**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**

# Helper function to hash the client's name (key)

def hash\_function(key, size):

return hash(key) % size

# ====================

# Separate Chaining Hash Table

# ====================

class HashTableSeparateChaining:

def \_\_init\_\_(self, size):

self.size = size

self.table = [[] for \_ in range(size)]

self.comparisons = 0 # To track the number of comparisons

def insert(self, key, value):

index = hash\_function(key, self.size)

# Check if the key already exists in the list

for idx, (k, v) in enumerate(self.table[index]):

self.comparisons += 1

if k == key:

self.table[index][idx] = (key, value) # Update existing record

return

self.table[index].append((key, value)) # Insert new record

self.comparisons += 1 # Adding comparison for new entry

def search(self, key):

index = hash\_function(key, self.size)

for idx, (k, v) in enumerate(self.table[index]):

self.comparisons += 1

if k == key:

return v # Return the value (telephone number)

return None # Return None if not found

# ====================

# Open Addressing Hash Table (Linear Probing)

# ====================

class HashTableOpenAddressing:

def \_\_init\_\_(self, size):

self.size = size

self.table = [None] \* size

self.comparisons = 0 # To track the number of comparisons

def insert(self, key, value):

index = hash\_function(key, self.size)

original\_index = index # Store original index to avoid infinite loops

while self.table[index] is not None:

self.comparisons += 1

if self.table[index][0] == key: # If key already exists, update it

self.table[index] = (key, value)

return

index = (index + 1) % self.size # Linear probing

if index == original\_index:

raise Exception("HashTable is full")

self.table[index] = (key, value) # Insert at found empty spot

self.comparisons += 1 # Adding comparison for new entry

def search(self, key):

index = hash\_function(key, self.size)

original\_index = index

while self.table[index] is not None:

self.comparisons += 1

if self.table[index][0] == key:

return self.table[index][1] # Return the value (telephone number)

index = (index + 1) % self.size

if index == original\_index:

break

return None # Return None if not found

# =========================

# Testing the Hash Tables

# =========================

# Create both hash tables with a given size

size = 10

clients = [

("Alice", "555-1234"),

("Bob", "555-2345"),

("Charlie", "555-3456"),

("David", "555-4567"),

("Eve", "555-5678")

]

# Initialize separate chaining hash table and open addressing hash table

hash\_table\_chaining = HashTableSeparateChaining(size)

hash\_table\_open\_addressing = HashTableOpenAddressing(size)

# Insert client data into both hash tables

for name, phone in clients:

hash\_table\_chaining.insert(name, phone)

hash\_table\_open\_addressing.insert(name, phone)

# Now, search for each client and compare the number of comparisons

search\_clients = ["Alice", "Bob", "Charlie", "David", "Eve"]

# Separate Chaining Search Comparisons

for client in search\_clients:

hash\_table\_chaining.comparisons = 0 # Reset comparisons count

print(f"Searching for {client} in Separate Chaining:")

phone = hash\_table\_chaining.search(client)

print(f"Phone number: {phone}, Comparisons: {hash\_table\_chaining.comparisons}")

# Open Addressing Search Comparisons

for client in search\_clients:

hash\_table\_open\_addressing.comparisons = 0 # Reset comparisons count

print(f"Searching for {client} in Open Addressing:")

phone = hash\_table\_open\_addressing.search(client)

print(f"Phone number: {phone}, Comparisons: {hash\_table\_open\_addressing.comparisons}")