社招面试总结----算法题篇

股票买卖(头条)

1. 买卖股票的最佳时机

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
```

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
```

解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

题解

纪录两个状态，一个是最大利润，另一个是遍历过的子序列的最小值。知道之前的最小值我们就可以算出当前天可能的最大利润是多少

```
class Solution {
    public int maxProfit(int[] prices) {
        // 7,1,5,3,6,4
        int maxProfit = 0;
        int minNum = Integer.MAX_VALUE;
        for (int i = 0; i < prices.length; i++) {
            if (Integer.MAX_VALUE != minNum && prices[i] - minNum > maxProfit)
{
                maxProfit = prices[i] - minNum;
            }

            if (prices[i] < minNum) {
                minNum = prices[i];
            }
        }
        return maxProfit;
    }
}
```

2. 买卖股票的最佳时机 II

这次改成股票可以买卖多次，但是你必须要在出售股票之前把持有的股票卖掉。

示例 1:

输入：[7,1,5,3,6,4]输出：7 解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。

示例 2:

输入：[1,2,3,4,5]输出：4 解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:

输入：[7,6,4,3,1]

输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。

题解

由于可以无限次买入和卖出。我们都知道炒股想挣钱当然是低价买入高价抛出，那么这里我们只需要从第二天开始，如果当前价格比之前价格高，则把差值加入利润中，因为我们可以昨天买入，今日卖出，若明日价更高的话，还可以今日买入，明日再抛出。以此类推，遍历完整个数组后即可求得最大利润。

```
class Solution {
    public int maxProfit(int[] prices) {
        // 7,1,5,3,6,4
        int maxProfit = 0;
        for (int i = 0; i < prices.length; i++) {
            if (i != 0 && prices[i] - prices[i-1] > 0) {
                maxProfit += prices[i] - prices[i-1];
            }
        }
        return maxProfit;
    }
}
```

LRU cache (头条、蚂蚁)

题目

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作？

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

题解

这道题在今日头条、快手或者硅谷的公司中是比较常见的，代码要写的还蛮多的，难度也是 hard 级别。

最重要的是 LRU 这个策略怎么去实现,

很容易想到用一个链表去实现最近使用的放在链表的最前面。

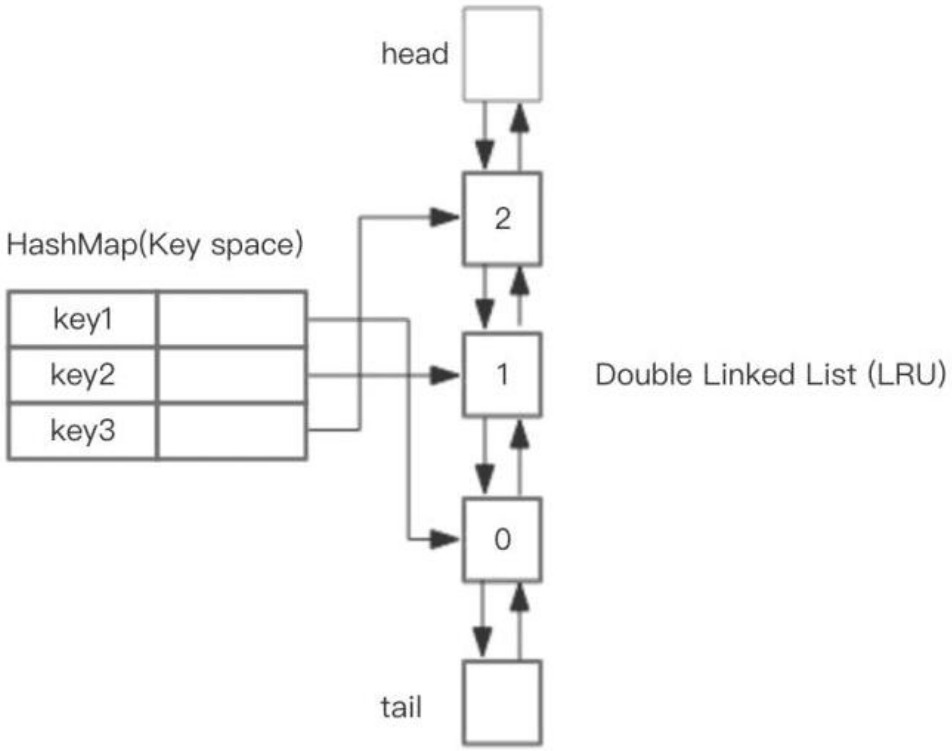
比如 get 一个元素,相当于被使用过了，这个时候它需要放到最前面，再返回值,

set 同理。

那如何把一个链表的中间元素，快速的放到链表的开头呢？
很自然的我们想到了双端链表。

基于 HashMap 和 双向链表实现 LRU 的

整体的设计思路是，可以使用 HashMap 存储 key，这样可以做到 save 和 get key 的时间都是 O(1)，而 HashMap 的 Value 指向双向链表实现的 LRU 的 Node 节点，如图所示。



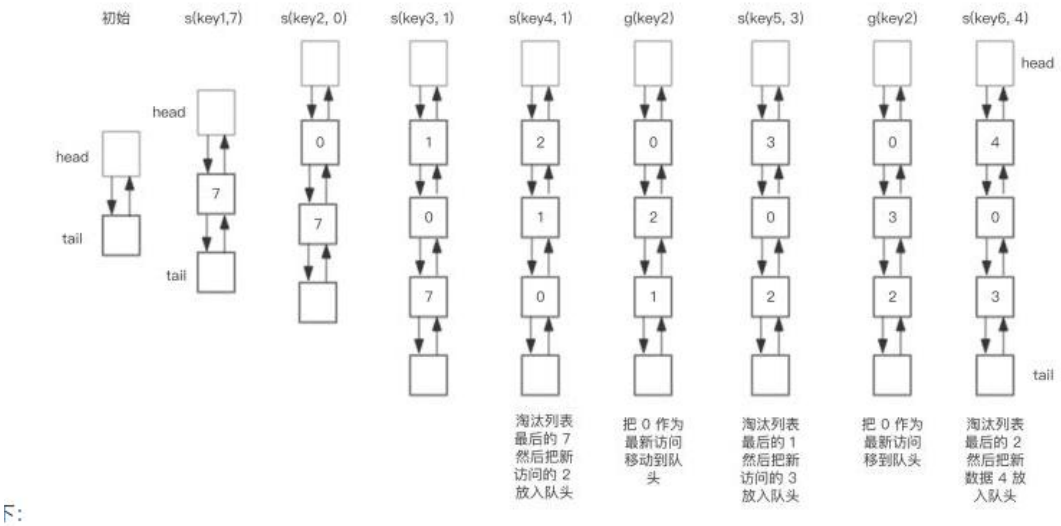
LRU 存储是基于双向链表实现的，下面的图演示了它的原理。其中 head 代表双向链表的表头，tail 代表尾部。首先预先设置 LRU 的容量，如果存储满了，可以通过 O(1) 的时间淘汰掉双向链表的尾部，每次新增和访问数据，都可以通过 O(1)的效率把新的节点增加到对头，或者把已经存在的节点移动到队头。

下面展示了，预设大小是 3 的，LRU 存储的在存储和访问过程中的变化。为了简化图复杂度，图中没有展示 HashMap 部分的变化，仅仅演示了上图 LRU 双向链表的变化。我们对这个 LRU 缓存的操作序列如下：

```
save("key1", 7)
save("key2", 0)
```

```
save("key3", 1)
save("key4", 2)
get("key2")
save("key5", 3)
get("key2")
save("key6", 4)
```

相应的 LRU 双向链表部分变化如下：



下：

总结一下核心操作的步骤:

save(key, value)，首先在 HashMap 找到 Key 对应的节点，如果节点存在，更新节点的值，并把这个节点移动队头。如果不存在，需要构造新的节点，并且尝试把节点塞到队头，如果 LRU 空间不足，则通过 tail 淘汰掉队尾的节点，同时在 HashMap 中移除 Key。

get(key)，通过 HashMap 找到 LRU 链表节点，因为根据 LRU 原理，这个节点是最新访问的，所以要把节点插入到队头，然后返回缓存的值。

```
private static class DLinkedNode {
        int key;
        int value;
```

```
        DLinkedNode pre;
        DLinkedNode post;
    }

    /**
     * 总是在头节点中插入新节点.
     */
    private void addNode(DLinkedNode node) {

        node.pre = head;
        node.post = head.post;

        head.post.pre = node;
        head.post = node;
    }

    /**
     * 摘除一个节点.
     */
    private void removeNode(DLinkedNode node) {
        DLinkedNode pre = node.pre;
        DLinkedNode post = node.post;

        pre.post = post;
        post.pre = pre;
    }

    /**
     * 摘除一个节点,并且将它移动到开头
     */
    private void moveToHead(DLinkedNode node) {
        this.removeNode(node);
        this.addNode(node);
    }

    /**
     * 弹出最尾巴节点
     */
    private DLinkedNode popTail() {
        DLinkedNode res = tail.pre;
        this.removeNode(res);
        return res;
    }
```

```
private HashMap<Integer, DLinkedNode>
        cache = new HashMap<Integer, DLinkedNode>();
private int count;
private int capacity;
private DLinkedNode head, tail;

public LRUCache(int capacity) {
    this.count = 0;
    this.capacity = capacity;

    head = new DLinkedNode();
    head.pre = null;

    tail = new DLinkedNode();
    tail.post = null;

    head.post = tail;
    tail.pre = head;
}

public int get(int key) {

    DLinkedNode node = cache.get(key);
    if (node == null) {
        return -1; // cache 里面没有
    }

    // cache 命中,挪到开头
    this.moveToHead(node);

    return node.value;
}


public void put(int key, int value) {
    DLinkedNode node = cache.get(key);

    if (node == null) {

        DLinkedNode newNode = new DLinkedNode();
        newNode.key = key;
        newNode.value = value;

        this.cache.put(key, newNode);
```

```
            this.addNode(newNode);

            ++count;

            if (count > capacity) {
                // 最后一个节点弹出
                DLinkedNode tail = this.popTail();
                this.cache.remove(tail.key);
                count--;
            }
        } else {
            // cache 命中,更新 cache.
            node.value = value;
            this.moveToHead(node);
        }
    }
```

二叉树层次遍历(头条)

这个题目之前也讲过，Leetcode 102 题。

题目

给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。

例如:

给定二叉树: [3,9,20,null,null,15,7],

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

返回其层次遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

题解

我们数据结构的书上教的层序遍历,就是利用一个队列,不断的把左子树和右子树入队。但是这个题目还要要求按照层输出。所以关键的问题是： **如何确定是在同一层的**。
我们很自然的想到：

如果在入队之前,把上一层所有的节点出队,那么出队的这些节点就是上一层的列表。

由于队列是先进先出的数据结构,所以这个列表是从左到右的。

```
/** * Definition for a binary tree node. * public class TreeNode { *     int val; *     TreeNode left; *     TreeNode right; *     TreeNode(int x) { val = x; } * }
 */class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            List<Integer> currentRes = new LinkedList<>();
            // 当前队列的大小就是上一层的节点个数，依次出队
            while (size > 0) {
                TreeNode current = queue.poll();
                if (current == null) {
                    continue;
                }
                currentRes.add(current.val);
                // 左子树和右子树入队.
                if (current.left != null) {
```

```
                queue.add(current.left);
            }
            if (current.right != null) {
                queue.add(current.right);
            }
            size--;
        }
        res.add(currentRes);
    }
    return res;
  }
}
```

这道题可不可以用非递归来解呢?

递归的子问题：遍历当前节点，对于当前层，遍历左子树的下一层层,遍历右子树的下一层

递归结束条件: 当前层,当前子树节点是 null

```
/** * Definition for a binary tree node. * public class TreeNode { *     int val; *     TreeNode left; *     TreeNode right; *     TreeNode(int x) { val = x; } * } */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }
        levelOrderHelper(res, root, 0);
        return res;
    }

    /**     * @param depth 二叉树的深度     */
    private void levelOrderHelper(List<List<Integer>> res, TreeNode root, int depth) {
        if (root == null) {
            return;
        }

        if (res.size() <= depth) {
            // 当前层的第一个节点,需要new 一个list 来存当前层.
            res.add(new LinkedList<>());
```

```
        }
        // depth 层,把当前节点加入
        res.get(depth).add(root.val);
        // 递归的遍历下一层.
        levelOrderHelper(res, root.left, depth + 1);
        levelOrderHelper(res, root.right, depth + 1);
    }
}
```

二叉树转链表(快手)

这是 Leetcode 104 题。

给定一个二叉树，原地将它展开为链表。

例如，给定二叉树

```
    1
   / \
  2   5
 / \   \
3   4   6
```

将其展开为：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

这道题目的关键是转换的时候递归的时候记住是先转换右子树，再转换左子树。所以需要记录一下右子树转换完之后链表的头结点在哪里。注意没有新定义一个 next 指针，而是直接将 right 当做 next 指针,那么 Left 指针我们赋值成 null 就可以了。

```
class Solution {
    private TreeNode prev = null;

    public void flatten(TreeNode root) {
        if (root == null)  return;
        flatten(root.right); // 先转换右子树
        flatten(root.left);
        root.right = prev;  // 右子树指向链表的头
        root.left = null; // 把左子树置空
        prev = root; // 当前结点为链表头
    }
}
```

用递归解法,用一个 stack 记录节点,右子树先入栈，左子树后入栈。

```
class Solution {
    public void flatten(TreeNode root) {
        if (root == null) return;
        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode current = stack.pop();
            if (current.right != null) stack.push(current.right);
            if (current.left != null) stack.push(current.left);
            if (!stack.isEmpty()) current.right = stack.peek();
            current.left = null;
        }
    }
}
```

二叉树寻找最近公共父节点(快手)

Leetcode 236 二叉树的最近公共父亲节点

题解

从根节点开始遍历，如果 node1 和 node2 中的任一个和 root 匹配，那么 root 就是最低公共祖先。 如果都不匹配，则分别递归左、右子树，如果有一个 节点出现在左子树，并且另一个节点出现在右子树，则 root 就是最低公共祖先. 如果两个节点都出现在左子树，则说明最低公共祖先在左子树中，否则在右子树。

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        //发现目标节点则通过返回值标记该子树发现了某个目标结点
        if(root == null || root == p || root == q) return root;
        //查看左子树中是否有目标结点，没有为null
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        //查看右子树是否有目标节点，没有为null
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        //都不为空，说明做右子树都有目标结点，则公共祖先就是本身
        if(left!=null&&right!=null) return root;
        //如果发现了目标节点，则继续向上标记为该目标节点
        return left == null ? right : left;
    }
}
```

数据流求中位数(蚂蚁)

面了蚂蚁中台的团队,二面面试官根据汇报层级推测应该是 P9 级别及以上，在美国面我，面试风格偏硅谷那边。题目是 hard 难度的,leetcode 295 题。

这道题目是 Leetcode 的 hard 难度的原题。给定一个数据流，求数据流的中位数，求中位数或者 topK 的问题我们通常都会想用堆来解决。

但是面试官又进一步加大了难度，他要求内存使用很小，没有磁盘，但是压榨空

间的同时可以忍受一定时间的损耗。且面试官不仅要求说出思路，要写出完整可经过大数据检测的 production code。

先不考虑内存

不考虑内存的方式就是 Leetcode 论坛上的题解。

基本思想是建立两个堆。左边是大根堆，右边是小根堆。
如果是奇数的时候，大根堆的堆顶是中位数。

例如：[1,2,3,4,5]

大根堆建立如下：

```
  3
 / \
1   2
```

小根堆建立如下:

```
  4
 /
5
```

偶数的时候则是最大堆和最小堆顶的平均数。

例如: [1, 2, 3, 4]

大根堆建立如下：

```
  2
 /
1
```

小根堆建立如下

```
  3
 /
4
```

然后再维护一个奇数偶数的状态即可求中位数。

```
public class MedianStream {
    private PriorityQueue<Integer> leftHeap = new PriorityQueue<>(5,
Collections.reverseOrder());
    private PriorityQueue<Integer> rightHeap = new PriorityQueue<>(5);

    private boolean even = true;

    public double getMedian() {
        if (even) {
            return (leftHeap.peek() + rightHeap.peek()) / 2.0;
        } else {
            return leftHeap.peek();
        }
    }

    public void addNum(int num) {
        if (even) {
            rightHeap.offer(num);
            int rightMin = rightHeap.poll();
            leftHeap.offer(rightMin);
        } else {
            leftHeap.offer(num);
            int leftMax = leftHeap.poll();
            rightHeap.offer(leftMax);
        }
        System.out.println(leftHeap);
        System.out.println(rightHeap);
        // 奇偶变换.
        even = !even;
    }
}
```

压榨内存

但是这样做的问题就是可能内存会爆掉。如果你的流无限大，那么意味着这些数据都要存在内存中，堆必须要能够建无限大。如果内存必须很小的方式，用时间换空间。

- 流是可以重复去读的, 用时间换空间;
- 可以用分治的思想,先读一遍流,把流中的数据个数分桶;

- 分桶之后遍历桶就可以得到中位数落在哪个桶里面,这样就把问题的范围缩小了。

```
public class Median {
    private static int BUCKET_SIZE = 1000;

    private int left = 0;
    private int right = Integer.MAX_VALUE;

    // 流这里用 int[] 代替
    public double findMedian(int[] nums) {
        // 第一遍读取 stream 将问题复杂度转化为内存可接受的量级.
        int[] bucket = new int[BUCKET_SIZE];
        int step = (right - left) / BUCKET_SIZE;
        boolean even = true;
        int sumCount = 0;
        for (int i = 0; i < nums.length; i++) {
            int index = nums[i] / step;
            bucket[index] = bucket[index] + 1;
            sumCount++;
            even = !even;
        }
        // 如果是偶数,那么就需要计算第 topK 个数
        // 如果是奇数, 那么需要计算第 topK 和 topK+1 的个数.
        int topK = even ? sumCount / 2 : sumCount / 2 + 1;

        int index = 0;
        int indexBucketCount = 0;
        for (index = 0; index < bucket.length; index++) {
            indexBucketCount = bucket[index];
            if (indexBucketCount >= topK) {
                // 当前 bucket 就是中位数的 bucket.
                break;
            }
            topK -= indexBucketCount;
        }

        // 划分到这里其实转化为一个 topK 的问题, 再读一遍流.
        if (even && indexBucketCount == topK) {
            left = index * step;
            right = (index + 2) * step;
            return helperEven(nums, topK);
            // 偶数的时候, 恰好划分到在左右两个子段中.
```

```
            // 左右两段中 [topIndex-K + (topIndex-K + 1)] / 2.
        } else if (even) {
            left = index * step;
            right = (index + 1) * step;
            return helperEven(nums, topK);
            // 左边 [topIndex-K + (topIndex-K + 1)] / 2
        } else {
            left = index * step;
            right = (index + 1) * step;
            return helperOdd(nums, topK);
            // 奇数, 左边 topIndex-K
        }
    }
}
```

这里边界条件我们处理好之后,关键还是 helperOdd 和 helperEven 这两个函数怎么去求 topK 的问题. 我们还是转化为一个 topK 的问题,那么求 top-K 和 top(K+1)的问题到这里我们是不是可以用堆来解决了呢? 答案是不能,考虑极端情况。

中位数的重复次数非常多

```
eg:
[100,100,100,100,100...] (1000 亿个 100)
```

你的划分恰好落到这个桶里面,内存同样会爆掉。

再用时间换空间

假如我们的划分 bucket 大小是 10000,那么最大的时候区间就是 20000。(对应上面的偶数且落到两个分桶的情况)

那么既然划分到某一个 bucket 了,我们直接用数数字的方式来求 topK 就可以了,用堆的方式也可以,更高效一点,其实这里问题规模已经划分到很小了,两种方法都可以。

我们选用 TreeMap 这种数据结构计数。然后分奇数偶数去求解。

```
private double helperEven(int[] nums, int topK) {
        TreeMap<Integer, Integer> map = new TreeMap<>();
```

```
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] >= left && nums[i] <= right) {
                if (!map.containsKey(nums[i])) {
                    map.put(nums[i], 1);
                } else {
                    map.put(nums[i], map.get(nums[i]) + 1);
                }
            }
        }

        int count = 0;
        int kNum = Integer.MIN_VALUE;
        int kNextNum = 0;
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            int currentCountIndex = entry.getValue();
            if (kNum != Integer.MIN_VALUE) {
                kNextNum = entry.getKey();
                break;
            }
            if (count + currentCountIndex == topK) {
                kNum = entry.getKey();
            } else if (count + currentCountIndex > topK) {
                kNum = entry.getKey();
                kNextNum = entry.getKey();
                break;
            } else {
                count += currentCountIndex;
            }
        }

        return (kNum + kNextNum) / 2.0;
    }

    private double helperOdd(int[] nums, int topK) {
        TreeMap<Integer, Integer> map = new TreeMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] >= left && nums[i] <= right) {
                if (!map.containsKey(nums[i])) {
                    map.put(nums[i], 1);
                } else {
                    map.put(nums[i], map.get(nums[i]) + 1);
                }
            }
        }
```

```
        int count = 0;
        int kNum = Integer.MIN_VALUE;
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            int currentCountIndex = entry.getValue();
            if (currentCountIndex + count >= topK) {
                kNum = entry.getKey();
                break;
            } else {
                count += currentCountIndex;
            }
        }

        return kNum;
    }
```

至此,我觉得算是一个比较好的解决方案。