International Journal of Innovation and Scientific Research
ISSN 2351-8014 Vol. 9 No. 2 Sep. 2014, pp. 258-267
© 2014 Innovative Space of Scientific Research Journals
http://www.ijisr.issr-journals.org/

Hash Table Collision Resolution Using a Multi-dimensional Array

Peter Nimbe[1], Michael Opoku[2], Samuel Ofori Frimpong[3], Audrey Asante[4], and Dominic-Mary Kornu[5]

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

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

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

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

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

ABSTRACT: This paper presents a new and innovative technique for handling collisions in hash tables based on a multi-dimensional array. The proposed strategy followed the standard ways of evaluating and implementing algorithms to resolve collisions in hash tables. This technique is an effective way of handling the problem of collisions in hash table slots or cells but at a slight expense of space. It was discussed that an optimal representation of this scheme is to minimize or completely remove the empty spaces created within the array.

KEYWORDS: Open Addressing, Separate Chaining, Linear Probing, Quadratic Probing, Double hashing, Hash Function

1 INTRODUCTION

Open addressing and separate chaining are the 2 broad ways of collision resolution to be reviewed. They play a vital role in the analysis and discussions [1]. In open addressing, all elements are stored in the hash table itself. Table slots are systematically examined during search until the desired element is found. Elements are not stored outside the table. This is not the case for separate chaining where elements are stored outside of the hash table by means of linked lists. Three techniques are commonly used to compute probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that (h(k,0), h(k,1), …, k(k,m-1)) is a permutation of (0,1,…,m-1) for each key k. However none of these techniques fulfills the assumption of uniform hashing [2].

1.1 LINEAR PROBING

In linear probing, original hash value is taken and successive values of a linear nature are added to the starting value. Linear probing uses the hash function h(k,i) = (h`(k) + i) mod m for i=0, 1, …, m-1. It suffers from primary clustering even though it is easy to implement. Clusters arise since an empty slot preceded by i full slots gets filled next with probability (i+1)/m. The average search time for linear probe increases due to the building up of long runs of occupied slots [2].

1.2 QUADRATIC PROBING

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 [3]. It uses the hash function of the form:

$h(k, i) = (h(k) + i^2) \bmod n$ for i = 0, 1, 2. . . n-1 [3]

1.3 DOUBLE HASHING

Double hashing is another method of collision resolution, but unlike the linear collision resolution, double hashing uses a second hashing function which normally limits multiple collisions. The idea is that if two values hash to the same spot in the table, a constant can be calculated from the initial value using the second hashing function which can then be used to change the sequence of locations in the table, but still have access to the entire table [4].

1.4 SEPARATE CHAINING

This is a collision resolution technique where linked list of items are chained to the various addresses of the hash table depending on where the keys hash to. Multiple lists could be created or built and these lists have to be maintained. In separate chaining, items that collide are chained together in separate linked lists. In the open addressing schemes, keys are converted into table addresses. As with elementary sequential search, lists can be kept in a sorted or unordered order. For separate chaining, the time savings are less significant (because the lists are short) and the space usage is more significant (because there are many lists) [5].

2 MATERIALS AND METHODS

Secondary data was used for the analysis basically from literature, journals, websites, and lecture notes. The NOF algorithm was implemented in C++ programming language using Dev-C++ IDE. The concept of multi-dimensional arrays was adopted in the design of the NOF collision resolution strategy.

3 RESULTS

The NOF strategy or technique is implemented as an algorithm and 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, rem, j, assign
Declare arrays: h[size], arr[size][size]
Assign zero to rem
Assign zero to j
Assign zero to max
FOR a=0 to size
    FOR b=0 to size
        Assign zero to arr[a][b]
    END FOR
END FOR

FOR i=0 to size
    Prompt user to enter a key
    Get key

    Compute rem as key%size
    Assign zero to j
    Assign 'false' to assign

    DO
        IF arr[rem][j] is equal to zero
            Assign key to arr[rem][j]
            Assign 'true' to assign
        ELSE
            Increment j by 1
        END IF
    WHILE assign is equal to 'false'
```

```
END FOR

End Line
FOR a=0 to size
    Display a
    FOR b=0 to size
        IF arr[a][b] is greater than zero
            Display arr[a][b]
        END IF
    End Line
    END FOR
END FOR
```

3.2 C++ Implementation

```
#include<iostream>

#include<iomanip>

using namespace std;

int main()

{

    int key, size, rem=0, j=0;

    bool assign;


    //Accepting the size of the hash table

    cout<<"Please specify the size of hash table:";

    cin>>size;

    int h[size], 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;


        rem=key%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<<setw(5)<<arr[a][b];
                cout<<endl;
        }

        return 0;
}
```

3.3 BIG O NOTATION

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

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



        //Initializing the array
        for(int a=0;a<size;a++)                                  }
                  for(int b=0;b<size;b++)  ──────────►  O(N)     }  O(N)
                            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;

                  rem=key%size;        ──────────────►  O(1)
                  j=0;                 ──────────────►  O(1)
                  assign=false;                        O(1)                     O(N)

                                      do
                                      {
                                                if(arr[rem][j]==0)
                                                {
                                                          arr[rem][j]=key; O(1)  }  O(N)
                                                          assign=true;     O(1)
                                                }
                                                else
                                                          j++;  ─────►  O(1)

                                      }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++)  ──────────→   O(N)      ⎪
                        if(arr[a][b]>0)                          ⎪
                                cout<<setw(5)<<arr[a][b];        ⎬   O(N)
                cout<<endl;                                      ⎪
        }                                                        ⎪
                                                                 ⎪
        return 0;                                                ⎪
    }                                                            ⎭
```

Big O-Notation Time Efficiency Analyses

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

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

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

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

As $N \rightarrow \infty$ all constants can be ignored
$T(N) \approx [O(5N^2) + O(4N)]$

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 $T(N) = O(N^2)$

3.4 ILLUSTRATION

Step by step operations are outlined using linear probe, separate chaining and NOF (algorithm being proposed). The elements to be hashed are 45, 67, 25, 13, 23, 5 and 88.

3.4.1 LINEAR PROBING

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

For the 1st Element (x=45)

h(45) = 45%7 = 3

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

For the 2nd Element (x=67)

h(67) = 67%7 = 4

This implies the 2nd element will be stored in slot 4 of bucket array.

For the 3rd Element (x=25)

h(25) = 25%7 = 4

This implies the 3rd element will be stored in slot 4 of bucket array.

For the 4th Element (x=13)

h(13) = 13%7 = 6

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

For the 5th Element (x=23)

h(23) = 23%7 = 2

This implies the 5th element will be stored in slot 2 of bucket array.

For the 6th Element (x=5)

h(5) = 5%7 = 5

This implies the 6th element will be stored in slot 5 of bucket array.

For the 7th Element (x=88)

h(88) = 88%7 = 4

This implies the 7th element will be stored in slot 4 of bucket array.

Table 1: Hash Table Representation

0		
1		
2	23	5th Element
3	45	1st Element
4	67	2nd Element
5	25	3rd Element here due to collision at slot 4
6	13	4th Element

Analysis

- The 3th Element (25) should have gone to slot 4 but there is a collision since the element 67 is already occupying that slot. The next available slot (slot 5) was empty hence element 25 was placed there.
- The 6th Element (5) which should have been stored in slot 5 is occupied by element 25. The next slot which is slot 6 is occupied. Afterwards, there is no next available slot due to the incremental approach adopted by linear probing.
- Again the 7th Element (88) which should have been stored in slot 4 is occupied by element 67. The next slot which is slots 5 and 6 are occupied. Afterwards there is no next available slot due to the incremental approach adopted by linear probing even though slot 0 and 1 are empty.

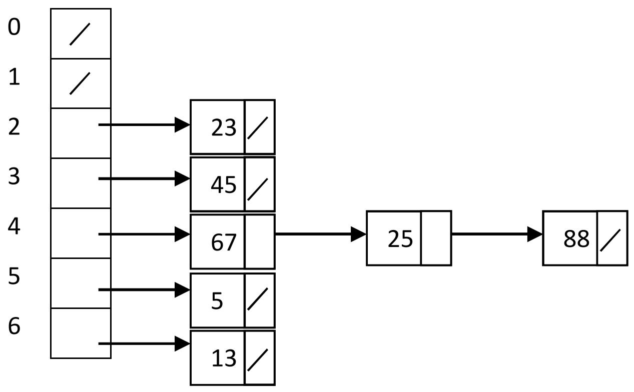
Limitations

- 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 5 had to be stored in slot 5 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.

3.4.2 Separate Chaining

Using the same function h(x) = x%7 and the elements (45, 67, 25, 13, 23, 5 and 88), separate chaining using linked list is represented in Table 3 below.

Table 2: Hash Table Representation



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.

3.4.3 NOF

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

For the 1st Element (x=45)

h(45) = 45%7 = 3

This implies the 1st element will be stored in row 3, column 0 of the n x n array.

For the 2nd Element (x=67)

h(67) = 67%7 = 4

This implies the 2nd element will be stored in row 4, column 0 of the n x n array.

For the 3rd Element (x=25)

h(25) = 25%7 = 4

This implies the 3rd element will be stored in row 4, column 1 of the n x n array.

For the 4th Element (x=13)

h(13) = 13%7 = 6

This implies the 4th element will be stored in row 6, column 0 of the n x n array.

For the 5th Element (x=23)

h(23) = 23%7 = 2

This implies the 5th element will be stored in row 2, column 0 of the n x n array.

For the 6th Element (x=5)

h(5) = 5%7 = 5

This implies the 6th element will be stored in row 5, column 0 of the n x n array.

For the 7th Element (x=88)

h(88) = 88%7 = 4

This implies the 7th element will be stored in row 4, column 2 of the n x n array.

Table 3: Hash Representation

Row numbers of the n x n array (7 x 7)

			0	1	2	3	4	5	6
0		→							
1		→							
2		→	23						
3		→	45						
4		→	67	25	88				
5		→	5						
6		→	13						

A better approach will be to use an n x m array where n is equal to the size of the hash table and m is equal to the number of elements in the row with the highest number of elements. In the case above, the maximum number of elements in row 4 is 3. Hence the value of m is 3. The new look of the array (7 x 3) is shown below in Table 4.

Table 4: Hash Representation

Row numbers of the n x m array

			0	1	2
0		→			
1		→			
2		→	23		
3		→	45		
4		→	67	25	88
5		→	5		
6		→	13		

Limitations

- The multidimensional array is not optimally used because there are empty slots or cells, hence not very efficient storage-wise.

4 Discussion

Studies review that there are other approaches for resolving collision in hash tables. Most common schemes such as linear probing, quadratic probing, double hashing, and separate chaining have some shortfalls. For example, performance is degraded in linear probing as the hash table get nearly full; in quadratic probing, there is no guarantee of finding an empty cell once the table gets more than half full. The proposed scheme is not very different in terms of disadvantages. Its primary issue of concern is the unutilized empty spaces which makes the scheme not efficient storage-wise. Despite this concern, it is proven to be very effective and efficient 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. It could be realized that an optimal implementation of the above technique is to remove or de-allocate the empty spaces.

5 CONCLUSION

NOF (an abbreviation for Nimbe-Opoku-Frimpong) is a collision resolution strategy experimented on numeric values. NOF is being proposed and is earnestly hoped it will add to the body of knowledge due to the fact that it has resolved the problem of collisions in hash tables. The adoption of NOF will in effect eliminate primary clustering and deterioration in hash table efficiency; which are common in the open addressing schemes. Future works to resolve collisions in hash tables will be conducted with 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] Knuth, D.E., *The Art of Computer Programming*: *Sorting and Searching*, volume3. Addison- Wesley Longman, second edition, 1998.
[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, Third Edition, 2009.
[3] 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, in press
[4] Kumar A., Data Structure for 'C' Programming, Second Revised Edition, Laxmi Publications Pvt. Ltd, 2004
[5] Sedgewick, R., *Algorithms in Java*, Third Edition, 2003.