1、树的含义：

树状图是一种数据结构，它是由n（n>=1）个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树。

1）每个元素称为结点（node）；

（2）有一个特定的结点被称为根结点或树根（root）。

（3）除根结点之外的其余数据元素被分为m（m≥0）个互不相交的集合T1，T2，……Tm-1，其中每一个集合Ti（1<=i<=m）本身也是一棵树，被称作原树的子树（subtree）。

2、树的分类：

二叉树：它是由n（n>=0）个有限结点的有限集合，该集合或者为空集，或者由一个根节点和两颗互不相交的分别为根节点的左字数和有子树的二叉树组成。

完全二叉树：完全二叉树是效率很高的数据结构，完全二叉树是由满二叉树而引出来的。对于深度为K的，有n个结点的二叉树，当且仅当其每一个结点都与深度为K的满二叉树中编号从1至n的结点一一对应时称之为完全二叉树。

满二叉树：除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树。

二叉搜索树：二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 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。

附上操作时间复杂度的图：

Data Structure	Time Complexity								Space Comple
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

3、相关的例题：

1、给定一个二叉树，判断其是否是一个有效的二叉搜索树。假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。（https://leetcode-cn.com/problems/validate-binary-search-tree/）

package tree;

public class BinarySearchTree {

```
//方法一：递归

```

// 复杂度分析

//

// 时间复杂度 : O(N)。每个结点访问一次。

// 空间复杂度 : O(N)。我们跟进了整棵树。

public boolean isValidBST(TreeNode root) {

return helper(root, null, null);

}

```
private boolean helper(TreeNode node, Integer low, Integer high) {
        // TODO Auto-generated method stub
        if (node==null)  return true;//空树也是二叉搜索树
        int val=node.val;
        if (low!=null&& val<=low) return false;
        if(high!=null&& val>high)return false;


    if (! helper(node.right, val, high)) return false;
    if (! helper(node.left, low, val)) return false;
    return true;

}

```

}

2、最近公共祖先:(https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/submissions/)

定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

时间复杂度：O(N)，N 是二叉树中的节点数，最坏情况下，我们需要访问二叉树的所有节点。

空间复杂度：O(N)，这是因为递归堆栈使用的最大空间位 N,斜二叉树的高度可以是 N。

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root==null|| p==root||q==root) return root;

```
    TreeNode left=lowestCommonAncestor(root.left, p, q);
    TreeNode right=lowestCommonAncestor(root.right, p, q);
    return left==null?right:right==null?left:root; //如果左右节点都不为空，则上级的节点就是最近公共祖先
}

```

}

3、二叉搜索树的最近公共祖先（https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/）

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

方法一：递归

从根节点开始遍历树

如果节点 pp 和节点 qq 都在右子树上，那么以右孩子为根节点继续 1 的操作

如果节点 pp 和节点 qq 都在左子树上，那么以左孩子为根节点继续 1 的操作

如果条件 2 和条件 3 都不成立，这就意味着我们已经找到节 pp 和节点 qq 的 LCA 了

时间复杂度：O(N)

其中 NN 为 BST 中节点的个数，在最坏的情况下我们可能需要访问 BST 中所有的节点。

空间复杂度：O(N)

所需开辟的额外空间主要是递归栈产生的，之所以是 N 是因为 BST 的高度为 N。

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

```
  if(root==null) return root;
    // Value of current node or parent node.
    int parentVal = root.val;

    // Value of p
    int pVal = p.val;

    // Value of q;
    int qVal = q.val;

    if (pVal > parentVal && qVal > parentVal) {
        // If both p and q are greater than parent
        return lowestCommonAncestor(root.right, p, q);
    } else if (pVal < parentVal && qVal < parentVal) {
```

```
        // If both p and q are lesser than parent
        return lowestCommonAncestor(root.left, p, q);
    } else {
        // We have found the split point, i.e. the LCA node.
        return root;
    }
}

```

方法二：迭代法

这个方法跟方法一很接近。唯一的不同是，我们用迭代的方式替代了递归来遍历整棵树。由于我们不需要回溯来找到 LCA 节点，所以我们是完全可以不利用栈或者是递归的。实际上这个问题本身就是可以迭代的，我们只需要找到分割点就可以了。这个分割点就是能让节点 pp 和节点 qq 不能在同一颗子树上的那个节点，或者是节点 pp 和节点 qq 中的一个，这种情况下其中一个节点是另一个节点的父亲节点。

时间复杂度：O(N)

其中 NN 为 BST 中节点的个数，在最坏的情况下我们可能需要遍历 BST 中所有的节点。

空间复杂度：O(1)

//方法二：迭代法

public TreeNode lowestCommonAncestor3(TreeNode root, TreeNode p, TreeNode q) {

```
    // Value of p
    int pVal = p.val;

    // Value of q;
    int qVal = q.val;

    // Start from the root node of the tree
    TreeNode node = root;

    // Traverse the tree
    while (node != null) {

        // Value of ancestor/parent node.
        int parentVal = node.val;

        if (pVal > parentVal && qVal > parentVal) {
            // If both p and q are greater than parent
            node = node.right;
```

```
        } else if (pVal < parentVal && qVal < parentVal) {
            // If both p and q are lesser than parent
            node = node.left;
        } else {
            // We have found the split point, i.e. the LCA node.
            return node;
        }
    }
    return null;
}

```

3、二叉树的遍历

package tree;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

public class Traversal {

```
//方法 1：迭代,深度优先搜索
//从根节点开始，每次迭代弹出当前栈顶元素，并将其孩子节点压入栈中，先压右孩子再压左孩子。

```

//在这个算法中，输出到最终结果的顺序按照 Top->Bottom 和 Left->Right，符合前序遍历的顺序。

// 时间复杂度：访问每个节点恰好一次，时间复杂度为 O(N)O(N) ，其中 NN 是节点的个数，也就是树的大小。

// 空间复杂度：取决于树的结构，最坏情况存储整棵树，因此空间复杂度是 O(N)O(N)。

```
public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result=new ArrayList();
        LinkedList<TreeNode> stack = new LinkedList<>();
        if (root==null) return result;

        stack.push(root);
        while(!stack.isEmpty()){
                TreeNode node = stack.pop();
                result.add(node.val);
                //压右孩子进栈
                if(node.right!=null){
                        stack.push(node.right);
                }
```

```
                if (node.left != null) {
                    stack.add(node.left);
                }
        }
        return result;

    }

//中序遍历 ，使用迭代
//时间复杂度：O(n)O。

```

//空间复杂度：O(n))。

```
public List<Integer> Traversal(TreeNode root) {
        List<Integer> result=new ArrayList();
        LinkedList<TreeNode> stack = new LinkedList<>();
        if (root==null) return result;
        TreeNode temp=root;
        while(!stack.isEmpty()|| temp!=null){

                while(temp!=null){   //左节点全部入栈
                        stack.push(temp);
                        temp=temp.left;
                }

                temp = stack.pop();
                result.add(temp.val);
                //压右孩子进栈
                temp=temp.right;
        }
        return result;

    }


//使用递归

public List < Integer > inorderTraversal(TreeNode root) {
        List < Integer > res = new ArrayList < > ();
        helper(root, res);
```

```
28          return res;
29      }
30
31      public void helper(TreeNode root, List < Integer > res) {
32          if (root == null) return;
33          if (root.left != null) {
34                  helper(root.left, res);
35            }
36              res.add(root.val);
37              if (root.right != null) {
38                  helper(root.right, res);
39              }
40
41      }
42
43
44
45      //后序遍历,使用迭代
46      //从根节点开始依次迭代，弹出栈顶元素输出到输出列表中，然后依次压入它的所有孩子节点，按照从上到下、从左至右的顺序依次压入栈中。
47
```

//因为深度优先搜索后序遍历的顺序是从下到上、从左至右，所以需要将输出列表逆序输出。

//其实就是根-右-左的逆序

```
1       public List<Integer> postorderTraversal(TreeNode root) {
2           LinkedList<TreeNode> stack = new LinkedList<>();
3           LinkedList<Integer> output = new LinkedList<>();
4           if (root == null) {
5             return output;
6           }
7
8           stack.add(root);
9           while (!stack.isEmpty()) {
10            TreeNode node = stack.pollLast();
11            output.addFirst(node.val);
12            if (node.left != null) {
13              stack.add(node.left);
14            }
15            if (node.right != null) {
```

```
            stack.add(node.right);
          }
        }
        return output;
      }

}
```
