

# Hashing, collisions

**Mr. Vikas Kumar**

Assistant Professor

Industry Embedded Program

## Lesson Plan

<b>Subject/Course</b>	<b>Competitive Coding</b>
<b>Lesson Title</b>	<b>Implement a hash table using separate chaining for collision handling. Perform operations like insertion, deletion, and search on the hash table.</b>

### Lesson Objectives

To implement a hash table using separate chaining for efficient data organization and collision handling.

To perform operations such as insertion, deletion, and search on the hash table.

To understand the working of a hash function and how it maps keys to indices in a table.

To analyze the time and space complexity of hashing operations and observe how collisions affect performance

# Problem Statement:

Implement a hash table using separate chaining for collision handling. Perform operations like insertion, deletion, and search on the hash table.

1. Insert key-value pairs.
2. Delete key-value pairs.
3. Search for a key
4. Display the complete hash table

# Concept

- Hashing converts data (key) into a fixed-size index using a hash function.
- Separate Chaining handles collisions by maintaining a linked list (or vector) at each index.
- This approach allows multiple key–value pairs to share the same index without overwriting.
- Provides  $O(1)$  average-time operations for insert, search, and delete.

# Algorithm/Logic

## 1. Initialize:

Create an array (vector) of buckets.

## 2. Hash Function:

$h = (h * \text{base} + \text{ASCII}(\text{char})) \bmod \text{TABLE\_SIZE}$

Base = 31 (a small prime)

Mod = Table size

# Algorithm/Logic

## 4. Insert (key, value):

- Compute index using hash function.
- If key exists, update its value.
- Else, append the new pair to the bucket.

## 5. Search (key):

- Compute index using the hash function.
- Traverse the bucket to find the key.
- Return its value or <NOT FOUND>.

# Algorithm/Logic

## 6. Delete (key):

- Compute index using hash function.
- Traverse the bucket and remove the key–value pair if found.

## 7. Display():

- Print all indices with their key–value pairs.

# Visualization

Hash Table (TABLE\_SIZE = 7):

0 : [apple → red]

1 : [banana → yellow]

3 : [grape → purple]

4 : [pear → green]

5 : [mango → orange]

If a collision occurs (same index), the new element is **chained** to the existing list.



# Code Implementation

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    const size_t TABLE_SIZE = 7;
    vector<vector<pair<string, string>>> table(TABLE_SIZE);

    auto insert = [&](string key, string value) {
        size_t idx = hashFunction(key, TABLE_SIZE);
        for (auto &p : table[idx])
            if (p.first == key) { p.second = value; return; }
        table[idx].push_back({key, value});
    };
}
```

```
auto searchKey = [&](string key) -> string {  
    size_t idx = hashFunction(key, TABLE_SIZE);  
    for (auto &p : table[idx])  
        if (p.first == key) return p.second;  
    return "<NOT FOUND>";  
};  
  
auto deleteKey = [&](string key) {  
    size_t idx = hashFunction(key, TABLE_SIZE);  
    auto &bucket = table[idx];  
    bucket.erase(remove_if(bucket.begin(), bucket.end(),  
        [&](auto &p){ return p.first == key; }),  
        bucket.end());  
};
```

```
auto display = [&]() {  
    for (size_t i = 0; i < TABLE_SIZE; ++i) {  
        cout << i << " : ";  
        for (auto &p : table[i])  
            cout << "[" << p.first << " -> " << p.second << "]" ";  
        cout << "\n";  
    } }  
insert("apple", "red");  
insert("banana", "yellow");  
insert("grape", "purple");  
insert("pear", "green");  
display();  
cout << "\nSearch pear: " << searchKey("pear") << endl;  
deleteKey("banana");  
cout << "\nAfter Deletion:\n"; display();  
}
```

# Output

Hash Table:

0 : [apple → red]

1 : [banana → yellow]

3 : [grape → purple]

4 : [pear → green]

Search pear: green

After Deletion:

0 : [apple → red]

3 : [grape → purple]

4 : [pear → green]

# Time & Space Complexity

Operation	Time Complexity	Space Complexity	Why?
Insert	$O(1)$	$O(n)$	Collision-free insert is constant time; many collisions cause linear time.
Search	$O(1)$	$O(n)$	Average $O(1)$ , but $O(n)$ if all keys hash to one bucket.
Delete	$O(1)$	$O(n)$	Same reasoning as insert/search.
space	$O(n)$	--	Stores all key-value pairs.

# Summary

- Hashing ensures fast access and updates.
- Separate chaining effectively handles collisions.
- Supports insert, search, and delete efficiently.
- Average performance remains  $O(1)$  for most operations.

# Practice Questions:

## 1. Design HashMap — LeetCode #706

↪ <https://leetcode.com/problems/design-hashmap/>

**Concept:** Implement hash table operations using chaining.

**Why Practice:** Same as our class example — perfect for reinforcement.

# Practice Questions:

## 2. Valid Anagram — LeetCode #242

∞ <https://leetcode.com/problems/valid-anagram/>

**Concept:** Uses hashing for frequency counting.



# Thanks