Hashmap

Introduction

There are numerous techniques for storing and accessing data in computer systems. Hashing utilizes an algorithm best suited for the users' needs and clubs similar pieces of data together or provides a unique id that helps in accessing the data stored in the system. Let us take a detailed look into it.

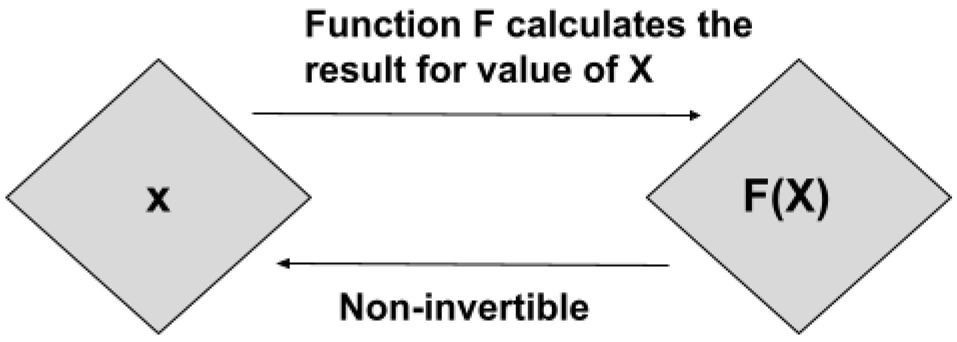
What is Hashing?

Hashing is the technique of modifying any given key and mapping its value into a hash table. This hash table stores the keys and the corresponding values, which can be directly accessed through the key.

The hash function is used to assign a value to a key, using which the element can be accessed in constant time(*O(1)*). So, operations like searching, insertion, and deletion become much easier and faster.

A good hash function uses a *one-way* hashing algorithm or *message-digest,* which makes it challenging to generate the original string from the hash function.

Hashing plays quite an important role in cryptography, data encryption, password authentication, and cyber security.



Collisions in Hashing

When the hash value of a key corresponds to an already occupied bucket in the hash table, it is referred to as a collision. It occurs when more than one value added to a particular hash table occupies the same slot in the table generated by the hash function.

For example, let us consider a hash table with size 10, where the following items are to be stored:*12, 24, 56, 30, 62*.
Let us assume that the hash function is given by: H1(K) = K % 10
We take the numbers one by one, apply the hash function to it, and the result tells us the index that the numbers will occupy in the hash table.
Applying the hash function, to the given numbers,

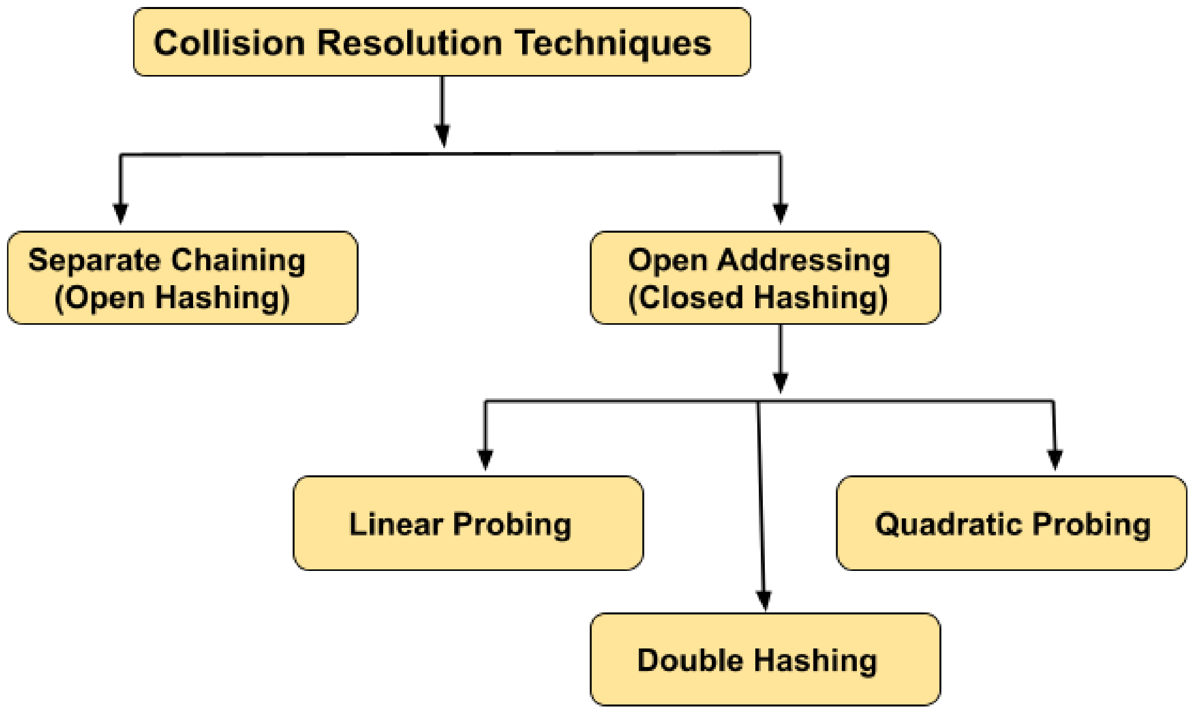
H1(12) = 12 % 10 = 2
H1(24) = 24 % 10 = 4
H1(56) = 56 % 10 = 6
H1(30) = 30 % 10 = 0
H1(62) = 62 % 10 = 2 ⟵ Collision Occurs

Index	Value
0	30
1	
2	12,62
3	
4	24
5	
6	56
7	
8	
9	

Thus, a collision occurs at the **2nd** index, where both the values produce the same value of **2** after passing through the hash function.

Thus, we need techniques to resolve a collision. Some of the main collision resolution techniques have been discussed below.

Collision Resolution Techniques


Collision Resolution Techniques
Separate Chaining (Open Hashing)
Open Addressing (Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

Let us briefly discuss Separate Chaining and Open Addressing before diving into Double Hashing to make the concept more precise.

Separate Chaining

The Hashing technique, which uses a linked list to handle collisions, is termed as separate chaining. Different elements at the same position in the hash table are stored in a linked list.

Open Addressing

Open addressing uses the concept of probing, a strategy used to handle a collision, and the type of probing depends on the type of hash table used. Unlike separate

chaining, no other additional data structure is required. Each slot, considered a hash bucket, maintains three states: *Empty*, *Occupied*, and *Deleted.*
Open addressing further consists of **3** types of methods of probing:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

Linear Probing is a simple collision resolution technique in Open Addressing. In case of a collision, it probes through the hash table linearly, looking for an empty slot, and assigns the free slot to the value. It is pretty easy to implement and occupies significantly less memory space.
However, it suffers from the problems of primary clustering, i.e., large blocks of cells in the hash table are continuously occupied, which makes the probe sequence quite lengthy.

Quadratic Probing

In quadratic probing, when a collision occurs, we probe the hash table for the $\boldsymbol{i^2}$ th bucket in the ***i*** th iteration, and we keep iterating till a suitable empty slot is found.
It suffers from the problem of secondary clustering, which means it may take a much longer time before insertion is made into the table.
So there was a need for a technique that could avoid both primary and secondary clustering, which is when double hashing came in.

Double Hashing

As discussed above, it is a collision-resolving technique in Open Addressing. It deploys the idea of using a 2nd hash function in case of a collision.

It uses the hash value generated by the first hash function as the starting point. In case of a collision, the second hash function, which is independent of the original function, determines the final location of the next value.

The following function denotes double hashing:

(H1(K) + i * H2(K)) % tableSize

Where ***H1(K)*** and ***H2(K)*** are the **2** hash functions and **tableSize** is the size of the hash table.

A typical first hash function is given by:

H1(K) = K % tableSize

A typical second hash function is given by:

H2(K) = cp - (K % cp)

Where **cp** is a number that must be smaller than the size of the hash table and should be coprime with the size of the table.

Example:

Let us understand double hashing using an example:

Keys, K = {**20**, **34**, **45**, **70**, **56**}

Let the size of the table be *11*. So the first hash function will be given by:

H1(K) = K % 11

For the second hash function, **cp** should be chosen in such a way that HCF(**cp**, 11) = 1

So, *let cp = 8*

The second hash function will be given by:

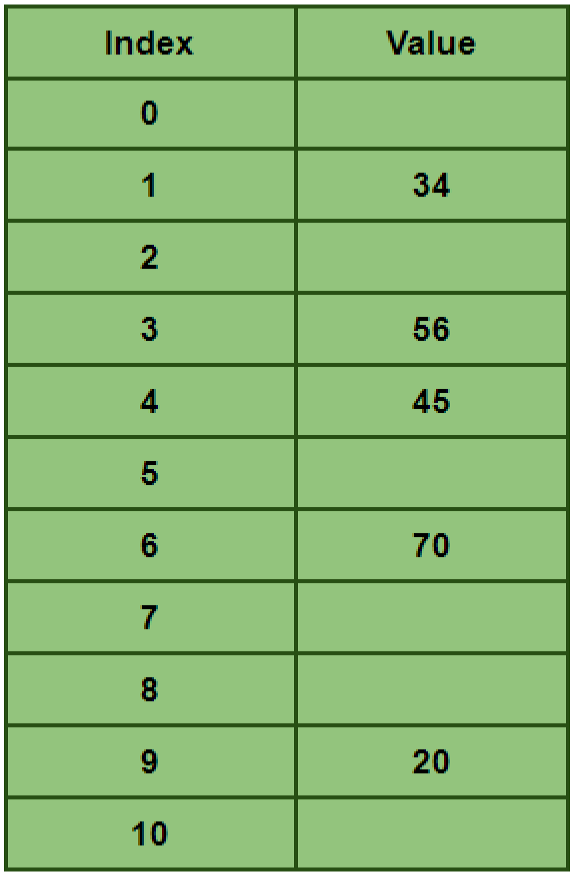
H2(K) = 8 - (K % 8)

And the hash function will be given by:

(H1(K) + i * H2(K)) % 11

Key	Hash Function	Index	Collision if any	New Index
20	H1(20) = 20 % 11 = 9	9	No	-
34	H1(34) = 34 % 11 = 1	1	No	-
45	H1(45) = 45 % 11 = 1 H2(45) = 8 - (45 % 8) = 3 (1 + 1 * 3) % 11 = 4	1 4	Yes	4
70	H1(70) = 70 % 11 = 4 H2(70) = 8 - (70 % 8) = 2 (4 + 1 * 2) % 11 = 6	4 6	Yes	6
56	H1(56) = 56 % 11 = 1 H2(70) = 8 - (56 % 8) = 8 (1 + 1 * 8) % 11 = 9 (1+ 2 * 8) % 11 = 6 (1+ 3 * 8) % 11 = 3	1 9 6 3	Yes Yes Yes	3

Thus, the hash table becomes:



Index	Value
0	
1	34
2	
3	56
4	45
5	
6	70
7	
8	
9	20
10	

Comparisons

Let us draw a comparison chart for the **3** techniques of Open Addressing:

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	YES	NO	NO
Secondary Clustering	YES	YES	NO
Cache performance	BEST	INTERMEDIATE	POOR

Implementation of Double Hashing

```
// Implementation of double hashing.
#include <bits/stdc++.h>
using namespace std;

// Predefining the size of the hash table.
#define SIZE_OF_TABLE 11

// Used for the second hash table.
// 8 and 11 are coprime numbers.
#define CO_PRIME 8

// Defining the hashTable vector.
vector<int> hashTable(SIZE_OF_TABLE);

class DoubleHash
{
    int size;

public:
    // To check whether the table is full or not.
    bool isFull()
    {
        // In case the table becomes full.
            return (size == SIZE_OF_TABLE);
    }

    // Calculating the first hash.
```

```
int hash1(int key)
{
        return (key % SIZE_OF_TABLE);
}

// Calculating the second hash.
int hash2(int key)
{
        return (CO_PRIME - (key % CO_PRIME));
}

DoubleHash()
{
        size = 0;
        for (int i = 0; i < SIZE_OF_TABLE; i++)
        hashTable[i] = -1;
}

// Function for inserting a key into the hash table.
void insertHash(int key)
{
    // We first check whether the hash is full or not.
        if (isFull())
        return;

    // Obtaining the index from the first hash table.
        int index = hash1(key);

    // In case a collision occurs.
        if (hashTable[index] != -1)
    {
       // Obtaining the index from second hash table.
        int index2 = hash2(key);
        int i = 1;
        while (1)
       {
          // Obtaining the new index.
        int newIndex = (index + i * index2) % SIZE_OF_TABLE;

          //If no collision occurs, the key is stored.
        if (hashTable[newIndex] == -1)
          {
                    hashTable[newIndex] = key;
                    break;
```

```
            }
          i++;
         }
      }

      //If no collision occurs, the key is stored.
          else
          hashTable[index] = key;
          size++;
  }

  // For searching a key in the hash table.
  void search(int key)
  {
          int i1 = hash1(key);
          int i2 = hash2(key);
          int i = 0;
          while (hashTable[(i1 + i * i2) % SIZE_OF_TABLE] != key)
      {
          if (hashTable[(i1 + i * i2) % SIZE_OF_TABLE] == -1)
         {
          cout << key << " does not exist" << endl;
          return;
         }
          i++;
      }
          cout << key << " found" << endl;
  }

  // For displaying the complete hash table.
  void displayHash()
  {
          for (int i = 0; i < SIZE_OF_TABLE; i++)
      {
          if (hashTable[i] != -1)
          cout << i << " --> "
                      << hashTable[i] << endl;
          else
          cout << i << endl;
      }
  }
};

// Driver's code.
```

```
int main()
{
    // Assuming the number of initial values to be 5.
    int n = 5, i, k;
    int a[100];

    // We first need to insert keys into the table.
    DoubleHash hash;
    cout << "Enter 5 values: \n";

    for (int i = 0; i < n; i++)
    {
            cin >> a[i];
            hash.insertHash(a[i]);
    }

    // First, we search for a key that is present in the table.
    cout << "Enter a key that you want to search \n";
    cin >> k;
    hash.search(k);

    // Lets display the hash table.
    // Since we took the SIZE_OF_TABLE as 11 we will get the indices from 0-10.
    cout << "\n The hash table looks like:\n";
    hash.displayHash();
    return 0;
}
```

Input:

```
Enter 5 values:
20
34
45
70
56
Enter a key that you want to search:
45
```

Output:

```
45 found
The hash table looks like:
0
1 --> 34
2
3 --> 56
4 --> 45
5
6 --> 70
7
8
9 --> 20
10
```

Complexity Analysis

Time Complexity

- Best Case: In the best case, no collision would occur and thus, the time complexity will be given by **O(1).**
- Average Case: The average case time complexity is also given by **O(1).** However, its proof is too complex to be discussed here.
- Worst Case: In the worst case, we have to probe over all **N** elements; thus the time complexity is given by **O(N).**

Space Complexity

Hash tables usually have a maximum capacity of some constant multiplied by **N** (the constant is predetermined and **N** is the total number of elements.)

Thus the space complexity of double hashing is **O(N).**

Advantages of Double Hashing

- No extra space is required.

- Primary Clustering does not occur.
- Secondary Clustering also does not occur.

Disadvantages of Double Hashing

- Poor cache performance.
- It's a little complicated because it uses two hash functions.
- Harder Implementation.