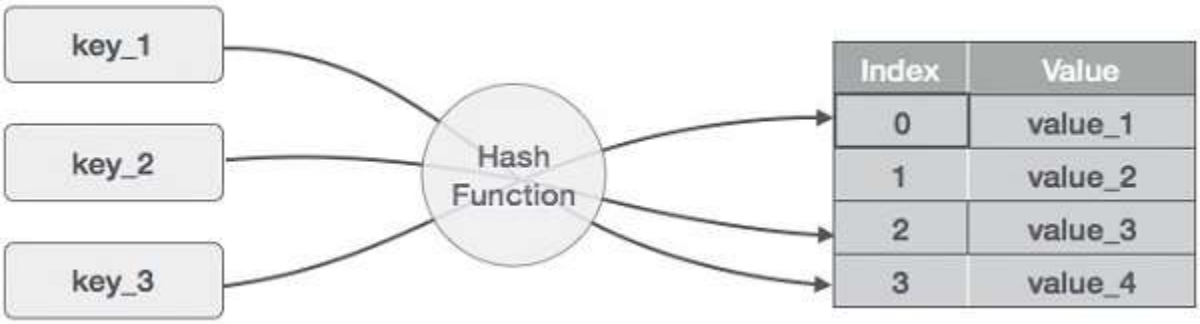
HASH TABLE:

It is a Data structure where the data elements are stored(inserted), searched, deleted based on the keys generated for each element, which is obtained from a hashing function. In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly implemented hash table would always promise an average insert/delete/retrieval time of *O(1).*

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



HASHING FUNCTION:

A function which employs some algorithm to computes the key *K* for all the data elements in the set *U,* such that the key *K* which is of a fixed size. The same key *K* can be used to map data to a hash table and all the operations like insertion,deletion and searching should be possible. The values returned by a **hash function** are also referred to as **hash** values, **hash** codes, **hash** sums, or **hashes**.

A good hash function should:

-be easy and quick to compute

-achieve an even distribution of the key values that actually occur across the index range supported by the table

-ideally be mathematically one-to-one on the set of relevant key values

Hash Function Techniques :

Division -the first order of business for a hash function is to compute aninteger value -if we expect the hash function to produce a valid index for our chosen table size, that integer will probably be out of range -that is easily remedied by moddingthe integer by the table size -there is some reason to believe that it is better if the table size is a prime, or at least has no small prime factors

Folding -portions of the key are often recombined, or foldedtogether -shift folding:123-45-6789 □123 + 456 + 789 -boundary folding:123-45-6789 □123 + 654 + 789 -can be efficiently performed using bitwise operations -the characters of a string can be xor'dtogether, but small numbers result -"chunks"of characters can be xor'dinstead, say in integer-sized chunks.

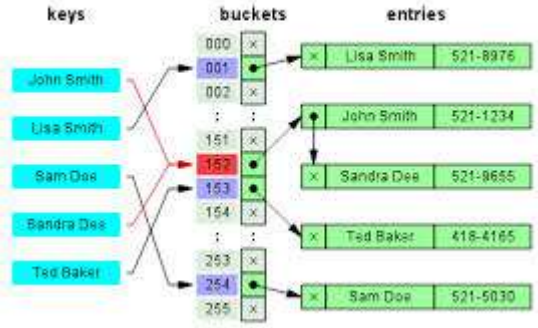
Mid-square function -square the key, then use the middle part as the result -e.g., 3121 □9740641 □406 (with a table size of 1000) -a string would first be transformed into a number, say by folding -idea is to let all of the key influence the result -if table size is a power of 2, this can be done efficiently at the bit level: 3121 □100101001010000101100001 □0101000010 (with a table size of 1024) Extraction -use only part of

the key to compute the result -motivation may be related to the distribution of the actual key values, e.g., VT student IDs almost all begin with 904, so it would contribute no useful separation.

HASH COLLISION RESOLUTION TECHNIQUES:

Open Hashing (Separate chaining)

Open Hashing, is a technique in which the data is not directly stored at the hash key index (*k)* of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.



In this technique when a data needs to be searched, it might become necessary (worst case) to traverse all the nodes in the linked list to retrieve the data.

Note that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

CLOSED HASHING (OPEN ADDRESSING)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

1. Liner Probing
2. Quadratic probing
3. Double hashing

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

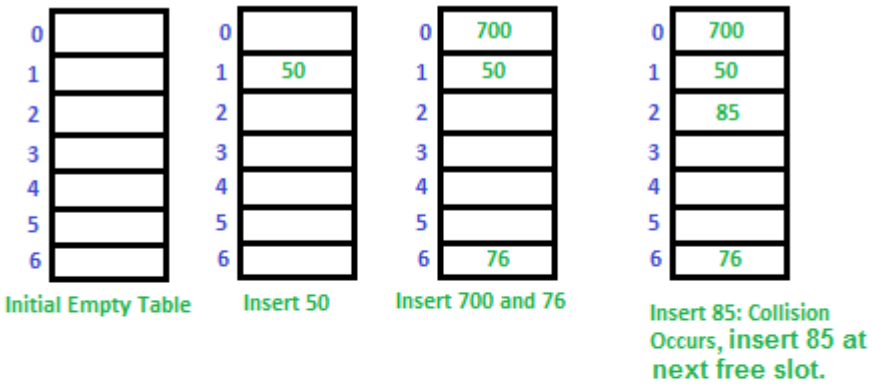
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

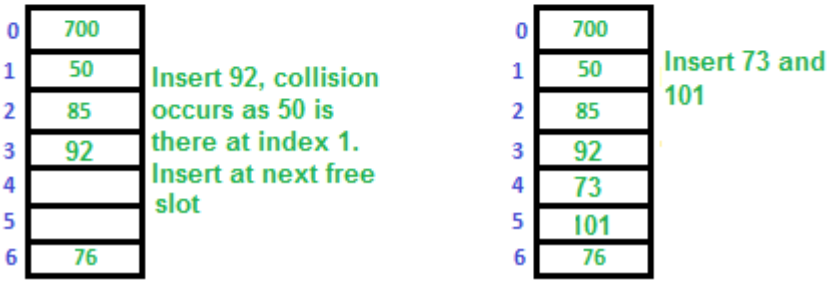
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

.....

.....

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.





Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) Quadratic Probing We look for i^2‘th slot in i’th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S

If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

.....

.....

c) <u>Double Hashing</u> Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

A comparative analysis of Closed Hashing vs Open Hashing

Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space