2. 程序框架搭建

3. 建立二叉树

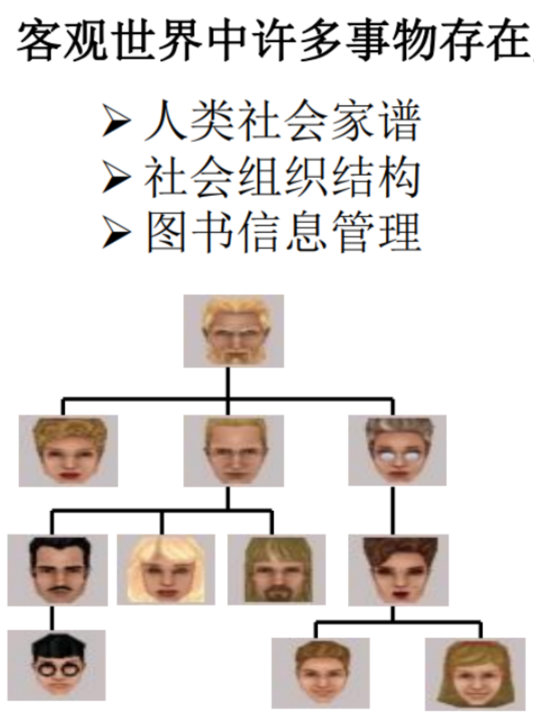
4. 判别两个二叉树是否同构

编程练习

1. 树的同构，见Part4
2. List Leaves
3. Tree Traversals Again

Part 1 树与树的表示

什么是树?



有效率地查找

查找：根据某个给定**关键字**K ，从**集合**R中找出关键字与K相同的记录

- 静态查找：集合中记录是固定的（没有插入，删除操作）
- 动态查找：集合中记录是动态变化的

静态查找

方法1：顺序查找 时间复杂度 `O(n)`

```c
int SequentialSearch(StaticTable* Tbl, ElementType K)
{
    //在Tbl[1]~Tbl[n]中查找关键字为K的数据元素
    Tb1->Element[0] = K;    //建立哨兵
    int i;
    for(i = Tbl->Length; Tbl->Element[i] != K; i--)
        ;
    return i;   //查找成功就返回对应下标，没找到K而退出循环时i走到了哨兵处，返回0
}
```

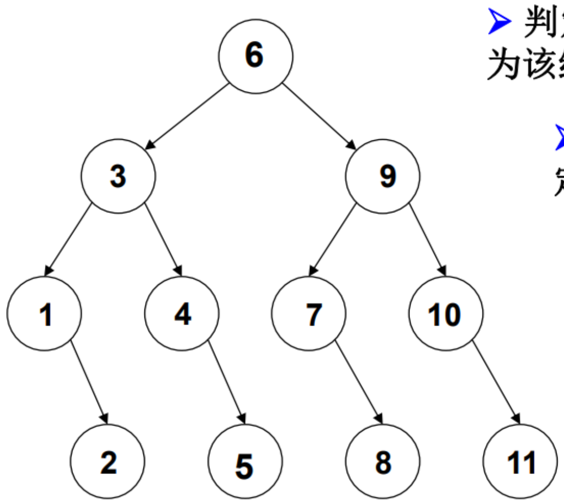
方法2：二分查找 时间复杂度 `O(logn)`

前提是，顺序表存储，关键码有序

```c
int BinarySearch(StaticTable* Tbl, ElementType K)
{
    int left, right, mid, NotFound = -1;
    left = 1, right = Tbl->Length;  //初始化左右边界
    while(left <= right)
    {
        mid = left + right >> 1;
        if(K < Tbl->Element[mid])       right = mid - 1;    //调整右边界
        else if(Tbl->Element[mid] < K)  left = mid + 1;     //调整左边界
        else return mid;    //查找成功
    }
    return NotFound;
}
```

11个元素的二分查找过程可以用下面的判定树来描述，包括了K取各种值的可能性

ASL Average Search Length



➢ 判定树上每个结点需要的查找次数刚好为该结点所在的层数;

➢ 查找成功时查找次数不会超过判定树的深度

➢ n个结点的判定树的深度为$[\log_2 n]+1$.

➢ ASL = (4*4+4*3+2*2+1)/11 = 3

二分查找的启示?

例如我们查找的是节点4，它位于第三层，查找它需要进入三次循环

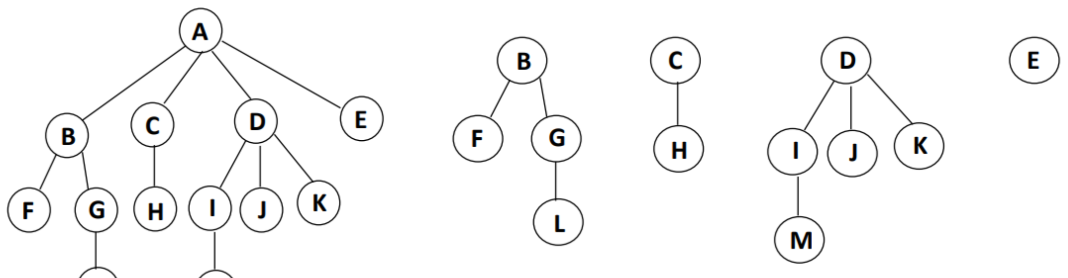
树的定义

树（Tree）：n（n≥0）个结点构成的有限集合。

当n=0时，称为空树；

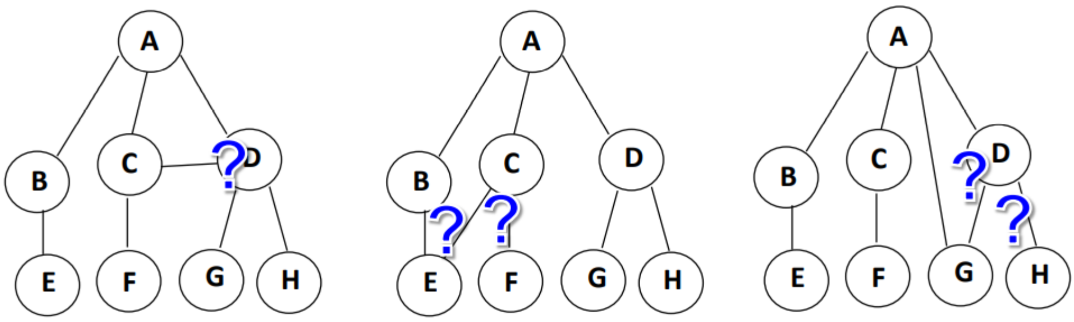
对于任一棵非空树（n> 0），它具备以下性质：

- 树中有一个称为“根（Root）”的特殊结点，用 r 表示；
- 其余结点可分为m(m>0)个互不相交的有限集T_1，T_2，… ，T_m，其中每个集合本身又是一棵树，称为原来树的“子树（SubTree）”



(a) 树 T　(b) 子树 T_{A1}　(c) 子树 T_{A2}　(d) 子树 T_{A3}　(e)子树 T_{A4}

❖ 树与非树?

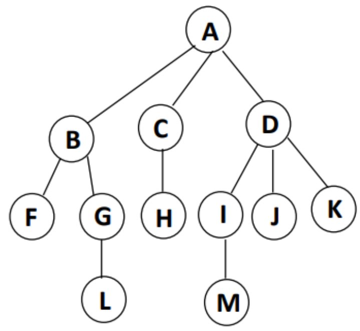


不满足下面条件的都不是树

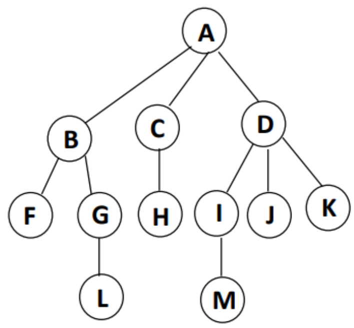
- 子树是**不相交的**
- 除了根节点外，**每个节点有且仅有一个父节点**
- 一颗N个节点的树有**N-1条边**

树的基本术语

1. 结点的度（Degree）：结点的子树个数
2. 树的度：树的所有结点中最大的度数
3. 叶结点（Leaf）：度为0的结点
4. 父结点（Parent）：有子树的结点是其子树的根结点的父结点
5. 子结点（Child）：若A结点是B结点的父结点，则称B结点是A结点的子结点；子结点也称孩子结点。
6. 兄弟结点（Sibling）：具有同一父结点的各结点彼此是兄弟结点。

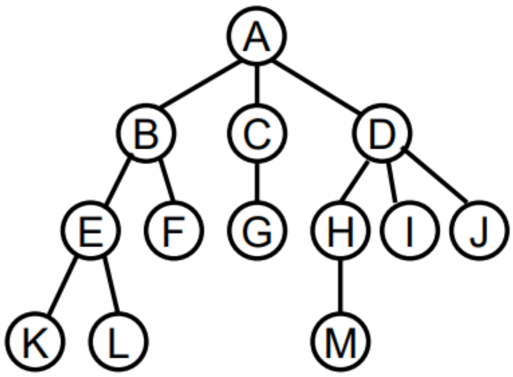


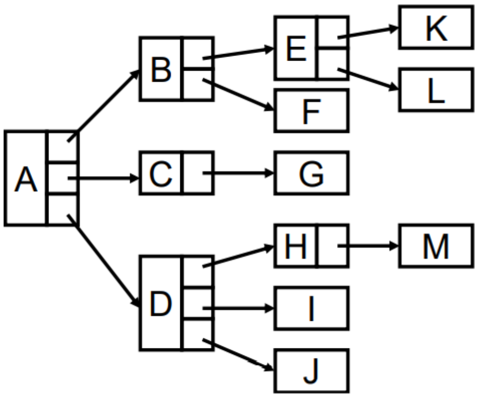
7. 路径和路径长度：从结点n_1到n_k的路径为一个结点序列n_1 , n_2 ,… , n_k , n_i是 n_{i+1}的父结点。路径所包含边的个数为路径的长度。
9. 祖先结点(Ancestor)：沿树根到某一结点路径上的所有结点都是这个结点的祖先结点。
10. 子孙结点(Descendant)：某一结点的子树中的所有结点是这个结点的子孙。
11. 结点的层次（Level）：规定根结点在1层，其它任一结点的层数是其父结点的层数加1。
12. 树的深度（Depth）：树中所有结点中的最大层次是这棵树的深度。



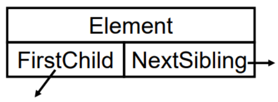
树的表示-儿子兄弟表示法

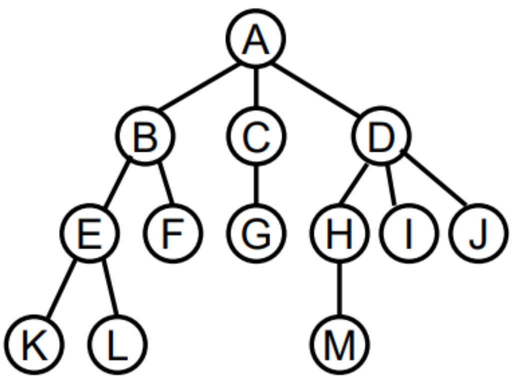
因为树的儿子数量不确定，我们如果使用链表分别表示父子之间的每个边，则需要预设每个节点都有树的度那么多个子节点，实际上往往并没有那么多子节点，会造成空间浪费

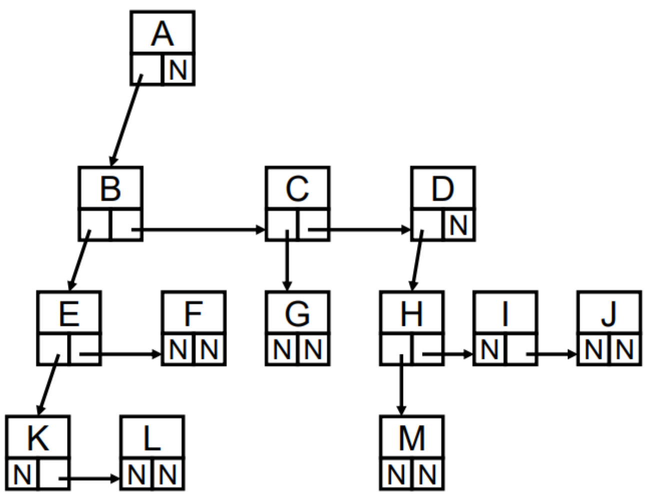




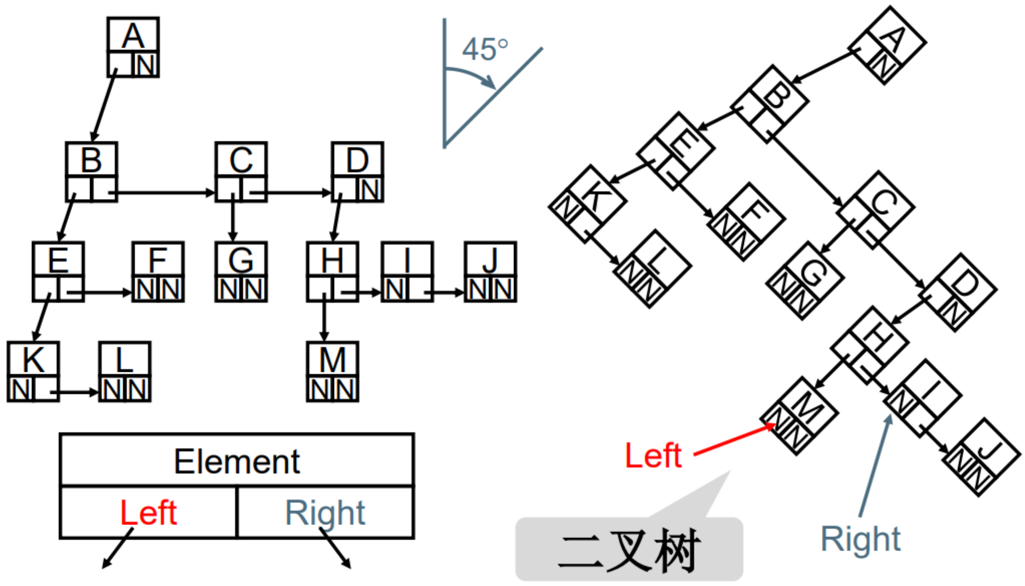
有一种方法有效解决了这个困境，就是我们给每个节点只设置两个指针，分别指向第一个儿子和下一个兄弟，可以表示为







使用这种结构还有一个好处就是，当我们将其顺时针旋转45度后，得到的必然是一个二叉树，也就是普通的树都可以通过这种方式转换为二叉树，解决了二叉树的问题就解决了大部分树的问题，所以，后面我们都将围绕二叉树展开



Part 2 二叉树及存储结构

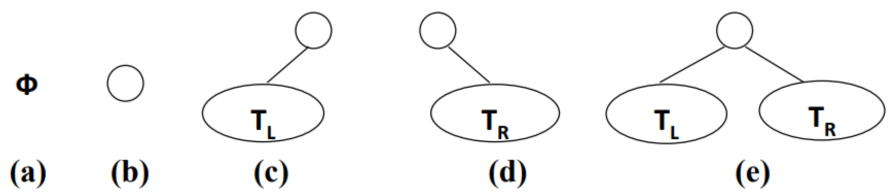
二叉树的定义

二叉树T：一个有穷的结点集合。

这个集合可以为空

若不为空，则它是由根结点和称为其左子树T_L和右子树T_R的两个不相交的二叉树组成。

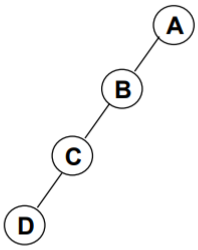
- 二叉树具体五种基本形态



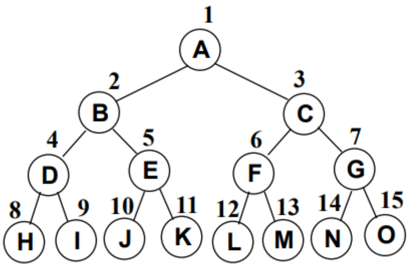
- 二叉树的子树有左右顺序之分

几种特殊二叉树

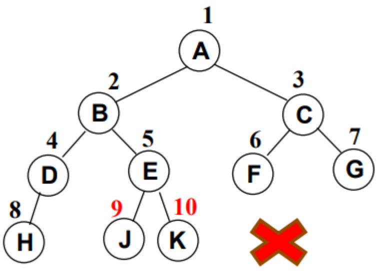
斜二叉树：退化为线性链表



完美二叉树或称满二叉树：除了叶节点外每个节点的度数都是2，深度为k时有2^k-1个节点



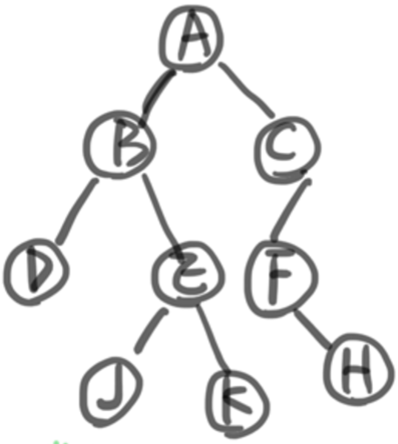
完全二叉树：有n个节点的二叉树，对树中节点按从上至下、从左到右顺序进行编号，编号为i（1 ≤ i ≤ n）结点与满二叉树 中编号为 i 结点在二叉树中位置相同，也就是只允许相对满二叉树减少最后几个节点，其他位置不能少，例如这样就不行，9应该是D的右孩子



二叉树的重要性质

- 一个二叉树第 *i* 层的最大结点数为：2^{i-1}，$i \geq 1$。
- 深度为*k*的二叉树有最大结点总数为：2^k-1，$k \geq 1$。
- 对任何非空二叉树 T，若n_0表示叶结点的个数、n_2是度为2的非叶结点个数，那么两者满足关系$n_0 = n_2 + 1$。

推算第三条性质如下



n_0：叶节点，度为0

n_1：度为1的节点

n_2：度为2的节点

(除根节点外)
每个节点必然有且仅有一条边与其父节点相连

$\Rightarrow$ 边总数 $= n_0+n_1+n_2-1$

通过 n_0, n_1, n_2 的特点

$\Rightarrow$ 边总数 $= 0\times n_0 + 1\times n_1 + 2\times n_2$

$$n_0+n_1+n_2-1 = n_1+2n_2 \Rightarrow \underline{n_0 = n_2+1}$$

二叉树的抽象数据类型定义

类型名称：二叉树

数据对象集：一个有穷的结点集合。

若不为空，则由根结点和其左、右二叉子树组成。

操作集： BT∈ BinTree, Item ∈ ElementType，重要操作有：

1、**Boolean IsEmpty(BinTree BT)**： 判别BT是否为空；

2、**void Traversal(BinTree BT)**：遍历，按某顺序访问每个结点；

3、**BinTree CreatBinTree( )**：创建一个二叉树。

常用的遍历方法有：

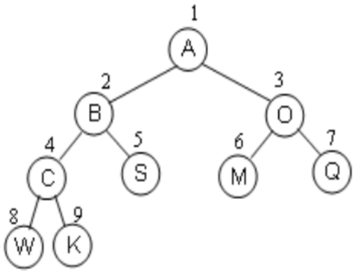
- void PreOrderTraversal(BinTree BT)：先序----根、左子树、右子树；
- void InOrderTraversal(BinTree BT)： 中序---左子树、根、右子树；
- void PostOrderTraversal(BinTree BT)：后序---左子树、右子树、根
- void LevelOrderTraversal(BinTree BT)：层次遍历，从上到下、从左到右

二叉树的存储结构

1. 顺序存储结构

完全二叉树：按从上至下、从左到右顺序存储

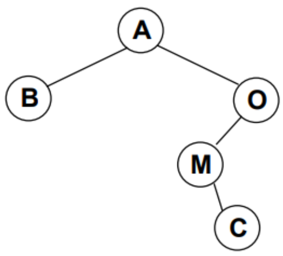
n个结点的完全二叉树的结点父子关系：



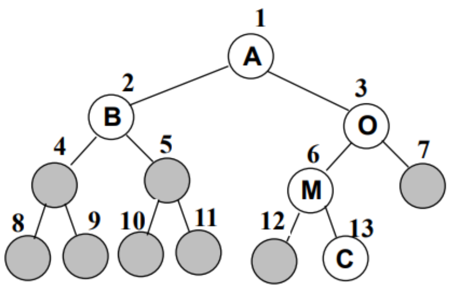
- 非根结点（序号 i > 1）的父结点的序号是 $\lfloor i / 2 \rfloor$;
- 结点（序号为 i ）的左孩子结点的序号是 2i，（若2 i <= n，否则没有左孩子）；
- 结点（序号为 i ）的右孩子结点的序号是 2i+1，（若2 i +1<= n，否则没有右孩子）；

结点	A	B	O	C	S	M	Q	W	K
序号	1	2	3	4	5	6	7	8	9

- 一般二叉树也可以采用这种结构，但会造成空间浪费……



(a)一般二叉树



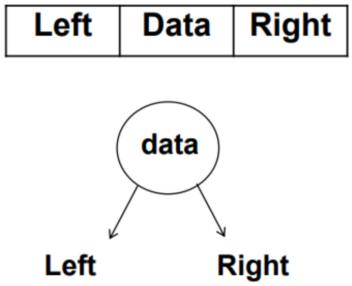
(b) 对应的完全二叉树

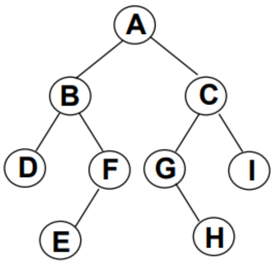
结点	A	B	O	∧	∧	M	∧	∧	∧	∧	∧	∧	C
序号	1	2	3	4	5	6	7	8	9	10	11	12	13

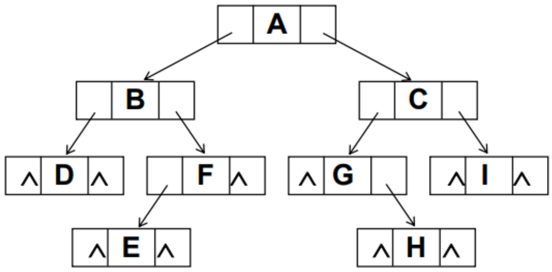
造成空间浪费！

2. 链表存储

```c
typedef struct TreeNode *BinTree;
typedef BinTree Position;
struct TreeNode{
        ElementType  Data;
        BinTree  Left;
        BinTree  Right;
}
```







Part 3 二叉树的遍历

先序，中序，后序都是根据根节点在遍历中的访问顺序而定的

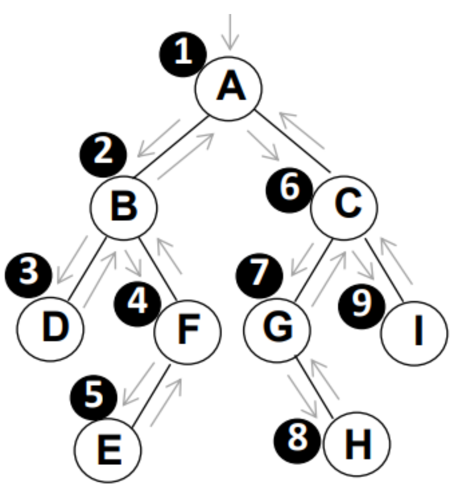
递归方法

• 先序遍历

遍历过程：**访问**根节点→先序遍历其左子树→先序遍历其右子树

A（B D F E）（C G H I）

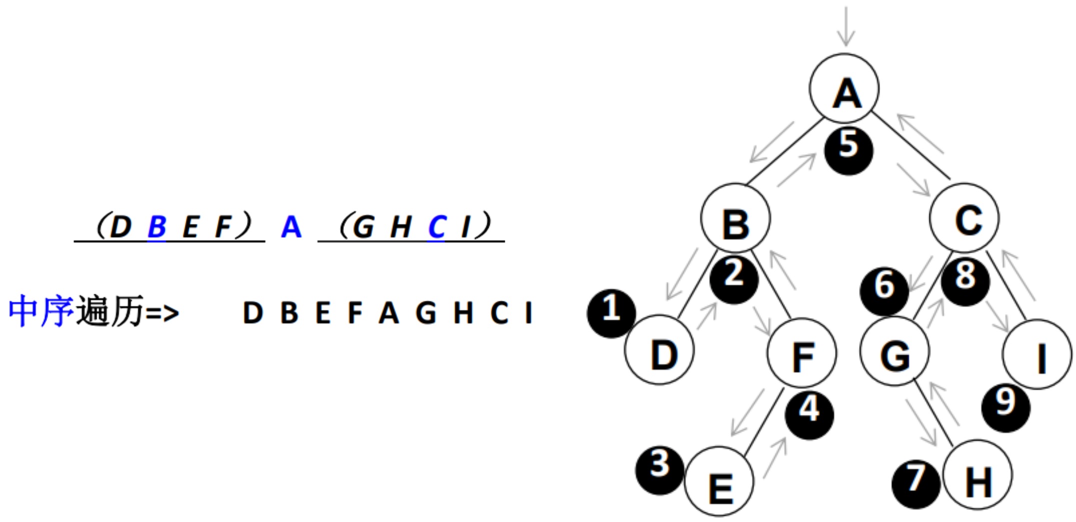
先序遍历=>　A B D F E C G H I



```
void PreOrderTraversal(BinTree BT)
{
    if(BT)
    {
        cout << BT->Data;
        PreOrderTraversal(BT->Left);
        PreOrderTraversal(BT->Right);
    }
}
```

- ## 中序遍历

遍历过程：中序遍历其左子树→**访问**根节点→中序遍历其右子树



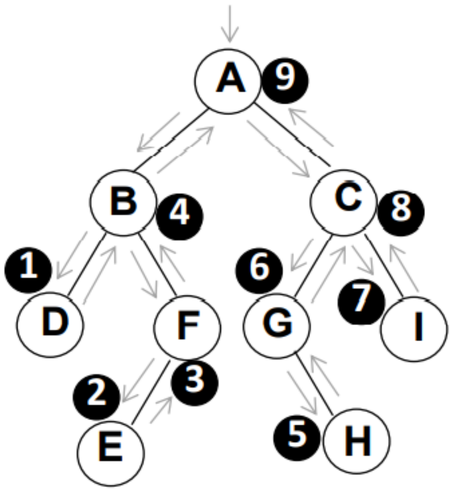

```
void InOrderTraversal(BinTree BT)
{
    if(BT)
    {
        InOrderTraversal(BT->Left);
        cout << BT->Data;
        InOrderTraversal(BT->Right);
    }
}
```

• 后序遍历

遍历过程：后序遍历其左子树➡后序遍历其右子树➡**访问**根节点

（D E F B）（H G I C）A

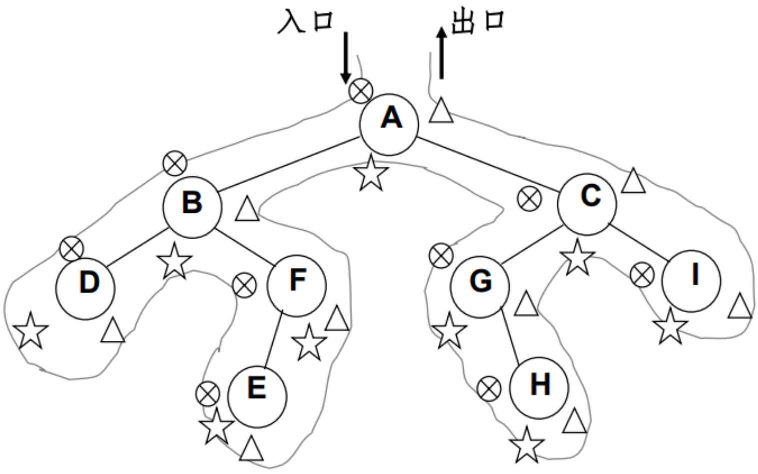
后序遍历=> D E F B H G I C A




```
void PostOrderTraversal(BinTree BT)
{
    if(BT)
    {
        PostOrderTraversal(BT->Left);
        PostOrderTraversal(BT->Right);
        cout << BT->Data;
    }
}
```

❖ 先序、中序和后序遍历过程：遍历过程中经过结点的路线一样，只是访问各结点的时机不同。

❖ 图中在从入口到出口的曲线上用⊗、☆ 和△三种符号分别标记出了先序、中序和后序访问各结点的时刻



二叉树的非递归遍历-使用堆栈

• 中序遍历

1. 遇到一个节点，就把它压栈，并遍历它的左子树
2. 当左子树遍历结束后，从栈顶弹出这个节点并访问它
3. 按其右指针再去中序遍历该节点的右子树

```
void InOrderTraversal(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreatStack(MaxSize);  //初始化堆栈S
    while(T || !IsEmpty(S))
    {
        while(T)    //一直向左并将沿途节点压入堆栈，直到最左边的叶子处停止
        {
            Push(S, T);
            T = T->Left;
        }
        if(!IsEmpty(S))
        {
            T = Pop(S); //节点弹出堆栈
```

```
                cout << T->Data;     //访问该节点
                T = T->Right;
            }
        }
}
```

前序遍历

1. 根节点入栈
2. 只要栈不为空
   1. 弹栈并访问
   2. 弹出节点的右儿子不为空则入栈
   3. 弹出节点的左儿子不为空则入栈

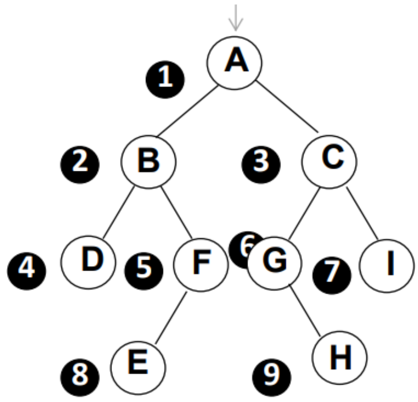
```
void PreOrderTraversal(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreatStack(MaxSize);
   if(T)    Push(S, T);
    while(!IsEmpty(S))
    {
        T = Pop(S);
        cout << T->Data;
        if(T->Right)        Push(S, T->Right);   //注意先右后左
        if(T->Left)         Push(S, T->Left);
    }
}
```

层次遍历

二叉树遍历的核心问题是**将二维结构（树）线性化（序列）**

A B C D F G I E H

层序遍历 => A B C D F G I E H



二叉树遍历的方式是通过节点访问其左右儿子节点，如果我们选择访问其左儿子，那么当前节点以及右儿子该怎么办？我们需要记住将访问节点的父节点或其兄弟，以便后来返回，可以访问其他未访问节点，这就需要使用队列或堆栈来存储，前面的三种遍历，我们都使用了堆栈，下面的层次遍历使用队列将更好理解

根节点入队→从队列中取出一个元素→访问其对应节点→将该节点非空的左右孩子节点入队

```
void LevelOrderTraversal(BinTree BT)
{
    Queue Q;
    BinTree T;
    if(!BT) return; //空树直接返回
    Q = CreatQueue(MaxSize);    //创建并初始化队列Q
    Add(Q, BT); //1.根节点入队
    while(!IsEmptyQ(Q))
    {
        T = DeleteQ(Q); //2.从队列中取出一个元素
        cout << T->Data;//3.访问其对应的节点
        //4. 将其非空的左右节点入队
        if(T->Left)     AddQ(Q, T->Left);
        if(T->Right)    AddQ(Q, T->Right);
    }
}
```

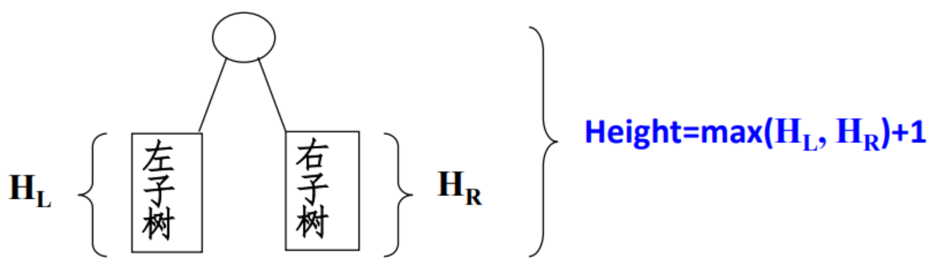
遍历二叉树的应用

1. 输出二叉树中的叶子节点

在二叉树的遍历算法中增加检测节点的 **左右子树是否为空** 即可，例如用前序遍历实现

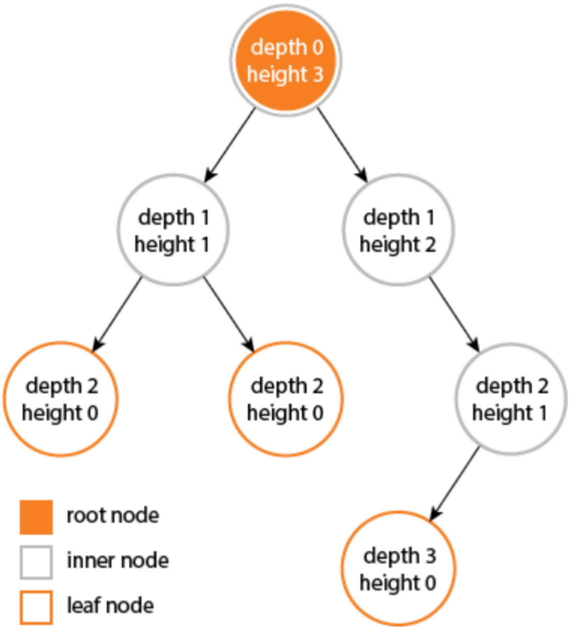
```
void PreOrderPrintLeaves(BinTree BT)
{
    if(BT)
    {
        if(!BT->Left && !BT->Right)
            cout << BT->Data;
        PreOrderPrintLeaves(BT->Left);
        PreOrderPrintLeaves(BT->Right);
    }
}
```

2. 求二叉树高度

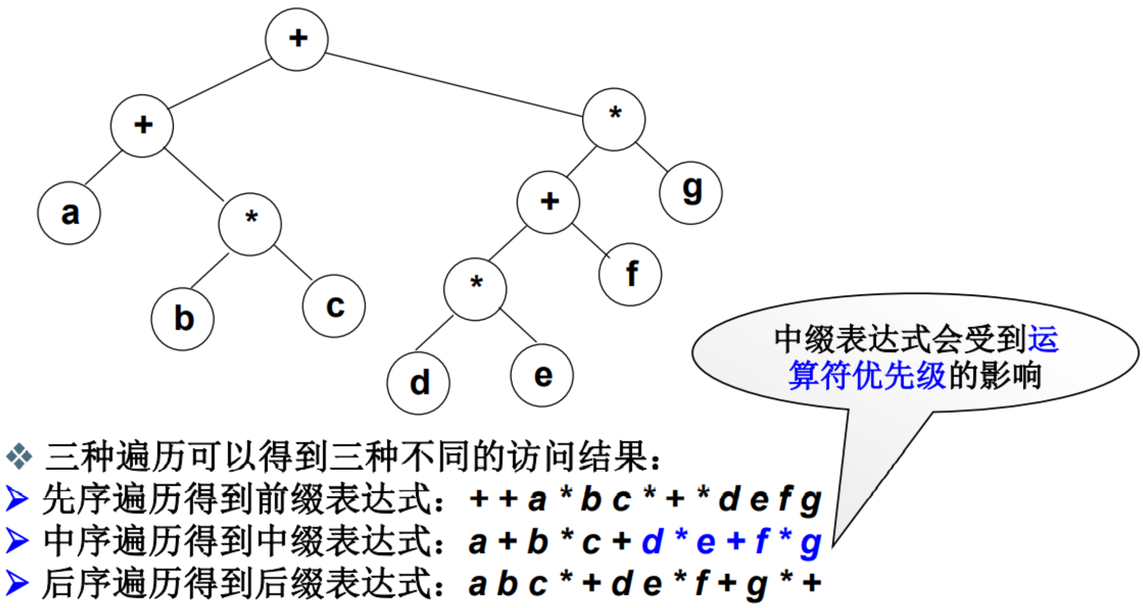



```
int PostOrderGetHeight(BinTree BT)
{
    int HL, HR, MaxH;
    if(BT)
    {
        HL = PostOrderGetHeight(BT->Left);   //求左子树高度
        HR = PostOrderGetHeight(BT->Right); //求右子树高度
        return max(HL, HR) + 1; //返回树的高度
    }
    else
        return 0;    //空树高度为0
}
```

节点的深度和高度的关系参照下面这张图（不同书籍定义可能不同），越靠上高度越大深度越小，越靠下高度越小深度越大，对于树的高度或深度等同于整棵树的最大层次数

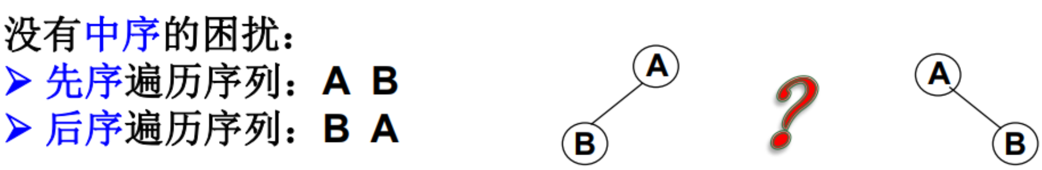


3. 二元运算表达式树及其遍历

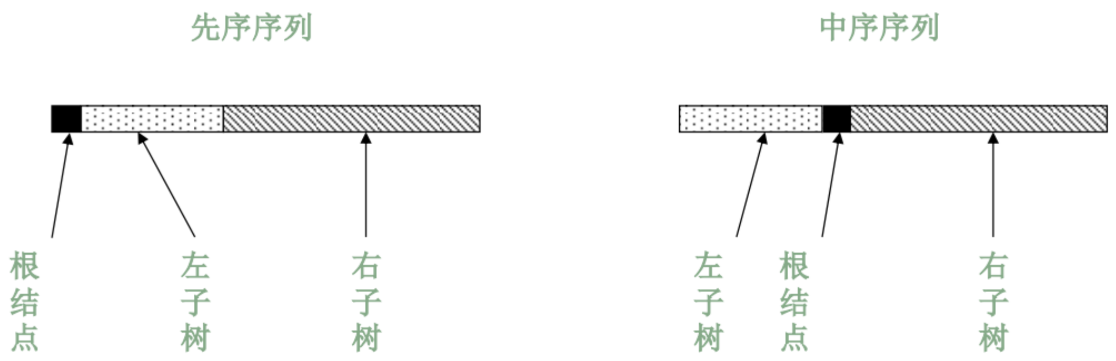


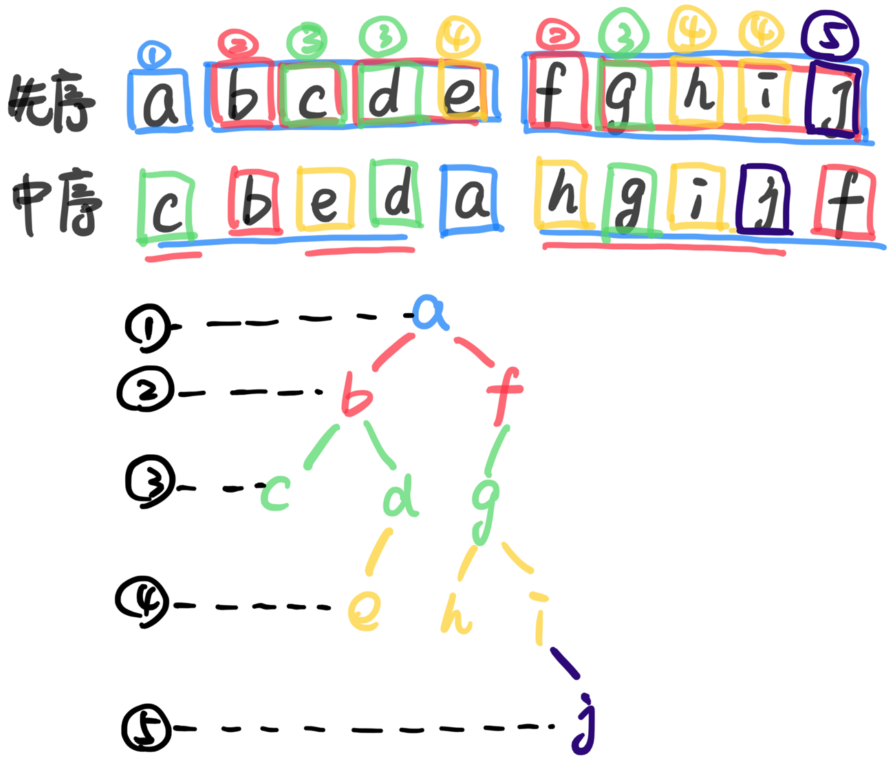
注意这里的中缀表达式因为运算符优先级的原因，直接计算的结果不一定准确，需要在遍历过程中加入括号

4. 由中序+前/后序遍历序列确定二叉树



1. 根据先序遍历序列第一个节点**确定根节点**
2. 根据根节点在中序遍历序列的位置， **确定左右子树**
3. 分别对左右子树重复以上两步

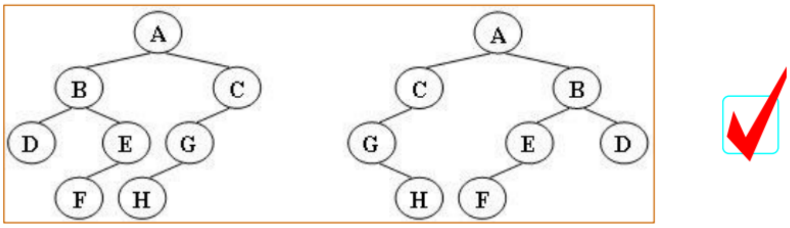


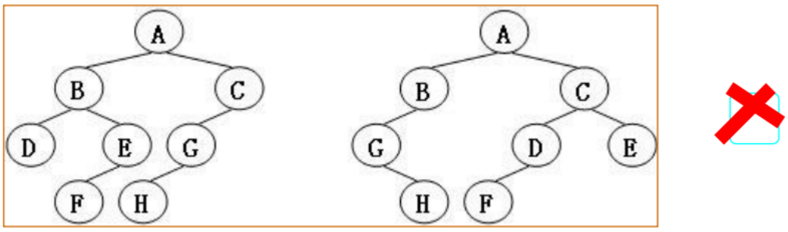


Part 4 树的同构

题意

给定两棵树T1和T2。如果T1可以通过若干次左右孩子互换 就变成T2，则我们称两棵树是“同构”的。 现给定两棵树，请你判断它们是否是同构的



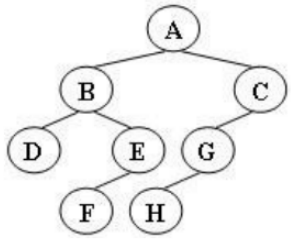


输入格式： 输入给出2棵二叉树的信息：

- 先在一行中给出该树的结点数，随后N行
- 第i行对应编号第i个结点，给出该结点中存储的字母、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“-”。

输入样例：

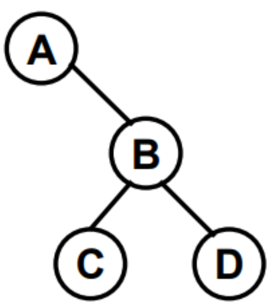
8
0 **A 1 2**
1 **B 3 4**
2 **C 5 -**
3 **D - -**
4 **E 6 -**
5 **G 7 -**
6 **F - -**
7 **H - -**
8
G - 4
B 7 6
F - -
A 5 1
H - -
C 0 -
D - -
E 2 -



求解

1. 二叉树表示

用结构数组表示二叉树，即静态链表法



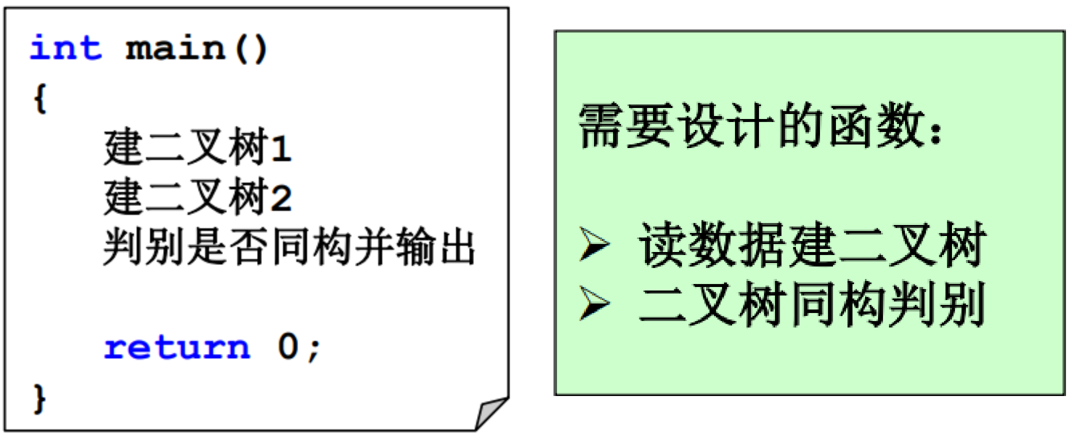
	A	B	C	D	
Left	-1	2	-1	-1	
Right	1	3	-1	-1	
	0	**1**	**2**	**3**	**4**

```c
#define MaxTree 10
#define ElementType char
#define Tree int
#define Null -1

struct TreeNode
{
    ElementType Element;
    Tree Left;
    Tree Right;
}T1[MaxTree], T2[MaxTree];
```

这种表示方式我们无法直接得知根节点的索引，一种方法是，我们观察所有节点的左右儿子索引，只有根节点不是任何节点的左右儿子，因此，Left和Right中都没有出现过的那个索引就指向根节点

- **2. 程序框架搭建**



```
int main()
{
    Tree R1, R2;

    R1 = BuildTree(T1);
    R2 = BuildTree(T2);

    if(Isomorphic(R1, R2))  cout << "Yes" ;
    else                    cout << "No" ;
    return 0;
}
```

• 3. 建立二叉树

第一行读入节点个数；接下来以字符形式读入每个节点的元素值以及左节点和右节点，并分不同情况处理，出现在左右儿子中的节点在 `check` 数组中设为 `1`；最后检测 `check` 中为零的节点就是根节点，将其返回

```
#include<iostream>
using namespace std;
int N;
Tree BuildTree(struct TreeNode T[])
{
    int Root = Null;
    int check[MaxTree];
    cin >> N;
    if(N)
    {
        for (int i = 0; i < N; i++)
            check[i] = 0;
        char cl, cr;
        for(int i = 0; i < N; i++)
        {
            cin >> T[i].Element >> cl >> cr;
            //处理左孩子
            if(cl != '-')
            {
                T[i].Left = cl - '0';
                check[T[i].Left] = 1;
            }
```

```
            else    T[i].Left = Null;
            //处理右孩子
            if(cr != '-')
            {
                T[i].Right = cr - '0';
                check[T[i].Right] = 1;
            }
            else    T[i].Right = Null;
        }
        //查找根节点
        for(int i = 0; i < N; i++)
            if(!check[i])
            {
                Root = i;
                break;
            }
    }
    return Root;
}
```

- **4. 判别两个二叉树是否同构**

```
bool Isomorphic(Tree R1, Tree R2)
{
    if(R1 == Null && R2 == Null)    return 1;    //全为空，默认同构
    if(R1 == Null && R2 != Null || R1 != Null && R2 == Null)
        return 0;    //一个空一个不空，不可能同构
    if(T1[R1].Element != T2[R2].Element)
        return 0;    //树根的值不同，不可能同构
    if(T1[R1].Left == Null && T2[R2].Left == Null)   //左子树都为空
        return Isomorphic(T1[R1].Right, T2[R2].Right);   //判断右子树是否同构
    if(T1[R1].Left != Null && T2[R2].Left != Null && T1[T1[R1].Left].Element == T2[T2[R2].Left].Element)    //两树的左子树都不为空且左子树元素相同
        return (Isomorphic(T1[R1].Left, T2[R2].Left) && Isomorphic(T1[R1].Right, T2[R2].Right));    //判断两树的左子树是否同构，右子树是否同构
    else
```

```
        return (Isomorphic(T1[R1].Left, T2[R2].Right) &&
    Isomorphic(T1[R1].Right, T2[R2].Left));    //一个树的左子树，右子
    树分别是否和另一个树的右子树，左子树同构
}
```

整合各部分代码即可AC此题

测试点	提示	结果	分数
0	sample 1 有双边换、单边换，节点编号不同但数据同	答案正确	7
1	sample 2 第3层开始错，每层结点数据对，但父结点不对	答案正确	7
2	结点数不同	答案正确	3
3	空树	答案正确	2
4	只有1个结点，结构同但数据不同	答案正确	3
5	最大N，层序遍历结果相同，但树不同	答案正确	3

编程练习

1. 树的同构，见Part4

2. List Leaves

3. Tree Traversals Again