FIT1008 Introduction to Computer Science (FIT2085: for Engineers)

Interview Prac 3 - Weeks 11 and 12

Semester 1, 2019

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].**
- **Provide documentation for each basic piece of functionality in your code (see below)**
- **The documentation needs to include:**
  - **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**
  - **at least two test cases per test function. These cases need to show that your functions/methods can handle both valid and invalid inputs (if it takes inputs).**
  - **assertions for testing the cases, rather than just printing out messages**
- **VERY IMPORTANT: As before the code cannot use any of the Python functionality that would make the job trivial (like dictionaries, slicing, etc). You need to implement the functions yourself by going through the array, as we did in the lectures.**

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 python files corresponding to these tasks. Remember: reaching the checkpoint is a **hurdle**.

Task 1 6 marks

Define a class `HashTable` implementing a hash table, using Linear Probing to resolve collisions. As in Prac 2, you should use a 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 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 universal 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, 30293, 36353, 43627, 52361, 62851, 75431, 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]

We have provided a file `TestHashTable.py` which 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 define your own tests for the `__setitem__` and `rehash` methods.

Task 2 5 marks

- Write a function `load_dictionary(hash_table, filename)` which reads a file `filename` containing one word per line, and adds each word to `hash_table` with integer 1 as the associated data.
  Important: you must define tests for the `load_dictionary` method. You should use one or more small dictionary files to do so.
- Download from Moodle the dictionary files `english_small.txt`, `english_large.txt` and `french.txt`. For each of these dictionaries, use a methodology similar to that studied in the Workshop of week 5 to time how long it takes for `load_dictionary` to run. Do this for each combination of the values specified below for TABLESIZE and `b` in the universal hash function (giving a total of 3*3=9 combinations per file – nested for loops would be a good way to implement this – you can use "in" for this).

b	TABLESIZE
1	250727
27183	402221
250726	1000081

  Present the times recorded in a table. Write a short analysis regarding what values work best and which work poorly. Explain why these might be the case. Be sure to include your table and explanation in `task_2.py` (as a multi-line string).

Note: You can use Ctrl+c to stop execution of your program if some combination of values takes too long. If a single combination of values takes more than 3 minutes to complete, you may report it as `TIMEOUT` (but think about the reasons why!).

Task 3 4 marks

When discussing open addressing, we introduced several concepts:

- A *collision* occurs when we attempt to insert a key, but the location for that key is already occupied by a *different* key.
- A *probe chain* is the sequence of positions we inspect before finding an empty space.

Consider inserting keys s_1, s_2 and s_3 (which all hash to location 10) into an empty table with linear probing:

- 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 define appropriate tests for your `statistics` method.

Once the hash table has been modified, repeat Task 2 printing not only the run times but also the four counters above, and comment on the relationship between these counters and the runtime. Also, comment on the length of the longest probe chain and the promise of O(1) time complexity. `rehash_count` should be 0 in all runs – explain why. Again, remember to include your table and explanations in your submission.

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. Compare the running time, and the value of the four counters obtained when loading each dictionary with Quadratic Probing against those found when using Linear Probing.

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 (if you want to store the number of elements per chain – up to you – then you should use instead something like a tuple that starts as (None,0), where 0 is the initial number of elements). Once 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 (or in the tuple as `(my_tree,1)`.

Once this is done, compare again the runtime and value of the counters obtained 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, implement 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

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

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, implement a method `rarity(self, word)` that given a word, returns its rarity score.

Download some ebooks as text files from `https://www.gutenberg.org/` and use `Freq` to read these files 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 define tests for the `add_file` and `rarity` methods.

Task 7 2 marks

Download another ebook from `https://www.gutenberg.org/` and using the frequencies of the files read by Freq in Task 6, print the percentage of common, uncommon, rare and misspelled words for the given ebook.

Task 8 2 marks

Add a `delete` method to your Linear Probing hash table. This function is to take `key` as input and then delete the entry corresponding to that key, reinserting all elements from the position where the key appears, until the first empty position. Raise a `KeyError` if `key` is not in the hash table. And please reuse (do not copy and paste) as much functionality of the `rehash` method as possible

Important: You must define appropriate tests for the `delete` method.