DATA STRUCTURES AND ALGHORITHMS

REPORTS OF TIME MEASUREMENTS

Ahmet Enes YILMAZ
ISTANBUL TECHINCAL UNIVERSITY
Computer Engineering Student

aenesywork@gmail.com
LinkedIn

INTRODUCTION

This report analyzes the execution times measured from the code solutions used for solving the assigned problems. The measurements were conducted using the time.h library. All datasets were shuffled, and the operation IDs range from 1 to 500,000.

GROUP 1

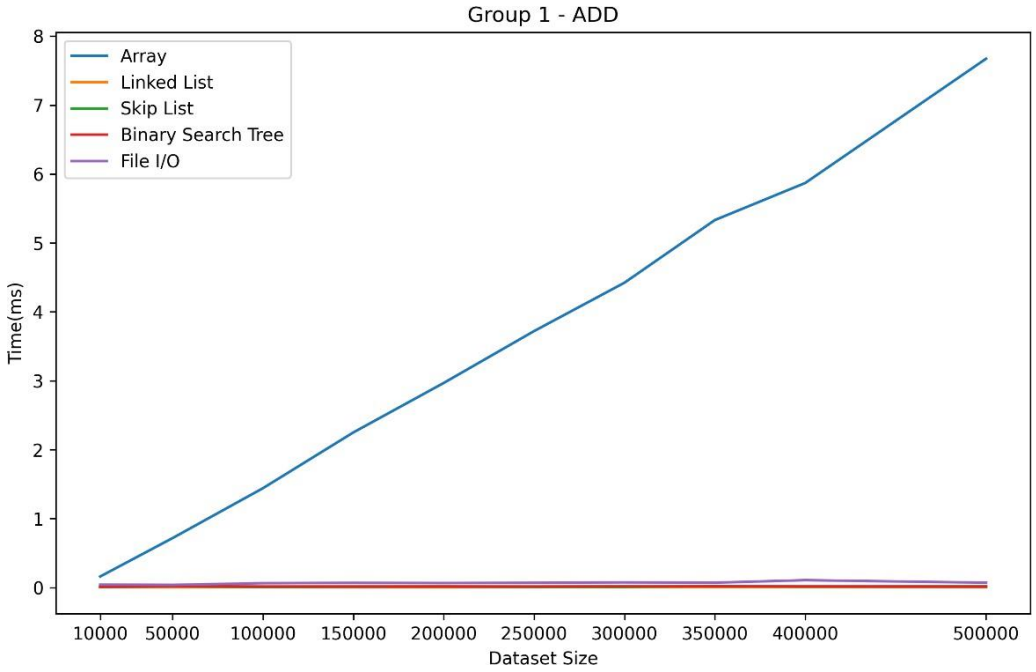

Figure 1

Figure 1 shows that the Dynamic Array solution took a significant amount of time for the Add operation due to its O(N) time complexity. In contrast, Skip List and BST, which have logarithmic time complexity (O(logN)), performed much better. In my Linked List solution, the presence of a tail pointer allowed for O(1) time complexity when adding the last employee. For the file solution, I used the String Stream library to quickly locate the end of the file, enabling a fast addition of an employee. (Figure 1)

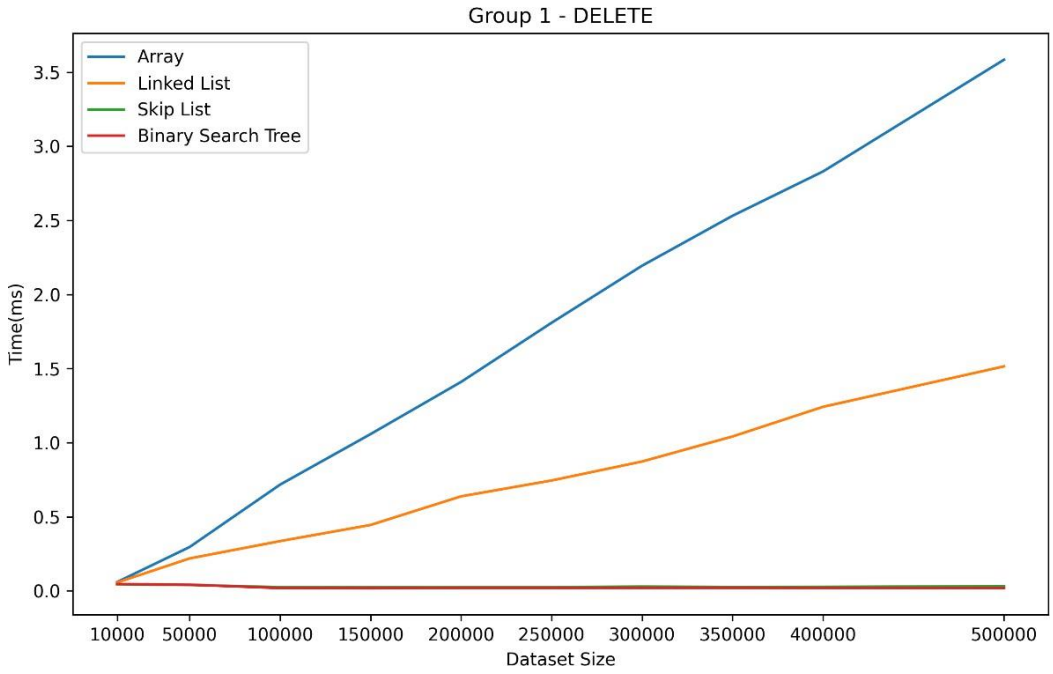

Figure 2

Array and Linked List have O(N) time complexity for deletion since they are linearly searching the ID's. In Skip List and BST, deletion time complexity had O(logN), since Skip List has layers and BST is in a tree form. (Figure 2)

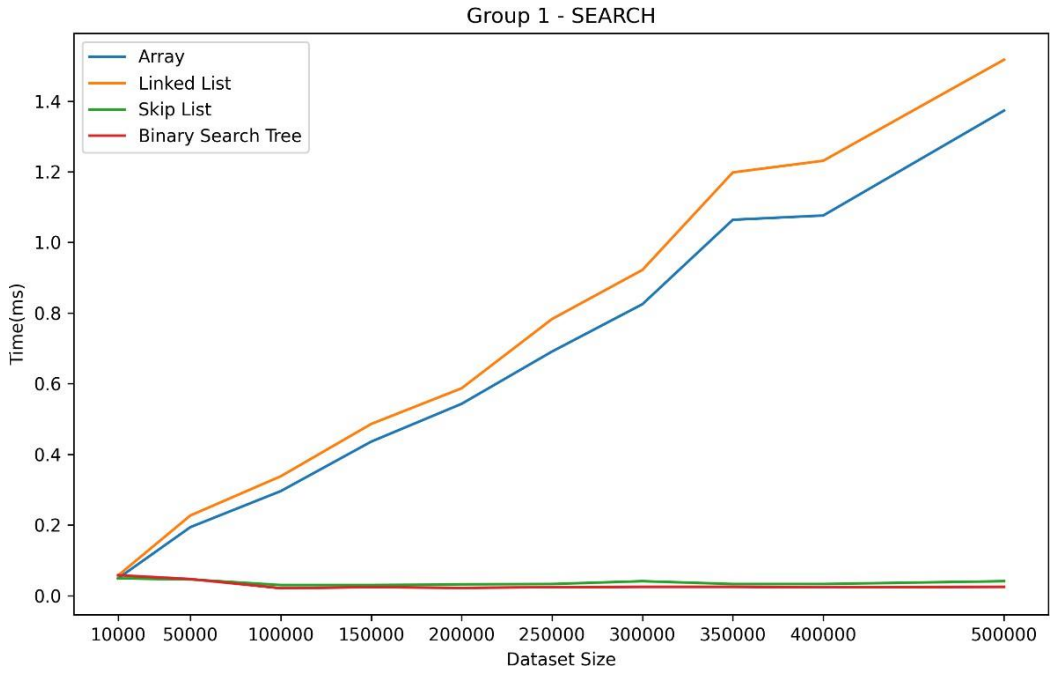

Figure 3

When searching for an employee in Array and Linked List, we have to use a linear search algorithm, which reduces performance for large datasets. Performance would be better if we used Skip List or BST, because they have logarithmic O(logN) time complexity. When searching for an ID in BST, we compare it with the root's ID. If it is greater than the root's ID, we search the right subtree. If it is smaller than the root's ID, we search the left subtree. In the worst case, we check one node for every level in the BST (which is equal to the height). The Skip List achieves logarithmic time complexity by maintaining multiple layers of linked lists, where each layer skips over a certain number of elements. During the search, the algorithm navigates through the layers to find the searching ID, effectively reducing the search space in each step. (Figure 3)

GROUP 2

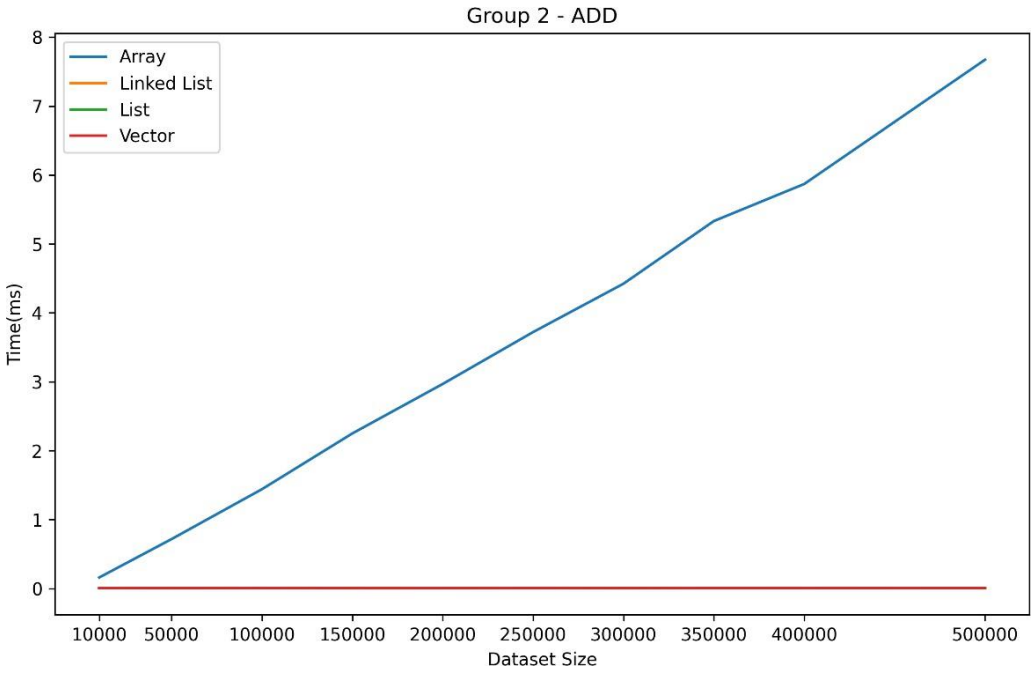

Figure 4

Since Arrays have O(N) time complexity, it took a significantly longer time compared to others. Since we do not know where the dynamic array ends, we search linearly, causing O(N) time complexity. For the other solutions, the Add operation has O(1) time complexity. However, if we did not have a tail for the Linked List, that would also be O(N). It appears that Array is not in a good position among Group 2. (Figure 4)

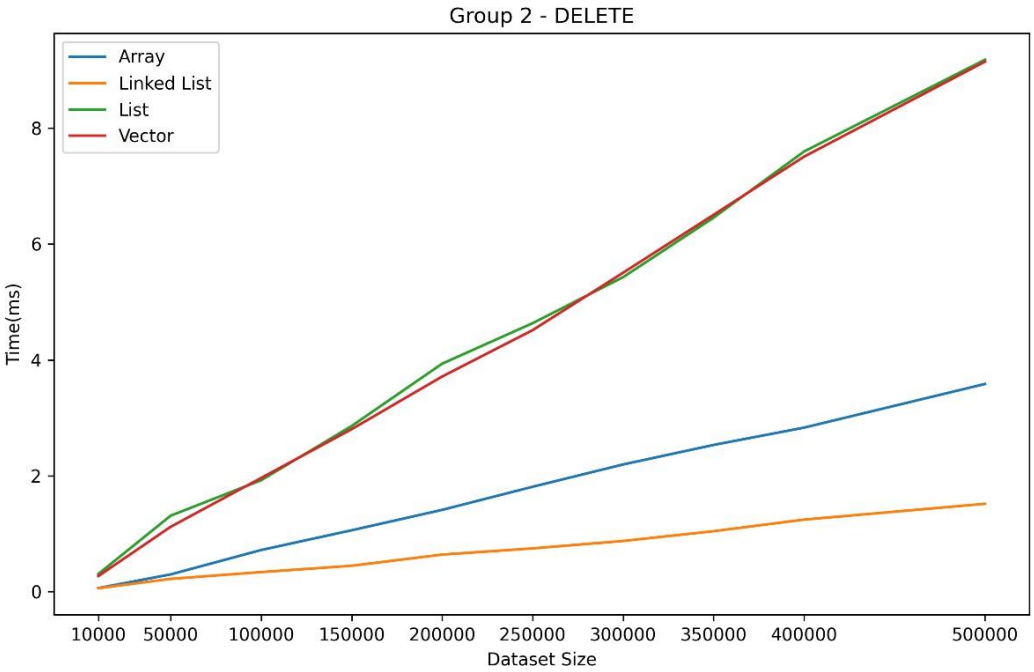

Figure 5

According to the graph, Group 2 has no good solution for Deletion. They all have O(N) time complexity, which significantly reduces performance. We are linearly searching for the target ID, and it takes massive time. (Figure 5)

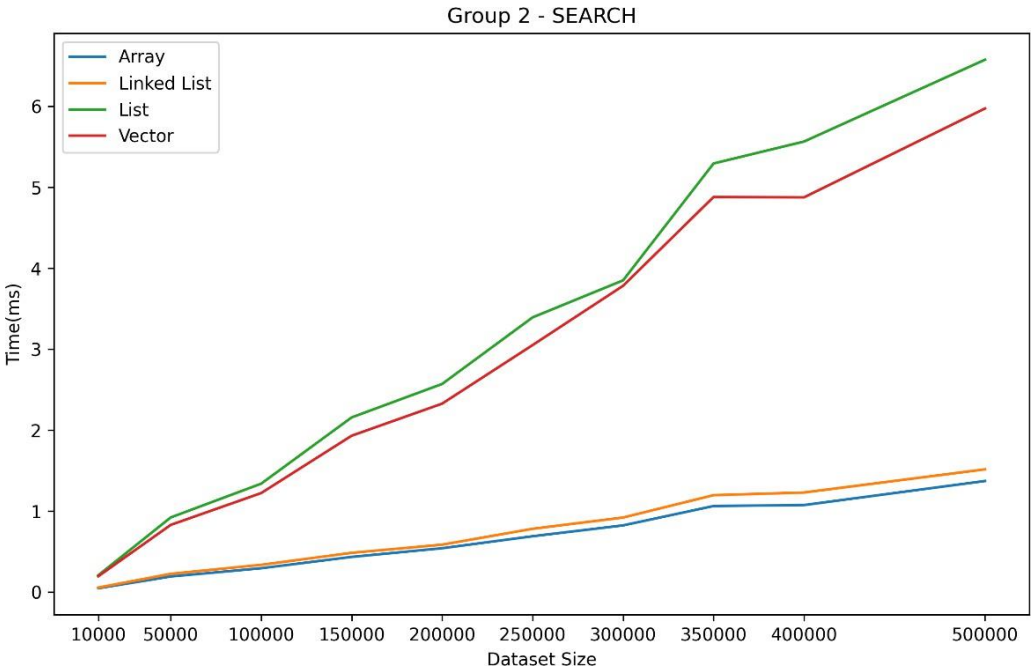

Figure 6

For the same reasons mentioned for GROUP 2 – DELETE, these data structures have O(N) time complexity for searching, and they are not useful for get by ID methods. Linear search is not a good choice for deletion. (Figure 6)

GROUP 3

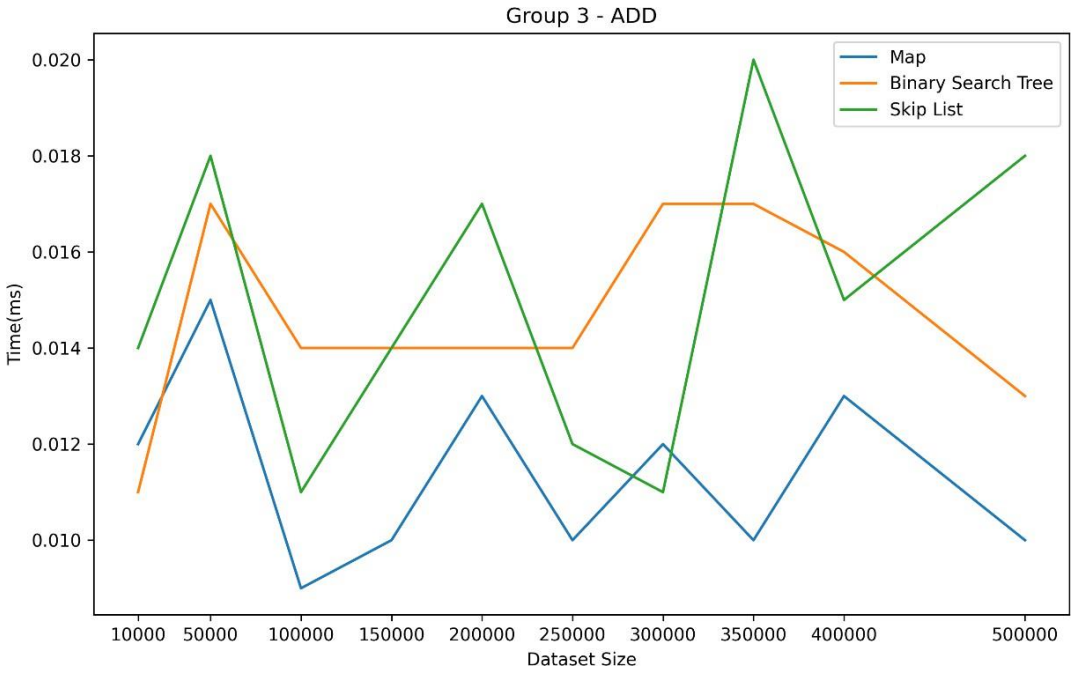

Figure 7

In this graph, we can observe that the performance results for Map, BST, and Skip List are very good. This is because their time complexity is O(logN). For BST and Map, the complexity is O(logN) because their elements are contained in a tree structure. Every time you examine a node of the tree, you determine if the element you're trying to find/insert is less than or greater than the node. However, Map is performing better because it implements a self-balancing BST (Red-Black Tree). Therefore, using a map seems to be a good way to handle large datasets. However, the map (Red-Black Tree) should balance itself after each insertion.

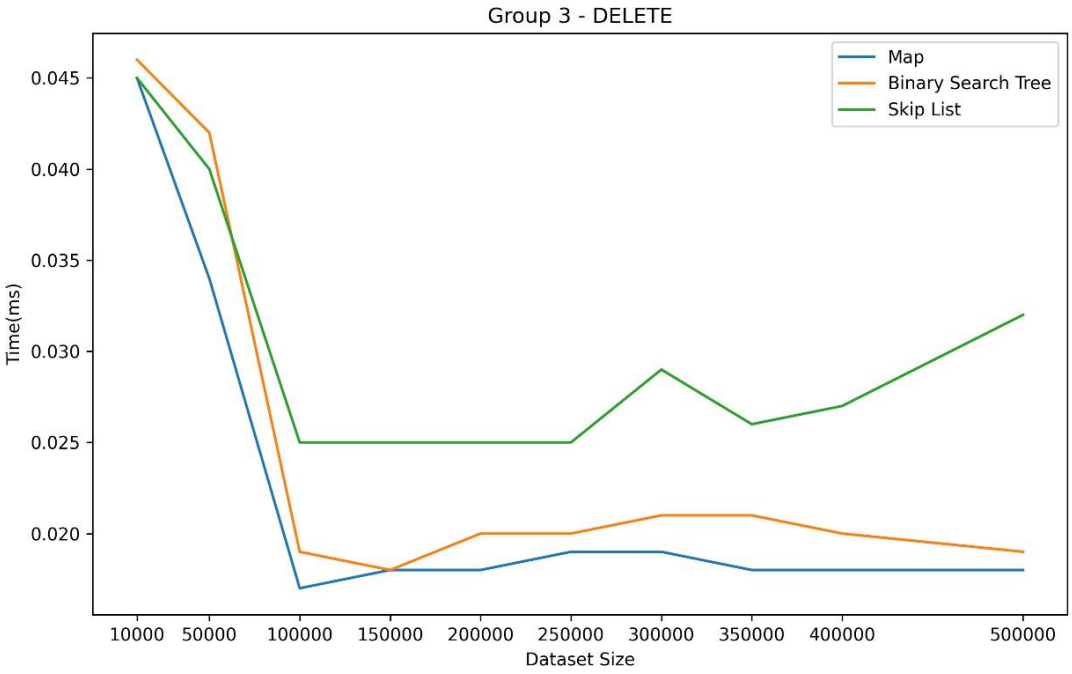

Figure 8

The results show that Group 3 also has good solutions for the Delete operation. They all have O(logN) time complexity, and the results are very close to each other. These results are better than the other groups. (Figure 8)

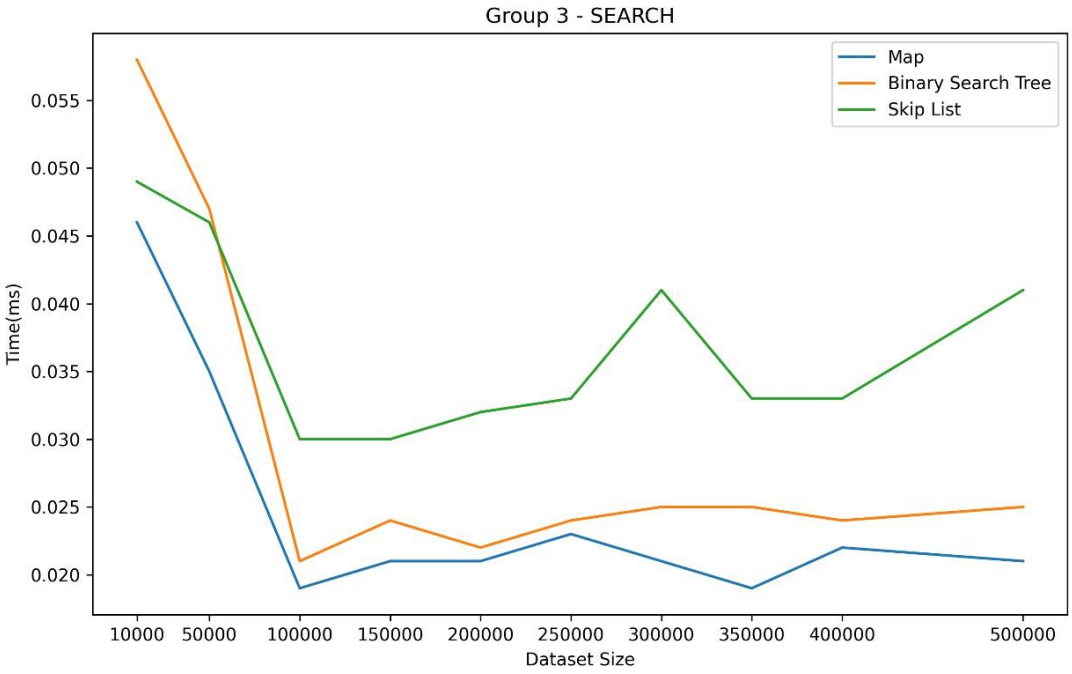

Figure 9

Since these data structures have O(logN) time complexity, it is easier to find employees by ID. Considering all the results, Group 3 has the best performance. (Figure 9)