INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 4.295
(Volume3, Issue2)

Cuckoo Hashing & Finding Loop

Nitesh Gupta
Dept. of Computer Science,
OPJS University, Churu,
Rajasthan, India

Dr. Om Prakash
Dept. of Computer Science,
OPJS University, Churu,
Rajasthan, India

Abstract: We will present a simple cuckoo hashing with an example of how it works. Along with it, we will present an algorithm to find a loop in cuckoo hashing.

Keywords: Hashing, Collision, Cuckoo Hashing, Memory Management.

I. INTRODUCTION

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

The name derives from the behavior of the bird CUCKOO where the bird lays eggs in another bird's nest and when these eggs are about to hatch, bird pushes other eggs and young out of the nest.

Algorithm behaves in the same way. If there is a collision then the existing element present in the location is pushed out and the new element takes the place. But then the question comes as where will this existing element go. Then algorithm states that you should have another hash function which takes this element and find a new location to store the key. This process will repeat until every element is placed in an empty location.

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.
4. If the 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 solves 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 has 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.

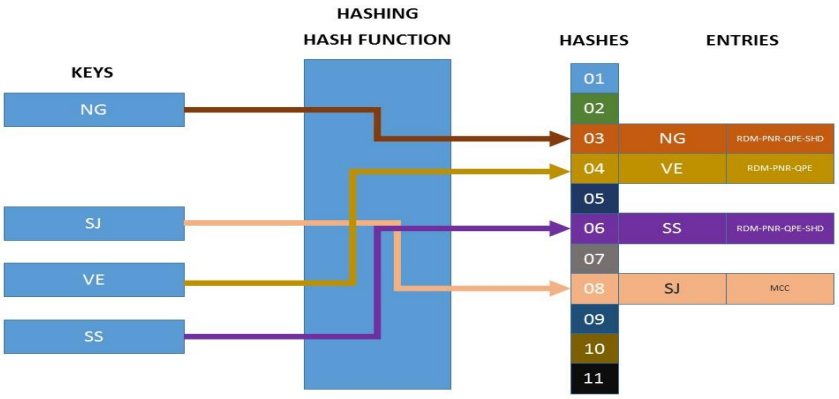


Fig. 1

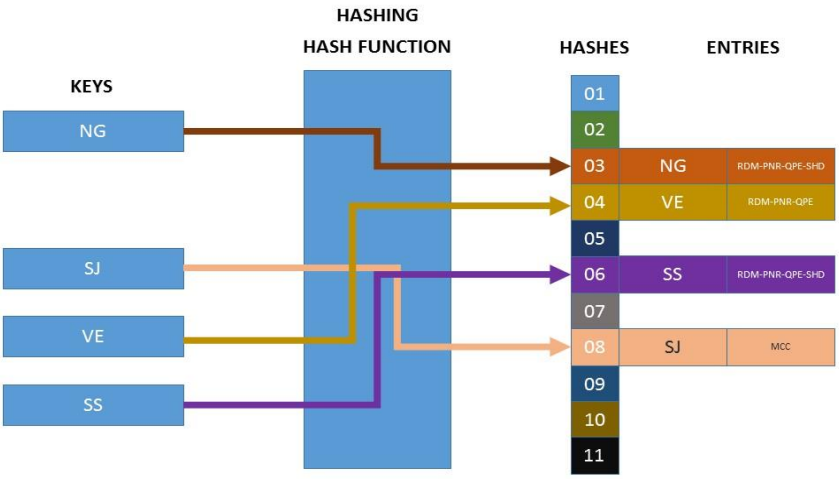


Fig. 2 – Cuckoo Hashing – Initial placement of keys

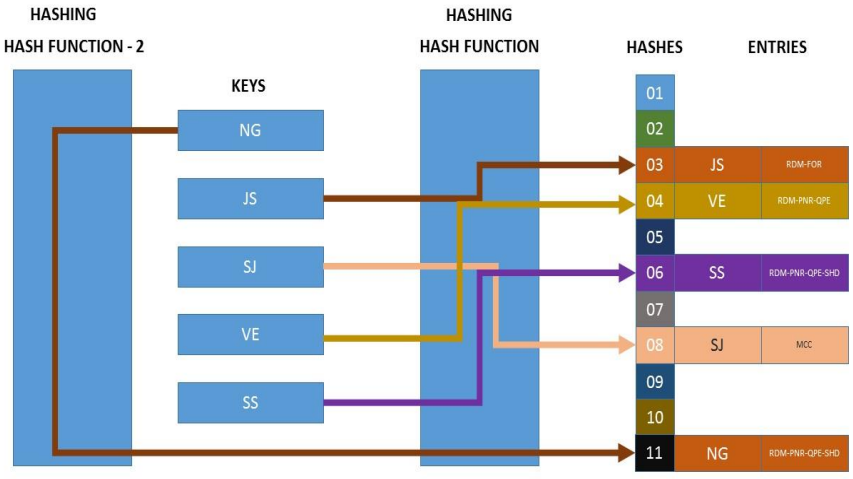


Fig. 3 – Cuckoo Hashing – Rehashing NG with hash fn 2 when JS get conflicted with NG

II. SELF-EVOLUTION

Cuckoo Hashing use two hash functions which guarantee that a key must be present in one of the two memory locations derived by the two functions. Since we know that this algorithm has the problem of the infinite loop because of which it cannot be used in any of the stable systems. Let's discuss the algorithm that can be used to identify such loop.

Identification of Loop

To evolve the algorithm, we first need to specify when a loop can occur in the algorithm. This can be identified easily by tracking what all elements are being replaced at the time of any insertion. If the number of replacement of any element is more than the number of functions present at that particular time, which means it went into an infinite loop.

Now let's see understand this algorithm with an example. Let's say first all the keys are placed in empty locations.

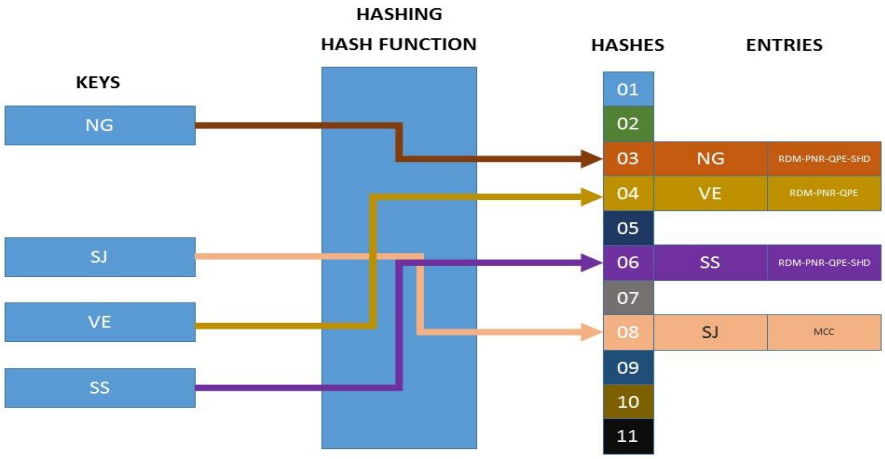


Fig. 1 – Initial placement

Table I

REPLACEMENT TABLE 1

Replacement Key								
# of Replacement								

Now JS came for the entry. We hash JS with function 1 and get location 03. Since 03 is already filled by NG, NG is moved out and JS is placed.

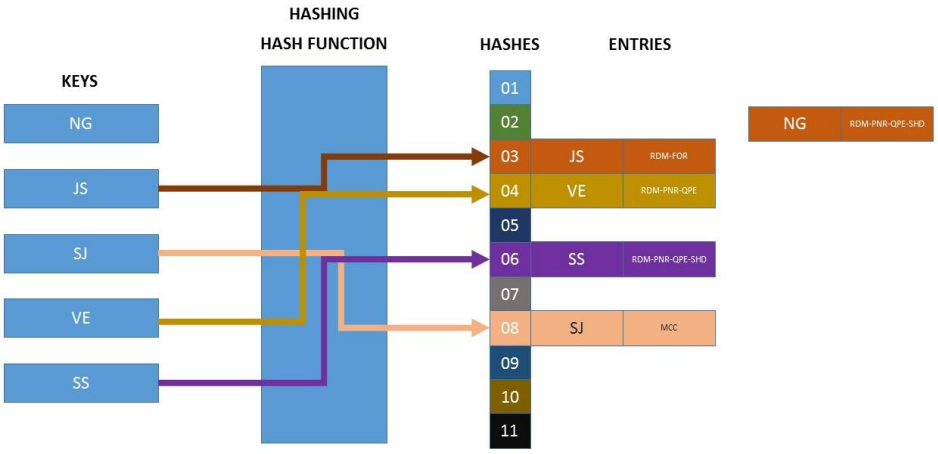


Fig. 2 – JS kicked out NG

Table II

REPLACEMENT TABLE 2

Replacement Key	NG							
# of Replacement	1							

Now NG is hashed with function 2 and a new location 04 is derived. Since VE is already at location 04, we will remove VE and NG is placed at 04.

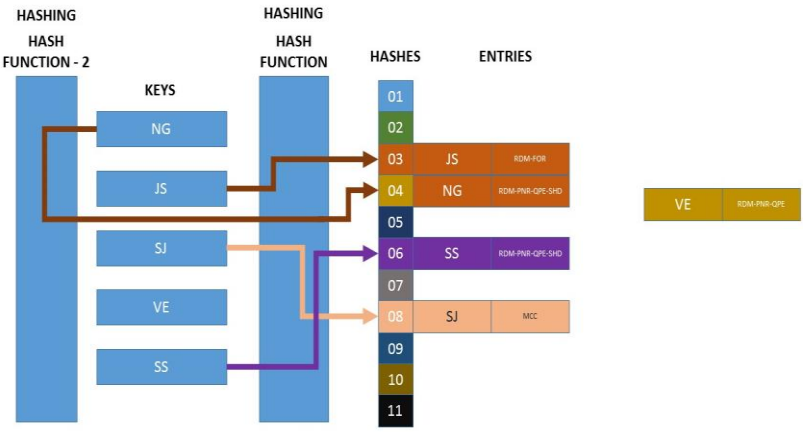


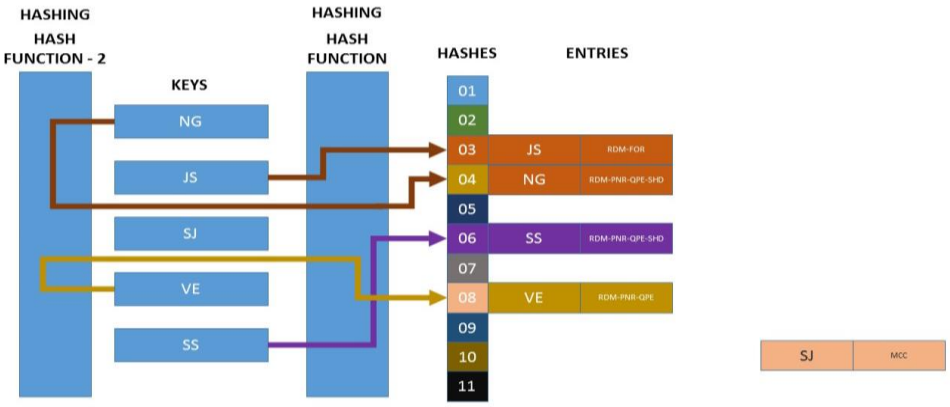
Fig. 3 – NG kicked out VE

Table III

REPLACEMENT TABLE 3

Replacement Key	NG	VE						
# of Replacement	1	1						

Now VE has hashed with function 2 and a new location is 08 derived. Since SJ is already at location 08, we will remove SJ and VE are placed at 08.



4 Fig. 4 – VE kicked out SJ

Table IV

REPLACEMENT TABLE

Replacement Key	NG	VE	SJ					
# of Replacement	1	1	1					

Now SJ is hashed with function 2 and a new location is 03 derived. Since JS is already at location 03, we will remove JS and SJ is placed at 03.

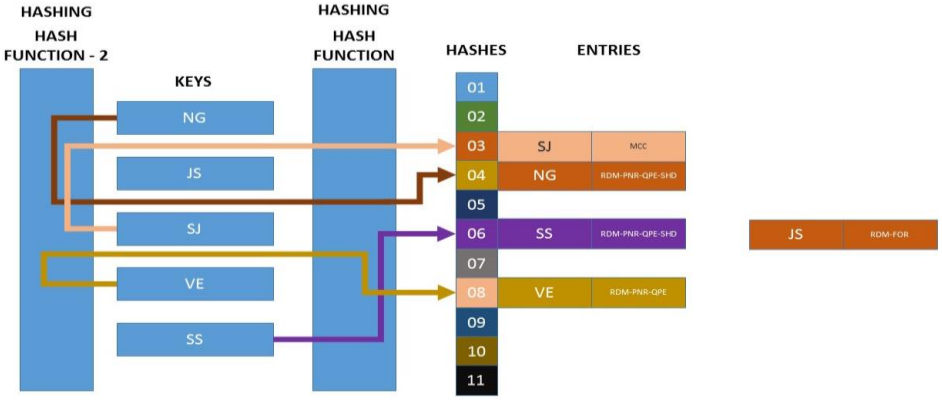


Fig. 5 – SJ kicked out JS

Table V

REPLACEMENT TABLE 5

Replacement Key	NG	VE	SJ	JS				
# of Replacement	1	1	1	1				

Now JS has hashed with function 2 and a new location is 08 derived. Since VE is already at location 08, we will remove VE and JS is placed at 08.

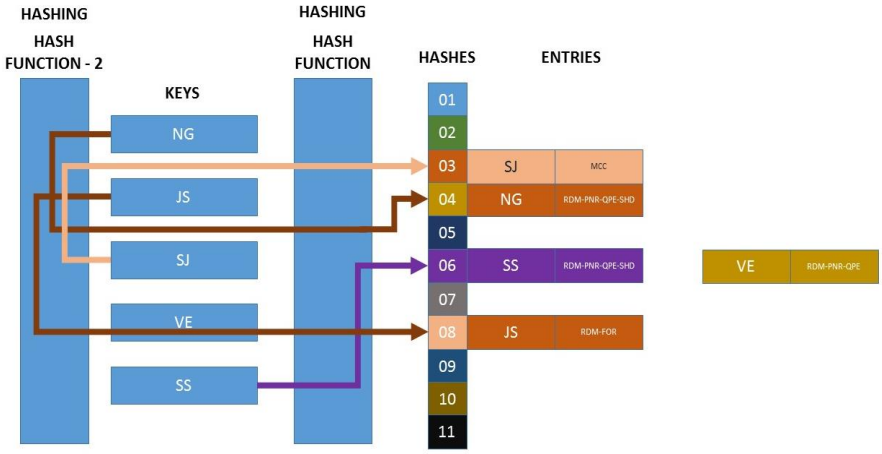


Fig. 6 – JS kicked out VE

Table VI

REPLACEMENT TABLE 6

Replacement Key	NG	VE	SJ	JS				
# of Replacement	1	2	1	1				

Now VE is hashed with function 1 and a new location is 04 derived. Since NG is already at location 04, we will remove NG and VE is placed at 04.

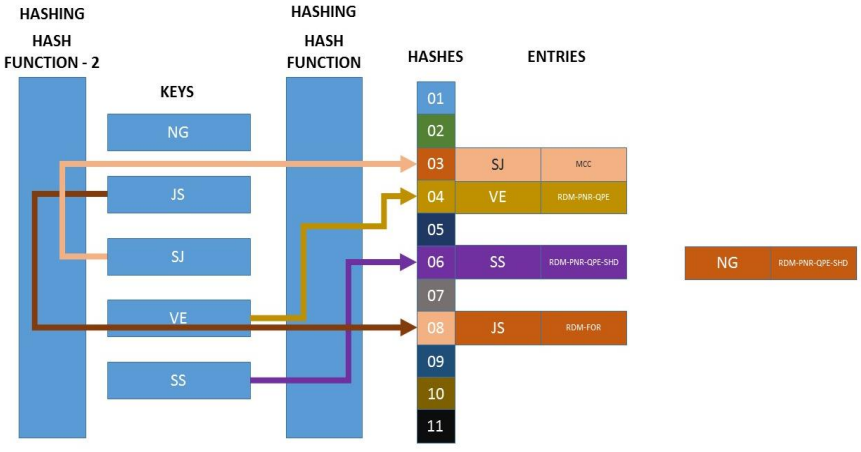


Fig. 7 – VE kicked out NG

Table VII

REPLACEMENT TABLE 7

Replacement Key	NG	VE	SJ	JS				
# of Replacement	2	2	1	1				

Now NG is hashed with function 1 and a new location is 03 derived. Since SJ is already at location 03, we will remove SJ and NG is placed at 04.

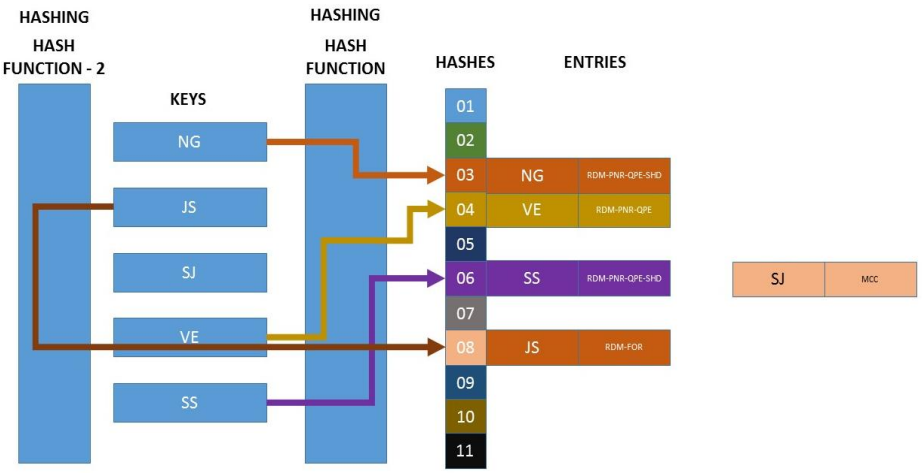


Fig. 20 – NG kicked out SJ

Table VIII

REPLACEMENT TABLE 8

Replacement Key	NG	VE	SJ	JS				
# of Replacement	2	2	2	1				

Now SJ has hashed with function 1 and a new location is 08 derived. Since JS is already at location 04, we will remove JS and SJ is placed at 04.

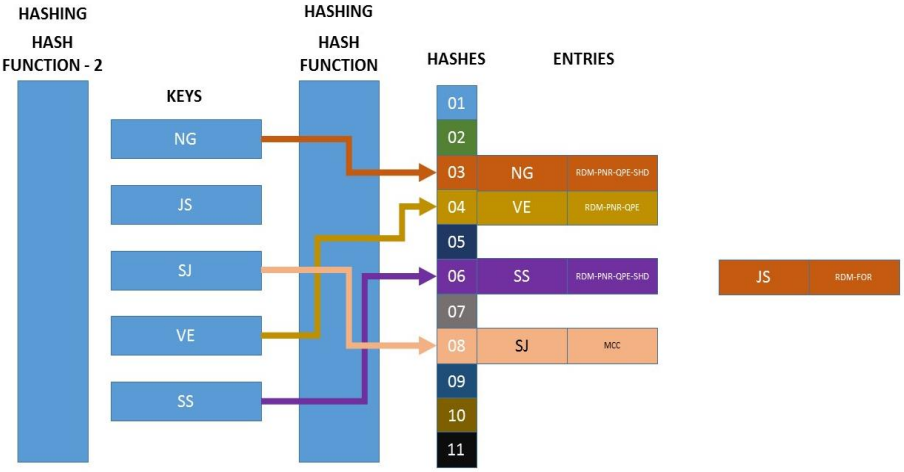


Fig. 8 – SJ kicked out JS

Table IX

REPLACEMENT TABLE 9

Replacement Key	NG	VE	SJ	JS				
# of Replacement	2	2	2	2				

Now JS is hashed with function 1 and a new location is 03 derived. Since NG is already at location 03, we will remove NG and JS is placed at 03.

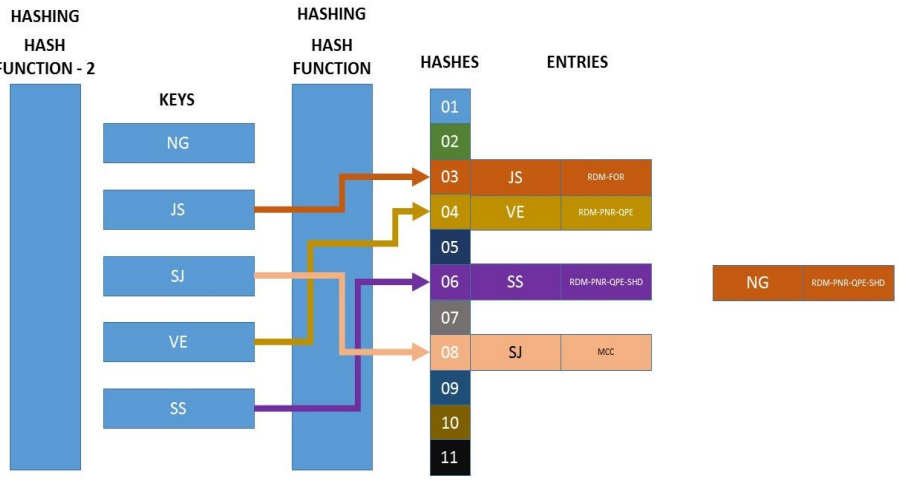


Fig. 9 – JS kicked out NG

Table IX

REPLACEMENT TABLE 9

Replacement Key	NG	VE	SJ	JS				
# of Replacement	3	2	2	2				

Now since NG is replaced 3 times, and functions for cuckoo hashing is 2, hence we can easily conclude that it's an infinite loop.

CONCLUSIONS

In this paper, we discussed the basic concept of Cuckoo Hashing. Then we discussed the derived algorithm that can be used to detect a loop while insertion. We can finally conclude that this algorithm can be the best possible way in which we can find out the loop and can terminate the infinite running of the algorithm.

ACKNOWLEDGMENT

Nitesh Gupta wishes to acknowledge Dr. Om Prakash and other contributors for continuous encouragement and helping me in writing this paper.

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.

