Linear Data Structures Lecture Notes

This module deals with the Abstract Data Type(ADT) Dictionary and its implementation using elementary data structures like arrays(unsorted and sorted) and linked lists. The second and third sessions introduce advanced data structures, namely hash tables and Bloom filters.

Introduction

Most real-life applications require storing a list of records and retrieving them later. The Dictionary or Map ADT is designed to facilitate this requirement. In a dictionary, each record or entry is identified using a key. Hence, these keys are required to be unique. For instance, in a phone book application to store and look up contacts using names, each contact is a record and its name is the key.

A dictionary supports three fundamental operations: ADD, FIND and DELETE.

1. The ADD operation takes a record as input and inserts it into the dictionary.
2. The FIND operation takes a key as input and outputs the corresponding record. If no such record exists, it outputs null.
3. The DELETE operation takes a key as input, returns the corresponding record and deletes the same. If no such record exists, it returns null.

Implementation using Elementary Data Structures

A dictionary may be implemented using various data structures. The choice of data structure is governed by its implications on the performance of a given application. The following table captures the running time of the ADD and FIND operations for arrays and linked lists.

	ADD	**FIND**
Unsorted Array	O(1)	O(n)
Sorted Array	O(n)	O(log n)
Linked list	O(1)	O(n)

1. Linked lists are preferred for applications where the total number of records cannot be estimated, e.g. a social networking database.
2. Arrays are preferred for applications with information about the total number of records.
    A) An unsorted array is used if the relative number of ADD operation calls is very high due to O(1) running time of the ADD operation.
    B) A sorted array is used if the relative number of FIND operation calls is very high due to O(log n) running time of the binary search operation.

Note: A dictionary is generally dynamic, i.e. records are added, retrieved and even deleted frequently. In special cases, a static dictionary may be required where all records are added at once and only the FIND operation is called subsequently. A sorted array is the preferred data structure in such instances.

Preprocessing Cost and Amortised Cost

Consider a dictionary in which all records have been added and sorted by key. If a large number of FIND operations are to be performed on this dictionary, then each of these has a running time of O(log n). However, the binary search operation with O(log n) running time may be used here only if the array has been preprocessed, i.e. sorted beforehand. Hence, a preprocessing cost needs to be associated to the running time of the FIND operation here.

At the same time it's unjustified to associate the preprocessing cost with every single FIND operation in a sorted array. Is there a number that can be associated with the expected running time of the FIND operation in a sorted array? This leads to the topic of amortized cost.

The **amortized cost** per operation for a sequence of operations is defined as the total cost of these operations divided by the number of operations.

- In our case, let us suppose we sort our map using one of the best algorithms you are familiar with: mergesort. We know that this method runs in O(n log n) time.

- In the sorted map case, sorting would be done in n log n time and each FIND operation would be carried out in O(log n) time using binary search. So, the amortized cost per operation for `m' search operations, where m >> n would be O(n log n + m log n)/m = O(log n) for our sorted map. On the other hand, the amortized cost for an unsorted map would be O(mn)/m = O(n) since each search operation takes linear time. This clearly shows the time optimization obtained by using sorted maps.

- For a sorted map, if m(search operations) is equal to one, then the amortized cost will be $\frac{n \log n + m \log n}{m} = (n+1)\log n \simeq O(n \log n)$. However, if m is close to n, then the amortized cost is $\frac{n \log n + m \log n}{m} = \frac{2n \log n}{n} \simeq O(\log n)$.

The preprocessing cost i.e. the cost to sort the array depends on the sorting algorithm. For instance

- Insertion sort is used in practice for small lists, when the list is small or the size of the list is small, then insertion sort works better than other algorithms.

- Quicksort is fairly efficient in practice; its expected time complexity is O(n log n), whereas it also gives the worst case of O(n^2). Quicksort is often the most commonly used algorithm in practice. If the list size is large, then quicksort performs better than insertion sort. Please note that both of these are used for in-memory sorting.

- The other most commonly used algorithm is merge sort. Merge sort is typically used when the data is in secondary memory, such as tape or disk or if the data is stored in a file. Since it's easier to access it sequentially, merge sort is preferred. Merge sort actually gives you O(n log n) algorithm in the worst case as well as average case, but the constant factor(the order complexity) for merge sort is higher than that of quicksort.

  Quicksort running time = $k_1 n \log n$ **,**
  Merge sort running time = $k_2 n \log n$ **,** $k_1 < k_2$

- Typically when the data is stored in secondary memory such as a tape or a disk or sometimes transferred over a network, you make use of merge sort and use quicksort for sorting the items present in primary memory.

	WORST CASE RUNNING TIME	AVERAGE RUNNING TIME	USE CASES(where the algorithm performs the best)
INSERTION SORT	$O(n^2)$	$O(n^2)$	Small lists stored in primary memory
QUICKSORT	$O(n^2)$	$O(n \log n)$	Large lists stored in primary memory
MERGE SORT	$O(n \log n)$	$O(n \log n)$	Lists stored in secondary memory and networks

The Bin sort Algorithm and Direct Addressing

You know how to sort the elements in an array based on comparison of keys and the minimum order for performing such a sort is O(n log n). But you can perform a sort without comparing keys using the Bin sort Algorithm.

The bin sort procedure or algorithm works for a list of records whose keys are unique and numeric and lie in a known range. It sorts them by indexing, for example, if there are 900 records whose keys lie in the range [100,...,999], then the record corresponding to each key K is indexed as (K−100), i.e. stored in an array of size 900 at the index (K−100).

For performing a Bin sort you need to make the following assumptions:

- The keys are numeric
- The keys are unique
- The keys are in a particular range that is known.

CODE FOR BIN SORTING is as follows:

```
for(i= 0; i< n; i++) {
lsnew[ls[i].key - low] = ls[i]
}
```

Here **ls** is a list of n records with unique keys and **low** is the lowest key in the list of n records, **lsnew** is the new sorted array obtained after bin sort. For every i^{th} record, you subtract the lowest key from its key value to obtain an index, say inew and then place the i^{th} record in index inew of the new array **lsnew**.

The time complexity of bin sort is **O(n)**, where n is the size of the list, since there will be **n** unique iterations for the **n** records present in the list. To **FIND** a certain record in the sorted list, you simply need to provide the key of the record and thus the result of performing bin sort on a list of records is that the FIND operation runs in constant time.

Direct addressing is a powerful method to implement a dictionary. Indeed, the fundamental operations run in constant time when direct addressing is used. But the major drawback of this method is its memory requirement and the associated sparsity i.e. direct addressing requires huge amount of memory, which could at times be impractical and virtually impossible. This paves the way to the next fundamental data structure Hash tables, which primarily solves these problems.

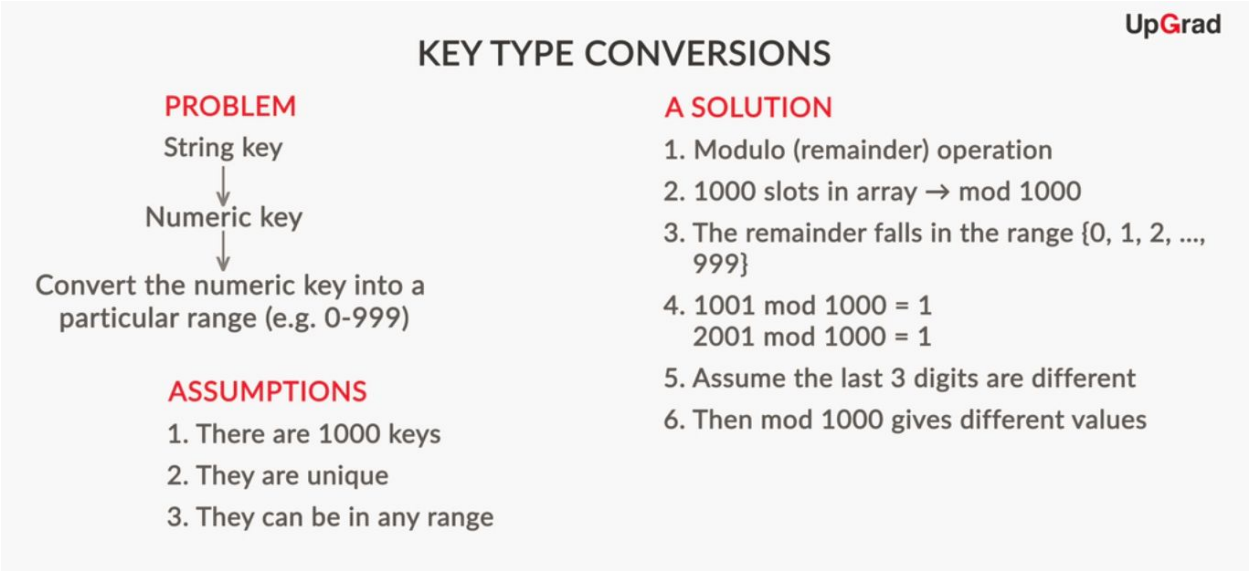
Hash Table Introduction

If your keys are in a known range and, they're numbers then you can map them to integer locations or indices of the array locations. If your keys are in a known range but they are not numeric then you can find a way to map any type of key to a numeric key and then restrict the range so that you will end up with locations or array indices and then use the same approach that you did in Bin sorting. For this process, you need to make use of Hash functions.

For instance, consider a range of strings which are known. You can map the string key to a numeric key using the hash function. Hash function also solves the issue faced with direct addressing, i.e. its huge memory requirement and the associated sparsity.

Hash Functions

A hash function is defined as a function that takes keys of any specified type as input and outputs numbers, corresponding to array indices. A hash function initially converts a given key to a possibly large integer and then reduces it to a smaller number, typically using the modulo operator. For a hash table of size n, the mod n operator is used to obtain indices less than n. These array indices are referred to as hash values. Each record is then stored in the hash value corresponding to its key.



Non-numeric keys such as strings are initially converted to integers(possibly large) using a hash code. A simple example of a hash code is to add the ASCII values of the individual characters of a string. The integer so obtained is then reduced to the range of available indices using a modulo operator.

Suppose you have 1000 records where one record has a string key “Hello”. To convert this string key to a numerical key you can make use of a Hash code. Assume this string to be an array of characters, then you can convert all the characters to the lowercase and add the individual ASCII values of each character. So “Hello” will give a sum of 372. To convert this integer to the corresponding numerical key you can make use of a modulo (Remainder) in this case. So this is a simple formula to convert any string of characters to a number once you have done this conversion you can do the modulo operation to get a value in the location range 0 to 999 so modulo operation will depend on what is the **range of numbers** or the number of locations. The mod value depends on the size of the table. If the table size is 1000, then you

need to do modulo thousand and if you have 2,000 values in your array, then the location range will go from 0 to 1999, therefore you will do modulo 2000. So for a table size of 1000 your hash will be 372.

This process is commonly referred as **Hashing**, where you convert any key to a desired numeric key. You need to keep in mind that each key must be unique or it may result in collision or overwriting of the 2 or more records at the same location.

Similarly you can take each value in the original list, compute a hash function which gives you a location and you can put that value or drop the value into that table in that location. This procedure is in line with the process of Bin Sort, where bin sort uses a simple subtraction operation instead of a hash operation.

You can find a particular value with the same efficiency you achieved in Bin sort. One of the results of doing Bin sort was that you could locate a particular element by simply obtaining the location in constant amount of time and therefore the cost of find operation was constant. Similarly, you can take the key and do a hash operation here and locate the index to insert or read a record. So you locate the index in constant time and hence, the effect is the same as that in Bin Sort.

Hash table Design and Performance Analysis

Hash functions enable adding and retrieving records in constant running time while requiring lesser memory than direct addressing. Let's introduce you to a data structure that is designed using hash functions.

By now you know that a HASH TABLE is an array of n locations. The indices in a Hash Table go from on to n-1. You need to assume that your hash function gives unique values in the range of {0, 1 ,..., n-1} and your input key can be any arbitrary value.

So let's assume that you have an array H with indices from 0 to (n-1). You have a hash function which you assume returns a location in this range 0 to (n-1) for any key K where K could be string type for example. So with this assumption, you can write your add function. The add function essentially takes a record R and R has a key and that key can be hashed. The hash function will give a location as an output for storing the record in the Hash Table.

UpGrad

THE ADD OPERATION

H[0... n-1]
hash(K)
ADD(R)
1. i = hash(R.key)
2. H[i] = R

RUNNING TIME
1. Constant running time
2. Two operations
   a. Compute the hash function to get the index
   b. Use the index to store the record

Hash Function offers a constant running time for the ADD operation as it requires 2 operations, each running in constant time-

a. Computing the Hash Function to get the index
b. Use the index to store the record

Similarly you can do a FIND operation. The find operation takes the key of a record and returns a value of the location where the record can be found in the Hash Table.

UpGrad

THE FIND OPERATION

FIND(K)
1. i = hash(K)
2. return H[i]

RUNNING TIME
1. Constant time operation
   a. Compute the hash to get an index
   b. Return record

Hash Function offers a constant running time for FIND operation as it requires 2 operations, each running in constant time-

a. Computing the Hash Function to get the index
b. Returning the record from the index

Note: Hashing the key and hashing the record mean the same things.

For a given application, a hash function should :

- Accept any possible key as input and
- Output one of the indices(locations) in the allocated array (i.e. table).

The latter i.e. requirement is under the control of the designer/implementer: i.e. one can ensure this requirement by choosing the size of the array (table) to be equal to the modulus of the hash function.

For a good Hash function you should always keep the following points in mind:

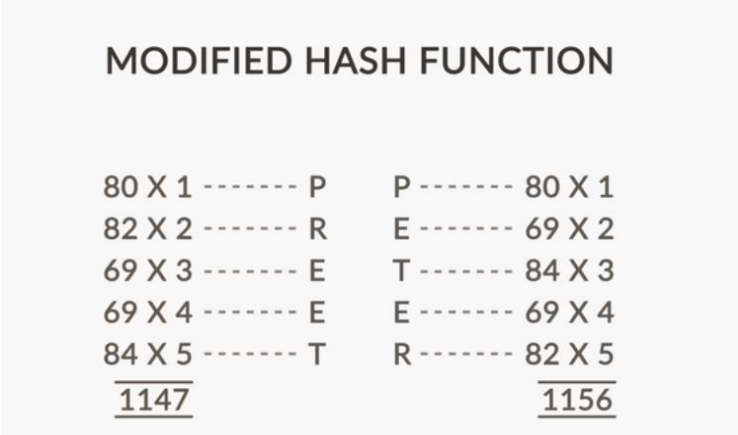
- "Use" all the information in the input (i.e. in the keys).
- Distribute the keys uniformly across the array indices.
- Output different hash values for keys that are "similar yet unequal".

If a Hash function allots all the records to only a set of hash table indices instead of uniformly distributing them or fails to make use of all the information provided by the record key, then such a hash function is usually unacceptable and is designated as a bad Hash function.

Collision and Chaining

For the ADD and FIND operations in the previous segment, we made an assumption that the keys will be mapped to unique locations. But that assumption is not always true. For instance, for the hash function that we designed, we took the ASCII values of the characters and added them up. If 2 strings have similar (PETER & PREET) names, then such a hash function will fail as the sum of the ASCII values of characters will be the same for both strings.

The records will then get stored in the same location which means one will overwrite the other. The phenomenon of two distinct keys K1 and K2 getting hashed to the same index is known as collision in hash tables. That is, K1 ≠ K2 but h(K1) = h(K2).
Collision cannot be eliminated, but may be reduced to a great extent by designing good hash functions. One method is shown in the figure here

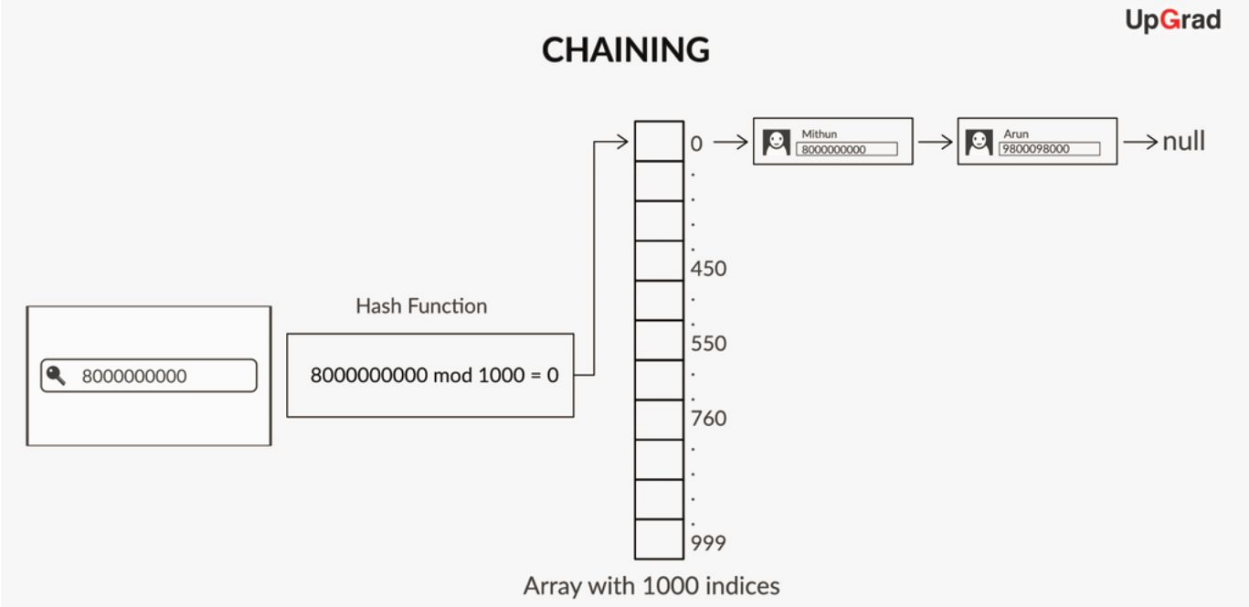


But again this method may fail under certain circumstances and still this function may not be collision proof.

Earlier the hash table was simply an array and each element in the array held one record. Now, in order to design a collision-proof hash function you should change the structure of this hash

table and that leads to designing the hash table as an array of linked lists.

Each location in the hash table will have a linked list of records. So you can go to any particular location and add one more element into the linked list. Therefore, if multiple keys hash on to the same location, you will end up storing both the records that hash on to the same location in two different nodes in the linked list.



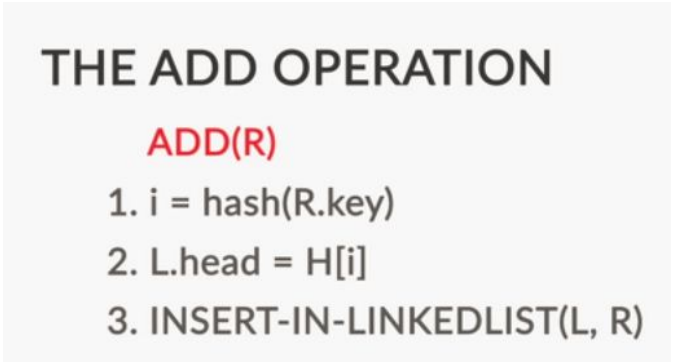
The memory requirement of a hash table, also referred to as its space complexity is O(n) where n is the number of records to be stored. The fundamental operations ADD, FIND and DELETE run in O(1) time for a hash table, as in the direct addressing method. This makes the hash table a preferred data structure for the implementation of a phone book. Collision is a drawback associated with hashing and this problem is alleviated using various methods like chaining and open addressing.

Performance Analysis of Hash tables with Chaining

You now have a hash table which is an array of n locations, each location is a linked list and if two values collide, i.e. if two keys are hashed onto the same location, then you will just add the records one after the other into the linked list.

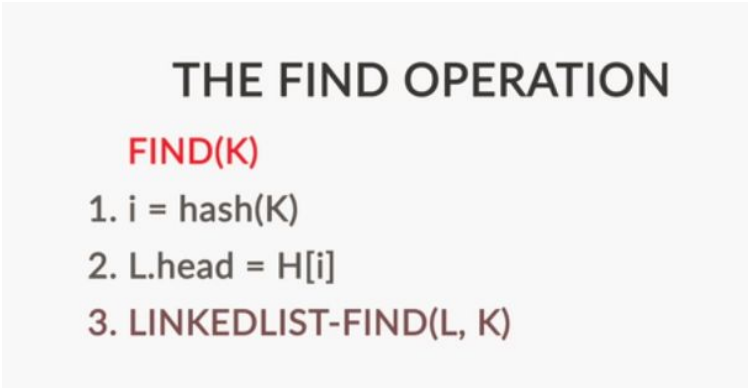
You have already seen how these functions, which are written with linked lists are slightly different because once you find the hash value, you only find the location and the record may lie in any node of the linked list of that corresponding location.

You can add a record in the linked list using the modified ADD operation as shown below:



Here each new record is allocated a specific location using the hash functions and its key. At that location, the record is stored in the first node of the linked list. The H[i] which was earlier just a simple value will now correspond to the head of the linked list. You will have to insert the new record at the first node of the linked list using the Insert-in-LinkedList operation. As you will be always adding the record to the first node, the add operation time will always remain constant, i.e. O(1).

Similarly, a FIND operation using linked list is shown in the figure below:



Earlier the hash function returned a location that was an index. So earlier you were simply returning H[i], where there was only one record in that location. Now H[i] is the head of the linked list which will correspond to L.head and therefore you will have to traverse through the linked list and then return a record using the LinkedList-Find operation.

As for finding the record you will be traversing through the linked list which could take a time of O(n) in the worst case.

Although the problem of Collision in hash tables is alleviated by using linked lists at each index location, in this process, the performance of the hash table is affected. Indeed, the running time of the FIND operation in a hash table with chaining grows to O(n) since you cannot return the array index directly. Rather, you need to search the linked list sequentially for the record.

Expected number of elements in linked list (n) = $\frac{Total\ number\ of\ items\ in\ the\ hash\ table\ (M)}{Size\ of\ the\ hash\ table\ (N)}$ **= (M)/(N)**

THE FIND OPERATION

FIND(K)

1. i = hash(K)
2. L.head = H[i]
3. LINKEDLIST-FIND(L, K)
   a. O(n) time in the worst case
   b. n = $\frac{M}{N}$
   c. Typically, we write O(1 + $\frac{M}{N}$)
   d. If M is close to N
   e. Then, $\frac{M}{N}$ is close to 1
   f. O (1 + $\frac{M}{N}$) is close to O(1)

The expected running time of the FIND operation in a hash table with chaining becomes O(1+M/N) where M is the number of stored records and N is it size, i.e. the number of indices in the hash table. O(1) represents the constant time required for hashing the key and O(M/N) represents the time taken to search the linked list sequentially. If M >>N, then the performance is adversely affected.

Sizing and Rehashing

Now that we have discussed about chaining in Hash tables, we move forward and review the performance factors of Hash Tables and discuss what is usually done to maintain a constant running time for the fundamental operations: Add, find and delete.

UpGrad

FIND OPERATION

FACTORS THAT DECIDE THE COST OF THE FIND OPERATION

1. Load factor, which is $\frac{M}{N}$

   M = Number of elements inserted

   N = Size of the table
2. Worst-case number in a linked list

WORST-CASE SCENARIO

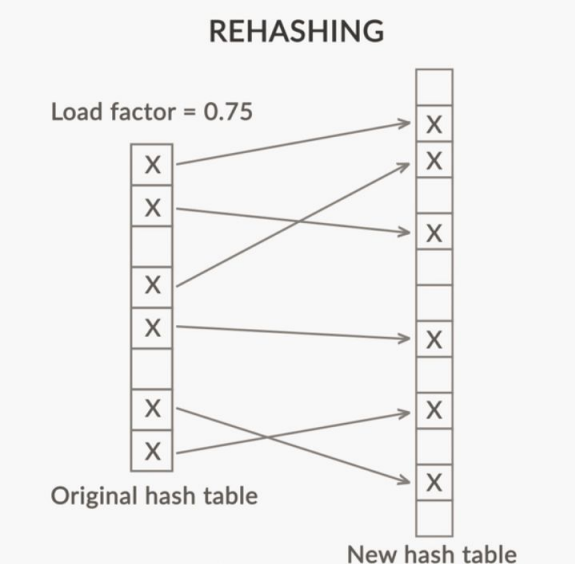
1. Cost of O $(1 + \frac{M}{N})$ for the FIND operation
2. Under the assumption that all the M elements are equally distributed
3. A good hash function ensures this
4. Some hash functions may distribute values unevenly
5. The values that arrive may be skewed
6. Some linked lists become long and some become short
7. The expected value of the length of a linked list is close to the total number of elements inserted

So the cost of the find operation depends on two things:

1. One is the load factor which is M over N where M is the number of elements inserted and N is the size of the table. This is one factor that determines the cost of the find operation.

2. The other factor is that the expected number M/N doesn't become the worst case number.

The running time of the FIND operation in a hash table is directly proportional to its load factor and hence, needs to be kept low. In practice, 0.75 is the load factor at which the hash table is sized and rehashed.



A new hash table double the original size is created and all existing records are rehashed into it. This helps in maintaining the constant running time of the FIND operation. To reduce the load on the Hash table, increase the size of the hash table. Therefore, M remains the same, N has been increased and hence, the Load factor $\frac{M}{N}$ correspondingly decreases.

You should not trigger rehashing quite often since every rehashing operation has a significant cost that is proportional to the current number of elements in the hash table.

Java API for Hash table

We have discussed a whole lot of theory regarding the dictionary ADT and its implementation using various data structures like sorted and unsorted arrays, linked lists and most importantly hash tables. Let's now look at how a real application like a contact list may be implemented in a programming language like Java.

UpGrad

A CLASS IMPLEMENTING INTERFACE CONTACTS

```
public class ContactList implements Contacts{
          public MobileNum findNum(String callName)
{//Method should be defined here}

          public void addEntry(String callName,
          MobileNum mb)
{//Method should be defined here}
}
```

The interface essentially gives a blueprint of the methods findNum and addEntry. The findNum method returns the phone number of the person that you are looking for on the contact list. The Add operation takes a contact list, adds a new entry with the name and a number and it returns a modified contact list and the user's understanding is that this is the list of entries.

Now the provider concerns can dictate which implementation to choose. You can choose one particular implementation. However, there could be alternatives for implementation. For instance, in this example, instead of a sorted list you could have used a hash table for the reasons show in the figure below.

UpGrad

REASONS FOR USING A HASH TABLE

1. The FIND operation runs for O(log n) time in a sorted list
2. But, O(1) time in a hash table
3. The operations are fast
4. An occasional delayed operation
5. In a hash table, the ADD operation is straightforward
   a. Time taken is O(1) in a hash table
   b. Time taken is O(n) in a sorted array

Java provides two APIs for hash tables, namely Hashtable and HashMap. These APIs provide several methods, the most important being 'put' and 'get'. 'put' is used to add records and 'get' is used to retrieve them using their respective keys. Recollect that records are considered as (key, value) pairs in the Java environment. In the phonebook example of looking up contacts using names, each name is a key and phone number is its value.

UpGrad

A CLASS IMPLEMENTING INTERFACE CONTACTS

```
import java.util.Hashtable;

public class ContactTable implements Contacts {
          private Hashtable namesAndNumbers;

          public ContactList(int initSize) {//Define
          method here}

          public MobileNum findNum(String callName){
          //Define the method here}

public void addEntry(String callName, MobileNum mb) {
          //Define the method here}
```

You have seen an implementation of the contact list using Hashtable class in Java. You can also use a HashMap implementation for the same.

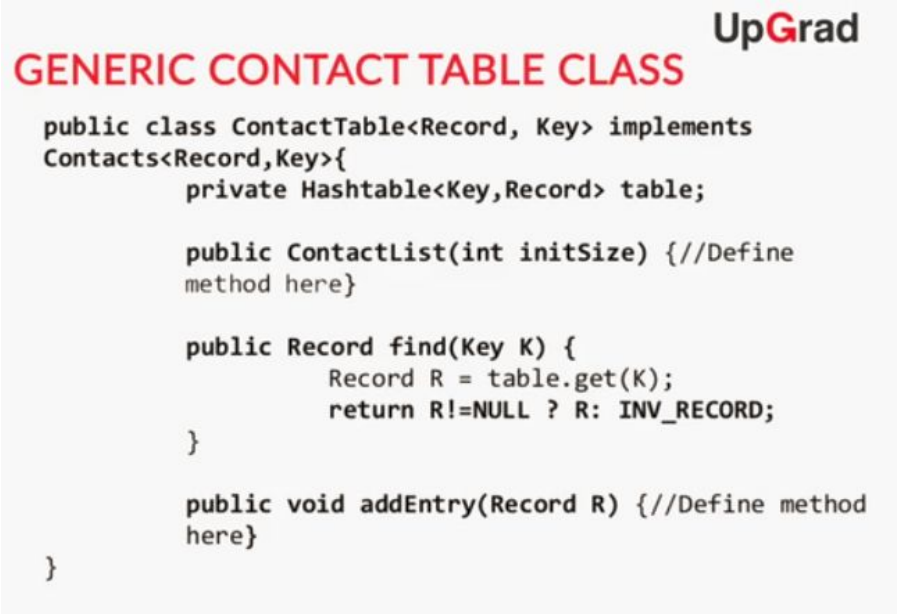
There are a few subtle differences between the Hashtable and HashMap APIs:

1. Hashtable is synchronized while HashMap is not.
2. Hashtable does not support null values while HashMap does.

ConcurrentHashMap is a Java class that provides thread-safe features like the Hashtable class while allowing concurrent get() calls. You may read further about the ConcurrentHashMap class and compare it with HashMap and Hashtable classes here.

Generic Hashtable in Java

Java supports generics which allows the developer to implement a dictionary to store and retrieve any type of key-value pairs using the Hashtable or HashMap APIs. In the last segment we looked at how to implement an ADT in Java and the example we used was that of a contact list. We then discussed how to generalize that ADT so that the contact list is not only in the context of a mobile phone but it could be any general contact list where the contact could be a list of email, phone, fax etc.



We have modified the contacts interface to accept record and key as type parameters. It means that when you implement a **public interface Contacts <Record , Key>** then the records could be any specific record **type** and that could be instantiated at the **point of use.** The interface above has two methods those methods operate on a generic type called record and a generic type called key.

Real-life applications require various types of keys and values like the Contacts app in your smartphone that supports both 'Search by Name' and 'Search by Number' features. These features may be implemented seamlessly in Java due to Generics in Java.

Let's recall some of the use cases for a generic interface in Java:

- Using generics, a hash table with keys of type Long(mobile numbers) and values of type String(names) is initialized as follows

  Hashtable numbersToNames<Long,String> = new Hashtable<Long, String>() ;

  The type parameters Long, String within the angle brackets <> declares the Hashtable to be constituted of keys of type Long and values of type String.

- For this hash table numbersToNames, a FIND function that looks up values of type String using keys of type Long is implemented as follows:

```
String FIND(Long callnumber){
        String R=numbersToNames.get(callnumber) ;
        return R!=null?  R : "Invalid Number";
      }
```

Note that it is not required to cast numbersToNames.get(callnumber) to String type because it is declared as String by the code generated by the compiler.

On the other hand, a hash table to implement the "Search by name" function would be as follows:

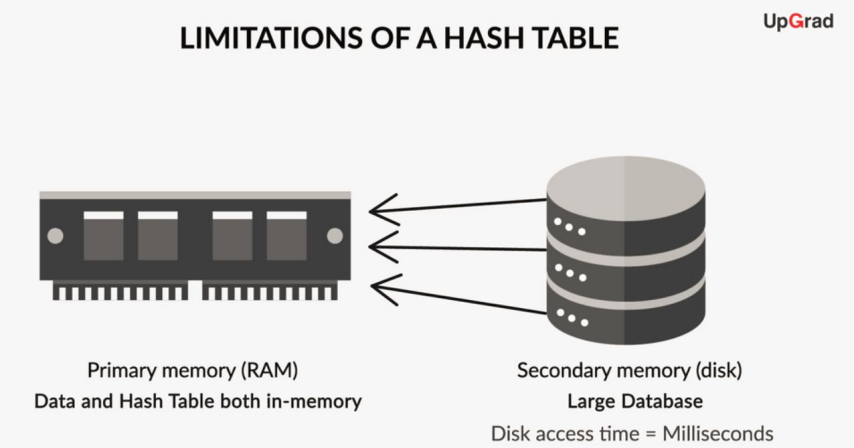
```
Hashtable namesToNumbers<String, Long> = new Hashtable<String, Long>();
Long FIND(String callname){
      R = namesToNumbers.get(callname);
      return R;
      }
```

Bloom Filters Introduction

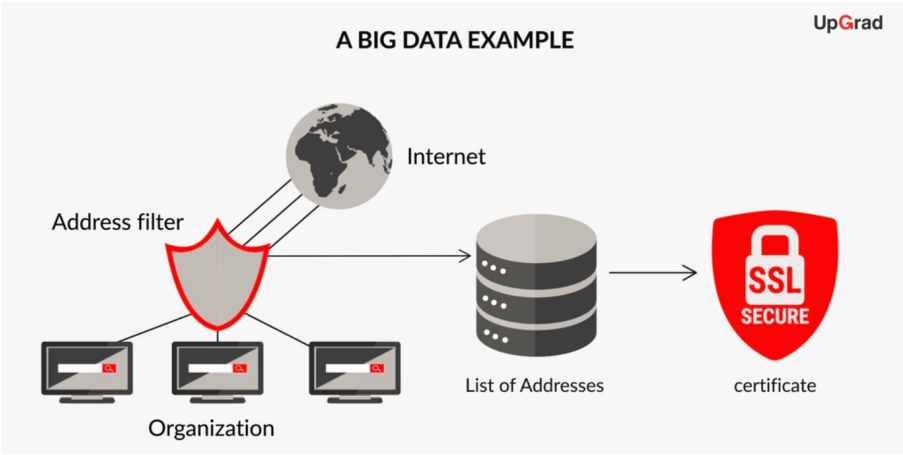
In this session, you were introduced to a compressed probabilistic data structure called Bloom filter. Bloom filters requires very less memory and are used to test the membership of records in a huge database.

Compressed Data Structures

So when you have situations where the **Hash table** gets really large then all the data cannot be stored in memory. Typically you store things in a hash table assuming that the data is all in **memory** and that the hash table is in memory. But when you have a large database, let's say of billions of records, you have to store the records on a disk.



Since you are unable to store them in memory, a portion of that is fetched at a time and then processed. This has a cost associated with it. The typical access time from hard disk is very large. In a disk, typically it's in milliseconds whereas in memory, the fetch time is in nanoseconds.



- So let's consider an example where you have a very large list of blocked addresses from which you don't want to receive any communication in your organisation.

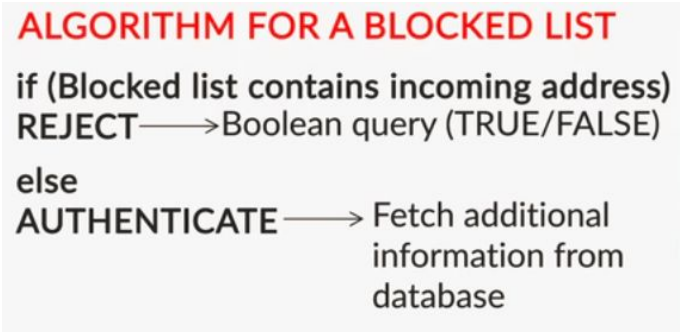
- So this list is fairly large because there are billions of addresses on the Internet and anybody could be sending packets which you don't want to receive in your network.

- So you have this large database on your network and the large database contains a list of addresses as well as authentication for valid addresses. For instance, you may have a certificate if the packet is coming from a valid address. The certificate allows you to check if the address is valid or you may have other information which says that information from this packet coming from a particular address is acceptable.

- So in your database, you have different categories of these addresses. Some of those addresses are blocked and if the address is blocked, you don't want to receive the packet from that address

To perform this task you can write a small algorithm, which is a 3 line algorithm which says **if an incoming address is in the block list then reject the packet. Else, authenticate the package using the certificate before passing it through.**

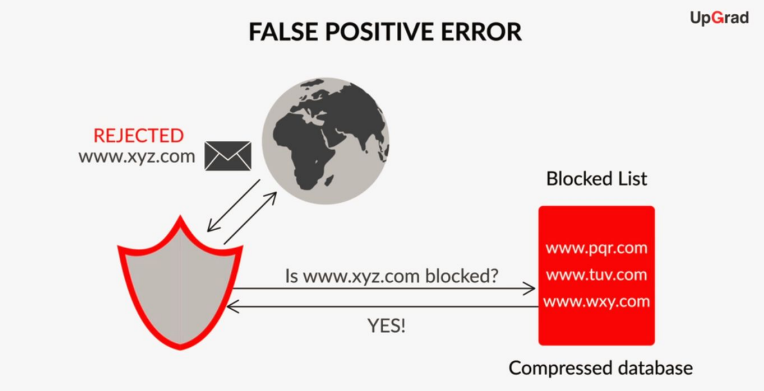
Now whenever an address is in block list, it hits the disk once since the address has to come from the hard disk, and when you say Authenticate, it hits the disk again. This two step operation can be converted to one by modifying the algorithm to “The first time you fetch the entry from the disk, store it in memory and then access it again and when you access it, take it from memory.” This operation is possible, so per packet you have to do one disk access. So, for n packets you will be requesting n disk accesses.



If you know that you only need to check whether the address is in the block list or not, this is a Boolean query and is going to return only a true or false value. So you don't need any other information from the database. But in case you need additional information from the database, such as owner information or a certificate, you will have to fetch this from the database and if the database is in the hard disc then you have no way of avoiding this second look up.

So the blocklist is usually a sublist of this total database, let's suppose you have only 10% of the address as your block list, now you don’t have to do a full query on the hard disk for every request for every package. Instead you can have a compressed database in memory. This

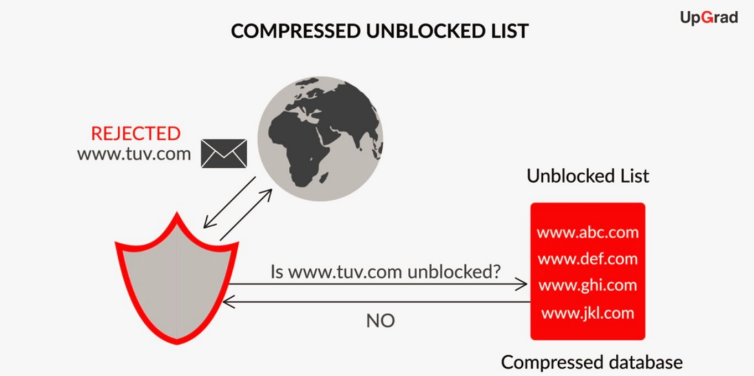
compressed database will only give you a yes or no answer. Now to do the boolean lookup, you will only use this database in memory and for that 10% of the requests, you will get a yes or no value. But you must keep in mind that if you try to compress the database you may lose information.



Assume that you have such a compressed database that answers the query with high probability accurately. It might be possible that sometimes it will give you a false answer saying that the address is in the block list although it is not in blocklist. So if there is a false positive, in that case you could reject packets which you are supposed to be allowing. To fix this consider the reverse scenario you can **modify** your earlier code as ***"****if address not in block-list then authenticate else reject."*

Now imagine a compressed database that has 90% of the items. Since you are compressing the database, you may lose some information. Due to the lost information, sometimes it is possible that you may get a false positive, i.e. it is going to say "the address is not in blocklist" even though it is a blacklisted operation. But if it says that the address is not on the blocklist then you are not immediately allowing the packet but first going to authenticate.

So when there is a false positive, you take a hit on your disk access time, because for a small percent of time you will be accessing the disk. But for a significant portion of the time, when it is not giving false positives, you can ignore the disk access and could do a reject.



Such a setup is a cost saving mechanism. When there is a scenario where you can actually construct the compressed table that may give some false positives, but most of the time it will give you the correct answer, and it does so in memory without having to store the entire database.

This solution is referred as a **Bloom filter** because it is typically used for filtering out accesses to the disk. When you have a large database in the disk and you can't store everything, you store a compressed database in the memory since your memory access is faster. So a Bloom filter is a solution for such a scenario where you have a large amount of data and quite often you have to go and hit the hard disk.

When you use a compressed database i.e. a Bloom filter and you have the compressed Bloom filter in local memory, you check and only go to the network when it is necessary. So at times when you are not going to the hard disk, you are not going to the network to access a distributed database and thus saving time. Hence, a reduction in running time is achievable using a Bloom filter which is a probabilistic database, which means it will give false positives occasionally but most of the time it will give a correct result and it will answer only yes or no queries rather than emitting the entire record.

Bloom filters are used to alleviate overheads due to disk and network fetch requests in data-intensive applications like web search. An important point to be kept in mind is that a Bloom filter is an array of bits, and hence a Bloom filter of size 1 MB is an array of 8×1024×1024 bits!

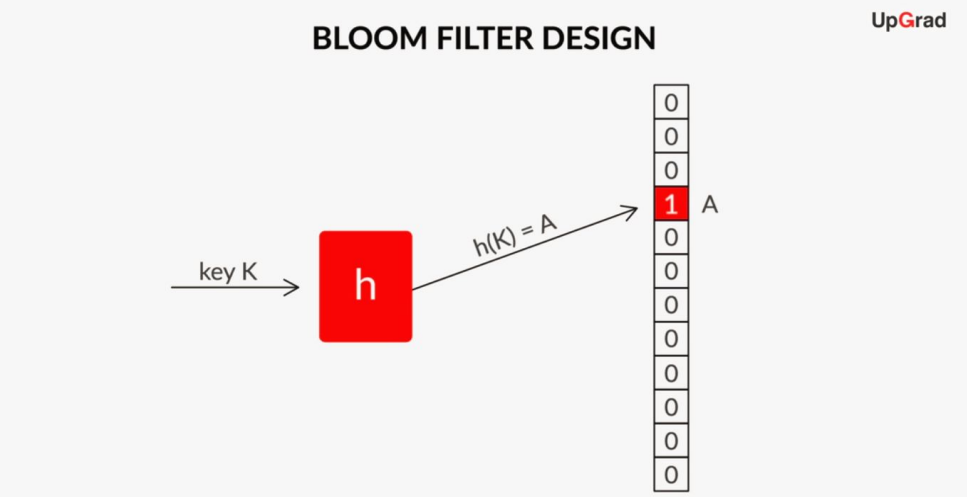
Bloom FIlter Design

By now you know that a Bloom filter is a probabilistic compressed data structure which only answers yes or no queries as opposed to Hash tables where the find operation can retrieve the whole record.

A Bloom filter is an array of bits coupled with multiple hash functions. Initially, all the bits are set to 0. When a record is added, its key is hashed by multiple hash functions and the bits

corresponding to their hash values are set to 1. If a record is to be looked up, its key is hashed by all the hash functions and the corresponding bits are checked if they are set to 1. If all the bits are set to 1, the Bloom filter returns YES or TRUE. If any of these bits are found to be 0, the Bloom filter returns NO or FALSE.

Essentially we use the Bloom filter to filter out queries which need not be considered further. As long as there are only false positives but no false negatives we can filter out queries and only when needed we go and check a secondary database.

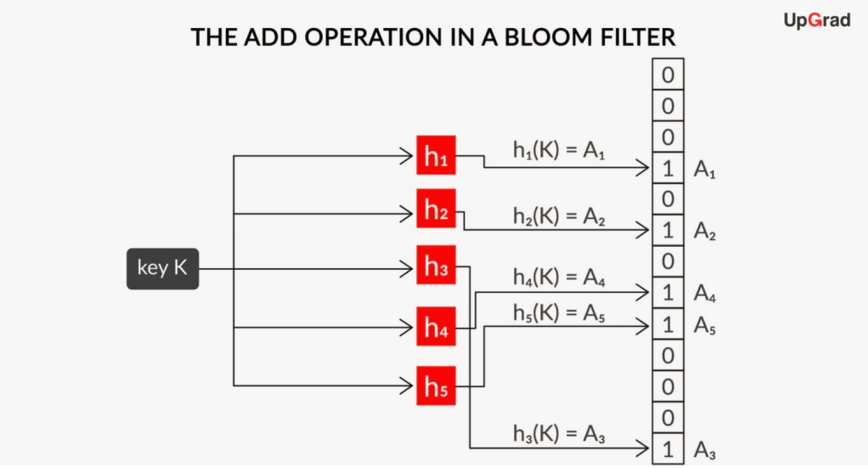


Now to ensure that one particular location does not get a 1 and many other values collide on to this and they all the end up turning to 1, we are going to use multiple Hash Functions.

Add Operation in Bloom Filter:

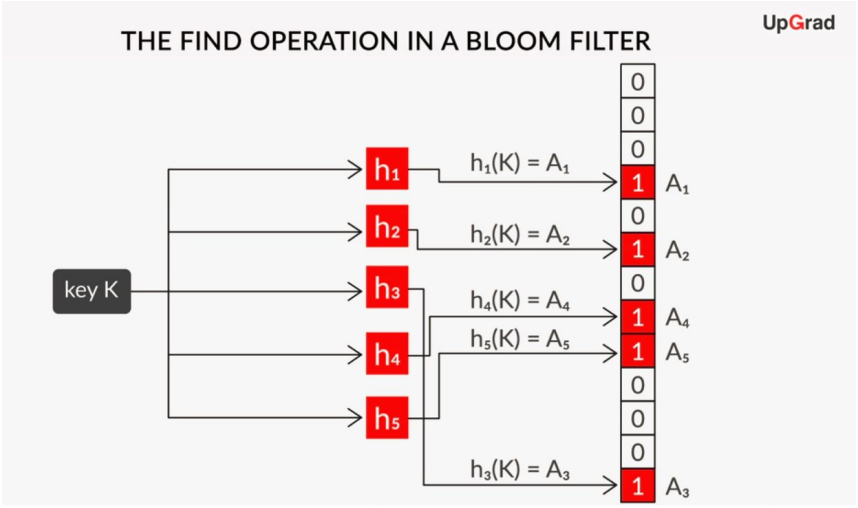
- Assume that you have 5 hash functions, H1, H2, H3, H4 and H5. When a key K is given to add a value, you compute the Hash Function 5 times using these different hash functions.

- Each of them is going to give a different location. H1 gives you A1, H2 gives A2 and then you get to A3, A4 and A5 accordingly. At all these locations you are going to set the bit to 1.

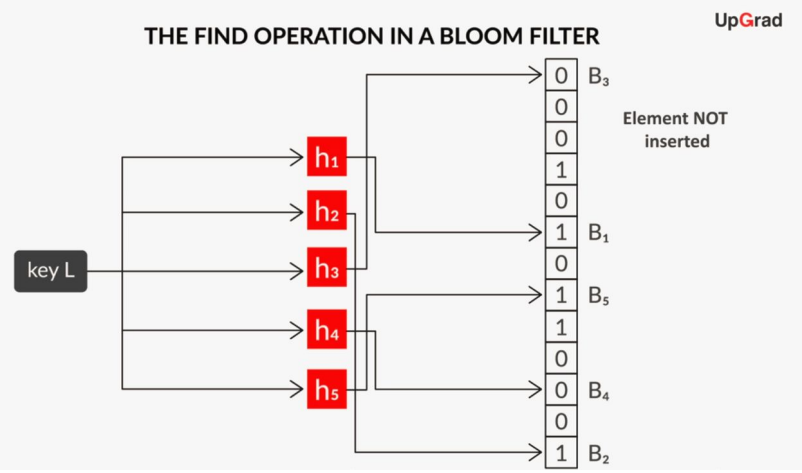


- All the 5 locations were obtained out of the same key using 5 different hash functions. All the hash functions are going to give you locations in the range of 0 to N -1 where you have N items in the Hash table and all these addresses or all these locations will be set to 1. Now this is the add operation.

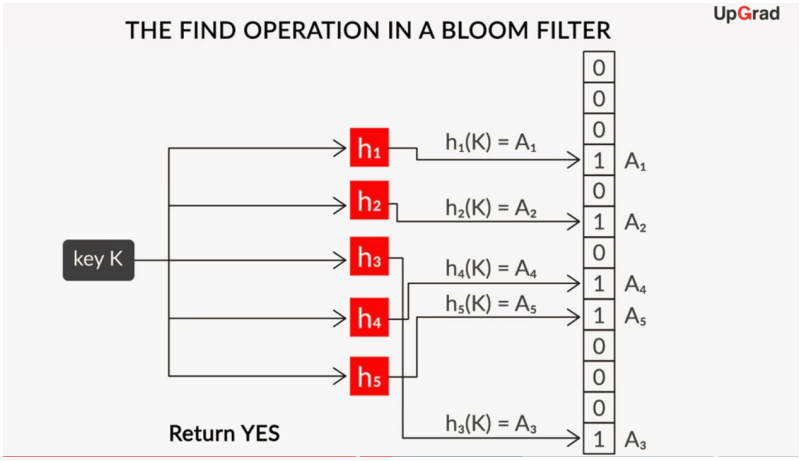
Find Operation in Bloom Filter:



Again compute the 5 hash functions for the same key as in the add operation. Each one will give you an address and you got 5 addresses. If all 5 addresses are 1, it implies that this particular value has been inserted.



If one of the locations is zero, i.e. if you get a 0, it implies that this element was never inserted. So you compute 5 hash functions and if one of the locations is 0, then you know definitely that this value has not been inserted.



If all 5 values happen to be 1, it returns a YES answer. But it is not guaranteed that the value has been inserted. Because of collisions, some other key could have been inserted causing this to be set to 1, which might have generated a false positive. If you get at least one 0, then you know that this particular hash value has not been touched, which in turn means that this particular key has not been inserted. Hence, whenever you find a 0, you can confidently return a NO answer.

ALGORITHM FOR THE ADD FUNCTION

UpGrad

ALGORITHM FOR THE ADD FUNCTION

ADD(Key K)

1. $h_1(K) = A_1$
2. $h_2(K) = A_2$
3. $h_4(K) = A_3$
4. $h_4(K) = A_4$
5. $h_5(K) = A_5$
6. $A_1=A_2=A_3=A_4=A_5=1$

For an ADD(key k), you compute 5 hash functions. So you compute H1(K) = A1 and H2(K) = A2 and so on. Now you have these 5 addresses and want to set your table for these 5 different locations. For all the given 5 address you will set the value as 1. Hence, the time complexity of the add operations depends upon setting the bits corresponding to these hash values to 1. So the order for add functions is O(1).

ALGORITHM FOR THE FIND FUNCTION

UpGrad

ALGORITHM FOR THE FIND FUNCTION

FIND(Key K)

1. $A_1 = h_1(K)$
2. $A_2 = h_2(K)$
3. $A_3 = h_3(K)$
4. $A_4 = h_4(K)$
5. $A_5 = h_5(K)$
6. if (A_1==1 && A_2==1 && A_3==1 && A_4==1 && A_5==1)
   return 1; ⟶ Sometimes wrong (False positive)
7. else return 0; ⟶ Always correct

The find operation is going to compute these locations and it's going to ask *"If all Ai's are 1 then return 1 else return zero."* (Here A[i] value is Hi(k)).
So you are going to compute all the hash values and ask *"If all the hash values are 1, then return 1, else return 0"*, returning 0 says that the element is not present. When you return 0, it is always correct. But when you are returned 1, you may be incorrect. This is referred to as a false positive and so this data structure may return a false positive. However, notice that the running

time complexity of the find operation is also O(1).

To summarize, a Bloom filter uses multiple hash functions to set certain bits to 1. When the FIND operation is called to search for a record, the exact same bits are checked if they have been set to 1, and if yes, it returns TRUE. Finally, False positive errors occur due to collision associated with these hash functions.