DATA AND KNOWLEDGE
ENGINEERING GROUP

Information Retrieval

Programming Task P01
Winter 2019

Parse, index and search plain text and HTML documents with Apache Lucene

#	Student Name	Immatrikulation Number	Email
1	Adel Memariani	218186	adel.memariani@st.ovgu.de
2	Chetan Singh	223732	chetan.singh@st.ovgu.de
3	Harsh Solanki	226254	harsh.solanki@st.ovgu.de
4	Poornima Venkatesha	226075	poornima.venkatesha@st.ovgu.de
5	Sediqullah Muslim	223881	sediqullah.muslim@st.ovgu.de

Summary:

We are a team of 5 students and this document presents the way in which the given programming task is addressed by our team.

The code is available in a GitHub repository here:
https://github.com/sediq-khan/Lucene_Indexing_Analysing_Searching

It can be run with the following steps:

1- Navigate to the **out\artifacts\LuceneDemo_jar** directory through command line

2- Execute the program with the following command:

java -jar LuceneDemo.jar F:\Lucene\Data word-to-search F:\Lucene\Index

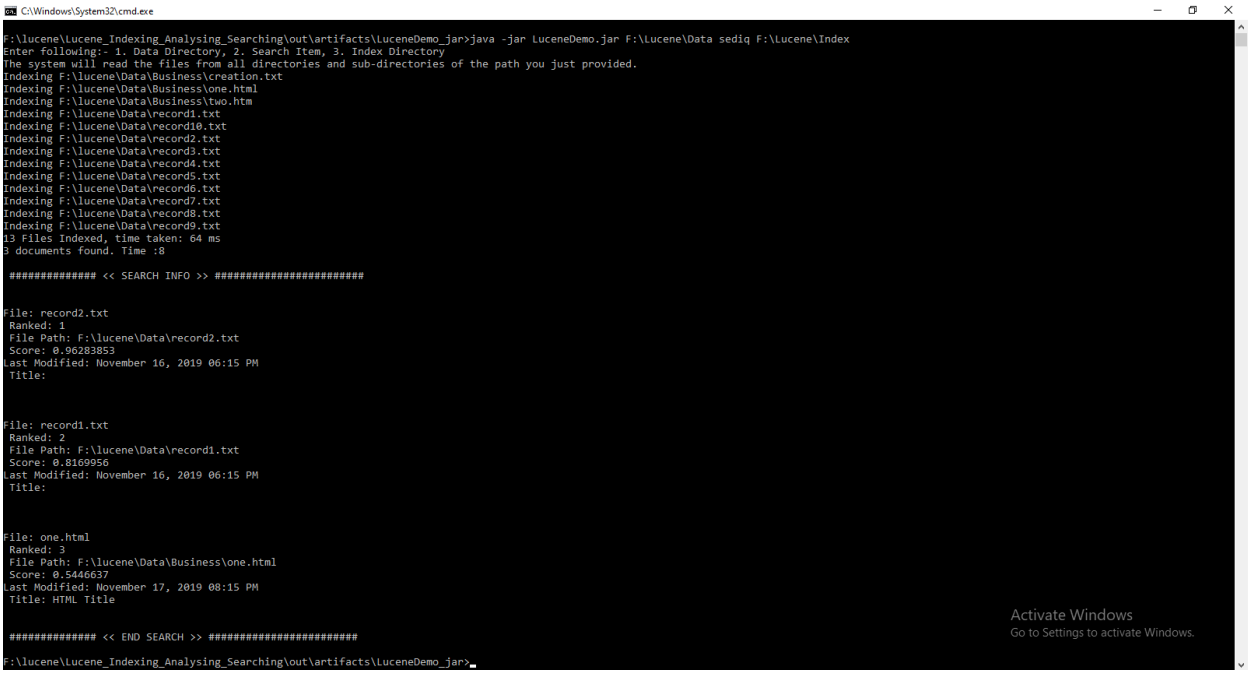
F:\Lucene\Data : Is the folder wich contains text and HTML files need to be indexed.

word-to-search : Is the search term.

F:\Lucene\Index : is the Index directory where lucene will place the output of index.

The output should look like the following figure:

note: the search term in the figure is “sediq”



Details:

The main functions are located in the following file:

Lucene_Indexing_Analysing_Searching-master\src\main.java

The Main class of the main.java consist of the following methods:

```java
public static void main(String[] args) throws IOException, ParseException {
    Main mainObject = new Main();

    System.out.println("Enter following:- 1. Data Directory, 2. Search Item, 3. Index Directory");

    if (args.length < 3){
        System.out.println("Please enter atleast the three said arguments.");
        return;
    }
    mainObject.searchItem = args[1].toLowerCase().trim();
    mainObject.userInputIndexDir = args[2].toLowerCase().trim().toString();
    mainObject.userInputDataDir = args[0].trim().toLowerCase();

    mainObject.createFileList();
    mainObject.createIndex();
    mainObject.search(mainObject.searchItem);
}
```

Figure 1 : Main method, when the user provides the required input, the rest of the operation is performed sequentially without the user interaction

The main() method main is expecting at least three parameters, without which it will not perform any other action.

```java
public void createFileList(){
    try {
        Path filePath = new File(userInputDataDir).toPath();
        boolean exists =      Files.exists(filePath);        // Check if the file exists
        boolean isDirectory = Files.isDirectory(filePath);   // Check if it's a directory
        boolean isFile =      Files.isRegularFile(filePath); // Check if it's a regular file
        System.out.println("The system will read the files from all directories and sub-directories of the path you just provided.");
        filesFromUserInputDir = fileExtensionLookup.fileExtensionLookup(userInputDataDir);
    }catch (Exception ex){
        System.out.println("<<< Please enter a valid directory or a file path. >>>");
    }
}
```

Figure 2 : createFileList() method

The createFileList() method, creates a list of all files that contain in the data directory and all its sub-directories. The list is is formed and then it is passed to the index function for further process.

```java
public void createIndex() throws IOException {
    luceneOperations = new LuceneOperations(userInputIndexDir);
    int numIndexed;
    long startTime = System.currentTimeMillis();
    numIndexed = luceneOperations.createIndex(filesFromUserInputDir);
    long endTime = System.currentTimeMillis();
    luceneOperations.close();
    System.out.println(numIndexed + " Files Indexed, time taken: " + (endTime - startTime) + " ms");

}
```

Figure 3 : createIndex() method in main() class

The luceneOperations() class has a constructor method which takes a directory and creates a writer object from the directory.

```java
public class LuceneOperations {

    private IndexWriter writer;

    public LuceneOperations(String indexDirectoryPath) throws IOException {
        Directory indexDirectory = FSDirectory.open(new File(indexDirectoryPath));

        writer = new IndexWriter(indexDirectory,
                new StandardAnalyzer(Version.LUCENE_36), create: true,
                IndexWriter.MaxFieldLength.UNLIMITED);
    }
```

Figure 4 : The constructor of luceneOperations() class

The createIndex() method of luceneOperations() class takes an object containing the list of files returns the indexed documents. This method, uses indexFile() method to do the indexing.

```java
public int createIndex(List<File> files) throws IOException{
    for (int i = 0; i < files.size(); i++){
        if (!files.get(i).isHidden()
                && files.get(i).exists()
                && files.get(i).canRead()
        ){
            indexFile(files.get(i));
        }
    }
    return writer.numDocs();
}
```

Figure 5 : The createIndex() method in luceneOperations() class

```java
private void indexFile(File file) throws IOException {
    File files = file;
    System.out.println("Indexing " + file.getCanonicalPath());
    Document document = getDocument(files);
    writer.addDocument(document);
}
```

Figure 6 : The indexFile() method in luceneOperations() class

The indexFile() method uses getDocument() method to create a document object containing three Apache Lucene Fields:

1- contentField
2- fileNameField
3- filePathField

We imported the following from the Lucene:

```
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
```

```java
private Document getDocument(File file) throws IOException{
    Document document = new Document();

    Field contentField = new Field( name: "CONTENTS", new FileReader(file));
    Field fileNameField = new Field( name: "FILE_NAME", file.getName(), Field.Store.YES, Field.Index.NOT_ANALYZED);
    Field filePathField = new Field( name: "FILE_PATH", file.getCanonicalPath(), Field.Store.YES, Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);
    return document;
}
```

Figure 7 : The getDocument() method

The search() method of main() class firstly, takes the search term as input and then search through the indices. It returns the documents wich contain the search term based on the ranking. The search() method uses Jsoup library to to pick the title for html documents as instructed by the task.

```java
private void search(String searchQuery) throws IOException {
    String htmlTitle = null;
    searcher = new Searcher(userInputIndexDir);
    TopDocs hits = null;
    long startTime = System.currentTimeMillis();
    try {

        hits = searcher.search(searchQuery);

    } catch (ParseException ex) {
        System.out.println("Parse Exception: " + ex.getMessage());
    }
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits + " documents found. Time :" + (endTime - startTime));
    int i = 0;
    System.out.println("\n ############## << SEARCH INFO >> ########################");
    String title = "";
    for(ScoreDoc scoreDoc : hits.scoreDocs) {
        i++;
        Document doc = searcher.getDocument(scoreDoc);
        File tempFile = new File(doc.get("FILE_PATH"));
        if (tempFile.getName().endsWith(".html") || tempFile.getName().endsWith(".htm")){
            org.jsoup.nodes.Document htmlDoc = Jsoup.parse(tempFile, charsetName: "UTF-8");
            title = htmlDoc.title();
        }
        long milliseconds = tempFile.lastModified();
        DateFormat format=new SimpleDateFormat( pattern: "MMMM dd, yyyy hh:mm a");
        long timeModified = tempFile.lastModified();
        System.out.println("\n");
        System.out.println("File: " + doc.get("FILE_NAME") + "\n" +
                " Ranked: " + i + "\n" +
                " File Path: " + doc.get("FILE_PATH") + "\n" +
                " Score: " + scoreDoc.score + "\n" +
                "Last Modified: " + format.format(milliseconds) +
                "\n Title: " + title
                + "\n"
        );
        title = "";
    }
    System.out.println("\n ############## << END SEARCH >> ########################");
    searcher.close();
}
```

Figure 8 : The search() method