Computational Problem Solving CSCI-603
Chained Hash Map Lab 7

1 Introduction

In this lab, you will be revising the hash table implementation of a map to use chainning for handling collisions. You will also measure hashing performance with different hash functions.

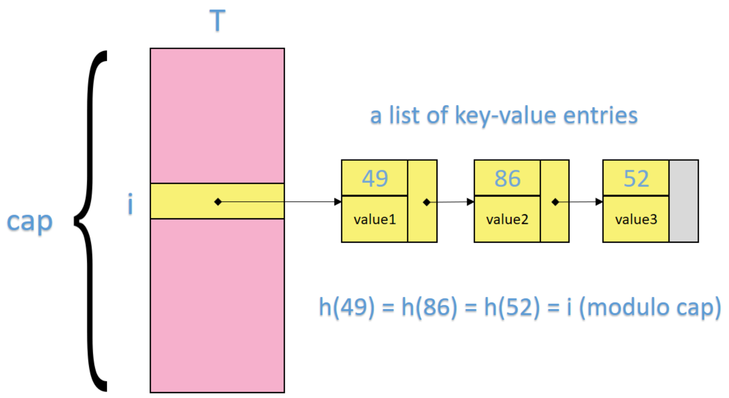

T
a list of key-value entries
cap
i
49
value1
86
value2
52
value3
h(49) = h(86) = h(52) = i (modulo cap)

Figure 1: The chained hash map design.

2 Implementation and Answers to Questions

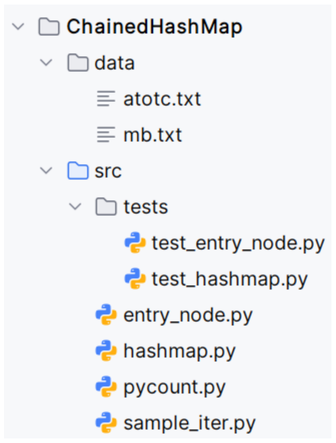
NOTE: Several questions are posed throughout the rest of this document. Enter your answers into a file called `answers.txt` *and submit it with your code.*

2.1 Starter code

Download the starter code from here.

Extract the zip file, and then copy the `data` and `src` directories into the root of your project folder in PyCharm. To make PyCharm aware the `src` directory has source code, you should right click on the `src` directory and select **Mark directory as... Sources root**. The `src` directory should be blue color.

Your project structure should look as follows:



```
ChainedHashMap
    data
        atotc.txt
        mb.txt
    src
        tests
            test_entry_node.py
            test_hashmap.py
        entry_node.py
        hashmap.py
        pycount.py
        sample_iter.py
```

2.2 Design

The documentation for the modules `entry_node.py` and `hashmap.py` you will write is available `here`. Your data structure design is required to look like the diagram in Figure 1. That is, you build a chained hash table that contains a sequence of list nodes at each location. Each unique key that hashes to the same location is simply added to that list with its associated value. You may use the course version of a hash map for reference, but for the best learning experience, you should code your hash map class from scratch.

Note that you have to write an iterator for your hash map to iterate over the entries `(key, value)`. The file **src.sample_iter.py** contains examples to show you how to do it. There are many online resources as well.

2.3 Design Constraints

Methods running time:

1. The `add`, `contains`, `get` and `remove` methods should all run in $O(1)$ time (assuming not much clustering due to excessive collisions). This means no linear searches besides the small chains of entries at specific "bucket" locations in the hash table.
2. The `len` method must run in $O(1)$.
3. The iterator, `str`, and `repr` methods are of course linear.

You are not allowed to use dictionaries or any built-in data structure from the Python library except for the list to implement the basic hash table. The data elements must be stored using linked nodes.

Basically you need to follow the design exemplified in **Figure 1**.

Here is the suggested order of implementation of all the `Hashmap` methods:

- `__init__` and `__len__` methods
- `__str__` and `__repr__` methods
- `__iter__` method
- `contains` method
- `get` method

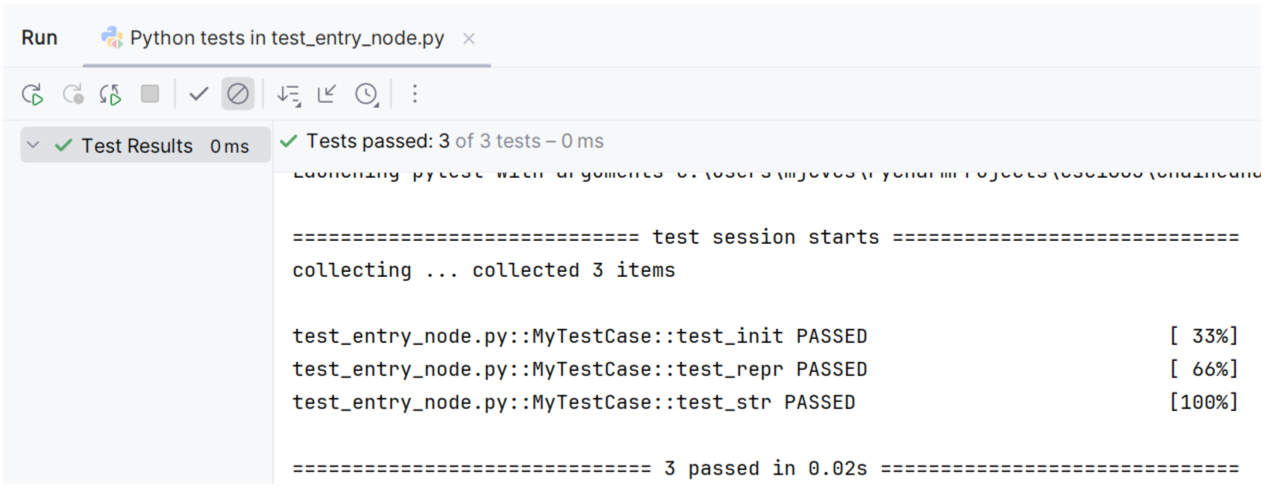
- `add` method
- `remove` method

2.3.1 Testing

Thorough testing is critical for any data structure development, as there will be special cases that your code will have to consider. A significant portion of your grade will be given for a good test suite.

There is a `test` package in the starter code that has 2 unit tests for the classes you must implement, `test_entry_node.py` and `test_hashmap.py`. Those unit tests have been implemented using the Python testing framework, `unittest`. You can find more information about this framework here.

The `test_entry_node.py` module is already complete. Once you finish the implementation of the `entry_node.py` module, run this unit test. Your goal is to make sure you pass all the tests, e.g.


Run
Python tests in test_entry_node.py
Test Results 0 ms
Tests passed: 3 of 3 tests – 0 ms

```
Launching pytest with arguments C:\Users\mjeves\PycharmProjects\[illegible]

============================= test session starts ==============================
collecting ... collected 3 items

test_entry_node.py::MyTestCase::test_init PASSED                         [ 33%]
test_entry_node.py::MyTestCase::test_repr PASSED                         [ 66%]
test_entry_node.py::MyTestCase::test_str PASSED                          [100%]

============================== 3 passed in 0.02s ===============================
```

The `test_hashmap.py` module is however half implemented. Your job is to complete this unit test by adding 3 additional test methods that verify non-trivial test cases scenarios. Each test method must be distinct from the others in terms of what it demonstrates. This test unit contains two finalized test methods available for you to use for adding test functions. This file should be thoroughly commented to explain what each test is testing.

You have also been provided with a program called `pycount.py` which counts the number of times each different word appears in a given file. Make sure your hash map works with this program. You can run it using the provided files `mb.txt` and `atotc.txt` as inputs.

2.4 Comparing Hash Functions

This section involves measuring the original hashing, in Problem Solving question 1, the improved version in Problem Solving question 3, and answering 2 questions. To make the measurements, we need to prevent the hash table from resizing. **Make sure to comment**

out the rehashing code from the `add` operation before answering the questions below.

Add to your hash map a function named `imbalance` to test how well a hash function works. The function must compute the average length of all non-empty chains and then subtract 1. A perfectly balanced, N-entry table would have an imbalance number of 0; the worst case imbalance value would be $N - 1$, where N is the number of elements currently in the hash map.

- **Question 1**: What is the imbalance measurement value for the original *string* hash function from the question 1 of the problem solving session that simply adds the values together, when you run `word_count.py` using your hash map on the files `mb.txt` and `atotc.txt` and for each file use the table sizes of 100 and 1000?

Rerun the program so that it computes the hashcode of a string using the improved formula. You can simply provide the hash table with a different hash function when it is created. Recall that the formula for a string `s` is given by:

$$\sum_{\texttt{i=0}}^{\texttt{len(s)-1}} \texttt{ord(s[i])*31}^{i} = \texttt{ord(s[0])}+\texttt{ord(s[1])*31}+\ldots+\texttt{ord(s[len(s)-1])*31}^{\texttt{len(s)-1}}$$

- **Question 2**: What is the imbalance value for the *improved* `hash_function` given above (i.e. the function from the PSS question 3)? Again, use table sizes of 100 and 1000.

Using any built-in hash function for answering these questions is forbidden.

3 Grading

- 20% Problem Solving
- 10% Design
- 50% Functionality:
  - EntryNode: 5%
  - HashMap: 45%
    - 9% constructor and others
    - 5% each: `contains, get, iterator, imbalance`
    - 8% each: `add, remove`
- 10% Testing
- 5% Answer Questions.
- 5% Style: Proper commenting of modules, classes and methods (e.g. using docstring's).

4 Submission

Create a ZIP file named lab7.zip that contains all your source code **(hashmap.py, entry_node.py, test_hashmap.py)** and the `answers.txt` file. Submit the ZIP file to the MyCourses assignment before the due date (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this lab).