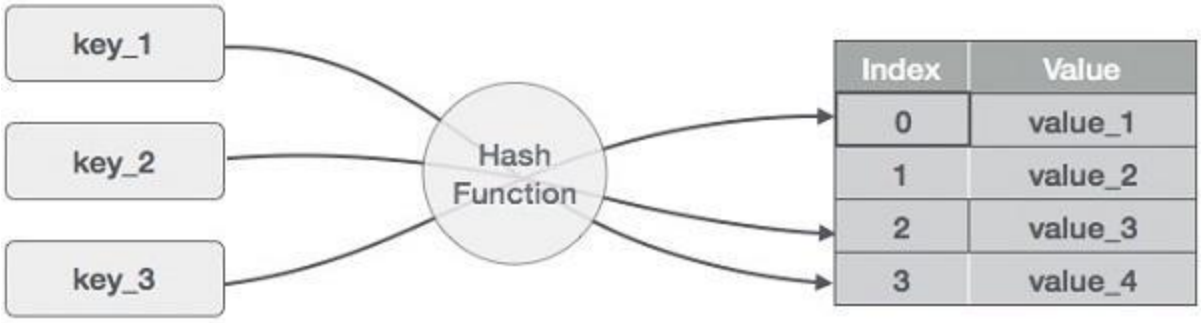
Hashing References

Please refer your text book for Hashing
Other references are

https://www.gatevidyalay.com/tag/linear-probing-example/
https://quescol.com/data-structure/linear-probing

Hashing

- It is a technique to convert a range of key values into a range of indexes of an array.
- Hashing is a technique to convert a range of key values into a range of indexes of an array.



- Hash Table:
  - Hash Table is a data structure which stores data in an associative manner.
  - In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
  - Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data.
  - Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from. Hash Table is
  -
- Hash Function
  - A fixed process converts a key to a hash key is known as a **Hash Function.**
  - This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
  - Hash value represents the original string of characters, but it is normally smaller than the original.

- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

What is Hashing?

- It is used to facilitate the next level searching method when compared with the linear or binary search. Hashing allows to update and retrieve any data entry in a constant time O(1). Constant time O(1) means the operation does not depend on the size of the data. Hashing is used with a database to enable items to be retrieved more quickly. It is used in the encryption and decryption of digital signatures.

  Example:

  hash table of size n=10

  Modulo function as hash function

  Item are in the (key,value) format.

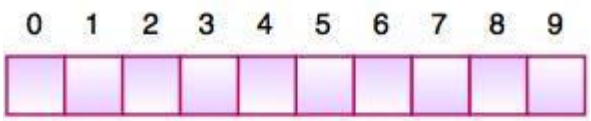


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10.
- Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on.
- Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {{26, Ajay}, {70, Reema}, {18, Paul], {31,Denish}, {54, Abdul}, {93, Bijal}}.
- One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3
36	36%10 = 6	6

Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure.
- In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **delete** − Deletes an element from a hash table.

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.

The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. **Less collisions:** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

   Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

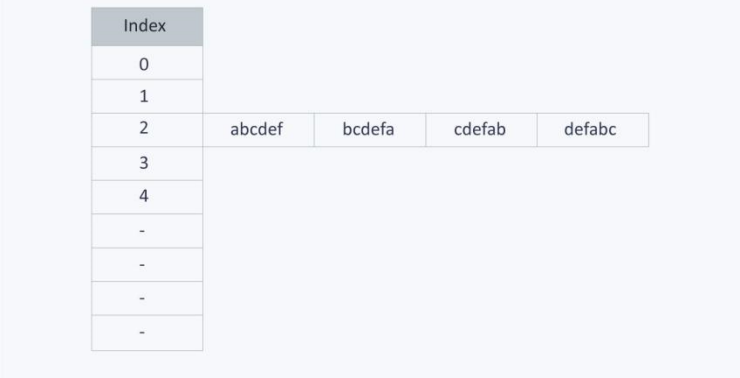
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



Here, it will take **O(n)** time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.

Let’s try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 **(prime number).**

Char * 10 + pos

String	Hash function	Index
abcdef	(97*1* + *98*2 + 99*3* + *100*4 + 101*5* + *102*6)%2069	38
bcdefa	(98*1* + *99*2 + 100*3* + *101*4 + 102*5* + *97*6)%2069	23
cdefab	(99*1* + *100*2 + 101*3* + *102*4 + 97*5* + *98*6)%2069	14
defabc	(100*1* + *101*2 + 102*3* + *97*4 + 98*5* + *99*6)%2069	11

Example: Sorting string

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Key : 123456879

Mid Square

M= 456

M^2 = 456 * 456

Folding

F1 = 12345

F2= 6879

F = F1+F2

Rehashing

Example: Counting frequency of character in the given string:

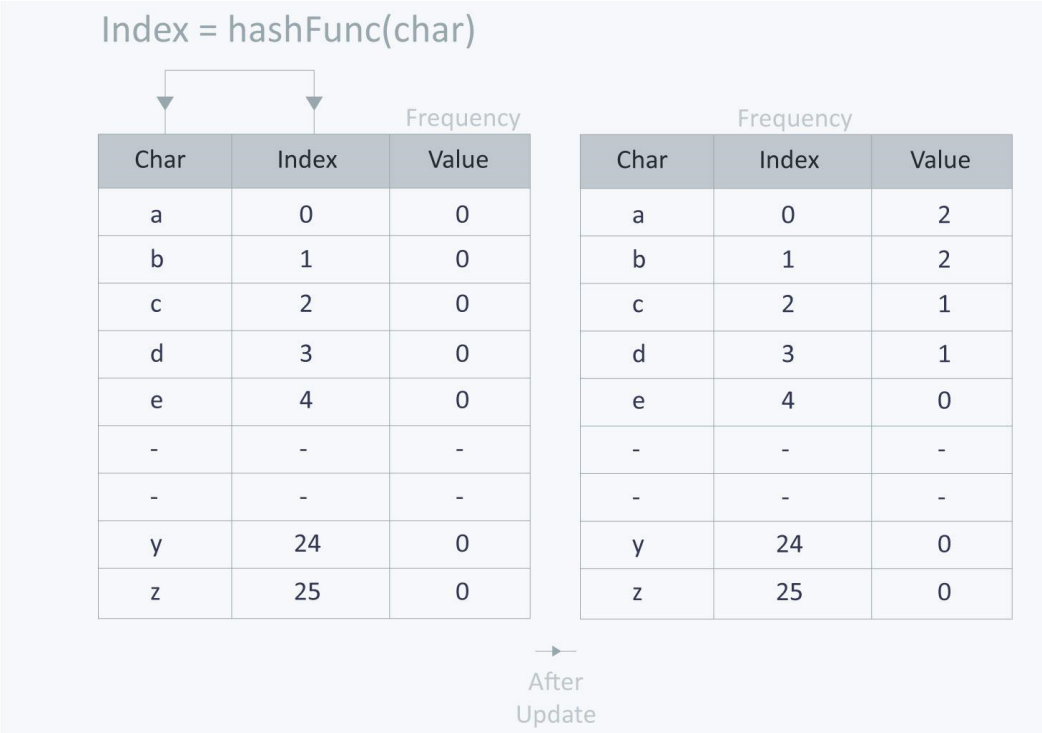
Logic1: Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is **O(N)** where **N** is the size of the string.

```
int Frequency[26];

  int hashFunc(char c)
  {
     return (c - 'a');
  }

  void countFre(string S)
  {
     for(int i = 0;i < S.length();++i)
     {
        int index = hashFunc(S[i]);
        Frequency[index]++;
     }
     for(int i = 0;i < 26;++i)
        cout << (char)(i+'a') << ' ' << Frequency[i] << endl;
  }
```

Output

```
a 2
b 2
c 1
d 1
e 0
f 0
...
```

```
z 0
```



S = “abaaacdef”

S[0] = s[0] – ‘a’ = 0

S[1] = 1

S[2] = 2

Collision resolution techniques

Collision:

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1, value	1: 1,Value
2	2	2 % 20 = 2	2, value	2: 2, Vaalue
3	42	42 % 20 = 2	2,Value	3: 42,Value
4	4	4 % 20 = 4	4	4: 4,Valaue
5	12	12 % 20 = 12	12	12:12,Valaue
6	14	14 % 20 = 14	14	14:14,Vlaue
7	17	17 % 20 = 17	17	17: 17,Value
8	13	13 % 20 = 13	13	13: 13,Value
9	37	37 % 20 = 17	17	18:37,value

Linear probing (open addressing or closed hashing)

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Hash = Mod 10

1 1%10= 1

12 = 2

13 = 3

11 11%10 =1

21 21%10 = 1

Primary Clustering

Secondary Clustering

0	21
0	**28**
0	13
0	11
0	20
0	15
0	16
0	17
0	18
0	19

Insert 28

28%10 = 8

Search 48

Delete 1

Search 11

11%10 = 1

Insert 21

21%10 =1

Search 31

31%10 =1

Not found

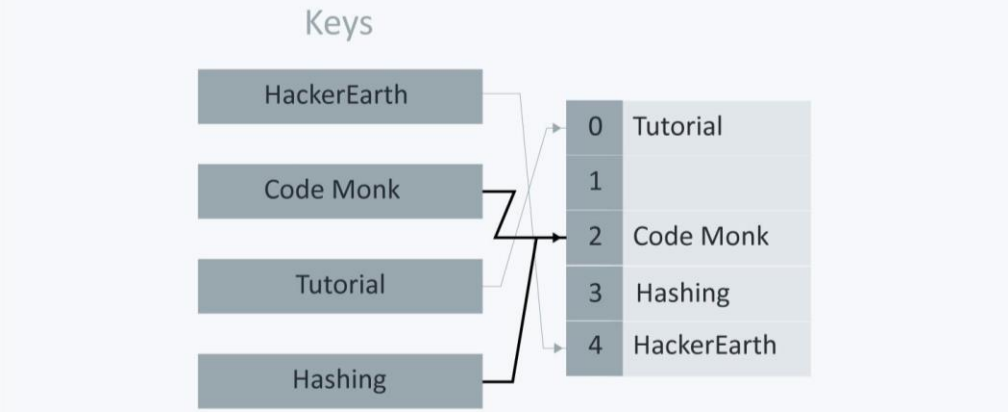
Logical Deletion/Physical Deletion

Linear search : O(n)

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be:

index = Key % hashTableSize
index = (Key + 1) % hashTableSize
index = (Key + 2) % hashTableSize
index = (Key + 3) % hashTableSize

and so on…



Hash collision is resolved by open addressing with linear probing.
Since **CodeMonk** and **Hashing** are hashed to the same index i.e. **2**, store **Hashing** at **3** as the interval between successive probes is **1**.

Quadratic Probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize

and so on…

Double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

and so on…

Here, **indexH** is the hash value that is computed by another hash function.

Linear probing technique explanation with example

Data Structure

- The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location.

- Here the idea is to place a value in the next available position. Because in this approach searches are performed sequentially so it's known as linear probing.
- Here array or hash table is considered circular because when the last slot reached an empty location not found then the search proceeds to the first location of the array.

Clustering is a major drawback of linear probing.

Applications

- *Associative arrays*: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- *Caches*: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- *Object representation*: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster

Associative Arrays and Hashing

The associative arrays are very similar to numeric arrays in term of functionality but they are different in terms of their index. Associative array will have their index as string so that you can establish a strong association between key and values.

To store the salaries of employees in an array, a numerically indexed array would not be the best choice. Instead, we could use the employees names as the keys in our associative array, and the value would be their respective salary.

NOTE − Don't keep associative array inside double quote while printing otherwise it would not return any value.

Example

```
<html>
   <body>

      <?php
         /* First method to associate create array. */
         $salaries = array("mohammad" => 2000, "qadir" => 1000,
"zara" => 500);

         echo "Salary of mohammad is ". $salaries['mohammad'] .
"<br />";
         echo "Salary of qadir is ".  $salaries['qadir']. "<br
/>";
         echo "Salary of zara is ".  $salaries['zara']. "<br
/>";

         /* Second method to create array. */
         $salaries['mohammad'] = "high";
         $salaries['qadir'] = "medium";
         $salaries['zara'] = "low";

         echo "Salary of mohammad is ". $salaries['mohammad'] .
"<br />";
         echo "Salary of qadir is ".  $salaries['qadir']. "<br
/>";
         echo "Salary of zara is ".  $salaries['zara']. "<br
/>";
      ?>

   </body>
</html>
```

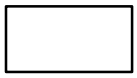
This will produce the following result −

```
Salary of mohammad is 2000
Salary of qadir is 1000
Salary of zara is 500
Salary of mohammad is high
Salary of qadir is medium
Salary of zara is low
```

Sequential Chaining

Index	Key Value	Chain (pointer)
1		
2		
3	10	100
4		NULL
5		
6		
7		

100



10%7 = 3
17 %7 = 3
Search 24 %7 = 3

Overflow Buket :