

BLOOM FILTER ADVANCEMENTS

Project Report submitted in partial fulfillment of the requirement
for the degree of

Bachelor of Technology.

in

Information Technology

under the Supervision of

Mr Shailendra Shukla

By

Niharika Verma - 111409

to



Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled “ Bloom Filter Advancements”, submitted by Niharika Verma (111409) in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Wagnaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Mr Shailendra Shukla

Date: 8th May,2015

Assistant Professor (Grade-II)

Acknowledgement

I would like to express my special thanks of gratitude to my project guide Mr. Shailendra Shukla as well as the university who gave me the golden opportunity to do this wonderful project on the topic “ Bloom Filter Advancements ” which also helped me in doing a lot of Research and i came to know about so many new things I am really thankful to them. Secondly i would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Date: 8th May,2015

Niharika Verma

111409

Table of Content

S. No.	Topic	Page No.
1.	Introduction	10
1.1	Operations	11
1.2	Applications	12
2.	Basic Bloom Filter	13
2.1	Basic Bloom Filter Design	13
2.2	Space and time Advantages	15
2.3	Probability of False Positive	16
2.4	Pseudocode for Bloom Filter Insertion	20
2.5	Pseudocode for Bloom Filter Test	21
3.	Basic Counting Bloom Filter	22
3.1	Basic Counting Bloom Filter Design	24
3.2	False Positive Probability Of Counting BF	25
3.3	Pseudocode for CBF Insertion	26
3.4	Pseudocode for CBF Deletion	27

4.	Basic Weighted Bloom Filter	28
4.1	Basic Weighted Bloom Filter Design	28
4.2	Generalization of Bloom Filter	31
5.	Implementation	32
5.1	Java Implementation	32
5.2	Java Implementation Results	38
5.2	Matlab Code for Basic Bloom Filter	46
5.3	Matlab Code for Counting Bloom Filter	49
6.	Results and Conclusion	51
6.1	Complexity Analysis of Basic Bloom Filter	51
6.2	Complexity Analysis of Counting Bloom Filter	53
6.3	Complexity Analysis of Weighted Bloom Filter	54
6.4	Result and conclusion	55

List of Figures

S.No.	Title	Page No.
1.	Bloom Filter Architecture	10
2.	Bloom Filter Example	14
3.	False Positive Rate	18
4.	Counting Bloom Filter Architecture	23
5.	Bloom Filter Operations	38
6.	Bloom Filter Insertion	39
7.	Bloom Filter Search	40
8.	Bloom Filter Check Empty	41
9.	Bloom Filter Clear	42
10.	Bloom Filter Size	43
11.	Bloom Filter Deletion	44

12.	Show Bloom Filter	45
13.	Matlab Implementation of BBF	48
14.	Matlab Implementation of CBF	50

List of Tables

S.No.	Title	Page No.
1.	Key Bloom Filter Parameters	16

Abstract

This project deals with the Bloom filter advancements where a basic bloom filter is designed. The main idea of designing a bloom filter is to reduce the time complexity of search by keeping the false positive probability as minimum as possible.

The drawbacks of the basic bloom filter is seen which are further dealt with in the next level of Bloom filters which is counting bloom filters. At each level, the false positive probability is reduced.

Another advancement is done by creating weighted bloom filter which is an advanced version of the basic bloom filter with further reduced false positivity rate. Hence the comparison between these three filters is done and evaluated.

The matlab implementation of the filters are done to see which filter is best in terms or reduced search time as well lower false positive rate. Also the complexity analysis is added to see how the time and space complexity increase or decrease with the advancements.

CHAPTER 1

1. INTRODUCTION

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not. In other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

The main aim of the bloom filter is to reduce the unnecessary disk access made while searching an element while it is not in the set.

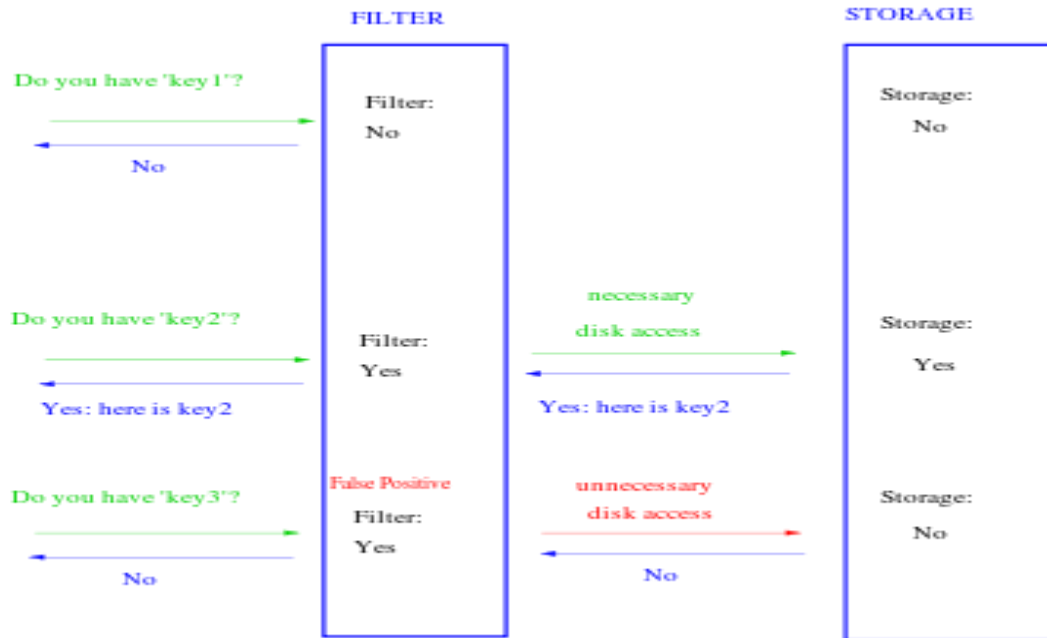


Figure : 1 Bloom Filter Architecture [8]

1.1 Operations

The basic bloom filter supports two operations: **test** and **add**.

Test is used to check whether a given element is in the set or not. If it returns:

- *false* then the element is definitely not in the set.
- *true* then the element is *probably* in the set. The *false positive rate* is a function of the bloom filter's size and the number and independence of the hash functions used.

Add simply adds an element to the set. Removal is impossible without introducing false negatives, but extensions to the bloom filter are possible that allow removal e.g. counting filters.

1.2 Applications

The classic example is using bloom filters to reduce expensive disk (or network) lookups for non-existent keys.

If the element is not in the bloom filter, then we know for sure we don't need to perform the expensive lookup. On the other hand, if it *is* in the bloom filter, we perform the lookup, and we can expect it to fail some proportion of the time (the false positive rate).

CHAPTER 2

BASIC BLOOM FILTER

The basic bloom filter is designed where the false positive rate is checked.

2.1 Basic Bloom Filter Design

- An empty Bloom filter is a bit array of m bits, all set to 0.
- There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution.
- To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
- To query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions.
- If any of the bits at these positions are 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted.
- If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive.

Element Removal :

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

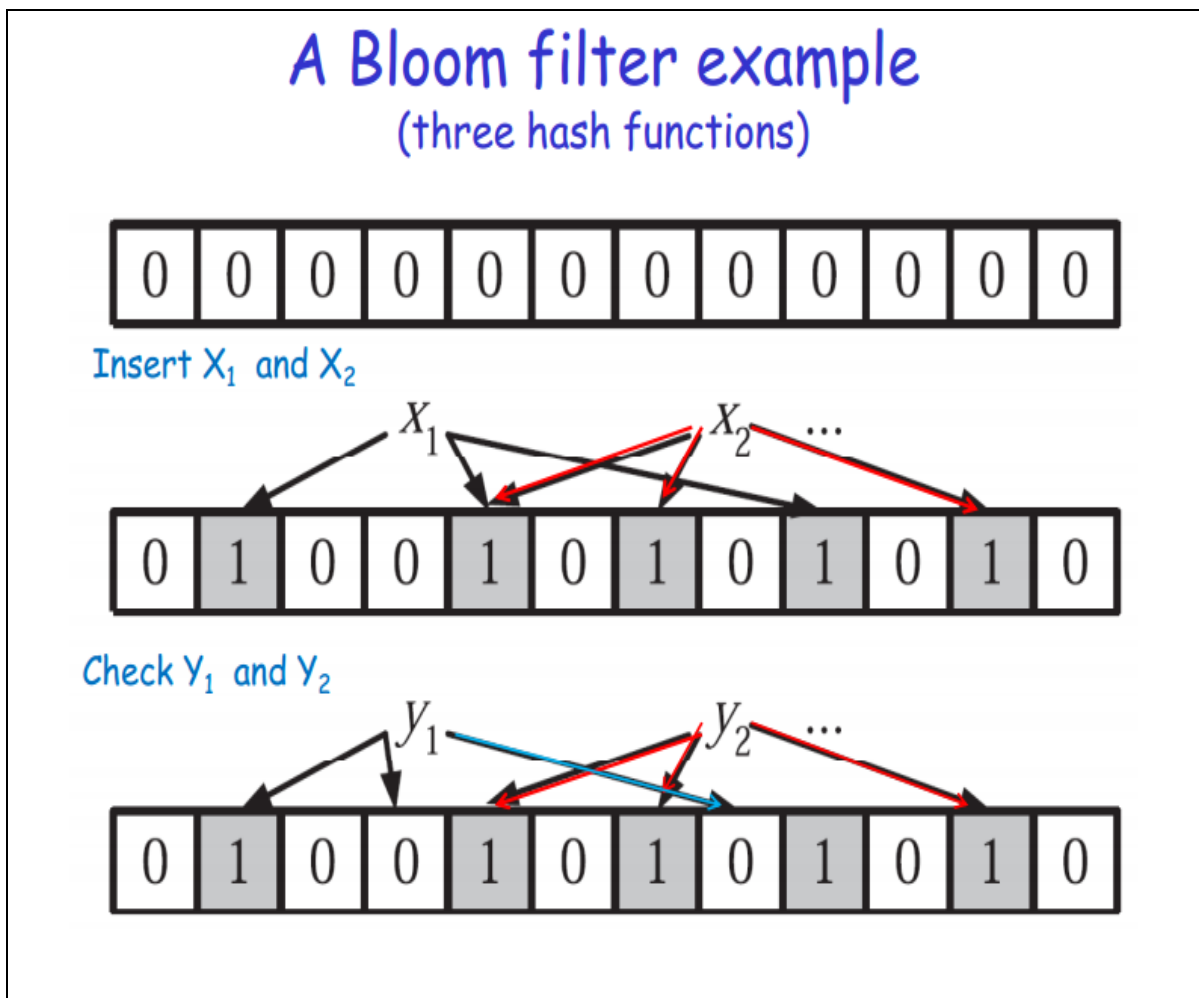


Figure : 2 Bloom Filter Example [6]

2.2 Space And Time Advantages

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, $O(k)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters (k and m) are chosen well, about half of the bits will be set,[2] and these will be apparently random, minimizing redundancy and maximizing information content.

2.3 Probability of False Positives[2]

Bloom Filter Size = m bits

Number of elements in set = n

Number of hash functions used = k

Let m denote the number of bits in the Bloom filter. When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is :

$$1 - \frac{1}{m}.$$

Now, there are k hash functions, and the probability of any of them not having set a specific bit to one is given by :

$$\left(1 - \frac{1}{m}\right)^k.$$

After inserting n elements to the filter, the probability that a given bit is still zero is :

$$\left(1 - \frac{1}{m}\right)^{kn}.$$

And consequently the probability that the bit is one is :

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set. The probability of this happening when the element is not part of the set is given by :

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

Therefore, the probability of false positive in bloom filter is :

$$P_{FP} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k = (1 - p)^k.$$

OPTIMAL NUMBER OF HASH FUNCTIONS :

The false positive probability decreases as the size of the Bloom filter, m , increases. The probability increases with n as more elements are added. Now, we want to minimize the probability of false positives, by minimizing $(1 - e^{-kn/m})^k$ with respect to k . This is accomplished by taking the derivative and equating to zero, which gives the optimal value of k :

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}.$$

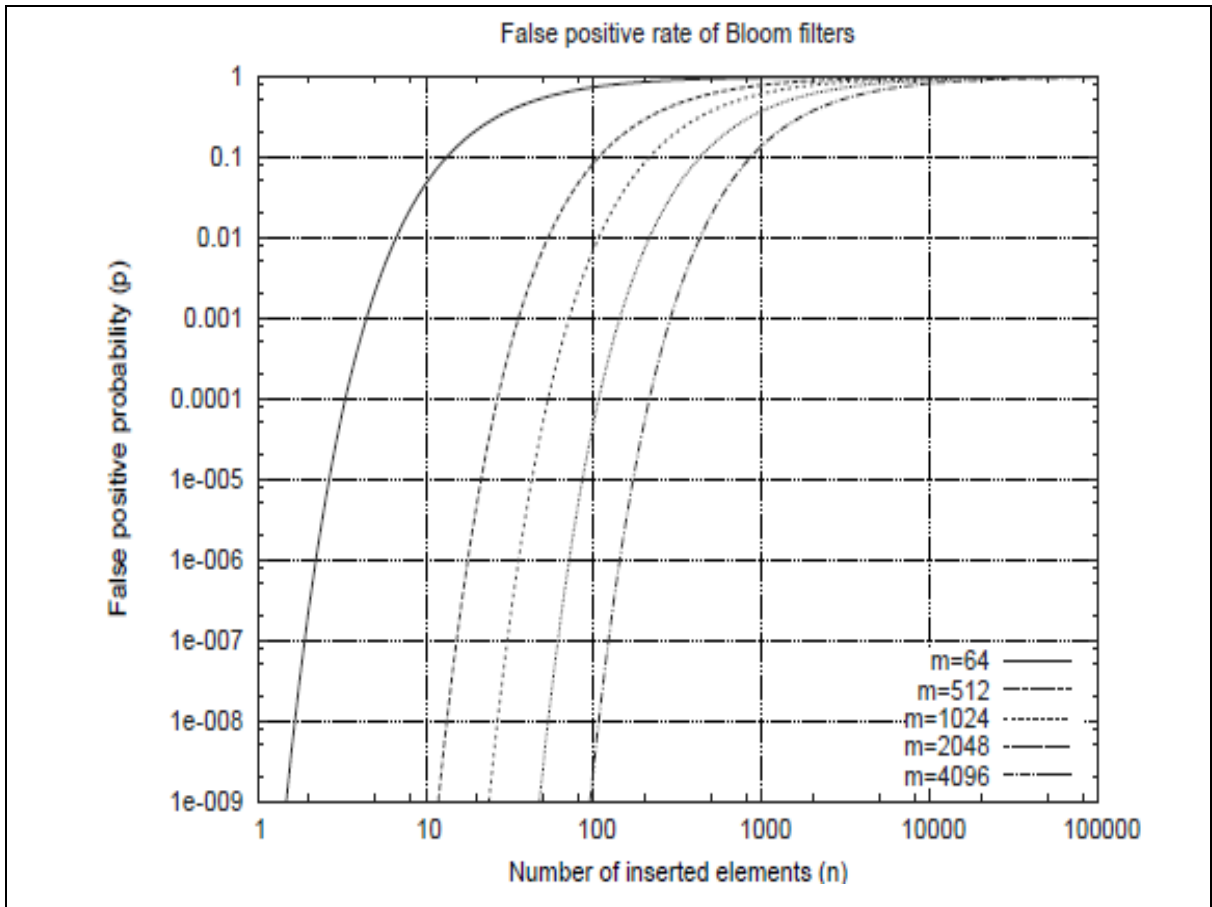


Figure : 3 Bloom Filter False Positivity Rate [2]

TABLE 1
(KEY BLOOM FILTER PARAMETERS)

PARAMETERS	INCREASE
<ul style="list-style-type: none"> • Number of hash functions (k) • Size of filter (m) • Number of elements in the set (n) <p>Higher false positive rate</p>	<ul style="list-style-type: none"> • More computation, lower false positive rate as $k = k_{opt}$ • More space is needed, lower false positive rate • Higher false positive rate

PSEUDOCODES FOR IMPLEMENTATION

The basic bloom filter algorithms are designed for two operations : insertion and checking if an element might be present in the set or not.

2.4 Pseudocode for Bloom Filter insertion

Data: x is the object key to insert into the Bloom filter.

Function: *insert(x)*

for j : 1 . . . k **do**

/* Loop all hash functions k */

i = $h_j(x)$;

if $B_i == 0$ **then**

/* Bloom filter had zero bit at
position i */

$B_i = 1$;

end

end

2.5 Pseudocode for Bloom Member Test

Data: x is the object key for which membership is tested.

Function: $ismember(x)$ returns true or false to the membership test

$m \leftarrow 1;$

$j \leftarrow 1;$

while $m == 1$ *and* $j \leq k$ **do**

$i \leftarrow h_j(x);$

if $B_i == 0$ **then**

$m \leftarrow 0;$

end

$j \leftarrow j + 1;$

end

 return $m;$

CHAPTER 3

BASIC COUNTING BLOOM FILTER

A Bloom filter can easily be extended to support deletions by adding a counter for each element of the data structure. A counting Bloom filter has m counters along with the m bits. Fan et al. first introduced the idea of a counting Bloom filter in conjunction with Web caches.

The structure works in a similar manner as a regular Bloom filter; however, it is able to keep track of insertions and deletions. In a counting Bloom filter, each entry in the Bloom filter is a small counter associated with a basic Bloom filter bit. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. To avoid counter overflow, we need choose sufficiently large counters.

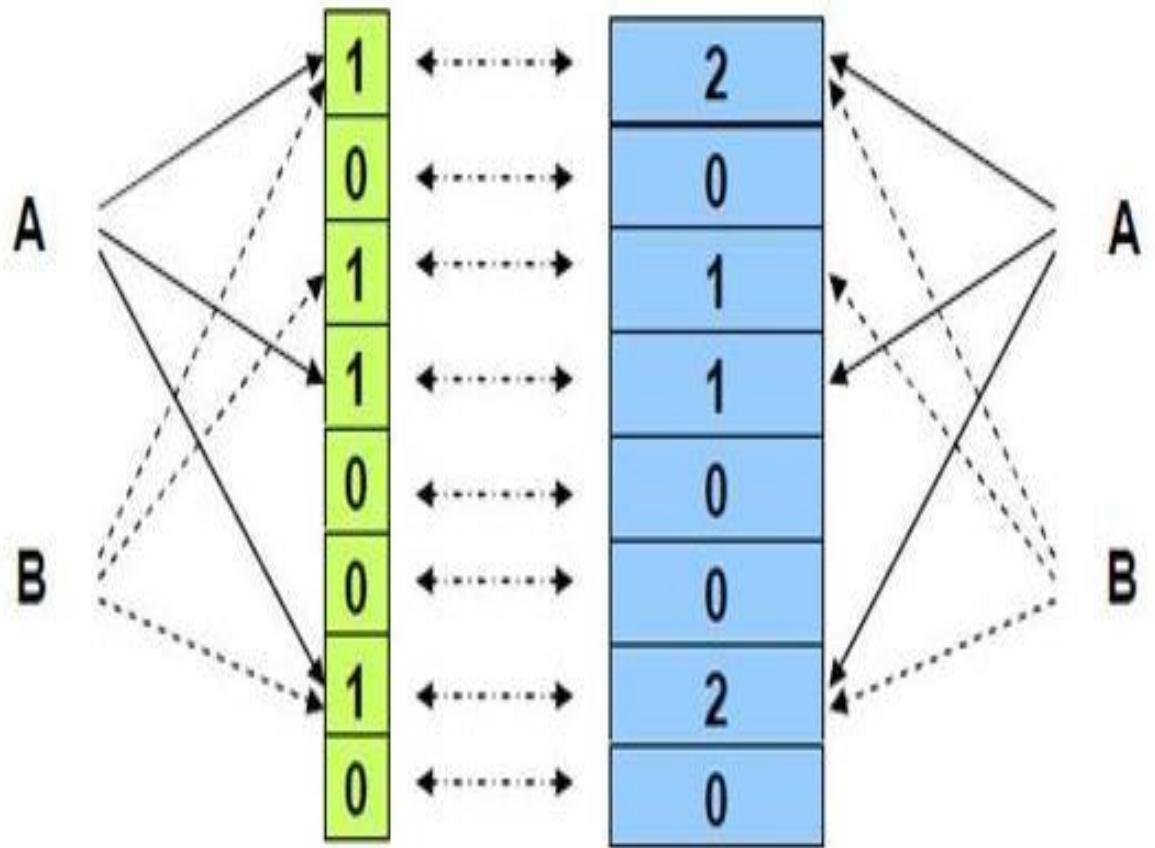


Figure :4 Counting Bloom Filter Architecture [7]

3.1 Basic Counting Bloom Filter Design

- A basic bloom filter with m bits size is taken.
- K number of hash functions are used using the optimal value of k
- A counting filter with size same as the basic bloom filter is used of m bits.
- Each time the bit at a particular index is set, the value of count corresponding to that array index is incremented.
- At the time of deletion, the value of count at the index provided by different hash functions is checked.
- If the value of count at all the positions is zero, then the element can be deleted since that position is not set by any of the elements in the set.
- If the values of count is non-zero , then that element cannot be deleted.
- The space required for such an architecture is twice of that used in the basic bloom filter design whereas the complexity remains same.

3.2 False Positive Probability of Counting Bloom Filter[2]

The probability that the i th counter is incremented j times is a binomial random variable:

$$P(c(i) = j) = \binom{nk}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{nk-j}$$

The counter counts the number of times that the bit is set to one. All the counts are initially zero. The probability that any count is greater or equal to j :

$$\Pr(\max(c) \geq j) \leq m \binom{nk}{j} \frac{1}{m^j} \leq m \left(\frac{enk}{jm}\right)^j .$$

As already mentioned the optimum value for k (over reals) is $\ln 2m/n$ so assuming that the number of hash functions is less than $\ln 2m/n$ we can further bound :

$$\Pr(\max(c) \geq j) \leq m \left(\frac{e \ln 2}{j}\right)^j$$

PSEUDOCODES FOR IMPLEMENTATION

3.3 Pseudocode for Counting Bloom Filter insertion

Data: x is the item to be inserted.

Function: $insert(x)$

for $j : 1 \dots k$ **do**

/* Loop all hash functions k */

$i = h_j(x);$

/* Increment counter C_i */

$C_i = C_i + 1;$

if $B_i == 0$ **then**

/* Bit is zero at position i */

$B_i = 1;$

end

end

3.4 Pseudocode for Counting Bloom Filter deletion

Data: x is the item to be removed.

Function: $delete(x)$

for $j : 1 \dots k$ **do**

/* Loop all hash functions k */

$i = h_j(x);$

/* Decrement counter C_i */

$C_i = C_i - 1;$

if $C_i = 0$ **then**

/* Reset bit at position i */

$B_i = 0;$

end

end

Algorithm 3.3 presents the pseudocode for the insert operation for element x with counting. The operation increments the counter of each bit to which x is hashed. The counting structure supports the removal of elements using the delete operation presented in Algorithm 3.4. The delete decrements the counter of each bit to which x is hashed. The corresponding bit is reset to zero when the counter becomes zero.

CHAPTER 4

BASIC WEIGHTED BLOOM FILTER

The basic weighted bloom filter is based on the elements' query frequencies and their probabilities of being members. In many applications, query frequencies and membership likelihoods are estimated or collected with well developed techniques. Statistics of such information are maintained, especially for a set of 'hot' categories.

The filter's false positive probability is a weighted sum of each individual element's false positive probability, where the weight corresponding to an individual element is positively correlated with the element's query frequency and is negatively correlated with the element's probability of being a member. Therefore, we would in general like to assign more hash functions to an element with a higher query frequency or with a lower probability of being a member, in order to reduce the false-positive probability of an element with a higher weight.

4.1 Basic Weighted Bloom Filter Design

- Same as the traditional bloom filter, it uses m bits to record the n elements of the set S .
- There are N elements in the universe and $N \gg n$
- Probability of an element 'e' being a member is x_e

$$X_e = \begin{cases} 1 & , \text{ if } e \in S \\ 0 & , \text{ if } e \notin S \end{cases}$$

- $E\{X_e\} = x_e$
- For each element $e \in U$, denote its query frequency by f_e .
- Denote the number of hash functions used for an element e by k_e

- Probability that a query is about an element $a \in U$:

$$P = \text{freq of element } a / \text{freq of all elements } \in U$$

- Total number of hash functions used for elements in S :

$$K = \sum_{e \in S} k_e = \sum_{e \in U} X_e \cdot k_e.$$

- Probability that a bit is not set :

$$p = \left(1 - \frac{1}{m}\right)^K \approx e^{-\frac{K}{m}}.$$

- Prob (Bit = 0) :p

$$\text{Prob (Bit = 1) : } 1-p$$

- The probability of a false positive is the weighted sum of the false positive probabilities of all non-members in the universe, denoted by $FPF[3]$:

$$\begin{aligned}
P_{FP} &= \frac{\sum_{e \in U-S} f_e (1-p)^{k_e}}{\sum_{e \in U-S} f_e} \\
&= \frac{\sum_{e \in U} (1-X_e) \cdot f_e (1-p)^{k_e}}{\sum_{e \in U} (1-X_e) \cdot f_e} \\
&= \sum_{e \in U} \frac{(1-X_e) f_e}{\sum_{i \in U} (1-X_i) \cdot f_i} \cdot (1-p)^{k_e}.
\end{aligned}$$

- Normalized query frequency :

$$r_e = \frac{(1-X_e) f_e}{\sum_{i \in U} (1-X_i) \cdot f_i}.$$

- False Positive Probability :

$$P_{FP} = \sum_{e \in U} r_e (1-p)^{k_e}.$$

- Value of k_e for all $e \in U$:

$$k_e = \frac{m}{n} \cdot \ln 2 + (\ln E\{r_e\} - \sum_{i \in U} \frac{x_i}{n} \cdot \ln E\{r_i\}) \cdot \frac{1}{\ln 2};$$

- Expectation of False Positive Probability :

$$E\{P_{FP}\} = 2^{-(m/n) \ln 2} \cdot N \cdot \prod_{e \in U} E\{r_e\}^{x_e/n}.$$

4.2 Generalization Of Bloom Filter

We compare the weighted Bloom filter to the traditional Bloom filter and show that our result is a generalization and further optimization of the traditional optimal configuration.

Optimal Configuration For Bloom Filter :

$$\begin{aligned}
 p &= 1/2; \\
 k_e &= (m/n) \ln 2, \forall e \in U; \\
 P_{FP} &= 2^{-(m/n) \ln 2}.
 \end{aligned}$$

In the weighted Bloom filter setting, all the elements should be treated the same when no knowledge about query frequencies or membership likelihood is available.

By plugging $x_e = n/N$ and $E\{r_e\} = 1/N$, we obtain the formulas $k_e = (m/n) \ln 2$ and $P_{FP} = 2^{-(m/n) \ln 2}$, the same as the second and the third formulas of the traditional Bloom filter's configuration.

$$\begin{aligned}
 E\{P_{FP}\} &= 2^{-(m/n) \ln 2} \cdot N \cdot \prod_{e \in U} E\{r_e\}^{x_e/n} \\
 &= 2^{-(m/n) \ln 2} \cdot \frac{\prod_{e \in U} E\{r_e\}^{1/N}}{\sum_{e \in U} E\{r_e\}/N} \\
 &\leq 2^{-(m/n) \ln 2}.
 \end{aligned}$$

CHAPTER 5

5.1 Implementation in Java

BLOOM FILTER :

```
package bloom.filter;
import java.util.*;
import java.security.*;
import java.math.*;
import java.nio.*;

/* Class BloomFilter */

class BloomFilter
{
    private byte[] set;
    private Integer[] countSet;
    private Integer[] frequency;
    private int keySize, setSize, size;
    private MessageDigest md;
    /* Constructor */

    public BloomFilter(int capacity, int k)
    {
        setSize = capacity;
        set = new byte[setSize];
        countSet = new Integer[setSize];
        frequency = new Integer[20];
        keySize = k;
        size = 0;
        for(int i=0; i<setSize ; i++){
            countSet[i] = 0;
        }
        for(int i=0; i<20; i++){
            frequency[i] = 0;
        }
        try
        {
            md = MessageDigest.getInstance("MD5");
        }
        catch (NoSuchAlgorithmException e)
        {
```



```
        throw new IllegalArgumentException("Error : MD5 Hash not found");
    }
}
```

```
/* Function to clear bloom set */
public void makeEmpty()
{
    set = new byte[setSize];
    countSet = new Integer[setSize];
    for(int i =0; i<setSize; i++){
        countSet[i]=0;
    }
    size = 0;
    try
    {
        md = MessageDigest.getInstance("MD5");
    }
    catch (NoSuchAlgorithmException e)
    {
        throw new IllegalArgumentException("Error : MD5 Hash not found");
    }
}
```

```
/* Function to check is empty */
public boolean isEmpty()
{
    return size == 0;
}
```

```
/* Function to get size of objects added */
public int getSize()
{
    return size;
}
```

```

/* Function to get hash - MD5 */
private int getHash(int i)
{
    md.reset();
    byte[] bytes = ByteBuffer.allocate(4).putInt(i).array();
    md.update(bytes, 0, bytes.length);
    return Math.abs(new BigInteger(1, md.digest()).intValue()) % (set.length - 1);
}

```

```

/* Function to add an object */
public void add(Object obj)
{
    int element = (Integer)obj;
    frequency[(Integer)obj]+=1;
    System.out.println("Element is : "+element);
    int[] tmpset = getSetArray(obj);
    System.out.println("hello");
    for (int i : tmpset){
        set[i] = 1;
        countSet[i] +=1;
    }
    size++;
}

```

```

/* Function to check is an object is present */
public boolean contains(Object obj)
{
    int[] tmpset = getSetArray(obj);
    for (int i : tmpset)
        if (set[i] != 1)
            return false;
    return true;
}

```

```

/* Function to get set array for an object */
private int[] getSetArray(Object obj)
{
    int freq ;
    int freqSum=0;
    for(int i =0; i<20; i++){
        if(frequency[i]>0)
            freqSum += 1;
    }
    freq = (frequency[(Integer)obj]/freqSum)*100;
    System.out.println("freq : "+freq);
    if(freq >= 50){
        keySize = keySize*2;
    }
    int[] tmpset = new int[keySize];
    tmpset[0] = getHash(obj.hashCode());
    for (int i = 1; i < keySize ; i++){
        tmpset[i] = (getHash(tmpset[i - 1]));
    }
    System.out.print("Keys are : ");
    for(int i =0; i<keySize; i++){
        System.out.print(tmpset[i]+"\\t");
    }
    return tmpset;
}

public void delete(Object obj)
{
    int[] tmpset = getSetArray(obj);
    for (int i : tmpset){
        if(countSet[i] == 1){
            set[i] = 0;
            countSet[i] -=1;
        }else
        {
            System.out.println("Element cannot be deleted");
            break;
        }
    }
    size--;
}

```

```

public void showBF(){
    System.out.println("Index \tBloom \t Count");
    for(int i=0; i<setSize ; i++){
        System.out.println(i +"\t"+set[i]+" \t" + countSet[i]);
    }
}
}
}

```

BLOOM FILTER TEST :

```

package bloom.filter;

import java.util.Scanner;

/**
 *
 * @author Niharika
 */
public class BloomFilterTest
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Bloom Filter Test\n");

        System.out.println("Enter set capacity and key size");
        BloomFilter bf = new BloomFilter(scan.nextInt() , scan.nextInt());

        char ch;
        /* Perform bloom filter operations */
        do
        {
            System.out.println("\nBloomFilter Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. contains");
            System.out.println("3. check empty");
            System.out.println("4. clear");
            System.out.println("5. size");
            System.out.println("6. Delete");
            System.out.println("7. Show Bloom Filter");

            int choice = scan.nextInt();

```

```

switch (choice)
{
case 1 :
    System.out.println("Enter integer element to insert");
    bf.add( new Integer(scan.nextInt() ));
    break;
case 2 :
    System.out.println("Enter integer element to search");
    System.out.println("Search result : "+ bf.contains( new Integer(scan.nextInt())
));
    break;
case 3 :
    System.out.println("Empty status = "+ bf.isEmpty());
    break;
case 4 :
    System.out.println("\nBloom set Cleared");
    bf.makeEmpty();
    break;
case 5 :
    System.out.println("\nSize = "+ bf.getSize() );
    break;
case 6:
    System.out.println("Enter integer element to delete");
    bf.delete( new Integer(scan.nextInt() ));
    break;
case 7:
    System.out.println("Bloom Filter is : \n");
    bf.showBF();
    break;
default :
    System.out.println("Wrong Entry \n ");
    break;
}

    System.out.println("\nDo you want to continue (Type y or n) \n");
    ch = scan.next().charAt(0);
} while (ch == 'Y' || ch == 'y');
}
}

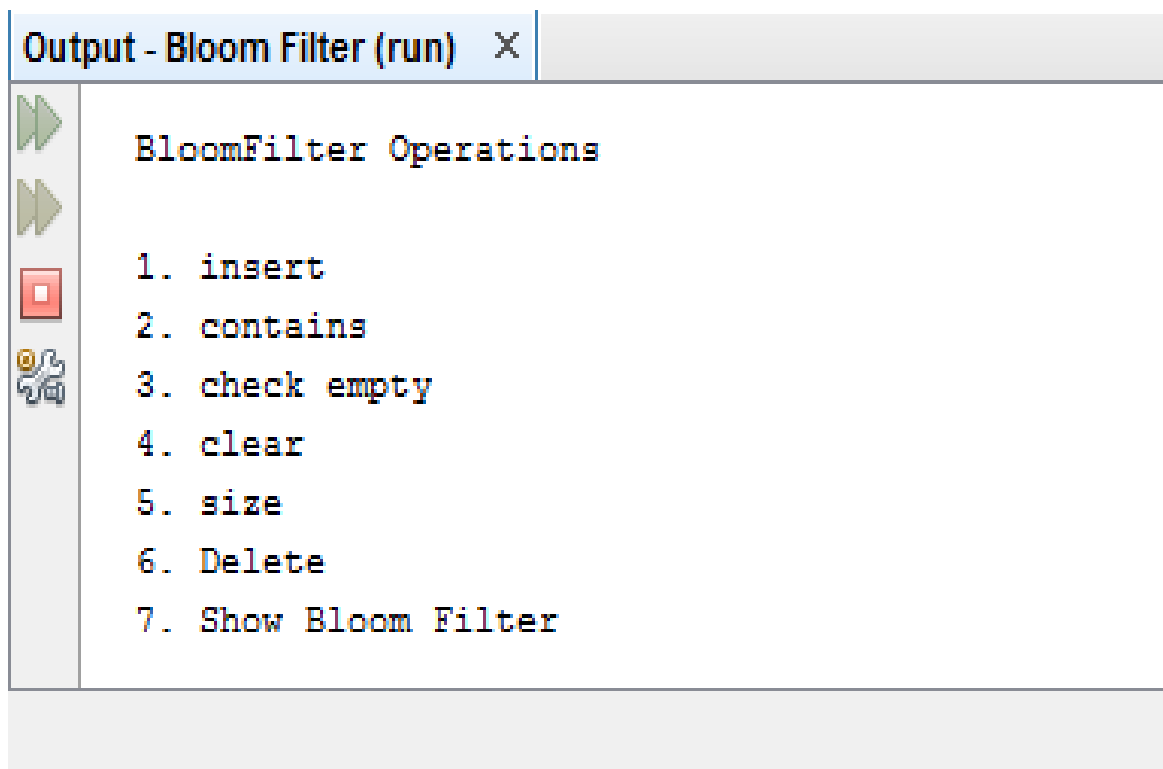
```

5.2 Results :

Operations :

The operations used in the Bloom filter are as follows:

- Insertion
- Contains
- Check Empty
- Clear
- Size
- Delete
- Show Bloom Filter



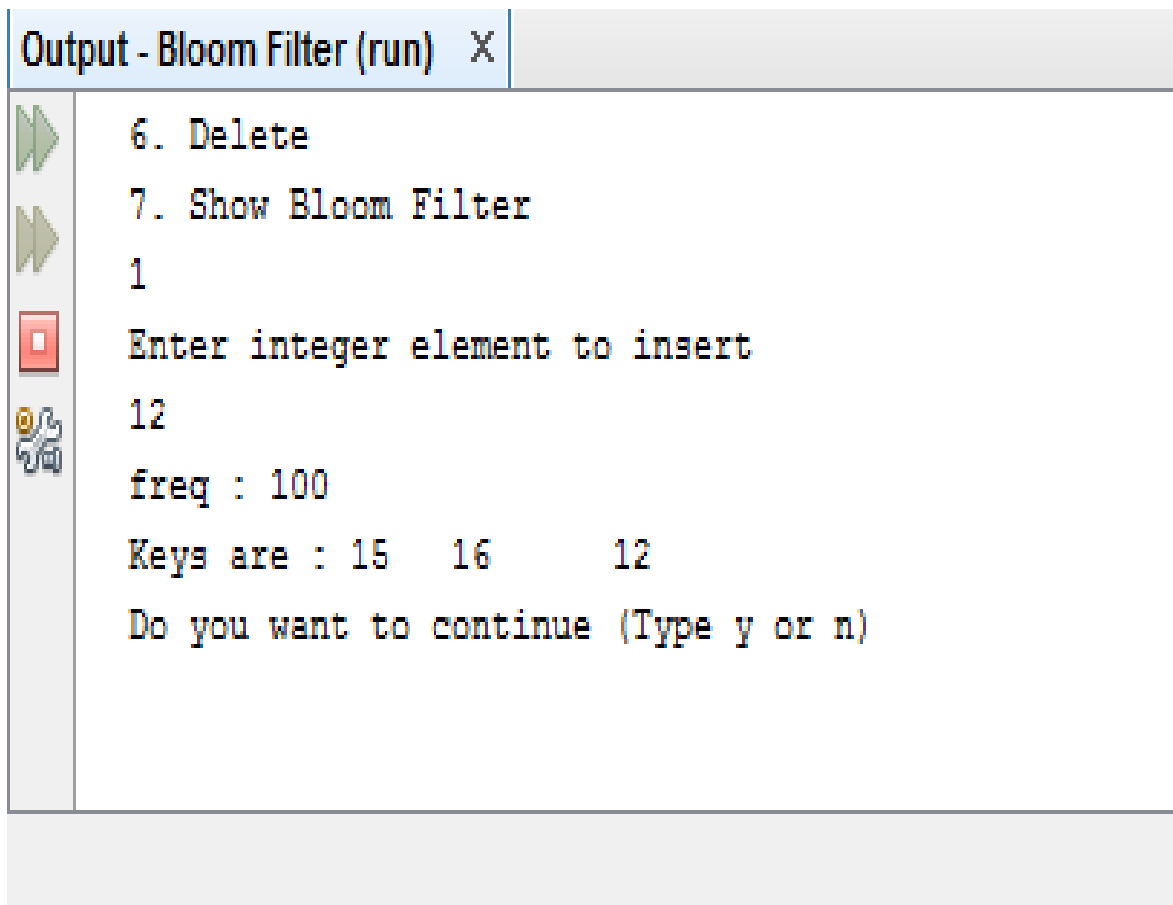
```
Output - Bloom Filter (run) X
BloomFilter Operations
1. insert
2. contains
3. check empty
4. clear
5. size
6. Delete
7. Show Bloom Filter
```

Figure : 5 Bloom Filter Operations

Insertion :

The insertion operation is performed to add elements into the Bloom Filter. After entering the element, the element's frequency is calculated for calculating the weight of the element in the Bloom filter to set the key size of the frequently occurring elements.

The following screenshot shows the element insertion , the frequency of the element being inserted. Also, the corresponding keys are calculated from the formula in the above code. The calculated keys are then set to 1 which shows the array index being set by the inserted element.



```
Output - Bloom Filter (run) X
6. Delete
7. Show Bloom Filter
1
Enter integer element to insert
12
freq : 100
Keys are : 15  16  12
Do you want to continue (Type y or n)
```

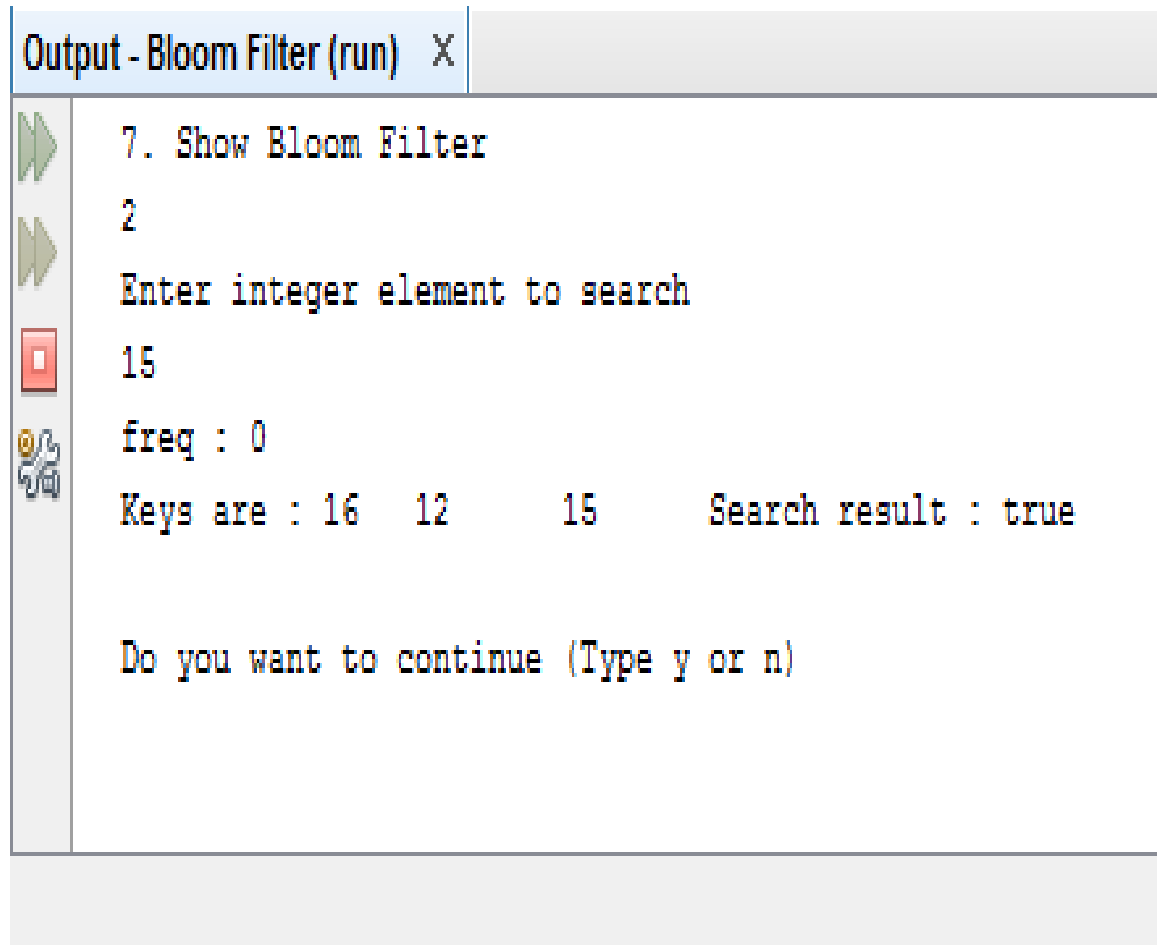
Figure 6 : Bloom Filter Insertion

Contains :

This function is used to check whether a given element is in the set or not. The frequency of the element is not calculated since no weight needs to be assigned for the element to be searched.

The search result returns true if the element may be in the set i.e. the search result if true of the keys calculated by the element are all set to 1.

If the calculated keys are all found out to be 0, it means the element is definitely not a part of the set, else, it may or may not be in the set.



```
Output - Bloom Filter (run) X
7. Show Bloom Filter
2
Enter integer element to search
15
freq : 0
Keys are : 16 12 15 Search result : true

Do you want to continue (Type y or n)
```

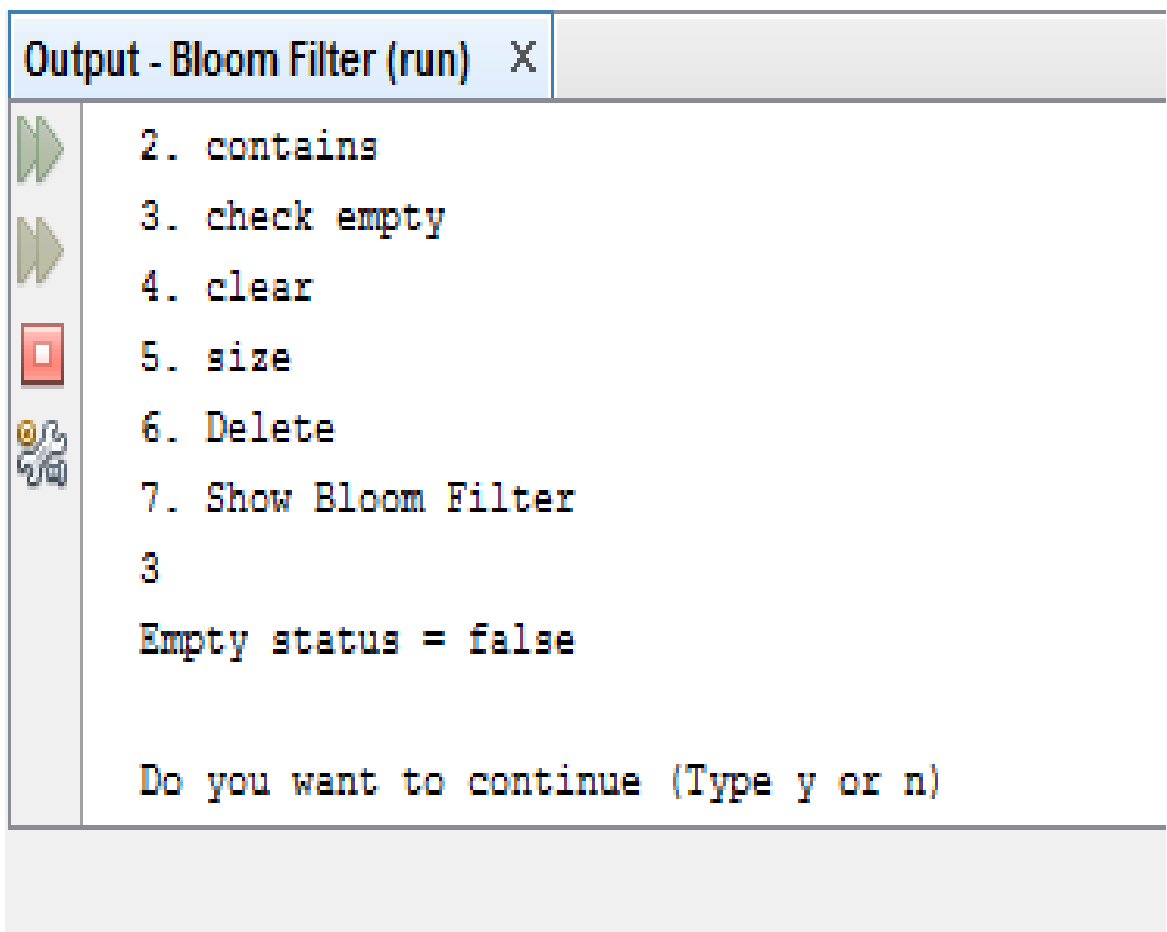
Figure 7 : Bloom Filter Search

Check Empty :

This function is used in the Bloom Filter to check whether the set or the bloom filter is empty or not.

If the bloom filter is empty and there are no elements in the set, then the search result shows true.

The search result is false if there are elements already inserted in the set.



```
Output - Bloom Filter (run) X
2. contains
3. check empty
4. clear
5. size
6. Delete
7. Show Bloom Filter
3
Empty status = false

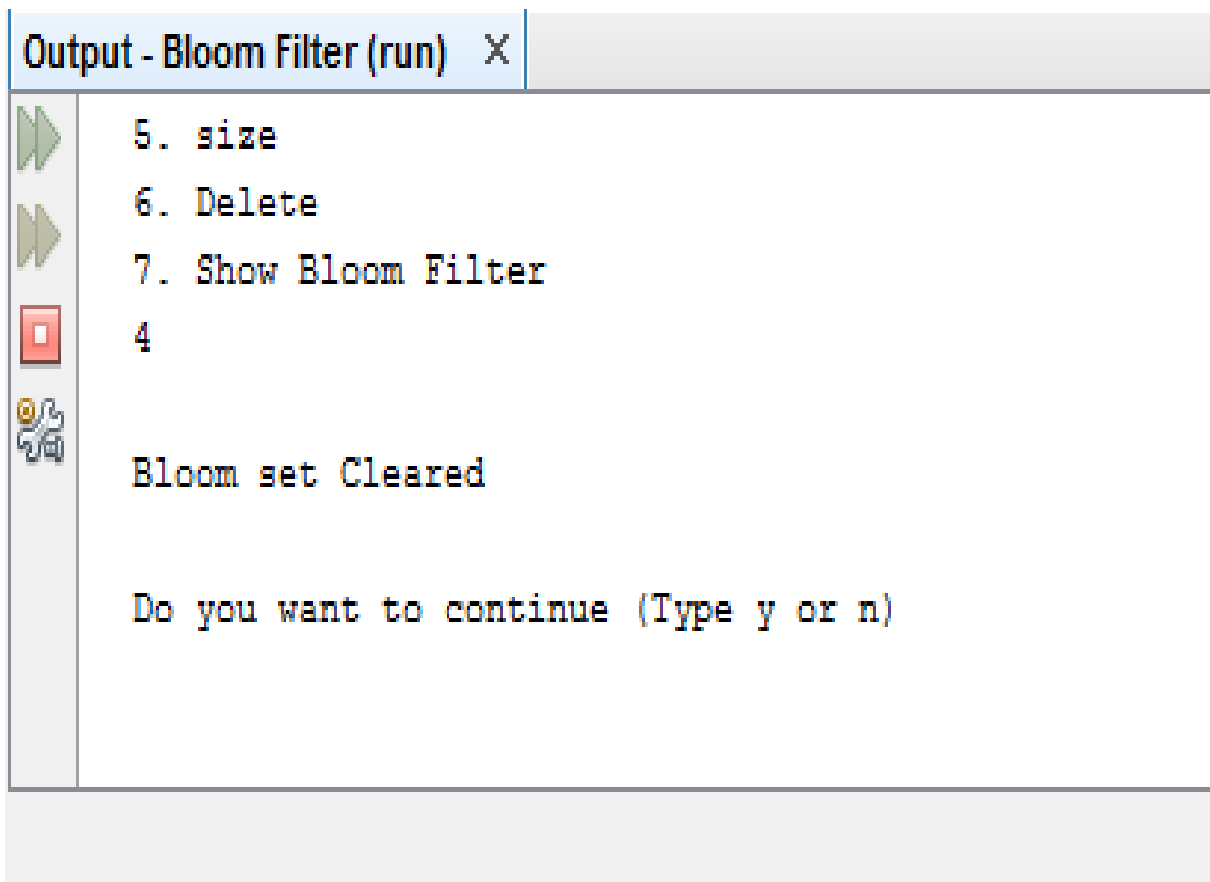
Do you want to continue (Type y or n)
```

Figure 8 : Bloom Filter Check Empty

Clear :

This function is used to clear the Bloom Filter. If the bloom filter contains already inserted elements, all the elements from the Bloom Filter are deleted and the array indexes previously set to 1 for different element's keys are again set to 0.

In case of counting bloom filter, along with the array index of the basic bloom filter, the index in the counter array are also set to 0 for all the corresponding array position of the keys.

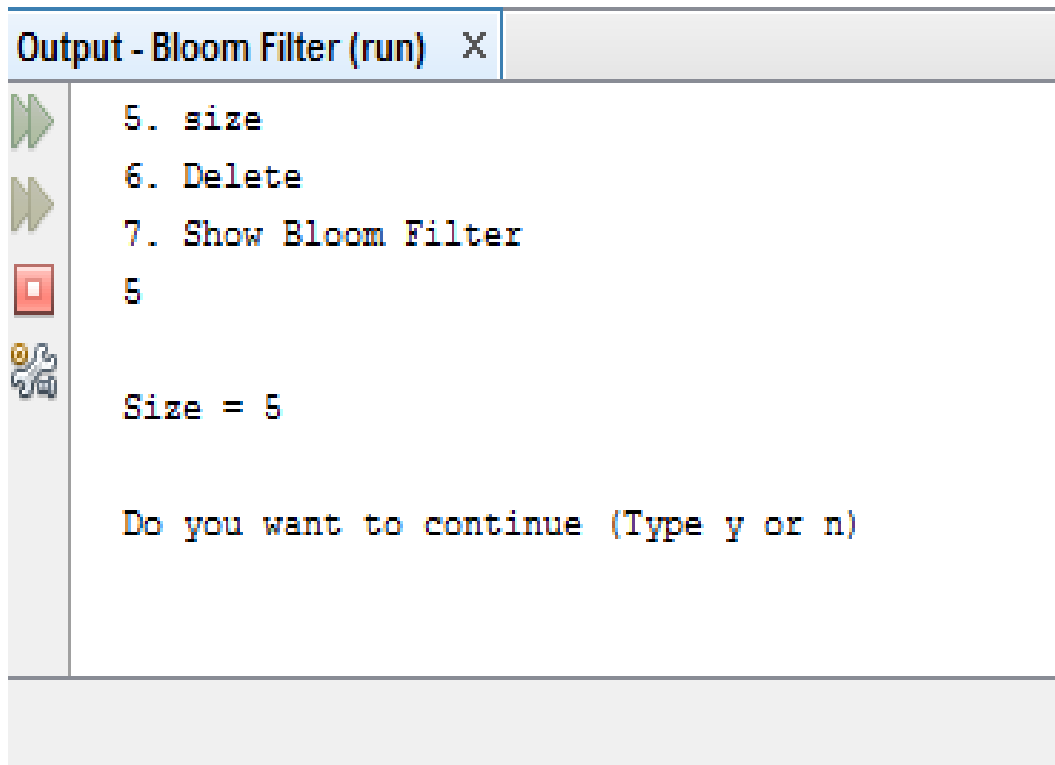


```
Output - Bloom Filter (run) X
5. size
6. Delete
7. Show Bloom Filter
4
Bloom set Cleared
Do you want to continue (Type y or n)
```

Figure 9 : Bloom Filter Clear

Size :

This function shows the size of the bloom filter initially entered by the user.

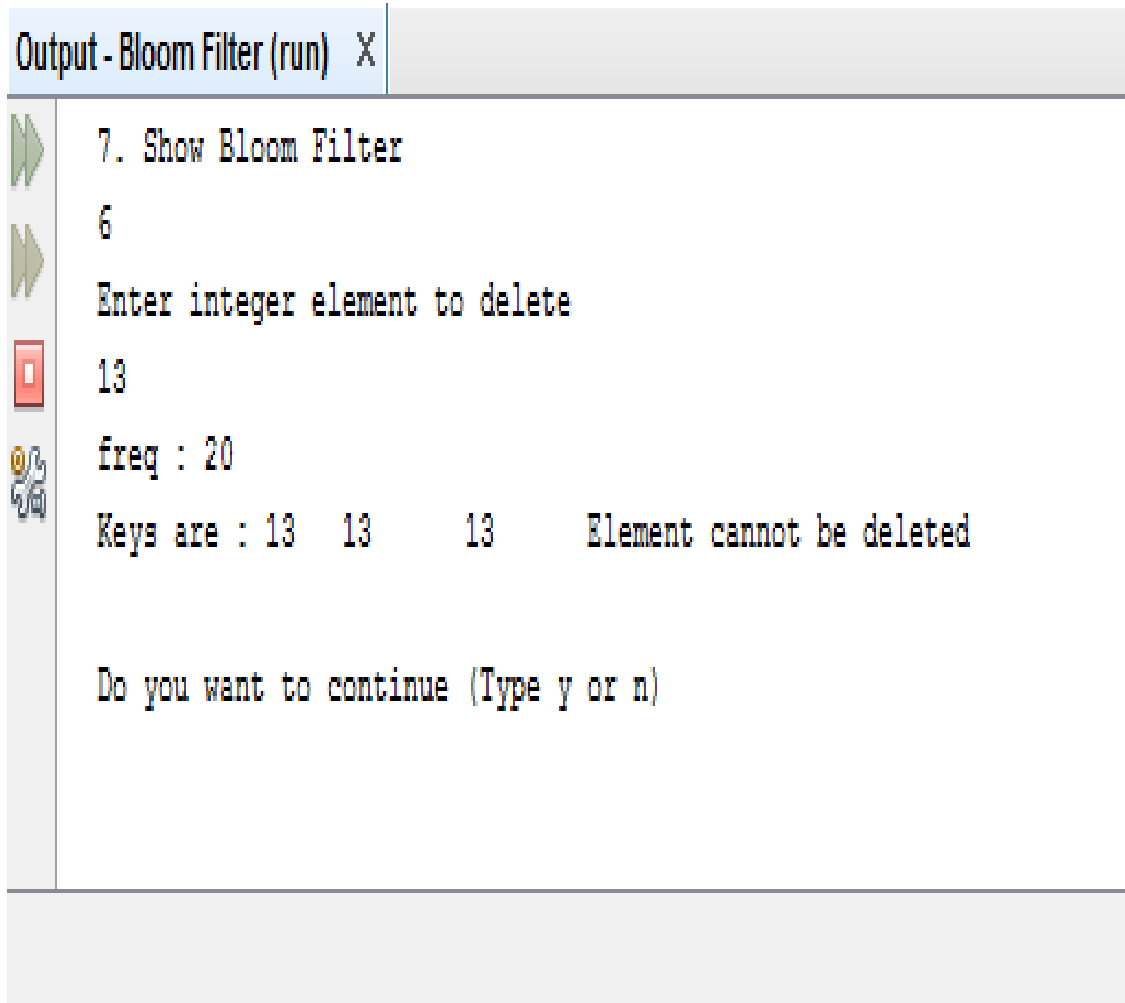


```
Output - Bloom Filter (run) X
5. size
6. Delete
7. Show Bloom Filter
5
Size = 5
Do you want to continue (Type y or n)
```

Figure 10 : Bloom Filter Size

Delete :

This function is used to delete a particular element from the filter. The keys of the element to be deleted are calculated. If the bit at even a single array index of the key is set to 1, then the element cannot be deleted otherwise it can be deleted.



```
Output - Bloom Filter (run) X
7. Show Bloom Filter
6
Enter integer element to delete
13
freq : 20
Keys are : 13 13 13 Element cannot be deleted

Do you want to continue (Type y or n)
```

Figure 11 : Bloom Filter Deletion

Show Bloom Filter :

This function is used to show the entire bloom filter. It shows the basic bloom filter array in which are the keys calculated are set to 1. Also the counting bloom filter array contains the number of times an index of the basic bloom filter array is set to 1.

Output - Bloom Filter (run) X			
	Index	Bloom	Count
▶▶	0	1	1
▶▶	1	0	0
■	2	0	0
🔧	3	0	0
	4	1	3
	5	0	0
	6	0	0
	7	0	0
	8	0	0

Output - Bloom Filter (run) X			
	Index	Bloom	Count
▶▶	9	0	0
▶▶	10	0	0
▶▶	11	1	1
■	12	1	2
🔧	13	1	3
	14	0	0
	15	1	2
	16	1	2
	17	1	1
	18	0	0

Figure 12 : Show Bloom Filter

5.3 Matlab Code For Basic Bloom Filter :

```
clc;
clear all;

m=64;
n=[1:0.001:1000];
k=(m./n)*log(2);
y=(1-exp((-k.*n)./m)).^k;
subplot(2,2,1);
plot(n,y,'red');
xlabel('Number of elements');
ylabel('False Positivity');

m=128;
n=[1:0.001:1000];
k=(m./n)*log(2);
y=(1-exp((-k.*n)./m)).^k;
subplot(2,2,2);
plot(n,y,'red');
xlabel('Number of elements');
ylabel('False Positivity');

m=512;
n=[1:0.001:1000];
k=(m./n)*log(2);
y=(1-exp((-k.*n)./m)).^k;
subplot(2,2,3);
plot(n,y,'red');
xlabel('Number of elements');
ylabel('False Positivity');
```

```
m=1024;
n=[1:0.001:1000];
k=(m./n)*log(2);
y=(1-exp((-k.*n)./m)).^k;
subplot(2,2,4);
plot(n,y,'red');
xlabel('Number of elements');
ylabel('False Positivity');

xlabel('Number of elements');
ylabel('False Positivity');
```

RESULT :

Here , we have considered four values of $m = 64, 128, 512, 1024$. We have seen that as size of the bloom filter increases, the false positive rate decreases for the basic bloom filter.

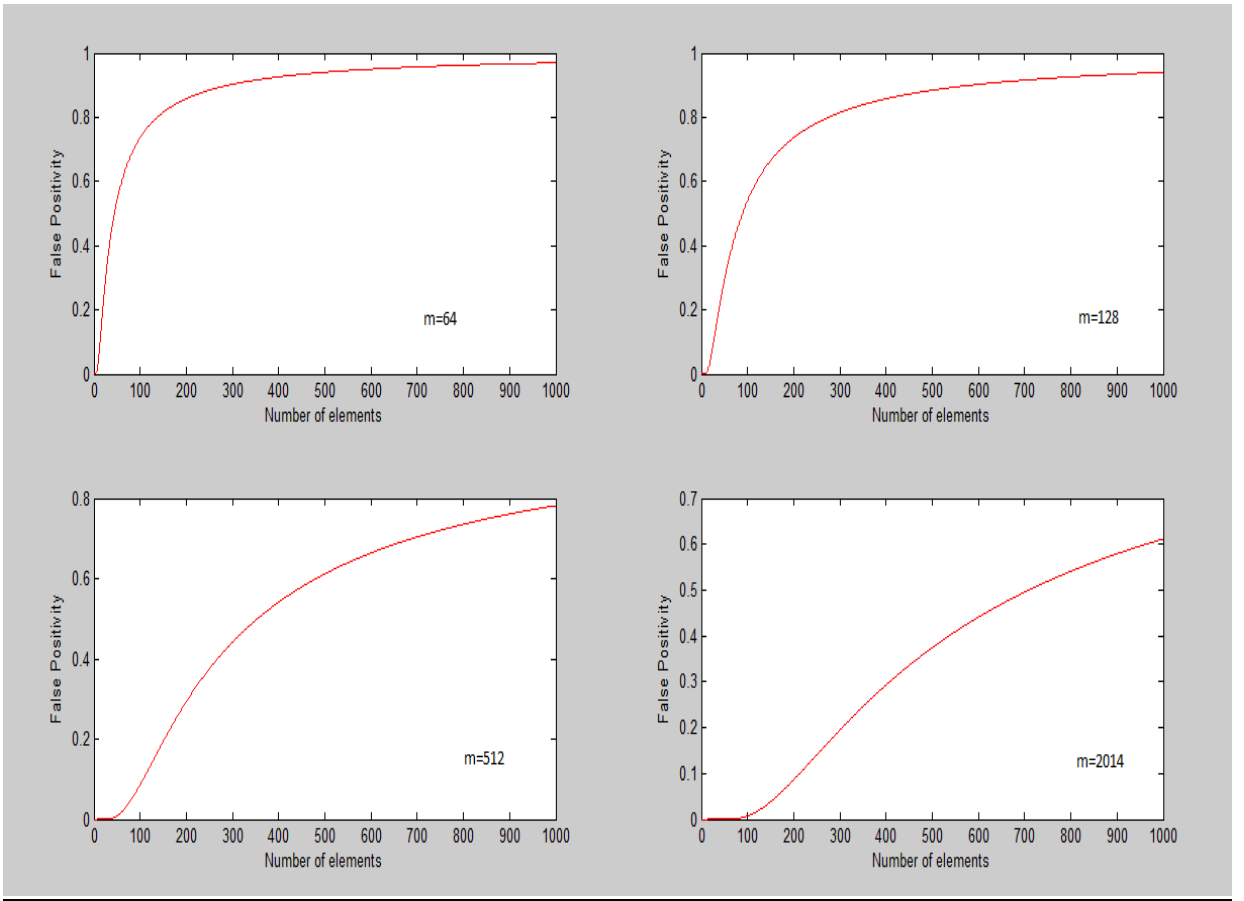


Figure : 13 Matlab Implementation of BBF.

5.4 Matlab Code For Counting Bloom Filter :

```
clc;
clear all;

m=64;
n=[1:0.001:1000];
k=round((m./n)*log(2));
j=16;
z = m .* (((2.71 .*n.*k)./(j.*m))).^j;
subplot(2,2,1);
plot(n,z);
xlabel('Number of elements');
ylabel('False Positivity');

m=128;
n=[1:0.001:1000];
k=round((m./n)*log(2));
j=16;
z = m .* (((2.71 .*n.*k)./(j.*m))).^j;
subplot(2,2,2);
plot(n,z);
xlabel('Number of elements');
ylabel('False Positivity');

m=512;
n=[1:0.001:1000];
k=round((m./n)*log(2));
j=16;
z = m .* (((2.71 .*n.*k)./(j.*m))).^j;
subplot(2,2,3);
plot(n,z);
xlabel('Number of elements');
ylabel('False Positivity');

m=1024;
n=[1:0.001:1000];
k=round((m./n)*log(2));
j=16;
z = m .* (((2.71 .*n.*k)./(j.*m))).^j;
subplot(2,2,4);
plot(n,z);
xlabel('Number of elements');
ylabel('False Positivity');
```

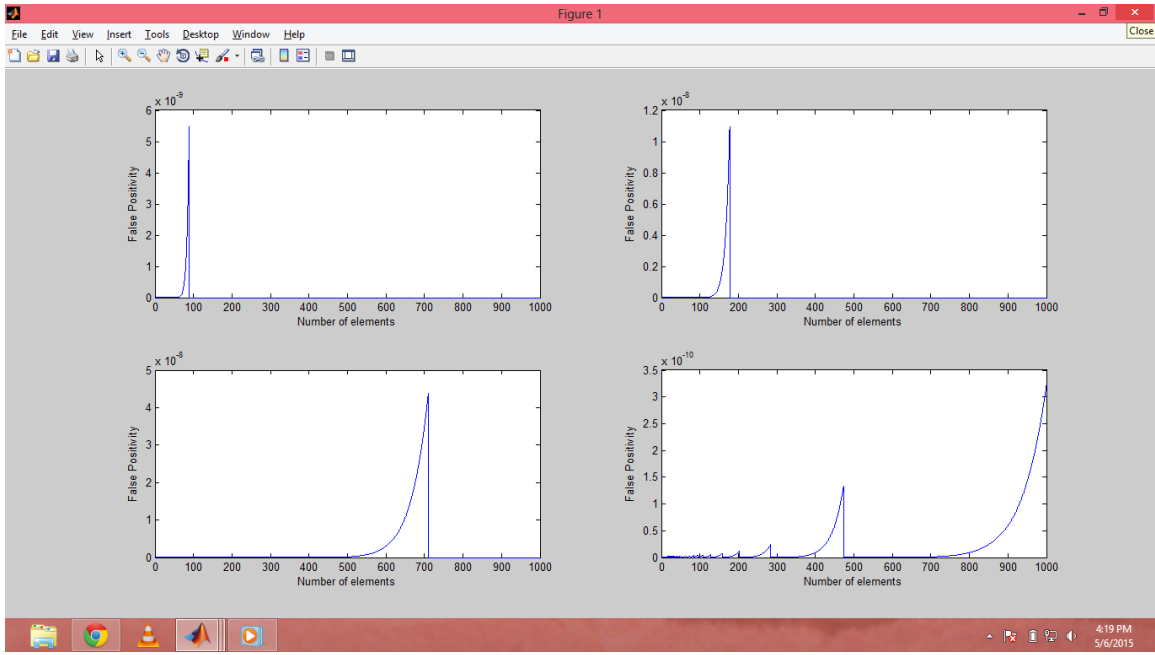


Figure :14 Matlab Implementation of Counting Bloom Filter

Result: The counting Bloom Filter is implemented for various values of $m = 64, 128, 512, 1024$. Hence, it is clear that as the size of bloom filter increases, the false positivity decreases for large value of n .

Comparison : The above figures show that the Counting Bloom Filter is better than the Basic Bloom Filter as the false positivity has significantly reduced in the counting filter along with the facility of removal of element deletion if necessary conditions are satisfied.

CHAPTER – 6

6.1 Complexity Analysis of Basic Bloom Filter

The complexity analysis of the Basic Bloom Filter include the searching time complexity required to test whether an element is a member of the set or not which in turn helps in finding the false positivity rate.

The working code required for checking whether the element is in the set or not is as follows:

```
private int getHash(int i)
{
    md.reset();
    byte[] bytes = ByteBuffer.allocate(4).putInt(i).array();
    md.update(bytes, 0, bytes.length);
    return Math.abs(new BigInteger(1, md.digest()).intValue()) % (set.length - 1);
}

private int[] getSetArray(Object obj)
{
    int freq ;
    int totalElements=0;
    for(int i =0; i<20; i++){
        totalElements += frequency[i];
    }

    freq =(frequency[(Integer)obj]*100)/totalElements;
    System.out.println("freq : "+freq);

    if(freq >= 50 && totalElements>10){

        keySize = keySize*2;

    }

    int[] tmpset = new int[keySize];
    tmpset[0] = getHash(obj.hashCode());
```

```

    for (int i = 1; i < keySize ; i++){
        tmpset[i] = (getHash(tmpset[i - 1]));
    }

    System.out.print("Keys are : ");

    for(int i =0; i<keySize; i++){
        System.out.print(tmpset[i)+"\t");
    }

    return tmpset;

}
public boolean contains(Object obj)
{
    int[] tmpset = getSetArray(obj);
    for (int i : tmpset)
        if (set[i] != 1)
            return false;
    return true;
}

```

In the above given code, the following functions are used :

- `getHash()` : This function has a constant time complexity since it has no loops in it. Hence the time complexity remains constant.
- `getSetArray()` : This function has for loop running twice to the length of the key size k . Since it has a for loop, the time complexity of this function comes out to be $O(k)$.
- `contains()` : This function gives the result as to whether the element is in the set or not. This function has the other function `getSetArray` with time complexity $O(k)$. Hence the complexity of the overall Basic Bloom filter to check whether an element is a member of the set or not is $O(k)$.

- **CONCLUSION :**

The complexity of adding an element into the array : $O(k)$

The complexity of checking set membership : $O(k)$

6.2 Complexity Analysis of Counting Bloom Filter

The counting bloom filter is an extension of the Basic Bloom Filter. Hence the basic running code for checking whether an element is in the set or not remains the same. The addition made comprises of creation of an extra array as the counter.

As we have already seen that the complexity of the Basic Bloom filter is $O(K)$. The time complexity for checking whether an element is member of the set or not remains unchanged. We directly go to the basic bloom filter to check the set membership.

The counting bloom filter is basically used at the time of element deletions from the set which cannot be achieved using the basic bloom filter.

The code for deletion of an element is as follows :

```
public void delete(Object obj)
{
    int[] tmpset = getSetArray(obj);

    for (int i : tmpset){

        if(countSet[i] == 1){
            set[i] = 0;
            countSet[i] -=1;
        }else
        {
            System.out.println("Element cannot be deleted");
            break;
        }
    }
    size--;
}
```

Since the above code for deletion again contains the function `getSetArray()` which has the time complexity $O(k)$. The other for loop runs till the length of `tempest` which again is of size k . Hence the overall complexity for deletion comes out to be $O(k)$ again.

- **Conclusion :**

The time complexity for adding an element : $O(k)$

The time complexity for testing set membership : $O(k)$

The time complexity for deleting an element : $O(k)$

6.3 Complexity Analysis of Weighted Bloom Filter

The weighted Bloom Filter is designed as an extension of the counting bloom filter and the basic bloom filter. It includes using more hash functions for the frequently occurring elements. The frequency of an element is calculated based on which decision is taken to find how many number of hash functions should be used.

In the function `getSetArray()` given above in the basic bloom filter, the frequency of each element is calculated as and when it is entered. As soon as the element is entered, based on the previously entered items, the frequency is calculated.

The formula used for finding the frequency is given below:

```
int freq ;
int totalElements=0;
for(int i =0; i<20; i++){
    totalElements += frequency[i];
}

freq =(frequency[(Integer)obj]*100)/totalElements;
System.out.println("freq : "+freq);

if(freq >= 50 && totalElements>10){
    keySize = keySize*2;
}
```

In the above code for calculating the frequency of each element, no loops are used . Hence the frequency calculation would not cause the overall complexity of the code to change.

- **Conclusion :**

The time complexity for adding an element : $O(k)$

The time complexity for testing set membership : $O(k)$

The time complexity for deleting an element : $O(k)$

Hence we have seen that the Bloom Filter Advancements reduce the false positivity rate without changing the time complexity throughout. The property of the Basic Bloom filter to have a constant search time is retained throughout the project.

Reference:

1. Bloom Filters Wikipedia
2. Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems”, Communications Surveys & Tutorials, IEEE, Volume:14 , Issue: 1, 2012,Pages 131-155
3. Jehoshua Bruck, Jie Gao, Anxiao (Andrew) Jiangz, “Weighted Bloom Filter”, Information Theory, 2006 IEEE International Symposium ,2006, Pages 187-191
4. Flavio Bonomi¹, Michael Mitzenmacher , Rina Panigrahy, Sushil Singh and George Varghese, “An Improved Construction for Counting Bloom Filters”, ESA'06 Proceedings of the 14th conference on Annual European Symposium - Volume 14,2006, Pages 684-695 .
5. Navendu Jain, Mike Dahlin, Renu Tiwari, “Using Bloom filters to refine web search results”, In Proceedings of the eighth International Workshop on the Web and Databases, 2005.
6. Basic Bloom Filter - <https://www.wikipedia.com/bloomfilter>
7. Counting Bloom Filter - <http://blog.csdn.net/jiaomeng/article/details/1498283>
8. Basic Bloom Filter Architecture - http://en.wikipedia.org/wiki/Bloom_filter#/media/File:Bloom_filter_speed.svg