Homework 3: Lexicon

CSCI 60

Due 11:59pm Sunday, January 24, 2021

Krehbiel

Overview. This program will have you implement a class called `Lexicon`, which is similar to the `Bag` data structure introduced Monday of Week 3 but it functions as a set of distinct words rather than a list of possibly repeated integers. The complete lexicon interface is given in `lexicon.h`; your implementation goes in `lexicon.cpp`. Part of your task is to modify the construction and mutation functionality of `Bag` to reflect the set-like nature of `Lexicon`, and part of your task is to overload several additional operators corresponding to set operations.

This handout overviews the interface and details how to implement each of the required functions. You will only upload your implementation in `lexicon.cpp`, but you should use the provided skeletal `main.cpp` driver program to do your own unit testing as you implement each `Lexicon` function. Do not change any of the boilerplate code at the top of each file or use functions defined in libraries not included by the starter files, and compile your code as follows:

```
$ g++ main.cpp lexicon.cpp -std=c++11
```

Look over the submission checklist from HW1 for usual submission instructions and style guidelines, and read announcements about common technical problems other students encounter throughout the week.

The interface. The interface file `lexicon.h` is fully written and should not be modified in any way. It is heavily documented with pre- and post-conditions for each function. Some simple functions are defined inline; these don't belong in your implementation file.

Notice that the fact that a `Lexicon` stores strings is reflected in the types in the function signatures, and the fact that it stores distinct values is reflected in the specification of `insert` and `remove`. In addition to the subscript operator defined inline, you will overload three binary operators as member functions; these correspond to set union, intersection, and symmetric difference. You will also overload six binary comparison operators as nonmember functions; these will correspond to whether one set is a superset of another.

The implementation. Your main task is to implement all the functions declared in the interface. You can tackle these tasks in the following stages:

1. Note that a 0-argument constructor creating an empty set is already implemented for you inline. Implement a 1-argument constructor to open a text file and populate the `data_` array with up to `CAPACITY` distinct words in the order that they appear in the file, keeping track of the number of distinct words in `size_`. Because the file may have duplicate words, it is possible for it to have more tokens than the capacity of a lexicon but for it to not fill a lexicon. For example, if you have a file `hello.txt` with `CAPACITY` instances of `"hello"` followed by one instance of `"world"`, then calling `Lexicon lex("hello.txt")` from main should result in an object `lex` where `lex.data_` is an array whose first two entries are `"hello"` and `"world"` and `lex.size_` is 2. However, if there are more than `CAPACITY` *distinct* values in your file, your constructor should fill the data array with the first `CAPACITY` distinct values and ignore the rest of the file.

   This program does not require you to do any string manipulation. Use == to test string inequality, which treats tokens `"Hello"`, `"hello"` and `"hello!"` as three distinct words.

2. Implement three boolean member functions `contains`, `insert`, and `remove` consistent with the pre- and post-conditions stipulated in the header file.

3. Overload three bitwise operators as member functions so that `lex1 | lex2` corresponds to the union of the two lexicons (i.e., any string in either or both of the underlying lexicons), `lex1 & lex2` corresponds to the intersection of the two lexicons (i.e., any string in both of the underlying lexicons),

and `lex1 ^ lex2` corresponds to the symmetric difference of the two lexicons (i.e., any string that appears in exactly one – not both – of the two underlying lexicons).

Note that because each of these operators is implemented as a member function, the function is written with `lex1` as the reference object and `lex2` as a single parameter, and the function constructs, populates, and returns a third lexicon object.

4. Overload six comparison operators as nonmember functions with the following functionality:

    - `lex1 == lex2` is true if and only if every string in one of the lexicons is also in the other (though not necessarily in the same position),
    - `lex1 != lex2` is true if and only if there is some string in one lexicon and not the other,
    - `lex1 <= lex2` is true if and only if the first lexicon is a subset of the second (i.e., each string in the first is also in the second),
    - `lex1 < lex2` is true if and only if the first lexicon is a strict subset of the second (i.e., each string in the first is also in the second, and the second contains at least one string not in the first),
    - `lex1 >= lex2` is true if and only if the second lexicon is a subset of the first, and
    - `lex1 > lex2` is true if and only if the second lexicon is a strict subset of the first.

    Note that because each of these operators is implemented as a nonmember function, the contents of the parameters can only be accessed through public functions such as `[]`, which is implemented for you, and `contains`, which you will have implemented in Step 2.

The test program. Use the `main.cpp` file to test your implementation as you go. Writing *unit* tests to test each function one at a time as you develop rather than running all tests at the end. This will make it much easier to identify the source of both syntactical and logical bugs, saving you lots of time and headache! Notice that << is implemented for you and should be used liberally as you test.

When your implementation is complete, run `foundingDocs` and `annaKarenina` to check that they compile and produce the following output when run with the sample text files provided with the assignment:

```
The US Constitution contains 1680 distinct words.
The Declaration of Independence contains 632 distinct words.
The two documents have 202 words in common.
There are 1908 words in their symmetric difference.
Does it make sense that there are 2000 words in the combined lexicon?

12.0238% of Constitution words are in the Declaration of Independence.
"Order" is one that is not.
31.962% of Declaration of Independence words are in the Constitution.
"Honor." is one that is not.

The Constitution's preamble contains 2.32143% of the distinct words
in the Constitution and no others.
The strict subset relationship remains true if we flip
the operator and args even though it's a different function call.

Scrambling the words of the preamble gives an equivalent lexicon,
which is equivalent to the union of the two.


Anna Karenina is 350188 words (with repetition).
The number of distinct words reaches our class's capacity.
```