UVIN ABEYSINGHE - 789931

This was done in A3 since the diagrams were too big to be read properly in A4.

Part 4

Our aim in this report is to understand the relationship between the hash table's load factor and the following:

1. Number of collisions.
2. Length of average probe sequence.

Steps taken to make the experiment fair:

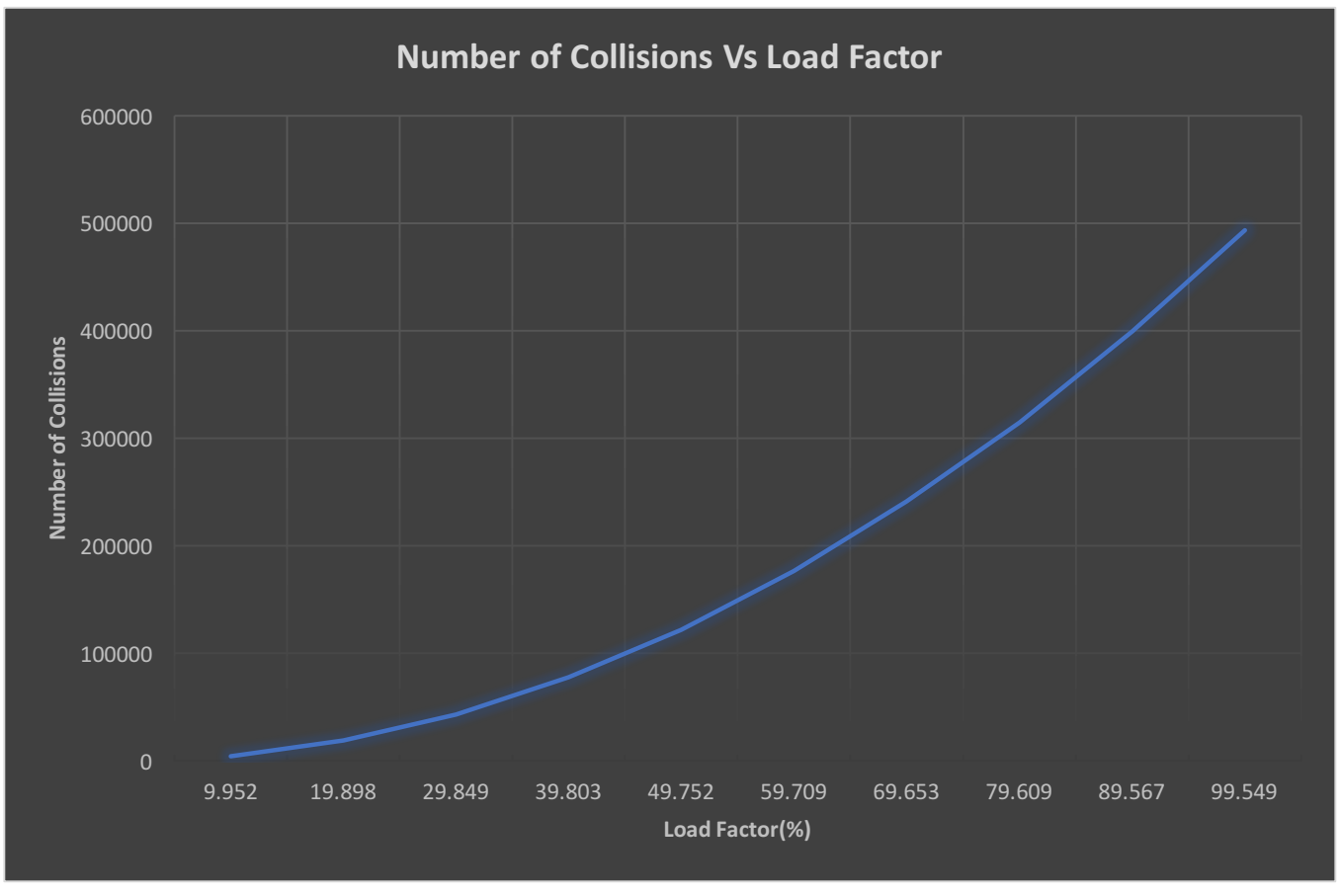
- We used text files with large number of *random* numbers with the smallest being 100,000 numbers including repetitions.
- Set the starting hash table size to 1000,000 so that hash table doubling will not affect the results.

We created 10 text files with number of numbers starting with 100,000 and ending with 1,000,000 with a step of 100,000 numbers. Using these file, we found the number of collisions and the length of average probe sequence and entered it to the table below.

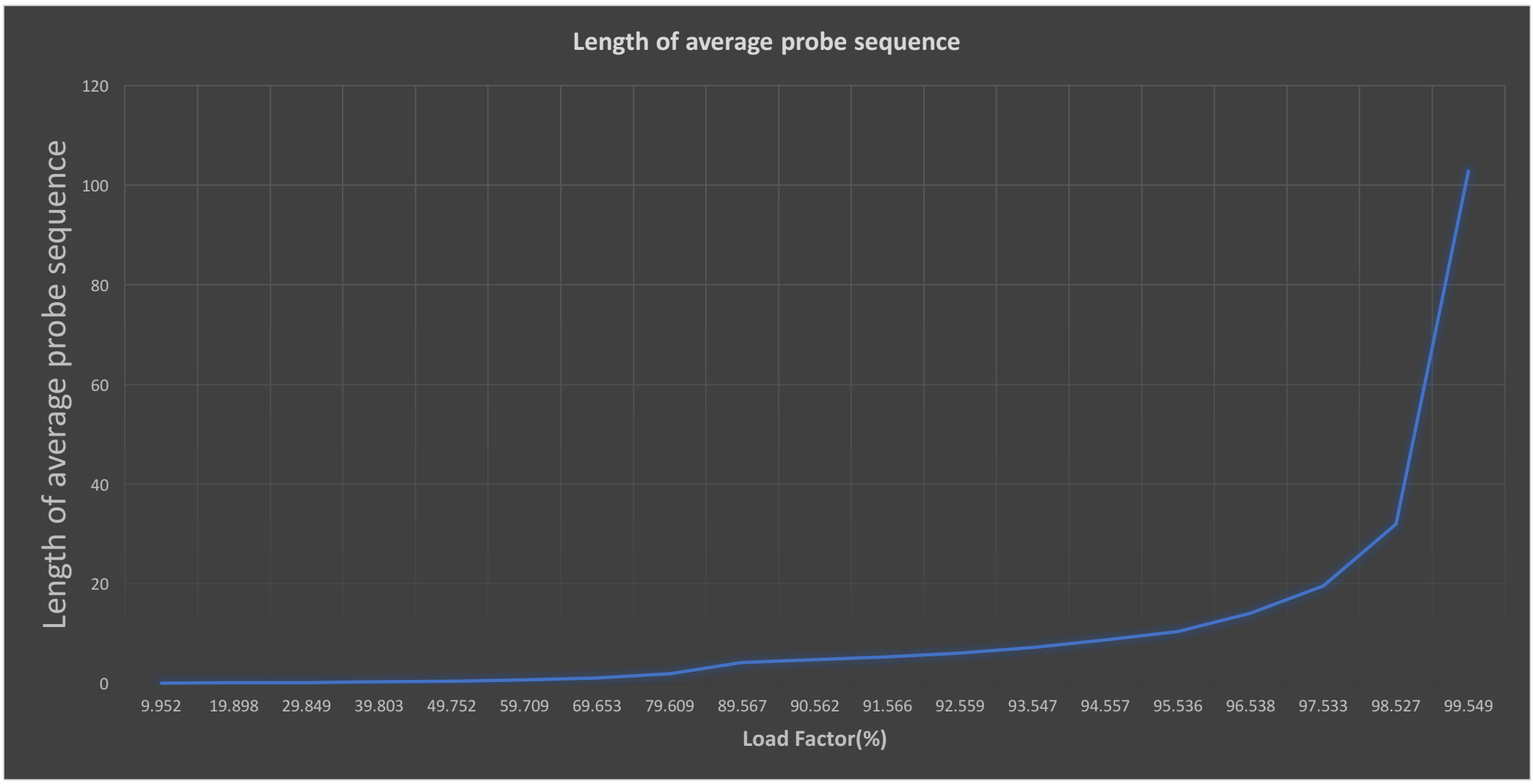
Number of Numbers (with repetition)	Load Factor	Number of Collisions	Length of average probe sequence
100000	9.952	4483	0.049656
200000	19.898	18883	0.1182
300000	29.849	43272	0.2048
400000	39.803	78069	0.322322
500000	49.752	122138	0.484959
600000	59.709	176289	0.732012
700000	69.653	241303	1.139861
800000	79.609	314550	1.9212
900000	89.567	399112	4.166004
1000000	99.549	493447	102.829272

However, we can see that there's a sharp increase in the length of average probe sequence between 900,000 and 1,000,000 numbers. Therefore, we created 9 more text files with number of numbers ranging between 910,000 and 1,000,000.

Number of Numbers (with repetition)	Load Factor	Length of average probe sequence
910000	90.562	4.818575
920000	91.566	5.355333
930000	92.559	6.116213
940000	93.547	7.272932
950000	94.557	8.74256
960000	95.536	10.443065
970000	96.538	14.086158
980000	97.533	19.540827
990000	98.527	31.998696



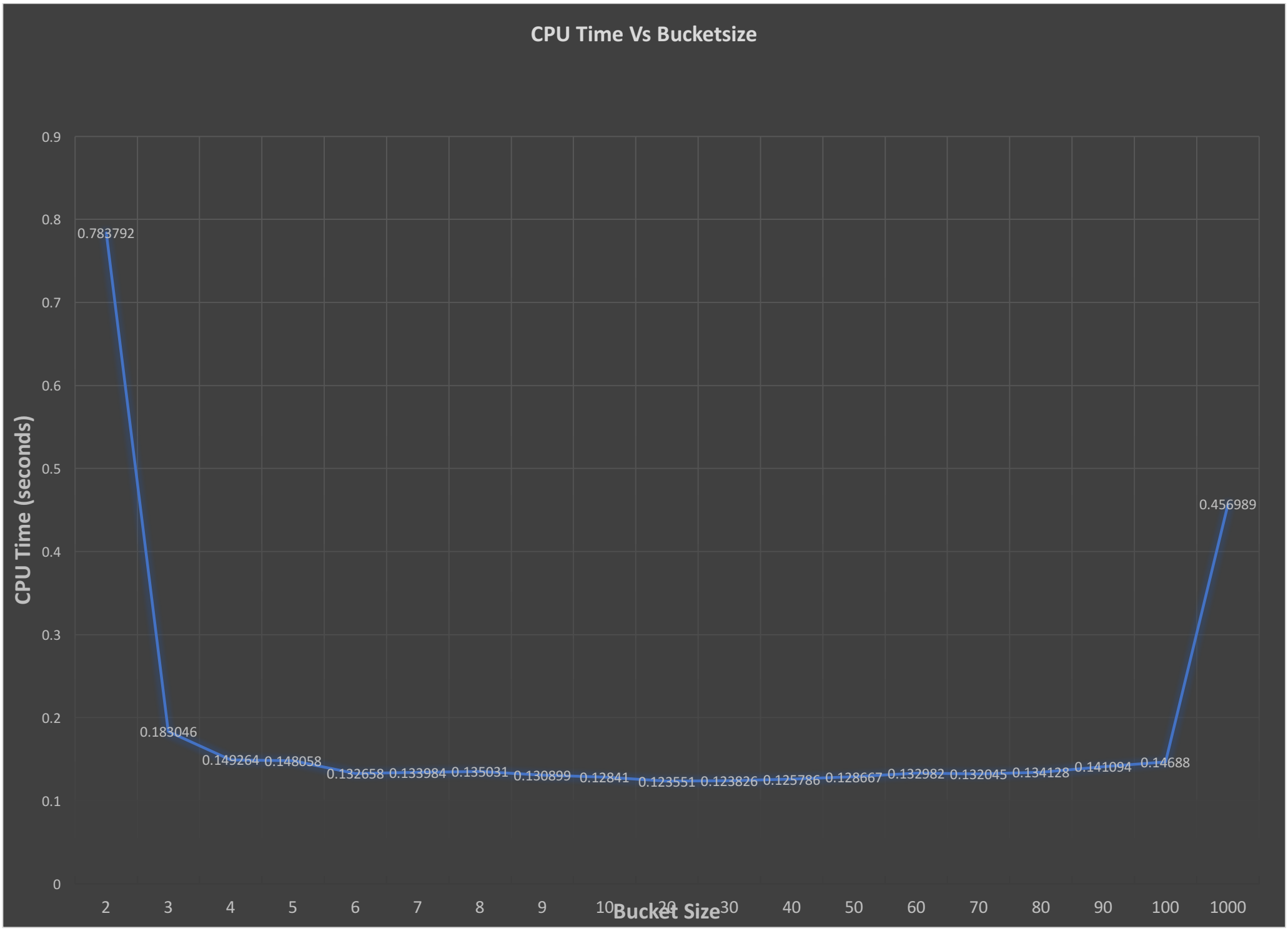
In this chart, we can see that the number of collisions increase when the load factor increases. The chart shows a curve, so the number of collisions doesn't increase linearly, so we can say that the number of collision are increasing at a faster rate when the load factor increases.



This graph shows that the length of average probe sequence starts to increase dramatically after reaching the load factor of around 95%. Therefore, it would be a good idea to double the table when the load factor reaches around 95 but could be a waste of memory if only a few more number are needed to be added and the doubled space is not needed. It depends on which attribute you need to constrict, memory or performance. The length of average probe sequence until 90-95 is very low which would speed up look ups.

Part 5

We created a text file with random 100,000 insertions (with repetitions) and random 100,000 look ups. Using this file, we checked the time taken to complete the insertions and look ups. *Insertion did not work when the bucket size was 1 since the table had to be doubled more than the maximum allowed size.* Therefore, ignoring bucket size = 1, we calculated the time for the rest of the bucket sizes from 2 to 10. Since the change in CPU time was very small when the bucket size was around 10, we checked for bucket sizes from 10 to 100 stepped by 10, and 1000. The following table shows the results.



The CPU time taken to finish the processes decrease dramatically from bucket size = 2 to bucket size =3, then it keeps decreasing and staying constant until a bucket size of around 35 and then starts increasing gradually. By the time the bucket size reaches 1000, the CPU time increases to 0.456989.
This shows that when the bucket size is too small, the program would spend more time doubling the table and copying down the pointers. On the other hand, when the bucket size is too large, there will be too many keys to look through to check if a key is present. Thus, when entering a key, the look up function takes more time going through the bucket of keys before inserting. When the bucket size is close to the number of keys being added, the look up function would be more of a linear search thus taking more time. Therefore, the ideal bucket size for 100,000 insertions and 100,000 lookups would be around 10-40 keys per bucket.

Bonus part:

Since we didn't build our programs focusing on efficiency, the Multi-Key Extendible Cuckoo Hashing algorithm takes too long to complete for extremely large number of insertions. Therefore, we tested using a text file with 20,000 insertions (with repetitions) and 20,000 look ups. The bucket sizes and the starting table sizes were selected as 8 wherever necessary. The table doubling for programs which use cuckoo was sets as the number of keys in both the tables plus 10.

Part1	Part2	Part3	Bonus
Cuckoo	Multiple-Key Extensible	xuckoo	xuckoon
0.152025 seconds	0.024182 seconds	7.021354 seconds	1.841397 seconds

To obtain the times, we ran the program thrice and obtained the smallest time. The table above Multiple-Key Extensible takes the smallest time to complete the actions, this maybe because it does not toggle between 2 tables thus not having to double according to a condition created by us. In other words, doubling of the table is efficient compared to the cuckoo programs we created. When considering cuckoo and xuckoon, cuckoo has space for 1 key so if another key is present it'll swap but xuckoon has to go through an array of n keys checking and then swapping with a random key in the array(splitting buckets as well). Cuckoo is faster here since its simpler compared to xuckoon, but it may take more memory. Xuckoo takes longer than xuckoon since it must split buckets more often since you can only save only key in one bucket compared to xuckoon's multiple key buckets.