**Hashing:**

Hashing is a method of storing and retrieving data from a database efficiently, suppose that we want to design a system for storing employee records keyed using phone numbers. And we want the following queries to be performed efficiently:

* Insert a phone number and the corresponding information.
* Search a phone number and fetch the information.
* Delete a phone number and the related information.

We can think of using the following data structures to maintain information about different phone numbers.

* An array of phone numbers and records.
* A linked list of phone numbers and records.
* A balanced binary search tree with phone numbers as keys.
* A direct access table.

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(log n) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order. With a balanced binary search tree, we get a moderate search, insert, and delete time. All these operations can be guaranteed to be in O(log n) time. Another solution that one can think of is to use a direct access table where we make a big array and use phone numbers as indexes in the array. An entry in the array is NIL if the phone number is not present, else the array entry stores pointer to records corresponding to the phone number. Time complexity wise this solution is the best of all, we can do all operations in O(1) time. For example, to insert a phone number, we create a record with details of the given phone number, use the phone number as an index and store the pointer to the record created in the table. This solution has many practical limitations. The first problem with this solution is that the extra space required is huge. For example, if the phone number is of n digits, we need O(m \* 10n) space for the table where m is the size of a pointer to the record. Another problem is an integer in a programming language may not store n digits. Due to the above limitations, the Direct Access Table cannot always be used. Hashing is the solution that can be used in almost all such situations and performs extremely well as compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing, we

get O(1) search time on average (under reasonable assumptions) and O(n) in the worst case.

Hashing is an improvement over Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as an index in a table called a hash table.

**Hash Function**: A function that converts a given big phone number 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 big number or string to a small integer that can be used as an index in the hash table. A good hash function should have following properties:

It should be efficiently computable.

It should uniformly distribute the keys (Each table position be equally likely for each key).

For example, for phone numbers, a bad hash function is to take the first three digits. A better function will consider the last three digits. Please note that this may not be the best hash function. There may be better ways. Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry. Collision Handling: Since a hash function gets us a small number for a big key, 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. Following are the ways to handle collisions:

**Chaining**: The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.

**Open Addressing**: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine the table slots until the desired element is found or the element is not present in the table.

**Applications of Hashing**

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

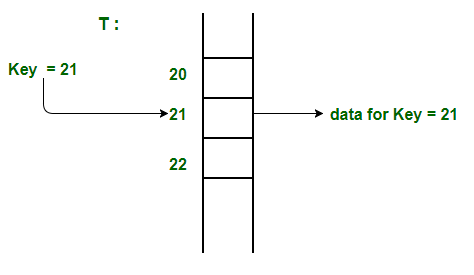
There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

* Message Digest
* Password Verification
* Data Structures (Programming Languages)
* Compiler Operation
* Rabin-Karp Algorithm
* Linking File name and path together
* Game Boards
* Graphics

**Direct Address Table**

Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. They facilitate fast searching, insertion and deletion operations.

We can understand the concept using the following example. We create an array of size equal to maximum value plus one (assuming 0 based index) and then use values as indexes. For example, in the following diagram key 21 is used directly as index.



**Advantages**:

* Searching in O(1) Time: Direct address tables use arrays which are random access data structure, so, the key values (which are also the index of the array) can be easily used to search the records in O(1) time.
* Insertion in O(1) Time: We can easily insert an element in an array in O(1) time. The same thing follows in a direct address table also.
* Deletion in O(1) Time: Deletion of an element takes O(1) time in an array. Similarly, to delete an element in a direct address table we need O(1) time.

**Limitations**:

* Prior knowledge of maximum key value
* Practically useful only if the maximum value is very less.
* It causes wastage of memory space if there is a significant difference between total records and maximum value.

Hashing can overcome these limitations of direct address tables.

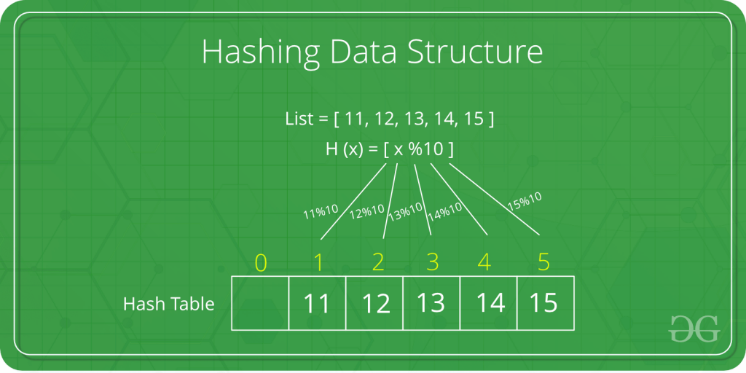
How to handle collisions?

Collisions can be handled like Hashing. We can either use Chaining or open addressing to handle collisions. The only difference from hashing here is, we do not use a hash function to find the index. We rather directly use values as indexes.

**Hashing Functions**

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.



The mod method:

In this method for creating hash functions, we map a key into one of the slots of table by taking the remainder of key divided by table\_size. That is, the hash function is

h(key) = key **mod** table\_size

i.e., key % table\_size

For Example

37599 % 17 = 12

But for **key = 573**, its hash function is also

573 % 17 = 12

**The multiplication method**:

* In multiplication method, we multiply the key k by a constant real number c in the range 0 < c < 1 and extract the fractional part of k \* c.
* Then we multiply this value by table\_size m and take the floor of the result. It can be represented as

**h(k) = floor (m \* (k \* c mod 1))**

**or**

**h(k) = floor (m \* frac (k \* c))**

**Collision Handling**

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

* Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple but requires additional memory outside the table.
* Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or the element is not in the table.

**Separate Chaining:**

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques to handle collisions.

The linked list data structure is used to implement this technique. So, what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain

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

**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). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

1. **Linear Probing**:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows:

**rehash(key) = (n+1)%table-size**

2. **Quadratic Probing**:

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above. This method is also known as the mid-square method. In this method, we look for the i2‘th slot in the ith iteration. We always start from the original hash location. If only the location is occupied, then we check the other slots.

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

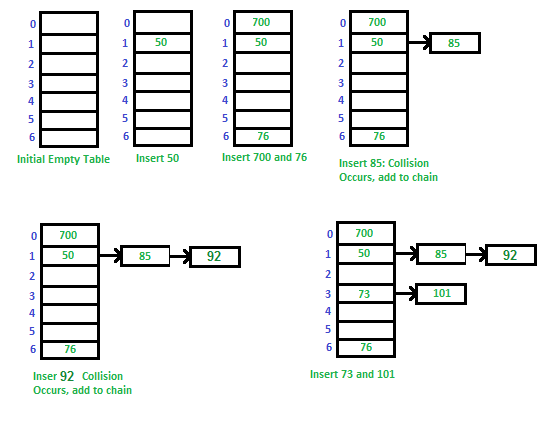
If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S

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

**Double Hashing**

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i\*hash2(x) slot in the ith rotation.



**Implementation of Chaining**

In hashing there is a hash function that maps keys to some values. But these hashing functions may lead to a collision that is two or more keys are mapped to same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let's create a hash function, such that our hash table has 'N' number of buckets.

To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

Example: hashIndex = key % noOfBuckets

Insert: Move to the bucket corresponding to the above-calculated hash index and insert the new node at the end of the list.

Delete: To delete a node from hash table, calculate the hash index for the key, move to the bucket corresponding to the calculated hash index, and search the list in the current bucket to find and remove the node with the given key (if found).

A screenshot of a computer

Description automatically generated

We use a list in C++ which is internally implemented as linked list (Faster insertion and deletion).

**Open Addressing**

Open Addressing: 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).

Important Operations:

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

Search(k): Keep probing until the 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 the deleted keys are marked specially as "deleted".

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:

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

**Quadratic Probing**: We look for i2'th slot in ith 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 .................................................. ..................................................

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 it suffers from clustering. One more advantage of Linear probing that it 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.

**Separate Chaining**

1. Chaining is Simpler to implement.

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

3. Chaining is Less sensitive to the hash function or load factors.

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

5. Cache performance of chaining is not good as keys are stored using linked list.

6. Wastage of Space (Some Parts of hash table in chaining are never used).

7. Chaining uses extra space for links.

**Open Addressing**

1. Open Addressing requires more computation.

2. In open addressing, table may become full.

3. Open addressing requires extra care for to avoid clustering and load factor.

4. Open addressing is used when the frequency and number of keys is known.

5. Open addressing provides better cache performance as everything is stored in the same table.

6. In Open addressing, a slot can be used even if an input doesn't map to it.

7. 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 α = n/m ( < 1 )

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

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