Nick Szul
Professor Tojeira
CSCI 33500
10 April 2024

Project 2 Report

1.)

Implementation	input1.txt(4000 instructions)	input2.txt(16000 instructions)	input3.txt(64000)
Vector	1879 microseconds	12107 microseconds	116139 microseconds
List	60938 microseconds	1007344 microseconds	34431816 microseconds
Heap	2859 microseconds	9422 microseconds	36648 microseconds
AVL Tree	2888 microseconds	12310 microseconds	58991 microseconds

2.)

Vector Method:

For the vectorMedian function, we have the initial loop, which has a time complexity of O(n), with n being the amount of instructions. For each iteration, we are either going to be inserting or erasing an element from the vector. Both of these functions operate on linear time, so their time complexity is O(n). This would make the time complexity of the vectorMedian Function O(n^2).

List Method:

For the listMedian function, we still have the initial loop with a time complexity of O(n), However, we use nested loops to reach the median item. Because of this, our time complexity for the listMedian function is O(n^2) since for each iteration, we will go through another loop.

Heap Method:

For the heapMedian function, again we have the initial loop with a time complexity of O(n). For inserting and removing from a Heap, the time complexity is O(logn). Since for each iteration we will either insert or remove an item, the time complexity of heapMedian is O(nlogn).

AVL Tree Method:

Similar to the heapMedian function, inserting and removing from an AVL Tree has a time complexity of O(logn). With the same loop for instructions, it has the same time complexity of O(nlogn).

3.)

Generally, my results seem to line up with the time complexities of each function. However, some functions were much closer to the worst case than others. As expected, the Heap and AVL Tree methods were the most efficient, while the Vector and List methods were significantly less efficient. For Vector and List, which both had a time complexity of O(n^2), listMedian comes much closer to the worst case than vectorMedian. This is probably because each element in a list is a node and nodes require extra space. Although the vectorMedian function doesn't have this issue, it still suffers in efficiency because of the linear time complexity of the insert and erase functions, but is much better than the worst case. The heapMedian and treeMedian are both efficient, but heapMedian runs fastest because all you have to do is access the top node of both heaps. In TreeMedian, you have to find the rightmost/leftmost node, which requires some traversal.

4.) Through implementing the methods in this project, I feel like I now have a much better grasp on the strengths and weaknesses of these data structures and their efficiencies. Now that I better understand the time complexities of these implementations better, it makes solving these tasks in the most efficient manner a lot easier. Before this project, I wouldn't have really known how to approach something like outputting the medians in the most efficient way, but after the implementation I began to think about things such as the time complexity and how certain implementations can increase the runtime. With the heap implementation, I noticed how small things like using a data structure that would store the median at the root saves us time, and I will keep things like this in mind for the future.