Homework 3: WordBench

Educational Objectives: On successful completion of this assignment, the student should be able to

- Define the concept of associative container as a re-usable component in programs
- State the distinction between unimodal and multimodal associative containers, and:
  ☐☐☐ Give examples of each type
  ☐☐☐☐ Describe use cases making each type appropriate
- State the distinction between ordered and unordered associative containers
  ☐☐☐ Give examples of each type
  ☐☐☐☐ Describe use cases making each type appropriate
- State the API for associative containers of these types:
  ☐☐☐ Unimodal Ordered Set
  ☐☐☐☐ Multimodal Ordered Set (aka Ordered Multiset)
  ☐☐☐☐☐ Unimodal Unordered Set
  ☐☐☐☐ Multimodal Unordered Set (aka Unordered Multiset)
  ☐☐☐ Unimodal Ordered Map (aka Ordered Table)
  ☐☐☐☐ Multimodal Ordered Map (aka Ordered Multimap)
  ☐☐☐☐☐ Unimodal Unordered Map (aka Unordered Table)
  ☐☐☐☐☐☐ Multimodal Unordered Map (aka Unordered Multimap)
  ☐☐☐☐ Ordered Associative Array
  ☐☐☐ Unordered Associative Array
  Describe the behavior and state the runtime expectations for each operation.
- Describe various implementation plans for ordered associative containers, and discuss whether and why runtime expectations are met by the implementation.

Background Knowledge Required: Be sure that you have mastered the material in these chapters before beginning the assignment:
Introduction to Sets, Introduction to Maps.

Operational Objectives: Create a client `WordBench` of the Set API that serves as a text analysis application.

Deliverables: `wordbench.h`, `wordbench.cpp`, `makefile`, `log.txt`.

Procedural Requirements

☐☐☐ Begin by copying all of the files from the assignment distribution directory, which will include:

```
hw3/main.cpp        # driver program for wordbench
hw3/data*           # sample word files
hw3/makefile        # makefile for project
hw3/hw3submit.sh    # submit script
```

☐☐☐ Define and implement the class `WordBench`, placing the class API in the header file `wordbench.h` and implementations in the code file `wordbench.cpp`

☐☐☐ Be sure to fully cite all references used for code and ideas, including URLs for web-based resources. These citations should be in the file documentation and if appropriate detailed in relevant code locations.

☐☐☐ Test your API using the distributed client program `main.cpp`.

☐☐☐ Keep a text file log of your development and testing activities in `log.txt`.

☐☐☐ Submit the assignment using the script `hw3submit.sh`.

> ***Warning:*** *Submit scripts do not work on the program and linprog servers. Use `shell.cs.fsu.edu` to submit assignments. If you do not receive the second confirmation with the contents of your assignment, there has been a malfunction.*

Functionality Requirements

☐☐☐ `WordBench` can read an arbitrary text file on command and extract all of the words in the file, maintaining the unique words, along with the frequency of occurrence of each word, in a set. Letters are converted to

lower case before comparison and storage. A word is understood to be a string of letters and/or digits, with certain other symbols allowed. Most non-alpha-numeric characters are ignored. Exceptions are hyphens and apostrophes, which are considered part of the word, so that contractions and hyphenated constructs are counted as individual words. (Note: two adjacent apostrophes are not considered part of a word, since they represent closing of a quotation.)

- ☐☐☐ WordBench can write an analysis of its current stored words. This analysis consists of a lexicographical listing of the unique words together with their frequencies, followed by a count of the total number of words and the vocabulary size (number of unique words). Note that this is a cumulative analysis over all of the input files read since starting up TA (or since the last clearing operation).
- ☐☐☐ Note that a component of the analysis and summary is a listing of the files whose contents contributed to the data.
- ☐☐☐ WordBench must operate with the supplied driver program LIB/hw3/main.cpp which has a user interface with the following options:
  - ☐☐☐ Read a file. Read the words of the file into the structure (and report summary to screen).
  - ☐☐☐ Write an analysis of the current data (including input file names) to a file (and report summary to screen).
  - ☐☐☐ Erase current data and clear all data from the structure.
  - ☐☐☐ Show current size and send a data summary to the screen.
  - ☐☐☐ display Menu.
  - ☐☐☐ eXit BATCH mode.
  - ☐☐☐ Quit program.

  Use the source code in the driver program main.cpp to determine the syntax requirements for the WordBench public interface. Use the executable in area51 to model expected behavior. The following shows the exact syntax of the API required by the driver program:

```
bool   ReadText     (const fsu::String& infile);
bool   WriteReport  (const fsu::String& outfile, unsigned short c1 = 15, unsigned
short c2 = 15) const;
void   ShowSummary  () const;
void   Erase        ();
```

- ☐☐☐ From any directory having access to the course library and containing your submission files, entering "make" should result in an executable called "wordbench.x". (NOTE: This requirement will necessitate only a name change for the executable in the distributed makefile.)

Implementation Requirements.

- ☐☐☐ You should define a class WordBench, declared in the file wordbench.h and implemented in the file wordbench.cpp. An object of type WordBench is used by the driver program to create the executable wordbench.x.
- ☐☐☐ Use the following to define internal types and private class variables for WordBench:

```
private:
  // the internal class terminology:
  typedef fsu::Pair      < fsu::String, unsigned long >  EntryType;
  typedef fsu::LessThan  < EntryType >                   PredicateType;

  // choose one associative container class for SetType:
  typedef fsu::UOList          < EntryType , PredicateType >      SetType;
  // typedef fsu::MOList          < EntryType , PredicateType >      SetType;
  // typedef fsu::UOVector        < EntryType , PredicateType >      SetType;
  // typedef fsu::MOVector        < EntryType , PredicateType >      SetType;
  // typedef fsu::RBLLT           < EntryType , PredicateType >      SetType;

  // declare the two class variables:
  SetType                   wordset_;
  fsu::List < fsu::String >  infiles_;
};
```

This will serve several useful purposes:

- ☐☐☐ Changing the structure used for SetType is as simple as changing which typedef statement is uncommented in the WordBench class definition.
- ☐☐☐☐ It is ensured that you are writing to the Set API
- ☐☐☐☐☐ The list of filenames is an fsu::List of fsu::String objects
- ☐☐☐☐ The "RBLLT" option will be used later when we develope left-leaning red-black trees.

You are free to add private helper methods to the class. You should not add any class variables other than

wordset_ and infiles_.

☐☐☐ Note that the fsu::Pair template class has comparison operators defined that emphasize the first coordinate of the (called the "first_", but playing the role of "key"), so that two pairs are considered equal, for example, if they have equal keys.

☐☐☐ The application should function correctly in every respect using fsu::UOList < EntryType > for SetType.

☐☐☐ The application should function correctly in every respect using fsu::UOVector < EntryType > for SetType.

☐☐☐ As usual, you should employ good software design practice. Your application should be completely robust and all classes you define should be thoroughly tested for correct function, robust behavior, and against memory leaks. Your wordbench.x should mimic, or improve upon, the behavior illustrated in area51/wordbench.x.