Fardeen Bhuiyan

Class: CSCI 335 Software Analysis and Design III

Professor: Justin Tojeira

Project 2 Methods Analysis Report

Pop-Median Runtimes (in microseconds) for input1.txt File

	vectorMedian	**listMedian**	**heapMedian**	**treeMedian**
Trial 1	4554	50419	4530	
Trial 2	4674	47612	4022	
Trial 3	4164	49164	4052	
Average	4464	49065	4201	

Pop-Median Runtimes (in microseconds) for input2.txt File

	vectorMedian	**listMedian**	**heapMedian**	**treeMedian**
Trial 1	15907	481572	13281	
Trial 2	15903	467156	12461	
Trial 3	14666	480792	12428	
Average	15492	476507	12723	

Pop-Median Runtimes (in microseconds) for input3.txt File

	vectorMedian	**listMedian**	**heapMedian**	**treeMedian**
Trial 1	69830	7688717	31535	
Trial 2	69802	7707510	30976	
Trial 3	70496	7700417	30996	
Average	70043	7698881	31169	

For void vectorMedian(const std::vector<int> * instructions), the O-complexity is **O(n^2) quadratic.**
The complexity of the overarching for-loop itself is O(n) complexity. When it calls upon the insertVector function, this function utilizes std::lower_bound to find the first integer that the vector already has that is greater than the integer being inserted, and this itself is an O(log n) complexity due to it using binary search. However, in the insertVector function, the STL insert() function is used to actually insert the new integer into the vector. This essentially makes insert Vector have a time complexity of O(n).
When we check inside the for loop for vectorMedian(...), vectorSTL.erase() is used in order to remove the median from the vector. erase() is an STL function that runs on O(n) complexity unless we are erasing at the end, because we are shifting elements after erasing one element.
So utilizing the complexity of the overarching for-loop along with its internal erase() call for each element that is not equal to -1, the calculation becomes O(n) * O(n), which results in vectorMedian(...) to have an O-complexity of **O(n^2) quadratic time.**
The results have lined up with this complexity. As the file's amount of inputs increases, the runtime increases roughly quadratically. From 4000 to 16000 to 64000 inputs, the data averages around 4400 to 15500 to 70000 and so the results are very close to squared in terms of rate of growth.

For void listMedian(const std::vector<int> * instructions), the O-complexity is **O(n^2) quadratic.**
The complexity of the overarching for-loop is O(n) complexity. When we use std::next(), we are inside the for-loop and we traverse the list to find the middle of the list which also becomes O(n) complexity. These are the main factors the lead to the overall calculation of O(n) * O(n) to get **O(n^2) quadratic time.**
Maybe I looked over something and miscalculated the complexity for listMedian(...) as it performed severely worse than the other methods for each file. The complexity also looks greater than linear, but less than quadratic, but perhaps it may still count as O(n^2). If this is the case, perhaps lists fare worse in regards to popping the median due to constant traversing.

For void heapMedian(const std::vector<int> * instructions), the O-complexity is **O(n log n).**
The complexity of the overarching for-loop is O(n) complexity. Within the loop, we utilize the STL top() and pop() functions. top() has a complexity of O(1), but pop() requires heapification. When you remove an element in a heap, this is not like an ordinary list because you need to balance the other elements. In a max heap, the largest integer would be at the root, so if you pop the top, you need to send the second largest integer to the root. In a min heap, the smallest integer would be at the root, so if you pop the top, you would need to send the second smallest integer to the top. This involves O(log n) complexity. Since these are the primary components of the method, the entire process calculation would be O(n) * O(log n) to get **O(n log n) complexity.**
The results seem to line up with this complexity because O(n log n) serves best when dealing with larger inputs. As was shown, the larger the file became, the better the heap method performed in comparison to the lists and vector methods.

As I was collecting the data, I had assumed that the heap method and vector method would have very similar data throughout and thus the similar complexity (before I calculated the complexities) because file 1 had shown them with very similar estimates. However, the moment I observed the runtimes for file 2, I realized that heaps are efficient for larger inputs and that was confirmed in file 3's data. It was very interesting to see the efficiency differences even whilst collecting the data and it reinforces why certain data structures must be used in specific circumstances.