Hashing in Simple Terms

Hashing is like organizing your books on a shelf. Imagine you have a book title (the key) and want to find it quickly without scanning the entire shelf. Hashing uses a simple rule (called a hash function) to decide exactly where each book should go. Later, when you search for a book, you use the same rule to go directly to its spot.

Basic Operations

1. **Insert**: Place an item in the hash table.
2. **Search**: Find an item in the hash table.
3. **Delete**: Remove an item from the hash table.

Collision Handling

Sometimes, two items might get the same position from the hash function. This is called a collision. Common solutions are:

- **Chaining**: Use a linked list at each index to store multiple items.
- **Linear Probing**: Move to the next empty spot.

Example Code in C

Below is an implementation of a simple hash table with chaining to handle collisions.

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

// Define the size of the hash table
#define TABLE_SIZE 10

// Define the structure for a node in the linked list
typedef struct Node {
    int key;
    struct Node* next;
} Node;

// Hash table array of pointers
Node* hashTable[TABLE_SIZE];

// Hash function: returns the index for a given key
int hashFunction(int key) {
```

```
    return key % TABLE_SIZE;
}

// Insert a key into the hash table
void insert(int key) {
    int index = hashFunction(key);

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

    // Insert the node into the linked list at the index
    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
    } else {
        // Collision: add the new node at the beginning of the list
        newNode->next = hashTable[index];
        hashTable[index] = newNode;
    }
    printf("Inserted key %d at index %d\n", key, index);
}

// Search for a key in the hash table
int search(int key) {
    int index = hashFunction(key);
    Node* temp = hashTable[index];

    while (temp != NULL) {
        if (temp->key == key) {
            return 1; // Key found
        }
        temp = temp->next;
    }
    return 0; // Key not found
}

// Delete a key from the hash table
void delete(int key) {
    int index = hashFunction(key);
    Node* temp = hashTable[index];
    Node* prev = NULL;

    while (temp != NULL) {
        if (temp->key == key) {
            // Key found, remove it
            if (prev == NULL) {
                hashTable[index] = temp->next;
```

```
            } else {
                prev->next = temp->next;
            }
            free(temp);
            printf("Deleted key %d from index %d\n", key, index);
            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("Index %d: ", i);
        Node* temp = hashTable[i];
        while (temp != NULL) {
            printf("%d -> ", temp->key);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

// Main function
int main() {
    // Initialize the hash table
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }

    // Perform operations
    insert(5);
    insert(15);
    insert(25);
    insert(35);
    display();

    printf("Search for key 15: %s\n", search(15) ? "Found" : "Not Found");
    printf("Search for key 100: %s\n", search(100) ? "Found" : "Not Found");

    delete(15);
    display();

    return 0;
}
```

Explanation

1. **Hash Function**:
    - `hashFunction(key)` maps a key to an index using modulus operation (`key % TABLE_SIZE`).
2. **Insertion**:
    - Keys are placed in a linked list at their hashed index.
    - Collisions are resolved by adding to the linked list.
3. **Search**:
    - The hash function gives the index.
    - Search the linked list at that index for the key.
4. **Deletion**:
    - Traverse the list at the hashed index and remove the matching key.
5. **Display**:
    - Print the contents of each index in the hash table.

Example Run

- Insert keys `5`, `15`, `25`, and `35`.
- Hash function places them all in index `5` (due to collision).
- Each is added to the linked list at index `5`.
- Search for `15` → Found.
- Delete `15` → Removed from the list.

This approach demonstrates basic hashing with chaining. It can be extended for more advanced use cases.