ECS 32B: Programming Assignment #4

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: You cannot use a dictionary.
- v.3: If `delete()` is used on an unmapped key, then the method should do nothing.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Thursday, September 10. Gradescope will say 12:30 AM on Friday, September 11, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

*This content is protected and may not be shared, uploaded, or distributed.

Some students tend to email me very close to the deadline. This is a bad idea. There is no guarantee that I will check my email right before the deadline.

3 Grading Breakdown

The assignment is out of 130 points. Here is a *tentative* breakdown of the worth of each part. It totals to 155 points, meaning that extra credit is possible. This means that you can earn up to 2.5% of extra credit to your final grade[1].

- `get_table_size()`: 5 points.
- `insert(self, key, value)`: 25 points.
- `print_keys(self)`: 25 points.
- `get_num_keys(self)`: 15 points.
- `get_load_factor(self)`: 5 points.
- `find(self, key)`: 15 points.
- The `in` Operator: 10 points.
- `update(self, key, value)`: 15 points.
- `delete(self, key)`: 15 points.
- Rehashing: 25 points.

4 Gradescope Autograder Details

You will submit two files:

- `hash_map.py`
- `unordered_list.py`

For each of `unordered_list.py` and `hash_map.py`, there should be no code placed outside of any of the functions that you define, unless that code is in the body of a conditional statement of the form `if __name__ == "__main__":`.

4.1 Released Test Cases vs. Hidden Test Cases

You will not see the results of certain test cases until after the deadline; these are the hidden test cases, and Gradescope will not tell you whether you have gotten them correct or not until after the deadline. One purpose of this is to promote the idea that you should test if your own code works and not depend solely on the autograder to do it. Unfortunately, Gradescope will display a dash as your score until after the deadline, but you can still tell if you have passed all of the visible test cases if you see no test cases that are marked red for your submission.

4.2 Changing Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talk about a small video that you can find on Canvas here. This video is from my ECS 32A course last summer session, so the autograder referenced therein is for a different assignment. Note that due to the hidden test cases, your score will show as a dash for all of the submissions, but you can still identify how many of the released test cases you got correct for each submission.

5 Unordered Map with Separate Chaining

In this programming assignment, you will implement an unordered map, meaning that you will implement your own version of a Python dictionary. To do this, you will implement a hash table of key-value pairs, where the key determines the location of the pair in the hash table, i.e. the key is what the hash function is used on. Your hash table implementation will use separate chaining (with unordered linked lists) for collision resolution.

[1] If, due to specific circumstances that we discussed over email, programming assignment #4 will be reweighted to be worth more for your grade, the extra credit will not be reweighted, i.e. scoring 140 out of 130 points will mean 1% of extra credit for you as it would for any other student.

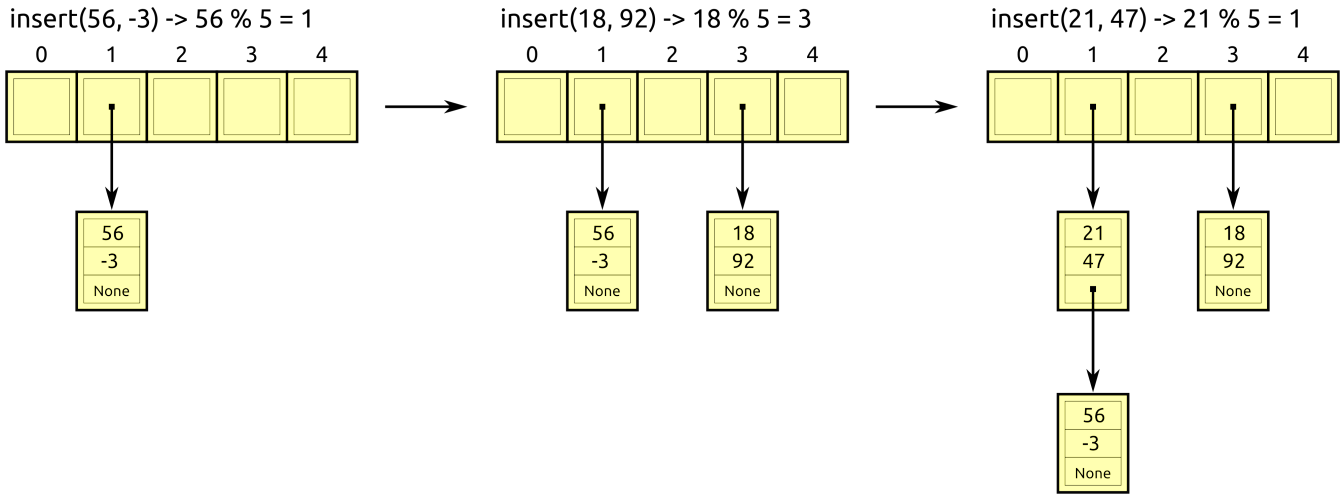
5.1 Background Information

A **map** is an ADT that stores key-value pairs. An **ordered map** is a map that maintains the relative ordering of the keys of the various key-value pairs, allowing one to, for example, print all of the key-value pairs in order of their keys. An **unordered map** is a map that sacrifices maintenance of the relative ordering (and operations that depend on knowing this relative ordering) in exchange for faster finding, insertion, and deletion of key-value pairs. An ordered map is typically implemented with a self-balancing binary search tree, such as an AVL tree or a red-black tree. An unordered map is typically implemented with a hash table; consequently, an unordered map is also known as a **hash map**. In C++, the standard library equivalents are `std::map` and `std::unordered_map`. In Python, an unordered map is a dictionary; I do not know of an ordered map that Python already provides you[2].

If a key-value pair (k, v) exists in a hash map H, then we say that k is **mapped** to v. If k is not mapped in H, then that means that there is no key-value pair in H such that k is the key; it does *not* mean that k is mapped to `None`, as that *would* be considered a key-value pair, i.e. (k, None).

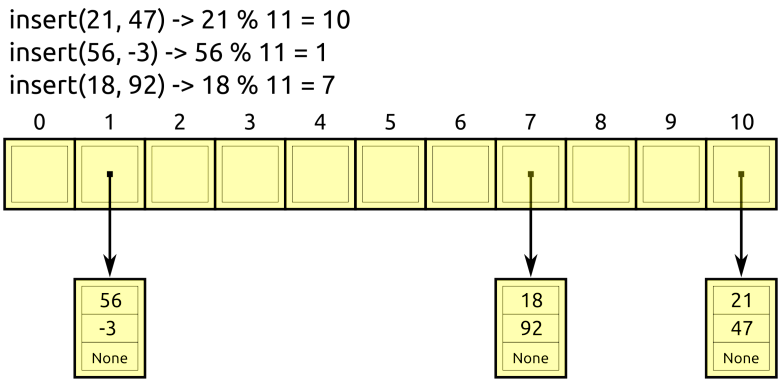
5.2 Example

Below is a visual example of how insertion of key-value pairs should behave in a hash table that uses separate chaining with unordered lists for collision resolution.



If rehashing is enabled, then ***after*** inserting the third key-value pair and observing that the load factor becomes at least $\frac{1}{2}$, the hash table should rehash to the one shown below.

[2]It is possible that there is one and that I did not look hard enough. Do note that `OrderedDict` is not an ordered map; an `OrderedDict` instance preserves the order in which the key-value pairs were inserted. While this may require use of a data structure such as a self-balancing binary search tree, it is not the same as an ordered map; there is a difference between preserving the relative order of the keys and preserving their insertion order.



5.3 Note about the Hash Table Reading

Although our primary textbook has a section about hash tables on section 6.5, I do not think that understanding the code in there helps. It might, but I do not expect it will and thus will not recommend it here. If you do look at it, do note that they have a different definition of "rehash" than the one that we (and Weiss' book) use.

5.4 `UnorderedList`

You are provided an almost certainly insufficient version of `unordered_list.py` on Canvas. You can make significant changes to it, or you could ignore it entirely and create your own `UnorderedList` implementation from scratch, so long as it is an unordered linked list. *Probably, the best way to go about this is to make modifications to `UnorderedList` as you go about implementing `HashMap`.*

At the top of the initial version of `hash_map.py` provided on Canvas, you can see the line `from unordered_list import UnorderedList`. This line should be in your final submission, and your `UnorderedList` implementation should be in `unordered_list.py`; you are not allowed to copy and paste your `UnorderedList` implementation into `hash_map.py`.

For my own solution, I made significant modifications to the `UnorderedList` implementation. Below are examples of modifications that I made. You may not need to make these modifications.

- My `Node` implementation has a `key` member and a `value` member.
- The `find` method of `UnorderedList` returns the node that was found or `None` otherwise.
- I removed the accessor methods in `Node` because I do not think accessor methods are appropriate for the `Node` class, but you can of course keep/modify those methods instead, if you wish.

5.5 `HashMap`

Each `HashMap` method that you must implement is described below, in the order in which I recommend that you implement them. Needless to say, you do not need to implement all of the methods to get some credit from the autograder. When possible, some test cases will test a specific method or methods. Of course, there are certain limits to this, e.g. if you cannot get `insert()` working, then it is hard to imagine you could get more than a few points of credit.

You are not allowed to use a dictionary. That would defeat the entire point of this. You are *implementing* a dictionary.

At the bottom of `hash_map.py`, you can see many examples of the hash map being used. Below are those examples and the expected output of those examples.

```
h = HashMap()
h.insert(3, 8)
h.insert(14, 15)
h.insert(50, 20)
h.insert(27, -5)
h.insert(36, 40)
print("=== Printing keys ===")
```

```
h.print_keys()
print("=== Done printing keys ===")
print("h.get_num_keys():", h.get_num_keys())
print("h.find(3):", h.find(3))
print("h.find(36):", h.find(36))
print("h.find(20):", h.find(20))
print("3 in h:", 3 in h)
print("36 in h:", 36 in h)
print("40 in h:", 40 in h)  # value doesn't work
h.delete(3)
h.delete(36)
h.delete(80)  # unsuccessful
print("=== Performed three deletions (two successful) ===")
h.print_keys()
print("=== By default, rehashing is disabled ===")
h.insert(11, 1)
h.insert(22, 1)
h.insert(33, 1)
h.print_keys()
print("h.get_num_keys():", h.get_num_keys())
print("Load factor (rounded):", round(h.get_load_factor(),2))
print("\n=== Let's try a hash table that supports rehashing ===")
h = HashMap(5, True)
h.insert(15, "values can be any type, by the way")
h.insert(20, "but the keys have to be integers")
print("===== Before insertion that causes rehash =====")
h.print_keys()
h.insert(26, 48)  # triggers rehashing, AFTER being inserted
print("===== After rehashing =====")
h.print_keys()
print("h.get_num_keys():", h.get_num_keys())
print("Load factor (rounded):", round(h.get_load_factor(),2))
```

```
=== Printing keys ===
0: None
1: None
2: None
3: 36 14 3
4: None
5: 27
6: 50
7: None
8: None
9: None
10: None
=== Done printing keys ===
h.get_num_keys(): 5
h.find(3): 8
h.find(36): 40
h.find(20): None
3 in h: True
36 in h: True
40 in h: False
=== Performed three deletions (two successful) ===
0: None
1: None
2: None
3: 14
4: None
5: 27
6: 50
7: None
8: None
9: None
10: None
=== By default, rehashing is disabled ===
0: 33 22 11
1: None
2: None
3: 14
4: None
5: 27
6: 50
7: None
```

```
8: None
9: None
10: None
h.get_num_keys(): 6
Load factor (rounded): 0.55

=== Let's try a hash table that supports rehashing ===
===== Before insertion that causes rehash =====
0: 20 15
1: None
2: None
3: None
4: None
===== After rehashing =====
0: None
1: None
2: None
3: None
4: 26 15
5: None
6: None
7: None
8: None
9: 20
10: None
h.get_num_keys(): 3
Load factor (rounded): 0.27
```

5.5.1 Initializer

This method should initialize any member variables that you feel are necessary. In particular, you should set up the underlying hash table, i.e. the list of linked lists. (Recall that in a separate chaining hash table, each bucket/slot in the table corresponds to a linked list.) You can use the following line for this. (You do not have to use the same variable names.) The line is based on the example in slide #40 of slide deck #2.

```
self.table = [UnorderedList() for i in range(self.table_size)]
```

5.5.2 `get_table_size()`

This method should return the size (i.e. number of buckets/slots) in the table.

5.5.3 `insert(self, key, value)`

This method should insert the given key-value pair into the table. At this point, do not worry about rehashing. If the given key was already mapped in the hash table, then this method should `raise` a `ValueError` with the message `"Key already mapped."` without modifying the hash table. Below are more examples of `insert()` in use. (It also uses the `find()` method.)

```
>>> h = HashMap()
>>> h.insert(5,17)
>>> h.find(5)
17
>>> h.insert(5,19)
Traceback (most recent call last):
...
ValueError: Key already mapped.
```

5.5.4 `print_keys(self)`

You can see this method's behavior in the example output above. This method should print each key in the hash table. While printing keys within the same linked list, they should be printed in the order in which they appear, i.e. the head of the list should have its key printed first.

5.5.5 `get_num_keys(self)`

Returns the number of key-value pairs in the table.

This method must take constant time. If it does not, you will lose any points for this method when we inspect your code.

5.5.6 `get_load_factor(self)`

Returns the load factor. (Don't round it.)

5.5.7 `find(self, key)`

Returns the value associated with the key passed in as argument.

5.5.8 The `in` Operator

Notice, in the example output above, that the `in` operator worked with an instance of `HashMap`. To make this happen, implement the `__contains__(self, key)` method defined at the bottom of `hash_map.py`. This method should return `True` if `key` exists in any key-value pair in the hash table and `False` otherwise.

5.5.9 `update(self, key, value)`

This method should change the value currently associated with the given key to be the new value. If the key was not already mapped in the hash table, then this method should `raise` a `ValueError` with the message `"Key not mapped."` without modifying the hash table.

5.5.10 `delete(self, key)`

This method should delete the key-value pair associated with the given key.
Update: If the given `key` is not mapped, then `delete(...)` should do nothing.

5.5.11 Rehashing

The second explicit argument (`can_rehash`) to the initializer dictates if the hash map supports rehashing or not. Modify your `insert()` method so that *after completing the insertion*, if the hash map supports rehashing and the load factor is at least $\frac{1}{2}$, then rehashing should occur. Assuming the old hash table size is m, the new hash table size should be the lowest prime number x such that $x \geq 2m$. All key-value pairs in the old table must be moved to the new table. When moving nodes within the same linked list in the old table, they should be reinserted in the order in which they appear. For example, if in the old table, there is a linked list containing (in order) nodes A, B, and C, then node A should be inserted into the new table first, followed by node B and then finally node C. (Of course, it is unlikely that they end up in the same linked list again.)