FIT1008 – Intro to Computer Science
Assessed Prac 3 – Weeks 11 and 12

Semester 1, 2017

Objectives of this practical session

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

- **You should provide documentation and testing for each piece of functionality in your code. Your documentation needs to include pre and post conditions, and information on any parameters used.**
- **Create a new file/module for each task or subtask.**
- **You are not allowed to use the Python built-in `dict` methods.**
- **Name your files task[num] to keep them organised.**

Testing

For this prac, you are required to write:

(1) a *function to test* each function you implement, and

(2) *at least two test cases* per function.

The cases need to show that your functions can handle both valid and invalid inputs.

Marks

For this assessed prac, there are a total of 30 marks. There are 10 marks allocated to your understanding of the solutions and your implementations in the prac overall. In addition to these, the marks for each task are listed with the tasks. A marking rubric is available online for you to know and understand how you will be marked.

Task 1 [7 marks]

Implement a complete version of a hash table using Linear Probing to resolve collisions. Include implementations for the following 4 functions:

- `__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. Called by `table[key]`
- `__setitem__(self, key, value)`: Sets the value corresponding to `key` in the hash table to be `value`. Raise an exception if the hash table is full and the `key` does not exist in the table yet. Called by `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`. Use the universal hash function given in the lectures.

Task 2 [6 marks]

Download from Moodle the dictionary files `english_small.txt`, `english_large.txt` and `french.txt`. For each of these dictionaries, time how long it takes to read all words in it into the hash table. Do this for each combination of the values for table size and `b` in the hash function specified below.

b	Table size
1	250727
27183	402221
250726	1000081

Note: You can use Ctrl+c to stop execution of your program in case some combination of values takes too long. In total you should consider 9 possibilities (3 Table sizes for each value of b). Here you can treat both the key and the data, as a word.
Present the times recorded in a table. Write a short analysis regarding what values work the best and which work poorly. Explain why these might be the case.

Task 3 [3 marks]

Modify your hash table implementation to now track the number of collisions. Repeat Task 2 and comment on the relationship between the number of collisions and runtime.

CHECKPOINT

(You should reach this point during week 11)

Task 4 [3 marks]

Modify your hash table from the previous tasks to use Quadratic Probing to resolve collisions. Compare the number of collisions and running time found when loading each dictionary against the number found using the Linear Probing hash table.

Task 5 [3 marks]

Implement a hash table which uses Separate Chaining to resolve collisions. Again, compare the number of collisions found when loading each dictionary against both Quadratic and Linear Probing hash tables. In this variation, we will use a Binary Search Tree instead of a Linked List to support the separate chaining. It is a good idea to implement and test the Bonary Tree Search separately, first.

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 that 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].

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

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 assume that any word that appears at least `max`/100 times appears in the top 100 words and is therefore common. The same can be applied as `max`/1000 times for the next 900 words, rounding out the top 1000 words, considered to be uncommon. All words that appear less than `max`/1000 times are considered to be rare.
In this prac we have been implementing hash tables and so we will use one here to facilitate a frequency analysis on a given text file.

Task 6 [4 marks]

Download some ebooks as text files from `https://www.gutenberg.org/` and use these files as a collection of English. Read these into your Quadratic Probing hash table to count the number of each word in the texts. Considering the data you collected in Task 2, select an appropriate table size for the text files. Use these occurrence counts to construct a table which can be used to look up whether a given word is common, uncommon or rare within written English.

Task 7 [2 marks]

Download another ebook from `https://www.gutenberg.org/` and using your table from Task 6, analyse the percentage of common, uncommon and rare words. If a word is not found in the table, consider it misspelled. Print the percentage of common, uncommon, rare and misspelled words for the given text.

Task 8 [2 marks]

Add a `delete` function to your Quadratic Probing hash table. This function is to take `key` as input and then deletes the entry corresponding to that key. Raise a `KeyError` if `key` is not in the hash table.
To facilitate the deletion of items, also implement a `rehash` function which takes `size` as input. This function changes the size of the table and reinserts all key-value pairs. Raise a `ValueError` if `size` < 1. This function can also be called to resize the hash table when required (this may be when it is full or some other time that you deem appropriate). Your class should be coded such that the supporting array is always at the most half full. When making the hash table larger, ensure that the sizes given are appropriate and continue the good performance of the hash table.