# Hash Functions Analysis Report

## 1. Individual Analysis of collision algorithms -:

## 1. Hash Chaining

Hash Function used = key % sizeofTable.

**Theoretical analysis**

a). Expected time per operation is O(1 + α) where α is load factor.

b) This algo requires extra space in the form of a linked list to store the key value pairs that collide to a single index after hashing

c). All the operations take almost the same time.

**Pros**

a) It works and is simple

**Cons**

a) Extra space for collisions

b) Extra time for two level access.

**Pseudo-Code**

**Search(k)**

```
i = h(k)
While H[i] != null && H[i] != k
        H[i] = H[i]->next
If H[i] != null, return H[i]->val
Else throw exception
```

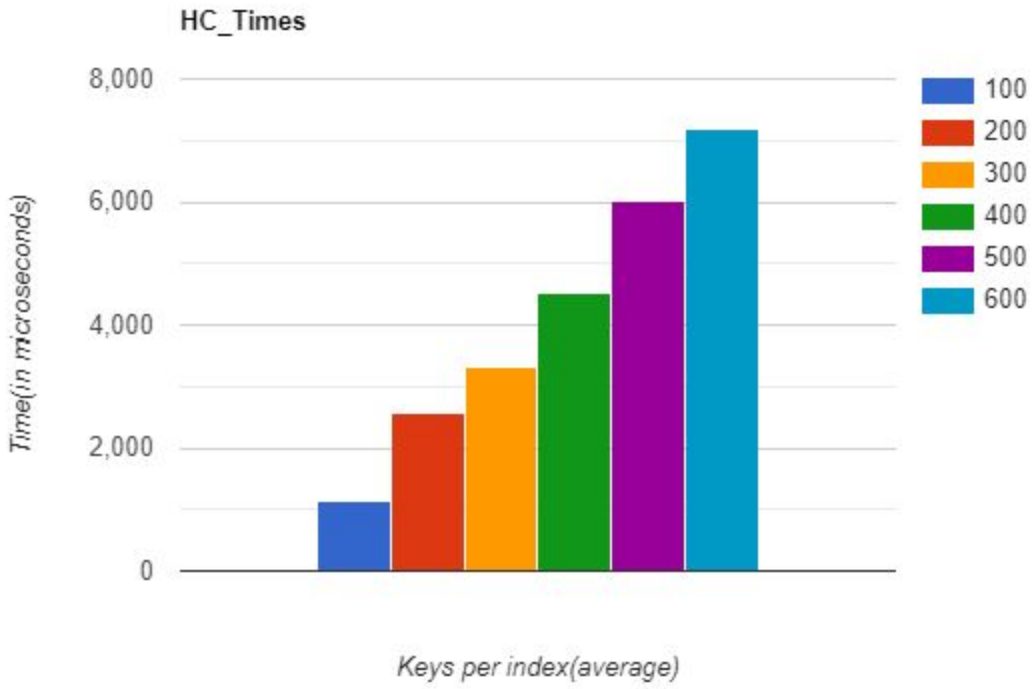
**Set(k,v)**

```
i = h(k)
While H[i] != null && H[i] != k
        H[i] = H[i]->next
If H[i] != null, H[i]->val = v          //update the value
Else    Insert new key(k) value(v) pair at the end of the list
```

**Delete(k)**

```
i = h(k)
While H[i] != null && H[i] != k
        H[i] = H[i]->next
If H[i] != null, delete the cell H[i].current
Else
        Throw exception
```

Runtimes of Hash Chaining with different average keys per index(load factor)




As you can see, the run time of inserting 1000 keys for different values of average load factors is shown above. We can see a linear relationship between time of operations and load factor in hash chaining, as was expected( Expected runtime of any operation in hash chaining is O(1+α) )

## 2. Linear Probing

Hash Function used = key % sizeofTable.

**Theoretical analysis -:**

a) Worst case search time is bad.

b) Time for search(k) operation ≤ length of largest full contiguous block of cells containing H[h(k)].

c) Average search time hashing time is constant but it depends on the load factor.

d) For successful search -:

$$\Theta\ (1 + 1/(1 - \alpha))$$

For unsuccessful search -:

$$\Theta\ (1 + 1/(1 - \alpha)^2)$$

**Pros**

a) It works and it is simple

b) Single-level store: space efficient, sequential memory is fast.

**Cons**

a) Degrades badly when α gets close to 1.

**Pseudocode**

**Search(k)**

```
i = h(k)
while H[i] is nonempty and contains a key != k
i = (i + 1) mod N
if H[i] is nonempty: return its value
else: exception
```

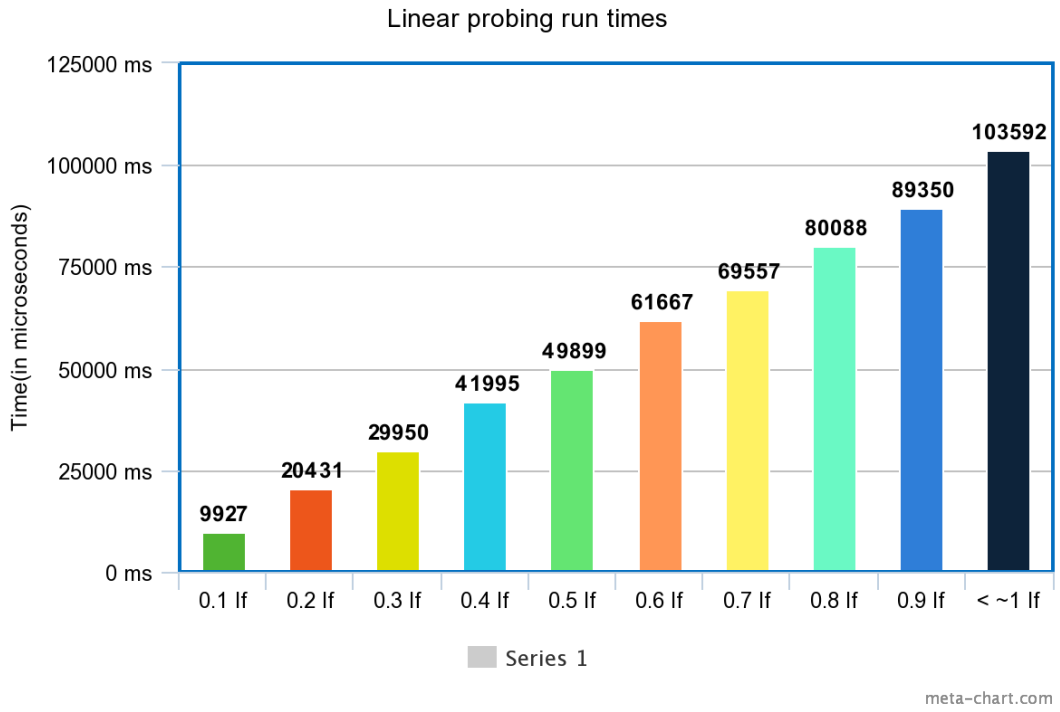
**Set(k,v)**

```
i = h(k)
while H[i] is nonempty and contains a key != k or is not unused
i = (i + 1) mod N
store (k,v) in H[i]
```

**Delete(k)  (I implement lazy delete)**

```
i  = h(k)
while H[i] is nonempty and contains a key != k
i = (i + 1) mod N
if H[i] is nonempty: delete the key value pair
else: exception
```

Runtimes of Linear Probing with different load factors -:




The above graph show time taken to insert 100 keys in a hashtable of size 100000 at different periods of the load factor. As can be seen from above, that the expected time per operation linearly increases with the load factor. However, when the load factor tends to 1, the performance degrades substantially.

## 3. Cuckoo Hashing

(both tables are part of one large table of size sizeOfTable)

Hash Functions used = key % (sizeOfTable/2)
(key/(sizeOfTable/2)) % (sizeOfTable/2) + sizeOfTable/2

**Theoretical analysis**

a). Any sequence of n operations takes O(1) expected time per operation
b) With probability 1 − Θ(1/n), Cuckoo works.
c) With probability Θ(1/n), we need to rehash and rebuild. .

**Pros**

a) Guaranteed O(1) search and delete

**Cons**

a) Slower set operation
b) More complex to implement.

**Pseudo-Code**

**Search(k)**

```
index1 = h1(k), index2 = h2(k)
if(H[index1] == key || H[index2] == key)
        return value of index that contains key
else
        Throw exception
```

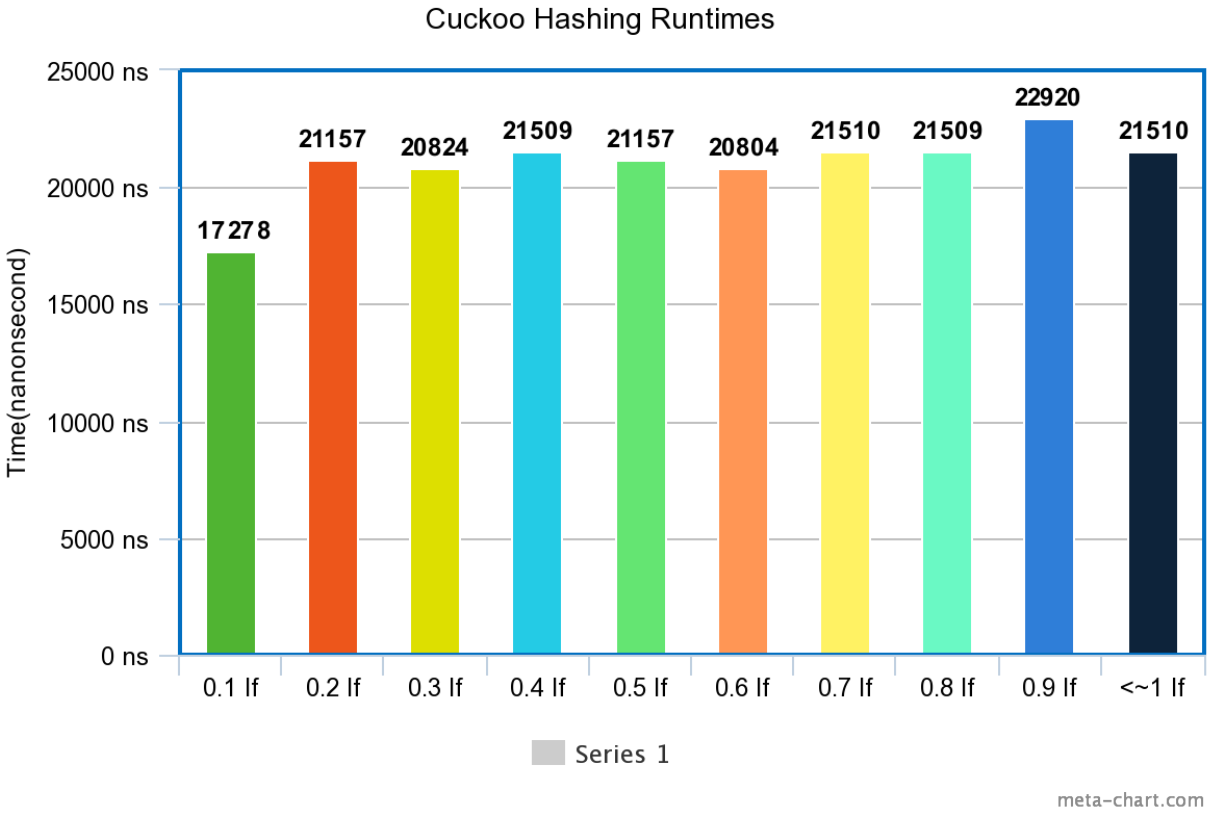
**Set(k,v)**

```
def set(k,v):
t = 0
while (k,v) is a nonempty pair:
        (k,v) ↔ Ht [ht(k)]
        t = 1 - t
```

**Delete(k)**

```
index1 = h1(k), index2 = h2(k)
if(H[index1] == key || H[index2] == key)
        Delete the cell that contains key
else
        Throw exception
```

Runtimes of Cuckoo Hashing with different load factors -:




The above graph show time taken to insert 100 keys in a hashtable of size 100000 at different periods of the load factor. As expected, the time to insert keys is quite independent of the load factor but rather depends more on the how many keys are kicked out and replaced(which depends on how well the hash function is designed).

## 4. Quadratic Probing

### Reasons why this algorithm is chosen

1. The performance is much better for quadratic probing than linear probing when load factor is way less than 1.
2. Doesn't require extra space as compared to hash chaining.
3. Simpler logic than cuckoo hashing and hash chaining.

Hash Functions used = key % sizeOfTable

**Theoretical analysis -:**

a) Worst case search time is bad.
b) Average search time is constant but depends heavily on load factor.
c) Doesn't require extra space as compared to hash chaining.

**Pros**

a) It works and it is simple
b) Single-level store: space efficient, sequential memory is fast.
c) Performance is much better than linear probing when load factor is way less than 1.

**Cons**

a) Degrades very badly when α gets close to 1.

**Pseudocode**

**Search(k)**

```
i = h(k) count = 1
while H[i] is nonempty and contains a key != k
i = (i + count^2) mod N; count++;
if H[i] is nonempty: return its value
else: exception
```

**Set(k,v)**

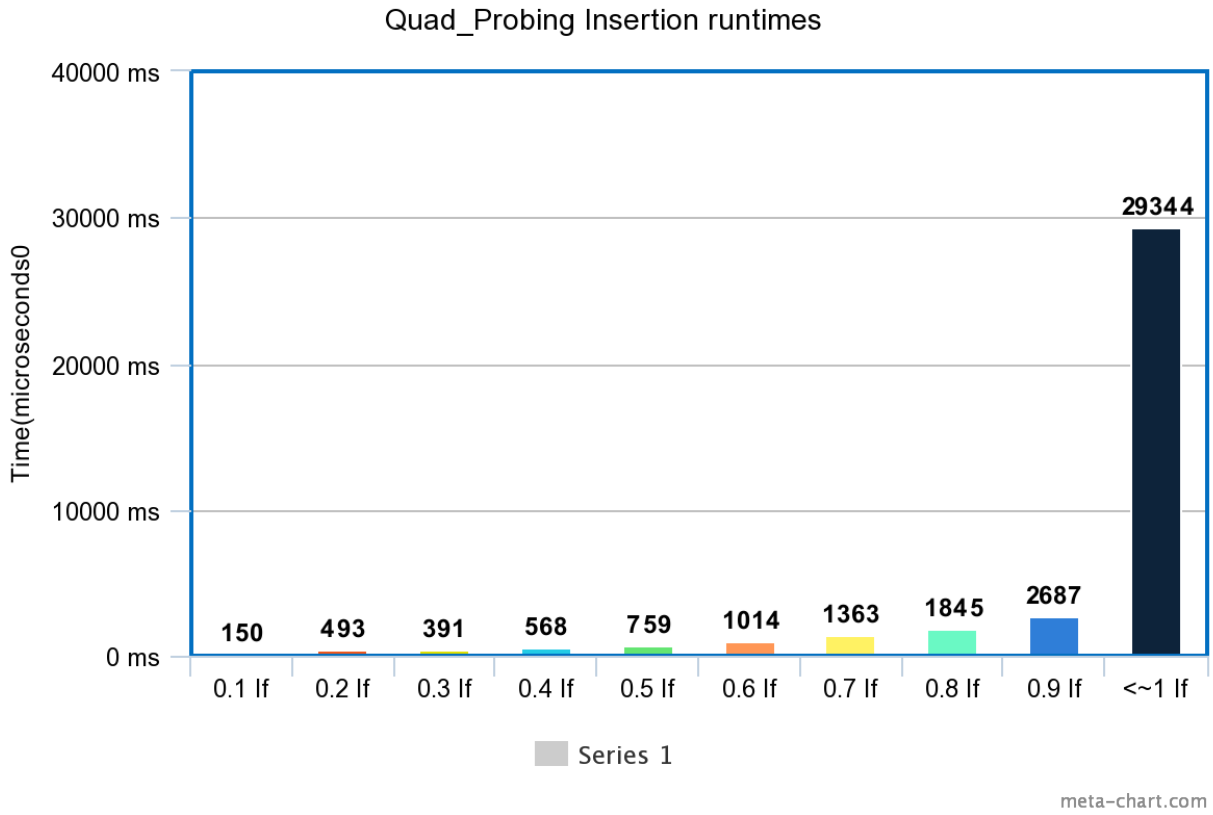
```
i = h(k) count = 1;
while H[i] is nonempty and contains a key != k or is not unused
i = (i + count^2) mod N ; count++;
store (k,v) in H[i]
```

**Delete(k) (I implement lazy delete)**

```
i  = h(k) ount = 1;
while H[i] is nonempty and contains a key != k
i = (i + count^2) mod N ; count++
if H[i] is nonempty: delete the key value pair
else: exception
```

Note-: I run the loops in quadratic probing for 10 * currentNoOfElements as I am want to avoid the case of infinite loops. (This may lead to some fallacies where the key cannot be found in this number, so this algo is not 100% correct

Runtimes of Quadratic Probing with different load factors -:




The above graph show time taken to insert 100 keys in a hashtable of size 100000 at different periods of the load factor. As can be seen, the degradation in performance is substantially worse as load factor tends to 1 in quadratic probing as compared to linear probing. However, the performance is much better for quadratic probing than linear probing when load factor is way less than 0.8. So, it is a better option than linear probing when load factor is low.

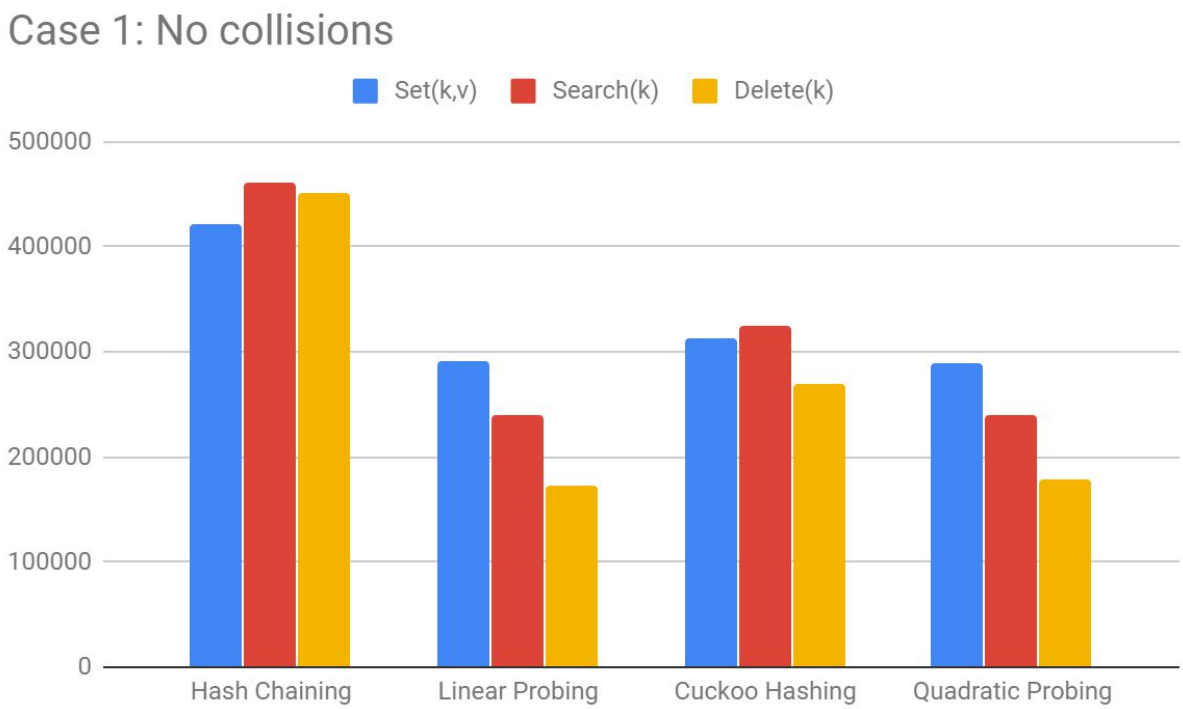
## 2. Comparative Analysis of Collision Algorithms -:

Here we analyze the various performances of the 4 above mentioned collision techniques based on various inputs and see how the individual operations of the 4 techniques perform.
Table size is 1000 cells for each collision technique

**Case 1(No collisions)**: no collisions occur in this case as 1000 sequential keys(from 1 to 1000) are inserted in table of size 1000. This is just to set the reference for the future test cases and see how they perform as compared to this base case. It itself is not very helpful when it comes to analyzing these 4 techniques

(All times are in nanoseconds)

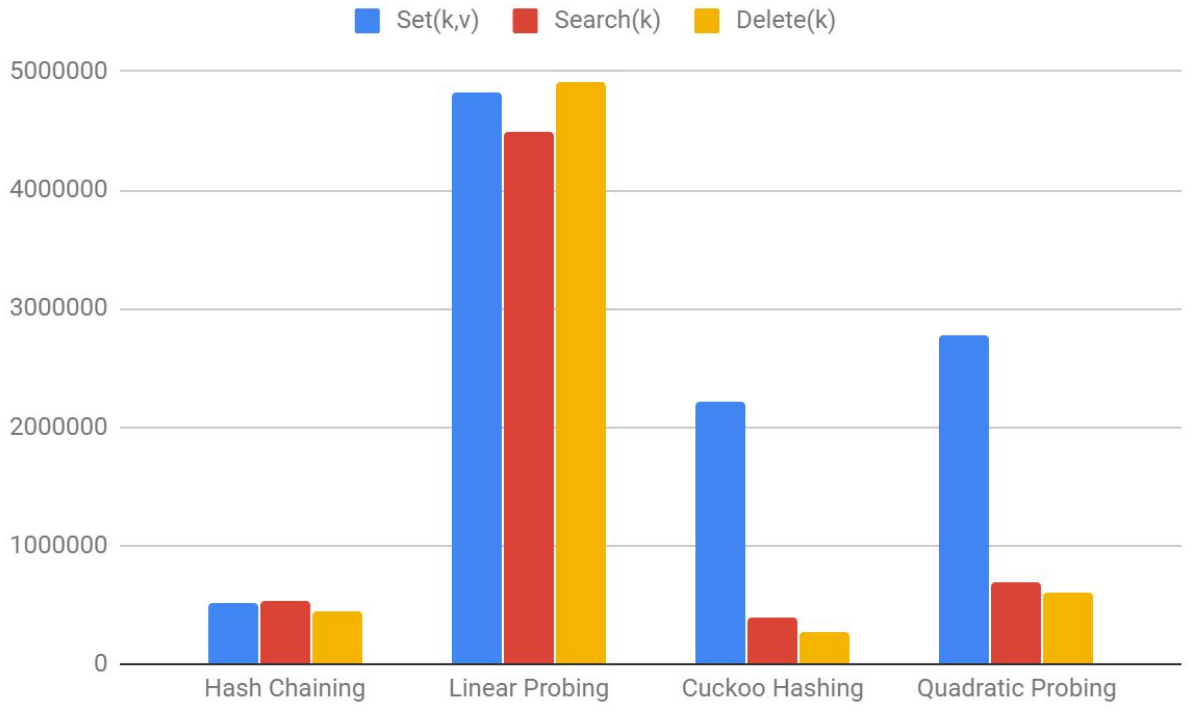
| Operations -> Collision technique \|\| | Set(k,v) | Search(k) | Delete(k) |
|---|---|---|---|
| Hash Chaining | 421024 | 461994 | 450357 |
| Linear Probing | 291966 | 240132 | 172077 |
| Cuckoo Hashing | 312418 | 325112 | 269398 |
| Quadratic Probing | 289850 | 240837 | 178777 |




For set(k,v)-: Quadratic Probing performs the best and Hash Chaining performs the worst
For search(k)-: Linear Probing performs the best and Cuckoo Hashing performs the worst
For delete(k)-: Linear Probing performs the best and Cuckoo Hashing performs the worst.
No conclusions can be drawn from the case where there are no collisions

**Case 2( Collisions with keys that have some order)**In this case again table size is 1000 and key number is also 1000. But, there are keys that collide as they are hashed to the same index over and over again. Let's see how the techniques perform in this case.

| Operations -> Collision technique \|\| | Set(k,v) | Search(k) | Delete(k) |
|---|---|---|---|
| Hash Chaining | 523988 | 528991 | 453883 |
| Linear Probing | 4830136 | 4497619 | 4906301 |
| Cuckoo Hashing | 2212456 | 392462 | 269398 |
| Quadratic Probing | 2768389 | 688659 | 611083 |



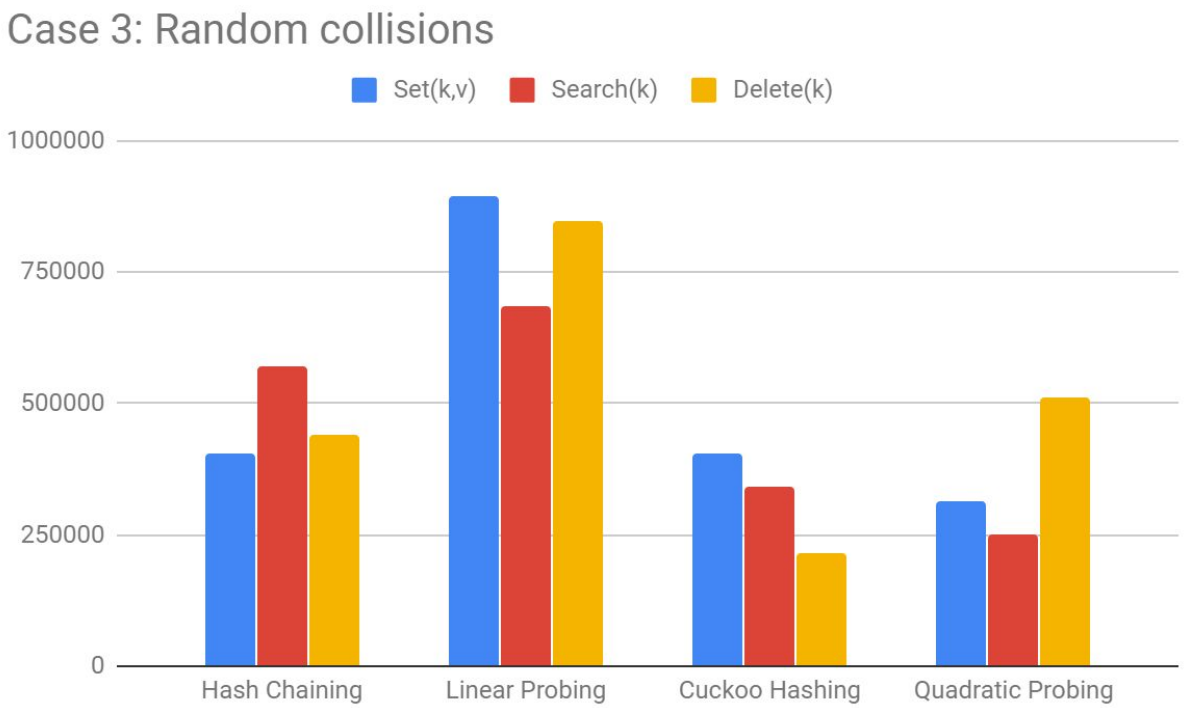

For set(k,v)-: Hash Chaining performs the best and Linear Probing performs the worst
For search(k)-: Cuckoo Hashing performs the best and Linear probing performs the worst
For delete(k)-: Cuckoo Hashing performs the best and linear probing performs the worst

Conclusions -:

1. From this data, we can analyze the **usefulness of cuckoo hashing**. As can be seen, it has no competition when it comes to searching or deleting keys (both Guaranteed O(1) if they exist). It easily beats its competitors when there is collision involved.
2. Almost all other techniques have become much worse when collision is involved as compared to the scenario when there was no collision.
3. Linear probing should be avoided in this case.
4. Hash Chaining is a viable option with its times almost the same across all operations.(Cuckoo doesn't do well when it comes to set(k,v)).

**Case 3(Random keys that collide with the same index)**.
In this scenario also, we have table of size 1000. And the number of keys that are to be inserted are also 1000. However, the keys are randomized(with modulo 1000) and have no order to them. Lets see how these techniques perform in this scenario.

| Operations -> Collision technique ‖ | Set(k,v) | Search(k) | Delete(k) |
|---|---|---|---|
| Hash Chaining | 406214 | 568694 | 439779 |
| Linear Probing | 895550 | 684504 | 848123 |
| Cuckoo Hashing | 405508 | 341685 | 215096 |
| Quadratic Probing | 315239 | 248947 | 512351 |




For search :- Quadratic Probing performs best and linear probing performs worst.
For set -: Quadratic probing performs best and linear probing worst.
For delete-: Cuckoo hashing performs best and linear probing worst.

The performance depends on how many collisions are there for the random keys generated. It looks like that the number of collisions is less in the above case. So performance is not so bad.

# Final conclusions

As can be seen from the various cases above, each collision technique has its merits and demerits..

When a lot of collisions are involved and main operations are search and delete, then cuckoo hashing should be the preferred technique for collision resolution.

Hash Chaining, though expensive in terms of space usage and complexity, is usually a good option in a variety of scenarios.

Quadratic Probing beats linear probing in various situations and can be used if auxiliary space is limited.

If the number of collisions is few and far between, then linear probing can be used.