CptS 223 PA #3 – Dictionary for Hashing

In this task, you will implement a program that creates an English dictionary for users to search through for definitions of words. You will implement a hashtable to store the dictionary and a user interface for managing/using the dictionary. The hash table will be indexed via the words in the English language. The dictionary will store the definition of the words for doing lookups after the dictionary is built.

Here are some of the key components of your implementation:

- Word class
  - holds a word (string)
  - holds the word's definition (string)
  - Already implemented for testing, but it's a tiny thing anyway.
- Hashtable class
  - Implements a hash table (a vector)
  - The hash table will start with size 101
  - The Hashtable class will keep track of how many Words it holds (which is size N)
  - The main table will be a C++ STL vector (see note in private: section of the class)
  - The hash table uses separate chaining to resolve collisions (Chapter 5.3)
    - Each chain is a linked list holding Words
    - You're welcome to decide how you want to do the linked list (STL list is fine)
  - The hash table needs to implement:
    - int hash_function(string key)
      - returns an integer based upon converting the string key to an int
      - See Figure 5.4 from the book for this algorithm
    - int hash_function(int key)
      - returns hashkey integer based upon table size
      - The function from the book using mod TableSize would work just fine
    - void insert(key, value)
      - Hashes the Word's key and adds a Word object to the hash table if it isn't already there
      - Updates (overwrites) old entry if there's a already the same Word in the hash table
    - bool contains(string key)
      - Searches the hash table by a string.
      - Will need to convert the string to a key, then hash it and search the vector of Words to see if it's there
    - Word remove(string key)
      - Finds an entry using the passed in word, erases it (list.erase()) from the hash table, then returns how many elements it removed [0,1]
    - void rehash()

- Is called if the table gets above a load factor (N / Table Size) of 1.0 during an insert of a new Word
- At least doubles the table size then finds the next prime number for the new table size (See figure 5.22 for an example)
- Example code for finding the next prime above a given integer can be found at: https://gist.github.com/alabombarda/f3944cd68dda390d25cb

- Dictionary class:
  - This class is the user interface for your dictionary.
  - I've stubbed in the basics for you, but you'll need to handle parsing the user's input they type in and handle JSON files with dictionaries in them.
  - I provided the canonical loop for C++ to handle user input that ends when the file ends (EOF) or ctrl-D because you should just know that one by heart.
  - This class needs to allow the user to enter these commands via STDIN:

help	-> print out command help
add "word" "definition"	-> Add (or update!) a word and it's definition. Must handle quotes
remove "word"	-> Remove a given word. Must handle quotes
define "word"	-> Define a word by printing out it's definition or "unknown word"
load "filename"	-> Load in a JSON file of dictionary words
unload "filename"	-> Remove words from a given JSON file of dictionary words
size	-> Print out current number of words in the dictionary
clear	-> Remove ALL words from the dictionary
print [#words]	-> Print out all words, unless user gives a maximum number
random	-> Print out a single word chosen randomly from the dictionary
quit	-> Quit the user interface and shut down

- The key parts include:
  - Words can be mixed case, but should still match!
    - Hello == hello == HELLO == hElL0
  - Words might have quotes around them if they have special characters like spaces, so make sure to strip those off when you parse
  - Filenames for JSON files *can* have directory paths. C++ fopen and istream will take those just fine. i.e. configs/fullDict.json
  - print has an optional number of words to limit printing
    - If the user doesn't specify, they get *everything* - test with care
    - print isn't in sorted order, just start from the first word you find
  - unload will read in the JSON file and call remove on every word it finds

A note about JSON files:

JSON is a very common format for web servers. It's used to pass data structures between programs and for storing configurations. The dictionaries I've provided are in JSON format. They live in the config/ directory. You may either use a C++ JSON library to load the files or write your own parser, which doesn't have to handle the full JSON spec, but only our dictionary files. The lines are simple enough that it should be reasonably easy to parse the files without a full on library for handling the full JSON specification, but you're welcome to approach it either way.

Main JSON page: http://www.json.org/

A couple of JSON libraries if you want to use those:

https://github.com/miloyip/rapidjson

http://uscilab.github.io/cereal/

Don't underestimate how much it will take to get a working parser for the dictionary files! Handling the various pieces of texts, quotes, and other string handling could be *interesting* depending on how you approach things. You can do most of it by reading in each line of the file, deciding if it's a line with a word and definition, then using string iterators to find the start and stop of the word and definition to copy out.

Testing

As per normal, the Makefile has test cases. In this case there are separate tests for the hash table and the dictionary. The hash table tests are very similar to the ones we've seen for trees. The tests for the dictionary are more UI-style where they pass in a series of user commands to view the output. Please have a look at how these tests are implemented to make sure you have a sense of what is expected of the program. They pass in the kinds of strings you would expect based on the documentation here, but the ones in the tests are the canonical ones if there's a discrepancy.

Deliverables

You must upload your program by checking it into blackboard on the PA3 branch. Please do not merge it into the master branch so the graders can find it more easily. Also, ensure your code stays in the PA3 directory so we know which code base to check. Grading should follow this pattern:

1) git clone your repository
2) cd into your repository's directory
3) git checkout PA3
4) cd PA3
5) make
6) make test
7) make bigtest
8) inspect your output, errors, and code for style and quality

Grading Criteria

Your assignment will be judged by the following criteria:

- [80] Your code compiles on the EECS servers using g++ -std=c++11 *.cpp (as is defined in the Makefile), passes inspection with a set of test cases (dictionary json files, Makefile, and general poking/prodding via make test and make bigtest.
- [10] Data structure usage. Your program uses a vector for the hashtable and separate chaining with linked lists for the collision resolution.
- [10] Your code is well documented and generally easy to read.