CSCI 204 – Data Structures and Algorithms

Lab 12 -- Map ADT using Hash Map

1 Objectives

The objectives of this lab are to

- Become familiar with hash map implementation of Map ADT, hash functions and their collision resolution methods;
- Write hash map methods.

In this lab you are given a partially completed hash table implementation and you are to complete the missing functions to make the hash table object work correctly.

As usual, read the tasks first before you begin to tackle them.

2 Introduction

Hash tables allow constant time O(1) for search and insertion. The basic idea of hashing is to map keys directly into a slot in the hash table without *searching* for the spot. When a collision happens, that is, two or more keys are mapped into the same slot, hash tables in general try to resolve the collision either in *open hashing* (a.k.a. *chained hashing*) using another data structure to store the items that are mapped into the same slot in the hash table, or in *closed hashing* which searches for an available spot within the hash table to store the items that are in conflict.

We concentrate on *closed hashing* in this lab, that is, the collisions are resolved by finding another slot within the hash table if we can. The primary indicator of hash table performance in our study is the number of collisions that occur when an item is inserted into the hash table. The fewer collisions needed to find a slot for insertion, the better the performance is.

You will be implementing the operations Map ADT using hash table in a class called HashMap.

3 Getting started

Begin by making a directory for this lab. Then copy the starter files from the Linux directory ~csci204//student-labs/lab12/. You can also retrieve the zip file from the course Moodle site. You should see the following set of files:

```
hashmap.py  testhash.py
```

You will work with these files to complete your lab exercises.

4 Your work

First, please load the program `testhash.py` and take a look at the program. This program runs a sequence of tests for a completed hash map. The tests include insertion of items into a hash map, printing the keys and values in the hash table using an iterator, removing some keys from the hash table, printing the collision statistics, and lastly testing a large number of insertions to make the statistics such as collision count more meaningful.

If you run the program `testhash.py`, you will find that the program does not work. It will work after you complete the `HashMap` class.

To run the testhash.py file, you must use terminal/command-line. Use the command below:

```
PYTHONHASHSEED=1 python testhash.py
```

You have to use this command because (as of Python 3.3 and to the currently newest 3.8) the built-in Python hash function, `hash()`, uses a random seed if the `PYTHONHASHSEED` environmental variable isn't explicitly set. If you just tried to run the `test_hash` file without setting that seed, you could have a different result every time you run it (or not, it's practically unpredictable!)

4.1 (TASK 1) Write the _lookup method

Complete the `_lookup` (helper) method for the hash map. This method will help you complete most of the other methods for the hash map. Lookup takes a hash function and a key. It first computes the slot for that key and gets the contents currently occupying that slot in the array.

Lookup returns a pair of the slot and its contents (the contents are the desired key,value pair in the array):

```
return slot, contents
```

If nothing is in those array contents, it returns the slot and None as a pair:

```
return slot, None
```

If the wrong key is in those array contents, it returns None and None as a pair:

```
return None, None
```

You won't be able to test your code just yet.

4.2 (TASK 2) Write the add method

Complete the `add` method for the hash map. Add takes a key and a value. If the key does not exist, add the new entry to the map. If the key exists, update its value. Each key could go into two possible slots in the array. A collision occurs when the key tries to go into a slot occupied by another key. Fatal collisions occur when both slots are already occupied by some other key. If there is a fatal collision, do nothing.

Use the `_lookup` method to help you out. We have provided two possible hash functions (`_hash1` and `_hash2`). Here is how to call `_lookup` with a hash function:

```
self._lookup(self._hash1, key)
```

The primary location to try is the slot found by `_hash1`. The backup location is the one found by `_hash2`. Don't forget to update the size as needed. When you find a location for a key,value pair, put a `_MapEntry` instance in that location:

```
self._array[slot] = _MapEntry(key, value)
```

Be careful... if a key is in its backup location and the thing in the primary location has since been removed, you still want to update the value in the backup location, not add the key to the primary location. If you make a mistake here, the key will appear in the hash map twice.

Once you complete the add method, you should run the `testhash.py` program. You should see the test for the `peek` method succeed. We have provided `peek` for you but it will only work if your `add` and `_lookup` methods work. (We will test one more feature of `add` later on).

4.3 (TASK 3) Write the iterator for the hash map

Complete the `iterator` for the hash map. Your iterator will return the keys in the map.

An iterator is special object that steps through another collection of objects one at a time, allowing programming structures such as `for` loop to iterate through the collection. Since the given program `hashmap.py` has not implemented the iterator for the hash table yet, the `for` loop in the `testhash.py` program doesn't work.

The basic idea of an iterator is to create a mechanism to step through collection of the objects one at a time. In our case, this collection of objects is the items (key/value pairs) in the hash map in the form of an array (or Python list). Stepping through the hash map basically is going through each item in the Python list (the `self._array` in the `HashMap` class). However a hash table may contain items that are empty, the iterator needs to skip over these items, not returning them for examination by the calling function. Your iterator has to take care of this scenario.

Once you complete the iterator, you should run the `testhash.py` program again. If the iterator you create works correctly, the test program should print out a complete list of the hash map contents, a total of 10 pairs of keys and values.

4.4 (TASK 4) Write the remove method

Complete the `remove` method for the hash map. The logic of removal follows a similar pattern to that of insertion. If the key exists in the hash map, set the contents of that slot to None. If the key is not in the hash map, do nothing. Again, use the _lookup method to help you out. Don't forget to update the size as needed.

Once the `remove` method is completed, run the program `testhash.py` again. The program now should print the correct contents of the hash map even after the removal of two keys. You should also see the peek function looks for a non-existent key and you should see the key,value pair 6,8 removed successfully.

If 6,8 is still in your hash map when it says it should be gone then you have an error to fix. (The primary slot for key 6 has become empty but 6 is in its backup slot).

4.5 (TASK 5) Adding collision counting

The `HashMap` class contains a variable called `_collisions` which is supposed to count the total number of collisions occurred during the insertions. Your next task is to increment this value in proper places.

Where should the collision count be incremented? Collisions occur when we try to insert a key into a slot that is already occupied by some other key. We can recover from some collisions by trying to insert the key in its backup location but we should still include those collisions in our count. Some collisions are fatal and result in the key failing to be added to the hash map, **however do not count fatal collisions**.

Having completed this part, you should run the program `testhash.py` again. This time, you should see the collision count being used correctly. You can also see how a larger capacity lowers the collision count (and allows the keys to possibly all fit).

5 Submitting the lab work

Upload `hashmap.py` to Moodle and double-check that it uploaded successfully.

Congratulations on finishing this lab (and lab component of the entire course)!