```
def _is_prime(number):
    if number < 2:
        return False

    if number != 2 and number % 2 == 0:
        return False

    for div in range(3, number // 2 + 1, 2):
        if number % div == 0:
            return False

    return True


def _next_capacity(old_capacity):
    current = old_capacity

    while not _is_prime(current):
        current += 1

    return current


class HashTable(dict):
    def __init__(self, capacity=11):
        super().__init__()
        self.capacity = capacity
        self.size = 0
        self.table = [None] * self.capacity
        self.load_factor_threshold = 0.7

    def h2(self, key):
        if type(key) == str:
            ascii_sum = 0

            for character in key:
                ascii_sum += ord(character)

            key = ascii_sum

        return key % self.capacity

    def h1(self, key, index):
        return (self.h2(key) + index) % self.capacity

    def _resize(self):
        """resize the hash table when the load factor exceeds the threshold."""
        old_table = self.table

        self.capacity = _next_capacity(self.capacity * 2)  # at least double the size
        self.size = 0  # reset the size, will be updated as we insert back
        self.table = [None] * self.capacity  # create the new table

        # insert back all the existing items into the new table
        for item in old_table:
```

```
            if item is not None:
                key, value = item
                self.__setitem__(key, value)

    def __setitem__(self, key, value):
        """insert or update a (key, value) pair in the hash table using open addressing with double
hashing"""
        if self.size / self.capacity > self.load_factor_threshold:
            self._resize()

        index = self.h2(key)
        i = 0

        while self.table[index] is not None:
            stored_key, _ = self.table[index]

            # same key values, update
            if stored_key == key:
                self.table[index] = (key, value)
                return

            # different key values, collision
            i += 1
            index = self.h1(key, i)

            if i >= self.capacity:
                break

        # key not found, insert
        self.table[index] = (key, value)
        self.size += 1

    def __getitem__(self, key):
        index = self.h2(key)
        i = 0

        while self.table[index] is not None:
            stored_key, stored_value = self.table[index]

            # same key values
            if stored_key == key:
                return stored_value

            # different key values, continue probing
            i += 1
            index = self.h1(key, i)

            if i >= self.capacity:
                break

        # key not found
        raise KeyError(f"Key '{key}' not found")

    def __delitem__(self, key):
        """remove a (key, value) pair from the hash table using open addressing with double hashing"""
```

```
    index = self.h2(key)
    i = 0

    while self.table[index] is not None:
        stored_key, _ = self.table[index]

        if stored_key == key:
            # key found, remove it by setting the table slot to "None"
            self.table[index] = None
            self.size -= 1

            # rehash the elements after removal
            self._rehash_after_removal(index)
            return

        # continue probing
        i += 1
        index = self.h1(key, i)

        if i >= self.capacity:
            break

    # key not found
    raise KeyError(f"Key '{key}' not found")

def _rehash_after_removal(self, remove_index):
    """rehash items after removal to maintain proper probing sequence"""
    i = 1
    index = self.h1(remove_index, i)

    while self.table[index] is not None:
        stored_key, stored_value = self.table[index]

        self.table[index] = None
        # self.size -= 1

        self.__setitem__(stored_key, stored_value)
        i += 1
        index = self.h1(remove_index, i)

def __len__(self):
    return self.size

def keys(self):
    table_keys = []

    for item in self.table:
        if item:
            table_keys.append(item[0])

    return table_keys

def values(self):
    table_values = []
```

```
    for item in self.table:
        if item:
            table_values.append(item[1])

    return table_values

def clear(self):
    for index in range(self.capacity):
        if self.table[index] is not None:
            self.table[index] = None

    self.size = 0

def __repr__(self):
    """return a string representation of the hash table"""
    return str(self)

def __str__(self):
    return str([item if item is not None else 'Empty' for item in self.table])
```