159.172 – Assignment 3 - Hash tables & performance evaluation

Worth 20% Due by 11pm Monday November 13

Part 1 - Hash Table Implementation – 7 marks

You're going to write a key-value store (hash table) using Python lists. ***Python dictionaries MUST NOT be used to implement the hash table for Parts 1 & 2.***

Hash Table Refresher: A hash table is used to store key-value pairs where the location to store the data (often called a ***slot***) is calculated from the key using a function (the *hash* function). The function takes the key and returns a number. This number is used as an index into in a table (or list) that's used to store the data. However, the hash function sometimes returns the same index for different keys – this is called a *collision* – so you need to store the ***key** and **the value*** in the table so you can ensure the value you return is associated with the correct key.

A good hash function gives different indexes for similar data, which results in the keys being distributed evenly over the hash table, whereas a poor hash function will result in many keys pointing to the same location (collisions), and can leave parts of the table empty.

Collisions are usually handled in one of two ways:.

1. **trying a sequence of alternate locations:** incrementing the initial index h giving h+1, h+2 (linear increments), h+2, h+4, h+9 (quadratic increments) or a similar scheme, wrapping around to the start of the table when the index exceeds the table size.

2. instead of storing the key/value directly in the hash table location, **each hash table location contains a sublist.** All keys that map to a particular location/sublist simply have their key-value pair added to the sublist. This gives a very simple algorithm:

```
h = hash(key);
sublist = table[h];
Do a linear search to add or search sublist for the key
```

You're going to use the second method. The main hash table is a Python list with a fixed size of 1009 entries, as 1009 is smallest table with more than 1000 entries where the number of entries is prime (which can help with hashing). Each list entry contains a sublist of the key-value pairs that hash to that location.

To start with, the entries in the table will be set to the empty sublist, so the after the table is initialised, and has a few entries added

```
insert(2, 'two')
insert(3032, 'xxx')
insert(1011, 'a big number')
```

Using the simple hash function: **hash(key) = key % M** (where M is the size of the hash table), the hash table would looks like this

```
[ [], [], [(2, 'two'), (1011, 'a big number)], [], [], [(3032, 'xxx')],
  [] ... for another thousand or so entries ]
```

>>> **Validate that your hash table works** by inserting 5,000 random integers as keys, where the value is simply the *key+5*, then lookup the keys and confirm the value are correct,

>>> **For 10 randomly chosen keys, display the key and the retrieved value,.**

Part 2 – Evaluating hash table performance with strings as keys - 9 marks

>>>> **Copy the previous program into a new file.**

>>>> **Modify it to use the following hash function, so strings can be used as keys.** This isn't a good hash function for strings as it simply adds the values of each character – so the keys **'ate', 'eat'** and **'tea'** all return the same hash value, but it gives us a place to start:

```
def hash(key, tableSize):
   if type(key) is str:
      # calculate a hash value by adding up all the characters
      total = 0
      char_count = 0
      for char in key:
         total = total + ord(char)     # ord() converts a character to a int
      return total % tableSize
   else:  # for integer keys
      return key % tableSize # the hash value used in the first problem
```

>>>> **Modify the *lookup(key)* function so it returns a *work_done* value,** reflecting the number of comparisons it needed to find a particular key.

>>>> **Add some code to return the following statistics** that shows the *in-use* and *empty* slots in the hash table, the total amount of work done in looking up ALL the keys, and from that calculate the work per key:

```
In use slots       : 293
Empty  slots       : 716
max sublist length: 599

No of keys    : 23376
Total Work    : 4263683.0
Work per key  : 182.4
```

Here, you can see what happens a really terrible hash function is used (much worse than the one above). If the keys were evenly distributed, 23276 keys spread over 1009 slots would result in around 23376/1009 → 23 entries in each sublist, whereas about 70% (716/1009) of the table is unused, and the maximum sublist length is nearly 30 times the ideal.

>>> **As a source of words to use as keys for testing, download the file *unique-words.txt* from Stream and add it to your folder.** This file contains around 23,000 unique strings.

>>>> **Load the file unique-words.txt into a list so each word is a list item.**

>>> Create a new hash table with 1009 entries and add all the words using the above hash function.

>>> **Display the statistics for the above hash function.**

>>>**Create TWO different variations of the hash function *so that the order of the characters matters* (so *eat, ate* and *tea* return different hash values) and display the statistics for these variations.**

Part 3 – Recursive functions – 4 marks

Write recursive functions

1. **count(char, L)** that takes a character and a list of strings as parameters. The function returns a count of how many lists items contain that character:
   e.g. **`count('a', ['a', 'really', 'nice', 'day'])`** would return 3

2. **reversejoin(L)** that takes a list of strings and returns a single string made up of all the original strings joined together in the opposite order:

   ```
   s = reversejoin(['a','really','nice','day'])
   print(s)
   ```

   would display **`daynicereallya`**

3. **mpy(M,N)** takes two numbers M & N and returns their product (M*N) by recursively totalling M copies of N (i.e. by repeated addition NOT multiplication). Totalling N copies of M is also fine.

These functions must do their work WITHOUT using ***while*** or ***for*** loops.

What to submit on Stream

>>> **Submit your three programs (Parts 1, 2 & 3) along with samples of their output copied to a .txt .DOC .RTF or .ODT file.**