ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 5, Issue 2, pp: (1-11), Month: April - June 2017, Available at: www.researchpublish.com

Evolution of Cuckoo Hashing

Nitesh Gupta

Amadeus Labs, Bangalore, India

Abstract: **We will present a simple cuckoo hashing with an example of how it works. We will also present the problem that can come in this algorithm. We will also present the evolution of this algorithm and will see if it solves the problem or how much the problem in minimized.**

Keywords: **Hashing, Collisions, Cuckoo Hashing, Memory Management.**

I. INTRODUCTION

Cuckoo Hashing is an algorithm for resolving hash collisions of the values of the hash functions in the table and enhance the worst case lookup time.

Algorithm:

1. Let F (n) used for hashing is a hash function and initial assignment is F (1).
2. Hash key (HK) with Function F (n).
3. Check if the place is empty or not. If place is empty, push the entry into location and terminate.
4. If place is not empty push the existing entry into TEMP.
5. Place HK into the location.
6. Check which function is used to place TEMP into the location.
7. Assign TEMP back to HK.
8. Assign alternate hash function into F (n).
9. Repeat from step 2 until there is no collision.

Now let's see how this algorithm solve the above collision problem. Let's say first all the keys are placed in empty locations (except JS). Now key JS came for the entry. It is hashed with function 1 and location is derived as 03. Now when the location is checked if it is free or not, it says that there is one entry stored at that location. Then entry (NG) is pushed out of the location and JS is stored at that location. Then this NG is hashed again and a new location is derived as 11. Now this new location is checked if it is free or not. Since location 11 is free, NG is placed at this location.

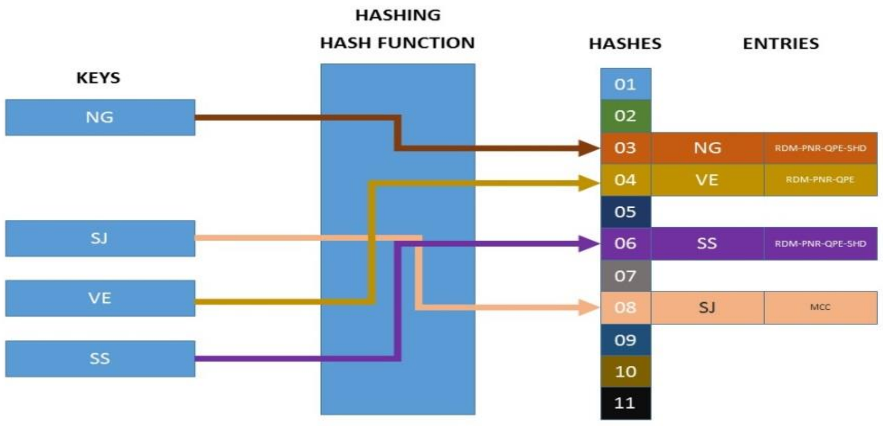

Fig. 1

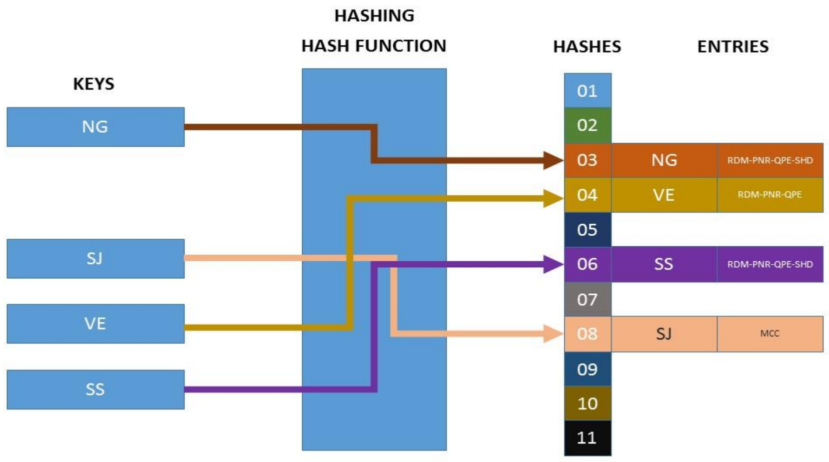

Fig. 2

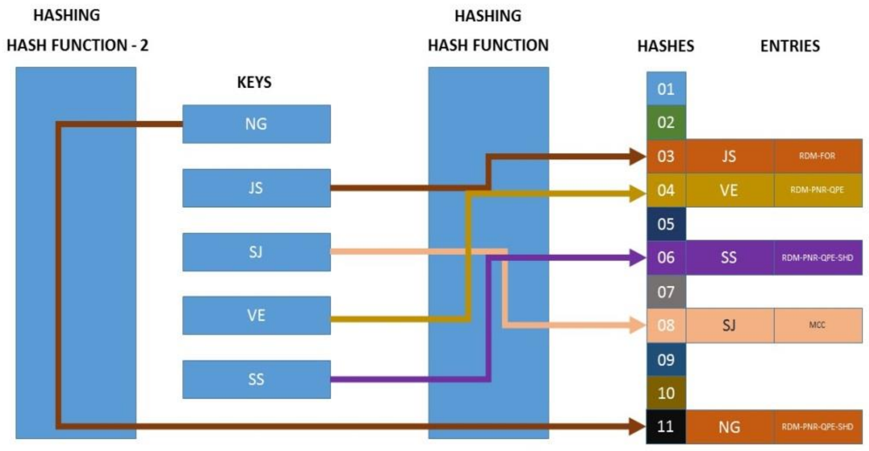

Fig. 3

II. PROBLEM WITH CUCKOO HASHING

As we have discussed earlier, cuckoo hashing needs two hash functions for avoiding collision. Now there is a possibility that collision occurs in such a way the algorithm enters into an infinite loop. This is one of the biggest problem with Cuckoo Hashing as these kind of problem can come with a probability of 20%.

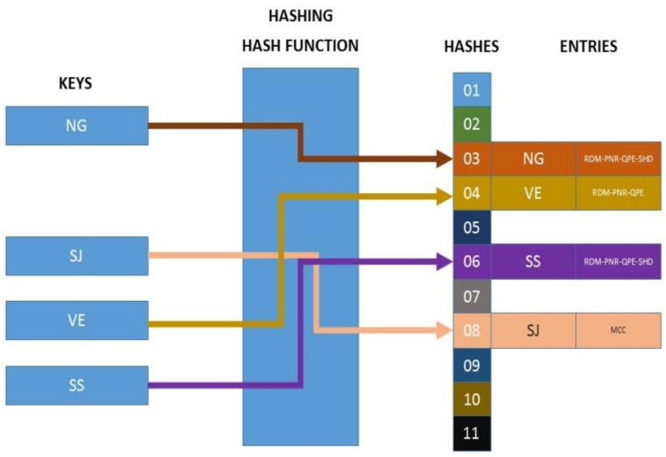

Fig. 4

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 5, Issue 2, pp: (1-11), Month: April - June 2017, Available at: www.researchpublish.com

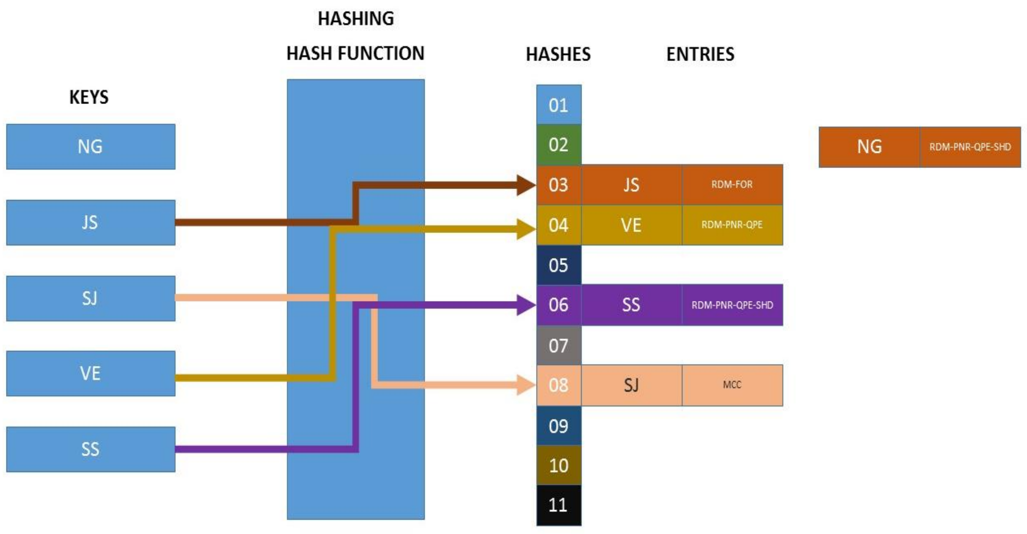

Fig. 5

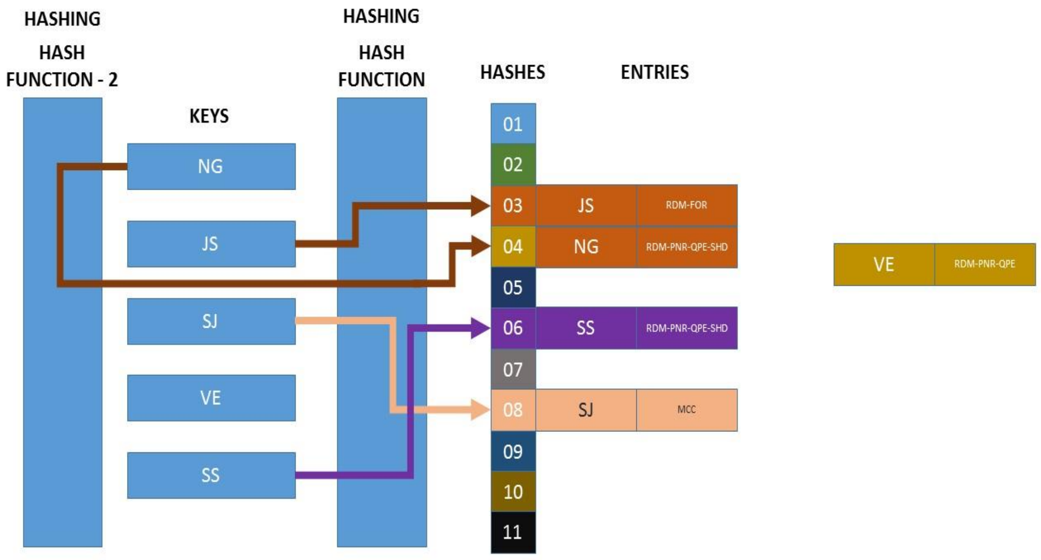

Fig. 6

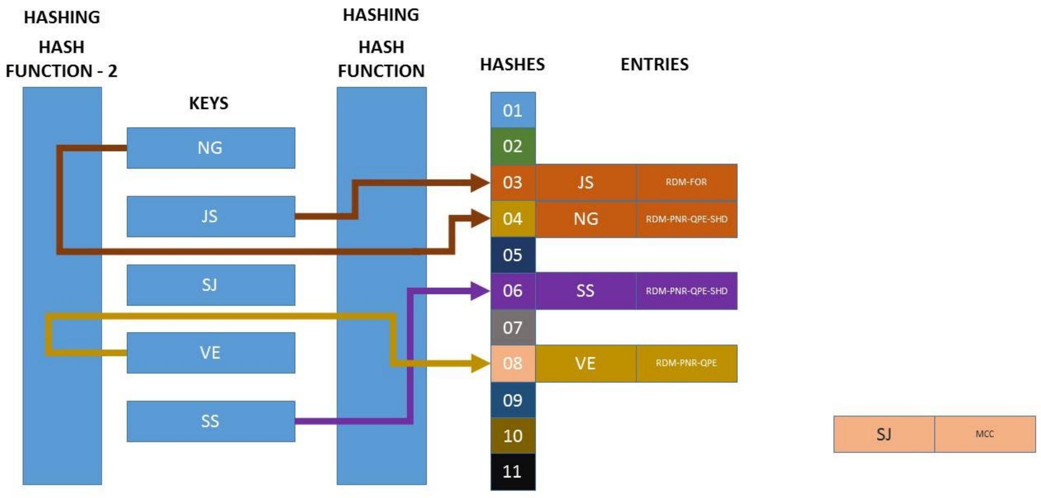

Fig. 7

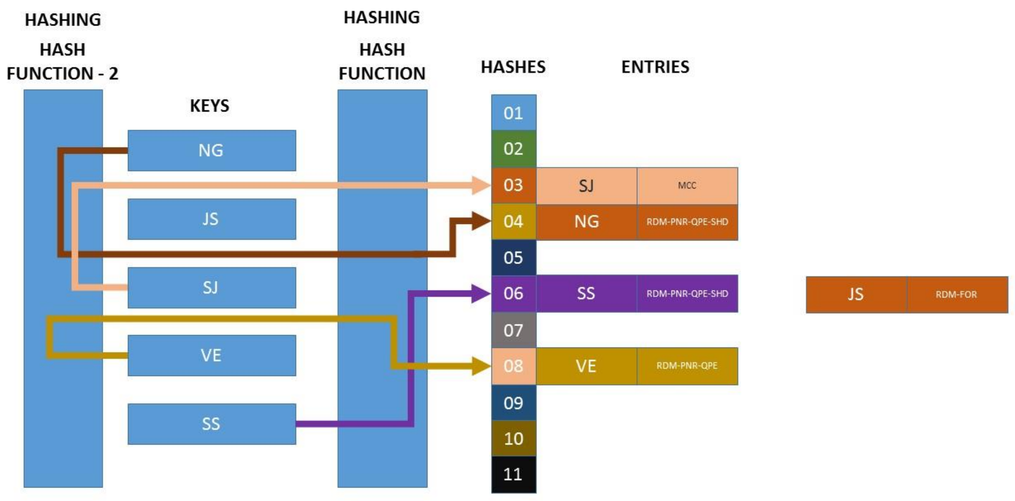

Fig. 8

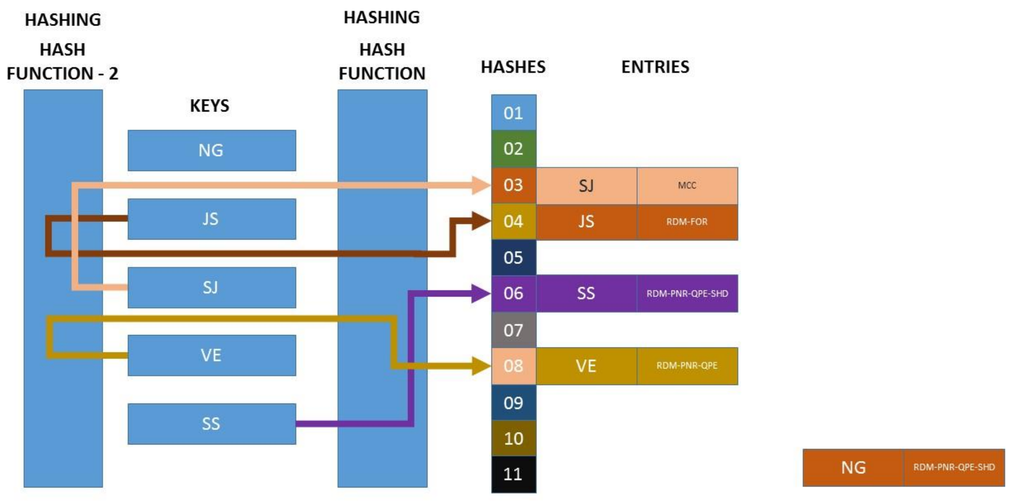

Fig. 9

III. EVOLUTION OF CUCKOO HASHING

Cuckoo Hashing, as we already know, use two hash functions which guarantees that a key must be present in one of the two memory locations derived by the two functions. But since the probability of loop is 20%, let’s discuss what will happen if we evolve the algorithm by adding more hash functions.

A. Cuckoo Hashing with three functions:

In this algorithm, instead of having two hash functions we will use three hash functions. Complexity of finding one element can be maximum of 3 locations, hence can be stated as O (1).

Algorithm:

1. Hash a key with first function F (1) to F (n) and get memory location M (n).
2. Check if M (n) is empty or not. If yes, then place the entry into the location and terminate.
3. If M (n) is not empty, then place the existing entry of M (n) to TEMP and new entry to M (n).
4. Get the function f (k) which is used hash key present in TEMP.
5. If f (k) is equals to F (1), assign F (2) to F (n), else if f (k) is equals to F (2), assign F (3) to F (n), or else assign F (1) to F (n).
6. Repeat from step 2 until there is no collision.

Now let's see understand this algorithm with an example. Let's say first all the keys are placed in empty locations (except JS). Now key JS came for the entry. It is hashed with function 1 and location is derived as 03. Now when the location is checked if it is free or not, it says that there is one entry stored at that location. Then entry (NG) is pushed out of the location and JS is stored at that location. Now NG is checked which function was used previously to get the location. We found that function 2 was used to hash it previously, it is hashed with function 3 to get the new location 11. Now since the new location is free NG is placed to that location.

Let's say a new entry as RK came. It is hashed with function 1 and location 10 is derived. Now since NG was already placed in this location, it is pushed out and RK is placed at location 11. Now we is again checked which function was used to previously to get location and found that function 3 was used. It will be hashed by function 1 to get the location 4, since VE is already placed at this location VE is pushed out and NG take location 4 (we can note that location 4 was and must be the previous location of NG and filled by some other entry because every entry was first hashed with function 1 and will move from its location only if any other entry takes the position). Now we will hash VE with function 2 and a new location 07 is derived, since 07 is empty, we place VE at 07.

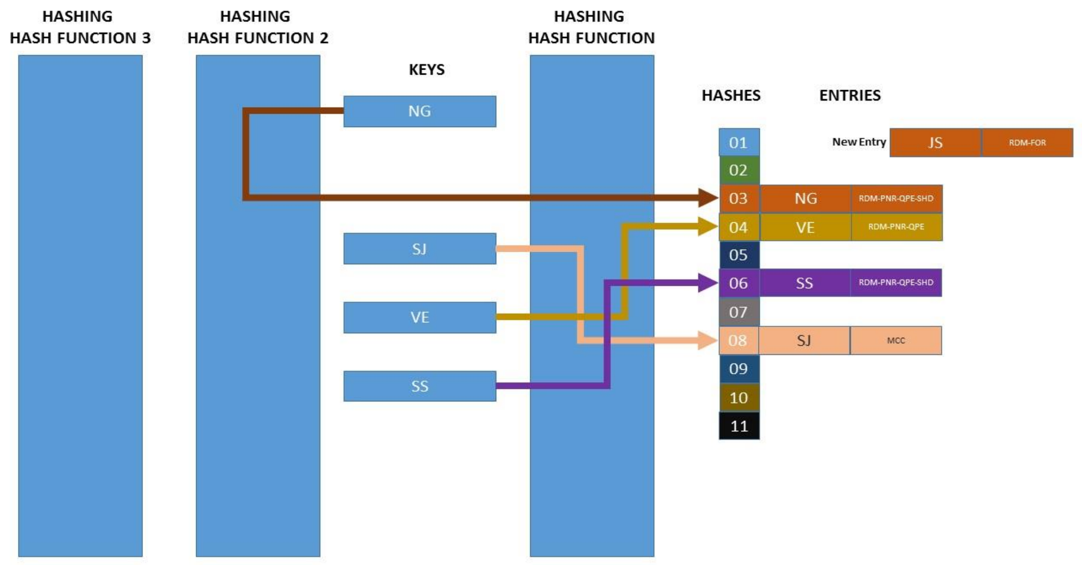

Fig. 10

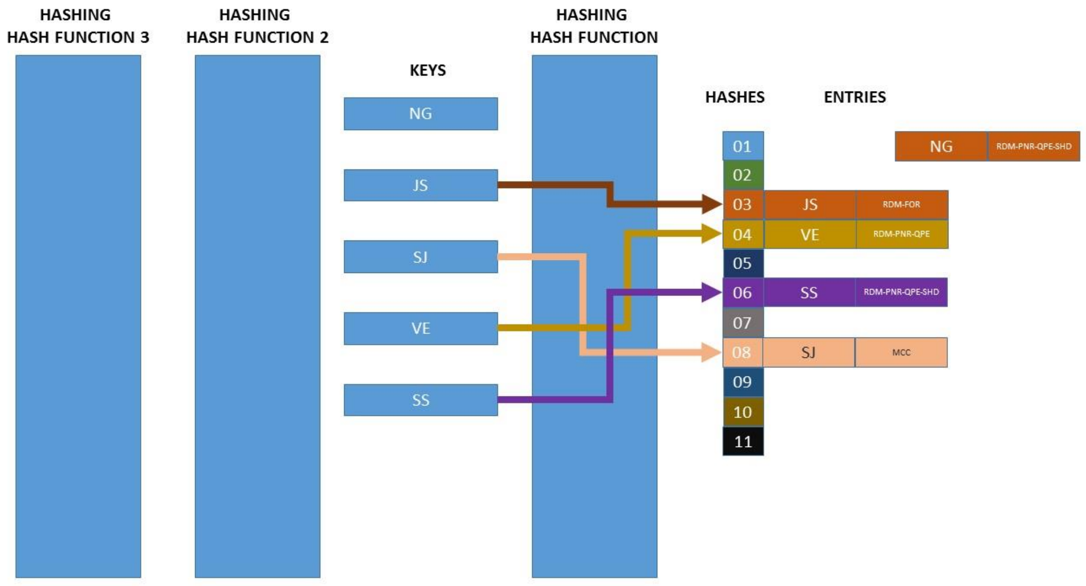

Fig. 11

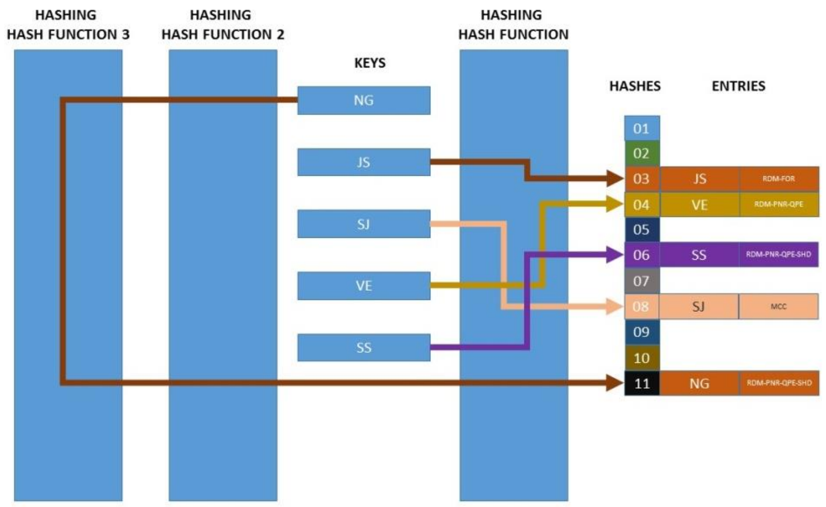

Fig. 12

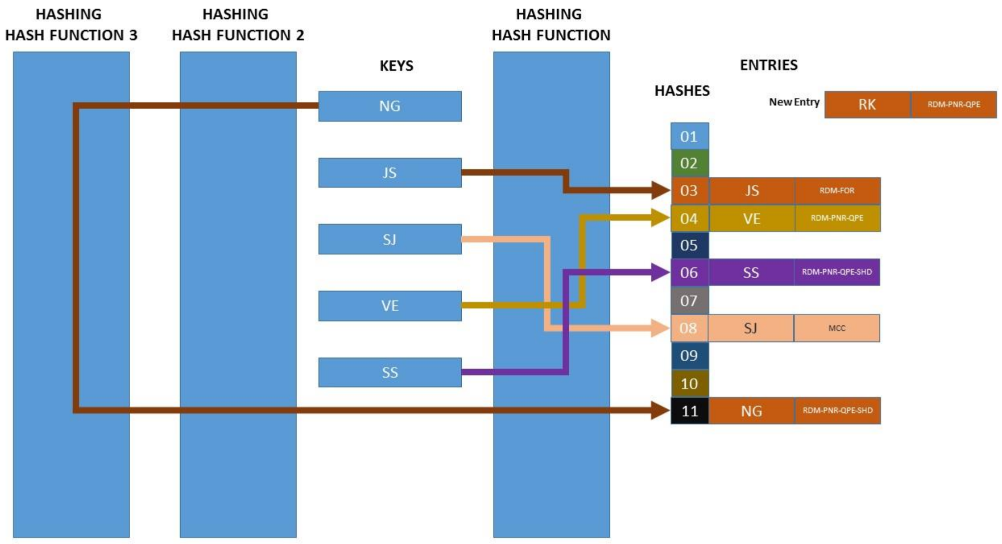

Fig. 13

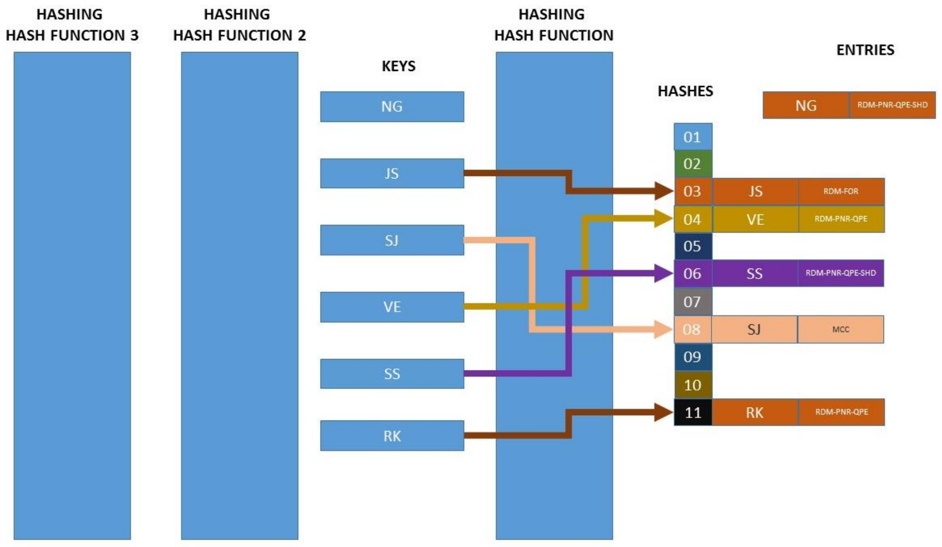

Fig. 14

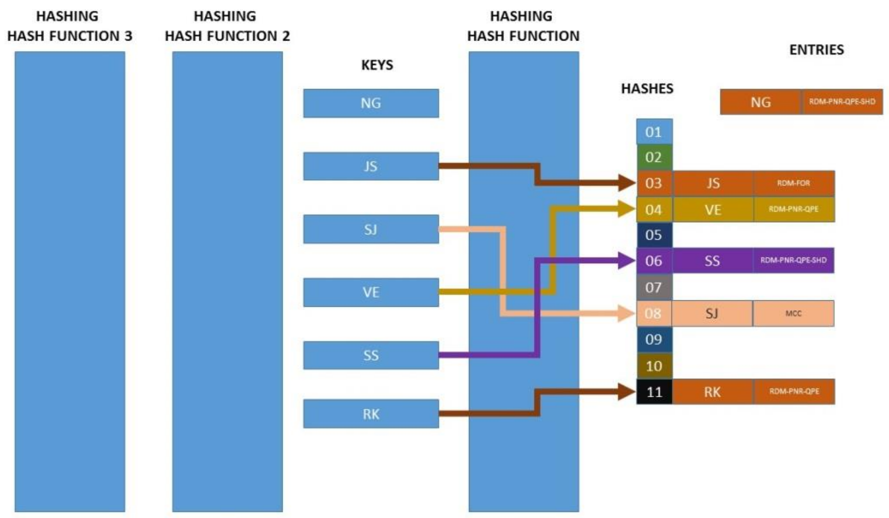

Fig. 15

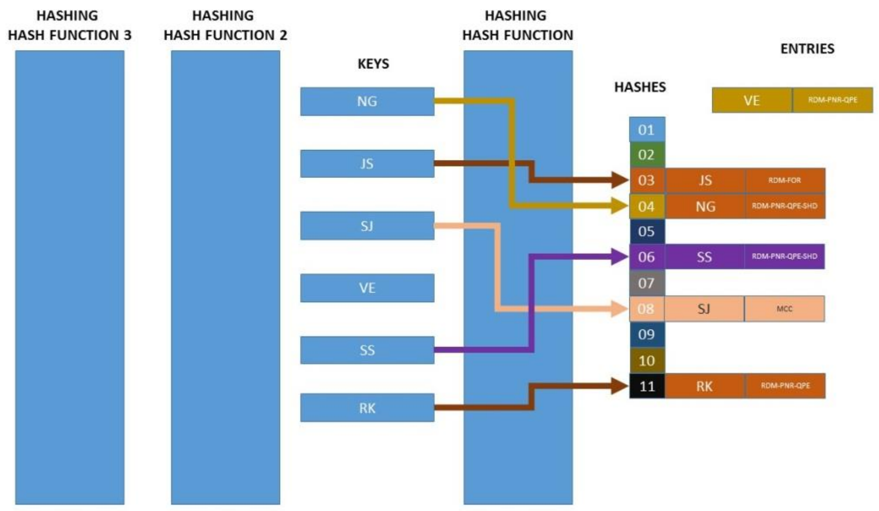

Fig. 16

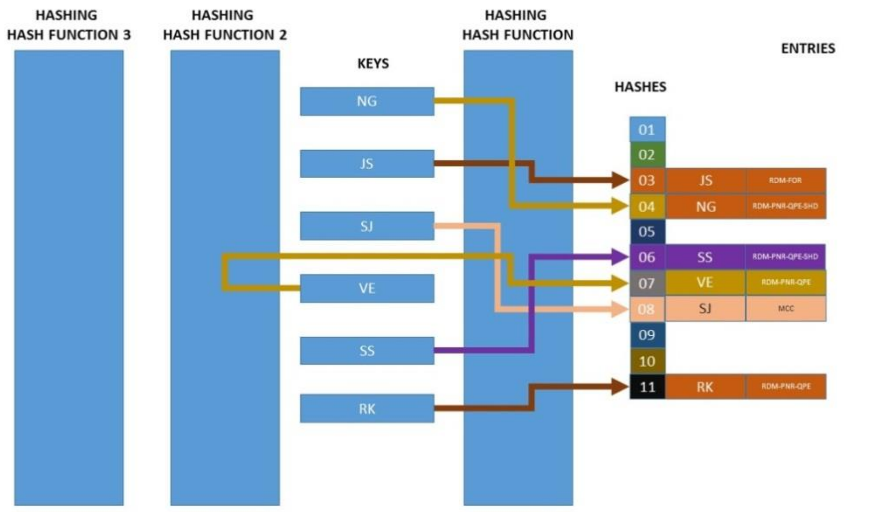

Fig. 17

This algorithm reduces the possibility of occurrence of loop in hash table. But this algorithm also doesn't guarantee that loop will never come.

B. Cuckoo Hashing with four functions:

In this algorithm, instead of having two hash functions we will use four hash functions. Complexity of finding one element can be maximum of 4 locations, hence can be stated as O (1).

Algorithm:

7. Hash a key with first function F (1) to F (n) and get memory location M (n).
8. Check if M (n) is empty or not. If yes, then place the entry into the location and terminate.
9. If M (n) is not empty, then place the existing entry of M (n) to TEMP and new entry to M (n).
10. Get the function f (k) which is used hash key present in TEMP.
11. If f (k) is equals to F (1), assign F (2) to F (n), else if f (k) is equals to F (2), assign F (3) to F (n), else if f (k) is equals to F (3), assign F (4) to F (n), or else assign F (1) to F (n).
12. Repeat from step 2 until there is no collision.

Now let's see understand this algorithm with an example. Let's say first all the keys are placed in empty locations (except JS). Now key JS came for the entry. It is hashed with function 1 and location is derived as 03. Now when the location is checked if it is free or not, it says that there is one entry stored at that location. Then entry (NG) is pushed out of the location and JS is stored at that location. Now NG is checked which function was used previously to get the location. We found that function 3 was used to hash it previously, it is hashed with function 4 to get the new location 11. Now since the new location is free NG is placed to that location.

Let's say a new entry as RK came. It is hashed with function 1 and location 11 is derived. Now since NG was already placed in this location, it is pushed out and RK is placed at location 11. Now we is again checked which function was used to previously to get location and found that function 4 was used. It will be hashed by function 1 to get the location 4, since VE is already placed at this location VE is pushed out and NG take location 4 (we can note that location 4 was and must be the previous location of NG and filled by some other entry because every entry was first hashed with function 1 and will move from its location only if any other entry takes the position). Now we will hash VE with function 2 and a new location 07 is derived, since 07 is empty, we place VE at 07.

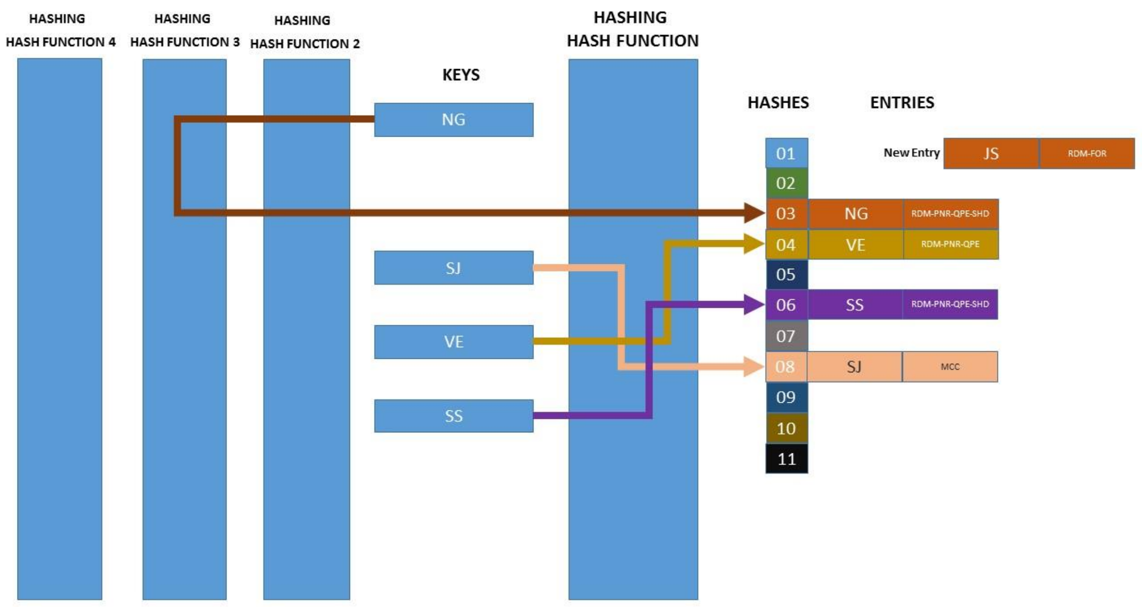

Fig. 18

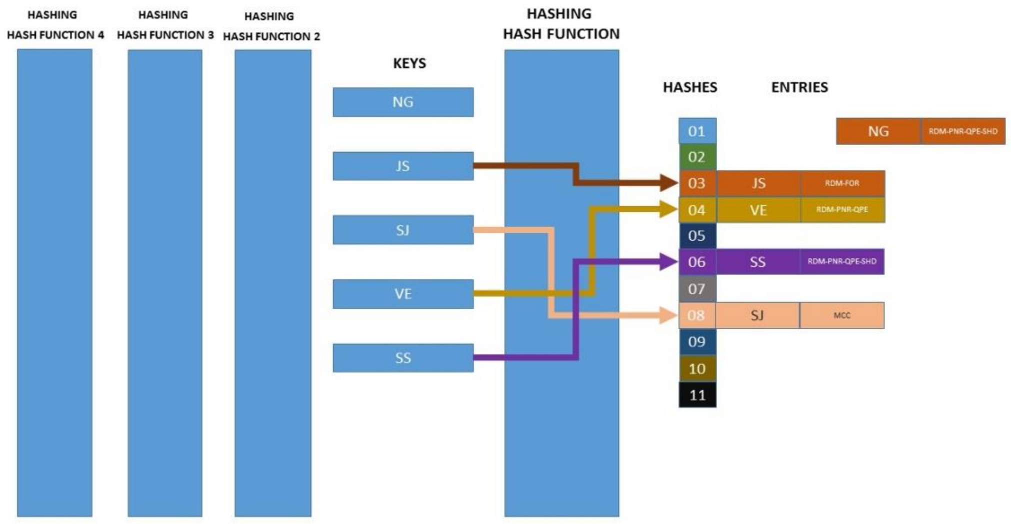

Fig. 19

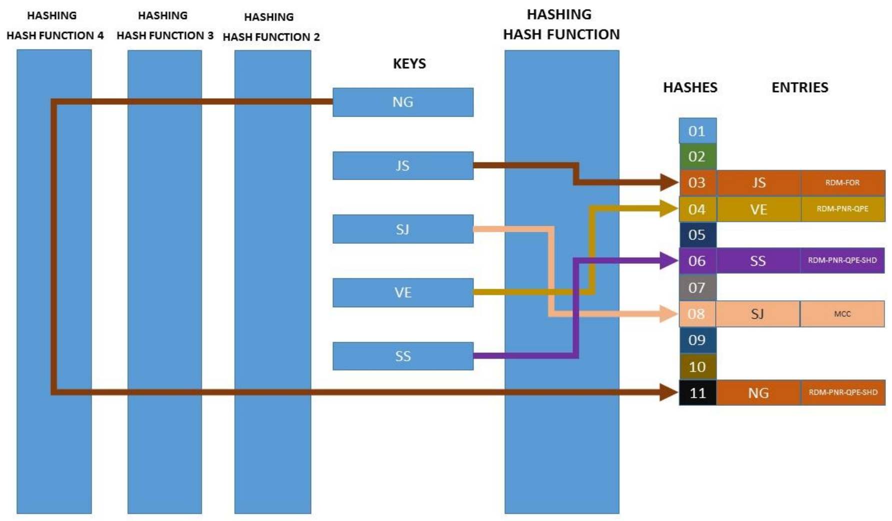

Fig. 20

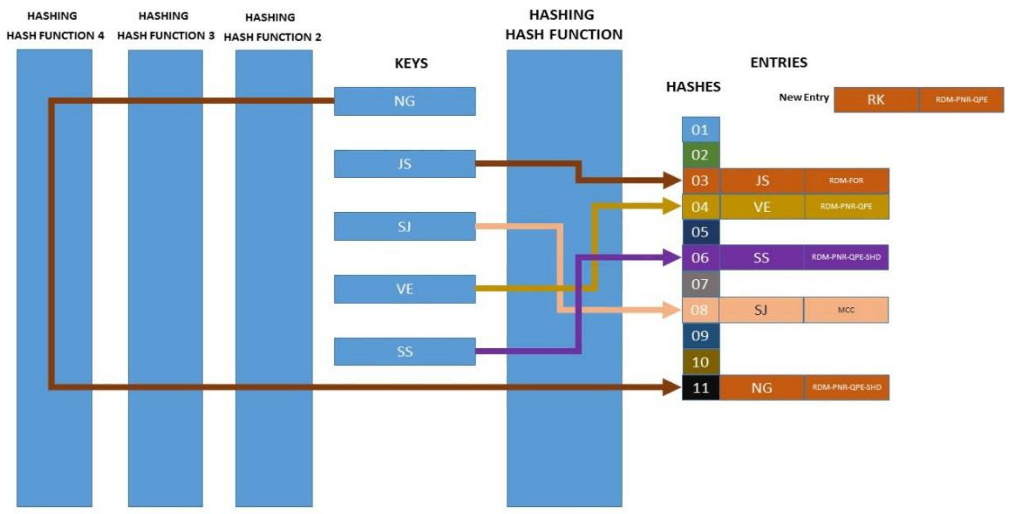

Fig. 21

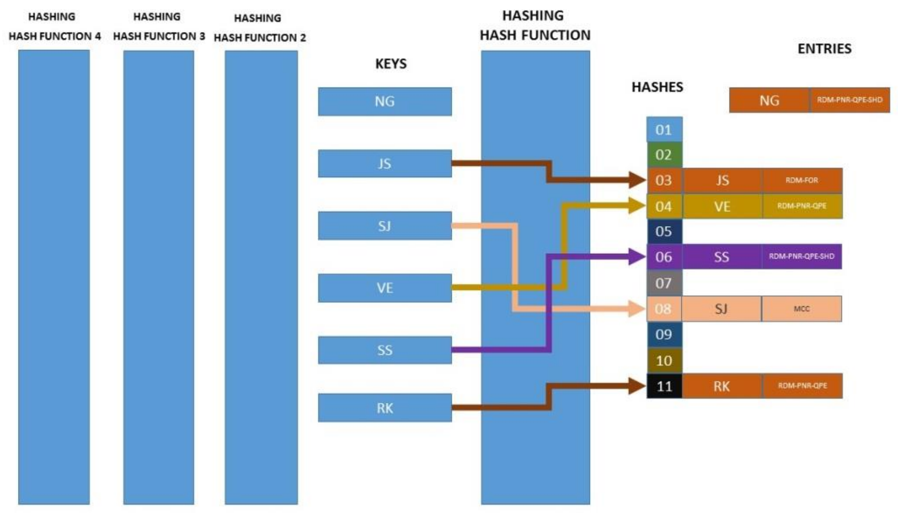

Fig. 22

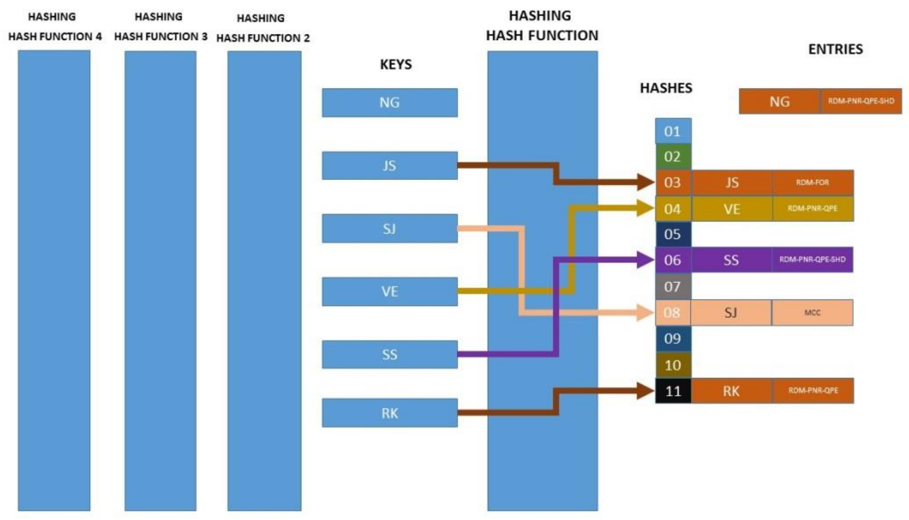

Fig. 23

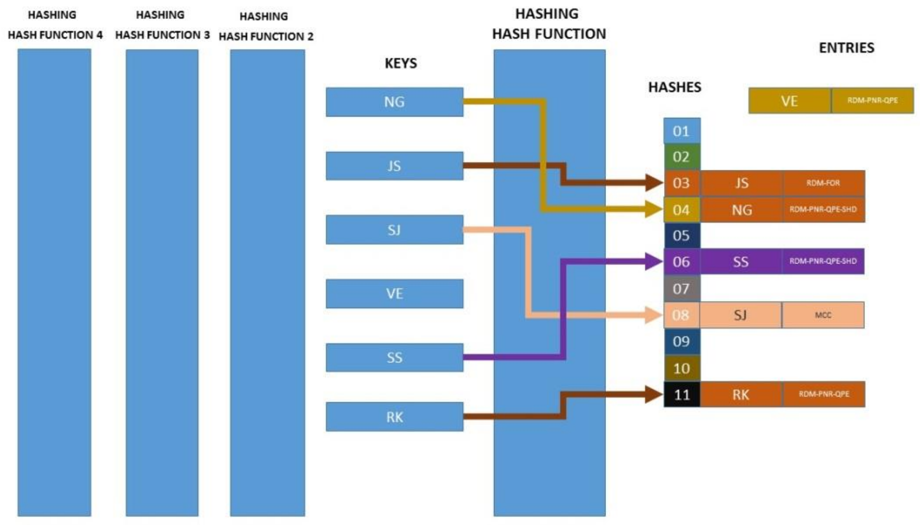

Fig. 24

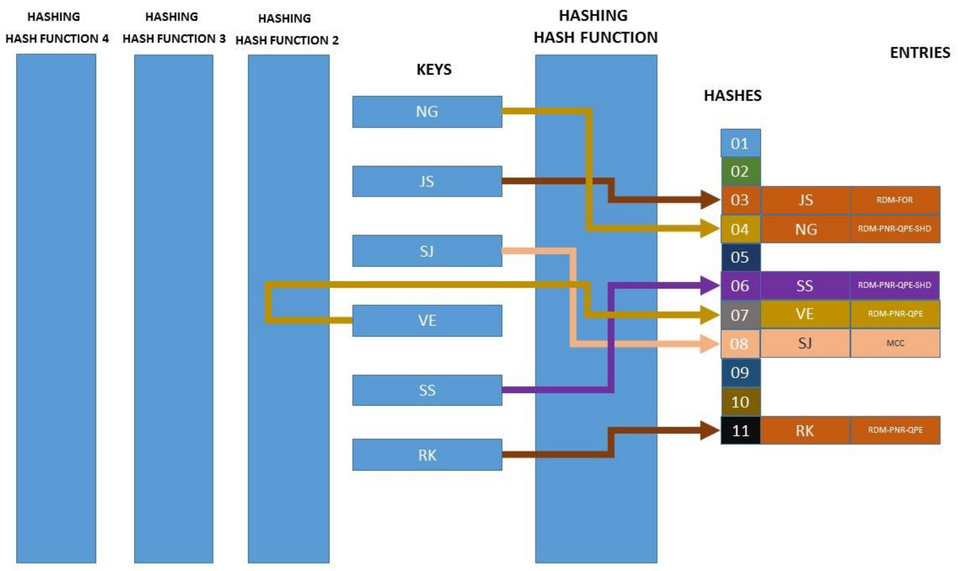

Fig. 25

This algorithm is also way more efficient than any of the above algorithm and also reduce the probability of the occurrence of loop from 20% to 8%. Now that is a huge reduction but still there is some probability of occurrence of the loop which can't be ignored in large systems where memory management is critical.

IV. CONCLUSION

Above we have discussed what is cuckoo hashing, its nature and the problem that can come is cuckoo hashing is used in any system. We discussed the various evolutions of Cuckoo Hashing where instead of just using two hash functions we use three and four hash functions. We can increase number of functions based on the requirement thus reducing the probability of occurrence of loop. But still we can conclude that Cuckoo Hashing cannot be used in any of the big systems where memory is critical. Not only that if we have to make any system stable, we can't use Cuckoo Hashing for storage purpose as there is no guarantee that a loop can never occur. Hence in this paper, we will conclude by stating that even if Cuckoo Hashing is one of the best algorithm for memory management, still we can't use it.

REFERENCES

[1] Pagh, Rasmus and Rodler, Flemming Friche (2001). "Cuckoo Hashing". Algorithms — ESA 2001. Lecture Notes in Computer Science 2161. pp. 121–133. doi:10.1007/3-540-44676-1_10. ISBN 978-3-540-42493-2.

[2] Knuth, Donald (1998). 'The Art of Computer Programming'. 3: Sorting and Searching (2nd ed.). Addison-Wesley. pp. 513–558.

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). Introduction to Algorithms (3rd ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.

[4] Cuckoo Hashing, Theory and Practice (Part 1, Part 2 and Part 3), Michael Mitzenmacher, 2007.

[5] Algorithmic Improvements for Fast Concurrent Cuckoo Hashing, X. Li, D. Andersen, M. Kaminsky, M. Freedman. EuroSys 2014.