CS 214: Systems Programming, Spring 2018
Assignment 2: Keyspace Construction

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!*

0. 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. The indexer should tokenize the files and produce an inverted index of how many times the word occurred in each file, sorted by word. Your output should be in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<fileIndex>
	<word text=”word0”>
		<file name=”filename0”>count0</file>
		<file name=”filename1”>count1</file>
	</word>
	<word text=”word1”>
		<file name=”filename2”>count2</file>
		<file name=”filename3”>count3</file>
		<file name=”filename4”>count4</file>
	</word>
</fileIndex>
```

The above depiction gives a logical view of the inverted index. In your program, you have to define data structures to hold the mappings (token to list) and the records (file name, count).

An inverted index is 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 that filename.

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

File Path	File Content
/adir/boo	A dog named named Boo
/adir/baa	A cat named Baa
/adir/bdir/baa	Cat cat

Your indexer should output:

```
<?xml version="1.0" encoding="UTF-8"?>
<fileIndex>
        <word text=”a”>
                <file name=”baa”>1</file>
                <file name=”boo”>1</file>
        </word>
        <word text=”baa”>
                <file name=”baa”>1</file>
        </word>
        <word text=”boo”>
                <file name=”boo”>1</file>
        </word>
        <word text=”cat”>
                <file name=”baa”>3</file>
        </word>
        <word text=”dog”>
                <file name=”boo”>1</file>
        </word>
        <word text=”name”>
                <file name=”boo”>2</file>
                <file name=”baa”>1</file>
        </word>
</fileIndex>
```

The inverted index file that your indexer writes must follow the XML format defined above. Words must be sorted in alphanumeric order. All characters of a word should be first converted to lowercase before the word is counted. Your output should print with the lists arranged in alphanumeric order (a to z, 0 to 9) of the tokens. The filenames in your output should be in descending order by frequency count (highest frequency to lowest frequency).If there is a word with the same frequency in two or more files, order them by path name alphanumerically (a to z, 0 to 9).

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

2. Implementation

Your program must implement the following command-line interface:

```
invertedIndex <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 combine all token frequencies for the same filename regardless of which directory it appears in. We define tokens as any sequence of consecutive alphanumeric characters (a-z, A-Z, 0-9) starting with an alphabetic character.

Examples of tokens according to the above definition include:

```
a, aba, c123
```

If a file contains

```
This an$example12 mail@rutgers
```

it should tokenize to

```
this
an
example12
mail
rutgers
```

The XML format lets us easily read the inverted index for debugging. You should carefully consider how the program may break and code it robustly. You should outline and implement a strategy to deal with potential problems and inconsistencies. 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 want 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 ought to consider.

3. Hints

Data structures that might be useful include a list that sorts as you insert 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 should probably approach this in steps:

o First, build a simple tokenizer to parse tokens from a file.
o Next, get your program to walk through a directory.
o Next, 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 named Asst2.tgz that contains a directory called Asst2 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 invertedIndex 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. (In your test plan, you should discuss the larger scale testing and the results, but you can skip including huge data sets).

A readme.pdf file that describes the design of your indexer.

This should also include an analysis of time and space usage of your program.

You do not need to 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)
Testing thoroughness (quality of your test cases).