二叉树专题

前言：

说道二叉树，就不得不说递归，很多同学对递归都是又熟悉又陌生，递归的代码一般很简短，但每次都是一看就会，一写就废。

在编写递归方法时，如果没思路，可以先把以下三点理出：

1. 终止条件（特例处理）
2. 递推工作（递归）
3. 返回值

基础概念：

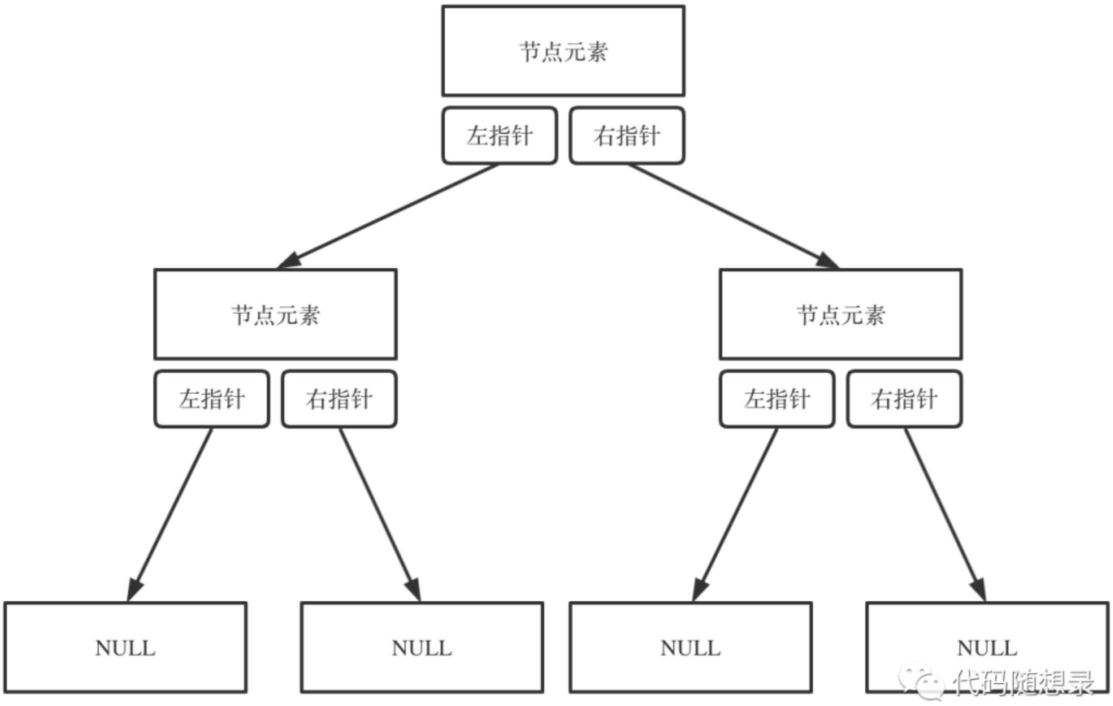
```
度：通俗的讲二叉树中连接节点和节点的线就是度，有n个节点，就有n-1个度，节点数总是比度要多一个，那么度为0的节点一定是叶子节点，因为该节点的下面不再有线；度为1的节点即：该节点只有一个分支；同理度为2的节点就是有两个分支。在二叉树中不可能存在度为3或大于3的节点

树的深度：树的深度就是树中节点的最大层次树。

满二叉树：在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。（如果一棵二叉树只有度为0的结点和度为2的结点，并且度为0的结点在同一层上，则这棵二叉树为满二叉树。）

完全二叉树：完全二叉树：对一颗具有n个结点的二叉树按层编号，如果编号为i(1<=i<=n)的结点与同样深度的满二叉树中编号为i的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。满二叉树一定是完全二叉树，但反过来不一定成立。（在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2h 个节点。）

二叉排序树，又称为二叉查找树、二叉搜索树。二叉排序树或者是一棵空树，或者是具有以下性质的二叉树：若其左子树不为空，则左子树上的所有节点的值均小于它的根结点的值；若其右子树不为空，则右子树上的所有节点的值均大于它的根结点的值；左右子树又分别是二叉排序树。


平衡二叉树（AVL树）：当且仅当两个子树的高度差不超过1时，这个树是平衡二叉树。（同时是排序二叉树）
1. 是二叉排序树
2. 任何一个节点的左子树或者右子树都是平衡二叉树（左右高度差小于等于 1）

红黑树：红黑树是一种平衡二叉查找树的变体，但是红黑树在某些情况下，自旋后并不是一颗平衡二叉树。为什么？因为红黑树在旋转和不旋转之间做了性能的权衡，有些时候，不旋转虽然不平衡，但是旋转带来的性能提升不明显，反而造成为了旋转而带来的开销。它的左右子树高差有可能大于1，所以红黑树不是严格意义上的平衡二叉树（AVL），但对之进行平衡的代价较低， 其平均统计性能要强于 AVL 。
```

二叉树的存储方式

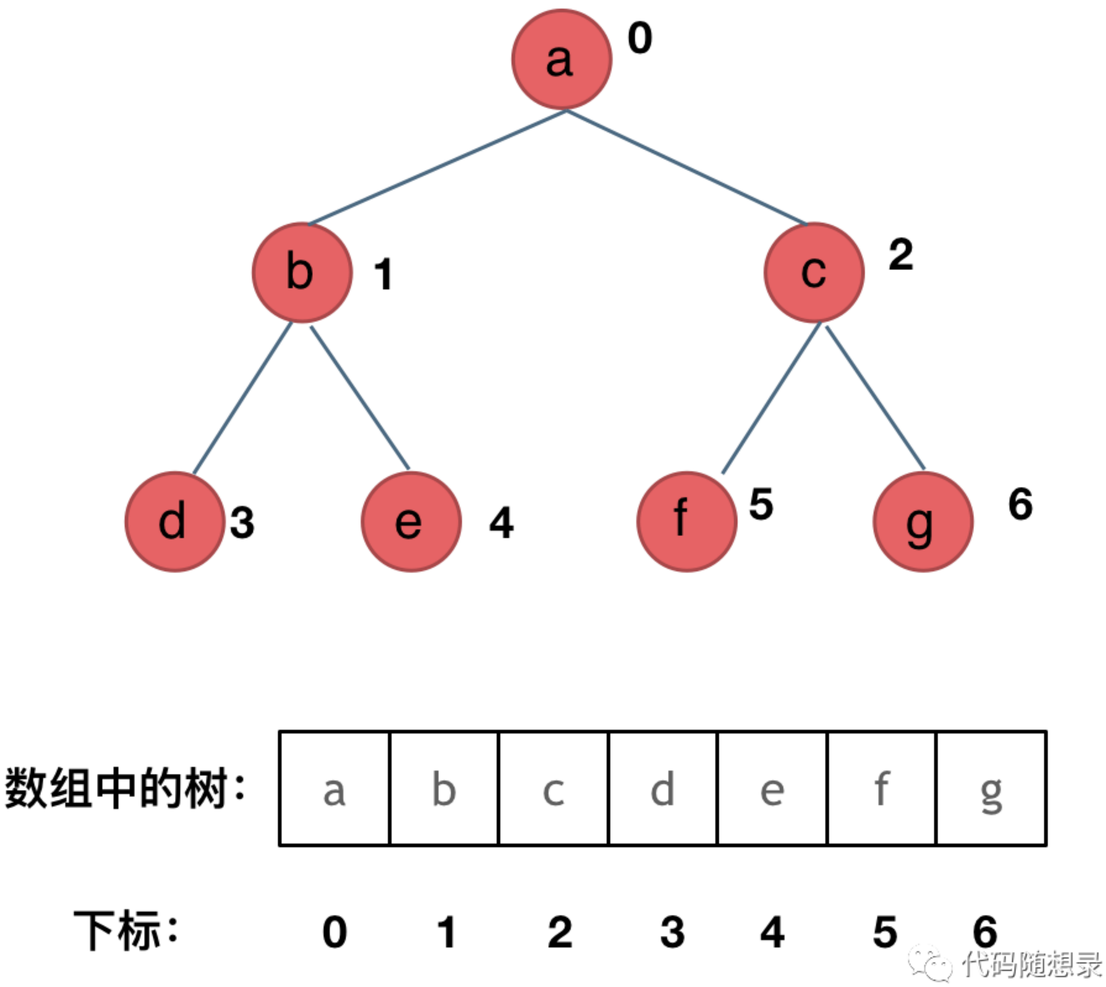
二叉树可以链式存储，也可以顺序存储。： 那么链式存储方式就用指针， 顺序存储的方式就是用数组。

顾名思义就是顺序存储的元素在内存是连续分布的，而链式存储则是通过指针把分布在散落在各个地址的节点串联一起。

链式存储如图：



顺序存储的方式如图：



数组来存储二叉树如何遍历的呢？：**如果父节点的数组下表是i，那么它的左孩子就是i * 2 + 1，右孩子就是 i * 2 + 2。**

二叉树的遍历方式

二叉树主要有两种遍历方式：

1. 深度优先遍历：先往深走，遇到叶子节点再往回走。
2. 广度优先遍历：一层一层的去遍历。

- 深度优先遍历（搜索）DFS
- - 前序遍历（递归法，迭代法）
  - 中序遍历（递归法，迭代法）
  - 后序遍历（递归法，迭代法）
- 广度优先遍历（搜索）BFS
- - 层次遍历（迭代法）

深度优先遍历实例：

剑指 Offer 27. 二叉树的镜像：请完成一个函数，输入一个二叉树，该函数输出它的镜像。

```
//方法一：深度优先搜索（递归法）
//  根据二叉树镜像的定义，考虑递归遍历（dfs）二叉树，交换每个节点的左 / 右子节点，即可生成二叉树的镜像。
//  递归解析：
//      终止条件： 当节点 rootroot 为空时（即越过叶节点），则返回 nullnull ；
//      递推工作：
//      1.初始化节点 tmptmp ，用于暂存 rootroot 的左子节点；
//      2.开启递归右子节点mirrorTree(root.right)mirrorTree(root.right)，并将返回值作为rootroot的左子节点
//      3.开启递归 左子节点 mirrorTree(tmp)mirrorTree(tmp) ，并将返回值作为 rootroot 的 右子节点 。
//      返回值： 返回当前节点 rootroot ；
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null) return null;
        TreeNode tmp = root.left;
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(tmp);
        return root;
    }
}
```

```
//方法二：广度优先搜索(辅助栈或队列)
//  利用栈（或队列）遍历树的所有节点 nodenode ，并交换每个 nodenode 的左 / 右子节点。
//  算法流程：
//  特例处理： 当 rootroot 为空时，直接返回 nullnull ；
//  初始化： 栈（或队列），本文用栈，并加入根节点 rootroot 。
//  循环交换： 当栈 stackstack 为空时跳出；
//  出栈： 记为 nodenode ；
//  添加子节点： 将 nodenode 左和右子节点入栈；
```

```java
//  交换： 交换 nodenode 的左 / 右子节点。
//  返回值： 返回根节点 rootroot 。
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null) return null;
        Stack<TreeNode> stack = new Stack<>() {{ add(root); }};
        while(!stack.isEmpty()) {
            TreeNode node = stack.pop();
            if(node.left != null) stack.add(node.left);
            if(node.right != null) stack.add(node.right);
            TreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;
        }
        return root;
    }
}
```

资料：https://mp.weixin.qq.com/s/_ymfWYvTNd2GvWvC5HOE4A