Zack Colello & Anna Genke

Assignment 4: Search

Search consists of 3 .c files: search.c, parsefile.c, and bst.c. A makefile creates a search executable with the interface

```
./search <inputfile>
```

where <inputfile> is is the index file .txt created by the index. In our pa4, we included the files necessary to run ./index and generate the index file to be used by Search (tokenizer.c, index.c, and sorted-list.c).

Interface

After running ./search on a valid index text file, the user can do a search of this index with the following commands:

so <terms> : Search **OR**. The program will output all paths that have at least one of the terms given.

sa <terms> : Search **AND**. The program outputs all paths that have all of the terms given.

sxo <terms> : Search **XOR** (exclusive or). The program outputs all paths that have only **one** of the terms given.

sno <terms> : Search **NOR**. The program outputs all paths that have **none** of the terms given.

sna <terms> : Search **NAND**. The program outputs all paths that do **not** have **all** of the terms given.

q: Quit and return out of the program.

Inverted Index Format

Running ./index on a file or directory outputs a text file with a name of the user's choice. It generates an alphabetical list of all words found in the file or directory, along with the files that contain each word, in the following format for each word found in the file/directory:

```
<list> (word)
(filepath), (wordcount)
(filepath), (wordcount)
… //potentially more filepaths here
</list>
```

Essentially, each word found in a file/directory has “<list>” written before it, to signal that we are looking at a word, and not a path. After we find a word, we list every path that contains at least one instance of that word. We print a comma after the path, followed by the number of times that word appears in the file (wordcount). After all paths with that word have been printed, we mark the end of the list of paths with a line that just contains </list> to tell our program we are done with that word. Each word follows this format.

Memory Requirements

For our Search program, we utilized the O(*logn*) search time of a binary search tree for the searching algorithm. First, we generate a 2-D linked list from the inputted index file with our **buildLL** function. This returns a list that contains the head of a list of words found in the index file, where each word node (called tokenNode) contains a list of files (fileNodes) that contain that word. Since our index file is already organized, we can create this 2D linked list in O(*n*) time with O(*n*) space, where n is the number of lines (words + file paths).

After generating the linked list, we generate another linked list that contains only the file paths (useful for our sno and sna functions). This again takes O(*n*) time.

We then call our **LLtoBST** function to create a Binary Search Tree from our linked list to be used for searching. This is implemented again in O(*n*) time, as we build the BST from the ground up recursively, taking O(*n*) space.

We then go into a loop that reads in input from the user (ex: sa words).

We generate another linked list to get the count of the fileNodes to be used before we call the search function (again, O(*n*) time and space complexity).

After, we call the search function necessary depending on the user input in the loop with a switch statement. We traverse our BST to find the words, taking O(*logn)* time for each term.