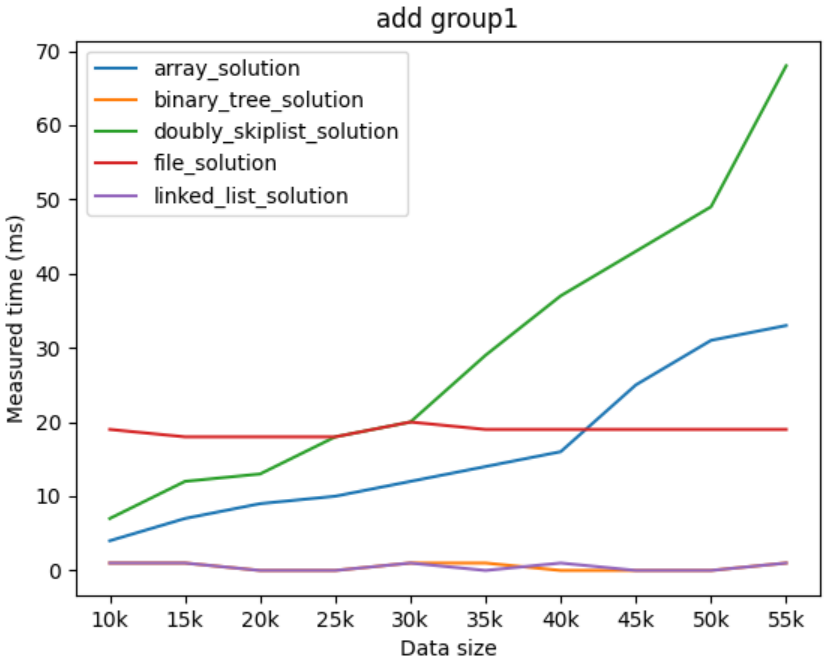
BLG 223E Homework 5 – Report

Name: Kemal Tahir Bıcılıoğlu Id: 150210083

- I generated my own datasets and operations from 10k to 55k which are randomly generated and not sorted with generate.cpp. I used python for plotting.



Array solution: Add function in the array solution has O(n) running time complexity since first it copies the elements to the new array and then adds the new id to the correct place which is the end of the array because all id's are unique and new id is determined to be last id + 1. As a result, we can see that the graph shows the linearity of the complexity of the add function for the array solution.

Binary tree solution: Add function in the binary tree solution has O(log n) running time complexity since we traverse from the head of the binary tree and check if the current id is less or greater than the new id and proceed, we get rid of the approximately half of the nodes. So that our complexity here is log n. Consequently, the graph shows the logarithmic behavior of the add function for the binary tree solution. Since the datasets are small log n behavior is not being observed properly but it is log n.

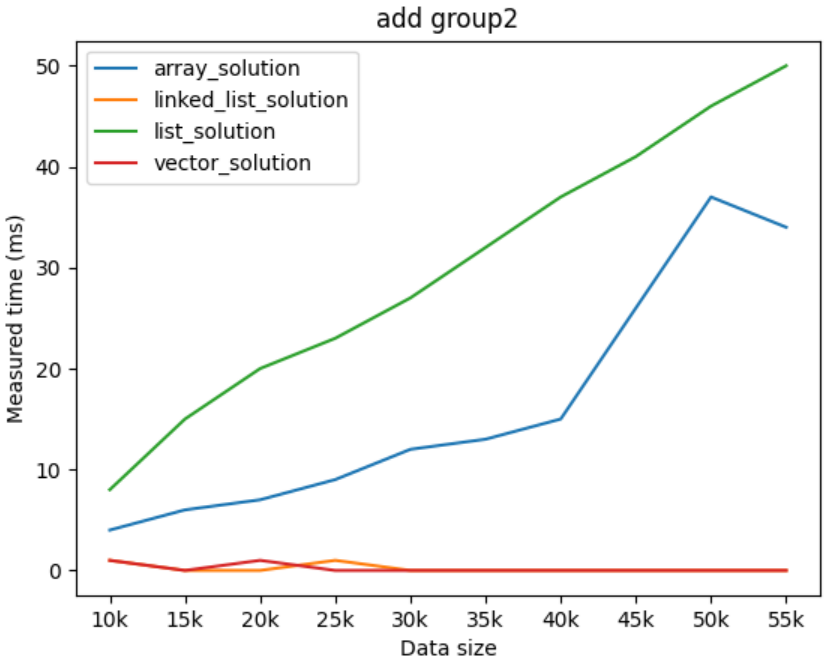
Doubly skip list solution: Add function in the skip list solution has O(log n) running time complexity. Skip list is a probabilistic data structure, and it approaches to log n time complexity for the add operation. Because skip list skips over the elements that are not relevant to the search, it does not traverse all the elements in the skip list and runs at O(log n). My skip list implementation seems not to have a log n time

complexity for adding operation as we can see from the graph. However, it is a probabilistic data structure, and the inputs are chosen to be max 100k and the order of the id's can make difference. We should consider these facts, but the graph shows that my skip list runs at O(n) time complexity for adding operation.

File solution: Add function in the file solution has O(1) running time complexity. Since we are guaranteed that the new id is always the last id + 1 for adding operation, we can just open the file in append mode and add the new employee to the end of the file which takes constant time. So, the graph shows the constant time behavior of the add function for the file solution.

Linked list solution: Add function in the linked list solution has O(1) running time complexity since the new id is always the last id + 1 for adding operation, we can just keep tail for the linked list and add the new id to the tail's next and make the new id as tail which takes constant time. As a result, we can see that the graph shows the constant time behavior of the add function for the linked list solution.

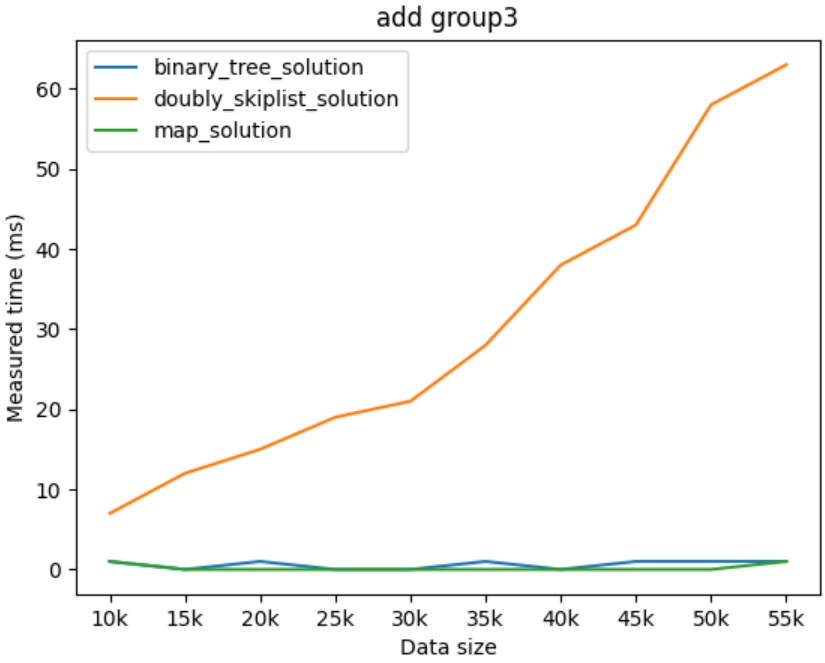
To sum up, we can interpret the graph as comparing the time complexities, linked list and the binary tree solution seems like the best data structure from all the data structures above.



Behavior of the array solution and the linked list solution for add operation explained previously. We can see the same behavior here as well.

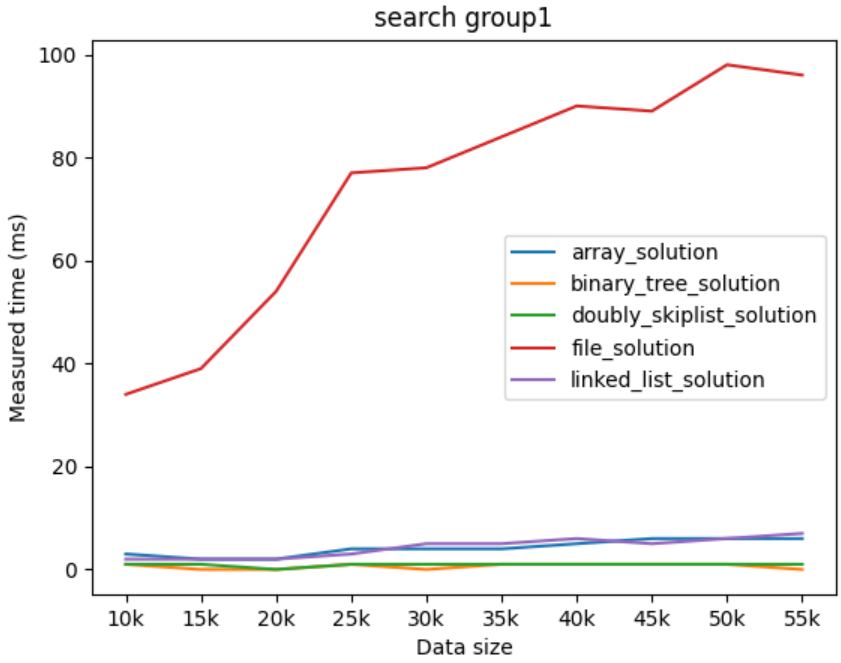
List solution: Add function in the list solution has O(n) running time complexity. It seems like list data structure which is defined in stl library does not use tail in the implementation. Because of that, for every addition it should traverse the nodes in the linked list. As a result, we can see that the graph shows the linear behavior of the add function for the list data structure.

Vector Solution: Add function in the vector solution has O(1) running time complexity. We know that the new id should be inserted to the last index of the vector. Push_back() function keeps size and capacity so that if size is enough to add the element to the end of the vector it adds in O(1) since the index is known. However, if the size is not enough it should increase the size of the vector. It doubles the capacity of the vector and puts the new id in the correct place. Because the size is doubled just, when necessary, push_back() function amortized to O(1). So, the graph shows the constant time behavior of the push_back function.



Behavior of the binary tree solution and the skip list solution for add operation explained previously. We can see the same behavior here as well.

Map Solution: Adding is O(log n) for stl map. So, we can see the logarithmic behavior from the graph. Since the datasets are small log n behavior is not being observed properly but it is log n.



Array solution: O(n)

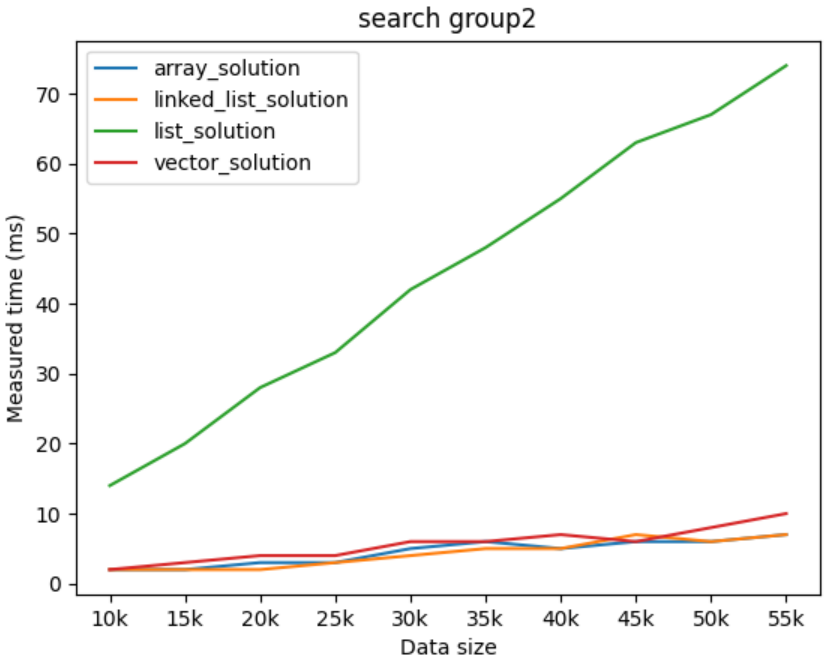
Binary tree solution: O(log n)

Skip list solution: O(log n)

File solution: O(n)

Linked list solution: O(n)

Because the file solution takes so much time, linearity of the array and linked list solutions, and the logarithmic behavior of the binary tree, skip list solutions are not being observed but they act like that.



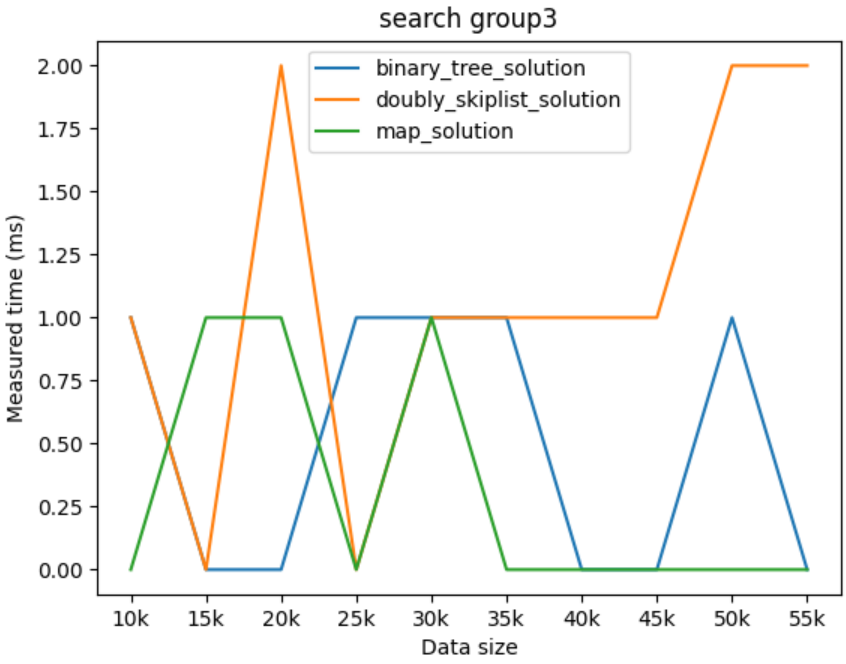
Array solution: O(n)

Linked list solution: O(n)

List solution: O(n)

Vector solution: O(n)

Because the list solution runs slower these linearities cannot be seen.

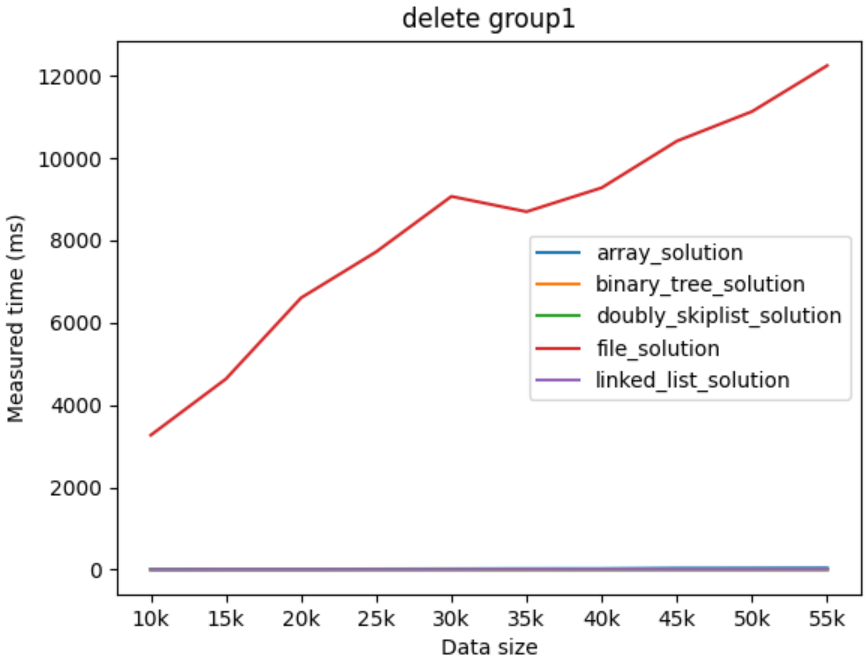


Binary tree solution: O(log n)

Skip list solution: O(log n)

Map solution: O(log n)

Since the execution time is very low and functions are very fast and the datasets are very small, log n behavior cannot be seen.



Array solution: O(n)

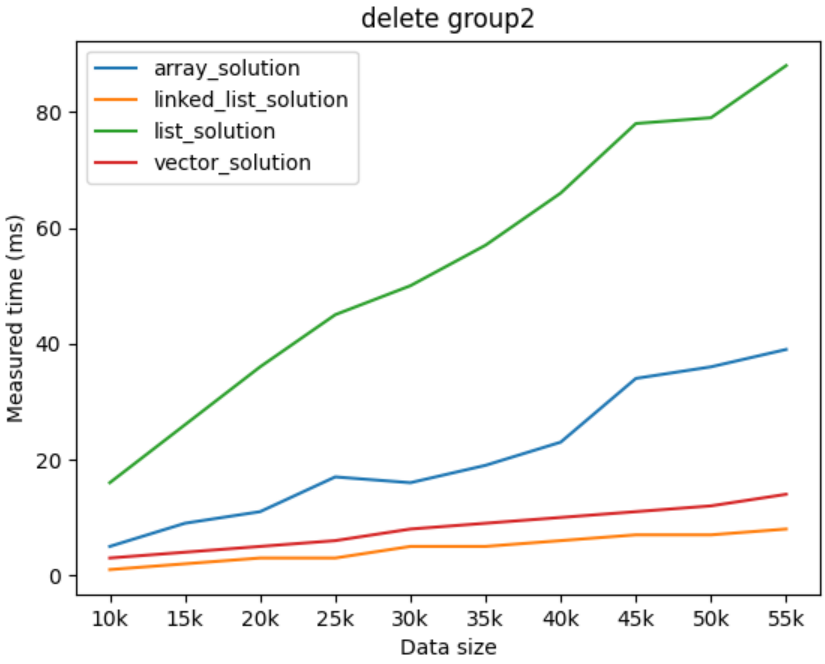
Binary tree solution: O(log n)

Skip list solution: O(log n)

File solution: O(n)

Linked list solution: O(n)

Because the file solution takes so much time, linearity of the array and linked list solutions, and the logarithmic behavior of the binary tree, skip list solutions are not being observed but they act like that.

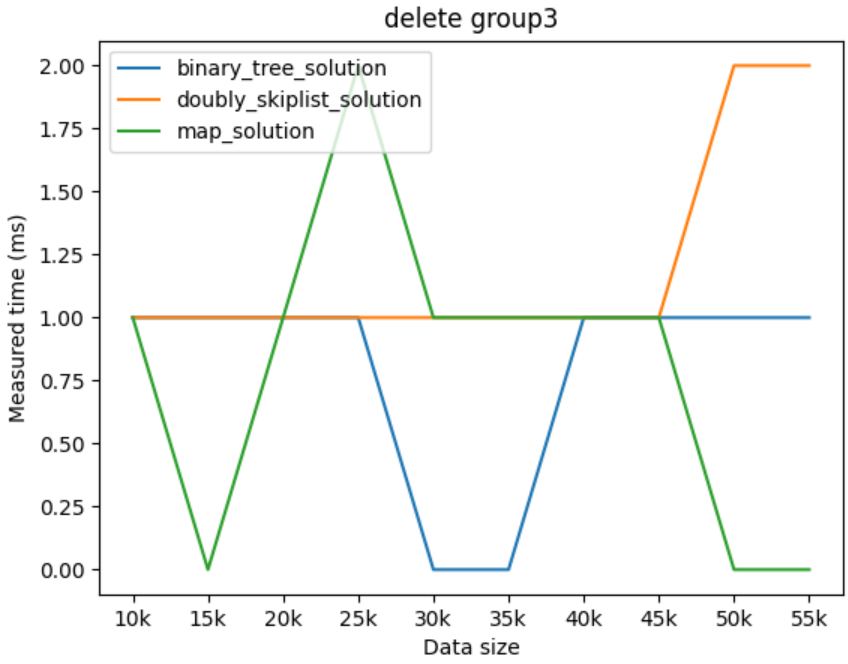


Array solution: O(n)

Linked list solution: O(n)

List solution: O(n)

Vector solution: O(n)



Binary tree solution: O(log n)

Skip list solution: O(log n)

Map solution: O(log n)

Since the execution time is very low and functions are very fast and the datasets are very small, log n behavior cannot be seen.