Heap

Task management

How to manage the processes running in the operating system.

Queue

- Short task may have to wait a long time.
- High priority task cannot get prioritized.

ArrayList

Have to sort every time a new task arrives.

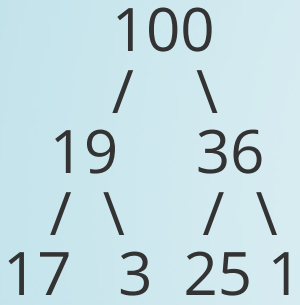
Heap (min, max)

Very fast for getting the lowest/highest priority element.

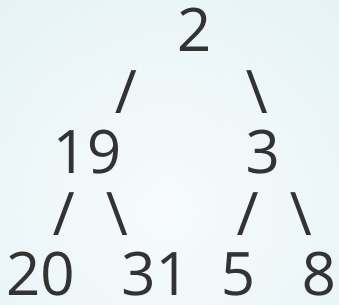
Heap (min, max)

- Specialized tree-based data structure.
- If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap
  - min-heap: root is always smaller than children.
  - max-heap: root is always larger than children.
- A common implementation would be binary heap, which is based on complete binary tree.

Heap (min, max)



max-heap



min-heap

Heap (min, max)

- Min-Heap
  - Root is the smallest node of the whole tree.
- Max-Heap
  - Root is the largest node of the whole tree.
- Complete Binary Tree
  - So if you construct a heap by yourself, then you only need an Array

Basic Operations

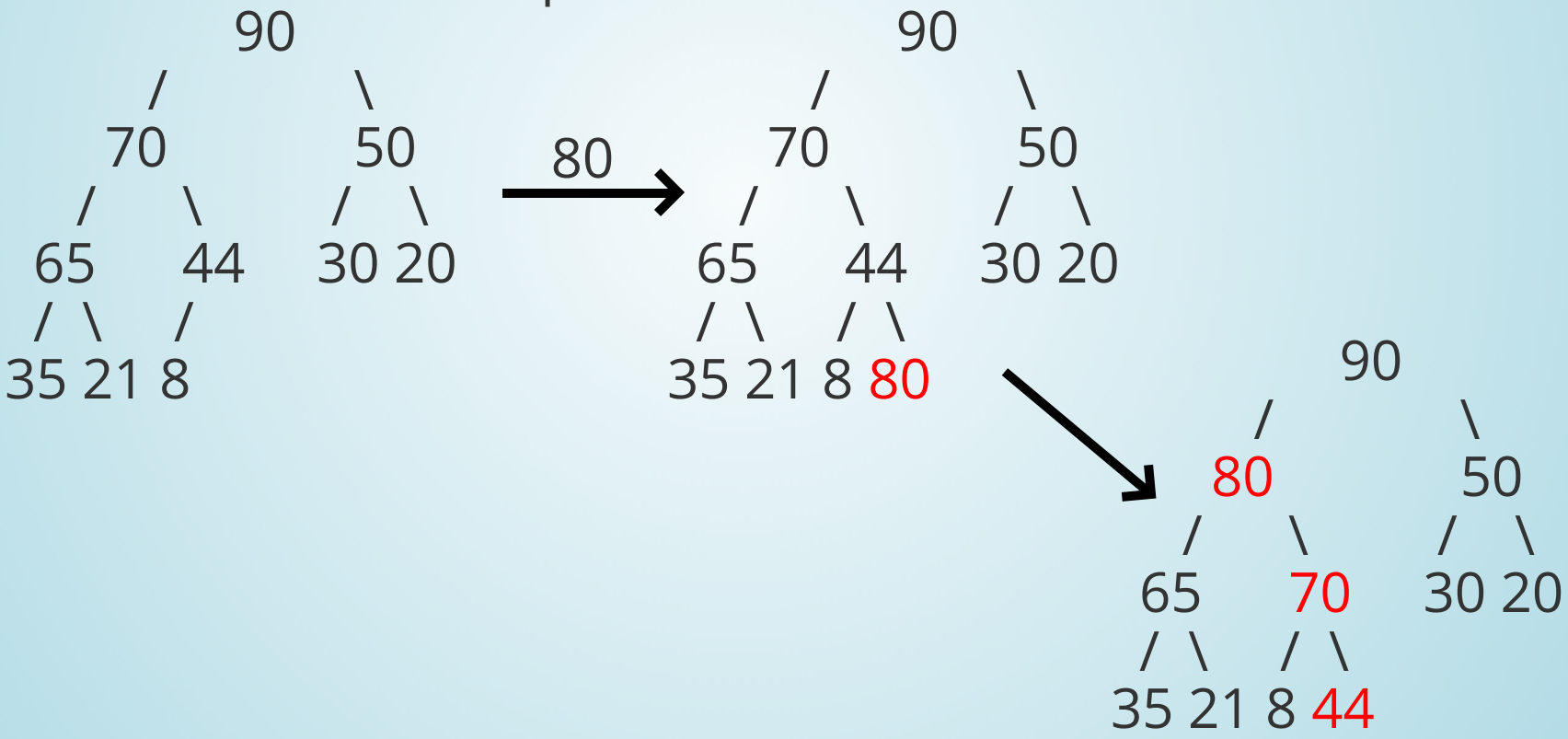
- insert, (offer)
- delete
  - delete root, (poll)
- initialize

Basic Operations

- insert
  - Insert the element to the end every time
  - Then do the bubble up
  - At most bubble up to the root, operation times would be the depth, log(N)
  - O(logN)

Basic Operations

- insert -> bubble up

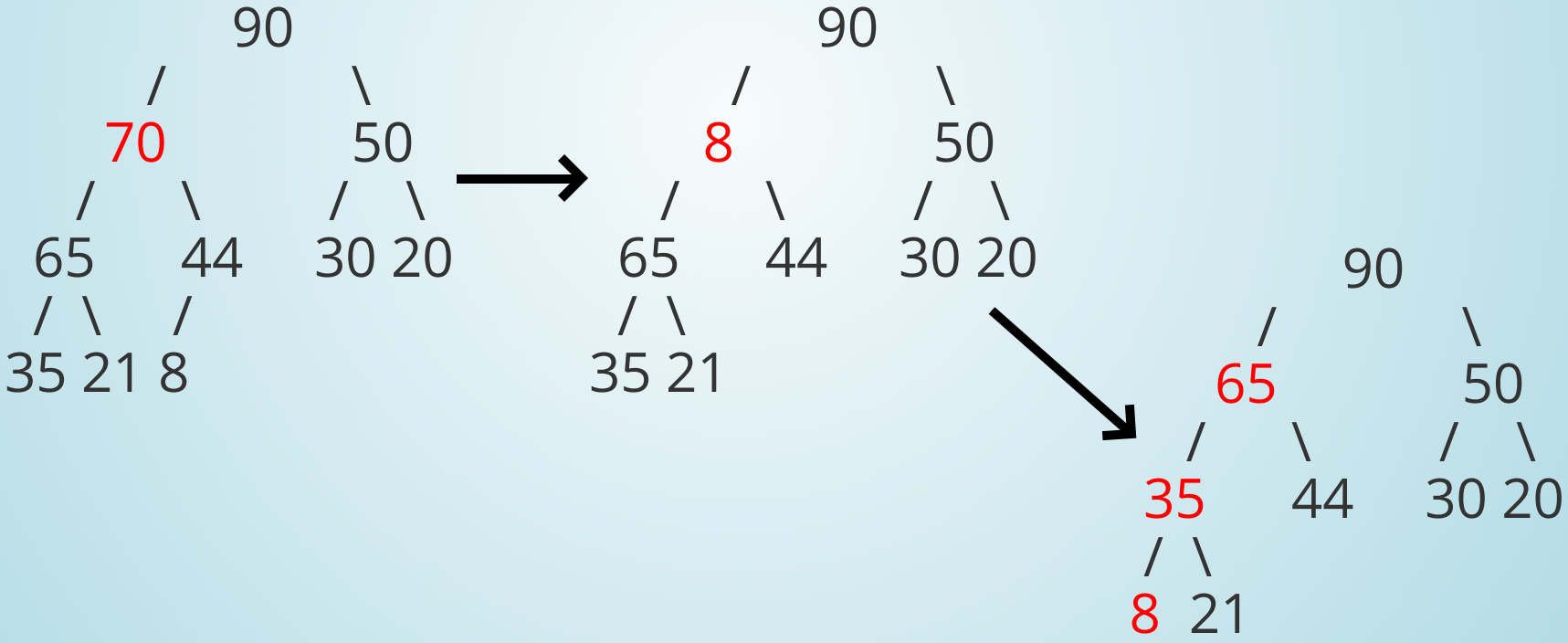


Basic Operations

- insert
- delete (pop, poll)
  - After swapping, at most sift down to the leaf, operation time would be the depth, log(N).
  - O(logN).
  - delete root, O(logN)
    - O(1), get the largest/smallest element.
    - O(logN), to moderate.

Basic Operations

- insert
- delete



Basic Operations

- insert
- delete
- initialize

Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).

Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9

Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9

  26

Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9

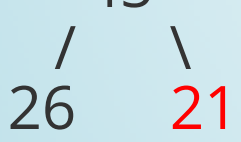
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9

45
/
26

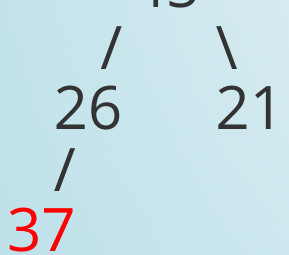
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



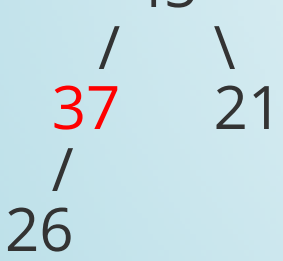
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



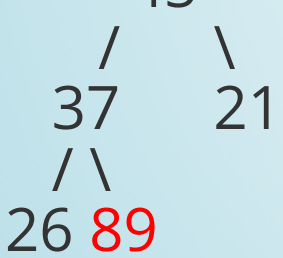
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



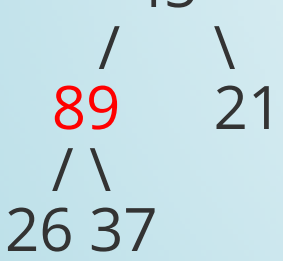
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



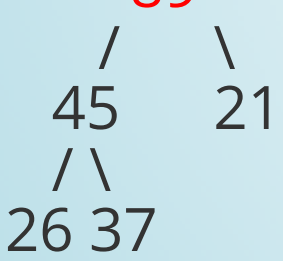
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



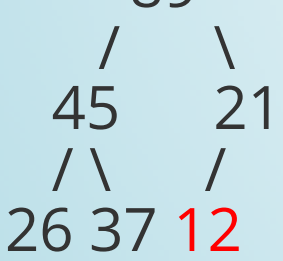
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



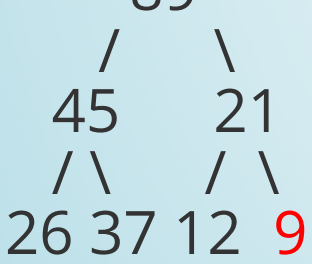
Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9



Basic Operations

- insert
- delete
- initialize
  - Insert elements one by one, O(NlogN).
  - 26, 45, 21, 37, 89, 12, 9

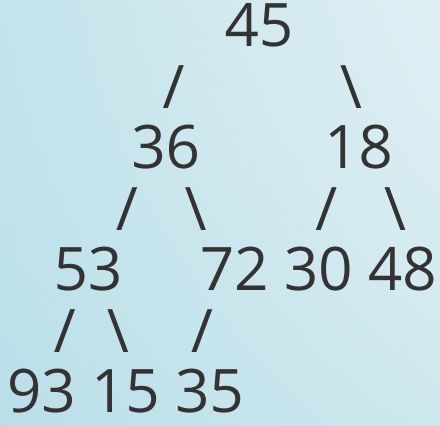


Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)

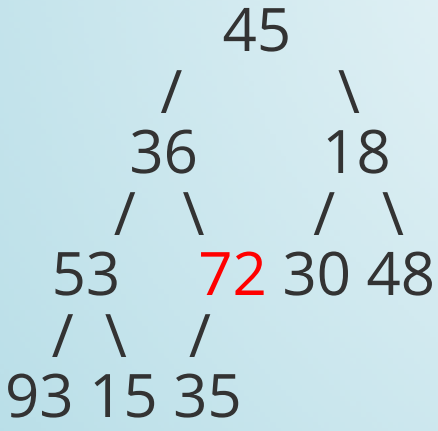
Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35



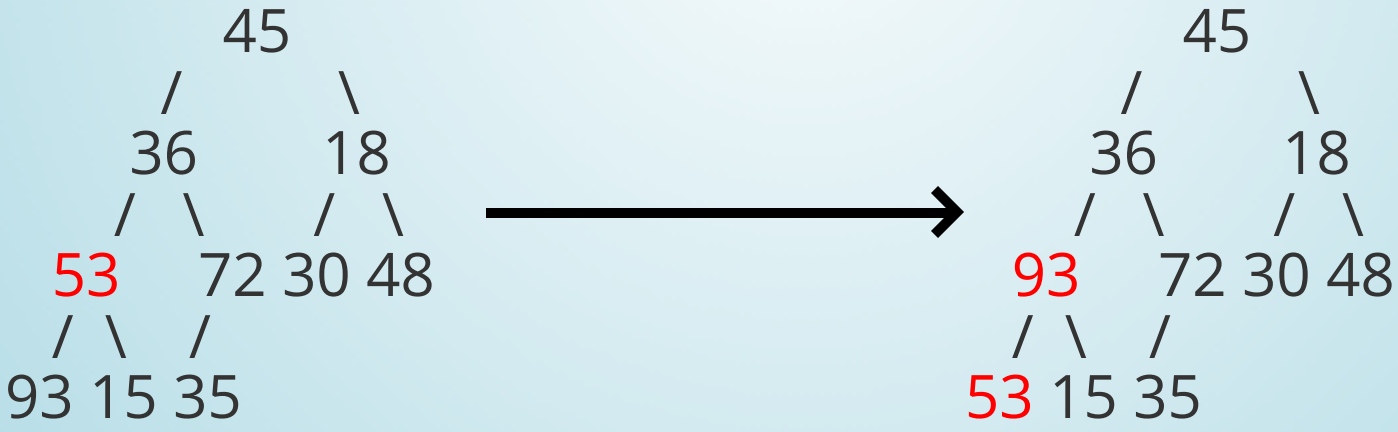
Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35



Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35

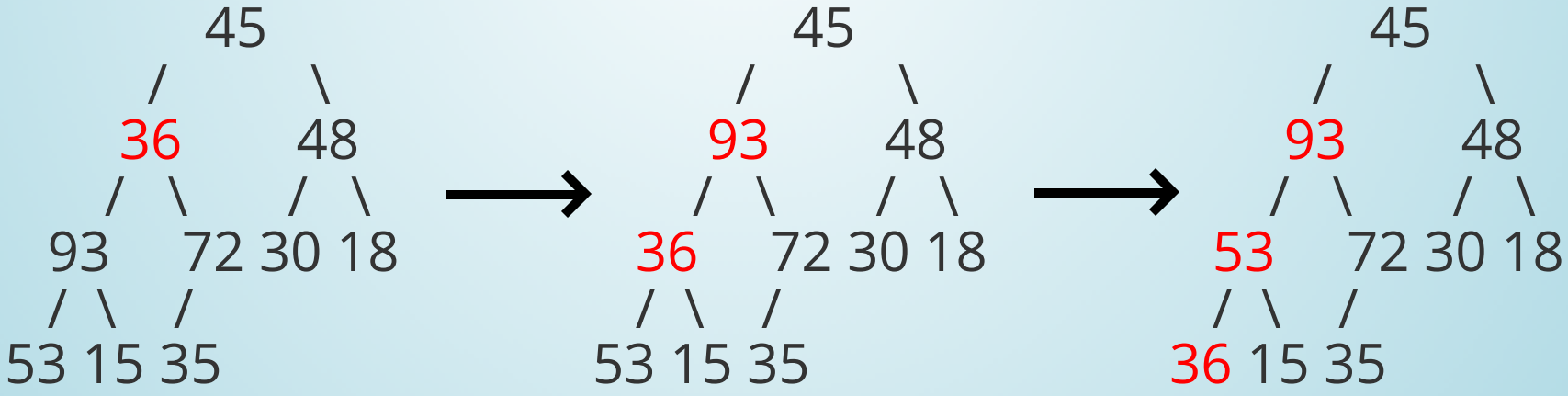


Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35

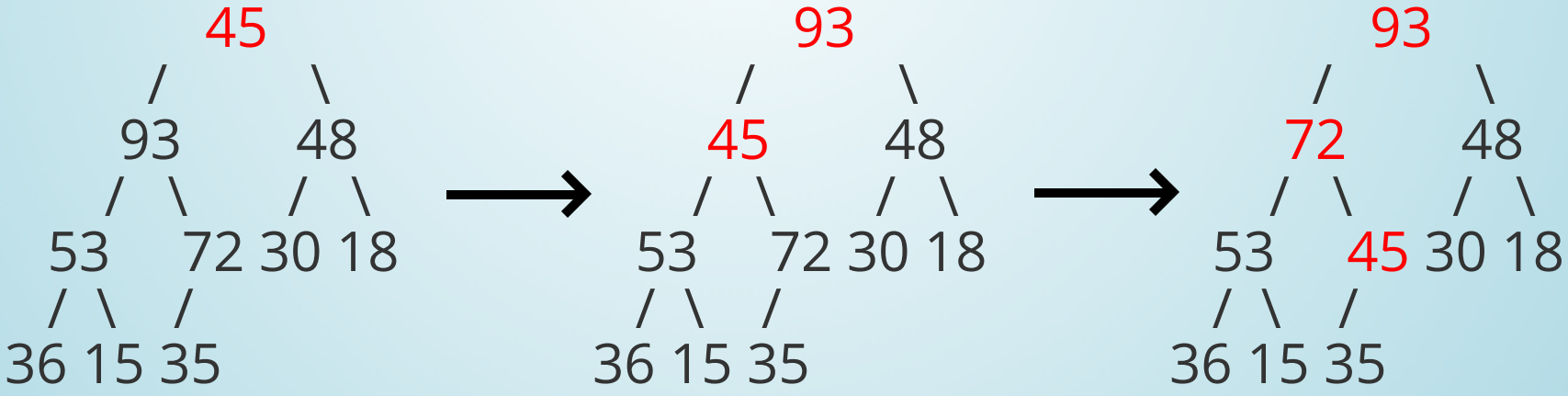
Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35



Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - 45, 36, 18, 53, 72, 30, 48, 93, 15, 35



Basic Operations

- insert
- delete
- initialize
  - sift down level by level, O(N)
  - Last level, do not need to move
  - second last level, at most sift down once

$$T = \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + ... + 1 \times (\log n - 1)$$

$$2T = \frac{n}{2} \times 1 + \frac{n}{4} \times 2 + ... + 2 \times (\log n - 1)$$

$$T = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... = O(n)$$

Basic Operations

- insert, O(logN)
- delete, O(logN)
- initialize, O(N)
  - Insert elements one by one, O(NlogN).
  - sift down level by level, O(N)

Heap in Java - PriorityQueue

- Operations
  - E peek();
  - E poll();
  - boolean offer(E element);
- Constructor
  - PriorityQueue(int capacity)
  - How does Java know how to compare elements in the heap.

Comparator v.s. Comparable

- Comparator
  - Helps other objects to compare
- Comparable
  - If some class is comparable, that means its instances know how to compare to other instances.

Comparator v.s. Comparable

```
class MyClass implements Comparable<MyClass> {
   @override
   public int compareTo(final MyClass o) {
       return this.val - o.val; // increasing (minHeap)
   }
}
PriorityQueue<> heap = new PriorityQueue<>(capacity);
```

Comparator v.s. Comparable

```
class MyComparator implements Comparator<MyClass> {
   public int compare(MyClass a, MyClass b) {
      return a.val - b.val; // increasing (minHeap)
   }
}
MyComparator myComparator = new MyComparator();
PriorityQueue<MyClass> heap = new PriorityQueue<MyClass>
(capacity, myComparator);
```

Comparator v.s. Comparable

```
PriorityQueue<MyClass> heap = new PriorityQueue<MyClass>
(cap, new Comparator<MyClass>() {
    public int compare(MyClass a, MyClass b) {
        return a.val - b.val;  // increasing (minHeap)
    }
});
```

Add Elements into Heap

- Default: min-heap (PriorityQueue)
- If need max-heap
  - Comparator & Comparable
  - add (-num) into Heap

Heap in C++ - priority_queue

- Operations
  - top()
  - pop()
  - push()
  - size()
- Similar as Java. Instead of using Comparator or Comparable, you need to overload the operator : >, <

Heap Summary

- insert, O(logN)
- delete, O(logN)
- initialize, O(N)
- Good at maintaining a dynamic data stream, from which largest/smallest is always needed, while there is no need for sorting.

Merge Sorted Array/List

- Two sorted array/list
  - Two pointer, move the smaller one backward.
- K sorted array/list
  - Impossible to manually maintain k pointers.
  - Data structure to maintain k pointers and capable of returning the largest/smallest element with high efficiency.
  - Heap

Merge k Sorted List

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

public ListNode merge(ListNode[] lists);

Merge k Sorted List

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Insert all head nodes into minHeap.

while (minHeap is not empty)

    root = minHeap.pop();

    Add root into result

    Insert root.next into minHeap.

return result

Merge k Sorted List

```
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0)
        return null;
    Comparator<ListNode> comparator = new Comparator<ListNode> () {
        public int compare(ListNode node1, ListNode node2) {
            return node1.val - node2.val;
        }
    };
    PriorityQueue<ListNode> minHeap =
        new PriorityQueue<ListNode>(lists.length, comparator);

    for (int i = 0; i < lists.length; i++) {
        if (lists[i] != null) {
            minHeap.add(lists[i]);
        }
    }

    ListNode dummy = new ListNode(-1);
    ListNode cur = dummy;
    while (!minHeap.isEmpty()) {
        cur.next = minHeap.poll();
        cur = cur.next;
        if (cur.next != null)
            minHeap.add(cur.next);
    }
    return dummy.next;
}
```

Merge k Sorted List

This can be used in external merge sort

But usually people will use tournament tree instead of a heap to do the external merge sort

Find Maximum Number

- O(N) to find maximum
- What if we need to update the list very often, that said, when the maximum is removed, a new number will come instantly.
  - Need to compare n times again. O(N)
  - Sort? Need to sort every time.
    - O(NlogN) for the first time and O(N) every time.
    - Inserting number is very time consuming.
  - Heap.
    - Good at dealing with data stream.

Sliding Window Maximum

Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

For example,

Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.

Return [3,3,5,5,6,7].

Note:

You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Sliding Window Maximum

```
public int[] maxSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return nums;
    }
    int[] result = new int[nums.length - k + 1];
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(k,
        new Comparator<Integer>() {
            public int compare(Integer a, Integer b) {
                return b - a;
            }
        });
    for (int i = 0; i < nums.length; i++) {
        if (i >= k) {
            maxHeap.remove(nums[i-k]);
        }
        maxHeap.offer(nums[i]);
        if (i >= k - 1) {
            result[i - k + 1] = maxHeap.peek();
        }
    }
    return result;
}
```

Sliding Window Maximum

```
public int[] maxSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return nums;
    }
    int[] result = new int[nums.length - k + 1];
    PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);
    for (int i = 0; i < nums.length; i++) {
        if (i >= k) {
            minHeap.remove(-nums[i-k]);
        }
        minHeap.offer(-nums[i]);
        if (i >= k - 1) {
            result[i - k + 1] = -minHeap.peek();
        }
    }
    return result;
}
```

Sliding Window Maximum

A better way: Use deque to get O(n) time complexity

```
public int[] maxSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return nums;
    }
    Deque<Integer> deque = new LinkedList<>();
    int[] result = new int[nums.length - k + 1];
    for (int i = 0; i < nums.length; i++) {
        if (!deque.isEmpty() && deque.peekFirst() == i - k) {
            deque.poll();
        }
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
            deque.pollLast();
        }
        deque.offer(i);
        if (i >= k - 1) {
            result[i - k + 1] = nums[deque.peekFirst()];
        }
    }
    return result;
}
```

Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,

Given [3,2,1,5,6,4] and k = 2, return 5.

Note:

You may assume k is always valid, 1 ≤ k ≤ array's length.

Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

- Find largest element for k times. O(kn)
- Sort. O(nlogn)
- Min-Heap
  - Build a min heap with capacity of k.
  - Insert all numbers into the heap.
    - If the heap is full, then only insert when number is larger than the root.
  - Root of the heap will be the kth largest element.

Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

```
public int findKthLargest(int[] nums, int k) {
    Comparator<Integer> comparator = new Comparator<Integer>() {
        public int compare(Integer a, Integer b) {
            return a - b;
        }
    };

    PriorityQueue<Integer> minHeap = new PriorityQueue(k, comparator);
    for (int i = 0; i < k; i++) {
        minHeap.add(nums[i]);
    }
    for (int i = k; i < nums.length; i++) {
        if (nums[i] >= minHeap.peek()) {
            minHeap.poll();
            minHeap.add(nums[i]);
        }
    }
    return minHeap.poll();
}
```

Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

- Find largest element for k times. O(kn)
- Sort. O(nlogn)
- Min-Heap. O(nlogk)
- Recursion. O(n)
  - Randomly select one number, find its position in array. O(n)
  - If pos > k, then recursion in left half.
  - Otherwise, recursion in right half.
  - Time: n + n/2 + n/4 + ... = 2n = O(n)

Find Median in Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3], the median is (2 + 3) / 2 = 2.5

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

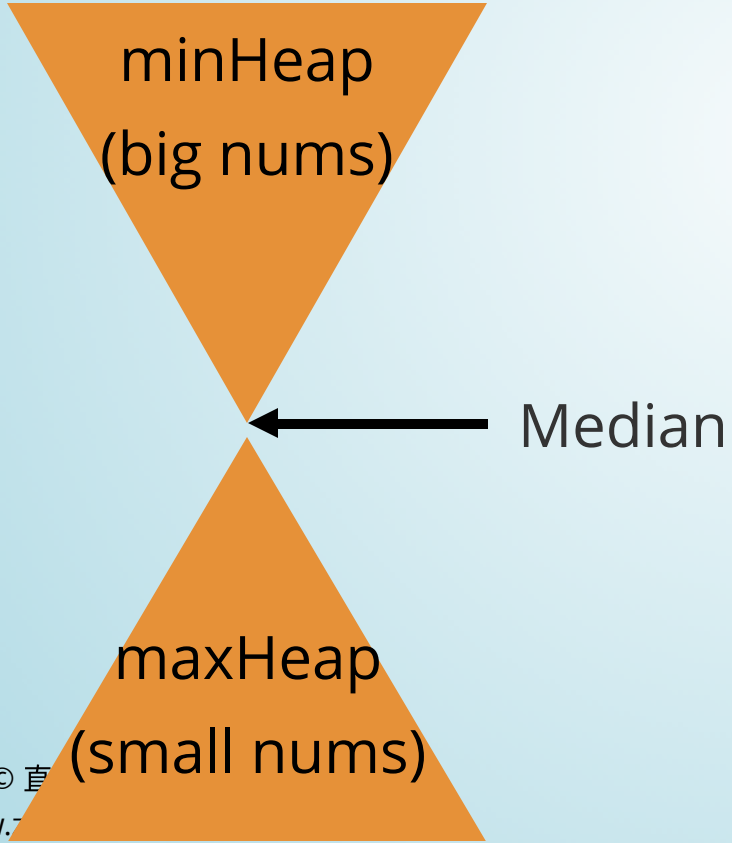
Find Median in Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

```
class MedianFinder {
  public void addNum(int num);
  public double findMedian();
}
```

Find Median in Data Stream

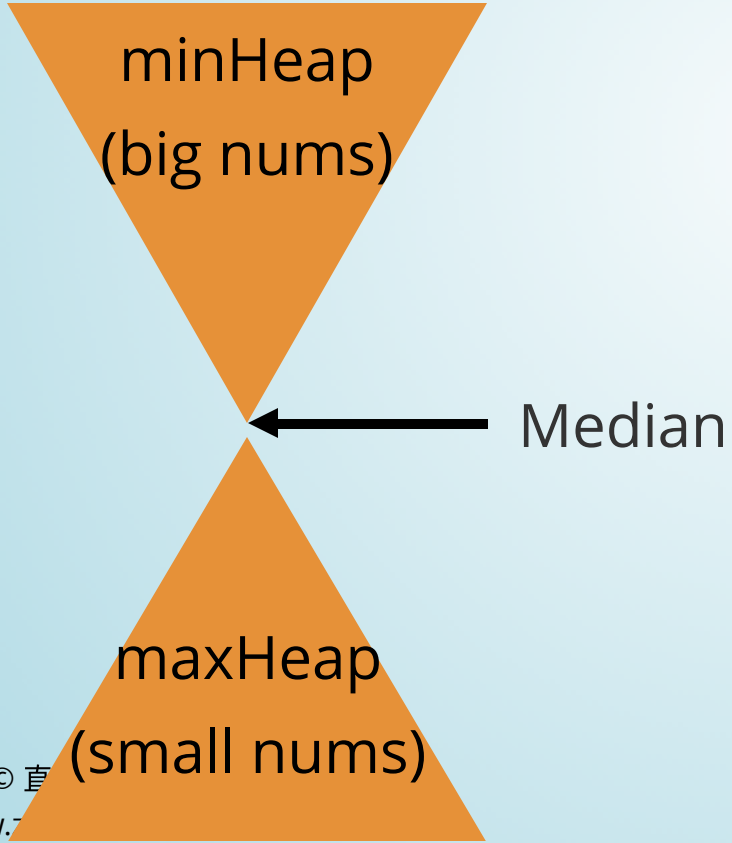
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- size of minHeap and maxHeap should be the same (diff <= 1).
- minHeap.peek > maxHeap.peek

Find Median in Data Stream

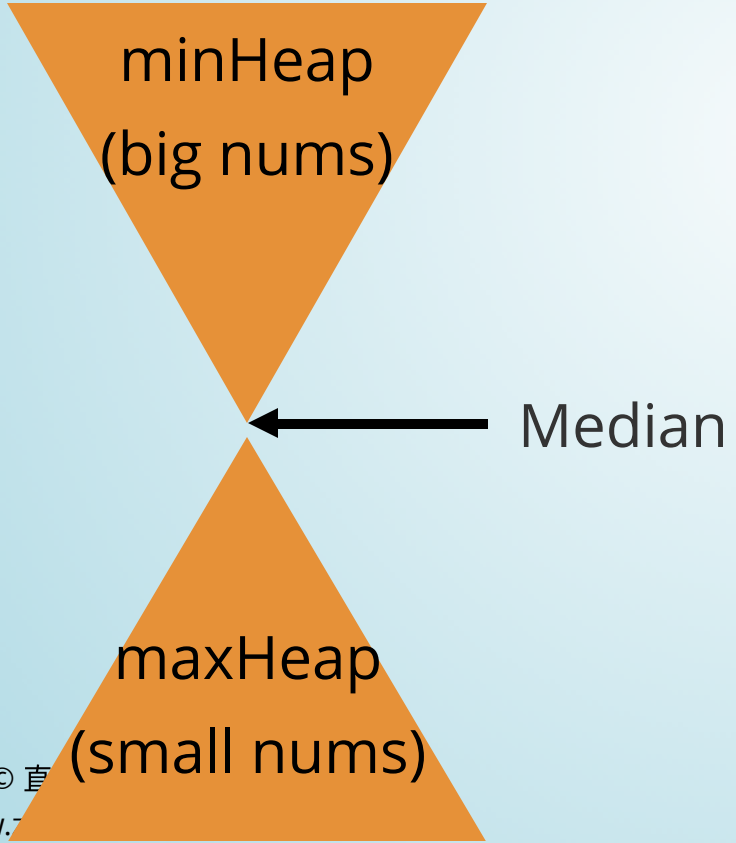
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff

Find Median in Data Stream

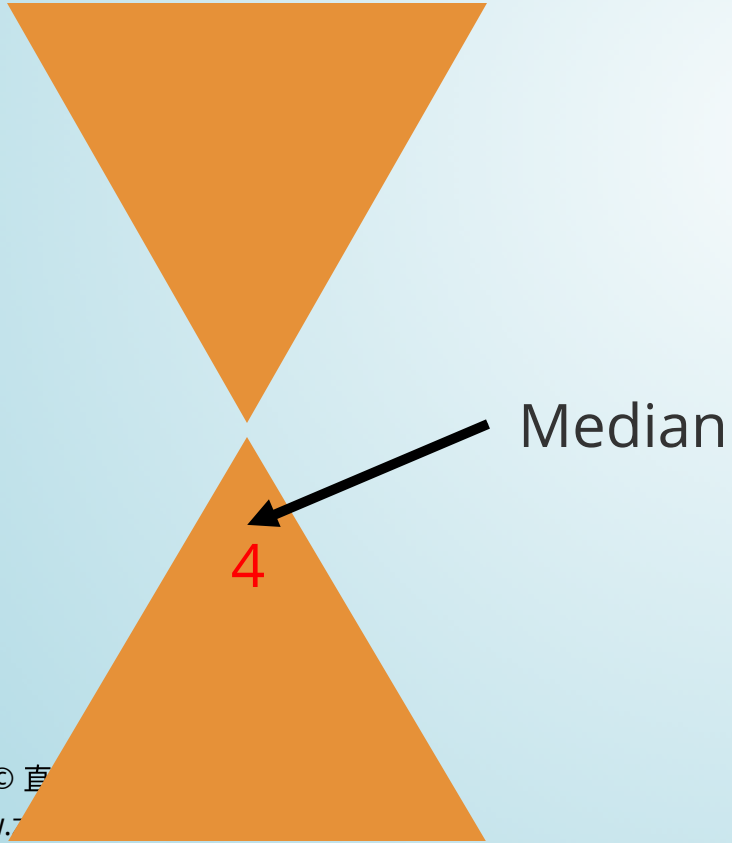
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

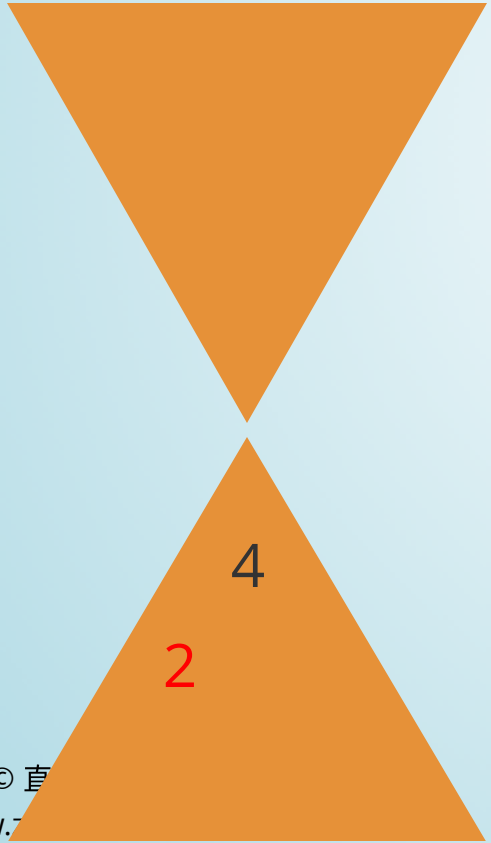
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

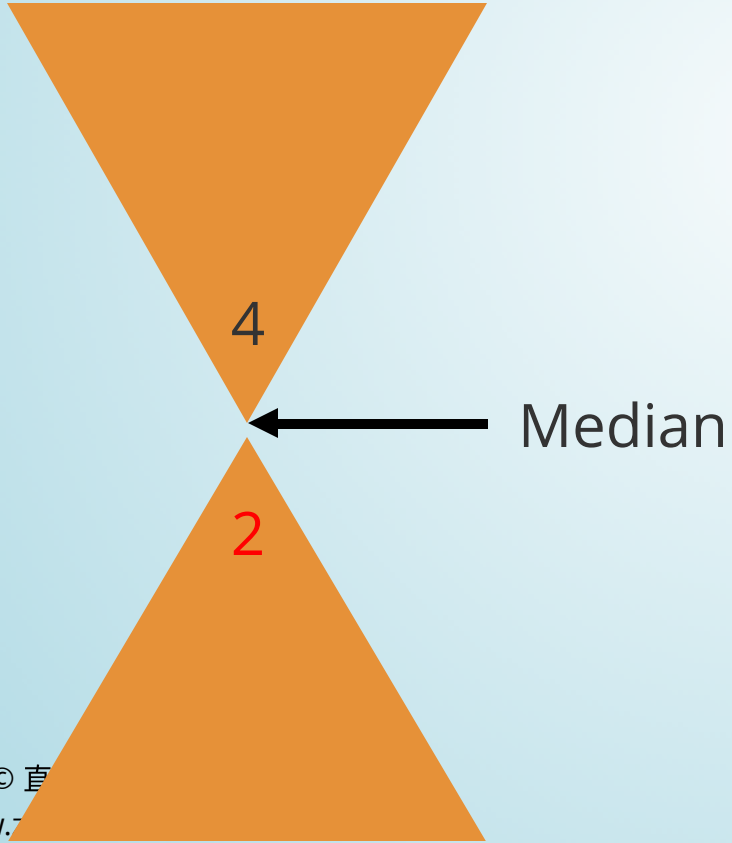
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

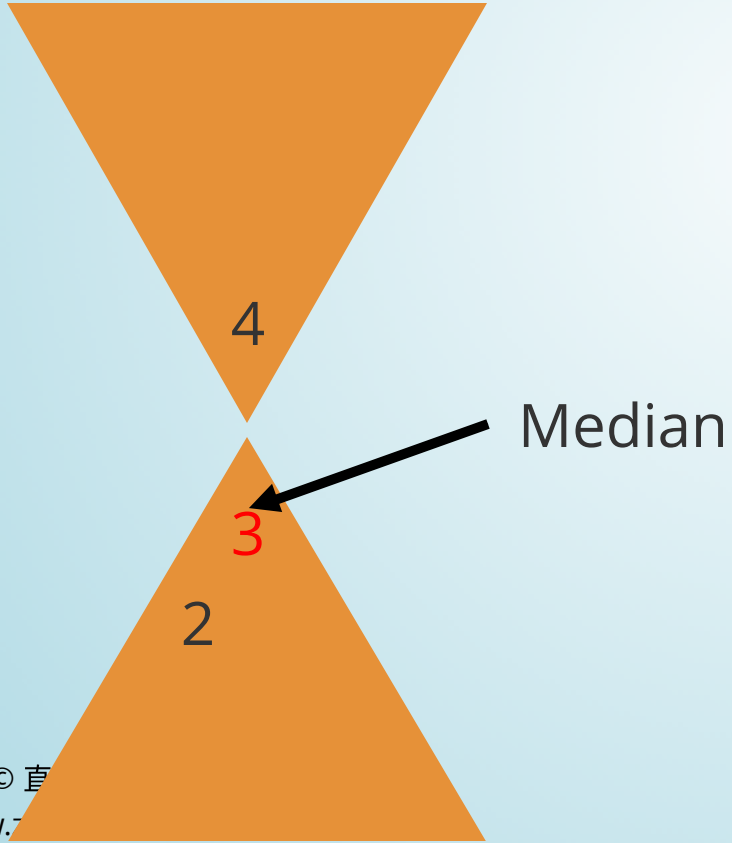
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

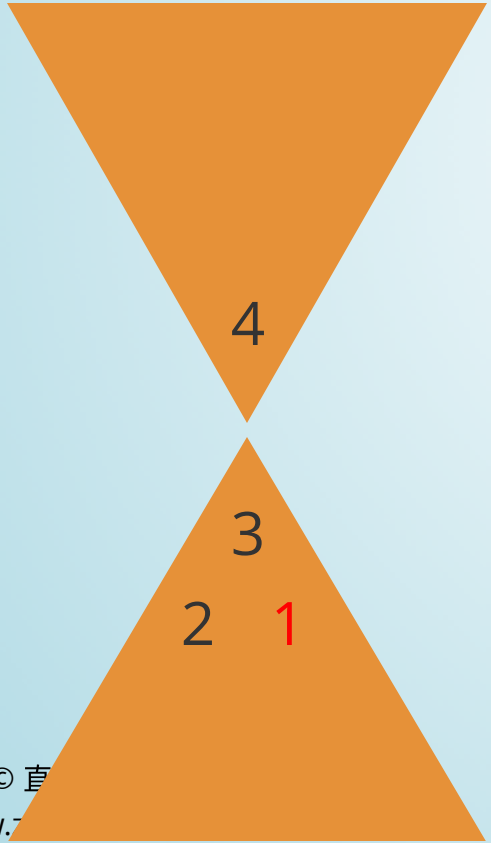
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

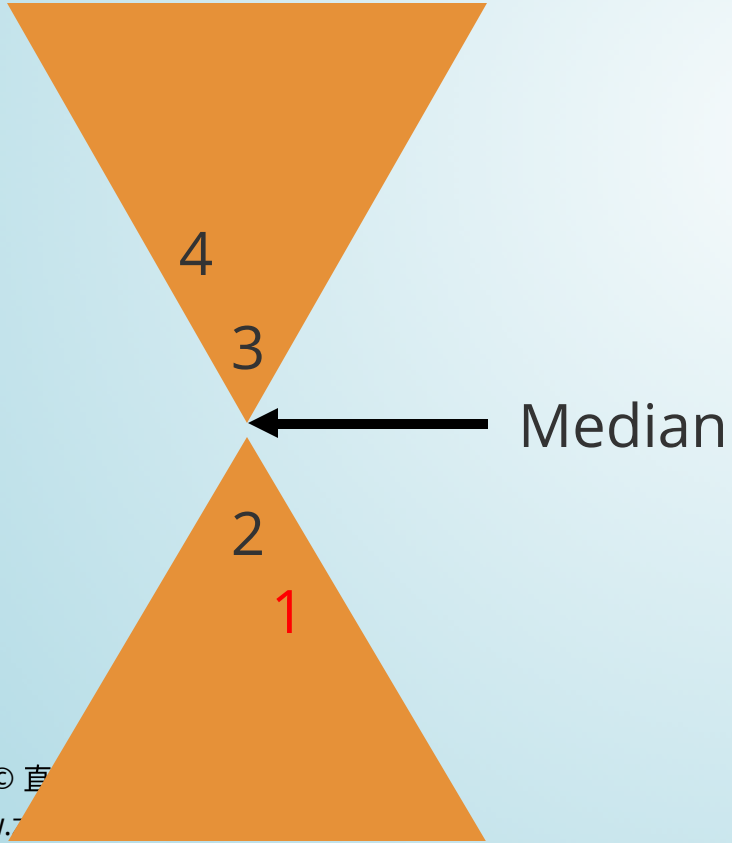
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

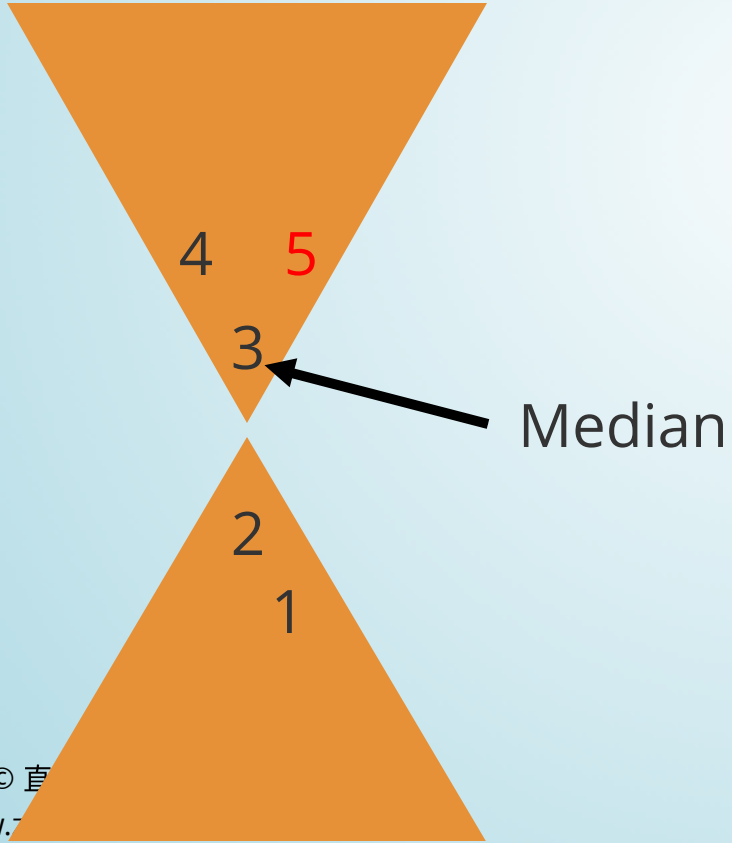
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.



- If num > minHeap.peek
  - Add into minHeap
- Otherwise
  - Add into maxHeap
- Maintain size diff
- [4, 2, 3, 1, 5]

Find Median in Data Stream

```
class MedianFinder {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>();

    // Adds a number into the data structure.
    public void addNum(int num) {
        if (!minHeap.isEmpty() && num > minHeap.peek()) {
            minHeap.offer(num);
        } else {
            maxHeap.offer(-num);
        }
        if (minHeap.size() - maxHeap.size() == 2) {
            maxHeap.offer(-minHeap.poll());
        } else if (maxHeap.size() - minHeap.size() == 2) {
            minHeap.offer(-maxHeap.poll());
        }
    }
    // Returns the median of current data stream
    public double findMedian() {
        if (minHeap.size() > maxHeap.size()) {
            return minHeap.peek();
        }
        if (minHeap.size() < maxHeap.size()) {
            return -maxHeap.peek();
        }
        return (double)(minHeap.peek() - maxHeap.peek()) / 2.0;
    }
```

Find Median in Data Stream

```
class MedianFinder {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>();

    // Adds a number into the data structure.
    public void addNum(int num) {
        minHeap.offer(num);
        maxHeap.offer(-minHeap.poll());
        if (maxHeap.size() - minHeap.size() > 1) {
            minHeap.offer(-maxHeap.poll());
        }
    }

    // Returns the median of current data stream
    public double findMedian() {
        if (minHeap.size() == maxHeap.size()) {
            return (double)(minHeap.peek() - maxHeap.peek()) / 2.0;
        }
        return -maxHeap.peek();
    }
};
```

Top N Numbers

Given **m** lines, each line have **n** numbers, if you choose one from each line and add them up, you could have **n^m** different ways. Please give the largest **n** results of these results. The output need to also in descending order.

Examples:

m= 3, n = 3

5 8 7

2 9 5

0 2 3

Result:

[20, 19, 19]

Hint: 20 = 8 + 9 + 3, 19 = 7 + 9 + 3, 19 = 8 + 9 + 2

Top N Numbers

Of course we can use Brute force way

Is there a better way

Think of this. We add numbers in one line each time, then we always maintain the top n. We know that data not in top n will never become part of the final result.

Therefore we definitely will reduce the time we compute the sum.

How to maintain the top n? We use a minHeap

```java
public List<Integer> topNumbers(int[][] numbers) {
        int m = numbers.length;
        int n = numbers[0].length;
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(n);
        for (int i: numbers[0]) {
                minHeap.add(i);
        }
        int[] list = new int[n];
        for (int i = 1; i < m; i ++) {
                for (int j = n - 1; j >= 0; j --) {
                        list[j] = minHeap.poll();
                }
                int[] cur = numbers[i];
                Arrays.sort(cur);
                int largest = cur[n - 1];
                for (int j = 0; j < n; j ++) {
                        minHeap.add(largest + list[j]);
                }
                for (int j = n - 2; j >= 0; j --) {
                        for (int r = 0; r < n ; r ++) {
                                if (cur[j] + list[r] < minHeap.peek()) {
                                        break;
                                }
                                minHeap.poll();
                                minHeap.add(cur[j] + list[r]);
                        }
                }
        }
        List<Integer> result = new ArrayList<>();
        result.addAll(minHeap);
        Collections.sort(result, comparator);
        return result;
}
```

```
Comparator<Integer> comparator = new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        return b - a;
    }
};
```

Heap Summary

- Data Stream
  - Need to be maintained/update all the time.
- Max or min is needed, but no need for sorting
- Other use cases
  - Heap sort