Project 4: WordBench

Analyzing vocabulary in texts

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:
  i. Give examples of each type
  ii. Describe use cases making each type appropriate
- State the distinction between ordered and unordered associative containers
  i. Give examples of each type
  ii. Describe use cases making each type appropriate
- State the API for associative containers of these types:
  i. Unimodal Ordered Set
  ii. Multimodal Ordered Set (aka Ordered Multiset)
  iii. Unimodal Unordered Set
  iv. Multimodal Unordered Set (aka Unordered Multiset)

  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`, `wordify.cpp`, `log.txt`.

Procedural Requirements

1. The official development/testing/assessment environment is specified in the Course Organizer.
2. Begin by copying all of the files from the assignment distribution directory, which will include:

```
LIB/proj4/main_wb.cpp       # driver program
LIB/proj4/data*             # sample word files
LIB/proj4/makefile          # makefile for project
LIB/scripts/submit.sh       # submit script
LIB/proj4/deliverables.sh   # submission configuration file
```

3. Define and implement the class `WordBench`, placing the class API in the header file `wordbench.h` and implementations in the code file `wordbench.cpp`
4. 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.
5. Test your API using the distributed client program `main.cpp`.
6. Keep a text file log of your development and testing activities in `log.txt`.
7. Submit the assignment using command `submit.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

1. `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.)
2. `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 wordbench.x (or since the last clearing operation).
3. Note that a component of the analysis and summary is a listing of the files whose contents contributed to the data.
4. `WordBench` must operate with the supplied driver program `LIB/proj4/main.cpp` which has a user interface with the following options:
   a. Read a file. Read the words of the file into the structure (and report summary to screen).
   b. Write an analysis of the current data (including input file names) to a file (and report summary to screen).
   c. Clear current data and clear all data from the structure.
   d. Show current size and send a data summary to the screen.
   e. display Menu.
   f. eXit BATCH mode.
   g. 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 WordBench 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; // c1,c2 are column widths
void   ShowSummary  () const;
void   ClearData    ();
```

5. 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.

1. 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`.
2. 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:

i. Changing the structure used for `SetType` is as simple as changing which `typedef` statement is uncommented in the `WordBench` class definition.
ii. It is ensured that you are writing to the Set API
iii. The optimal choice (other than RBLLT) is UOVector - unimodal set API with very fast search time. However it is important that the project is functional with all of the choices currently available, even if functionality isn't what you want for the multimodal options. This tests the generality and genericity of your code. Then, later, when you have created RBLLT, you can switch over and have fast insert times along with fast search times.
iv. The list of filenames is an fsu::List of fsu::String objects

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

3. Add a private helper method as follows:

```
private: // string cleaner-upper
  static void Wordify  (fsu::String&);
```

Wordify is used to "clean up" the string passed by reference according to the processing rules above. The implementation of Wordify should be in the separate file `wordify.cpp`. (This function may be used again in a future assignment.)

4. Note that the `fsu::Pair` template class has comparison operators defined that emphasize the first coordinate of the pair (called "first_", but playing the role of "key"), so that two pairs are considered equal, for example, if they have equal keys.
5. The application should function correctly in every respect using `fsu::UOList < EntryType >` for `SetType`.
6. The application should function correctly in every respect using `fsu::UOVector < EntryType >` for `SetType`.
7. 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`.

Hints

- It is critical to keep track of the various APIs you are dealing with. Here is a partial list:

  fsu::List
  the Set API
  fsu::Pair
  fsu::String
  WordBench

  In addition you have:

  The user interface defined in `main.cpp`, which amounts to a driver program for WordBench plus some control commands

  You are tasked to *implement* the WordBench API. In this implementation you will need to *write to* the various fsu APIs.

- Test your WordBench by uncomment/comment the various possible SetType definitions (except for the last one using RBLLT, which we will get to in a later assignment). How does it behave with the other unimodal implementation UOVector? How does it behave with the multimodal (MultiSet) versions, MOList and MOVector ?