# Hashing

Prakash Hegade

~*~*~*~*~*~*~*~

**Hashing**

By Prakash Hegade

**Smashwords Edition**



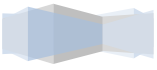
~*~*~*~*~*~*~*~



~*~*~*~*~*~*~*~

For any query or questions kindly mail: **prakash.hegade@gmail.com**

~*~*~*~*~*~*~*~

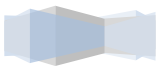


# Hashing

## First edition

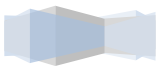
## 2017

## ISBN: 9781370432943



## Contents



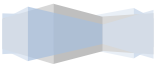


## Preface

This ebook talks about ‘Hashing’ data structure. Hashing can go deep. However the intention of this ebook is to understand the basics of the data structure and walk through with conceptual development.

The material covers introduction, definition, designing hash functions, collision resolution techniques and applications. There is also a sample and simple code (C language) given to make the definitional understanding better.

- **Prakash B Hegade**



## 01. Introduction

**"Give the key, Get the address,**

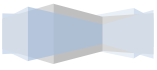
**Makes the data, faster access."**

This is a # and this is a #Tag. Hash table and Hashing has yet lot to do. So, we have yet lot to understand. A first-class amount of background information will make the study more than interesting. Let's settle down the dust and chew the fat!

Say you have an inclination of writing a daily journal (not on Facebook), every day one page, as already dated in the diary. The diary has the tag to identify the month and days are easily accessible. On a lonely day with no work, if you feel like reading something which you scribbled on May 31 when you had something special with your crush, you can easily access the page of necessitate and read it. You precisely know where exactly to go.

***Dear Diary,***
***#unnecessary #hash #tags #everywhere***

Days and years pass by and diaries pile up. One fine day you decide to make it electronic and transfer everything to your computer. Every page is a separate document. Each page is saved somewhere on the disk. Thinking like a computer



engineer it's our duty to make sure to provide a mechanism where for each date, when the page has to be searched, it's quickly accessible.

Let us say there is some magic black box which takes the date as input and gives out the location on disk where that page can be fetched and loaded into memory.

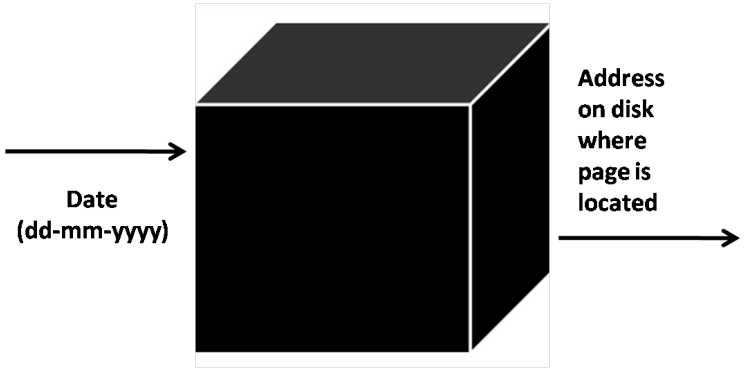



*Figure 1.1: Magic Box*

Well, yes, we have dig up to the point. That is the hash function sitting inside the magic black box and converting the required. Let us pre-empt all the necessary questions.

**It is a search?**

Yes, off course and indeed.

But we need a mechanism to make the search faster. We don't want to search page by page and then give out the result. We want a faster access.

**Who designs this magic sitting inside the black box?**

We will. We have to.

That is the challenging task. Quite difficult but with some noise we can easily have a good enough solution.

**What exactly is this Hash?**

Function and an array!

Funny, but that's it. One function which computes based on the supplied input which we call it as key and an array to store all the values associated with keys. Yes, you got it right, a [key-value] pair.

Let us have a closer look inside the black box!

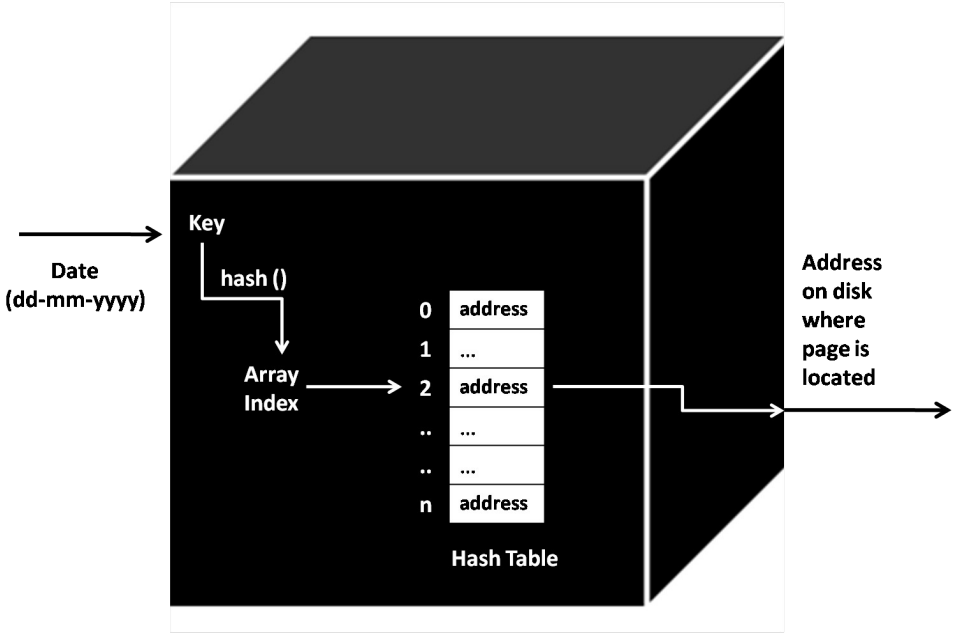



*Figure 1.2: Zoom inside the Magic Box*

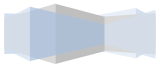
The date is the key which is given as input to the hash function. Hash function computes using the key where the result is an array index. There is an array, which is a hash table, housing the addresses. Based on the generated index value, the address is fetched from the index and sent out of the box as the value.

**We now have some serious questions to answer!**

- How do we make sure that every key is hashed to unique index?
- How do we make sure that the result of hash function is within the index of hash table?
- What do we do if the hash table is full?
- Is there a possibility that a few values in hash table never be used?
- Is there a possibility that multiples keys hash to same index location?
- How do we design a hash function? Are there any rules?
- How do we know we have designed a good hash function?

Let us unpack the answers. Let us see a very naïve implementation of the hash table. The requirements are as follows:

- We need a hash table to hash records of 100 students with roll number 101 to 200.
- The size of hash table is 100



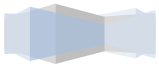
- The hash table will have entries set to -1 if no entries exist yet or when deleted.
- The key is the roll number and as well as the value. For example: Roll number 102 should be hashed to index '2' in the hash table and value in the table at that index should be 102.
- There is a global variable called 'hash_table' which maintains the data
- The operations performed are insert, delete and search of keys

### Quick Quiz

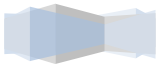
Deep Down, hash table is after all an array. Agree more or agree less?

Program below implements the said constraints. The code is self explanatory.

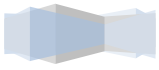
```
#include <stdio.h>
#include <stdlib.h>

// A global variable to hold the values for the supplied key
int hash_table[100];

// Function: menu
// Description: prints the menu option
// Input Params: NIL
// Return Type: void
void menu() {
   printf("\n ** MENU ** \n");
   printf("------------------------\n");
   printf("1-Insert a key\n");
   printf("2-Search a Key\n");
   printf("3-Delete a key\n");
   printf("4-Display Hash Table\n");
   printf("5-Exit\n");
   printf("------------------------\n");
}

// Function: insert_key
// Description: computes hash and inserts the key into the hash table
// Input Params: NIL
// Return Type: void
void insert_key()
```



```
{
   int roll = 0, index = 0;

   printf("Enter the roll number\n");
   scanf("%d", &roll);

   // A simple hash computation using % operation.
  // You can Notice here that hash function simple takes the
 // mod 100 of the supplied key
   index = roll % 100;
   hash_table[index] = roll;
}

// Function: search_key
// Description: searches the value associated for the given key
// Input Params: NIL
// Return Type: void
void search_key()
{
   int roll = 0, index = 0;

   printf("Enter the roll number to be searched\n");
   scanf("%d", &roll);

   index = roll % 100;

   if(hash_table[index] == -1)
      printf("No Such Data Exists\n");
   else
      printf("You have reached the %d Roll number Student\n", hash_table[index]);
}

// Function: delete_key
// Description: deletes the supplied key entry from the hash table
// Input Params: NIL
// Return Type: void
void delete_key()
{
   int roll = 0, index = 0;

   printf("Enter the roll number\n");
   scanf("%d", &roll);

   index = roll % 100;

   if(hash_table[index] == -1)
      printf("No Such Data Exists\n");
   else {
      hash_table[index] = -1;
      printf("Associated Data Deleted\n");
   }
}

// Function: display_key
```



```
// Description: prints the hash table, -1 with no entries
// Input Params: NIL
// Return Type: void
void display_keys()
{
   int index = 0;

   printf("\nIndex\tRoll Number\n");
   for(index=0;index<100;index++)
      printf("%d\t%d\n", index, hash_table[index]);
}

int main()
{
   int choice = 0, index = 0;

   //Initialize the hash table with the value -1
   for(index=0;index<100;index++)
      hash_table[index] = -1;

   while(1) {
      menu();
      printf("Enter your choice\n");
      scanf("%d", &choice);

      switch(choice) {
         case 1: insert_key();
                 break;

         case 2: search_key();
                 break;

         case 3: delete_key();
                 break;

         case 4:display_keys();
                 break;

         case 5: printf("Terminating the Program\n");
                 exit(1);

         default: printf("Invalid Option\n");
                 break;
      }
   }

   return 0;
}
```



## 02. Definition

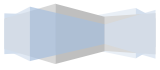
With all that now understood, let us see the definition of hashing. This is what Wikipedia says (as of in July 2017):

**"A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found".**

- Associative array?
- Array of buckets or slots?

**Those were something new! Isn't it? But, just the terms!**

Associative arrays are nothing but data types which are composed of key-value pairs. That solves one. We have already discussed a problem that multiple keys can be housed at same index. One fix is that each location is collection of records (buckets). An array or a linked list connected to each index and hence so why we call it an array of buckets or slots.



*Good to know!*

Given a hash table with **m** slots that stores **k** keys, we define the **load factor** as **k / m**.

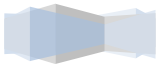
Coming to example that we discussed, what we saw is one variant of hashing. Hashing can be used to generate a key-value pair for any required scenario. That is the main summary you should be taking away from all this. Look out the applications section for more examples to find out where all hashing has put down its ass.

**Quickie** –

Let us say we have to maintain the whole of Indian population data. Which one do you think would be the key if a hash table is maintained? AADHAR card number? Can it be?

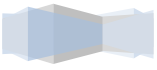
Knowing all this unknowingly puts some restrictions on the design of hash function. Naturally, one will have to adhere to the following:

- Must be **easy** to compute. Who doesn't want that! Check out the type of data coming in for the key. It can help in designing a suitable hash function.
- To possible extent it should maintain a **uniform distribution** of keys across the hash table. Too many keys should not be mapped to same index. The better way is to look out for any patterns if present in the keys.



- **Reducing** the collisions! The size of the table and an attempt to reduce the number of collisions is a 'to-be-followed'.

Hashing is also sometimes referred as **randomizing** as there is no immediate obvious connection between the key and the location of the corresponding record.



## 03. Designing Hash Functions

Let us do some cooking. A hot potato is on the table! Well, it could turn out to be a smelly dish after all.

**3.1.** Design a hash function to hash integer data of range 0 to 1000 over table size 100.

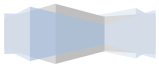
**One possible solution:**

```
int hash(int key, int maxAddress)
{
    int sum = 0;
    sum = key % 149;
    return sum % maxAddress;
}
```

Here maxAddress is size of the table which is 100.The hash function is % with 149 which is just a prime number. There is no hard defined logic here. We have integer range of 0 to 1000 and table size is 100. So the collisions are bound to happen.

**Question:** If the input key is 560, what is the address generated?

You can code the complete program and find out the right answer among the following:



a. 15

b. 13

c. 16

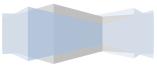
d. 14

**Answer:** Option b. 13

**3.2.** Design a hash function to hash a string of 6 characters over table size of 1000.

**One possible solution:**

```
int hash(char key[7], int maxAddress)
{
    int sum = 1, index = 0;
    int ascii=0;
    for(index = 0; index < 6; index++)
    {
        ascii = key[index];
        sum = (sum * 100 * ascii) % 7867;
    }
    return sum % maxAddress;
}
```



Did you see what have we just done? You can program a whole different one serving the purpose.

**Question:**

Input any key of 6 characters and find out what the address generated will be.

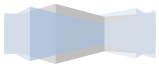
**3.3.** Design a hash function to hash a string over table size of 1000

**One possible solution:**

```
int hash(char * key, int maxAddress)
{
    int sum = 0;
    while( *key != '\0'){
        sum = sum + *key;
        key++;
    }
    return sum % maxAddress;
}
```

Here the input key string can be of any length.

**Question:** If the input key is “college”, what is the address generated?



You can code the complete program and find out the right answer among the following:

a. 731

b. 688

c. 580

d. 462

**Answer:** Option a. 731

**3.4.** Design a hash function to hash a string of atleast 3 characters long over table size of 10000.

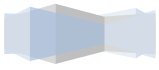
**One possible solution:**

```
int hash(char key[], int maxAddress)
{
    int sum = 0;
    sum = (key[0] + 27 * key[1] + 729 * key[2]);
    return sum % maxAddress;
}
```

Can you try some other function?

**Quick Quiz**

It is easy to implement a bad hash function. Good or Bad?



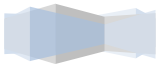
## 04. Collision Resolution Techniques

Research is made on several techniques on resolving the collisions in hash table. As now we already know that it's inevitable and cannot be prevented no matter how good the hash function is, the only way is to have a good mechanism to handle it. This section discusses few popular such collision resolution techniques.

Can we apply the hash function again and again with a little variation? Yes! We name the technique as **double hashing**. When a collision occurs, a second hash function is applied to the key to produce a number 'K' which is relatively prime to the number of addresses. If the overflow address is already occupied, K is added to it to produce another overflow address. This procedure continues until a free overflow address is found.

Alternatively, you hashed a key and got to know that the index is already taken. What would you prefer to do? Search sequentially for the next available slot? That exactly is a **Progressive Overflow** technique. It is also known as **Linear Probing** for obvious reasons. The method is simple and for many cases looks to be perfectly adequate solution.

Let us see an illustration.



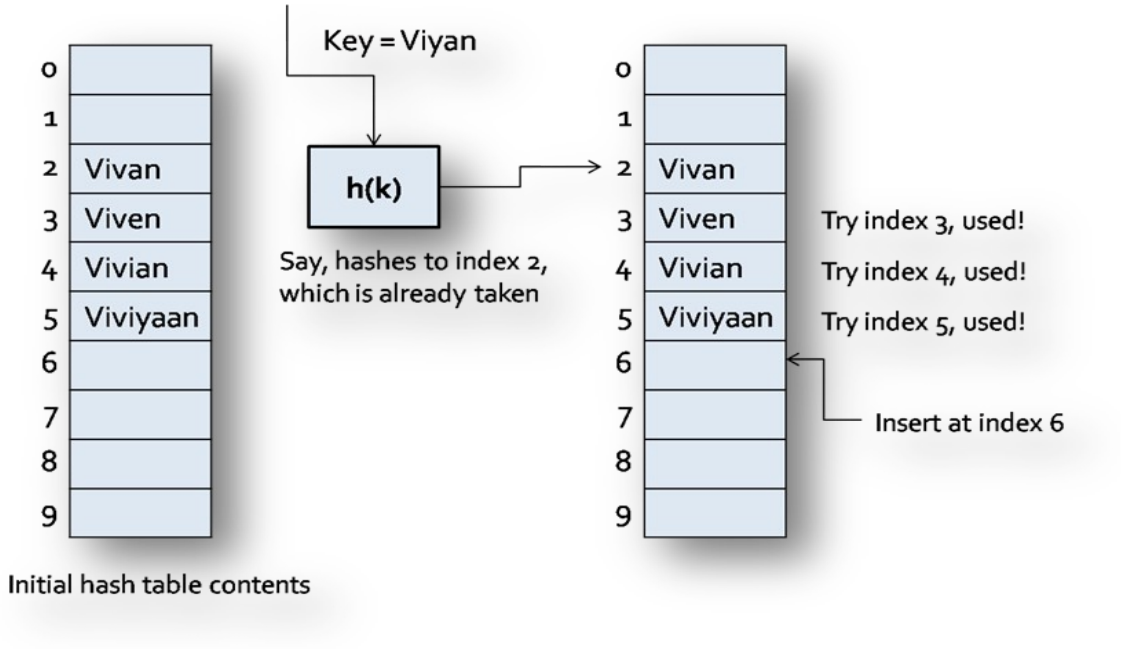



*Figure 1.3: Linear Probing Example*

As shown in figure 1.3, the table already has indexed for 2 to 5. The new key 'Viyan' which is supplied to hash function generates the index 2, which is already occupied. The next available free index is searched and inserted. So the key gets inserted at the index 6.

What if we want to improve this technique? We want to know that a key which was supposed to be hashed at index 2, is now available at index 6. We are telling the hash table, where exactly to search next while accessing the data. This technique is called as **Chained Progressive Overflow**.

In chained progressive overflow we maintain one more index called the 'synonym-index'. Once a free slot available is found after collision, we come back to original and update the synonym-index with available index. This can continue for any number of synonyms.

Below is an illustration with same example in figure 1.4.

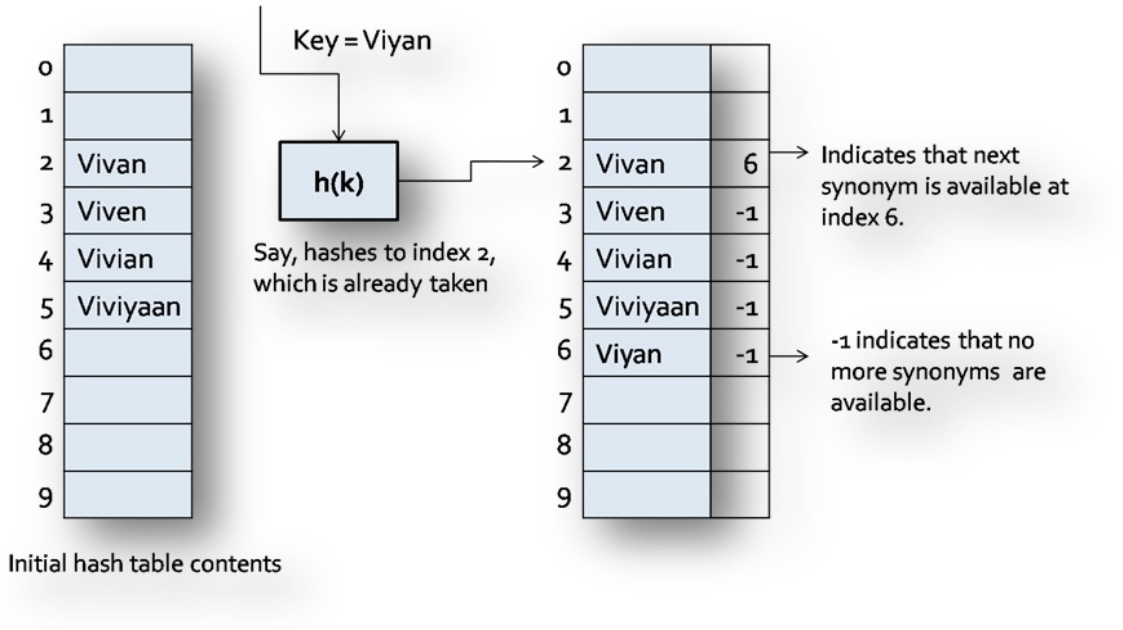



*Figure 1.4: Chained Progressive Overflow*

As seen in figure above, synonym-index of 2 is 6. It means while search, if index generated was 2 and data is not what we required, then we need to search at index 6. The one at 6 is also if not the one required then the data is not found in the table.

We can also maintain a separate overflow area. When a collision occurs, we map it to another hash table. We can now mix and match several approaches. This over flow area can now be maintained with a chained progressive overflow technique.

We can use the concept of **buckets** to avoid collisions. Let us say we have the five keys and the addresses generated are as shown in the figure below. A bucket concept would allow us to create a buckets of size 3 so that the respective keys can be stored are same index. Figure 1.5 demonstrates the concept.

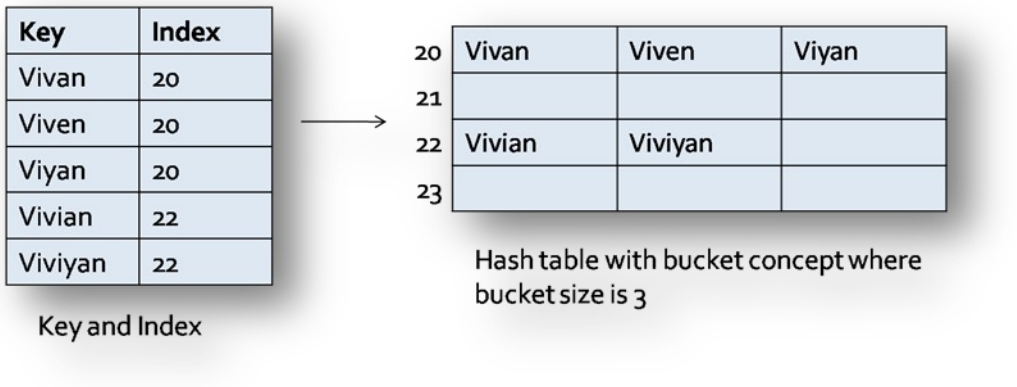



*Figure 1.5: Bucket Concept*

The disadvantages are that, we need to carefully pick the bucket size. Otherwise we would end up in wasting a lot of space.

If you are thinking a little ahead with known concepts so far, your head might have already bounced questions that why not use a linked list. **Scatter table** concept is all

about the same. For the same example a scatter table looks like the following as shown in figure 1.6.

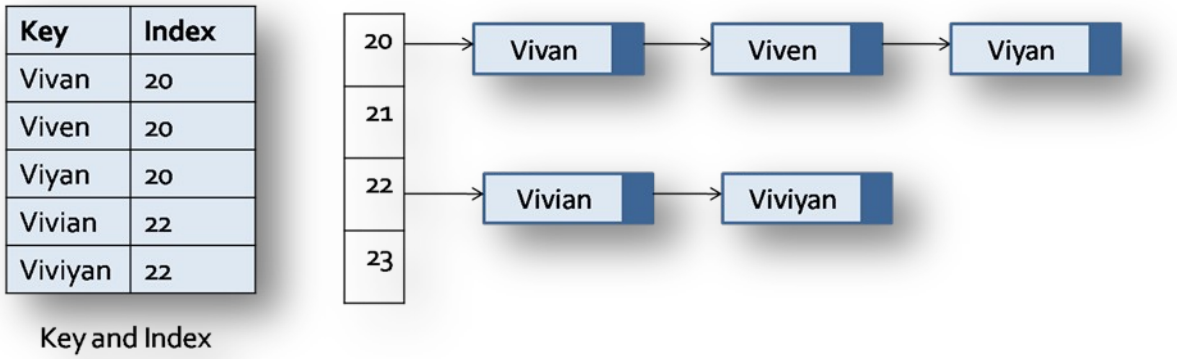



*Figure 1.6: Scatter Table*

Using a linked list we can dynamically allocate the memory as and when the keys are hashed to the index.

This is not the end. Most of the times the collision resolution techniques are appropriate mix and match of learnt techniques based on the type of application they work on. A lot of other new techniques can also be browsed over.

## 05. Applications

In the heat of the moment, let us look at the applications of hashing. This is why we did study all of the above, to know and understand all of the below. They are generic, leaving out for you all to explore the specifics.

### 1. The Undercover Agent for Passwords

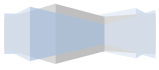
We all have made our accounts safe by having passwords. What if passwords file is on the loose? Rather than saving passwords, it can be hashed and saved using a strong hash function. What is actually dumped on the disk is not the password but the hash of them, adding a level of security.

### 2. The Network Authenticator

Plentiful applications in area of cryptography like digital signatures, key generation, etc using a secure hash function. Long story short – look up a network security material or wait until you take the course.

### 3. Influential Personality

Hash data structure has motivated for the discovery of many new efficient data structures based on the same logic and idea. Bloom filter stands ahead in the line. It is a space efficient, probabilistic data structure to test if an item is member of a set.

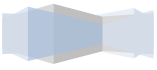


### 4. Marrying Key and Values

Where ever there is a need of mapping between key and values, hashing could be invited. It sure is an assisting guest. Key and values are going to stay happily forever. Checking duplicate records, finding a similar record, cutting long representations to short, etc, rings a bell?

### 5. You wear the Cap

As now you wear the cap, go ahead and add some feathers. Exploring and learning other new applications from a library book or web is going to be very welcoming!



## About the Author

Academician, always a research student and blogger who loves to read and write!

Twitter Handle: **@itsPhTweet**

Blog: **itsphblog.wordpress.com** and **itsphbytes.wordpress.com**

~*~*~*~*~*~*~*~