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:
>
> 1. **Insert(1, 5)**: Assign the pair {1, 5} at the index (1%20 =1) in the Hash Table.
> 2. **Insert(2, 15)**: Assign the pair {2, 15} at the index (2%20 =2) in the Hash Table.
> 3. **Insert(3, 20)**: Assign the pair {3, 20} at the index (3%20 =3) in the Hash Table.
> 4. **Insert(4, 7)**: Assign the pair {4, 7} at the index (4%20 =4) in the Hash Table.
> 5. **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.
> 6. **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}.
> 7. **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

Below is the implementation of the above approach:

C++ C

```
// 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;

// Function to add key value pair
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");
}

// This code is contributed by Lovely Jain
```


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)*