# Bloom Filters Introduction and Implementation

*GeeksforGeeks*

Suppose you are creating an account on Geekbook, you want to enter a cool username, you entered it and got a message, "Username is already taken". You added your birth date along username, still no luck. Now you have added your university roll number also, still got "Username is already taken". It's really frustrating, isn't it?
But have you ever thought about how quickly Geekbook checks availability of username by searching millions of username registered with it. There are many ways to do this job –

- **Linear search** : Bad idea!
- **Binary Search** : Store all username alphabetically and compare entered username with middle one in list, If it matched, then username is taken otherwise figure out, whether entered username will come before or after middle one and if it will come after, neglect all the usernames before middle one(inclusive). Now search after middle one and repeat this process until you got a match or search end with no match. This technique is better and promising but still it requires multiple steps.
  But, there must be something better!!

**Bloom Filter** is a data structure that can do this job. It is mainly a spaced optimized version of hashing where we may have false positives. The idea is to not store the actual key rather store only hash values. It is mainly a probabilistic and space optimized hashing where less than 10 bits per key are required for a 1% false positive probability and is not dependent on the size of individual keys.

### What is Bloom Filter?

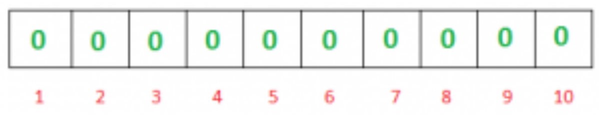
A Bloom filter is a **space-efficient probabilistic** data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it 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.

### Interesting Properties of Bloom Filters

- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate **false negative** result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

### Working of Bloom Filter

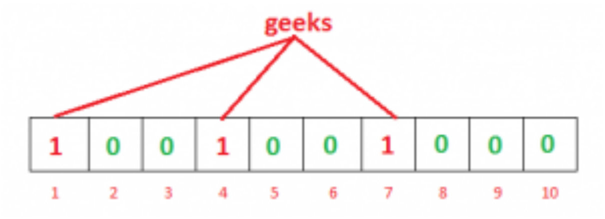
A empty bloom filter is a **bit array** of **m** bits, all set to zero, like this –



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.
Example – Suppose we want to enter "geeks" in the filter, we are using 3 hash functions and a bit array of length 10, all set to 0 initially. First we'll calculate the hashes as follows:

```
h1("geeks") % 10 = 1
h2("geeks") % 10 = 4
h3("geeks") % 10 = 7
```

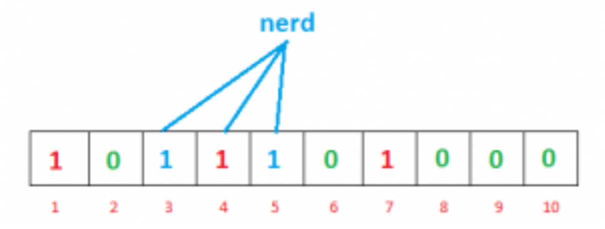
**Note:** These outputs are random for explanation only.
Now we will set the bits at indices 1, 4 and 7 to 1



Again we want to enter "nerd", similarly, we'll calculate hashes

```
h1("nerd") % 10 = 3
h2("nerd") % 10 = 5
h3("nerd") % 10 = 4
```

Set the bits at indices 3, 5 and 4 to 1



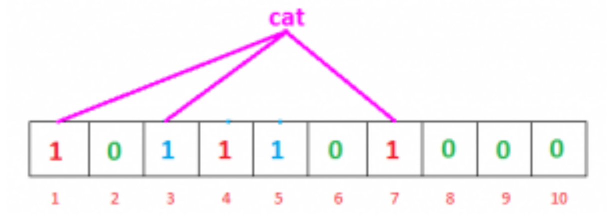
Now if we want to check "geeks" is present in filter or not. We'll do the same process but this time in reverse order. We calculate respective hashes using h1, h2 and h3 and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that "geeks" is **probably present**.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

## False Positive in Bloom Filters

The question is why we said **"probably present"**, why this uncertainty. Let's understand this with an example. Suppose we want to check whether "cat" is present or not. We'll calculate hashes using h1, h2 and h3

```
h1("cat") % 10 = 1
h2("cat") % 10 = 3
h3("cat") % 10 = 7
```

If we check the bit array, bits at these indices are set to 1 but we know that "cat" was never added to the filter. Bit at index 1 and 7 was set when we added "geeks" and bit 3 was set we added "nerd".



So, because bits at calculated indices are already set by some other item, bloom filter erroneously claims that “cat” is present and generating a false positive result. Depending on the application, it could be huge downside or relatively okay.
We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want to decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition to the item and checking membership.

**Operations that a Bloom Filter supports**

- insert(x) : To insert an element in the Bloom Filter.
- lookup(x) : to check whether an element is already present in Bloom Filter with a positive false probability.

NOTE : We cannot delete an element in Bloom Filter.
**Probability of False positivity:** Let **m** be the size of bit array, k be the number of hash functions and **n** be the number of expected elements to be inserted in the filter, then the probability of false positive **p** can be calculated as:

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

**Size of Bit Array:** If expected number of elements **n** is known and desired false positive probability is **p** then the size of bit array **m** can be calculated as :

$$m = -\frac{n \ln P}{(ln2)^2}$$

**Optimum number of hash functions:** The number of hash functions **k** must be a positive integer. If **m** is size of bit array and **n** is number of elements to be inserted, then k can be calculated as :

$$k = \frac{m}{n} ln2$$

**Space Efficiency**

If we want to store large list of items in a set for purpose of set membership, we can store it in hashmap, tries or simple array or linked list. All these methods require storing item itself, which is not very memory efficient. For example, if we want to store “geeks” in hashmap we have to store actual string “ geeks” as a key value pair {some_key : ”geeks”}.
Bloom filters do not store the data item at all. As we have seen they use bit array which allow hash collision. Without hash collision, it would not be compact.

**Choice of Hash Function**

The hash function used in bloom filters should be independent and uniformly distributed. They should be fast as possible. Fast simple non cryptographic hashes which are independent enough include murmur, FNV series of