**Task 1**

In this implementation, hash\_T represents our hash table class. We initialize it with a size of 100, but you can change that value if you need a larger or smaller table. The keys and values lists store our keys and values, respectively.

The \_\_getitem\_\_ function retrieves the value for a given key. We use linear probing to handle collisions. If the key is not found, we raise a KeyError.

The \_\_setitem\_\_ function sets a key-value pair in the hash table. If the table is full and the key is not already in the table, we raise an exception. If the key already exists in the table, we update its value.

The \_\_contains\_\_ function checks if a key is in the table. We use linear probing to handle collisions.

Finally, the hash function calculates the hash value for a given key. In this implementation, we use a custom hash function that sums the ASCII values of the characters in the key and returns the result modulo the table size.

**Task 2**

In this code, we define a HashTable class with methods to insert and retrieve key-value pairs. The hash method calculates the hash value for a given key, and the insert method stores the key-value pair in the appropriate bucket. The get method retrieves the value associated with a given key, or raises a KeyError if the key is not found.

To read the contents of a file, we define the read\_file function, which opens the file and returns a list of its lines with any leading or trailing whitespace removed.

We then use this hash table to store the words from two dictionary files, English\_small.txt and English\_large.txt, for three different hash table sizes: 200000, 300000, and 400000. We measure the time taken to insert each dictionary into the hash table, and print the results.

The runtime of a hash table is influenced by the size of the hash table because it affects the number of collisions that occur. A collision happens when two or more keys are mapped to the same hash value, and the hash table needs to search through the corresponding bucket to find the correct key-value pair. If the hash table size is too small, then there will be more collisions, which will increase the time taken to insert and retrieve values. If the hash table size is too large, then there will be more unused space in the table, which may waste memory. Therefore, we need to choose a hash table size that balances the trade-off between memory usage and performance.

**Task 3**

Linear probing is a simple method that probes consecutive slots until an empty slot is found. It is easy to implement but can lead to clustering and long probe lengths, especially when the table is almost full.

Quadratic probing uses a quadratic function to calculate the next probe slot. It has better clustering resistance compared to linear probing but can still lead to long probe lengths, especially when the table is nearly full.

Double hashing uses a secondary hash function to calculate the next probe slot. It has good clustering resistance and can significantly reduce the likelihood of long probe lengths compared to the previous two methods.

To compare these methods, we can measure the number of collisions and average probe length for each method when inserting a given number of items into a hash table of a certain size. The number of collisions is the number of times an item is attempted to be inserted into a slot that is already occupied. The average probe length is the average number of probes required to insert an item.

The following table summarizes the comparison:

|  |  |  |
| --- | --- | --- |
| Method | Number of collisions | Average probe length |
| Linear probing | High | High |
| Quadratic probing | Medium | Medium |
| Double hashing | Low | Low |

Overall, double hashing is the best method among the three in terms of both criteria. It has the lowest number of collisions and the lowest average probe length, which indicates that it is the most efficient and effective method for hash table insertion.