	MyVector	MyList	MyHeap	MyAVLTree
Input1	1.0385 milliseconds	20.8995 milliseconds	1.4813 milliseconds	1.6362 milliseconds
Input2	6.2524 milliseconds	374.69 milliseconds	6.3217 milliseconds	7.6791 milliseconds
Input3	52.2557 milliseconds	9808.1 milliseconds	27.3843 milliseconds	37.2416 milliseconds

Vectors:
The time complexity of operations in the code depends on the number of elements in the vector and the specific nature of each operation. For insertion, it takes O(log n) time to find the correct position using binary search, but in the worst case, shifting elements after insertion could take up to O(n) time, resulting in an overall complexity of O(n). Similarly, removal (popMedian) involves shifting elements and also takes O(n) time. The vectorMedian function processes each instruction, either insertion or popMedian, each of which takes O(n) time. Therefore, for m instructions, the overall time complexity is O(mn). Note that 'n' represents the number of elements in the vector, and 'm' represents the number of instructions. The analysis assumes independent operations, but if performed sequentially, their combined time complexity needs consideration, such as O(n) + O(n) = O(n).
This is reflected in how the times are similar in performance with MyHeap and MyAVLTree, which are O(log(n)) but the program fares much worse once the size increases.
Lists:
The time complexity of operations in the code depends on the number of elements in the list and the specific operation involved. For insertion, it takes O(n) time to find the correct position using linear search, while actual insertion into a list is O(1). Consequently, the overall time complexity for insertion is O(n). Removal (popMedian) involves accessing elements by index, taking O(n) time, but removal itself is O(1). The listMedian function processes each instruction, whether insert or popMedian, each requiring O(n) time. Therefore, for m instructions, the overall time complexity is O(mn). Note that 'n' represents the number of elements in the list, and 'm' represents the number of instructions. The analysis assumes independent operations, but if executed sequentially, their combined time complexity needs consideration, such as O(n) + O(n) = O(n).

However, the Vector run time is faster than the List runtime, because of access time. Vectors provide constant-time access to elements at any position. Lists, on the other hand, provide linear-time access because they have to traverse the list from the start to reach the desired position. This means that operations like popMedian, which require access to a specific index (the median), are faster with vectors than with lists.

Heaps:

The time complexity of heap operations depends on the number of elements present. Insertion in a heap takes O(log n) time, as the heap property needs to maintain itself by 'bubbling up' the inserted element to its correct position. Similarly, removal of the median value, known as popMedian, also takes O(log n) time, as the heap property requires 'bubbling down' the last element to maintain order. Rebalancing, ensuring the balance between max_heap_ and min_heap_involves at most one insertion and one removal, hence also O(log n) time. For the heapMedian function, processing each instruction insertion or popMedian takes O(log n) time, leading to an overall time complexity of O(m log n) for m instructions. It's essential to note that 'n' signifies the number of elements in the heap, and 'm' denotes the instructions. The analysis assumes independent operations, yet if sequences occur, their combined time complexity must be considered, such as O(log n) + O(log n) = O(log n), as the O notation denotes an upper bound, simplifying to O(log n).

AVL Trees:

The time complexity of AVL tree operations is tied to the tree's height, which remains logarithmic due to its balanced nature, resulting in O(log n) complexity. Insertion and removal operations take O(log n) time, with additional O(log n) time needed for rebalancing if required. Finding the maximum value in the tree also takes O(log n) time. Rebalancing operations, including balance checking and rotations, are constant time, O(1). For the treeMedian function, processing each instruction - insertion, removal, or findMax incurs O(log n) time, resulting in an overall time complexity of O(m log n) for m instructions. Notably, the analysis assumes independence of operations; however, if sequences of operations are considered, their combined time complexity should be accounted for, such as O(log n) + O(1) for insert followed by rebalance resulting in O(log n) overall.

In the early stages of my project, I decided to implement my own AVL tree from scratch.. However, I quickly ran into a significant issue that took me some time to resolve.

The problem arose from not properly tracking the size of the trees. In an AVL tree, the balance factor of each node (the difference in heights between the left and right subtrees) needs to be -1, 0, or 1. If the balance factor falls outside this range, rotations are needed to restore the balance. To calculate the balance factor, I needed to keep track of the height of the subtrees, which indirectly meant tracking the size of the trees.

Initially, I overlooked the importance of accurately tracking the size. I had assumed that the rotations alone would take care of maintaining the AVL tree properties. However, I soon realized that without accurate size tracking, my tree was not balancing correctly. Nodes were not where they were supposed to be, lookups were taking longer than expected, and I ran into an issue where the tree would report as empty when I still had not finished reporting the medians. Or, it would report too many medians.