	4000 Instructions	16,000 Instructions	64,000 Instructions
List	56.9013ms	488.071ms	8288.5ms
Vector	2.86917ms	10.8959ms	69.5545ms
Heap	2.19596ms	7.012ms	30.6073ms
AVL Tree	2.01892ms	9.64267ms	46.1398ms

List

- The algorithm complexity for insertion in a list is O(n), due to the requirement of finding the correct position, and then O(1) to actually insert the data.
- Finding the median of the list is also O(n) since list traversal is required in order to find the middle element (by value)
- Since lists do not support random-access, operations that require specific elements require traversing through the list, which falls under O(n) complexity.
- Despite this, the growth factor between instruction count is not linear, in fact, the time grows from roughly 4x to 16x, which is similar to $O(n^2)$

Vector

- Insertion in a sorted vector will be O(n) in the worst case, meaning that elements would need to be shifted to make room for the new element. Best case it may happen in O(log n), but shifting elements may be necessary still.
- Once the vector is sorted however, the median can be found in O(1), this is because vectors support random access meaning traversal is not necessary.
- Between 4000 and 16,000 instructions, the time increased 4x, and from 16 to 64 it was about 6.5x. This does not reflect O(n) or O(log n).

Heap

- Insertion in a priority queue heap is O(log n) since the element has to be in the correct place in order to maintain the heap's properties.
- Traditionally it is not a straightforward process to find the median in a heap since it is not always fully sorted, but since we used two heaps, it should be O(1) since we only need to look at the top or bottom of one or both heaps. Adjusting heaps after removal could be O(log n) though.
- Since a heap follows a binary tree structure, each node can be smaller or larger than their children, but that does not necessarily make it in proper order.
- Between the 3 instruction counts, the growth goes from roughly 3.2x to 4.4x, which is greater than linear, but less than quadratic, which is about where O(log n) falls in terms of growth factor.

AVL Tree

- Generally all operations are O(log n), this is because there is a strict balance through the heights of child subtrees not differing by more than one. If this ever occurs, the tree is automatically rebalanced to maintain itself.
- Between instruction counts, the time grew from 4.77x to 4.79, this should reflect the O(log n) nature of the AVL tree, since the growth does not follow a linear pattern, but is also not as drastic as an exponential complexity. The growth is positive, but at a slight increment, which is how log algorithms are expected to grow.