International Journal of Innovation and Scientific Research
ISSN 2351-8014 Vol. 13 No. 1 Jan. 2015, pp. 89-97

http://www.ijisr.issr-journals.org/

Hashing String Keys Using NFO and NOF Collision Resolution Strategies

Peter Nimbe[1], Michael Opoku[1], and Samuel Ofori Frimpong[2]

[1]Department of Computer Science,
K.N.U.S.T,
Kumasi, Ghana

[2]Department of Computer Science,
C.U.C.G,
Sunyani-Fiapre, Ghana

ABSTRACT: This paper presents the systematic way of hashing string values using NFO and NOF collision resolution strategies. NFO and NOF are techniques used for hashing numeric keys. The same principles and techniques for hashing numeric keys are deployed in the hashing of string values but with slight modifications in the hashing process and implementations. These variants followed the standard ways of evaluating and implementing algorithms to resolve collisions in hash tables. They are very effective in resolving the problem of collisions of string keys or values in the same slot of a hash table.

KEYWORDS: NFO Strategy, NOF Strategy, Hash Function, Hash Tables, Collision Resolution, ASCII Value, Algorithm.

1 INTRODUCTION

NFO and NOF are collision resolution strategies to be reviewed in this paper. They play a vital role in the results, analysis and discussions. Over the years, various collision resolution techniques have been developed. The goal of collision resolution is to find a free slot in the hash table when the home position for the key is already occupied. The various collision resolution strategies generate a sequence of hash table slots that can hold a value or key. A probe function is used by any collision strategy or policy to generate probe sequences (sequence of slots) for keys to occupy. The collision resolution policy generates a next slot if the home position is occupied. If this is occupied as well, then another slot must be found, and so on. The ideal behavior for a collision resolution mechanism is that each empty slot in the table will have equal probability of receiving the next record inserted (assuming that every slot in the table has equal probability of being hashed to initially). Its probe sequence should be able to cycle through slots in the hash table before returning to the home position [1].

1.1 NFO STRATEGY

NFO is a dot strategy technique for collision resolution based on single dimensional arrays. The dot feature which is incorporated as part of the implementation of this technique helps in the placement of numeric key values in appropriate slots of the bucket array or hash table after they have been hashed. It is very efficient storage-wise, and does not require use of 3 state flag in cells. This technique has no primary clustering, no long probe sequences and no deterioration in hash table efficiency. Collision resolution is very efficient. Tracking an element and its slot number is made very easy by means of the dot feature. It has a running time of $O(n^2)$ [2].

Corresponding Author: Peter Nimbe

1.2 NOF STRATEGY

NOF is a technique for collision resolution based on multidimensional arrays (n x n array). It was discussed that an n x m array is used for better and more efficient implementations. In this case, n is equal to the number of rows and m is equal to the number of columns. The use of the multidimensional array ensures that elements are placed in appropriate slots by matching each of the slot numbers with a specific row of the multidimensional array. The primary issue of concern of this technique is that there are unutilized empty spaces or slots which makes the scheme not efficient storage-wise. Despite this concern, it is proven to be very effective in collision resolution. In the implementation of NOF, a further and better representation of the array size which was trimmed down or resized enhanced the space utilization though it could not eliminate it completely. An optimal implementation of the above technique is to remove or deallocate the empty spaces. Tracking an element and its slot number is made very easy by means of the multidimensional array. It has a running time of $O(n^2)$ [3].

2 MATERIALS AND METHODS

Secondary data was used for the analysis basically from literature, journals, websites, and lecture notes. The Variants of NFO and NOF algorithms for string hashing were implemented in C++ programming language using Dev-C++ IDE. The concepts of single and multi-dimensional arrays were adopted in this paper just as in the ones for numeric keys.

3 RESULTS

The variants are implemented using C++ programming language. Further illustration is given with respect to how the algorithms function.

3.1 C++ IMPLEMENTATION USING NFO

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

      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;

        sum=0;
            for(int d=0;d<key.length();d++)
                  sum=sum+int(key[d]);
```

```
                chk= val = sum;

                while(chk>0)
                {
                        chk/=10;
                        val*=0.1;
                }
                h[i]=(sum%size)*1.0+val;
        }

        cout<<endl;
        cout << std::fixed;
        for (int i=0;i<size;i++)
                cout<<i<<setw(7)<<setprecision(3)<<h[i]<<"\n";

        cout<<endl;
    system("pause");
        return 0;
}
```

3.2 C++ IMPLEMENTATION USING NOF

```
#include<iostream>
#include<iomanip>
#include<string>
using namespace std;
int main()
{
        int  size, rem=0, j=0, val=0;
        bool assign;
    string key="";

        //Accepting the size of the hash table
        cout<<"Please specify the size of hash table:";
        cin>>size;
        int h[size];
        string arr[size][size];
```

```
//Initializing the array
for(int a=0;a<size;a++)
        for(int b=0;b<size;b++)
                arr[a][b]="0";

//Accepting the keys, hashing and placing them in appropriate location in array
for(int i=0;i<size;i++)
{
        cout<<"Please enter the keys to be hashed:";
        cin>>key;

        val=0;
        for(int d=0;d<key.length();d++)
             val=val+int(key[d]);

        rem=val%size;
        j=0;
        assign=false;

                        do
                        {
                                if(arr[rem][j]=="0")
                                {
                                        arr[rem][j]=key;
                                        assign=true;
                                }
                                else
                                        j++;

                        }while(assign==false);
}

cout<<endl;

//Displaying the contents of the array
for(int a=0;a<size;a++)
```

```
        {
                cout<<a;
                for(int b=0;b<size;b++)
                        if(arr[a][b]!="0")
                                cout<<"  "<<arr[a][b];
                cout<<endl;
        }

        cout<<endl;
        system("pause");
        return 0;
}
```

3.3 BIG O NOTATION

Big O-Notation Time Efficiency Analyses for Variant of NFO

$T(N) = O(N)\ [O(1) + O(N)(O(1))\ + O(1)\ + O(N)[O(1) + O(1)] + O(1)] + O(N)$

$T(N) = O(N)\ [O(1) + O(N) + O(1) + O(N) + O(N) + O(1)] + O(N)$

$T(N) = [O(N) + O(N^2) + O(N) + O(N^2) + O(N^2) + O(N)] + O(N)$

$T(N) = [O(N^2) + O(N^2) + O(N^2) + O(N) + O(N) + O(N) + O(N)]$

As $N \rightarrow \infty$ all constants can be ignored

$T(N) \approx [O(3N^2) + O(4N)]$

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 the algorithm's time efficiency complexity can be measured by $T(N) = O(N^2)$

Big O-Notation Time Efficiency Analyses for Variant of NOF

$T(N) = [O(N)(O(N)+O(1))] + [O(N)(O(1) + O(N)(O(1)) +O(1) + O(1) + O(1) + O(N)(O(1)+O(1)+O(1))] + [O(N)(O(N))]$

$T(N) = [O(N^2) + O(N)] + O(N) + O(N) + O(N^2)(O(1))+O(N)+O(N)+ O(N)) + O(N^2)+O(N^2)+O(N^2)) + O(N^2)]$

$T(N) = [O(N^2) + O(N) + O(N) + O(N) + O(N^2) + O(N) + O(N) + O(N) + O(N^2) + O(N^2) + O(N^2) + O(N^2)]$

$T(N) = [O(N^2) + O(N^2) + O(N^2) + O(N^2)+ O(N^2) + O(N^2) + O(N) + O(N) + O(N) + O(N) + O(N)\ + O(N)]$

As $N \rightarrow \infty$ all constants can be ignored

$T(N) \approx [O(6N^2) + O(6N)]$

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 the algorithm's time efficiency complexity can be measured by $T(N) = O(N^2)$

3.4 ILLUSTRATION

Step by step operations are outlined using the variants of NFO and NOF. The string elements to be hashed are mellon, tomato, orange, potato, okra, carrot, banana, olive, salt, mushroom, onion, cabbage and cucumber. The ASCII values of the

characters in the string elements will be used in the computation process. The various characters and their ASCII code is given below [4].

Table 1. Characters and their ASCII Code

Character	a	b	c	d	e	f	g	h	i	j	k	l	m
ASCII Code	97	98	99	100	101	102	103	104	105	106	107	108	109

Character	n	o	p	q	r	s	t	u	v	w	x	y	Z
ASCII Code	110	111	112	113	114	115	116	117	118	119	120	121	122

3.4.1 NFO

The hash function is given by h(x) = x%13, where x is the sum of all the ASCII values for the characters in a particular string. The value 13 here represents the size or number of slots of the bucket array.

For the 1st Element

h(mellon) = (109 + 101 + 108 + 108 + 111 + 110) % 13 = 647 % 13 = 10

This implies the 1st element will be stored in slot 10 of bucket array.

For the 2nd Element

h(tomato) = (116 + 111 + 109 + 97 + 116 + 111) % 13 = 660 % 13 = 10

This implies the 2nd element will be stored in slot 10 of bucket array.

For the 3rd Element

h(orange) = (111 + 114 + 97 + 110 + 103 + 101) % 13 = 636 % 13 = 12

This implies the 3rd element will be stored in slot 12 of bucket array.

Applying the same process above the table below is generated:

Table 2. String elements, sum of ASCII values of the characters and slots

	Element	**X**	**h(x)**
1.	mellon	647	10
2.	tomato	660	10
3.	orange	636	12
4.	potato	663	0
5.	Okra	429	0
6.	carrot	651	1
7.	banana	609	11
8.	olive	543	10
9.	Salt	436	7
10.	mushroom	890	6
11.	Onion	547	1
12.	Cabbage	693	4
13.	Cucumber	854	9

Table 3. Hash Table Representation (NFO)

0	10.647	1st Element
1	10.660	2nd Element
2	12.636	3rd Element
3	0.663	4th Element
4	0.429	5th Element
5	1.651	6th Element
6	11.609	7th Element
7	10.543	8th Element
8	7.436	9th Element
9	6.890	10th Element
10	1.547	11th Element
11	4.693	12th Element
12	9.854	13th Element

The snapshot below is the results displayed after the hashing of the string elements using the NFO technique.

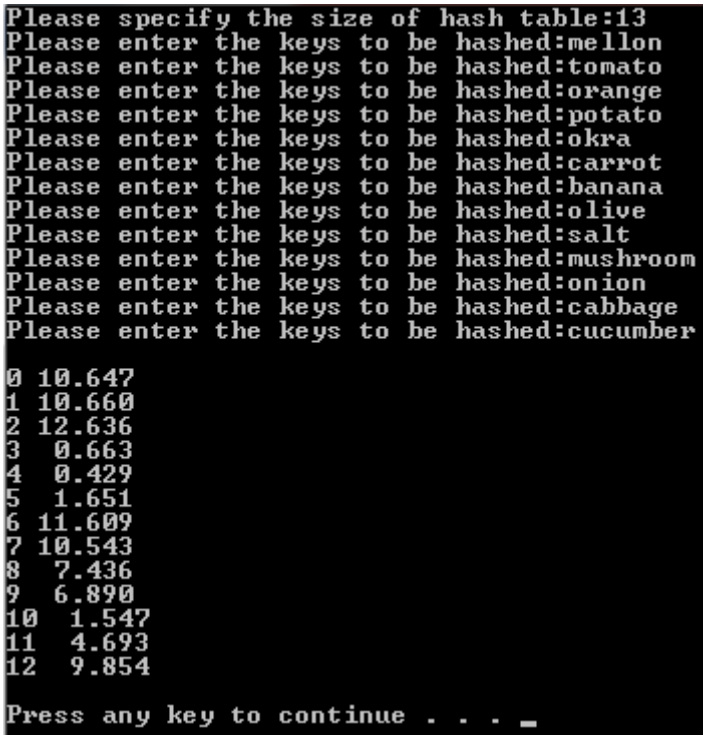

Fig. 1. Output from a Console (NFO)

ANALYSIS

Assuming we have two string values to be hashed i.e. “abc” and “bac”. This will return the same slot number after the hashing process is complete. The numeric representations of the string is then kept in the hash table. The question therefore is which of the values is “abc” or “bac”? The answer is any of them could be “abc” or “bac” and does not really make any difference if any of them is presented as “abc” or “bac”. All that needs to be done in searching for a string element is to compute the sum of the ASCII values of the characters in the string and searching through the hash table for that value.

Table 4. Hash Table Representation (NOF) – n x m array

0		potato	okra	
1		carrot	onion	
2				
3				
4		cabbage		
5				
6		mushroom		
7		salt		
8				
9		cucumber		
10		mellon	tomato	olive
11		banana		
12		orange		

The snapshot below is the results displayed after the hashing of the string elements using NOF technique.

```
Please specify the size of hash table:13
Please enter the keys to be hashed:mellon
Please enter the keys to be hashed:tomato
Please enter the keys to be hashed:orange
Please enter the keys to be hashed:potato
Please enter the keys to be hashed:okra
Please enter the keys to be hashed:carrot
Please enter the keys to be hashed:banana
Please enter the keys to be hashed:olive
Please enter the keys to be hashed:salt
Please enter the keys to be hashed:mushroom
Please enter the keys to be hashed:onion
Please enter the keys to be hashed:cabbage
Please enter the keys to be hashed:cucumber

0  potato  okra
1  carrot  onion
2
3
4  cabbage
5
6  mushroom
7  salt
8
9  cucumber
10  mellon  tomato  olive
11  banana
12  orange

Press any key to continue . . .
```

Fig. 2. Output from a Console (NOF)

3.5 COMPARATIVE ANALYSIS

		NFO	NOF
1.	**Array type**	Single dimensional	Multi-dimensional
2.	**Storage-wise**	Efficient	Not efficient
3.	**Keys type hashed**	Numeric and string	Numeric and string
4.	**Technique**	Dot	Chain
5.	**Running Time**	**$O(N^2)$**	**$O(N^2)$**

4 DISCUSSION

There are a number of collision resolution strategies in hash tables which have been used for numeric and string value hashing. This paper shows the step by step process involved in hashing string values or keys into appropriate slots of a bucket array. Even though there have been slight modification or variations to the codes or implementations, the technique and processes are identical. The major concepts used in NFO and NOF strategies for numeric key hashing is basically the same for string key hashing. It could be seen that the strategies can be used for hashing string values as well. It is strongly believed that the work of NFO and NOF will not have been complete if it was unable to hash string values. Whilst NFO and NOF has been proven to be very efficient there are some problems associated with them just like any other collision resolution strategy. This paper intended to demonstrate how these 2 collision resolution strategies could be used on string values. It has been successful in terms of hashing string keys and will makes an authentic contribution to the field of data structures and computational theory.

5 CONCLUSION

NFO and NOF collisions are also very efficient for string value hashing just like in the case of numeric key values. It is earnestly hoped this paper will add to the body of knowledge due the proven nature of the two strategies to handle collision problems of both numeric and string values. The adoption of these techniques for hashing either numeric and string values or keys will help minimize high overhead incurred as a result of collision. Future works will be to design collision resolution strategies using other data structures.

ACKNOWLEDGMENT

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.

REFERENCES

[1] Shaffer, C.A., "Hashing Tutorial - Collision Resolution", 2011

[2] Nimbe, P., Frimpong, S.O., Opoku, M., "An Efficient Strategy for Collision Resolution in Hash Table", International Journal of Computer Applications, Volume 99 – No. 10, August 2014, pp. 35-41.

[3] Nimbe, P., Opoku, M., Frimpong, S.O., Asante, A., Kornu, D., "Hash Table Collision Resolution Using a Multi-dimensional Array", International Journal of Innovation and Scientific Research, Vol. 9 No.2, Sep. 2014, pp. 258-267.

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