

## Lecture Notes

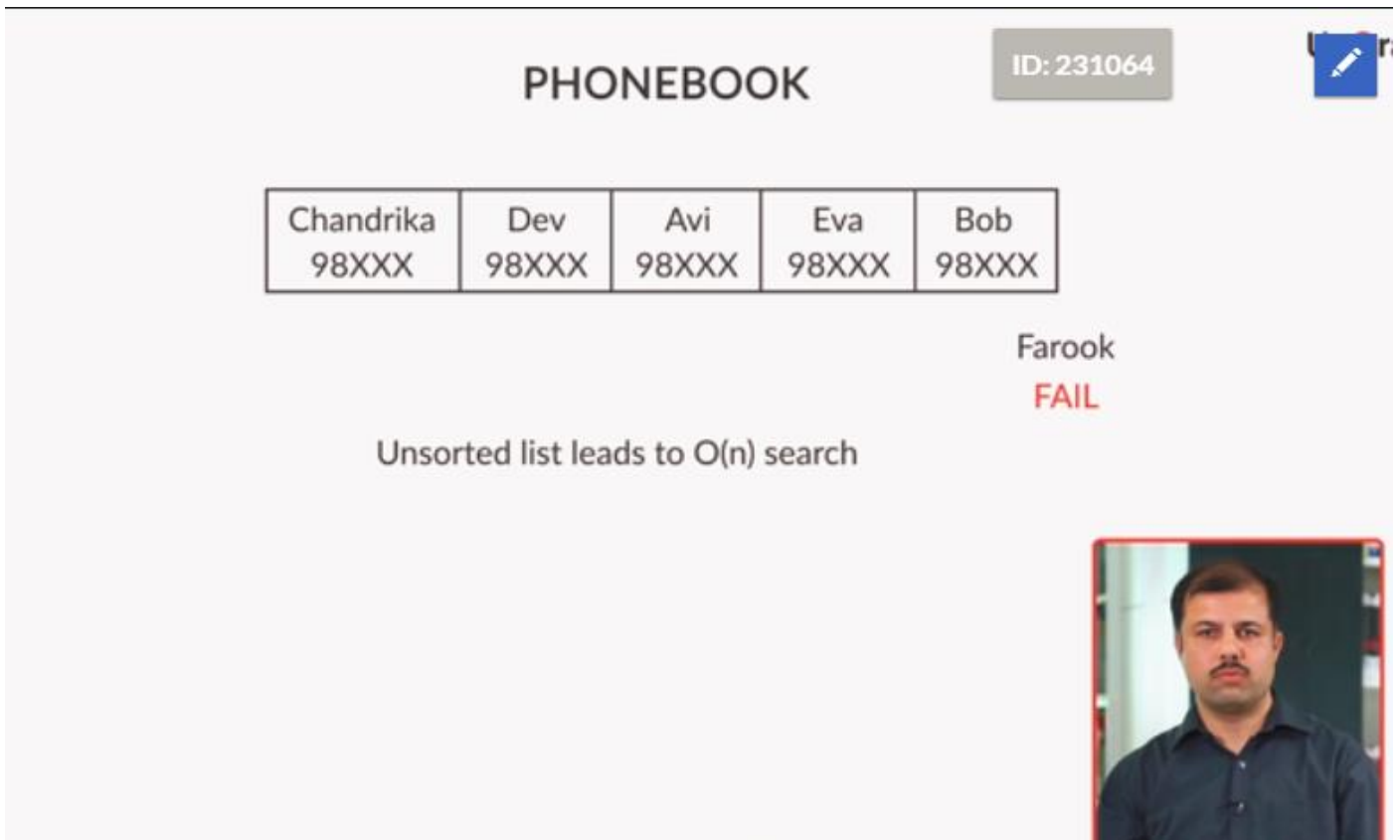
### Hash Tables

Let's revisit the topics learnt in the session on Hash Tables.

In the Searching and Sorting module, you implemented a phone book using an array of names, which, in the worst case, took linear time to search for a name in the phone book when the names were unsorted. However, it would be a better move if you store the names in a sorted manner, thus reducing the overall time complexity to  $O(\log n)$  by applying binary search functionalities to the phone book. But is this what you desire to achieve in the case of a phone book? No, you want fast retrieval, something in the order of  $O(1)$ . In this summary, you will get to grips with the data structure known as Hashtable, which retrieves results in  $O(1)$ .

### Array Implementation of a Phone book

You saw in the video that searching for a name in a phone book implemented using an array required you to traverse the entire list in the worst case. This leads to a time complexity of  $O(n)$ .



**PHONEBOOK** ID: 231064

|           |       |       |       |       |
|-----------|-------|-------|-------|-------|
| Chandrika | Dev   | Avi   | Eva   | Bob   |
| 98XXX     | 98XXX | 98XXX | 98XXX | 98XXX |

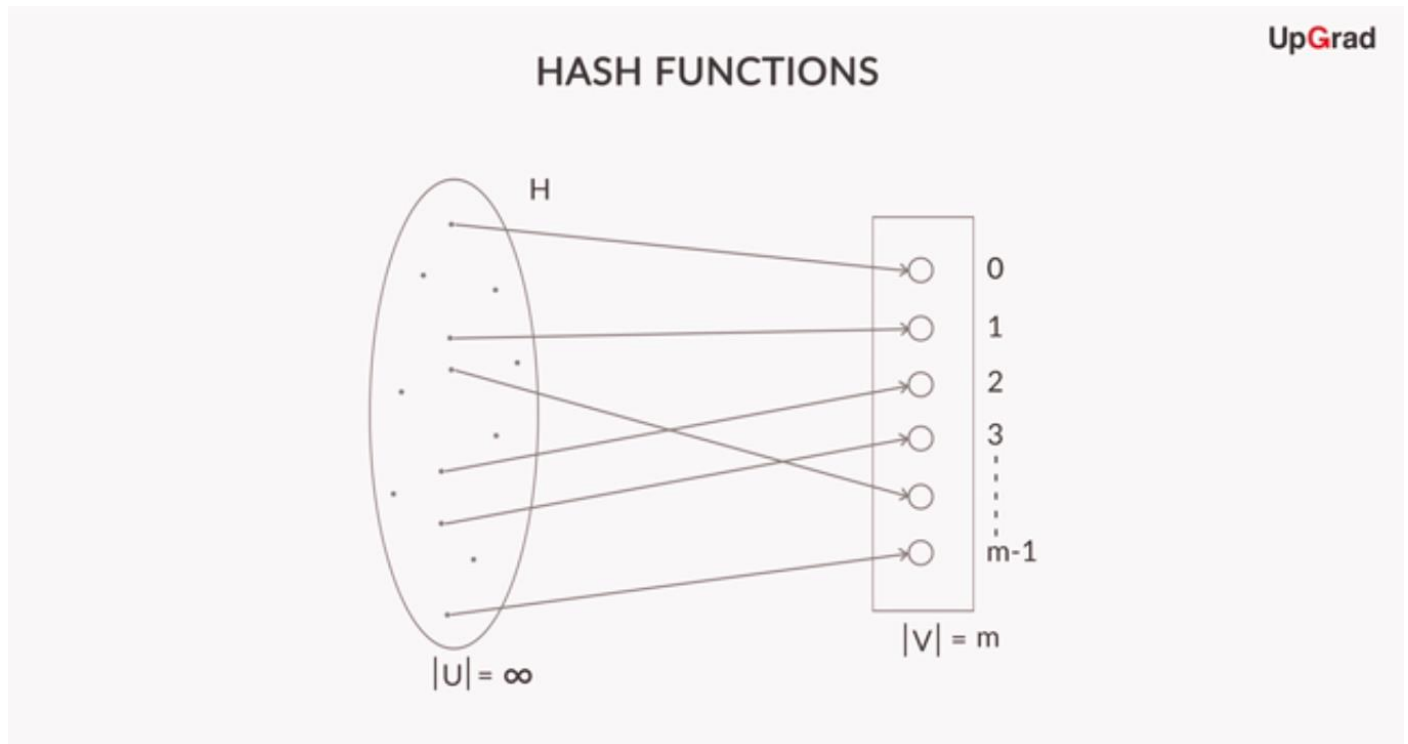
Farook  
**FAIL**

Unsorted list leads to  $O(n)$  search

However, if you are cautious to sort the names in the phone book, you can apply the method of binary search, which will lead to an overall decline in the time complexity to  $O(\log n)$ .

## Hashing and Hash Functions

The idea of hashing is based on the central concept of the **hash function**. A hash function is any [function](#) that can be used for mapping arbitrarily sized [data](#) to fixed size data. As you can see in the figure below, the cardinality of the input set is infinite, whereas the output is a range with a fixed size  $M$ , where  $0 < M \leq N$  ( $N$  is the set of natural numbers).



The basic idea behind a hash function is to map the input domain, which could be any generic data type, into the output range, which in most cases is a natural number that can be realised in the program memory.

An important point to consider here is that the hash function should be fast to compute, or else, the whole objective of having a constant time to search or add won't come to pass if you have a hash function that itself is computationally expensive.

So, let's take a look at a hash function that takes up the first alphabet of an individual's name, and then based on the position of the letter in the English alphabet, it maps the name to the corresponding index in the hash table.

In the example below, you can see that the name 'Avi' starts with an 'A' and since A is the first letter of the English alphabet, it gets mapped to the first index of the hash table, i.e. 0. Similarly, 'Bob' gets mapped to the hash index 1, and so on.

Now, when a new entry, let's say 'Farook', comes in for search, you can directly apply the hash function to 'Farook'. Here, see that the first letter of the name is 'F', which means  $H(F) \rightarrow 5$ , so we can directly go to index 5 of the hash table and see whether the entry 'Farook' is there in the hash table or not. This search for the name 'Farook' by applying the hash function and directly checking the hash index 5 would take a total time of  $O(1)$ , i.e. constant time.

This way, the use of hashing and hash functions can help you realise the constant time complexity, which is desired in the case of the phone book.

## HASH FUNCTION

H

|   |    |
|---|----|
| A | 0  |
| B | 1  |
| C | 2  |
| : | :  |
| F | 5  |
| : | :  |
| Z | 25 |

Avi  
Bob  
Chandrika  
Dev  
Eva

|    |           |
|----|-----------|
| 0  | Avi       |
| 1  | Bob       |
| 2  | Chandrika |
| 3  | Dev       |
| 4  | Eva       |
| 5  |           |
| :  | :         |
| 25 |           |

Hash Table

Farook  
 $H(F) \rightarrow 5$



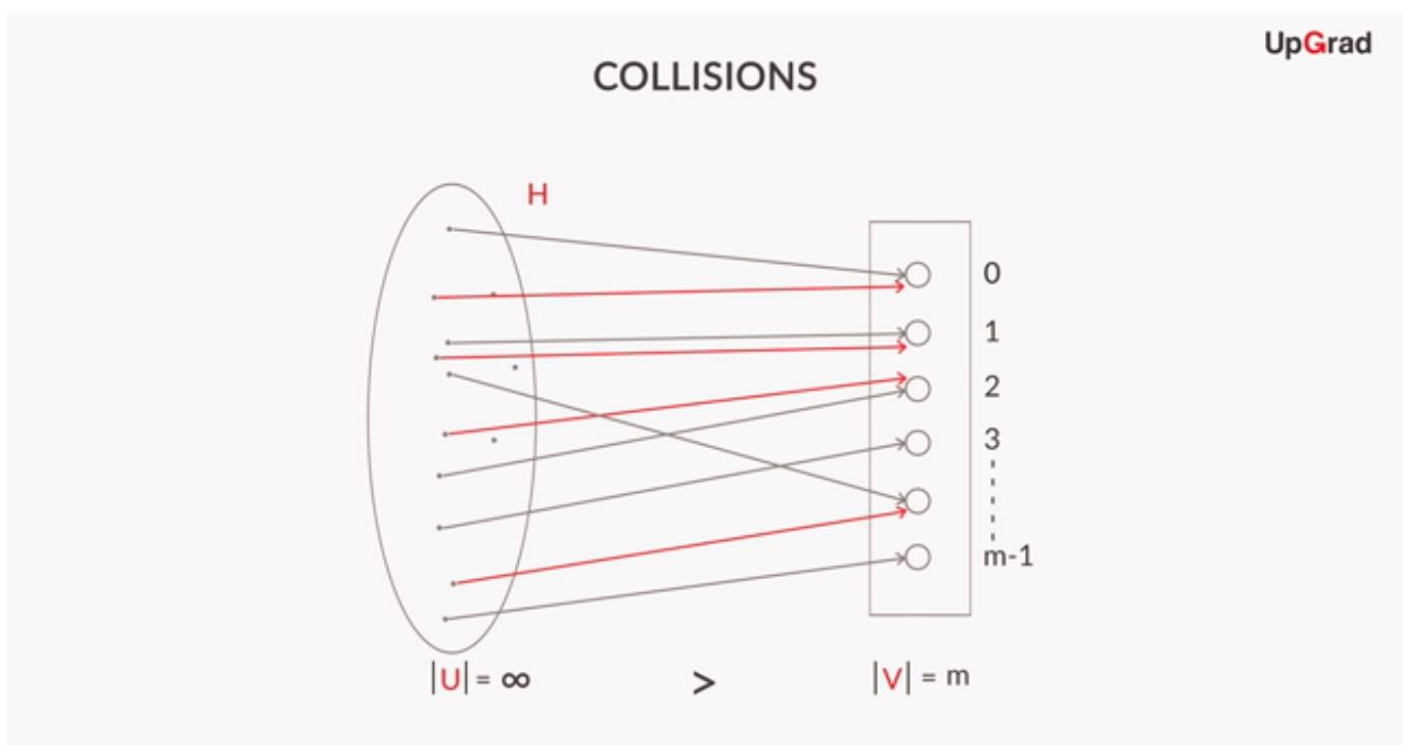
Let's say that you're working with some mathematical operations such as division, and modulus, which play a great role in many hash functions. For instance, the hash function  $H = i \% 10$  would map any integer in the range of 0-9 depending on the remainder you get on dividing 'i' by 10. If  $i = 23$ , then  $H = 3$ , since 23 divided by 10 would give you a remainder of 3.

## Collisions in Hash Tables

Let's understand a major limitation of hash tables that was overlooked in the previous segment, collision.

Let's say that you maintained a phone book with a hash function that takes the first character of each name to decide the hash key. What would happen if there were two names with the same first letter?

As discussed, this hash function maps an input domain which is infinitely long to an output range that lies in the range of natural numbers. So, applying the basic laws of mathematics, if the input from a large dataset is mapped to the output of a small dataset, collisions are inevitable. The following diagram illustrates an example of a collision:



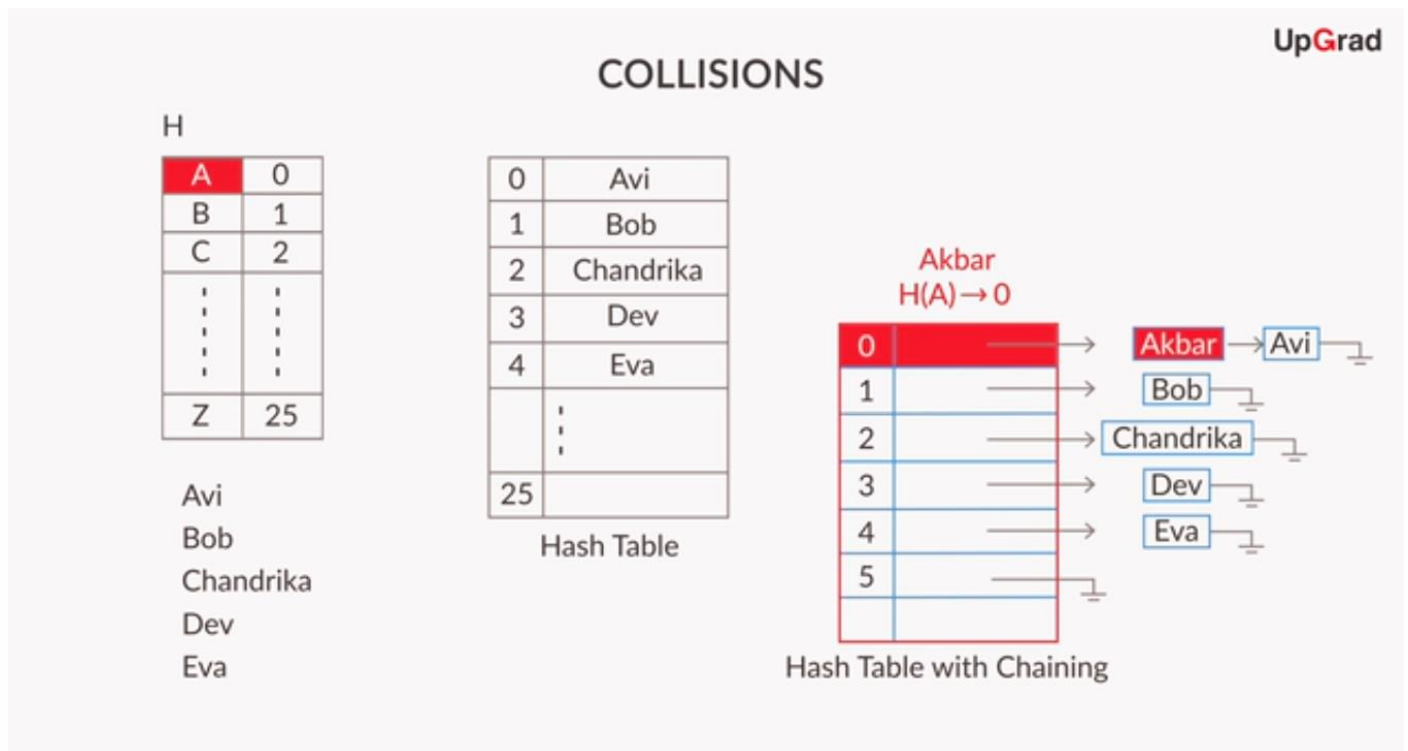
So, the main challenge that lies here is how we resolve this conflict. In other words, how do we come up with a solution to resolve this issue?

Let's say that you already have 'Avi' in the hash table and now another entry is made with the name 'Akash'. Since the first letter of both names is 'A', which means both get hashed to the same index, i.e. 0, now either we can store 'Akash' and delete 'Avi' from the table or the other way around.

But both of these ways are not what we desire because we want the data to be complete in every sense.

Thus, we come up with an appropriate way of handling collisions, i.e. Chaining. In this method, rather than storing a single entry at each index, there is a linked list of entries maintained at each index. So, every time a new entry with the same hash index comes into the picture, it gets added as the head of the linked list at that particular index.

In the figure below, we can see 'Akbar' getting added to the linked list at the 0<sup>th</sup> index where 'Avi' is already present, thus increasing the size of the list and also giving the opportunity to store multiple keys with the same hash index at the same position. The following illustration shows how the linked list:

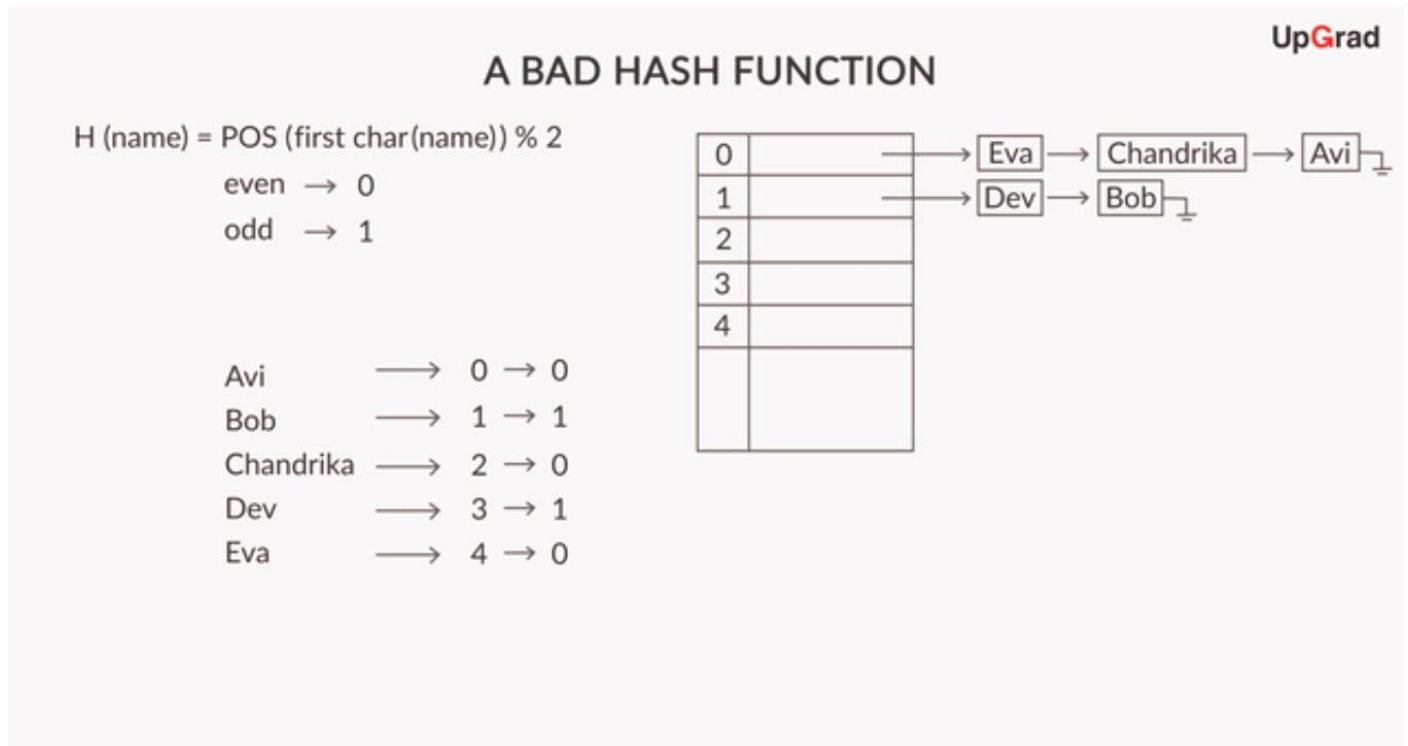


While chaining, the addition of a new entry into the hash table still is an  $O(1)$  operation because you are adding the element to the head of the list, whereas searching can take  $O(n)$  time when the list at an index grows proportional to the length of the hash table. Regardless of this, maintaining a hash table with chaining is a good way to eliminate collisions.

## How the Choice of a Hash Function Affects Hashing

Now that you know that the overall idea of hashing is based on the central concept of hash functions, you will now learn how the choice of a hash function affects the overall process of hashing.

So, for example, let's choose a hash function that is proven to be a bad choice, as shown in the illustration below:



As you can see, the hash function considers the position of the first letter of each name in the English alphabet and then considers whether it lies in an even position or odd position in the alphabet. If it's located in an even position, like 'A' comes at 0 (considering a 0-based index), then the names starting with A, C, E, etc. will get mapped to the hash index 0, whereas names starting with B, D, F, etc. will get mapped to index 1.

In this scenario, only two of the indices of the hash table get occupied and that too with both indices getting chained with long linked lists. This way of storing prevents us from calculating the constant run-time complexity, since we would have to traverse the entire list at a single index in the worst case.

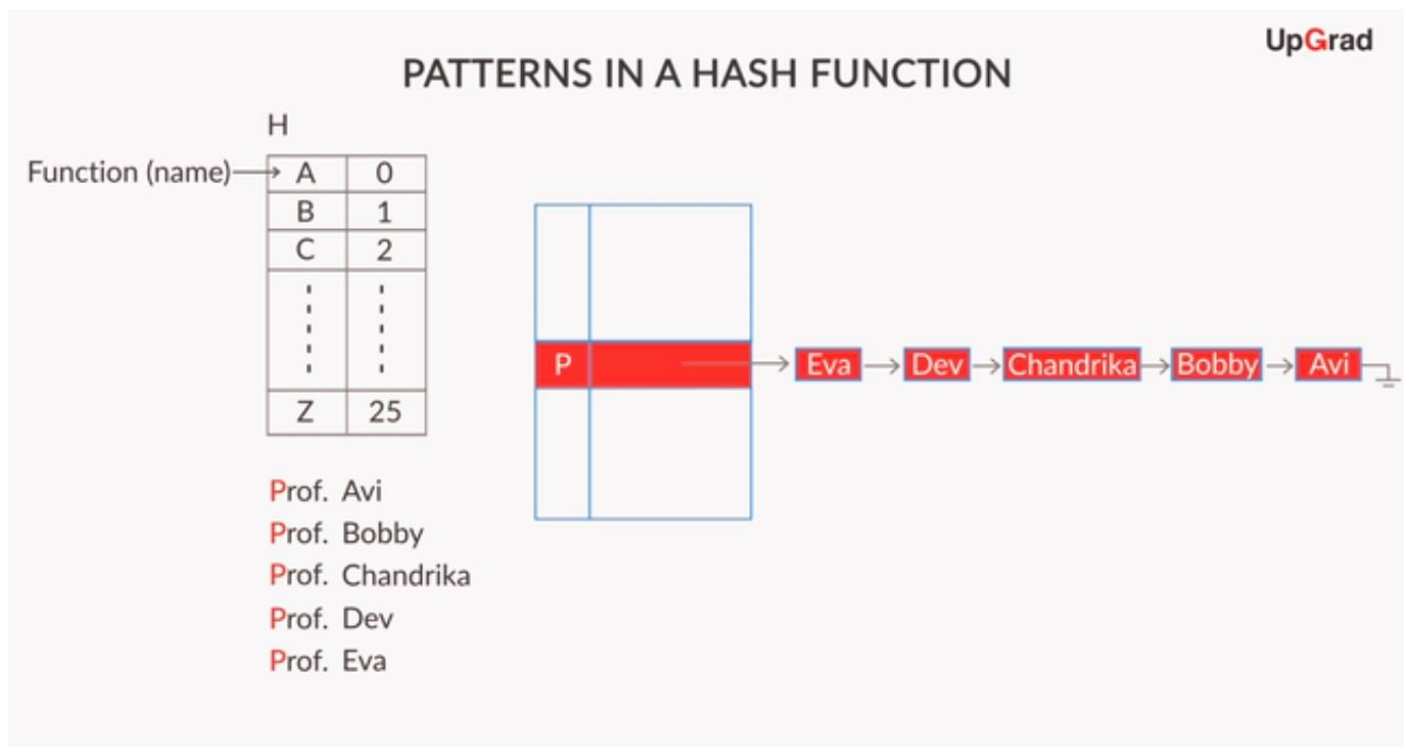
Besides, since only two of the indices are have taken up the load of the entire hash table, a lot of useful space is being wasted.

However, if we were careful in choosing the hash function, the scenario would have been quite different. For instance, we wouldn't be wasting so much space and could have avoided creating the long chains at only two of the indices of the hash table.

Next thing that comes up in hashing is patterns, which you need to avoid in order to ensure better performance of your hash functions.

Let's consider the case where all the names that appear in the phone book unexpectedly have the same first letter. Let's say that we are considering the case of a university, so the names are all saved with the title 'Prof.' ahead of them, e.g. 'Prof. Avi', 'Prof. Bob', and so on.

Now, all of the names get mapped to the same index since they all start with 'P'. In this scenario, a well-defined hash function, which worked well in some other cases, would fail as soon as it encounters a pattern in the input domain. Consider the image below:



What we see above is an example of a hash function that happens to work well in certain cases but doesn't produce desirable results when the input values have a pattern in them.

There is no single hash function that is universally applicable. For a given application, a good hash function should be designed with the following characteristics in mind:

1. It should use all the keys.
2. It should distribute the keys uniformly across the array indices.
3. It should output different hash values for similar, yet unequal, keys.

## Phonebook Implementation Using Hash Tables in Java

Now, you will see the hash table API provided by Java and implement a phone book using the functionalities provided by the hash table API.

You basically require two main functions called **put** and **get**, where put does the work of adding values to your hash table and get retrieves the results from the hash table using the key.

First, you must import the hash table to your class, which is done by simply writing 'import java.util.Hashtable;' where all the import statements go in the code. After that, you have to declare the hash table by writing —

```
Hashtable<int, string> name = new Hashtable<int, String>;
```

In the preceding code snippet, int is the key of the hash table, string is the value that is intended to be stored at that key, and name is the general name that would be assigned to your hash table.

As for the put function, you have to write 'name.put(key, value)', which hashes the value to the particular key specified in the function itself. And as for the get function, you simply write 'name.get(key)', which returns the value stored at the specified key. The code that we used in the demonstration is shown below:

```
package com.company;

import java.util.*;
import java.util.Hashtable;           //import the Hashtable API.
class Main{
    public static void main(String args[]){
        Hashtable<String,Integer> contacts=new Hashtable<String,Integer>(); //Hashtable API
        provided by Java (initialize).

        contacts.put("Ross",24434); //the put function adds value to the hash table.
        contacts.put("Rachel",24244);
        contacts.put("Chandler",12444);
        contacts.put("Monica",13144);

        //to check if any key is contained in the hash table.
        System.out.println(contacts.containsKey("Chandler")); //returns a bool value.

        //Let's search for some names in this phone book.
        System.out.println(contacts.get("Chandler")); //the get function gets the value of the
        key sent in the function.

        Set<String> keys = contacts.keySet(); //to get all the keys present in the hash table.
        for(String key: keys){
            System.out.println("Number of "+key+" is: "+contacts.get(key));
        }

        // to remove an entry from the hash table.
        contacts.remove("Chandler");

        System.out.println(contacts.containsKey("Chandler")); //to check if the key has been
        removed.

        contacts.clear(); //to clear the hashtable completely.
    }
}
```

There are other functions as well provided by the hash table API, such as `containsKey(key)`, which checks whether a particular key is contained in a hash table or not.

The difference between `containsKey()` and `get()` is that `containsKey()` returns a Boolean value whether the key remains in the hash table or not, whereas `get()` returns the value at that key.



Moreover, `Hashtable.keySet()` returns the set of keys that are contained in the hash table, and you can traverse the set of keys to get a list of all the keys contained in the hash table.

Also, `Hashtable.remove(key)` deletes the value entered with the key as given in the function from the hash table, thus clearing out the memory assigned to that key.

Lastly, `Hashtable.clear()` completely clears the hash table and leaves it empty.

## Summary

In this module, you learnt about Hashtable, a data structure used for implementing phone book systems. The main objective of a hash table is that it provides constant time for additions or retrievals, which is the main reason why it's so prevalent. Also, you learnt how choosing the wrong hash function can affect the performance of hashing. Next, you got insights on collisions, an integral part of hashing, and learnt how to resolve them using the method of chaining. Finally, you went through the hash table API provided by Java and its various functions.