Basics of Hash Tables

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

- Assume that you have an object and you want to assign a key to it to make searching easy.
- To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.
- However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.
- In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**.
- The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key).
- By using that key you can access the element in **O(1)** time.
- Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

   hash = hashfunc(key)

   index = hash % array_size

- In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

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.
-

Hashing | (Separate Chaining)

Collision resolution techniques

- In separate chaining, each element of the **hash** table is a linked list.
- To store an element in the **hash** table you must insert it into a specific linked list. If there is any **collision** (i.e. two different elements have same **hash** value) then store both the elements in the same linked list.

What is Collision?

- Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value.
- The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

- Collisions are very likely even if we have big table to store keys.

How to handle Collisions?

There are mainly two methods to handle collision:

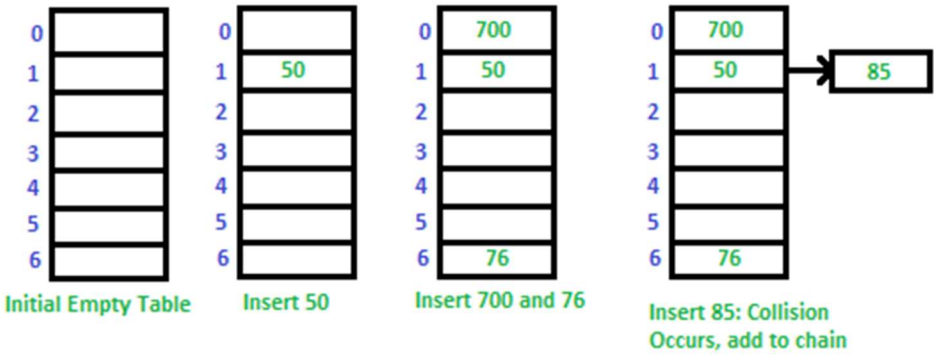
1) Separate Chaining

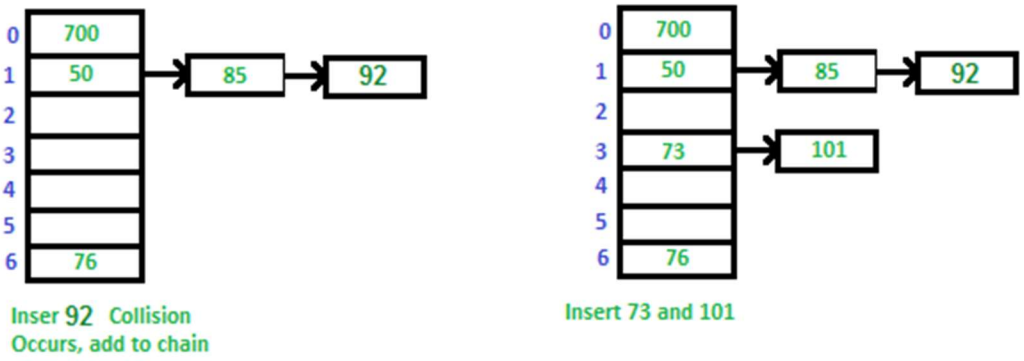
2) Open Addressing

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

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





Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in the worst case.

4) Uses extra space for links.

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table

Load factor $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$

Expected time to delete = $O(1 + \alpha)$

Time to insert = O(1)

Time complexity of search insert and delete is
$O(1)$ if α is $O(1)$

Hashing | (Open Addressing)

Open Addressing:

- Like separate chaining, open addressing is a method for handling collisions.
- In Open Addressing, all elements are stored in the hash table itself.
- So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

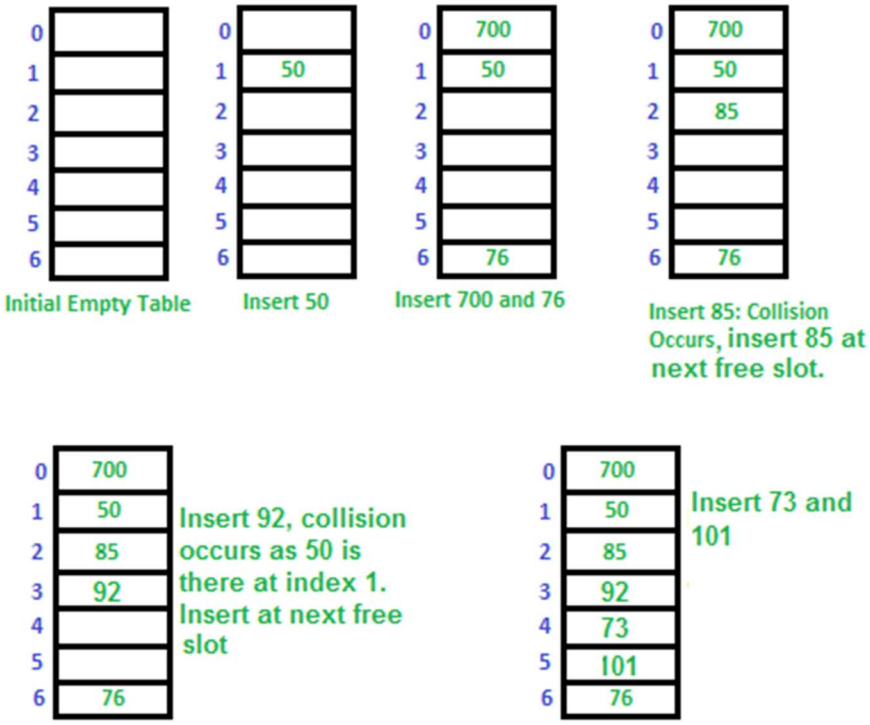
Open Addressing is done in the following ways:

a) Linear Probing:

- In linear probing, we linearly probe for next slot.
- For example, the typical gap between two probes is 1 as taken in below example also.
- Let **hash(x)** be the slot index computed using a 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.



Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.
2. **Secondary Clustering*:*** Secondary clustering is less severe, two records do only have the same collision chain (Probe Sequence) if their initial position is the same.

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) Double Hashing

We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

Let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

.....
.....

Comparison of above three:

- Linear probing has the best cache performance but suffers from clustering.
- One more advantage of Linear probing is easy to compute. Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering.
- Double hashing requires more computation time as two hash functions need to be computed.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table

Load factor ? = n/m (< 1)

Expected time to search/insert/delete < 1/(1 - ?)

So Search, Insert and Delete take (1/(1 - ?)) time