Assignment 1: Building an Index

The objective of this assignment was to create an inverted file system from a given corpus of documents and provide a way to interact with the inverted file system created.

Program Structure:

There are two java class files which are being submitted:

- **InvertedFile.java**: Consists of methods and data structures to create and interact with an inverted file system. The constructor takes three attributes: path to the directory where the documents are kept, path to the file consisting of stop list words on separate lines and path to the output directory where the word list and postings file will be written.
- **UserInterface.java**: This class provides methods to handle the query entered by the user. Three attributes (directory path to corpus files, path to stop list file and directory for exporting word list and postings file) need to be provided to use the interface. These attributes are the ones as required by the InvertedFile class as explained above.
  In order to use the system run this class and provide the required attributes.

Solution Explanation:

The constructor for **UserInterface** takes three arguments: directory path to documents, path to stop list file and directory path to a directory where word list and postings file is to be created on disk. When this constructor is invoked, it creates an object of InvertedFile class with the arguments it gets.

The **prepareData** function is then called which in turn calls the **createStopList** function and then the **iterateOverDirectory** function. After these two function calls, the word list and postings file is ready and so we go ahead and create copies of them on the disk at the directory path provided. Wordlist is created in the file **wordlist.txt** and the postings file is called **postings.txt**. The system is now ready to take user queries. Depending on the query (single term or multi term) appropriate functions (**handleSingleTerm** or **handleMultipleTerms**) are called and the response is given to the user. This continues as long as the user does not enter 'ZZZ' as a query.

When the **createStopList** function is called, it goes over the stop list file and creates an array list of words. Using this data structure allows use of functions like *contains* which help later in checking whether a particular word is in the stop list or not.

The **iterateOverDirectory** function goes over each file in the directory as provided and builds up the word list. It takes one file at a time and calls the **parseFile** function which parses a file line by line and splits each line into respective words.

Java provides the TreeMap data structure and that is what I have used in this programming assignment. A tree map data structure always keeps its keys in the ascending order. If it was a regular hash map (HashMap) then the keys would be arranged in the order they are added. A tree map data structure is different as it always keeps keys sorted and is thus useful for creating an index as when new words are encountered they are added to the data structure in the sorted order. In my case the key would be the word.
I am using the TreeMap for a number of things in the program.

- **wordList** is a TreeMap which keeps track of the document frequency for the word. Its key is the word and value is the number of documents the word appears in. Once generated, it is written into the file wordlist.txt.
- **wordPosition** is a TreeMap which keeps track of where inside a particular document a particular word appears at i.e. the word position. The key here is the word and the value is another TreeMap. This other TreeMap has key as the document name and value as the array list consisting of positions of the word inside the document. Using TreeMap at the first level ensures that it is sorted by the words and using TreeMap at the next level ensures that file names are also sorted in the ascending order. Once generated, this wordPosition tree map is written into the file postings.txt in the appropriate format.

I also use TreeMap for storing document length (documentLength). Storing document length is helpful as it serves a double purpose: firstly, because it is keyed by document name, it helps in generating a listing of the documents in the corpus which is helpful while querying the system. Secondly, it helps in detecting boundary conditions while displaying the context a word appears in.

There are other functions in InvertedFile.java which help in computing the term frequency (**getTF** and **getTFInDocument**), inverse document frequency (**getIDF**) and displaying the context for a particular word (**printTermInDocument**). These functions are called appropriately as per the user query.

Assumptions:

- If all but **one** terms in a query are stop list words then the query is treated as a single term query and results provided will be the same as those if it only that single term was input.
- If there are duplicate terms in a query then the query will still be treated as a multi term query. For e.g. if the query entered is **good good** then it will be treated as two terms and not a single term. Same applies if there are other terms also in the query i.e. **good weather good** is different from **good weather**.
- The stop list file always consists of terms which are on separate lines.
- Hidden files and sub directories inside the directory path where documents are located are ignored.

Execution Instructions:

Unzip **a1.zip** to get this Report, the Binary, Data, and Source folders. The Data folder consists of the word list (wordlist.txt) generated as a result of running the program on the given corpus of 40 documents. It also consists of the postings file (postings.txt) with information about the file a term appears in with the number of occurrences and the positions at which it appears. The Source folder consists of the src (consisting of the invert package which contains InvertedFile.java and UserInterface.java files) and bin (consisting of the invert package which contains InvertedFile.class and UserInterface.class) folders.

Inside the Binary folder there is a file called InvertedFile.jar. Go to the command prompt and type the following command:

> java -classpath **<path to InvertedFile.jar>** invert.UserInterface **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept> <directory path on system where the generated wordlist and postings file would be written>**

The system thus takes three command line arguments as shown above. Once running the system will request you to enter a query.

Another way to run the system is to take the bin folder inside the Source folder. Change directory to go inside the bin folder. There will be a folder called invert (the name of the package). There are classes (InvertedFile.class and UserInterface.class) inside this package. While inside the invert folder type the following command to run the system:

> java invert.UserInterface **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept> <directory path on system where the generated wordlist and postings file would be written>**

Alternatively, we can run the system by taking the src folder inside the Source folder and change the directory to go inside the src folder. The invert package here consists of the the .java files. While inside the invert folder type the following commands to run the system:

> 1. javac -classpath . invert/*
> 2. java invert.UserInterface **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept> <directory path on system where the generated wordlist and postings file would be written>**

Please note that the first command here needs to be run once only.

Note: Please use jdk 6. It is available on machines in the CSUGLAB

Sample Output:

1. Entering a single term (“higher”) we get the following output (first the IDF (Inverse Document Frequency) and then for each file the positions at which the word appears, the TF (Term Trequency), the TF.IDF score and then the context within the file:

```
Enter your query: higher

IDF = 2.0

file00.txt[2]: [215, 299]
TF = 1.3010299956639813
TF.IDF = 2.6020599913279625
Context is as follows:
rapid turnaround to support much higher flight rates Evolving systems

file18.txt[3]: [129, 251, 344]
TF = 1.4771212547196624
TF.IDF = 2.9542425094393248
Context is as follows:
was 2.45 metres 0.29 metres higher than the previous observed

file19.txt[1]: [649]
TF = 1.0
TF.IDF = 2.0
Context is as follows:
is expressed at a much higher concentration than any of

file37.txt[1]: [523]
TF = 1.0
TF.IDF = 2.0
Context is as follows:
top groups has led to higher cell efficiencies increasing the
```

2. Entering the multiple term query (“also good”) we get the following output (file name followed by TF.IDF score):

```
Enter your query: also good
file05.txt 3.9264962627083224
file04.txt 3.536367256164429
file22.txt 3.39655849739314
file02.txt 2.8676602488073955
file28.txt 2.8676602488073955
file36.txt 2.8676602488073955
```