Kenneth Zhang and Kevin Pei

CS 214: Systems Programming, Spring 2017
Assignment 2: Recursive Indexing

Function

This program takes two inputs: a file to write to and either a file or directory to read from. If a directory is given to read from, this program recursively enters the directory and any directories it contains and returns all files within them. Then, every token within the files is put into a hashtable, mapping to a hashtable of file names containing the frequency of the appearance of the given token in that file. A token is defined as an alphanumerical string beginning with an alphabetical character. After creating the hashtable of tokens, it is then converted to XML and an alphabetical list of tokens with a list of the files they appear in, sorted by frequency, is written into the file chosen to write to. If the file to write to does not exist, one is created; otherwise, the user will be prompted whether they want to overwrite it.

Input

The input should be of the form:

> ./invertedIndex <output file name> <directory or file to read from>

The program is coded so that it must accept 3 inputs (program name, output file name, and the directory or file to read from). If there aren't 3 inputs, then it prints:

> You did not pass enough arguments. Please indicate inverted_index file and target file/directory...

and closes. If the output file name you provide already exists, then it prints:

> An output file or with same the name already exists...
> Would you like to override the file...?
> Enter 'y' to override or 'n' to choose another name or press Ctrl-C to exit the program...

If you press 'y', then your output will override the file that was already there. If you press 'n', then it prints:

> Choose a new name for your file...

And the program creates a new output file with that name. If it once again matches the name of a file already in the directory, it will print the following:

> A file/directory with that name already exists. Please try again...

If you do not enter 'y' or 'n', then it prints:

Invalid input. Please reply with 'y' or 'n'...

And prompts you again.

Structure

The program is contained in invertedIndex.c, with header files in readAndWriteFile.h and dirFunctions.h. invertedIndex.c includes the main function and all utility functions needed for its operation, such as searchDir, readFile, and getToken.

Sorting

The tokens are output in alphabetical order. For each token, the files it appears in are listed first by frequency, and if the frequencies are the same, by alphabetical order. Files with the same name under different directories have their frequencies aggregated. The alphabetical sorting is done using the default strcmp() sorting, with numbers coming before letters. All tokens are converted to lowercase and results of tokens with different cases are aggregated.

Output

The program outputs an XML file which lists the string tokens in ascending alphabetical order. Under each string token are the files it is contained in, in descending order by frequency of appearances in that file. If multiple files have the same frequency, they are sorted in ascending alphabetical order. Here is an example of the output:

```
<?xml version="1.0" encoding="UTF-8"?>
<fileIndex>
        <word text="a">
                <file name="Fun.txt">4</file>
                <file name="testcases.txt">2</file>
        </word>
        <word text="and">
                <file name="testcases.txt">5</file>
                <file name="Fun.txt">4</file>
                <file name="Hints.txt">4</file>
        </word>
        <word text="empty">
                <file name="testcases.txt">1</file>
        </word>
</fileIndex>
```

Analysis

In order to create our hashtable, we create two structs: stringTable to hold our tokens, and fileTable to hold the files for each string. Each stringTable contains its token, a pointer to the first fileTable in its file hashtable, and a pointer to the next stringTable in the hashtable. fileTable contains its file name, the given token's frequency in that file, and a pointer to the next fileTable in the file hashtable. Because the hashtables for tokens and files are linked lists, iterating through them takes linear time O(n). The worst case scenario for analyzing a token in a file would be O(nm), where n is the size of the token hashtable and m is the size of its corresponding file hashtable.

Our buffer reads the file 1 byte at a time, which would result in a high run time. Additionally, because we analyze each character for whether it is non-alphanumerical, there is a high run time associated with each character we add to the token.