# Introduction to Hashing

# 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:

- 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.

# Why hashing?

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, **O(n).**

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 * $10^n$) 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

**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).

**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.

# 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.

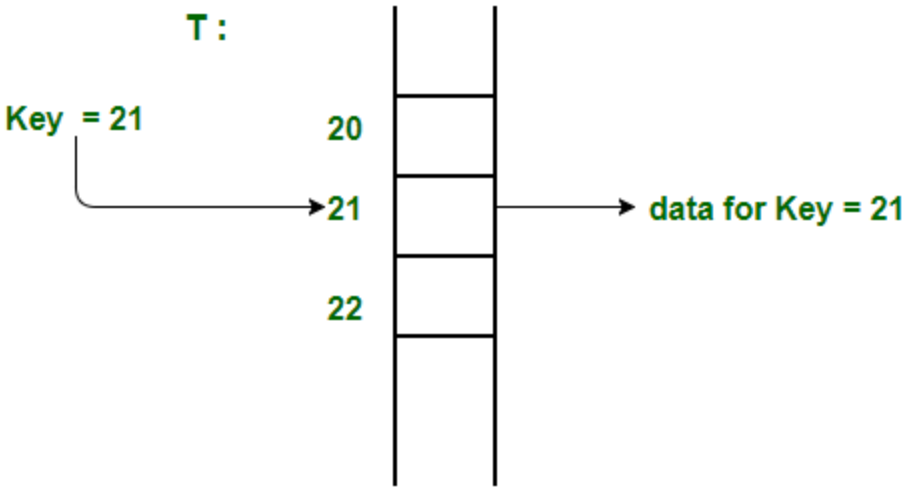
There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

- **Message Digest:** Creating a unique fingerprint of data to ensure its integrity and detect changes.
- **Password Verification**: Securely storing passwords as hashes instead of plain text.
- **Data Structures(Programming Languages):** Implementing hash tables for efficient lookups, insertion, and deletion of data.
- **Compiler Operation:** Using hashing techniques for symbol tables and faster code optimization.
- **Rabin-Karp Algorithm:** Efficient pattern searching within text.
- **Cryptography**: Hashing forms a basics of modern cryptography.
- **Caches**: Caches are temporary storage areas that speed up data access. Hashing helps caches work efficiently.

# 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.

```
int arr[]={10,21,40,5,7,8,20,25};
int key=21;
int dat[41];
for(int i=0; i<sizeOfarray; i++)
        dat[arr[i]]=arr[i];
```



**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:**

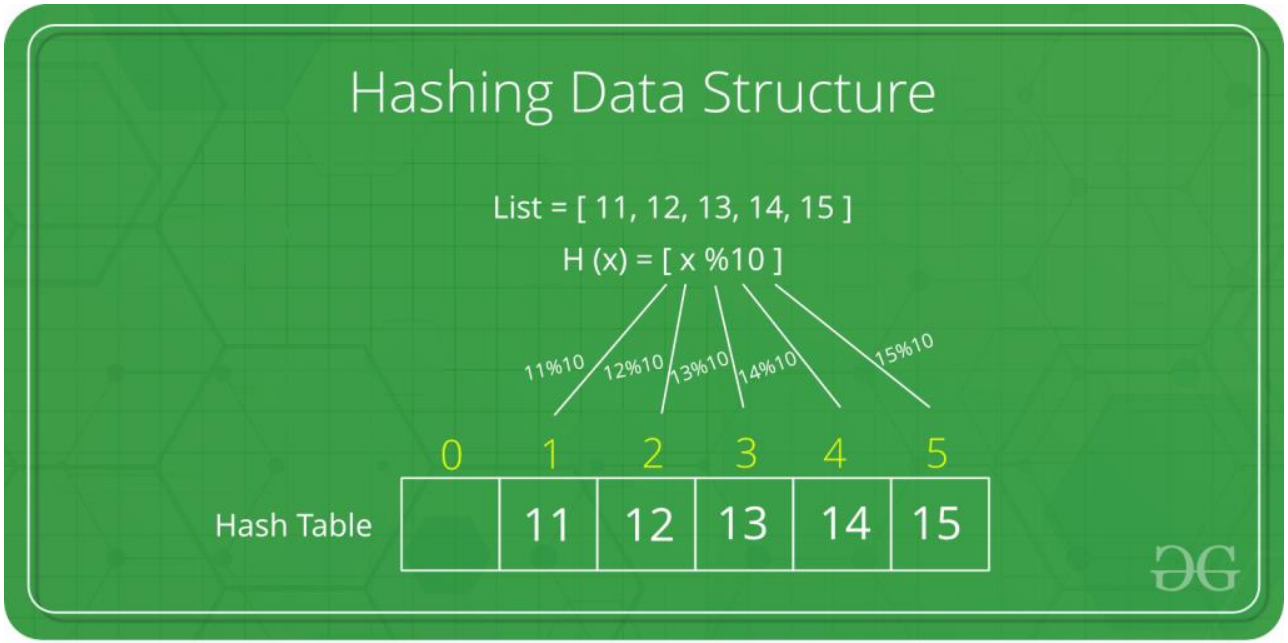
- Prior knowledge of maximum key value
- Practically useful only if the maximum value is very less.
- It causes wastage of memory space if there is a significant difference between total records and maximum value.
- Hashing can overcome these limitations of direct address tables.

**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))**

# Implementation of Chaining

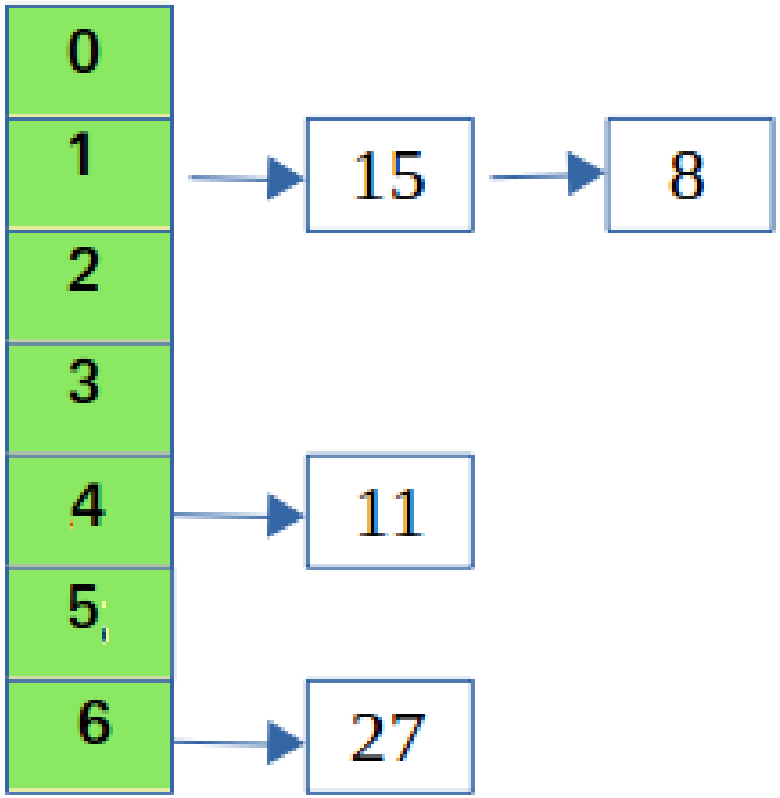
In hashing there is a hash function that maps keys to some values. But these hashing functions may lead to a collision that is two or more keys are mapped to same value. **Chain hashing** avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash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

# Implementation of Chaining

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



```
//program to implement hashing with chaining
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);
}
// 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;
      }
}
```

```
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);
}
```

```
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
```

# Complexity

**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 ($\propto$) 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 : $O(\propto)$
- Expected time to search : $O( 1 + \propto )$
- Expected time to insert/ delete : $O( 1 + \propto )$

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

# 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".

# Open Addressing ways

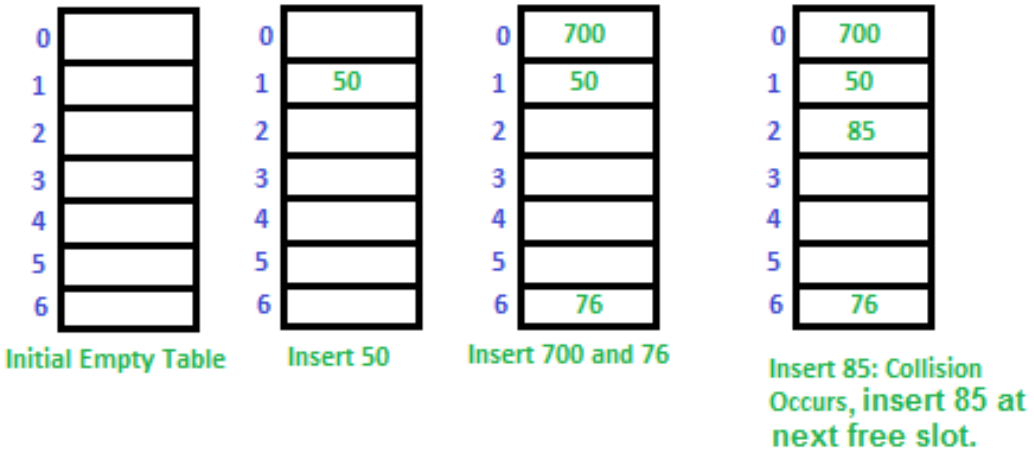
***Linear Probing:*** In linear probing, we linearly probe for the next slot.

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

# Open Addressing ways

***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
- .....

**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**:
  - poor cache performance but no clustering.
  - Double hashing requires more computation time as two hash functions need to be computed.

# Comparison

| S.No. | **Seperate 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 |

# 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 **$i^{th}$** rotation.

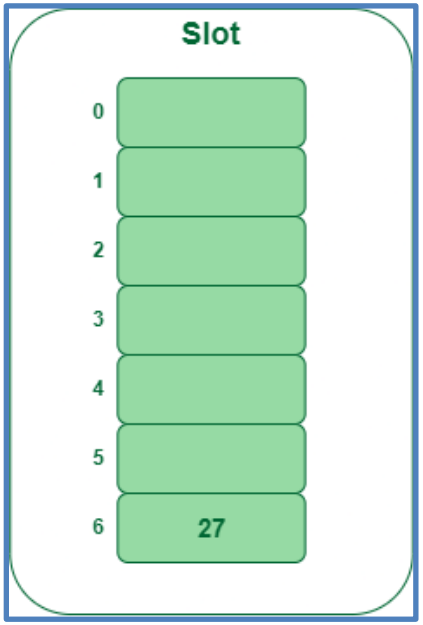
*let hash(x) be the slot index computed using hash function.*

- *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*
- *……..*

**Example:** Insert the keys 27, 43, 97, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

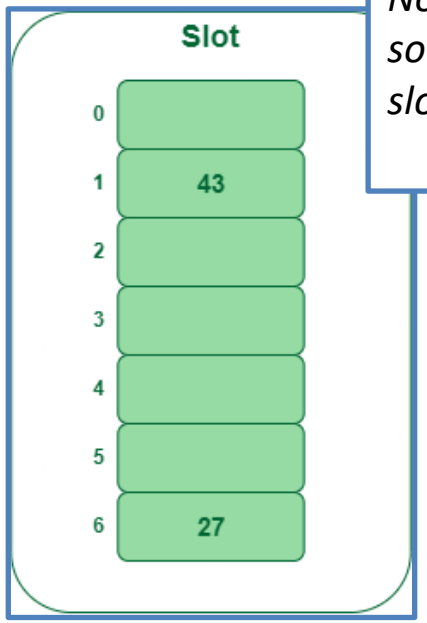
**Step 1:** Insert 27

- 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.




**Step 2:** Insert 43

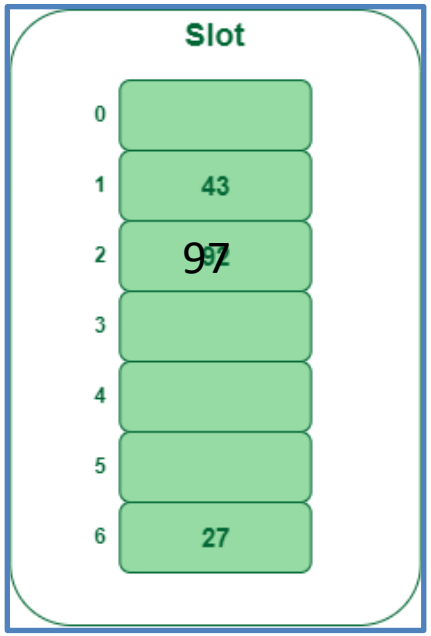
43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.




**Step 3:** Insert 97

97 % 7 = 6, but location 6 is already being occupied and this is a collision

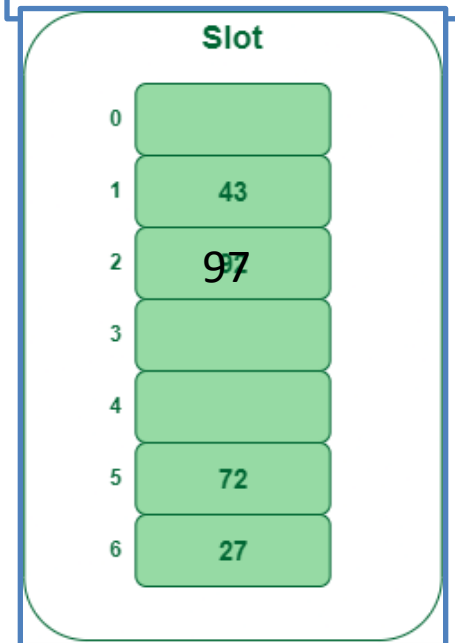
*$h_{new}$ = [h1(97) + i * (h2(97] % 7*
*= [6 + 1 * (1 + 97 % 5)] % 7*
*= 9 % 7 = 2*
*Now, as 2 is an empty slot, so we can insert 97 into 2nd slot.*




**Step 4:** Insert 72

72 % 7 = 2, but location 2 is already being occupied and this is a collision.

*$h_{new}$ = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
*= 5 % 7 = 5,*
*Now, as 5 is an empty slot, so we can insert 72 into 5th slot.*



# Performance of Open Addressing:

Cache performance of chaining is not good because when we traverse a Linked List, we are basically jumping from one node to another, all across the computer's memory.

For this reason, the CPU cannot cache the nodes which aren't visited yet, this doesn't help us.

But with Open Addressing, data isn't spread, so if the CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access.

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 $\alpha = n/m$ ( $< 1$ )*
- *Expected time to search/insert/delete $< 1/(1 - \alpha)$*
- *So Search, Insert and Delete take $(1/(1 - \alpha))$ time*

# Implementation of Open Addressing

The task is to design a general Hash Table data structure with Collision case handled and that supports the **Insert()**, **Find(),** and **Delete()** functions.

**Examples:**

- Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table:
- **Insert(1, 5**): Assign the pair {1, 5} at the index (1%20 =1) in the Hash Table.
- **Insert(2, 15**): Assign the pair {2, 15} at the index (2%20 =2) in the Hash Table.
- **Insert(3, 20**): Assign the pair {3, 20} at the index (3%20 =3) in the Hash Table.
- **Insert(4, 7**): Assign the pair {4, 7} at the index (4%20 =4) in the Hash Table.
- **Find(4):** The key 4 is stored at the index (4%20 = 4). Therefore, print the 7 as it is the value of the key, 4, at index 4 of the Hash Table.
- **Delete(4):** The key 4 is stored at the index (4%20 = 4). After deleting Key 4, the Hash Table has keys {1, 2, 3}.
- **Find(4):** Print -1, as the key 4 does not exist in the Hash Table.

**Approach:** The given problem can be solved by using the modulus Hash Function and using an array of structures as Hash Table, where each array element will store the **{key, value}** pair to be hashed. The collision case can be handled by Linear probing, open addressing. Follow the steps below to solve the problem:

- Define a node, structure say **HashNode,** to a key-value pair to be hashed.
- Initialize an array of the pointer of type **HashNode**, say ***arr[]** to store all key-value pairs.
- **Insert(Key, Value):** Insert the pair **{Key, Value}** in the Hash Table.
  - Initialize a **HashNode** variable, say **temp**, with value **{Key, Value}.**
  - Find the index where the key can be stored using the, Hash Function and then store the index in a variable say **HashIndex**.
  - If **arr[HashIndex]** is not empty or there exists another **Key**, then do linear probing by continuously updating the **HashIndex** as **HashIndex =(HashIndex+1)%capacity.**
  - If **arr[HashIndex]** is not null, then insert the given Node by assigning the address of **temp** to **arr[HashIndex].**

- **Find(Key):** Finds the value of the **Key** in the Hash Table.
  - Find the index where the **key** may exist using a Hash Function and then store the index in a variable, say **HashIndex**.
  - If the **arr[HashIndex]** contains the key, **Key** then returns the value of it.
  - Otherwise, do linear probing by continuously updating the **HashIndex** as **HashIndex =(HashIndex+1)%capacity.** Then**,** if **Key** is found, then return the value of the **Key** at that **HashIndex** and then return **true**.
  - If the **Key** is not found, then return **-1** representing not found. Otherwise, return the value of the **Key.**
- **Delete(Key):** Deletes the **Key** from the Hash Table.
  - Find the index where the **key** may exist using a Hash Function and then store the index in a variable, say **HashIndex**.
  - If the **arr[HashIndex**] contains the key, **Key** then delete by assigning **{-1, -1}** to the **arr[HashIndex]** and then return **true**.
  - Otherwise, do linear probing by continuously updating the **HashIndex** as **HashIndex =(HashIndex+1)%capacity.** Then**,** if **Key** is found then delete the value of the **Key** at that **HashIndex** and then return **true**.
  - If the **Key** is not found, then the return is false.

```cpp
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;

struct HashNode {
    int key;
    int value;
};

const int capacity = 20;
int size = 0;

struct HashNode** arr;
struct HashNode* dummy;
```

```cpp
void insert(int key, int V)
{

    struct HashNode* temp
        = (struct HashNode*)malloc(sizeof(struct HashNode));
    temp->key = key;
    temp->value = V;

    // Apply hash function to find
    // index for given key
    int hashIndex = key % capacity;

    // Find next free space
    while (arr[hashIndex] != NULL
           && arr[hashIndex]->key != key
           && arr[hashIndex]->key != -1) {
        hashIndex++;
        hashIndex %= capacity;
    }

    // If new node to be inserted
    // increase the current size
    if (arr[hashIndex] == NULL || arr[hashIndex]->key == -1)
        size++;

    arr[hashIndex] = temp;
}
```

```
// Function to delete a key value pair
int deleteKey(int key)
{
    // Apply hash function to find
    // index for given key
    int hashIndex = key % capacity;

    // Finding the node with given
    // key
    while (arr[hashIndex] != NULL) {
        // if node found
        if (arr[hashIndex]->key == key) {
            // Insert dummy node here
            // for further use
            arr[hashIndex] = dummy;

            // Reduce size
            size--;

            // Return the value of the key
            return 1;
        }
        hashIndex++;
        hashIndex %= capacity;
    }

    // If not found return null
    return 0;
}
```

```
// Function to search the value
// for a given key
int find(int key)
{
    // Apply hash function to find
    // index for given key
    int hashIndex = (key % capacity);

    int counter = 0;

    // Find the node with given key
    while (arr[hashIndex] != NULL) {

        int counter = 0;
        // If counter is greater than
        // capacity
        if (counter++ > capacity)
            break;

        // If node found return its
        // value
        if (arr[hashIndex]->key == key)
            return arr[hashIndex]->value;

        hashIndex++;
        hashIndex %= capacity;
    }

    // If not found return
    // -1
    return -1;
}
```

```
// Driver Code
int main()
{
    // Space allocation
    arr = (struct HashNode**)malloc(sizeof(struct HashNode*)
                                    * capacity);
    // Assign NULL initially
    for (int i = 0; i < capacity; i++)
        arr[i] = NULL;

    dummy
        = (struct HashNode*)malloc(sizeof(struct HashNode));

    dummy->key = -1;
    dummy->value = -1;

    insert(1, 5);
    insert(2, 15);
    insert(3, 20);
    insert(4, 7);
    if (find(4) != -1)
        cout << "Value of Key 4 = " << find(4) << endl;
    else
        cout << ("Key 4 does not exists\n");

    if (deleteKey(4))
        cout << ("Node value of key 4 is deleted "
                 "successfully\n");
    else {
        cout << ("Key does not exists\n");
    }

    if (find(4) != -1)
        cout << ("Value of Key 4 = %d\n", find(4));
    else
        cout << ("Key 4 does not exists\n");
}
```

**Output**

Value of Key 4 = 7
Node value of key 4 is deleted successfully
Key 4 does not exists

***Time Complexity:*** *O(capacity), for each operation*

***Auxiliary Space:*** *O(capacity)*

# Hashing in C++ using STL

Hashing in C++ can be implemented using different containers present in STL as per the requirement. Usually, STL offers the below-mentioned containers for implementing hashing:

- set
- unordered_set
- map
- unordered_map

# Set

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. **The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.**

Sets are used in the situation where it is needed to check if an element is present in a list or not. It can also be done with the help of arrays, but it would take up a lot of space. Sets can also be used to solve many problems related to sorting as the elements in the set are arranged in a sorted order.

Some basic functions associated with Set:

- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows last element in the set.
- **size()** – Returns the number of elements in the set.
- **insert(val)** – Inserts a new element *val* in the Set.
- **find(val)** - Returns an iterator pointing to the element *val* in the set if it is present.
- **empty()** – Returns whether the set is empty.

```
#include <iostream>
#include <set>
#include <iterator>
int main()
{
        set <int> s;                                    // empty set container
        int arr[] = {40, 20, 60, 30, 50, 50, 10};       // List of elements
        for(int i = 0; i < 7; i++)                      // Insert the elements of the List  to the set
                s.insert(arr[i]);
// Print the content of the set The elements of the set will be sorted without any duplicates
        cout<<"The elements in the set are: \n";
        for(auto itr=s.begin(); itr!=s.end(); itr++)
                cout<<*itr<<" ";
        if(s.find(50)!=s.end())                         // Check if 50 is present in the set
                cout<<"\n\n50 is present";
        else
                cout<<"\n\n50 is not present";
        return 0;
}
```

**Output**:
The elements in the set are: 10 20 30 40 50 60
50 is present

# unordered_set

The unordered_set container is implemented using a hash table where keys are hashed into indices of this hash table so it is not possible to maintain any order. All operation on unordered_set takes constant time O(1) on an average which can go up to linear time in the worst case which depends on the internally used hash function but practically they perform very well and generally provide constant time search operation.

The unordered-set can contain key of any type – predefined or user-defined data structure but when we define key of a user-defined type, we need to specify our comparison function according to which keys will be compared.

**set vs unordered_set**

- Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered.
- Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(Log n) while for unordered_set, it is O(1).

**Note**: Like set containers, the Unordered_set also allows only unique keys.

```
#include <iostream>
#include <unordered_set>
#include <iterator>
using namespace std;
int main()
{
    unordered_set <int> s;                  // empty set container
    int arr[] = {40, 20, 60, 30, 50, 50, 10}; // List of elements
    for(int i = 0; i < 7; i++)              // Insert the elements of the List to the set
        s.insert(arr[i]);
    // Print the content of the unordered set The elements of the set will not be sorted
    without any duplicates
    cout<<"The elements in the unordered_set are: \n";
    for(auto itr=s.begin(); itr!=s.end(); itr++)
        cout<<*itr<<" ";                    // Check if 50 is present in the set
    if(s.find(50)!=s.end())
        cout<<"\n\n50 is present";
    else
        cout<<"\n\n50 is not present";
     return 0;
}
```

**OUTPUT:**

```
The elements in the unordered_set are:
10 50 30 60 40 20

50 is present
```

# Map container

As a set, the Map container is also associative and stores elements in an **ordered** way but Maps **store elements in a mapped fashion**. Each element has a **key value and a mapped value**. No two mapped values can have the same key values.

Some basic functions associated with Map:

- **begin()** – Returns an iterator to the first element in the map.
- **end()** – Returns an iterator to the theoretical element that follows last element in the map.
- **size()** – Returns the number of elements in the map.
- **empty()** – Returns whether the map is empty.
- **insert({keyvalue, mapvalue})** – Adds a new element to the map.
- **erase(iterator position)** – Removes the element at the position pointed by the iterator
- **erase(const g)**– Removes the key value ‘g’ from the map.
- **clear()** – Removes all the elements from the map.

# C++ program to illustrate Map container

```
#include <iostream>
#include <iterator>
#include <map>
int main()
{
    map<int, int> mp;                    // empty map container
    // Insert elements in random order First element of the pair is the key
    second element of the pair is the value
    mp.insert(pair<int, int>(1, 40));
    mp.insert(pair<int, int>(2, 30));
    mp.insert(pair<int, int>(3, 60));
    mp.insert(pair<int, int>(4, 20));
    mp.insert(pair<int, int>(5, 50));
    mp.insert(pair<int, int>(6, 50));
    mp.insert(pair<int, int>(7, 10));
    map<int, int>::iterator itr;         // printing map mp
    cout << "The map mp is : \n";
    cout << "KEY\tELEMENT\n";
    for (itr = mp.begin(); itr != mp.end(); ++itr)
        cout << itr->first << '\t' << itr->second << '\n';
    return 0;
}
```

**Output**:

The map mp is :

| KEY | ELEMENT |
|---|---|
| 1 | 40 |
| 2 | 30 |
| 3 | 60 |
| 4 | 20 |
| 5 | 50 |
| 6 | 50 |
| 7 | 10 |

# unordered_map Container

The unordered_map is an associated container that stores elements formed by a combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.

Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of a hash table that is why the performance of data structure depends on hash function a lot but on an average, the cost of search, insert and delete from hash table is O(1).

# *C++ program to demonstrate functionality of unordered_map*

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main()
{
    // Declaring umap to be of <string, int> type key will be of string type and
    mapped value will be of double type
    unordered_map<string, int> umap;
    umap.insert({"GeeksforGeeks", 10});      // inserting values
    umap.insert({"Practice", 20});
    umap.insert({"Contribute", 30});
    cout << "The map umap is : \n";          // Traversing an unordered map
    cout << "KEY\t\tELEMENT\n";
    for (auto itr = umap.begin(); itr!= umap.end(); itr++)
        cout << itr->first << '\t' << itr->second << '\n';
    return 0;
}
```

**Output**:

```
The map umap is :
KEY             ELEMENT
Contribute      30
GeeksforGeeks   10
Practice        20
```