/\*功能介绍：

1.首先可以通过输入端选择输入方式，1为括号表示法，2位双亲表示法，3位前序 + 层次表示法, 然后输入相应的树的表达方式；

2.将输入数据读入到inputTree数组(通过convert函数),inputTree数组存有data与parent两个成员，并得到结点数n与最高层次m；

3.建树，将inputTree数组转化为链式表达（通过creatTreeFromParent函数）,每个结点含有data, child的指针数组,指向parent的指针以及孩子个数num与层次lev；

4.通过相应的函数进行前序,后序与层次遍历(前序与后序有递归与非递归两种方法);

5.输出这棵树的结点总个数，叶子结点的情况(通过get\_leaf\_node函数)与这棵树的最高层次(通过get\_level函数)

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// 树的次树：最大MAXM次，树最多有MAXN个结点

#define MAXN 100

#define MAXM 10

typedef struct nodeParent

{

char data;

int parent;

}NODE\_PARENT;

typedef struct linkNode

{

char data; //节点数据

struct linkNode \*child[MAXM]; //子节点的地址

struct linkNode \*parent; //父节点的地址

int lev; //节点的层次(根节点为0）

int num; //子节点的个数

}LINK\_NODE;

LINK\_NODE \*addr\_NODE[MAXN]; //储存链式存储一棵树的每个结点的指针

int AddChild(LINK\_NODE \*parent,LINK\_NODE \*child,int m) //m次树中，parent下插入一个新的子结点child

{

if(parent == NULL || child == NULL || parent -> num == m) return -1; //如果插入子节点失败

parent -> child[parent -> num] = child;

parent -> num++;

child -> lev = parent -> lev + 1;

child -> parent = parent;

return 1;

}

LINK\_NODE \*Node\_initail(int m) //m次树结点初始化，根据结点类型，完成结点的初始化操作

{

LINK\_NODE \*a = (LINK\_NODE\*)malloc(sizeof(LINK\_NODE));

for(int i = 0; i < m; i++)

a -> child[i] = NULL;

a -> num = 0;

return a;

}

LINK\_NODE \*creatTreeFromParent(NODE\_PARENT inputTree[],int m,int n) //根据inputTree建树成链式存储，用一个数组addr\_NODE来记录每个结点的指针，

{ //m表示次数，n表示节点个数

int i;

LINK\_NODE \*root, \*p, \*q;

if(n < 1) return NULL; //若为空树

root = Node\_initail(m); //建立根节点

root -> data = inputTree[0].data;

root -> lev = 0;

addr\_NODE[0] = root;

for(i = 1; i < n; i++)

{

q = Node\_initail(m);

q -> data = inputTree[i].data;

addr\_NODE[i] = q;

p = addr\_NODE[inputTree[i].parent]; //找到子节点q的父节点p

if(AddChild(p, q, m) < 0)

printf("建树失败！\n");

}

return root;

}

void r\_preorder(LINK\_NODE \*t, int m) //递归前序遍历，t:根，m次数

{

int i;

if(t != NULL)

{

printf("%c ", t -> data);

for(i = 0; i < t -> num; i++)

r\_preorder(t -> child[i], m);

}

}

void r\_postorder(LINK\_NODE \*t, int m) //递归后序遍历，t:根，m次数

{

int i;

if(t != NULL)

{

for(i = 0; i < t -> num; i++)

r\_postorder(t -> child[i], m);

printf("%c ", t -> data);

}

}

void hierachicalorder(LINK\_NODE \*t, int m) //非递归层次遍历

{

LINK\_NODE \*q[MAXN], \*p;

int head = 0, tail = 1, i;

q[0] = t;

while(head < tail)

{

p = q[head++];

printf("%c ", p -> data);

for(i = 0; i < p -> num; i++)

q[tail++] = p -> child[i];

}

}

void preorderNoRecursion(LINK\_NODE \*t, int m) //递归前序遍历，t:根，m次数

{

LINK\_NODE\* s[MAXN];

int top, i;

if(t == NULL) return;

s[0] = t;

top = 1; //根节点入栈，top指向栈顶的下一个位置

while(top > 0)

{

t = s[--top];

printf("%c ", t -> data);

for(i = t -> num - 1; i >= 0; i--)

s[top++] = t -> child[i];

}

}

void postorderNoRecursion(LINK\_NODE \*t, int m) //非递归后序遍历，t:根，m次数

{

LINK\_NODE \*s[MAXN], \*p;

int mark[MAXN], top = -1, j;

s[++top] = t;

mark[top] = 0; //根节点入栈，top指向栈顶

while(top >= 0)

{

p = s[top];

if(mark[top] == 0 && p -> num > 0)

{

mark[top] = 1;

for(j = p -> num - 1; j >= 0; j--)

{

s[++top] = p -> child[j];

mark[top] = 0;

}

}

if(s[top] -> num == 0 || mark[top] == 1)

printf("%c ", s[top--] -> data);

}

}

void convert1(NODE\_PARENT inputTree[], int \*a, int \*b) //处理括号表示法 a来记录最大次数，b来记录节点数

{

int s[MAXN], num[MAXN], top = -1, m = 0, n = 0; //top记录当前栈顶元素,栈顶元素为父亲节点

char ch; //num记录当前父节点的子节点个数

scanf(" %c", &ch);

inputTree[n].data = ch; inputTree[n++].parent = -1;

while((ch = getchar()) != '\n')

{

if(isalpha(ch))

{

inputTree[n].data = ch;

inputTree[n++].parent = s[top];

num[top]++;

}

else

{

switch(ch)

{

case '(': s[++top] = n - 1;

num[top] = 0;

break;

case ')': m = m > num[top] ? m : num[top];

top--;

}

}

}

\*a = m; \*b = n;

}

void convert2(NODE\_PARENT inputTree[], int \*a, int \*b) //处理双亲表示法 a来记录最大次数，b来记录节点数

{

scanf("%d", b);

int num[MAXN];

int mx = 0, i;

for(i = 0; i < \*b; i++)

{

scanf("%d %c", &inputTree[i].parent, &inputTree[i].data);

num[i] = 0;

if(inputTree[i].parent >= 0) num[inputTree[i].parent]++; //记录父节点的孩子数

}

for(int i = 0; i < \*b; i++) mx = mx > num[i] ? mx : num[i];

\*a = mx;

}

void convert3(NODE\_PARENT inputTree[], int \*a, int \*b) //处理前序+层号表示法 a来记录最大次数，b来记录节点数

{

scanf("%d", b);

int i, tmp, s1[MAXN], s2[MAXN], num[MAXN], top = 0, mx = 0; //s1记录上个节点的序号，s2记录层次，num记录父节点的孩子个数

scanf("%d %c", &tmp, &inputTree[0].data); //top指向栈顶

inputTree[0].parent = -1;

s1[top] = 0;

s2[top] = tmp;

num[top] = 0;

for(i = 1; i < \*b; i++)

{

scanf("%d %c", &tmp, &inputTree[i].data);

while(s2[top] >= tmp) //如果上个节点的层次不比当前节点低，则出栈直到找到层次比当前低的节点

{

mx = mx > num[top] ? mx : num[top];

top--;

}

num[top]++;

inputTree[i].parent = s1[top];

s1[++top] = i;

s2[top] = tmp;

num[top] = 0;

}

while(top >= 0)

{

mx = mx > num[top] ? mx : num[top];

top--;

}

\*a = mx;

}

void get\_leaf\_node(int m, int n) //获得叶子结点的信息

{

int i, cnt = 0;

printf("这棵%d次树的叶子结点为： ", m);

for(i = 0; i < n; i++)

{

if(addr\_NODE[i] -> num == 0)

{

printf("%c ", addr\_NODE[i] -> data);

cnt++;

}

}

printf("\n叶子结点的总个数为%d\n", cnt);

}

void get\_mx\_level(int m, int n) //获得最大层次的信息

{

int i, mx = -1;

for(i = 0; i < n; i++)

mx = mx > addr\_NODE[i] -> lev ? mx : addr\_NODE[i] -> lev;

printf("这棵%d次树的最大层次为%d\n", m, mx);

}

int main()

{

int m, n, tag;

LINK\_NODE \*root;

NODE\_PARENT inputTree[MAXN]; //将输入数据（每一个节点的data与parent）读入到inputTree数组，建树

while(true)

{

printf("请先选择输入的类型：1.括号表示 2.双亲表示 3.前序 + 层号表示\n输入其他数字跳出循环\n");

scanf("%d", &tag);

printf("请输入：\n"); //第2与第3种表示法要首先给定节点个数

switch(tag)

{

case 1: convert1(inputTree, &m, &n); break;

case 2: convert2(inputTree, &m, &n); break;

case 3: convert3(inputTree, &m, &n); break;

default: tag = -1;

}

if(tag == -1) break;

root = creatTreeFromParent(inputTree, m, n); //根节点的parent为-1

printf("前序遍历(递归): "); r\_preorder(root, m);

printf("\n前序遍历(非递归): "); preorderNoRecursion(root, m);

printf("\n\n后序遍历(递归): "); r\_postorder(root, m);

printf("\n后序遍历(非递归): "); postorderNoRecursion(root, m);

printf("\n\n层次遍历: "); hierachicalorder(root, m);

printf("\n\n");

printf("这棵%d次树的节点总数为%d\n", m, n);

get\_leaf\_node(m, n);

get\_mx\_level(m, n);

putchar('\n');

}

return 1;

}

/\*

以下实现了所有必须要完成的函数功能：

1.运行程序，首先先选择不同的二叉树读入方式，有前序+附加两个标志位，前序序列+中序序列，后序序列+中序序列三种读入方式，然后输入相应格式；

2.选择不同的遍历方式，有前序，中序，后序，层次四种遍历方式，每种方式都有递归与非递归两种方式实现；

3.选择对这棵二叉树的不同询问，有节点个数，树的高度，是否为满树，是否为完全树，是否为左小右大的查找树5种询问；

4.最后将这棵二叉树看成森林，分别输出没棵树的前序遍历

\*/

#include <stdio.h>

#include <stdlib.h>

#define MAXN 1024

char PreOrder[MAXN];//"ABDECGHI" 示例1的前序：“ABCDEFSGHIJKLMNO”

char inOrder[MAXN];//"DBEACHGI" 示例1的中序：“BCDASFHIJGEMNOLK”

char postOrder[MAXN];//"DEBHIGCA"

typedef struct binaryTreeNode//二叉树结构

{

char data;//节点值

struct binaryTreeNode \*lchild;//左子节点

struct binaryTreeNode \*rchild;//右子节点

//根据需要加入其它字段

}BiNODE; //树的结构

typedef struct LRtagBiTree//左右标志表示树节点

{

int ltag;

char data;

int rtag;

}LRBTree;

LRBTree LRtag[MAXN];

BiNODE\* initial()//树节点初始化

{

BiNODE\* node;

node= (BiNODE\*)malloc(sizeof(BiNODE));

node->lchild = NULL;

node->rchild = NULL;

//node->level=0;

//node->parent='\0';

//node->data='\0';

return node;

}

BiNODE\* BinaryTreeFromLRtag(int length) //前序+附加两个标志位生成标准链式结构

{

BiNODE \*root = initial(), \*p = root, \*q; //创建根节点与工作指针p,q

BiNODE\* St[MAXN]; //建立栈

int top = 0; //top指向栈顶的下一个元素

for(int i = 0; i < length - 1; i++)

{

p->data = LRtag[i].data;

if(LRtag[i].rtag == 0) //若有右子节点，进栈

St[top++] = p;

q = initial();

if(LRtag[i].ltag == 0)

p->lchild = q;

else //若没有左子节点，则下一个节点为栈顶元素的右子节点

{

p = St[--top];

p->rchild = q;

}

p = q;

}

p->data = LRtag[length-1].data;

return root;

}

//中序遍历的[L1, R1]对应着前序遍历的[L2，R2], 返回所表示树树的根节点

BiNODE\* BiTreeFromInPre\_recursion(char\* inorder, int L1, int R1, char\* preorder, int L2, int R2)

{

if(L1 > R1) return NULL;

BiNODE \*root = initial();

int i;

for(i = L1; i <= R1; i++)

{

if(inorder[i] == preorder[L2]) //在中序遍历中找到根节点的下标i

break;

}

root->data = inorder[i];

root->lchild = BiTreeFromInPre\_recursion(inorder, L1, i - 1, preorder, L2 + 1, L2 - L1 + i); //分别对左右子树进行递归

root->rchild = BiTreeFromInPre\_recursion(inorder, i + 1, R1, preorder, L2 - L1 + i + 1, R2);

return root;

}

BiNODE\* BiTreeFromInPre(char\* inorder, char\* preorder, int length)//根据前序中序生成子树

{

return BiTreeFromInPre\_recursion(inorder, 0, length - 1, preorder, 0, length - 1);

}

BiNODE\* BiTreeFromInPost\_recursion(char\* inorder, int L1, int R1, char \*postorder, int L2, int R2)

{

if(L1 > R1) return NULL;

BiNODE \*root = initial();

int i;

for(i = L1; L1 <= R1; i++)

{

if(inorder[i] == postorder[R2])

break;

}

root->data = inorder[i];

root->lchild = BiTreeFromInPost\_recursion(inorder, L1, i - 1, postorder, L2, L2 - L1 + i - 1);

root->rchild = BiTreeFromInPost\_recursion(inorder, i + 1, R1, postorder, L2 - L1 + i, R2 - 1);

return root;

}

BiNODE\* BiTreeFromInPost(char\* inorder, char\* postorder, int length)//根据前序中序生成子树

{

return BiTreeFromInPost\_recursion(inorder, 0, length - 1, postorder, 0, length - 1);

}

void r\_preorder(BiNODE\* root) //递归实现前序遍历

{

if(root != NULL)

{

printf("%c ", root->data);

r\_preorder(root->lchild);

r\_preorder(root->rchild);

}

}

void non\_r\_preorder(BiNODE\* root) //非递归实现前序遍历

{

BiNODE\* st[MAXN], \*p = root;

int top = 0;

while(top != 0 || p != NULL)

{

while(p != NULL)

{

printf("%c ", p->data);

st[top++] = p; //节点入栈

p = p->lchild; //指向右孩子

}

if(top != 0)

{

p = st[--top];

p = p->rchild;

}

}

}

void r\_inorder(BiNODE\* root) //递归实现中序遍历

{

if(root != NULL)

{

r\_inorder(root->lchild);

printf("%c ", root->data);

r\_inorder(root->rchild);

}

}

void non\_r\_inorder(BiNODE\* root) //非递归实现中序遍历

{

BiNODE\* st[MAXN], \*p = root;

int top = 0;

while(top != 0 || p != NULL)

{

while(p != NULL)

{

st[top++] = p; //节点入栈

p = p->lchild; //指向右孩子

}

if(top != 0)

{

p = st[--top]; //取出栈顶元素

printf("%c ", p->data);

p = p->rchild;

}

}

}

void r\_postorder(BiNODE\* root) //递归实现后序遍历

{

if(root != NULL)

{

r\_postorder(root->lchild);

r\_postorder(root->rchild);

printf("%c ", root->data);

}

}

void non\_r\_postorder(BiNODE\* root) //非递归后序遍历

{

BiNODE \*s[MAXN], \*p;

int mark[MAXN], top = -1; //mark为0表示此节点没有展开子节点，1表示已经展开

s[++top] = root;

mark[top] = 0; //根节点入栈，top指向栈顶

while(top >= 0)

{

p = s[top];

if(mark[top] == 0 && (p->lchild || p->rchild)) //若是这个节点有子孩子，且子节点没有展开，展开子节点

{

mark[top] = 1;

if(p->rchild) //将存在的子节点入栈

{

s[++top] = p->rchild;

mark[top] = 0;

}

if(p->lchild)

{

s[++top] = p->lchild;

mark[top] = 0;

}

}

if((!s[top]->lchild && !s[top]->rchild) || mark[top] == 1) //若是栈顶元素的子节点已经展开或者没有子节点

printf("%c ", s[top--]->data);

}

}

void levelorder(BiNODE\* root)

{

BiNODE\* q[MAXN], \*p;

q[0] = root;

int head = 0, tail = 1; //head指向队首，head指向队尾下一个

while(head != tail)

{

p = q[head++];

if(p->lchild) q[tail++] = p->lchild;

if(p->rchild) q[tail++] = p->rchild;

printf("%c ", p->data);

}

}

int get\_node(BiNODE\* root) //获得一棵树的节点

{

if(root == NULL) return 0;

return 1 + get\_node(root->lchild) + get\_node(root->rchild);

}

int get\_height(BiNODE\* root) //获得一棵树的高度，高度从1开始计数，最后答案要减1

{

if(root == NULL) return 0;

return 1 + (get\_height(root->lchild) > get\_height(root->rchild) ? get\_height(root->lchild) : get\_height(root->rchild));

}

int judge\_fulltree(BiNODE\* root) //判断一棵树是否为满树

{

int height = get\_height(root) - 1, node = get\_node(root);

if((1<<(height + 1)) -1 == node) return 1;

else return -1;

}

int judge\_completetree(BiNODE\* root) //判断一棵树是否为完全二叉树

{

int flag = 0;

BiNODE\* q[MAXN], \*p;

q[0] = root;

int head = 0, tail = 1; //head指向队首，head指向队尾下一个;

while(head != tail)

{

p = q[head++];

if(flag == 1 && (p->lchild || p->rchild)) //已经标记好之后又出现了有左孩子或者右孩子的节点

{

flag = -1;

break;

}

if(!p->lchild || !p->rchild) //遇到第一个没有左儿子或者右儿子的节点，设置标志位

flag = 1;

if(p->lchild) q[tail++] = p->lchild;

if(p->rchild) q[tail++] = p->rchild;

}

if(flag == -1) return 0;

return 1;

}

int judge\_findtree(BiNODE\* root) //判断一棵树是否为查找树

{

char ans[MAXN]; //记录中序遍历的结果

int cnt = 0;

BiNODE\* st[MAXN], \*p = root; //进行中序遍历

int top = 0;

while(top != 0 || p != NULL)

{

while(p != NULL)

{

st[top++] = p; //节点入栈

p = p->lchild; //指向右孩子

}

if(top != 0)

{

p = st[--top]; //取出栈顶元素

ans[cnt++] = p->data;

p = p->rchild;

}

}

for(int i = 1; i < cnt; i++)

{

if(ans[i] <= ans[i - 1])

return 0;

}

return 1;

}

void binary\_tree\_to\_forest(BiNODE \*root)

{

int cnt = 0;

printf("第%d棵树的前序遍历为：%c ", ++cnt, root->data);

non\_r\_preorder(root->lchild);

putchar('\n');

while(root->rchild)

{

root = root->rchild;

printf("第%d棵树的前序遍历为：%c ", ++cnt, root->data);

non\_r\_preorder(root->lchild);

putchar('\n');

}

}

int main()

{

int cas, n;

BiNODE\* root;

printf("请选择二叉树的读入方式：\n1. 前序+附加两个标志位\n2. 前序序列+中序序列\n3. 后序序列+中序序列\n");

scanf("%d", &cas);

if(cas == 1)

{

printf("请在第一行输入树的节点数n，再输出n行节点的信息\n");

scanf("%d", &n);

for(int i = 0; i < n; i++)

scanf("%d %c %d", &LRtag[i].ltag, &LRtag[i].data, &LRtag[i].rtag);

root = BinaryTreeFromLRtag(n);

}

else if(cas == 2)

{

printf("请在第一行输入树的节点数n，第二行输入前序序列，第三行输入中序序列\n");

scanf("%d", &n);

for(int i = 0; i < n; i++)

scanf(" %c", &PreOrder[i]);

for(int i = 0; i < n; i++)

scanf(" %c", &inOrder[i]);

root = BiTreeFromInPre(inOrder, PreOrder, n);

}

else

{

printf("请在第一行输入树的节点数n,第二行输入中序序列，第三行输入后序序列\n");

scanf("%d", &n);

for(int i = 0; i < n; i++)

scanf(" %c", &inOrder[i]);

for(int i = 0; i < n; i++)

scanf(" %c", &postOrder[i]);

root = BiTreeFromInPost(inOrder, postOrder, n);

}

while(1)

{

printf("\n请选择相应的遍历方式：1.前序遍历 2.中序遍历 3.后序遍历 4.层次遍历\n输入其他数字退出\n");

scanf("%d", &cas);

if(cas == 1)

{

printf("递归前序遍历： ");

r\_preorder(root); printf("\n非递归前序遍历： ");

non\_r\_preorder(root); putchar('\n');

}

else if(cas == 2)

{

printf("递归中序遍历： ");

r\_inorder(root); printf("\n非递归中序遍历： ");

non\_r\_inorder(root); putchar('\n');

}

else if(cas == 3)

{

printf("递归后序遍历： ");

r\_postorder(root); printf("\n非递归后序遍历： ");

non\_r\_postorder(root); putchar('\n');

}

else if(cas == 4)

{

printf("层次遍历： ");

levelorder(root); putchar('\n');

}

else break;

}

while(1)

{

printf("\n请选择对树的询问操作：\n1.树的节点数 2.树的高度\n 3.是否为满树 4.是否为完全二叉树\n");

printf("5.判断一棵二叉树是否左子树上的结点的值都小于根，右子树上的结点的值都大于根\n输入其他数字退出\n");

scanf("%d", &cas);

if(cas == 1)

printf("这棵树的节点数为：%d\n", get\_node(root));

else if(cas == 2)

printf("这棵树的高度为：%d\n", get\_height(root) - 1);

else if(cas == 3)

{

if(judge\_fulltree(root) > 0) printf("这棵树是满树\n");

else printf("这棵树不是满树\n");

}

else if(cas == 4)

{

if(judge\_completetree(root) > 0) printf("这棵树是完全二叉树\n");

else printf("这棵树不是完全二叉树\n");

}

else if(cas == 5)

{

if(judge\_findtree(root) > 0) printf("是一棵左小右大的查找树\n");

else printf("不是一棵左小右大的查找树\n");

}

else break;

}

printf("这棵二叉树可以转化为森林：\n");

binary\_tree\_to\_forest(root);

}

皇后问题

#include<bits/stdc++.h>

#define INF 0x3f3f3f3f

const int maxn = 20;

using namespace std;

int x[maxn], y[maxn]; //表示一行与一列

int a[2\*maxn], b[2\*maxn]; //表示主对角线与次对角线

int cnt, tmp[14], ans[14], n;

void dfs(int k1, int k2)

{

if(x[k1] || y[k2] || a[k1+k2] || b[k1-k2+n]) //剪枝

return;

tmp[k1] = k2; //记录

if(k1 == n) //若找到最后答案

{

cnt++;

if(cnt == 1)

{

for(int i = 1; i <= n; i++)

ans[i] = tmp[i];

}

return;

}

x[k1] = y[k2] = a[k1+k2] = b[k1-k2+n] = 1;

for(int i = 1; i <= n; i++)

dfs(k1+1, i);

x[k1] = y[k2] = a[k1+k2] = b[k1-k2+n] = 0;

}

int main()

{

scanf("%d", &n);

for(int i = 1; i <= n; i++)

dfs(1, i);

printf("%d\n", cnt);

if(!cnt) return 0;

for(int j = 1; j < n; j++)

printf("%d ", ans[j]-1);

printf("%d\n", ans[n]-1);

}