**What is Hashing?**

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value **x** at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



*Hashing Data Structure*

A **Hash Function**is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a “function” that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

* **Division Method.**
* **Mid Square Method.**
* **Folding Method.**
* **Multiplication Method.**
* **Division Method:**
* This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.
* **Formula:**
* ***h(K) = k mod M***
* *Here,****k****is the key value, and****M****is the size of the hash table.*
* *k = 12345  
  M = 95  
  h(12345) = 12345 mod 95   
                 = 90*
* *k = 1276  
  M = 11  
  h(1276) = 1276 mod 11   
               = 0*

**Pros:**

* This method is quite good for any value of M.
* The division method is very fast since it requires only a single division operation.

**Cons:**

* This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
* Sometimes extra care should be taken to choose the value of M.

**2. Mid Square Method:**

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

* Square the value of the key k i.e. k2
* Extract the middle **r** digits as the hash value.

**Formula:**

***h(K) = h(k x k)***

*Here,****k****is the key value.*

The value of **r**can be decided based on the size of the table.

**Example:**

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

*k = 60  
k x k = 60 x 60  
        = 3600  
h(60) = 60*

*The hash value obtained is 60*

**Pros:**

* The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
* The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

**Cons:**

* The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
* Another disadvantage is that there will be collisions but we can try to reduce collisions.

**3. Digit Folding Method:**

This method involves two steps:

* Divide the key-value **k**into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
* Add the individual parts. The hash value is obtained by ignoring the last carry if any.

**Formula:**

***k = k1, k2, k3, k4, ….., kn******s = k1+ k2 + k3 + k4 +….+ kn******h(K)= s***

*Here,****s****is obtained by adding the parts of the key****k***

**Example:**

*k = 12345  
k1 = 12, k2 = 34, k3 = 5  
s = k1 + k2 + k3  
  = 12 + 34 + 5  
  = 51   
h(K) = 51*

**4. Multiplication Method**

This method involves the following steps:

* Choose a constant value A such that 0 < A < 1.
* Multiply the key value with A.
* Extract the fractional part of kA.
* Multiply the result of the above step by the size of the hash table i.e. M.
* The resulting hash value is obtained by taking the floor of the result obtained in step 4.

**Formula:**

***h(K) = floor (M (kA mod 1))***

*Here,****M****is the size of the hash table.****k****is the key value.****A****is a constant value.*

**Example:**

*k = 12345  
A = 0.357840  
M = 100*

*h(12345) = floor[ 100 (12345\*0.357840 mod 1)]  
               = floor[ 100 (4417.5348 mod 1) ]  
               = floor[ 100 (0.5348) ]  
               = floor[ 53.48 ]  
               = 53*

**Pros:**

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

**Cons:**

The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

**Collisions in Hashing and Collision Resolution Techniques**

Collisions

Definition: A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.

Example Hash Table With Collisions:

Let’s take the exact same hash function from before: take the value to be hashed mod 10, and place it in that slot in the hash table.

Numbers to hash: 22, 9, 14, 17, 42

As before, the hash table is shown to the right.

As before, we hash each value as it appears in the string of values to hash, starting with the first value. The first four values can be entered into the hash table without any issues. It is the last value, 42, however, that causes a problem. 42 mod 10 = 2, but there is already a value in slot 2 of the hash table, namely 22. This is a collision.



The value 42 must end up in one of the hash table’s slots, but arbitrarly assigning it a slot at random would make accessing data in a hash table much more time consuming, as we obviously want to retain the constant time growth of accessing our hash table. There are two common ways to deal with collisions: chaining, and open addressing.

Collision Resolution Techniques

There are two types of collision resolution techniques.

Separate chaining (open hashing)

Open addressing (closed hashing)

------------------------------

Separate chaining: This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list. Separate chaining is the term used to describe how this connected list of slots resembles a chain. It is more frequently utilized when we are unsure of the number of keys to add or remove.

1. Open Hashing (Separate chaining)

Collisions are resolved using a list of elements to store objects with the same key together.

Suppose you wish to store a set of numbers = {0,1,2,4,5,7} into a hash table of size 5.

Now, assume that we have a hash function H, such that H(x) = x%5

So, if we were to map the given data with the given hash function we'll get the corresponding values

H(0)-> 0%5 = 0

H(1)-> 1%5 = 1

H(2)-> 2%5 = 2

H(4)-> 4%5 = 4

H(5)-> 5%5 = 0

H(7)-> 7%5 = 2

Clearly 0 and 5, as well as 2 and 7 will have the same hash value, and in this case we'll simply append the colliding values to a list being pointed by their hash keys.



Open addressing: To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table’s size is never equal to or less than the number of keys. Additionally known as closed hashing.

The following techniques are used in open addressing:

1) Linear probing

2) Quadratic probing

3) Double hashing

**Linear probing:** This involves doing a linear probe for the following slot when a collision occurs and continuing to do so until an empty slot is discovered.

The worst time to search for an element in linear probing is O. The cache performs best with linear probing, but clustering is a concern.

Linear Probing

The idea of linear probing is simple, we take a fixed sized hash table and every time we face a hash collision we linearly traverse the table in a cyclic manner to find the next empty slot.

Assume a scenario where we intend to store the following set of numbers = {0,1,2,4,5,7} into a hash table of size 5 with the help of the following hash function H, such that H(x) = x%5.

So, if we were to map the given data with the given hash function we'll get the corresponding values

H(0)-> 0%5 = 0

H(1)-> 1%5 = 1

H(2)-> 2%5 = 2

H(4)-> 4%5 = 4

H(5)-> 5%5 = 0



**Quadratic probing:** When a collision happens in this, we probe for the i2-nd slot in the ith iteration, continuing to do so until an empty slot is discovered. In comparison to linear probing, quadratic probing has a worse cache performance. Additionally, clustering is less of a concern with quadratic probing.

This method lies in the middle of great cache performance and the problem of clustering. The general idea remains the same, the only difference is that we look at the Q(i) increment at each iteration when looking for an empty bucket, where Q(i) is some quadratic expression of i. A simple expression of Q would be Q(i) = i^2, in which case the hash function looks something like this:

H(x, i) = (H(x) + i^2)%N

Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that H(x, i) = (x%5 + i^2)%5.



**Double hashing:** In this, you employ a different hashing algorithm, and in the ith iteration, you look for (i \* hash 2(x)). The determination of two hash functions requires more time. Although there is no clustering issue, the performance of the cache is relatively poor when using double probing.

Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that

H(x, i) = (H1(x) + i\*H2(x))%5

H1(x) = x%5 and H2(x) = P - (x%P), where P = 3

(3 is a prime smaller than 5)



Clearly 5 and 0 will face a collision, in which case we'll do the following:

- we look at 5%5 = 0 (collision)

- we look at (5%5 + 1\*(3 - (5%3)))%5 = 1 (collision)

- we look at (5%5 + 2\*(3 - (5%3)))%5 = 2 (collision)

- we look at (5%5 + 3\*(3 - (5%3)))%5 = 3 (empty -> place element here)