**Assignment 1**

In this implementation, we have used a hash function that simply computes the remainder of the key by the size of the hash table. This is a common and simple hash function that works well in many cases. We could also use more complex hash functions, but the simplicity of this one makes it easy to understand and implement.

In our implementation, we have used linear probing to resolve collisions. Linear probing is a technique that involves checking the next available slot in the hash table if the slot of the hash value calculated for a key is already occupied by another key. If that slot is also occupied, then we keep moving to the next available slot until we find an empty slot. This method works well for small hash tables and has the advantage of being simple to implement.

We have implemented four functions in our implementation of a hash table in Python:

\_\_getitem\_\_(self, key): returns the value corresponding to key in the hash table. Raise a KeyError if the key does not exist. To use this function: call by table[key].

\_\_setitem\_\_(self, key, value): set a key-value pair. Raise an exception if the hash table is full and the key does not exist in the table yet. To use this function: call by table[key]=value.

\_\_contains\_\_(self, key): returns True if the key is in the table and False otherwise.

hash(self, key): calculates the hash value for the given key, using our own hash function.

In the main function, we have written two test functions to make sure that each function works properly. We have used two test cases in each function to test the functionality of each of these four functions. We have tested that the \_\_getitem\_\_ function is able to retrieve the values for the given keys, and that it raises a KeyError when trying to access a key that does not exist. We have tested that the \_\_setitem\_\_ function is able to set a key-value pair, and that it raises an exception when trying to set a key-value pair when the hash table is already full. We have tested that the \_\_contains\_\_ function returns True if the key is in the table and False otherwise. Finally, we have tested that the hash function is able to calculate the hash value for the given key, using our own hash function.

The implementation of the hash table using linear probing to resolve collisions is a simple yet effective way to store key-value pairs. Our choice of hash function works well for many cases, and our implementation of four functions provides a simple and easy-to-use interface to access and modify the hash table. By testing the implementation using

**Assignment Two**

The larger the hash table size, the longer it takes to read the file and store it into the hash table. This is because a larger hash table requires more memory and more time to compute the hash function for each key. However, a larger hash table also reduces the likelihood of collisions, which can improve the overall performance of the hash table for lookups and inserts. The optimal hash table size depends on the specific use case and the size of the data being stored.