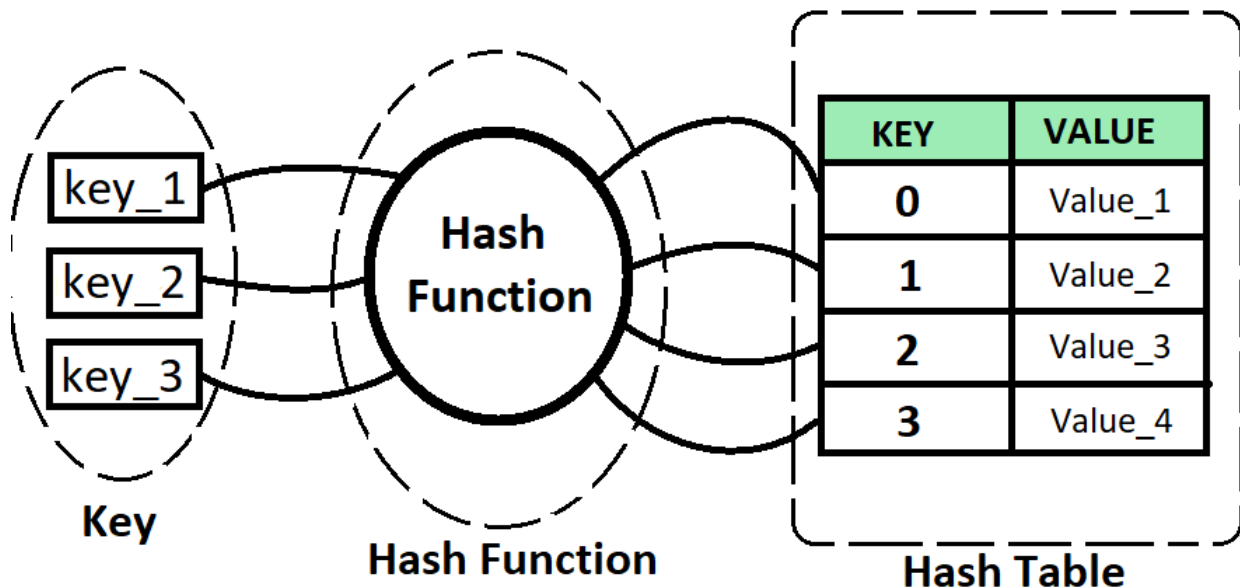


# Hash Tables

A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.



```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10 //The number of buckets in the hash table

// Node structure for linked list (chaining)
struct Node
{
    int key;
    int value;
    struct Node* next;
};

// The Literal Hash Function
int hashFunction(int key)
{
    return key % TABLE_SIZE; // Simple modulo-based hash function
}
```

# Load Factor

A hash table's load factor is determined by how many elements are kept there in relation to how big the table is. The table may be cluttered and have longer search times and collisions if the load factor is high. An ideal load factor can be maintained with the use of a good hash function and proper table resizing.

# Hash Functions

A Function that translates keys to array indices is known as a hash function. The keys should be evenly distributed across the array via a decent hash function to reduce collisions and ensure quick lookup speeds.

- **Integer universe assumption:** The keys are assumed to be integers within a certain range according to the integer universe assumption. This enables the use of basic hashing operations like division or multiplication hashing.
- **Hashing by division:** This straightforward hashing technique uses the key's remaining value after dividing it by the array's size as the index. When an array size is a prime number and the keys are evenly spaced out, it performs well.
- **Hashing by multiplication:** This straightforward hashing operation multiplies the key by a constant between 0 and 1 before taking the fractional portion of the outcome. After that, the index is determined by multiplying the fractional component by the array's size. Also, it functions effectively when the keys are scattered equally.

# Insert a Key-Value Pair

```
void insert(int key, int value) {
    int index = hashFunction(key);

    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->value = value;
    newNode->next = NULL;

    // Insert at head (chaining method)
    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
    } else {
        newNode->next = hashTable[index]; // Link new node to existing chain
        hashTable[index] = newNode;
    }
}
```

# Search For a key

```
int search(int key)
{
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];

    while (temp)
    {
        if (temp->key == key)
        {
            return temp->value; // Key found, return value
        }
        temp = temp->next;
    }
    return -1; // Key not found
}
```

# Delete a Key

```
void deleteKey(int key)
{
    int index = hashFunction(key);
    struct Node* temp = hashTable[index];
    struct Node* prev = NULL;

    while (temp)
    {
        if (temp->key == key)
        {
            if (prev == NULL)
            {
                hashTable[index] = temp->next; //Remove head node
            } else {
                prev->next = temp->next; //Remove middle or last node
            }
            free(temp);
            printf("Key %d deleted.\n", key);
            return;
        }
        prev = temp;
        temp = temp->next;
    }
    printf("Key %d not found.\n", key);
}
```

# Display the Hash Table

```
void display()
{
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("[%d]: ", i);
        struct Node* temp = hashTable[i];
        while (temp) {
            printf("(%d, %d) -> ", temp->key, temp->value);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

# Big-O Complexity of Arrays

Operation	Average Case $O(1)$	Worst Case $O(n)$
Insert	$O(1)$	$O(n)$ (if all elements hash to the same index)
Search	$O(1)$	$O(n)$ (worst-case collisions)
Delete	$O(1)$	$O(n)$

## Interview Tips for Hash Tables

### 1. Understand Different Collision Handling Techniques


- Chaining vs. Open Addressing.
- Linear probing vs. Quadratic probing vs. Double Hashing.

### 2. Know When to Use a Hash Table

- **Fast lookups?** Use a hash table.
- **Need ordered data?** Use a balanced BST instead.

### 3. Common Hash Table Interview Questions **Two Sum** (Find two numbers that add up to a target sum)

 **Longest Consecutive Subsequence**

 **Group Anagrams** (Use a hash table with sorted words as keys)

 **Find Duplicate Elements**