作业 HW3 实验报告

姓名：郑博远 学号：2154312 日期：2022 年 10 月 20 日

1. 涉及数据结构和相关背景

本次作业涉及的数据结构为树型结构这一重要非线性数据结构中的树与二叉树，二者从直观上来看是以分支关系定义的层次结构。树是 n 个结点的有限集，其能够满足：1）有且仅有一个特定的称为根的结点；2）当 n>1 时，其余的结点可分为 m 个互不相交的有限集 $T_1, \cdots, T_m$，其中每个集合本身又是一棵树。

二叉树是另一种树型结构，它的特点是每个结点至多只有两棵子树（即在二叉树中，不存在度大于 2 的结点）。并且，二叉树的子树有左右之分，其左右子树的次序不能够任意颠倒。

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 中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，就能够得到另外一棵树。同理可得，图 2 中的两棵树就不是同构的。

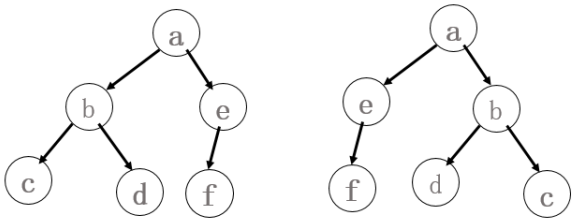


图 1 同构的二叉树

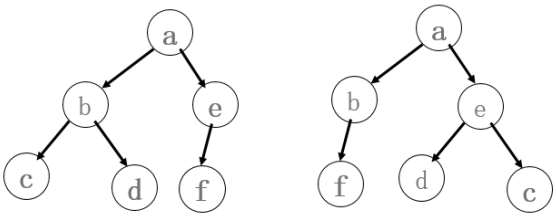


图 2 不同构的二叉树

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。

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

```
//队列 用于类似层次遍历的方式求树的深度
template <class QElemType>
class SqQueue {
private:
    QElemType* base;     //申请空间的指针
    int MAXQSIZE;        //队列的最大长度
    int front;           //队首指针
        int rear;        //队尾指针
};

typedef struct BiTNode {
    TElemType data;          //树元素数据
    BiTNode* lchild, * rchild;     //左右孩子
}*BiTree;
```

2.1.4 功能说明（函数、类）

1. SqQueue 队列类的成员函数：

```
/*
* SqQueue(int maxqsize)
```

```
* @param maxqsize 队列的最大长度
*/
template <class QElemType>
SqQueue<QElemType>::SqQueue(int maxqsize);
//空队列的建立（构造函数）

/*
* SqQueue()
*/
template <class QElemType>
SqQueue<QElemType>:: ~SqQueue();
//销毁已有的队列（析构函数）

/*
* Status ClearQueue()
*/
template <class QElemType>
Status SqQueue<QElemType>::ClearQueue();
//把现有队列置空队列

/*
* Status GetHead(QElemType& e)
* @param e 取到的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::Top(QElemType& e);
//取队首元素

/*
* Status DeQueue(QElemType& e)
* @param e 取出的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::DeQueue(QElemType& e);
//弹出队首元素

/*
* Status EnQueue(QElemType e)
* @param e 入队列的元素
*/
template <class QElemType>
Status SqQueue<QElemType>:: EnQueue(QElemType e);
//新元素入队列
```

```
/*
* bool QueueEmpty()
*/
template <class QElemType>
bool SqQueue<QElemType>::QueueEmpty();
//当前队列是否为空队列

/*
* int QueueLength ()
*/
template <class QElemType>
int SqQueue<QElemType>::QueueLength();
//返回当前队列的长度
```

2. 操作 BiTree 的相关函数:

```
/*
* Status CreateBiTree(BiTree& T)
* @param T 二叉树的根结点指针
*/
Status CreateBiTree(BiTree& T)
//二叉树的构建

/*
* bool IsSimilarTree(BiTree T1, BiTree T2)
* @param T1 二叉树1的根结点指针
* @param T2 二叉树2的根结点指针
*/
bool IsSimilarTree(BiTree T1, BiTree T2)
//递归方法判断两颗二叉树是否同构

/*
* int GetDepth(BiTree T)
* @param T 二叉树的根结点指针
*/
int GetDepth(BiTree T)
//非递归方式求二叉树的深度
```

2.1.5 调试分析

作为本次作业的第一题，本题考查对二叉树的基本建立和遍历，比较简单。

在调试过程中，出现了如下三个问题:

1. 判断两棵树是否同构时，可以使用递归的方式：即，两棵树同构当且仅当根结点相等，且两棵树的左子树与右子树均同构（或其中一棵树的左子树与另外一棵树的右子树，其中一棵树的右子树与另外一棵树的左子树均同构）。这含有很明显的递归思想，因此采用递归的方式解决。但我在调用递归时，错误的只判断了两棵树的左子树同构或一棵树的左子树与另一棵树的右子树同构这一条件，忽略了此时另一侧对应的两个子树也要判断是否同构。这样的同构判断条件过于宽松，便导致了错误。判断同构的方式增加了条件之后，结果便正确了；

2. 使用递归的方式来求算树的深度是一种写法简单且较容易想到的方式。但是，当树的结构类似于一条单链、树的深度很大时，很容易导致栈溢出。本题中，我一开始采用递归方式求算树的深度，由于测试样例中的数据量比较大，导致了许多测试点 Runtime Error 错误。后来，我根据层次遍历的算法进行了改进，通过队列的方式求算树的深度，解决了该问题。

图 3　使用递归方式求树深度导致栈溢出的 Runtime Error 报错

3. 在建二叉树时，给出的左右孩子信息可能是数字，也可能是代表不存在左右孩子的“-”。一开始我选择使用 char 类型变量输入，但测试点中结点个数不

止 10 个，数字编号会超过个位数，这就导致了建树的错误。后来改用 string 存储输入的左右孩子信息，再进行特别判断“-”解决问题。

2.1.6 总结和体会

在本题中，我体会了与课本中不同的另一种二叉树的建立方式。不同于课本中的先序插入方式，本题中的输入方式是给出每个结点的左右孩子信息。在分别创建各个结点的对应关系之后，还需要遍历一遍所有结点，找出不是任何结点的左右孩子的结点，即为二叉树的根结点。

判断两棵二叉树是否同构的方式是一种不同于书本中的前序、中序、后序的另一种递归方式，是本题当中我学习到最多的地方。本题中判断同构的方法即：两棵树同构当且仅当根结点相等，且两棵树的左子树与右子树均同构（或其中一棵树的左子树与另外一棵树的右子树，其中一棵树的右子树与另外一棵树的左子树均同构）；通过这样的方式继续递归其左右子树直至走到 NULL 停止。

通过递归方式求树深度比较容易思考且写法简单，但当层数过多容易导致栈溢出。因此可以采用类似层次遍历的方式使用非递归的方法求算二叉树深度。

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 4 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。

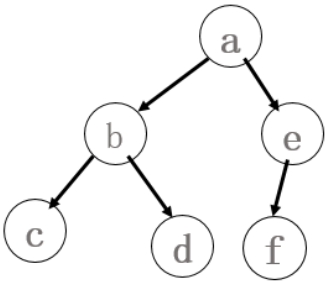


图 4　abc##d##ef###先序建立的二叉树

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。

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)

输出：多行，若给定的出站序列可以得到，输出 yes 一行，后序遍历序列。

2.2.3 数据结构设计

```
template <class SElemType>
struct SqStack{
private:
    SElemType* base;          //栈的底部
    SElemType* top;           //栈的顶部
    int stacksize;            //栈的空间大小
};

typedef struct BiTNode {
    TElemType data;           //树元素数据
    BiTNode* lchild, * rchild;     //左右孩子
}*BiTree;
```

2.2.4 功能说明（函数、类）

1. SqStack 栈类的成员函数：

```
/*
* SqStack()
*/
SqStack();                    //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                   //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();          //把现有栈置空栈

/*
* Status Top(SElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素

/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. 二叉树的操作函数:

```
/*
* Status CreateBiTree(BiTree& T)
* @param T 二叉树的根结点指针
*/
```

```
Status CreateBiTree(BiTree& T)
//二叉树的构建

/*
* Status PostOrderTraverse(BiTree T, Status(*visit)(TElemType e))
* @param T 二叉树的根结点指针
* @param visit 访问该结点的函数
*/
Status PostOrderTraverse(BiTree T, Status(*visit)(TElemType e))
//后序遍历访问该树

/*
* Status VisitPrintNormal(TElemType e)
* @param e 要访问的结点信息
*/
Status VisitPrintNormal(TElemType e)
//打印当前结点
```

2.2.5 调试分析

在 CreateBiTree 建立二叉树函数的编写上，我进行了多次的修改。最起初时，我简单的采用 push 和 pop 的栈顶元素来做子结点的插入，若当前结点无左子结点就将 push 的新结点作为左子结点，若当前结点以及有左子结点则将 push 的新结点作为右子结点。由于左子结点、右子结点插入位置等许多细节处理没有考虑到，这会导致插入的结点关系的错误。

修改之后，我在压入结点信息的栈元素中加入 state 元素。当 state 为 0 时，表示当前结点的左子结点与右子结点都没有被设置过；当 state 为 1 时，表示设置过左节；state 为 2 时，表示左右结点均被设置过。每次 push 操作会使当前栈顶的元素 state++，若 state 为 2 则元素应该出栈（中序，左右结点都被遍历过）。

2.2.6 总结和体会

本题的主要难点是通过 push 和 pop 操作来建立二叉树。这需要在对二叉树的中序遍历的非递归算法有所了解的基础上进行设计。通过本题，我复习了中序

遍历的递归方式与两种非递归方式，更深入的了解了二叉树的遍历过程，起到了很好的复习巩固作用。

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:

```
      +
     / \
    a   *
       / \
      b   c
```

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：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：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），

然后是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。

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

```
//队列 用于类似层次遍历的方式输出图形化的树
template <class QElemType>
class SqQueue {
private:
    QElemType* base;     //申请空间的指针
    int MAXQSIZE;        //队列的最大长度
    int front;           //队首指针
        int rear;        //队尾指针
};
```

```
template <class SElemType>
//栈 用于计算逆波兰表达式在值
struct SqStack{
private:
    SElemType* base;         //栈的底部
    SElemType* top;          //栈的顶部
    int stacksize;           //栈的空间大小
};

typedef struct BiTNode {
    TElemType data;          //树元素数据
    BiTNode* lchild, * rchild;     //左右孩子
}*BiTree;
```

2.3.4 功能说明（函数、类）

1. SqStack 栈类的成员函数:

```
/*
* SqStack()
*/
SqStack();                   //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                  //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();         //把现有栈置空栈

/*
* Status Top(SElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素
```

```
/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. SqQueue 队列类成员函数:

```
* SqQueue(int maxqsize)
* @param maxqsize 队列的最大长度
*/
template <class QElemType>
SqQueue<QElemType>::SqQueue(int maxqsize);
//空队列的建立（构造函数）

/*
* SqQueue()
*/
template <class QElemType>
SqQueue<QElemType>:: ~SqQueue();
//销毁已有的队列（析构函数）

/*
* Status ClearQueue()
*/
template <class QElemType>
Status SqQueue<QElemType>::ClearQueue();
//把现有队列置空队列

/*
* Status GetHead(QElemType& e)
* @param e 取到的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::Top(QElemType& e);
//取队首元素

/*
```

```
* Status DeQueue(QElemType& e)
* @param e 取出的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::DeQueue(QElemType& e);
//弹出队首元素

/*
* Status EnQueue(QElemType e)
* @param e 入队列的元素
*/
template <class QElemType>
Status SqQueue<QElemType>:: EnQueue(QElemType e);
//新元素入队列

/*
* bool QueueEmpty()
*/
template <class QElemType>
bool SqQueue<QElemType>::QueueEmpty();
//当前队列是否为空队列

/*
* int QueueLength ()
*/
template <class QElemType>
int SqQueue<QElemType>::QueueLength();
//返回当前队列的长度
```

3. 二叉树的操作函数:

```
/*
* Status CreateBiTree(BiTree& T)
* @param T 二叉树的根结点指针
*/
Status CreateBiTree(BiTree& T)
//二叉树的构建

/*
* void MakeExpressionTree(SqStack<BiTree>& Value,
SqStack<BiTree>& Operator)
* @param Value 操作数的栈
* @param Operator 操作符的栈
* @param visit 访问该结点的函数
```

```
    */
    void MakeExpressionTree(SqStack<BiTree>& Value, SqStack<BiTree>&
Operator)
    //做一次某符号运算（构建运算符和操作数的父子关系）

    /*
    *  Status  PostOrderTraverse(BiTree  T,  SqStack<int>&  stack,
    Status(*visit)(SqStack<int>& stack, TElemType e, set* list, int
    n), set* list, int n)
    * @param T 二叉树的根结点
    * @param stack 存放逆波兰式表达式的栈
    * @param visit 访问结点的函数
    * @param list 存放各变量值的数组指针
    * @param n 变量的个数
    */
    Status PostOrderTraverse(BiTree T, SqStack<int>& stack,
Status(*visit)(SqStack<int>& stack, TElemType e, set* list, int n),
set* list, int n)//打印当前结点

    /*
    * Status Print(SqStack<int>& stack, TElemType e, set* list, int
    n)
    * @param stack 存放逆波兰式表达式的栈
    * @param e 当前访问的结点
    * @param list 存放各变量值的数组指针
    * @param n 变量的个数
    */
    Status Print(SqStack<int>& stack, TElemType e, set* list, int n)
    //访问结点的函数（本题负责打印值 并 计算表达式）

    /*
    * Status PrintTree(BiTree T)
    * @param T 要打印的二叉树根结点指针
    */
    Status PrintTree(BiTree T)
    //图形化打印二叉树
```

4. 其他函数:

```
    /*
    * int GetValue(set* list, int n, char e)
    * @param list 存放各变量值的数组指针
    * @param n 变量的个数
    * @param e 要取值的变量名
```

```
*/
int GetValue(set* list, int n, char e)  //取某变量的值

/*
* void PrintSpace(int n)
* @param n 打印的空格数目
*/
void PrintSpace(int n)      //打印 n 个空格（类似 setw）

/*
* int pow2(int n)
* @param n 欲求的2的幂次
*/
int pow2(int n)             //求 2 的 n 次幂
```

2.3.5 调试分析

1. 右括号 ')' 入栈时，应该弹出所有操作符进行相应操作，直至遇到左括号停止出栈，继续向后进行入栈分析。但起初的程序在弹出所有的操作符并进行相应操作之后，忘记将左括号从存放操作符的栈中弹出，导致此后的括号匹配错误。每次匹配后将左括号 Pop 弹出即可解决；

2. 在函数之前通过参数传递栈的过程中，应该传递栈类的引用，否则会由于复制构造函数没有被定义从而造成浅拷贝。这会导致在栈被销毁的过程中有空间被反复释放而出错。将形参改为引用传递即可；

3. 在处理中缀表达式的时候，我的处理逻辑与栈与队列的表达式求解作业相类似。但是，我没有考虑到“*”、“/”与“+”、“-”算术符的左结合性质。也就是，当操作符栈顶是“/”时，若遇到优先级相同的“*”或“/”时，应该先进行栈内运算符的运算。若表达式为“6/2/2”，按照右结合则表达式值为 6，左结合则表达式值为 1。“*”与“+”由于其结合性不会有值的问题，但建立的二叉树会有不同。因此，修改程序使得遇到优先级相同的“*”或“/”时，先进行栈内运算符的运算。

2.3.6 总结和体会

本题的建树过程比较新颖。我采用了类似栈与队列作业中的布尔表达式一题计算表达式的方式，不同的点在于建树时不进行计算，而将两操作数作为操作符的左右结点，再加操作符压入操作数的栈内（如图 5）。本题中针对操作数的不同变量与操作符加减乘除与括号的表达式计算也是对栈这一数据结构的复习。

```
void MakeExpressionTree(SqStack<BiTree>& Value, SqStack<BiTree>& Operator)
{
    BiTree opr, a, b;
    Operator.Pop(opr);
    Value.Pop(a);
    Value.Pop(b);
    opr->lchild = b;
    opr->rchild = a;
    Value.Push(opr);
}
```

图 5　执行栈顶操作符相应操作的函数

建树的过程中，分别用两个栈记录操作数与操作符。读取到操作数时直接入栈；读取到操作符时，依次比较存放操作符栈顶的符号。若当前读取到的符号优先级低于栈顶的符号，则将其入栈；否则，不断弹出栈顶符号并执行其对应操作（让操作数栈栈顶两个元素为操作符的左右孩子，并将操作符压入操作数栈中），直至栈顶符号优先级低于入栈符号或栈为空。所有操作数、操作符入栈之后，依次弹出栈中所有操作符。最终操作数栈顶（且栈中唯一的）元素即为根结点。

图形化打印表达式二叉树比较繁琐，需要调整空格的数量、位置，左右斜杠的位置等。打印时存在需要计算 2 的次方来确认空格个数等细节问题，但总的来说属于层次遍历的一种，用队列结构即可大致解决。

计算表达式的值既可以直接在建树的过程中解决，也可以借助求出的逆波兰式。逆波兰式的压栈过程中，遇到运算符就将栈顶两个变量取出计算再压栈，最终即可得到表达式的值。

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： 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：一行，代表计算出的区域数量。

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 相等的所有结点个数）

每一组按其代表结点的中序遍历顺序排列（即按中序遍历搜索线索树，找到结点即可输出，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

```
enum class PointerTag { LINK, THREAD };
typedef char TElemType;
typedef struct BiThrNode {
    TElemType data;                    //结点元素
    BiThrNode* lchild, * rchild;       //左右孩子
    PointerTag LTag, RTag;             //左右孩子的Tag
}*BiThrTree;
```

```
//顺序表 在遍历时将每个结点加入 便于最后查找
typedef BiThrTree SqList_ElemType;
class SqList {
    SqList_ElemType* base;      //头指针
    int length;                 //表长
    int listsize;               //申请的表空间
}
```

2.4.4 功能说明（函数、类）

1. SqList 顺序表类的成员函数

```
/*
*SqList()
*/
SqList()  //顺序表的建立（构造函数）


/*
*~SqList()
*/
~SqList()  //顺序表的销毁（析构函数）


/*
* int ListLength()
*/
int ListLength()  //获取顺序表的表长


/*
* Status ListInsert(int i, SqList_ElemType e)
* @param i 插入元素的位置
* @param e 插入的元素
*/
Status ListInsert(int i, SqList_ElemType e) //在顺序表中插入元素


/*
* Status GetElem(int i, SqList_ElemType& e)
* @param i 获取元素的位置
* @param e 获取的元素
*/
Status GetElem(int i, SqList_ElemType& e) //在顺序表中获取元素
```

2. 二叉树的操作函数

```
    /*
    * Status CreateBiTree(BiTree& T)
    * @param T 二叉树的根结点指针
    */
    Status CreateBiTree(BiTree& T)
    //二叉树的构建

    /*
    * void InTHREADing(BiThrTree p, BiThrTree& pre)
    * @param p 当前正在线索化的结点指针
    * @param pre 上一个线索化的结点指针
    */
    void InTHREADing(BiThrTree p, BiThrTree& pre)
    //二叉树的线索化（由InOrderTHREADing调用）

    /*
    * Status InOrderTHREADing(BiThrTree& Thrt, BiThrTree T)
    * @param Thrt 线索二叉树的头结点
    * @param T 未线索化的二叉树根结点
    */
    Status InOrderTHREADing(BiThrTree& Thrt, BiThrTree T)
    //二叉树的线索化

    /*
    *  Status  InorderTraverse_Thr(BiThrTree  T,  SqList&  traverse,
    Status(*Visit)(BiThrTree e, SqList& traverse))
    * @param T 二叉树的头结点
    * @param traverse 存放遍历序列的顺序表
    * @param Visit 访问结点的函数
    */
    Status InorderTraverse_Thr(BiThrTree T, SqList& traverse,
Status(*Visit)(BiThrTree e, SqList& traverse)) //中序遍历

    /*
    * Status MyPrint(BiThrTree e, SqList& traverse)
    * @param e 当前访问的结点
    * @param traverse 存放序列的顺序表（为了和MyVisiit统一，此处无用）
    */
    Status MyPrint(BiThrTree e, SqList& traverse)
    //打印中序访问序列

    /*
    * Status MyTraverse(BiThrTree e, SqList& traverse)
    * @param e 当前访问的结点
```

```
* @param traverse 存放序列的顺序表
*/
Status MyVisit(BiThrTree e, SqList& traverse)
//存放中序访问序列

/*
* Status FindTarget(char tgt, SqList& traverse)
* @param tgt 要找到结点信息
* @param traverse 存放序列的顺序表
*/
Status FindTarget(char tgt, SqList& traverse)
//查找要找的结点并打印相关信息
```

2.4.5 调试分析

在输出最后的序列时，我统一将左孩子当作当前结点的前驱，将右孩子作为当前结点的后继，然后打印 LTag 和 RTag。实际上，当 LTag 与 RTag 不是 Thread 时，左孩子与右孩子存放的并非真正的前驱后继。最后我在遍历二叉树时将每个结点塞入顺序表中。最终查找某个元素的前驱后继时，再从头到尾 O(n)遍历一次整个顺序表来打印前驱后继及其对应的 Tag 便解决了问题。

2.4.6 总结和体会

本题主要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。主要的操作都比较基础，但由于是初次实践二叉树线索化比较不熟练，我在本次作业中得到了很好的锻炼。

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（任意结点的子树是有序的）。

每个结点的子结点数是可变的，并且数量没有限制。 一般而言， 有序树由

有限结点集合 T 组成，并且满足：

1.其中一个结点置为根结点，定义为 root(T);

2.其他结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。结点 root(T1),...root(Tm)是兄弟结点。

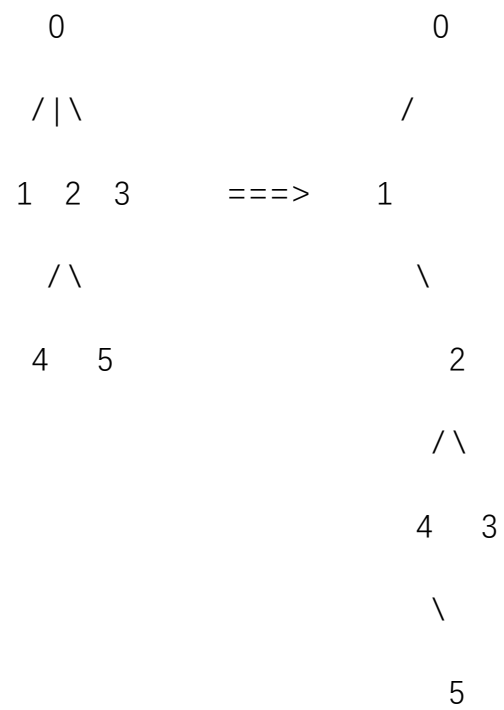
通常将一个有序树表示为二叉树是更加有用的，这样每个结点可以存储在相同内存空间中。有序树到二叉树的转化步骤为：

1.去除每个结点与其子结点的边;

2.对于每一个结点，在它与第一个孩子结点（如果存在）之间添加一条边，作为该结点的左孩子;

3.对于每一个结点，在它与下一个兄弟结点（如果存在）之间添加一条边，作为该结点的右孩子

如图所示：



在大多数情况下，树的深度（从根结点到叶子结点的边数的最大值）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。

现在，需要你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：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 个结点，最多 10000 个结点。

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 是转化后的树的深度。

2.5.3 数据结构设计

```
//队列 用于类似层次遍历的方式求树的深度
template <class QElemType>
class SqQueue {
private:
    QElemType* base;     //申请空间的指针
    int MAXQSIZE;        //队列的最大长度
    int front;           //队首指针
        int rear;        //队尾指针
};

template <class SElemType>
//栈 用于模拟 du 的入栈出栈操作
struct SqStack{
private:
    SElemType* base;         //栈的底部
    SElemType* top;          //栈的顶部
    int stacksize;           //栈的空间大小
};
```

```
typedef struct BiTNode {
    TElemType data;          //树元素数据
    BiTNode* lchild, * rchild;     //左右孩子
}*BiTree;
```

2.3.4 功能说明（函数、类）

1. SqStack 栈类的成员函数:

```
/*
* SqStack()
*/
SqStack();                  //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                 //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();        //把现有栈置空栈

/*
* Status Top(SElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素

/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
```

```
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. SqQueue 队列类成员函数:

```
* SqQueue(int maxqsize)
* @param maxqsize 队列的最大长度
*/
template <class QElemType>
SqQueue<QElemType>::SqQueue(int maxqsize);
//空队列的建立（构造函数）

/*
* SqQueue()
*/
template <class QElemType>
SqQueue<QElemType>:: ~SqQueue();
//销毁已有的队列（析构函数）

/*
* Status ClearQueue()
*/
template <class QElemType>
Status SqQueue<QElemType>::ClearQueue();
//把现有队列置空队列

/*
* Status GetHead(QElemType& e)
* @param e 取到的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::Top(QElemType& e);
//取队首元素

/*
* Status DeQueue(QElemType& e)
* @param e 取出的队首元素
*/
template <class QElemType>
Status SqQueue<QElemType>::DeQueue(QElemType& e);
//弹出队首元素

/*
* Status EnQueue(QElemType e)
```

```
* @param e 入队列的元素
*/
template <class QElemType>
Status SqQueue<QElemType>:: EnQueue(QElemType e);
//新元素入队列

/*
* bool QueueEmpty()
*/
template <class QElemType>
bool SqQueue<QElemType>::QueueEmpty();
//当前队列是否为空队列

/*
* int QueueLength ()
*/
template <class QElemType>
int SqQueue<QElemType>::QueueLength();
//返回当前队列的长度
```

3. 操作 BiTree 的相关函数：

```
/*
* Status CreateBiTree(BiTree& T, string input)
* @param T 二叉树的根结点指针
* @param input 输入的du序列
*/
Status CreateBiTree(BiTree& T, string input)
//二叉树的构建

/*
* int GetDepth(BiTree T)
* @param T 二叉树的根结点指针
*/
int GetDepth(BiTree T)
//非递归方式求二叉树的深度
```

2.5.5 调试分析

总体编写很顺利，没有遇到大的逻辑问题，很迅速地通过了。重点在于需要判断当前栈顶的结点是否已经被访问过，这决定了下一个 d 入的元素是接在已有的左孩子后还是当前栈顶的左孩子上；起初由于判断不清有一些小错误。

2.5.6 总结和体会

本题在概念上比较新颖，但是理解了从有序树到二叉树的转换之后，我能够比较逻辑清晰地高效完成。因为结点关系的建立在于：每一个节点在它与第一个孩子节点之间添加一条边，作为该节点的左孩子；在它与下一个兄弟节点间添加一条边，作为该节点的右孩子。因此需要判断当前栈顶的结点是否已经被访问过（即当前的栈顶元素是否已经访问过第一个左孩子），这决定了下一个 d 入的元素是接在已有的左孩子后还是当前栈顶的左孩子上。若已经有被访问过的左孩子，就需要将新 d 入的结点接在其右子树上；因此我用一个指针 pre 记录 pop 过的上一个被访问结点，这样就可以把新 d 入的结点接在 pre 结点的右子树上。

3. 实验总结

这次作业中，我巩固了对树型结构中二叉树的了解与使用，在不同的题目中使用了先序方式建树以及各种不同方式树的建立（如表达式树一题中类似于栈计算的方式来建树），掌握了二叉树的先序、中序、后序遍历三种方式，运用了递归与非递归的不同方式来解决问题。

通过这次作业，我在不同的题目之中也对之前作业中的顺序表、栈、队列这些数据结构进行了复习，学会了更灵活地使用它们。本次作业中用到的顺序表、栈、队列均使用了之前作业中的模板类，验证了前几周代码的健壮性。

总的来说，通过本次作业，我加深了树型结构尤其是其中二叉树的理解，实践了二叉树的线索化，也复习了各种基础操作。我希望通过这次作业的练习在未来能更加熟练掌握使用已学过的这些数据结构。