These numbers are the averages of three executions of each input file rounded to the nearest whole number, all raw data is at the end of the report. All computation was done on the Linux machine.

	Vector implementation	Linked list implementation	Heap implementation	AVL tree implementation
input1.txt	532 μs	9109 μs	234 μs	10030 μs
Input2.txt	5661 μs	217369 μs	757 μs	277495 μs
Input3.txt	86334 μs	7269102 μs	3533 μs	8984802 μs

vectorMedian

Insertion into the sorted vector is executed using std::upper_bound, taking O(log n) time per insertion, where n represents the current size of the vector. With k insertions, the total insertion complexity becomes O(k log n). Subsequently, finding and removing medians from the sorted vector involves O(n) time per operation in the worst case, with at most k removal operations, leading to a total complexity of O(k O(n)). Thus, the overall time complexity of the vectorMedian function is O(k * n) in its simplest form, where k is the number of instructions provided.

listMedian

Inserting into the sorted list requires finding the correct position, typically taking O(n) time where n is the current size of the list, though, on average, this can be considered O(n/2). Similarly, finding and removing the median, which involves traversing half of the list elements, has a time complexity of O(n). Consequently, for a sequence of k instructions, the total time complexity is O(k * n), where k is the number of instructions and n is the size of the list at any given point. Therefore, the overall time complexity of the algorithm is O(k * n).

heapMedian

It iterates through each instruction in the instructions vector, resulting in a time complexity of O(k), where k is the number of instructions. Within each iteration, the function performs operations such as pushing, popping, and balancing the priority queues, each of which has a time complexity of O(log n), where n is the size of the priority queue. As a result, the overall time complexity of the algorithm is O(k log n).

treeMedian

Each insertion or removal operation in AVL trees takes O(log n) time, where n is the number of elements in the tree. The function iterates through each instruction in the input vector, leading to a time complexity of O(k log n), where k represents the number of instructions. Rebalancing AVL trees in the worst case also takes O(log n) time per operation. Additionally, calculating and storing medians has a time complexity of O(k). Combining all these factors, the overall average-case time complexity of the function is O(k log n).

Analyzing the algorithms for finding the median vectorMedian, listMedian, heapMedian, and treeMedian reveals trade-offs between speed and memory usage. When dealing with large datasets, vectorMedian and heapMedian shine due to their impressive time to run. However, heapMedian takes the crown for its superior performance due to its more efficient runtime of O(k log n). This is because heap operations have lower inherent costs compared to manipulating vectors. On the other hand, listMedian and treeMedian come with a higher time to run with complexities of O(k * n) and O(k log n) respectively. ListMedian suffers from inefficiencies due to linear search and deletion within a linked list. While TreeMedian offers efficiency gains over listMedian, maintaining balanced AVL trees introduces some overhead. Additionally, frequent calls to the size function within conditional statements might negate some of the AVL tree's performance benefits, as determining the size has a time complexity of O(n).

While all four algorithms use additional space proportional to the input size (O(n)), runtime becomes increasingly important for larger datasets. This is especially true for algorithms with higher time complexities, where the slowdown becomes more significant as the data grows. Therefore, the best choice depends on a careful evaluation of not just time and space complexity, but also how runtime behaves as the input size increases. In ascending order of runtime for growing datasets, we have: heapMedian, vectorMedian, and listMedian, with treeMedian being the slowest of the four.

Initially, I expected significant differences between implementations, particularly when comparing vectorMedian and listMedian. However, their core structures were surprisingly similar. Likewise, heapMedian and treeMedian both utilize two underlying containers, exhibiting a similar design pattern. Furthermore, I was surprised by how few helper functions were necessary – only two for the AVL tree class. Exploring the STL offered valuable insights into the capabilities of base container classes like vectors, lists, and heaps. The most unexpected finding was the dramatic difference in runtime between implementations as input sizes increased. Despite all having similar time complexity with vectorMedian and listMedian having O(k * n), and heapMedian and treeMedian having O(k log n) time complexity, their execution times diverged considerably with larger inputs. This finding highlights the importance of considering implementation details beyond just theoretical time complexity. Furthermore, execution times varied significantly across machines with different architectures.

Raw results

input1.txt	Vector implementation	Linked list implementation	Heap implementation	AVL tree implementation
Attempt one	555 μs	9168 μs	233 μs	10123 μs
Attempt two	496 μs	8896 μs	235 μs	9807 μs
Attempt three	544 μs	9264 μs	234 μs	10160 μs

Input2.txt	Vector implementation	Linked list implementation	Heap implementation	AVL tree implementation
Attempt one	5638 μs	213804 μs	757 μs	280819 μs
Attempt two	5680 μs	205409 μs	757 μs	273593 μs
Attempt three	5665 μs	232893 μs	758 μs	278074 μs

Input3.txt	Vector implementation	Linked list implementation	Heap implementation	AVL tree implementation
Attempt one	84001 μs	7340897 μs	3539 μs	9035226 μs
Attempt two	93656 μs	7136742 μs	3531 μs	8894266 μs
Attempt three	81346 μs	7329666 μs	3528 μs	9024914 μs