FIT1008 Introduction to Computer Science (FIT2085: for Engineers)

Interview Prac 3 - Weeks 11 and 12

Semester 2, 2019

Learning objectives of this practical session

To be able to implement and use hash tables in Python.

Important requirements

For this prac, you are required to:

- **Create a new file/module for each task named `task[num].py` (that is, `task1.py`, `task2.py`, and so on).**
- **Provide documentation for each basic piece of functionality in your code (see below)**
- **The documentation needs to include (in the docstring):**
  - **pre and post conditions**
  - **Big O best and worst time complexity**
  - **information on any parameters used**
- **Provide testing for those functions/methods explicitly specified in the questions. The testing needs to include:**
  - **a function to test it**
  - **a comment indicating the purpose of each test**
  - **at least two test cases per test function. These cases need to show that your functions/methods can handle both valid and invalid inputs.**
  - **assertions for testing the cases, rather than just printing out messages**

  Your tests should be added to the corresponding `test_task[num].py` file provided
- **VERY IMPORTANT: As before the code cannot use any of the Python functionality that would make the job trivial (like dictionaries), but this time your are allowed to use any Python list or string functionality you want.**

Supporting Files

To get you started, you will find some supporting files on Moodle under Week 11:

- `HashTable.py`, `Freq.py`: Skeleton files for the `HashTable` and `Freq` classes, which you may use as starting points.
- `test_prac3.py`, `test_task[num].py`, `test_common.py`: Testing harnesses for each of the tasks.
- Additional text files for testing your dictionaries (and used by the testing harness).

Your code will be tested with an extended version of this harness, so **make sure your code can run inside the harness**. To make the testing work, we have made assumptions about the naming of instance variables, so please follow the naming used in the skeletons. Your modules should guard any computation/printing/etc. for the tasks with `if __name__ == '__main__':`, so they can be safely `import`ed by the testing harness.

The harness you have been provided is not exhaustive, so don't rely on it to ensure correctness! Remember to add your own tests, as well.

Important: Checkpoint for Week 11

To reach the check point you must complete Tasks 1, 2 and 3 by the end of your lab in Week 11, and submit all files corresponding to these tasks **before leaving the lab**. Remember: reaching the checkpoint is a **hurdle**.

Task 1 6 marks

Define a class `HashTable` that implements a hash table (accepting only strings as keys), using Linear Probing to resolve collisions. As in Prac 2, you should use a Python list to represent the underlying array. Include implementations for the following methods:

- `__init__(self, table_capacity, hash_base)`: Creates a new hash table, with initial table capacity `table_capacity`, and with using base `hash_base` for the hash function (see `hash` below). If `table_capacity` or `hash_base` are not specified, the hash table should use appropriate default values.
- `__getitem__(self, key)`: Returns the value corresponding to `key` in the hash table. Raises a `KeyError` if `key` does not exist in the hash table. Remember that this can be called from Python code as `table[key]`
- `__setitem__(self, key, value)`: Sets the value corresponding to `key` in the hash table to be `value`. If the hash table is full and the `key` does not exist in the table yet, it first calls the `rehash` method (see below) and then reinserts the `key` and `value`. Called from Python as `table[key] = value`
- `__contains__(self, key)`: Returns `True` if `key` is in the table and `False` otherwise.
- `hash(self, key)`: Calculates the hash value for the given `key`, using the uniform hash function specified in lecture 27 (page 35) with the base `hash_base` given on table creation.
- `rehash(self)`: It first creates a new array using as size the smallest prime number in the `Primes` list below that is larger than twice the current size. It then updates the size of the hash table and reinserts all key-value pairs in the old array into the new array using the new `size`. Raises a `ValueError` if there is no such prime in the list. For now, this method is only called when the table is full and we want to insert an element.
  `Primes` = [3, 7, 11, 17, 23, 29, 37, 47, 59, 71, 89, 107, 131, 163, 197, 239, 293, 353, 431, 521, 631, 761, 919, 1103, 1327, 1597, 1931, 2333, 2801, 3371, 4049, 4861, 5839, 7013, 8419, 10103, 12143, 14591, 17519, 21023, 25229, 30313, 36353, 43627, 52361, 62851, 75521, 90523, 108631, 130363, 156437, 187751, 225307, 270371, 324449, 389357, 467237, 560689, 672827, 807403, 968897, 1162687, 1395263, 1674319, 2009191, 2411033, 2893249, 3471899, 4166287, 4999559, 5999471, 7199369]

The file `test_task1.py` defines tests for some of the `HashTable` methods. But be warned: these tests are not comprehensive, and it is in your interest to think about what kinds of inputs we have not covered. **Important:** you must add your own tests for the `__setitem__` and `rehash` methods.

Task 2 5 marks

- Write a function `load_dictionary(hash_table, filename, time_limit)` that reads a file `filename` containing one word per line, and adds each word to `hash_table` with integer 1 as the associated data. `time_limit` is an optional argument – if it is specified and loading the dictionary takes more than `time_limit` seconds, the function should immediately raise an exception.
  Important: you must add your own tests for the `load_dictionary` method, using one or more very small dictionary files to do so.
- Write a function `load_dictionary_time(hash_base, table_size, filename, max_time)` that creates a new hash table with base `hash_base` and `table_size`, and returns the tuple `(words,time)`, where `words` is the number of (distinct) words added to the table, and `time` is the time taken (in seconds) for `load_dictionary` to complete if less or equal than `max_time`, and `None` otherwise.
- Download from Moodle the dictionary files `english_small.txt`, `english_large.txt` and `french.txt`. Write a Python function `table_load_dictionary_time(max_time)` that does the following: for each of these dictionaries and each combination of the values specified in the table below for TABLESIZE and `b` in the universal hash function, uses `load_dictionary_time` to time how long it takes for `load_dictionary` to run and prints a line to file `output_task2.csv` containing the name of the dictionary, the TABLESIZE and `b` used, the number of words added to the hash table, and either the time taken to run `load_dictionary` for that combination, or `TIMEOUT` if the time exceeded `max_time`.

b	TABLESIZE
1	250727
27183	402221
250726	1000081

  Note that the above table gives a total of 3*3=9 combinations per dictionary file. Nested for loops would be a good way to implement these combinations, and you are allowed to use "in" for this.

- Execute `table_load_dictionary_time(max_time)` for `max_time = 120` (i.e., 2 minutes) and use the information in the newly created file `output_task2.csv` to graph the time taken for each file combination (i.e., a graph where the `x` axis has the 9 combinations, the `y` has the time, and each combination shows 3 columns, one per file – an example is given at the end of the prac sheet), using the same methodology as you used in the code review prac of week 5 (giving `TIMEOUT` the value `max_time+10`). Create also an `explanation_task2.pdf` file that contains this graph together with a short analysis regarding what values work best and which work poorly, and an explanation of why these might be the case.
  IMPORTANT: The dictionary files use the `utf-8` encoding. Depending on your system configuration, you may need specify this when you open the file (e.g. `open(a_file, 'r', encoding='utf-8')`).

Note: If you are playing around with no time out, you can use Ctrl+c to stop execution of your program if some combination of values takes too long.

Task 3 4 marks

When discussing open addressing, we introduced two concepts: that of *collision*, which occurs when we attempt to insert a key, but the location for that key is already occupied by a *different* key); and that of *probe chain*, which is the sequence of positions we inspect before finding an empty space.

Consider inserting keys s_1, s_2 and s_3 into an empty table with linear probing, assuming all three hash to location 10. When we insert s_1, location 10 is empty, so there is no conflict. If we then insert s_2, location 10 is full, but 11 is empty. This is a conflict, with probe chain length 1. When we insert s_3, 10 and 11 are full, but 12 is free. This is one conflict, with probe chain length 2.

- Extend your `HashTable` class with a method `statistics(self)`, which returns a tuple `(collision_count, probe_total, probe_max, rehash_count)`, consisting of: (a) the total number of collisions, (b) the total length of the probe chains, (c) the length of the longest probe chain, and (d) the number of times `rehash` has been called.
  Important: you must add your own tests for your `statistics` method.

- Write a function `load_dictionary_statistics(hash_base, table_size, filename, max_time)` which does the same as `load_dictionary_time` (from Task 2), but returns the following tuple: `(words, time, collision_count, probe_total, probe_max, rehash_count)`, where the tuple elements represent the same information described above.

- Using the function above, write a function `table_load_dictionary_statistics(max_time)` that is similar to that in Task 2, but also prints the four counters above in the `output_task3.csv` file.

- Execute `table_load_dictionary_statistics(120)` and create an `explanation_task3.pdf` file that contains the associated graph and a comment regarding the relationship between these counters and the runtime, and regarding the length of the longest probe chain and the promise of O(1) time complexity. Explain why `rehash_count` is 0 in all runs (it should be!) Again, remember to include your csv and pdf files in your submission. **Hint:** To avoid duplicating this code later in the prac, you may wish to pass the HashTable *class* as an optional argument to `load_dictionary_statistics`.

CHECKPOINT

(You should reach this point during week 11)

Task 4 3 marks

Modify your hash table from the previous task to use Quadratic Probing to resolve collisions. Repeat the analysis from Task 3 on your Quadratic Probing hash table, and add to your `explanation_task4.pdf` file the graph associated with the table in `output_task4.csv` together with a comparison of the running time and reported statistics against those found when using Linear Probing.

Hint: If you followed the previous hint, you should not need to modify `load_dictionary_statistics`.

Task 5   4 marks

Implement a hash table that uses Separate Chaining to resolve collisions, where the linked data structure used is a Binary Search Tree (we will see this data type in Week 11, and you will be provided with an implementation of the Binary Search Tree, so do not worry about this just yet).

Note that the content of the array cells should initially be `None`. Whenever you need to add a `(key,data)` pair in that cell, you should create an empty binary search tree (say `my_tree = BinarySearchTree()`), then insert the `(key,data)` into `my_tree`, and then put `my_tree` in the cell.

Once this is done, compare again the runtime and value of the counters obtained in `explanation_task5.pdf` for Separate Chaining against those obtained for Quadratic and for Linear Probing.

Background

In most large collections of written language, the frequency of a given word in that collection is inversely proportional to its rank in the words. That is to say: the second most common word appears about half as often as the most common word, the third most common word appears about a third as often as the most common word and so on[1].

If we count the number of occurrences of each word in such a collection, we can use just the number of occurrences to determine the approximate rank of each word. Taking the number of occurrences of the most common word to be `max` and the relationship described earlier, we can give a *rarity score* to each word:

- Any word that appears at least `max`/100 times is considered to be common (score 0).
- Any word that appears less than `max`/1000 times is considered to be rare (score 2).
- Any word in between (less than `max`/100 and greater or equal to `max`/1000) is considered to be uncommon (score 1).
- Any word which has never been observed is a misspelling (score 3).

In the following we will use a hash table to facilitate a frequency analysis on a given text file.

Task 6   4 marks

Create a new class `Freq` that uses the Linear Probing hash table to perform a frequency analysis of a set of files as follows. First, add to this class the method `add_file(self,filename)`, which reads each word from the file into the hash table, in such a way that the data associated to the word is its "occurrence count", i.e., the number of times the word has been "added" to the hash table (which is the same as the number of times it has already appeared in the file). The class (and its methods) must keep track of the number of occurrences `max` for the most common word read.

Then, add to this class a method `rarity(self, word)` that given a word, returns its rarity score (an integer in the range $[0, 5]$).

We have provided ebooks of some classic texts from `https://www.gutenberg.org/` in the Prac folder. Use `Freq` to read the provided `1342-0.txt`, `2600-0.txt` and `98-0.txt` into your hash table. At the end of this process the table will store the number of times each word appears in the texts. We will consider the resulting hash table as a reasonably accurate representation of the frequency of words in the English language. Make sure you use the data you collected in the previous tasks to select an appropriate table size for the selected text files. Also, modify `__setitem__` to call `rehash` when the load of the hash table is above 0.5, rather than when full (this will not be used if you choose an appropriate table size, but is more realistic for cases where you do not know the number or size of the books being read).

Important: You must add your own tests for the `add_file` and `rarity` methods.

Task 7   2 marks

Add to the `Freq` class a method `evaluate_frequency(self, other_filename)` that returns a tuple `(common, uncommon, rare, errors)` of the *percentage* of words appearing in `other_filename` which have the corresponding rarity (with respect to the current frequency counts).

[1] This is known as Zipf's law. You can read more at `https://en.wikipedia.org/wiki/Zipf%27s_law`

Use `evaluate_frequency` to compute the frequency of each rarity score in `84-0.txt`, using the expected frequencies you constructed in Task 6.

Example Plot

This is an example of the style of plot you should be generating: