作业 HW3 试验报告

姓名: 范潇　学号: 2254298

2023 年 11 月 12 日

1　涉及数据结构和相关背景

本次实验主要涉及到的数据结构为树和二叉树。

树是由 $n(n \geq 0)$ 个结点组成的有限集合。如果 $n = 0$，称为空树；如果 $n > 0$，则有一个特定的称为根的结点，它只有后继，没有前驱。除根以外的其他结点划分为 $m(m \geq 0)$ 个互不相交的有限集合 $T_0, T_1, \cdots, T_{m-1}$，每个集合本身又是一棵树，并且称之为根的子树。每棵子树的根节点有且仅有一个直接前驱，但可以有 0 个或多个后继。

一颗二叉树则是结点的一个有限集合，该集合或者为空，或者是由一个根节点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

二叉树的物理存储结构主要分为顺序存储结构和链式存储结构。前者是借助数组来实现，缺点是会造成存储空间的浪费，后者又可以进一步分为二叉链表和三叉链表。

二叉树的生成和遍历十分重要。二叉树的遍历方式可以分为先序，中序和后序。给定二叉树的先序遍历序列和中序遍历序列或者给定后序遍历序列和中序遍历序列，都可以还原出另外一个未给定的遍历序列。

二叉树相关的算法一般用递归实现。这样实现的算法通常较为简洁。但是当应用环境资源受限，或者有特殊需求时，则采用栈来模拟递归程序。

树的存储结构可以分为定长和不定长结构。一般采用定长结构，和不定长结构相比，指针的利用率不高，但是算法更加简单。当定长结构未二叉树时，指针利用效率最高。树和二叉树之间可以进行相互转换。当从树转换至二叉树时，应该遵循“左孩子，右兄弟”的规则来进行转换。

2 实验内容

2.1 二叉树的非递归遍历

2.1.1 问题描述

问题描述：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。

2.1.2 基本要求

输入要求：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)

输出要求：一行，后序遍历序列。

2.1.3 数据结构设计

```
typedef char ElemType;
typedef struct Node{
   ElemType data;
   struct Node *lchild,*rchild;
}Node,*Tree;
```

2.1.4 功能说明

Algorithm 1 递归后序遍历

输入: 待遍历树 Tree T, 访问函数 visit

输出: 后序遍历输出

1: **function** PosOrderTraverse(Tree T,Status(*visit)(Tree))
2: **if** 树是空树 **then**
3: 树为空
4: **return** OK
5: **end if**

6: 访问左子树
7: 访问右子树
8: 访问根节点
9: **return** OK
10: **end function**

```
//根据二叉树中序遍历的栈的操作序列还原二叉树
int main()
{
    int n,flag=0;//用于判断是否已经读入了push
    char opt[MAXOPTSIZE];//存储操作指令
    ElemType val ;//存储输入数据
    scanf("%d",&n);//获取待还原结点个数
    std::stack<Tree> s;
    Tree p=(Tree)malloc(sizeof(Node)),T = p,temp;//工作指针初始化
    if(!T)
      OVERFLOW;
    while(n){
        while(flag||((scanf("%s",opt)==1&&!strcmp(opt,"push")))){
            flag =0;
            p ->lchild =(Tree)malloc(sizeof(Node));
            p ->rchild =(Tree)malloc(sizeof(Node));
            scanf("␣%c",&val);
            p->data = val;
            s.push(p);
            p = p->lchild;//沿左链下降
        }
        //此时读入了一个pop,应该释放左节点,同时修改栈顶结点的左节点为NULL
        free(p);
        p = s.top();//回溯
        s.pop();
        p->lchild = NULL;
        n--;
        while(n&&(scanf("%s",opt)==1&&!strcmp(opt,"pop"))){
            //每循环一次说明有一个右孩子结点为NULL
            free(p->rchild);
            p->rchild = NULL;
            p = s.top();
            s.pop();
            n--;
        }
```

```
36          //此时读入了一个push
37          flag = 1;//用于判断是否已经读入了一个push
38          if(!n){//处理边际情况
39              free(p->rchild);
40              p->rchild = NULL;
41          }
42          p = p->rchild;
43      }
44      PosOrderTraverse(T,visit);
45      return 0;
46  }
```

2.1.5 调试分析

调试时采用输出日志的方式来辅助查找出错的位置。在遇到内存泄漏的问题时，则使用 VS 调试模式中的自动变量窗口来查看哪里出现泄露，并根据泄露位置倒推原因。

2.1.6 总结与体会

本题的易错点之一是读入存储的值时，应该使用" %c"，而非"%c"。另外一个易错点，同时也是难点的是内存的管理。压入栈的结点的左右子树不应该置 NULL，否则最后所形成的二叉树只是一个个结点。为此，在创建一个新结点的同时，便要为其左右子树申请内存空间，然后根据后续的栈操作序列来判断是否需要释放。释放完成后还需注意要进行置 NULL。

2.2 二叉树的同构

2.2.1 问题描述

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。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。

2.2.2 基本要求

输入要求：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。

输出要求：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。

2.2.3 数据结构设计

```
typedef struct {
    char data;
    int lchild;
    int rchild;
}ElemType;
typedef struct {
    ElemType* elem;
    int root;
} Tree;
```

2.2.4 功能说明

Algorithm 2 判断两棵树是否同构

输入： 树 T1,T2, 根节点 root1,root2

输出： 是否相同的判断

```
1: function VALID(Tree T1,int root1,Tree T2,int root2)
2:     if 两个根节点均为空 then
3:         return FALSE
4:     end if
5:     if 两个根节点一个为空，另一个不为空 then
6:         return FALSE
```

```
    end if
    if 两个根节点中的数据不一样 then
        return FALSE
    end if
    return (valid(T1, T1.elem[root1].lchild, T2, T2.elem[root2].lchild)
     && valid(T1, T1.elem[root1].rchild, T2, T2.elem[root2].rchild)
     || valid(T1, T1.elem[root1].lchild, T2, T2.elem[root2].rchild)
     && valid(T1, T1.elem[root1].rchild, T2, T2.elem[root2].lchild));
end function
```

Algorithm 3 非递归求二叉树深度

输入: 树 T

输出: 深度 Depth

```
function DEPTH(Tree T)
    初始化分别用于存储结点以及对应结点深度的栈 s1,s2
    初始化工作指针 p，指向树 T 的根节点
    初始化记录深度的变量 depth 和 maxdepth 为 1
    while s1 不为空且工作指针不为空 do
        while 工作指针不为空 do
            s1.push(p)
            p 更新为其左孩子结点
            s2.push(depth)
            depth++
        end while
        s1.pop(p)
        p 更新为右孩子结点
        if p 不为空 then
            depth=s2.top()+1
        else
            depth= s2.top
        end if
        maxdepth = max(depth,maxdepth)
    end while
    return maxdepth
```

22: **end function**

2.2.5 调试分析

在本题的调试上我花费了很长时间。一开始 OJ 的反馈是能够通过前 4 个样例，而后 6 个样例是 TLE。再优化了初始化树的算法以及递归算法后，仍然没有任何改变，有些时候后面 6 个样例变成了 MLE。后来我下载了数据生成文件后，发现运行时判断的结果很快就能给出，而深度的结果虽然正确，但是要等待相当长一端时间，因此判断是由于求深度时采用了递归算法。修改为非递归算法后，发现 OJ 仍反馈错误，经过排查后发现是因为读入左右孩子结点时使用的是 getchar，当结点个数超过一位数时，便会产生错误。后来采用了 scanf 和 atoi 配合进行数据读入。

2.2.6 总结与体会

易错点之一是读取数据是的函数选择，数据域始终为 1 个字母，可以选择用 getchar，而左右孩子结点可能是一个字符 '-'，也可能是一位或多位数字，需要多加注意。另一个易错点是求深度的方法选择，由于深度可能高达 30 层左右，如果选择递归方式的话将肯定超时，应该选用非递归方式。难点之一是用于判断同构函数的编写，需要覆盖所有的可能，并且设置好中止条件。

2.3 感染二叉树需要的时间

2.3.1 问题描述

问题描述：给你一棵二叉树的根节点 root ，二叉树中节点的值互不相同。另给你一个整数 start 。在第 0 分钟，感染将会从值为 start 的节点开始爆发。每分钟，如果节点满足以下全部条件，就会被感染：

1. 节点此前还没有感染。
2. 节点与一个已感染节点相邻。

返回感染整棵树需要的分钟数。

2.3.2 基本要求

输入要求：第一行包含两个整数 n 和 start。接下来包含 n 行，描述 n 个节点的左、右孩子编号。0 号节点为根节点。

输出要求：一个整数，表示感染整棵二叉树所需要的时间。

2.3.3 数据结构设计

```
typedef struct{//三叉
    int val;//为真代表还未被感染
    int lchild;//左孩子结点
    int rchild;//右孩子结点
    int parent;//双亲结点
}Node;
typedef Node Tree[MAXN];//用静态数组实现
```

2.3.4 功能说明

```
//初始化树
scanf("%d%d",&n,&start);
for(int i =0;i<n;i++){
    T[i].val = 1;//未被感染
    scanf("%d%d",&T[i].lchild,&T[i].rchild);//设置左右孩子结点
    T[T[i].lchild].parent= i;//设置左孩子结点的双亲结点
    T[T[i].rchild].parent= i;//设置右孩子结点的双亲结点
}
T[start].val = 0;//感染起始点
T[0].parent = -1;//将根节点的双亲结点置空
```

Algorithm 4 感染

输入: 感染起始处 start, 待感染树 T

输出: 感染所需时间 count

1: **function** INFECT(int start,Tree T)

2: 初始化队列 p,q

```
3:     将 start 压入 p 中
4:     while 1 do
5:         while p 不为空 do
6:             p 出队
7:             if 双亲结点不为空，且未被感染 then
8:                 将双亲结点标记为感染，并进入 q 中
9:             end if
10:            if 左孩子结点不为空，且未被感染 then
11:                将左孩子结点标记为感染，并进入 q 中
12:            end if
13:            if 右孩子结点不为空，且未被感染 then
14:                将右孩子结点标记为感染，并进入 q 中
15:            end if
16:        end while
17:        将 q 中的元素依次出队，并进入 p 中
18:        if p 为空 then
19:            break
20:        end if
21:        count++
22:    end while
23:    return count
24: end function
```

2.3.5 调试分析

采用输出中间变量的方式进行调试，关键的中间变量为 count 的不同值之间进入 q 的结点的序号。

2.3.6 总结与体会

在处理“不重复地从某一位置像周围扩展”这一类问题时，可以采用队列配合辅助标记的方式来解决。

2.4 树的重构

2.4.1 问题描述

问题描述：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

2.4.2 基本要求

输入要求：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)。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。

输出要求：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。

2.4.3 数据结构设计

```
1 char input[MAXINFOLENGTH]
```

2.4.4 功能说明

Algorithm 5 根据深度优先遍历序列求顺序树深度

输入: 深度优先遍历序列 input

输出: 顺序树深度

```
1: function SqDepth(char input[])
2:     工作指针 p 指向 input 起始位置
3:     SqDepth，SqMaxDepth 置零
4:     while p 不指向尾零 do
5:         if p 指向 d then
6:             SqDepth++
7:         else
8:             SqDepth- -
9:         end if
10:        SqMaxDepth = max(SqDepth,SqMaxDepth)
11:        工作指针 p 后移
12:    end while
13:    return SqMaxDepth
14: end function
```

Algorithm 6 根据深度优先遍历序列求二叉树深度

输入: 深度优先遍历序列 input

输出: 顺序树深度

```
1: function BiDepth(char input[])
2:     工作指针 p 指向 input 起始位置
3:     BiDepth，BiMaxDepth 置零
4:     初始化辅助栈 s，并将 0 压入栈中
5:     while p 不指向尾零 do
6:         if p 指向 d then
7:             BiDepth++
8:             s.push(BiDepth)
9:         else
10:            s.pop(BiDepth)
11:        end if
12:        BiMaxDepth = max(BiDepth,BiMaxDepth)
```

```
13:         工作指针 p 后移
14:     end while
15:     return SqMaxDepth
16: end function
```

2.4.5 调试分析

采用输出关键中间变量的方式来调试，关键的中间变量是当前深度优先遍历序列中的各个位置所对应的深度。

2.4.6 总结与体会

一开始我设计的算法是先根据深度优先遍历序列把顺序树给构造出来，然后使用递归来求对应的二叉树的深度。但是这种方法不仅编写时较为复杂，容易出错，而且时间复杂度较大。

在编写与遍历序列相关的树的程序时，要利用好遍历序列中蕴含的信息，以便编写出原地算法等复杂度较低的算法。

2.5 最近公共祖先

2.5.1 问题描述

问题描述：给出一颗多叉树，请你求出两个节点的最近公共祖先。一个节点的祖先节点可以是该节点本身，树中任意两个节点都至少有一个共同祖先，即根节点。

2.5.2 基本要求

输入要求：输入数据包含 T 个测试样本，每个样本 i 包含 Ni 个节点和 Ni-1 条边和 Mi 个问题，树中节点从 1 到 Ni 编号输入第一行是测试样本数 T。每个测试样本 i 第一行为两个整数 Ni 和 Mi。接下来 Ni-1 行，每行 2 个整数 a、b，表示 a 是 b 的父节点。接下来 Mi 行，每行两个整数 x、y，表示询问 x 和 y 的共同祖先。

输出要求：对于每一个询问输出一个整数，表示共同祖先的编号

2.5.3 数据结构设计

```
1 typedef int Tree[MAXN];//静态数组实现，存放的是对应序号的结点的父节点
```

2.5.4 功能说明

Algorithm 7 search

输入： 树 T，询问序列，询问序列个数 N

输出： 最近公共祖先编号

```
1: function SEARCH(Tree T,int N)
2:     while 还有未处理的询问 do
3:         获取带查询结点 x，y
4:         初始化辅助栈 s1，s2
5:         x 压入 s1
6:         y 压入 s2
7:         while x 的父节点非空 do
8:             x 更新为其父节点
9:             x 压入 s1 中
10:        end while
11:        while y 的父节点非空 do
12:            y 更新为其父节点
13:            y 压入 s2 中
14:        end while
15:        while s1，s2 均非空，且栈顶元素相同 do
16:            ans 更新为栈顶元素
17:            s1 出栈
18:            s2 出栈
19:        end while
20:        打印 ans
21:    end while
22:    return OK
23: end function
```

2.5.5 调试分析

通过输出中间变量来辅助查询漏洞，并且结合提供的输入输出来倒退漏洞。关键的中间变量为进出辅助栈的元素值。

2.5.6 总结与体会

该题的一个易错点在于每个测试样本中的关系的个数不是 N，而是 N-1 。同时，由于涉及到多个测试样本的处理，要注意样本前后之间不能相互影响，即当切换到新样本时，相关的变量应该要初始化。

2.6 求树的后序

2.6.1 问题描述

问题描述：给出二叉树的前序遍历和中序遍历，求树的后序遍历。

2.6.2 基本要求

输入要求：输入包含若干行，每一行有两个字符串，中间用空格隔开。同行的两个字符串从左到右分别表示树的前序遍历和中序遍历，由单个字符组成，每个字符表示一个节点。字符仅包括大小写英文字母和数字，最多 62 个。输入保证一颗二叉树内不存在相同的节点。

输出要求：每一行输入对应一行输出。若给出的前序遍历和中序遍历对应存在一棵二叉树，则输出其后序遍历，否则输出 Error

2.6.3 数据结构设计

```
typedef struct Node{
    char data;//数据域
    Node* lchild;//左孩子结点
    Node* rchild;//右孩子结点
}*Tree,Node;
```

2.6.4 功能说明

Algorithm 8 重构二叉树

输入: 前序遍历序列 pre，中序遍历序列 in，待重构树 T

输出: 重构后的树 T

```
1: function Pos(string pre,string in,Tree &T)
2:     if 前序遍历序列和中序遍历序列的长度不相等 then
3:         return ERROR
4:     end if
5:     if 前序遍历序列和中序遍历序列均为空 then
6:         将树 T 置为空
7:         return OK
8:     end if
9:     在中序遍历序列中寻找前序遍历序列的第一个结点，即根节点 root，
   并置于 T 的数据域中
10:    return Pos(pre.substr(1,root),in.substr(0,root),T->lchild)
         *Pos(pre.substr(root+1),in.substr(root+1),T->rchild)
11: end function
```

2.6.5 调试分析

调试时出现的问题集中在递归时子序列的起止点的计算上。我通过 VS 调试模式中的临时变量窗口来辅助修改。

2.6.6 总结与体会

给定特定的遍历方式的遍历序列，我们可以从中得到树的结构等其他有用的信息，这一点在其他题目中也有体现。

2.7 表达式树

2.7.1 问题描述

问题描述：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。

2.7.2 基本要求

输入要求：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。

输出要求：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。

2.7.3 数据结构设计

```
typedef struct Node{
    char ch;//存放变量名的数据域
    int data;//存放变量值的数据域
    Node* lchild;
    Node* rchild;
}Node,*Tree;
```

2.7.4 功能说明

Algorithm 9 中缀表达式转后缀表达式

输入： 中缀表达式 in，待转换的后缀表达式 pos

输出： 转换后的后缀表达式 pos

```
1: function IN__TO__POS(string in,string &pos)
2:     while 遍历中缀表达式 do
3:         if 当前字符为运算符 then
4:             while 运算符栈非空且栈顶运算符优先级更高 do
5:                 if 栈顶元素不为左括号 then
6:                     栈顶元素接到 pos 尾部
7:                 end if
8:                 栈顶元素出栈
9:             end while
10:            if 当前字符不为右括号 then
11:                当前字符入栈
12:            end if
13:        else
14:            当前字符接到 pos 尾部
15:        end if
16:    end while
17: end function
```

Algorithm 10 根据后缀表达式生成表达式树

输入: 后缀表达式 pos，变量值映射关系 map<char,int>val

输出: 表达式树 T

```
1: function MAKE__TREE(string pos,map<char,int>val)
2:     初始化用于存储结点的辅助栈
3:     while 遍历后缀表达式 pos do
4:         初始化新结点
5:         if 当前字符是运算符 then
6:             弹出结点栈中的两个结点，依次赋给新结点的左右孩子结点
7:             当前字符赋给新结点的变量域
8:             将左右孩子的变量值域中的值分别作为左右运算数，执行当
   前字符对应的运算并赋给新结点的变量值域
9:             将新结点压入栈中
10:        else
11:            将读入的变量赋给新结点的变量域
12:            将新结点的左右孩子结点置空
```

```
            根据 map 将读入变量对应的变量值赋给新结点的变量值域
            将新结点压入栈中
        end if
    end while
    return 栈顶元素
end function
```

Algorithm 11 打印表达式树

输入: 表达式树 T

```
function DISPLAY(Tree T)
    求树的深度 depth
    for(int i = 0;i<depth-1;i++);offset = 2*offset+1;
    初始化队列 to_display,temp
    T 进入 to_display
    while 1 do
        输出 offset 个空格
        初始化用于判断是否结束的 flag，置 1
        while to_display 非空 do
            if 队首元素不为空 then
                to_display 队首元素出队，并将其左右子树进入 temp
                flag 置 0
            else
                将两个空指针进入 temp
            end if
            打印队首元素中的字符，若为空则为空格，后跟 2*offset+1 个空格
        end while
        if flag 为 1 then
            return OK
        end if
        输出回车后跟 offet-1 个空格
        初始化计数器 c=1
        while temp 非空 do
            取 temp 队首结点后将其出队，然后再进入 to_display
```

```
25:             if 该结点非空 then
26:                 if c 为偶数 then
27:                     打印 /
28:                 else
29:                     打印 \ 后跟 offset*2-1 个空格
30:                 end if
31:             else
32:                 if c 为偶数 then
33:                     打印三个空格
34:                 else
35:                     打印 offset*2-1 个空格
36:                 end if
37:                 c++
38:             end if
39:         end while
40:         offset=(offset-1)/2
41:     end while
42: end function
```

2.7.5 调试分析

该程序要实现的功能较为复杂，但是可以拆分成“求后缀表达式”，“求值”，“构造表达式树”，“打印表达式树”这四大部分。调试程序时，各个部分分别调试。

2.7.6 总结与体会

当程序要实现的目标较为复杂时，常常会涉及到多种数据结构之间的相互组合。例如在本题中，打印表达式树时采用了层次遍历，这就需要队列来辅助。因此在学习数据结构的时候，不仅仅要掌握其结构，也要熟悉其应用场景。

3 实验总结

与树相关的问题常常采用递归的方式来解决。在设计递归函数的时候，重点在于终止条件的设置，以及确保程序逐步“靠近”中止条件。对于和树

相关的递归函数，常常使用树为空来作为终止条件，这样通常相比于使用条件语句来判断是否继续递归要来得更加简洁。但是，递归函数也不是万能的，由于递归的特性，其时间复杂度高，在设计递归函数时，要避免在一次递归中调用多次自身，因为这会使得大幅度增加时间复杂度。同时，如果数据的规模较大，那么也应考虑使用栈辅助来模拟递归程序。

此外，二叉树以及树的结点设计对于顺利解决问题至关重要。例如，在感染二叉树需要的时间中，采用三叉链表便可以通过已被的结点轻松找到下一个时刻被感染的结点；在最近公共祖先这道题中，采用双亲表示法的设计和题目的目标相匹配，使得最终的程序较为简单。

在解决和树相关的问题时，常常需要栈和队列来进行辅助，应该要抓住所要相关元素在特定遍历方法下的特性，即是 FIFO 还是 FILO 来选取相应的辅助数据结构。