An Efficient Strategy for Collision Resolution in Hash Tables

Peter Nimbe
Dept. of Computer Science
K.N.U.S.T – Kumasi, Ghana
Dept. of Computer Science
C.U.C.G, Fiapre-Sunyani
Ghana

Samuel Ofori Frimpong
Faculty of I.C.S.T
Dept. of Computer Science
C.U.C.G, Fiapre-Sunyani
Ghana

Michael Opoku
Dept. of Computer Science
K.N.U.S.T – Kumasi, Ghana
Dept. of Computer Science
C.U.C.G, Fiapre-Sunyani
Ghana

ABSTRACT

This paper presents NFO, a new and innovative technique for collision resolution based on single dimensional arrays. Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys and should be seen as an event that can disrupt the normal operations or flow of hash functions computing an index into an array of buckets or slots. Hash tables provide efficient table implementations but then its performance is greatly affected if there are high loads of collisions. This new approach intends to manage these collisions effectively and properly although there are some algorithms for handling collisions currently. NFO incorporates certain features to resolve some problems of existing techniques. The performance of our approach is quantified via analytical modeling and software simulations. Efficient implementations that are easily realizable and productive in modern technologies are discussed. The performance benefits are significant and require machines with moderate memory and speed specifications. Depending on observations of the results of implementation of the proposed approach or technique on a set of real data of several types, all results are registered and analyzed.

General Terms

Algorithm, Collision, Performance, Implementation

Keywords

Hash Function, Open Addressing, Separate Chaining, Linear Probing, Quadratic Probing, Double Hashing

1. INTRODUCTION

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the O(log *n*) average cost required to do a binary search on a sorted array of *n* records, or the O(log *n*) average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system [1].

A hash table is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a hash function h that maps every possible item *x* to a small integer h(x). Then we store *x* in slot h(x) in an array. The array is the hash table. Most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket will occur and must be accommodated in some way [2].

Alternatively, a hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array [3].

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is a 95% chance of at least two of the keys being hashed to the same slot [4].

Therefore, most hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values [4].

Some of these strategies include:

- Open Addressing (Linear Probing, Quadratic Probing, Rehashing/Double Hashing)
- Separate Chaining with linked lists
- Separate Chaining with other structures
- Separate Chaining with list heads [4]

2. RELATED WORK

There are many collision resolution strategies. Open addressing and separate chaining are considered in this paper. Focus is placed on these two broad strategies even though there are other strategies for resolving collisions in hash tables. They are the 2 broad ways of collision resolution and play a vital role in the analysis and comparisons [5]. Cache-Conscious collision resolution strategy used in string hash tables is also reviewed in this paper.

2.1 Open Addressing

This strategy uses array implementation where all items are stored in the hash table itself. Each of the cells in the hash table or array has three states namely: OCCUPIED, EMPTY, DELETED. Alternative cells which are empty are found by the hash function when collision occurs [6]. This hash table has a probe sequence which is usually in the form:

h_i(key) = [h(key) + c(i) % n, for i=0,1,..,n-1 where h is the hash function and n is the size of the hash table. The function c(i) is required to have the following two properties:

Property 1: c(0) = 0

Property 2: The set of values {c(0) % n, c(1)%n, c(2)%n, …, c(n-1)%n} must be a permutation of {0,1,2,…,n-1}, that is, it must contain every integer between 0 and n-1 inclusive [6].

The function c(i) is used to resolve collisions.

To insert item r, we examine array location h0(r) = h(r). If there is a collision, array locations $h_1(r)$, $h_2(r)$,..., $h_{n-1}(r)$ are examined until an empty slot is found [6].

Similarly, to find item r, we examine the same sequence of locations in the same order.

For a given hash function h(key), the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function c(i). Common definitions of c(i) are:

Table 1. Definitions of collision resolution techniques

Collision Resolution Strategy	**c(i)**
Linear Probing	i
Quadratic Probing	$+=i^2$
Double Hashing	$i*h_p(key)$

where $h_p(key)$ is another hash function

Open Addressing come with some merits like:

- No need of a new data structure
- Efficient storage-wise

And its demerits include:

- Requires the use of 3 state flag in each cell
- The keys of objects to be hashed must be distinct
- Proper table size must be chosen

Usually in order to determine an appropriate table size, the following formula is of essence:

$$\text{Table size} = \text{Smallest prime} \geq \frac{\text{Number of items in table}}{\text{Desired load factor}}$$

In linear probing, when collision occurs, the table is searched sequentially for an empty slot. This is accomplished using two values - one as an initial value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the *increment*, is repeatedly added to the initial value until a free space is found, or the entire table is traversed [7]. The algorithm for this technique is

nextLocation = (initialValue + increment) % tableSize

The *increment* takes the following value: 1, 2, 3, 4, and so on. Given an ordinary hash function h(x), a linear probing function would be:

H(x, i) = (h(x) + i) (mod n)   for i = 0, 1, 2,. . . n-1   [7]

Linear probing has some disadvantages which include:

- Primary clustering
- Large clusters lead to long probe sequence
- Large clusters lead to deterioration in hash table efficiency [6]

In quadratic probing, original hash value is taken and successive values of an arbitrary quadratic polynomial are added to the starting value. The idea here is to skip regions in the table with possible clusters [7]. It uses the hash function of the form:

$H(k, i) = (h(k) + i^2) \bmod n$ for i = 0, 1, 2, . . . , n-1   [7]

In double hashing, a second hash function h2(key) is applied to the key when collision takes place. The result of the second hash function will be the number of positions from the point of collision to insert. There are some requirements for the second function. These requirements include:

- It must never evaluate to zero
- Must make sure that all cells can be probed [7].

The probing sequence is then computed as follows:

Hi(x) = (h(x) + ih2(x)) mod n [7]

Where h(x) is the original function, h2(x) the second function, i the number of collisions and n the table size. So the table is searched as follows:

H0 = (h(x) + 0*h2(x)) mod n

H1 = (h(x) + 1*h2(x)) mod n

H2 = (h(x) + 2*h2(x)) mod n

And so on [7].

2.2 Separate Chaining

This strategy uses an array of linked list implementation. It could also use other data structures other than linked lists. Here, many linked lists are connected or chained to various cells of the hash table. This tends to cause some problems especially tracking these linked lists. For example, if a hash table has 2500 cells, it implies there will also be a maximum of 2500 linked lists (if collisions occur in all the cells) which are obviously very difficult to track [6]. The separate chaining comes with some advantages like:

- Efficient collision resolution
- Deletion is easy
- Table size need not be a prime number [6].

It also has some disadvantages like:

- Separate data structure for chains required and code to manage it.
- Extra space required for linked lists.
- Creating new nodes is expensive and slows down the machine for some languages [6].

The separate chaining is depicted in Figure 1.

Loading the keys 23, 13, 21, 14, 7, 8, and 15, in a hash table of size 7 using the separate chaining using linked lists, the following hash values emerge.

h(23) = 23%7 = 2

h(13) = 13%7 = 6

h(21) = 21%7 = 0

h(14) = 14%7 = 0   collision

h(7) = 7%7 = 0   collision

h(8) = 8%7 = 1

h(15) = 15%7 = 1

The hash table representation is depicted in Figure. 2.

2.3 Cache-Conscious Collision Resolution

In this strategy, two alternatives to the standard representation were explored. They included:

- The simple expedient of including the string in its node
- And the more drastic step of replacing each list of nodes by a contiguous array of characters

The Cache-Conscious Collision Resolution Strategy is significant for large set of strings and the new structure gives substantial savings in space at no cost in time. In the best case, the overhead space required for pointers is reduced by a factor of around 50 to less than two bits per string (with total space required, including 5.68 megabytes of strings, falling from 20.42 megabytes to 5.81 megabytes), while access times are also reduced [8].

Askitis et al suggested cache-conscious strategy as oppose to a standard-chain hash table which uses two pointer traversals, one to reach the node and one to reach the string. In the cache-conscious strategy, strings are assumed to have sequences of 8-bit bytes, and a character such as null is available as a terminator. This strategy eliminates the chain altogether, and store the strings in a contiguous array [8].

The cache-conscious strategy is highly effective with array-based structures use for storing strings. Each array element is represented as a contiguous list of items which in effect depict the array as a resizable bucket. The cost of access is a single pointer traversal, to fetch a bucket, which is then processed linearly [8]. Although there seem to be an improvement in this strategy, it is best suited for string values. Askitis et al experiment did not reveal how numeric values would perform with their proposed scheme. Their prime focus was on cache (space) utilization, and memory management [8]. The cache-conscious collision resolution strategy is depicted in Figure 3.

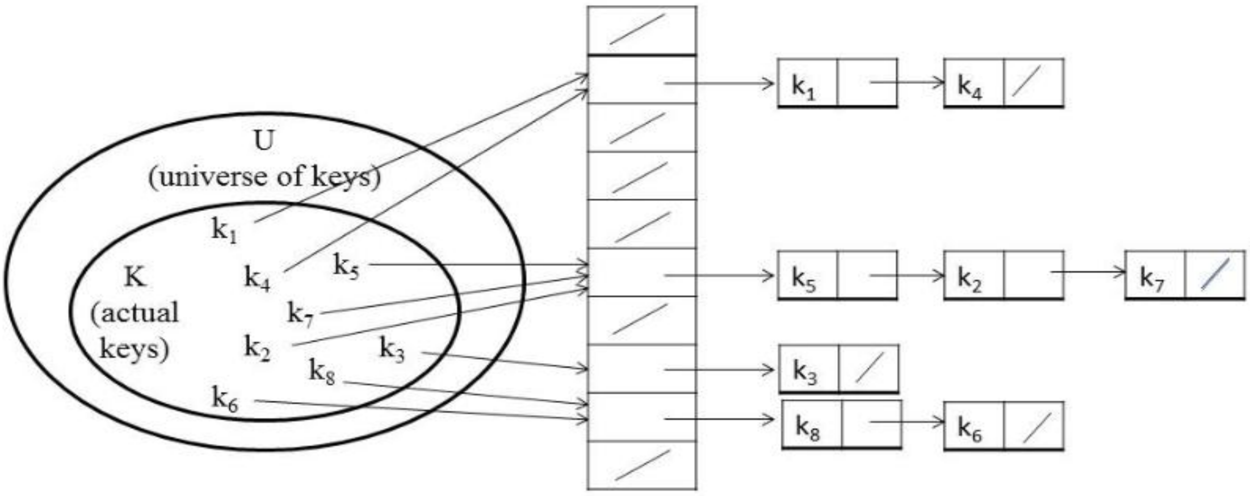


Fig.1: Separate chaining using linked lists

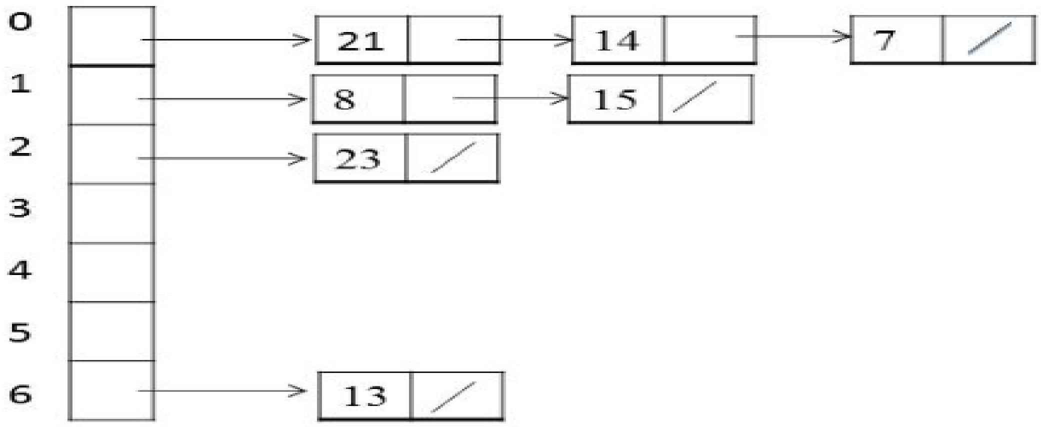


Fig.2: Hash table representation for separate chaining

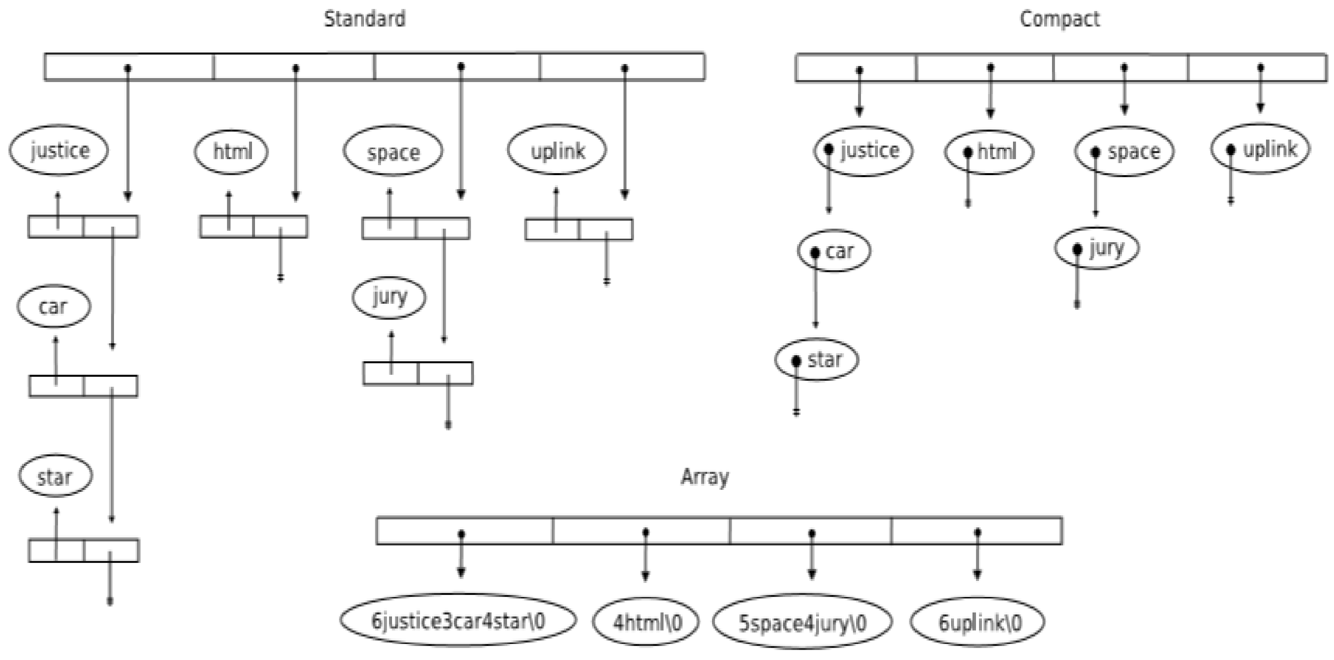


Fig.3: The standard-chain (left), compact-chain (right) and array (below) hash tables

3. NFO STRATEGY

The algorithm for the NFO strategy is expressed as a pseudocode. It is then implemented using C++ programming language. Further illustration is given with respect to how the algorithm functions.

3.1 Pseudocode

```
Declare variables: key, size, chk, val
Prompt user to enter size
Get size
FOR i=0 to size
    Prompt user to enter keys
    Get key
    Assign key to val
    Assign key to chk
    WHILE chk is greater than zero
        Calculate chk as chk/10
        Calculate val as val*0.1
    END WHILE
    Calculate h_i as (key%size)*1.0+val
END FOR
FOR i=0 to size
    Display i, h_i
END FOR
.
```

3.2 C++ Implementation

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
        int key, size, chk;

        cout<<"Please specify the size of hash table:";
        cin>>size;
        double h[size],  val=0.0;

        for(int i=0;i<size;i++)
        {
            cout<<"Please enter the keys to be   hashed:";
            cin>>key;
            chk = val = key;
            while(chk>0)
            {
                chk/=10;
                val*=0.1;
             }
                h[i]=(key%size)*1.0+val;
        }

        for (int i=0;i<size;i++)
                cout<<i<<setw(7)<<h[i]<<"\n";
        return 0;
}
```

3.3 Illustration

Step by step operations are outlined using linear probe, separate chaining and NFO (algorithm being proposed). The elements to be hashed are 23, 13, 8, 15, 45, 88, and 67.

3.3.1 Linear Probing

The hash function is given by h(x) = x%7

For the 1st Element (x=23)

h(23) = 23%7 = 2

This implies the 1st element will be stored in slot 2 of the bucket array.

For the 2nd Element (x=13)

h(13) = 13%7 = 6

This implies the 2nd element will be stored in slot 6 of the bucket array.

For the 3rd Element (x=8)

h(8) = 8%7 = 1

This implies the 3rd element will be stored in slot 1 of the bucket array.

For the 4th Element (x=15)

h(15) = 15%7 = 1

This implies the 4th element will be stored in slot 1 of the bucket array

For the 5th Element (x=45)

h(45) = 45%7 = 3

This implies the 5th element will be stored in slot 3 of the bucket array

For the 6th Element (x=88)

h(88) = 88%7 = 4

This implies the 6th element will be stored in slot 4 of the bucket array

For the 7th Element (x=67)

h(67) = 67%7 = 4

This implies the 7th element will be stored in slot 4 of the bucket array. The hash table representation is shown in Table 2 below.

Table 2: Hash table representation for linear probing

0			
1	8	-	3rd Element
2	23	-	1st Element
3	15	-	4th Element here due to collision at slot 1
4	45	-	5th Element here due to collision at slot 3
5	88	-	6th Element here due to collision at slot 4
6	13	-	2nd Element

Analysis

- The 4th Element (15) should have gone to slot 1 but there is a collision since the element 8 is already occupying that slot. The next available slot which is slot 2 is also occupied by the element 23. The 3rd available slot was empty hence element 15 was placed there.

- Again the 5th Element (45) which should have been stored in slot 3 is moved to slot 4 which is the next available slot which is not occupied.

- The 6th Element (88) should have gone to slot 4 but then it is occupied by element 45. The next available slot is slot 5 which is empty. Hence it's placed there.

Limitations

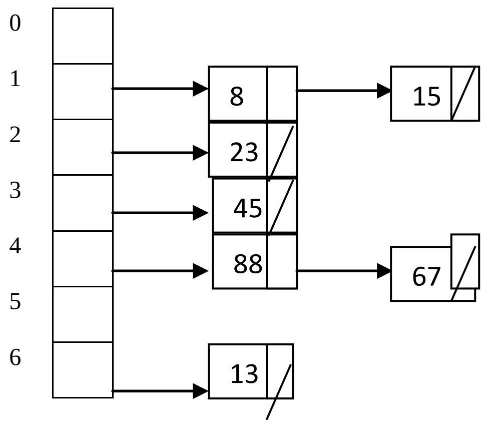
- The hash table is not full to capacity yet not all the elements can be stored in the hash table due to the incremental approach adopted by the linear probe. For instance element 67 had to be stored in slot 4 but then that slot is occupied. So the next available slots which could have stored the element were not available. Hence there is a dead end.

- The hash function should have placed elements in specific slots but because of collision they have been displaced or moved elsewhere. Hence relating some of the elements to their corresponding slots is complex.

3.3.2 Separate Chaining

Using the same function h(x) = x%7 and the elements (23, 13, 8, 15, 45, 88, and 67), separate chaining using linked list is represented in Table 3 below.

Table 3: Hash representation for separate chaining



Limitations

- The hash table is not optimally used because there are empty slots at 0 and 5.
- Keeping track of the multiple linked lists becomes extremely difficult.

3.3.3 NFO

Using the same function h(x) = x%7 and the elements 23, 13, 8, 15, 45, 88, and 67, the values returned are 2, 6, 1, 1, 3, 4, and 4. The values returned are then joined or concatenated to the elements hashed by using a dot (.) and then placed in the next available cell or slot starting from the first slot (slot 0). The hash table is represented below in Table 4.

Table 4: Hash representation for NFO

0	2.23
1	6.13
2	1.8
3	1.15
4	3.45

5	4.88
6	4.67

From the hash table above, the elements 8 and 15 are found in slots 2 and 3 whereas the elements 88 and 67 are found in slots 5 and 6. Elements 23, 13 and 45 are found in slots 0, 1 and 4 respectively. The preceding numbers before the dot (.) in each slot indicates the value returned by the hash function. There is a value of 1 preceding the elements in slots 2 and 3 likewise there is a value of 4 preceding the elements in slots 5 and 6. This indicates that had it not being the adoption of NFO, the elements 8 and 15 would have collided in the same bucket slot (slot 1). The same applies to elements 88 and 67 which would have also collided in the same bucket (slot 4).

4. RESULT ANALYSIS

The Big-O-Notation and the Number of Primitive Operations are used to characterize the running times of the algorithms. The characterization is done in terms of n, of the running time of the algorithm.

4.1 Big O Notation

The outer loop iterates 'n' times. The inner loop also iterates 'n' times. In computing the big O, the number of iterations of each of the loop must be multiplied together. The running time of the algorithm (T(n)) is computed as follows:

$T(n) = O(n)[O(1) + O(n)[O(1) + O(1)] + O(1)] + O(n)$
$T(n) = O(n) [O(1) + O(n)+ O(n)] + O(1)] + O(n)$
$T(n) = O(n) + O(n^2) + O(n^2) + O(n) + O(n)$
$T(n) = O(2n^2) + O(3n)$.

In this case as N becomes very large, $O(n^2)$ is considered the most significant factor of the Big O-Notation obtained from the above T(N) deductions. Hence in the worst case scenario NFO algorithm's time efficiency complexity can be measured by $T(n) = O(n^2)$

4.2 Number of Primitive Operations

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
        int key, size, chk;

        cout<<"Please specify the size of hash table:";
        cin>>size; //1 primitive operation
        double h[size],  val=0.0; //2 primitive operations

        for(int i=0;i<size;i++) //1+ n*1 primitive operations
        {
             cout<<"Please enter the keys to be   hashed:";
             cin>>key;       //n*1 primitive operation
             chk = val = key;     //n*2 primitive operation
             while(chk>0)       //n*n primitive operations
             {
                chk/=10;    //(n*n*2) primitive operations
                val*=0.1;    //(n*n*2) primitive operations
              }
                //n*5 primitive operations
                h[i]=(key%size)*1.0+val;
        }

        for (int i=0;i<size;i++) //1+ n*1 primitive operations
                //2 primitive operation
                cout<<i<<setw(7)<<h[i]<<"\n";
        return 0;
        }
```

At least: $t(n) = 3$

As $n \rightarrow 3, 3 \rightarrow \infty$ => $O(n^2)$
At most: $t(n) = 1 + 2 + 1+ n*1 + n*1 +n*2 + n*n + n*n*2 + n*n*2 +n* 5 + 1+n*1+2 = 7+10n+4n^2$ As n , $(4n^2 + 10n + 7)$ => $O(n^2)$.

4.3 Merits and Demerits of NFO

The merits of NFO strategy include:

- Efficient storage-wise
- Does not require use of 3 state flag in cells
- Table size is known (size of table = number of values to be hashed)
- No primary clustering
- No long probe sequence
- No deterioration in hash table efficiency
- Collision resolution is efficient
- Easy to search and track an element and its slot or bucket number.
- Table size need not be a prime number
- The keys of the objects to be hashed need not be unique.
- No need to use another data structure

The demerits of NFO strategy include:

- Extra computation is required
- Extra feature is required i.e. dot (.)

5. CONCLUSION

NFO (an abbreviation for Nimbe-Frimpong-Opoku) is an efficient collision resolution strategy experimented on numeric values. NFO is being proposed and is earnestly hoped it will go a long way to add to the body of knowledge due to the numerous merits it has, including but not limited to efficient storage-wise, no primary clustering and no deterioration in hash table efficiency. It is also known as the dot (.) strategy. Future works to resolve collisions in hash tables will be conducted with a multidimensional array and other data structures. A variant of NFO will be designed to resolve collision in string hash tables. This will require some modifications to the original NFO, thus making it more efficient, reliable and adoptable to values of different data types.

6. ACKNOWLEDGMENTS

It has been God‘s grace and revelation that this paper has been completed. All praises and gratitude is given unto God. Amen! Families and Friends are acknowledged for their encouragement and unyielding support. Reviewers and experts are also shown gratitude for their assessments, contributions and comments towards this paper.

7. REFERENCES

[1] Clifford, A. Shaffer., 2007. Hashing Tutorial.

[2] Erickson, J., 2009. Hash Tables

[3] Bruno, D.G., 1999. Data structures and algorithm with object oriented design in C++ (1* Ed). Addison Wesley Publishing Company-America. PP. 225-248

[4] Archaya, A., 2012. Input Segmented Universal Hashing

[5] D.E. Knuth., 1998. The Art of Computer Programming: Sorting and Searching, volume3. Addison- Wesley Longman, second edition.

[6] Jauhar, A., 2008. Hashing: Collision Resolution Schemes

[7] Bello, S.A., Liman, A.M., Gezawa, A.S., Garba, A., Ado, A., “Comparative Analysis of Linear Probing, Quadratic Probing and Double Hashing Techniques for Resolving Collusion in a Hash Table”, International Journal of Scientific & Engineering Research, 2014

[8] Askitis, N, Zobel, J., 2005. Cache-Conscious Collision Resolution in String Hash Tables