# 树

**路径：**从树中一个结点到另一个结点之间的分支构成两个结点的路径。

**路径长度：**某条路径上边的条数。

**树的路径长度WPL**：从树根到每一个叶子结点的路径长度 乘以 叶子节点权值 的之和。

# 二叉树

叉不超过2的树

## 概念

**节点度数**：有几个分叉

**整棵树的度数**：度数最大的节点的度数

有结论：任何二叉树里表示度数是i的节点个数，= -1

**平衡二叉树**：对于任何节点，它的左右两个子树的高度差的绝对值不超过1，空树也算平衡二叉树。

**满二叉树**：每一个层的结点数都达到最大值的二叉树,节点数恰好是的二叉树，h是高度

**完全二叉树**：除了最后一层，其余层都达到最大个数，最后一层不一定达最大个数，所以慢二叉树一定也是完全二叉树，但完全不一定是满的

**层序遍历**：相当于广度优先搜索

**前序遍历**：相当于在前面执行代码的深度优先搜索

**中序遍历**：相当于在中间执行代码的dfs深度优先搜索

**后序遍历**：相当于在最后执行代码的dfs深度优先搜索

## 编码细节：

### 索引方式

**指针索引：**就是定义结构体时，储存左右位置的数据用指针表示

1. **struct** Node{
2. **int** sum=0;
3. Node \*son[2];
4. };  //用的时候要一个个实例化节点

**乘法索引：**就是定义结构体时，不储存左右位置的，而是通过乘法获取，对于m叉树,tree[now]代表当前节点的话，tree[now\*m+i]代表左起第i个子树,i∈[0,m-1]

1. **struct** Node{
2. **int** sum=0;
3. };
4. Node tree[MAX];
5. //可以证明tree[now\*2]代表左，tree[now\*2+1]代表右，能严格紧凑的把节点排列在数组里

任何静态的完全二叉树都可以用这种方式，因为完全二叉树按照自上而下自左到右编码顺序正好能满足tree[now\*2]代表左，tree[now\*2+1]代表右，且能严格紧凑的把节点排列在数组里

**拟指针索引：**就是定义结构体时，储存左右位置的数据,且用数组取模拟指针，对于m叉树,tree[now]代表当前节点的话，tree[tree[now].son[i]] 代表左起第i个子树，i∈[0,m-1]

1. **struct** Node{
2. **int** sum,son[2];
3. };
4. Node tree[MAX];

以上三种索引方式，实际用法里效率依次提高，最高的是拟指针索引，因为它一次性申请了一大片数据，相比指针索引(用的时候要一个个实例化节点)，一次性申请MAX个节点比分着申请快，相比乘法索引，不需要通过乘法和加法运算来得到位置。

## 前序中序后序遍历，知其二：

1：首先确立根的位置。   
2：根的左右子树交给递归完成，分析其中的每次找到根径的偏移量

# 广义表