BLG223E-HOMEWORK5-REPORT

Name Surname: Racha Badreddine

Student Number: 150210928

Introduction:

In this report there are 3 different groups of data structures, for each data structure run time of insert, search, delete operations was calculated separately and the results are shown below.

To measure the run time 10 datasets were used. The data consist of employee's ID, Salary and Department, which were stored in 10 different CSV files as follows: 10k, 20k, 30k, 40k, 50k, 60k, 70k, 80, 90k and 100k. All the files used have shuffled data (the employees' IDs are not ordered).

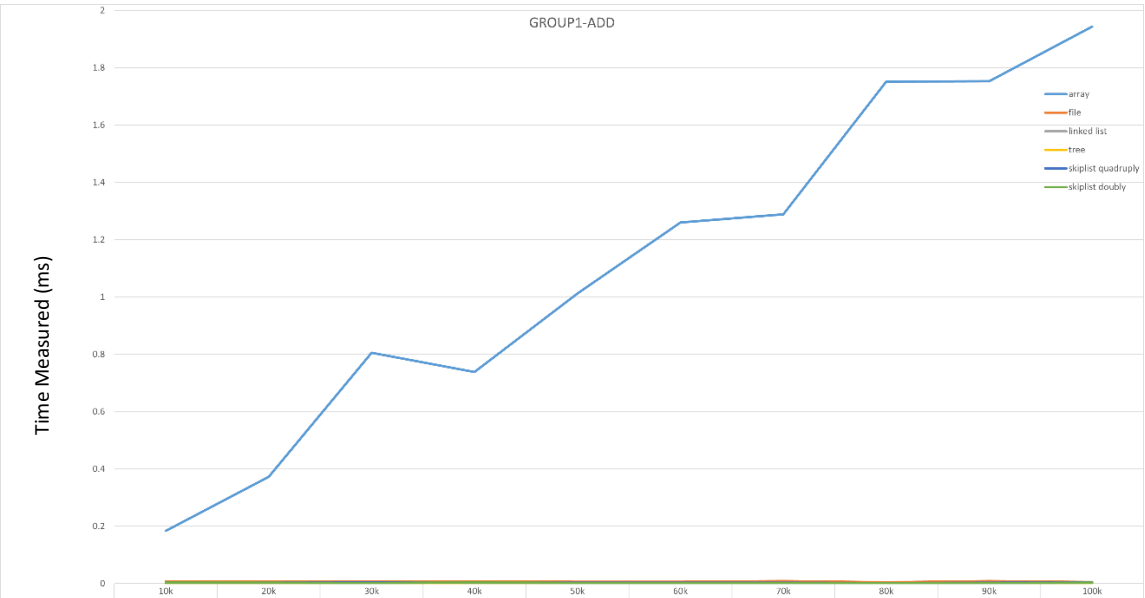
The run time was calculated for 3 main operations, Insert where a random node was created and inserted to the data structure, Search operation where a random ID was generated and searched for in the data structure, Delete operation where a random ID was generated and the employee with that ID was deleted if it exists.

The first Group consists of the following data structures: Array, File I/O, linked list, Binary Search Tree and skip list both quadruply and doubly. The second one consists of Array, Linked List, Vector STL, List STL. And the third one: Binary Search Tree, Map and skip list.

The time was measured in milliseconds.

Group1:

Insert Operation:



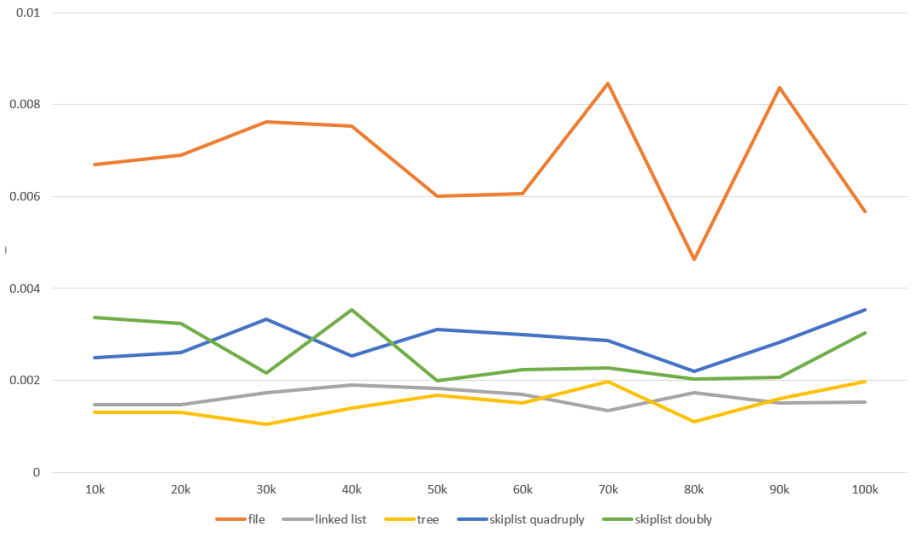
	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
array	0.1845	0.3731	0.8057	0.7384	1.0128	1.2597	1.2888	1.7516	1.7529	1.9437
file	0.0067	0.0069	0.007633333	0.007533333	0.006	0.006066667	0.008466667	0.004633333	0.008366667	0.005666667
linked list	0.001466667	0.001466667	0.001733333	0.0019	0.001833333	0.0017	0.001333333	0.001733333	0.0015	0.001533333
tree	0.0013	0.0013	0.001034483	0.0014	0.001666667	0.0015	0.001966667	0.0011	0.0016	0.001966667
skiplist quadruply	0.0025	0.0026	0.003333333	0.002533333	0.0031	0.003	0.002866667	0.0022	0.002833333	0.003533333
skiplist doubly	0.003366667	0.003233333	0.002166667	0.003533333	0.002	0.002233333	0.002266667	0.002033333	0.002066667	0.003033333

Figure1: Group1 – Insert

As it can be observed from the graph (Figure 1) the array structure is the slowest in this group, it takes many milliseconds compared to the other structures and the reason is that the array was implemented as

follows: it creates a new array with the previous size plus one then it copies all the previous array's data then it adds the new one as the last element. And after that it deletes the previous array to free the memory. This takes more time compared to the other structures where there is not copying and deleting and all this long process.

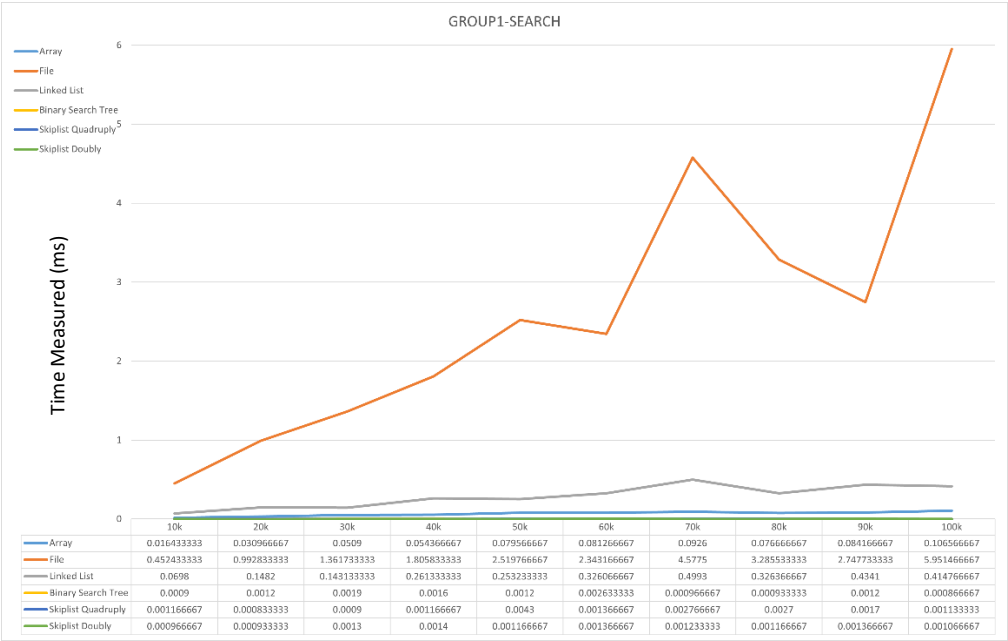
A zoomed picture of the other structures is shown below:



Zoomed view of figure 1

Concerning the other structures in this group the File I/O is slower compared to the others but it is kind of constant since every time it just goes to the end of the file and adds. The other structures are close to each other. The binary search tree is the fastest (O(log(n)) complexity) then the linked list. SkipList is slightly slower due to the random height of each node, the node is actually a linked list inside, this may be the reason that it is slower than tree and linked list where only one node is created.

Search Operation:



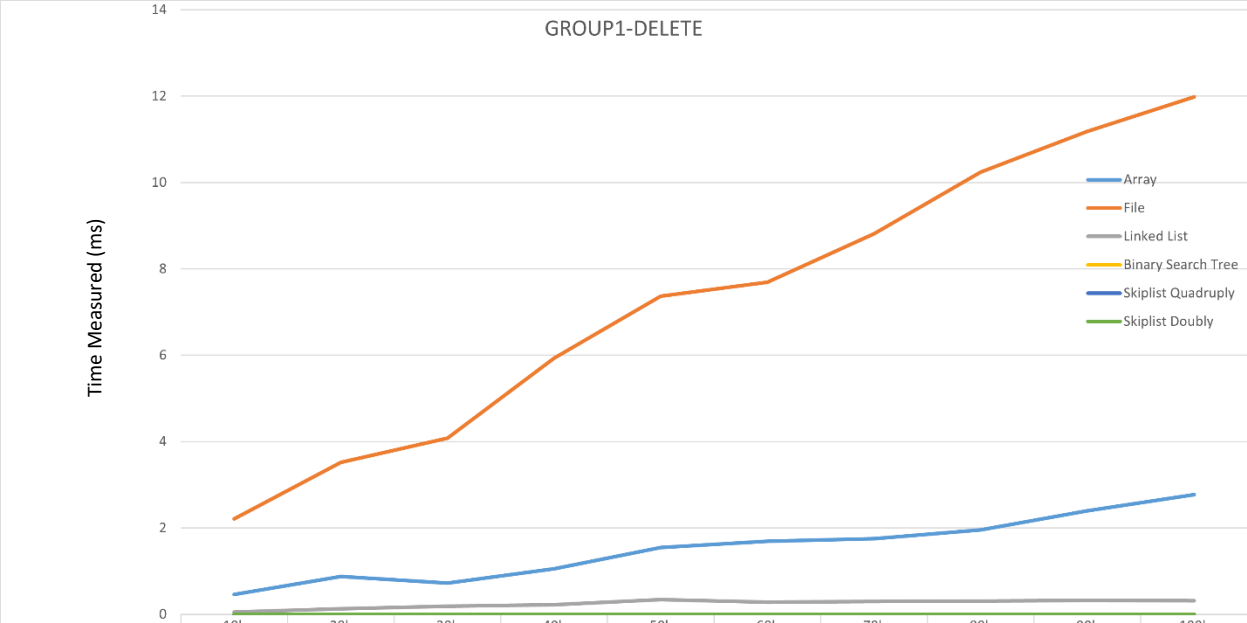
	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Array	0.016433333	0.030966667	0.0509	0.054366667	0.079566667	0.081266667	0.0926	0.076666667	0.084166667	0.106566667
File	0.452433333	0.992833333	1.361733333	1.805833333	2.519766667	2.343166667	4.5775	3.285533333	2.747733333	5.951466667
Linked List	0.0698	0.1482	0.143133333	0.261333333	0.253233333	0.326066667	0.4993	0.326366667	0.4341	0.414766667
Binary Search Tree	0.0009	0.0012	0.0019	0.0016	0.0012	0.002633333	0.000966667	0.000933333	0.0012	0.000866667
Skiplist Quadruply	0.001166667	0.000833333	0.0009	0.001166667	0.0043	0.001366667	0.002766667	0.0027	0.0017	0.001133333
Skiplist Doubly	0.000966667	0.000933333	0.0013	0.0014	0.001166667	0.001366667	0.001233333	0.001166667	0.001366667	0.001066667

Figure2: Group1 – Search

For search operation, Binary Search tree, quadruply skiplist and doubly skiplist are the fastest structures, it is already known that they have O(log(n)) complexity which is the worst case. Array and LinkedList are slower than them since in the worst case we can get O(n) complexity which means traversing all the array or linked list, and this is the reason of the increase observed with larger dataset (running time increases significantly with the data)

Searching using File I/O takes so long which makes it a bad option when we want to handle data. A substantial increase is observed when data gets larger.

Delete Operation:



	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Array	0.460233333	0.876066667	0.7232	1.0555	1.546033333	1.692466667	1.747366667	1.958	2.397733333	2.774566667
File	2.2137	3.5212	4.0795	5.933466667	7.367366667	7.690433333	8.815833333	10.23536667	11.17933333	11.9775
Linked List	0.053033333	0.1271	0.1911	0.2192	0.3388	0.278766667	0.2997	0.3087	0.325666667	0.3185
Binary Search Tree	0.001733333	0.001866667	0.003333333	0.0013	0.0011	0.001666667	0.0017	0.001033333	0.001133333	0.001233333
Skiplist Quadruply	0.001033333	0.001333333	0.0025	0.001466667	0.0016	0.001033333	0.001366667	0.001433333	0.0035	0.0013
Skiplist Doubly	0.001566667	0.001133333	0.001	0.0026	0.002666667	0.003266667	0.001133333	0.001266667	0.002033333	0.0012

Figure3: Group1 – Delete

Since the delete operation starts by searching for that data to delete, a similar behavior to the search operation can be observed.

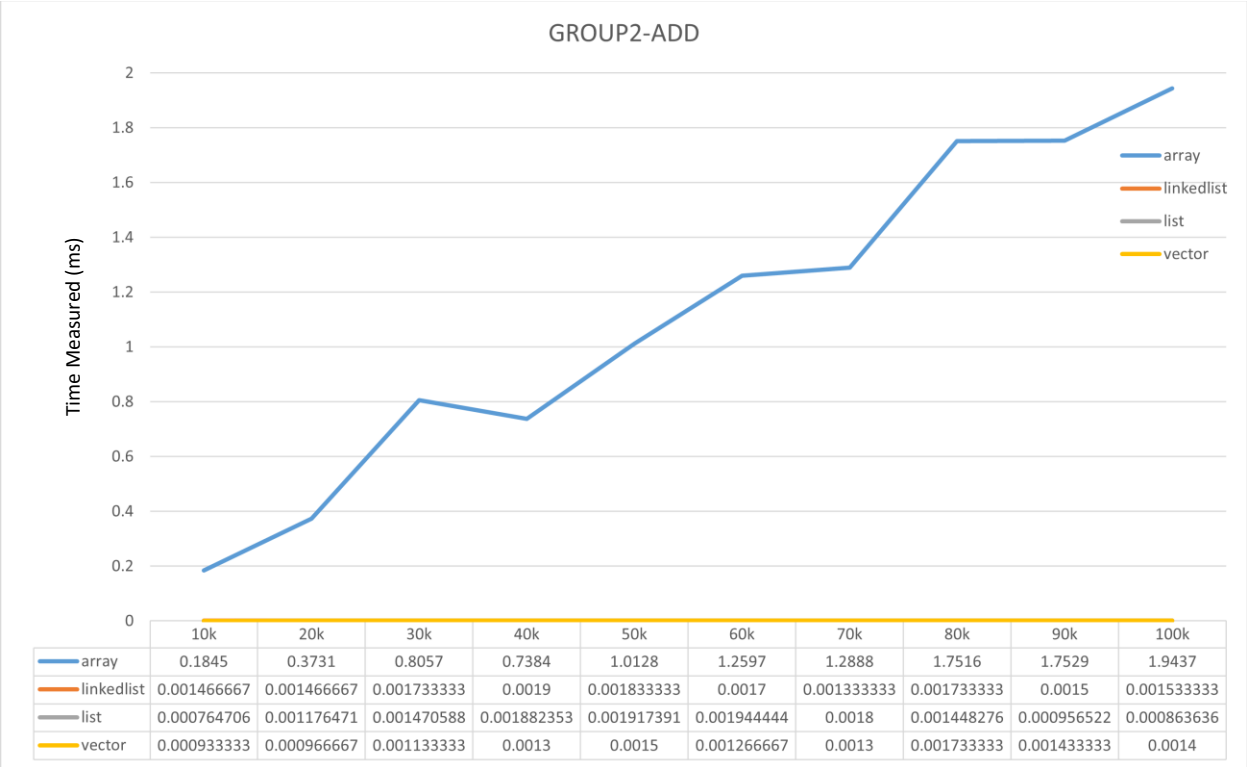
The fastest structures are Binary Search Tree, Quadruply Skiplist and Doubly Skiplist since they have O(log(n)) complexity in worst case. And similar to the search when the data increases the running time does not increase significantly (logarithmic growth).

For the delete the array is slower than the linked list since more operations are executed to delete. The linked List it only searches for the node, deletes it and updates the pointers. However, for the array when the element is found a new array is created with previous size minus 1, then all the elements except the found element are copied to the new one then the previous one is deleted to free the memory. This takes more time and increases significantly when the data is larger since more elements are copied and deleted.

File I/O is the slowest just like the search but for the delete is slower even more because the code was implemented as follows: when data is found a new file is created and all the data except that one is copied to the new one then the previous one is deleted.

Group2:

Insert Operation:

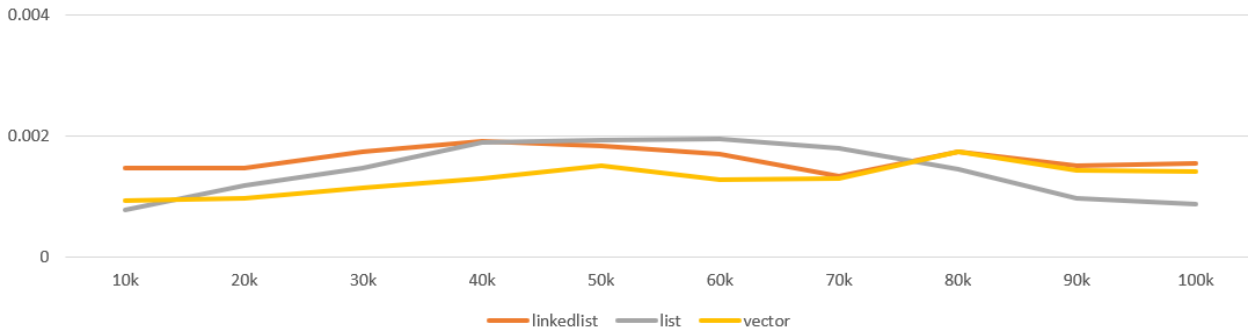


	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
array	0.1845	0.3731	0.8057	0.7384	1.0128	1.2597	1.2888	1.7516	1.7529	1.9437
linkedlist	0.001466667	0.001466667	0.001733333	0.0019	0.001833333	0.0017	0.001333333	0.001733333	0.0015	0.001533333
list	0.000764706	0.001176471	0.001470588	0.001882353	0.001917391	0.001944444	0.0018	0.001448276	0.000956522	0.000863636
vector	0.000933333	0.000966667	0.001133333	0.0013	0.0015	0.001266667	0.0013	0.001733333	0.001433333	0.0014

Figure4: Group2 - ADD

In this group the array is slowest structure compared to these structures too and this is due to the multiple operations done when new element is added (In group1 insert operation it has been explained).

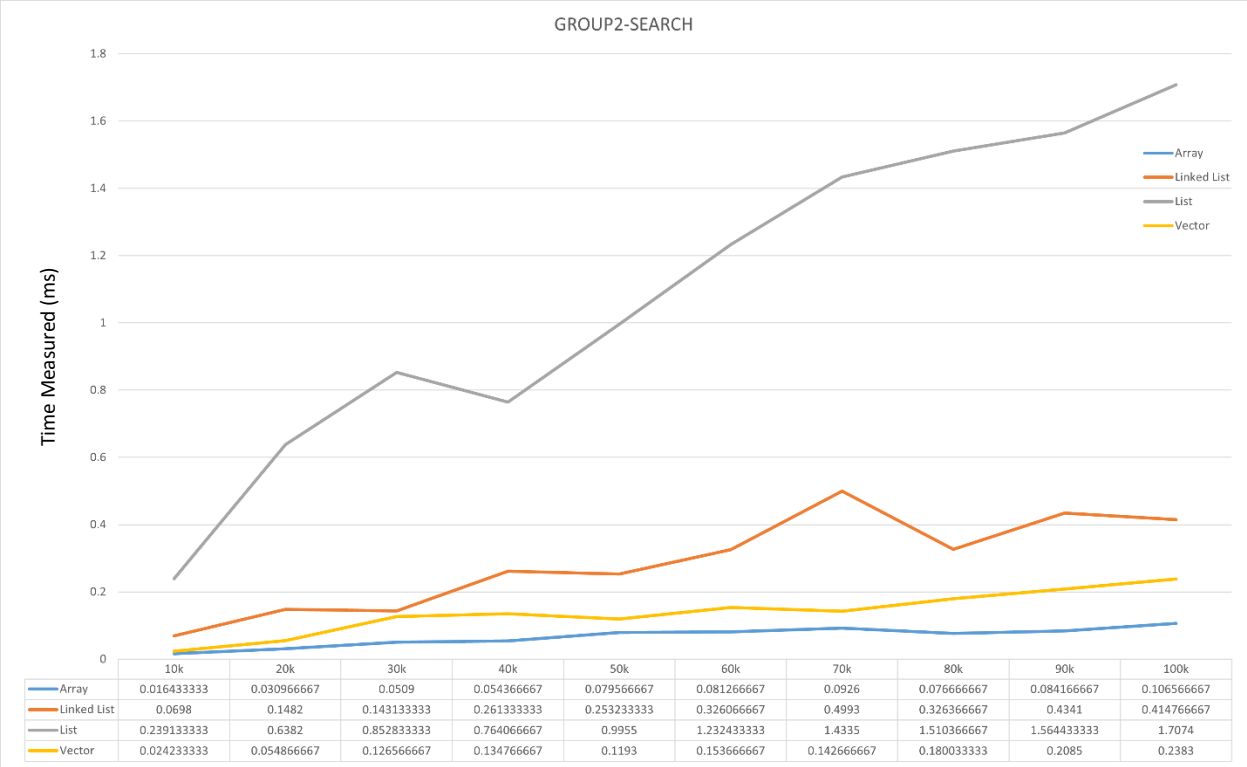
A zoomed picture of the other structures is shown below:



Zoomed view of figure 4

Linked List, List and vector have close running time because adding a new node to all of them has same logic: iterating to the end of the structure and add the new node. When the data is larger a small increase is observed for both vector and LinkedList.

Search Operation:

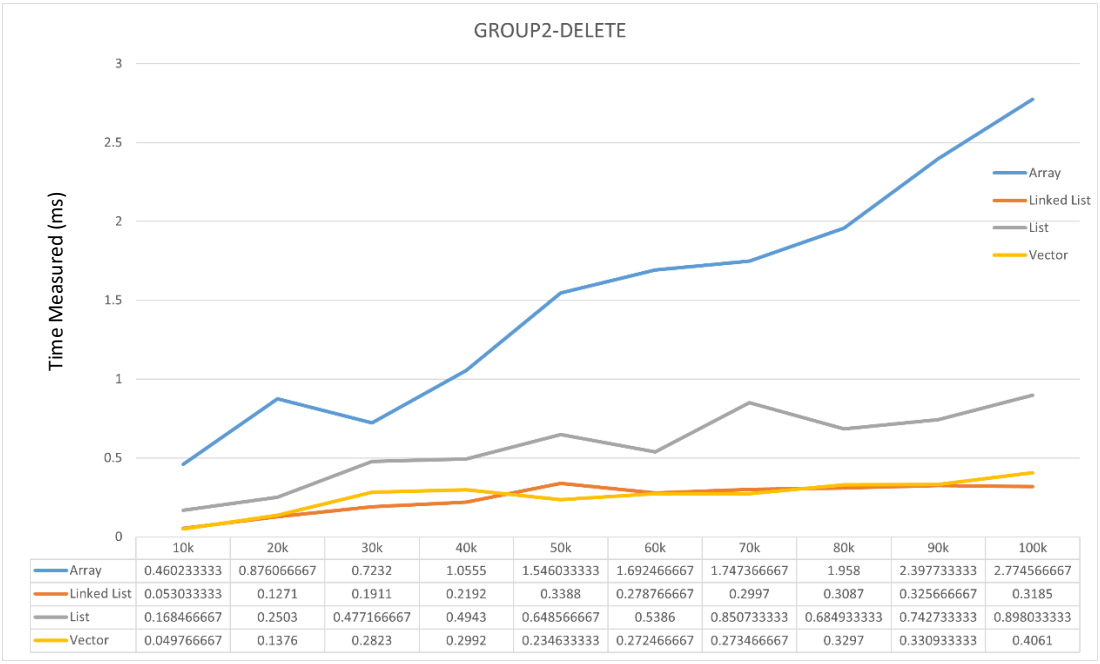


	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Array	0.016433333	0.030966667	0.0509	0.054366667	0.079566667	0.081266667	0.0926	0.076666667	0.084166667	0.106566667
Linked List	0.0698	0.1482	0.143133333	0.261333333	0.253233333	0.326066667	0.4993	0.326366667	0.4341	0.414766667
List	0.239133333	0.6382	0.852833333	0.764066667	0.9955	1.232433333	1.4335	1.510366667	1.564433333	1.7074
Vector	0.024233333	0.054866667	0.126566667	0.134766667	0.1193	0.153666667	0.142666667	0.180033333	0.2085	0.2383

Figure5: Group2 – Search

The slowest structure is the STL list, when larger data is used substantial increase is observed. A slight increase is noticed with larger data in the vector and array. The LinkedList is faster than the STL list but slower than the others, it gets slower when the data is larger.

Delete Operation:



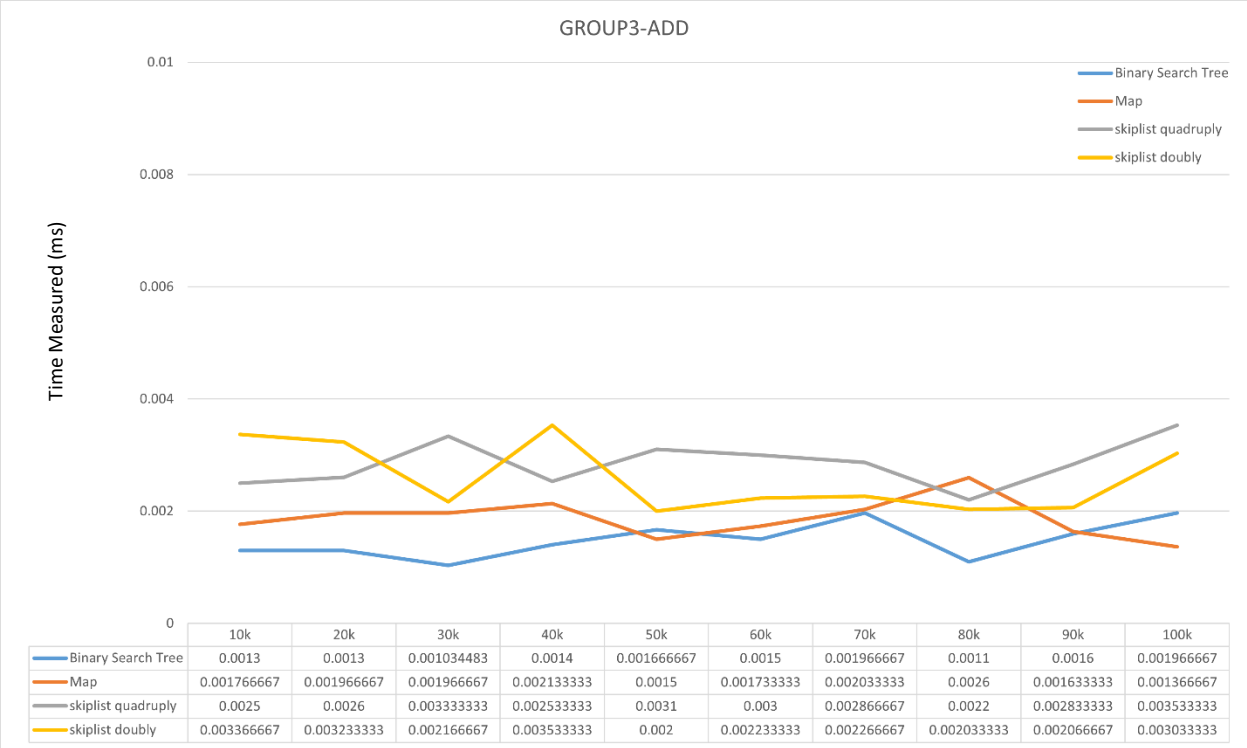
	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Array	0.460233333	0.876066667	0.7232	1.0555	1.546033333	1.692466667	1.747366667	1.958	2.397733333	2.774566667
Linked List	0.053033333	0.1271	0.1911	0.2192	0.3388	0.278766667	0.2997	0.3087	0.325666667	0.3185
List	0.168466667	0.2503	0.477166667	0.4943	0.648566667	0.5386	0.850733333	0.684933333	0.742733333	0.898033333
Vector	0.049766667	0.1376	0.2823	0.2992	0.234633333	0.272466667	0.273466667	0.3297	0.330933333	0.4061

Figure6: Group2 – Delete

The array is the slowest data structure since a lot of operations are executed when an element is deleted (it has been explained in group 1 delete operation). STL List is also slower than the vector and LinkedList but faster than the array. Then comes the LinkedList and STL vector which have close running time since it has same logic of deleting: iterating to find the element then delete it.

Group3:

Insert Operation:



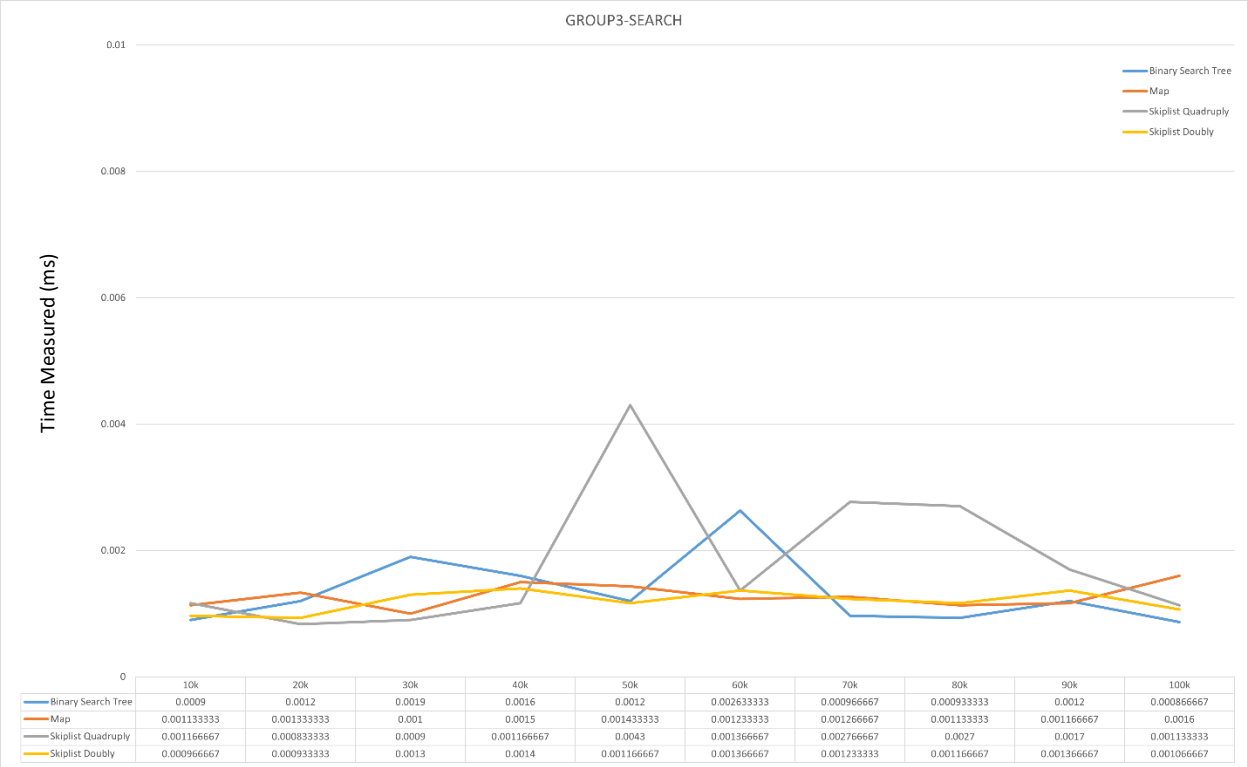
	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Binary Search Tree	0.0013	0.0013	0.001034483	0.0014	0.001666667	0.0015	0.001966667	0.0011	0.0016	0.001966667
Map	0.001766667	0.001966667	0.001966667	0.002133333	0.0015	0.001733333	0.002033333	0.0026	0.001633333	0.001366667
skiplist quadruply	0.0025	0.0026	0.003333333	0.002533333	0.0031	0.003	0.002866667	0.0022	0.002833333	0.003533333
skiplist doubly	0.003366667	0.003233333	0.002166667	0.003533333	0.002	0.002233333	0.002266667	0.002033333	0.002066667	0.003033333

Figure7: Group3 – ADD

It can be observed that all the structures of this group have close running time since they all have O(log(n)) complexity as worst case. The Binary Search Tree is the fastest structure in this group then the STL map which is actually a Red-Black tree (balanced tree) in C++. In this case even the Binary Search Tree is balanced due to the random dataset provided however when an ordered dataset is provided it is slower (similar to a linked list).

The changes observed in the skiplist is due to its probability characteristic, when the program runs, a random height is assigned to each node which changes every time the program runs, but actually this characteristic is the reason to make it a fast data structure with same complexity with the others.

Search Operation:

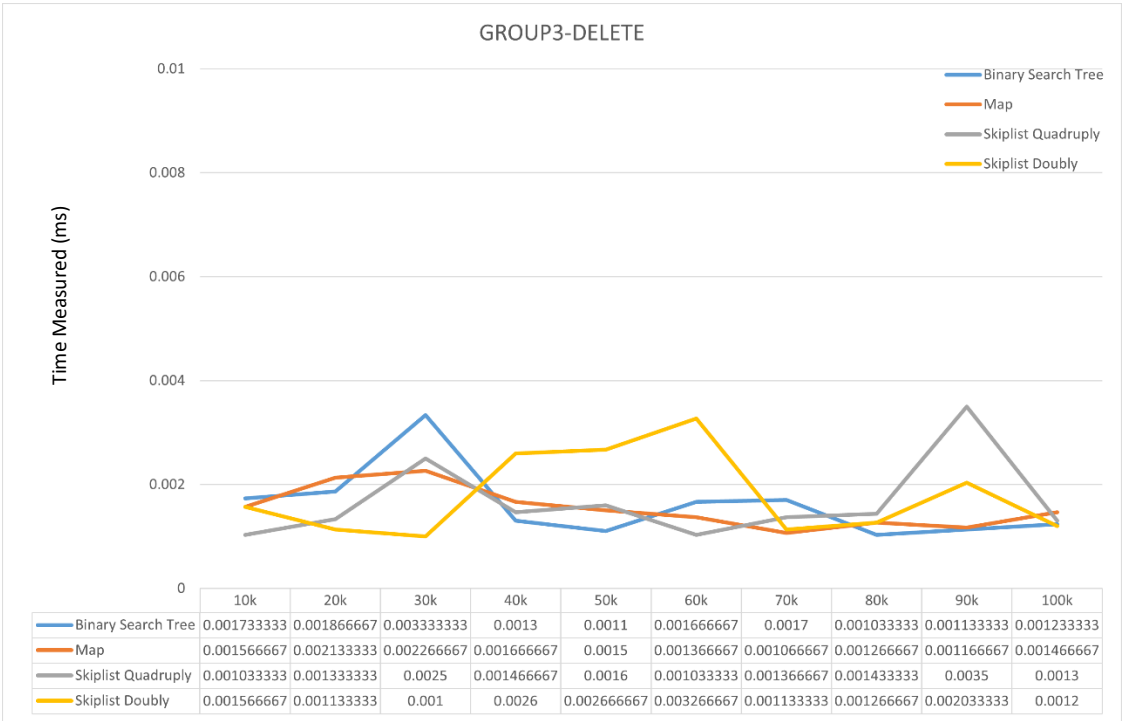


	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Binary Search Tree	0.0009	0.0012	0.0019	0.0016	0.0012	0.002633333	0.000966667	0.000933333	0.0012	0.000866667
Map	0.001133333	0.001333333	0.001	0.0015	0.001433333	0.001233333	0.001266667	0.001133333	0.001166667	0.0016
Skiplist Quadruply	0.001166667	0.000833333	0.0009	0.001166667	0.0043	0.001366667	0.002766667	0.0027	0.0017	0.001133333
Skiplist Doubly	0.000966667	0.000933333	0.0013	0.0014	0.001166667	0.001366667	0.001233333	0.001166667	0.001366667	0.001066667

Figure8: Group3 – SEARCH

For search operation same behavior to the add where all structures have close running time, even though an unexpected behavior of the quadruply skiplist in the 50k employees' data set occurred which is also may be due to the probability property. All these structures have O(log(n)) complexity which explains the behavior shown in the graph.

Delete Operation:



	10k	20k	30k	40k	50k	60k	70k	80k	90k	100k
Binary Search Tree	0.001733333	0.001866667	0.003333333	0.0013	0.0011	0.001666667	0.0017	0.001033333	0.001133333	0.001233333
Map	0.001566667	0.002133333	0.002266667	0.001666667	0.0015	0.001366667	0.001066667	0.001266667	0.001166667	0.001466667
Skiplist Quadruply	0.001033333	0.001333333	0.0025	0.001466667	0.0016	0.001033333	0.001366667	0.001433333	0.0035	0.0013
Skiplist Doubly	0.001566667	0.001133333	0.001	0.0026	0.002666667	0.003266667	0.001133333	0.001266667	0.002033333	0.0012

Figure9: Group3 – DELETE

Similar to search and add all the structures have close running times. The skiplist has unexpected behavior which maybe due to its probability property as it has been explained above. So, its running time changes every time due to the change in node's height which may make the searching for this node faster or slower than the time before.

- Search and Delete operations using data structures in the third group are faster than the other ones due to the time complexity O(log(n)).
- These results rely on the logic I have followed while implementing my codes in the previous homeworks.