# 树

# 一：基本篇

# 解决问题的思想

参考链接：

https://zhuanlan.zhihu.com/p/59505888?utm\_source=wechat\_session&utm\_medium=social&utm\_oi=1180477396925583360&from=singlemessage

## 自顶向下（被认为先序）

意味着在每个递归层级，我们将首先访问节点来计算一些值，并在递归调用函数时将这些值传递到子节点。 所以 “自顶向下” 的解决方案可以被认为是一种前序遍历**。例如树的构建中基本用的都是自顶向下。**

**伪码：**

1. return specific value for null node

2. update the answer if needed // anwer <-- params

3. left\_ans = top\_down(root.left, left\_params) // left\_params <-- root.val, params

4. right\_ans = top\_down(root.right, right\_params) // right\_params <-- root.val, params

5. return the answer if needed// answer <-- left\_ans,

right\_ans

例如，思考这样一个问题：给定一个二叉树，请寻找它的最大深度。

我们知道根节点的深度是1。 对于每个节点，如果我们知道某节点的深度，那我们将知道它子节点的深度。 因此，在调用递归函数的时候，将节点的深度传递为一个参数，那么所有的节点都知道它们自身的深度。 而对于叶节点，我们可以通过更新深度从而获取最终答案。 这里是递归函数 maximum\_depth(root, depth) 的伪代码：

1. return if root is null

2. if root is a leaf node:

3. answer = max(answer, depth) // update the answer if needed

4. maximum\_depth(root.left, depth + 1) // call the function recursively for left child

5. maximum\_depth(root.right, depth + 1) // call the function recursively for right child

## 自底向上（被认为后序）

自底向上” 是另一种递归方法。 在每个递归层次上，我们首先对所有子节点递归地调用函数，然后根据返回值和根节点本身的值得到答案。 这个过程可以看作是**后序遍历**的一种。递归的函数一般形如bottom\_up(root)

伪码：

1. return specific value for null node

2. left\_ans = bottom\_up(root.left) // call function recursively for left child

3. right\_ans = bottom\_up(root.right) // call function recursively for right child

4. return answers // answer <-- left\_ans, right\_ans, root.val

继续讨论前面关于树的最大深度的问题，但是使用不同的思维方式：对于树的单个节点，以节点自身为根的子树的最大深度x是多少？

如果我们知道一个根节点，以其**左**子节点为根的最大深度为l和以其**右**子节点为根的最大深度为r，我们是否可以回答前面的问题？ 当然可以，我们可以选择它们之间的最大值，再加上1来获得根节点所在的子树的最大深度。 那就是 x = max（l，r）+ 1。

这意味着对于每一个节点来说，我们都可以在解决它子节点的问题之后得到答案。 因此，我们可以使用“自底向上“的方法。下面是递归函数 maximum\_depth(root) 的伪代码：

1. return 0 if root is null // return 0 for null node

2. left\_depth = maximum\_depth(root.left)

3. right\_depth = maximum\_depth(root.right)

4. return max(left\_depth, right\_depth) + 1 // return depth of the subtree rooted at root

# 层次遍历

这个答案不是，下面的是自己的东西，可以配合下面树的构建一起用。

还有好处是当构建的树自己不知道是啥样的时候（当然可以通过先序加中序的序列来自己画，但是不够直观），可以用这种层次遍历出来，一目了然。

每一层处理完，每一层的节点个数其实和队列长度是一样的。

int\* LevelTraverse(TREENODE\_T\* root, int\* returnSize)

{

    int\* res = NULL;

    int res\_count = 0;

    int i, levelSize;

    TREENODE\_T\* headNode = NULL;

    QUEUE\_T queue = {0};

    QueueReInit(&queue);

    QueuePush(&queue, root);

    res = (int\*)calloc(QUEUE\_MAX, sizeof(int));

    while (QueueIsEmpty(&queue) != 1) {

        levelSize = QueueSize(&queue);

        for (i = 0;i < levelSize; i++) {

            headNode = QueuePop(&queue);

            if (headNode == NULL) {

                res[res\_count++] = -1;

            } else {

                res[res\_count++] = headNode->val;

// 避免每个叶子节点的最后都打-1，-1，造成浪费

if (headNode->left == NULL && headNode->right == NULL) {

continue;

}

                QueuePush(&queue, headNode->left);

                QueuePush(&queue, headNode->right);

            }

        }

    }

    \*returnSize = res\_count;

    return res;

}

#define QUEUE\_MAX 100

typedef struct TreeNode{

    struct TreeNode\* left;

    struct TreeNode\* right;

    int val;

}TREENODE\_T;

typedef struct Queue {

    TREENODE\_T\* nodePool[QUEUE\_MAX];

    int head;

    int tail;

    int size;

}QUEUE\_T;

int QueueReInit(QUEUE\_T\* queue);

int QueueIsFull(QUEUE\_T\* queue);

int QueueIsEmpty(QUEUE\_T\* queue);

TREENODE\_T\* QueuePop(QUEUE\_T\* queue);

int QueueSize(QUEUE\_T\* queue);

int QueueReInit(QUEUE\_T\* queue)

{

    queue->head = 0;

    queue->tail = 0;

    queue->size = 0;

    memset(queue->nodePool, 0, QUEUE\_MAX \* sizeof(TREENODE\_T\*));

}

int QueueIsFull(QUEUE\_T\* queue)

{

    return queue->size >= QUEUE\_MAX ? 1 : 0;//(queue->tail + 1) % QUEUE\_MAX == queue->head;

}

int QueueIsEmpty(QUEUE\_T\* queue)

{

    return queue->size == 0 ? 1 : 0;//queue->head == queue->tail;

}

int QueuePush(QUEUE\_T\* queue, TREENODE\_T\* root)

{

    if (QueueIsFull(queue) == 1) {

        return 0;

    }

    queue->nodePool[queue->tail] = root;

    queue->tail = (++queue->tail) % QUEUE\_MAX;

    queue->size++;

    return 1;

}

TREENODE\_T\* QueuePop(QUEUE\_T\* queue)

{

    TREENODE\_T\* node = NULL;

    if (QueueIsEmpty(queue) == 1) {

        return NULL;

    }

    node = queue->nodePool[queue->head];

    queue->head = (++queue->head) % QUEUE\_MAX;

    queue->size--;

    return node;

}

int QueueSize(QUEUE\_T\* queue)

{

    return queue->size;

}

# 树的构建

写学会先中后序以及层次遍历再到这里看，写在前面是为了可以好构造用例。

## 1-105.根据完全二叉树数组创建二叉树【非leetcode题但关键】

顺序结构的数组需要恢复成普通的树需要这个数组表示的是完全二叉树，即空节点也要表示。否则树不会唯一确定，有多个匹配.

1. 已知一颗树按层次遍历之后放在数组中，数组中的-1被认为是空节点。请恢复二叉树,并按照先序遍历打印.

例如树   的层次遍历之后肯定是数组[1,2,3,4,5,-1,6,-1,-1,7,8,-1,-1],请利用数组恢复二叉树,并先序遍历.

1  
   / \  
  2   3  
 / \   \  
4   5   6  
   / \  
  7   8

思路：

由表示完全二叉树状的数组我们知道:

当前节点的序号为index，那么：

左孩子序号为2\*index + 1

右孩子序号为2\*index + 2

因此可以通过递归的形式。每遇到一个当前非-1的节点就将其建立为一个父节点，然后递归其左右。

/\*\*树的节点定义\*/

typedef struct TreeNode{

    struct TreeNode\* left;

    struct TreeNode\* right;

    int val;

}TREENODE\_T;

/\*\*从层次遍历出的序列（完全二叉树）恢复原来的树，原来不一定是完全二叉树\*/

TREENODE\_T\* CreateBinaryTreeFromLevelTraverseVec(int\* vec, int start, int vecSize)

{

    TREENODE\_T\* root = NULL;

    /\*\*节点用完了，就不用继续了\*/

    if (start >= vecSize) {

        return NULL;

    }

    /\*\*如果是空节点的话也不进行构造\*/

    if (vec[start] == -1) {

        return NULL;

    }

    root = (TREENODE\_T\*)calloc(1, sizeof(TREENODE\_T));

    root->val = vec[start];

    root->left = CreateBinaryTreeFromLevelTraverseVec(vec, start \* 2 + 1,vecSize);

root->right = CreateBinaryTreeFromLevelTraverseVec(vec, start \* 2 + 2,vecSize);

// 最后返回根节点

    return root;

}

/\*\*销毁二叉树，其实是后续遍历\*/

void DestroyBinaryTree(TREENODE\_T\*\* root)

{

    if (\*root == NULL) {

        return;

    }

    DestroyBinaryTree(&((\*root)->left));

    DestroyBinaryTree(&((\*root)->right));

    free(\*root);

    \*root = NULL;

}

## 2. [105. 从前序与中序遍历序列构造二叉树](https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/)



思路：从前序和中序可以唯一确定一颗树。

前序：根左右

中序：左根右。

那么对于2个当前树的序列，我们可以根据当前树的前序序列的首元素知道根结点，再根据当前树的中序序列知道根的左树中有哪些元素，右树中有那些元素。

于是其实这里的每一次递归都可以理解为将2个序列分别拆为左树的前序和中序，以及右树的前序和中序。

比如：第一步知道preorder[0]为根节点。

那么inorder中可以知道3左边9是左树的中序序列，右边15，20，7是右树的中序序列。

进而知道preorder中的9是左树的前序序列，20，15，7是右树的前序序列。

代码：

struct TreeNode\* BuildreeByPreAndInorder(int\* preOrder, int preLeft, int preRight,

                                         int\* inOrder, int inLeft, int inRight)

{

    int iRoot;

    int lenLeft;

    int lenRight;

    struct TreeNode\* root = NULL;

    if (preLeft >= preRight || inLeft >= inRight) {

        return NULL;

    }

    /\*\*1.创建根结点\*/

    root = (struct TreeNode\*)calloc(1, sizeof(struct TreeNode));

    root->val = preOrder[preLeft];

    /\*\*2.寻找根节点在inorder中的位置,试图找到找出当前根节点的左树的中序序列以及右树的中序序列\*/

    for (iRoot = inLeft; iRoot < inRight; iRoot++) {

        if (inOrder[iRoot] == root->val) {

            break;

        }

    }

    /\*\*3.根据iRoot切分序列,体现在参数上\*/

    lenLeft = iRoot - inLeft;

    /\*\*4.左树的前序和中序序列继续递归构造左树\*/

    root->left = BuildreeByPreAndInorder(preOrder, preLeft + 1, preLeft + 1 + lenLeft,

                                         inOrder, inLeft, iRoot);/\*\*iRoot没有减1是因为开区间\*/

    /\*\*5.右树的前序和中序序列继续递归构造右树\*/

    root->right = BuildreeByPreAndInorder(preOrder, preLeft + 1 + lenLeft, preRight,

                                          inOrder, iRoot + 1, inRight);

    return root;

}

struct TreeNode\* buildTree(int\* preorder, int preorderSize, int\* inorder, int inorderSize)

{

    struct TreeNode\* root = NULL;

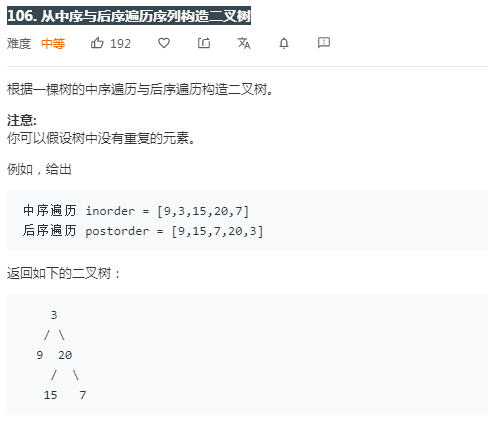
    root = BuildreeByPreAndInorder(preorder, 0, preorderSize,

                                   inorder, 0, inorderSize);

    return root;

}

## 3-105.[106. 从中序与后序遍历序列构造二叉树](https://leetcode-cn.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/)



思路同上：

/\*\*

中序遍历 inorder = [9,3,15,20,7]    左根右

后序遍历 postorder = [9,15,7,20,3]  左右根

\*/

struct TreeNode\* BuildTreeByPostAndInorder(int\* postOrder, int postLeft, int postRight,

                                           int\* inOrder, int inLeft, int inRight)

{

    struct TreeNode\* root = NULL;

    int iRoot;

    int lenLeft;

    if (postLeft >= postRight || inLeft >= inRight) {

        return NULL;

    }

    root = (struct TreeNode\*)calloc(1, sizeof(struct TreeNode));

    root->val = postOrder[postRight - 1];

    /\*\*从中序遍历的序列中寻找根的位置，进一步如何分割序列\*/

    for (iRoot = inLeft; iRoot < inRight; iRoot++) {

        if (inOrder[iRoot] == root->val) {

            break;

        }

    }

    lenLeft = iRoot - inLeft;

    /\*\*构建左子树\*/

    root->left = BuildTreeByPostAndInorder(postOrder, postLeft, postLeft + lenLeft,

                                           inOrder, inLeft, iRoot);

    /\*\*构建右子树\*/

    root->right = BuildTreeByPostAndInorder(postOrder, postLeft + lenLeft, postRight - 1,

                                            inOrder, iRoot + 1, inRight);

 return root;

}

struct TreeNode\* buildTree(int\* inorder, int inorderSize, int\* postorder, int postorderSize)

{

    struct TreeNode\* root = NULL;

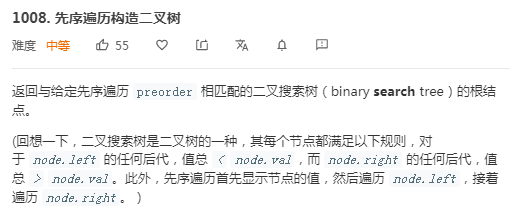
    root = BuildTreeByPostAndInorder(postorder, 0, postorderSize,

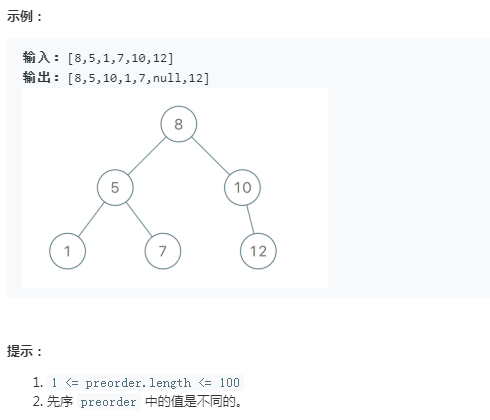
                                     inorder, 0, inorderSize);

    return root;

}

## 4. [1008. 先序遍历构造二叉树](https://leetcode-cn.com/problems/construct-binary-search-tree-from-preorder-traversal/)





思路：

和前面的一样，切割好序列，递归的去构建左右子树即可。

由于是先序的，那么任何一个序列的开始处即preOrder[preLeft]必定是根，

而由于是BST，那么该节点的左子树序列肯定是[preLeft+ 1, index)，右子树是[index, preRight)

struct TreeNode\* BuildOneBstTreeByPreorder(int\* preOrder, int preLeft, int preRight)

{

    struct TreeNode\* root = NULL;

    int index = 0;

    if (preLeft >= preRight) {

        return NULL;

    }

    root = (struct TreeNode\*)calloc(1, sizeof(struct TreeNode));

    root->val = preOrder[preLeft];

    /\*\*找到第一个比根大的下标，为了分割。虽然剩余的不是严格排序的，仍然可以利用二分查找的方法寻找这个index\*/

    for (index = preLeft + 1; index < preRight; index++) {

        if (preOrder[index] > root->val) {

            break;

        }

    }

    root->left = BuildOneBstTreeByPreorder(preOrder, preLeft + 1, index);

    root->right = BuildOneBstTreeByPreorder(preOrder, index, preRight);

    return root;

}

struct TreeNode\* bstFromPreorder(int\* preorder, int preorderSize)

{

    struct TreeNode\* root = NULL;

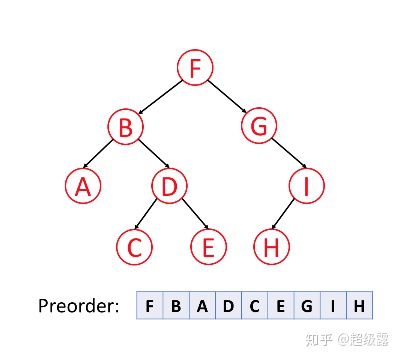
    root = BuildOneBstTreeByPreorder(preorder, 0, preorderSize);

    return root;

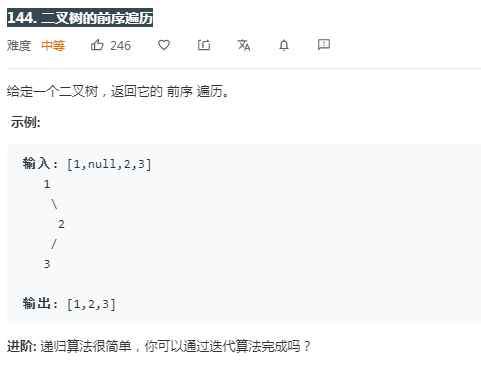
}

# 先序遍历preorder

遍历的顺序是先root、left、right,对于每个节点也是如此。



## [144. 二叉树的前序遍历](https://leetcode-cn.com/problems/binary-tree-preorder-traversal/)



方法一递归：

#define NODE\_MAX\_NUM 100

int g\_count = 0;

void PreOrderDfs(struct TreeNode\* root, int\* res)

{

    if (root == NULL) {

        return;

}

// 1.访问根节点

    res[g\_count++] = root->val;

    // 2.分别递归访问左右

PreOrderDfs(root->left, res);

    PreOrderDfs(root->right, res);

}

int\* preorderTraversal(struct TreeNode\* root, int\* returnSize)

{

    int\* res = NULL;

    res = (int\*)calloc(NODE\_MAX\_NUM, sizeof(int));

    g\_count = 0;

    PreOrderDfs(root, res);

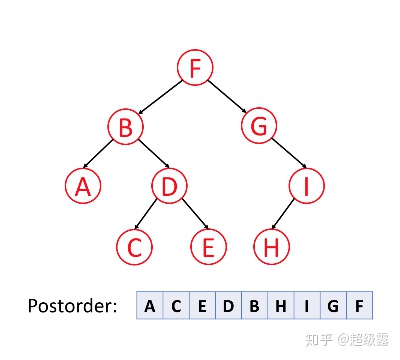
    \*returnSize = g\_count;

    return res;

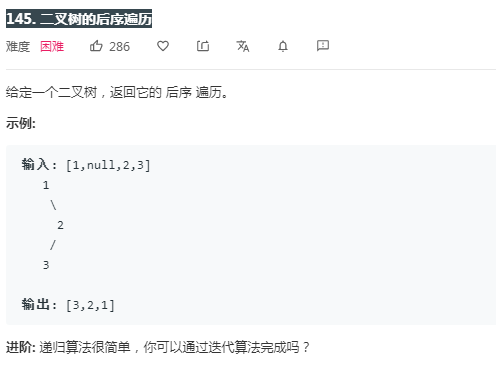
}

方法二非递归（迭代，栈）：

# 后序遍历



## 1.[145. 二叉树的后序遍历](https://leetcode-cn.com/problems/binary-tree-postorder-traversal/)



方法一递归：

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

#define MAX\_NODE\_NUM 100

int g\_count = 0;

void PostOrderDfs(struct TreeNode\* root, int\* res)

{

    if (root == NULL) {

        return;

    }

    PostOrderDfs(root->left, res);

    PostOrderDfs(root->right, res);

    res[g\_count++] = root->val;

}

int\* postorderTraversal(struct TreeNode\* root, int\* returnSize)

{

    int\* res = NULL;

    res = (int\*)calloc(MAX\_NODE\_NUM, sizeof(int));

    g\_count = 0;

    PostOrderDfs(root, res);

    \*returnSize = g\_count;

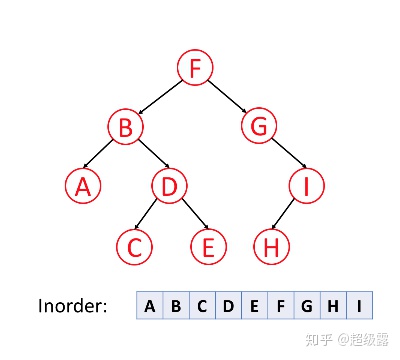
    return res;

}

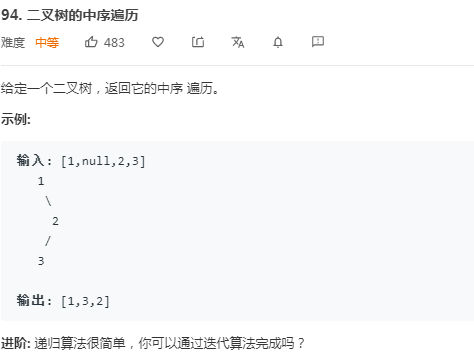
方法二非递归（迭代）

# 中序遍历

遍历的顺序是先left、root、right,对于每个节点也是如此。



## 1..[94. 二叉树的中序遍历](https://leetcode-cn.com/problems/binary-tree-inorder-traversal/)



方法一：递归

#define MAX\_NODE\_NUM 100

int g\_count = 0;

void InorderDfs(struct TreeNode\* root, int\* res)

{

    if (root == NULL) {

        return;

    }

    InorderDfs(root->left, res);

    res[g\_count++] = root->val;

    InorderDfs(root->right, res);

}

int\* inorderTraversal(struct TreeNode\* root, int\* returnSize)

{

    int\* res = NULL;

    g\_count = 0;

    res = (int\*)calloc(MAX\_NODE\_NUM, sizeof(int));

    InorderDfs(root, res);

    \*returnSize = g\_count;

    return res;

}

方法二非递归（迭代）

# 二：树的自身特性问题

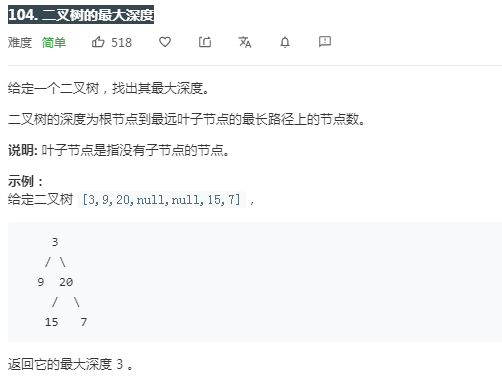
参考链接：

https://zhuanlan.zhihu.com/p/59505888?utm\_source=qq&utm\_medium=social&utm\_oi=1180477396925583360

## 基本特性

包括对称，翻转，打印，合并、剪枝、右视图、深度和宽度

## [104. 二叉树的最大深度](https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/)



思路：

方法一：【自顶向下】

可以采用自顶向下的方法。假设知道当前节点的深度depth，以该节点为根的左树的深度其实就是depth+1,同理右边也是。

int g\_MaxDepth = 0;

#define GET\_MAX(x, y) ((x) < (y)) ? (y) : (x)

void DfsMaxDepthUpDown(struct TreeNode\* root, int depth)

{

    // 到达叶子节点更新最大值

if (root == NULL) {

        g\_MaxDepth = GET\_MAX(g\_MaxDepth, depth);

        return;

}

// 子节点的深度为当前的节点的深度+1

    DfsMaxDepth(root->left, depth + 1);

    DfsMaxDepth(root->right, depth + 1);

}

int maxDepth(struct TreeNode\* root)

{

    g\_MaxDepth = INT\_MIN;

    DfsMaxDepthUpDown (root, 0);

    return g\_MaxDepth;

}

方法二：【自底向上】

对于任何一个节点，他的深度其实是左树的高度depthLeft与右树的高度depthRight的较大者+1，可以在遍历完之后进行计算，最终直接把最大的值返回。

#define GET\_MAX(x, y) ((x) < (y)) ? (y) : (x)

int DfsMaxDepthDownUp(struct TreeNode\* root)

{

    int depthLeft;

    int depthRight;

    int max = 0;

    if (root == NULL) {

        return 0;

    }

    depthLeft = DfsMaxDepthDownUp(root->left);

depthRight = DfsMaxDepthDownUp(root->right);

//如果直接写成return GET\_MAX(depthLeft, depthRight) + 1;不行

//这个写法也是OK的

//max = GET\_MAX(depthLeft, depthRight);

//return max + 1;

//当前节点的深度为左子树的深度和右子树的深度的最大值

    return GET\_MAX(depthLeft + 1, depthRight + 1);

}

int maxDepth(struct TreeNode\* root)

{

    if (root == NULL) {

        return 0;

    }

    return DfsMaxDepthDownUp(root);

}

## 构造问题

参见一基本篇的“树的构建”

## 节点问题

包括祖先、父子、兄弟等亲戚关系；选取符合要求的节点子集；查找、添加、修改、删除节点。

## 路径问题

前中后序遍历；层次遍历；特殊定义的顺序遍历；路径和

# 三：树的类型