AutoIndexer

Due Date: **Monday Feb 28, 2022 @ 6am to Github Assignment Repo**
Early Submission Deadline: **Saturday Feb 26, 2022 @ 6am to Github Assignment 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. But, who or what creates these indexes? Do humans create them? Do computers create them? 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 keeps a list of words on each page that should be included in the index. However, time is short, and she needs to get the book edited AND indexed quickly. She's enlisted your help to write an AutoIndexer.

Implementation Requirements

You'll read two input files:
1) the text of the book separated by page indications
2) the keywords files that Prof. Jackson has been constructing while editing.

The full index will be written to a third file output will be written to a third file.

Book Text Input File

This file will contain the text that Professor Jackson is editing. The book is currently in raw text (ASCII); layout and page design isn't part of her job. 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. Listing 1 gives an example of a book text file:

```
<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.

The Keyword Input File

The input text file will contain a list of keywords and phrases from the book that need to be indexed. When making it, Prof. Jackson didn't pay attention to letter-case, so you'll need to account for that in your program. This means that 'tree' and 'Tree' should be considered as the same word. Each word or

phrase to be indexed will be on a separate line. No line will exceed 80 characters. There is no relationship between the order of the index file and the order of appearance in the text. Additionally, because she edits one chapter at a time, items to be indexed may actually be repeated; don't have two entries in the index for that though. A simple Keyword Input File can be found in Listing 2.

```
b+ Tree
algorithm
analysis
doubly linked list
A*
```

Listing 2: Sample keyword file.

```
[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 3: 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 Index File

The output Index 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. An index entry will consist of the indexed word, a colon, then a list of page numbers where that word was found in ascending order. 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.

The DSVector Class

You don't have any idea how many individual words, index entries, etc. will be present in the input data file. And since 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 vector, **DSVector** specifically. You'll need to implement a DSVector class that should minimally include the following features/functionality:

- contiguously allocated sequential container
- homogeneously typed, but should be able to hold any type (in other words, you class should be templated)
- grow as needed
    - In other words, don't start with an array of 500,000 elements. Start with a modest size (10 sounds like a good place to start) and **double the size as needed**.
    - You are permitted to have a constructor that accepts an 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 client code of DSVector.
    - access elements using the [] operator
    - remove an element from the container given a particular value or an index.
    - 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. Your DSVector Class should be thoroughly tested using CATCH2.

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.

Assumptions

You may make the following simplifying assumptions in your project:

- The input file will be properly formatted according to the rules above
- You need to remove punctuation from the input file words. 'Data!!!' and 'data' should be considered the same word
- 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).
- A word may appear as part of more than one phrase or as an individual keyword. For example "algorithm" and "algorithm analysis".

Execution

There will be two modes of execution for this project:

1. CATCH2 test mode - No command line args
2. run mode - 3 command line args
    a. book text file name
    b. keyword file name
    c. index output file name

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
Reasonable OO Design of source code base	20
Dynamic Memory Management including minimal mem leaks	10
Book Indexing functionality	30
Source Code Quality including Comments	10