Project 4 (Hashing)

CSC 202 Spring 2020

Due: 6/6 @10:00 PM

Make sure solve the \r\n issue. Remove it or find a solution.

Concordance (an application of hash tables)

This assignment has several parts: implementing different versions of the hash table from the text (with some additional functionality) and writing an application that builds a concordance. 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.**

Word and Line Concordance Application

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 dictionary ADT is perfect to store the word concordance with the word being the dictionary key and a list of its line numbers being the associated value for the key. Since the concordance should only keep track of the "main" words, there will be a second 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 not include strings that **represent numbers**. E.g. "24" or "2.4" should not appear. There is no distinction between upper and lower case letters. (For simplicity convert all characters to lowercase letters.)

The following is an example and the output file.

Sample `stop_words.txt` file

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

Sample `data.txt` file

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

The real data file is much bigger.
```

Notes:

1) Words are defined to be sequences of letters delimited by any non-letter. (e.g., white space, punctuation, parentheses, dashes, double quotes, etc.)
2) There is to be no distinction made between upper and lower case letters. (e.g., "CAT" is the same word as "cat")
3) Blank lines are to be counted in the line numbering. (e.g., line 3 above is blank)

Sample output 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
```

The general algorithm for the word-concordance program is:

1) Read the `stop_words.txt` file into **your implementation of a hashtable** containing only stop words. Make sure to check the number of stop words so that you can size the dictionary appropriately! (WARNING: Make sure you do not include the newline character ('\n') when adding the stop words.)

2) Process the `input` file one line at a time to build the **word-concordance dictionary**. This dictionary should contain the **non-stop words** as the keys in separate hash table (use the stop words hash table to filter out the

stop words). Associated with each key is its **value** where the **value** consists of a list containing the line numbers where the key appears in the file. DO NOT INCLUDE DUPLICATE LINE NUMBERS. **(Note: If a word appears more than once on a line only print out the line number once for that word.)**

3) Generate a text file containing the concordance words printed out **in alphabetical order** along with their line numbers. One word per line (followed by a colon), and spaces separating items on each line. See the sample output files.
Note there is no space after the last line number-make sure to match the sample output files.
(Note: *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.*)

DICTIONARY ADT COMPARISON

Implement 2 dictionary ADT implementations:

- Open Addressing using linear probing
- Open Addressing using quadratic probing

Your basic hash function should take a string containing one or more characters and return an integer. Use **Horner's rule** for string hashing to compute the hash efficiently. (This function explained below.)
In addition, your hash table size should have the capability to grow if the input file is large. You should start with a default hash table size of 251, then 503, then if further increases are necessary, use
new table size = 2* old table size + 1 , and use this "new table size" even if it is no longer a prime. (If load factor > .75 increase the size of hash table.)

- **def honor_hash(self, key)** and return an integer from 0 to the (size of the hash table) – 1. Compute the hash value by h_value(str) = $\sum_{i=0}^{n-1} ord(str[i]) * 31^{n-1-i}$

  where n = the **minimum** of **len(key) and 8** (e.g., if len (key) > 8 assume n=8) , i = the index of each character of the key. If result of your computation of hashval returns float, use round function to return an integer number.

  Example:

  key = 'Cat', which is same as 'cat'

  n= len(key) = 3 and i = 0 for 'c', i = 1 for 'a', and i = 2 for 't',

  h_vslue = round ((ord('c') * 31^{3-1-0} + ord('a') * 31^{3-1-1} + ord('t') * 31^{3-1-2}) % table_size)

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.txt - contains no punctuation to be removed
  - file2.txt, file2.txt - contains punctuation to be removed
  - declaration.txt, declaration_sol.txt – large file for test

Only files for hash_quad is provided. The hash table classes you implement **should each contain** the following methods with exact signatures below. These methods may be called in grading your program. (more detail in you sample hash_quad.py)

- `def __init__(self, table_size)` : creates and initializes the hash table size to size
- `def insert(self, key, value)` : inserts an entry into hash table using Horner hash function

- `def horner_hash(self, key): compute and returns index between 0 and size of table -1`
- `def in_table(self, key):returns True if key is in table`
- `def get_index(self, key):` returns the index of the hash table entry containing the provided key.
- `def get_all_keys(self):` returns a Python list of all keys in the hash table
- `def get_value(self, key):` returns the value (list of line numbers) associated with the key.
- `def get_num_items(self):` returns the number of entries (words) in the table
- `def get_table_size(self):` returns the size of the hash table
- `def get_load_factor(self):` returns the load factor of the hash table (entries / table_size)

SUBMISSION

Six files:

- hash_linear.py containing -- class HashTableLinPr for Linear Probing
- hash_quad .py containing -- class HashTableQuadPr for Quadratic Probing
- hash_linear_tests.py – tests for HashTableLinPr methods
- hash_quad_tests.py – tests for HashTableQuadPr methods
- concordance.py – containing class Concordance with all of the specified methods
- concordance_tests.py – tests for Concordance methods

Helpful resources:

- A "How to on sorting in Python": https://docs.python.org/3/howto/sorting.html . You may use the built-in sorting routines in Python so you may find this reference helpful.

- Python has some build in capability to eliminate punctuation that you may find helpful. See string. constants https://docs.python.org/3.1/library/string.html (apostrophes should be removed, hyphens should be replaced by a space – other punctuation can be removed or replaced by a space)