Implementation of hash function usage to store and retrieve key-value pairs.

Mr. Vikas Kumar

Assistant Professor

Industry Embedded Program

Lesson Plan

Subject/Course	**Competitive Coding**
Lesson Title	**Implementation of hash function usage to store and retrieve key-value pairs.**

Lesson Objectives
To implement a hash function for storing and retrieving key–value pairs efficiently.
To design functions for insertion, search, and display operations in a hash table.
To design functions for insertion, search, and display operations in a hash table.

Problem Statement:

Write a Program for a basic hash function in a programming language of your choice. Demonstrate its usage to store and retrieve key-value pairs.

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

Concept

Hashing converts data (key) into a fixed-size index using a hash function.

- Enables fast insertion and lookup in O(1) average time.
- Collision: Two keys produce the same index.
- Chaining: Handle collisions by maintaining a list of key-value pairs at each index.

Algorithm/Logic

1. Initialize:

Create an array (vector) of buckets.

2. Hash Function:

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

Base = 31 (a small prime)

Mod = Table size

Algorithm/Logic

4. Compute index using hash function .

If key exists → update value.

Else → append to that bucket.

5. Find (key):

Compute index using hash function.

Search for the key in the bucket.

Return its value or <NOT FOUND>.

6. Display():

Visualization

Hash Table (TABLE_SIZE = 11):

0 : [banana → yellow]

3 : [apple → red]

5 : [grape → purple]

8 : [mango → orange]

9 : [pear → green]

10: [peach → pink]

Code Implementation

```
const size_t TABLE_SIZE = 11;              // small size to show collisions
  vector<vector<pair<string,string>>> buckets(TABLE_SIZE);

  // helper lambdas
  auto insert = [&](const string &key, const string &value) {
    size_t idx = simpleStringHash(key, TABLE_SIZE);
    // update if key already exists
    for (auto &p : buckets[idx]) {
      if (p.first == key) { p.second = value; return; }
    }

    buckets[idx].push_back({key, value});
  };
```

Code Implementation

```
auto findValue = [&](const string &key) -> string {
        size_t idx = simpleStringHash(key, TABLE_SIZE);
        for (auto &p : buckets[idx]) {
            if (p.first == key) return p.second;
        }
        return "<NOT FOUND>";
    };

    auto showTable = [&]() {
        cout << "Hash table buckets (index : [key -> value, ...])\n";
        for (size_t i = 0; i < TABLE_SIZE; ++i) {
            cout << i << " : ";
 for (auto &p : buckets[i]) cout << "[" << p.first << " -> " << p.second << "] ";
            cout << "\n";
        }
    };
```

Output

```
Hash table buckets (index : [key → value])
0 : [banana → yellow]
3 : [apple → red]
5 : [grape → purple]
8 : [mango → orange]
9 : [pear → green]
10: [peach → pink]

Retrieve:
apple → red
banana → yellow
pear → green
kiwi → <NOT FOUND>
```

Time & Space Complexity

Operation	Time Complexity	Space Complexity	Why?
Insert	O(1)	O(n)	
Search	O(1)	O(n)	

Summary

- Hashing improves search efficiency.
- Collision handling via chaining ensures stable performance.
- Each bucket can store multiple key-value pairs.
- Effective for dictionaries, databases, and symbol tables.

Practice Questions:

1. **Design HashMap —** LeetCode #706

🔗 https://leetcode.com/problems/design-hashmap/

Concept: Build a simple hash map with chaining.

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

Practice Questions:

2. Valid Anagram — LeetCode #706

🔗 https://leetcode.com/problems/valid-anagram/

Concept: Uses hashing for frequency counting.

Thanks