CS 214: Systems Programming, Spring 2015
Programming Assignment 3: Indexer

Warning: As you will see below, the descriptions of the assignments will be increasingly complex because we are asking you to build increasingly bigger programs. *Make sure to read the assignment carefully!* This is critical because this document essentially describes the requirements for your program.

1. Introduction

In this assignment you will practice using the file system API (as well as pointers in different data structures). In particular you will be creating, opening, reading, writing, and deleting files.

Your task is to write an indexing program, called an *indexer*. Given a set of files, an indexer will parse the files and create an *inverted index*, which maps each token found in the files to the subset of files that contain that token. In your indexer, you will also maintain the frequency with which each token appears in each file.

Here is an example of how the indexer should work. If you are given the following set of files:

File Name	File Content
boo	A dog name name Boo
baa	A cat name Baa

The indexer should read the files and produce the following inverted index, in sorted order by word:

"a" → ("boo", 1), ("baa", 1)

"baa" → ("baa", 1)

"boo" → ("boo", 1)

"cat" → ("baa", 1)

"dog" → ("boo", 1)

"name" → ("boo", 2), ("baa", 1)

After constructing the entire inverted index in memory, the indexer will save it to a file.

Some observations:

- An inverted index is just a sequence of mappings, where each mapping maps a token (e.g., "dog") to a list of records, with each record containing the name of a file whose content contains the token and the frequency with which the token appears in the file.

- The above depiction just gives a logical view of the inverted index. In your program, you have to define data structures to hold the mappings (token → list) and the records (file name, count).
- The mappings are maintained in sorted order of the tokens. You will see later why this is useful. Sorting in ascending or descending order doesn't matter so much. We will just arbitrarily say for this assignment that the sequence should be maintained in ascending sorted order based on the ASCII coding of characters (i.e., "a" before "b" and "aa" before "ab").
- Records in each list are maintained in descending sorted order based on frequency counts of the tokens in the files. Again, you will see later why this is useful.
- Tokens are not case sensitive. All upper-case letters should be converted to lower-case.
- The tokenizer and sorted-list that you wrote in earlier assignments may be useful for this assignment (although you have to modify the tokenizer to work with a file, rather than a string). You also have the option of using the improved tokenizer.c file attached to this assignment.

2. Implementation

Since you are implementing a program in this assignment, there is no programming interface to follow. Instead, your program must implement the following command-line interface:

index <inverted-index file name> <directory or file name>

The first argument, <inverted-index file name>, gives the name of a file that you should create to hold your inverted index. The second argument, <directory or file name>, gives the name of the directory or file that your indexer should index. If the second argument is a directory, you need to recursively index all files in the directory (and its sub-directories). If the second argument is a file, you just need to index that single file.

When indexing files in a directory, you may have files that have the same name in separate directories. You should prepend the pathname (relative to the input directory name) in every record in the inverted index, rather than just the file name.

Tokenization is a little different in this assignment than in the previous assignment. You are not given a set of separators. Instead, we define tokens as any sequence of consecutive alphanumeric characters (a-z, A-Z, 0-9). All other characters are separators. Note that you can use the entire ASCII coding minus the alphanumeric characters as your separators to minimize the change to your tokenizer. But, this is not efficient, since the alphanumeric characters is only a small subset of ASCII. This would be even truer if we extend the character set beyond ASCII. So, you should not take this easy way out.

Examples of tokens according to the above definition include:

a, aba, c123, 1, 454

If a file contains

This an$example12 mail@rutgers

it should tokenize to

this an example12 mail rutgers

The inverted index file that your indexer writes must follow the following format, where I'm showing each space as an _ to make it more clear:

<list>_token

name1_count1_name2_count2_name3_count3_name4_count4_name5_count5

name6_count6_name7_count7_name8_count8_name9_count9_name10_count10

...

</list>

with the lists arranged in ascending sorted order of the tokens. Note you must obey the line breaks as shown. Each line containing the (file name, count) records can contain at most 5 records.

So, the example inverted index from Section 1 could look like:

<list> a

boo 1 baa 1

</list>

<list> baa

baa 1

</list>

<list> boo

boo 1

</list>

<list> cat

baa 1

</list>

<list> dog

boo 1

<list> name

boo 2 baa 1

</list>

This format is quite inefficient in a number of ways. We will optimize later. For now, we want to be able to easily read the inverted index for debugging.

You should carefully consider preexisting conditions and how they relate to the execution of your program. You should outline and implement a strategy to deal with these circumstances. For example, if a file already exists with the same name as the inverted-index file name, you should ask the user if they wish to overwrite it. If the name of the directory or file you are to index does not exist, your indexer should print an error message and exit gracefully rather than crash. There are many other error cases that you will need to consider.

You should use multi-file compilation to carefully organize your code. For example, the tokenizer should be in its own .c file, with a .h file that callers should include. The same applies for the sorted-list. You should also write a Makefile to efficiently compile and link your indexer.

3. Hints

- Data structures that might be useful include the sorted list you just implemented and/or a hash table.
- A custom record type (e.g., a record {"baa", 3}) that can be inserted into multiple data structures, such as a sorted list and/or a hash table).
- You can use your sorted list to maintain the set of tokens in ascending order. Since we are asking for the records for each token to be sorted in descending order by frequency, you have to flip the meaning of < and > in your record comparator function.
- You should probably approach this in steps.
    - First, you might get your tokenizer to generate correct tokens from a file.
    - Next, you might get your program to walk through a directory.
    - Next, you might implement a data structure that allows you to count the number of occurrences of each unique token in a file.
    - And so on …

4. What to Turn In

A tarred gzipped file name pa3.tgz that contains a directory called pa3 with the following files in it:

- All the .h and .c files necessary to produce an executable named index.
- A makefile used to compile and produce the index executable. It must have a target clean to prepare a fresh compilation of everything.
- A file called testplan.txt that contains a test plan for your indexer. You should include the example files and/or directories that you test your indexer on but keep these from being too large, please. (We might test your program with a very large data set though so don't skip testing your program for scalability. In your test plan, you should discuss the larger scale testing and the results, but you can skip including the data set).
- A readme.pdf file that describes the design of your indexer. This should also include the usual analysis of time and space usage of your program. Starting in this assignment, you do not need analyze every single function. Rather, you need to analyze the overall program. (So, for example, analyzing initialization code is typically not too important unless this initialization depends on the size of the inputs.)

As usual, your grade will be based on:

- Correctness (how well your code is working),
- Quality of your design (did you use reasonable algorithms),
- Quality of your code (how well written your code is, including modularity and comments),
- Efficiency (of your implementation), and
- Testing thoroughness (quality of your test cases).