Experiment No. 6
Name: 9_Amruta Poojary
Data Stream Algorithms: Implement Bloom Filter algorithm using any programming language
Date of Performance: 14/9/2023
Date of Submission: 21/9/2023

Aim: Data Stream Algorithms: Apply Bloom Filter algorithm using any programming language

Theory:

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it's not.

Working of Bloom Filter

A empty bloom filter is a bit array of m bits, all set to zero

We need k number of hash functions to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions.

Now if we want to check whether a string is present in filter or not. We'll do the same process but this time in reverse order. We calculate respective hashes using h1, h2 and h3 and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that input string is probably present. If any of the bit at these indices are 0 then the given string is definitely not present.

Code:

```
import java.util.*;
import java.security.*;
import java.math.*;
import java.nio.*;
/* Class BloomFilter */
class BloomFilter
{
   private byte[] set;
   private int keySize, setSize, size;
   private MessageDigest md;

   /* Constructor */
   public BloomFilter(int capacity, int k)
   {
      setSize = capacity;
      set = new byte[setSize];
      keySize = k;
      size = 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];
      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[] tmpset = getSetArray(obj);
        for (int i : tmpset)
            set[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[] tmpset = new int[keySize];
        tmpset[0] = getHash(obj.hashCode());
        for (int i = 1; i < keySize; i++)
            tmpset[i] = (getHash(tmpset[i - 1]));
        return tmpset;
    }
}
/* Class BloomFilterTest */
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");
            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;
            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');
    }
}
```

Output:

Bloom Filter Test

Enter set capacity and key size

1000 1000

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. size

1

Enter integer element to insert 57

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. size

1

Enter integer element to insert 97

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. Size

1

Enter integer element to insert 91

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. Size

1

Enter integer element to insert 23

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. size

1

Enter integer element to insert 67

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. Size

2

Enter integer element to search 67

Search result : true

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. size

2

Enter integer element to search 25

Search result : false

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

BloomFilter Operations

1. Insert
2. Contains
3. check empty
4. Clear
5. size

2

Enter integer element to search 33

Search result : false

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

BloomFilter Operations

1. Insert
2. Contains

3. check empty

4. Clear

5. size

2

Enter integer element to search 97

Search result : true

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

BloomFilter Operations

1. Insert

2. Contains

3. check empty

4. Clear

5. size

5

Size = 5

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

BloomFilter Operations

1. Insert

2. Contains

3. check empty

4. Clear

5. Size

4

Bloom set Cleared

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

BloomFilter Operations

1. Insert

2. Contains

3. check empty

4. Clear

5. size

3

Empty status = true

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

Conclusion:

A Bloom filter is a data structure designed to tell rapidly and memory- efficiently whether an element is present in a set. The tradeoff is that it is probabilistic; it can result in False positives. Nevertheless, it can definitely tell if an element is not present. Bloom filters are space-efficient; they take up O(1) space, regardless of the number of items inserted as the bloom filter size remains the same. However, their accuracy decreases as more elements are added. Insert, and lookup operations are O(1) time.