1、满二叉树:如果一棵二叉树**只有度0的结点和度为2**的结点，并且**度为0的结点**在同一层，则称该二叉树为**满二叉树**

结论：深度为k(k>=1)，则结点数为**2^k-1**

2、完全二叉树:在完全二叉树中，除了**最底层结点可能没填满外**，其余**每层节点数**都达到最大值，并且最下面一层的结点都集中在**该层最左边**的若干位置。

优先级队列其实是一个堆，堆就是一棵**完全二叉树**，与此同时满足**父子结点之间的顺序关系**

3、二叉搜索树：

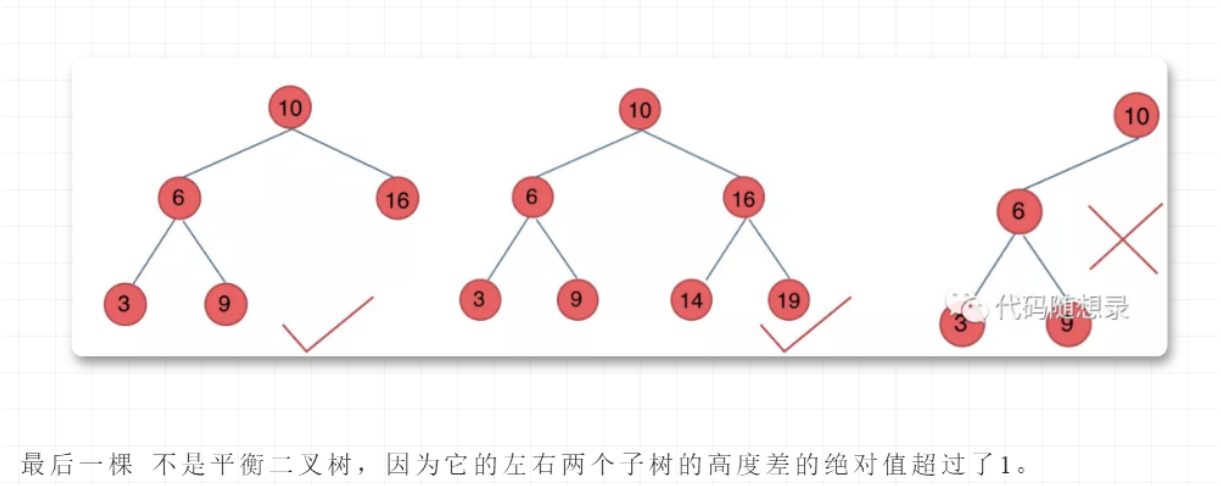
二叉搜索树是一个有序树。

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉排序树

4、平衡二叉树

平衡二叉搜索树，又被称为AVL树

定义：它是一棵空树或者左右子树的高度差的绝对值小于1，且左右子树同时都是平衡二叉树AVL



最后一棵 不是平衡二叉树，因为它的左右两个子树的高度差的绝对值超过了1。

> C++中map、set、multimap、multiset的底层实现都是**平衡二叉树**，增删时间复杂度都是O(logn),而unordered_set和unordered_map的实现是**哈希表**

注意点：

但是用链式表示的二叉树，更有利于我们理解，所以一般我们都是用链式存储二叉树。

「所以大家要了解，用数组依然可以表示二叉树。」

BFS和DFS与二叉树的遍历关系

> 深度优先遍历：(**实现方式:栈和递归**)

先序遍历算法
中序遍历算法
后序遍历算法

以上的“序”是中间节点相对于左子树和右子树的遍历顺序

广度优先遍历:（**实现方式:队列和递归**）
层次遍历算法

一入递归深似海，从此offer是路人

1、确定递归函数的**参数和返回值,(类型)**
2、确定终止条件(运行递归算法的时候，经常会出现栈溢出的错误，就是没有写终止条件或者终止条件写的不对,)
3、确定单层递归的逻辑
确定每一层递归需要处理的信息，在这里也就会重复调用自己来实现递归的过程

二叉树的各种迭代算法

为什么我们可以用栈来实现二叉树
因为递归算法就是每一次递归调用的时候都会把函数的局部变量、参数和返回地址等压入调用栈中，返回时就是弹栈的过程

前序遍历的迭代算法（中左右）

前序遍历是中左右，每次先处理中间结点(此时不需要把根节点放入栈中)，然后将右孩子放入栈中，再加入左孩子。

```
class solution{
 public:
 vector<int>preorderTraversal(TreeNode* root){
 stack<TreeNode*> sta;
 vector<int> result;
 if(root==null) return result;
 sta.push(root);
 result.push_back(root->value);
 while(!sta.empty()){
  TreeNode* cur=sta.top();
  sta.pop();
  result.push_back(cur->value);
  if(cur->right)sta.push(cur->right);//先加入右孩子
  if(cur->left)sta.push(cur->left);//再加入左孩子
  }
  return result;
  }
}
```

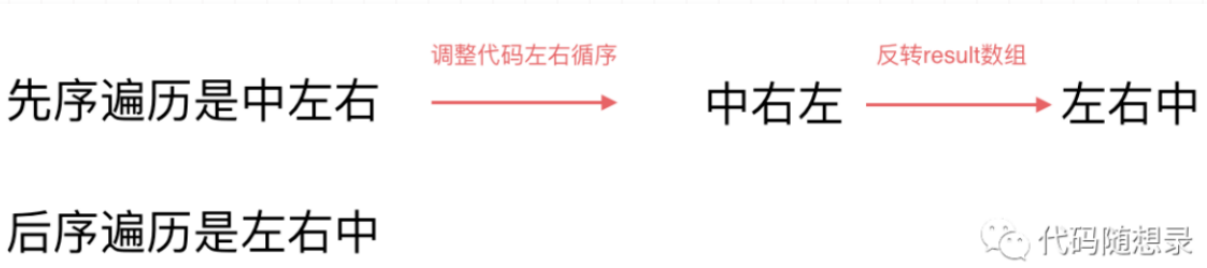
中序遍历的迭代算法（左中右）

由于中序遍历的顺序为**左中右**，只有**当一直向下访问到最左子树**时才开始处理元素，因此必须要用**一个指针的遍历**来帮助访问结点，栈则用来**处理结点上的元素**

```cpp
class Solution{
 public:
 vector<int> inorderTraversal(TreeNode* root){
  stack<TreeNode*> sta;
        vector<int> result;
        if(root==NULL) return result;
        //sta.push(root);
        TreeNode* cur=root;
        while(cur!=NULL||!sta.empty()){
             if(cur){
                 sta.push(cur);//左
                 cur=cur->left;
             }else{
                 cur=sta.top();
                 sta.pop();
                 result.push_back(cur->val);//中
                 cur=cur->right;//右
             }
        }
         return result;
  }
}
```

后序遍历算法（左右中）

> 对于后序遍历算法我们只要调整一下前序遍历算法中**左右子树的入栈顺序，**（先入左树再入右树），然后对于**输出的结果result反转**




```cpp
class Solution{
 public:
 vector<int> postorderTraversal(TreeNode* root){
 stack<TreeNode*> sta;
 vector<int> result;
 if(root==null) return result;
```

```
sta.push(root);
while(!sta.empty()){
TreeNode* node=sta.top();
 sta.pop();
 result.push_back(node);
 if(node->left)sta.push(node->left);
 if(node->right)sta.push(node->right);
 }
 reverse(result.begin(),result.end());
 return result;
 }
}
```

层次遍历算法就是BFS算法

> 层次遍历算法中每次while进入时队列里面存放的都是树中每层的所有元素值

102.二叉树的层序遍历

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。（即逐层地，从左到右访问所有节点）。

> 层次遍历算法就是广度优先算法，只不过应用在二叉树上。

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
 queue<TreeNode*>que;
 if(root!=NULL) que.push(root);
 vector<vector<int>> result;
 while(!que.empty()){
 int size=que.size();
 vector<int> vec;
 for(int i=0;i<size;i++){//一定要用size,que.size的值是在变化的
 TreeNode* node=que.front();
 que.pop();
 vec.push_back(node->val);
 if(node->left) que.push_back(node->left);
 if(node->right) que.push_back(node->righ);
 }
 result.push_back(vec);
 }
 return result;
    }
};
```

根据以上的模板可以完成4道题

107.二叉树的层次遍历 II

给定一个二叉树，返回其节点值自底向上的层次遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

> 根据二叉树的层次遍历然后把result反转一下不就行了

```
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
 vector<vector<int>> result;
 queue<TreeNode*> que;
 if(root==NULL) return result;
 que.push(root);
 while(!que.empty()){
 int size=que.size();
  vector<int> in;
  for(int i=0;i<size;i++){
  TreeNode* t=que.front();
  que.pop();
  in.push_back(t->val);
  if(t->left!=NULL)que.push(t->left);
  if(t->right!=NULL)que.push(t->right);
  }
  result.push_back(in);
  }
  reverse(result.begin(),result.end());
  return result;
    }
};
```

199.二叉树的右视图

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

```
输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

右视图就是每层中的最右边的元素
在内层for循环里面就是每一层的遍历

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
 vector<int> ans;
 queue<TreeNode*> que;
 if(root==null) return ans;
 que.push_back(root);
 while(!que.empty()){
 int size=que.size();
  for(int i=0;i<size;i++){//从队列中弹出的元素依次为该层从左到右的元素
  TreeNode* node=que.front();
  que.pop();
  if(i==size-1)ans.push_back(node->val);
  if(node->left) que.push(node->left);
  if(node->right)que.push(node->right);
  }
  }
  return ans;
    }
```

```
20 };
```

637.二叉树的层平均值

给定一个非空二叉树，返回一个由每层节点平均值组成的数组。

示例 1：

输入：

```
    3
   / \
  9  20
    /  \
   15   7
```

输出：[3, 14.5, 11]

解释：

第 0 层的平均值是 3 ， 第1层是 14.5 ，第2层是 11 。因此返回 [3, 14.5, 11] 。

每层元素求和取平均值

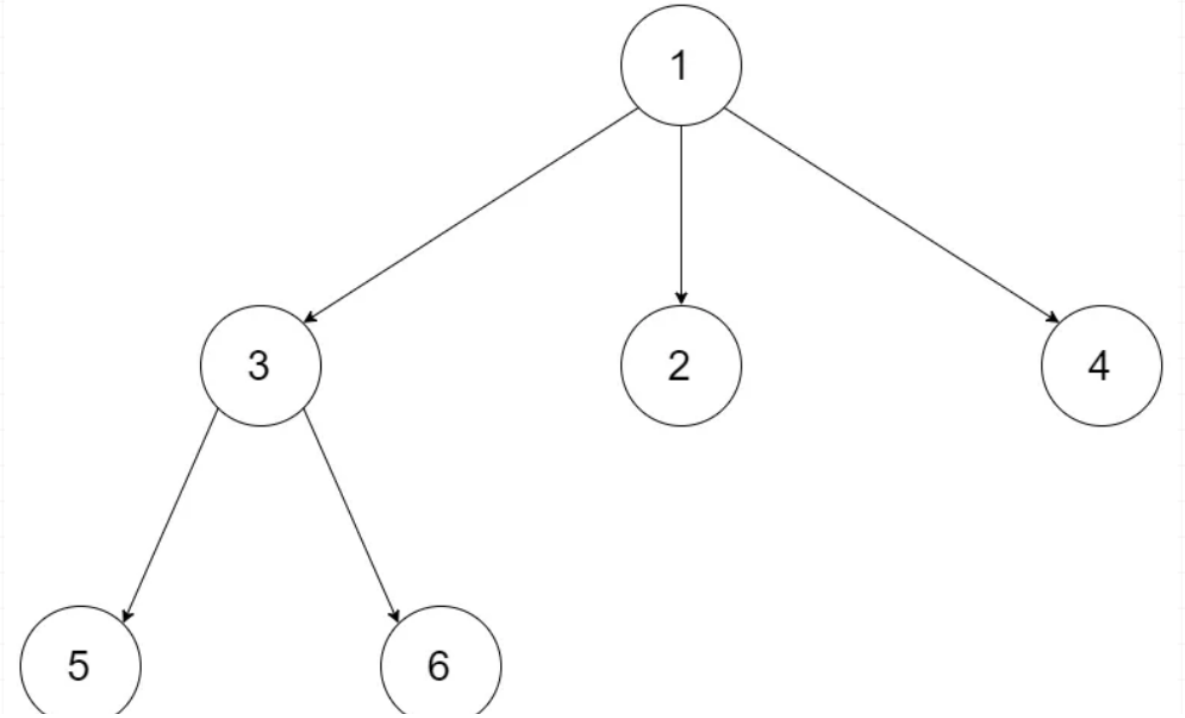
```cpp
class Solution {
public:
    vector<double> averageOfLevels(TreeNode* root) {
 queue<TreeNode*> que;
 vector<double> ans;
 if(root==NULL) return ans;
 que.push(root);
 while(!que.empty()){
 int size=que.size();
  double a=0;
  for(int i=0;i<size;i++){
  TreeNode* node=que.front();
  que.pop();
  a+=node->val;
  if(node->left)que.push(node->left);
  if(node->right)que.push(node->right);
  }
  ans.push_back(a/=size);
  }
```

```
20   return ans;
21       }
22  };
```

429.N叉树的层序遍历

给定一个 N 叉树，返回其节点值的层序遍历。(即从左到右，逐层遍历)。

例如，给定一个 3叉树 ：




```
1  class Solution {
2  public:
3      vector<vector<int>> levelOrder(Node* root) {
4          queue<Node*> que;
5   vector<vector<int>> ans;
6   if(root==NULL)return ans;
7   que.push(root);
8   while(!que.empty()){
9   int size=que.size();
10   vector<int> t;
11   for(int i=0;i<size;i++){
12   Node* node=que.front();
13   que.pop();
14   t.push_back(node->val);
```

```
 int nodeSize=node->children.size();
 for(int j=0;j<nodeSize;j++){
 que.push(node->children[j]);
 }
 }
 ans.push_back(t);
 }
 return ans;
    }
};
```