AutoIndexer

Due: Sept 28, 2015 submitted to GitHub.

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.

Your Task

You will implement a piece of software that can read in Professor Jackson's keyword file (raw ASCII text with page indications), process the keyword data from the book, and output the complete index to separate file. All of this must be done within specific implementation constraints described in the forthcoming sections.

Implementation Details

You'll read from the ASCII text file generated by Prof Jackson. We'll call this the **intput text file**. Once you read in all of the data and process it, you'll write the index to an output file. We'll call this the **output text file**.

The Input Text File

The **input text file** will contain a list of keywords and phrases from the book separated into groups based on the page each word or phrase appears on. The end of the list of keywords will be indicated by <-1> at the end of the file. If a phrase is to be indexed, the words that comprise the phrase will be surrounded by square brackets (ex: [binary search tree]). No index word or phrase will exceed 40 characters in length (not including square brackets for phrases).

Here are a few things you should know about Prof. Jackson's messy style for keeping track of the keywords. She 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. Page numbers will appear in angle brackets (ex: <8>) and will always be on their own individual line. Page number will not necessarily be in order. Because of the editing process, Jackson may accidentally repeat page numbers due to re-reading the same section multiple times. This may mean she accidentally lists a word twice on the same page. In this case, there's no need to list the word or phrase twice in the index. A (very very) simple input text file can be found in **Listing 1**.

```
<15>
algorithm [binary tree] analysis heap
<1>
[binary search tree] analysis
complexity algorithm [2-3 tree]
<5>
tree [b+ tree] [Binary Tree]
<8>
graph clique Tree
<5>
tree [full binary tree]
[complete binary tree]
<-1>
```

Listing 1: Sample Input Text 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
[G]
graph: 8
[H]
heap: 15
[T]
tree: 5, 8
```

Listing 2: Sample Output Text File.

The Output Text File

The **output text file** will be organized in ascending order with numeric index categories appearing before alphabetic categories. Each category header 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 50 characters. The line should wrap before 50 characters and subsequent lines for that particular index entry should be indented 4 spaces. An example output text file can be found in **Listing 2**.

Prof. Jackson's Peculiarities with C++ Dev

Professor Jackson is a purist and doesn't trust many of the container classes and algorithms from the C++ standard library. ***Therefore, she has instructed you to not use any of them***. This includes her *aversion* to string objects. However, through much pressuring from her students and colleagues, she has come to accept and trust the streaming libraries that are part of the STL (iostream, fstream, stringstream.). Jackson trusts your skills though, so she encourages you to implement your own container class(es). Prof Jackson is also a sticker for efficiency of memory usage. So, she requires some very strict limits/constraints on memory management. See the section on Data Structure Implementation for more info.

Data Structure Implementation

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. Using dynamic memory allocation smartly, you can simulate the idea of an array growing in capacity. Therefore, this would allow you to handle really small books as well as really large books without being insensitive to memory usage issues. There is no need (and you definitely shouldn't) allocate arrays with 50,000 elements and cross your fingers in hopes that you'll not encounter 50,001 items that need to go into your array.
In particular, you'll implement functionality that will resize your arrays as needed for the various different dimensions of your data structure (such as words, pages, page list, etc.). Figure 1 shows an overview of a potential memory layout for this project.

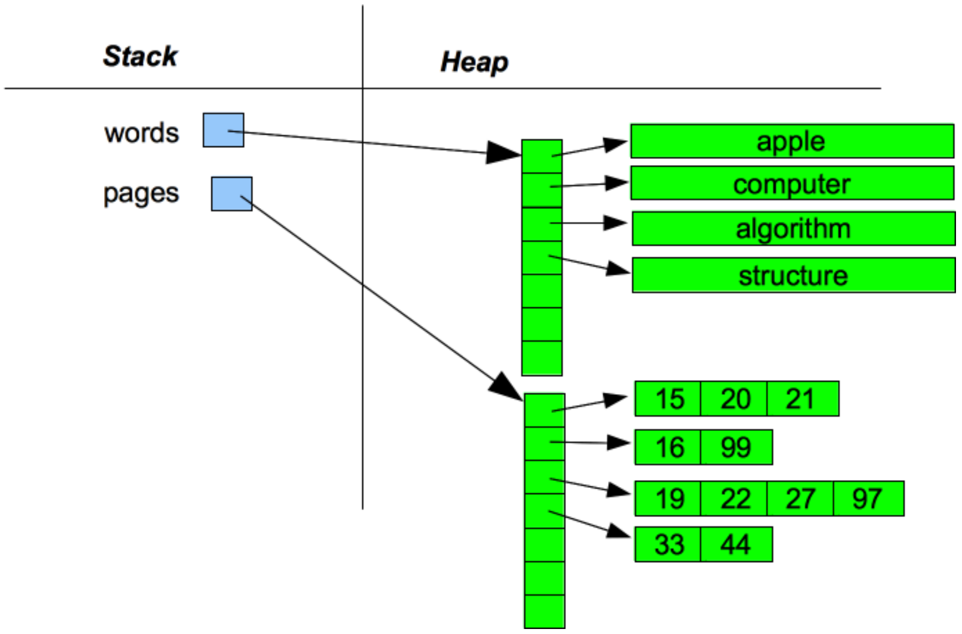

Figure 1 – Possible Memory Layout

At a minimum, your implementation should contain a pointer to a char pointer (char**) and a pointer to an int pointer (int**). You may have other data members as you see fit. Your implementation will likely always have some "extra" space to store more words and their page lists. **However, you may never have more than 10 unused spaces in your data structure in any particular array.** Of course, for the purposes of reading from the input text file, because the lines of text may be up to 80 characters, you may have a statically allocated array big enough to hold one line.

Assumptions

You may make the following simplifying assumptions in your project:

- The input file will be properly formatted according to the rules above
- With the exception of angle brackets and square brackets for page number and phrases, you may ignore the presence of other punctuation marks. For example, "Data" and "Data!!!" would be considered two different index entries.
- No line of text in the input file will contain more than 80 characters
- No word or phrase will be longer than 40 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 individual words)

Execution

The executable for this project will be run from the command line with two arguments:

- the name of the input text file,
- the name of the output file to write the index to.

Example:

```
prompt$ ./indexer input.txt index.output
```

What to Submit

You should submit:

- well formatted and documented source code
- any design documents you created up front in order to help you get started on the project
  - o Keep anything you jot down while thinking about how to structure the project. Scan it, take a picture of it, or otherwise reproduce it as part of your submission.
- any sample data files you used to test your program.

Strategies for Success

Just some friendly words of wisdom from your professor and TAs:

- The first 10% of a project is always the hardest. Don't sit down in front of an empty .cpp file hoping/waiting for inspiration. This is likely to turn into exasperation, desperation, exhaustion, etc. very quickly
- THINK BEFORE YOU CODE.
  - o Design before you start. Draw class diagrams; connect the classes with lines. Brainstorm about what classes/functionality you'd need to make this happen. Think about the major steps of processing that you'll have to go through. ***Do this step with a***

friend/buddy/pal/BFF that's in the class. That is completely acceptable. Challenge each other's design. Critique. Question. Explore.

- Consider the analogue of writing a paper by starting with an outline. After reading this handout in detail, what are the big "roman numeral" things that have to get done. Try to keep the list to 5 or less big tasks. Write them down (or type them into a Word doc). Break each of them into smaller tasks.
- When coding, THINK BEFORE YOU TYPE. You don't want a carpenter to start randomly putting nails in walls or drilling holes in your ceiling before they measure, re-measure, think about it, etc. Don't just mindlessly write code. Be intentional about every line you write.

Your TA's will also give you their guidance in each of the respective labs. Please don't dismiss our suggestions; they come from experience of making many mistakes. This is a completely do-able project in the time frame you've been given as long as you use your time wisely.

Grading

	Points Possible	**Points Awarded**
Correct Underlying Data Structure Implementation	30	
Complete functionality implemented	30	
Dynamic memory managed correctly	20	
Proper class infrastructure (constructors, destructors, accessors, mutators, etc.) and design	10	
Class documentation, formatting, comments, design documents	10	