MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

Project Report
Topic : Bloom Filter

Subject : ADSA Lab CSE 5141

Submitted By :
Uday Yadav : 230913008
Blen Joswin : 230913009

Under the guidance of :
Prof. Gururaj
Department of Computer Science and Engineering

Bloom Filters

1. Introduction

A Bloom filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set or not. It utilizes a bit array and a set of hash functions to represent the presence or absence of elements. When an element is inserted, its hash values are used to set corresponding bits in the array. To check for membership, the filter hashes the query element and checks if all corresponding bits are set; if any bit is not set, the element is not in the set. However, false positives are possible, but false negatives are not. Bloom filters are commonly used in applications like data deduplication, spell checking, and network routing for quickly eliminating irrelevant data. They provide a trade-off between memory usage and accuracy in set membership tests.

2. What is a Probabilistic Data Structure (PDS)?

It is a type of data structure that uses probabilistic algorithms and techniques to provide approximate answers or solutions to various problems with limited memory or computational resources. Instead of providing exact results, these data structures return estimates or approximations with controlled error rates. This results in speed up of queries that would otherwise take a full search. Here are some examples :

- *Web Caching*: In web caching systems, PDS can be used to check whether a requested web page is in the cache. If the page is not in the cache, there's no need to retrieve it from the server, which saves time and bandwidth.
- *Duplicate Elimination*: In distributed systems or databases, PDS can be employed to eliminate duplicate records. They can quickly identify whether a record is likely a duplicate, reducing the overhead of checking the full dataset.
- *Network Firewall Rules*: In network security, PDS can be used to check if a specific IP address or URL should be allowed or blocked. This can speed up packet filtering by quickly eliminating traffic that does not match known rules.
- *Uniqueness Checkers*: PDS can be used in spell checkers to quickly determine whether a word is valid or exists without consulting a larger dataset.

3. How does a bloom filter work ?

The bloom filter data structure is a bit array of length n as shown in Figure below. The position of the buckets is indicated by the index (0–9) for a bit array of length ten. All the bits in the bloom filter are set to zero when the bloom filter is initialized (an empty bloom filter). The bloom filter discards the value of the items but stores only a set of bits identified by the execution of hash functions on the item.

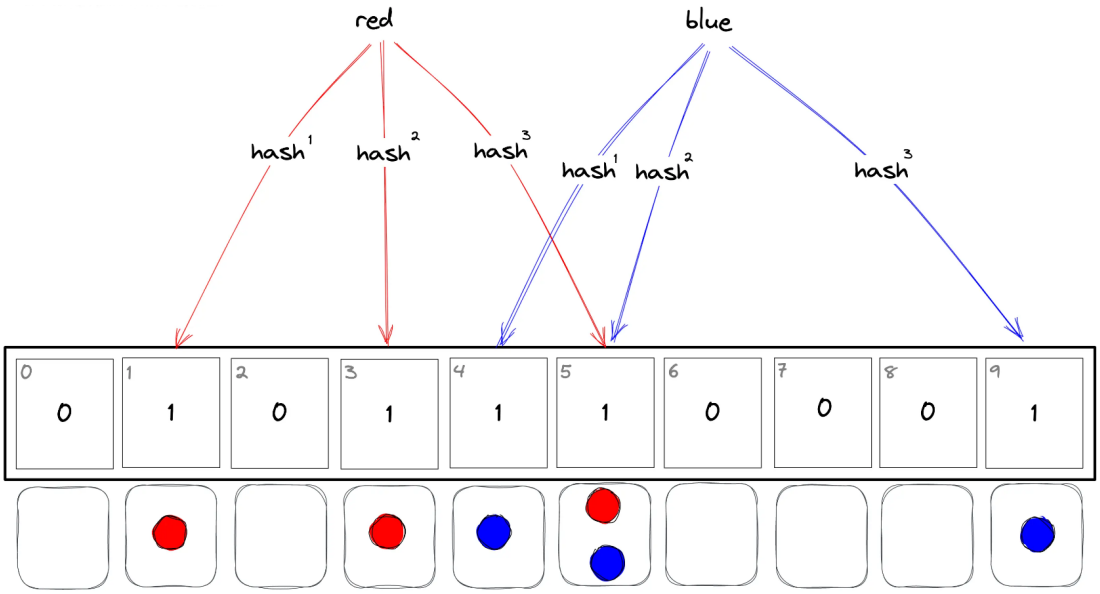
3.1 Adding Items to Bloom Filter

In order to add data to the bloom filter, we require k hash functions and n-bit arrays, k can be small, and n needs to be reasonably big considering the amount of data expected to be processed. We need good hash functions to avoid collisions.

1. The item is processed through k hash functions
2. Each hash function with produce different value, mark the corresponding bit to 1
3. Do this for all hash functions

There is a probability that some bits on the array are set to one multiple times due to hash collisions.

Let's say we have 2 words : red and blue and 3 hash functions : hash1, hash2, hash3



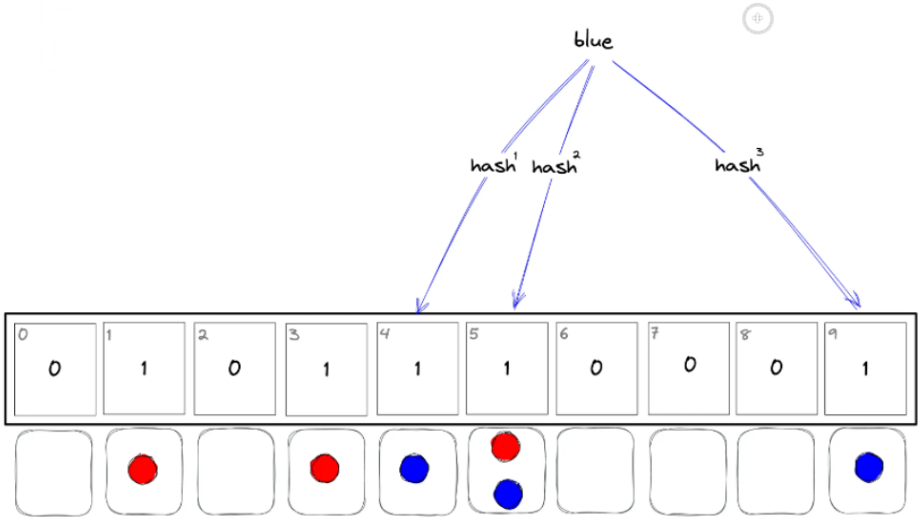
- h1(red) mod 7 = 1
- h2(red) mod 11 = 3
- h3(red) mod 17 = 5
- h1(blue) mod 7 = 4
- h2(blue) mod 11 = 5
- h3(blue) mod 17 = 9

Item red is hashed using 3 different functions and the resultant bit is set to one. It is important to note that the hash function should be different and avoid collision even for the same datapoint. When the item blue is a hash, one of its values collides with the red item, this is an acceptable scenario as the other 2 results are different. The bit array at index 5 stores both item red and blue

3.2 Check whether Item exists or not in Bloom Filter

In Order to check whether an item exists in bloom filter or not, we perform the steps of adding item to bloom filter but instead of marking the corresponding bit as 1, we verify whether it is already marked or not :

- The item is processed through k hash functions
- Each hash function will produce a different value, note the result of marking.
- Do this for all hash functions



An item is not included in the bloom filter if any of the specified bits are set to zero. The item may be a member of the bloom filter if all the bits are set to one. Because of hash function collisions or the potential for some bits to be changed to one by other things, there is uncertainty regarding an item's membership.

To determine if item blue is a member, a query is sent to the bloom filter. The modulo operator is used to the generated hash value to identify the buckets that need to be examined.

- h1(blue) mod 7 = 4
- h2(blue) mod 11 = 5
- h3(blue) mod 17 = 9

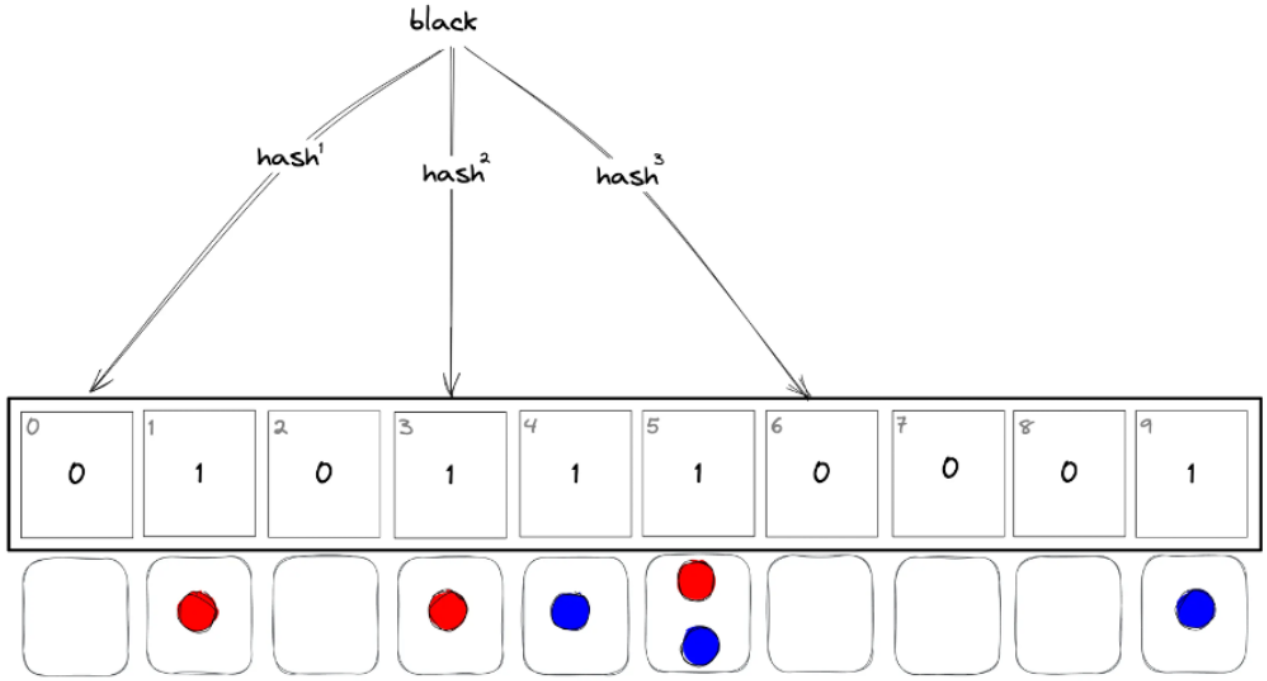
Given that all of the bits are set to one, the item blue may be a part of the bloom filter.

3.2.1 Checking for false positive

In order to determine if item black, which is not a member of the bloom filter, is included, the bloom filter is using the same steps as done previously.

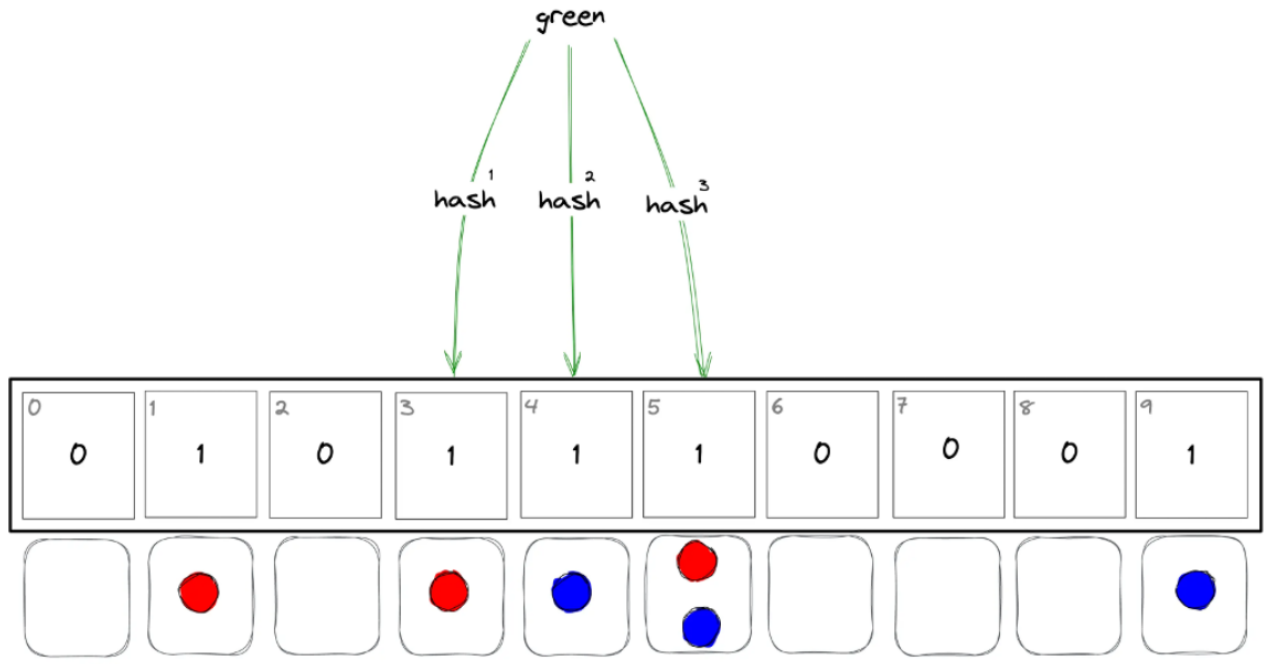
- h1(black) mod 7 = 0
- h2(black) mod 11 = 3
- h3(black) mod 17 = 6

Because the bit at position zero is set to zero, the item black is not a part of the bloom filter. It is not necessary to verify the remaining bits.



4. False Positive and False Negatives in Bloom Filters

4.1 False positive in Bloom Filter



Let's check whether the item green exists in the bloom filter or not.

- h1(*green*) mod 7 = 3
- h2(*green*) mod 11 = 4
- h3(*green*) mod 13 = 5

The bloom filter will say yes although the item green is not a member of the bloom filter as the bits were set to one by items blue and red.

In a Bloom filter, false positives can occur, while false negatives do not. This behavior is a fundamental characteristic of Bloom filters and is a consequence of their design. Here's an explanation of why false positives can occur in Bloom filters while false negatives are not possible:

False Positives

- When you insert an element into a Bloom filter, it is hashed to multiple positions (typically using multiple hash functions), and these positions are set in the filter.
- When you want to check whether an element is in the filter, you hash the element in the same way and check whether all the corresponding positions in the filter are set. If they are, the filter indicates that the element is "possibly" in the set.
- False positives occur when multiple elements happen to hash to the same positions, setting bits in the filter. This makes the filter think that an element is in the set when it is not. Since different elements can produce the same hash positions, false positives can happen.

False Negatives

- In a Bloom filter, once a bit is set to 1 (indicating that an element is in the set), it cannot be reset to 0. This is because other elements might have set that bit, and resetting it could lead to incorrect results for those elements.
- As a result, if an element has never been inserted into the Bloom filter, there's no way for the filter to mistakenly indicate that it's in the set. This means that false negatives, where the filter incorrectly claims that an element is not in the set when it actually is, do not occur.

In Bloom Filter, Deletion and Updation operation are not possible, because they also avoid false negatives. This limitation can be overcome by counting Bloom Filters.

5. Time/Space Complexity Analysis

5.1 Time complexity analysis

We must examine each computing step in order to assess the bloom filter's complexity. Setting and verifying the value in the n-bit array by hashing the value using k distinct hash functions. The hash functions that are utilized determine how well the bloom filter performs. Each bloom filter operation takes less time overall the faster the hash

function is computed. It is possible to think about value checking in an n-bit array as a constant process.

Also the amortized cost of the hash function can be considered as O(1).

- Amortized Addition operation : O(k)
- Amortized cost for checking item : O(k)

K is the number of hash functions. The time complexity of the bloom filter is independent of the number of items already in the bloom filter. The k lookups in the bloom filter are independent and can be parallelized.

Space complexity

By setting aside a little amount of bits for each item, the bloom filter uses a fixed amount of bits regardless of the number of objects it contains. The data pieces are not stored by the bloom filter, resulting in an O(n) space complexity.

6. Advantages of bloom filter

The advantages of the bloom filter are the following:

- K amortized time complexity
- N space complexity
- operations are parallelizable
- no false negatives
- enables privacy by not storing actual items

6.1 Disadvantages of Bloom filter

The limitations of the bloom filter are the following:

- bloom filter doesn’t support the delete operation
- false positives rate can’t be reduced to zero
- bloom filter on disk requires random access due to random indices generated by hash functions

Removing an item from the bloom filter is not supported because there is no possibility to identify the bits that should be cleared. There might be other items that map onto the same bits and clearing any of the bits would introduce the possibility of false negatives.

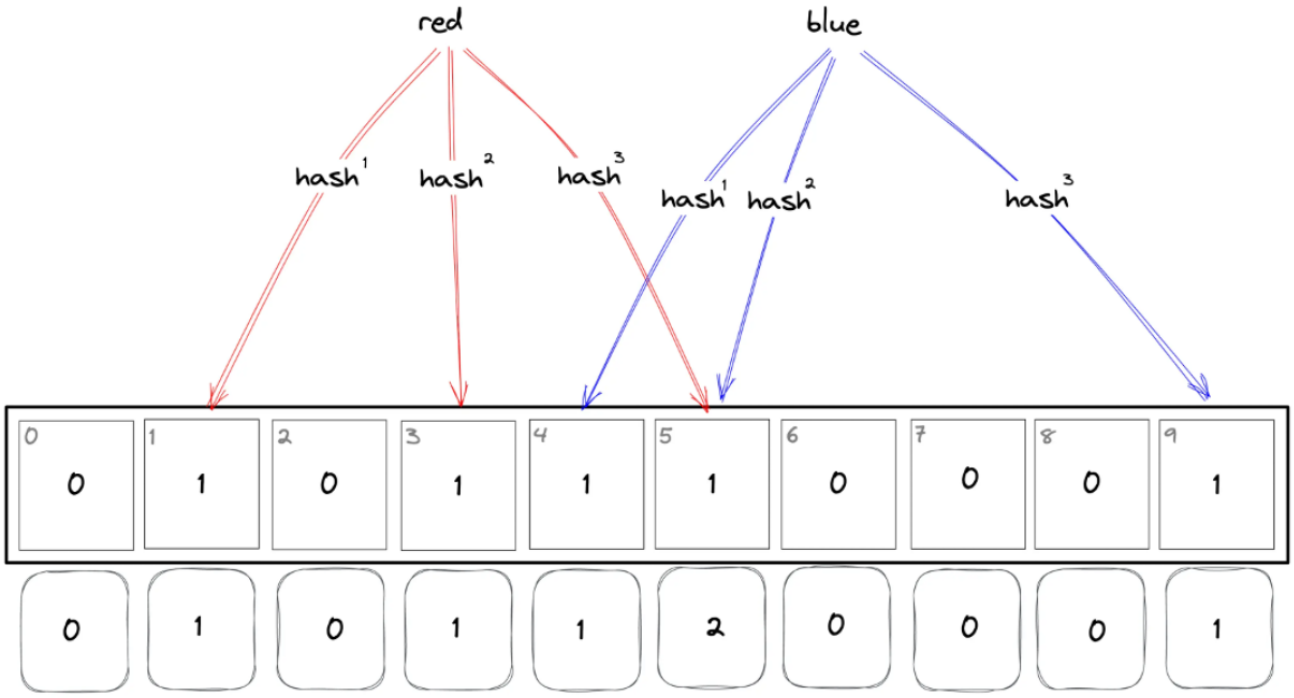
6.2 Bloom filter use cases

The bloom filters are useful in high-ingestion systems to prevent expensive disk seeks. For instance, the server performing a lookup of an item in a large table on the disk can degrade the throughput of the service. The bloom filter in memory can be used to

serve the lookups and reduce the unnecessary disk I/O except when the bloom filter returns a false positive. The applications of the bloom filter are the following:

- reducing disk lookups for the non-existing keys in a database
- determining whether a user ID is already taken
- filtering out previously shown posts on recommendation engines
- checking words for misspellings and profanity with a spellchecker
- identifying malicious URLs, blocked IPs, and fraudulent transactions
- Log-structured merge tree (LSM) storage engine used in databases such as Cassandra uses a bloom filter to check if the key exists in the SSTable
- MapReduce uses the bloom filter for the efficient processing of large datasets

7. Extension of bloom filter : Counting bloom filter



The counting bloom filter includes an array of counters for each bit in the bloom filter array. The counter array is initialized to zeros. The counter for associated bits is incremented when an item is added to the counting bloom filter. The membership query is the same as the classical bloom filter. The **counting bloom filter supports the delete operation**. The following operations are executed to delete an item from the bloom filter:

1. the item is hashed through the same k-hash functions
2. the modulo n (length of bit array) operation is executed on the output of the hash functions to identify the k array positions (buckets)
3. the counter of the associated bits is decreased
4. the corresponding bit is unset to zero if the associated counter is decremented to zero

Naturally, Counting Bloom filters are much bigger than classical Bloom filters because additional memory has to be allocated for the counters even if most of them are zeros. Therefore, it is important to estimate how large such counters can become and how their size depends on the filter's length m and the number of hash functions k.

Proof Bloom filters doesn't have false negative

Let A be not false positive given by one hash function:
=> P(A) = [(n-1)/n]
In bloom filter, we use k different hash functions :
=> P(A) for all k = $[(n-1)/n]^k$

Lets calculate false negative probability for item purple not in bloom filter
P(purple) = P(!false[red]) x P(!false[blue]) …. M times

M : is the number of elements in bloom filter

=>P(purple) = $[((n-1)/n)^k]^m$
=> P (!purple) = 1 - P (purple) = 1 - $[((n-1)/n)^k]^m$

For very big n, (n-1)/n = 1 - 1/n = 1, where, 1/n is approx to 0
=> P (!purple) = $[1 - (\text{approx } 1)^k]^m$
=> P (!purple) = approx 0

8. Conclusion

In conclusion, this report has provided a comprehensive exploration of Bloom filters, shedding light on their fundamental principles and basic operations. Bloom filters, as probabilistic data structures, excel in optimizing memory usage for set-membership tests. Their simplicity, efficiency, and versatility make them valuable in a wide array of applications.

Code:

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;

int h1(string s, int arrSize);
int h2(string s, int arrSize);
int h3(string s, int arrSize);
int h4(string s, int arrSize);

// lookup operation
bool lookup(bool *bitarray, int arrSize, string s)
{
    int a = h1(s, arrSize);
    int b = h2(s, arrSize);
    int c = h3(s, arrSize);
    int d = h4(s, arrSize);

    if (bitarray[a] && bitarray[b] && bitarray && bitarray[d])
        return true;
    else
        return false;
}

// insert operation
void insert(bool *bitarray, int arrSize, string s)
{
    // check if the element in already present or not
    if (lookup(bitarray, arrSize, s))
        cout << s << " is Probably already present" << endl;
    else
    {
        int a = h1(s, arrSize);
        int b = h2(s, arrSize);
        int c = h3(s, arrSize);
        int d = h4(s, arrSize);

        bitarray[a] = true;
        bitarray[b] = true;
        bitarray[c] = true;
```

```
        bitarray[d] = true;

        cout << s << " inserted" << endl;
    }
}

// Driver Code
int main()
{
    bool bitarray[100] = {false};
    int arrSize = 100;
    string sarray[33] = {"abound", "abounds", "abundance",
                         "abundant", "accessible", "bloom",
                         "blossom", "bolster", "bonny",
                         "bonus", "bonuses", "coherent",
                         "cohesive", "colorful", "comely",
                         "comfort", "gems", "generosity",
                         "generous", "generously", "genial",
                         "bluff", "cheater", "hate",
                         "war", "humanity", "racism",
                         "hurt", "nuke", "gloomy",
                         "facebook", "geeksforgeeks", "twitter"};
    for (int i = 0; i < 33; i++)
    {
        insert(bitarray, arrSize, sarray[i]);
    }
    return 0;
}

int h1(string s, int arrSize)
{
    ll int hash = 0;
    for (int i = 0; i < s.size(); i++)
    {
        hash = (hash + ((int)s[i]));
        hash = hash % arrSize;
    }
    return hash;
}
```

```cpp
int h2(string s, int arrSize)
{
    ll int hash = 1;
    for (int i = 0; i < s.size(); i++)
    {
        hash = hash + pow(19, i) * s[i];
        hash = hash % arrSize;
    }
    return hash % arrSize;
}


int h3(string s, int arrSize)
{
    ll int hash = 7;
    for (int i = 0; i < s.size(); i++)
    {
        hash = (hash * 31 + s[i]) % arrSize;
    }
    return hash % arrSize;
}

// hash 4
int h4(string s, int arrSize)
{
    ll int hash = 3;
    int p = 7;
    for (int i = 0; i < s.size(); i++)
    {
        hash += hash * 7 + s[0] * pow(p, i);
        hash = hash % arrSize;
    }
    return hash;
}
```