Claudia Pan cp728

Yelin Shin ys521

The general design of our code is that it takes two arguments from argv. Argv[1] represents where our outputted inverted index file will go and argv[2] represents the path that we must index. If all parameters are given correctly, and argv[2] is a path to a directory, we will call our indexer. The indexer is a recursive function that keeps calling itself if the new path is also a directory. If the path is a file, we will call tokenize and the function will make valid tokens of the file. The tokenizer is in the form of a link list and every time a new token is created we will insert it to the list sorted (alphabetically). After all files and directories have been checked we will call our sort_count function which we resort our link list based on the highest frequency. A nice thing we did with the sorting is that we have a while loop that checks through the whole link list but within that is a while loop that will detect all tokens with the same name and different file names and only sort that portion based on frequency and then continues to look if the rest of the tokens need sorting. After all tokens are in the correct order, we called createFile which will make the output file with the indexed tokens in xml format. Of course throughout the code we have error statements to check if the inputs are correct, or if such file/ directory exists and so on.

Based on our program using linked list for the tokens, I would give our function analysis as big O(n). When we insert new tokens, we always have to search through each token until we find the correct spot, the same goes for sort_count and the createFile function. The program space usage is all based on the amount of tokens we would have to create from the files. So depending on our testcases, our size usage can be very large or just minimal.