Bloom Filters for System Design Interviews

PREV | **HOME** | **NEXT**

Please consider supporting us by signing up for a paid Medium account or just buy us a coffee :)

When designing applications, programmers often face the use case where they need to check whether an element is present in a set. For example, when creating a new email account, you need to type in an email address. The system will tell you whether the email address is already taken or not. If already taken, you will test a different one until you find one that's available.

At the backend, the system is checking your email address against millions of existing email addresses to check if there's a match. Yet, the system answers you in a fraction of a second. Conventional linear search of an index could not give a result that fast. A hash map could do the job but it will take a large amount of memory space.

Bloom filter is the best solution to the above use case. It's ideal for applications in which:

- It needs to be quickly checked whether or not an item is in a list.
- The list is large and the memory space is limited.

What Are Bloom Filters?

Bloom filter is a probabilistic data structure, designed by Burton Howard Bloom in 1970 to check whether an element is a member of a set. 'Probabilistic' because the data structure tells whether an element is NOT in the set, or MAY BE present.

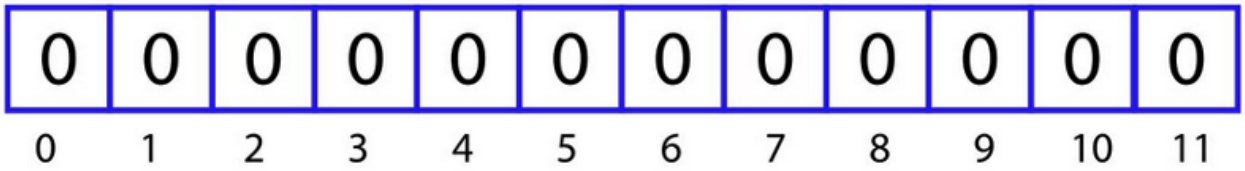
So, the fast lookups offered by bloom filters have a catch — false positives. A false positive is the case in which an element is not present in the set, but the system tells you that it is

present. The probability of false positives is typically low.

How Does A Bloom Filter Work?

A bloom filter is a bit vector of *m* bits, initially all set to 0.

As an example, below is a 12-bit bloom filter. All the bits are 0 initially. The number below the bits represent the index of that bit. The index starts from 0 to *m-1* (in this case, 11).



To add items to the bloom filter, we need *k* number of hash functions. Each element that you want to add to the bloom filter is passed through the *k* hash functions to obtain k fixed-sized hash values.

Next, we take the mod of *m* for each of these hash values (*hash value % 12*in our case) to obtain indexes less than or equal to m-1. The bits present at the indexes obtained through the operation are set to 1 in the bit vector. Continue adding each element to the bloom filter using the same method.

Other than adding elements to the filter, we can also check their presence in the filter. To check if an element is present, we perform hash and mod on the element using the same procedure as described above. Once you have the indexes, check values of the bits present at these indexes to deduce whether the element is present or not. You'll understand the concept better with the example below.

Note that you can only add elements to the bloom filter or check their membership. Once added, elements cannot be deleted from the filter.

Example

Let's suppose we have a 12-bit bloom filter and 3 hash functions, h1(x), h2(x), h3(x). First we will add elements to the bloom filter. Next, we will check if an element is present in the filter or not.

Adding Elements To The Filter

We want to add the string “white” to the empty bloom filter.

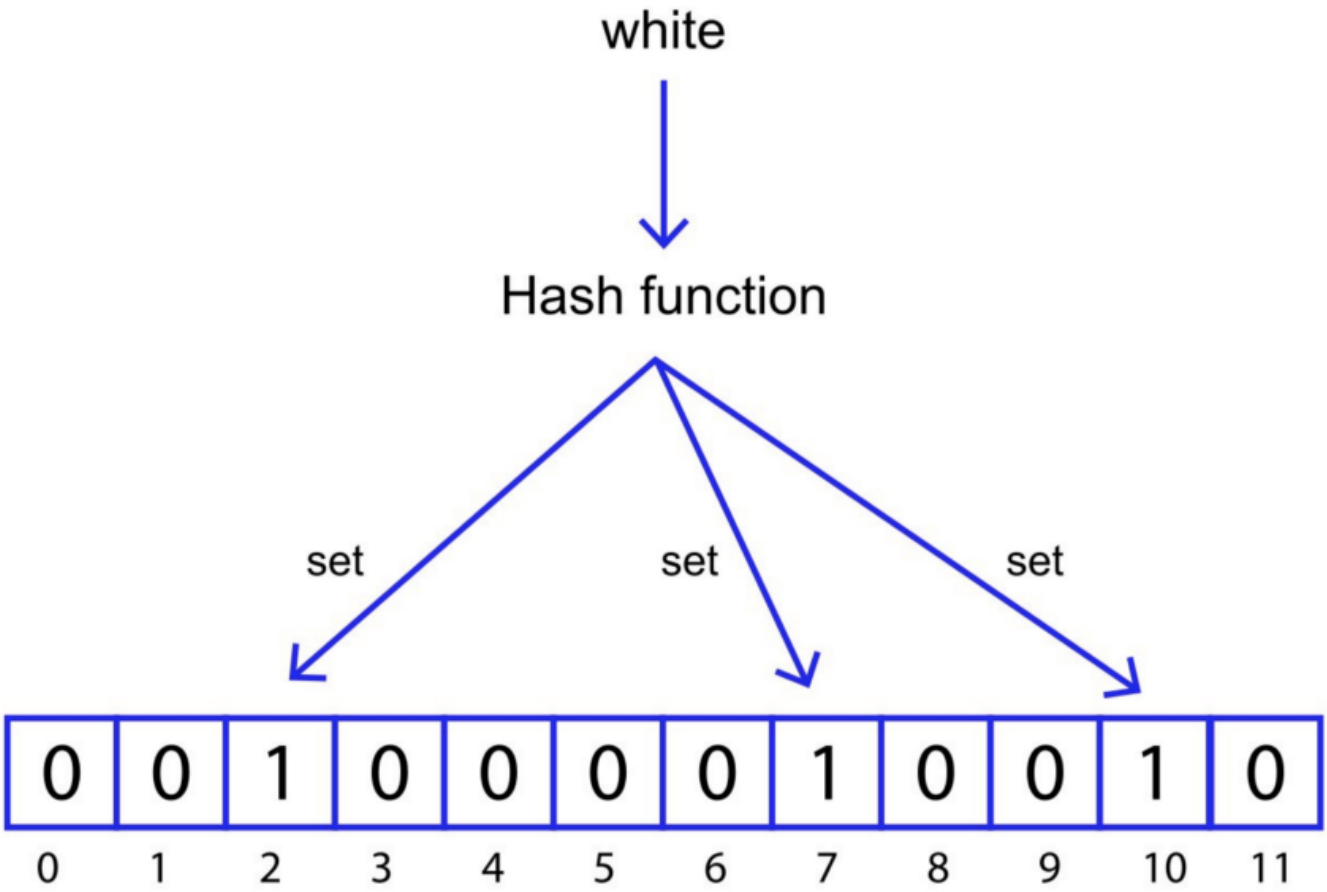
Feeding it to the three hash functions in our example and taking the mod of 12 for the result, suppose we have:

$$h_1(\text{"white"}) \% 12 = 2$$

$$h_1(\text{"white"}) \% 12 = 10$$

$$h_1(\text{"white"}) \% 12 = 7$$

Let’s set the bits at the indexes 2, 10 and 7 to 1. The bloom filter looks like this now:



Next, let's add another element, "blue" to the bloom filter.

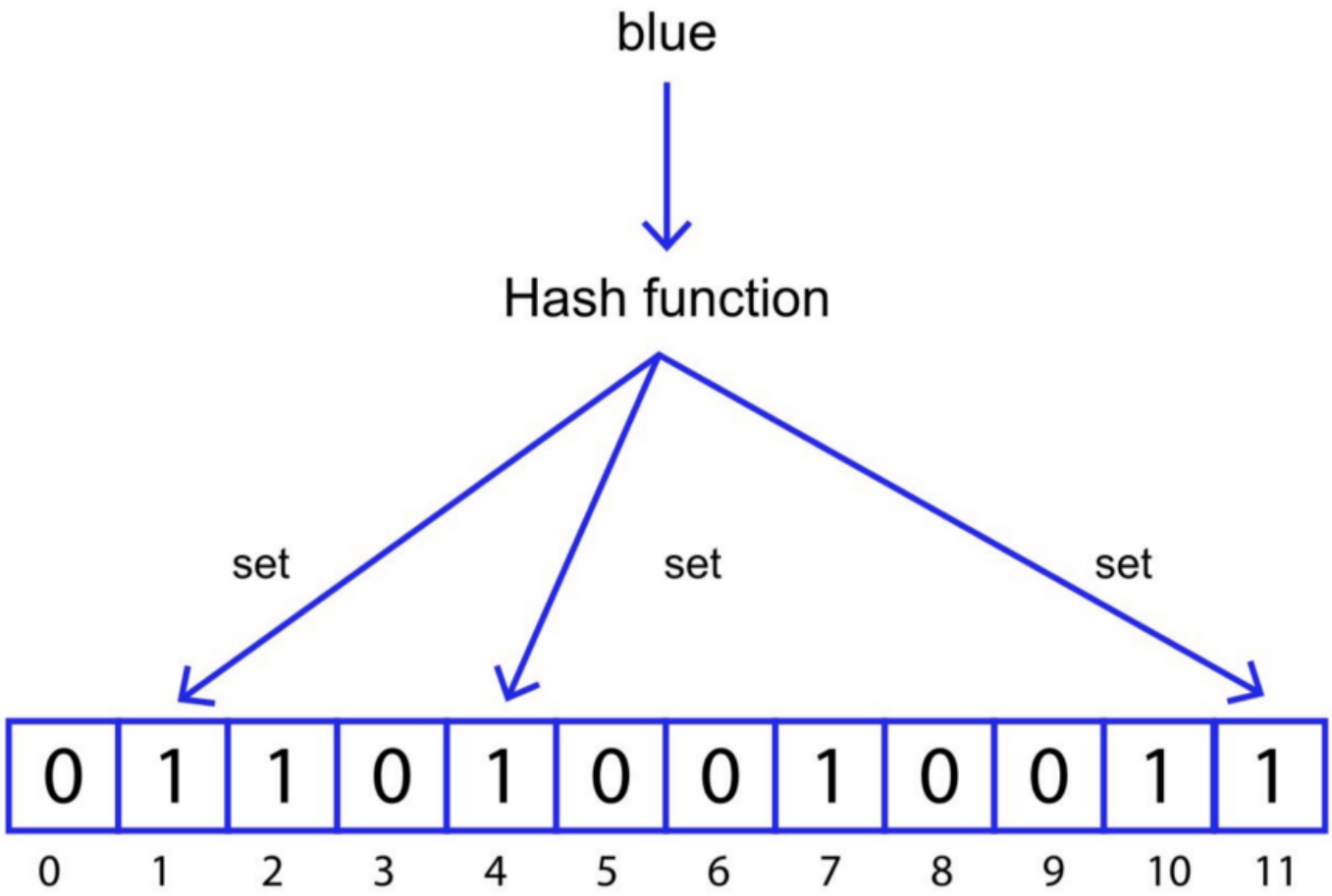
Feeding the string to the three hash functions and taking the mod, we get 3 further indexes to set:

$$h_1(\text{"blue"}) \% 12 = 4$$

$$h_1(\text{"blue"}) \% 12 = 1$$

$$h_1(\text{"blue"}) \% 12 = 11$$

The bits at indexes 4, 1, and 11 will also be set to 1. Now the bloom filter looks like this:



Checking If An Element Is In The Filter

Now our bloom filter has some elements (“white” and “blue” for this example). Let’s check if the element “purple” is present in the set.

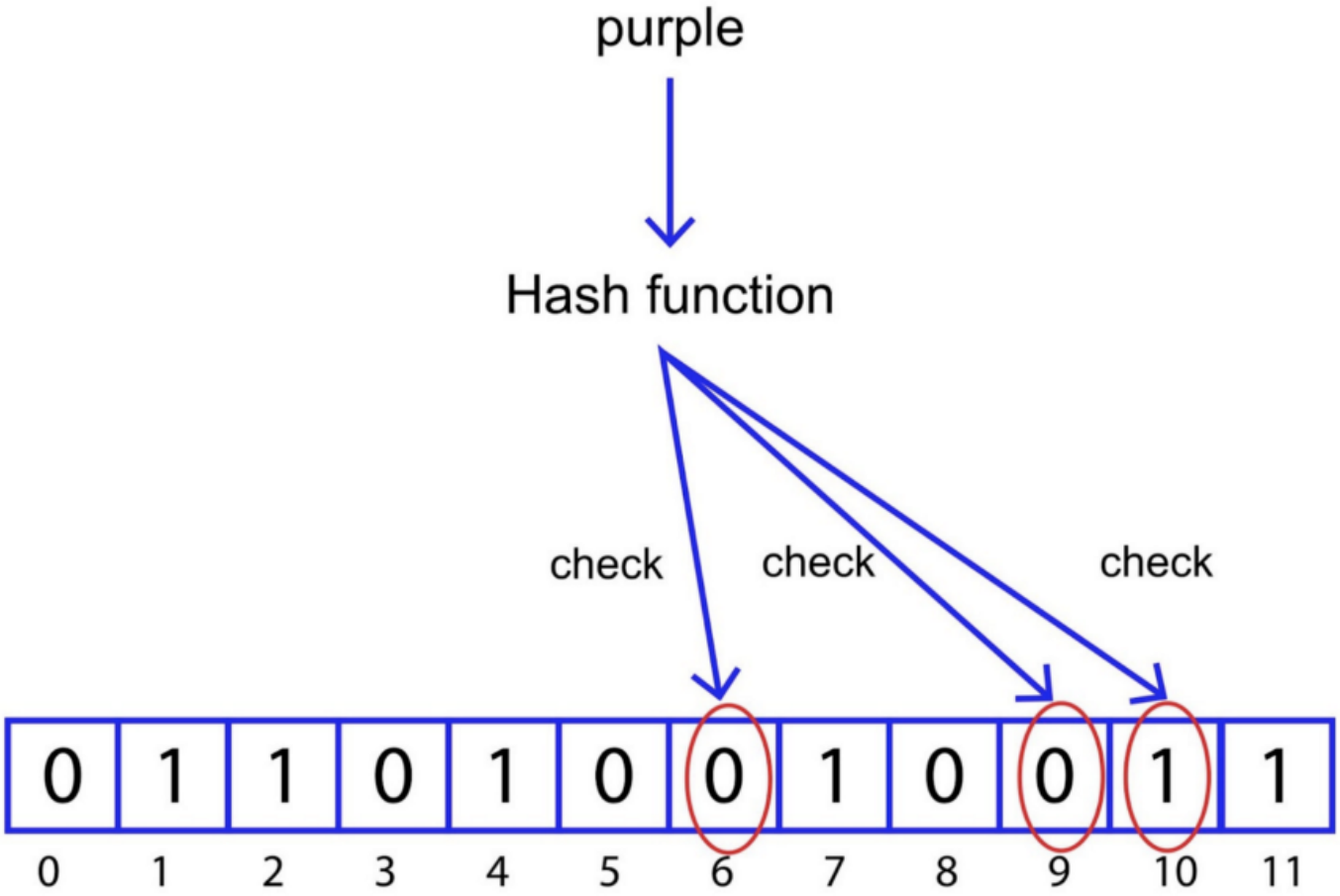
We’ll perform the same operation on “purple”, finding its hashes and taking the mod:

$$h_1("purple") \% 12 = 9$$

$$h_1("purple") \% 12 = 6$$

$$h_1("purple") \% 12 = 10$$

Check the value of the bits at the indexes calculated above. If the bits at all three indexes are 1, we can say that “purple” **may be** present in the filter. If at least one of the bits at these indexes are 0, we can say that “purple” is **not** present in the filter.



Since the bits at indexes 6 and 9 are 0 in the diagram above, we know that “purple” is not in the filter.

Let’s check if “blue” is in the filter.

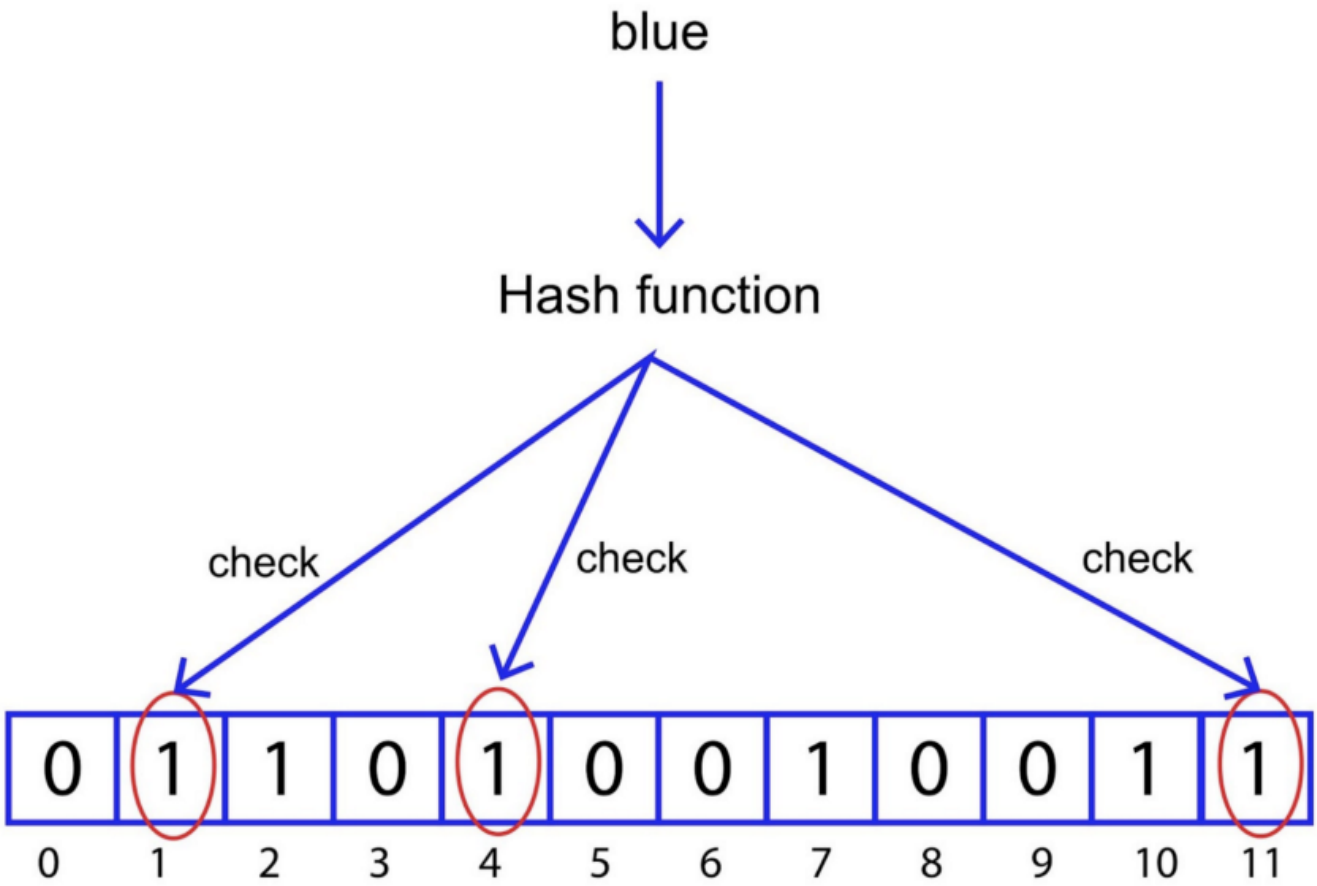
Perform hash functions and mod on “blue” to obtain the indexes to check,

$$h_1(\text{"blue"}) \% 12 = 4$$

$$h_1(\text{"blue"}) \% 12 = 1$$

$$h_1(\text{"blue"}) \% 12 = 11$$

Next, let’s check the values of the bits at the above indexes,



The bits at all three positions are set, telling you that the element “blue” may be present in the filter.

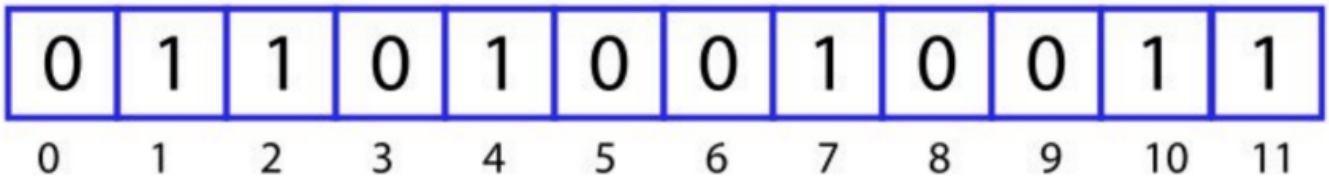
Please consider supporting us by signing up for a paid Medium account or just buy us a coffee :)

How Does The Bloom Filter Give False Positives?

We mentioned earlier that bloom filters sometimes give false positive results. This is why we can only say that an element “may be” present in the set if the bloom filter gives a positive result upon checking for an element’s presence. Why so? Why isn’t a positive result 100% accurate?

Let’s demonstrate with an example.

For the same example above, where we have the bloom filter,



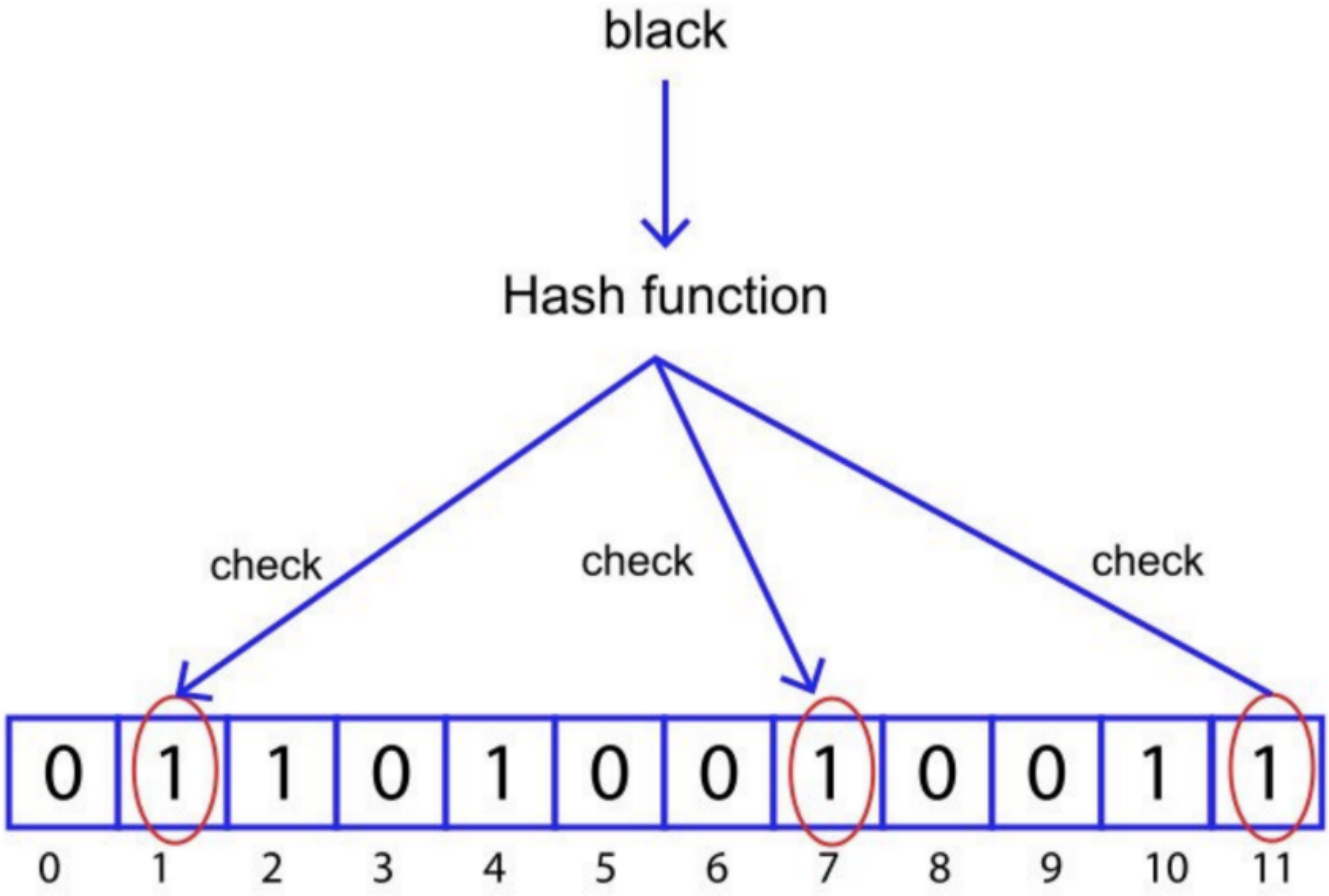
Let's check if "black" is present in the filter. Performing hash and mod on "black", we have the following indexes to check,

$$h_1("black") \% 12 = 11$$

$$h_1("black") \% 12 = 7$$

$$h_1("black") \% 12 = 1$$

Next, let's check what we have in the bloom filter at the indexes 11, 7 and 1.



Incidentally, the bits at all the three indexes are set. So the bloom filter tells us “black” **may be** present in the set.

However, since we added only “white” and “blue” to the filter, we know that “black” was never there in the first place! So the bloom filter gave a “false positive” in this case.

What happened was that the bit at index 7 was set when “white” was added to the filter, and the bits at 1 and 11 were set when “blue” was added to the filter. Now when the algorithm sees that bits at 11, 7 and 1 are set, it suggests that “black” could be in the filter.

Reducing The Probability Of False Positives

If your application demands a lower probability of false positives, there are some ways in which you can control it. Increasing the size of the bit array and the number of hash functions can improve the efficiency of the results and lower the probability of a false positive.

However, increasing the number of hash functions also adds latency to the insertion and lookup operations of the bloom filter. The time complexity for a bloom filter is O(k) where k

is the number of hash functions involved.

Applications Of Bloom Filters

As a space-efficient data structure that can rapidly check the membership of elements, bloom filter has numerous applications. Here are some examples:

- Google Chrome uses bloom filters to spot malicious URLs. The URL entered by the user is looked up in a bloom filter. If it returns a positive result, the URL goes through some further tests before the user is warned.
- CDNs use bloom filters to prevent caching “one-hit-wonders” (elements requested by users only once), saving disk space.
- Medium uses it to detect articles already read by the user, preventing it from appearing in the recommended posts.
- Databases like Apache Cassandra, Google BigTable and Apache HBase optimize performance of query operations by lowering the number of searches for non-existent rows and columns. Again, bloom filters help achieve this.

Conclusion

By now, you’ll hopefully have a better understanding of what a simple bloom filter is, how it works and some ideas on how it can be implemented to real life use cases. There can be variations to the basic design, depending on what the application calls for. Counting bloom filters, for instance, can be implemented in applications where deletion of elements is also a requirement.