This lab exercise involves implementing a hash table data structure in C using separate chaining and open addressing for collision resolution.

Objectives:

- Understand the concept of hash tables, separate chaining and open addressing.
- Implement a hash table structure in C using linked lists for separate chaining and open addressing.
- Develop functions for searching, insertion, and removal of elements in the hash table.

Data Structure:

1. Define a structure named `HashTableEntry` to store a key-value pair. The key (unique identifier) can be an integer, and the value can be any data type you choose (e.g., integer, string).
2. Define a structure named `HashTable` that holds an array of pointers to linked lists. This array represents the hash table itself, where each index acts as a bucket for storing key-value pairs using separate chaining. This structure should contain the size of the array and its capacity when created.
3. Define the structure `Node` used for storing inside the key-value pairs with a pointer to the next node.
4. Define a structure named `LinkedList` containing a pointer to the head, a pointer to the last and the size (number of elements in a linked list).

Functions:

1. **`int getCode(int key):`** This function calculates the hash code for a given key using the modulo operator.
2. **`HashTable* createHashTable(int size):`** This function creates a new hash table by allocating memory for the hash table structure and initializing the array of pointers to linked lists with NULL values (representing empty buckets).
3. **`Value* get(HashTable* table, int key):`** This function searches for a key-value pair in the hash table. It calculates the hash code for the key, finds the corresponding bucket in the hash table array, and then iterates through the linked list in that bucket to find the matching key. If the key is found, it returns the value associated with the key. Otherwise, it returns NULL.
4. **`void put(HashTable* table, int key, Value value):`** This function inserts a new key-value pair into the hash table. It calculates the hash code for the key, finds the corresponding bucket in the hash table array, and then adds a new node containing the key-value pair at the beginning of the linked list in that bucket.
5. **`void remove(HashTable* table, int key):`** This function removes a key-value pair from the hash table. It calculates the hash code for the key, finds the corresponding bucket in the hash table array, and then iterates through the linked list to find the node with the matching key. If the key is found, it removes the node from the linked list.
6. **`int load_factor(HashTable* table):`** This function returns the load factor of the hash table.

7. **`int collision_rate(HashTable* table):`** This function returns the collision rate in the hash table. Recall that the collision rate is a value between 0 and 100 representing the rate of collision in a hash table.

Functions:

Implement the above data structure using linear probing (quadratic and double hashing is a plus) in open addressing as a method for collision resolution.

Applications:

1. Given two arrays of values T1 and T2, write a function that tests whether all values in array T1 are included in array T2.
2. Write a function that, for each value in an input array T1, displays the number of times that value appears in the array.
3. Given two arrays T1 and T2, create a union array containing the union of elements present in both given arrays. The order of elements in the output array does not matter.
4. Write a function that, given an array A[] of n values and another number x, determines whether there exist two elements in A whose sum is exactly x. For example, if A[] = {1, 4, 45, 6, 10, 8} and x = 16, the output is (10, 6).