Programming Assignment -3

Group -14

Kirubananth Sankar
Savya Sachi Pandey

Introduction:

The objective of this assignment is to understand the implementation of hash table, understand and study different techniques to manage collisions and understand their respective implementation and efficiencies.

Theoretical Analysis:

Hash Table - Chaining:

In this implementation, where there is a collision, the new key value pair is just stored alongside the existing key value pairs in the same location as a linked list or an array. This is more efficient than other implementations in terms of time complexity, but it is the worst implementation in terms of space complexity.

Insertion:

As mentioned above we first find the hash of the key, this is O(1). Then we check if the hash table at that position is already occupied, if not then we add it otherwise, we append the key value pair to the existing list. So in both case, insertion will only take O(1)

Amortized time complexity = O(1)
Total time complexity for N elements = O(N)

Removal:

When we have to remove a key, we will again find the index and get to the correct position in the table. But here there may be multiple keys stored in the same position in a list. So we have to search each element of this list to get our key value pair. So if there are ‘n’ pairs in this list the time taken would be O(n). In the worst case scenario, n = N, so the time taken is O(N).

Amortized time complexity = O(N)
Total time complexity for N elements = $O(N^2)$

Hash Table - LinearProbing:

Insertion:

In this method, when there is a collision, the algorithm linearly checks for the next available spot in the hash table. Let us say for an insertion we have to perform probing

‘n’ times, in the worst case scenario this ‘n’ would be equal to ‘N-1’ which is the total capacity of the table. So the insertion would be in linear time.

Amortized time complexity = O(N)
Total time complexity for N elements = $O(N^2)$

Removal:
In this method, just like in the case of insertion, to find the key itself it will take linear time. But once we find it we can immediately remove it since the position of the key is unique

Amortized time complexity = O(N)
Total time complexity for N elements = $O(N^2)$

Hash Table - DoubleHashing:

Insertion:
In this method, when there is a collision, we use a secondary hash function to recalculate a new position of the key and then insert in that position. Let us say for an insertion we have to calculate the index ‘n’ times, in the worst case scenario this ‘n’ would be equal to ‘N-1’ which is the total capacity of the table. So the insertion would be in linear time.

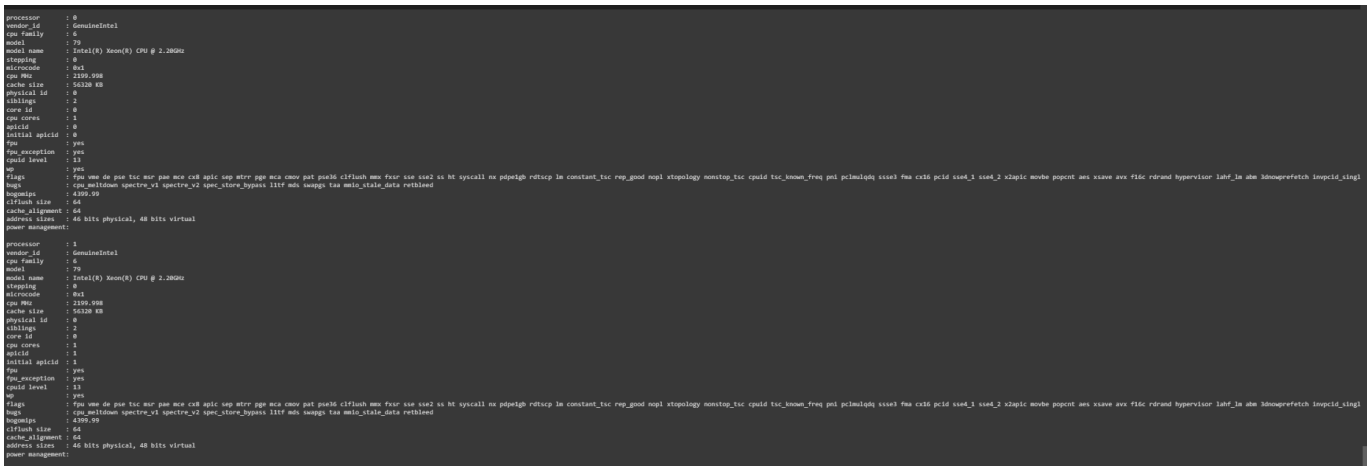
Amortized time complexity = O(N)
Total time complexity for N elements = $O(N^2)$

Removal:
In this method, just like in the case of insertion, to find the key itself it will take linear time. But once we find it we can immediately remove it since the position of the key is unique

Amortized time complexity = O(N)
Total time complexity for N elements = $O(N^2)$

Experimental setup:

Machine Specifications:



The processor speed is 2.2 GHz.

Test Inputs:
We used the given dictionary.txt as the input and tested the insertion times for various capacities.

The process:
We recorded the time for every 10 insertion operations and stored it in an array, for each of the implementations and finally plotted it to compare.

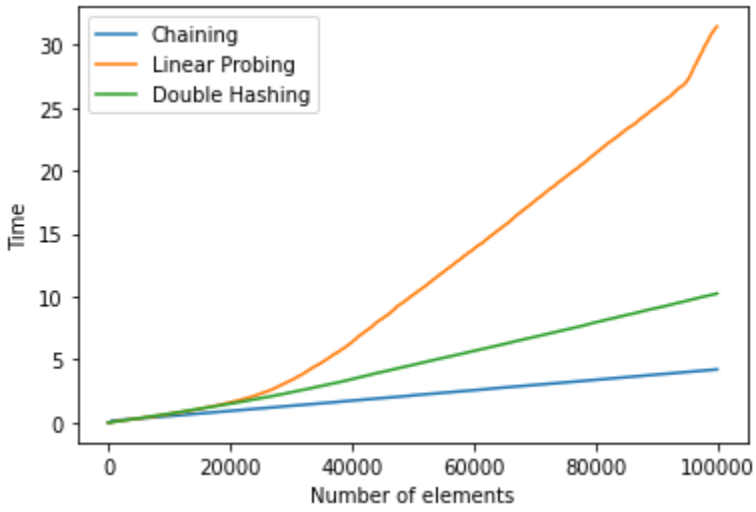
Experimental Results:

The number of unique values in the file is 58000

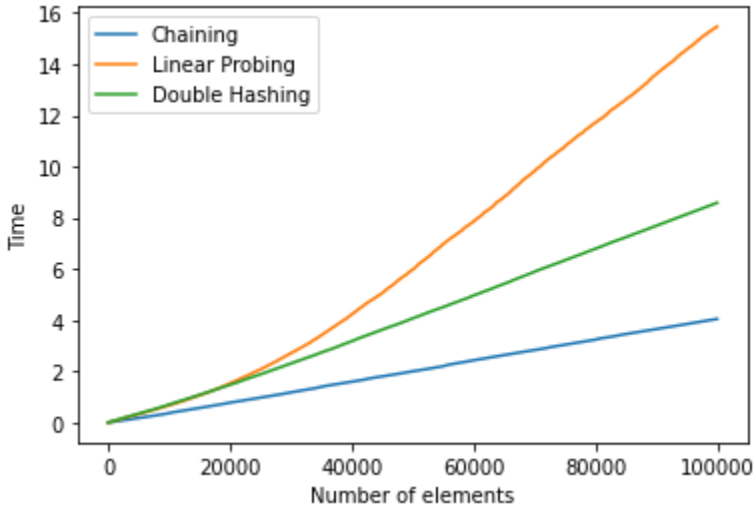
Capacity < 58,000:
In this case, it is not possible to even use linear probing or double hashing since the capacity itself is less than the unique keys.

Capacity = 58,000 (comparable to number of unique keys):
Even though theoretically it was possible for linear probing and double hashing to work in this case, they were still taking an impossible amount of time to complete the task making it impractical.

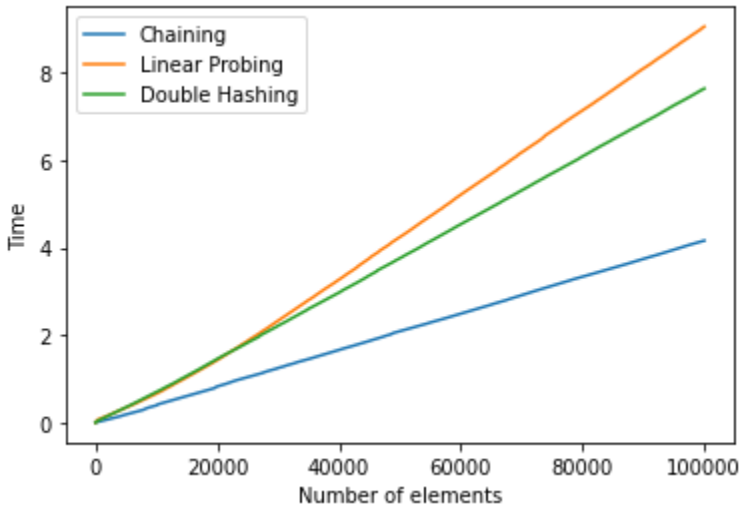
Capacity =59,000:



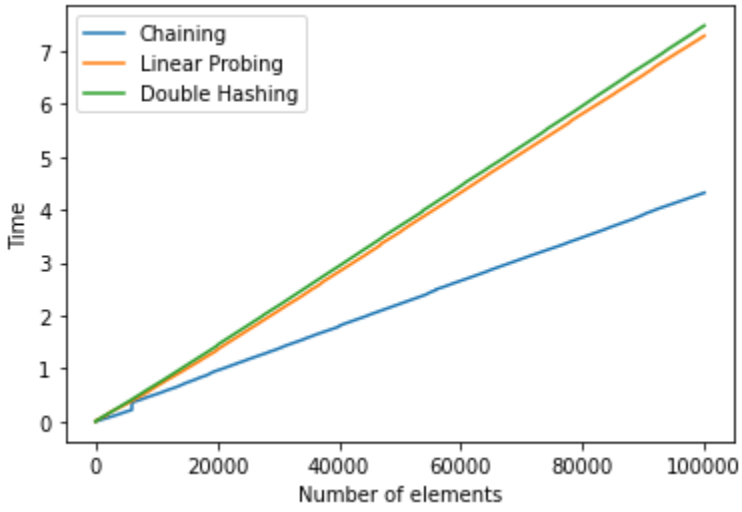
Capacity = 60,000:



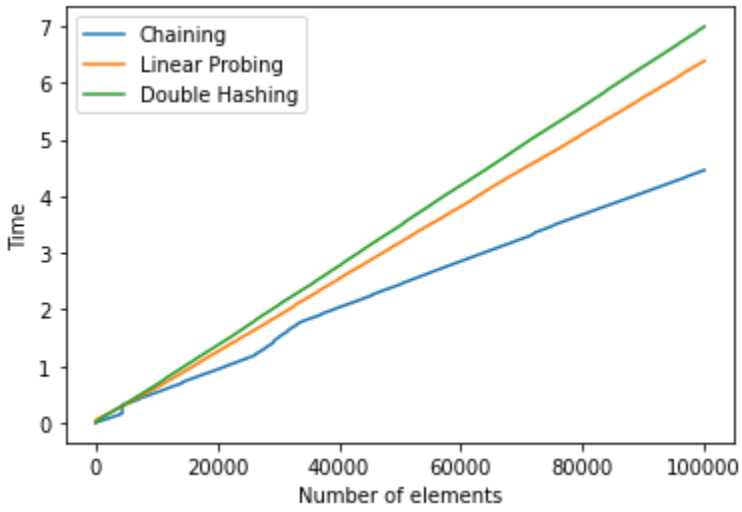
Capacity = 65,000:



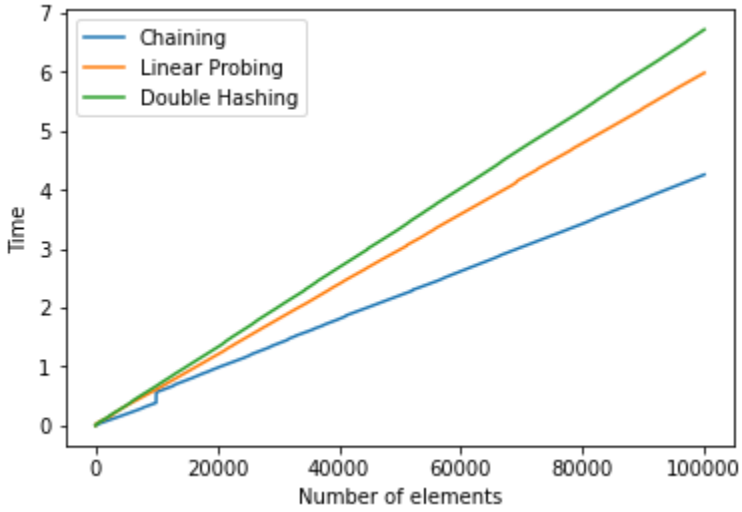
Capacity = 70,000:



Capacity = 100,000



Capacity = 10,00,000:



Observations:

- As expected from the theoretical data, the chaining implementation performed consistently better than both the other implementations.
- Again as expected from the theoretical analysis, the chaining implementation performs at O(N). But in the case of linear probing and Double hashing, initially when the capacity size was lesser and comparable to the number of unique key values, they performed at $O(N^2)$, but as the capacity increased they also performed at O(N).
- The Performance also depended on the initial size of the hash table. As we can see from the above graphs, as capacity increases, the difference in performance between linear

probing and double hashing decreases and after a capacity of 70000 linear probing starts performing better than double hashing. This is because the number of collisions starts decreasing as the capacity increases.

In conclusion, if space was not the concern it is always better to use the chaining implementation. Also when the capacity of the hash table is very low, again it's better to use the chining implementation. When the capacity of the hash table is high and we can’t use extra space, then it is better to use linear probing. Otherwise we can use the double hashing.