作业 HW3 实验报告

姓名：刘博洋 学号：2153538 日期：2022 年 11 月 5 日

1. 涉及数据结构和相关背景

树型结构是一种非常重要的非线性数据结构，其中树和二叉树是最为常用的数据结构。树和二叉树采用分支关系来定义层次，可以较为清晰的表示事物的组成结构或者逻辑结构，也是信息的重要组成形式之一，许多广泛被人们使用的算法例如决策树、哈夫曼树等等都是建立在基本的树和二叉树结构之上，MySQL 数据库生成索引的数据结构，也是应用了排序二叉树。对树进行分类，主要可以分为有序树和无序树，常见的有序树主要有：完全二叉树、满二叉树、二插搜素树、平衡二叉树等等。

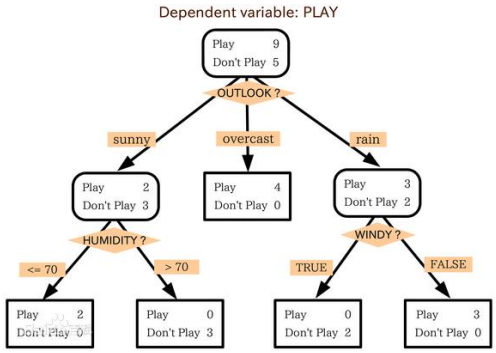


图 1 一种决策树

2. 实验内容

2.1 二叉树的同构

2.1.1 问题描述

给定两棵树 T1 和 T2。如果 T1 可以通过若干次左右孩子互换变成 T2，则我们称两棵树是“同构”的。例如图 1 给出的两棵树就是同构的，因为我们把其中一棵树的结点 A、B、G 的左右孩子互换后，就得到另外一棵树。而图 2 就不是同构的。

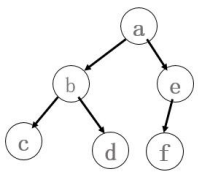
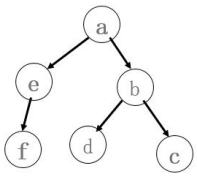
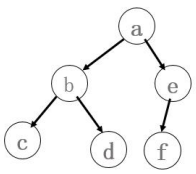


图 2





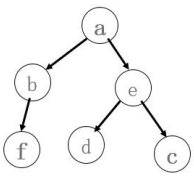


图 3

现给定两棵树，请你判断它们是否是同构的。并计算每棵树的深度。

2.1.2 基本要求

输入:

第一行是一个非负整数 N1，表示第 1 棵树的结点数；

随后 N 行，依次对应二叉树的 N 个结点（假设结点从 0 到 N-1 编号），每行有三项，分别是 1 个英文大写字母、其左孩子结点的编号、右孩子结点的编号。如果孩子结点为空，则在相应位置上给出“-”。给出的数据间用一个空格分隔。

接着一行是一个非负整数 N2，表示第 2 棵树的结点数；

随后 N 行同上描述一样，依次对应二叉树的 N 个结点。

对于 20%的数据，有 0<N1=N2<=10
对于 40%的数据，有 0<=N1=N2<=100
对于 100%的数据，有 0<=N1,N2<=10100
注意：题目不保证每个结点中存储的字母是不同的。
输出:
共三行。
第一行，如果两棵树是同构的，输出“Yes”，否则输出“No”。
后面两行分别是两棵树的深度。

2.1.3 数据结构设计

解决本题总体上要解决两个子问题：一是判断是否同构，二是计算二叉树的深度。
首先要构造两颗二叉树，根据题目所给输入信息的特点，采用顺序表进行存储，在顺序表每一个节点构造三个数据域，data 存放节点元素，lt 存放左孩子的数组下标，rt 存放右孩子的数组下标，如果不存在则对其赋值为-1。
对于计算深度问题可以编写递归函数，分别对左子树和右子树进行计算深度，最后取两者的较大值进行计算。由于递归函数可读性不好，在编写的时候给我造成了不少困惑，这里举出例子分析递归的执行过程。

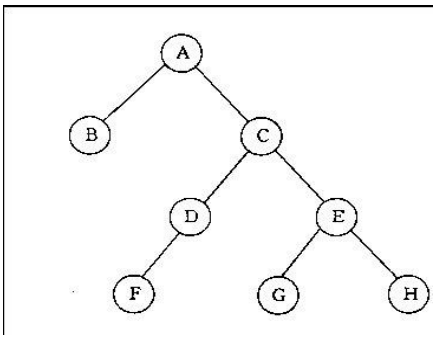


图 4

Depth（T） ——》进入函数 ——》访问 A 0
int a = Depth(T->lchild)；1 ——》访问 B 1
int a = Depth(T->lchild)；0 ——》访问 B 的左空节点 2
int b = Depth(T->rchild)；0 ——》访问 B 的右空节点 3
此时标号 2,3 函数完成，得到 a=0，由步骤 2,3 得到步骤 1 函数的 a 值为 1，再由步骤 1 得到步骤 0 对应的返回值为 2，此时计算到树的高度为 2，这只是根节点左部的子树高度，此时运行到刚开始进入函数内的第二个 GetTreeDeep，所以接下来该访问右部子树：
int b =Depth(T->rchild)；—— 》访问 C 4
int a = Depth(T->lchild)；2 ——》访问 D 5
int a = Depth(T->lchild)；1 ——》访问 F 6
int a = Depth(T->lchild)；0 ——》访问 F 的左空节点 7
int b = Depth(T->rchild)；0 ——》访问 F 的右空节点 8
步骤 7,8 的返回值为 0，由此得到步骤 6 的返回值为 1，步骤 4 对应的返回值为 2，接下来，运行到该访问 C 的右节点：
int b = Depth(T->rchild)；——》访问 E 9
int a = Depth(T->lchild)；1 ——》访问 G 10
int a = Depth(T->lchild)；0 ——》访问 G 的左空节点 11

int b =Depth(T->rchild)；0 ——》访问 G 的右空节点 12
以此类推：可知步骤 8 的返回值为 1，现在该访问 E 的右节点：
int b = Depth(T->rchild)；1 ——》访问 H 13
int a = Depth(T->lchild)；0 ——》访问 H 的左空节点 14
int b = Depth(T->rchild)；0 ——》访问 H 的右空节点 15
现在开始返回，比较各个节点的返回值孰大孰小
1， 首先比较的是步骤 13 和步骤 10 的返回值，二者一样大，返回 1+1，步骤 9 得返回值 2
2， 比较步骤 9 和 5，二者同样为 2，故步骤 4 的返回值为 2+1，为 3
3， 比较步骤 4 和步骤 1，前者为 3，后者为 1，取前者，所以最后返回 3+1，得步骤 0 的返回值为 4，即为最终结果。

其次，我们需要解决如何判断两颗树是否同构的问题。首先还是从两棵树的头结点开始比较，从根开始，对于每一个节点进行判断，有以下几种情况：
1. 两个节点都是空节点，则可以直接判断为两树同构。
2. 一个节点为空，一个节点不为空，显然不同构。
3. 两个节点都不为空，但是数据不相同，显然不同构。
4. 两个节点不为空，数据也相同，那么开始判断孩子节点
   a. 左子树和左子树同构且右子树和右子树同构，显然同构。
   b. 左子树和右子树同构且右子树和左子树同构，显然也同构
   c. 除了 a，b 两种情况之外的左右子树都不可能同构。

这也是一个递归程序，先看成整棵树，再分别判断左子树和右子树，左子树的左子树和右子树和右子树的左子树和右子树.....一直递归到遍历完整棵树，再开始回溯返回值。

2.1.4 功能说明（函数、类）

```
#define MaxTree 10100
struct TreeNode
{
    char data;
    int lt;
    int rt;
};//顺序表存储树的信息，三个数据域分别为节点数据，左子树下标和右子树下标
TreeNode* T1 = new TreeNode[MaxTree];
TreeNode* T2 = new TreeNode[MaxTree];
//建树
int BuildTree(struct TreeNode* T, int N)
{
    int i;
    string cl, cr;
    int check[10500];//判断是否为根节点的标记
    if (N)
    {
        for (i = 0; i < N; i++)
```

```
        {
            check[i] = 1;  //初始化
        }
        for (i = 0; i < N; i++)
        {
            cin >> T[i].data >> cl >> cr;
            if (cl != "-")
            {
                T[i].lt = atoi(cl.c_str());//读入字符串型，这里用 atoi 函数将
string 转化为 int 型
                check[T[i].lt] = -1;
            }
            Else//如果是“-”则表示子树为空，标记为-1
            {

                T[i].lt = -1;
            }

            if (cr != "-")
            {
                T[i].rt = atoi(cr.c_str());
                check[T[i].rt] = -1;
            }
            else
            {
                T[i].rt = -1;
            }
        }
        //判断是不是根节点，根节点没有双亲结点
        for (i = 0; i < N; i++)
        {
            if (check[i] == 1)
            {
                return i;  //返回根结点的编号
            }
        }
        return -1;
    }
    return -1;//如果整棵树为空，即没有根节点，那么返回-1
}
int depth1(struct TreeNode  T)
{
    int m, n;
    if (T.lt != -1 && T.rt == -1)
```

```
    {
        m = depth1(T1[T.lt]);
        return m + 1;
    }
    if (T.lt == -1 && T.rt != -1)
    {
        n = depth1(T1[T.rt]);
        return n + 1;
    }
    if (T.lt == -1 && T.rt == -1)
        return 1;
    m = depth1(T1[T.lt]); //递归计算左子树的深度记为 m
    n = depth1(T1[T.rt]); //递归计算右子树的深度记为 n
    if (m > n)
        return (m + 1); //二叉树的深度为 m 与 n 的较大者加 1
    else
        return (n + 1);

}
int depth2(struct TreeNode  T)
{
    int m, n;
    if (T.lt != -1 && T.rt == -1)
    {
        m = depth2(T2[T.lt]);
        return m + 1;
    }
    if (T.lt == -1 && T.rt != -1)
    {
        n = depth2(T2[T.rt]);
        return n + 1;
    }
    if (T.lt == -1 && T.rt == -1)
        return 1;
    m = depth2(T2[T.lt]); //递归计算左子树的深度记为 m
    n = depth2(T2[T.rt]); //递归计算右子树的深度记为 n
    if (m > n)
        return (m + 1); //二叉树的深度为 m 与 n 的较大者加 1
    else
        return (n + 1);

}
int tonggou(int x, int y)
{
```

```
    if (x == -1 && y == -1)
    {//如果两个都恰好是空节点则其实是相同的
        return 1;
    }
    else if ((x != -1 && y == -1) || (x == -1 && y != -1))
    {//两边一个有子节点，一个没有子节点
        return 0;
    }
    else if (T1[x].data != T2[y].data)
    {//两个位置的数值不相等，则判定为直接为不一样的
        return 0;
    }
    else if (tonggou(T1[x].lt, T2[y].lt) && tonggou(T1[x].rt, T2[y].rt))
    {//如果左边的等于左边的且右边的等于右边的
        return 1;
    }
    else if (tonggou(T1[x].lt, T2[y].rt) && tonggou(T1[x].lt, T2[y].rt))
    {//左边的等于右边的，右边的等于左边的
        return 1;
    }
    else
    {
        return 0;
    }
}
int main()
{
    int R1, R2;
    int N1, N2;
    cin >> N1;
    R1 = BuildTree(T1, N1);
    cin >> N2;
    R2 = BuildTree(T2, N2);
    if (tonggou(R1, R2))
    {
        printf("Yes\n");
    }
    else
    {
        printf("No\n");
    }
    cout << depth1(T1[R1]) << endl << depth2(T2[R2]);
    return 0;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

本题主要是判断同构的程序编写十分困难，很容易漏掉情况导致判断错误。而递归程序的查找漏动和调试也比较困难，在多层递归的情况下，很难看出是否出现逻辑漏洞，兵器题目所给测试数据数量大，难以判断问题所在。刚开始编写这个递归程序的时候，我把两棵树割裂开来，导致不能对两颗树进行逐层对比判断，然后我又想对于每一个根节点，先判断左子树再判断右子树，但是这样会导致无法判断左右子树互换相等的情况。最后还是把整个问题细化成一个小问题，即判断一个节点和它的左子树右子树是否同构，然后逐层递归得到判断结果。

2.1.6 总结和体会

本题难点就在于判断同构的递归操作，通过编写这个递归程序我对于递归函数的理解有了不少加深。由于曾经编写的递归程序判断返回条件都比较单一，层数也不多，分支也不复杂，所以在遇到这种复杂的分支和递归条件时我应对不足，消耗了大量时间，实际上，对于递归问题，都要从小处着眼，把它转化为每一个子问题来解决，并且要注意解决的顺序和层次。

2.2 二叉树的非递归遍历

2.2.1 问题描述

二叉树的非递归遍历可通过栈来实现。例如对于由 abc##d##ef###先序建立的二叉树，如下图 1 所示，中序非递归遍历（参照课本 p131 算法 6.3）可以通过如下一系列栈的入栈出栈操作来完成：push(a) push(b) push(c) pop pop push(d)pop pop push(e) push(f) pop pop。

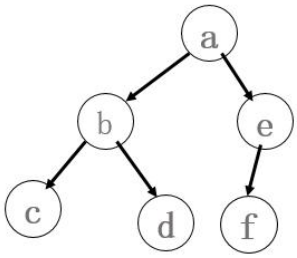


图 5

如果已知中序遍历的栈的操作序列，就可唯一地确定一棵二叉树。请编程输出该二叉树的后序遍历序列。

提示：本题有多种解法，仔细分析二叉树非递归遍历过程中栈的操作规律与遍历序列的关系，可将二叉树构造出来。

2.2.2 基本要求

输入：

第一行一个整数 n，表示二叉树的结点个数。

接下来 2n 行，每行描述一个栈操作，格式为：push X 表示将结点 X 压入栈中，pop 表示从栈中弹出一个结点。

(X 用一个字符表示)

对于 20%的数据，0<n<=10

对于 40%的数据，0<n<=20

对于 100%的数据，0<n<=83

输出：

一行，后序遍历序列。

2.2.3 数据结构设计

本题实际上要仔细分析非递归遍历时的操作序列和二叉树遍历的顺序之间的关系，便可以比较容易的得到结论。即按照先序序列进栈，出栈的序列一定是一个中序序列。

那么如何确定这一结论呢？

可以参照先序遍历和中序遍历的非递归算法，不难得到：

对一棵树或者子树进行先序或者中序遍历时入栈顺序相同，都是根节点、左子树、右子树，先序遍历时访问后入栈，中序遍历是出栈后访问，则先序序列就是入栈顺序，中序序列就是出栈顺序。如下面代码所示：

```
void preOrder2(BinTree *root)     //非递归前序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL || !s.empty())
    {
        while(p!=NULL)
        {
            cout<<p->data<<"";  //入栈前输出节点的值
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            s.pop();
            p=p->rchild;
        }
    }
}
void inOrder2(BinTree *root)      //非递归中序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            cout<<p->data<<"";  //出栈前输出栈顶节点的值
            s.pop();
```

```
            p=p->rchild;
        }
    }
}
```

所以本题可以根据键盘输入的 push 和 pop 操作，此为树的中序遍历非递归算法时的进出栈顺序，也即为建立二叉树的先序序列，将这个序列存在数组中，并通过先序遍历建立二叉树，由于题目要求输出二叉树的后序遍历序列，那么将二叉树建好之后，再用后序遍历的方法输出序列。

2.2.4 功能说明

```
typedef struct BiNode {
    char data;
    struct BiNode* lchild, * rchild;
}BiNode,*BiTree;
BiTree T;//树的节点类型
char sq[200] = {};
int c = 0;
Status CreateBiTree(BiTree& T,int n)
{
    static int i = 0;
    char a = sq[i];
    i++;
    /*if (i == 2 * n - 1)
        return OK;*/
    if (a == ' ')
    {
        T = NULL;
    }
    else {
        if (!(T = (BiNode*)malloc(sizeof(BiNode))))
            exit(OVERFLOW);
        T->data = a;
        CreateBiTree(T->lchild,n);
        CreateBiTree(T->rchild,n);
    }
    return OK;
}//通过先序序列建立二叉树
Status visit(char a)
{
    cout << a;
    return OK;
}//访问节点
Status PostorderTraverse(BiTree& T)//后序遍历输出二叉树
{
    if (T == NULL)
```

```
        return 0;
    PostorderTraverse(T->lchild);
    PostorderTraverse(T->rchild);
    visit(T->data);
}
int main()
{
    int n;
    cin >> n;
    for (int i = 0;i < 2 * n;i++)
    {
        string s;
        cin >> s;
        if (s == "push")
        {
            char a;
            cin >> a;
            sq[c] = a;
            c++;
        }
        else if (s == "pop")
        {
            sq[c] = ' ';
            c++;
        }
    }//根据输入得到先序序列
    sq[2 * n] = ' ';
    CreateBiTree(T,n);
    PostorderTraverse(T);
    return 0;

}
```

2.2.5 调试分析

本题调试没有遇到什么大的困难，主要是在读题后不要急着写程序，要找规律，发现隐藏的结论后就十分容易。

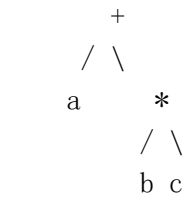
2.2.6 总结与收获

本题是一个探索结论性的问题，让我很深刻的理解了二叉树非递归算法出栈入栈顺序和遍历序列的关系，同时启示我在完成题目时，不能仅仅盯着结果，要从过程性的去思考某一已学知识与题目的关系，可能会隐藏一些结论，如果能够找到这些结论，可能可以大幅度简化程序，让算法更加简单高效。

2.3 表达式树

2.3.1 问题描述

任何一个表达式，都可以用一棵表达式树来表示。例如，表达式 a+b*c，可以表示为如下的表达式树：



现在，给你一个中缀表达式，这个中缀表达式用变量来表示（不含数字），请你将这个中缀表达式用表达式二叉树的形式输出出来。

2.3.2 基本要求

输入格式:

输入分为三个部分。

第一部分为一行，即中缀表达式(长度不大于 50)。

中缀表达式可能含有小写字母代表变量（a-z），也可能含有运算符（+、-、*、/、小括号），

不含有数字，也不含有空格。

第二部分为一个整数 n(n <= 10)，表示中缀表达式的变量数。

第三部分有 n 行，每行格式为 C x，C 为变量的字符，x 为该变量的值。

对于 20%的数据，1<=n<=3，1<=x<=5；

对于 40%的数据，1<=n<=5，1<=x<=10；

对于 100%的数据，1<=n<=10，1<=x<=100；

输出格式：

输出分为三个部分，第一个部分为该表达式的逆波兰式，即该表达式树的后根遍历结果。占一行。

第二部分为表达式树的显示，如样例输出所示。

如果该二叉树是一棵满二叉树，则最底部的叶子结点，分别占据横坐标的第 1、3、5、7……个位置（最左边的坐标是 1），

然后它们的父结点的横坐标，在两个子结点的中间。

如果不是满二叉树，则没有结点的地方，用空格填充（但请略去所有的行末空格）。

每一行父结点与子结点中隔开一行，用斜杠（/）与反斜杠（\）来表示树的关系。

/出现的横坐标位置为父结点的横坐标偏左一格，\出现的横坐标位置为父结点的横坐标偏右一格。

也就是说，如果树高为 m，则输出就有 2m-1 行。

第三部分为一个整数，表示将值代入变量之后，该中缀表达式的值。需要注意的一点是，除法代表整除运算，即舍弃小数点后的部分。

同时，测试数据保证不会出现除以 0 的现象。

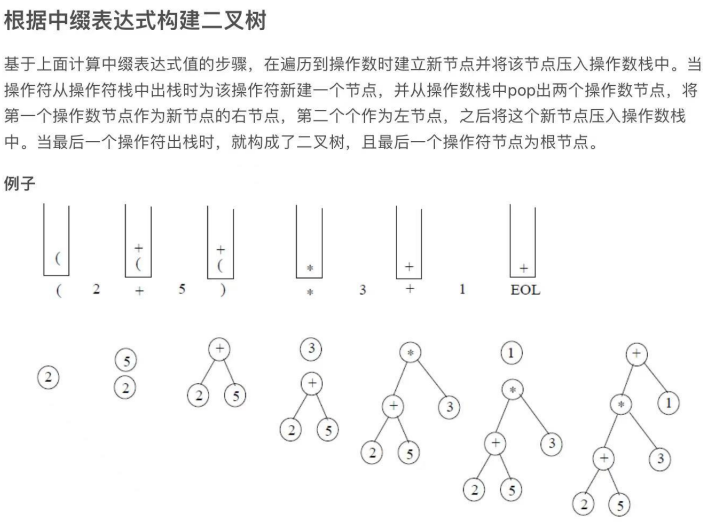
2.3.3 数据结构设计

本题可以分为三个子问题，首先是根据中缀表达式得到后缀表达式，然后是给表达式赋值，计算表达式，最后是按照格式打印表达式树的图形。

根据中缀表达式得到后缀表达式可以用栈来实现，由于只有加减乘除四种运算，所以只要遇到括号就进栈，遇到操作数则直接复制到 result 中，然后遇到优先级高的就进栈，最后得到含有后缀表达式的 result 数组。

给表达式赋值的操作我们可以在开头定义一个结构体 index，设置两个数据域，一个为变量的名称，一个为变量的值，这样的话我们可以用后缀表达式进行计算，读到某个变

量就对它的值进行运算。后缀表达式计算只需要一个栈来存放操作符，从左到右扫描表达式，遇到数字就将其压入栈，遇到操作符表示可以计算，这时取出栈顶的两个元素进行操作，然后再次将结果压入栈，最后栈里会留下一个元素，该元素就是运行结果。
打印图形是本题最难的地方，首先要用会层序遍历打印，其次要从叶子节点开始计算空格。假设树的深度为 h，在第 i 层的第一个节点距离最左边的距离为 $2^{h-i}-1$，然后同一层的两个节点之间距离为 2^{h-i}，同时还要注意到表示枝干关系的/和\需要在每个节点下方左右一个空格处输出，有了这种关系，我们便可以按照格式要求输出这颗二叉树。
下图是一个很好的形象化表述：



2.3.4 功能说明

```
void trans(string s,char result[])//将中缀表达式转为后缀表达式
{
    int i = 0;
    int j = 0;
    while (s[i] != '\0')
    {
        switch (s[i])
        {
        case'(':push(L, s[i]);
            break;
        case')':while (L.top != NULL && L.top != L.base && *L.top != '(')
        {
            result[j] =*L.top ;
            j++;
            pop(L);
        }
               pop(L);
               break;
        case'+':
        case'-':
            while (L.top &&L.top!=L.base&& *L.top != '(')
```

```
            {
                result[j] = *L.top;
                j++;
                pop(L);
            }
            push(L,s[i]);
            break;
        case'*':
        case'/':
            while (L.top && L.top != L.base && *L.top != '(' && *L.top == '*'
|| *L.top == '/')
            {
                result[j] = *L.top;
                j++;
                pop(L);
            }
            push(L, s[i]);
            break;
        default:result[j] = s[i];
            j++;
            break;

        }
        i++;
    }
    while (L.top != L.base)
    {
        result[j] = *L.top;
        L.top--;
        j++;
    }
    result[j] = '\0';
}
Status CreateBiTree(BiTree& T,char result[])//根据后缀表达式创建树
{
    BiTree stack[200]={};
    int i = 0;
    int top = -1;
    while (result[i] != '\0')
    {
        BiTree p;
        p = (BiNode*)malloc(sizeof(BiNode));//生成根节点
        if (!p)exit(OVERFLOW);
        p->data = result[i];
```

```
        p->lchild = NULL;
        p->rchild = NULL;
        if ((int)result[i] >= 97 && (int)result[i] <= 122)
        {
            top++;
            stack[top] = p;
        }
        else {
            p->lchild = stack[top-1];
            p->rchild = stack[top];
            stack[top-1] = p;
            top--;
            T = p;
        }
        i++;
    }
    T = stack[0];
}
void printtree(Bitree* root)//输出树
{
    int n = depth(root);

    char* r[10];
    Bitree* tmp = root, * queue[N], * p;
    int head = 0, tail = 0, i, t, h, j = 0;
    queue[tail] = root;//用队列实现层序遍历
    tail++;
    while (j < n)
    {
        r[j] = new char[300];
        for (i = head; i < tail; i++)
            r[j][i - head] = queue[i]->val;
        r[j][i - head] = '\0';
        t = tail;
        h = head;
        head = tail;
        for (i = h; i < t; i++)
        {
            if (queue[i]->lchild)
                queue[tail++] = queue[i]->lchild;
            else
            {
                p = new Bitree(' ');
                queue[tail++] = p;
```

```
            }
            if (queue[i]->rchild)
                queue[tail++] = queue[i]->rchild;
            else
            {
                p = new Bitree(' ');
                queue[tail++] = p;
            }
        }
        j++;
    }

    int d = 1, k, _n = 1;
    for (k = 0; k < n; k++)
        _n *= 2;
    for (i = 0; i < n; i++)
    {
        d *= 2;
        for (j = 0; r[i][j] != '\0'; j++)
        {
            print(j == 0 ? _n / div - 1 : _n * 2 / div - 1, ' ');
            cout << r[i][j];
        }
        cout << endl;
        if (i < n - 1)
        {
            for (j = 0; r[i][j] != '\0'; j++)
            {
                print(j == 0 ? _n / d - 2 : _n * 2 / d - 3, ' ');
                if (r[i + 1][2 * j] != ' ' && r[i + 1][2 * j + 1] != ' ')
                    cout << "/ \\";
                else
                    cout << "   ";
            }
            cout << endl;
        }
    }
}
int calculate(string s, index p[])
{
    int r[10], i, top = -1, j;
    char tmp;
    for (i = 0; i < s.length(); i++)
    {
```

```
        tmp = s[i];
        if (tmp == '+' || tmp == '-' || tmp == '*' || tmp == '/')
        {
            if (tmp == '+')
                r[top - 1] += re[top];
            else if (tmp == '-')
                r[top - 1] -= re[top];
            else if (tmp == '*')
                r[top - 1] *= re[top];
            else
                r[top - 1] /= re[top];
            top--;
        }
        else
        {
            for (j = 0; j < num; j++)
                if (p[j].name == tmp)
                {
                    r[++top] = p[j].val;
                    break;
                }
        }
        tmp = '\0';
    }
    return r[0];
}
```

2.3.5 调试与分析

本题是我这次作业耗时最长的一题，主要问题一是出在对栈的使用不够熟悉，每次编写程序可能还存在不少错误导致调试和改错时间过长，问题二是在于本题的输出问题，一开始想用一个二维数组进行输出，但是在层序遍历判断层数时出了问题，经过和同学的讨论，最后基本弄清楚了整个输出的过程。这也反映出我对基础知识的掌握不够牢固，应用不够熟练。本题调试问题一方面出在开始没有想到用一个结构体表示变量，而后缀表达式的 result[]数组又是 char 型的，导致如果运算的数值超过一位数就无法放入数组中进行存储，（其实也可以进行转换但是太过麻烦），在打印的问题上也遇到了不少困难。

2.3.6 总结与收获

本题是一种复杂的综合性问题，要把它细化为几个子问题进行求解，每个子问题都用到不同的算法思想，要逐个击破，特别是在遇到这种输出要求比较严苛的问题时，更应该严格要求自己，努力去达到高标准。

2.4 中序遍历线索二叉树

2.4.1 问题描述

练习线索二叉树的基本操作，包括二叉树的线索化，中序遍历线索二叉树，查找某元素在中序遍历的后继结点、前驱结点。

2.4.2 基本要求

输入

2 行

第 1 行，输入先序序列，内部结点用一个字符表示，空结点用#表示

第 2 行，输入一个字符 a，查找该字符的后继结点元素和前驱结点元素

输出

第 1 行，输出中序遍历序列

第 2 行，输出查找字符的结果，若不存在，输出一行 Not found，结束

接下来包含 N 组共 2N 行（N 为树种与 a 相等的所有节点个数）

每一组按其代表节点的中序遍历顺序排列（即按中序遍历搜索线索树，找到节点即可输出，直到遍历完整棵树）。

第 2k+3 行输出该字符的直接后继元素及该后继元素的 RTag,格式为:succ is 字符+RTag

第 2k+4 行输出该字符的直接前驱继元素及前驱元素的 LTag,格式为:prev is 字符+LTag

若无后继或前驱，用 NULL 表示。上述 k 有（0<=k<N/2）

2.4.3 数据结构设计

本题可以分为三个模块，首先根据先序序列建立二叉树；其次用中序遍历将其线索化；最后通过遍历查找节点的前驱后继。

先序建立二叉树可以采用递归建树，每一次读入一个元素，再分别建立左子树和右子树，比较容易实现。

中序遍历线索化的过程，首先创建头结点，令其 lchild 域的指针指向二叉树的根节点，其 rchild 域指向中序遍历时访问的最后一个节点；反之，令二叉树中序序列中的第一个节点的 lchild 域指针和最后一个节点 rchild 域的指针均指向头结点，这样好比为二叉树建立了一个双向链表，方便遍历。中序线索化是修改空指针的过程，用一个指针 pre 始终指向刚刚访问过的节点，指针 p 指向当前节点，那么 pre 永远是 p 的前驱。

线索化完成后，我们开始中序遍历线索二叉树并输出，其过程大致为：先一直向左下找到中序遍历的第一个元素，从此开始输出，输出一个便把指针指向其后继节点直到指向为空。

在查找某一特定节点的前驱后继时，我们可以用遍历线索二叉树的方法，找到 p->data=a 的节点 p，然后构造两个函数 findprev（）和 findsucc（），在线索二叉树中，对任意一个节点进行分析，不难得到任意一个节点前驱节点是其左子树最右下的节点，任意一个节点的后继节点是其右子树最左下的节点。

同时，由于题目要求对前驱为空和后继为空进行判断并输出 NULL，由于线索化的时候，我们把头结点的 lchild 域的指针指向二叉树的根节点，rchild 域指向中序遍历时访问的最后一个节点，令二叉树中序序列中的第一个节点的 lchild 域指针和最后一个节点 rchild 域的指针均指向头结点，此时可以把 p->lchild=Thrt 作为没有前驱节点的判断条件，把 p->rchild=Thrt 作为没有后继节点的判断条件。而不能采用 p->rchild(lchild)=NULL 来判断，因为指向的头结点并不为空，会导致即使为空也输出一个空值。

2.4.4 功能说明

typedef struct BiThrNode

```
{
    char data;
    struct BiThrNode* lchild, * rchild;
    int LTag, RTag;    //线索标志
}BiThrNode, * BiThrTree;
BiThrTree pre=new BiThrNode;//pre 先驱节点，定义为一个全局变量方便使用
void CreateBiTree(BiThrTree& T)
{
    char ch;
    /*按先序次序输入二叉树中结点的值(一个字符)，创建二叉链表表示的二叉树 T*/
    cin >> ch;
    if (ch == '#') T = NULL;
    else
    {
        T = (BiThrNode*)malloc(sizeof(BiThrNode));
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
}//先序创建二叉树
/*以结点 p 为根的子树中序线索化*/
void InThrding(BiThrTree p)
{

    /*pre 是全局变量，初始化时其右孩子指针为空，便于在树的最左点开始建线索*/
    if (p)
    {
        InThrding(p->lchild);            /*左子树递归线索化*/
        if (!(p->lchild))                          {
            p->LTag = 1;
            p->lchild = pre;                  //p 的左孩子指针指向 pre(前驱)
        }
        else
        {
            p->LTag = 0;
        }
        if (!(pre->rchild))
        {
            pre->RTag = 1;
            pre->rchild = p;                  //pre 的右孩子指针指向 p(后继)
        }
        else
        {
            pre->RTag = 0;
```

```
        }
        pre = p;                                /*保持pre指向p的前驱*/
        InThrding(p->rchild);           /*右子树递归线索化*/
    }
}
void InOrderThrding(BiThrTree& Thrt, BiThrTree T)
{
    /*中序遍历二叉树T，并将其中序线索化，Thrt指向头结点*/
    Thrt = (BiThrTree)malloc(sizeof(BiThrNode));          /*建头结头*/
    if (!Thrt)
        exit (OVERFLOW);
    Thrt->LTag = 0;
    Thrt->RTag = 1;
    Thrt->rchild = Thrt;
    if (!T) Thrt->lchild = Thrt;   //头结点初始化
    else
    {
        Thrt->lchild = T; pre = Thrt; /*头结点的左孩子指向根，pre初值指向头
结点*/
        InThrding(T);             /*调用上述算法，对以T为根的二叉树进行中序
线索化*/
        pre->rchild = Thrt;         /*算法结束后，pre为最右结点，pre的右线索
指向头结点*/
        pre->RTag = 1;
        Thrt->rchild = pre;         /*头结点的右线索指向pre*/
    }
}
char InOrderTraverse_Thr(BiThrTree T)
{
    char last=' ';
    /*T指向头结点，头结点的左链lchild指向根结点*/
    /*中序遍历二叉线索树T的非递归算法，对每个数据元素直接输出*/
    BiThrTree p = T->lchild;    /*p指向根结点*/
    while (p != T)
    {
        while (p->LTag == 0)       /*沿左孩子向下*/
        {
            p = p->lchild;
        }
        cout << p->data ;
        last = p->data;/*访问其左子树为空的结点*/
        while (p->RTag == 1 && p->rchild != T)  /*沿右线索访问后继结点*/
        {
            p = p->rchild;
```

```
            cout << p->data ;
            last = p->data;
        }
        p = p->rchild;
    }
    return last;
}
int findsucc(BiThrTree p)//找到p右子树的最左下节点即为后继
{
    int flag = 0;
    if (p->RTag == 0)
    {
        BiThrTree q = p->rchild;
        while (q->LTag == 0)
            q = q->lchild;
        if (q->data!='!')
        cout<<"succ is "<< q->data << q->RTag<<endl;
        flag = 1;
    }
    else if (p->RTag == 1)
    {
        if (p->lchild->data!='!')
            cout << "succ is " << p->rchild->data << p->rchild->RTag<<endl;
            flag = 1;
    }
    return flag;
}
int findprev(BiThrTree p)//找到p左子树的最右下节点即为前驱
{
    int flag = 0;
    if (p->LTag == 0)
    {
        BiThrTree q = p->lchild;
        while (q->RTag == 0)
            q = q->rchild;
        if(q->data!='!')
        cout << "prev is " << q->data << q->LTag<<endl;
        flag = 1;
    }
    else if (p->LTag == 1)
    {
        if (p->lchild->data!='!')
            cout << "prev is " << p->lchild->data << p->lchild->LTag<<endl;
```

```
            flag = 1;
    }
    return flag;
}
void search(BiThrTree& Thrt,char a,char last)//查找前驱后继并输出
{
    BiThrTree p = Thrt->lchild;    /*p 指向根结点*/
    int flag = 0;
    while (p != Thrt)
    {
        while (p->LTag == 0)       /*沿左孩子向下*/
        {
            p = p->lchild;
        }
        if (p->data == a)
        {
            flag = 1;//找到了 flag 为 1
            if (p->rchild == Thrt)
                cout << "succ is NULL" << endl;
            else findsucc(p);
            if (p ->lchild==Thrt)
                cout << "prev is NULL"<<endl;
            else
                findprev(p);
        }
        while (p->RTag == 1 && p->rchild != Thrt)  /*沿右线索访问后继结点*/
        {
            p = p->rchild;
            if (p->data == a)
            {
                flag = 1;//找到了 flag 为 1
                if (p->rchild == Thrt)
                    cout << "succ is NULL" << endl;
                else findsucc(p);
                if (p ->lchild==Thrt)
                    cout << "prev is NULL" << endl;
                else
                 findprev(p);
            }
        }
        p = p->rchild;
    }
    if(flag==0)//没找到，输出 Not found
    cout << "Not found" << endl;
```

```
}
int main()
{
    BiThrTree T;
    BiThrTree Thrt=new BiThrNode;
    Thrt->data = '!';
    CreateBiTree(T);
    /*Thrt = T;*/
    pre->RTag = 1;
    pre->rchild = NULL;
    char a;
    cin >> a;
    InOrderThrding(Thrt, T); /*带头结点的中序线索化*/
    char last=InOrderTraverse_Thr(Thrt);/*遍历中序线索二叉树*/
    cout << ans;
    cout << endl;
    search(Thrt, a,last);
    return 0;
}
```

2.4.5 调试与分析

本题是实现中序线索化二叉树并查找前驱后继的问题，仍然可以分为三个部分，第一，建立先序二叉树，第二，中序线索化二叉树并中序遍历输出，第三，查找某一节点的前驱后继并输出。

此题在调试过程中遇到的比较难的地方是查找前驱后继，开始由于我的对于判断前驱后继为空的条件是想当然的写错 p->rchild（lchild）=NULL，但是却忘记了或者说没有理解线索化过程实际上是建立了一个头结点 Thrt 和二叉树的双向链表，所以虽然没有给头结点赋值，但是头指针不是空，是为它分配了空间的，而中序序列的最后一个节点的右指针指向头结点，第一个节点的左指针指向头结点，实际上是方便了我们进行遍历判断的。同时在判断输出时，一定要确定节点数据不为空再输出，在之前判断条件不正确的时候，输出可能为变成‘？’并且卡住。

同时，在调试测试数据时，当数据量过大时如果用 vs 窗口直接输入会出现如下情况，即程序无响应，甚至还可以继续输入，如图 6。一方面可能是剪切板容量有限，没有输入完全全部数据，一方面可能是编译器设置的输入上限等问题。

图 6

解决方法是用 cmd 窗口管道运算符进行判断，cd 到测试数据素哟在文件夹，把 exe 文件也复制过去，输入 demo.exe<input6.txt>result 6.txt 即可以在文件夹中打开 result6

文本文档看到输出结果。再输入：fc result6.txt output6.txt
即可比较两输出文件是否相同，即答案是否正确。

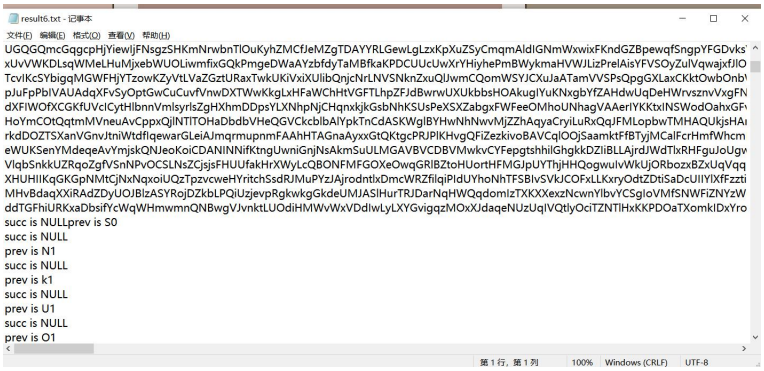


图 7

2.4.6 总结与收获

本题看似是课本上的基础操作，却也需要自己一步步去实现，并且对于查找前驱节点和后继节点的操作也需要对中序遍历有着比较深的理解，要发现其实前驱节点都是左孩子的右下元素，后继节点就是右孩子的最左下元素，充分利用二叉树的遍历顺序和性质进行编程。对于空节点的判断和处理也要注意，开始时要把 pre 的右子树置空，方便在左边开始进行遍历，并且头指针 Thrt 的初始化也要每一步都不能漏掉，整个线索化是一个严谨的系统工程，任意一个环节出差错都会导致遍历和查找时出现问题。并且在调试的时候要尽量用 cmd 窗口进行测试数据，更为精确和专业。

2.5 树的重构

2.5.1 问题描述

树木在计算机科学中有许多应用。 也许最常用的树是二叉树，但也有其他的同样有用的树类型。 其中一个例子是有序树（任意节点的子树是有序的）。
每个节点的子节点数是可变的，并且数量没有限制。 一般而言， 有序树由有限节点集合 T 组成，并且满足：

1.其中一个节点置为根节点，定义为 root(T)；

2.其他节点被划分为若干子集 T1,T2,...Tm,每个子集都是一个树.

同样定义 root(T1),root(T2),...root(Tm)为 root(T)的孩子，其中 root(Ti)是第 i 个孩子。节点 root(T1),...root(Tm)是兄弟节点。
通常将一个有序树表示为二叉树是更加有用的，这样每个节点可以存储在相同内存空间中。有序树到二叉树的转化步骤为：

1.去除每个节点与其子节点的边

2.对于每一个节点，在它与第一个孩子节点（如果存在）之间添加一条边，作为该节点的左孩子

3.对于每一个节点，在它与下一个兄弟节点（如果存在）之间添加一条边，作为该节点的右孩子

如图所示：

```
    0                          0
  / | \                       /
```

在大多数情况下，树的深度（从根节点到叶子节点的边数的最大值）都会在转化后增加。这是不希望发生的事情，因为很多算法的复杂度都取决于树的深度。
实现一个程序来计算转化前后的树的深度。

2.5.2 基本要求

输入由多行组成，每一行都是一棵树的深度优先遍历时的方向。其中 d 表示下行(down)，u 表示上行(up)。
例如上面的树就是 dudduduudu,表示从 0 下行到 1,1 上行到 0,0 下行到 2 等等。输入的截止为以#开始的行。
可以假设每棵树至少含有 2 个节点，最多 10000 个节点。

输入

输入由多行组成，每一行都是一棵树的深度优先遍历时的方向。其中 d 表示下行(down)，u 表示上行(up)。
例如上面的树就是 dudduduudu,表示从 0 下行到 1,1 上行到 0,0 下行到 2 等等。输入的截止为以#开始的行。
可以假设每棵树至少含有 2 个节点，最多 10000 个节点。

输出

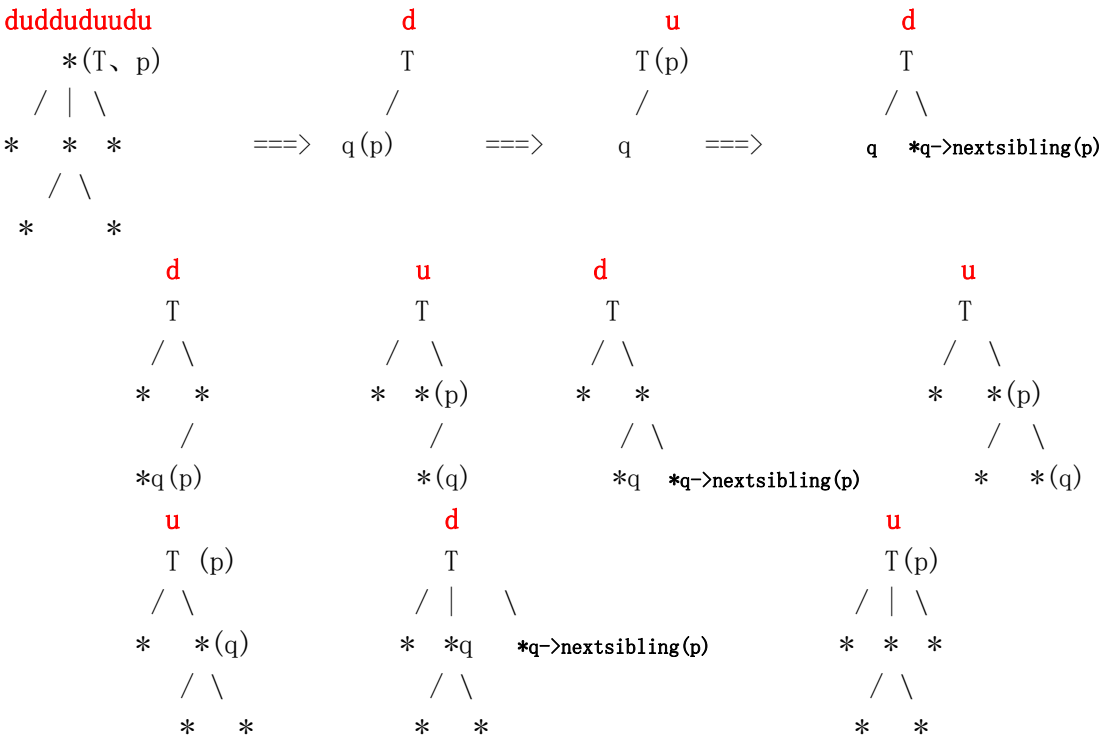
对每棵树，打印转化前后的树的深度，采用以下格式 Tree t: h1 => h2。其中 t 表示样例编号(从 1 开始)，h1 是转化前的树的深度，h2 是转化后树的深度。

2.5.3 数据结构设计

本题实际上是要求把一颗树按规则转化为二叉树，再分别计算深度。首先我们考虑树的储存结构，有孩子表示法、双亲表示法、孩子兄弟表示法等等，根据本题的要求，把树变成二叉树时，对于每一个节点在它与第一个孩子节点（如果存在）之间添加一条边，作为该节点的左孩子，在它与下一个兄弟节点（如果存在）之间添加一条边，作为该节点的右孩子。这显然符合 firstchild、nextsibling 的存储结构，于是我们尝试用孩子兄弟表示法建树并计算深度。
建树过程是本题相对来说的难点。由于输入给出的是对原来树的深度优先搜索方向序列 d 和 u，所以我们不妨按照其搜索顺序来建树，但是又由于存在向上方向的遍历，我们需要多次找一个节点的父亲节点，不妨在类型定义中加入一个指针域 parent 指向节点的父亲节点。首先初始化树，令根节点的数据域为*，其他域全部为空，令指针 p=T，然后开始读取字符串建树。设置一个指针 q 初始化为 NULL 辅助遍历。

如果读到的元素是 d,代表向下遍历,这时候分两种情况,首先如果 p 没有 firstchild 域,说明此时遍历的是 p 的 firstchild,分配内存并使 q 指向它,q 的 parent 域指向 p,其他指针域置空，p 也更新到 q，此时 q=p；第二种情况在叙述完情况为 u 的情况再进行解释。

如果读到的元素是 u，代表向上遍历，此时 p 更新到 p 的 parent 域，q 是否更新要看情况，如果在 p 更新前 q=p，代表这是 q=p 后第一次向上遍历，此时 q 不需要更新，保存上一次遍历的节点，如果更新前 q 不等于 p，代表在 q=p 前已经向上遍历过，此时 q 需要更新到 p 的下一层，即 q=q->parent。
对于读到元素是 d 的第二种情况，如果 p 已经有 firstchild 了，那么 p 遍历的下一个节点一定是和 firstchild 一层的节点，由以上可知 q 每次储存的就是这个节点，所以只需要为 q->nextsibling 分配空间，然后 q 更新到 q->nextsibling 再将 q->parent 指向 p，值为*，其他指针域置空最后将 p 更新过来即可。
用图示更加好理解，以样例中的树为例：



然后就是求深度问题，求深度算法之前已经实现过很多遍了，现在需要从两个角度进行求深度，一个是孩子兄弟链表实际上就是转换之后的二叉树，直接根据二叉树求深度的方法递归求解即可。
而在计算原来的树的深度时，因为此时的右孩子对应的是原来树中该结点临近的兄弟，它们位于同一层次，实际上树的深度等于 max(左子树的深度加+1,右树深度)。

2.5.4 功能说明

```
typedef struct CSNode {
    char data;
    struct CSNode* firstchild, * nextsibling,*parent;
}CSNode, * CSTree;//孩子兄弟链表，加上 parent 域
CSTree T;
Status InitCSTree(CSTree& T)
{
    T = (CSNode*)malloc(sizeof(CSNode));
    if (!T)
```

```
        exit(OVERFLOW);
    T->data = '*';
    T->parent = NULL;
    T->firstchild = NULL;
    T->nextsibling = NULL;
    return OK;
}//初始化树
Status CreateCSTree(CSTree &T, string s)
{
    int i = 0;
    int count = 0;
    CSTree p = T;
    CSTree q = NULL;//辅助指针 p
    while (s[i] != '\0')
    {
        if (s[i] == 'd')
        {
            if (p->firstchild == NULL)
            {
                q = (CSNode*)malloc(sizeof(CSNode));
                if (!q)
                    exit(OVERFLOW);
                q->data = '*';
                q->parent = p;
                q->firstchild = NULL;
                q->nextsibling = NULL;
                p->firstchild=q;
                p = q;
            }
            else {
                q->nextsibling = (CSNode*)malloc(sizeof(CSNode));
                if (!q->nextsibling)
                    exit(OVERFLOW);
                q = q->nextsibling;
                q->parent = p;
                q->data = '*';
                q->firstchild = NULL;
                q->nextsibling = NULL;
                p = q;
            }
        }//根据以上两种情况讨论
        else if (s[i] == 'u')
        {
            if (q != p)
```

```
                q = q->parent;
            p = p->parent;
        }
        i++;
    }
    return OK;
}
int depth_BiTree(CSTree& T)
{
    if (T == NULL)
        return 0;
    else {
        int i = depth_BiTree(T->firstchild);
        int j = depth_BiTree(T->nextsibling);
        if (i > j)
            return i+1;
        else return j+1;
    }
}//递归求二叉链表深度
int depth_CSTree(CSTree& T)
{

    if (T == NULL)
            return 0;
        int firstchild_depth = depth_CSTree(T->firstchild);
        int nextsibling_depth = depth_CSTree(T->nextsibling);

        return (firstchild_depth + 1 > nextsibling_depth ? firstchild_depth +
1 : nextsibling_depth);

}//递归求树深度
int main()
{
    string s;
    int i = 1;
    while (1)
    {
        cin >> s;
        if (s == "#")
            break;//停止条件
        InitCSTree(T);
        CreateCSTree(T, s);
        cout << "Tree " << i << ": " << depth_CSTree(T)-1 << " => " <<
depth_BiTree(T)-1<<endl;
```

```
        i++;
    }
    return 0;

}
```

2.5.5 调试与分析

本题主要是建树时指针和指针域比较多，容易造成没有初始化或者指针域没有更新的情况，有几个细节是 q 更新的条件要进行判断，是在 p 不等于 q 的时候 q 才和 p 一起更新，其次是不能给没有分配内存的指针域赋值，要先分配内存在进行更新和赋值。最后调试遇到的一点问题是主函数中输入时误写成 while（s!=”#”）这样的话最后输入#也会执行操作，但是 vs 编译器没有显示输出导致结果一直出错，这是写循环的习惯不够好，没有仔细判断终止条件。

2.5.6 总结与收获

本题主要是让我们自己实现了孩子兄弟表示法，首先需要看出来本题需要用这种特殊的存储结构更方便，其次是课本上没有的通过深度优先遍历顺序建造孩子兄弟二叉链表。这种算法不是简简单单模版类的操作，我在本题中通过不断画图用想出来这种用两个指针遍历建立的算法，自己也比较有成就感，同时自己的代码和算法能力有一定的提高。

3.实验总结

本次实验是学习数据结构以来个人感觉比较难的一次作业，每个题都侧重这一板块的不同知识点，比如树的同构考察我们对递归程序的编写以及求深度的操作，非递归遍历要求我们理解栈遍历操作和先序中序后序遍历的关系和过程，表达式树是让我们理解一种比较特别的二叉树的构造以及树形输出二叉树的同时考察了层序遍历与队列结合的知识，后面两题线索二叉树以及一般化树的存储结构，考察了我们线索化以及查找的知识，以及一种较为常用的孩子兄弟表示法的知识，透过 ppt 上的图示我们可能只是知道这种数据结构的思想，但是当我们真正去实现的时候才知道自己的漏洞在哪里，作为转专业的学生我的数组、链表以及代码基础并不算好，在高强度的学习数据结构知识的同时也是在加大我的代码量，提高我的代码能力，巩固一些基础知识，很高兴自己能面对对我来说很难的内容没有掉队没有跟不上，也感谢老师和同学们的耐心解答，题目 AC 的成就感让我感觉比别人花的长的多的时间没有白花，希望自己继续加油！