Bloom Filter

Sergios Anestis Kefalidis
Konstantinos Nikoletos
Kostas Plas &
Manolis Koubarakis

What is an Bloom Filter?

A Bloom FIlter is a space efficient **probabilistic** data structure that helps in search operations in a set of elements.

Bloom Filters are space efficient because they utilize a bit array to represent the data instead of saving them fully.

Bloom Filters are probabilistic because they can show us that element **definitely does NOT exists** or that it maybe exists in the structure.

How does a Bloom Filter function?

Bloom Filters are very similar to hash tables in nature. They use an array and hash functions to store elements into that array.

However, unlike hash tables, bloom filters do not store the data into the array. The only store 1 bit for each entry, 0 or 1, to indicate if the entry is in the array or not. Thus, bloom filters utilize a bit array for their data.

The Bit Array (1)

A bit array is simply a collection of bits. They are very similar to other types of arrays like integer, float arrays etc. However, since in c we cannot declare a variable to be of type bit, we usually represent bit arrays using **unsigned char** arrays.

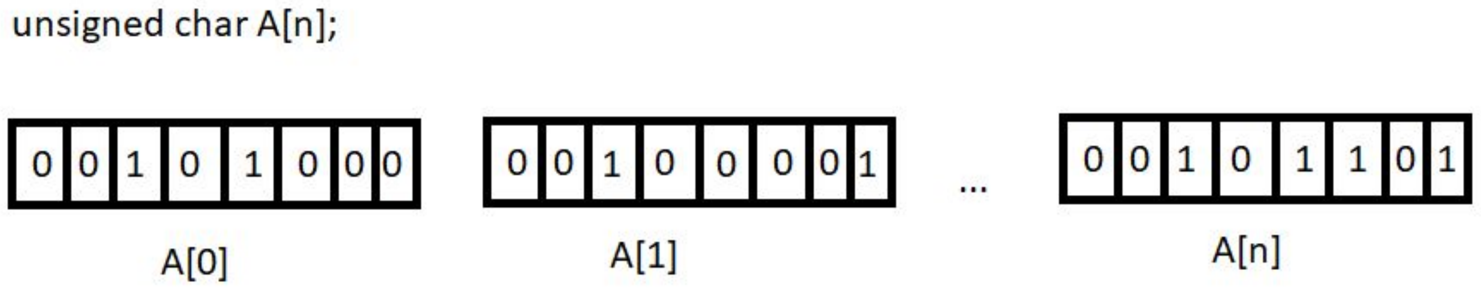
For example if we want a bit array of 200 spaces, we can declare:

```
unsigned char bitArray[25];
```

Why 25 spaces; Because char type variables consist of 1 byte, so 25*8= 200 bits!

The Bit Array (2)

A example of a bit array using an unsigned char array for its implementation:



Inserting data into a Bloom Filter (1)

To insert an element into a bloom filter, we first pass through **k** hash functions. We will discuss why k hash functions are used instead of 1. The k values are then store into the bit array, by changing the bit from 0 to 1, in the position the hash value indicates. For example, let’s imagine we want to insert the word “apples” into the filter.

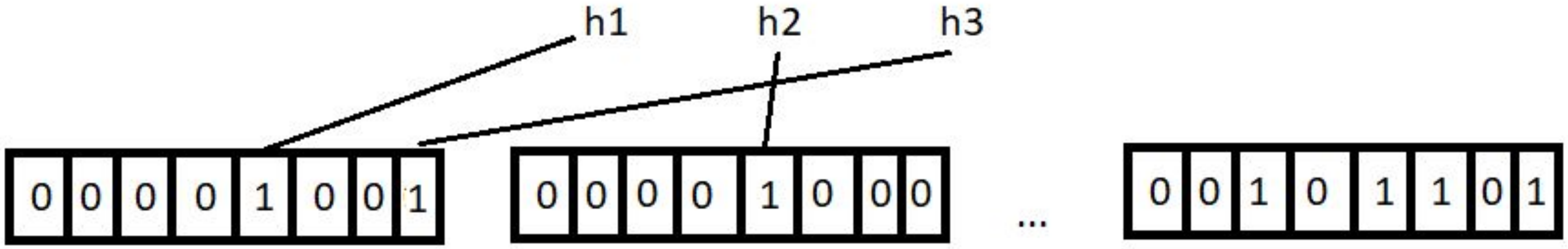
h1(“apples”) = 4
h2(“apples”) = 12
h3(“apples”) = 7

We must now change the bits 4, 7 & 12 into the bit array.

Inserting data into a Bloom Filter (2)

To get the correct indexes and bit positions we divide by 8 and get the modulo of 8 respectively (why?)

h1(“apples”) = 4, index = 4/8 = 0, pos = 4%8 = 4
h2(“apples”) = 12, index = 12/8 = 1, pos = 12%8 = 4
h3(“apples”) = 7, index = 7/8 = 0, pos 7%8 = 7



Checking if elements exist in Bloom Filters (1)

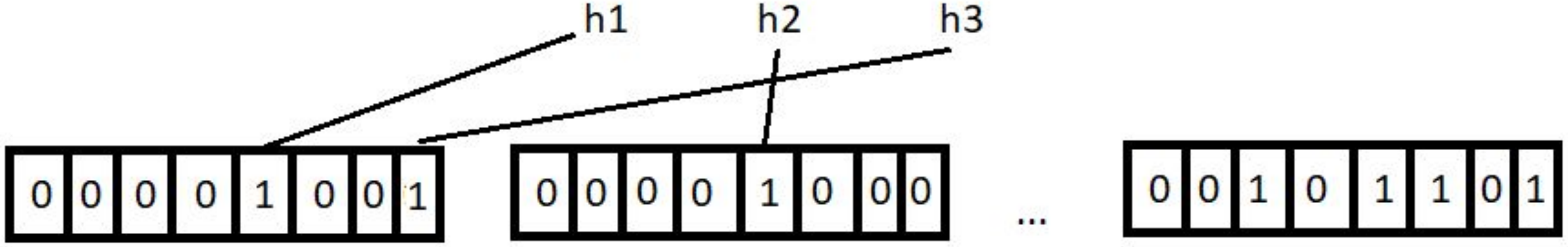
In the bloom filter structure, it is impossible to know for sure if an element actually exists in the array. If the position bit is 1, the element might truly exist, or we might find a **false positive**. The true strength of bloom filters lie in the fact, that when an element is not present in the set, we **definitely** know that it is **not** there.

To check if an element exists in the filter or not, we follow the same operation with insert. We first pass the key through our k hash functions and then check the bits in the positions of the array (division and modulo by 8).

If at least one of the positions provided by the k hash functions contains a 0 bit, then the element does not exist. If every k position bit is 1, then the element might exist.

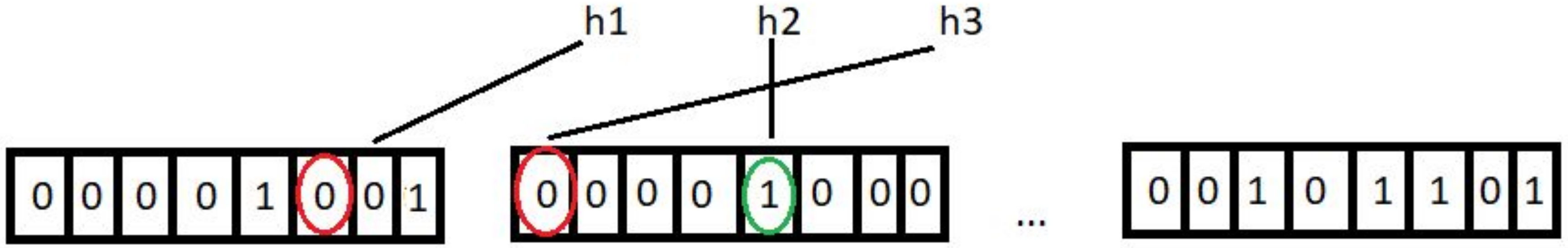
Checking if elements exist in Bloom Filters (2)

h1(“apples”) = 4 [index:0,pos:4]
h2(“apples”) = 12 [index:1,pos:4]
h3(“apples”) = 7 [index:0,pos:7]



Checking if elements exist in Bloom Filters (3)

h1(“cherries”) = 6 [index:0,pos:6]
h2(“cherries”) = 12 [index:1,pos:4]
h3(“cherries”) = 8 [index:1,pos:0]



Collisions in Bloom Filters

Collisions is a problem that bloom filters, like any other structure using hashes, faces. Collisions, are the main reason for **false positive** values.

To reduce the number of false positives, on the other hand, we use the k hash functions, so the possibility of getting a false positive is much smaller.

Bitwise operations

A reminded on bit arithmetics to help you implement a bloom filter:

To **change** a certain bit p in a given index of the array:

```
mask = 1 left_shift p

bitArray[i] = bitArray[i] OR mask
```

To **check** a certain bit p in a given index of the array:

```
mask = 1 left_shift p

value = bitArray[i] AND mask
```