CS214: Systems Programming
Assignment 2: Recursive Indexing by Eric Giovannini and Chris Gong

I. Objective

The point of this project is to create an indexer that utilizes file system functions which include creating, opening, reading, and writing files. The program will traverse through a given directory or file and parse the files. The result should be an inverted index file that maps each token in the files that were traversed through. The XML file should first list out all the words that appear throughout the traversed directory/file and then for each word, it lists the associated files and corresponding frequencies. So after running the make command in the terminal,

```
[cg646@null cg646]$ make
```

you'll get an executable called indexer. Then to execute the program, another command is needed in this format, *./indexer <inverted-index file name> <directory or file name>*

```
[cg646@null cg646]$ ./indexer xml directory
```

The above command will write the inverted-index file to a file called xml inside the directory that holds the executable whether a file called xml exists or not. If xml already exists, then it'll prompt the user on whether or not the program is allowed to overwrite the file. And then the directory/file called directory will be parsed by the indexer in order to appropriately write up the XML file, which can look something like this with the given set of files,

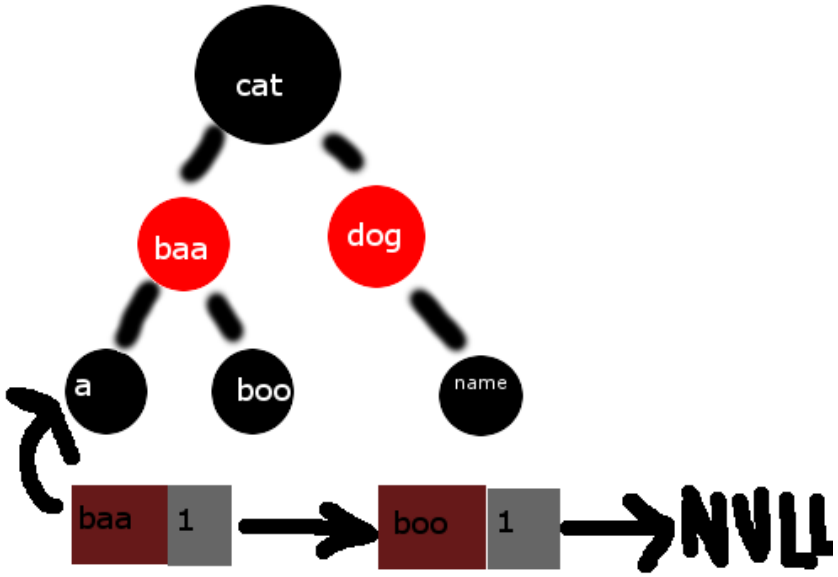
File Path	File Content
/adir/boo	A dog named named Boo
/adir/baa	A cat named Baa
/adir/bdir/baa	Cat cat

```
<?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>
```

II. Data Structures Involved

So in order to parse the inputted file or directory's files, we need to keep track of all the words found and their counts. So in order to do this, we use a redblack tree whose implementation can be found in *redblack.c* and *redblack.h* to hold each token. The benefit to this is that the words will be sorted in order (letters in alphabetical then numbers in ascending order). Inserting the words will take O(nlogn) time with n being the total number of words. Each word, however,

contains a linked list of nodes that contain their associated filename and frequency. Inserting all the nodes to the linked list and updating frequencies of existing nodes can take up to O(n^2) time with n being the total number of times a word showing up in a directory/file because if nodes towards the end of the list get updated more often, the program will run slower. The implementation for the linked list can be found in *linkedlist.c* and *linkedlist.h*. Once all the files have been traversed through, a merge sort is run on every single red black tree's node's linked list so that each linked list ends up being sorted by descending count. And if a tie occurs between two file's counts, the order of tied linked list nodes is by filename in ascending order (numbers come after letters). Here is a simple diagram depicting the structures involved,



III. Traversing Directories/Files

The main part of this program are how we utilized file system API functions in two methods in *invertedIndex.c* called *traverseFile* and *traverseDir*. First, we check if the second argument is a file or directory using *opendir*. If *opendir* returns null, then the inputted file path points to a regular file assuming no error codes have been activated. If it doesn't return null, then the inputted file path points to a directory.

In the case where the file path points to a directory, we called *traverseDir*. This method makes a call to *opendir* as well. But what's the point if we already know it's a directory? The point is that this is a recursive method. Because once we open the directory, we begin to call *readdir* to traverse through the elements within the directory. Then we check whether the element is a file or directory by checking its *d_type* (check out C's implementation of the *direct* struct for more information). After that, if it's a file we call *traverseFile* on its path. On the other hand, if it's a directory, we make a recursive call to *traverseDir* on the new directory's path.

traverseFile is very different than *traverseDir* in the sense that it is more responsible for filling up the red black tree and linked list data structures. *traverseDir* was more responsible for making sure each file got called by *traverseFile.* Whereas *traverseFile* calls *open* to make the file useable for reading. We also use *lseek* to get the file's length in bytes. Then, we read in 20 bytes

at a time from the file using *read* and parse each buffer of 20 bytes at a time. This way, less system I/O calls are made as well as malloc and free calls (since we avoiding realloc'ing one byte at a time), drastically making our program faster.

The last part of the program consisted of actually writing the XML, which was simply done using *fprintf* and *fopen*. *fopen* creates or overwrites a file for writing purposes. And while doing an inorder traversal of the red black tree, and traversing each of its nodes' linked lists, we do *fprintf* statements to properly load up the inverted-index file.