Project 5: Word Frequency Database

1. Introduction

In this assignment, you will implement an application that allows a user to search the frequencies of words based on a corpus dataset constructed by the Google (Books) Ngram Viewer Team (see Section 2 for more details). You application also allows the user to insert new records as well as to replace and remove existing records.

The code skeleton is under **/comp/15/files/p5** on the CS server. However, unlike the previous assignments, the code skeleton provides you with only **main.cpp**, **test.cpp**, and two datasets, **dataset_large.tsv** and **dataset_small.tsv**. That is, this assignment is designed for you to gain experience not only implementing the application but also designing your own modules for the application.

As a part of the requirements, you are asked to implement a hash table with chaining to represent a word frequency database. (See Section 3 for more details.) However, it will be completely up to you that how exactly the application is implemented. Note: You are **NOT** allowed to use C++ Standard Template Library (STL).

1. Log in to the CS homework server.
2. Move to your **comp15** directory that you created in Lab 1.
3. Create a directory named **project5** under the comp15 directory.
4. Move to the project directory.
5. Copy the code skeleton (except for the two dataset files, as they are large (approximately 50 MB in total)) to the current working directory.
6. Leverage the features of Git to manage your progress toward writing the program. (Note: Do **NOT** commit the dataset files.)

Suggestion:

- Consider spending enough time to understand what you are being asked to achieve in this assignment.
- Consider using the concept of a **class** to represent modules, and make the role of each module crystal clear.
- As a part of your design process, think thoroughly how each of your modules communicates with one another to achieve the overall goal.
- Consider performing unit testing and integration testing whenever appropriate.
- Consider refining your design whenever you can.

2. Datasets

The two dataset files are based on the corpus datasets constructed by the Google (Books) Ngram Viewer Team. Please take a look at the web page for more details about the original dataset. http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

The original dataset files can be found at:
American English > Version 20090715 > 1-grams > 0 - 9
(Licensed under the Creative Commons Attribution 3.0 Unported License)

dataset_large.tsv includes **3,738,012** records from the original dataset files, 0-8
dataset_small.tsv includes **414,941** records from the original dataset file, 9

In terms of the two dataset files, **dataset_large.tsv** and **dataset_small.tsv**:

- "tsv" stands for tab-separated values.

- Each line has the form: **word**		**frequency**
    - The line begins with **word**,
    - followed by a tab character,
    - followed by **frequency**,
    - followed by a newline control.

- **word** is composed of only ascii characters.
- **frequency** is the number of books that contain **word**.

3. Requirements

Part 1: Makefile

You will provide your Makefile, so that the grader can compile your source files using the make command. Your Makefile must include at least one target task, named **build**. The task is to produce an executable, named **database**, which is the name of your application. The usage is:

```
$ make build #This produces the executable named "database"
```

Part 2: The database executable

The database executable takes one argument that is the path to the dataset file. An example usage is:

```
$ ./database /comp/15/files/p5/dataset_large.tsv
```

With a successful run, the application first checks the existence of the dataset file, opens the file, and then stores all records in your hash table.

To read a file and to split a line into tokens with a specific delimiter character, you may find the following useful:

- **`std::ifstream`** (You will need to include **`fstream`** header.)
    - `open()`, `close()`, `is_open()`, `good()`, `eof()`
- **`std::istringstream`** (You will need to include **`sstream`** header.)
    - `str()`, `clear()`, `good()`, `eof()`
- `std::getline()` (You will need to include **`string`** header.)
    - tab character: `\t`

If no path or more than one path is given or if any unexpected errors happened with the given file, then the application prints out the following message to **stdout**, and then terminates. The message is followed by the newline control. (See an example in the given **main.cpp**.) An example, in which no path is given, is:

```
$ ./database
Error
```

After successfully storing the word frequency records in the hash table, the application waits for user input (**stdin**). The application should be able to respond to the following four commands, `:q`, `:p`, `:g`, and `:r`. The application keeps running until the user inputs the `:q` command. Note that for grading purposes, the application should not print out anything to **stdout** other then the outcome of each user command.

`:q`

(q)uit. Terminate the application.

`:p word frequency`

(p)ut. Add the word and its frequency record to the database.
Each of `:p` and `word` is followed by a space. Therefore, `word` itself cannot contain any spaces.
If `frequency` is not a 0 (zero) or not a positive integer, then the application prints out the following message to **stdout** (case-sensitive): **Invalid**
With a successful execution, the application prints out the following message to **stdout** (case-sensitive): **Added**
Both error and successful message is followed by a newline control.
If the given `word` exists in the database, then the application replaces the existing record with the new record.

`:g word`

(g)et. Retrieve the record that is corresponding to the given `word`.
If the record exists, then the application prints out the message that has the following form to **stdout** (case-sensitive): **word frequency**

- where **word** is the `word` that the user typed in,
- followed by a space,
- followed by **frequency** which is the frequency of the given word.

If the record does not exist, then the application prints out the following message to **stdout** (case-sensitive): **Not found**
Both message is followed by a newline control.

`:r word`

(r)emove. Delete the record that is corresponding to the given `word`.
With a successful execution, the application prints out the following message to **stdout** (case-sensitive): **Deleted**
If the record does not exist, then the application prints out the following message to **stdout** (case-sensitive): **Not found**
Both successful and error message is followed by a newline control.

In case any other command is given, then the application prints out the following message to **stdout** (case-sensitive): **Unknown command**
The message is followed by a newline control.

Part 3. The specification

You will provide the specification document of your modules. (The document must be in PDF format.) The document must describe all of the public methods of each module, including the default constructor, non-default constructors (if any), copy constructor, assignment operator and destructor. There is no specific format for this document; however, it is highly recommended to leverage powers of colors, fonts, layouts, figures, and so on. Lastly, your specification document must be precise, and the content of your specification document must match with your implementations.

Another requirements

- Your hash table should keep track of its load factor and should be able to expand its size according to its load factor. (In other words, your hash table may not be of a fixed size.)
- You are responsible for your modules not producing any memory leaks and memory errors.

4. Testing

Modify the **test.cpp** by adding your own test cases to ensure that your module function correctly. You are encouraged to add as many test cases as necessary to make you feel confident about your modules.

Run the application, whose main() is written in **main.cpp**, and manually type in some commands to test the overall functionalities. You can also prepare an input file and an expected output file, and then use redirects to automate user inputs as well as to save outputs of your application.

5. README

Create the README file that includes the following categories with appropriate section headers.

1. **Name**: The programmer's name.
2. **Date**: The last updated date of the program code, including README.
3. **Summary**: A brief summary of the project. (e.g. What does the program do? How do you use the modules? and so on.)
4. **Files**: A list of files that are necessary to build and test the program.
5. **Instructions**: A sequence of commands to compile and test/run the program. Note that you are expected to report procedures without using the make command.
6. **References**: A list of citations to information used to complete the project.
7. **Post evaluation on planning**: Did your project go smoothly, as planned? If not, which parts were difficult? What were the causes? How will you improve your planning for future assignments?

6. Submission

Submit your files listed below using Gradescope.
Files: **main.cpp test.cpp Makefile README specification.pdf**
and all .hpp and .cpp files that are necessary to build your application