CSCI 33500
Project 2
MINGJUN LU
(1)

Time is microseconds	time of vector	time of list	time of heap	time of AVLtree
methods-> input(below)	vector Median	list Median	Heap Median	Tree Median
test input 400	55.2	1,119.99	54.298	492.415
input1 4000	597.502	31,450.1	247.14	1,036.26
input2 16000	5,698	4.08672e+06	977.047	7,562.97
input3 64000	110,367	1.70425e+08	12,123	36,004.1

According to the table, It can be seen that Vector is the fastest when processing the median, followed by Heap, third by AVLtree, and last is List. List takes a particularly long time to use when dealing with larger databases.

In the vectorMedian function, a traversal operation is performed to read the m instructions in instructions. For each instruction, std::lower_bound is used to find the insertion position in the ordered vector, and then the element is inserted at that position. The time complexity of this insertion operation is O(log n), where n is the current size of the vector vec. If the instruction is -1, the median is calculated and removed. First, the position of the median is calculated using a constant-time operation, and then the element at that position is removed from the vector. The time complexity of this removal operation is also O(n) due to the efficient removal from an ordered vector. The overall time complexity of the entire loop is mainly determined by the insertion operations and the calculation and removal of medians. Since each insertion and removal operation is O(n), and there are m such operations in the loop, the time complexity of the entire loop is O(n).

The listMedian function traverses an instructions vector of size N, resulting in a time complexity of O(N) due to the linear iteration. Each insertion operation involves sorting the list, which contributes O(N log N) to the overall time complexity. Additionally, the median calculation and removal operations have a time complexity of O(N/2) in the worst case, as finding the median index involves traversing half of the list. This results in an overall time complexity of O(N^2 log N) for the listMedian function.

The heapMedian function begins its operation by traversing through each element in the instructions vector, resulting in a time complexity of O(N), where N denotes the number of instructions. Subsequently, the insertion and deletion operations within the heaps, facilitated by std::priority_queue::push and std::priority_queue::pop respectively, have a time complexity of O(log N) per operation. These complexities are influenced by the size of the heap, with N representing the heap's size.

Moreover, the function involves rebalancing the heaps, which also incurs a time complexity of O(log N) per rebalancing operation. Combining these complexities, the heapMedian function demonstrates an overall time complexity of O(N log N)

AVL trees are self-balancing binary search trees that maintain balance during node insertion and deletion, ensuring an average time complexity of O(log n) for search operations. This efficiency is achieved through rotations and adjustments that keep the tree's height logarithmic relative to the number of nodes.

When dealing with two AVL trees simultaneously, such as smallTree and largeTree, additional time complexities arise during rebalancing. Insertion and deletion operations in AVL trees have a time complexity of O(log n), involving traversal and rotations for balance restoration. Rebalancing due to imbalance caused by insertions or deletions adds another O(log n) factor to the time complexity. Considering these complexities, the overall time complexity for managing two AVL trees and ensuring their balance during common operations remains O(log n).

Through observation, it's evident that list operations such as insertion, deletion, and sorting can be costly, leading to slower performance compared to other data structures. List structures require traversals for insertions and deletions, and sorting operations add to the time complexity. On the other hand, vectors provide efficient insertion and access due to their contiguous memory allocation, resulting in lower time complexity for these operations. As a result, vectors generally outperform lists in scenarios involving frequent insertions and deletions of elements.

Heap's insertion and deletion operations involve adjusting only specific nodes and don't require global balancing operations. As a result, when it comes to tasks like insertion, deletion, and finding the median, Heap outperforms AVL trees in terms of performance.