Project 2, Indexing using BSTs and overloading

Due: Please check due date on BlackBoard

Objectives

The objective of this programming assignment is to have you practice using Binary Search Trees (called BSTs from now on), Recursion, Queues, Strings, Overloaded operators, and File I/O.

Introduction

The creation of a textbook’s index requires the ability to determine the frequency and location of words in a document. The information is stored in a structure where it lists the distinct words in alphabetical order and makes references to each line on which the word is used. For instance, consider the text below:

The Hinge tool lets you select a hinged element on your model and move it in the directions indicated by the arrows. A selected element can be moved using your mouse or the arrow keys on your keyboard. When an element can be rotated in multiple directions, the selected direction is indicated with a green arrow. To select a different direction, click one of the yellow arrows or use the TAB key on your keyboard.

The word “element” occurs 3 times in the text and appears on lines 1, 2 and 3. The word “keyboard” occurs 2 times and appears on lines 3 and 5. **Case is not considered.** Keyboard and keyboard are the same word.

There are many words in the textbook that should not be indexed. These are called “Stop Words”. While there are many different wordlists that can help in filtering common words and phrases, the stop words used in this project will come from the “General Service List” and is provided (and edited slightly), and named “gsl.txt”. It will not be perfect, but it will do for this project.

Application Setup

Your application in command line format, will accept:

1. Filename of words NOT to be included in the index (the stop words). This will usually be gsl.txt file.
   1. This file needs to be validated that it is present.
   2. The filter list will not be in alphabetical order but will be one lower case word per line.
   3. There will be no punctuation in the file.
2. Filename of data (text) to be indexed.
   1. This will need to be validated that it is present as well.
   2. certain punctuation will need to be removed from the data file

Example command line:

**./Proj2.out gsl.txt data.txt**

Your application, ***in order***, will:

1. Validate the input files.
2. Read in the stop words file (gst.txt)
3. Build a BST of Word(s) to be filtered called “filteredBST”. Remember this will the list of ***stop words***.
4. Read in a file containing the data using the second command line variable, while:
   1. Building a filtered **case insensitive** BST of Word(s) “indexedBST”
   2. Storing the line number where the word appears.
   3. The stored object needs to know how often the word has appeared in the data file.
5. output the results **(using an overloaded << stream)** of:
   1. filteredBST in alphabetical order to “filterResults.txt”
   2. indexedBST in alphabetical order to “indexResults.txt”

BST Class:

The BinarySeachTree(.h) code for this class is given. You will need to add some features to be successful. Remember, you may edit the file, but **the BST must stay generic** and we are NOT using the C++11 version since GL cannot support it. You are ***required*** to use the code given, but please do not remove any features since the graders may use those features to test your program.

Indexer Class:

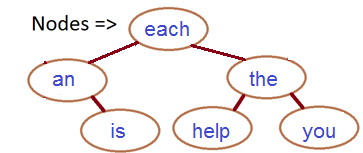
The Indexer class is where most of the actual work is done in the project. The following design goals should be met:

1. You should provide a constructor:
   1. That allows passing in the file names you will be working with and doing the validation needed.
   2. Does all other initialization for the object.
2. A method DoIndex() that will build the BSTs and show the proper output. This method should not directly do the work but should be using other methods you define.
   1. Build the filteredBST.
   2. Build the indexedBST.
   3. Output the results.

The rest of the design of this class is up to you.

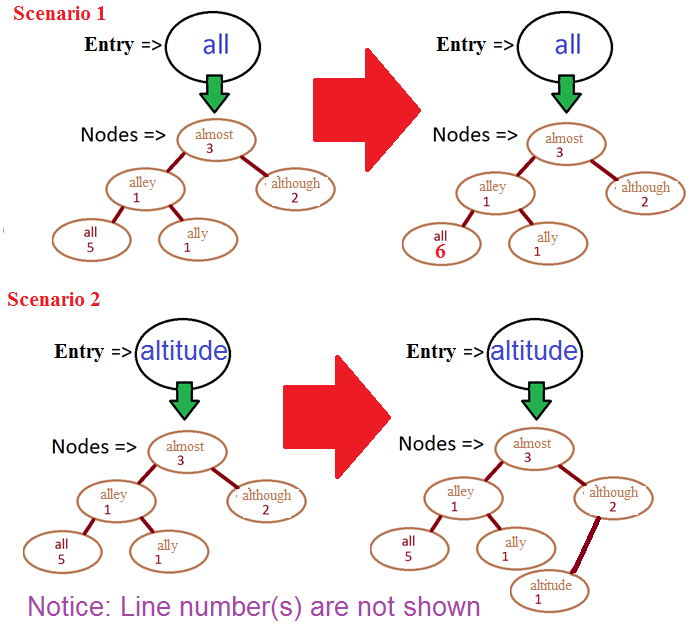
Filter BST (filteredBST)

This will be a very simple BST of WORD that contains words that will be filtered. The words that will be filtered will be in a file given named gsl.txt. DO NOT assume that the stop word file will be the same each time.



Index BST (indexedBST)

This will be much more complicated. An instance of the IndexBST will be created in the Indexer class. The Index BST will contain nodes of WORD(s) but named nodes. Each node is covered in the text below. The Index BST will grow as new words are entered, but it needs to see if the word has already exists. If the word already has an entry, (this will use the == overload) then count and queue listing need to be updated. Otherwise, a new entry is added to the indexedBST. There will be other functions to create for testing and debugging purposes.



Word Class

The Word class will be the most comprehensive classes that you will complete on your own in this project. The Word class will contain the member variables “wordText”, “count”, and an STL Queue of Line numbers named “lineNumbers”. It will also contain a:

* 1. constructor
  2. CountWord
     1. increments the word’s occurrence
     2. add this unique value to the queue, remember to add only once if in the same line
  3. member getters/setters
  4. The = operator for Word IS OPTIONAL. (The other overloaded operators are not). I placed it on the UML just to show you what I did, but notice it was no mentioned in the project description or requirements.
  5. comparison operator == AND < for inserting into the BST
  6. stream operator <<
     1. <text>……………………… <count>: q1, q2, q3, q4 … qn
     2. the <count> portion is right justified, with “.” as the fill value
     3. see sample output below

Other Class(es)

You are allowed to create other classes for your needs. We suggest a Utility class for some basic string manipulation and an exception class of your own for handling any exceptions. Just make sure to follow the requirements below.

Sample Output

The output is done by Word’s overloaded << operator. The output will be the enter data file entered and filter. The output should look like in a real index (except count), whereas the <count> portion is right justified, with “.” as the fill value

<text>……………………… <count>: q1, q2, q3, q4 … qn

(BTW, it is 24 characters from the start of the line to the colon “:”)

An example is shown below:

apple..................3: 1 2

axel...................1: 12

battle.................4: 1 12 45 89

-

-

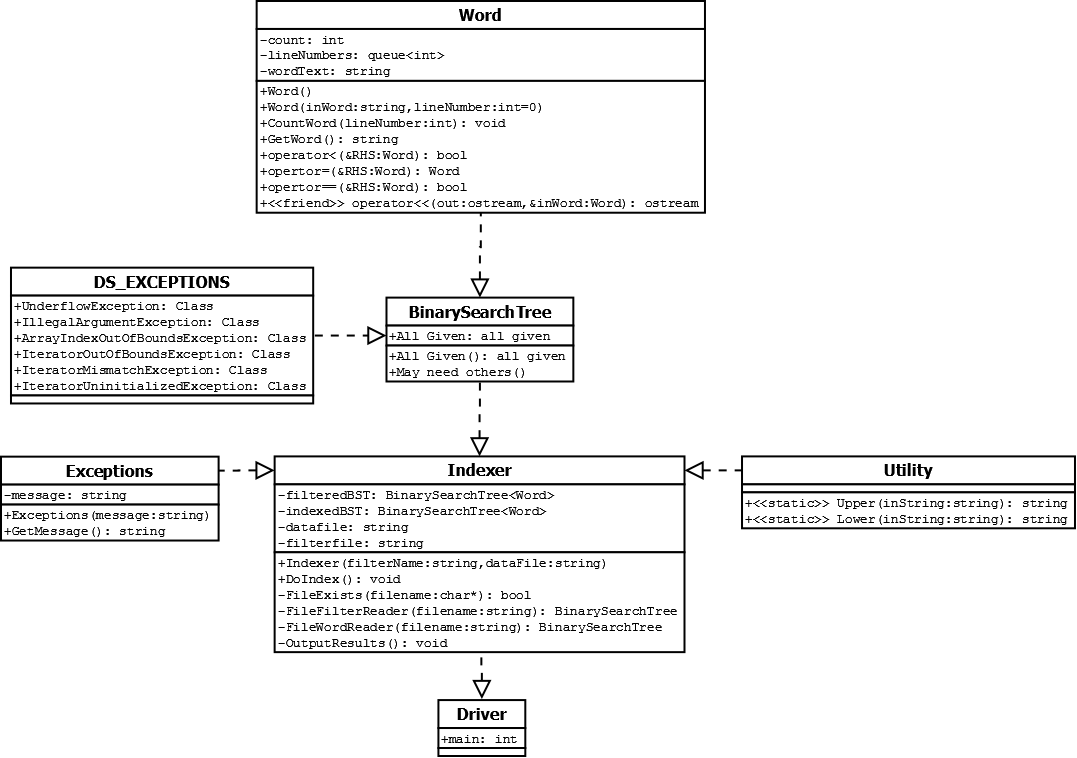
they’re...............11: 34 26 245 234 223 543 678 888

The result is accomplished by printing the text and writing the count right justified in a field with the fill character set to “.”. The lines numbers are printed by traversing the linked list.

Coding and Implementation Requirements

*Note: Running time is one of the most important considerations in the implementation of a data structure. Programs that produce the desired output but exceed the required running times are considered wrong implementations and will receive substantial deductions during grading.*

1. You will be required to use C++’s Queues to hold the generic Lines Numbers. It is to be named *lineNumbers*.
2. You will be required to develop a generic BST. You must use the BST class from the textbook.
3. A primary objective of this programming assignment is to have you use recursion. So, one component of grading will evaluate how elegantly you employ recursion to implement this data structure. (Yes, you are being graded on aesthetics!) There will be some additions that will ***not*** use recursion.
4. The stop words file will ***MAY BE*** named “gsl.txt”. An example is given. All entries in the file will ultimately be stored in a BST named “filterBST”.
5. The run time for the entire project should not be anything more than O(n), BUT:
   1. all BST CREATION operations will run in O(n) time
   2. all BST operations will run in O(n log n) time
   3. printing in the entire BST operations will run O(n)
6. The Indexer class **must** contain:
   1. the function FileFilterReader(String filename)
      1. this function will open the file, places those words into the appropriate BST in the O(n) run time
   2. the function FileWordReader(String filename)
      1. this function will open the file, filter the useless text it to the distinct words, and frequency, then places those words into the appropriate BST in O(n) time
7. The class Word:
   1. Word.cpp/.h is a very basic class that holds an instance of a word and it’s frequency.
   2. The Word’s frequency and word is given IMMEDIATETLY in the file.
   3. This class will also contain the overloads discussed above.
      1. = overload is optional
   4. Please see the before mentioned description and UML below.
8. You can check that your code compiles correctly with this sample **main** (has the main() ) program: Driver.cpp. Your code must compile with example Driver.cpp without alteration. The executable should be named Driver.out.
9. To help further your organization, you must use this UML as part of your layout.



What to Submit

Submit to Blackboard as usual. You should copy over all of your C++ source code and have your .cpp/.h under the src directory. You must also supply an MAKE build file.

Make sure that your code is in the ~/csce221proj/proj2/ directory and not in a subdirectory of ~/csce221proj/proj2/. In particular, the following Unix commands should work.

cd ~/csce221proj/proj2/src

make compile

make run **FILTER=gsl.txt DATA=data.txt**

make clean

Addendum

If your project is taking a LONG time to complete, it could be who you are breaking up the tokens within the line. Remove all useless punctuation from the ENTIRE line read in first, then start breaking up the line into smaller tokens.

**In the example file with text from the Quran, there are some words that begin with an apostrophe ('Ad, 'Iddat, maybe others) - do we need to keep that starting apostrophe when adding to the index?**  
  
Yes, do not break it up. It should not have any problems in ordering a character over a letter.  
  
**There are also words with 'internal' apostrophes (Quran actually appears as Qur'an, for example) - I've kept that format but is that what we're supposed to do?**  
  
Same as above, to keep it simple.  
  
**I was testing with the other example input file (containing Bible extracts and Shakespeare) and ran across an instance of several words ran all together.  The resulting 'word' ("sixorseventimeshonoured") is 23 characters long, so it threw an exception during output.  My question for this is: should we be checking the length of each word and ignoring if it's too long?  Or should we try to handle a situation like this some other way (like breaking this up into component words)?**

It should handle this as one massive word. Breaking up the word would be another project in itself. Again, trying to keep it simple.