Heap Data Structure Java

Implementation Of Heap Data Structure in Java

Here’s a basic implementation of a Min Heap data structure in Java:

MinHeap.java

```java
public class MinHeap {
    private int[] heap;
    private int size;
    private int maxSize;

    // Constructor to initialize the heap with a maximum size
    public MinHeap(int maxSize) {
        this.maxSize = maxSize;
        this.size = 0;
        heap = new int[this.maxSize + 1];
        heap[0] = Integer.MIN_VALUE;
    }

    // Get the position of the parent of the node at position pos
    private int parent(int pos) {
        return pos / 2;
    }

    // Get the position of the left child of the node at position pos
    private int leftChild(int pos) {
        return (2 * pos);
    }

    // Get the position of the right child of the node at position pos
    private int rightChild(int pos) {
        return (2 * pos) + 1;
    }

    // Check if the node at position pos is a leaf node
    private boolean isLeaf(int pos) {
        return pos >= (size / 2) && pos <= size;
    }

    // Swap two nodes of the heap
    private void swap(int fpos, int spos) {
        int tmp;
        tmp = heap[fpos];
        heap[fpos] = heap[spos];
        heap[spos] = tmp;
    }

    // Heapify the node at position pos
    private void minHeapify(int pos) {
        if (!isLeaf(pos)) {
            if (heap[pos] > heap[leftChild(pos)] || heap[pos] > heap[rightChild(pos)]) {
                if (heap[leftChild(pos)] < heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    minHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    minHeapify(rightChild(pos));
                }
            }
        }
    }

    // Insert an element into the heap
    public void insert(int element) {
        if (size >= maxSize) {
            System.out.println("Heap is full");
            return;
        }
        heap[++size] = element;
        int current = size;

        while (heap[current] < heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
        }
    }

    // Remove and return the minimum element from the heap
    public int remove() {
        int popped = heap[1];
        heap[1] = heap[size--];
        minHeapify(1);
        return popped;
    }
```

```
        // Print the elements of the heap
        public void print() {
            for (int i = 1; i <= size / 2; i++) {
                System.out.print("Parent: " + heap[i]
                        + " Left Child: " + heap[2 * i]
                        + " Right Child: " + heap[2 * i + 1]);
                System.out.println();
            }
        }

        public static void main(String[] arg) {
            System.out.println("The Min Heap is ");
            MinHeap minHeap = new MinHeap(15);
            minHeap.insert(5);
            minHeap.insert(3);
            minHeap.insert(17);
            minHeap.insert(10);
            minHeap.insert(84);
            minHeap.insert(19);
            minHeap.insert(6);
            minHeap.insert(22);
            minHeap.insert(9);

            minHeap.print();
            System.out.println("The Min value is " + minHeap.remove());
        }
    }
```

Explanation of Code:

- **Constructor**: Initializes the heap array and sets its initial size to zero.
- **parent, leftChild, rightChild**: Helper functions to get indices of the parent, left, and right children of a given index.
- **isLeaf**: Checks if a node is a leaf node (i.e., it has no children).
- **minHeapify**: Ensures that the heap property is maintained for a given node and its subtree.
- **insert**: Adds a new element to the heap and ensures the heap property by "bubbling up" the element as needed.
- **remove**: Removes and returns the minimum element (root) of the heap, then restores the heap property.
- **print**: Displays the heap's elements in terms of parent and child relationships.

Implementation Of Max Heap Data Structure in Java

Here’s an implementation of a Max Heap data structure in Java:

MaxHeap.java

java

```java
public class MaxHeap {
    private int[] heap;
    private int size;
    private int maxSize;

    // Constructor to initialize the max heap with a specified size
    public MaxHeap(int maxSize) {
        this.maxSize = maxSize;
        this.size = 0;
        heap = new int[this.maxSize + 1];
        heap[0] = Integer.MAX_VALUE; // Sentinel value at index 0
    }

    // Get the index of the parent node
    private int parent(int pos) {
        return pos / 2;
    }

    // Get the index of the left child
    private int leftChild(int pos) {
        return (2 * pos);
    }

    // Get the index of the right child
    private int rightChild(int pos) {
        return (2 * pos) + 1;
    }

    // Check if a node is a leaf
    private boolean isLeaf(int pos) {
        return pos >= (size / 2) && pos <= size;
    }

    // Swap two nodes in the heap
    private void swap(int fpos, int spos) {
        int tmp = heap[fpos];
        heap[fpos] = heap[spos];
        heap[spos] = tmp;
    }
    // Heapify the node at position pos to maintain the max heap property
    private void maxHeapify(int pos) {
        if (!isLeaf(pos)) {
            if (heap[pos] < heap[leftChild(pos)] || heap[pos] < heap[rightChild(pos)]) {
                if (heap[leftChild(pos)] > heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    maxHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    maxHeapify(rightChild(pos));
                }
            }
        }
    }

    // Insert an element into the heap
    public void insert(int element) {
        if (size >= maxSize) {
            System.out.println("Heap is full");
            return;
        }
        heap[++size] = element;
        int current = size;

        // Move up until the heap property is restored
        while (heap[current] > heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
        }
    }

    // Remove and return the maximum element (root) from the heap
    public int remove() {
        int popped = heap[1];
        heap[1] = heap[size--];
        maxHeapify(1);
        return popped;
    }

    // Print the elements of the heap
    public void print() {
        for (int i = 1; i <= size / 2; i++) {
            System.out.print("Parent: " + heap[i]
                    + " Left Child: " + heap[2 * i]
                    + " Right Child: " + heap[2 * i + 1]);
            System.out.println();
        }
    }

    public static void main(String[] arg) {
        System.out.println("The Max Heap is ");
        MaxHeap maxHeap = new MaxHeap(15);
        maxHeap.insert(5);
        maxHeap.insert(3);
        maxHeap.insert(17);
        maxHeap.insert(10);
        maxHeap.insert(84);
        maxHeap.insert(19);
        maxHeap.insert(6);
        maxHeap.insert(22);
        maxHeap.insert(9);

        maxHeap.print();
        System.out.println("The Max value is " + maxHeap.remove());
    }
}
```

Explanation of Code:

- **Constructor**: Initializes the max heap array and sets its size to zero. `` `heap[0]` `` is set to `` `Integer.MAX_VALUE` `` to act as a sentinel value.
- **parent, leftChild, rightChild**: Helper functions to get indices of the parent, left, and right children.
- **isLeaf**: Checks if a node is a leaf (i.e., has no children).
- **maxHeapify**: Maintains the max heap property by checking if a node violates the max heap property and fixing it if necessary.
- **insert**: Adds a new element to the heap and restores the max heap property by "bubbling up" the new element.
- **remove**: Removes and returns the maximum element (root) and then restores the max heap property.
- **print**: Displays the heap's elements with their parent and child relationships.

This code provides a fully functional Max Heap that supports insertion, deletion of the maximum element, and display of the heap structure.

Given an array A of size N, the task is to check if the given array represents a Binary MIN Heap or MAX Heap.Kindly explan it withh example

To determine if an array represents a **Binary Min Heap** or **Max Heap**, we need to verify specific properties based on how elements are organized in the array.

Properties of Min Heap and Max Heap:

1. **Binary Min Heap**:
    - The parent node is always **smaller** than or equal to its child nodes.
    - For an array `A`, this means `A[i] <= A[2 * i + 1]` and `A[i] <= A[2 * i + 2]` for all valid parent indices `i`.
2. **Binary Max Heap**:
    - The parent node is always **greater** than or equal to its child nodes.
    - For an array `A`, this means `A[i] >= A[2 * i + 1]` and `A[i] >= A[2 * i + 2]` for all valid parent indices `i`.

How Heaps Are Represented in Arrays:

- Given a heap in an array form, each element at index `i` has:
    - **Left child** at index `2 * i + 1`
    - **Right child** at index `2 * i + 2`
    - **Parent** at index `(i - 1) / 2` (for `i > 0`)

Algorithm to Check if an Array is a Min Heap or Max Heap:

1. Loop through each element in the array up to `N / 2` (as only non-leaf nodes need to be checked).
2. For each node `A[i]`:
    - **Min Heap Check**: Verify that `A[i] <= A[2 * i + 1]` and `A[i] <= A[2 * i + 2]` (if child indices are within bounds).
    - **Max Heap Check**: Verify that `A[i] >= A[2 * i + 1]` and `A[i] >= A[2 * i + 2]` (if child indices are within bounds).
3. If all nodes satisfy the min heap property, the array is a min heap. If all nodes satisfy the max heap property, the array is a max heap. If neither property is satisfied for all nodes, the array is neither a min heap nor a max heap.

Example:

Consider an array `A = {10, 5, 3, 2, 4}` and `N = 5`.

Step-by-Step Check:

1. **Check if it's a Min Heap**:
    - `A[0] = 10`, left child `A[1] = 5`, right child `A[2] = 3`
        - 10 > 5 and 10 > 3, so it **does not satisfy min heap property** at root node.
    - Since `A[0]` fails the min heap property, this array **is not a Min Heap**.

2. **Check if it’s a Max Heap**:
   - `A[0] = 10`, left child `A[1] = 5`, right child `A[2] = 3`
     - 10 >= 5 and 10 >= 3, so it **satisfies max heap property**.
   - `A[1] = 5`, left child `A[3] = 2`, right child `A[4] = 4`
     - 5 >= 2 and 5 >= 4, so it **satisfies max heap property**.
   - `A[2] = 3` is a leaf node, so no further check is needed.

   Since all nodes satisfy the max heap property, this array **is a Max Heap**.

Code Implementation

Here’s a Java function to check if an array is a Min Heap, Max Heap, or neither:

```java
public class HeapCheck {
    // Function to check if array is a Min Heap
    public static boolean isMinHeap(int[] arr, int n) {
        for (int i = 0; i <= (n - 2) / 2; i++) {
            // If left child is greater than the parent
            if (2 * i + 1 < n && arr[i] > arr[2 * i + 1]) {
                return false;
            }
            // If right child is greater than the parent
            if (2 * i + 2 < n && arr[i] > arr[2 * i + 2]) {
                return false;
            }
        }
        return true;
    }

    // Function to check if array is a Max Heap
    public static boolean isMaxHeap(int[] arr, int n) {
        for (int i = 0; i <= (n - 2) / 2; i++) {
            // If left child is smaller than the parent
            if (2 * i + 1 < n && arr[i] < arr[2 * i + 1]) {
                return false;
            }
            // If right child is smaller than the parent
            if (2 * i + 2 < n && arr[i] < arr[2 * i + 2]) {
                return false;
            }
        }
        return true;
    }
    // Main function to determine the type of heap
    public static void main(String[] args) {
        int[] A = {10, 5, 3, 2, 4};
        int n = A.length;

        if (isMinHeap(A, n)) {
            System.out.println("The array represents a Min Heap");
        } else if (isMaxHeap(A, n)) {
            System.out.println("The array represents a Max Heap");
        } else {
            System.out.println("The array is neither a Min Heap nor a Max Heap");
        }
    }
}
```

Output for Example

For `A = {10, 5, 3, 2, 4}`, the output will be:

```c
The array represents a Max Heap
```

This approach efficiently determines the type of heap in `O(N)` time complexity by examining each non-leaf node once and checking the heap property.

Kth Largest Element and smallest element in an Array.Kindly solve it using Priority queue.Also explain it with example and output

To find the **Kth largest** and **Kth smallest** elements in an array using a **Priority Queue** in Java, we can use **Min Heaps** and **Max Heaps**.

Approach

1. **Kth Largest Element**:

- Use a **Min Heap** (Priority Queue in Java).
- Insert the first `K` elements of the array into the min heap.
- For each remaining element, if it's greater than the root of the heap (the smallest element in the heap), remove the root and insert the new element.
- At the end, the root of the min heap will be the Kth largest element.

2. **Kth Smallest Element**:
   - Use a **Max Heap**.
   - Insert the first `K` elements of the array into the max heap.
   - For each remaining element, if it's smaller than the root of the heap (the largest element in the heap), remove the root and insert the new element.
   - At the end, the root of the max heap will be the Kth smallest element.

Example

Consider an array `A = {7, 10, 4, 3, 20, 15}` and `K = 3`.

- **Finding the 3rd Largest Element**:
  - We use a **Min Heap** of size `K = 3`.
  - Insert the first three elements: `{7, 10, 4}` → Min Heap becomes `{4, 10, 7}` (smallest element at the root).
  - Next, consider `3`. Since `3` is smaller than the root (`4`), we ignore it.
  - Next, consider `20`. Since `20` is greater than `4`, remove `4` and add `20` → Min Heap becomes `{7, 10, 20}`.
  - Next, consider `15`. Since `15` is greater than `7`, remove `7` and add `15` → Min Heap becomes `{10, 15, 20}`.
  - The root of the min heap is `10`, so the 3rd largest element is **10**.
- **Finding the 3rd Smallest Element**:
  - We use a **Max Heap** of size `K = 3`.
  - Insert the first three elements: `{7, 10, 4}` → Max Heap becomes `{10, 7, 4}` (largest element at the root).
  - Next, consider `3`. Since `3` is smaller than `10`, remove `10` and add `3` → Max Heap becomes `{7, 4, 3}`.
  - Next, consider `20`. Since `20` is larger than `7`, we ignore it.
  - Next, consider `15`. Since `15` is larger than `7`, we ignore it.
  - The root of the max heap is `7`, so the 3rd smallest element is **7**.

Java Code Implementation

Here's how you can implement this in Java using `PriorityQueue` for both min and max heaps.

java

```java
import java.util.Collections;
import java.util.PriorityQueue;

public class KthLargestSmallest {

    // Function to find Kth largest element
    public static int findKthLargest(int[] arr, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Insert first k elements into the min heap
        for (int i = 0; i < k; i++) {
            minHeap.add(arr[i]);
        }

        // Process the remaining elements
        for (int i = k; i < arr.length; i++) {
            if (arr[i] > minHeap.peek()) {
                minHeap.poll();
                minHeap.add(arr[i]);
            }
        }

        return minHeap.peek(); // The root is the Kth largest element
    }

    // Function to find Kth smallest element
    public static int findKthSmallest(int[] arr, int k) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        // Insert first k elements into the max heap
        for (int i = 0; i < k; i++) {
            maxHeap.add(arr[i]);
        }

        // Process the remaining elements
        for (int i = k; i < arr.length; i++) {
            if (arr[i] < maxHeap.peek()) {
                maxHeap.poll();
                maxHeap.add(arr[i]);
            }
        }

        return maxHeap.peek(); // The root is the Kth smallest element
    }

    public static void main(String[] args) {
        int[] A = {7, 10, 4, 3, 20, 15};
        int K = 3;
        int kthLargest = findKthLargest(A, K);
        int kthSmallest = findKthSmallest(A, K);

        System.out.println("The " + K + "rd largest element is " + kthLargest);
        System.out.println("The " + K + "rd smallest element is " + kthSmallest);
    }
}
```

Explanation of the Code

1. **findKthLargest**:
   - A `PriorityQueue` (min heap) is used to maintain the largest `K` elements.
   - After processing, the root of this min heap (smallest element in the top `K` largest) is the Kth largest element.
2. **findKthSmallest**:
   - A `PriorityQueue` with `Collections.reverseOrder()` is used to create a max heap.
   - This max heap maintains the smallest `K` elements.
   - After processing, the root of this max heap (largest element in the top `K` smallest) is the Kth smallest element.

Output

For the array `A = {7, 10, 4, 3, 20, 15}` with `K = 3`, the output will be:

```
csharp

  The 3rd largest element is 10
  The 3rd smallest element is 7
```

This solution is efficient with time complexity **O(N log K)** for both functions, as we only maintain a heap of size `K` and process each element of the array.

Merge M sorted list using Brute force and optimized manner

To merge `M` sorted lists, we can approach the problem in two ways:

1. **Brute Force Method**: Simply combine all lists and then sort the result.
2. **Optimized Method using a Min Heap**: Merge the lists efficiently using a min heap to keep track of the smallest element across lists.

Let's go through both approaches.

Example

Consider the following sorted lists:

```plaintext
list1 = [1, 4, 7]
list2 = [2, 5, 8]
list3 = [3, 6, 9]
```

Approach 1: Brute Force

1. Combine all elements from all lists into a single list.
2. Sort this combined list.
3. The result will be a single sorted list containing all elements from the input lists.

Brute Force Code

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MergeSortedListsBruteForce {
    public static List<Integer> mergeKSortedLists(List<List<Integer>> lists) {
        List<Integer> mergedList = new ArrayList<>();
        // Combine all elements into one list
        for (List<Integer> list : lists) {
            mergedList.addAll(list);
        }

        // Sort the combined list
        Collections.sort(mergedList);

        return mergedList;
    }

    public static void main(String[] args) {
        List<List<Integer>> lists = new ArrayList<>();
        lists.add(List.of(1, 4, 7));
        lists.add(List.of(2, 5, 8));
        lists.add(List.of(3, 6, 9));

        List<Integer> result = mergeKSortedLists(lists);
        System.out.println("Merged List (Brute Force): " + result);
    }
}
```

Output:

```java
Merged List (Brute Force): [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complexity Analysis

- **Time Complexity**: $O(N \log N)$, where N is the total number of elements across all lists, since sorting is $O(N \log N)$.
- **Space Complexity**: $O(N)$, as we store all elements in a combined list.

Approach 2: Optimized Method using Min Heap (Priority Queue)

In this approach, we use a **Min Heap** (Java's `` `PriorityQueue` ``) to keep track of the smallest element across the lists. By using a min heap, we can efficiently merge the sorted lists in $O(N \log M)$ time, where M is the number of lists.

Steps:

1. Initialize a min heap.
2. Insert the first element of each list into the min heap along with the list’s index and element’s index.
3. While the heap is not empty:
   - Remove the smallest element from the heap and add it to the result.
   - If the list of the removed element has more elements, insert the next element from that list into the min heap.
4. Continue until all elements are processed.

Optimized Code Using Min Heap

```java
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

class Element {
    int value;
    int listIndex;
    int elementIndex;

    public Element(int value, int listIndex, int elementIndex) {
        this.value = value;
        this.listIndex = listIndex;
        this.elementIndex = elementIndex;
    }
}

public class MergeSortedListsOptimized {
    public static List<Integer> mergeKSortedLists(List<List<Integer>> lists) {
        PriorityQueue<Element> minHeap = new PriorityQueue<>((a, b) -> Integer.compare(a.value, b.value));
        List<Integer> mergedList = new ArrayList<>();

        // Add the first element of each list to the min heap
        for (int i = 0; i < lists.size(); i++) {
            if (!lists.get(i).isEmpty()) {
                minHeap.add(new Element(lists.get(i).get(0), i, 0));
            }
        }

        // Extract elements one by one from the min heap
        while (!minHeap.isEmpty()) {
            Element smallest = minHeap.poll();
            mergedList.add(smallest.value);

            // If there are more elements in the same list, add the next element to the heap
            if (smallest.elementIndex + 1 < lists.get(smallest.listIndex).size()) {
                int nextValue = lists.get(smallest.listIndex).get(smallest.elementIndex + 1);
                minHeap.add(new Element(nextValue, smallest.listIndex, smallest.elementIndex + 1));
            }
        }

        return mergedList;
    }

    public static void main(String[] args) {
        List<List<Integer>> lists = new ArrayList<>();
        lists.add(List.of(1, 4, 7));
        lists.add(List.of(2, 5, 8));
        lists.add(List.of(3, 6, 9));

        List<Integer> result = mergeKSortedLists(lists);
        System.out.println("Merged List (Optimized): " + result);
    }
}
```

Output:

```java
Merged List (Optimized): [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Explanation of Code

- We define an `` `Element` `` class to store each element's value, the index of its list, and its index within that list.
- We use a `` `PriorityQueue` `` (min heap) to store these elements, ordering by the `` `value` `` field.
- Initially, we add the first element of each list to the min heap.
- We repeatedly poll the smallest element from the heap and add it to the merged list.
- If the polled element's list has more elements, we add the next element from that list into the min heap.

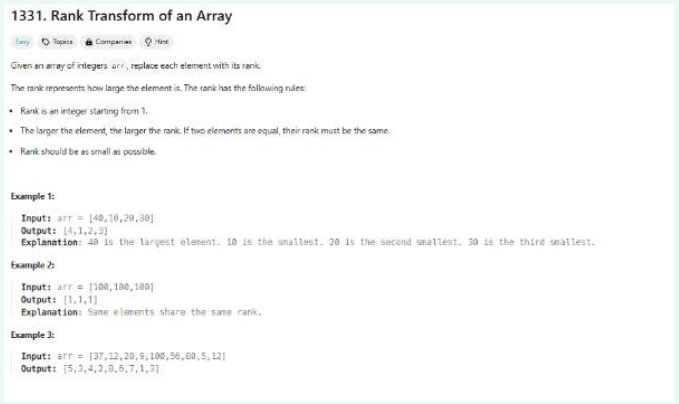
Complexity Analysis

- **Time Complexity**: $O(N \log M)$, where N is the total number of elements across all lists and M is the number of lists. Each insertion and removal from the heap takes $O(\log M)$ time, and we perform these operations N times.
- **Space Complexity**: $O(M)$, for storing elements in the heap (one element from each list at most).

Comparison of Approaches

Method	Time Complexity	Space Complexity	Explanation
Brute Force	$O(N \log N)$	$O(N)$	Simple but inefficient for large lists.
Optimized (Heap)	$O(N \log M)$	$O(M)$	Efficient for merging many sorted lists.

The **optimized method using a min heap** is preferable, especially when `` `M` `` is large, as it avoids unnecessary sorting and only maintains a small heap of size `` `M` ``.


1331. Rank Transform of an Array

Easy · Topics · Companies · Hint

Given an array of integers `arr`, replace each element with its rank.

The rank represents how large the element is. The rank has the following rules:

- Rank is an integer starting from 1.
- The larger the element, the larger the rank. If two elements are equal, their rank must be the same.
- Rank should be as small as possible.

Example 1:

```
Input: arr = [40,10,20,30]
Output: [4,1,2,3]
Explanation: 40 is the largest element. 10 is the smallest. 20 is the second smallest. 30 is the third smallest.
```

Example 2:

```
Input: arr = [100,100,100]
Output: [1,1,1]
Explanation: Same elements share the same rank.
```

Example 3:

```
Input: arr = [37,12,28,9,100,56,80,5,12]
Output: [5,3,4,2,8,6,7,1,3]
```

Kindly try to solve the above problemn using Brute force and Optimized manner

To solve the **Rank Transform of an Array** problem, let's go over both the **Brute Force** and **Optimized** approaches.

Problem Summary:

Given an array of integers `` `arr` ``, replace each element with its rank. The rank is determined by the following rules:

- Rank starts from 1.
- The larger the element, the larger the rank.

- If two elements are equal, they should have the same rank.
- The rank should be as small as possible.

Example Walkthrough

For `` `arr = [40, 10, 20, 30]` ``:

1. Sort the array in ascending order: `` `[10, 20, 30, 40]` ``.
2. Assign ranks based on sorted order:
    - `` `10` `` gets rank `` `1` ``
    - `` `20` `` gets rank `` `2` ``
    - `` `30` `` gets rank `` `3` ``
    - `` `40` `` gets rank `` `4` ``
3. Transform the original array to `` `[4, 1, 2, 3]` `` based on ranks.

Approach 1: Brute Force

1. **Sort and Rank**:
    - Copy the array and sort it.
2. **Assign Ranks**:
    - For each unique element in the sorted array, assign a rank (starting from 1).
3. **Replace Elements**:
    - Replace each element in the original array with its corresponding rank.

Brute Force Code

```
java
```

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class RankTransformBruteForce {
    public static int[] arrayRankTransform(int[] arr) {
        int[] sortedArr = arr.clone();
        Arrays.sort(sortedArr);

        Map<Integer, Integer> rankMap = new HashMap<>();
        int rank = 1;

        // Assign ranks to each unique element in the sorted array
        for (int num : sortedArr) {
            if (!rankMap.containsKey(num)) {
                rankMap.put(num, rank++);
            }
        }

        // Replace each element in the original array with its rank
        int[] result = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            result[i] = rankMap.get(arr[i]);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {40, 10, 20, 30};
        int[] rankedArray = arrayRankTransform(arr);
        System.out.println("Ranked Array (Brute Force): " + Arrays.toString(rankedArray));
    }
}
```

Output:

```java
Ranked Array (Brute Force): [4, 1, 2, 3]
```

Complexity Analysis

- **Time Complexity**: $O(N \log N)$ due to sorting.
- **Space Complexity**: $O(N)$ for the `` `rankMap` `` and `` `result` `` array.

Approach 2: Optimized Using HashMap and Sorting

1. **Sort and Use Indexing**:
    - Use a **sorted list of unique values** from the original array.

   - Assign ranks to each unique element as we iterate through the sorted list.
2. **Rank Map Lookup**:
   - Use a **HashMap** to store the rank of each unique element.
3. **Replace Elements**:
   - Replace each element in the original array by looking up its rank in the HashMap.

Optimized Code

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class RankTransformOptimized {
    public static int[] arrayRankTransform(int[] arr) {
        int[] sortedArr = arr.clone();
        Arrays.sort(sortedArr);

        Map<Integer, Integer> rankMap = new HashMap<>();
        int rank = 1;

        // Assign ranks to each unique element in sorted order
        for (int num : sortedArr) {
            rankMap.putIfAbsent(num, rank++);
        }

        // Replace each element in the original array with its rank
        int[] result = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            result[i] = rankMap.get(arr[i]);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {40, 10, 20, 30};
        int[] rankedArray = arrayRankTransform(arr);
        System.out.println("Ranked Array (Optimized): " + Arrays.toString(rankedArray));
    }
}
```

Output:

```java
Ranked Array (Optimized): [4, 1, 2, 3]
```

Explanation of Optimized Approach

- **Sorting**: Sort the array, which takes $O(N \log N)$.
- **Assigning Ranks**: Use `` `putIfAbsent` `` to assign ranks only to unique elements.
- **Replacing Elements**: Replace elements in $O(N)$ by looking up each element's rank in the HashMap.

Complexity Analysis

- **Time Complexity**: $O(N \log N)$, dominated by sorting.
- **Space Complexity**: $O(N)$, for storing the ranks in the `` `rankMap` ``.

Summary

Approach	Time Complexity	Space Complexity	Explanation
Brute Force	$O(N \log N)$	$O(N)$	Sort and use HashMap to store ranks for each unique element.
Optimized	$O(N \log N)$	$O(N)$	Similar to brute force but slightly optimized with `` `putIfAbsent` ``.

The **optimized method** is more efficient, especially with `` `putIfAbsent` ``, as it skips redundant checks. Both methods are suitable for this problem given the constraints.

Merge k Sorted Arrays in Java

To merge `k` sorted arrays in Java, we can use a **Min Heap (Priority Queue)** to efficiently merge these arrays. This approach ensures that we always get the smallest (or next smallest) element across all arrays at each step, making the process more efficient than a brute-force approach.

Approach: Using a Min Heap (Priority Queue)

1. **Initialize a Min Heap**:
   - Use a min heap (Priority Queue) to keep track of the smallest elements across all arrays.
   - Each element in the heap stores the value, the index of the array it came from, and its index within that array.
2. **Insert the First Element of Each Array**:
   - Insert the first element of each array into the min heap along with its array index and position within the array.
3. **Extract and Insert Next Element**:
   - While the heap is not empty:
     - Extract the smallest element (this is the smallest available element across all arrays).
     - Add this element to the merged result.
     - If there’s a next element in the same array from which the element was extracted, add that next element to the min heap.
4. **Repeat Until All Elements Are Merged**:
   - Continue the process until the min heap is empty, meaning all elements have been added to the merged result.

Code Implementation

Here’s how you can implement this approach in Java:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

class Element {
    int value;
    int arrayIndex;
    int elementIndex;

    public Element(int value, int arrayIndex, int elementIndex) {
        this.value = value;
        this.arrayIndex = arrayIndex;
        this.elementIndex = elementIndex;
    }
}

public class MergeKSortedArrays {

    public static List<Integer> mergeKSortedArrays(List<int[]> arrays) {
        PriorityQueue<Element> minHeap = new PriorityQueue<>((a, b) -> Integer.compare(a.value, b.value));
        List<Integer> mergedList = new ArrayList<>();

        // Initialize the min heap with the first element of each array
        for (int i = 0; i < arrays.size(); i++) {
            if (arrays.get(i).length > 0) {
                minHeap.add(new Element(arrays.get(i)[0], i, 0));
            }
        }

        // Process the min heap until it is empty
        while (!minHeap.isEmpty()) {
            Element smallest = minHeap.poll();
            mergedList.add(smallest.value);

            // If there is a next element in the same array, add it to the heap
            if (smallest.elementIndex + 1 < arrays.get(smallest.arrayIndex).length) {
                int nextValue = arrays.get(smallest.arrayIndex)[smallest.elementIndex + 1];
                minHeap.add(new Element(nextValue, smallest.arrayIndex, smallest.elementIndex + 1));
            }
        }

        return mergedList;
    }

    public static void main(String[] args) {
        List<int[]> arrays = new ArrayList<>();
        arrays.add(new int[]{1, 4, 7});
        arrays.add(new int[]{2, 5, 8});
        arrays.add(new int[]{3, 6, 9});

        List<Integer> mergedArray = mergeKSortedArrays(arrays);
        System.out.println("Merged Array: " + mergedArray);
    }
}
```

Explanation of the Code

1. **Element Class**:
    - This helper class is used to store each element’s value, the index of the array it belongs to, and its index within that array.

2. **PriorityQueue Initialization**:
   - We initialize the `PriorityQueue` (min heap) with a custom comparator to always retrieve the smallest element.
3. **Inserting First Elements**:
   - We add the first element of each array into the min heap, which ensures we start with the smallest elements across all arrays.
4. **Processing the Heap**:
   - We extract the smallest element, add it to the result list, and if there are more elements in the array from which it came, add the next element of that array into the heap.
5. **Final Result**:
   - The final `mergedList` contains all elements in sorted order.

Complexity Analysis

- **Time Complexity**: $O(N \log k)$, where N is the total number of elements across all arrays and k is the number of arrays.
  - Adding and removing from the min heap takes $O(\log k)$, and we perform these operations for each element across all arrays.
- **Space Complexity**: $O(k)$, for storing elements in the heap (at most one element from each array at a time).

Example Output

For the input arrays:

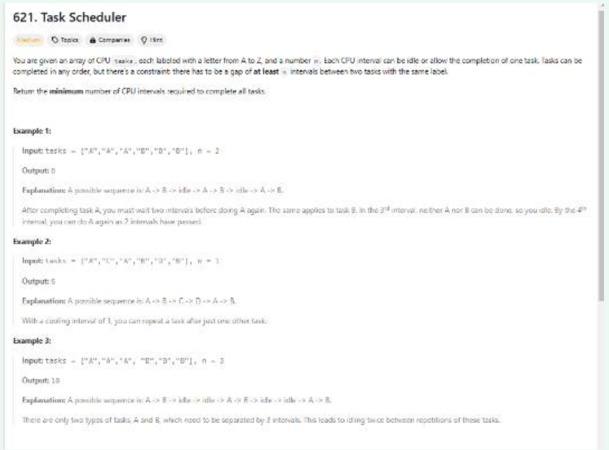
```plaintext


  arrays = [
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9]
  ]
```

The output would be:

```plaintext
Merged Array: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This solution efficiently merges `k` sorted arrays using a min heap and is optimal for cases where `k` is much smaller than the total number of elements `N`.



621. Task Scheduler

Medium Topics Companies Hint

You are given an array of CPU `tasks`, each labeled with a letter from A to Z, and a number `n`. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of **at least** `n` intervals between two tasks with the same label.

Return the **minimum** number of CPU intervals required to complete all tasks.

Example 1:

Input: `tasks = ["A","A","A","B","B","B"], n = 2`

Output: 8

Explanation: A possible sequence is: A -> B -> idle -> A -> B -> idle -> A -> B.

After completing task A, you must wait two intervals before doing A again. The same applies to task B. In the 3rd interval, neither A nor B can be done, so you idle. By the 4th interval, you can do A again as 2 intervals have passed.

Example 2:

Input: `tasks = ["A","C","A","B","D","B"], n = 1`

Output: 6

Explanation: A possible sequence is: A -> B -> C -> D -> A -> B.

With a cooling interval of 1, you can repeat a task after just one other task.

Example 3:

Input: `tasks = ["A","A","A", "B","B","B"], n = 3`

Output: 10

Explanation: A possible sequence is: A -> B -> idle -> idle -> A -> B -> idle -> idle -> A -> B.

There are only two types of tasks, A and B, which need to be separated by 3 intervals. This leads to idling twice between repetitions of these tasks.

Kindly try to solve the above problemnd and explain the code as well

The **Task Scheduler** problem requires us to find the minimum number of intervals (CPU units) required to complete a list of tasks, where each task has a cooling interval `n` that prevents the same task from being scheduled within `n` intervals after it was last executed.

Problem Analysis

- Each task is represented by a character from `'A'` to `'Z'`.
- We are allowed to execute tasks in any order.
- The goal is to minimize the total intervals needed, which may include idle intervals to satisfy the cooling period requirement.
- If a task is executed, it must wait for `n` intervals before it can be executed again.

Approach to Solution

1. **Frequency Counting**:
   - Count the frequency of each task to know which tasks are the most frequent. The most frequent tasks will largely determine the minimum intervals due to the cooling period requirement.
2. **Calculate Minimum Intervals Using the Most Frequent Task**:
   - Let `maxFrequency` be the frequency of the most frequent task.
   - Let `maxCount` be the number of tasks that appear with this `maxFrequency`.
   - To arrange tasks with cooling, we can think of the task execution structure as "buckets" separated by `n` idle periods, where each bucket can hold a task.
3. **Calculate Idle Intervals**:
   - The required intervals are determined by the formula:

     ```scss
     intervals = (maxFrequency - 1) * (n + 1) + maxCount
     ```

   - This formula accounts for `maxFrequency - 1` full slots (each separated by `n` intervals) plus an extra slot for the remaining `maxCount` tasks with the highest frequency.
   - If the calculated intervals are less than the total number of tasks, it means we don't need any idle intervals. Thus, the answer is the maximum of `intervals` and the total number of tasks.
4. **Edge Cases**:
   - If `n = 0`, no cooling period is required, so the answer is simply the length of the tasks array.

Code Implementation

Here's the Java code implementing the above approach:

```java
import java.util.HashMap;
import java.util.Map;
```

```java
public class TaskScheduler {
    public static int leastInterval(char[] tasks, int n) {
        Map<Character, Integer> taskFrequency = new HashMap<>();

        // Count the frequency of each task
        for (char task : tasks) {
            taskFrequency.put(task, taskFrequency.getOrDefault(task, 0) + 1);
        }

        // Find the maximum frequency
        int maxFrequency = 0;
        for (int freq : taskFrequency.values()) {
            maxFrequency = Math.max(maxFrequency, freq);
        }

        // Count how many tasks have the maximum frequency
        int maxCount = 0;
        for (int freq : taskFrequency.values()) {
            if (freq == maxFrequency) {
                maxCount++;
            }
        }

        // Calculate the minimum intervals needed
        int intervals = (maxFrequency - 1) * (n + 1) + maxCount;

        // The result is the maximum of intervals and the total number of tasks
        return Math.max(intervals, tasks.length);
    }

    public static void main(String[] args) {
        char[] tasks1 = {'A', 'A', 'A', 'B', 'B', 'B'};
        int n1 = 2;
        System.out.println("Output for Example 1: " + leastInterval(tasks1, n1)); // Output: 8

        char[] tasks2 = {'A', 'A', 'A', 'B', 'B', 'B'};
        int n2 = 0;
        System.out.println("Output for Example 2: " + leastInterval(tasks2, n2)); // Output: 6

        char[] tasks3 = {'A', 'A', 'A', 'B', 'B', 'B'};
        int n3 = 1;
        System.out.println("Output for Example 3: " + leastInterval(tasks3, n3)); // Output: 6
    }
}
```

Explanation of Code

1. **Frequency Counting**:

- We use a `HashMap` to count the occurrences of each task. This lets us easily access each task's frequency.

2. **Finding `maxFrequency` and `maxCount`**:
   - We determine `maxFrequency` as the highest count of any single task.
   - `maxCount` counts how many tasks have this `maxFrequency`.
3. **Calculating Intervals**:
   - Using the formula `(maxFrequency - 1) * (n + 1) + maxCount`, we calculate the minimum intervals needed to arrange the tasks with idle intervals if required.
   - We take the maximum of this result and the total number of tasks, as idle intervals are unnecessary if the tasks can be arranged without needing extra idle intervals.

Example Walkthrough

Example 1

```plaintext
tasks = ['A', 'A', 'A', 'B', 'B', 'B'], n = 2
```

- Task frequencies: `A = 3`, `B = 3`
- `maxFrequency = 3` (for both A and B), and `maxCount = 2`
- Minimum intervals = `(3 - 1) * (2 + 1) + 2 = 8`

Thus, the output is `8`.

Example 2

```plaintext
tasks = ['A', 'A', 'A', 'B', 'B', 'B'], n = 1
```

- Task frequencies: `A = 3`, `B = 3`
- `maxFrequency = 3`, `maxCount = 2`

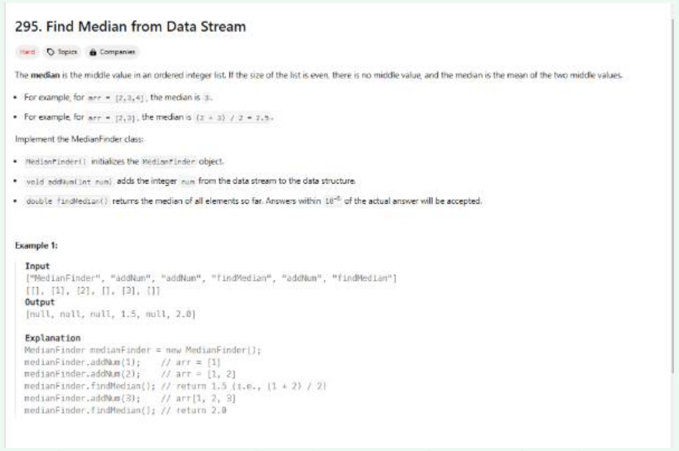
- Minimum intervals = `(3 - 1) * (1 + 1) + 2 = 6`

Thus, the output is `6`.

Complexity Analysis

- **Time Complexity**: $O(N)$, where N is the number of tasks. We perform frequency counting in $O(N)$, then find `maxFrequency` and `maxCount` in $O(1)$ since we only have 26 possible tasks (A to Z).
- **Space Complexity**: $O(1)$, assuming we only store up to 26 tasks in the map.

This solution is efficient and works well within the constraints.



295. Find Median from Data Stream

Hard · Topics · Companies

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is `3`.
- For example, for `arr = [2,3]`, the median is `(2 + 3) / 2 = 2.5`.

Implement the MedianFinder class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:

```
Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Kindly try to solve the above problemnd and explain the code as well

To solve the **Find Median from Data Stream** problem, we need to implement a data structure that can add numbers and find the median efficiently at any time. This problem can be efficiently solved using two heaps (priority queues).

Problem Analysis

1. **Median**:
    - The median is the middle value in an ordered list.
    - If the list has an odd number of elements, the median is the middle element.
    - If the list has an even number of elements, the median is the average of the two middle elements.
2. **Requirements**:
    - We need to add numbers to the data structure.
    - We need to retrieve the median efficiently.

Approach: Using Two Heaps (Min Heap and Max Heap)

1. **Max Heap**:
    - This heap (priority queue) will store the smaller half of the numbers.
    - The root of the max heap will be the largest number in the smaller half.
2. **Min Heap**:
    - This heap will store the larger half of the numbers.
    - The root of the min heap will be the smallest number in the larger half.
3. **Balancing the Heaps**:
    - After adding a number, balance the heaps to ensure:
        - The max heap either has the same number of elements as the min heap or one more element.
        - This balance allows us to find the median efficiently.
4. **Calculating the Median**:
    - If the max heap has more elements, the median is the root of the max heap.
    - If both heaps have the same number of elements, the median is the average of the roots of both heaps.

Code Implementation

Here’s the Java code implementing the above approach:

```java
import java.util.PriorityQueue;

class MedianFinder {
    private PriorityQueue<Integer> maxHeap; // Max heap for the lower half
    private PriorityQueue<Integer> minHeap; // Min heap for the upper half

    public MedianFinder() {
        // Max heap to store the smaller half of the numbers
        maxHeap = new PriorityQueue<>((a, b) -> b - a);
        // Min heap to store the larger half of the numbers
        minHeap = new PriorityQueue<>();
    }

    public void addNum(int num) {
        // Add the new number to maxHeap by default
        maxHeap.add(num);

        // Move the largest element from maxHeap to minHeap to maintain order
        if (!maxHeap.isEmpty() && !minHeap.isEmpty() && maxHeap.peek() > minHeap.peek()) {
            minHeap.add(maxHeap.poll());
        }

        // Balance the heaps if the size difference is more than 1
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.add(maxHeap.poll());
        } else if (minHeap.size() > maxHeap.size()) {
            maxHeap.add(minHeap.poll());
        }
    }

    public double findMedian() {
        // If maxHeap has more elements, median is the root of maxHeap
        if (maxHeap.size() > minHeap.size()) {
            return maxHeap.peek();
        }
        // If both heaps are balanced, median is the average of both roots
        return (maxHeap.peek() + minHeap.peek()) / 2.0;
    }
}
```

Explanation of Code

1. **Constructor (`MedianFinder`)**:
    - Initializes two priority queues (`maxHeap` and `minHeap`).
    - `maxHeap` is implemented as a max heap to hold the smaller half of numbers.
    - `minHeap` is a min heap to hold the larger half of numbers.
2. **addNum(int num)**:
    - First, add `num` to `maxHeap`.

    - Ensure the order by moving the root of `` `maxHeap` `` to `` `minHeap` `` if the max element in `` `maxHeap` `` is greater than the min element in `` `minHeap` ``.
    - Balance the heaps so that `` `maxHeap` `` has at most one more element than `` `minHeap` ``.
3. **findMedian()**:
    - If `` `maxHeap` `` has more elements than `` `minHeap` ``, the median is the root of `` `maxHeap` ``.
    - If both heaps are balanced, the median is the average of the roots of `` `maxHeap` `` and `` `minHeap` ``.

Example Walkthrough

For the sequence of operations in Example 1:

```
plaintext

1. addNum(1):    maxHeap = [1], minHeap = []                -> Median = 1.0
2. addNum(2):    maxHeap = [1], minHeap = [2]               -> Median = (1 + 2) / 2 = 1.5
3. addNum(3):    maxHeap = [2, 1], minHeap = [3]            -> Median = 2.0
```

The output for this sequence would be:

```
graphql

[null, null, null, 1.5, null, 2.0]
```

Complexity Analysis

- **Time Complexity**:
    - **addNum**: $O(\log N)$, where N is the number of elements. Inserting into a heap takes $O(\log N)$.
    - **findMedian**: $O(1)$, since we just peek at the root(s) of the heaps.
- **Space Complexity**: $O(N)$, where N is the total number of elements added, as we are storing all elements in the heaps.

This solution is efficient for a continuous stream of data, where we need to maintain and retrieve the median dynamically.

Top K Frequent Elements
Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

To solve the **Top K Frequent Elements** problem, we need to identify the `k` elements that appear most frequently in the given array `nums`.

Problem Analysis

1. **Input**:
    - `nums`: an array of integers.
    - `k`: an integer representing the number of most frequent elements we need to return.
2. **Output**:
    - An array of the `k` most frequent elements in `nums`.
3. **Constraints**:
    - The order of the returned elements does not matter.
    - We can assume that `k` is always valid (i.e., `1 <= k <= number of unique elements in nums`).

Approach

We can solve this problem efficiently using a **HashMap** to count frequencies and a **Min Heap (Priority Queue)** to keep track of the top `k` frequent elements.

Steps

1. **Frequency Counting**:
    - Use a `HashMap` to count the frequency of each element in `nums`.
2. **Maintain a Min Heap**:
    - Use a min heap (priority queue) of size `k` to store the top `k` elements based on frequency.
    - For each element in the frequency map:
        - If the heap size is less than `k`, add the element to the heap.
        - If the heap size is `k` and the current element's frequency is greater than the root of the heap, remove the root and add the current element.
    - After processing all elements, the heap will contain the `k` most frequent elements.

3. **Extract the Elements from the Heap**:
    - Convert the elements in the heap to the result list.

Code Implementation

Here’s the Java code for this approach:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;

public class TopKFrequentElements {
    public static int[] topKFrequent(int[] nums, int k) {
        // Step 1: Frequency map to count occurrences of each element
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Step 2: Min Heap (Priority Queue) to store top k frequent elements
        PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
                new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
            minHeap.add(entry);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        // Step 3: Extract elements from the heap into the result array
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = minHeap.poll().getKey();
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = topKFrequent(nums, k);
        System.out.println("Top " + k + " frequent elements: ");
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Explanation of Code

1. **Frequency Map**:
   - We count the occurrences of each element using a `` `HashMap` ``.
   - For example, if `` `nums = [1, 1, 1, 2, 2, 3]` ``, the frequency map will be `` `{1=3, 2=2, 3=1}` ``.
2. **Min Heap**:
   - We use a min heap of size `` `k` `` to store the `` `k` `` most frequent elements.
   - Each entry in the min heap is a map entry from the frequency map.
   - If the size of the min heap exceeds `` `k` ``, we remove the element with the smallest frequency to ensure that only the top `` `k` `` frequent elements are retained.
3. **Extracting the Result**:
   - After processing all elements, the min heap contains the `` `k` `` most frequent elements.
   - We extract these elements from the min heap and store them in the result array.

Complexity Analysis

- **Time Complexity**:
  - Building the frequency map takes $O(N)$, where N is the number of elements in `` `nums` ``.
  - Adding elements to the min heap takes $O(M \log k)$, where M is the number of unique elements in `` `nums` `` and `` `k` `` is the heap size.
  - Extracting elements from the heap takes $O(k \log k)$.
  - Overall, the time complexity is $O(N + M \log k)$, which is efficient for this problem.
- **Space Complexity**:
  - The space complexity is $O(M + k)$, where M is the number of unique elements in `` `nums` `` for the frequency map, and `` `k` `` is the size of the min heap.

Example Walkthrough

For the input:

```plaintext
nums = [1, 1, 1, 2, 2, 3], k = 2
```

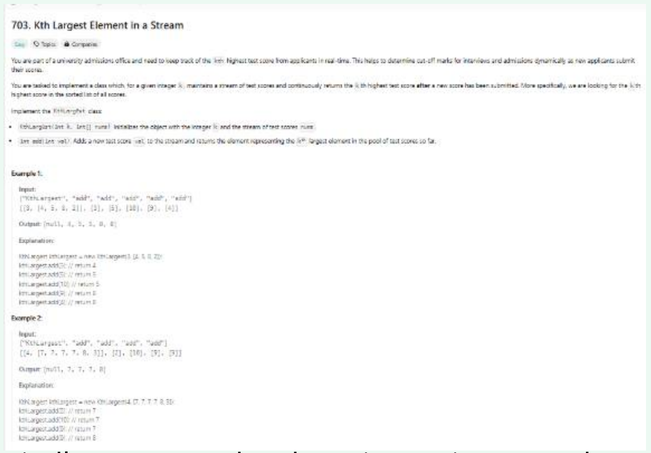
1. **Frequency Map**: `{1=3, 2=2, 3=1}`
2. **Min Heap**:
    - After processing all elements, the min heap will contain the two most frequent elements: `1` and `2`.
3. **Result**:
    - The result array will be `[1, 2]`.

Output

The output for this example will be:

```plaintext
Top 2 frequent elements: 1 2
```

This approach efficiently finds the `k` most frequent elements, using the heap to maintain only the top elements and removing less frequent ones dynamically.



703. Kth Largest Element in a Stream

Easy · Topics · Companies

You are part of a university admissions office and need to keep track of the `kth` highest test score from applicants in real-time. This helps to determine cut-off marks for interviews and admissions dynamically as new applicants submit their scores.

You are tasked to implement a class which, for a given integer `k`, maintains a stream of test scores and continuously returns the `k`th highest test score **after** a new score has been submitted. More specifically, we are looking for the `k`th highest score in the sorted list of all scores.

Implement the `KthLargest` class:

- `KthLargest(int k, int[] nums)` Initializes the object with the integer `k` and the stream of test scores `nums`.
- `int add(int val)` Adds a new test score `val` to the stream and returns the element representing the `k`th largest element in the pool of test scores so far.

Example 1:

Input:
```
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
```
Output: `[null, 4, 5, 5, 8, 8]`

Explanation:

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

Example 2:

Input:
```
["KthLargest", "add", "add", "add", "add"]
[[4, [7, 7, 7, 7, 8, 3]], [2], [10], [9], [9]]
```
Output: `[null, 7, 7, 7, 8]`

Explanation:

```
KthLargest kthLargest = new KthLargest(4, [7, 7, 7, 7, 8, 3]);
kthLargest.add(2); // return 7
kthLargest.add(10); // return 7
kthLargest.add(9); // return 7
kthLargest.add(9); // return 8
```

Kindly represent the above image in text and try to solve the problemn and also explain the code along with output

Problem: 703. Kth Largest Element in a Stream

You are part of a university admissions office and need to keep track of the **k-th highest test score** from applicants in real-time. This helps to determine cut-off marks for interviews and admissions dynamically as new applicants submit their scores.

You are tasked to implement a class that, for a given integer `` `k` ``, maintains a stream of test scores and continuously returns the k-th highest test score after a new score has been submitted. More specifically, we are looking for the **k-th highest element** in the sorted list of all scores.

Implement the `` `KthLargest` `` class:

1. **KthLargest(int k, int[] nums)**:
   - Initializes the object with the integer `` `k` `` and the stream of test scores `` `nums` ``.
2. **int add(int val)**:
   - Adds a new test score `` `val` `` to the stream and returns the element representing the `` `k` ``-th largest element in the pool of test scores so far.

Examples

Example 1:

```plaintext
Input:
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]

Output:
[null, 4, 5, 5, 8, 8]

Explanation:
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
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
```

Example 2:

```plaintext
Input:
["KthLargest", "add", "add", "add", "add", "add"]
[[4, [7, 7, 7, 7, 8, 31]], [2], [10], [9], [9], [9]]

Output:
[null, 7, 7, 8, 8, 8]

Explanation:
KthLargest kthLargest = new KthLargest(4, [7, 7, 7, 7, 8, 31]);
kthLargest.add(2);   // returns 7
kthLargest.add(10);  // returns 7
kthLargest.add(9);   // returns 8
kthLargest.add(9);   // returns 8
kthLargest.add(9);   // returns 8
```

Solution Approach

1. **Min Heap (Priority Queue)**:
    - Use a min heap to store the k largest elements.

- The root of the min heap will hold the k-th largest element.
- If a new element is larger than the root (k-th largest element), we replace the root to ensure that the heap always contains the top k elements.

2. **Process**:

   - Initialize the heap with the first `k` elements of the `nums` array (or fewer if `nums` has less than `k` elements).
   - For every new element added:
     - If the size of the heap is less than `k`, just add the element.
     - If the heap already has `k` elements, only add the new element if it's larger than the smallest element (root of the heap). Remove the smallest element before adding the new one.
   - The root of the heap will always be the k-th largest element after each addition.

Code Implementation

Here's the Java implementation of the `KthLargest` class:

```java
import java.util.PriorityQueue;

class KthLargest {
    private PriorityQueue<Integer> minHeap;
    private int k;

    public KthLargest(int k, int[] nums) {
        this.k = k;
        minHeap = new PriorityQueue<>(k);

        // Initialize the min heap with the first k elements or all elements if fewer than k
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int val) {
        // Add the new value to the heap
        if (minHeap.size() < k) {
            minHeap.offer(val);
        } else if (val > minHeap.peek()) {
            minHeap.poll();
            minHeap.offer(val);
        }

        // The root of the min heap is the k-th largest element
        return minHeap.peek();
        }
    }
```

Explanation of Code

1. **Constructor (`KthLargest(int k, int[] nums)`)**:
    - We initialize the `minHeap` with a size of `k`.
    - We add each element from `nums` to the min heap by calling `add()`.
    - The `add()` function ensures that only the top `k` largest elements are kept in the heap, and the smallest of these `k` elements is at the root (representing the `k`-th largest element).
2. **add(int val)**:
    - If the size of `minHeap` is less than `k`, add `val` to the heap.
    - If `minHeap` already contains `k` elements, only add `val` if it's larger than the smallest element (the root). In this case, remove the root and insert the new value.
    - Return the root of `minHeap`, which is the `k`-th largest element.

Complexity Analysis

- **Time Complexity**:
    - Initializing the heap with `k` elements takes $O(N \log k)$, where N is the length of `nums`.
    - Each `add()` operation takes $O(\log k)$ for insertion into the min heap.
- **Space Complexity**:
    - The space complexity is $O(k)$, since we store only the top `k` largest elements in the min heap.

Example Walkthrough

Example 1

```plaintext

kthLargest = new KthLargest(3, [4, 5, 8, 2])

1. add(3) -> minHeap = [4, 5, 8] -> Returns 4 (3rd largest element)
```

```
  2. add(5) -> minHeap = [5, 5, 8] -> Returns 5 (3rd largest element)
  3. add(10) -> minHeap = [5, 8, 10] -> Returns 5 (3rd largest element)
  4. add(9) -> minHeap = [8, 9, 10] -> Returns 8 (3rd largest element)
  5. add(4) -> minHeap = [8, 9, 10] -> Returns 8 (3rd largest element)
```

Output:

```plaintext
  [null, 4, 5, 5, 8, 8]
```

Example 2

```plaintext
  kthLargest = new KthLargest(4, [7, 7, 7, 7, 8, 31])

  1. add(2) -> minHeap = [7, 7, 7, 8] -> Returns 7 (4th largest element)
  2. add(10) -> minHeap = [7, 8, 10, 31] -> Returns 7 (4th largest element)
  3. add(9) -> minHeap = [8, 9, 10, 31] -> Returns 8 (4th largest element)
  4. add(9) -> minHeap = [8, 9, 10, 31] -> Returns 8 (4th largest element)
  5. add(9) -> minHeap = [8, 9, 10, 31] -> Returns 8 (4th largest element)
```

Output:

```plaintext
  [null, 7, 7, 8, 8, 8]
```

This solution efficiently maintains the k-th largest element in a stream of numbers, using a min heap to manage the top `k` elements in a way that keeps retrieval and insertion time efficient.