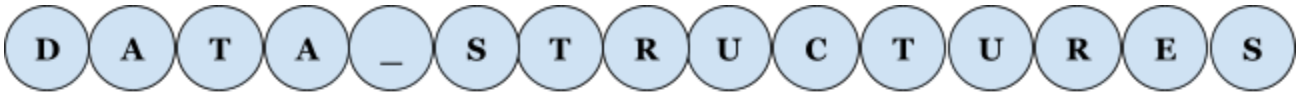
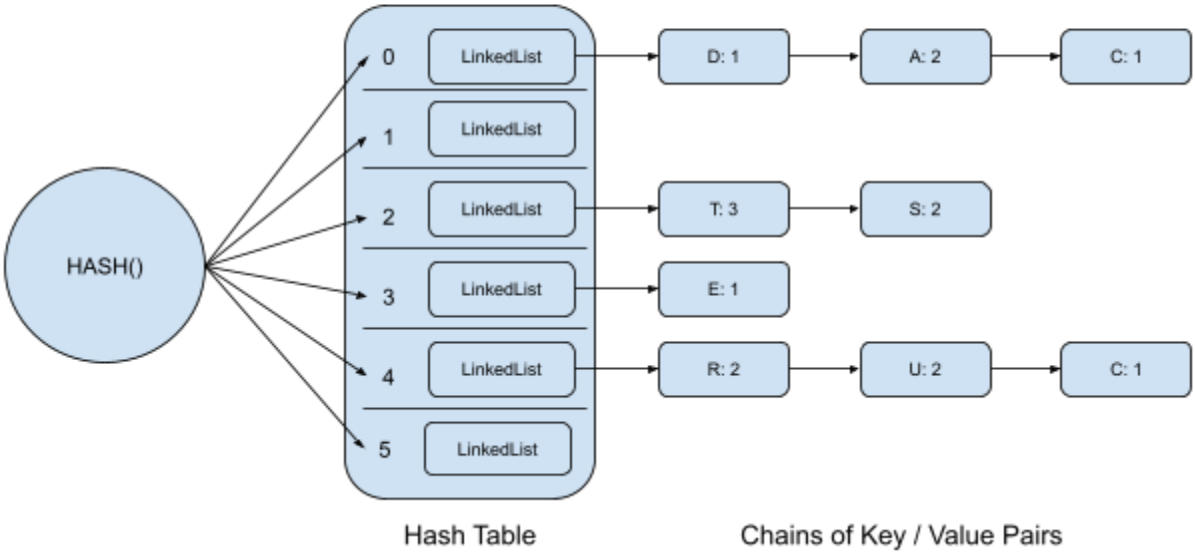
CS261 Data Structures

Assignment 6

Fall 2022

HashMap Implementation





Contents

General Instructions

1. Programs in this assignment must be written in Python 3 and submitted to Gradescope before the due date specified on Canvas and in the Course Schedule. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

2. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. Your goal is to pass all tests.

3. We encourage you to create your own test programs and cases, even though this work won't need to be submitted. Gradescope tests are limited in scope and may not cover all edge cases. Your submission must work on all valid inputs. We reserve the right to test your submission with more tests than Gradescope.

4. Your code must have an appropriate level of comments. At a minimum, each method should have a descriptive docstring. Additionally, add comments throughout the code to make it easy to follow and understand any non-obvious code.

5. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in skeleton code must retain their names and input/output parameters. Variables defined in skeleton code must also retain their names. We will only test your solution by making calls to methods defined in the skeleton code and by checking values of variables defined in the skeleton code.

   We have provided `_is_prime()` and `_next_prime()`, which are used by `__init__()` and are available for your use. You may add more helper methods and variables, and you are allowed to add optional default parameters to method definitions.

   However, certain classes and methods cannot be changed in any way. Please see the comments in the skeleton code for guidance. In particular, the content of any methods pre-written for you as part of the skeleton code must not be changed.

   Half points will be deducted from find_mode() for the incorrect time complexity.

6. The skeleton code and code examples provided in this document are part of assignment requirements. They have been carefully selected to demonstrate requirements for each method. Refer to them for detailed descriptions of expected method behavior, input/output parameters, and handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

7. **For each method, you are required to use an iterative solution.**

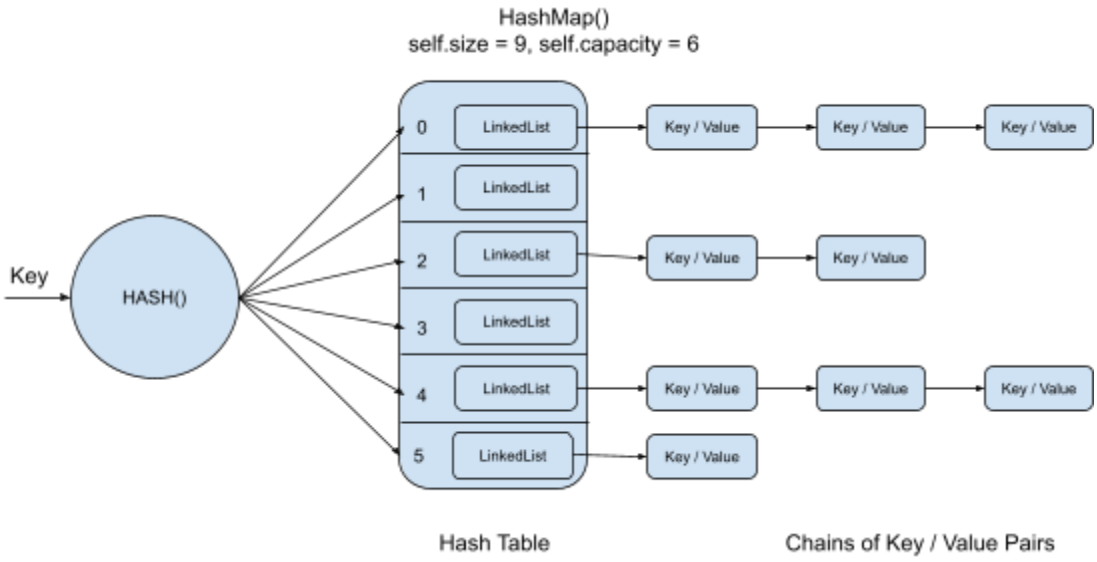
8. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementation of methods `__eq__()`, `__lt__()`, `__gt__()`, `__ge__()`, `__le__()`, and `__str__()`.

Part 1 - Summary and Specific Instructions

1. Implement the HashMap class by completing the provided skeleton code in the file `hash_map_sc.py`. Once completed, your implementation will include the following methods:

```
put()
get()
remove()
contains_key()
clear()
empty_buckets()
resize_table()
table_load()
get_keys()
find_mode()
```

2. Use a dynamic array to store your hash table, and implement **chaining for collision resolution** using a singly linked list. Chains of key/value pairs must be stored in linked list nodes. The diagram below illustrates the overall architecture of the HashMap class:



3. Two pre-written classes are provided for you in the skeleton code - DynamicArray and LinkedList (in `a6_include.py`). You **must** use objects of these classes in your HashMap class implementation. Use a DynamicArray object to store your hash table, and LinkedList objects to store chains of key/value pairs.

4. The provided DynamicArray and LinkedList classes may provide different functionality than those described in the lectures, or implemented in prior homework assignments. Review the docstrings in the skeleton code to understand the available methods, their use, and input/output parameters.

5. The number of objects stored in the hash map will be between 0 and 1,000,000 inclusive.

6. Two pre-written hash functions are provided in the skeleton code. Make sure you test your code with both functions. We will use these two functions in our testing of your implementation.

7. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods.

   You are NOT allowed to directly access any variables of the DynamicArray or LinkedList classes. All work must be done only by using class methods.

8. Variables in the SLNode class are not private. You ARE allowed to access and change their values directly. You do not need to write any getter or setter methods.

9. You may not use any imports beyond the ones included in the assignment source code provided.

put(self, key: str, value: object) -> None:

This method updates the key/value pair in the hash map. If the given key already exists in the hash map, its associated value must be replaced with the new value. If the given key is not in the hash map, a new key/value pair must be added.

For this hash map implementation, the table must be resized to double its current capacity when this method is called and the current load factor of the table is greater than or equal to 1.0.

Example #1:

```
m = HashMap(53, hash_function_1)
for i in range(150):
    m.put('str' + str(i), i * 100)
    if i % 25 == 24:
        print(m.empty_buckets(), round(m.table_load(), 2), m.get_size(),
m.get_capacity())
```

Output:

```
39 0.47 25 53
39 0.94 50 53
82 0.7 75 107
79 0.93 100 107
184 0.56 125 223
181 0.67 150 223
```

Example #2:

```
m = HashMap(41, hash_function_2)
for i in range(50):
    m.put('str' + str(i // 3), i * 100)
    if i % 10 == 9:
        print(m.empty_buckets(), round(m.table_load(), 2), m.get_size(),
m.get_capacity())
```

Output:

```
37 0.1 4 41
34 0.17 7 41
31 0.24 10 41
28 0.34 14 41
26 0.41 17 41
```

empty_buckets(self) -> int:

This method returns the number of empty buckets in the hash table.

Example #1:

```
m = HashMap(101, hash_function_1)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key1', 10)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key2', 20)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key1', 30)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key4', 40)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
```

Output:

```
101 0 101
100 1 101
99 2 101
99 2 101
98 3 101
```

Example #2:

```
m = HashMap(53, hash_function_1)
    for i in range(150):
        m.put('key' + str(i), i * 100)
        if i % 30 == 0:
            print(m.empty_buckets(), m.get_size(), m.get_capacity())
```

Output:

```
52 1 53
39 31 53
83 61 107
80 91 107
184 121 223
```

table_load(self) -> float:

This method returns the current hash table load factor.

Example #1:

```
m = HashMap(101, hash_function_1)
print(round(m.table_load(), 2))
m.put('key1', 10)
print(round(m.table_load(), 2))
m.put('key2', 20)
print(round(m.table_load(), 2))
m.put('key1', 30)
print(round(m.table_load(), 2))
```

Output:

```
0.0
0.01
0.02
0.02
```

Example #2:

```
m = HashMap(53, hash_function_1)
    for i in range(50):
        m.put('key' + str(i), i * 100)
        if i % 10 == 0:
            print(round(m.table_load(), 2), m.get_size(), m.get_capacity())
```

Output:

```
0.02 1 53
0.21 11 53
0.4 21 53
0.58 31 53
0.77 41 53
```

clear(self) -> None:

This method clears the contents of the hash map. It does not change the underlying hash table capacity.

Example #1:

```
m = HashMap(101, hash_function_1)
print(m.get_size(), m.get_capacity())
m.put('key1', 10)
m.put('key2', 20)
m.put('key1', 30)
print(m.get_size(), m.get_capacity())
m.clear()
print(m.get_size(), m.get_capacity())
```

Output:

```
0 101
2 101
0 101
```

Example #2:

```
m = HashMap(53, hash_function_1)
print(m.get_size(), m.get_capacity())
m.put('key1', 10)
print(m.get_size(), m.get_capacity())
m.put('key2', 20)
print(m.get_size(), m.get_capacity())
m.resize_table(100)
print(m.get_size(), m.get_capacity())
m.clear()
print(m.get_size(), m.get_capacity())
```

Output:

```
0 53
1 53
2 53
2 101
0 101
```

resize_table(self, new_capacity: int) -> None:

This method changes the capacity of the internal hash table. All existing key/value pairs must remain in the new hash map, and all hash table links must be rehashed.

First check that new_capacity is not less than 1; if so, the method does nothing.

If new_capacity is 1 or more, make sure it is a prime number. If not, change it to the next highest prime number. You may use the methods `_is_prime()` and `_next_prime()` from the skeleton code.

Example #1:

```
m = HashMap(23, hash_function_1)
m.put('key1', 10)
print(m.get_size(), m.get_capacity(), m.get('key1'), m.contains_key('key1'))
m.resize_table(30)
print(m.get_size(), m.get_capacity(), m.get('key1'), m.contains_key('key1'))
```

Output:

```
1 23 10 True
1 31 10 True
```

Example #2:

```
m = HashMap(79, hash_function_2)
keys = [i for i in range(1, 1000, 13)]
for key in keys:
    m.put(str(key), key * 42)
print(m.get_size(), m.get_capacity())

for capacity in range(111, 1000, 117):
    m.resize_table(capacity)
    m.put('some key', 'some value')
    result = m.contains_key('some key')
    m.remove('some key')
    for key in keys:
        result &= m.contains_key(str(key))
        result &= not m.contains_key(str(key + 1))
    print(capacity, result, m.get_size(), m.get_capacity(),
          round(m.table_load(), 2))
```

Output:

```
77 79
111 True 77 113 0.68
228 True 77 229 0.34
345 True 77 347 0.22
462 True 77 463 0.17
579 True 77 587 0.13
696 True 77 701 0.11
813 True 77 821 0.09
930 True 77 937 0.08
```

get(self, key: str) -> object:

This method returns the value associated with the given key. If the key is not in the hash map, the method returns None.

Example #1:

```
m = HashMap(31, hash_function_1)
print(m.get('key'))
m.put('key1', 10)
print(m.get('key1'))
```

Output:

```
None
10
```

Example #2:

```
m = HashMap(151, hash_function_2)
for i in range(200, 300, 7):
    m.put(str(i), i * 10)
print(m.get_size(), m.get_capacity())
for i in range(200, 300, 21):
    print(i, m.get(str(i)), m.get(str(i)) == i * 10)
    print(i + 1, m.get(str(i + 1)), m.get(str(i + 1)) == (i + 1) * 10)
```

Output:

```
15 151
200 2000 True
201 None False
221 2210 True
222 None False
242 2420 True
243 None False
263 2630 True
264 None False
284 2840 True
285 None False
```

contains_key(self, key: str) -> bool:

This method returns True if the given key is in the hash map, otherwise it returns False. An empty hash map does not contain any keys.

Example #1:

```
m = HashMap(53, hash_function_1)
print(m.contains_key('key1'))
m.put('key1', 10)
m.put('key2', 20)
m.put('key3', 30)
print(m.contains_key('key1'))
print(m.contains_key('key4'))
print(m.contains_key('key2'))
print(m.contains_key('key3'))
m.remove('key3')
print(m.contains_key('key3'))
```

Output:

```
False
True
False
True
True
False
```

Example #2:

```
m = HashMap(79, hash_function_2)
keys = [i for i in range(1, 1000, 20)]
for key in keys:
    m.put(str(key), key * 42)
print(m.get_size(), m.get_capacity())
result = True
for key in keys:
    # all inserted keys must be present
    result &= m.contains_key(str(key))
    # NOT inserted keys must be absent
    result &= not m.contains_key(str(key + 1))
print(result)
```

Output:

```
50 79
True
```

remove(self, key: str) -> None:

This method removes the given key and its associated value from the hash map. If the key is not in the hash map, the method does nothing (no exception needs to be raised).

Example #1:

```
m = HashMap(53, hash_function_1)
print(m.get('key1'))
m.put('key1', 10)
print(m.get('key1'))
m.remove('key1')
print(m.get('key1'))
m.remove('key4')
```

Output:

```
None
10
None
```

get_keys_and_values(self) -> DynamicArray:

This method returns a dynamic array where each index contains a tuple of a key/value pair stored in the hash map. The order of the keys in the dynamic array does not matter.

Example #1:

```
m = HashMap(11, hash_function_2)
for i in range(1, 6):
    m.put(str(i), str(i * 10))
print(m.get_keys_and_values())

m.put('20', '200')
m.remove('1')
m.resize_table(2)
print(m.get_keys_and_values())
```

Output:

```
[('1', '10'), ('2', '20'), ('3', '30'), ('4', '40'), ('5', '50')]
[('2', '20'), ('3', '30'), ('20', '200'), ('4', '40'), ('5', '50')]
```

find_mode(arr: DynamicArray) -> (DynamicArray, int):

Write a standalone function outside of the HashMap class that receives a dynamic array (that is not guaranteed to be sorted). This function will return a tuple containing, in this order, a dynamic array comprising the mode (most occurring) value/s of the array, and an integer that represents the highest frequency (how many times they appear).

If there is more than one value with the highest frequency, all values at that frequency should be included in the array being returned (the order does not matter). If there is only one mode, the dynamic array will only contain that value.

You may assume that the input array will contain at least one element, and that all values stored in the array will be strings. You do not need to write checks for these conditions.

For full credit, the function must be implemented with O(N) time complexity. For best results, we recommend using the separate chaining hash map provided for you in the function's skeleton code.

Example #1:

```
da = DynamicArray(["apple", "apple", "grape", "melon", "peach"])
mode, frequency = find_mode(da)
print(f"Input: {da}\nMode : {mode}, Frequency: {frequency}")
```

Output:

```
Input: ['apple', 'apple', 'grape', 'melon', 'peach']
Mode : ['apple'], Frequency: 2
```

Example #2:

```
test_cases = (
      ["Arch", "Manjaro", "Manjaro", "Mint", "Mint", "Mint", "Ubuntu",
       "Ubuntu", "Ubuntu"],
      ["one", "two", "three", "four", "five"],
      ["2", "4", "2", "6", "8", "4", "1", "3", "4", "5", "7", "3", "3", "2"]
)

for case in test_cases:
    da = DynamicArray(case)
    mode, frequency = find_mode(da)
    print(f"{da}\nMode : {mode}, Frequency: {frequency}\n")
```

Output:

```
Input: ['Arch', 'Manjaro', 'Manjaro', 'Mint', 'Mint', 'Mint', 'Ubuntu',
'Ubuntu', 'Ubuntu']
Mode : ['Mint', 'Ubuntu'], Frequency: 3

Input: ['one', 'two', 'three', 'four', 'five']
Mode : ['one', 'four', 'two', 'five', 'three'], Frequency: 1

Input: ['2', '4', '2', '6', '8', '4', '1', '3', '4', '5', '7', '3', '3', '2']
Mode : ['2', '3', '4'], Frequency: 3
```

Part 2 - Summary and Specific Instructions

1. Implement the HashMap class by completing the provided skeleton code in the file `hash_map_oa.py`. Your implementation will include the following methods:

   ```
   put()
   get()
   remove()
   contains_key()
   clear()
   empty_buckets()
   resize_table()
   table_load()
   get_keys()
   __iter__(), __next__()
   ```

2. Use a dynamic array to store your hash table, and implement **Open Addressing with Quadratic Probing** for collision resolution inside that dynamic array. key/value pairs must be stored in the array. Refer to the Explorations for an example of this implementation.

3. Use the pre-written DynamicArray class in `a6_include.py`. You **must** use objects of this class in your HashMap class implementation. Use a DynamicArray object to store your Open Addressing hash table.

4. The provided DynamicArray class may provide different functionality than the one described in the lectures, or implemented in prior homework assignments. Review the docstrings in the skeleton code to understand the available methods, their use, and input/output parameters.

5. The number of objects stored in the hash map will be between 0 and 1,000,000 inclusive.

6. Two pre-written hash functions are provided in the skeleton code. Make sure you test your code with both functions. We will use these two functions in our testing of your implementation.

7. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods. You are NOT allowed to directly access any variables of the DynamicArray class. All work must be done only by using class methods.

8. Variables in the HashEntry class are not private. You ARE allowed to access and change their values directly. You do not need to write any getter or setter methods.

9. You may not use any imports beyond the ones included in the assignment source code provided.

put(self, key: str, value: object) -> None:

This method updates the key/value pair in the hash map. If the given key already exists in the hash map, its associated value must be replaced with the new value. If the given key is not in the hash map, a new key/value pair must be added.

For this hash map implementation, the table must be resized to double its current capacity when this method is called and the current load factor of the table is greater than or equal to 0.5.

Example #1:

```
m = HashMap(53, hash_function_1)
for i in range(150):
    m.put('str' + str(i), i * 100)
    if i % 25 == 24:
        print(m.empty_buckets(), m.table_load(), m.get_size(),
              m.get_capacity())
```

Output:

```
28 0.47 25 53
57 0.47 50 107
148 0.34 75 223
123 0.45 100 223
324 0.28 125 449
299 0.33 150 449
```

Example #2:

```
m = HashMap(41, hash_function_2)
for i in range(50):
    m.put('str' + str(i // 3), i * 100)
    if i % 10 == 9:
        print(m.empty_buckets(), m.table_load(), m.get_size(),
              m.get_capacity())
```

Output:

```
37 0.1 4 41
34 0.17 7 41
31 0.24 10 41
27 0.34 14 41
24 0.41 17 41
```

table_load(self) -> float:

This method returns the current hash table load factor.

Example #1:

```
m = HashMap(101, hash_function_1)
print(m.table_load())
m.put('key1', 10)
print(m.table_load())
m.put('key2', 20)
print(m.table_load())
m.put('key1', 30)
print(m.table_load())
```

Output:

```
0.0
0.01
0.02
0.02
```

Example #2:

```
m = HashMap(53, hash_function_1)
    for i in range(50):
        m.put('key' + str(i), i * 100)
        if i % 10 == 0:
            print(m.table_load(), m.get_size(), m.get_capacity())
```

Output:

```
0.02 1 53
0.21 11 53
0.4 21 53
0.29 31 107
0.38 41 107
```

empty_buckets(self) -> int:

This method returns the number of empty buckets in the hash table.

Example #1:

```
m = HashMap(101, hash_function_1)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key1', 10)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key2', 20)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key1', 30)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
m.put('key4', 40)
print(m.empty_buckets(), m.get_size(), m.get_capacity())
```

Output:

```
101 0 101
100 1 101
99 2 101
99 2 101
98 3 101
```

Example #2:

```
m = HashMap(53, hash_function_1)
    for i in range(150):
        m.put('key' + str(i), i * 100)
        if i % 30 == 0:
            print(m.empty_buckets(), m.get_size(), m.get_capacity())
```

Output:

```
52 1 53
76 31 107
162 61 223
132 91 223
328 121 449
```

resize_table(self, new_capacity: int) -> None:

This method changes the capacity of the internal hash table. All existing key/value pairs must remain in the new hash map, and all hash table links must be rehashed.

First check that new_capacity is not less than the current number of elements in the hash map; if so, the method does nothing.

If new_capacity is valid, make sure it is a prime number; if not, change it to the next highest prime number. You may use the methods `_is_prime()` and `_next_prime()` from the skeleton code.

Example #1:

```
m = HashMap(23, hash_function_1)
m.put('key1', 10)
print(m.get_size(), m.get_capacity(), m.get('key1'), m.contains_key('key1'))
m.resize_table(30)
print(m.get_size(), m.get_capacity(), m.get('key1'), m.contains_key('key1'))
```

Output:

```
1 23 10 True
1 31 10 True
```

Example #2:

```
m = HashMap(79, hash_function_2)
keys = [i for i in range(1, 1000, 13)]
for key in keys:
    m.put(str(key), key * 42)
print(m.get_size(), m.get_capacity())

for capacity in range(111, 1000, 117):
    m.resize_table(capacity)
    m.put('some key', 'some value')
    result = m.contains_key('some key')
    m.remove('some key')
    for key in keys:
        result &= m.contains_key(str(key))
        result &= not m.contains_key(str(key + 1))
    print(capacity, result,m.get_size(), m.get_capacity(),
          round(m.table_load(), 2))
```

Output:

```
77 163
111 True 77 227 0.34
228 True 77 229 0.34
345 True 77 347 0.22
462 True 77 463 0.17
579 True 77 587 0.13
696 True 77 701 0.11
813 True 77 821 0.09
930 True 77 937 0.08
```

get(self, key: str) -> object:

This method returns the value associated with the given key. If the key is not in the hash map, the method returns None.

Example #1:

```
m = HashMap(31, hash_function_1)
print(m.get('key'))
m.put('key1', 10)
print(m.get('key1'))
```

Output:

```
None
10
```

Example #2:

```
m = HashMap(151, hash_function_2)
for i in range(200, 300, 7):
    m.put(str(i), i * 10)
print(m.get_size(), m.get_capacity())
for i in range(200, 300, 21):
    print(i, m.get(str(i)), m.get(str(i)) == i * 10)
    print(i + 1, m.get(str(i + 1)), m.get(str(i + 1)) == (i + 1) * 10)
```

Output:

```
15 151
200 2000 True
201 None False
221 2210 True
222 None False
242 2420 True
243 None False
263 2630 True
264 None False
284 2840 True
285 None False
```

contains_key(self, key: str) -> bool:

This method returns True if the given key is in the hash map, otherwise it returns False. An empty hash map does not contain any keys.

Example #1:

```
m = HashMap(53, hash_function_1)
print(m.contains_key('key1'))
m.put('key1', 10)
m.put('key2', 20)
m.put('key3', 30)
print(m.contains_key('key1'))
print(m.contains_key('key4'))
print(m.contains_key('key2'))
print(m.contains_key('key3'))
m.remove('key3')
print(m.contains_key('key3'))
```

Output:

```
False
True
False
True
True
False
```

Example #2:

```
m = HashMap(79, hash_function_2)
keys = [i for i in range(1, 1000, 20)]
for key in keys:
    m.put(str(key), key * 42)
print(m.get_size(), m.get_capacity())
result = True
for key in keys:
    # all inserted keys must be present
    result &= m.contains_key(str(key))
    # NOT inserted keys must be absent
    result &= not m.contains_key(str(key + 1))
print(result)
```

Output:

```
50 163
True
```

remove(self, key: str) -> None:

This method removes the given key and its associated value from the hash map. If the key is not in the hash map, the method does nothing (no exception needs to be raised).

Example #1:

```
m = HashMap(53, hash_function_1)
print(m.get('key1'))
m.put('key1', 10)
print(m.get('key1'))
m.remove('key1')
print(m.get('key1'))
m.remove('key4')
```

Output:

```
None
10
None
```

clear(self) -> None:

This method clears the contents of the hash map. It does not change the underlying hash table capacity.

Example #1:

```
m = HashMap(101, hash_function_1)
print(m.get_size(), m.get_capacity())
m.put('key1', 10)
m.put('key2', 20)
m.put('key1', 30)
print(m.get_size(), m.get_capacity())
m.clear()
print(m.get_size(), m.get_capacity())
```

Output:

```
0 101
2 101
0 101
```

Example #2:

```
m = HashMap(53, hash_function_1)
print(m.get_size(), m.get_capacity())
m.put('key1', 10)
print(m.get_size(), m.get_capacity())
m.put('key2', 20)
print(m.get_size(), m.get_capacity())
m.resize_table(100)
print(m.get_size(), m.get_capacity())
m.clear()
print(m.get_size(), m.get_capacity())
```

Output:

```
0 53
1 53
2 53
2 101
0 101
```

get_keys_and_values(self) -> DynamicArray:

This method returns a dynamic array where each index contains a tuple of a key/value pair stored in the hash map. The order of the keys in the dynamic array does not matter.

Example #1:

```
m = HashMap(11, hash_function_2)
for i in range(1, 6):
    m.put(str(i), str(i * 10))
print(m.get_keys_and_values())

m.resize_table(2)
print(m.get_keys_and_values())

m.put('20', '200')
m.remove('1')
m.resize_table(12)
print(m.get_keys_and_values())
```

Output:

```
[('1', '10'), ('2', '20'), ('3', '30'), ('4', '40'), ('5', '50')]
[('1', '10'), ('2', '20'), ('3', '30'), ('4', '40'), ('5', '50')]
[('4', '40'), ('5', '50'), ('20', '200'), ('2', '20'), ('3', '30')]
```

__iter__():

This method enables the hash map to iterate across itself. Implement this method in a similar way to the example in the Exploration: Encapsulation and Iterators.

You **ARE** permitted (and will need to) initialize a variable to track the iterator's progress through the hash map's contents.

You can use either of the two models demonstrated in the Exploration - you can build the iterator functionality inside the HashMap class, or you can create a separate iterator class.

Example #1:

```
m = HashMap(10, hash_function_1)
for i in range(5):
    m.put(str(i), str(i * 10))
print(m)
for item in m:
    print('K:', item.key, 'V:', item.value)
```

Output:

```
0: None
1: None
2: None
3: None
4: K: 0 V: 0 TS: False
5: K: 1 V: 10 TS: False
6: K: 2 V: 20 TS: False
7: K: 3 V: 30 TS: False
8: K: 4 V: 40 TS: False
9: None
10: None

K: 0 V: 0
K: 1 V: 10
K: 2 V: 20
K: 3 V: 30
K: 4 V: 40
```

__next__():

This method will return the next item in the hash map, based on the current location of the iterator. Implement this method in a similar way to the example in the Exploration: Encapsulation and Iterators. It will need to only iterate over active items.

Example #2:

```
m = HashMap(10, hash_function_2)
for i in range(5):
    m.put(str(i), str(i * 24))
m.remove('0')
m.remove('4')
print(m)
for item in m:
    print('K:', item.key, 'V:', item.value)
```

Output:

```
0: None
1: None
2: None
3: None
4: K: 0 V: 0 TS: True
5: K: 1 V: 24 TS: False
6: K: 2 V: 48 TS: False
7: K: 3 V: 72 TS: False
8: K: 4 V: 96 TS: True
9: None
10: None

K: 1 V: 24
K: 2 V: 48
K: 3 V: 72
```