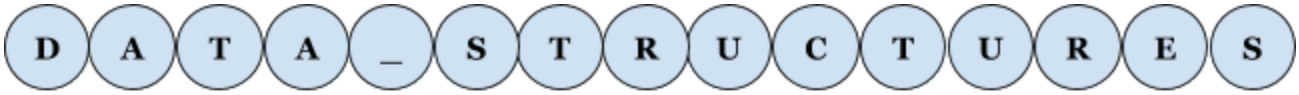
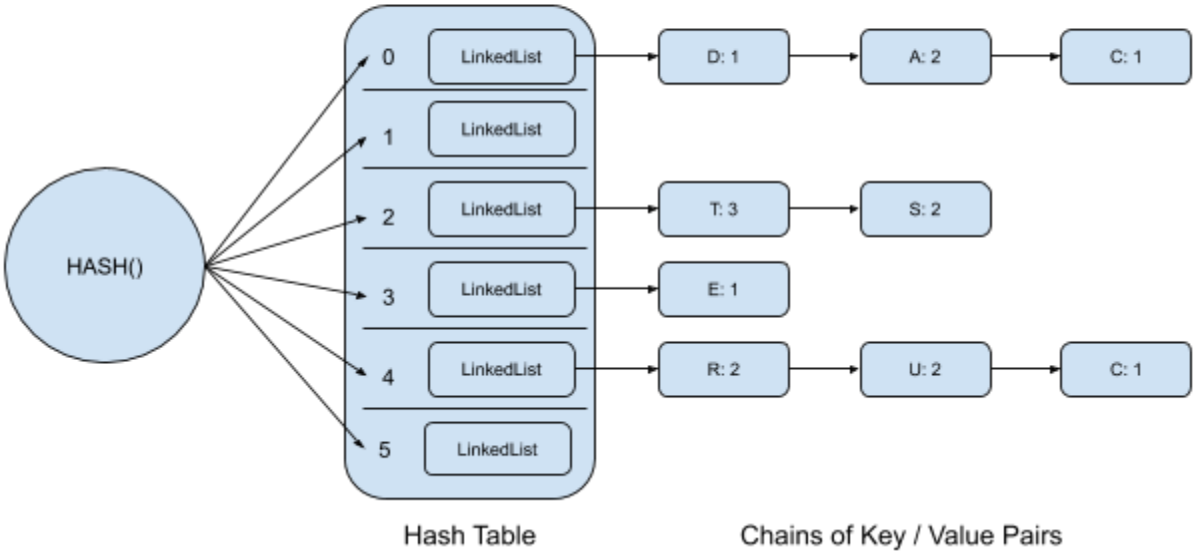
CS261 Data Structures

Assignment 5

v 1.01 (revised 5/15/2020)

Your Very Own Hash Map





Contents

General Instructions

1. Your project must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. 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.

2. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

3. You are encouraged to develop your own additional tests even though this work won't have to be submitted and won't be graded. **You are not permitted to share any additional tests with other students.** Understanding the functionality of methods and developing unit tests is part of the assignment and must be done individually. 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.

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. You can add more methods and variables, as needed. However, certains classes and methods can not be changed in any way. Please see comments in the skeleton code for guidance.

6. **Both 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 the detailed description of expected method behavior, input / output parameters, and handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

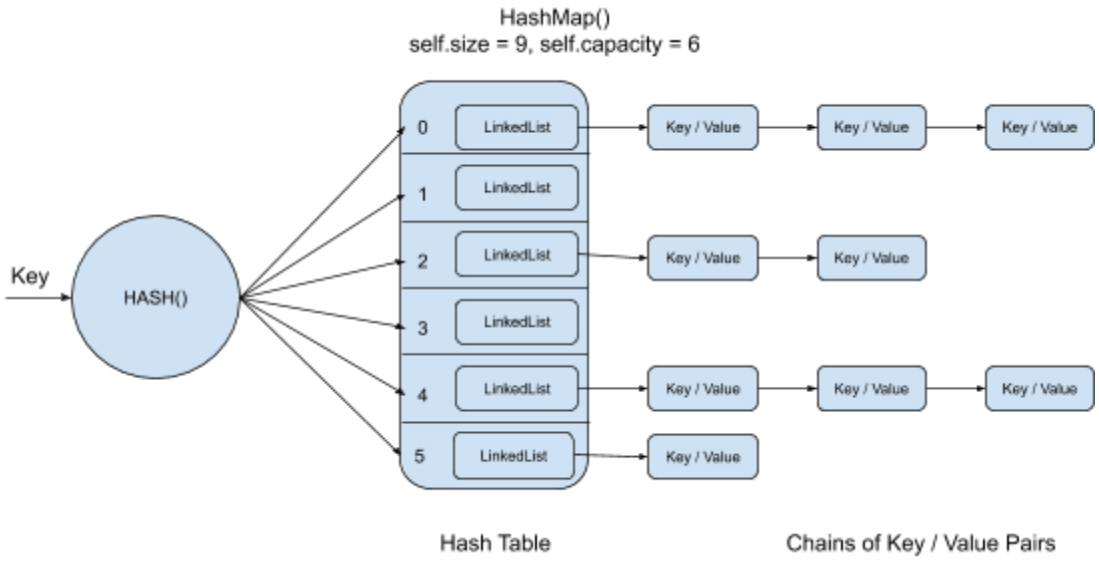
7. For each method, you can choose to implement a recursive or iterative solution. When using a recursive solution, be aware of maximum recursion depths on large inputs. We will specify the maximum input size that your solution must handle.

Part 1 - Summary and Specific Instructions

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

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

2. Use Python built-in list structure to store your hash table. Implement chaining for collision resolution. Chains of key / value pairs must be stored in linked list nodes. Pre-written LinkedList class is provided for you in the skeleton code. You should use objects of this class in your HashMap class implementation.



3. Variables in the HashMap() 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 for the HashMap() class.

4. Name of the file for Gradescope submission must be hash_map.py

empty_buckets(self) -> int:

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

Example #1:

```
m = HashMap(100, hash_function_1)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key1', 10)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key2', 20)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key1', 30)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key4', 40)
print(m.empty_buckets(), m.size, m.capacity)
```

Output:

```
100 0 100
99 1 100
98 2 100
98 2 100
97 3 100
```

Example #2:

```
m = HashMap(50, hash_function_1)
    for i in range(150):
        m.put('key' + str(i), i * 100)
        if i % 30 == 0:
            print(m.empty_buckets(), m.size, m.capacity)
```

Output:

```
49 1 50
39 31 50
36 61 50
33 91 50
30 121 50
```

table_load(self) -> float:

This method returns the current hash table load factor.

Example #1:

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

Output:

```
0.02 1 50
0.22 11 50
0.42 21 50
0.62 31 50
0.82 41 50
```

clear(self) -> None:

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

Example #1:

```
m = HashMap(100, hash_function_1)
print(m.size, m.capacity)
m.put('key1', 10)
m.put('key2', 20)
m.put('key1', 30)
print(m.size, m.capacity)
m.clear()
print(m.size, m.capacity)
```

Output:

```
0 100
2 100
0 100
```

Example #2:

```
m = HashMap(50, hash_function_1)
print(m.size, m.capacity)
m.put('key1', 10)
print(m.size, m.capacity)
m.put('key2', 20)
print(m.size, m.capacity)
m.resize_table(100)
print(m.size, m.capacity)
m.clear()
print(m.size, m.capacity)
```

Output:

```
0 50
1 50
2 50
2 100
0 100
```

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

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

Example #1:

```
m = HashMap(50, 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.size, m.capacity)
```

Output:

```
39 0.5 25 50
37 1.0 50 50
35 1.5 75 50
32 2.0 100 50
30 2.5 125 50
30 3.0 150 50
```

Example #2:

```
m = HashMap(40, 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.size, m.capacity)
```

Output:

```
36 0.1 4 40
33 0.175 7 40
30 0.25 10 40
27 0.35 14 40
25 0.425 17 40
```

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(50, 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(75, hash_function_2)
keys = [i for i in range(1, 1000, 20)]
for key in keys:
    m.put(str(key), key * 42)
print(m.size, m.capacity)
result = True
for key in keys:
    # all inserted keys must be present
    result = result and m.contains_key(str(key))
    # all NOT inserted keys must be absent
    result = result and not m.contains_key(str(key + 1))
print(result)
```

Output:

```
50 75
True
```

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(30, hash_function_1)
print(m.get('key'))
m.put('key1', 10)
print(m.get('key1'))
```

Output:

```
None
10
```

Example #2:

```
m = HashMap(150, hash_function_2)
for i in range(200, 300, 7):
    m.put(str(i), i * 10)
print(m.size, m.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:

```
15 150
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
```

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

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

Example #1:

```
m = HashMap(50, 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
```

resize_table(self, 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.

Example #1:

```
m = HashMap(20, hash_function_1)
m.put('key1', 10)
print(m.size, m.capacity, m.get('key1'), m.contains_key('key1'))
m.resize_table(30)
print(m.size, m.capacity, m.get('key1'), m.contains_key('key1'))
```

Output:

```
1 20 10 True
1 30 10 True
```

Example #2:

```
m = HashMap(75, hash_function_2)
keys = [i for i in range(1, 1000, 13)]
for key in keys:
    m.put(str(key), key * 42)
print(m.size, m.capacity)
for capacity in range(111, 1000, 117):
    m.resize_table(capacity)
    result = True
    for key in keys:
        result = result and m.contains_key(str(key))
        result = result and not m.contains_key(str(key + 1))
    print(capacity, result, m.size, m.capacity, round(m.table_load(), 2))
```

Output:

```
77 75
111 True 77 111 0.69
228 True 77 228 0.34
345 True 77 345 0.22
462 True 77 462 0.17
579 True 77 579 0.13
696 True 77 696 0.11
813 True 77 813 0.09
930 True 77 930 0.08
```