```
            if (largest != index) {
                swap(heap, largest, index);
            } else {
                break;
            }
            index = largest;
            left = index * 2 + 1;
            right = index * 2 + 2;
        }
    }

    public void heapInsert(HeapNode[] heap, int index, int row, int col,
            int value) {
        heap[index] = new HeapNode(row, col, value);
        int parent = (index - 1) / 2;
        while (index != 0) {
            if (heap[index].value > heap[parent].value) {
                swap(heap, parent, index);
                index = parent;
                parent = (index - 1) / 2;
            } else {
                break;
            }
        }
    }

    public void swap(HeapNode[] heap, int index1, int index2) {
        HeapNode tmp = heap[index1];
        heap[index1] = heap[index2];
        heap[index2] = tmp;
    }

    public boolean isContains(int row, int col, HashSet<String> set) {
        return set.contains(String.valueOf(row + "_" + col));
    }

    public void addPositionToSet(int row, int col, HashSet<String> set) {
        set.add(String.valueOf(row + "_" + col));
    }
```

出现次数的 TOP *K* 问题

【题目】

给定 String 类型的数组 strArr，再给定整数 k，请严格按照排名顺序打印出现次数前 k 名的字符串。

【举例】

strArr=["1","2","3","4"]，k=2

No.1: 1, times: 1

No.2: 2, times: 1

这种情况下，所有的字符串都出现一样多，随便打印任何两个字符串都可以。

strArr=["1","1","2","3"]，k=2

输出：

No.1: 1, times: 2

No.2: 2, times: 1

或者输出：

No.1: 1, times: 2

No.2: 3, times: 1

【要求】

如果 strArr 长度为 N，时间复杂度请达到 $O(N\log k)$。

【进阶题目】

设计并实现 TopKRecord 结构，可以不断地向其中加入字符串，并且可以根据字符串出现的情况随时打印加入次数最多前 k 个字符串，具体为：

1．k 在 TopKRecord 实例生成时指定，并且不再变化（k 是构造函数的参数）。

2．含有 add(String str)方法，即向 TopKRecord 中加入字符串。

3．含有 printTopK()方法，即打印加入次数最多的前 k 个字符串，打印有哪些字符串和对应的次数即可，不要求严格按排名顺序打印。

【举例】

```
TopKRecord record = new TopKRecord(2); // 打印 Top 2 的结构
record.add("A");
record.printTopK();
```

此时打印：

TOP:
Str: A Times: 1
record.add("B");
record.add("B");
record.printTopK();
此时打印：
TOP:
Str: A Times: 1
Str: B Times: 2
或者打印
TOP:
Str: B Times: 2
Str: A Times: 1
record.add("C");
record.add("C");
record.printTopK();
此时打印：
TOP:
Str: B Times: 2
Str: C Times: 2
或者打印
TOP:
Str: C Times: 2
Str: B Times: 2

【要求】

1. 在任何时刻，add 方法的时间复杂度不超过 $O(\log k)$。
2. 在任何时刻，printTopK 方法的时间复杂度不超过 $O(k)$。

【难度】

原问题 尉 ★★☆☆

进阶问题　校　★★★☆

【解答】

原问题。首先遍历 strArr 并统计字符串的词频，例如，strArr=["a","b","b","a","c"]，遍历后可以生成每种字符串及其相关词频的哈希表如下：

key（字符串）	value（相关词频）
"a"	2
"b"	2
"c"	1

用哈希表的每条信息可以生成 Node 类的实例，Node 类如下：

```
public class Node {
        public String str;
        public int times;

        public Node(String s, int t) {
                str = s;
                times = t;
        }
}
```

哈希表中有多少信息，就建立多少 Node 类的实例，并且依次放入堆中，具体过程为：

1．建立一个大小为 k 的小根堆，这个堆放入的是 Node 类的实例。

2．遍历哈希表的每条记录，假设一条记录为(s,t)，s 表示一种字符串，s 的词频为 t，则生成 Node 类的实例，记为(str,times)。

1）如果小根堆没有满，就直接将(str,times)加入堆，然后进行建堆调整(heapInsert 调整)，堆中 Node 类实例之间都以词频（times）来进行比较，词频越小，位置越往上。

2）如果小根堆已满，说明此时小根堆已经选出 k 个最高词频的字符串，那么整个小根堆的堆顶自然代表已经选出的 k 个最高词频的字符串中，词频最低的那个。堆顶的元素记为(headStr,minTimes)。如果 minTimes<times，说明字符串 str 有资格进入当前 k 个最高词频字符串的范围。而 headStr 应该被移出这个范围，所以把当前的堆顶(headStr,minTimes)替换成(str,times)，然后从堆顶的位置进行堆的调整(heapify)。如果 minTimes>=times，说明字符串 str 没有资格进入当前 k 个最高词频字符串的范围，因为 str 的词频还不如目前选出的 k 个最高词频字符串中词频最少的那个，所以什么也不做。

3．遍历完 strArr 之后，小根堆里就是所有字符串中 k 个最高词频的字符串，但要求严

格按排名打印，所以还需要根据词频从大到小完成 k 个元素间的排序。

遍历 strArr 建立哈希表的过程为 $O(N)$。哈希表中记录的条数最多为 N 条，每一条记录进堆时，堆的调整时间复杂度为 $O(\log k)$，所以根据记录更新小根堆的过程为 $O(N\log k)$。k 条记录排序的时间复杂度为 $O(k\log k)$。所以总的时间复杂度为 $O(N)+O(N\log k)+O(k\log k)$，即 $O(N\log k)$。具体过程请参看如下代码中的 printTopKAndRank 方法。

```
public void printTopKAndRank(String[] arr, int topK) {
        if (arr == null || topK < 1) {
                return;
        }
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        // 生成哈希表(字符串词频)
        for (int i = 0; i != arr.length; i++) {
                String cur = arr[i];
                if (!map.containsKey(cur)) {
                        map.put(cur, 1);
                } else {
                        map.put(cur, map.get(cur) + 1);
                }
        }
        Node[] heap = new Node[topK];
        int index = 0;
        // 遍历哈希表，决定每条信息是否进堆
        for (Entry<String, Integer> entry : map.entrySet()) {
                String str = entry.getKey();
                int times = entry.getValue();
                Node node = new Node(str, times);
                if (index != topK) {
                        heap[index] = node;
                        heapInsert(heap, index++);
                } else {
                        if (heap[0].times < node.times) {
                                heap[0] = node;
                                heapify(heap, 0, topK);
                        }
                }
        }
        // 把小根堆的所有元素按词频从大到小排序
        for (int i = index - 1; i != 0; i--) {
                swap(heap, 0, i);
                heapify(heap, 0, i);
        }
        // 严格按照排名打印 k 条记录
        for (int i = 0; i != heap.length; i++) {
                if (heap[i] == null) {
                        break;
                } else {
                        System.out.print("No." + (i + 1) + ": ");
                        System.out.print(heap[i].str + ", times: ");
```

```
                System.out.println(heap[i].times);
            }
        }
    }

    public void heapInsert(Node[] heap, int index) {
        while (index != 0) {
            int parent = (index - 1) / 2;
            if (heap[index].times < heap[parent].times) {
                swap(heap, parent, index);
                index = parent;
            } else {
                break;
            }
        }
    }

    public void heapify(Node[] heap, int index, int heapSize) {
        int left = index * 2 + 1;
        int right = index * 2 + 2;
        int smallest = index;
        while (left < heapSize) {
            if (heap[left].times < heap[index].times) {
                smallest = left;
            }
            if (right < heapSize && heap[right].times < heap[smallest].times) {
                smallest = right;
            }
            if (smallest != index) {
                swap(heap, smallest, index);
            } else {
                break;
            }
            index = smallest;
            left = index * 2 + 1;
            right = index * 2 + 2;
        }
    }

    public void swap(Node[] heap, int index1, int index2) {
        Node tmp = heap[index1];
        heap[index1] = heap[index2];
        heap[index2] = tmp;
    }
```

进阶问题。原问题是已经存在不再变化的字符串数组，所以可以一次性统计词频哈希表，然后建小根堆。可是进阶问题不一样，每个字符串词频可能会随时增加，这个过程一直是动态的。当然也可以在加入一个字符串时，在词频哈希表中增加这种字符串的词频，这样，add 方法的时间复杂度就是 $O(1)$。可是当有 printTopK 操作时，你只能像原问题一样，

根据所有字符串的词频表来建立小根堆，假设此时哈希表的记录数为 N，那么 printTopK 方法的时间复杂度就成了 $O(N\log k)$，但明显是不达标的。本书提供的解法依然是利用小根堆这个数据结构，但在设计上更复杂。下面介绍 TopKRecord 的结构设计。

TopKRecord 结构重要的 4 个部分如下：

- 依然有一个小根堆 heap。小根堆里装的依然是原问题中 Node 类的实例，每个实例表示一个字符串及其词频统计的信息。小根堆里装的都是加入过的所有字符串中词频最高的 Top K。heap 的大小在初始化时就确定，是 Node 类型的数组结构，数组的总大小为 k。
- 整型变量 index。表示如果新的 Node 类的实例想加入到 heap，该放在 heap 的哪个位置。
- 哈希表 strNodeMap。key 为字符串类型，表示加入的某种字符串。value 为 Node 类型。strNodeMap 上的每条信息表示一种字符串及其所对应的 Node 实例。
- 哈希表 nodeIndexMap，key 为 Node 类型，表示一种字符串及其词频信息。value 为整型，表示 key 这个 Node 类的实例对应到 heap 上的位置，如果不在 heap 上，为-1。

关于 strNodeMap 和 nodeIndexMap 的说明如下：

比如，"A"这个字符串加入了 10 次，那么在 strNodeMap 表中就会有类似这样的记录(key="A",value=("A",10))，value 是一个 Node 类的实例。如果"A"加入的次数很多，使"A"成为加入的所有字符中词频最高的 Top K 之一，那么"A"应该在堆上。假设"A"在堆上的位置为 5，那么在 nodeIndexMap 表中就会有类似这样的记录(key=("A",10),value=5)。如果"A"加入的次数不算多，没有使"A"成为加入的所有字符中词频最高的 Top K 之一，那么"A"不在堆上，则在 nodeIndexMap 表中就会有这样的记录(key=("A",10),value=-1)。strNodeMap 是字符串及其所对应的 Node 实例信息的哈希表，nodeIndexMap 是字符串的 Node 实例信息对应在堆中（heap）位置的哈希表。

以下为加入一个字符串时，TopKRecord 类中 add 方法所做的事情：

1．当加入一个字符串时，假设为 str。首先在 strNodeMap 中查询 str 之前出现的词频，如果查不到，说明 str 为第一次出现，在 strNodeMap 中加入一条记录(key=str,value=(str,1))。如果可以查到，说明 str 之前出现过，此时需要把 str 的词频增加，假设之前出现过 10 次，那么查到的记录为(key=str,value=(str,10))，变更为(key=str,value=(str,11))。

2．建立或调整完 str 的 Node 实例信息之后，需要考虑这个 Node 的实例信息是否已经在堆上，通过查询 nodeIndexMap 表可以得到 Node 的实例对应的堆上的位置，如果没有或

者查询结果为-1，表示不在堆上，否则表示在堆上，位置记为 pos。

1）如果在堆上，说明 str 词频没增加之前就是 Top K 之一，现在词频既然增加了，就需要考虑调整 str 对应的 Node 实例信息在堆中的位置，从 pos 位置开始向下调整小根堆即可(heapify)。特别注意：为了保证 nodeIndexMap 表中位置信息的始终准确，调整堆时，每一次两个堆元素（Node 实例）之间的位置交换都要更新在 nodeIndexMap 表中的位置。比如，在堆上的一个 Node 实例("A",10)原来在 2 位置，在 nodeIndexMap 表中的信息为(key=("A",10),value=2)。现在又加入了一个"A"，词频增加，信息当然要变成(key=("A",11),value=2)。然后从位置 2 调整堆时，发现这个实例需要和自己的一个孩子实例("B",10)交换，假设这个 Node 实例的位置是 6，即在 nodeIndexMap 表中记录为(key=("B",10),value=6)。那么在彼此交换位置之后，在 heap 数组中的两个实例当然很容易互换位置，但同时在 nodeIndexMap 上各自的信息也要变更，分别变更为(key=("A",11),value=6)，(key=("B",10),value=2)。也就是说，任何 Node 实例在堆中的位置调整都要改相应的 nodeIndexMap 表信息，这也是整个 TopKRecord 结构设计中最关键的逻辑。

2）如果不在堆中，则看当前的小根堆是否已满(index?=k)。如果没有满(index<k)，那么把 str 的 Node 实例放入堆底(heap 的 index 位置)，自然也要在 nodeIndexMap 表中加上位置信息。然后做堆在插入时的调整（heapInsert），同样，任何交换都要改 nodeIndexMap 表。如果已满(index==k)，则看 str 的词频是否大于小根堆堆顶的词频（heap[0]），如果不大于，则什么都不做。如果大于堆顶的词频，把 str 的 Node 实例设为新的堆顶，然后从位置 0 开始向下调整堆（heapify），同样，任何堆中位置的变更都要改 nodeIndexMap 表。

3．过程结束。

在加入新的字符串时，都可能会调整堆，而堆最大也仅是 k 的大小，所以 add 方法时间复杂度为 $O(\log K)$。随时更新的小根堆就是每时每刻的 Top K，打印时又没有排序的要求，所以 printTopK 方法直接依次打印小根堆数组即可，时间复杂度为 $O(K)$。

TopKRecord 类的全部实现请参看如下代码：

```
public class Node {
        public String str;
        public int times;

        public Node(String s, int t) {
                str = s;
                times = t;
        }
}

public class TopKRecord {
```

```
private Node[] heap;
private int index;
private HashMap<String, Node> strNodeMap;
private HashMap<Node, Integer> nodeIndexMap;

public TopKRecord(int size) {
        heap = new Node[size];
        index = 0;
        strNodeMap = new HashMap<String, Node>();
        nodeIndexMap = new HashMap<Node, Integer>();
}

public void add(String str) {
        Node curNode = null;
        int preIndex = -1;
        if (!strNodeMap.containsKey(str)) {
                curNode = new Node(str, 1);
                strNodeMap.put(str, curNode);
                nodeIndexMap.put(curNode, -1);
        } else {
                curNode = strNodeMap.get(str);
                curNode.times++;
                preIndex = nodeIndexMap.get(curNode);
        }
        if (preIndex == -1) {
                if (index == heap.length) {
                        if (heap[0].times < curNode.times) {
                                nodeIndexMap.put(heap[0], -1);
                                nodeIndexMap.put(curNode, 0);
                                heap[0] = curNode;
                                heapify(0, index);
                        }
                } else {
                        nodeIndexMap.put(curNode, index);
                        heap[index] = curNode;
                        heapInsert(index++);
                }
        } else {
                heapify(preIndex, index);
        }
}

public void printTopK() {
        System.out.println("TOP: ");
        for (int i = 0; i != heap.length; i++) {
                if (heap[i] == null) {
                        break;
                }
                System.out.print("Str: " + heap[i].str);
                System.out.println(" Times: " + heap[i].times);
        }
}
```

```
    private void heapInsert(int index) {
        while (index != 0) {
            int parent = (index - 1) / 2;
            if (heap[index].times < heap[parent].times) {
                swap(parent, index);
                index = parent;
            } else {
                break;
            }
        }
    }

    private void heapify(int index, int heapSize) {
        int l = index * 2 + 1;
        int r = index * 2 + 2;
        int smallest = index;
        while (l < heapSize) {
            if (heap[l].times < heap[index].times) {
                smallest = l;
            }
            if (r < heapSize && heap[r].times < heap[smallest].times) {
                smallest = r;
            }
            if (smallest != index) {
                swap(smallest, index);
            } else {
                break;
            }
            index = smallest;
            l = index * 2 + 1;
            r = index * 2 + 2;
        }
    }

    private void swap(int index1, int index2) {
        nodeIndexMap.put(heap[index1], index2);
        nodeIndexMap.put(heap[index2], index1);
        Node tmp = heap[index1];
        heap[index1] = heap[index2];
        heap[index2] = tmp;
    }

}
```

Manacher 算法

【题目】

给定一个字符串 str，返回 str 中最长回文子串的长度。