Joseph Klaszky
Mohammad Memon

Overview of how our code works:

The main struct that will hold most of our data is a Binary Search Tree (BST). Inside each BST node we'll hold: a string token, pointers to the next two nodes in the tree, and a pointer to a linked list. The linked list will hold the names of the files and the frequency of appearance. We chose a BST because it has a relativity simple implementation, but it will sort it self as items are inserted and insertion has an average O(log(n)), where n = number of nodes. Then the linked list that is contained within the BST nodes was chosen for its simplicity. We didn't need to do anything interesting with these linked lists other then occasional search and insertion, both worst case O(n), where n = number of links.. We didn't bother sorting them as we went, we just made a quick selection sort for the linked list, O(n^2), where n = number of links, that we called just before the writing the output.

The main loop of the code was run by our function fileIterator(), O(n), where n = total number of files in the main directory and each subdirectory. FileIterator() opens each directory and file and pulls the contents of each file it finds with extract(), O(n), where n = number of characters in the file. We then tokenize() the contents of the file. Tokenize() iterates over the contents of the file and looks for anything we define as a string. When it finds one it pulls it out with pullString() and puts it into the tree with addToTree().

After all files have been iterated over and all data has been pulled and tokenized we simply call our function finalOutput() that performs an in-order traversal of our tree and prints out the xml tags, the string token and the all the files that these tokens were found in. Finally we destroy our tree and all of the linked lists contained within with destroyTree().

Efficiency:

The overall time complexity is somewhat hard to put a number on. There are so many functions that depend on different types of input to define their complexity. If we're just going to look at all of our functions and define the overall complexity based on the least efficient one then the big-O would be O(n^2), where n = the number of links in any given node, because we chose to use selection sort for out linked list. However, I would say looking at the broader picture, the things that the code spends most of its time doing are opening files and pull the data within. These two things are done in linear time and depend on the total number of files to be looked at and the sizes of those files.

The space complexity is based on the number of files examined and number of unique string tokens contained within those files. For each unique string token there will be a BST node created with that will contain three pointers and a hold a string. Also, for each unique string token one of the pointers in the node will point to a linked list. Each link in the list will contain one pointer and a string with a file's name and an int. It was possible to use a short instead of an int, but we felt it was better to be safe than sorry.

Breakdown of the most important functions:

main()

- Create our BST head node that will store our tokens
- Call: fileIterator(thrid cmdln arg, head of BST)

- Call: finalOutput(head, second cmdln arg)
- Call: destroyTree(head)
- Return

fileItreator(char * currentPath, treeNode * head)

- The first few lines are just setting up vars to pull out the data for what ever files we're looking at and doing some error checking for exceptional cases.
- After that is the main loop of the whole project. readdir(dir) will return null when it hits the end of the a directory so it'll keep looping as long as there are files or directories left to explore.
- While(readdir != null): if it hits a directory, call fileIterator() on that directory, if it hits a file, pull the contents with extract() and tokenize() them.
- Always returns a head node, but it will make the node null if there were exceptional issues that couldn't be resolved.

extract(char * pathToAFile)

- Error check access.
- Error check for ability to open the given file path.
- Error check for empty file.
- Open file.
- If all of that is cool, it pull the information out using the annoying, nonblocking read() and a while loop.
- Closes file.
- Return null if there was an error or the file's contents if all was good.

tokenize(char * fileContents, treeNode * head, char * currentFile)

- Error check for empty file. Redundant, but why not?
- Iterator over contents until hitting an alpha character. Starts a counter that keeps track until it hits a non-alpha, non-numeric character or the end of the file.
- It then pulls that string of characters with pullString()
- It then searches the BST:
    - if that node is in the BST, it searches that node's linked list of files and either inserts a new link or if the file was already in there it increases the counter that keeps track of frequency.
    - If that node isn't in the BST, it creates a new node and puts it in at it's appropriate place.
- Returns the updated BST

finalOutput(treeNode * head, char * outputFileName)

- Error check for access.
- Error check to open the file.
- Open outputFile.
- Check if BST is empty and warn the user.
- Then is a lot of non-blocking write() calls with while loops to make sure they actually output everything I want them to write. (I know this could have been done a lot easier with other library functions, but the Professor seemed to want us to use these low level ones)

- Traverses the tree recursively by alphabetic order. For each node it prints out the string token and then every file it appears in and how many times it appeared in said file.
- Close(file)

destroyTree(treeNode * head)

- Recursively traverses the tree going to the deepest depth first.
- Upon hitting the deepest Node it recursively frees the linked list associated with each node.
- Then frees the string token associated with the node, then frees the node it self and continues up the tree.