Project 4 CPE 202

Hash Table and Concordance (An application of hash tables)

There are two parts to this assignment:

- Implementing a hash table
- Implementing a concordance builder

Hash Table

Implement a hash table with quadratic probing, similar to what is described in the text book, with some additional functionality. Much of the functionality of the hash table is described in the starter files provided to you via GitHub. Your hash table must:

- Use **Open Addressing using quadratic probing** for collision resolution.
- Use strings as keys, and use the following Horner Hash function:
    - h(str) = $\sum_{i=0}^{n-1} ord(str[i]) * 31^{n-1-i}$ where n = the **minimum** of **len(str) and 8**
- Always use a prime number for the table size. You **may use algorithms/code from the web** to find the next prime number. I used the algorithm found here: https://www.geeksforgeeks.org/program-to-find-the-next-prime-number/
- Have the capability to grow if the load factor becomes too high. After insertion of an item, if the load factor exceeds 0.5, you should grow (and rehash) the hash table.

Note that the hash table implementation is separate from the Concordance. It must be a generic hash table, capable of inserting any kind of data along with a string key and must not implement any logic that is specific to the Concordance problem. It is strongly encouraged that you implement ***and test*** your hash table before moving on to the Concordance problem.

Word and Line Concordance Application

A Webster's dictionary definition of concordance is: "an alphabetical list of the main words in a work." In addition to the main words, your program will keep track of all the line numbers where these main words occur. The goal of this assignment is to process a textual data file to generate a word concordance with line numbers for each main word.

A Hash Table ADT is perfect to store the word concordance with the word being the key. This will allow fast lookup of words that occur multiple times in the file. Since the concordance should only keep track of the "main" words, there will be another file containing words to ignore, namely a **stop-words file** (stop_words.txt). The **stop-words file** will contain a list of stop words (e.g., "a", "the", etc.) -- these words will not be included in the concordance even if they do appear in the data file. You should also not include strings that represent numbers (e.g., "24" or "2.4" should not appear).

Sample stop-words file

```
a
about
be
by
can
do
i
in
is
it
of
on
the
this
to
was
```

Sample text file

```
This is a sample data ((text)) file, to be processed
by your word-concordance program!!!

A REAL data file is MUCH bigger. Gr8!
```

Sample result file

```
bigger: 4
concordance: 2
data: 1 4
file: 1 4
much: 4
processed: 2
program: 2
real: 4
sample: 1
text: 1
word: 2
your: 2
```

Notes:

1. Words are defined as sequences of characters delimited by any non-letter (whitespace, punctuation).
2. There is no distinction made between upper and lower case letters (CaT is the same word as cat)
3. Blank lines are counted in line numbering.

The general algorithm for the word-concordance program is:
1) Read the stop-words file into **<u>your implementation of a hash table</u>** containing the stop words. Start with a table size of 191 and let the table grow as described as described in the hash table implementation, if necessary. Note: You will use your hash table implementation for both the stop-words and the concordance. In the case of the stop-words, you just won't use the line number information (can either store the actual line number from the file, or just use a default value).

2) The word-concordance will be in a separate hash table from the stop words hash table. Process the input file one line at a time to build the word-concordance. This hash table should only contain the **non-stop words** as the keys (use the stop words hash table to "filter out" the stop words). Associated with each key is its **value** where the **value** consists of the line numbers where the key appears. **Do not include duplicate line numbers**. Also, make sure you find an efficient way to not include duplicate line numbers. (For example, do not use the "`in`" operator as this will do a linear search.)

3) Generate a text file containing the concordance words printed out **in alphabetical order** along with their line numbers. You may use Python's built-in sort method. One word per line (followed by a colon), and spaces separating items on each line:

```
data: 1 4
```

It is strongly suggested that the logic for reading words and assigning line numbers to them be developed and tested separately from other aspects of the program. This could be accomplished by reading a sample file and printing out the words recognized with their corresponding line numbers without any other word processing.

Removing Punctuation

- It is recommended that you process the input file one line at a time.
- For each line in the input file, do the following:
  - Remove all occurrences of the apostrophe character (')
  - Convert all characters in `string.punctuation` to spaces.
  - Split the string into tokens using the `.split()` method (with no parameters).
  - Each token that returns True when the isalpha() method is called should be considered a "word". All other tokens should be ignored.

Provided Data Files

- the stop words in the file `stop_words.txt`
- six sample data files that can be used for preliminary testing of your programs:
  - `file1.txt, file1_sol.txt` - contains no punctuation to be removed
  - `file2.txt, file2_sol.txt` - contains punctuation to be removed
  - `declaration.txt, declaration_sol.txt` – larger file for test
- six large data files that can be used for performance testing. DO NOT include any of your tests of these files in your submitted tests. They will take too long to run and will be killed by the grader.
  - `dictionary_a-c.txt, dictionary_a-c_sol.txt`
    - Should take 3-ish seconds to insert each word and call each O(1) method for each insertion
    - Should take 5-ish seconds for concordance
  - `file_the.txt, file_the_sol.txt`
    - Should take 3-ish seconds for concordance, but you MUST remove "the" from the stop words.
  - `War_And_Peace.txt, War_And_Peace_sol.txt`
    - Should take 7-ish seconds for concordance.
- The following starter files (containing the methods that you must implement) are available on GitHub. **Do not change the names of the classes, methods, or attributes specified in the starter files.**
  - hash_quad.py
  - concordance.py

SUBMISSION

Six files:

- **hash_quad.py**: containing class HashTable with all of the specified methods, using quadratic probing for collision resolution
    - You **MAY NOT** use a Python dictionary in this file
- **hash_quad_tests.py** – tests for the HashTable methods specified by the assignment
    - This file should contain tests only for the functions specified above and should only test functionality required by the specification. These tests must run properly on a valid instructor solution and will be tested to see if they can catch bugs in incorrect solutions.
- **hash_quad_helper_tests.py** – tests for any HashTable methods NOT specified by the assignment
    - This file should contain all tests for your solution, including tests for any helper functions that you may have used. These tests will be used to verify that you have 100% coverage of your solution. Your solution must pass these tests.
- **concordance.py**: containing class Concordance with all of the specified methods
- **concordance_tests.py** – tests for the Concordance methods specified by the assignment
    - This file should contain tests only for the functions specified above and should only test functionality required by the specification. These tests must run properly on a valid instructor solution and will be tested to see if they can catch bugs in incorrect solutions.
- **concordance_helper_tests.py** – tests for the Concordance methods specified by the assignment
    - This file should contain all tests for your solution, including tests for any helper functions that you may have used. These tests will be used to verify that you have 100% coverage of your solution. Your solution must pass these tests.