HASHING:

Hashing is an efficient data structure that was improved from the DIRECT ADDRESSING concept. This hashing concept overcomes almost all the disadvantages of the DA such as

- The need for a larger memory
- Incompatibility with the data.
- Wastage of memory space.

Hashing works as key-value pairs to successfully store and retrieve data. Hashing usually includes a hash table and a Hash function. The Hash function shortens the key into a very small value and the key is used to as an index in the Hash table

FUNCTIONS OF HASHING:

There are two main functions for hashing

- Storing or mapping
- Retrieval

STORING:

Storing in a Hash table occurs in two stages, first the key is passed into the Hash function to get a shortened key, then the key is used as an index to the hash table.

RETRIEVAL:

A key is passed and the values are checked in the Hash table at that index. Usually the search time for a Hash table is O (1).

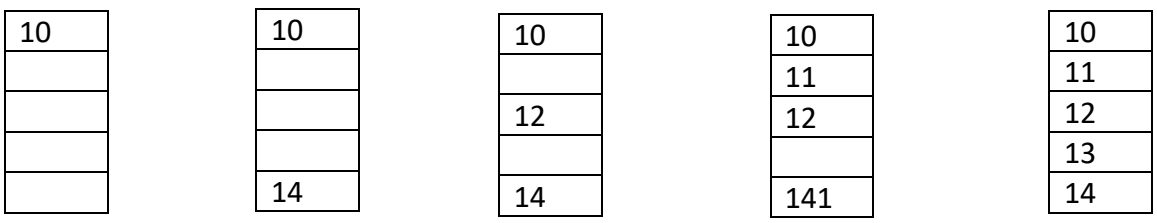
EXAMPLE:

Consider that there is a total of 5 keys and the size of the Hash table is 5.

Key= {10,14,12,11,13}

Let the hash function hash(x) be key%5.

Then



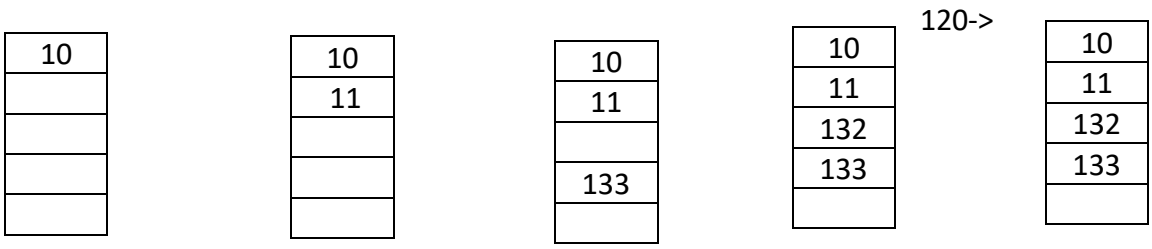
COLLISIONS:

As the Hash table uses the same Hash function then there is a huge possibility for collision to occur, in that case the collision has to handled to avoid overwriting data which may lead to a serious data loss.

EXAMPLE:

Let us consider that the size of the Hash table is 5.

Key= {10,11,120,133,132}



Here collision occurs as 120 % 5 = 0 but there is already an element in zero.

COLLISION HANDLING:

Once collision occurs in a Hash table it can be handled in different ways

1. Closed Hashing
    a. Linear probing
    b. Quadratic probing
    c. Double hashing
2. Open Hashing
    a. Separate chaining

CLOSED HASHING:

This method is called closed hashing as the data never leaves outside the Hash table.

a) LINEAR PROBING

In linear probing when a collision is encountered, we probe through the Hash table till an empty slot is found

EXAMPLE:

Consider a Hash table of size 5. Hash function(x)= key % 5

Key= {10,11,13,121,100}

Here the 10,11,13 are inserted into the table without any problems.
When 121 is passed 121 % 5 = 1. Since 1 is occupied by 11 we look in the (x + i) position in the i^{th} iteration. As 2 is empty 121 is placed there. The same process is repeated for 100 and 100 is placed at 5. The table looks like

10
11
13

10
11
121
13

10
11
121
13
100

b) QUADRATIC PROBING

Quadratic probing is similar to linear probing the only difference in quadratic probing is that the probe values are quadratically increasing. In quadratic probing when a collision is encountered at the x^{th} position instead of looking at the (x+1) we probe to (x+(1*1)) and if it fails then (x+(i*i)) and so on.

c) DOUBLE HASHING

In Double hashing we use another hash function to eliminate collision (Hash(x)+ i * hash(x)). This is named double hashing as the key is hashed two times.

SEPARATE CHAINING:

Here when collision is encountered the conflicting element is placed in a linked list to overcome it

REHASHING:

As the name suggests Rehashing means hashing the values again. To be able to understand why we rehash we should know about the Load factor of a hash table

Load factor α = $Total\ number\ of\ keys\ /\ size\ of\ the\ table$.

The load factor of a Hash table determines when to increase the size of the Hash table to maintain its time complexity to O (1). The time complexity increases with the increase in keys for an index. When the load factor crosses its predefined value then the size of the Hash table has to be increased to maintain the time complexity. Usually the predefined value of the Load factor is 0.75. (i.e): 75% of the table's capacity.

PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;

int TABLE_SIZE = 10;
class HashTable{
    public:
    int size = 0;
    class pair
    {
        public:
        int key, value;
        struct node *next;
        pair(int k,int v) : key{k}, value{v}, next{nullptr} {};
    };
    typedef struct pair node;

    vector<node *> table;

    static int hashFunction(int key)
    {
        return key % TABLE_SIZE;
    }

    HashTable()
    {
        for (int i = 0; i < TABLE_SIZE;i++)
            table.push_back(nullptr);
    }

    void insert(int key,int val)
    {
        int index = hashFunction(key);
        node *temp = table[index];
        if(!node *temp)
        {
            table[index] = new node(key, val);
            size++;
            return;
```

```
        }
        while(temp->next)
        {
            if(temp->key==key)
            {
                temp->value = val;
                return;
            }
            temp = temp->next;
        }
        node *newnode = new node(key, val);
        node *temp = table[index];
        newnode->next = temp;
        table[index] = newnode;
        size++;
    }

    void printTable()
    {
        for (int i = 0; i < TABLE_SIZE;i++)
        {
            cout << i << " ---> ";
            node *temp = table[i];
            while(temp)
            {
                cout << "(" << temp->key << "," << temp->value << ")   ";
            }
            cout << endl;
        }
    }
};


int main()
{
    HashTable Hash;
    int n;
    cin >> n;
    while(n--)
    {
        int key,val;
        cin >> key>>val;
        Hash.insert(key,val);
    }
    Hash.printTable();
}
#include<bits/stdc++.h>
using namespace std;
```

```
int TABLE_SIZE = 10;
class HashTable{
    public:
    int size = 0;
    class pair
    {
        public:
        int key, value;
        struct node *next;
        pair(int k,int v) : key{k}, value{v}, next{nullptr} {};
    };
    typedef struct pair node;

    vector<node *> table;

    static int hashFunction(int key)
    {
        return key % TABLE_SIZE;
    }

    HashTable()
    {
        for (int i = 0; i < TABLE_SIZE;i++)
            table.push_back(nullptr);
    }

    void insert(int key,int val)
    {
        int index = hashFunction(key);
        node *temp = table[index];
        if(!node *temp)
        {
            table[index] = new node(key, val);
            size++;
            return;
        }
        while(temp->next)
        {
            if(temp->key==key)
            {
                temp->value = val;
                return;
            }
            temp = temp->next;
        }
        node *newnode = new node(key, val);
        node *temp = table[index];
        newnode->next = temp;
```

```
        table[index] = newnode;
        size++;
    }

    void printTable()
    {
        for (int i = 0; i < TABLE_SIZE;i++)
        {
            cout << i << " ---> ";
            node *temp = table[i];
            while(temp)
            {
                cout << "(" << temp->key << "," << temp->value << ")   ";
            }
            cout << endl;
        }
    }
};


int main()
{
    HashTable Hash;
    int n;
    cin >> n;
    while(n--)
    {
        int key,val;
        cin >> key>>val;
        Hash.insert(key,val);
    }
    Hash.printTable();
}
```