- 需要的基础知识
- 高质量的代码
- 解决面试题的思路
- 优化时间和空间效率

目录

思路及代码实现

代码中用到的数据结构

- 链表

```
public class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}
```

- 二叉树

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}
```

1. 赋值运算函数

思路：

- 将返回值类型声明为该类型的引用
- 把传入的参数类型声明为常量引用
- 释放实例自身已有的内存
- 判断传入的参数和当前的实例是不是同一个实例

代码实现：略

2. 单例设计模式

题目描述：设计一个类，只能生成该类的一个实例。

思路：非线程安全与线程安全

代码实现：

- 线程安全的懒汉式：静态内部类

```
public class Singleton {
    private static class SingletonHodler {
        private static Singleton ourInstance = new Singleton();
    }
    public static Singleton getInstance() {
        return SingletonHodler.ourInstance;
    }
    private Singleton() {
    }
}
```

3. 二维数组中查找目标值

题目描述：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

思路：从右上角或左下角开始找，逐行排除，或者用二分法查找

代码实现：

- 解法一：双指针，时间复杂度：O(mn)，空间复杂度：O(1)

```
public static boolean find1(int[][] array, int target) {
    if (array == null || array.length == 0) {
        return false;
    }
```

```
    int row = 0;
    int column = array[0].length - 1;

    while (row < array.length && column >= 0) {
        if (array[row][column] == target) {
            return true;
        }
        if (array[row][column] > target) {
            column--;
        } else {
            row++;
        }
    }
    return false;
}
```

- 解法二：二分法，时间复杂度：O(log mn)，空间复杂度：O(1)

```
public boolean find2(int[][] array, int target) {
    if (array == null || array.length == 0) {
        return false;
    }

    int left = 0;
    int right = array.length * array[0].length - 1;
    int col = array[0].length;

    while (left <= right) {
        int mid = (left + right) / 2;
        int value = array[mid / col][mid % col];

        if (value == target) {
            return true;
        } else if (value < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

4. 替换字符串中的空格

题目描述：将一个字符串中的空格替换成“%20”。例如：当字符串为 We Are Happy.则经过替换之后的字符串为 We%20Are%20Happy。

思路：从后往前复制，数组长度会增加，或使用 StringBuilder、StringBuffer 类

代码实现：时间复杂度：O(n)，空间复杂度：O(n)

```
public String replaceSpace(StringBuffer str) {
    if (str == null)
        return null;
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < str.length(); i++) {
        if (String.valueOf(str.charAt(i)).equals(" ")) {
            sb.append("%20");
        }else {
            sb.append(str.charAt(i));
        }
    }
    return String.valueOf(sb);
}
```

5. 从尾到头打印链表

题目描述：输入一个链表，从尾到头打印链表每个节点的值。

思路：借助栈实现，或使用递归的方法。

代码实现：

- 解法一：借用栈

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
    ArrayList<Integer> list = new ArrayList<>();
    Stack<ListNode> stack = new Stack<>();
    while (headNode != null) {
        stack.push(headNode);
        headNode = headNode.next;
    }

    while (!stack.isEmpty()) {
        list.add(stack.pop().val);
    }
    return list;
}
```

- 解法二：递归

```
public static ArrayList<Integer> printListReverse2(ListNode headNode) {
    ArrayList<Integer> list = new ArrayList<Integer>();

    if (headNode != null) {
        if (headNode.next != null) {
```

```
            list = printListReverse2(headNode.next);
        }
        list.add(headNode.val);
    }
    return list;
}
```

6. 由前序和中序遍历重建二叉树

题目描述：输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

思路：先找出根节点，然后利用递归方法构造二叉树

代码实现：时间复杂度：O(n)，空间复杂度：O(n)

- 解法一：递归（传入数组的拷贝）

```
public TreeNode reConstructBinaryTree(int[] pre, int[] in) {
    if (pre == null || in == null || pre.length == 0 || in.length == 0)
 {
        return null;
    }
    if (pre.length != in.length) {
        return null;
    }

    TreeNode root = new TreeNode(pre[0]);
    for (int i = 0; i < pre.length; i++) {
        if (pre[0] == in[i]) {
            root.left = reConstructBinaryTree(Arrays.copyOfRange(pre, 1,
 i + 1), Arrays.copyOfRange(in, 0, i));
            root.right = reConstructBinaryTree(Arrays.copyOfRange(pre,
i + 1, pre.length), Arrays.copyOfRange(in, i + 1, in.length));
        }
    }
    return root;
}
```

- 解法二：递归（传入子数组的边界索引）

```
public TreeNode reConstructBinaryTree2(int[] preorder, int[] inorder) {
    if (preorder == null || preorder.length == 0 ||
            inorder == null || inorder.length == 0) return null;
    return helper(preorder, 0, preorder.length - 1, inorder, 0, inorder.
length - 1);
}
```

```
private TreeNode helper(int[] preorder, int preL, int preR, int[] inord
er, int inL, int inR) {
    if (preL > preR || inL > inR) {
        return null;
    }
    int rootVal = preorder[preL];
    int index = 0;
    while (index <= inR && inorder[index] != rootVal) {
        index++;
    }
    TreeNode root = new TreeNode(rootVal);
    root.left = helper(preorder, preL + 1, preL - inL + index, inorder,
 inL, index);
    root.right = helper(preorder, preL - inL + index + 1, preR, inorder,
 index + 1, inR);
    return root;
}
```

- **拓展题**：根据中序遍历与后序遍历构造二叉树。

7. 用两个栈实现队列

题目描述：用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。 队列中的元素为 int 类型。

思路：一个栈压入元素，而另一个栈作为缓冲，将栈 1 的元素出栈后压入栈 2 中。也可以将栈 1 中的最后一个元素直接出栈，而不用压入栈 2 中再出栈。

代码实现：

```
// 入栈，时间复杂度：O(1)，空间复杂度：O(n)
public void push(int node) {
    stack1.push(node);
}

// 出栈，时间（摊还）复杂度：O(1)，空间复杂度：O(1)
public int pop() throws Exception {
    if (stack1.isEmpty() && stack2.isEmpty()) {
        throw new Exception("栈为空！");
    }

    if (stack2.isEmpty()) {
        while(!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
```

```
    return stack2.pop();
}
```

8. 求旋转数组的最小数字

题目描述：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为 1。 NOTE：给出的所有元素都大于 0，若数组大小为 0，请返回-1。假设数组中不存在重复元素。

思路：利用二分法，找到数组的中间元素 mid。如果中间元素 > 数组第一个元素，在 mid 右边搜索变化点。如果中间元素 < 数组第一个元素，我们需要在 mid 左边搜索变化点。当找到变化点时停止搜索，满足 nums[mid] > nums[mid + 1]（mid+1 是最小值）或 nums[mid - 1] > nums[mid]（ mid 是最小值）即可。

代码实现：

- 解法一：二分查找（变化点），时间复杂度：O(log n)，空间复杂度：O(1)

```
public static int minInReversingList(int[] array) {
    if (array == null || array.length == 0) {
        return -1;
    }
    if (array.length == 1 || array[array.length - 1] > array[0]) {
        return array[0];
    }

    int left = 0;
    int right = array.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (array[mid] > array[mid + 1]) {
            return array[mid + 1];
        }
        if (array[mid - 1] > array[mid]) {
            return array[mid];
        }

        if (array[mid] > array[0]) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

- 解法二：二分查找（最左下标），时间复杂度：O(log n)，空间复杂度：O(1)

```
public int minInReversingList(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return nums[left];
}
```

- **拓展题**：若非递减排序数组中有重复元素，求最小元素的解法

9. 斐波那契数列的应用

9.1 输出斐波那契数列的第 n 项

题目描述：现在要求输入一个整数 n，请你输出斐波那契数列的第 n 项。n<=39

思路：递归的效率低，使用循环方式。

代码实现：

```
public long fibonacci(int n) {
    long result=0;
    long preOne=1;
    long preTwo=0;
    if(n==0) {
        return 0;
    }
    if(n==1) {
        return 1;
    }
    for (int i = 2; i <= n; i++) {
        result = preOne+preTwo;
        preTwo = preOne;
        preOne = result;
    }
    return result;
}
```

9.2. 青蛙跳台阶(1 或 2 级)

题目描述：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：斐波那契数列思想

代码实现：

```
public int JumpFloor(int n) {
    if(n < 3) {
        return n;
    }
    int result=0;
    int preOne=2;
    int preTwo=1;
    for (int i = 3; i <= n; i++) {
        result = preOne+preTwo;
        preTwo = preOne;
        preOne = result;
    }
    return result;
}
```

9.3 小矩形无重叠覆盖大矩形

题目描述：我们可以用 2X1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2X1 的小矩形无重叠地覆盖一个 2Xn 的大矩形，总共有多少种方法？

思路：斐波那契数列思想

代码实现：

```
public int Fibonaccik(int n) {
    int number = 1;
    int sum = 1;
    if (n <= 0)
        return 0;
    if (n == 1 ) {
        return 1;
    }

    while (n-- >= 2) {
        sum += number;
        number = sum - number;
    }
    return sum;
}
```

9.4. 青蛙跳台阶(n 级)

题目描述：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：2^(n-1)

代码实现：

```
public int JumpFloor2(int target) {
    return (int) Math.pow(2,target-1);
}
```

10. 二进制中 1 的个数

题目描述：输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

思路：a&(a-1)的结果会将 a 最右边的 1 变为 0，直到 a = 0，还可以先将 a&1 != 0，然后右移 1 位，但不能计算负数的值，

代码实现：

```
public int NumberOf1(int n) {
    int count = 0;
    while (n != 0) {
        count++;
        n = (n-1) & n;
    }
    return count;
}
```

11. 数值的整数次方

题目描述：给定一个 double 类型的浮点数 base 和 int 类型的整数 exponent。求 base 的 exponent 次方。不得使用库函数，不需要考虑大数问题

思路：不能用==比较两个浮点数是否相等，因为有误差。考虑输入值的多种情况。

代码实现：

```
public double Power(double base, int exponent) {
    double res = 0;
    if (equal(base,0)) {
        return 0;
    }
```

```
    if (exponent == 0) {
        return 1.0;
    }
    if (exponent > 0) {
        res = mutiply(base,exponent);
    }else {
        res = mutiply(1/base,-exponent);
    }
    return res;
}

public double mutiply(double base, int e) {
    double sum = 1;
    for (int i = 0; i < e; i++) {
        sum = sum * base;
    }
    return sum;
}

public boolean equal(double a, double b) {
    if (a - b < 0.000001 && a - b > -0.000001) {
        return true;
    }
    return false;
}
```

12. 求 1 到最大的 n 位数

题目描述：输入数字 n，按顺序打印从 1 到最大的 n 位数十进制数，比如：输入 3，打印出 1 到 999.

思路：考虑大数问题，使用字符串或数组表示。

代码实现：

```
public void printToMaxOfNDigits(int n) {
    int[] array=new int[n];
    if(n <= 0)
        return;
    printArray(array, 0);
}

private void printArray(int[] array,int n) {
    for(int i = 0; i < 10; i++) {
        if(n != array.length) {
            array[n] = i;
            printArray(array, n+1);
        } else {
            boolean isFirstNo0 = false;
```

```
            for(int j = 0; j < array.length; j++) {
                if(array[j] != 0) {
                    System.out.print(array[j]);
                    if(!isFirstNo0)
                        isFirstNo0 = true;
                } else {
                    if(isFirstNo0)
                        System.out.print(array[j]);
                }
            }
            System.out.println();
            return ;
        }
    }
}
```

13. O(1)时间删除链表节点

题目描述：给定单向链表的头指针和一个节点指针，在 O(1)时间复杂度内删除该节点。

思路：将要删除节点的下一个节点的值赋给要删除的节点，然后指向下下一个节点

代码实现：

```
public void deleteNode(ListNode head, ListNode deListNode) {
    if (deListNode == null || head == null)
        return;

    if (head == deListNode) {
        head = null;
    } else {
        // 若删除节点是末尾节点，往后移一个
        if (deListNode.nextNode == null) {
            ListNode pointListNode = head;
            while (pointListNode.nextNode.nextNode != null) {
                pointListNode = pointListNode.nextNode;
            }
            pointListNode.nextNode = null;
        } else {
            deListNode.data = deListNode.nextNode.data;
            deListNode.nextNode = deListNode.nextNode.nextNode;
        }
    }
}
```

14. 将数组中的奇数放在偶数前

题目描述：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变

思路：使用双指针法，或移动偶数位置

代码实现：时间复杂度 O（n)，空间复杂度 O（1)

```
public void reOrderArray(int[] array) {
    if (array == null || array.length == 0) {
        return;
    }

    int left = 0;
    int right = array.length - 1;
    while (left < right) {
        while (left < right && array[left] % 2 != 0) {
            left++;
        }
        while (left < right && array[right] % 2 == 0) {
            right--;
        }

        if (left < right) {
            int tmp = array[left];
            array[left] = array[right];
            array[right] = tmp;
        }
    }
}
```

15. 求链表中倒数第 K 个节点

题目描述：输入一个链表，输出该链表中倒数第 k 个结点。

思路：定义一快一慢两个指针，快指针走 K 步，然后慢指针开始走，快指针到尾时，慢指针就找到了倒数第 K 个节点。

代码实现：时间复杂度：O(n)，空间复杂度：O(1)

```
public ListNode findKthToTail(ListNode head, int k) {
    if (head == null || k < 1) {
        return null;
    }
    ListNode fast = head;
```

```
    ListNode slow = head;
    while (k-- > 1) {
        if (fast.next == null) {
            return null;
        }
        fast = fast.next;
    }
    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```

- **拓展题**：给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

16. 输出反转后的链表

题目描述：输入一个链表，反转链表后，输出新链表的表头。

思路：定义两个指针，反向输出

代码实现：

- 解法一：迭代：两个指针，反向输出，时间复杂度：O(n)，空间复杂度：O(1)

```
public ListNode reverseList(ListNode head) {
    ListNode pre = null;
    ListNode curr = head;

    while (curr != null) {
        ListNode tmp = curr.next;
        curr.next = pre;
        pre = curr;
        curr = tmp;
    }
    return pre;
}
```

- 解法二：递归，时间复杂度：O(n)，空间复杂度：O(n)

```
public ListNode reverseList2(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }

    ListNode p = reverseList3(head.next);
    head.next.next = head.next;
```

```
    head.next = null;
    return p;
}
```

17. 合并两个有序链表

题目描述：输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

思路：递归与非递归求解，小数放在前面。

代码实现：

- 解法一：递归，时间复杂度：O(m+n)，空间复杂度：O(m+n)

```
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    if (list1 == null) {
        return list2;
    }
    if (list2 == null) {
        return list1;
    }

    if (list1.val < list2.val) {
        list1.next = mergeTwoLists(list1.next, list2);
        return list1;
    } else {
        list2.next = mergeTwoLists(list1, list2.next);
        return list2;
    }
}
```

- 解法二：迭代，时间复杂度：O(m+n)，空间复杂度：O(1)

```
public ListNode mergeTwoLists2(ListNode list1, ListNode list2) {
    ListNode preHead = new ListNode(-1);
    ListNode pre = preHead;

    while (list1 != null && list2 != null) {
        if (list1.val < list2.val) {
            pre.next = list1;
            list1 = list1.next;
        } else {
            pre.next = list2;
            list2 = list2.next;
        }
        pre = pre.next;
    }
```

```
    pre.next = list1 == null ? list2 : list1;
    return preHead.next;
}
```

18. 判断二叉树 A 中是否包含子树 B

题目描述：输入两棵二叉树 A，B，判断 B 是不是 A 的子结构。（ps：我们约定空树不是任意一个树的子结构）

思路：若根节点相等，利用递归比较他们的子树是否相等，若根节点不相等，则利用递归分别在左右子树中查找。

代码实现：

```
public boolean hasSubTree(TreeNode source, TreeNode target) {
    if (target == null) {
        return true;
    }
    if (source == null) {
        return false;
    }

    if (doesTree1HaveTree2(source, target)) {
        return true;
    }
    return hasSubTree(source.left, target) || hasSubTree(source.right,
target);
}

public static boolean doesTree1HaveTree2(TreeNode source, TreeNode targ
et) {
    if (source == null && target == null) {
        return true;
    }
    if (source == null || target == null) {
        return false;
    }
    if (source.val != target.val) {
        return false;
    }

    return doesTree1HaveTree2(source.left, target.left) && doesTree1Hav
eTree2(source.right, target.right);
}
```

19. 二叉树的镜像

题目描述：操作给定的二叉树，将其变换为源二叉树的镜像。

思路：使用递归或非递归方式交换每个节点的左右子树位置。

代码实现：

- 解法一：递归，间复杂度：O(n)，空间复杂度：O(n)

```
public boolean isSymmetric(TreeNode root) {
    if (root == null) {
        return true;
    }
    return isMirror(root.left, root.right);
}

private boolean isMirror(TreeNode leftNode, TreeNode rightNode) {
    if (leftNode == null && rightNode == null) {
        return true;
    }
    if (leftNode == null || rightNode == null) {
        return false;
    }
    return leftNode.val == rightNode.val && isMirror(leftNode.left, rig
htNode.right) && isMirror(leftNode.right, rightNode.left);
}
```

- 解法二：迭代，时间复杂度：O(n)，空间复杂度：O(n)

```
public boolean isSymmetric2(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    stack.push(root);
    while (!stack.isEmpty()) {
        TreeNode t1 = stack.pop();
        TreeNode t2 = stack.pop();
        if (t1 == null && t2 == null) {
            continue;
        }
        if (t1 == null || t2 == null) {
            return false;
        }
        if (t1.val != t2.val) {
            return false;
        }
        stack.push(t1.left);
        stack.push(t2.right);
        stack.push(t1.right);
        stack.push(t2.left);
    }
    return true;
}
```

20. 顺时针打印矩阵

题目描述：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下 4X4 矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

思路：按层模拟：终止行号大于起始行号，终止列号大于起始列号，

代码实现：时间复杂度：O(n)，空间复杂度：O(n)，其中 n 表示矩阵元素个数

```
public List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> res = new ArrayList<>();
    if (matrix == null || matrix.length == 0) {
        return res;
    }

    int r1 = 0, r2 = matrix.length - 1;
    int c1 = 0, c2 = matrix[0].length - 1;
    while (r1 <= r2 && c1 <= c2) {
        // 从左往右
        for (int c = c1; c <= c2; c++) {
            res.add(matrix[r1][c]);
        }
        // 从上往下
        for (int r = r1 + 1; r <= r2; r++) {
            res.add(matrix[r][c2]);
        }
        // 判断是否会重复打印
        if (r1 < r2 && c1 < c2) {
            // 从右往左
            for (int c = c2 - 1; c > c1; c--) {
                res.add(matrix[r2][c]);
            }
            // 从下往上
            for (int r = r2; r > r1; r--) {
                res.add(matrix[r][c1]);
            }
        }
        r1++;
        r2--;
        c1++;
        c2--;
    }
    return res;
}
```

21. 包含 main 函数的栈

题目描述：定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 min 函数。

思路：定义两个栈，一个存放入的值。另一个存最小值。

代码实现：

```
public void push(int node) {
    stack1.push(node);
    if (stack2.isEmpty()) {
        stack2.push(node);
    }else {
        if (stack2.peek() > node) {
            stack2.push(node);
        }
    }
}

public void pop() {
    if (stack1.pop() == stack2.peek()) {
        stack2.pop();
    }
}

public int top() {
    return stack1.peek();
}

public int min() {
    return stack2.peek();
}
```

22. 判断一个栈是否是另一个栈的弹出序列

题目描述：输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4，5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

思路：用栈来压入弹出元素，相等则出栈。

代码实现：

```
public boolean IsPopOrder(int [] pushA,int [] popA) {
```

```
    if (pushA == null || popA == null) {
        return false;
    }

    Stack<Integer> stack = new Stack<>();
    int index = 0;
    for (int i = 0; i < pushA.length; i++) {
        stack.push(pushA[i]);
        while (!stack.isEmpty() && stack.peek() == popA[index]) {
            stack.pop();
            index++;
        }
    }
    return stack.isEmpty();
}
```

23.层序遍历二叉树

题目描述：从上往下打印出二叉树的每个节点，同层节点从左至右打印。

思路：利用队列（链表）辅助实现。

代码实现：

- 解法一：迭代，间复杂度：O(n)，空间复杂度：O(n)

```
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
    ArrayList<Integer> list = new ArrayList<>();
    if (root == null) {
        return list;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        list.add(node.val);
        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }
    return list;
}
```

- 解法二：递归，间复杂度：O(n)，空间复杂度：O(n)

```
public ArrayList<Integer> PrintFromTopToBottom2(TreeNode root) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    if (root == null) {
        return list;
    }

    list.add(root.val);
    levelOrder(root, list);
    return list;
}

public void levelOrder(TreeNode root, ArrayList<Integer> list) {
    if (root == null) {
        return;
    }

    if (root.left != null) {
        list.add(root.left.val);
    }
    if (root.right != null) {
        list.add(root.right.val);
    }
    levelOrder(root.left, list);
    levelOrder(root.right, list);
}
```

- **拓展题**：层序遍历按层返回（即返回类型是 List<List)

24. 后序遍历二叉搜索树

题目描述：输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出 Yes,否则输出 No。假设输入的数组的任意两个数字都互不相同。

思路：先找到右子树的开始位置，然后分别进行左右子树递归处理。

代码实现：

```
public boolean VerifySquenceOfBST(int[] sequence) {
    if (sequence == null || sequence.length == 0) {
        return false;
    }

    int rstart = 0;
    int rootIndex = sequence.length - 1;
    for (int i = 0; i < rootIndex; i++) {
        if (sequence[i] < sequence[rootIndex])
```

```
            rstart++;
        }

        if (rstart == 0) {
            VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,rootIndex));
            return true;
        }

        for (int i = rstart; i < rootIndex; i++) {
            if (sequence[i] <= sequence[rootIndex]) {
                return false;
            }
        }
        VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,rstart));
        VerifySquenceOfBST(Arrays.copyOfRange(sequence,rstart,rootIndex));
        return true;
    }
```

25. 二叉树中和为某值的路径

题目描述：输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

思路：先保存根节点，然后分别递归在左右子树中找目标值，若找到即到达叶子节点，打印路径中的值

代码实现：

```
private ArrayList<ArrayList<Integer>> listAll = new ArrayList<>();
private ArrayList<Integer> list = new ArrayList<>();

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root, int target)
 {
    if (root == null) {
        return listAll;
    }
    list.add(root.val);
    target -= root.val;

    if (target == 0 && root.left == null && root.right == null) {
        listAll.add(new ArrayList<>(list));
    }
    FindPath(root.left, target);
    FindPath(root.right, target);
    // 回退
    list.remove(list.size() - 1);
```

```
    return listAll;
}
```

- **拓展题**：给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

26. 复杂链表的复制

题目描述：输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

思路：先复制链表的 next 节点，将复制后的节点接在原节点后，然后复制其它的节点，最后取偶数位置的节点（复制后的节点）。

代码实现：

```
public RandomListNode Clone2(RandomListNode pHead) {
    if(pHead == null)
        return null;
    RandomListNode head = new RandomListNode(pHead.label) ;
    RandomListNode temp = head ;

    while(pHead.next != null) {
        temp.next = new RandomListNode(pHead.next.label) ;
        if(pHead.random != null) {
            temp.random = new RandomListNode(pHead.random.label) ;
        }
        pHead = pHead.next ;
        temp = temp.next ;
    }
    return head ;
}
```

27. 二叉搜索树转换为双向链表

题目描述：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

思路：定义一个链表的尾节点，递归处理左右子树，最后返回链表的头节点

代码实现：

```
public TreeNode Convert(TreeNode pRootOfTree) {
    TreeNode lastlist = covertNode(pRootOfTree,null);
    TreeNode pHead = lastlist;
```

```
    while (pHead != null && pHead.left != null) {
        pHead = pHead.left;
    }
    return pHead;
}

public TreeNode covertNode(TreeNode root, TreeNode lastlist) {
    if (root == null)
        return null;
    TreeNode cur = root;
    if (cur.left != null) {
        lastlist = covertNode(cur.left,lastlist);
    }
    cur.left = lastlist;
    if (lastlist != null) {
        lastlist.right = cur;
    }
    lastlist = cur;
    if (cur.right != null) {
        lastlist = covertNode(cur.right,lastlist);
    }
    return lastlist;
}
```

28. 打印字符串中所有字符的排列

题目描述：输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc,则打印出由字符 a,b,c 所能排列出来的所有字符串 abc,acb,bac,bca,cab 和 cba。

思路：将当前位置的字符和前一个字符位置交换，递归。

代码实现：

```
public ArrayList<String> Permutation(String str) {
    ArrayList<String> res = new ArrayList<>();
    if (str == null || str.length() == 0) {
        return res;
    }
    helper(res, 0, str.toCharArray());
    // 符合结果的输出顺序
    Collections.sort(res);
    return res;

}

private void helper(ArrayList<String> res, int index, char[] s) {
    if (index == s.length - 1) {
```

```
            res.add(String.valueOf(s));
            return;
        }

        for (int i = index; i < s.length; i++) {
            if (i == index || s[index] != s[i]) {
                swap(s, index, i);
                helper(res, index + 1, s);
                swap(s, index, i);
            }
        }
    }

    public void swap(char[] c, int a,int b) {
        char temp = c[a];
        c[a] = c[b];
        c[b] = temp;
    }
```

- **拓展题**：给定一个没有重复数字的序列，返回其所有可能的全排列。

29. 数组中出现次数超过一半的数字

题目描述：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。如果不存在则输出 0。

思路：将首次出现的数 count+1，与之后的数进行比较，相等则+1，否则—1，最后进行校验是否超过长度的一半。

代码实现：

```
    public static int moreThanHalfNum(int[] nums) {
        int count = 0;
        int candidate = 0;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }

        return checkMoreThanHalf(nums, candidate) ? candidate : 0;
    }

    private static boolean checkMoreThanHalf(int[] array, Integer number) {
        int times = 0;
        for (int i : array) {
```

```
            if (i == number) {
                times++;
            }
        }
        return times * 2 >= array.length;
    }
```

30. 找出最小的 K 个数

题目描述：输入 n 个整数，找出其中最小的 K 个数。

思路：先将前 K 个数放入数组，进行堆排序，若之后的数比它还小，则进行调整

代码实现：

```
public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k)
 {
    ArrayList<Integer> list = new ArrayList<>();
    if (input == null || k <= 0 || k > input.length) {
        return list;
    }
    int[] kArray = Arrays.copyOfRange(input,0,k);
    // 创建大根堆
    buildHeap(kArray);

    for(int i = k; i < input.length; i++) {
        if(input[i] < kArray[0]) {
            kArray[0] = input[i];
            maxHeap(kArray, 0);
        }
    }

    for (int i = kArray.length - 1; i >= 0; i--) {
        list.add(kArray[i]);
    }

    return list;
}

public void buildHeap(int[] input) {
    for (int i = input.length/2 - 1; i >= 0; i--) {
        maxHeap(input,i);
    }
}

private void maxHeap(int[] array,int i) {
    int left=2*i+1;
    int right=left+1;
    int largest=0;
```

```
    if(left < array.length && array[left] > array[i])
        largest=left;
    else
        largest=i;

    if(right < array.length && array[right] > array[largest])
        largest = right;

    if(largest != i) {
        int temp = array[i];
        array[i] = array[largest];
        array[largest] = temp;
        maxHeap(array, largest);
    }
}
```

31. 连续子数组的最大和

题目描述：输入一个整型数组，数组中有正数也有负数，数组中一个或连续的多个整数组成一个子数组，求连续子数组的最大和

思路：若和小于 0，则将最大和置为当前值，否则计算最大和。

代码实现：时间复杂度：O(n)，空间复杂度：O(1)

```
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int sum = 0;
        int result = nums[0];
        for (int num : nums) {
            sum = sum > 0 ? sum + num : num;
            result = Math.max(sum, result);
        }
        return result;
    }
```

32. 从 1 到非负整数 n 中 1 出现的次数

题目描述：输入一个整数 n，求从 1 到整数 n 的十进制表示中 1 出现的次数

思路：若百位上数字为 0，百位上可能出现 1 的次数由更高位决定；若百位上数字为 1，百位上可能出现 1 的次数不仅受更高位影响还受低位影响；若百位上数字大于 1，则百位上出现 1 的情况仅由更高位决定

代码实现：

- 解法一：计算高低位

```
public static int NumberOf1Between1AndN2(int n) {
    int count = 0; // 1 的个数
    int i = 1;  // 当前位
    int current = 0, after = 0, before = 0;

    while ((n / i) != 0) {
        before = n / (i * 10); // 高位
        current = (n / i) % 10; // 当前位
        after = n - (n / i) * i;  // 低位

        if (current == 0) {
            //如果为0,出现1 的次数由高位决定,等于高位数字 * 当前位数
            count = count + before * i;
        } else if (current == 1) {
            //如果为1,出现1 的次数由高位和低位决定,高位*当前位+低位+1
            count = count + before * i + after + 1;
        } else if (current > 1) {
            // 如果大于1,出现1 的次数由高位决定,（高位数字+1）* 当前位数
            count = count + (before + 1) * i;
        }
        //前移一位
        i = i * 10;
    }
    return count;
}
```

- 解法二：公式法

```
public int NumberOf1Between1AndN_Solution(int n){
    int count=0;

    for(int i = 1; i <= n; i *= 10){
        int a = n / i; // 高位
        int b = n % i; // 低位
        count += (a+8) / 10 * i;
        if(a % 10 == 1){
            count += b + 1;
        }
    }
    return count;
}
```

33. 把数组中的数排成一个最小的数

题目描述：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个

思路：先将整型数组转换成 String 数组，然后将 String 数组排序，最后将排好序的字符串数组拼接出来。关键就是制定排序规则。或使用比较和快排的思想，将前面的数和最后的数比较，若小则放到最前面，最后再递归调用。

代码实现：

```
public String PrintMinNumber(int [] numbers) {
    if(numbers == null || numbers.length == 0)
        return "";
    int len = numbers.length;
    String[] str = new String[len];
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < len; i++){
        str[i] = String.valueOf(numbers[i]);
    }
    Arrays.sort(str,new Comparator<String>(){
        @Override
        public int compare(String s1, String s2) {
            String c1 = s1 + s2;
            String c2 = s2 + s1;
            return c1.compareTo(c2);
        }
    });

    for(int i = 0; i < len; i++){
        sb.append(str[i]);
    }
    return sb.toString();
}
```

34.求第 N 个丑数

题目描述：求从小到大的第 N 个丑数。丑数是只包含因子 2、3 和 5 的数，习惯上我们把 1 当做是第一个丑数。

思路：乘 2 或 3 或 5，之后比较取最小值。

代码实现：

```
public static int getUglyNumber(int n) {
    if (n <= 0) {
        return 0;
```

```
    }
    int[] arr = new int[n];
    arr[0] = 1;
    int multiply2 = 0;
    int multiply3 = 0;
    int multiply5 = 0;

    for (int i = 1; i < n; i++) {
        int min = Math.min(arr[multiply2] * 2, Math.min(arr[multiply3]
* 3, arr[multiply5] * 5));
        arr[i] = min;
        if (arr[multiply2] * 2 == min) {
            multiply2++;
        }
        if (arr[multiply3] * 3 == min) {
            multiply3++;
        }
        if (arr[multiply5] * 5 == min) {
            multiply5++;
        }
    }
    return arr[n - 1];
}
```

35. 第一个出现一次的字符

题目描述：在一个字符串(1<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置

思路：利用 LinkedHashMap 保存字符和出现次数。

代码实现：

```
public int FirstNotRepeatingChar(String str) {
    if (str == null || str.length() == 0)
        return -1;
    char[] c = str.toCharArray();
    LinkedHashMap<Character,Integer> hash=new LinkedHashMap<Character,I
nteger>();

    for(char item : c) {
        if(hash.containsKey(item))
            hash.put(item, hash.get(item)+1);
        else
            hash.put(item, 1);
    }

    for(int i = 0;i < str.length(); i++){
        if (hash.get(str.charAt(i)) == 1) {
```

```
                return i;
            }
        }
        return -1;
    }
```

36. 数组中逆序对的个数

题目描述：在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数 P

思路：本质是归并排序，在比较时加入全局变量 count 进行记录逆序对的个数，若 data[start] >= data[index]，则 count 值为 mid+1-start

代码实现：

```
int count = 0;
public int InversePairs(int [] array) {
    if(array==null)
        return 0;
    mergeSort(array,0,array.length-1);
    return count;
}

private void mergeSort(int[] data,int start,int end) {
    int mid = (start + end) / 2;
    if (start < end) {
        mergeSort(data, start, mid);
        mergeSort(data, mid + 1, end);
        merge(data, start, mid, end);
    }
}

private void merge(int[] data,int start,int mid,int end) {
    int arr[] = new int[end - start + 1];
    int c = 0;
    int s = start;
    int index = mid + 1;
    while (start <= mid && index <= end) {
        if (data[start] < data[index]) {
            arr[c++] = data[start++];
        } else {
            arr[c++] = data[index++];
            count += mid +1 - start;
            count %= 1000000007;
        }
    }
```

```
    while (start <= mid) {
        arr[c++] = data[start++];
    }

    while (index <= end) {
        arr[c++] = data[index++];
    }

    for (int d : arr) {
        data[s++] = d;
    }
}
```

37. 两个链表的第一个公共节点

题目描述：输入两个链表，找出它们的第一个公共结点。

思路：先求出链表长度，然后长的链表先走多出的几步，然后两个链表同时向下走去寻找相同的节点，代码量少的方法需要将两个链表遍历两次，然后从头开始相同的节点。

代码实现：

```
public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
    ListNode p1 = pHead1;
    ListNode p2 = pHead2;
    while (p1 != p2){
        p1 = (p1 != null ? p1.next : pHead2);
        p2 = (p2 != null ? p2.next : pHead1);
    }
    return p1;
}
```

38. 求某个数在数组中出现次数

题目描述：统计一个数字在排序数组中出现的次数。

思路：利用二分查找+递归思想，进行寻找。当目标值与中间值相等时进行判断

代码实现：

```
public static int getNumberOfK(int[] arraySorted, int k) {
    if (arraySorted == null || arraySorted.length == 0) {
        return 0;
    }
    if (arraySorted.length == 1) {
        return arraySorted[0] == k ? 1 : 0;
    }
```

```
    int result = 0;
    int mid = arraySorted.length / 2;
    if (k < arraySorted[mid]) {
        result += getNumberOfK(Arrays.copyOfRange(arraySorted, 0, mid),
 k);
    } else if (k > arraySorted[mid]) {
        result += getNumberOfK(Arrays.copyOfRange(arraySorted, mid, arr
aySorted.length), k);
    } else {
        result += getCount(arraySorted, mid);
    }

    return result;
}

private static int getCount(int[] arraySorted, int mid) {
    int k = arraySorted[mid];
    int result = 0;

    for (int i = mid; i < arraySorted.length; i++) {
        if (arraySorted[i] == k) {
            result++;
        } else {
            break;
        }
    }

    for (int i = mid - 1; i >= 0; i--) {
        if (arraySorted[i] == k) {
            result++;
        } else {
            break;
        }
    }
    return result;
}
```

39. 二叉树的深度

39.1 求某个二叉树的深度

题目描述：输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

思路：利用递归遍历分别返回左右子树深度

代码实现：时间复杂度：O(n)，空间复杂度：O(log n)

```
public int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    return Math.max(left, right) + 1;
}
```

39.2 判断某二叉树是否是平衡二叉树

题目描述：输入一棵二叉树，判断该二叉树是否是平衡二叉树。

思路：平衡二叉树的条件：左子树是平衡二叉树，右子树是平衡二叉树，左右子树高度不超过 1

代码实现：

```
public boolean isBalanced(TreeNode root) {
    if(root == null) {
        return true;
    }
    boolean condition = Math.abs(maxDepth(root.left) - maxDepth(root.ri
ght)) <= 1;
    return condition && isBalanced(root.left) && isBalanced(root.right);
}
```

40. 找出只出现一次的数字

题目描述：一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

思路：两个相同的数异或后为 0，一个数和 0 异或还是它本身，将所有数异或后即得到 A、B 异或的结果，然后求得 1 在该数最右边出现的 index，然后判断每个数右移 index 后是不是 1。

代码实现：

```
public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
    if (array == null)
        return;
    num1[0] = 0;
    num2[0] = 0;
    int number = array[0];
    for (int i = 1; i < array.length; i++)
        number ^= array[i];
    // 异或后的数 1 出现在第几位
```

```
    int index = 0;
    while ((number & 1) == 0) {
        number = number >> 1;
        index++;
    }

    for (int i = 0; i < array.length; i++) {
        boolean isBit = ((array[i] >> index) & 1) == 0;
        if (isBit) {
            num1[0] ^= array[i];
        } else {
            num2[0] ^= array[i];
        }
    }
}
```

41. 整数序列的查找

41.1 和为 S 的连续整数序列

题目描述：输出所有和为 S 的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

思路：定义两个指针，分别递增，寻找和为 s 的序列。

代码实现：

```
public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {
    ArrayList<ArrayList<Integer>> arrayList = new ArrayList<>();
    ArrayList<Integer> list = new ArrayList<>();
    if (sum < 3)
        return arrayList;
    int small = 1;
    int big = 2;
    while (small < (sum + 1) / 2) {
        int s = 0;
        for (int i = small; i <= big; i++) {
            s += i;
        }
        if (s == sum) {
            for (int i = small; i <= big; i++) {
                list.add(i);
            }
            arrayList.add(new ArrayList<>(list));
            list.clear();
            small++;
        } else {
            if (s > sum) {
                small++;
```

```
            } else {
                big++;
            }
        }
    }
    return arrayList;
}
```

41.2 求两个乘积最小的数

题目描述：输入一个递增排序的数组和一个数字 S，在数组中查找两个数，是的他们的和正好是 S，如果有多对数字的和等于 S，输出两个数的乘积最小的。

思路：定义两个指针，分别从前面和后面进行遍历。间隔越远乘积越小，所以是最先出现的两个数乘积最小

代码实现：

```
public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
    ArrayList<Integer> list = new ArrayList<>();
    if (array == null )
        return list;
    int left = 0;
    int right = array.length - 1;

    while (left < right) {
        int s = array[left] + array[right];
        if (s == sum) {
            list.add(array[left]);
            list.add(array[right]);
            return list;
        }else {
            if (s > sum) {
                right--;
            }else {
                left++;
            }
        }
    }
    return list;
}
```

42.字符串中字符的移动

42.1 反转字符串

题目描述：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变

思路：先将整个字符串翻转，然后将每个单词反转。

代码实现：

```
public String reverseSentence(String sentence) {
    if (sentence == null || sentence.length() == 0 || sentence.trim().l
ength() == 0) {
        return sentence;
    }
    String blank = " ";
    String sentenceReverse = reverse(sentence);
    String[] splitStrings = sentenceReverse.split(blank);
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < splitStrings.length - 1; i++) {
        sb.append(reverse(splitStrings[i])).append(blank);
    }
    sb.append(reverse(splitStrings[splitStrings.length - 1]));
    return String.valueOf(sb);
}

public String reverse(String str) {
    StringBuilder sb = new StringBuilder();
    for (int i = str.length() - 1; i >= 0 ; i--) {
        sb.append(str.charAt(i));
    }
    return String.valueOf(sb);
}
```

42.2 将字符串循环左移 K 位

题目描述：对于一个给定的字符序列 S，请你把其循环左移 K 位后的序列输出

思路：拼接或反转三次字符串

代码实现：

```
public String LeftRotateString(String str,int n) {
    if (str == null || str.length() == 0)
        return str;
    String s1 = reverse(str.substring(0,n));
    String s2 = reverse(str.substring(n,str.length()));
    return reverse(s2)+reverse(s1);
}
```

43. n 个骰子的点数及出现的概率

题目描述：把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 s,输入 n,打印出 s 的所有可能出现的概率

思路：递归一般是自顶向下的分析求解，而循环则是自底向上，占用更少的空间和更少的时间，性能较好。定义一个二维数组，第一次掷骰子有 6 种可能，第一个骰子投完的结果存到 probabilities[0]；第二次开始掷骰子，在下一循环中，我们加上一个新骰子，此时和为 n 的骰子出现次数应该等于上一次循环中骰子点数和为 n-1,n-2,n-3, n-4,n-5，n-6 的次数总和，所以我们把另一个数组的第 n 个数字设为前一个数组对应 n-1,n-2,n-3,n-4,n-5，n-6 之和

代码实现：

```
public Map<Integer, Double> printProbability(int number) {
    Map<Integer, Double> probabilityMap = new HashMap<>();
    if (number < 1) {
        return probabilityMap;
    }
    int g_maxValue = 6;
    int[][] probabilities = new int[2][];
    probabilities[0] = new int[g_maxValue * number + 1];
    probabilities[1] = new int[g_maxValue * number + 1];
    int flag = 0;

    // 当第一次抛掷骰子时，有6 种可能，每种可能出现一次
    // 第一个骰子投完的结果存到了probabilities[0]
    for (int i = 1; i <= g_maxValue; i++) {
        probabilities[0][i] = 1;
    }

    //从第二次开始掷骰子，假设第一个数组中的第n 个数字表示骰子和为n 出现的次数，
    for (int k = 2; k <= number; ++k) {

        // 第k 次掷骰子，和最小为k，小于k 的情况是不可能发生的,令不可能发生的次数设置为0!
        for (int i = 0; i < k; ++i) {
            probabilities[1 - flag][i] = 0;
        }

        // 第k 次掷骰子，和最小为k，最大为g_maxValue*k
        for (int i = k; i <= g_maxValue * k; ++i) {
            // 初始化，因为这个数组要重复使用，上一次的值要清0
            probabilities[1 - flag][i] = 0;
            for (int j = 1; j <= i && j <= g_maxValue; ++j)
```

```
                probabilities[1 - flag][i] += probabilities[flag][i -
j];
            }

            // 若flag=0，1-flag用的就是数组1，而flag=1，1-flag用的就是数组0
            flag = 1 - flag;
        }

        double total = Math.pow(g_maxValue, number);
        for (int sum = number; sum <= g_maxValue * number; sum++) {
            double ratio = (double) probabilities[flag][sum] / total;
            probabilityMap.put(sum, ratio);
        }

        return probabilityMap;
    }
```

44. 扑克牌的顺子

题目描述：从扑克牌中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2~10 为数字本身，A 为 1，J 为 11，Q 为 12，K 为 13，大小王可以看成任意数字。

思路：用数组记录五张扑克牌，将数组调整为有序的，若 0 出现的次数>=顺子的差值，即为顺子。

代码实现：

```
public static boolean isContinuous(int [] numbers) {
    if (numbers == null || numbers.length == 0) {
        return false;
    }
    int count = 0;
    int diff = 0;
    Arrays.sort(numbers);

    for (int i = 0; i < numbers.length - 1; i++) {
        if (numbers[i] == 0) {
            count++;
            continue;
        }
        if (numbers[i] == numbers[i+1]) {
            return false;
        }

        diff += numbers[i+1] - numbers[i] - 1;
    }
```

```
    return diff <= count;
}
```

45. 圆圈中最后剩下的数

题目描述：圆圈中最后剩下的数字（约瑟夫环）

思路：利用公式法：f[n] = (f[n-1] + k) mod n，或使用循环链表实现

代码实现：

```
public static int lastRemain(int n, int m) {
    if (n < 1 || m < 1) {
        return -1;
    }
    int last = 0;
    for (int i = 2; i <= n; i ++) {
        // i 个人时删除数的索引等于 i-1 个人时删除数的索引+k(再对 i 取余)
        last = (last + m) % i;
    }
    return last;
}
```

46. 求 1 到 n 的和

题目描述：求 1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case 等关键字及条件判断语句（A?B:C）。

思路：巧用递归（返回值类型为 Boolean）

代码实现：

```
    public int Sum_Solution(int n) {
        int sum = n;
        boolean result = (n > 0) && ((sum += Sum_Solution(n-1)) > 0);
        return sum;
    }
```

47. 不用加减乘除做加法

题目描述：写一个函数，求两个整数之和，要求在函数体内不得使用+、-、*、/四则运算符号。

思路：利用位运算

代码实现：

```java
public int Add(int num1,int num2) {
    while (num2 != 0) {
        // 计算个位
        int temp = num1 ^ num2;
        // 计算进位（1+1）
        num2 = (num1 & num2) << 1;
        num1 = temp;
    }
    return num1;
}
```

48. 不能被继承的类

题目描述：设计一个不能被继承的类

思路：私有构造器的类不能继承

49. 将字符串转换为整数

题目描述：将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。 数值为 0 或者字符串不是一个合法的数值则返回 0

思路：若为负数，则输出负数，字符 0 对应 48,9 对应 57，不在范围内则返回 false。

代码实现：

```java
public int StrToInt(String str) {
    if (str == null || str.length() == 0)
        return 0;
    int mark = 0;
    int number = 0;
    char[] chars = str.toCharArray();

    if (chars[0] == '-')
        mark = 1;

    for (int i = mark; i < chars.length; i++) {
        if (chars[i] == '+') {
            continue;
        }
        if (chars[i] < 48 || chars[i] > 57) {
            return 0;
        }
        number = number * 10 + chars[i] - 48;
    }
```

```
    return mark == 0 ? number : -number;
}
```

50. 树中两个节点的最低公共祖先

题目描述：给定一个二叉搜索树，输入两个节点，求树中两个节点的最低公共祖先

思路：从根节点开始遍历树，如果节点 p 和 q 都在右子树上，那么以右孩子为根节点递归，如果节点 p 和节点 q 都在左子树上，那么以左孩子为根节点递归，否则就意味找到节 p 和节点 q 的最低公共祖先了。

代码实现：

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNod
e q) {
    if (root == null || p == null || q == null) {
        return null;
    }
    int parentVal = root.val;
    int pVal = p.val;
    int qVal = q.val;

    if (pVal > parentVal && qVal > parentVal) {
        return lowestCommonAncestor(root.right, p, q);
    } else if (pVal < parentVal && qVal < parentVal) {
        return lowestCommonAncestor(root.left, p, q);
    } else {
        return root;
    }
}
```

51. 找出重复的数

题目描述：在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内，找出数组中任意一个重复的数字

思路：快慢指针：先用快慢两个下标都从 0 开始，快下标每轮映射两次，慢下标每轮映射一次，直到两个下标再次相同。这快下标从 0 开始，这两个下标都继续每轮映射一次，当这两个下标相遇时，就是重复的数。。或使用辅助空间（HashSet）

代码实现：

```
public int findDuplicate(int[] numbers) {
    if (numbers == null || numbers.length == 0) {
        return -1;
```

```
    }
    int slow = 0;
    int fast = 0;
    for (int i = 0; i < numbers.length; i++) {
        slow = numbers[slow];
        fast = numbers[numbers[fast]];
        if (fast == slow) {
            fast = 0;
            while (fast != slow) {
                fast = numbers[fast];
                slow = numbers[slow];
            }
            return slow;
        }
    }
    return -1;
}
```

52. 构建乘积数组

题目描述：给定一个数组 A[0,1,...,n-1],请构建一个数组 B[0,1,...,n-1],其中 B 中的元素 B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]...A[n-1]。其中 A[i] = 1。不能使用除法，

思路：使用矩阵法求解，将矩阵分为上三角矩阵和下三角矩阵，分别求乘积

代码实现：

```
public int[] multiply(int[] A) {
    int length = A.length;
    int[] B = new int[length];
    if(length != 0 ){
        B[0] = 1;
        //计算下三角连乘
        for(int i = 1; i < length; i++){
            B[i] = B[i-1] * A[i-1];
        }
        int temp = 1;
        //计算上三角连乘
        for(int j = length-2; j >= 0; j--){
            temp *= A[j+1];
            B[j] *= temp;
        }
    }
    return B;
}
```

53. 正则表达式匹配

题目描述：请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符， 而* 表示它前面的字符可以出现任意次（包含 0 次）

思路：当字符串只有一个字符时，进行判断，否则就有两种递归情况，（1）当模式中的第二个字符不是“*”时：如果字符串第一个字符和模式中的第一个字符相匹配或是点那么字符串和模式都后移一个字符，然后匹配剩余的；如果 字符串第一个字符和模式中的第一个字符相不匹配，直接返回 false。（2）当模式中的第二个字符是“*”时：如果字符串第一个字符跟模式第一个字符不匹配，则模式后移 2 个字符，继续匹配；如果字符串第一个字符跟模式第一个字符匹配或是点，可以有 3 种匹配方式：1 >模式后移 2 字符，相当于 x*被忽略；2>字符串后移 1 字符，模式后移 2 字符；3>字符串后移 1 字符，模式不变，即继续匹配字符下一位，因为 * 可以匹配多位；

代码实现：

```
public boolean match(char[] str, char[] pattern) {
    if (str == null || pattern == null)
        return false;
    // 若字符串的长度为1
    if (str.length == 1) {
        if (pattern.length == 1){
            if (str[0] == pattern[0] || pattern[0] == '.')
                return true;
            return false;
        }
    }
    int sindex = 0;
    int pindex = 0;
    return matchIndex(str,sindex,pattern,pindex);
}

public boolean matchIndex(char[] str,int sindex, char[] pattern, int pi
ndex) {
    // str 和pattern 同时到达末尾，则匹配成功
    if (sindex == str.length && pindex == pattern.length)
        return true;
    // 若pattern 先到尾，而str 没有到达末尾，则匹配失败
    if (sindex != str.length && pindex == pattern.length)
        return false;
    // 若pattern 第二个字符是*
    if (pindex + 1 < pattern.length && pattern[pindex + 1] == '*') {
        if (sindex != str.length && pattern[pindex] == str[sindex] ||
                sindex != str.length && pattern[pindex] == '.') {
            return matchIndex(str,sindex+1,pattern,pindex+2)
```

```
                    || matchIndex(str,sindex,pattern,pindex+2)
                    || matchIndex(str,sindex+1,pattern,pindex);
        } else {
            return matchIndex(str,sindex,pattern,pindex+2);
        }
    }
    // 若pattern第二个字符不是*
    if (sindex != str.length && pattern[pindex] == str[sindex] ||
            sindex != str.length && pattern[pindex] == '.')
        return matchIndex(str,sindex+1,pattern,pindex+1);
    return false;
}
```

54. 表示数值的字符串

题目描述：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）

思路：逐个字符进行判断，e 或 E 和小数点最多出现一次，而 e 或 E 的前一个必须是数字，且不能是第一个或最后一个字符，符号的前一个字符不能是 e 或 E。也可用正则表达式判断！

代码实现：

```
public boolean isNumeric(char[] str) {
    if (str == null)
        return false;
    int index = 0;
    int ecount = 0;
    int point = 0;
    // 如果第一个字符是符号就跳过
    if (str[0] == '-' || str[0] == '+')
        index++;

    for (int i = index; i < str.length; i++) {
        if (str[i] == '-' || str[i] == '+') {
            if (str[i-1] != 'e' && str[i-1] != 'E')
                return false;
            continue;
        }

        if (str[i] == 'e' || str[i] == 'E') {
            ecount++;
            if (ecount > 1)
                return false;
            if (i == 0 || str[i-1] < 48 || str[i-1] > 57 || i == str.le
ngth-1)
                return false;
            point++;
            continue;
```

```
        }

        if (str[i] == '.') {
            point++;
            if (point > 1)
                return false;
            continue;
        }
        // 出现非数字且不是e/E 则返回false（小数点和符号用continue 跳过了）
        if ((str[i] < 48 || str[i] > 57) && (str[i] != 'e') && (str[i] !
= 'E'))
            return false;
    }
    return true;
}
```

55. 字符流中第一个不重复的字符

题目描述：请实现一个函数用来找出字符流中第一个只出现一次的字符。

思路：借助辅助空间进行判断，如字符数组。

代码实现：

```
char[] chars = new char[256];
StringBuilder sb = new StringBuilder();

public void Insert(char ch) {
    sb.append(ch);
    chars[ch]++;
}

public char FirstAppearingOnce() {
    char[] str = sb.toString().toCharArray();
    for (char c : str) {
        if (chars[c] == 1) {
            return c;
        }
    }
    return '#';
}
```

56. 链表中环的入口节点

题目描述：一个链表中包含环，请找出该链表的环的入口结点。

思路：定义快慢两个指针，相遇后（环中相汇点）将快指针指向 pHead 然后一起走，每次往后挪一位，相遇的节点即为所求。详细分析：相遇即 p1==p2 时，p2

所经过节点数为 2x,p1 所经过节点数为 x,设环中有 n 个节点,p2 比 p1 多走一圈有 2x=n+x; n=x;可以看出 p1 实际走了一个环的步数，再让 p2 指向链表头部，p1 位置不变，p1,p2 每次走一步直到 p1==p2; 此时 p1 指向环的入口。

代码实现：

```
public ListNode EntryNodeOfLoop(ListNode pHead) {
    if (pHead == null || pHead.next == null)
        return null;
    ListNode slow = pHead;
    ListNode fast = pHead;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast){
            fast = pHead;
            while (fast != slow) {
                fast = fast.next;
                slow = slow.next;
            }
            if (fast == slow)
                return slow;
        }
    }
    return null;
}
```

57. 删除链表中重复的节点

题目描述：在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。

思路：先新建一个头节点，然后向后查找值相同的节点，重复查找后删除

代码实现：

```
    public ListNode deleteDuplication(ListNode pHead) {
        if (pHead == null)
            return null;
        // 新建一个节点，防止头结点被删除
        ListNode first = new ListNode(-1);
        first.next = pHead;
        ListNode p = pHead;
        // 指向前一个节点
        ListNode preNode = first;
```

```
        while (p != null && p.next != null) {
            if (p.val == p.next.val) {
                int val = p.val;
                // 向后重复查找
                while (p != null && p.val == val) {
                    p = p.next;
                }
                // 上个非重复值指向下一个非重复值：即删除重复值
                preNode.next = p;
            }else {
                // 如果当前节点和下一个节点值不等，则向后移动一位
                preNode = p;
                p = p.next;
            }
        }
        return first.next;
    }
```

- **拓展题**：删除排序链表所有重复的元素，使得每个元素只出现一次。

58. 二叉树的下一个节点

题目描述：给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

思路：若节点右孩子存在，则设置一个指针从该节点的右孩子出发，一直沿着指向左子结点的指针找到的叶子节点即为下一个节点；若节点不是根节点。如果该节点是其父节点的左孩子，则返回父节点；否则继续向上遍历其父节点的父节点，重复之前的判断，返回结果

代码实现：

```
public TreeLinkNode GetNext(TreeLinkNode pNode) {
    if (pNode == null)
        return null;
    if (pNode.right != null) {
        pNode = pNode.right;
        while (pNode.left != null) {
            pNode = pNode.left;
        }
        return pNode;
    }

    while (pNode.next != null) {
        // 找第一个当前节点是父节点左孩子的节点
        if (pNode.next.left == pNode)
            return pNode.next;
```

```
            pNode = pNode.next;
        }
        return null;
    }
```

59. 对称的二叉树

题目描述：请实现一个函数，用来判断一颗二叉树是不是对称的。注意：如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

思路：利用递归进行判断，若左子树的左孩子等于右子树的右孩子且左子树的右孩子等于右子树的左孩子，并且左右子树节点的值相等，则是对称的。

代码实现：时间复杂度：O(n)，空间复杂度：O(n)

```
public static boolean isSymmetrical(TreeNode pRoot){
    return pRoot == null || isCommon(pRoot.left, pRoot.right);
}

public static boolean isCommon(TreeNode leftNode, TreeNode rightNode) {
    if (leftNode == null && rightNode == null) {
        return true;
    }
    if (leftNode == null || rightNode == null) {
        return false;
    }

    return leftNode.val == rightNode.val && isCommon(leftNode.left,righ
tNode.right) && isCommon(leftNode.right,rightNode.left);
}
```

60. 按之字形顺序打印二叉树

题目描述：请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，依此类推。

思路：利用两个栈的辅助空间分别存储奇数偶数层的节点，然后打印输出。或使用链表的辅助空间来实现，利用链表的反向迭实现逆序输出。

代码实现：

```
public static List<List<Integer>> Print(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if (root == null) {
        return res;
    }
```

```
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int depth = 0;
    while (!queue.isEmpty()) {
        LinkedList<Integer> tmp = new LinkedList<>();
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (depth % 2 == 0) {
                tmp.add(node.val);
            } else {
                tmp.addFirst(node.val);
            }
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
        res.add(tmp);
        depth++;
    }
    return res;
}
```

61. 把二叉树打印成多行

题目描述：从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

思路：利用辅助空间链表或队列来存储节点，每层输出。

代码实现：

```
public static ArrayList<ArrayList<Integer>> printTree(TreeNode pRoot) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<>();
    if (pRoot == null) {
        return res;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(pRoot);
    ArrayList<Integer> tmp = new ArrayList<>();
    int start = 0;
    int end = 1;

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        tmp.add(node.val);
```

```
            start++;
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }

            if (start == end) {
                start = 0;
                end = queue.size();
                res.add(new ArrayList<>(tmp));
                tmp.clear();
            }
        }
        return res;
    }
```

62. 序列化二叉树

题目描述：请实现两个函数，分别用来序列化和反序列化二叉树

思路：序列化：前序遍历二叉树存入字符串中；反序列化：根据前序遍历重建二叉树。

代码实现：

```
public String Serialize(TreeNode root) {
    StringBuffer sb = new StringBuffer();
    if (root == null){
        sb.append("#,");
        return sb.toString();
    }

    sb.append(root.val + ",");
    sb.append(Serialize(root.left));
    sb.append(Serialize(root.right));
    return sb.toString();
}

public int index = -1;
public TreeNode Deserialize(String str) {
    index++;
    int len = str.length();
    String[] strr = str.split(",");
    TreeNode node = null;

    if (index >= len)
        return null;
```

```
    if (!strr[index].equals("#")){
        node = new TreeNode(Integer.valueOf(strr[index]));
        node.left = Deserialize(str);
        node.right = Deserialize(str);
    }
    return node;
}
```

63. 求二叉搜索树的第 K 小的节点

题目描述：给定一棵二叉搜索树，请找出其中的第 k 小的结点

思路：二叉搜索树按照中序遍历的顺序打印出来正好就是排序好的顺序，第 k 个结点就是第 K 大的节点，分别递归查找左右子树的第 K 个节点，或使用非递归借用栈的方式查找，当 count=k 时返回根节点。

代码实现：

```
private int count = 0;
public TreeNode KthNode(TreeNode pRoot, int k) {
    if(pRoot == null) {
        return null;
    }
    TreeNode node = KthNode(pRoot.left, k);
    if(node != null) {
        return node;
    }
    count++;
    if(count == k) {
        return pRoot;
    }
    node = KthNode(pRoot.right, k);
    return node;
}
```

64. 数据流中的中位数

题目描述：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

思路：创建优先级队列维护大顶堆和小顶堆两个堆，并且小顶堆的值都大于大顶堆的值，2 个堆个数的差值小于等于 1，所以当插入个数为奇数时：大顶堆个数就比小顶堆多 1，中位数就是大顶堆堆头；当插入个数为偶数时，使大顶堆个数跟小顶堆个数一样，中位数就是 2 个堆堆头平均数。也可使用集合类的排序方法。

代码实现：

```
int count = 0;
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(16, new Comparator
<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
});

public void Insert(Integer num) {
    count++;
    // 当数据的个数为奇数时，进入大根堆
    if ((count & 1) == 1) {
        minHeap.offer(num);
        maxHeap.offer(minHeap.poll());
    } else {
        maxHeap.offer(num);
        minHeap.offer(maxHeap.poll());
    }
}

public Double GetMedian() {
    if (count == 0)
        return null;

    // 当数据个数是奇数时，中位数就是大根堆的顶点
    if ((count & 1) == 1) {
        return Double.valueOf(maxHeap.peek());
    } else {
        return Double.valueOf((minHeap.peek() + maxHeap.peek())) / 2;
    }
}
```

65. 滑动窗口的最大值

题目描述：给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值

思路：两个 for 循环，第一个 for 循环滑动窗口，第二个 for 循环滑动窗口中的值，寻找最大值。还可以使用时间复杂度更低的双端队列求解。

代码实现：

```
public ArrayList<Integer> maxInWindows(int [] num, int size) {
    ArrayList<Integer> list = new ArrayList<>();
    if (num == null || size < 1 || num.length < size)
        return list;
```

```
    int length = num.length - size + 1;

    for (int i = 0; i < length; i++) {
        int current = size + i;
        int max = num[i];
        for (int j = i; j < current; j++) {
            if (max < num[j]) {
                max = num[j];
            }
        }
        list.add(max);
    }
    return list;
}
```

66. 矩阵中的路径

题目描述：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。

思路：回溯法，双层 for 循环，判断每一个点，每次递归调用上下左右四个点，用 flag 标志是否已经匹配（return），进行判断点的位置是否越界，是否已经正确匹配，判断矩阵的路径与模式串的第 index 个字符是否匹配。

代码实现：

```
public boolean hasPath(char[] matrix, int rows, int cols, char[] str) {
    int flag[] = new int[matrix.length];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (helper(matrix, rows, cols, i, j, str, 0, flag))
                return true;
        }
    }
    return false;
}

private boolean helper(char[] matrix,int rows,int cols,int i,int j,char
[] str,int k,int[] flag) {
    int index = i * cols + j;
    if (i < 0 || i >= rows || j < 0 || j >= cols || matrix[index] != st
r[k] || flag[index] == 1)
        return false;

    if(k == str.length - 1)
        return true;
```

```
    flag[index] = 1;

    if (helper(matrix, rows, cols, i - 1, j, str, k + 1, flag)
            || helper(matrix, rows, cols, i + 1, j, str, k + 1, flag)
            || helper(matrix, rows, cols, i, j - 1, str, k + 1, flag)
            || helper(matrix, rows, cols, i, j + 1, str, k + 1, flag))
{
        return true;
    }
    flag[index] = 0;
    return false;
}
```

67. 机器人的运动范围

题目描述：地上有一个 m 行和 n 列的方格。一个机器人从坐标 0,0 的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于 k 的格子。请问该机器人能够达到多少个格子？

思路：利用递归实现，每次只能走上下左右四个点，进行判断点的位置是否越界，点数之和是否大于 K，是否已经走过了。

代码实现：

```
public int movingCount(int threshold, int rows, int cols) {
    int flag[][] = new int[rows][cols]; //记录是否已经走过
    return helper(0, 0, rows, cols, flag, threshold);
}

private int helper(int i, int j, int rows, int cols, int[][] flag, int
threshold) {
    if (i < 0 || i >= rows || j < 0 || j >= cols ||
            numSum(i) + numSum(j) > threshold || flag[i][j] == 1)
        return 0;
    flag[i][j] = 1;
    return helper(i - 1, j, rows, cols, flag, threshold)
            + helper(i + 1, j, rows, cols, flag, threshold)
            + helper(i, j - 1, rows, cols, flag, threshold)
            + helper(i, j + 1, rows, cols, flag, threshold) + 1;
}

private int numSum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += i % 10;
        i = i / 10;
    }
```

```
    return sum;
}
```

常用数据结构

1. 二叉树

1.1 二叉树的前序遍历

- 递归

```
private List<Integer> res = new ArrayList<>();
public List<Integer> preorderTraversal(TreeNode root) {
    if (root == null) {
        return res;
    }
    res.add(root.val);
    preorderTraversal(root.left);
    preorderTraversal(root.right);
    return res;
}
```

- 非递归

```
public List<Integer> preorderTraversal2(TreeNode root) {
    List<Integer> list = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            list.add(root.val);
            stack.push(root);
            root = root.left;
        }

        root = stack.pop();
        root = root.right;
    }
    return list;
}
```

1.2 二叉树的中序遍历

- 递归

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> list = new ArrayList<>();
    this.inorder(root, list);
    return list;
}
```

```
private void inorder(TreeNode root, List<Integer> list) {
    if (root == null) {
        return ;
    }
    inorder(root.left, list);
    list.add(root.val);
    inorder(root.right, list);
}
```

- 非递归

```
public List<Integer> inorderTraversal2(TreeNode root) {
    List<Integer> list = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }

        root = stack.pop();
        list.add(root.val);
        root = root.right;
    }
    return list;
}
```

1.3 二叉树的后序遍历

- 递归

```
private List<Integer> res = new ArrayList<>();
public List<Integer> postorderTraversal(TreeNode root) {
    if (root == null) {
        return res;
    }
    postorderTraversal(root.left);
    postorderTraversal(root.right);
    res.add(root.val);
    return res;
}
```

- 非递归（DFS）

```
public List<Integer> postorderTraversal(TreeNode root) {
    LinkedList<Integer> list = new LinkedList<>();
    if (root == null) {
        return list;
    }
```

```
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();
        list.addFirst(node.val);
        if (node.left != null) {
            stack.push(node.left);
        }
        if (node.right != null) {
            stack.push(node.right);
        }
    }
    return list;
}
```

1.4 二叉树的层序遍历

- 递归

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> levels = new ArrayList<>();
    if (root == null) {
        return levels;
    }

    levelOrderHelper(root, 0, levels);
    return levels;
}

private void levelOrderHelper(TreeNode root, int depth, List<List<Integ
er>> levels) {
    if (root == null) {
        return;
    }
    if (depth == levels.size()) {
        levels.add(new ArrayList<Integer>());
    }
    levels.get(depth).add(root.val);

    if (root.left != null) {
        levelOrderHelper(root.left, depth + 1, levels);
    }
    if (root.right != null) {
        levelOrderHelper(root.right, depth + 1, levels);
    }
}
```

- 非递归（BFS）

```
public List<List<Integer>> levelOrder2(TreeNode root) {
```

```
    List<List<Integer>> levels = new ArrayList<>();
    if (root == null) {
        return levels;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        List<Integer> list = new ArrayList<>();
        int levelLength = queue.size();
        for (int i = 0; i < levelLength; ++i) {
            TreeNode node = queue.poll();
            list.add(node.val);
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        levels.add(list);
    }
    return levels;
}
```

2. 链表

2.1 单链表的插入

```
public void insert(int data){
    ListNode node = new ListNode(data);
    if (null == head){
        head = node;
        return;
    }

    ListNode tmp = head;
    while(tmp.next != null){
        tmp = tmp.next;
    }
    tmp.next = node;
}
```

2.2 单链表的删除

```
public void remove(int data){
    if (null == head){
        return;
    }
```

```
    if (head.val == data){
        head = head.next;
        return;
    }

    ListNode p = this.head;
    while(null != p.next){
        if (p.next.val == data){
            break;
        }
        p = p.next;
    }

    if (null != p.next){
        p.next = p.next.next;
    }
}
```

2.3 单链表的反转

```
public static ListNode reverseList2(ListNode head) {
    ListNode pre = null;

    while (head != null) {
        ListNode p = head.nextNode;
        head.nextNode = pre;
        pre = head;
        head = p;
    }
    return pre;
}
```

========== **end** ==========

本书来自于白夜行的博客 博客地址：https://blog.csdn.net/baiye_xing/