**Practical No:01**

**Problem Statement:**

**Consider telephone book database of n clients make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers.**

**Algorithm:**

**Data Structures:**

1. **Linear Probing Table**: Uses an array where each slot stores an entry and keeps track of whether the slot is occupied.
2. **Chaining Table**: Uses an array of linked lists to handle collisions by storing multiple entries at the same index.

**1. Insert Operation:**

**Linear Probing:**

1. **Calculate hash index**:
   * Use a hash function to find the index for the given name.
2. **Find an empty slot**:
   * Start at the calculated index and check if it's empty.
   * If not, move to the next index (index + 1) and check again.
   * Repeat this process until an empty slot is found or you’ve checked all slots.
3. **Insert entry**:
   * Once an empty slot is found, insert the name and phone number.

**Chaining:**

1. **Calculate hash index**:
   * Use the same hash function to find the index for the given name.
2. **Insert into linked list**:
   * Add the entry (name, phone number) to the **linked list** at the calculated index.

**2. Search Operation:**

**Linear Probing:**

1. **Calculate hash index**:
   * Use the hash function to find the index for the given name.
2. **Check the slot**:
   * Look at the slot at the calculated index.
   * If it’s empty, return “not found”.
   * If the name matches, return the phone number.
   * If not, move to the next slot (index + 1) and check again.
   * Repeat this process until the name is found or you’ve checked all slots.

**Chaining:**

1. **Calculate hash index**:
   * Use the hash function to find the index for the given name.
2. **Check the linked list**:
   * Go through the **linked list** at the calculated index.
   * If an entry matches the name, return the phone number.
   * If not, continue to the next entry in the list.

**Example :**

Let's say we insert **"Alice"** and **"Bob"** with phone numbers and then search for them. The steps would be:

**Step 1: Insert "Alice" and calculate its hash value**

**ASCII Sum for "Alice":**

* 'A' = 65
* 'l' = 108
* 'i' = 105
* 'c' = 99
* 'e' = 101

**Sum** = 65+108+105+99+101==478

**Hash Index** = 478%10=8

**Storage in the Hash Table:**

* **Linear Probing**: "Alice" is stored at index 8 because it is empty.
* **Chaining**: "Alice" is added to the linked list at index 8.

**Step 2: Insert "Bob" and calculate its hash value**

**ASCII Sum for "Bob":**

* 'B' = 66
* 'o' = 111
* 'b' = 98

**Sum** = 66+111+98=275

**Hash Index** = 275%10=5

**Storage in the Hash Table:**

* **Linear Probing**: "Bob" is stored at index 5 because it is empty.
* **Chaining**: "Bob" is added to the linked list at index 5.

**Step 3: Search for "Alice"**

**Linear Probing:**

* The hash index for "Alice" is 8.
* At index 8, "Alice" is found in **1 comparison**.

**Chaining:**

* The hash index for "Alice" is 8.
* In the linked list at index 8, "Alice" is found in **1 comparison**.

**Step 4: Search for "Charlie"**

**ASCII Sum for "Charlie":**

* 'C' = 67
* 'h' = 104
* 'a' = 97
* 'r' = 114
* 'l' = 108
* 'i' = 105
* 'e' = 101

**Sum** = 67+104+97+114+108+105+101=696

**Hash Index** = 696%10=6

**Linear Probing:**

* The hash index for "Charlie" is 6.
* Index 6 is empty, so "Charlie" is not found.

**Chaining:**

* The hash index for "Charlie" is 6.
* The linked list at index 6 is empty, so "Charlie" is not found.

**Final State of the Hash Tables**

**Linear Probing Hash Table:**

| **Index** | **Entry** | **Occupied** |
| --- | --- | --- |
| 0 |  | No |
| 1 |  | No |
| 2 |  | No |
| 3 |  | No |
| 4 |  | No |
| 5 | Bob | Yes |
| 6 |  | No |
| 7 |  | No |
| 8 | Alice | Yes |
| 9 |  | No |

**Chaining Hash Table:**

| **Index** | **Entries** |
| --- | --- |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 | Bob |
| 6 |  |
| 7 |  |
| 8 | Alice |
| 9 |  |

**Program :**

#include <iostream>

#include <vector>

#include <list>

using namespace std;

const int TABLE\_SIZE = 10;

// Entry structure

struct Entry {

string name;

string phone;

};

// Linear Probing Hash Table

class LinearProbingHashTable {

vector<Entry> table;

vector<bool> occupied;

public:

LinearProbingHashTable() : table(TABLE\_SIZE), occupied(TABLE\_SIZE, false) {}

int hashFunction(const string& key) {

int hash = 0;

for (char ch : key) {

hash = (hash + ch) % TABLE\_SIZE;

}

return hash;

}

void insert(const string& name, const string& phone) {

int index = hashFunction(name);

int originalIndex = index;

while (occupied[index]) {

index = (index + 1) % TABLE\_SIZE;

if (index == originalIndex) {

cout << "Hash table is full!\n";

return;

}

}

table[index] = {name, phone};

occupied[index] = true;

}

bool search(const string& name, string& phone, int& comparisons) {

int index = hashFunction(name);

int originalIndex = index;

comparisons = 0;

while (occupied[index]) {

comparisons++;

if (table[index].name == name) {

phone = table[index].phone;

return true;

}

index = (index + 1) % TABLE\_SIZE;

if (index == originalIndex) break;

}

return false;

}

};

// Chaining Hash Table

class ChainingHashTable {

vector<list<Entry>> table;

public:

ChainingHashTable() : table(TABLE\_SIZE) {}

int hashFunction(const string& key) {

int hash = 0;

for (char ch : key) {

hash = (hash + ch) % TABLE\_SIZE;

}

return hash;

}

void insert(const string& name, const string& phone) {

int index = hashFunction(name);

table[index].push\_back({name, phone});

}

bool search(const string& name, string& phone, int& comparisons) {

int index = hashFunction(name);

comparisons = 0;

for (const auto& entry : table[index]) {

comparisons++;

if (entry.name == name) {

phone = entry.phone;

return true;

}

}

return false;

}

};

int main() {

LinearProbingHashTable linearTable;

ChainingHashTable chainingTable;

int choice;

string name, phone;

do {

cout << "\nTelephone Book Menu:\n";

cout << "1. Insert\n2. Search\n3. Exit\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter name: ";

cin >> name;

cout << "Enter phone number: ";

cin >> phone;

linearTable.insert(name, phone);

chainingTable.insert(name, phone);

break;

case 2:

cout << "Enter name to search: ";

cin >> name;

int linearComparisons, chainingComparisons;

if (linearTable.search(name, phone, linearComparisons)) {

cout << "Linear Probing: Found \"" << name << "\" with phone " << phone << " in " << linearComparisons << " comparisons.\n";

} else {

cout << "Linear Probing: \"" << name << "\" not found.\n";

}

if (chainingTable.search(name, phone, chainingComparisons)) {

cout << "Chaining: Found \"" << name << "\" with phone " << phone << " in " << chainingComparisons << " comparisons.\n";

} else {

cout << "Chaining: \"" << name << "\" not found.\n";

}

break;

case 3:

cout << "Exiting program.\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 3);

return 0;

}

**Output for the Example:**

Telephone Book Menu:

1. Insert

2. Search

3. Exit

Enter your choice: 1

Enter name: Alice

Enter phone number: 12345

Telephone Book Menu:

1. Insert

2. Search

3. Exit

Enter your choice: 1

Enter name: Bob

Enter phone number: 67890

Telephone Book Menu:

1. Insert

2. Search

3. Exit

Enter your choice: 2

Enter name to search: Alice

Linear Probing: Found "Alice" with phone 12345 in 1 comparisons.

Chaining: Found "Alice" with phone 12345 in 1 comparisons.

Telephone Book Menu:

1. Insert

2. Search

3. Exit

Enter your choice: 2

Enter name to search: Charlie

Linear Probing: "Charlie" not found.

Chaining: "Charlie" not found.

Telephone Book Menu:

1. Insert

2. Search

3. Exit

Enter your choice: 3

Exiting program.