Smitha Gurunathakrishnan Balagurunatan | 58328870

CompSci 261P Project #1: Hashing algorithms

Hashing algorithms map data of arbitrary size to a hash of a fixed size. 4 hash collision methods have been implemented here.

•Linear probing
•Hash chaining
•Cuckoo hashing
•Quadratic probing

1) Linear Probing:

- In this, we linearly probe for next slot. The hash function used is – **h = key%array-size**. To implement this algorithm, an **array** is used as the hash table. Arraylist data structure is not being used here as it doesn't initialize the buckets with the given size unlike array.
- **Resizing** is also implemented in this algorithm, while inserting if load factor >= 1, the array is doubled, and while deleting load factor if <= 0.3, array is halved.
- Three main operations have been implemented for this algorithm –
  Insert(int key) : Inserts given element into the HashTable (array) – O(1)
  Find (int key): Finds given Element in the HashTable (array) – O(1)
  Delete (int key): Delete given Element in the HashTable (array) – O(1)
- Code is written in Java. Only positive elements are considered. Unsigned numbers ranging from 0 to 1000000.
- The pseudocode for this algorithm is given below -

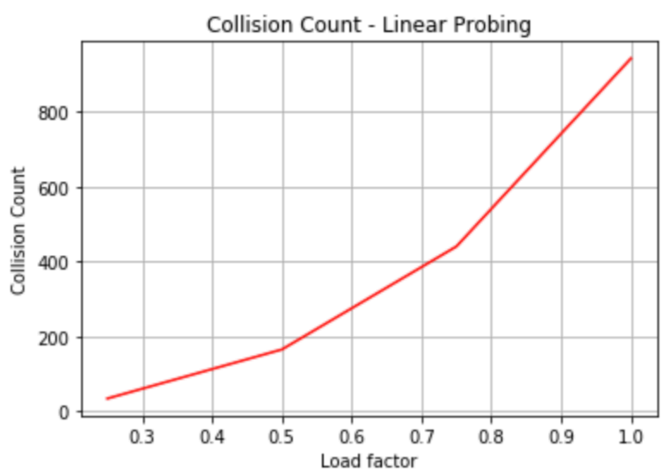
```
void Insert(int key)
           if no. of elements inserted >= array-size // Load Factor 1
              double the hashTable and rehash all existing elements
           compute index = key%arrsize
           while hashTable(index) is not empty
              increment index to next bucket in array
              increment the collision counter
           if hashTable(index) is empty
              place key in hashTable(index)
              increment no. of elements inserted

       void delete(int key)
           if no. of elements inserted <= array-size/3 // Load Factor 0.3
              Half the hashTable and rehash all existing elements
           compute index = key%arrsize
           while hashTable(index) is not empty
              if hashTable(index) matches key
                     set hashTable(index) as -1
                      return
              increment index to next bucket in array
           return
```

```
int find(int key)
    compute index = key%arrsize
    while hashTable(index) is not empty
       if hashTable(index) matches key
              return hashTable(index) element
       increment index to next bucket in array
    if not return -1
```

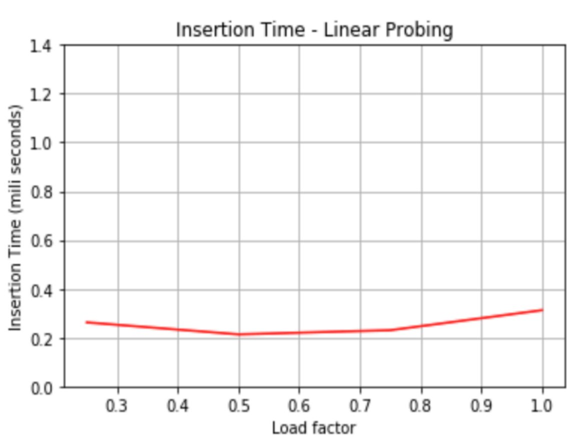
- Running time for Insert, Find and Delete in my implementations are similar as I am doing lazy delete. I set the bucket of the deleted item in array as -1 and hence it can be reused whenever necessary.

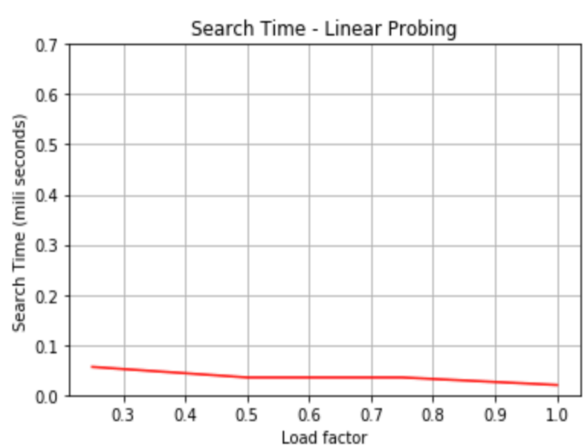
Experimental Analysis -

- I have fixed the hashtable (array) size as 1000 and I inserted 250, 500, 750 , 1000 elements. It is observed as the load factor increases, number of collisions that occur also increases.



- Time for Insert and Find operations are computed for varying load factors. It is noted that they remain constant for varying inputs sizes / load factors. This shows that Insert and Find operations have O(1) time in this implementation.





- Delete is implemented as a lazy delete, and internally calls find method, and so it is also done in O(1) time.

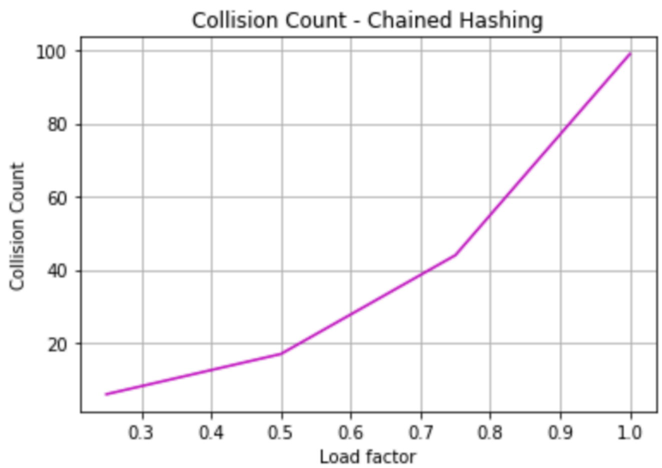
2) Chained hashing:

- In this, each cell of hash table(array) points to a linked-list of records that have same hash function value.
- The hash function used is – **h = key%array-size**. To implement this algorithm, an **array of linked-list** is used as the hash table. Arraylist data structure is not being used here as it doesn't initialize the buckets with the given size unlike arrays.
- Resizing is not implemented in this algorithm, as adding more elements leads to longer linked-list chains from a given array Index.
- Three main operations have been implemented for this algorithm –
  Insert(int key) : Inserts given element into the HashTable (array) – O(1)
  Find (int key): Finds given Element in the HashTable (array) – O(1)
  Delete (int key): Delete given Element in the HashTable (array) – O(1)
- Code is written in Java. Only positive elements are considered. Unsigned numbers ranging from 0 to 1000000.
- Delete here is implemented by finding and removing the element from linked-list which is O(1) as proved below in analysis.
- The pseudocode for this algorithm is given below -

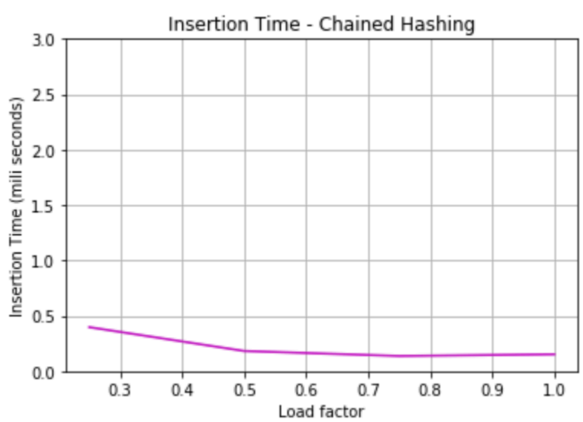
```
void insert(int key)
   compute index = key%arrsize
   if hashTable(index) contains key
       increment the collision counter
       add key to linked-link in hashTable(index) bucket of array

void delete(int key)
   compute index = key%arrsize
   while hashTable(index) is not empty
       if hashTable(index) contains key
              remove key from linked-list in the bucket hashTable(index)
       return;

int find(int key) {
   compute index = key%arrsize
   while hashTable(index) is not empty
   if hashTable(index) contains key
       return key from linked-list in the bucket hashTable(index)
   if not return -1
```

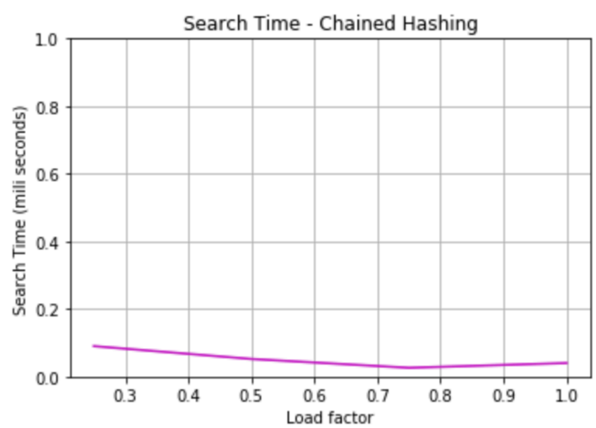
Experimental Analysis -

- I have fixed the hashtable (array) size as 1000 and I inserted 250, 500, 750 , 1000 elements. It is observed as the load factor increases, number of collisions that occur also increases. The Linked-list inside the array cells are empty initially.



- Time for Insert and Find operations are computed for varying load factors. It is noted that they remain constant for varying inputs sizes i.e load factors. This shows that Insert and Find operations have O(1) time in this implementation. Worst case search/ find become O(n) if the chains of linked-list increases.





3) Cuckoo hashing:

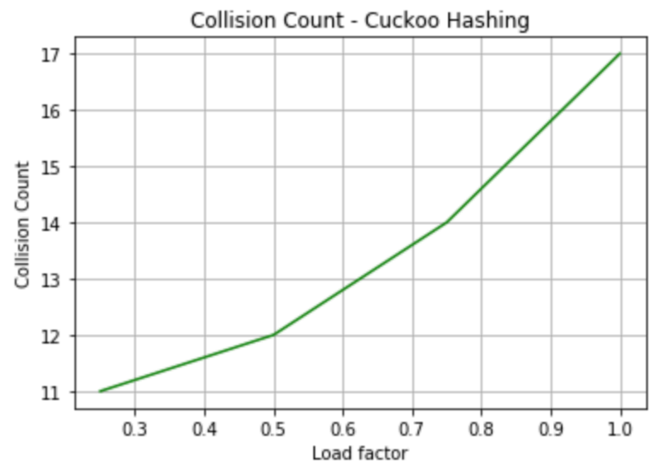
- This algorithm works by inserting a new key into a 1^{st} row of 2D array may push an older key to a different location 2^{nd} row of **2D array**. If there is already an element in 2^{nd} row it is in turn pushed to 1^{st} row and the algorithm goes on.
- The hash function used is – **h1 = key % array-size** and **h2 = (key/array-size) % array-size**.
- To implement this algorithm, an 2D array of size 2*N is used as the hash table. 1^{st} row contains entries for hash function 1 and 2^{nd} row contain entries for hash function 2.
- **Resizing** is implemented in this algorithm, while inserting if load factor >= 1, the array is doubled, and while deleting load factor if <= 0.3, array is halved.

- **Cycles or infinity loops** will occur in this hashing technique. It is handled by stopping the loops if the loop count equals or exceeds the size of input array is N. So N is maximum number of times the function can recursively call itself and try to insert keys.
- If such cycles are detected, the algorithm stops and resizing and rehashing is done. In this way or infinity loops are resolved.
- Three main operations have been implemented for this algorithm –
  - **Insert(int key) :** Inserts given element into the HashTable (2D array) – O(1)
  - **Find (int key):** Finds given Element in the HashTable (2D array) – O(1)
  - **Delete (int key):** Delete given Element in the HashTable (2D array) – O(1)
- Code is written in Java. Only positive elements are considered. Unsigned numbers ranging from 0 to 1000000.
- Delete here is implemented by finding the key using either of 2 hash functions and freeing that array index location. Thus this operation takes O(1) time.
- The pseudocode for this algorithm is given below -

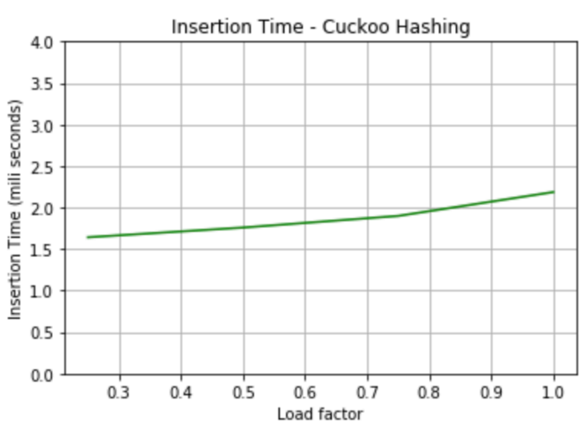
```
void insert(int key, int cycles)
while(true)
       if cycles >= array-size or no. of elements inserted >= (array-size *2)
       // double 2D array size if infinite loop or load factor = 1
              double the hashTable and rehash all existing elements
       compute index1 = key% array-size
       compute index2 = key/ array-size % array-size
       if hashTable[0][index1] or hashTable[1][index2] equals key)
              return;
       if hashTable[0][index1] is not empty
              temp key1 = hashTable[0][index1]
              hashTable[0][index1]= key;
              increment the collision counter
              key = temp key1; cycles+=1
       //similar to recursive push & insert of elements in empty slot, while loop continues
       else if hashTable[1][index2] is not empty
              temp key2 = hashTable[1][index2]
              hashTable[1][index2]= key;
              increment the collision counter
              key = temp key2; cycles+=1
       //similar to recursive push & insert of elements in empty slot, while loop continues
       else if slot is empty
              insert key in hashTable[0][index1] or hashTable[1][index2] based on recursion
              increment the no. of elements inserted
              return;

       void delete(int key) // find and set it 0
              if no. of elements inserted ((array-size*2)/3))
                     Half the hashTable and rehash all existing elements
              If key is found in location pointed by h1 or h2 hash functions
                     Set the element in that bucket/slot of 2D array to free(0)
              return

       int find(int key)
              If key is found in location pointed by h1 or h2 hash functions
                     return the element
              else return -1
```

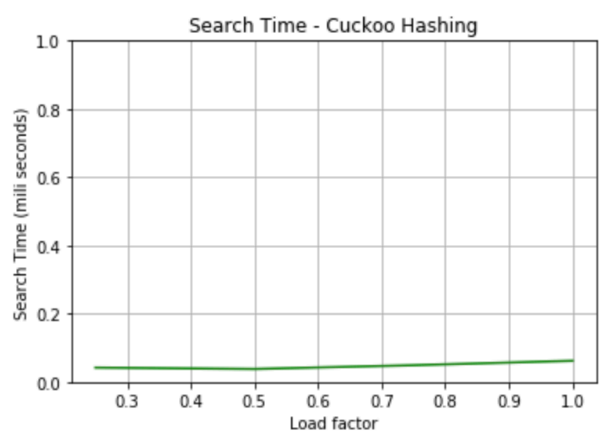
Experimental Analysis -

- I have fixed the hashtable (2D array) size as 100 and I inserted 250, 500, 750 , 1000 elements. It is observed as the load factor increases, number of collisions that occur also increases.



- Time for Insert and Find operations are computed for varying load factors. It is noted that they remain constant for varying inputs sizes i.e load factors. This shows that Insert and Find operations have O(1) time in this implementation.





4) Quadratic Probing:

- I have implemented this for optional hash function as it optimizes the collision resolving strategy used by linear probing. It reducing the probing time significantly.
- In this, we probe for i^2th slot in i^{th} iteration as next slot. The hash function used is – **h = key%array-size**. To implement this algorithm, an **array** is used as the hash table. Arraylist data structure is not being used here as it doesn't initialize the buckets with the given size unlike array.

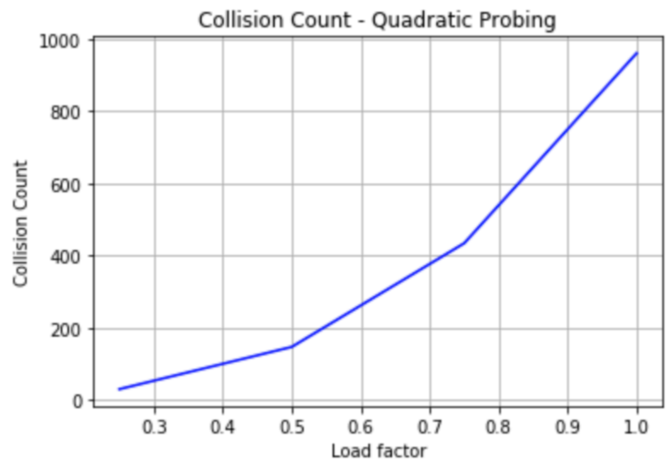
- **Resizing** is also implemented in this algorithm, while inserting if load factor >= 1, the array is doubled, and while deleting load factor if <= 0.3, array is halved.
- Three main operations have been implemented for this algorithm –
  Insert(int key) : Inserts given element into the HashTable (array) – O(1)
  Find (int key): Finds given Element in the HashTable (array) – O(1)
  Delete (int key): Delete given Element in the HashTable (array) – O(1)
- Code is written in Java. Only positive elements are considered. Unsigned numbers ranging from 0 to 1000000.
- The pseudocode for this algorithm is given below -

```
void Insert(int key)
          Let i be 1
          if no. of elements inserted >= array-size // Load Factor 1
             double the hashTable and rehash all existing elements
          compute index = (index + i * i++) % arrsize;
          while hashTable(index) is not empty
             increment index to next bucket in array
             increment the collision counter
          if hashTable(index) is empty
             place key in hashTable(index)
             increment no. of elements inserted

      void delete(int key)
          if no. of elements inserted <= array-size/3 // Load Factor 0.3
             Half the hashTable and rehash all existing elements
         compute index = (index + i * i++) % arrsize
          while hashTable(index) is not empty
             if hashTable(index) matches key
                    set hashTable(index) as -1
                     return
             increment index to next bucket in array
          return

      int find(int key)
         compute index = (index + i * i++) % arrsize
          while hashTable(index) is not empty
             if hashTable(index) matches key
                    return hashTable(index) element
             increment index to next bucket in array
          if not return -1
```

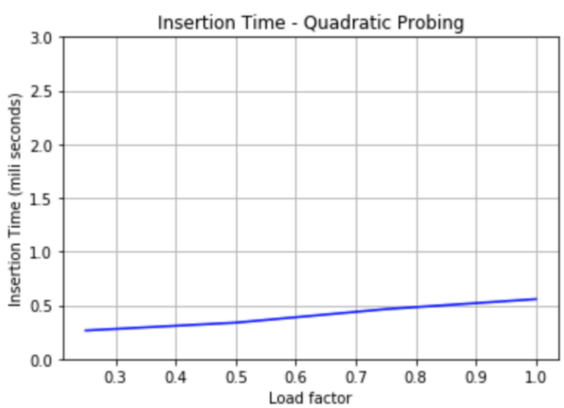
- Running time for Insert, Find and Delete in my implementations are similar as I am doing lazy delete. I set the bucket of the deleted item in array as -1 and hence it can be reused whenever necessary.

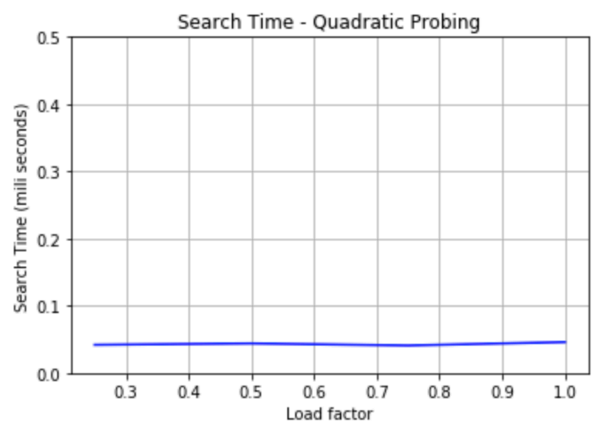
Experimental Analysis -

- I have fixed the hashtable (array) size as 1000 and I inserted 250, 500, 750 , 1000 elements. It is observed as the load factor increases, number of collisions that occur also increases. The Linked-list inside the array cells are empty initially.



- Time for Insert and Find operations are computed for varying load factors. It is noted that they remain constant for varying inputs sizes i.e load factors. This shows that Insert and Find operations have O(1) time in this implementation.





- Delete is implemented as a lazy delete, and internally calls find method, and so it is also done in O(1) time.

Comparitive Analysis between 4 hashing techniques –

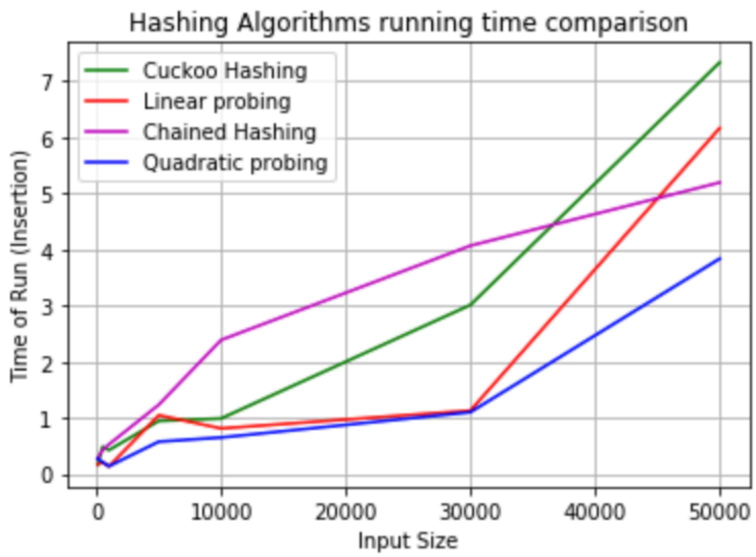
Time for algorithm runs (Insertion) -

1. For Cuckoo resizing and rehashing is done when there is cycle detected or if the load factor is one. This leads to overhead while inserting and hence performance for it is slower.

2. Linear Hashing takes more time as its collision resolution strategy is not very efficient and spends a lot of time probing one by one for empty slots as array size increases.

3. Chained hashing works efficiently as there is no overhead of re-sizing, re-hashing or probing while inserting.

4. Quadratic Probing works the best as its collision resolution strategy if very efficient and less time consuming for larger input / array sizes. Instead of probing every next slot, it probes only slots locations at the power of that iteration. This makes insert time faster.

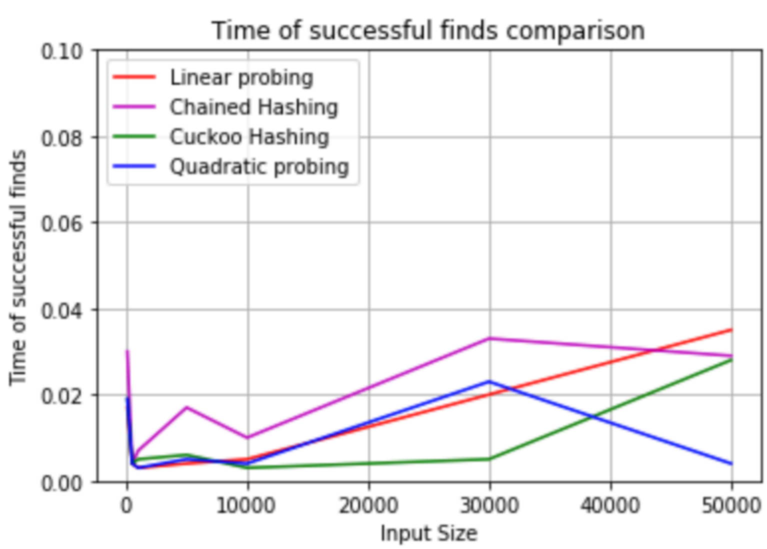


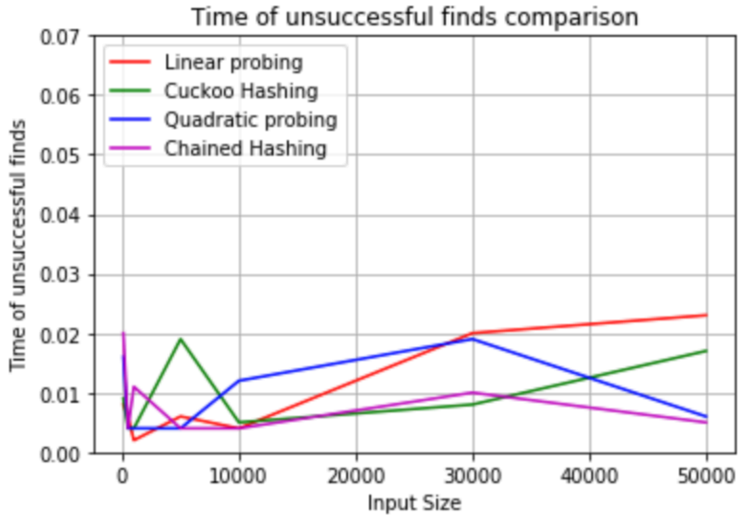
Time for successful & unsuccessful finds –

1. Linear Hashing takes more time to find an element if there are more collisions as it probes every next element.

2. Quadratic Hashing which is improvement to linear takes the least time to find as it reduces probing time.

3. Chaining and double hashing can have erratic behavior in worst cases and have more or less similar find times for larger values.





Time of unsuccessful finds comparison
Time of unsuccessful finds
Input Size
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
10000
20000
30000
40000
50000
Linear probing
Cuckoo Hashing
Quadratic probing
Chained Hashing