PW SKILLS

Lesson Plan

# Priority Queue-2

# Java

PW SKILLS

# Today's Checklist:

- STL of Priority Queue
- STL based in Given Priority
- Question Based on Priority Queue
- Question on Priority Queue

# STL of Priority Queue

A priority queue is a data structure that stores elements based on their priority. In C++ STL, std::priority_queue is an adapter class that implements a priority queue using a heap, where the element with the highest priority (or lowest, depending on the comparator) is served first.

**Max Heap:** A binary heap data structure in which the parent node has a higher value than its children. In the context of std::priority_queue, the default behavior is to create a max heap, ensuring the highest-priority element is at the front.

**Min Heap:** A binary heap data structure in which the parent node has a lower value than its children. In C++ STL, a min heap can be achieved by using a custom comparator, such as std::greater, with std::priority_queue.

**Comparator:** A function or object used to define the order of elements in a data structure. In the context of std::priority_queue, a comparator is used to determine whether one element should come before or after another. The default comparator creates a max heap, while a custom comparator can be used to create a min heap.

**push:** A member function of std::priority_queue used to insert an element into the priority queue while maintaining the heap property.

**pop:** A member function of std::priority_queue used to remove the highest-priority element from the front of the priority queue, adjusting the heap accordingly.

**top:** A member function of std::priority_queue used to access the highest-priority element at the front of the priority queue without removing it.

**size:** A member function of std::priority_queue that returns the number of elements in the priority queue. empty: A member function of std::priority_queue that returns true if the priority queue is empty and false otherwise.

**Example Code:**

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

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Creating a max heap using PriorityQueue
        PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(30);
        maxHeap.add(10);
        maxHeap.add(25);
```

```
        System.out.println("Top element: " + maxHeap.peek());

        // Creating a max heap using PriorityQueue
        System.out.println("Top element (max heap): " + maxHeap.peek());

        // Creating a min heap using custom comparator (Collections.reverseOrder())
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(Collections.reverseOrder());
        minHeap.add(30);
        minHeap.add(10);
        minHeap.add(25);

        System.out.println("Top element (min heap): " + minHeap.peek());

        // Using add to insert elements into a PriorityQueue
        maxHeap.add(50);
        System.out.println("Top element after add: " + maxHeap.peek());

        // Using poll to remove the top element from a PriorityQueue
        maxHeap.poll();
        System.out.println("Top element after poll: " + maxHeap.peek());

        // Using peek to access the top element of a PriorityQueue
        System.out.println("Top element: " + maxHeap.peek());

        // Using size to get the number of elements in a PriorityQueue
        System.out.println("Size of the PriorityQueue: " + maxHeap.size());

        // Using isEmpty to check if a PriorityQueue is empty
        System.out.println("Is the PriorityQueue empty? " + (maxHeap.isEmpty() ? "Yes" :
"No"));
    }
}
```

## Ques : Max Kth element in the array.

Given an integer array nums and an integer k, return the kth largest element in the array.
Note that it is the kth largest element in the sorted order, not the kth distinct element.

**Example Code:**

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }
        return minHeap.peek();
    }
}
```

**Explanation:** We can use a heap data structure to efficiently find the kth largest element in the array. There are two types of heaps we can use for this problem: a min heap and a max heap.
We can use a min heap to keep track of the k largest elements seen so far. We iterate over the array, and for each element, we insert it into the heap. If the size of the heap becomes larger than k, we remove the minimum element from the heap. At the end, the top element of the heap will be the kth largest element in the array.

**Time Complexity** - O(n log k) - Iterates through n elements, performing log k operations for each heap operation to maintain k largest elements.

**Space Complexity-** O(k) - Requires O(k) space to store the top k largest elements in the heap.

**Examples:**
Input: K = 3, N = 4, arr = { {1, 3, 5, 7}, {2, 4, 6, 8}, {0, 9, 10, 11}}
Output: 0 1 2 3 4 5 6 7 8 9 10 11
Explanation: The output array is a sorted array that contains all the elements of the input matrix.

Input: k = 4, n = 4, arr = { {1, 5, 6, 8}, {2, 4, 10, 12}, {3, 7, 9, 11}, {13, 14, 15, 16}}
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Explanation: The output array is a sorted array that contains all the elements of the input matrix.

**Code:**

```
class Solution {
    public int[] mergeKArrays(int[][] arr, int K, int N) {
        // To store the output array
        int[] output = new int[N * K];
        // Priority queue to hold elements and their positions
        PriorityQueue<Pair<Integer, Pair<Integer, Integer>>> pq = new PriorityQueue<>(
                Comparator.comparing(Pair::getKey)
        );

        // Push the first element of each array into the priority queue
        for (int i = 0; i < K; ++i) {
            pq.offer(new Pair<>(arr[i][0], new Pair<>(i, 0)));
        }

        int idx = 0;
        // Merge the arrays
        while (!pq.isEmpty()) {
            Pair<Integer, Pair<Integer, Integer>> pair = pq.poll();
            int val = pair.getKey();
            Pair<Integer, Integer> pos = pair.getValue();
            int i = pos.getKey();
            int j = pos.getValue();

            output[idx++] = val;
            if (j + 1 < N) { // If there are more elements in the current array
                pq.offer(new Pair<>(arr[i][j + 1], new Pair<>(i, j + 1)));
            }
        }
        return output;
    }
}
```

**Explanation:** The idea is to use Min Heap. This MinHeap based solution has the same time complexity which is O(NK log K). But for a different and particular sized array, this solution works much better. The process must start with creating a MinHeap and inserting the first element of all the k arrays. Remove the root element of Minheap and put it in the output array and insert the next element from the array of removed element. To get the result the step must continue until there is no element left in the MinHeap.

**Time Complexity:** O(N * K * log K), Insertion and deletion in a Min Heap requires log K time.

**Space Complexity:** O(K), If Output is not stored then the only space required is the Min-Heap of K elements.

## Merge k Sorted Lists [Leetcode-23]

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.
Merge all the linked-lists into one sorted linked-list and return it.

**Example 1:**

Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]

**Explanation:** The linked-lists are:
[
 1->4->5,
 1->3->4,
 2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6

**Example 2:**

Input: lists = []
Output: []

**Code:**

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        // To store the output linked list
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        // Priority queue to hold nodes from different lists
        PriorityQueue<ListNode> pq = new
PriorityQueue<>(Comparator.comparingInt(node -> node.val));

        // Add the first node from each list to the priority queue
        for (ListNode list : lists) {
            if (list != null) {
                pq.offer(list);
            }
        }
```

```
        // Merge the lists
        while (!pq.isEmpty()) {
            ListNode node = pq.poll();
            current.next = node;
            current = current.next;

            // Move to the next node in the list
            if (node.next != null) {
                pq.offer(node.next);
            }
        }

        return dummy.next;
    }
}
```

**Explanation:** The idea is to use a Min Heap. This MinHeap-based solution has a time complexity of O(NK log K), where N is the average number of nodes in a linked list. The process starts by creating a MinHeap and inserting the first node of each linked list. The root of the MinHeap is the smallest node. Remove the root element of the MinHeap and append it to the output linked list. Insert the next node from the list of the removed element. Repeat these steps until there are no nodes left in the MinHeap.

**Time Complexity:** O(N * K * log K), where N is the average number of nodes in a linked list. Insertion and deletion in a Min Heap require log K time.

**Space Complexity:** O(K), If the output is not stored, then the only space required is the Min-Heap of K elements.

# Find Median from Data Stream [Leetcode-295]

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

**Code:**

```
import java.util.PriorityQueue;

class MedianFinder {
    private PriorityQueue<Integer> maxHeap; // stores the smaller
half of numbers
    private PriorityQueue<Integer> minHeap; // stores the larger
half of numbers

    public MedianFinder() {
        maxHeap = new PriorityQueue<>((a, b) -> b - a);
        minHeap = new PriorityQueue<>();
    }

    public void addNum(int num) {
        if (maxHeap.isEmpty() || num <= maxHeap.peek()) {
            maxHeap.offer(num);
        } else {
            minHeap.offer(num);
        }

        // Balance the heaps
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.offer(maxHeap.poll());
        } else if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    public double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        } else {
            return maxHeap.peek();
        }
    }
}
```

**Explanation:** The solution uses two Priority Queues, one for the smaller half (maxHeap) and one for the larger half (minHeap). The addNum method inserts a new number into the appropriate heap, ensuring that the heaps are balanced. The findMedian method calculates and returns the current median based on the heaps' sizes.

**Time Complexity:** O(log N), as adding a number takes O(log N) time, where N is the number of elements seen so far. Finding the median takes constant time.

**Space Complexity:** O(N), where N is the number of elements seen so far.

PW SKILLS

# THANK YOU !

PW SKILLS