最大堆 + 最小堆：查找和最小的k对数字

困难/堆、最大堆，最小堆

学习目标

- 掌握堆、最大堆，最小堆的基本概念和性质
- 掌握最大堆，最小堆的运用

题目描述

给定两个升序的整型数组nums1和nums2，以及一个整数k

定义一对值（u,v）,其中第一个元素来自nums1，第二个元素来自nums2

请返回和最小的k对数字

输入: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
输出: [1,2],[1,4],[1,6]
解释: 返回序列中的前 3 对数:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

输入: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
输出: [1,1],[1,1]
解释: 返回序列中的前 2 对数:
[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

一. Comprehend 理解题意

目中给定的

两个数组

已经**升序**排好了

注意数组为空

以及k特别大

的**边界**情况

二. Choose 数据结构及算法思维选择

方案一：遍历组合（暴力解法）

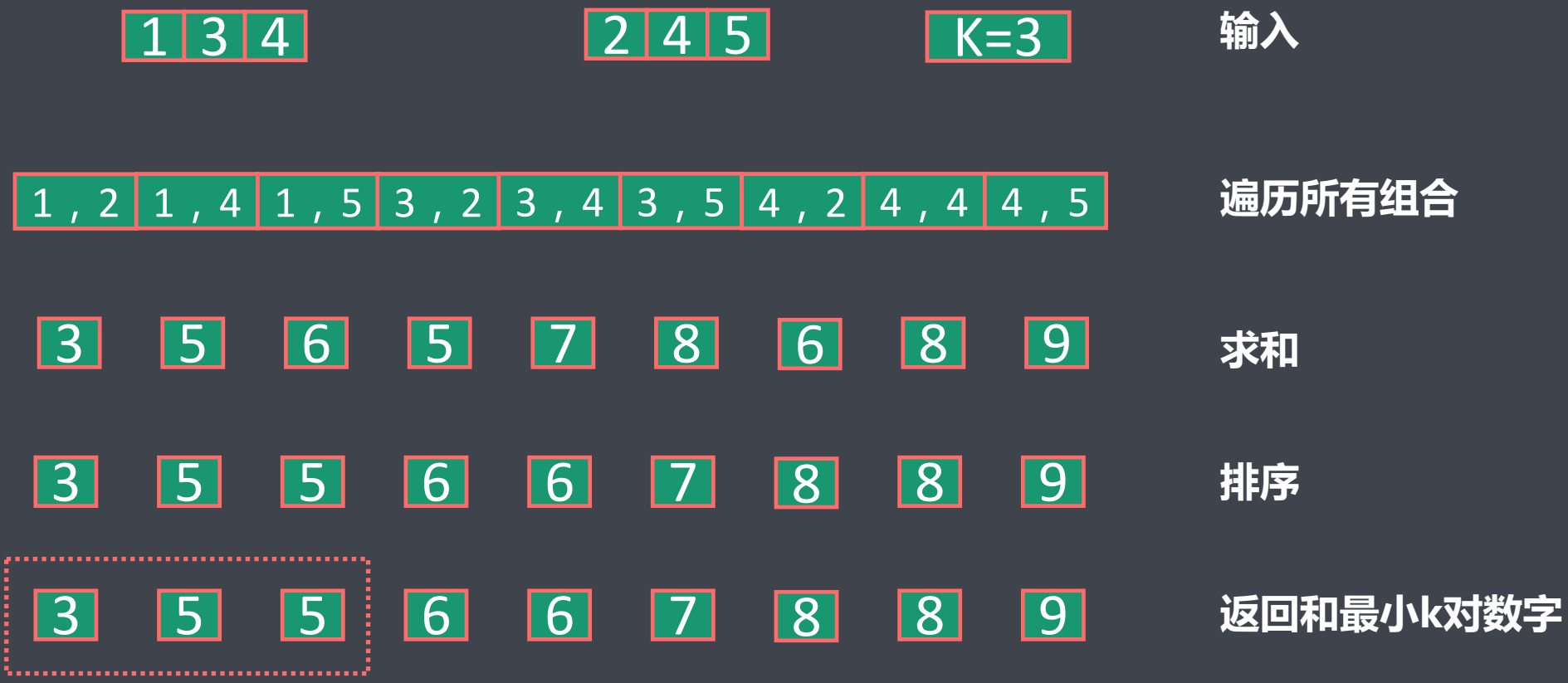
- 遍历所有数组的组合
- 数据结构：数组
- 算法思维：暴力搜索

方案二：最大堆（优化解法）

- 使用大顶堆存储和最小的k对数组
- 数组结构：大顶堆
- 算法思维：遍历

三. Code 基本解法及编码实现

解法一： 遍历组合



三. Code 基本解法及编码实现

解法一： 遍历组合边界和细节问题

边界问题

- 考虑数组为空和k特别大的情况

细节问题

- 如果用List来存储数组的和，那么在得到k个最小的数组和后如何找到对应的数组？
- 能否设计一个数据结构满足
  - 存储数组和
  - 按照和排序
  - 存储数组与和之间对应关系

三. Code 基本解法及编码实现

解法一： 遍历组合编码实现

```
class Solution {
    private class Pair{// 定义新的数据结构来存储解题所需信息
        int x,y;
        public Pair(int x,int y) {
            this.x=x;
            this.y=y;
        }
    }
    private class mySort implements Comparator<Pair>{
        @Override //重写排序方法，按照和的升序排列
        public int compare(Pair o1, Pair o2) {
            return o1.x+o1.y-o2.x-o2.y; // 自定义排序方法
        }
    }
```

三. Code 基本解法及编码实现

解法一： 遍历组合编码实现

```
class Solution {
    // 定义Pair和mySort  见上一页
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        if(nums1.length==0 || nums2.length==0) return new ArrayList<List<Integer>>();
        List<Integer> tmp=new ArrayList<>();
        List<List<Integer>> ans=new ArrayList<>();
        Pair[] pairs=new Pair[nums1.length*nums2.length];

        int len=0;
        for(int i=0;i<nums1.length;i++)
            for(int j=0;j<nums2.length;j++)
                pairs[len++]=new Pair(nums1[i],nums2[j]);

        Arrays.sort(pairs,new mySort());// 按照和升序排列
        // 取和最小的前k对
        for(int i=0;i<Math.min(pairs.length, k);i++) {// 注意如何处理k>pairs.length的情况
            tmp.add(pairs[i].x);tmp.add(pairs[i].y);
            ans.add(new ArrayList<>(tmp));
            tmp.clear();
        }
        return ans;
    }
}
```

执行结果： 通过 显示详情 >
执行用时： 116 ms，在所有 Java 提交中击败了 20.37% 的用户
内存消耗： 49.1 MB，在所有 Java 提交中击败了 23.31% 的用户

三. Code 基本解法及编码实现

解法一： 遍历组合复杂度分析

时间复杂度：O(Nlog(N))

- N=n1*n2
- n1、n2为两个数组的元素个数
- 遍历求和时候需要O(N)
- 排序平均需要O(Nlog(N))

空间复杂度：O(N)

- 需要存储所有组合的元素需要O(N)
- 如果使用快排，递归调用最差需要O(N)

三. Code 基本解法及编码实现

解法二： 最大堆

思考

- 解法一需要存储所有的数组组合
- 但是最终只需要返回前k个和最小的数组，后面的不需要存储
- 能否有一种数据结构
  - 存储k个元素
  - k个元素是所有元素里最小的

三. Code 基本解法及编码实现

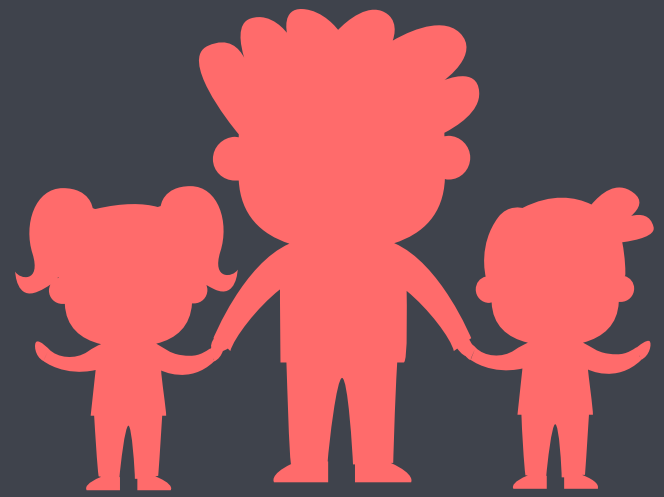
知识点：堆

堆树的定义如下：

（1）堆树是一颗完全二叉树；

（2）堆树中某个节点的值总是不大于或不小于其孩子节点的值；

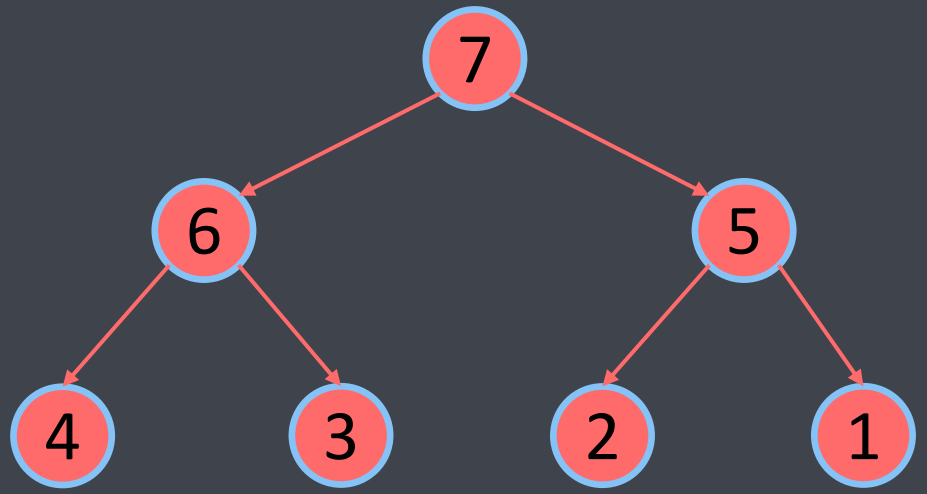
（3）堆树中每个节点的子树都是堆树。



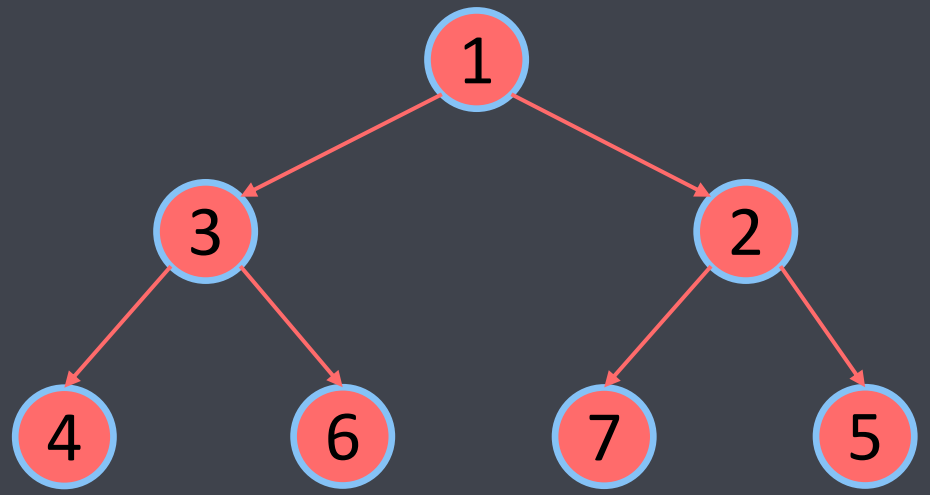
三. Code 基本解法及编码实现

知识点：堆

当父节点的键值总是大于或等于任何一个子节点的键值时为最大堆。 当父节点的键值总是小于或等于任何一个子节点的键值时为最小堆。如下图所示。



最大堆



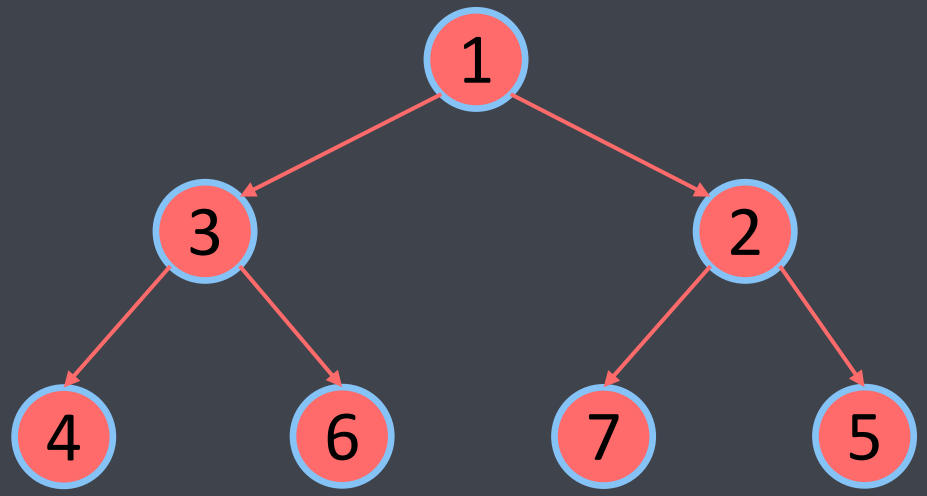
最小堆

三. Code 基本解法及编码实现

知识点：最小堆的建立

我们知道完全二叉树适合用数组来存储，不会出现空闲的位置，堆也是用数组来实现的，假设现在我们有以下数组：[6, 5, 4, 3, 2, 1, 7]

存入二叉树的格式为：



假设树的节点个数为n，

以1为下标开始编号，直到n结束。

对于节点i，其父节点为i/2；左孩子节点为i*2，

右孩子节点为i*2+1。

最后一个节点的下标为n，其父节点的下标为n/2。

三. Code 基本解法及编码实现

知识点：最小堆的建立

在构造堆的时候，首先需要找到最后一个节点的父节点，从这个节点开始构造最小堆，直到该节点前面所有分支节点都处理完毕

1、倒序遍历数列，因为下标在size/2 - 1之后的节点都是叶子结点，所以可以从size/2-1位置开始倒序遍历，减少比较次数。

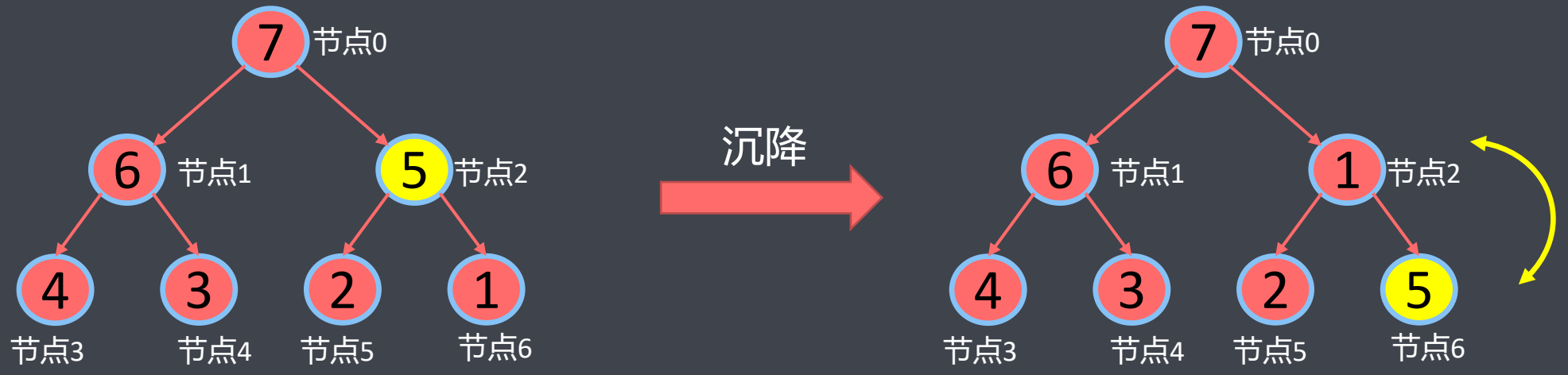
2、对二叉树中的元素挨个进行沉降处理，沉降过程为：把遍历到的节点与左右子节点中的最小值比对，如果比最小值要大，那么和孩子节点交换数据，反之则不作处理，继续倒序遍历。

3、 沉降后的节点，再次沉降，直到叶子节点。

三. Code 基本解法及编码实现

知识点：最小堆的建立

- 如对于下面的堆，首先从size/2 - 1 (节点2）开始进行沉降
- 节点2的子节点为节点5和节点6，最小值为1
- 由于节点2的值大于节点6的值，交换位置



三. Code 基本解法及编码实现

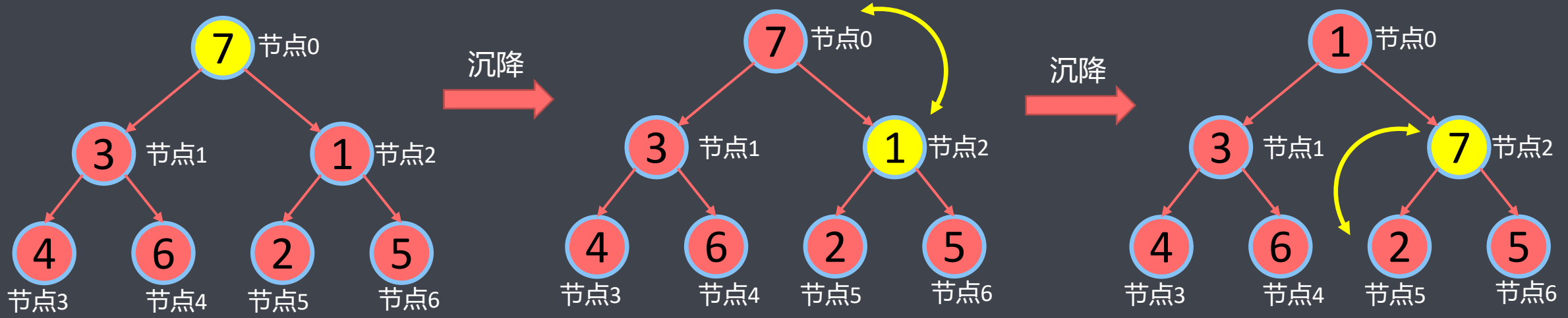
知识点：最小堆的建立

- 节点2沉降之后，后序遍历到节点1，对节点1进行沉降
- 节点1的子节点为节点3和节点4，最小值为3
- 由于节点1的值大于节点4的值，交换位置

三. Code 基本解法及编码实现

知识点：最小堆的建立

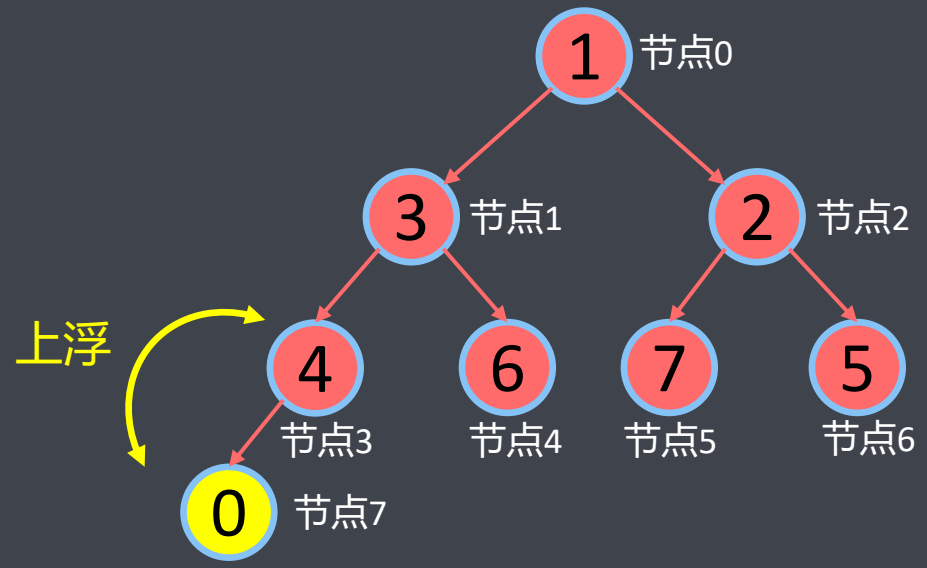
- 节点1沉降之后，后序遍历到节点0，对节点0进行沉降
- 节点0的子节点为节点1和节点2，最小值为1，交换节点0和节点2
- 继续沉降到叶子节点，最小堆建立完成



三. Code 基本解法及编码实现

知识点：最小堆的插入

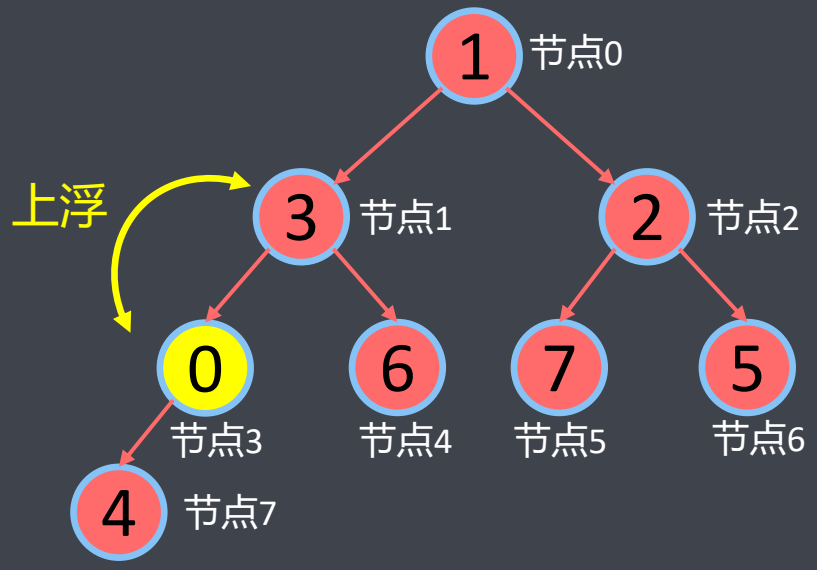
- 以上个最小堆为例，插入数字0。
- 数字0的节点首先加入到该二叉树最后的一个节点，依据最小堆的定义，自底向上，将数字0上浮，如果数字0小于其父节点的值，就交换位置，递归调整。



三. Code 基本解法及编码实现

知识点：最小堆的插入

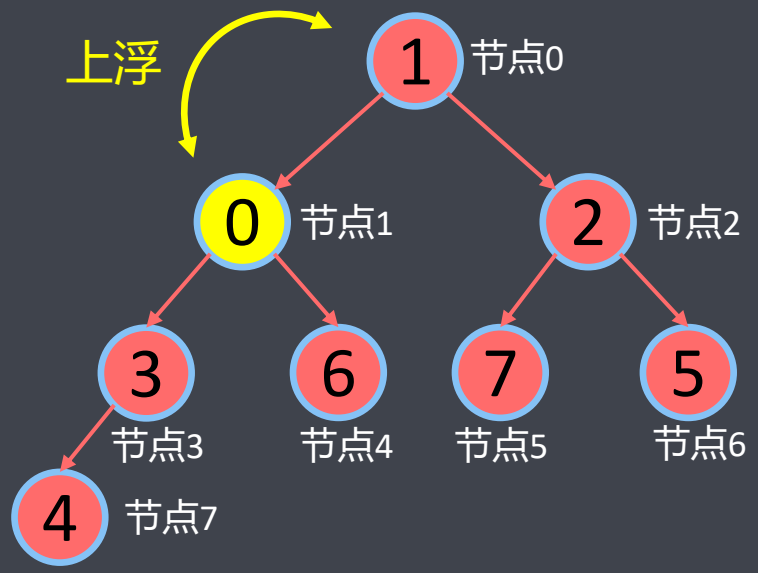
- 以上个最小堆为例，插入数字0。
- 数字0的节点首先加入到该二叉树最后的一个节点，依据最小堆的定义，自底向上，将数字0上浮，如果数字0小于其父节点的值，就交换位置，递归调整。



三. Code 基本解法及编码实现

知识点：最小堆的插入

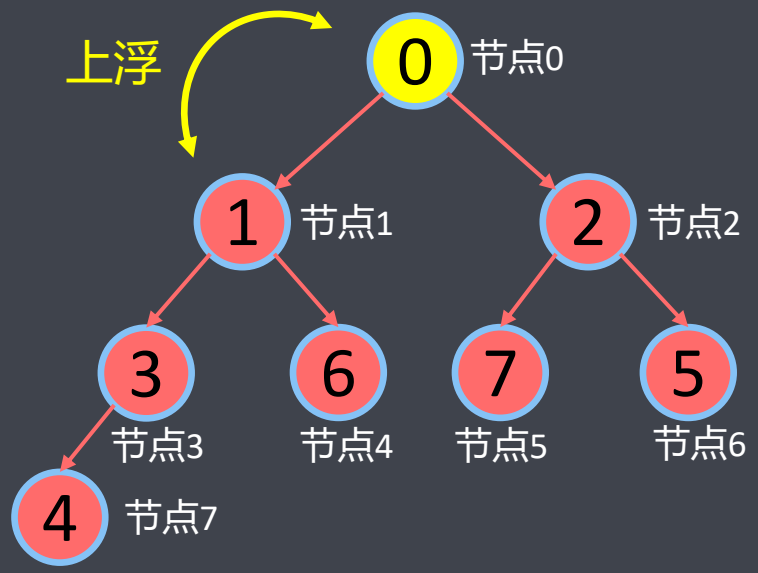
- 以上个最小堆为例，插入数字0。
- 数字0的节点首先加入到该二叉树最后的一个节点，依据最小堆的定义，自底向上，将数字0上浮，如果数字0小于其父节点的值，就交换位置，递归调整。



三. Code 基本解法及编码实现

知识点：最小堆的插入

- 以上个最小堆为例，插入数字0。
- 数字0的节点首先加入到该二叉树最后的一个节点，依据最小堆的定义，自底向上，将数字0上浮，如果数字0小于其父节点的值，就交换位置，递归调整。

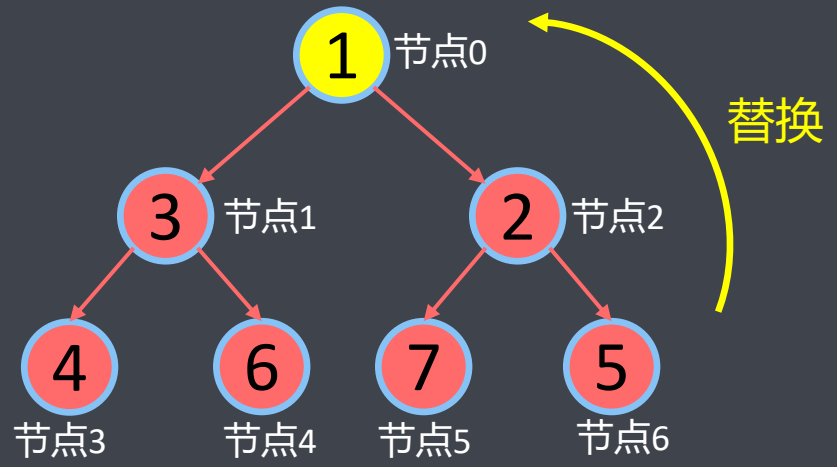


插入数字0后最小堆调整完成

三. Code 基本解法及编码实现

知识点：最小堆的删除

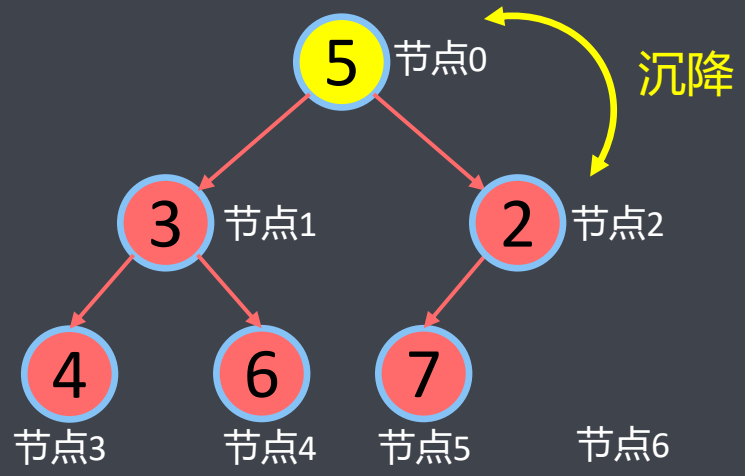
- 对于最小堆和最大堆而言，删除是针对于根节点而言。
- 对于删除操作，将二叉树的最后一个节点替换到根节点，然后自顶向下，递归调整，向下沉降。
- 以下面最小堆为例，删除堆顶数字1



三. Code 基本解法及编码实现

知识点：最小堆的删除

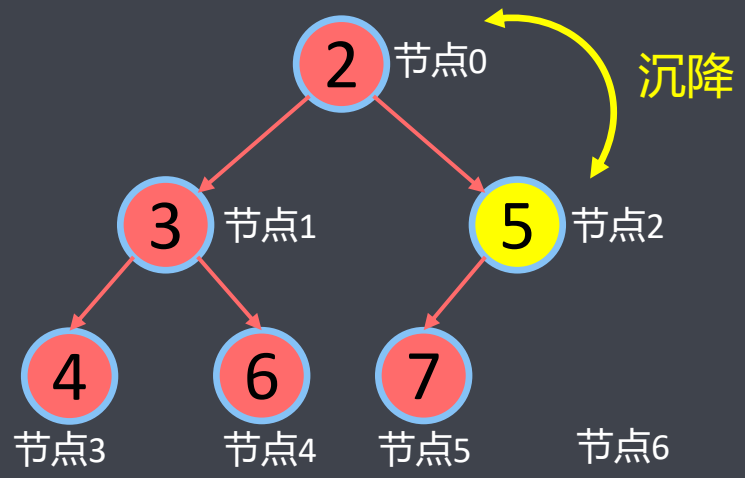
- 对于最小堆和最大堆而言，删除是针对于根节点而言。
- 对于删除操作，将二叉树的最后一个节点替换到根节点，然后自顶向下，递归调整，向下沉降。
- 以下面最小堆为例，删除堆顶数字1



三. Code 基本解法及编码实现

知识点：最小堆的删除

- 对于最小堆和最大堆而言，删除是针对于根节点而言。
- 对于删除操作，将二叉树的最后一个节点替换到根节点，然后自顶向下，递归调整，向下沉降。
- 以下面最小堆为例，删除堆顶数字1



删除堆顶后最小堆调整完成

三. Code 基本解法及编码实现

数据结构——PriorityQueue

- PriorityQueue是一个基于优先级的无界队列，Java中PriorityQueue通过二叉小顶堆实现，可以用一棵完全二叉树表示。
- Java中PriorityQueue默认是最小堆，可以通过传入Comparator实现最大堆
- 队列的每次插入、删除操作之后，优先队列会自动调整，维持一个最小堆(或最大堆)，一个长度为n的数组形成一颗完全二叉树深度为**logn，故下沉或上浮时间复杂度不会超过O(logn)**

三. Code 基本解法及编码实现

数据结构——PriorityQueue


```
//小顶堆，默认容量为11
PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

//大顶堆，容量11
PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11,new Comparator<Integer>(){
    @Override
    public int compare(Integer i1,Integer i2){
        return i2-i1;
    }
});
```

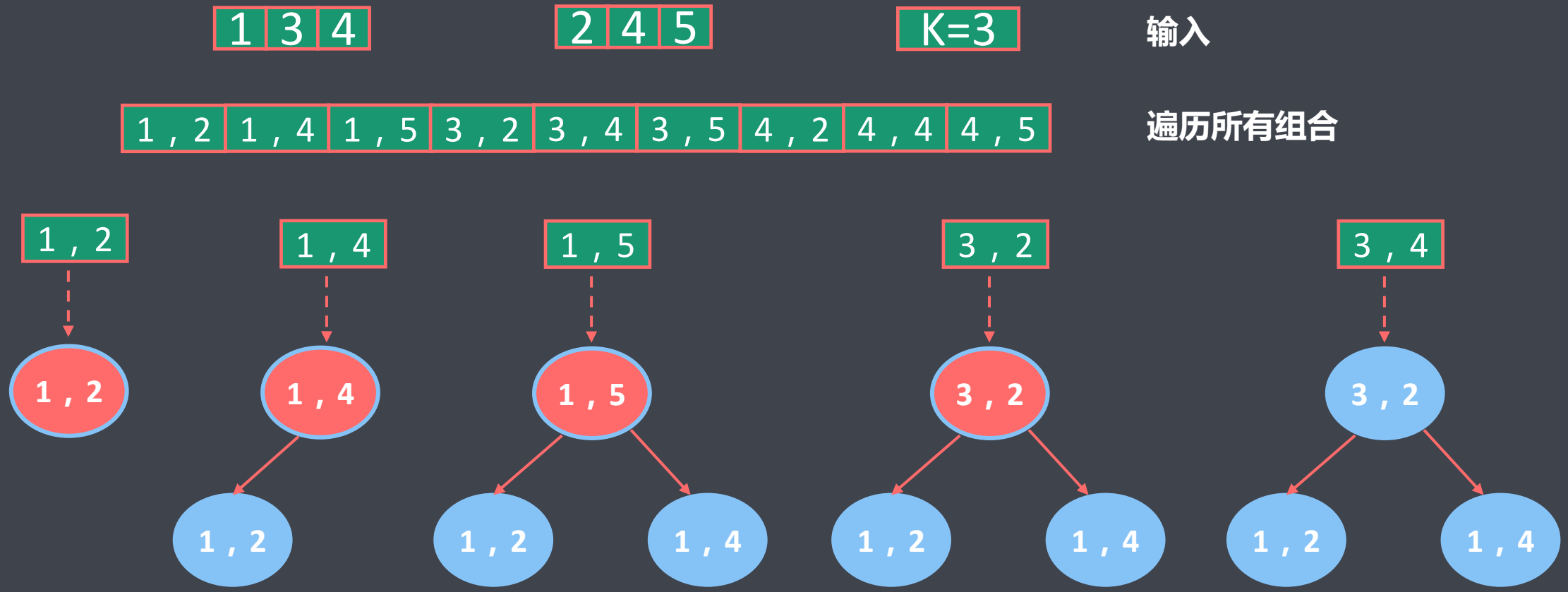
三. Code 基本解法及编码实现

实现最小堆的数据结构——PriorityQueue

- 插入方法(offer(),pull(),add())等方法时间复杂度为O(nlog(n))
- remove(Object) contains(Object) 方法时间复杂度为O(n)
- 检索方法peek element size 等方法时间复杂度为O(1)

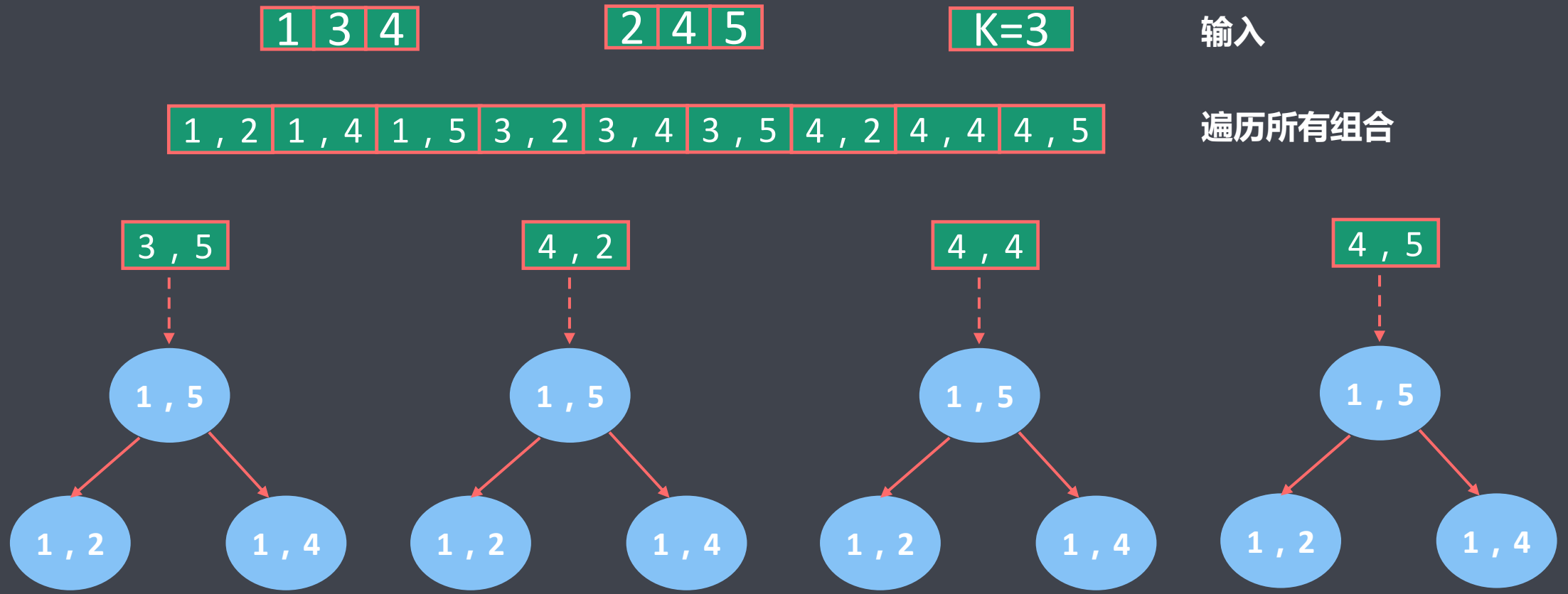
三. Code 基本解法及编码实现

解法二： 最大堆



三. Code 基本解法及编码实现

解法二： 最大堆



三. Code 基本解法及编码实现

解法二： 最大堆参考代码

```
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        PriorityQueue<List<Integer>> queue = new PriorityQueue<>(k, (o1, o2)->{
            return (o2.get(0) + o2.get(1)) - (o1.get(0) + o1.get(1));// 大顶堆，比较器使用lambda表达式，更简洁
        });
        for(int i = 0; i < Math.min(nums1.length, k); i++){
            for(int j = 0; j < Math.min(nums2.length, k); j++){
                // 如果最大堆未满或者当前数组和比堆顶的数组和小，加入最大堆
                if(queue.size() != k || nums1[i]+nums2[j] <= queue.peek().get(0) + queue.peek().get(1)){
                if(queue.size() == k)  queue.poll();
                List<Integer> pair = new ArrayList<>();
                pair.add(nums1[i]);
                pair.add(nums2[j]);
                queue.add(pair);
                }
            }
        }
        List<List<Integer>> res = new LinkedList<>();
        for(int i =0; i < k && !queue.isEmpty(); i++){
            res.add(0, queue.poll()); // 最后将元素弹出，倒序插入数组即可
        }
        return res;
    }
}
```

执行结果： 通过 显示详情 >
执行用时： 37 ms ，在所有 Java 提交中击败了 33.13% 的用户
内存消耗： 39.4 MB ，在所有 Java 提交中击败了 94.12% 的用户

三. Code 基本解法及编码实现

解法二：最大堆复杂度分析

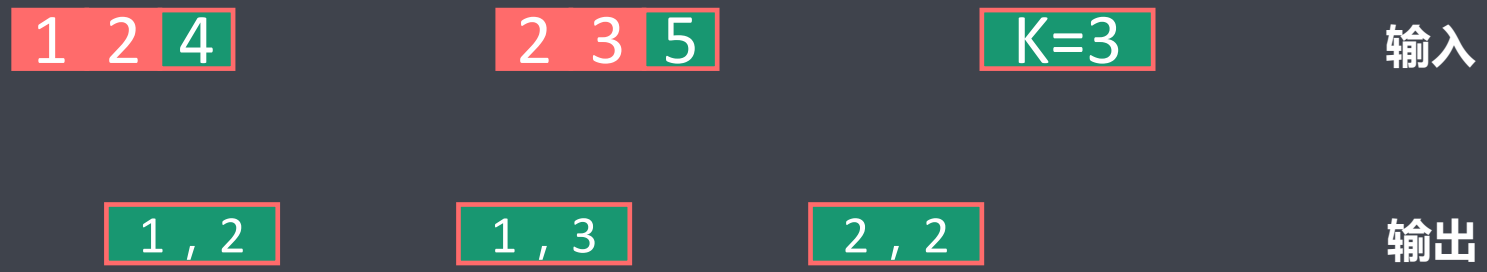
时间复杂度：O(N)

- N=n1*n2
- n1、n2为两个数组的元素个数

空间复杂度：O(k)

- k为所需结果数目
- 最大堆只需要存储k对元素即可

四. Consider 思考更优解



- 在前面的解法中，需要遍历所有的数组组合才能得到结果
- 在上面例子中，最终结果只需遍历每个数组前两个元素即可得到结果
- 是否存在一种解法只需遍历部分数组组合即可？

五. Code 最优解思路及编码实现

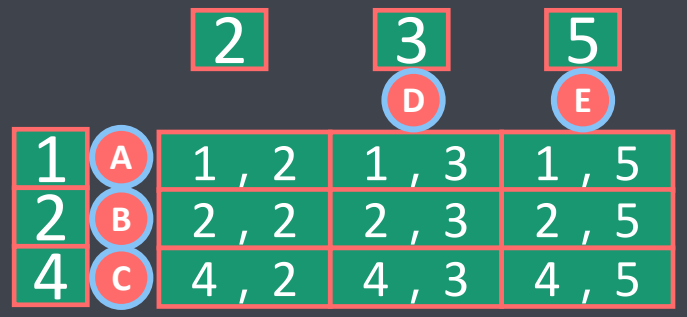
最优解： 最小堆

1	2	4

2	3	5

K=3

输入



- A所在的组合和最小
- 从A点开始A->B->C以及A->D->E分别升序
- 所有组合按照和升序排列后，A、B、C、D、E位于前面部分
- 但是B、C、D、E之间的位置不定

五. Code 最优解思路及编码实现

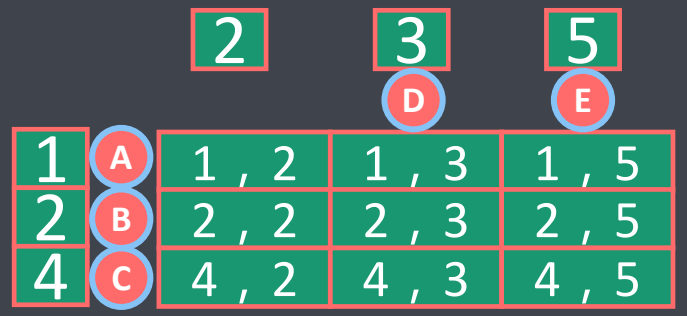
最优解： 最小堆

1	2	4

2	3	5

K=3

输入



- 如果我们可以对B、C、D、E排序后(本题中是A->B->D->C->E)，由于题目只需要前3个和最小的，直接取出A、B、D即可
- 无需再考虑剩下的组合，从而减少复杂度

五. Code 最优解思路及编码实现

最优解： 最小堆

1	2	4

2	3	5

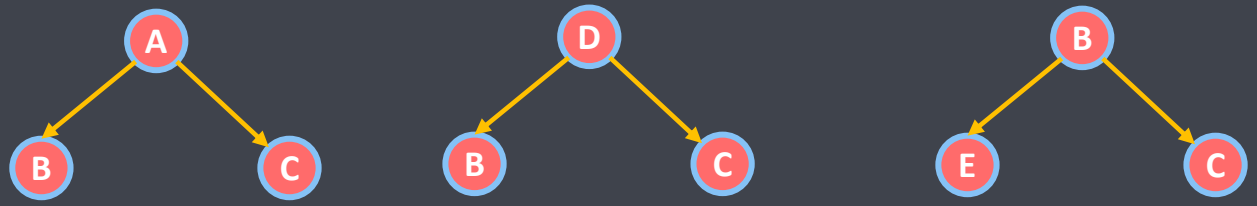
K=3

输入

		2	3	5
			D	E
1	A	1，2	1，3	1，5
2	B	2，2	2，3	2，5
4	C	4，2	4，3	4，5

算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



五. Code 最优解思路及编码实现

最优解： 最小堆

1	2	4

2	3	5

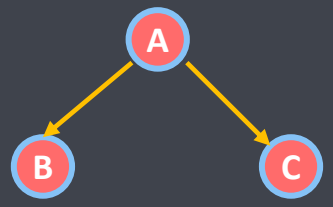
K=3

输入

		2	3	5
			D	E
1	A	1，2	1，3	1，5
2	B	2，2	2，3	2，5
4	C	4，2	4，3	4，5

算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



大小为3的最小堆

五. Code 最优解思路及编码实现

最优解： 最小堆

1	2	4

2	3	5

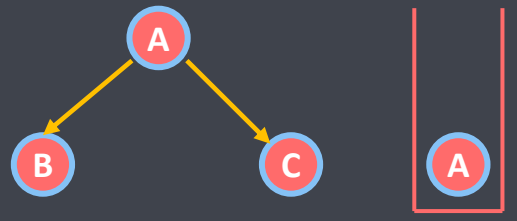
K=3

输入

		2	3 (D)	5 (E)
1	A	1，2	1，3	1，5
2	B	2，2	2，3	2，5
4	C	4，2	4，3	4，5

算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



K=1

弹出堆顶数组A

五. Code 最优解思路及编码实现

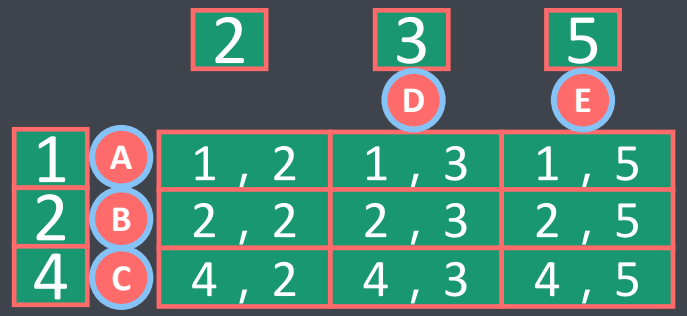
最优解： 最小堆

1	2	4

2	3	5

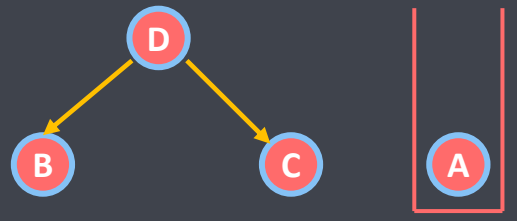
K=3

输入



算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



K=1

堆顶数组A第一个元素是来自nums1的 1，含有这个元素的下一个数组是D

五. Code 最优解思路及编码实现

最优解： 最小堆

1	2	4

2	3	5

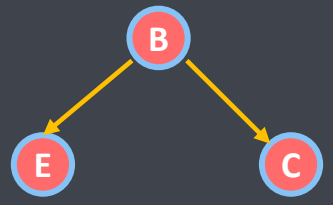
K=3

输入

		2	3	5
			D	E
1	A	1，2	1，3	1，5
2	B	2，2	2，3	2，5
4	C	4，2	4，3	4，5

算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



K=2

五. Code 最优解思路及编码实现

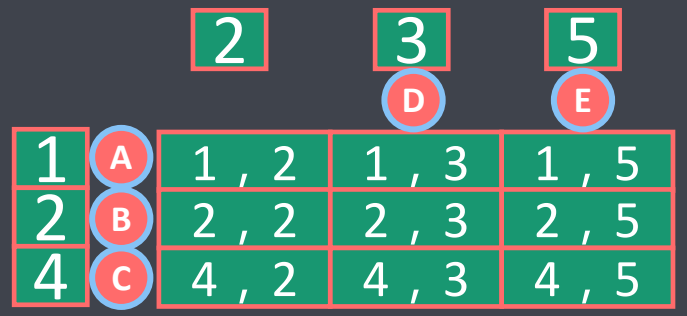
最优解： 最小堆

1	2	4

2	3	5

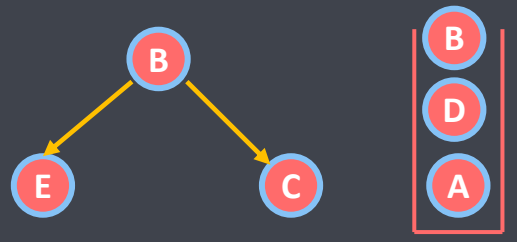
K=3

输入



算法流程

1. 初始化大小为k的最小堆，将含nums1第一个元素的前k个组合加入最小堆
2. 重复k次
   - 取出堆顶数组，加入结果队列
   - 如果还存在堆顶数组第一个元素的组合，则将下一个组合加入最小堆



K=3

五. Code 最优解思路及编码实现

最优解：最小堆编码实现

```
class Solution {
    public static List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        PriorityQueue<int[]> queue = new PriorityQueue<>(
                    (o1, o2) -> (nums1[o1[0]] + nums2[o1[1]]) - (nums1[o2[0]] + nums2[o2[1]]));
        List<List<Integer>> res = new LinkedList<>();

        if (nums1.length == 0 || nums2.length == 0) return res;// 两个数组有一个为空，返回空

        // 维护一个大小为k的小顶堆，将某个升序序列加入
        for (int i = 0; i < Math.min(nums1.length, k); i++) {
            queue.add(new int[]{i, 0}); // 注意加入的是坐标，这样有利于得到堆顶元素的下一个元素
        }

        while (k > 0 && !queue.isEmpty()) {
            int[] pair = queue.poll(); // 弹出堆顶元素
            List<Integer> item = new ArrayList<>();
            item.add(nums1[pair[0]]);
            item.add(nums2[pair[1]]);

            if (pair[1] < nums2.length - 1) {
                queue.add(new int[]{pair[0], pair[1] + 1});// 加入堆顶元素的下一个元素
            }
            res.add(item);
            k--;
        }
        return res;
    }
}
```

六. Change 变形延伸

题目变形

- （练习）最小的k个数(剑指offer 40)

延伸扩展

- 最大堆最小堆的最大特点是有序的存储k个元素

本题来源

- Leetcode 373 https://leetcode-cn.com/problems/find-k-pairs-with-smallest-sums/solution/javada-ding-dui-xiao-ding-dui-jie-fa-yi-dong-by-vi/

总结

- 掌握堆、最大堆、最小堆的基本概念和性质
- 掌握最大堆、最小堆的运用

课后练习

1. 数据流中的第k大元素(Leetcode703/简单)
2. 数组中的第k个最大元素（ Leetcode 215 /中等 ）
3. 有序矩阵中第k小的元素（ Leetcode 378 /中等 ）
4. 合并k个升序链表（ Leetcode 23 /困难 ）

课后练习

1. 数据流中的第k大元素(Leetcode703/简单)

   提示：设计一个找到数据流中第k大元素的类。注意是排序后第k大的元素，不是第k个不同的元素。

   你的KthLargest类需要同时接收整数k和整数数组nums 的构造器，它包含数据流中的初始元素。

   每次调用 KthLargest.add，返回当前数据流中第K大的元素。

```
int k = 3;
int[] arr = [4,5,8,2];
KthLargest kthLargest = new KthLargest(3, arr);
kthLargest.add(3);   // returns 4
kthLargest.add(5);   // returns 5
kthLargest.add(10);  // returns 5
kthLargest.add(9);   // returns 8
kthLargest.add(4);   // returns 8
```

课后练习

2. 数组中的第k个最大元素（Leetcode 215 /中等）

提示：在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

示例 2:

```
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

课后练习

3. 有序矩阵中第k小的元素（Leetcode 378 /中等）

提示：给定一个 n x n 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。

请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

课后练习

4. 合并k个升序链表（ Leetcode 23 /困难 ）

提示：给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容