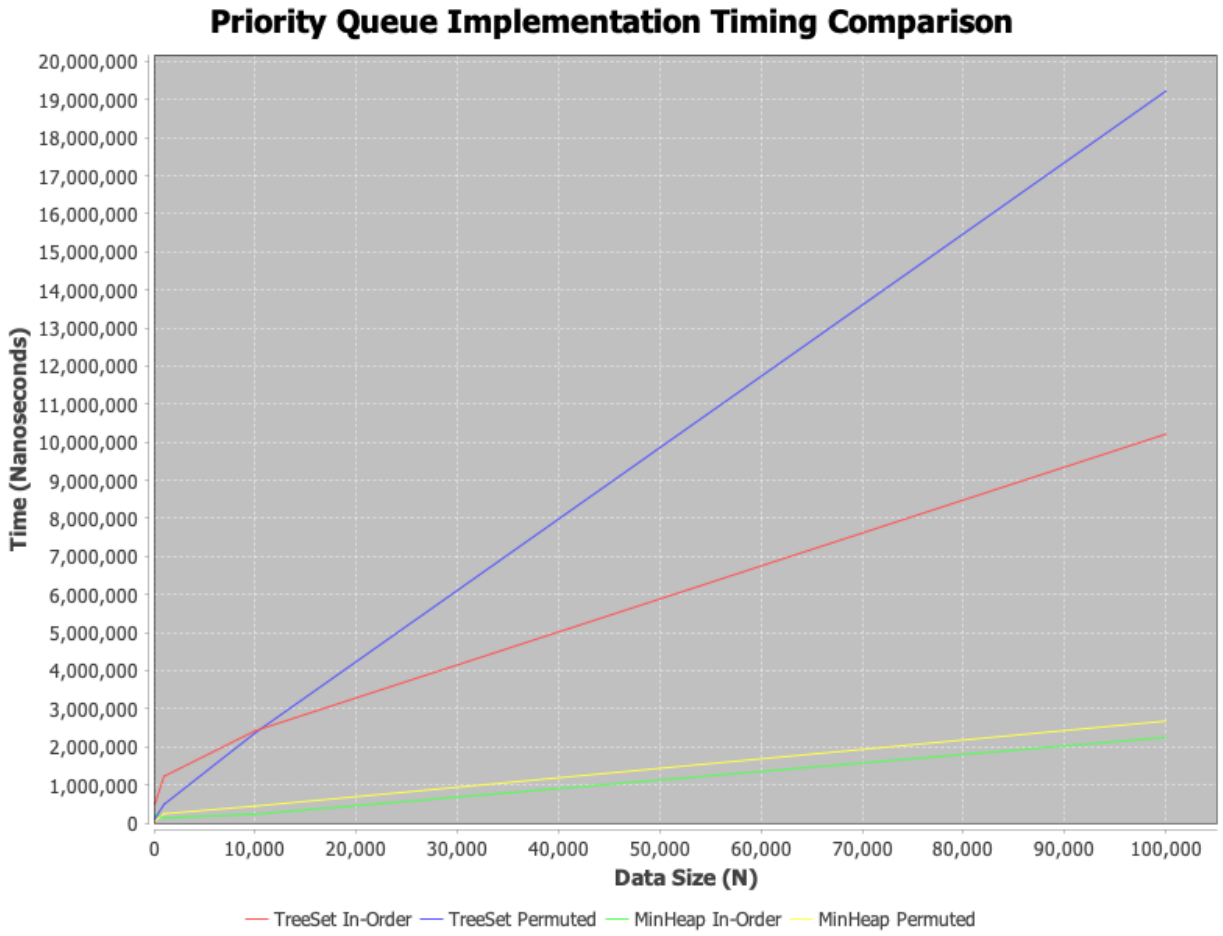
My partner is XiaoHan.

First plot



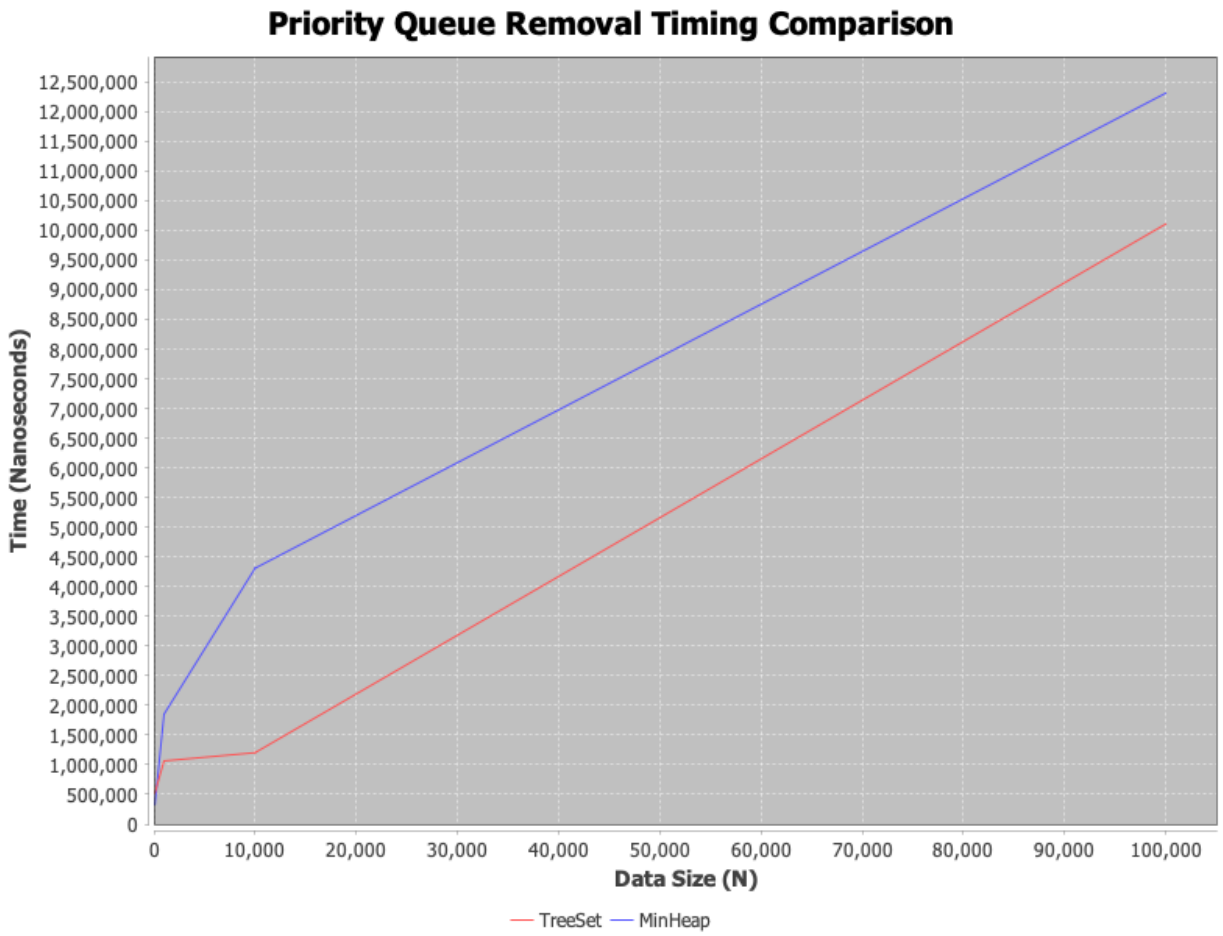
The chart shows the performance comparison of two priority queue implementations: a TreeSet-based queue and a MinHeap-based queue, each tested with in-order and permuted data sets.

- **TreeSet In-Order**: The linear increase in time with data size suggests that operations are likely O(n) for in-order data insertion. This could be because in a balanced tree, inserting in-order data might degrade performance as the tree becomes unbalanced, requiring a re-balance after each insertion.
- **TreeSet Permuted**: The steeper slope compared to the in-order case implies a greater than O(n) complexity, likely O(n log n) due to the cost of rebalancing the tree after each insertion.

- **MinHeap In-Order**: The flat line suggests an O(n) time complexity for constructing a heap from in-order data, which is better than expected. Theoretically, building a heap is O(n log n), but if the data is already in a particular order, it can approach O(n) due to the reduced need for element swaps.
- **MinHeap Permuted**: The slight increase in time with larger data sets could indicate a complexity close to O(n log n), which is expected for heap operations involving unsorted data.

In summary, the MinHeap-based priority queue consistently outperforms the TreeSet-based implementation for both in-order and permuted datasets across all sizes tested. The TreeSet struggles with permuted data due to the overhead of tree rebalancing. The MinHeap's performance is robust with varying data orders, making it a better choice for a priority queue when the data is not guaranteed to be in any particular order.

Second plot



From the graph, we can observe that the TreeSet outperforms the MinHeap in terms of removal time across all data sizes, with a less steep linear increase and a widening performance gap as the data size grows:

- **TreeSet**: Typically implemented as a self-balancing binary search tree (like a Red-Black tree), the expected average time complexity for removal operations is O(log n), because removing an element may require searching for the element and potentially rebalancing the tree afterward.
- **MinHeap**: This is a binary heap where the average time complexity for removing the root element (which is the priority element in a priority queue) is O(log n), due to the need to reheapify the tree after removal. However, if we are considering the removal of arbitrary elements (not the root), this could involve a linear search through the heap to find the element, followed by removal and reheapification, which could result in an average time complexity that approaches O(n).

If the TreeSet's removal time is less than that of the MinHeap as the data size increases, it suggests either an especially efficient implementation or specific characteristics of the data that favor the TreeSet. For instance, if most removal operations in the TreeSet are near the root or leaf nodes, this could reduce the average time complexity. Conversely, if the MinHeap's removals often require significant reheapification or involve removal of non-root elements, this could explain the poorer performance.