Introduction to 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:

1. Insert a phone number and the corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and the related information.

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

1. An array of phone numbers and records.
2. A linked list of phone numbers and records.
3. A balanced binary search tree with phone numbers as keys.
4. 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(Logn) 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 of these operations can be guaranteed to be in O(Logn) 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**](http://en.wikipedia.org/wiki/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:

1. It should be efficiently computable.
2. 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**](http://en.wikipedia.org/wiki/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 it is clear that the element is not present in the table

Hashing Application

**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](https://www.geeksforgeeks.org/print-distinct-elements-given-integer-array/), 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:

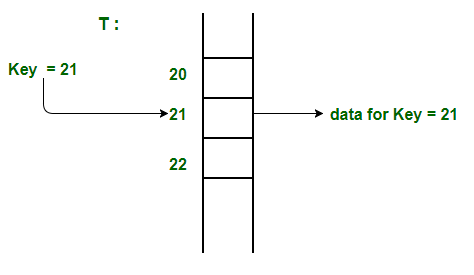
* Message Digest
* Password Verification
* Data Structures(Programming Languages)
* Compiler Operation
* Rabin-Karp Algorithm
* Linking File name and path together
* Game Boards
* Graphics

Direct Address Table

**Direct Address Table**

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:**

1. **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.
2. **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.
3. **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:**

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

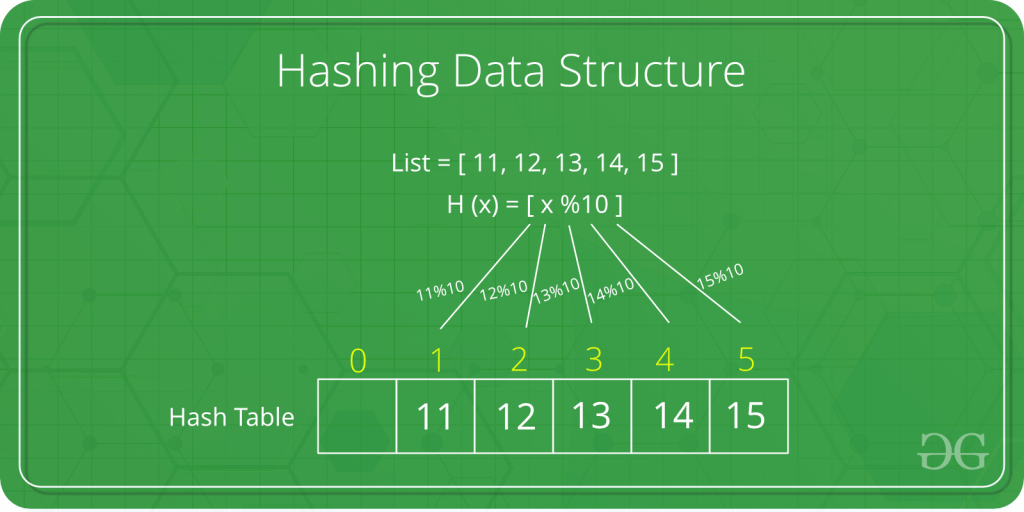
Hashing can overcome these limitations of direct address tables.

**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 valueat 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.



1. **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

1. **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:

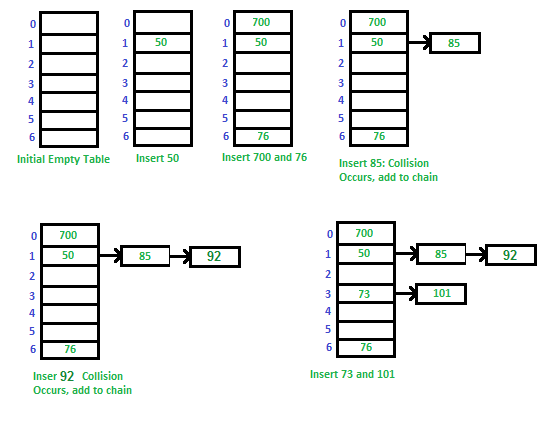
* **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
* **Open Addressing:**In open addressing, all elements are stored in the hash table itself. Each table 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 it is clear that 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 in order 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.

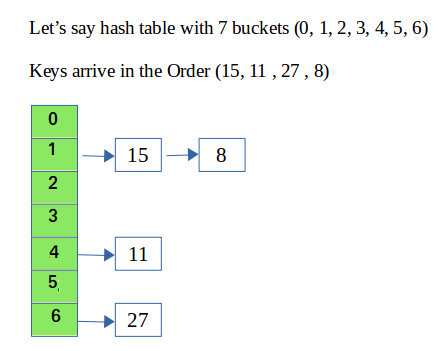
Implementation of Chaining

In [hashing](https://www.geeksforgeeks.org/hashing-data-structure/) 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).



Please refer [**Hashing | Set 2 (Separate Chaining)**](https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/)for details.  
We use a [list in C++](https://www.geeksforgeeks.org/list-cpp-stl/) which is internally implemented as linked list (Faster insertion and deletion).

**Method - 1 :**

This method has not concept of rehashing. It only has a fixed size array i.e. fixed numbers of buckets.

CPP

// CPP program to implement hashing with chaining

#include<bits/stdc++.h>

using namespace std;

class Hash

{

int BUCKET; // No. of buckets

// Pointer to an array containing buckets

list<int> \*table;

public:

Hash(int V); // Constructor

// inserts a key into hash table

void insertItem(int x);

// deletes a key from hash table

void deleteItem(int key);

// hash function to map values to key

int hashFunction(int x) {

return (x % BUCKET);

}

void displayHash();

};

Hash::Hash(int b)

{

this->BUCKET = b;

table = new list<int>[BUCKET];

}

void Hash::insertItem(int key)

{

int index = hashFunction(key);

table[index].push\_back(key);

}

void Hash::deleteItem(int key)

{

// get the hash index of key

int index = hashFunction(key);

// find the key in (index)th list

list <int> :: iterator i;

for (i = table[index].begin();

i != table[index].end(); i++) {

if (\*i == key)

break;

}

// if key is found in hash table, remove it

if (i != table[index].end())

table[index].erase(i);

}

// function to display hash table

void Hash::displayHash() {

for (int i = 0; i < BUCKET; i++) {

cout << i;

for (auto x : table[i])

cout << " --> " << x;

cout << endl;

}

}

// Driver program

int main()

{

// array that contains keys to be mapped

int a[] = {15, 11, 27, 8, 12};

int n = sizeof(a)/sizeof(a[0]);

// insert the keys into the hash table

Hash h(7); // 7 is count of buckets in

// hash table

for (int i = 0; i < n; i++)

h.insertItem(a[i]);

// delete 12 from hash table

h.deleteItem(12);

// display the Hash table

h.displayHash();

return 0;

}

**Output**

0

1 --> 15 --> 8

2

3

4 --> 11

5

6 --> 27

**Time Complexity:**

* **Search :**O(1+(n/m))
* **Delete :**O(1+(n/m))  
  where n =  Total elements in hash table  
               m = Size of hash table
* Here n/m is the **Load Factor**.
* Load Factor (∝) must be as small as possible.
* If load factor increases,then possibility of collision increases.
* Load factor is trade of space and time .
* Assume , uniform distribution of keys ,
* Expected chain length : O(∝)
* Expected time to search : O( 1 + ∝ )
* Expected time to insert/ delete :  O( 1 + ∝ )

**Auxiliary Space:**O(1), since no extra space has been taken.

**Method - 2 :**

Let's discuss another method where we have no boundation on number of buckets. Number of buckets will increase when value of load factor is greater than 0.5.

We will do [rehashing](https://www.geeksforgeeks.org/load-factor-and-rehashing/)when the value of load factor is greater than 0.5. In rehashing, we double the size of array and add all the values again to new array (doubled size array is new array) based on hash function. Hash function should also be change as it is depends on number of buckets. Therefore, hash function behaves differently from the previous one.

* Our Hash function is :**(ascii value of character \* some prime number ^ x) % total number of buckets.**In this case prime number is 31.
* Load Factor = number of elements in Hash Map / total number of buckets
* Our key should be string in this case.
* We can make our own Hash Function but it should be depended on the size of array because if we do rehashing then it must reflect changes and number of collisions should reduce.

C++

#include <iostream>

#define ll long long int

using namespace std;

// Linked List

template <typename T>

class node

{

public:

string key;

T value;

node \*next;

node(string key, T value) // constructor

{

this->key = key;

this->value = value;

this->next = NULL;

}

node(node &obj) // copy constructor

{

this->key = obj.key;

this->value = obj.value;

this->next = NULL;

}

~node() // destructor

{

node \*head = this;

while (head != NULL)

{

node \*currNode = head;

head = head->next;

delete currNode;

}

}

};

// hash table

template <typename T>

class unordered\_map

{

public:

int numOfElements, capacity;

node<T> \*\*arr; // want a array which stores pointers to node<T> i.e. head of a Linked List

unordered\_map() // constructor

{

this->capacity = 1;

this->numOfElements = 0;

this->arr = new node<T> \*[this->capacity];

this->arr[0] = NULL;

}

int hashFunction(string key) // hash function for hashing a string

{

int bucketIndex;

ll sum = 0, factor = 31;

for (int i = 0; i < key.size(); i++)

{

// sum = sum + (ascii value of character \* (prime number ^ x)) % total number of buckets

// factor = factor \* prime number i.e. prime number ^ x

sum = ((sum % this->capacity) + ((int(key[i])) \* factor) % this->capacity) % this->capacity;

factor = ((factor % INT16\_MAX) \* (31 % INT16\_MAX)) % INT16\_MAX;

}

bucketIndex = sum;

return bucketIndex;

}

float getLoadFactor()

{

// number of elements in hash table / total numbers of buckets

return (float)(this->numOfElements + 1) / (float)(this->capacity);

}

void rehashing()

{

int oldCapacity = this->capacity;

node<T> \*\*temp = this->arr; // temp is hodling current array

this->capacity = oldCapacity \* 2; // doubling the size of current capacity

this->arr = new node<T> \*[this->capacity]; // points to new array of doubled size

for (int i = 0; i < this->capacity; i++)

{

arr[i] = NULL;

}

for (int i = 0; i < oldCapacity; i++) // copying all the previous values in new array

{

node<T> \*currBucketHead = temp[i];

while (currBucketHead != NULL) // copying whole linked list

{

this->insert(currBucketHead->key, currBucketHead->value); // insert function have now updated hash function as capacity is doubled

currBucketHead = currBucketHead->next;

}

}

delete[] temp; // deleting old array from heap memory

return;

}

void insert(string key, T value)

{

while (this->getLoadFactor() > 0.5f) // when load factor > 0.5

{

this->rehashing();

}

int bucketIndex = this->hashFunction(key);

if (this->arr[bucketIndex] == NULL) // when there is no linked list at bucket

{

node<T> \*newNode = new node<T>(key, value);

arr[bucketIndex] = newNode;

}

else // adding at the head of current linked list

{

node<T> \*newNode = new node<T>(key, value);

newNode->next = this->arr[bucketIndex];

this->arr[bucketIndex] = newNode;

}

return;

}

int search(string key)

{

int bucketIndex = this->hashFunction(key); // getting bucket index

node<T> \*bucketHead = this->arr[bucketIndex];

while (bucketHead != NULL) // searching in the linked list which is present at bucket for given key

{

if (bucketHead->key == key)

{

return bucketHead->value;

}

bucketHead = bucketHead->next; // moving to next node in linked list

}

cout << "Oops!! Data not found." << endl; // when key is not matched...

return -1;

}

};

int main()

{

unordered\_map<int> mp; // int is value....in our case key must be of string type

mp.insert("Manish", 16);

mp.insert("Vartika", 14);

mp.insert("ITT", 5);

mp.insert("elite\_Programmer", 4);

mp.insert("pluto14", 14);

mp.insert("GeeksForGeeks", 11);

cout << "Value of GeeksForGeeks : " << mp.search("GeeksForGeeks") << endl;

cout << "Value of ITT : " << mp.search("ITT") << endl;

cout << "Value of Manish : " << mp.search("Manish") << endl;

cout << "Value of Vartika : " << mp.search("Vartika") << endl;

cout << "Value of elite\_Programmer : " << mp.search("elite\_Programmer") << endl;

cout << "Value of pluto14 : " << mp.search("pluto14") << endl;

// prints Oops!! Data not found and return -1

mp.search("GFG"); // case when there is no key present in Hash Map..

return 0;

}

**Output**

Value of GeeksForGeeks : 11

Value of ITT : 5

Value of Manish : 16

Value of Vartika : 14

Value of elite\_Programmer : 4

Value of pluto14 : 14

Oops!! Data not found.

**Complexity analysis of Insert:**

* **Time Complexity:** O(N), It takes O(N) time complexity because we are checking the load factor each time and when it is greater than 0.5 we call rehashing function which takes O(N) time.
* **Space Complexity**: O(N), It takes O(N) space complexity because we are creating a new array of doubled size and copying all the elements to the new array.

**Complexity analysis of Search:**

* **Time Complexity:** O(N),  It takes O(N) time complexity because we are searching in a linked list of size N.
* **Space Complexity:** O(1),  It takes O(1) space complexity because we are not using any extra space for searching.