Lab Experiment No. 9

Objective

Implementation and analyze Hashing techniques.

Theory

Hashing is a technique used for fast data retrieval. It involves mapping data to a hash table using a hash function. The primary objective of hashing is to optimize search, insertion, and deletion operations by achieving constant average time complexity, O(1). However, collisions can occur when multiple keys map to the same index. To handle such cases, various collision resolution techniques are used such as:

- **Open Addressing** (Linear Probing, Quadratic Probing, Double Hashing).
- **Separate Chaining.**

Algorithm:

1. Hash Function

1. Input: Key k, Table size m
2. Compute: index = k % m
3. Output: Index position in the hash table

2. Insertion (Using Separate Chaining)

1. Input: Key k, Value v
2. Compute hash index using the hash function.
3. If the bucket at that index is empty, create a new linked list and insert the key-value pair.
4. If a list already exists, append the key-value pair to the list (or update if key exists).
5. End.

3. Search

1. Input: Key k
2. Compute the hash index using the hash function.
3. Traverse the linked list at that index.
4. If the key is found, return the value.
5. Else, return “Key not found”.
6. End.

4. Deletion

1. Input: Key k
2. Compute the hash index using the hash function.
3. Traverse the linked list at that index.
4. If the key is found, delete the corresponding node.
5. If not found, return "Key not found".
6. End.

5. Traversal

1. For each index i from 0 to m-1:
    - Print index i.
    - If a list exists at that index, print all key-value pairs.
2. End.

6. Updation

1. Input: Key k, New value v
2. Compute the hash index using the hash function.
3. Traverse the list at that index to find the key.
4. If found, update the value to v.
5. Else, return "Key not found".
6. End.

Code

T-5.1. Implementation using Arrays. Depth-First Search (DFS)

```
#include <iostream>
#include <list>
using namespace std;

const int TABLE_SIZE = 10;

class HashTable {
private:
   list<pair<int, string>> table[TABLE_SIZE];

   int hashFunction(int key) {
      return key % TABLE_SIZE;
   }

public:
   void insert(int key, const string& value) {
      cout << "Insert " << key << endl;
      int index = hashFunction(key);
      table[index].push_back({key, value});
   }
   void display() {
      cout << "Display operation\n";
      bool hasElements = false;
      for (int i = 0; i < TABLE_SIZE; i++) {
         if (!table[i].empty()) {
            hasElements = true;
            cout << "Index " << i << ": ";
            for (auto& pair : table[i]) {
               cout << pair.first << "->" << pair.second
<< " ";
            }
            cout << endl;
         }
      }
      if (!hasElements) {
         cout << "Hash Table is empty.\n";
      }
   }
};
```

```
    void remove(int key) {
        cout << "Delete operation\n";
        int index = hashFunction(key);
        for (auto it = table[index].begin(); it !=
table[index].end(); ++it) {
            if (it->first == key) {
                table[index].erase(it);
                return;
            }
        }
    }
    void search(int key) {
        cout << "Search operation\n";
        int index = hashFunction(key);
        for (auto& element : table[index]) {
            if (element.first == key) {
                cout << "Found " << key << ": " <<
element.second << endl;
                return;
            }
        }
        cout << "Key not found.\n";
    }
```

```
int main() {
    HashTable ht;

    ht.insert(1, "Apple");
    ht.insert(11, "Banana");
    ht.insert(21, "Mango");

    ht.display();

    ht.search(11);
    ht.remove(11);
    ht.search(11);

    ht.display();

    return 0;
}
```

Sample Output

```
Insert 1
Insert 11
Insert 21
Display operation
Index 1: 1->Apple 11->Banana 21->Mango
Search operation
Found 11: Banana
Delete operation
Search operation
Key not found.
Display operation
Index 1: 1->Apple 21->Mango
```

Complexity Analysis

Table 9.1 Analysis of time complexity for Hashing Operation.

OPERATION	AVERAGE CASE	WORST CASE
Insertion	O(1)	O(n)

Search	O(1)	O(n)
Deletion	O(1)	O(n)

Conclusion

Hashing provides efficient data access, with average-case constant time for insertion, deletion, and search operations.

- **Collision resolution using separate chaining** effectively handles cases where multiple keys map to the same index.
- **Performance heavily depends on the quality of the hash function** - a good hash function ensures uniform distribution and minimizes collisions.
- **Table size selection is crucial** - choosing a prime number and maintaining a suitable load factor helps in maintaining optimal performance.

The experiment demonstrates that **hashing outperforms linear data structures** (like arrays and linked lists) for search-based operations, especially in large datasets