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

Semester 2, 2018

## *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.**

**Assessment: During your practical class in week 12**
**Submission: via Moodle, immediately after assessment**

## *Aim*

In this assignment, we will use hash tables to determine how common particular words are in the English language.

## *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** (*three or more if you want to achieve the excellent category*).

**The cases need to show that your functions can handle both valid and invalid inputs. There is no need to test menu functions.**

## *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 [10 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`. If the key does not exist and there is space in the table the key value pair should be added.
- `__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 hash function given below.

```python
def hash_value(self, key):
        prime_mult = 101
        result = 0
        for character in key:
                result = (result * prime_mult + ord(character)) % self.table_size
        return result
```

## *Task 2 [3 marks]*

Modify your hash table implementation to now track the number of collisions[1], the load, as well as the average probe length[2].
Using collisions, and probe length, choose appropriate values of *prime_mult* (in your hash function) and `table_size`. You want to find values that perform well across the three files (*english_small.txt, english_large.txt, and french.txt*). For this task use a maximum table size of 400000.
You should try at least 10 values, and explain your choice by presenting all data behind your reasoning recorded in a table [3].

[1] a collision is simply when an item *hashes* to an already filled position

[2] the probe length of an item is the number of positions to consider in order to place/find that item.
Average Probe length is hence the sum of this over all items divided by the number of items on the table

[3] Just for fun we'll collect the best performer in each lab class

## CHECKPOINT

(You should reach this point during week 11)

## *Task 3 [6 marks]*

Modify your hash table from the previous tasks to:

- use Quadratic Probing to resolve collisions.
- implement dynamic hashing, by doubling the size of the underlying array (and rehashing) every time the load exceeds $\frac{2}{3}$

Write up a paragraph comparing the number of collisions, probe length and running time found when loading each dictionary against the best combination found using the Linear Probing hash table.

## *Task 4 [5 marks]*

Implement a hash table which uses Separate Chaining to resolve collisions. Use your implementation of a linked list from the prior assessed prac for this. Write up a paragraph comparing the performance of Separate Chaining against linear probing in this case.

## *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[4].

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.

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

## *Task 5 [6 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. Use a table size of 399989 for the text files and 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 6 – Just for fun [0 marks]*

Implement the program in Task 3 using Python's `dict` class. How does the performance of your own implementations compare to Python's in terms of the wall time? What factors may drive the differences? How close can you get to the reference implementation?