Olti Gjoni CSCI-335

Project 2 Report

1) Timing Results:

Implementation	input1.txt	input2.txt	input3.txt
vectorMedian	44450 µs	52020 µs	53131 µs
listMedian	4169125 µs	4276764 µs	4223290 µs
heapMedian	2630 µs	2837 µs	2613 µs
treeMedian	12748 µs	No output	12881 µs

2) Complexity Analysis:

vectorMedian: Inserting an element into a sorted vector requires O(n) time in the worst case, where n is the number of elements in the vector. Popping the median from a sorted vector also takes O(n) time since it involves removing an element from the middle of the vector.

listMedian: Inserting an element into a sorted list takes O(n) time in the worst case, where n is the number of elements in the list. Popping the median from a sorted list also takes O(n) time since it involves removing an element from the middle of the list.

heapMedian: Inserting an element into a heap takes O(log n) time in the worst case, where n is the number of elements in the heap. Popping the median from a heap also takes O(log n) time since it involves removing the root element and then rebalancing the heap.

AVLtree: Inserting an element into an AVL tree takes O(log n) time in the worst case, where n is the number of elements in the tree. Popping the median from an AVL tree also takes O(log n) time since it involves finding and removing the median element while maintaining the AVL tree properties.

3) Results vs. Complexity:

The timing results generally align with the expected complexities of the implementations. Both vectorMedian and listMedian exhibit O(n) time complexity for each input, as they involve linear operations for insertion and popping. The heapMedian implementation shows better performance, which aligns with its O(log n) time complexity for both insertion and popping. The AVLtree implementation also demonstrates reasonable performance consistent with its O(log n) time complexity for insertion and popping operations.

4) Additional Observations:

The heapMedian implementation outperforms the others, especially for larger inputs.

- This highlights the efficiency of heap-based data structures for median tracking.

AVL trees offer a balance between efficient insertion and retrieval operations.

- This makes them suitable for scenarios where dynamic updates are frequent.

List-based implementations exhibit poor performance compared to the others.

- This is due to the linear time complexity of insertion and retrieval operations in sorted lists.

The choice of data structure depends on the specific requirements of the application, such as the frequency of updates and the size of the dataset.

Additionally, I wanted to note that for the analysis presented in this report, I decided to send the outputs from each input file into separate output.txt files. This approach allowed for easier viewing of all the outputs for each method.