COSC122 (2020) Lab 3

Linked Lists

Goals

This lab will provide you with some practice with *Linked Lists*. In this lab you will:

- implement stack and queue data structures using linked lists and
- work with a linked list structure for counting characters

You should be familiar with the Linked List material in Chapter 3 of the textbook before attempting this lab (online link here).

The provided `linked_list_structures` module contains two skeleton classes: `Stack` and `Queue` that use a linked list to implement the interfaces for working with a stack and a queue. However, they're missing implementations for the most important methods: `push`, `pop`, `peek`, `enqueue` and `dequeue`!

Implementing a Stack and a Queue

Remember that a stack is 'last-in–first-out' (LIFO), and a Queue is 'first-in–first-out' (FIFO). The stack implementation should push and pop items from the head of the list; with the queue implementation it works out easiest if items are enqueued at the tail and dequeued from the head of the list. Complete the missing implementations for both the `Stack` and `Queue` classes in the `linked_list_structures` module. Finally you should implement the `__len__` method so that calling `len(x)` will work when `x` is a stack or a queue.

It is important to test the extreme cases such as deleting from a list of 1 item, adding to a list of 0 items etc. These cases are included in the doc tests. When you have completed your implementations you can test them with the doctests for each of the classes (by running the file in *Wing*) and by manually experimenting with the classes, for example:

```
>>> from linked_list_structures import Stack
>>> s = Stack()
>>> s.push('a')
>>> print(s)
List for stack is: a -> None
>>> s.pop()
'a'
>>> print(s)
List for stack is: None
>>> s.push('b')
>>> print(s)
List for stack is: b -> None
>>> s.push('c')
>>> len(s)
2
>>> s.peek()
'c'
>>> print(s)
List for stack is: c -> b -> None
>>> s.pop()
'c'
```

```
>>> from linked_list_structures import Queue
>>> q = Queue()
>>> q.enqueue('a')
>>> print(q)
List for queue is: a -> None
>>> len(q)
1
>>> q.enqueue('b')
>>> print(q)
List for queue is: a -> b -> None
>>> q.enqueue('c')
>>> print(q)
List for queue is: a -> b -> c -> None
>>> len(q)
3
>>> q.dequeue()
'a'
>>> print(q)
List for queue is: b -> c -> None
```

> ***Complete* Linky stacks and queues *questions in Lab Quiz 3.***

Now that you have working stack and queue data structures, you should make the enqueue method faster by adding a tail pointer, pointing to the end of the list. Open the module `queue2.py` and complete the queue class so that it uses both head and tail pointers. You should be able to copy most of your code over from your previous queue but you will need to re-write the `enqueue` and `dequeue` methods—we have supplied new doctests that will check your code deals with the head and tail pointers appropriately.

> ***Complete the* Heads and Tails *questions in Lab Quiz 3.***

Calculating Letter Frequencies

The next task is to read a text corpus and produce the statistics counting the number of times each character (or pair of characters) appeared in the text using a linked list. To see applications of this type of analysis, check out Frequency Analysis on Wikipedia. [1] Frequency information can be used to help with predictive text on cellphones, email completion in your email client, etc. For example, if someone types `'t'` what is the most likely next character? What is the most likely character after `'ther'` has been typed? etc...

The output from your program should look like the following (for letter pairs):

```
1:  't ' = 12345
2:  ' e' = 12222
3:  'th' = 12013
4:  'he' = 11987
... etc
```

Where the first number in each line is the index number of the letter (or group of letters) and the last number is the frequency (for example the first line has an index on 1, and shows that the letter pair `'t '` has a frequency of 12345). The index number shows you the order in which the items are stored in the linked list. The index number also shows the frequency rank in the case of the sorted frequency list. The example output is taken from a sorted frequency list and therefore the most frequent pair was `'t '` and the second most frequent was `' e'`.

The linked lists need to store two values in each node: an item (such as a character or pair of characters), and the number of times it has been seen in the text. Initially the linked list is empty.

Note: We will start by doing frequencies for characters but will move on to calculating frequencies for pairs of characters, so your methods should not assume we are always using single characters.

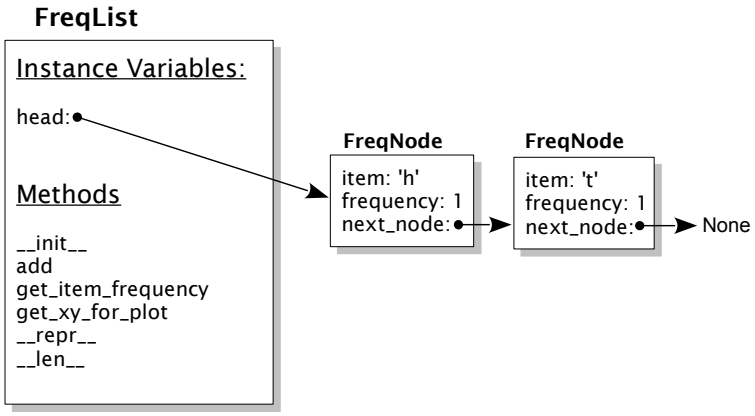

Figure 1: Simple object model overview

For each item in the text, your program should check whether it already exists in the list. If so, increment its frequency count by 1. If the item does not exist, then add a node to the linked list *(initially add at the start of the list as this will be the quickest way to add the node)* for that item and set its frequency count to 1.

Three sample text files are given in `le_rire.txt`, `ulysses.txt`, and `war_and_peace.txt` - listed from smallest to largest file-size. Tolstoy's War and Peace is one of the longest novels ever written, weighing in at around half a million words, but it is less than half the size of Proust's *In Search of Lost Time*, which has around 1.2 million words [2]. The `letter_frequencies` module provides some skeleton code. This

[1] http://en.wikipedia.org/wiki/Frequency_analysis

[2] It's too bad that *In Search of Lost Time* is not available on project Gutenberg.

module makes no distinction between UPPERCASE and lowercase letters.

Single Character Frequencies

Complete the add method in the `UnsortedFreqList` class in the `letter_frequencies` module. Run it on all the sample text files to provide the counts for each letter in each corpus (eg, to run on `le_rire.txt` you should uncomment `run_tests("le_rire.txt")` and un-comment the line in the `run_tests` function that adds the `UnsortedFreqList` test for single characters to the list of tests to run).

We recommend testing your programs fully with `le_rire.txt` before you move on to the longer files. You can ignore the `doctest` errors for other functions at this stage. Your output from a single character run, with `UnsortedFreqList`, should give you something like the left column of the following table.

```
--------------------------------------------------
Tests for: le_rire.txt
Doc size:  246941 chars
--------------------------------------------------
1 char(s) -> t = 0.8862s  (40 items)

Unsorted Frequency List
-----------------------------------
 1:  '@' = 2
 2:  '%' = 1
 3:  '/' = 26
 4:  '!' = 43
 5:  '"' = 262
 6:  ';' = 159
 7:  '?' = 89
 8:  'q' = 194
 9:  'x' = 387
10:  'z' = 24
11:  ''' = 112
12:  '*' = 28
13:  '#' = 1
14:  '.' = 1851
15:  'v' = 1897
16:  'd' = 6114
17:  'w' = 3824
18:  ',' = 3178
19:  'i' = 15917
20:  'm' = 5441
21:  'y' = 3461
22:  's' = 13342
23:  ':' = 181
24:  'a' = 15336
25:  'l' = 7836
26:  'f' = 4953
27:  'k' = 1001
28:  'b' = 2846
29:  'n' = 13878
30:  'u' = 5545
31:  'g' = 4055
32:  'c' = 6811
33:  'j' = 294
34:  'o' = 15142
35:  'r' = 11188
36:  'p' = 3679
37:  ' ' = 42837
38:  'e' = 24941
39:  'h' = 10597
40:  't' = 19468
```

```
--------------------------------------------------
Tests for: le_rire.txt
Doc size:  246941 chars
--------------------------------------------------
1 char(s) -> t = 0.4403s  (40 items)

Nicer Unsorted Frequency List
-----------------------------------
 1:  't' = 19468
 2:  'h' = 10597
 3:  'e' = 24941
 4:  ' ' = 42837
 5:  'p' = 3679
 6:  'r' = 11188
 7:  'o' = 15142
 8:  'j' = 294
 9:  'c' = 6811
10:  'g' = 4055
11:  'u' = 5545
12:  'n' = 13878
13:  'b' = 2846
14:  'k' = 1001
15:  'f' = 4953
16:  'l' = 7836
17:  'a' = 15336
18:  ':' = 181
19:  's' = 13342
20:  'y' = 3461
21:  'm' = 5441
22:  'i' = 15917
23:  ',' = 3178
24:  'w' = 3824
25:  'd' = 6114
26:  'v' = 1897
27:  '.' = 1851
28:  '#' = 1
29:  '*' = 28
30:  ''' = 112
31:  'z' = 24
32:  'x' = 387
33:  'q' = 194
34:  '?' = 89
35:  ';' = 159
36:  '"' = 262
37:  '!' = 43
38:  '/' = 26
39:  '%' = 1
40:  '@' = 2
```

Now, implement the add method in the `NicerUnsortedFreqList` class. This class should add new items to the end of the list rather than the beginning. Adding an item to the end of the list will take longer than adding to the start of the list but, as the name suggests, this works better overall. See the right column of the above output. Compare the speed of this nicer method to the speed of the original method.

Think about how many times new letters will be added to the frequency list in the course of processing a document and the position of characters such as `'t'`, `'h'` and `'e'` in the frequency list. Explain why inserting new letters at the end of the list vs. the beginning is likely to speed up corpus processing.

> ***Complete the* Simple character frequencies *questions in Lab Quiz 3.***

Character Pair Frequencies

Now that you have your code running for single characters, try running it for character pairs (by uncommenting `run_settings.append((UnsortedFreqList, 2, verbose))` for example). There should be a more noticeable gap between the `UnsortedFreqList` and the `NicerUnSortedFreqList`.

Complete the `SortedFreqList` to calculate the letter frequencies and store the nodes from highest frequency count to lowest. This is usually faster because the majority of characters being incremented

will be near the front of the list. To keep the list in order you will need to move nodes to their sorted position when needed. If a node has its count incremented and its count is now greater than the node before it then the node needs to be moved. It will obviously need to be before the previous node but it may need to go further as their may be many nodes with the same frequency as the previous node.

One way to move a node to its sorted position is to 'unlink' the node that needs to be moved then run through the list again, from the front, to find where the node should be moved to—we supply `_insert_in_order` that will help with this approach. Please read the comments in the `add` and `_insert_in_order` methods so you understand how and when to use the `_insert_in_order`, for example, you will never need to use `_insert_in_order` on an empty list as a node can't have higher frequency than its previous node if the list is empty.

Time your `SortedFreqList` implementation and compare it to the `UnsortedFreqList` and `NicerUnsortedFreqList` implementations. *Why is there only a small speed-up when compared to the `NicerUnsortedFreqList`?*

> ***Complete the* Sorted Frequencies *questions in web quiz 3.***

Extras

- To get a graphical representation of the frequencies use the `plot_freq_list` function provided in the source file. Only plot one graph per run - python will give you grief if you try more. We recommend only graphing the single character frequencies as the graph for character pairs will have far too many x-axis items! If you get a crash when running graphs you may need to restart the python shell in Wing, to do this simply click on the options button in the shell window and select restart shell...

- Implement a `FrequencyDictionary` that stores frequencies in a Python dictionary and see how well it stacks up versus the `SortedFreqList`. The timings should show why this the way most people would build the frequency values in the real-world.

- Write a function such as `make_sorted_words_freq_list` to calculate the frequencies of words in the documents. For simplicity assume a word is any set of characters between spaces, eg, in `'Wiggle and Woggle are two, very strange, dances.'` we would consider `'two,'` and `'strange,'` to be words, along with the more obvious words such as `Wiggle` and `are`. Basically using `.split()` will return the list of words that you want. But you may be able to process the files quicker if you scan through, character by character, building words and adding to the `freq_list` as you go—this saves building a huge list of words before you start processing.

- Another approach to implementing the `SortedFreqList` is to use a doubly linked list and move back through the list until the appropriate spot is found and insert the node there—this should be slightly quicker when moving nodes as it won't have to go all the way back to the start of the list each time. But, you will need to rewrite your nodes to have two links and make sure that both links are updated properly when operating on lists of them. Making a version that uses this approach is a good exercise for more motivated students. Take a copy of your code and change your implementation to use a doubly linked list. *Note: don't submit this code to the quiz question - use the singly linked list implementation (the quiz server will provide the Node class as given in the original lab code...).*