COSC122 (2020) Lab 5

Hashing

Goals

This lab will give you some practice with hashing and hash tables; in this lab you will:

- test a simple spell-checker that uses linear search;
- complete a spell-checker that uses the following hash table implementations;
  - chaining
  - linear probing
  - quadratic probing
- Investigate how the hash table load factor affects performance.

You should be familiar with the material in Section 5.2.3 - Hashing [1] of the textbook before attempting this lab.

Important Note

To get the most from this lab you really need to carefully work through this handout doing each task in order. If you jump sections then your code may not work as expected, for example, in the chaining hash table we start with a simple hashing function and then fix it to get better results. If you jump ahead then you miss the point where it is fixed and it will take a very long time to do trial runs! Also, some doc-tests will fail initially and we explain where this is ok.

Hashing

The goal of hashing and hash tables is to produce a collection with $\mathcal{O}(1)$ performance by reducing items to a *hash* value and storing them in a table of possible hash values. There are three main components to this process:

1. the *hash table*,
2. the *hash function*, and
3. the method for dealing with hash collisions.

Spell Checking

Automated spell checkers compare every word in a document to the words in a pre-defined dictionary. [2] While this may seem like a trivially easy task, the challenge is in making the process *fast*; there are over 600,000 words in the *Oxford English Dictionary*, and many more that are used every day, but aren't on that list—grammatical variants, jargon and technical terms, slang, foreign words, *etc.* Checking a single sentence could easily cost *millions* of comparisons if you're not careful.

The provided `spelling.py` module contains a few methods and stubs that will be used in this lab, such as:

[1] http://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html

[2] Modern spell checkers are a bit more advanced, but for this lab, we'll keep things simple.

- `load_dict_words`: loads the words from a dictionary file into a list.
- `load_doc_words`: reads the words (sequences of characters between A–Z or a–z) from a document, converts them all into lowercase, and returns them as a list.
- `ChainingHashTable`: an initially inefficient chaining hash table class.
- `LinearHashTable`: an open addressing hash table class with linear probing.
- `QuadraticHashTable`: an open addressing hash table class with quadratic probing.

There are several other files provided in the lab archive, which will be used as test input:

- `tester.py` contains some starter tests, that you will expand upon.
- `words_latin-1.txt`: a file containing over 610,000 words [3] - the dictionary we will use to check spelling.
- `sherlock.txt`: the first adventure of Sir Arthur Conan Doyle's *Memoirs of Sherlock Holmes*; ~1,000 words.
- `trgov.txt`: John Locke's *Two Treatises of Government*; ~56,000 words.
- `humnature.txt`: David Hume's *A Treatise of Human Nature*; ~225,000 words.

The easy, but slow, way

Complete the `spellcheck_with_list` stub method in the `spelling` module. The function receives lists of all words in the document and dictionary, and should go through every word in the document to see if it is also in the dictionary, using a naïve, linear method (ie, 'in' will be handy). The function should print out each word in the document that doesn't appear in the dictionary and keep a running total of the number of words that weren't in the dictionary, along with a set of words that were misspelled (ie, `unique_errors` is a set of words that were misspelled and therefore contains each word only once).

You can test your spell checker using the shell, as in the example below. But, it will be better to use the `tester.py` module that we provide and add in various tests of your own. You can then use Wing's *Debug* function to see the output as it happens, because full runs take some time.

```
>>> from spelling import *
>>> dictionary = load_dict_words("words_latin-1.txt")
>>> document = load_doc_words("sherlock.txt")
>>> spellcheck_with_list(document, dictionary)
-------------------------------------------------
Misspelled words:
-------------------------------------------------
pyland
favorite
...
```

Many of the "incorrect" words are actually correctly spelt (or the American spelling of) names and places! Even with just over 613,000 words, our dictionary is far from complete.

As you should observe, it takes quite a while to check the 10,000 words in `sherlock.txt`. Think about how long it would take to check the larger documents (with up to 225,000 words).

> ***Complete the* Sherlock *question(s) in Lab quiz Quiz 5.***

[3] From SCOWL: http://wordlist.sf.net/

The `ChainingHashTable` Class

If we kept the dictionary in a hash table, we could significantly reduce the number of comparisons it takes to check if a word is in it. (In theory! The actual efficiency in practice is dependent on the hashing function we use and the load factor.)

The `ChainingHashTable` class contains the implementation for a simple chaining hash table. The constructor creates a list of empty lists used to store the items. The outer list is sized to a certain number of slots (by default, 11); the inner lists contain the items for each slot (initially there is just an empty list in each slot). The `__repr__` method is implemented in the class and the example below shows a print of a simple hash table (note we have a *simple* hash function at the moment...):

```
>>> from spelling import ChainingHashTable
>>> hash_table = ChainingHashTable(5)
>>> hash_table.store("Paul")
>>> hash_table.store("Peter")
>>> hash_table.store("Paula")
>>> hash_table.store("David")
>>> hash_table.store("Bob")
>>> hash_table.store("Di")
>>> print(hash_table)

ChainingHashTable:
slot        0 = ["Paul", "Peter", "Paula", "David", "Bob", "Di"]
slot        1 = []
slot        2 = []
slot        3 = []
slot        4 = []
```

Notice how all the names are in slot 0. This is due to our currently very lazy `_hash` function that always returns 0. This is intended to demonstrate the worst case scenario for a chaining hash table.

You should now implement the `__contains__` method, this will allow you to use the `in` operator to see whether a given item is in the hash table, for example:

```
>>> print('Bob' in hash_table)
True
>>> print('Knob' in hash_table)
False
```

Spell Checking with `ChainingHashTable`

Complete the `spellcheck_with_hashtable(document, dictionary, ht_type, ht_size)` function. This function is similar to that for the simple list based search but it also includes a parameter for hash table type and hash table size. We provide the code that loads the dictionary words in to the hash table. You will need to insert the code to check the words in the document against the words in the hash table.

Once you have completed the function you can call it with calls such as:

- `spellcheck_with_hashtable(document, dictionary, 'Chaining', 11)`.

Stick to a hash table size of 11 for now - we will crank it up later when we look at the load factor.

After completing `spellcheck_with_hashtable`, test it in the same way that you did for `spellcheck_with_list`. You can, at this stage, ignore the doctest failure relating to the `ChainingHashTable`, we will fix this later... To start with you should spell check the `sherlock.txt` file as it is the shortest. Then try `trgov.txt`, this will take some time!

You should notice, however, that this doesn't seem any better than the linear method! The example print out of a chaining hash table above gives you an indication of what is going wrong. Are the items spread around the hash table evenly?

The `ChainingHashTable` class uses its `_hash` method to compute the hash value for items in the table; but at the moment it always returns 0—all items have the same hash value! This means that all words will be chained into the same slot, and when it comes to searching for an item, it will perform a linear search across that slot—effectively doing a linear search over the list of every single item that was inserted. So, now let's implement a more sensible hash function in `ChainingHashTable`.

Better Hash Functions

A perfect hash function transforms every item into a unique hash value. However, this usually isn't possible—it requires knowing what every item in the hash table will be before you write the function.[4] Instead, hash tables typically settle for a hash function that gives a *uniform distribution*—the hash values are spread across a large range of possible values.

In Python, defining a hash function for an object is done by adding a `__hash__` method that returns an integer.[5] Most built-in objects (numbers, strings, lists, *etc.*) already have an implementation of this method; it's called using the built-in `hash` function:

```
>>> hash("a")
-1008721619
>>> hash("Alpha")
1222515508
```

But, Python's inbuilt hash function doesn't necessarily return the same result on different runs (even on the same operating system) and it can return negative values that aren't much fun for list indexes. So, we have provided a `nice_hash` function that is platform independent and consistent across runs but still uses an algorithm very close to Python's inbuilt hash function.

```
>>> nice_hash("a")
3826102752
>>> nice_hash("Alpha")
2147701689
```

However, you can't just use the hash value as-is: the range of possible hash values spans over 4 billion integers—far more slots than will exist in our table. The hash values need to be scaled down to be in the range of the number of slots—typically done using the modulo operation (`%`). If using this method then it is wise to set the number of slots to a prime number, to minimise collisions and spread values more evenly.

Re-implement the `_hash` function in `ChainingHashTable` to return the `nice_hash` of the object, scaled to the number of slots in the hashtable.

[4] We *could* construct a perfect hash function for our dictionary of words, since it won't change over the course of this lab; but this typically isn't the case.

[5] If you do this you must also write an `__eq__` method that ensures `h(x) == h(y)` when `x == y`.

For the test example above (adding 'Paul','Peter','Paula','David','Bob' and 'Di', in that order, to a 5 slot hash table) should result in a hash table that looks like:

```
ChainingHashTable:
slot        0 = []
slot        1 = []
slot        2 = ["David", "Bob"]
slot        3 = ["Paul", "Peter", "Di"]
slot        4 = ["Paula"]
```

Notice the collisions at slots 2 and 3. In this example, collisions will be unavoidable given the number of items is greater than the number of slots. [6]

Test your `spellcheck_with_hashtable` method again, still using 'Chaining' and a hash table size of 11.

The Load Factor

Your spell checker should now be faster than the first version you implemented. However, you can still do much better.

The *load factor* of a hash table is a metric of how full the table is; it's calculated using the following formula:

$$\lambda = \frac{\text{number of items}}{\text{number of slots}}$$

A load factor of 0 would indicate an empty table, while a load factor greater than 1 would indicate a table that has more items than slots (and needs to chain them). Note that the load factor doesn't consider the hash function—you can have a low load factor and still have poor performance because your hash function doesn't distribute items well.

By default, our `HashTable` creates a table with 11 slots. If you calculate the load factor for our table when it's populated with the entire dictionary of words, you'll note that it's *incredibly* high. Assuming a perfect hash function, this means that it is creating chains of tens-of-thousands of items that are to be searched through!

You can adjust the number of slots in the table, by changing the parameter to the `HashTable` constructor or indirectly by changing the hash table size value that you send to `spellcheck_with_hashtable` with some more reasonable hash table sizes. Will a size of 23 help much? how about 101? Try some load factors under 1.0 —this will require more slots that words in the dictionary[7]. Try to find the sweet spot. Test out the other two documents—`trgov.txt` and `humnature.txt`—to ensure larger documents don't melt your computer.

Checking the spelling of a document should now be almost-instant. You will notice that the construction and population of the dictionary hash table is the most time-consuming task, spell checking the document takes less than a few seconds.

> ***Complete the* More bad spalling etc *question(s) in Lab Quiz 5.***

Dealing with Collisions

There are two main ways of dealing with hash collisions:

1. *Chaining*: A list is maintained at each hash slot with the list containing all items that hash to the slot.

[6] One benefit of chaining is that the hash table can hold more items than it has slots so you don't need to know how many items are to be added when you create the hash table and you won't get an error if you add large numbers of items - *until your computer runs out of memory.*

[7] see http://www.bigprimes.net/archive/prime/ for a selection of big prime numbers

2. *Open Addressing*: If a collision occurs then a *probe* is done to find another empty slot in the hash table. All items are stored in the slots of the hash table, therefore space is limited...

So far we have covered *chaining* and will now experiment with *open addressing*.

Open addressing

When using open addressing items are stored one per slot in the hash table, therefore the hash table cannot store more items than it has slots. For example, a five slot hash table that uses open addressing can store a maximum of 5 items. If an item hashes to a slot that already contains an item then we need to find another empty location in the table using a *probing* function. The hash plus probe sequence must be consistent so that a search will find the item. Depending on the hash table size, probing function and load factor an empty slot may not be located, even if the table isn't full! So it is important to get a feel for how open addressing works.

Linear Probing

If a collision occurs, increment the slot number and try that slot, continuing until an empty slot is found or the original slot is reached (which indicates the table is full). To avoid running off the end of the hash table use the modulo function to ensure the new slot number is less than the number of slots. For example the ith probe will be at $slot = (curr\_slot + i)\%table\_size$ - or the next probe will be at the current slot plus one. Our lab classes have an object variable called n_slots to make this easy for you.

Now is the time to implement the `store` method in the `LinearHashTable` class, so that the doc test returns no errors.

It is good to work through the addition of items in a step by step process, looking at the hash table as it grows. For example, running commands such as:

```
from spelling import *
hash_table = LinearHashTable(7)
hash_table.store("Tennis")
print(hash_table)
hash_table.store("Cricket")
print (hash_table)
hash_table.store("Swimming")
print (hash_table)
hash_table.store("Underwater Motorbike Hockey")
print (hash_table)
hash_table.store("Soccer")
print(hash_table)
```

> ***Complete the* Linear Probing *questions in Lab Quiz 5***

Once you are happy that you have your `LinearHashTable` class working, test your spell checking routine using `LinearHashTables` of size 650011, 700001, 800011, 2000003, 3000017, 5000011, 10000019. Then try using a size of 600011. What should happen here? Does your code deal with this?

- eg, `spellcheck_with_hashtable(document, dictionary, 'Linear', 2000003)`.

How does the speed change when the size of the hash table is increased?

Quadratic Probing

With quadratic probing, the sequence for probing is based on a quadratic function and in this lab we will use a simple quadratic function. That is, the ith probe will be at $slot = (h + i^2)\%table\_size$, where h is the original hash collision slot. So, if a clash occurred at slot 0 (in a hash table of size 101) then the sequence of slots to probe would be 1, 4, 9, 16, 25, 36,... With quadratic probing it helps to set the

number of slots in the hash table to a prime number, especially if you are going to have a high load factor. You should be aware that, for load factors over 0.5, quadratic probing isn't guaranteed to find an empty slot.

Now is the time to implement the `store` and `__contains__` methods in the `QuadraticHashTable` class, so that the doc test returns no errors. Then do some of your own testing by creating a hash table and adding in some items, printing out the table after each item is added.

Once you are happy that you have your `QuadraticHashTable` class working, test your spell checking routine using `QuadraticHashTable`'s of size 650011, 700001, 800011, 2000003, 3000017, 5000011, 10000019. Then try using a size of 600011. What should happen here? Does your code deal with this?

- eg, `spellcheck_with_hashtable(document, dictionary, 'Quadratic', 700001)`.

How does the speed change when the size of the hash table is increased?

> ***Complete the* Quadratic Probing *questions in lab Quiz 5***

Extras

- Instead of using `spellcheck_with_list`, write a function that uses a *binary search* of the dictionary, call it something entertaining like `spellcheck_bin` . How does it compare with the three hash table implementations? Will this always be the case?
- Change the `store` method of `ChainingHashTable` to keep the items in each chain sorted; adjust the `__contains__` method to take advantage of this. Given a load factor λ, and number of items n, what's the average number of comparisons needed to find an item using this method (assuming a perfect hash function)?