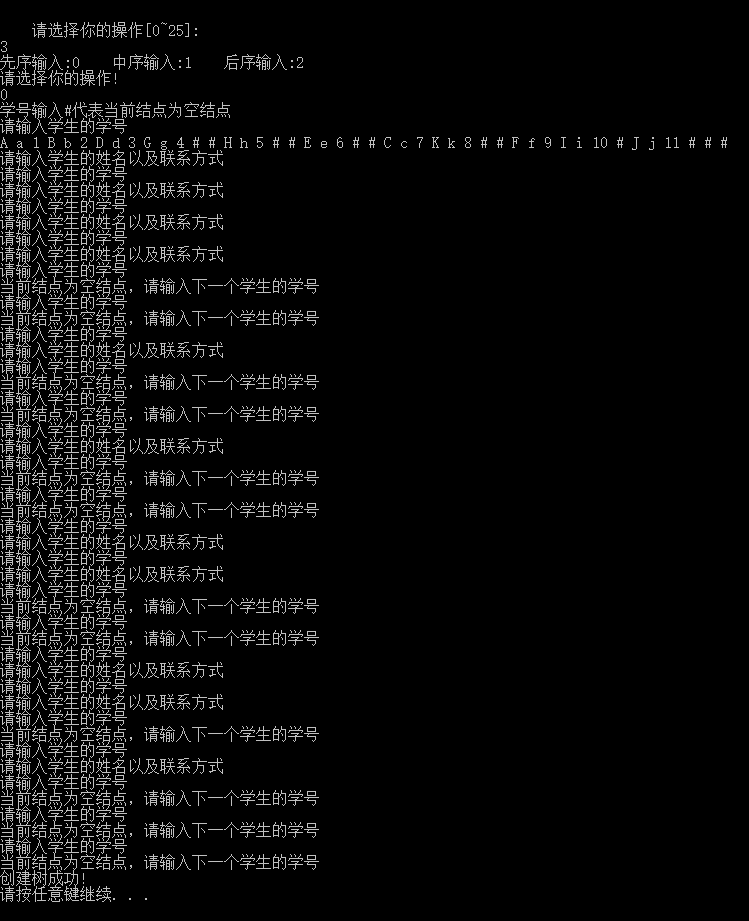
O o 12 P p 13 # # Q q 14 # #

### 3.4.4 测试结果

1. 初始化一个二叉树。

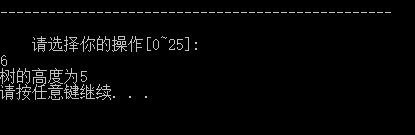


1. 构建一棵有11个元素的满二叉树。

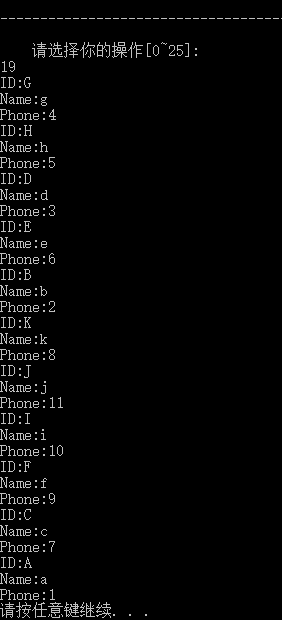
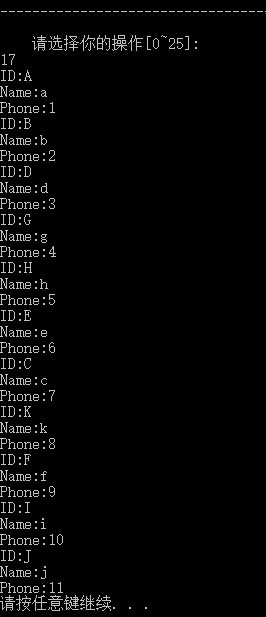


1. 测试该树是否为空以及该树的深度。

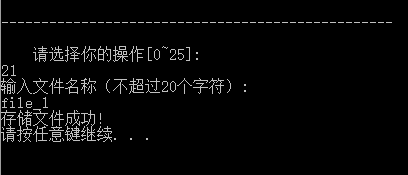




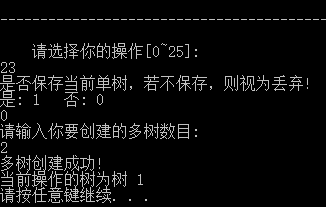
1. 先序以及后序遍历该二叉树。



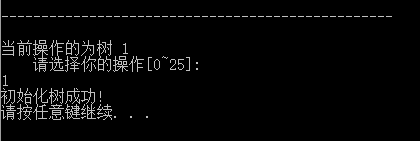
1. 将该二叉树写入文件。

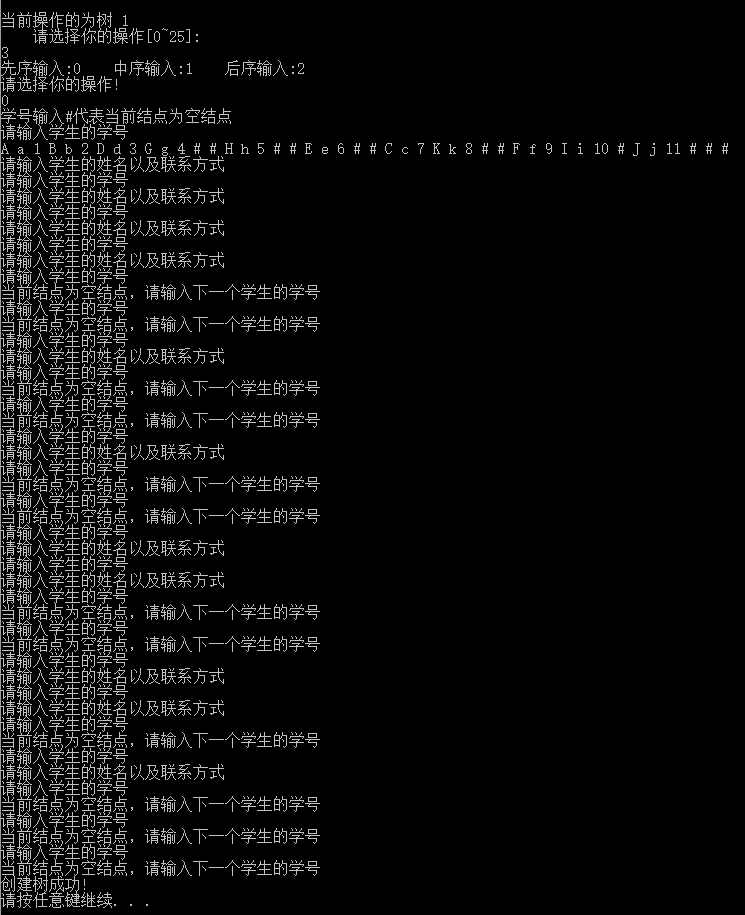


1. 执行多树功能并创建两个二叉树



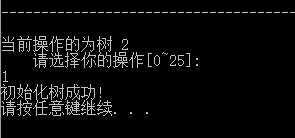
1. 创建第一个二叉树，二叉树的元素和结构与单树相同

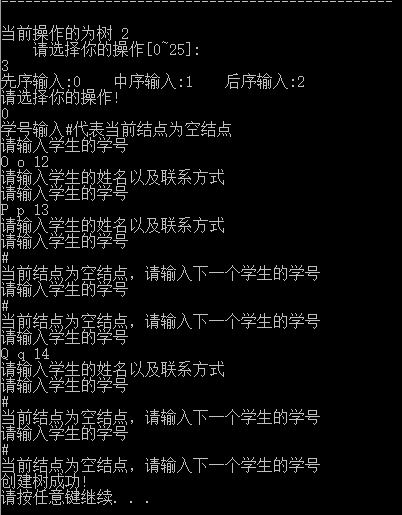




1. 切换到第二个二叉树，创建树使树为有3个元素的满二叉树



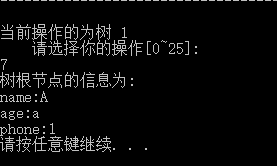




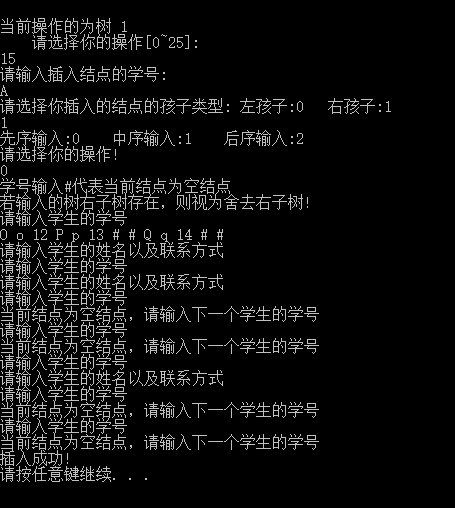
1. 切换到第一个二叉树



1. 打印根结点的值。

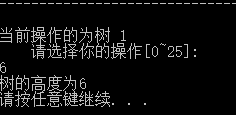


1. 将第二颗树的元素插入进第一棵树的右子树。

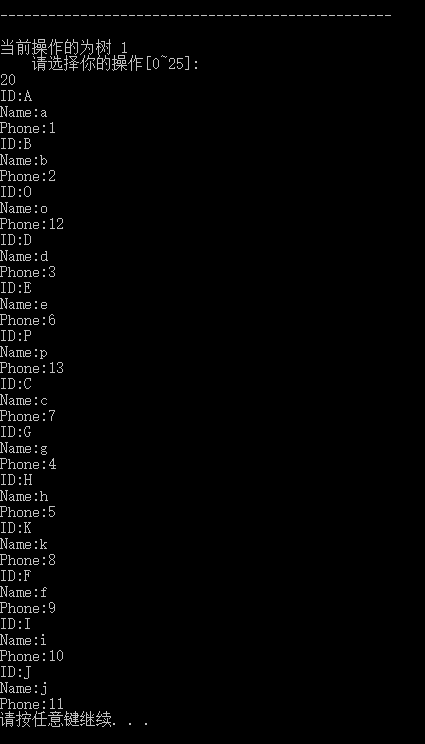


1. 测试该树的深度。

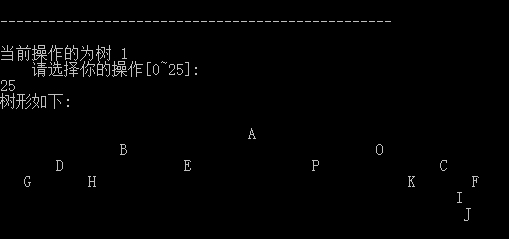
此时我们插入的树存在右子树，所以舍弃了插入树的右子树



1. 层序遍历该二叉树。



1. 以树形打印第一颗二叉树



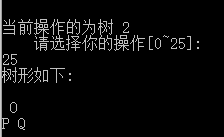
1. 销毁第一个二叉树



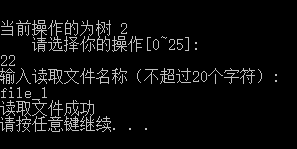
1. 自动切换到第二个二叉树



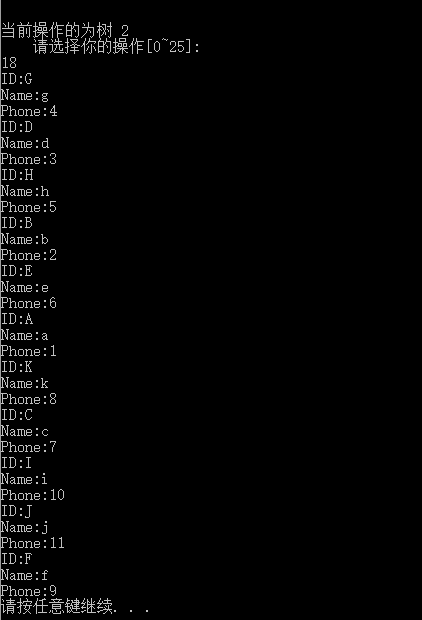
1. 树形打印该树



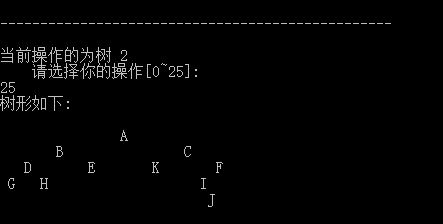
1. 从文件中加载单树于该树中。



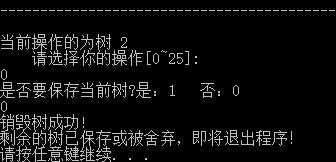
1. 中序遍历该二叉树。



1. 打印该树树形



1. 退出程序



## 3.5 实验总结

通过本次实验，我加深了对于基于二叉链表的二叉树的理解。

在本次实验中，我对递归有了更深层次的了解。实验中，有一个比较意外的收获是在清空树的时候，我重复释放内存空间。在不同的编译器产生了不同的反应，也因此让我对递归和编译器有了更深的理解。

此外，在这次的实验中，我多次使用栈进行遍历，对非递归遍历二叉树也有了更好的理解，本次实验加深了我对于二叉树操作的理解，也使得我的二叉树更加的全面和完整。