In [21]:

```
# Create your own hash table

class MyHashTable():
    def __init__(self, size):
            self.size = size
            self.data = [None] * self.size

    def __str__(self):
        return str(self.__dict__)
# This method above is used to print the attributes of the class object in

    def _hash(self, key):
        hash = 0
        for i in range(len(key)):
            hash = (hash + ord(key[i]) * i) % self.size
        return hash
# This is our custom hash function
# ord(key[i]) gives the code point in unicode of the character key[i]
# Then it returns the hash value obtained after applying our custom hash f

    def get(self, key):
        hash = self._hash(key)
        if self.data[hash]:
            for i in range(len(self.data[hash])):
                if self.data[hash][i][0] == key:
                    return self.data[hash][i][1]
        return None
# This function returns the value of the key entered by the user
# When passing the key to the _hash function, it calculates the hash value
# As multiple items may have been stored in the position of said hash valu
# So we loop over possible lists that may be in the same 'hash' position
# And compare the 1st element of the list with the given key
# If we match, the value is returned (which is the 2nd element)
# Otherwise, we return None

    def set(self, key, value):
        hash = self._hash(key)
        if not self.data[hash]:
            self.data[hash] = [[key, value]]
        else:
            self.data[hash].append([key, value])
        print(self.data)
# This function inserts a new key, value pair
# It calculates the hash value of the key, when passing the key to the _ha
# If the 'hash' position of the data array is empty, we insert the key, va
# And if the 'hash' position is not empty (collision), we append the list
    def keys(self):
        keys_array = []
        for i in range(self.size):
            if len(self.data[i]) > 1:
                for j in range(len(self.data[i])):
                    keys_array.append(self.data[i][j][0])
            else:
                keys_array.append(self.data[i][0][0])
        return keys_array
```

```
# This function returns all the keys
# It creates an array(list) to hold the keys
# Then loops over the whole table
# If it finds a bucket with something:
# It goes in and loops over all k, v pairs that the bucket may contain
# And adds the key of each list to the keys_array

    def values(self):
        values_array = []
        for i in range(self.size):
            if self.data[i]:
                for j in range(len(self.data[i])):
                    values_array.append(self.data[i][j][1])
        return values_array
# This function returns all the keys, similarly to the keys function
# But instead of adding the 1st element, we are adding the last one of eac
```

In [22]:

```
# Testing the functions

new_hash = MyHashTable(2)
print(new_hash)
```

```
{'size': 2, 'data': [None, None]}
```

In [23]:

```
new_hash.set('one',1)
new_hash.set('two',2)
new_hash.set('three',3)
new_hash.set('four',4)
new_hash.set('five',5)
print(new_hash)
```

```
[[['one', 1]], None]
[[['one', 1]], [['two', 2]]]
[[['one', 1]], [['two', 2], ['three', 3]]]
[[['one', 1]], [['two', 2], ['three', 3], ['four', 4]]]
[[['one', 1], ['five', 5]], [['two', 2], ['three', 3], ['four', 4]]]
{'size': 2, 'data': [[['one', 1], ['five', 5]], [['two', 2], ['three', 3],
['four', 4]]]}
```

In [24]:

```
print(new_hash.get('one'))
```

```
1
```

In [25]:

```
print(new_hash.keys())
```

```
['one', 'five', 'two', 'three', 'four']
```

In [26]:

```
print(new_hash.values())
```

```
[1, 5, 2, 3, 4]
```

Even though there are loops in the class 'MyHashTable', the time complexity is not O(n) because n corresponds to the size of the input. In this case, it will be the number of k, v pairs in the table.

The for loop in the '*hash' function* only runs for the length of the key, which can be rather small compared to the number of entries in general. The same applies to the loop in the 'get' function, as it runs for the length of the collisioined array, which won't take long in the majority of cases. Compared to the number of total entries, the time complexity for those loops won't reach as far, hence why the time complexity remains lower than O(n) and O(log n).

However, the keys and values methods are slightly worse than O(n), because we will have to loop over the entire size of the table once and then, loop over all the lists in the buckets which have collisions.