```
/*
树的定义：
    根 + 子树，以一种递归的方式定义
    子树是不相交的
    除了根结点外，每个结点有且仅有一个父结点
    一棵 N 个节点的树有 N-1 条边
    规定根结点在第 1 层

树的表示：
    用数组表示比较困难
    用链表实现比较方便，但是每个结点的结构会因子树数目的不同而不同
    如果令结点结构一致，会造成空间上的浪费

    建议使用"儿子-兄弟表示法"
    左指针指向第一个儿子，右指针指向下一个兄弟
    这样，一般的树就转换成了二叉树

二叉树的定义：
    这个集合可以为空
    若不为空，则它由根结点和左子树、右子树两个不相交的二叉树组成
    二叉树的子树有左右顺序之分

特殊的二叉树：
    斜二叉树
    完美/满二叉树
    完全二叉树

二叉树的几个重要性质：
    二叉树第 i 层的最大结点数为 2^(i-1) ， i>=1 ， 层数从第一层开始计算
    深度为k的二叉树最大结点数为 2^k-1 ， k>=1
    n0表示叶结点数，n2表示度为2的结点数，n0 = n2 + 1

二叉树的遍历：
    先序遍历
    中序遍历
    后序遍历
    光有前序和后序序列，无法唯一确定二叉树
    层次遍历
*/


#include<stdio.h>
#include<malloc.h>
#include<vector>
#include<queue>
#include<algorithm>
using namespace std;


typedef struct TreeNode *BinTree;
struct TreeNode{
    int Data;       // 存值
    BinTree Left;  // 左儿子结点
    BinTree Right; // 右儿子结点
};
BinTree CreatBinTree();     // 创建一个二叉树
bool IsEmpty(BinTree BT);  // 判断树 BT 是否为空
void PreOrderTraversal_1(BinTree BT);  // 先序遍历，根左右
void PreOrderTraversal_2(BinTree BT);  // 先序遍历，根左右
void InOrderTraversal_1(BinTree BT);   // 中序遍历，左根右
void InOrderTraversal_2(BinTree BT);   // 中序遍历，左根右
void PostOrderTraversal_1(BinTree BT); // 后序遍历，左右根
void PostOrderTraversal_2(BinTree BT); // 后序遍历，左右根


typedef struct SNode *Stack;
struct SNode{
    BinTree Data;
    Stack Next;
};
Stack CreateStack();    // 初始化链栈
int IsEmpty(Stack S);  // 判断链栈是否为空
void Push(Stack S,BinTree item); // 入栈
```

```
74  BinTree Pop(Stack S);  // 出栈
75
76
77  // 初始化
78  Stack CreateStack()
79  {
80      Stack S;
81      S = (Stack)malloc(sizeof(struct SNode));
82      S->Next = NULL; // 头结点不保存元素
83      return S;
84  }
85
86  // 判断是否为空
87  int IsEmpty(Stack S)
88  {
89      return (S->Next == NULL);
90  }
91
92  // 入栈
93  void Push(Stack S, BinTree item)
94  {
95      Stack tmp;
96      tmp = (Stack)malloc(sizeof(struct SNode));
97      tmp->Data = item;
98      tmp->Next = S->Next;
99      S->Next = tmp;
100 }
101
102 // 出栈
103 BinTree Pop(Stack S)
104 {
105     Stack popNode;
106     BinTree popVal;
107     if(IsEmpty(S))
108     {
109         printf("堆栈空");
110         return 0;
111     }
112     else
113     {
114         popNode = S->Next;
115         S->Next = popNode->Next;
116         popVal = popNode->Data; // 取出被删除结点的值
117         free(popNode); // 释放空间
118         return popVal;
119     }
120 }
121
122 BinTree Insert(int Data)
123 {
124     BinTree BT;
125     BT =  (BinTree)malloc(sizeof(struct TreeNode));
126     BT->Data = Data;
127     BT->Left = NULL;
128     BT->Right = NULL;
129     return BT;
130 }
131
132 // 初始化二叉树
133 BinTree CreatBinTree()
134 {
135     BinTree BT;
136     BT =  (BinTree)malloc(sizeof(struct TreeNode));
137     BT->Data = 1;
138     BT->Left = Insert(2);
139     BT->Right = Insert(3);
140     BT->Left->Left = Insert(4);
141     BT->Left->Right = Insert(6);
142     BT->Left->Right->Left = Insert(5);
143     BT->Right->Left = Insert(7);
144     BT->Right->Right = Insert(9);
145     BT->Right->Left->Right = Insert(8);
146     return BT;
```

```
}

// 根据前序序列初始化二叉树
// 1 2 4 0 0 6 5 0 0 0 3 7 0 8 0 0 9 0 0
void CreatBinTreeFomPreorder(BinTree &BT)
{
    int val;
    scanf("%d", &val);
    if(val == 0)
        BT = NULL;
    else
    {
        BT = (BinTree)malloc(sizeof(struct TreeNode));
        BT->Data = val;
        CreatBinTreeFomPreorder(BT->Left);
        CreatBinTreeFomPreorder(BT->Right);
    }
}

// 先序递归
void PreOrderTraversal_1(BinTree BT)
{
    if(BT)
    {
        printf("%d", BT->Data);        // 打印根
        PreOrderTraversal_1(BT->Left);  // 进入左子树
        PreOrderTraversal_1(BT->Right); // 进入右子树
    }
}

// 先序非递归
void PreOrderTraversal_2(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈 S
    while(T || !IsEmpty(S))  // 当树不为空或堆栈不空
    {
        while(T)
        {
            Push(S, T);    // 压栈，第一次遇到该结点
            printf("%d", T->Data); // 访问结点
            T = T->Left;   // 遍历左子树
        }
        if(!IsEmpty(S))    // 当堆栈不空
        {
            T = Pop(S);    // 出栈，第二次遇到该结点
            T = T->Right;  // 访问右结点
        }
    }
}

// 中序递归
void InOrderTraversal_1(BinTree BT)
{
    if(BT)
    {
        InOrderTraversal_1(BT->Left);  // 进入左子树
        printf("%d", BT->Data);       // 打印根
        InOrderTraversal_1(BT->Right); // 进入右子树
    }
}

// 中序非递归
void InOrderTraversal_2(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈 S
    while(T || !IsEmpty(S))  // 当树不为空或堆栈不空
    {
        while(T)
        {
            Push(S, T);    // 压栈
            T = T->Left;   // 遍历左子树
```

```
        }
        if(!IsEmpty(S))     // 当堆栈不空
        {
            T = Pop(S);     // 出栈
            printf("%d", T->Data); // 访问结点
            T = T->Right;  // 访问右结点
        }
    }
}

// 后序递归
void PostOrderTraversal_1(BinTree BT)
{
    if(BT)
    {
        PostOrderTraversal_1(BT->Left);  // 进入左子树
        PostOrderTraversal_1(BT->Right); // 进入右子树
        printf("%d",BT->Data);           // 打印根
    }
}

// 后序非递归
void PostOrderTraversal_2(BinTree BT)
{
    BinTree T = BT;
    Stack S = CreateStack(); // 创建并初始化堆栈 S
    vector<BinTree> v;
    Push(S, T);
    while(!IsEmpty(S)) // 当前结点非空
    {
        T = Pop(S);
        v.push_back(T);
        if(T->Left)
            Push(S, T->Left);
        if(T->Right)
            Push(S, T->Right);
    }
    reverse(v.begin(), v.end()); // 逆转
    for(int i=0; i<v.size(); i++)
        printf("%d", v[i]->Data);
}

// 层次遍历
void LevelOrderTraversal(BinTree BT)
{
    queue<BinTree> q;
    BinTree T;
    if(!BT)
        return;

    q.push(BT); // BT 入队
    while(!q.empty())
    {
        T = q.front();  // 访问队首元素
        q.pop();        // 出队
        printf("%d", T->Data);
        if(T->Left)
            q.push(T->Left);
        if(T->Right)
            q.push(T->Right);
    }
}

// 输出叶子结点
void  FindLeaves(BinTree BT)
{
    if(BT)
    {
        if(!BT->Left && !BT->Right)
            printf("%d", BT->Data); // 打印叶子结点
        FindLeaves(BT->Left);  // 进入左子树
        FindLeaves(BT->Right); // 进入右子树
    }
```

```
}

// 求树高度
int  GetHeight(BinTree BT)
{
    int hl, hr, maxh;
    if(BT)
    {
        hl = GetHeight(BT->Left);  // 求左子树高度
        hr = GetHeight(BT->Right); // 求右子树高度
        maxh = (hl>hr)?hl:hr;
        return maxh + 1;  // 当前结点高度为左右子树最大的高度+1
    }
    else
        return 0;
}

int main()
{
    BinTree BT, ST;

    printf("请按照前序序列输入二叉树，空结点用0表示，结点之间用空格分隔：\n");
    // CreatBinTreeFomPreorder(BT);
    BT = CreatBinTree();

    printf("先序遍历：\n");
    PreOrderTraversal_1(BT);
    printf("\n");
    PreOrderTraversal_2(BT);

    printf("\n中序遍历：\n");
    InOrderTraversal_1(BT);
    printf("\n");
    InOrderTraversal_2(BT);

    printf("\n后序遍历：\n");
    PostOrderTraversal_1(BT);
    printf("\n");
    PostOrderTraversal_2(BT);

    printf("\n层次遍历：");
    LevelOrderTraversal(BT);

    printf("\n输出叶子结点:");
    FindLeaves(BT);

    printf("\n输出树的高度：%d", GetHeight(BT));
    return 0;
}
```