AutoIndexer

Due: Monday Sept 27, 2021 @ 6am to Github Classroom Repo

Introduction

Professor Jackson was just assigned to be the editor of a riveting textbook titled "Advanced Data Structure Implementation and Analysis". She is super excited about the possibility of delving into the material and checking it for technical correctness. However, one of the more mundane tasks she must perform is creating an index for the book. Everyone has used the index at the back of a book before. An index organizes important words or phrases in alphabetical order together with a list of pages on which they can be found. As a comp sci prof, Jackson decides she wants to automate the process as much as possible because she knows that an automated indexer is faster and more accurate, and because it can be reused later when she finishes writing her own book. So as she is editing the book, she indicates important words and phrases with square brackets ('[' and ']') in the text. She's enlisted your help to write an AutoIndexer that will extract these bracketed terms and create an ordered index for them.

Input and Output

There will be 1 input file and one output file for this project.

Book Text Input File

This file will contain the text that Professor Jackson is editing. The book is currently in raw text (ASCII). The book is separated into pages - each page number is in angle brackets on its own line. The end of the text is indicated by <-1> as the page number. Important words and phrases are enclosed in square brackets([]). As you'll see in Listing 1, the indexable terms could be nested.

```
<3>
Preface:

[Advanced [Data Structures] and [Algorithm Implementation]] is a new
exciting text from Mustang Publishers aimed at upper division
undergraduate courses or introductory graduate courses.
<4>
Some of the topics covered are:
* [Fast String Matching]
* [[B+ Tree] Implementation]
* [A* Graph Traversal]
<10>
In a [B+ Tree], the leaf nodes are connected as a [doubly linked
list].
<-1>
```

Listing 1: Example book text.

Notice that the bracketed terms can be nested. For example `[[B+ Tree] Implementation]`. This means that the appropriate page (4 in this case) should appear in the list for `B+ Tree` AND for `B+ Tree Implementation.` Nesting will never be more than 1 level deep.

```
[2]
2-3 tree: 1
[A]
algorithm: 1, 15
analysis: 1, 15
[B]
b+ tree: 5
binary search tree: 1
binary tree: 5, 15
[C]
clique: 8
complexity: 1
complete binary tree: 5
[F]
full binary tree: 5
...
```

Listing 2: Sample Output Text File. **IMPORTANT NOTE**: This index doesn't match the text and keywords from Listing 1 and Listing 2. It is for example only.

The Output Text File

The output text file will be organized in ascending order by keyword/phrase with numeric index categories appearing before alphabetic categories. Each category header(A, B, C, etc.) will appear in square brackets followed by index entries that start with that letter in ascending alphabetic or numeric order. All the index entries should be converted to lower case. So, `b+ tree` and `B+ Tree` in the text should appear under the same index entry of `b+ tree`.

An index entry will consist of the indexed word, a colon, then a list of page numbers in ascending order that contain said word. No output line should be longer than 70 characters. The line should wrap before 70 characters and subsequent lines for that particular index entry should be indented 4 spaces. An example output text file can be found in Listing 3.

Implementation Requirements

The Vector Class

You don't have any idea how many individual words, index entries, etc. will be present in the input data file. And since Dr. Jackson doesn't like the container classes from the c++ standard library, you can't use the vector class that automatically grows as you insert elements into it. You'll need to implement some "data structure" that is capable of "growing" as needed. This sounds like a good place to use a templated vector class, **DSVector<>** specifically.

You'll need to implement a vector class that should minimally include the following features/functionality:

- a vector shall be able to hold any data type (template your class)
- a vector shall be a contiguously allocated, homogeneously typed sequential container
- a vector shall grow as needed
    - The default size of the vector should be 10 elements.
    - You may provide an additional constructor that allows creation of a vector with a larger

initial size.
- a vector shall minimally contain the following functionality:
    - add a new item to the container - you can choose how many options you would like to give to the user of DSVector. minimally, you should have a push_back-type function.
    - access elements using the [] operator
    - remove an element from the container given a location.
    - follow rule of three
    - search the container for an element and return a location.
    - functionality to allow the user to iterate over elements in the vector. You could even go so far as to implement something that mimics `std::vector<>::iterator`.

There's a great deal of other functionality that SHOULD be included, but this is the minimum amount needed. You should make sure your vector class is adequately tested using the CATCH library.

Other Requirements for your implementation:
- Your submission should be fully Object Oriented.
- You must use your DSString class from PA01; no c-strings (except inside DSString, and no std::strings). The one exception is using a char array buffer to temporarily store info when reading from the file.
- Your code should have a minimal amount of memory leaks per Valgrind or a similar profiling tool.

Assumptions
You may make the following simplifying assumptions in your project:
- The input file will be properly formatted according to the rules above
- Dr. Jackson hasn't included any unnecessary punctuation in the square brackets. (IOW, you don't need to remove punctuation).
- No line of text in the input file will contain more than 80 characters
- No keyword or phrase will be longer than 80 characters
- Different forms of the same word should be considered as individual entries in the index (e.g. run, runs, and running would each be considered separate entries words).

Execution
There will be two methods of running your project:
1. test mode - No command line args - your catch tests will run
2. run mode - 2 command line args
    a. book text file name
    b. index output file name

NOTE: This may be updated in the next 24 hours.

What to Submit
You should submit:
- well formatted and documented source code
- Your sample testing files

Grading

	Points Possible
DSVector class and CATCH2 Tests	20
Proper Templating Implementation for DSVector	10
Well-designed OO structure	20
Proper Memory Management including minimal mem leaks	10
Book Indexing functionality	30
Source Code Quality including Comments	10