DoubleMapMinPQ

removeMin

The removeMin method primarily operates on a TreeMap, specifically using methods like firstKey() and remove(), which have a time complexity of O(logN) due to the underlying balanced tree structure. Other operations within the method, such as removing an element from a Set or Map, are O(1) but do not dominate the overall complexity. Therefore, t**he worst-case runtime of is Θ(logN)** because the logarithmic operations on the TreeMap are the most time-consuming steps.

changePriority

The changePriority method involves removing the element from its current priority group and adding it to a new one, both of which require operations on the TreeMap like get(), remove(), and put(). These TreeMap operations have a time complexity of O(logN) because they involve navigating a balanced tree to locate or modify entries. Since these logarithmic operations are the most significant in terms of time, the **worst-case runtime is Θ(logN).**

UnsortedArrayMinPQ

removeMin

The removeMin method first calls peekMin, which scans all N elements to find the one with the minimum priority, taking O(N) time. It then scans the list again to locate and remove the corresponding node, an operation that also takes O(N) time due to the list traversal and element removal from an array-based list. Consequently, the **overall worst-case time complexity is Θ(N)**

changePriority

Begins by calling contains, which checks for the element's existence by scanning through all N elements, resulting in O(N) time. It then iterates through the list again to find the node associated with the element and updates its priority, another O(N) operation. As both steps involve traversing the entire list, the **worst-case time complexity is Θ(N)**.

HeapMinPQ

removeMin

The removeMin method efficiently removes the minimum element from the priority queue using the poll() method of the PriorityQueue, which is backed by a binary heap. This operation involves removing the root of the heap and performing a sift-down to maintain the heap property, resulting in a time complexity of O(logN). Therefore, the worst-case runtime **is Θ(logN).**

changePriority

The changePriority method first checks if the element exists in the priority queue using the contains method, which requires scanning through the heap and thus takes O(N) time. It then removes the element using remove(element), another O(N) operation due to the need to locate the element within the heap's internal array. Afterward, it adds the element back with the new priority in O(logN) time, but since the earlier steps dominate, the **worst-case runtime is Θ(N).**

OptimizedHeapMinPQ

removeMin

The removeMin method removes the minimum element (root of the heap) by swapping it with the last element and removing it from the data structures, which are O(1) operations. It then calls percolateDown to restore the heap property, which involves moving the swapped element down the tree. Since the heap has a height of logN, percolateDown takes O(logN) time in the worst case, **making the complexity Θ(logN).**

changePriority

The changePriority method updates the priority of an element using O(1) operations by leveraging the elementsToIndex map for direct access. Depending on whether the new priority is less than or greater than the old priority, it calls percolateUp or percolateDown to adjust the element's position in the heap. Both percolation methods have a worst-case time complexity of O(logN) due to the heap's height, resulting in an overall **worst-case time complexity Θ(logN).**

Explain the impact of tree bucket optimization assuming an even distribution of elements across the underlying array. Does the tree bucket optimization help, hurt, or not affect the asymptotic analysis given our assumptions?

The tree bucket optimization has minimal impact on the asymptotic analysis. This is because, with a good hash function and a maintained load factor, the average number of elements per bucket remains constant (i.e., O(1)). Under these conditions, operations like contains, get, and put already have average-case time complexities of O(1), regardless of whether the buckets are implemented as linked lists or red-black trees. Therefore, the tree bucket optimization does not affect the asymptotic analysis given our assumptions; it neither helps nor hurts because it primarily improves performance in scenarios with uneven distribution, which we have assumed away.

Explain how your tests account for the possibility of ties in the remove order since multiple tags can share the same count.
The tests account for the possibility of ties in the remove order by focusing on the counts rather than the exact order of the tags with identical counts. I compared the sequences of counts obtained from both the reference and testing implementations to ensure they match. For tags sharing the same count, I verified that the sets of these tags are identical between both implementations, knowing that their removal order may vary due to ties. This approach ensures that our tests accurately validate the handling of ties without being affected by the arbitrary order of tags with equal counts.

Explain how you upweighted the occurrence of the top 3 most commonly-reported tags.
To upweight the occurrence of the top 3 most commonly reported tags, I first defined a list called topTags containing these tags. Then, when creating the tagPool from which tags are randomly sampled, I added each top tag multiple times based on an upweightFactor. Specifically, for each top tag, I inserted it into the tagPool upweightFactor times, while other tags were added only once. This approach increases the probability of selecting a top tag during random sampling and allows easy adjustment by changing the topTags list and the upweightFactor without extensive code modifications.