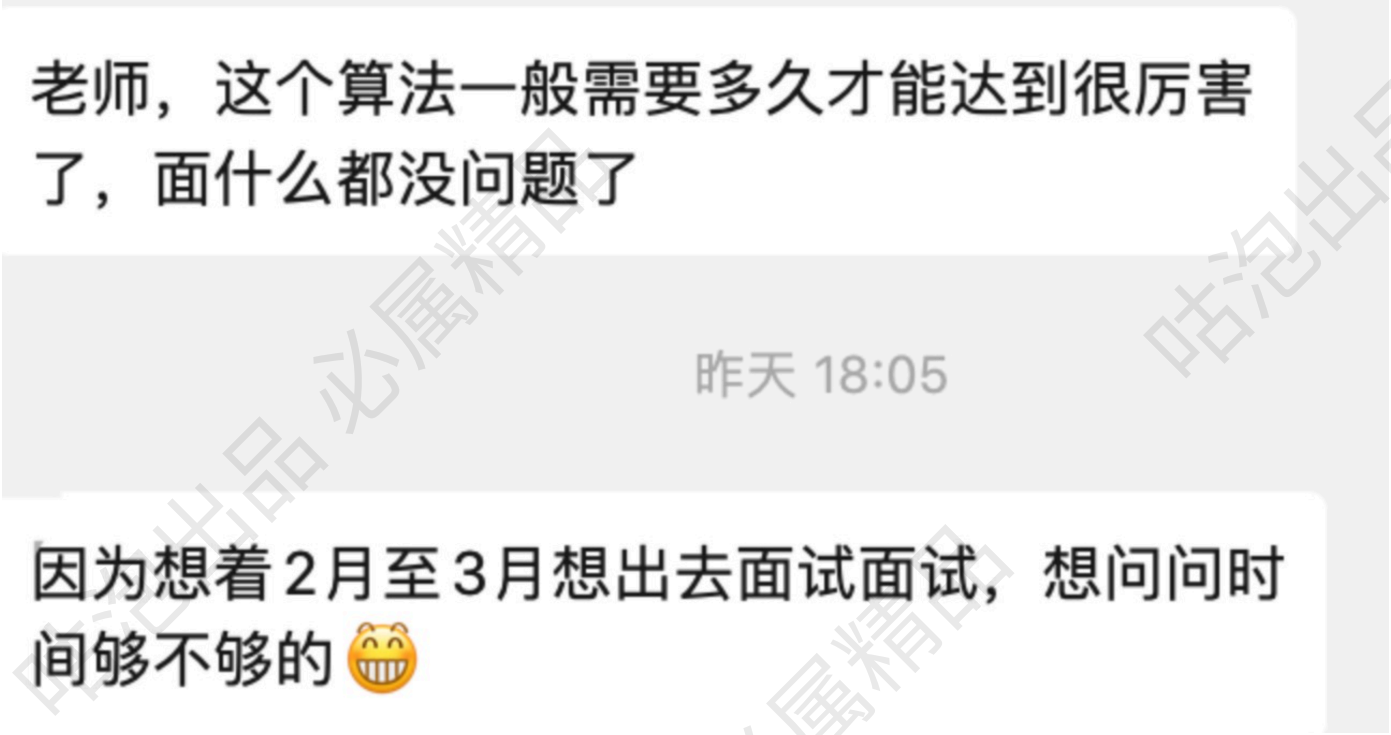
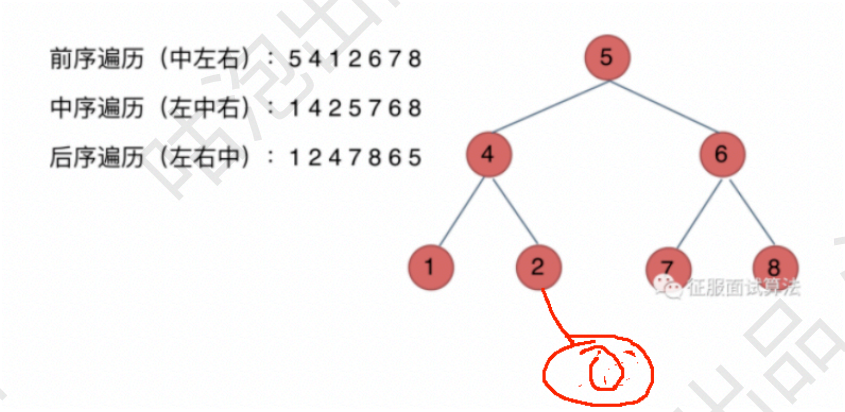
2023年1月7日 第三次直播课



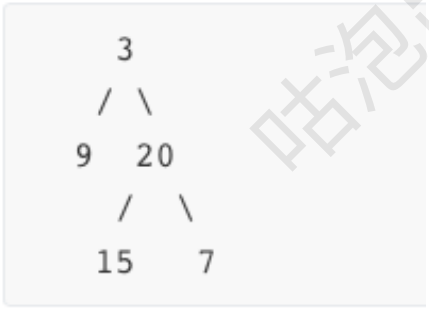
1.本章作业：

1.搞清楚前中后序到底咋回事，将下面增加0之后的序列写出来



2. 讲义1.5 通过序列构造二叉树，自己根据中序和后序

3.自己画下面的处理图的中序处理结果：




```
public static void inOrder(TreeNode root, List<Integer> res) {
    if (root == null) { return;}
    inOrder(root.left, res);
    res.add(root.val);
    inOrder(root.right, res);
}
```

4. 理解《数组和双指针》中的合并两个数组和对撞型双指针，然后来理解归并排序和快速排序
5. 快速排序有个极其重要的拓展题，在无序数组中找第K大，原理是基于pivot执行一次对撞后，pivot的位置唯一确定，此时可以根据pivot的索引和K的关系来判断，接下来该往左侧还是右侧找。

6.二分：将递归和非递归两种方式，背下来。通过案例来测试边界

7.二叉树的层次遍历，本身不难，核心问题是搞清楚如何将每层给分开。

核心设计：用size表示上一层元素的个数，变成0时就代表上一层已经处理完毕，此时队列里的元素刚好就是下一层的所有元素。

2. 树和递归

2.1 递归的地位

二叉树：绝对的重点

二叉树：前中后

二分查找（中序），快速排序（前序）、归并排序（后序）

回溯思想：递归的拓展->解题模板

动态规划：公认最难的思想。->解题，从后向前找递归，从前向后来计算。

贪心：不必单独学习理论，将常见的题目刷会就行了。(青蛙)

2.2 今天的安排：

递归 二叉树

下午：层次遍历、二分查找

递归的学习：

1.看懂别人代码

2.再尝试自己写，自己验证

怎么理解：

debug不好用

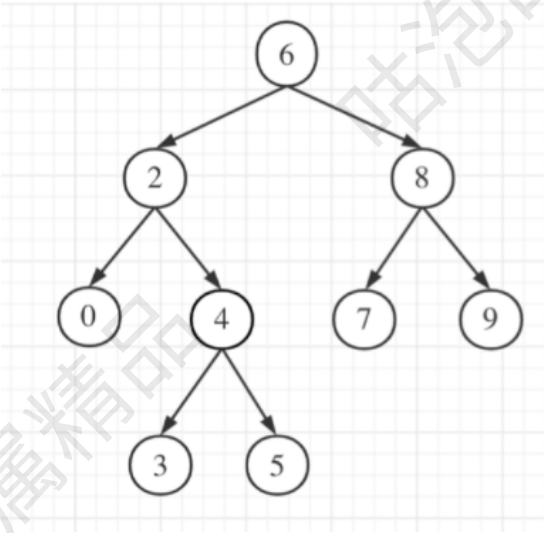
递归有两个基本的特征：

①终止判断在调用递归的前面。

②执行时范围不断缩小，这样才能触底反弹，这里的底正好就是上面的终止条件。

怎么写递归：

3.树怎么考



类别：

1.层次遍历以及相关变形问题　简单的问题，

2.树前序和后序，不好区分，递归，整体难，考察多

3.树的中序遍历：核心框架与二分查找的思想是一致的->提高查询效率->（不用写代码）平衡树->（不用写代码）红黑树（和技术面结合）

4.树的拓展问题：二分查找、归并快速和快速排序（理解，会写，能对）。

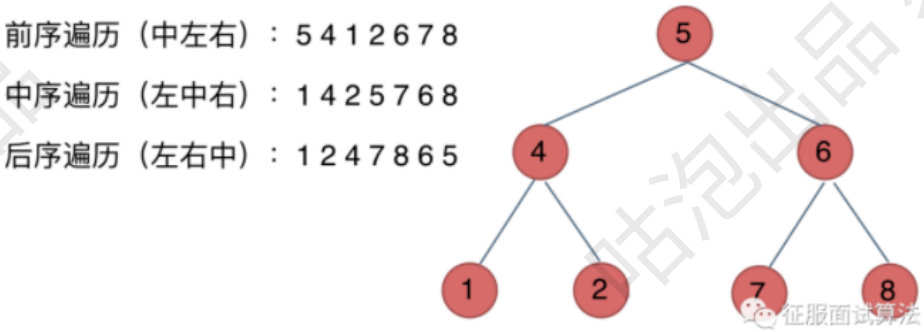
5.树的拓展：堆、红黑树、线段树、哈夫曼、B+，跳表(理解就行，不必写代码)

　红黑树：和技术面试结合，Map hashMap

　nginx

4. 前中后序的概念

前中后序遍历是针对谁来说的



中：【1 4 2 5 7 6 8】

规律：

1.前序的第一个元素是当前子树根节点

2.后序的最后一个元素是当前子树根节点

3.根据根节点和中序遍历序列，可以将根的左右子树分开

中：【1 4 2】5【7 6 8】

【1】4【2】

【7】 6【8】

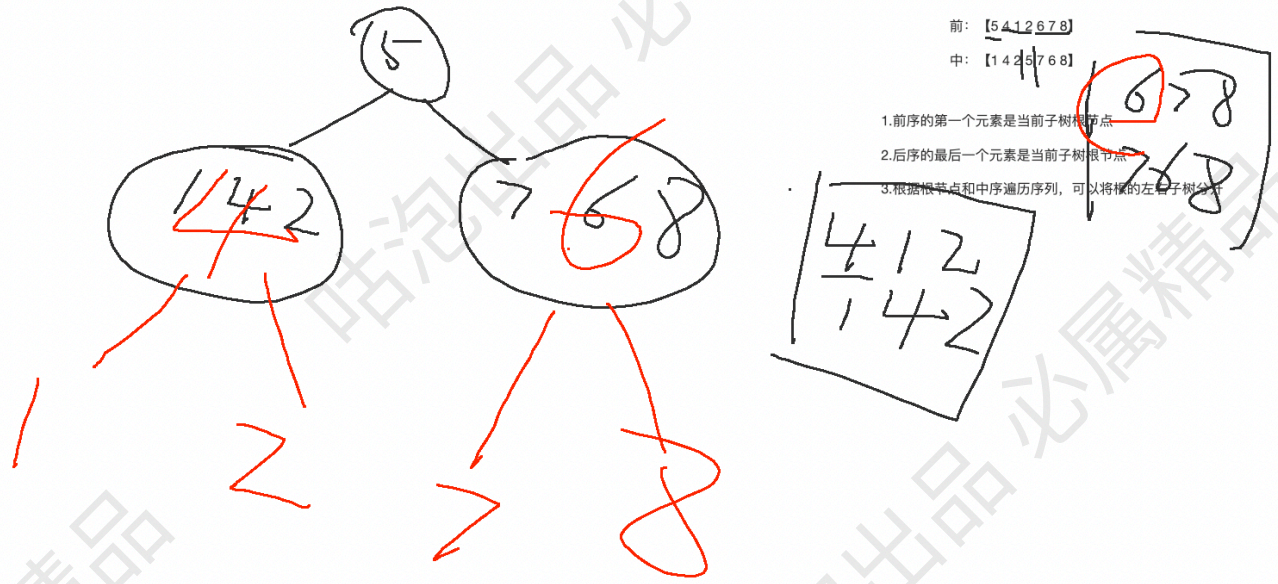
2.给你遍历后的序列，能否将原始二叉树恢复出来？ 可以

假如能的话，需要几个序列?

A：前中

B 前后 X

C 中后



前中为例

前：【5 4 1 2 6 7 8】

中：【1 4 2 5 7 6 8】

后：【1 2 4 7 8 6 5】

5 二叉树的题目

第一类：高度 深度的问题 ， Math.max,min

最小深度的时候，注意是从根到叶子结点。

注意的坑：

找最大	初始化为0
找最小	初始为最大

0
max(left ,right) =2

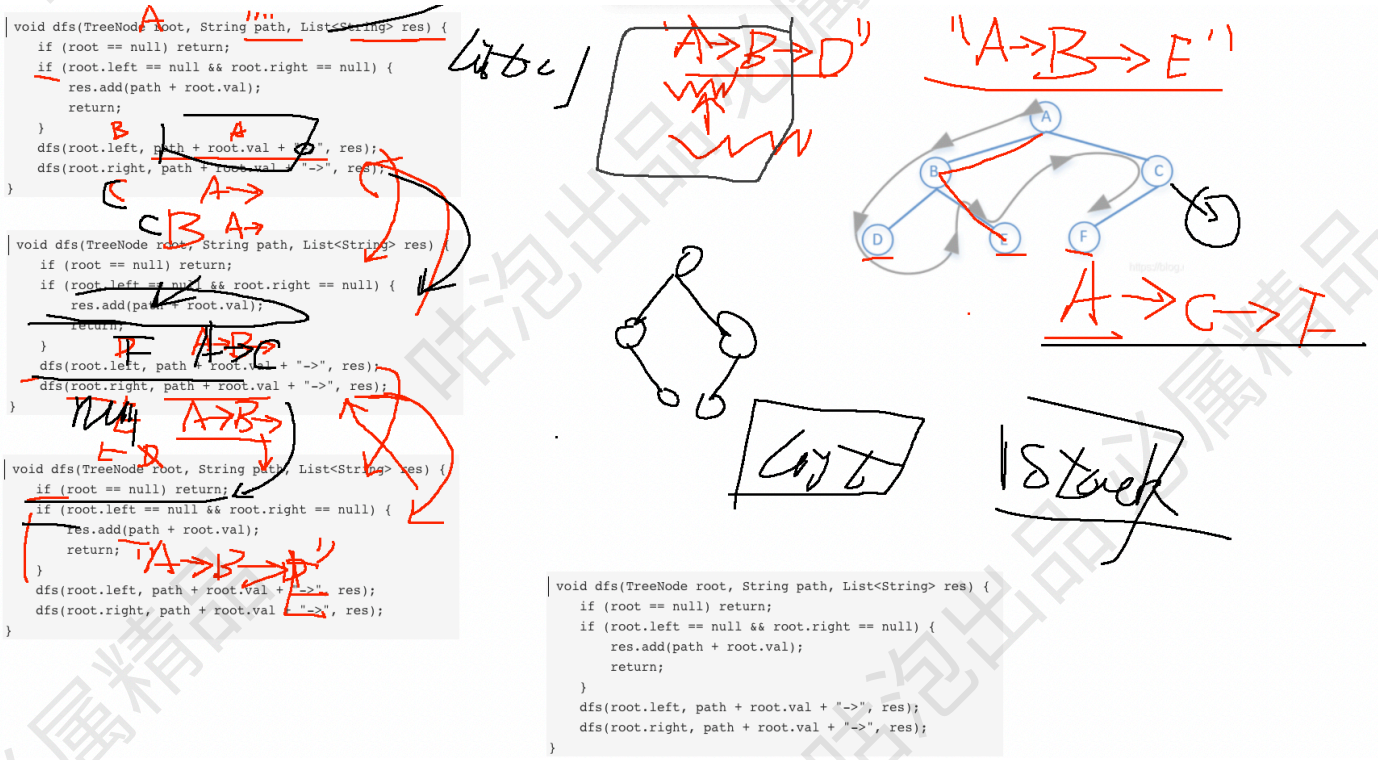
最小：
min(0 ,right)

学习策略，针对一个单元：

第一阶段：看懂，20min，直接去学习别人写更好。

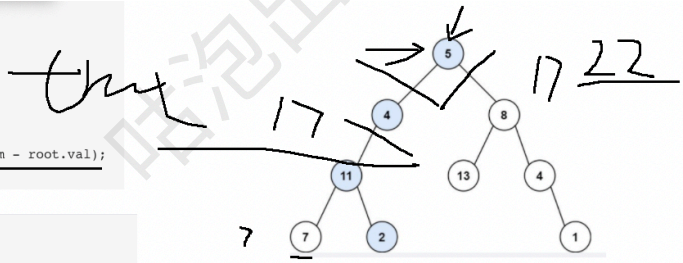
第二阶段：自己写

第三步：白纸 ， LeetCode



```
void dfs(TreeNode root, String path, List<String> res) {
    if (root == null) return;
    if (root.left == null && root.right == null) {
        res.add(path + root.val);
        return;
    }
    dfs(root.left, path + root.val + "->", res);
    dfs(root.right, path + root.val + "->", res);
}
```

```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        return sum == root.val;
    }
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
}
```



```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        return sum == root.val;
    }
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
}
```

```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        return sum == root.val;
    }
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
}
```

```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        return sum == root.val;
    }
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.r
}
```

6.归并算法

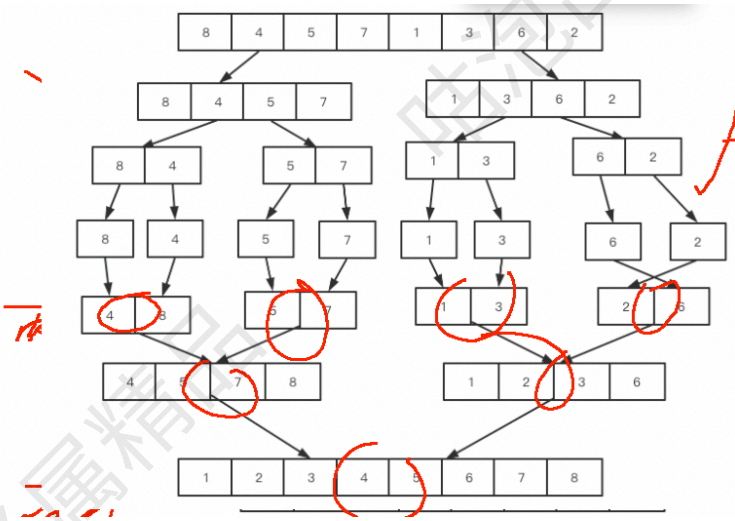
学习的时候，先将《数组和双指针》一章的合并两个数组的代码搞清楚再研究，否则晕！

1.归并里存在一个合并两个有序数组的过程，详细见《数组和双指针》一章。

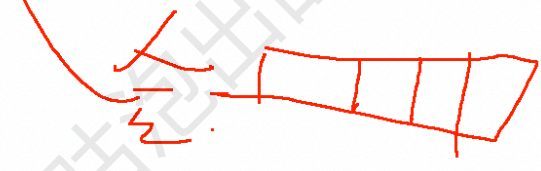
2.有序数组合并比单纯的比较整个数组，花费(比较)的时间要少很多。

3.归并是二叉树的后序遍历+两个有序数组的合并。

4.所有的操作（拆分）是在一个数组上的（对照代码理解）



```
void mergeSort(int[] array, int start, int end, int temp[]) {
    if (start >= end) {
        return;
    }
    //先递归访问到每个元素(也在结点)，得到元素之后再分别处理(两个合并)
    //与二叉树的后序遍历非常像
    mergeSort(array, start, (start + end) >>1, temp);
    mergeSort(array, (start + end) >>1 + 1, end, temp);
    merge(array, start, end, temp);
}
```

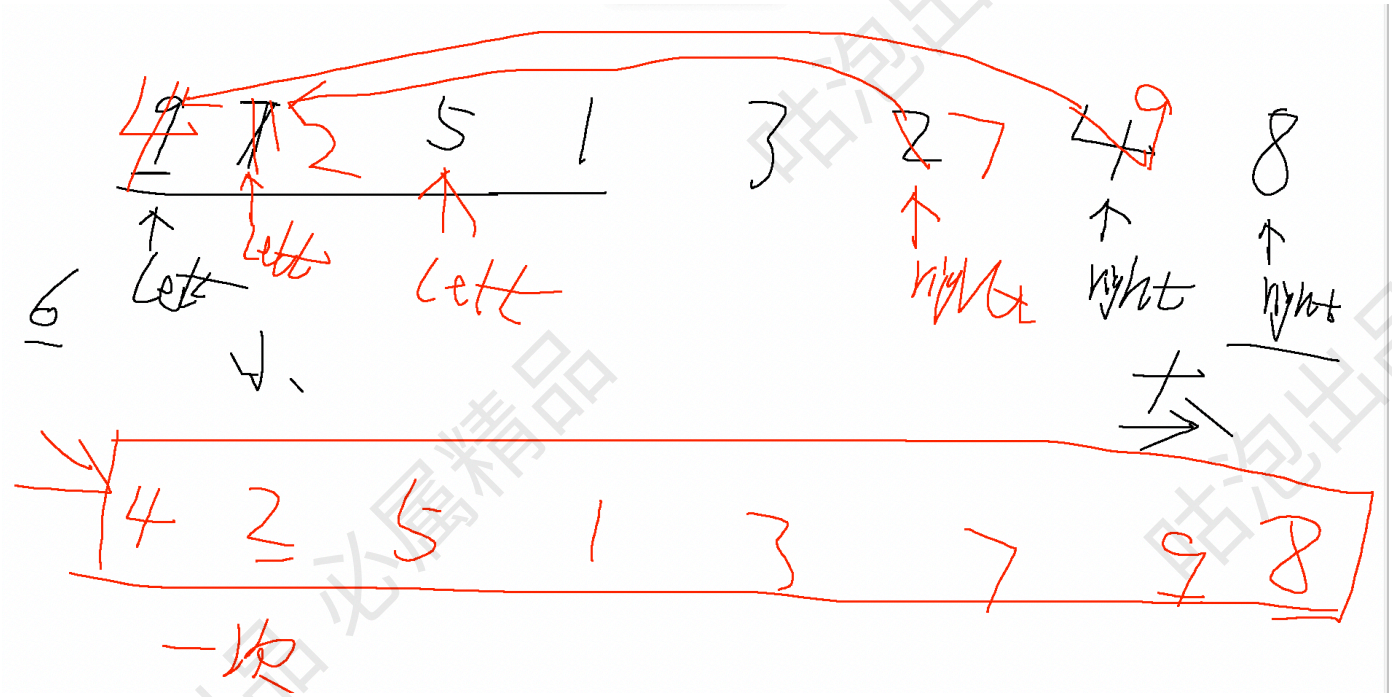


7.快速排序

《数组和双指针》

双指针的两种类型：快慢和对撞

要学习快速排序，先将《数组和双指针》中对撞双指针的几个问题搞清楚，否则晕！



上图有个不规范的，6不在数组序列中

快速排序根据pivot执行一次排序，就是执行了一次对撞型双指针。

- 提前选定一个哨兵元素pivot，选的常见方式：就选第一个、选中间或者随机选择一个位置。
- 比较的原则是所有比pivot小的放到左侧，比pivot大的放在右侧
- 中间空出来的位置放pivot。

可能出现的情况：每次都选了分组中最大或者最小的元素，这是快速排序的最差情况，此时的时间复杂度为O(n^2)，而平均复杂度是O(nlogn)。

- 完成一轮之后，此时pivot的位置完全确定，不会再变。
- 此时pivot左右两侧仍然是无序的，可以分别视为pivot的左右子树，因此可以继续进行递归处理。

8.二分查找和二叉树的中序遍历

老师曾经参加过的线上面试：

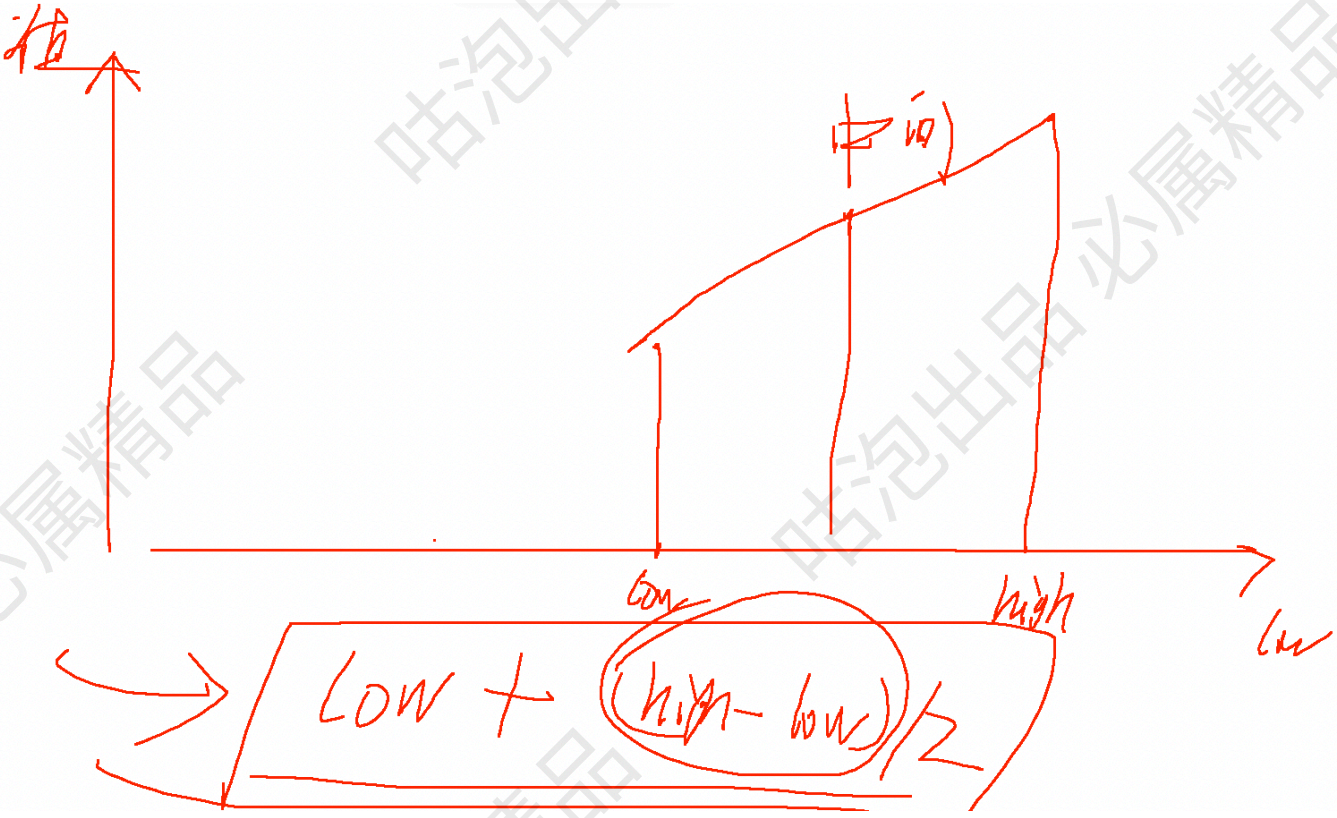
Java(javac 1.8) 重置

```
        //int a = in.nextInt();
        //System.out.println(a);


        System.out.println("Hello World!");
    }

    pubic int search( List list){

        int left=0,right=list.length-1;
        int mid=0;
        while(left<=right){
            mid=left+(right-left)>>1;
            if((list[mid]>list[mid+1])&&(list[mid]<list[mid-1]){
                right=mid-1;
            }else if((list[mid]<list[mid+1])&&(list[mid]>list[mid-1]){
                left=mid+1;
            }else if((list[mid]>list[mid+1])&&(list[mid]>list[mid-1]) ){
            return list[mid];
        }

        return -1;

    }
}
```

第1轮面试题目已锁定，不可修改

```
int mid = (low + high)/2;
int mid=low+(high-low)/2;->
 int mid=2low+(high-low)/2 =(low + high)/2
```

理解，如果传入进来的low和high不溢出，那么 low+(high-low)/2 一定不会溢出。



```
mid=low+(high-low)>>1;
```

左移一位：相当于乘以2

右移一位：相当于除以2

天坑：

```
mid=low+(high-low)     >>1;
变成 mid=high>>1;
```

正确的做法：

mid=low+((high-low) >>1);

1 1 1 2 2 2 2 2 3 3 3

二分查找的几个要点：

1.不假思索就能写

2.假如序列里面有重复，该怎么办

3.只要涉及到部分序列是有序的，此时让你优化的话，就要考虑是否可以用二分。参考《2.5 优化求平方根》

1 5 6 7 9 | 8 4 3

9.二叉树的层次遍历

首先需要使用一个队列，作为缓存。

如果不需要区分层次，则可以一边遍历树，一边用队列来缓存。

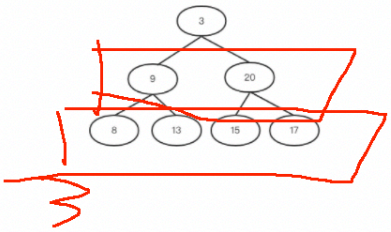
需要区分层次的时候，需要增加一个size变量。

size初始值为queue.size()。

然后每一个元素node出队，则size-1，并且将node的左右子节点入队，

当size=0 的时候，上一层的元素已经全部出队，而此时队列里的元素恰好就是下一层的全部元素。

所以，循环，继续让size=queue.size()，然后继续让node出队、size-1和node的左右子节点入队。



size=1

3

3 出 size=0

9 20

size= queue.size=2

size-1=1 size-1=0

8 13 15 17