SEARCH BENCHMARKING

Project Report Submitted to

Professor Arijit Das

International Technological University

SEN 850 – Data Structure

Fall 2014

Submitted By:

Shweta Verdia

Student ID: 84224

Abstract

The project Search Benchmarking aims at determining the performance comparison between the two major Data Structure algorithms i.e. Binary Search and Sequential Search, in terms of how quickly and efficiently each algorithm performs search operation on string in a text file. This project depicts implementation of Binary search and Sequential search, also Graphical User Interface is being developed to showcase the performance of both algorithm, which helps user in understanding the concept of searching mechanism and how properties of String search algorithms influence your choice of algorithm.

String searching algorithms are important in all sorts of applications that we meet every day. In text editors, we might want to search through a very large document (say, a million characters) for the occurrence of a given string (maybe dozens of characters). In text retrieval tools, we might potentially want to search through thousands of such documents (though normally these files would be indexed, making this unnecessary). Other applications might require string-matching algorithms as part of a more complex algorithm.

Therefore, Search Benchmarking search for all occurrences of a word in a list in order to find the word entered by user, whether it exist in the file or not, text file containing words will be given by user as input. The application returns the results if the word is found, as well as time taken by each search on GUI. For Binary search, we sort the items on a list into alphabetical order or dictionary order for binary search and for sequential search it iterate through every item of the list on unsorted data Java is used as programming language to develop the application and also the application is tested with the help of test cases. Test cases are written in JUnit framework to ensure that code written works fine without bugs.

Table of Contents

List of Figures

1. Introduction

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. Searching algorithms are used to find specific data in given data structures. As the quantity of the given data may be sufficiently large, the efficiency of the searching algorithm should be paid special attention to.

In this project Java based web application is developed using Spring MVC framework, Apache Tomcat as a server and html web page to display the results to the user. The web page will display the line number on which word exists, time taken by search algorithm and its complexity.

The data structure used behind the implementation is HashMap and ArrayList.

The HashMap is another sequence structure that can be used to store multiple objects in Java. HashMaps do not organize their data elements using an integer index value. Instead, HashMaps use object references as their indexes. In a map, an object that serves as an "index" is called a key. The object that is associated with a key is called a value. We used Strings as the index value, or “key” into the Hash Map. A key can have at most one associated value, but the same value can be associated to several different keys. When we construct a HashMap we specify two data types – they “key” and the “value.” The “key” data type is what we use to access an element in a HashMap. The “value” data type is the type of data that we want to store in your HashMap. A map can be considered to be a set of "associations," where each association is a key/value pair. A map is defined by "get" and "put" operations.

ArrayList (in the java.util package) is one of the simplest and most useful container classes. An ArrayList object works much like an array, but unlike an array, it has no fixed size, and expands as needed to store however many objects the program requires.

Project Requirement

- To find whether a specific word exist in the given text file or not using binary search and sequential search.

- Then find out the time taken by each of the algorithm in order to search the input word and perform a detail performance analysis on both the algorithm.

Searching here refers to finding an item in the list that meets some specified criterion. Sorting refers to rearranging all the items in the array into dictionary order.

Project Constraint

- Application can search only through text files and
- Each line of text file should contain only one word.
- Search is not case-sensitive

2. Searching & Sorting

In this section searching and sorting algorithm are described in general with their java implementation.

Searching algorithms are used to find specific data in given data structure. In this project, Sequential Search and Binary Search have been used for finding out the specific word from the list. Sorting algorithm puts element of a list in a certain order. We have used Merge sort in this project as binary search performs search on sorted data only.

The complexity of an algorithm is the amount of a resource, such as time, that the algorithm requires. It is a measure of how 'good' the algorithm is at solving the problem. The complexity of a problem is defined as the best algorithm that solves a problem.

2.1 Sequential Search

Sequential search, also known as linear search, is the simplest of all searching algorithms. It is a brute-force approach to locating a given element in a list. It finds the element by starting at the first element of the list and examining each subsequent element until the matching element is found or the list exhausts.

Sequential search is proved very useful in the context when you need to search an element in a list quite frequently, and the list may or may not be ordered. In that situation sequential search gets the job done in a brute-force manner.

Sequential search might be the most effective search method, depending upon n, the number of elements in the list, and the number of times you will perform such a search. If n is relatively small or you won't be performing the search over the list often, the cost of sorting the elements or using a complex data structure might outweigh the resulting benefits.

2.1.1 Algorithm for Sequential Search

Sequential search algorithm performs basic search through the entire collection. It starts with first element and move through collection looking for a match. In best-case scenario sequential search can find element on first position and in worst-case scenario it need to iterate through entire collection either because either the item is last one or because the item in not found.

```
for (each item in list) {
        compare search term to current item
        if match,
        save index of matching item
```

```
        break
        }
```

2.1.2 Java Implementation of Sequential Search

```
public int sequentialSearch(int item, int[] list) {
// if index is still -1 at the end of this method, the item is not
// in this array.
        int index = -1;
// loop through each element in the array. if we find our search
// term, exit the loop.
        for (int i=0; i<list.length; i++) {
            if (list[i] == item) {
                index = i;
                break;
            }
        }
        return index;
    }
```

2.1.3 Time Complexity of Sequential Search

- Linear search looks at every element until it finds a match
- The Best-case scenario would be when it finds the element on the first position.
- The worst-case would be
  - The element is not found or
  - It is the last element in the list
  - If there are n items in the list, then the loop will iterate at most n times.
- Therefore, time complexity of **O(n)**

2.2 Binary Search

Binary search delivers better performance than sequential search by sorting the elements of the list ahead of searching.

Binary search looks at the middle element of the list and see if the element sought-for is found. If not, than it compares the sought-for element with the middle element of the list and check whether the middle element is larger than the element we are trying to find. If so, we keep looking in the first half of the list; otherwise, we look in the second half. It cuts the number of elements to be compared by half.

We keep going in the same way until the item is found or it is determined that the sought-for element is not present in the list.

The input to binary search is an indexed collection A of elements. Each element A[i]has a key value ki that can be used to identify the element. These keys are totally ordered, which means that given two keys, ki and kj, either ki<kj, ki==kj, or ki>kj.

2.2.1 Algorithm for Binary Search

Binary search relies on a divide and conquer strategy to find a value within an already-sorted collection. It divides the array into the two parts first and then comparison determines whether the element equals the input, less than the input or greater. When the element being compared to equals the input the search stops and typically returns the position of the element. If the element is not equal to the input then a comparison is made to determine whether the input is less than or greater than the element. Depending on which it is the algorithm then starts over but only searching the top or bottom subset of the array's elements.

```
set first = 1, last = N, mid = N/2
while (item not found and first < last) {
        compare search term to item at mid
        if match
        save index
        break
        else if search term is less than item at mid,
        set last = mid-1
        else
        set first = mid+1
        set mid = (first+last)/2
        }
        return index of matching item, or -1 if not found
```

2.2.2 Java Implementation of Binary Search

```
    public int binarySearch(int item, int[] list) {
// if index = -1 when the method is finished, we did not find the
// search term in the array
        int index = -1;
// set the starting and ending indexes of the array; these will
// change as we narrow our search
        int low = 0;
        int high = list.length-1;
        int mid;
```

```
// Continue to search for the search term until we find it or
// until our ''low'' and ''high'' markers cross
        while (high >= low) {
            mid = (high + low)/2;// calculate the midpoint of the
current array
            if (item < list[mid]) { // value is in lower half, if at
all
                high = mid - 1;
            } else if (item > list[mid]) {
// value is in upper half, if at all
                low = mid + 1;
            } else {
// found it! break out of the loop
                index = mid;
                break;
            }
        }
        return index;
    }
```

2.2.3 Time Complexity of Binary Search

- Every recursive call to the binary search algorithm cuts the input size in half which results in logarithmic performance.
- In the best case, the item would be found is the middle in the array
- In the worst case, the item does not exist in the array at all
- Time complexity of Binary Search is **O(log n).**

2.3 Merge Sort

Merge sort is a neat algorithm, because it's "the sort that sorts itself". This means that merge sort requires very few comparisons and swaps; it instead relies on a "divide and conquer" strategy that's slightly different from the one that quicksort uses.

Merge sort starts by dividing the list to be sorted in half. Then, it divides each of these halves in half. The algorithm repeats until all of these "sublists" have exactly one element in them. At that point, each sublist is sorted. In the next phase of the algorithm, the sublists are gradually merged back together (hence the name), until we get our original list back — sorted.

3. Technologies & Tools

This section describes the underlying tools and technologies, which are being used to implement the project.

3.1 Programming Language

The application is developed using the high-level and object-oriented programming language **Java** using **Spring MVC framework**. **HTML** is used as a front-end language to develop a simple User Interface for user to view result or output. More information on java can be found on http://docs.oracle.com/javase/tutorial/ and for HTML http://www.w3schools.com/tags/

3.2 Spring MVC Framework

This project uses Spring Framework, which is a lightweight alternative to Java's EE complexity. One of the key benefits that it pioneered is dependency injection (DI). DI is a design pattern that separates an application's dependencies (and dependency configuration) from the code that uses those dependencies. Spring implements DI by allowing to "wire" together an application from beans, using either an XML configuration file or annotations. The main goal of an application is it should focus on solving its business problems rather than on the overhead required to construct resources and objects. Spring creates your objects for you and makes them available at runtime.

Spring MVC is a full MVC implementation that follows the patterns and paradigms that Spring is known for, including DI. Spring provides a front controller servlet named DispatcherServlet. Spring provides various controllers for to use as base classes for creating controllers, depending on project needs. More information can be found on http://spring.io/docs.

3.3 Apache Tomcat

Apache tomcat is an open source web container used in the application to run the server locally on the PC and servlets and Java serve pages(JSP). This means application is deployed using Apache tomcat for implementation. Apache Tomcat provides by default a HTTP connector on port 8080 i.e. Tomcat can also be used as HTTP server.More information on Apache Tomcat can be found on http://tomcat.apache.org/tomcat-7.0-doc/

3.4 Maven

Maven is used in the project as a software project management and comprehension tool. Maven manages a project's build, reporting and documentation from a central piece of information. Maven allows creating or importing Maven modules, downloading artifacts and performing the goals of the build lifecycle and plugins. More information on Maven can be found on http://maven.apache.org/guides/

3.5 IntelliJ IDEA

IntelliJ Idea is a Java integrated development environment (IDE) used in this project for developing web application. IntelliJ IDEA offers outstanding framework-specific coding assistance and productivity-boosting features for Spring along with deployment tools for most application servers. It provides advanced coding assistance for JavaScript, HTML and CSS technologies. IntellijIDEA can be more explored on this site https://www.jetbrains.com/idea/documentation/

3.6 JUnit

JUnit is a testing framework available for Java which uses annotations to identify method that specify a test. JUnit uses annotations to mark methods and to configure the test run. More information on JUnit can be found on http://junit.sourceforge.net/javadoc/

4. Search BenchMarking Web Application

Search benchmarking application is a collection of dynamic resources (such as Servlets, JavaServer Pages, Java classes and jars) and static resources (HTML pages and pictures). This web application can be deployed as a *WAR* (Web Archive) file.

This application is running inside a web container on the server. The container provides a runtime environment for Java web applications. The container is for Java web applications what the JVM (Java Virtual Machine) is for local running Java applications. The container itself runs in the JVM. Web containers are Tomcat or Jetty. A web container supports the execution of Java servlets and JavaServer Pages. Servlets are the Java programming language classes that dynamically process requests and construct responses, usually for HTML pages. Java Server Pages (JSP) are Text-based documents that are compiled into servlets and define how dynamic content can be added to static pages, such as HTML pages.

Search Benchmarking web application works by using web pages to display the content and a servlet to process the requests from the client that is when a user makes a query from the web page the servlet will process the query and return the result to the client. On the client side requests to the servlet is done by filling an input field and submitting the form by clicking the button.

4.1 Web MVC Framework

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

4.2 Web Application Architecture

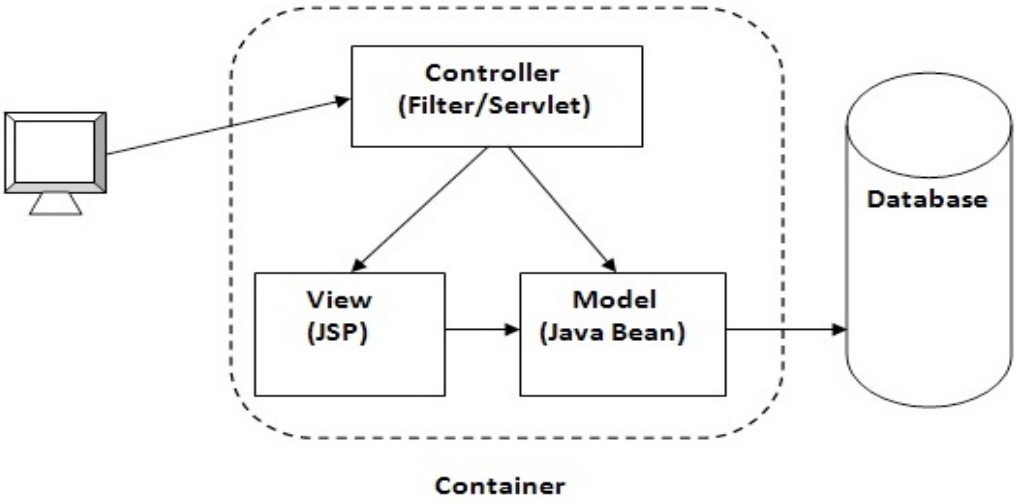

Figure 1 MVC Architecture

The Model 2 architecture is based on the MVC (Model View Controller) design pattern, shown in figure above, is a hybrid approach for serving dynamic content, since it combines the use of both servlets and JSP. It takes advantage of the predominant strengths of both technologies, using JSP to generate the presentation layer and servlets to perform process-intensive tasks. Here, the servlet acts as the *controller* and is in charge of the request processing and the creation of any beans or objects used by the JSP, as well as deciding, depending on the user's actions, which JSP page to forward the request to. There is no processing logic within the JSP page itself; it is simply responsible for retrieving any objects or beans that may have been previously created by the servlet, and extracting the dynamic content from that servlet for insertion within static templates.

The model is typically a loose JavaBean, or Plain Old Java Object (POJO) that has getters and setters to manipulate its contents. It can be a very simple object or a very complicated one with dozens of sub-objects, but it's just an object nonetheless.

The view is usually a JSP page. After the business logic is complete and the model has been generated, the controller passes the model to the view to be presented to the user.

The controller is typically implemented by a servlet that calls into business services to perform some action. The controller usually hosts additional logic such as authentication and authorization: it *controls* which actions a user can execute.

In Spring MVC architecture, controllers make calls into spring service beans to perform business actions and then send a model object to one of your views for presentation.

5. Project Set Up

This section describes the step-by-step procedure to create project, maven integration, node js integration with Spring framework and JUnit for testing.

5.1 Steps to Create Project in intelliJIDEA

1. Download and install **Java, IntellijIDEA** from the website: https://java.com/en/download/ https://www.jetbrains.com/idea/download/

Set the JAVA_HOME Variable

- Once you have identified the JRE installation path:
- Right-click the My Computer icon on your desktop and select Properties.
- Click the Advanced tab.
- Click the Environment Variables button.
- Under System Variables, click New.
- Enter the variable name as JAVA_HOME.
- Enter the variable value as the installation path for the Java Development Kit.
- Click OK. Click Apply Changes.

2. After downloading we have to create Maven Project. **Steps to create Maven module** in IntelliJIDEA

- Open the project you want to add a module to, and select File | New Module.
- As a result, the New Module wizard opens. (There are also other ways of accessing this wizard, see Opening the New Module wizard.)
- On the first page of the wizard, in the left-hand pane, select Maven.
- In the right-hand part of the page:

  -If the Module SDK field is present, specify the SDK (JDK) to be used.

  -If the necessary SDK is already defined in IntelliJ IDEA, select it from the list. Otherwise, click New and select the installation folder of the desired JDK.
- If you want to create a Maven module based on one of the Maven Archetypes, select the Create from archetype check box, and select the desired archetype from the list.

  Click Next.

- On the Maven page of the wizard, specify the aggregator and parent Maven projects, if any, and Maven project coordinates. Note that if a parent Maven project is specified, the new Maven project can inherit its GroupId and ArtifactId. Click Next.
- Specify the name and location settings for your module. Click Finish.

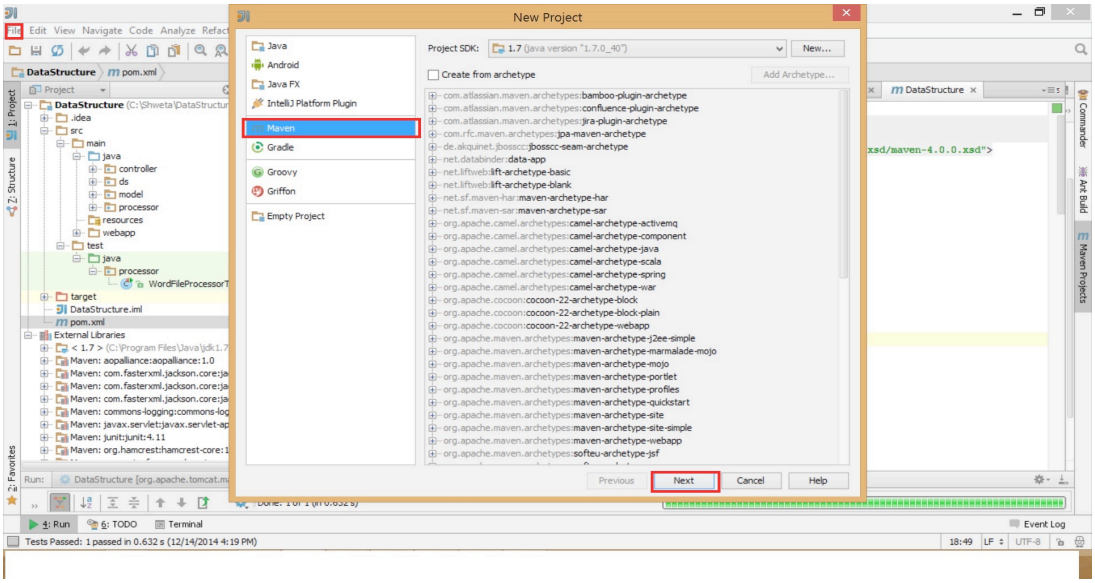

Figure 2 Create Maven Project

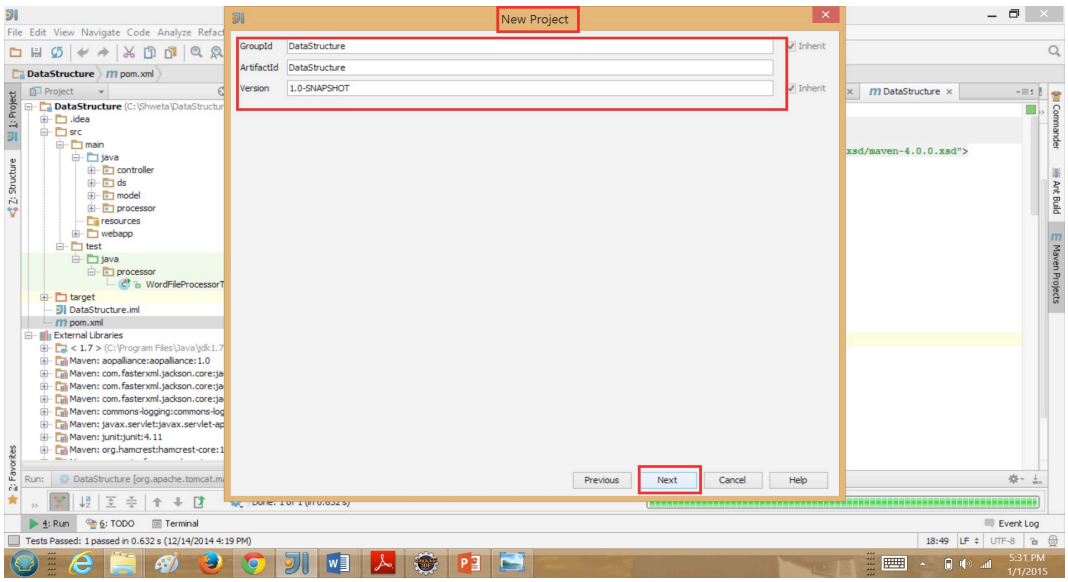

Figure 3 GAVC Values

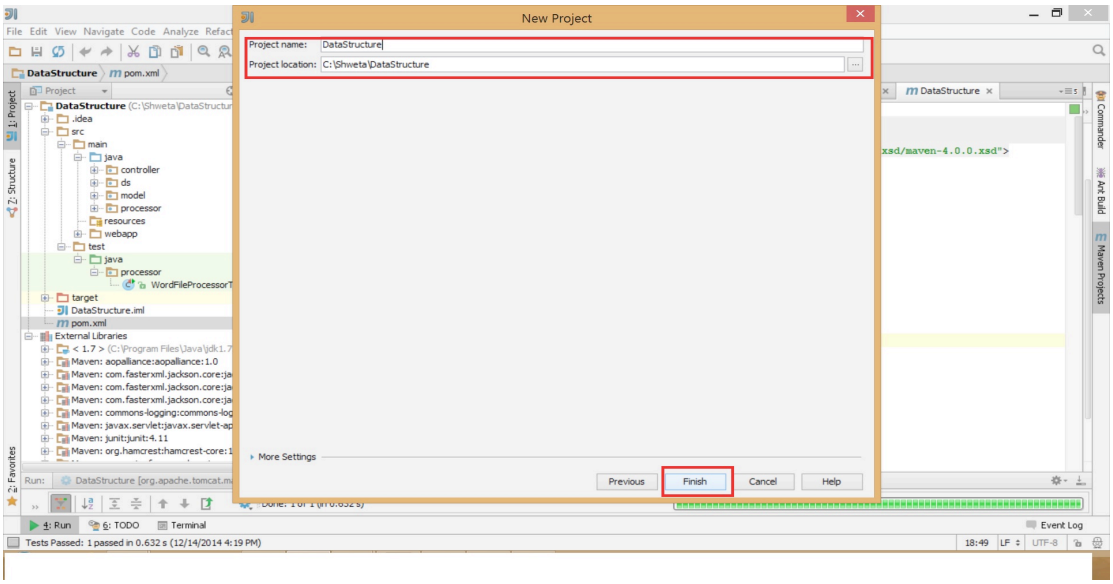

Figure 4 Maven Project Name

3. Getting Started with Maven:

- Maven projects assume a standard directory structure, which looks - at a minimum - like this

```
./pom.xml
./src
./src/main
./src/main/java
./src/main/webapp
./src/test
./src/test/java
./src/test/respurces
```

- At the root of the directory structure is a XML file (always called pom.xml) that Maven expects. The pom.xml (POM is short for Project Object Model) describes the things specific to your project that can't be inferred automatically like dependencies, the name of the project, etc.

DIRECTORY	**DESCRIPTION DIRECTORY'S CONTENTS (RELATIVE TO THE PROJECT ROOT)**
src/main/java	Contains the Java source code for project
src/main/resources	Contains any classpath-relative resources for project (like, a Spring application context .xml file)
src/test/java	Contains the java source code for test classes. This directory will not be included in the final build. All tests herein will be compiled

	and all tests will be run. If the tests fail, it aborts the build.
src/main/webapp	This directory contains standard web application structure.

- As we select the maven project, after set up we get a default pom.xml file. Add the packaging type as war.

-
```
<groupId>DataStructure</groupId>
<artifactId>DataStructure</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
```

d

ependencies in pom.xml

As we know Maven uses pom.xml to manage project dependencies, dependencies are added with their groupId, artifactId, and version. In Maven, the web project settings are configure via this single pom.xml file.

i. Add project dependencies – Spring, JUnit, json processing

ii. Add plugins to configure this project – Maven Tomcat, JDK Complier level

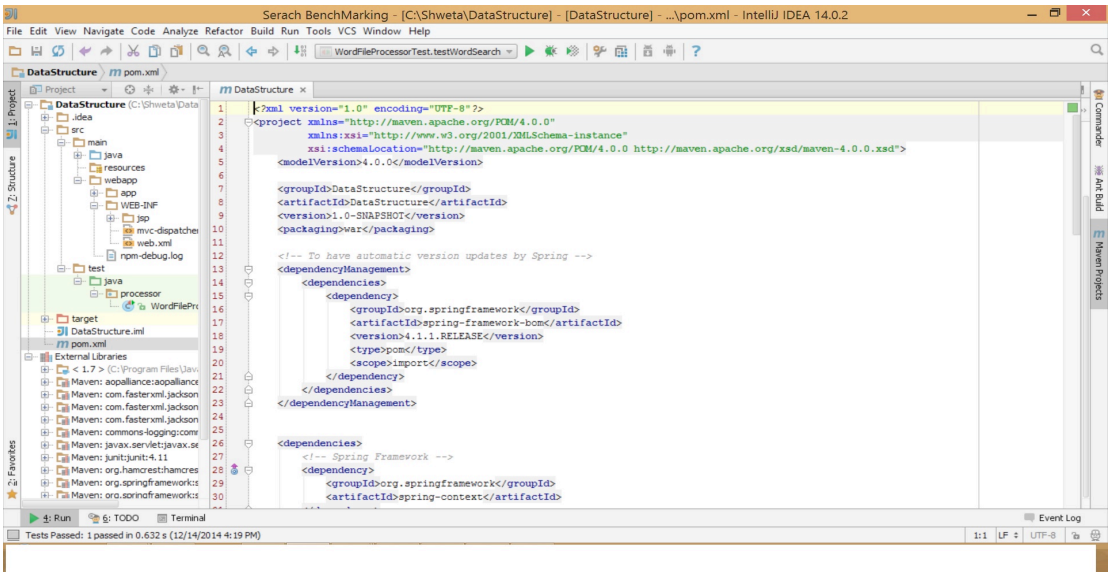

Figure 5 POM.xml

- Create a folder named “webapp” inside src/main/webapp. Create another folder inside webapp called WEB-INF. The WEB-INF directory contains the deployment descriptors for the Web application (web.xml and mvc-dispatcher-servlet.xml).

  The primary entry point for a Spring application is the DispatcherServlet, so this first step is to create a DispatcherServlet in the web.xml file. Next, that DispatcherServlet needs to be mapped to a URI pattern. You can choose any pattern that you want, but most Spring applications map requests to pages ending in .html. In this case, all requests that end in .html will be sent to the mvc-dispatcher Servlet. By default, Spring looks for Spring beans in a file whose name starts with the servlet name followed by -servlet.xml. So we created a mvc-dispatcher-servlet.xml file that contains, our URL mapping strategy (to map URLs to controllers), and our view resolver, and place it in the WEB-INF directory.

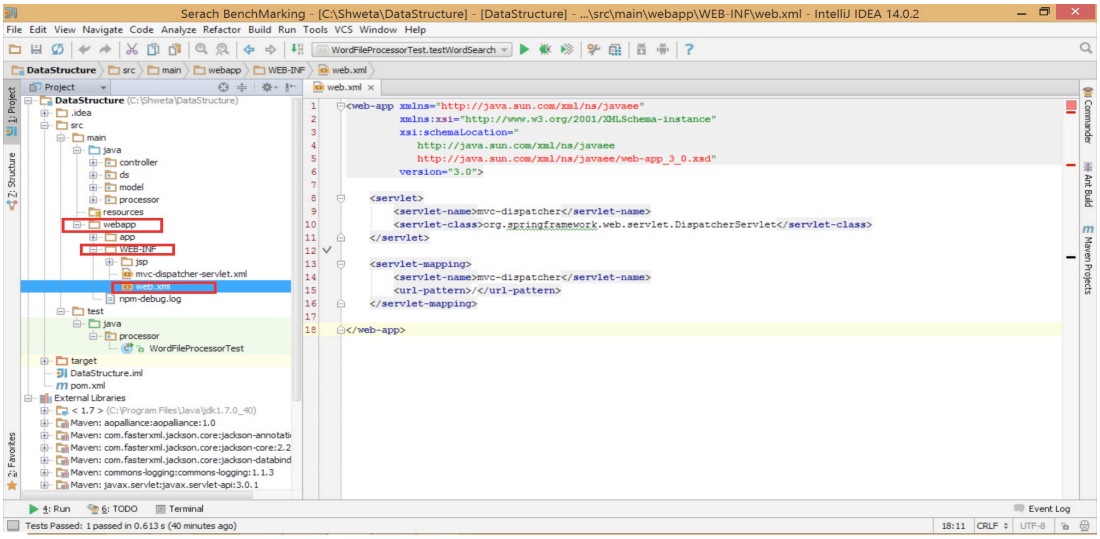

Figure 6 Web.xml

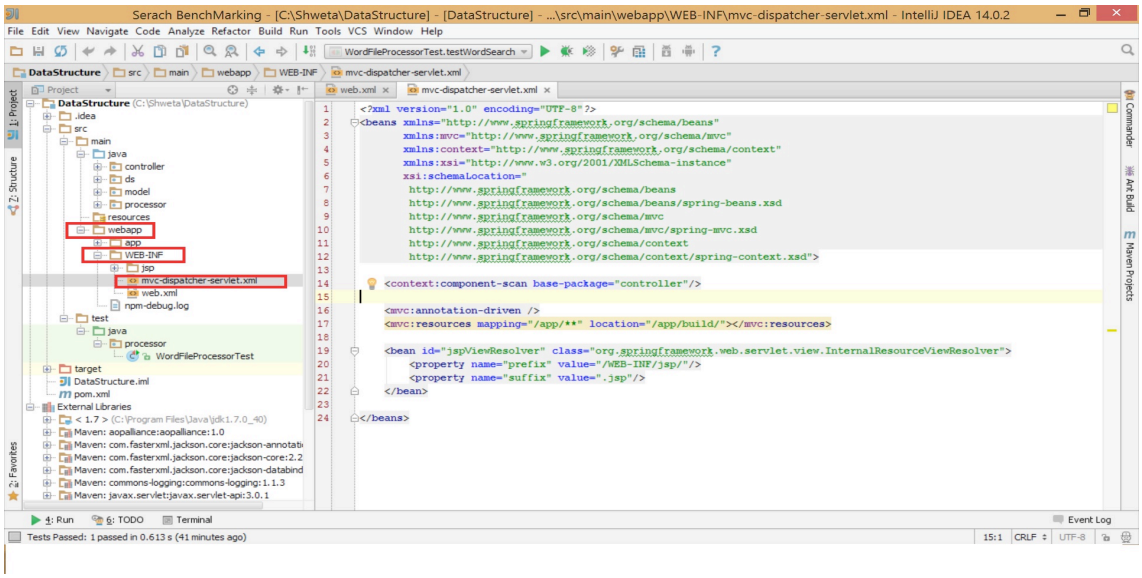

Figure 7 MVC-dispatcher-servlet.xml

- Create a folder named "controller" in src/main/java. Inside that create a class called WordSearchController. In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method. @RequestMapping indicates that all handling methods on this controller are relative to the / path. Attached is the snapshot below:

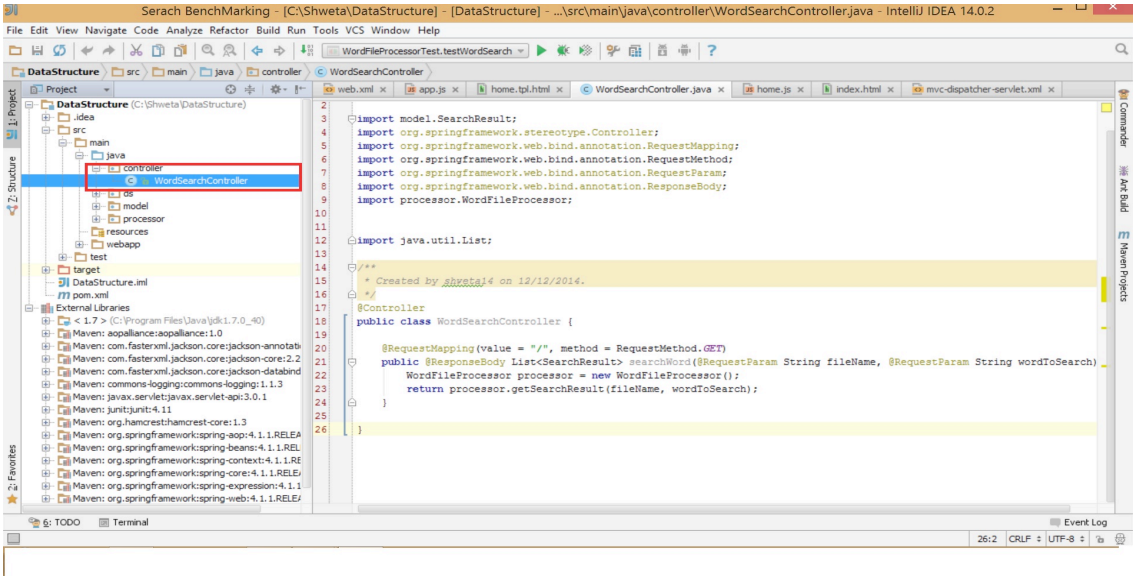

Figure 8 @Controller Class

- Create a folder named “ds” in src/main/java. Make 3 different classes StringBinarySearch, StrinhLinearSearch and StringMergeSort which include code that defines the searching and sorting of the strings present in the list.

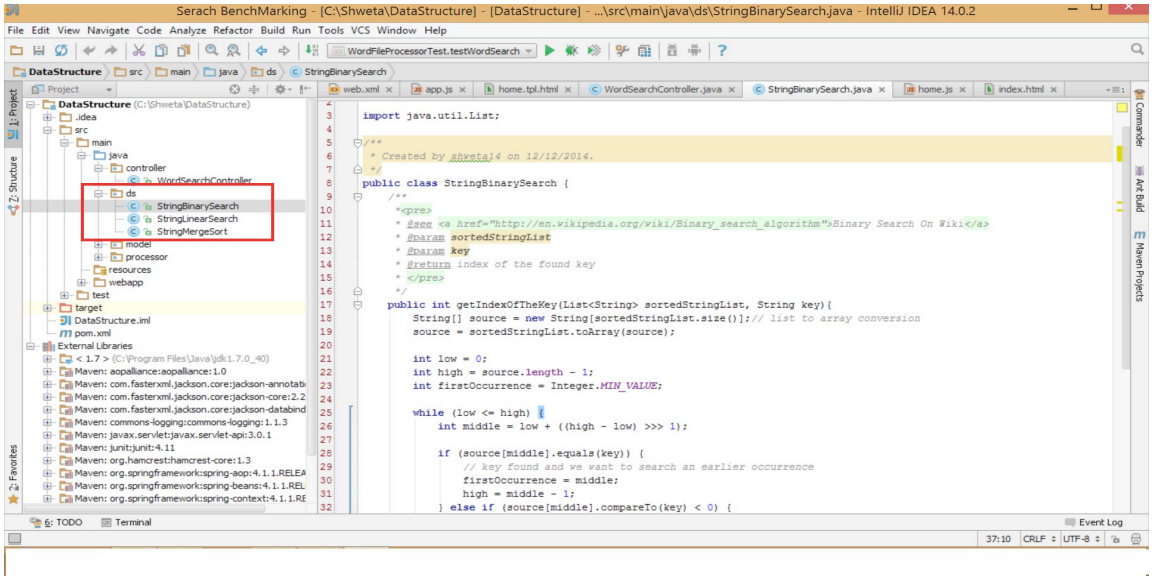

Figure 9 Search and Sort classes

- Create another folder named “model” in src/main/java. Make a class directory names SearchResult which will hold the result after searching and sorting is done. The model (the M in MVC) is a Map interface, which allows for the complete abstraction of the view technology.

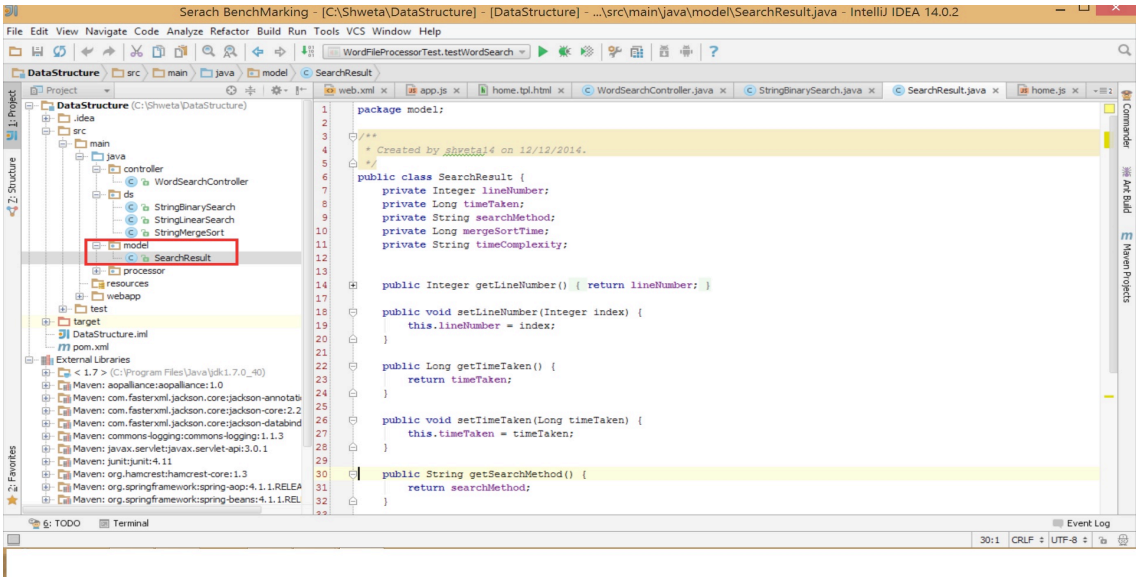

Figure 10 Model class

- Create a folder called “processor” inside src/main/java. Create a java class called WordFileProcessor. This is the core class which performs the main functionality for reading the input text file and return if the word exist or not.

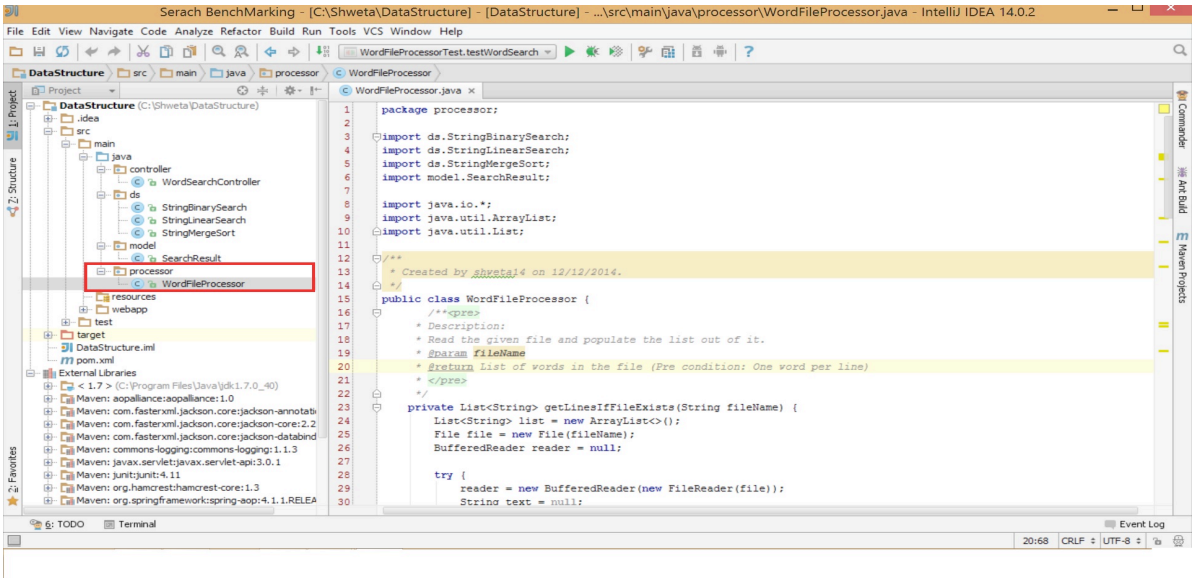

Figure 11 Processor Class

- Create a folder named “test” inside src/main. In this we have created a test class called WordFileProcessorTest for testing the code and functionality. This class is created inside src.main/test/java/processor.

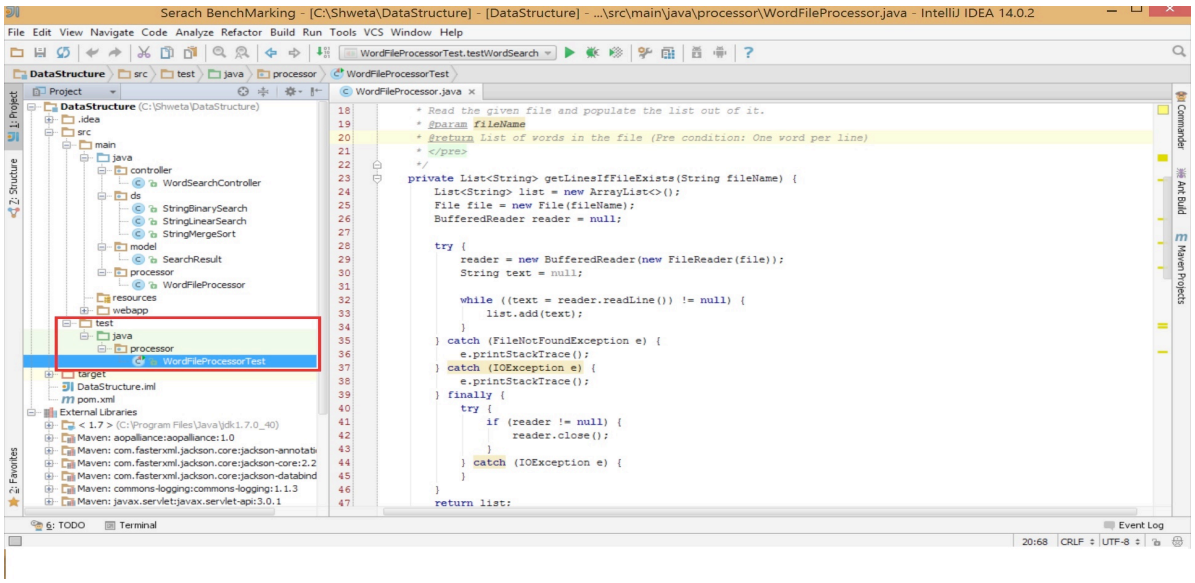

Figure 12 Test Class

4. Getting started with User Interface (Front End View)

- Installing Node and npm: The versions of Node and npm are downloaded from http://nodejs.org/dist, extracted and put into anode folder created in working directory. Node/npm will only be "installed" locally to your project. Create a file package.json in src/main/webapp/app. The package.json has pretty much the same responsibility for npm as the pom has for maven.

- After installation of Node. Perform following steps on Command Line:

  $ git clone git://github.com/joshdmiller/ng-boilerplate
  $cd ng-boilerplate
  $ sudo npn –g install grunt-cli karma bower
  $npm install
  $grunt watch

- ngBoilerplate is designed to make life easy by providing a basic framework with which to kickstart AngularJS projects. It contains a best-practice directory structure to ensure code reusability and maximum scalability. It contains a sophisticated Grunt-based build system to ensure maximum productivity. All you have to do is clone it and start coding.

- Directory Structure:

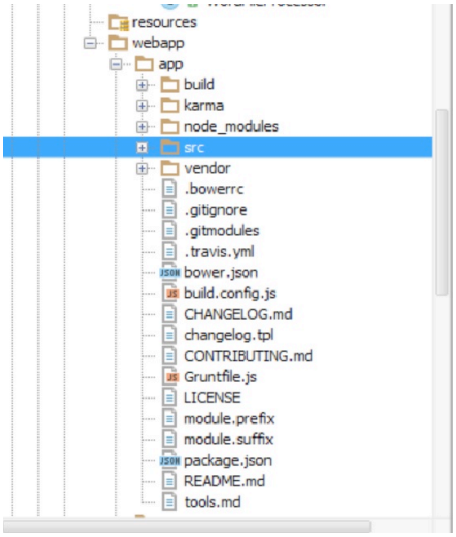

Figure 13 1 ngBoilerPlate Directory Structure

  - karma/ - test configuration.

- src/ - our application sources. Read more »
- vendor/ - third-party libraries. Bower will install packages here. Anything added to this directory will need to be manually added to build.config.js and karma/karma-unit.js to be picked up by the build system.
- .bowerrc - the Bower configuration file. This tells Bower to install components into the vendor/directory.
- bower.json - this is our project configuration for Bower and it contains the list of Bower dependencies we need.
- build.config.js - our customizable build settings.
- Gruntfile.js - our build script.
- module.prefix and module.suffix - our compiled application script is wrapped in these, which by default are used to place the application inside a self-executing anonymous function to ensure no clashes with other libraries.
- package.json - metadata about the app, used by NPM and our build script. Our NPM dependencies are listed here.

- ngBoilerplate uses Grunt as its build system, so Node.js is required. Also, Grunt by default no longer comes with a command-line utility and Karma and Bower must end up in your global path for the build system to find it, so they must be installed independently. Once you have Node.js installed, you can simply use npm to make it all happen:

  $ npm -g install grunt-cli karma bower
  $ git clone git://github.com/joshdmiller/ng-boilerplate my-project-name
  $ cd my Project name
  4.5 And then install the remaining build dependencies locally
  $ npm install
  $ bower install

  Technically, ngBoilerplate is now ready to go.

- To ensure your setup works, launch grunt:
  $ grunt watch

- Create a template html file in webapp/app/src/app/home. Added the HTML code for the page as to be shown on UI for user.

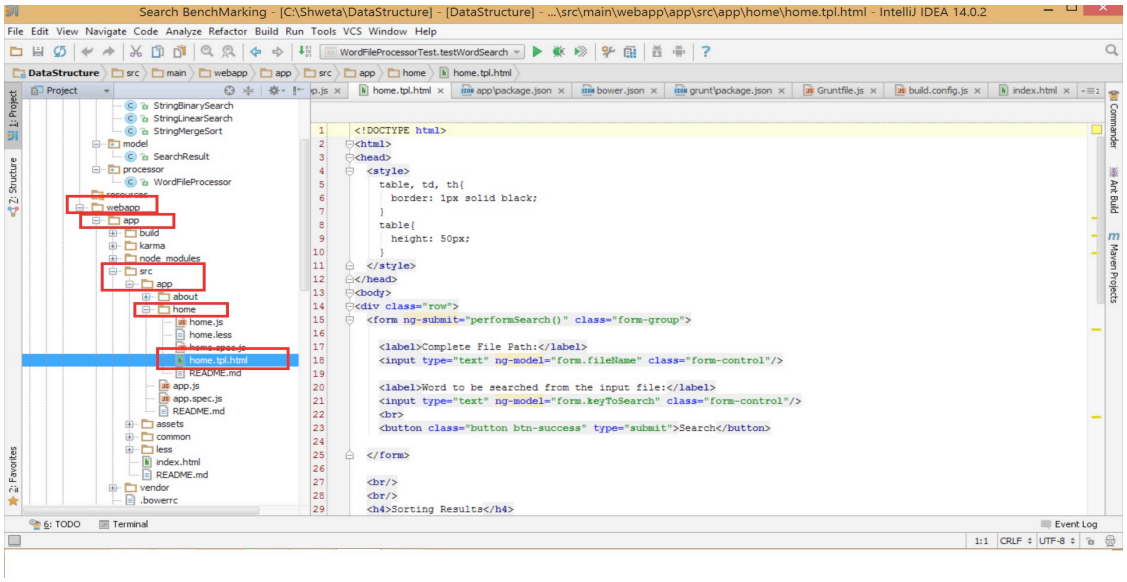

Figure 14 home.html

- One of the sub-module provided by ng-boiler plate is app.js. This is our main app configuration file. It kickstarts the whole process by requiring all the modules from src/app that we need. We must load these now to ensure the routes are loaded. There exist a sub-module called home.js in webapp/app/src/app/home. It contains the code for calling the controller from the UI page for handling searching and sorting. Our home module is where we want to start, which has a defined route for/home in src/app/home/home.js.

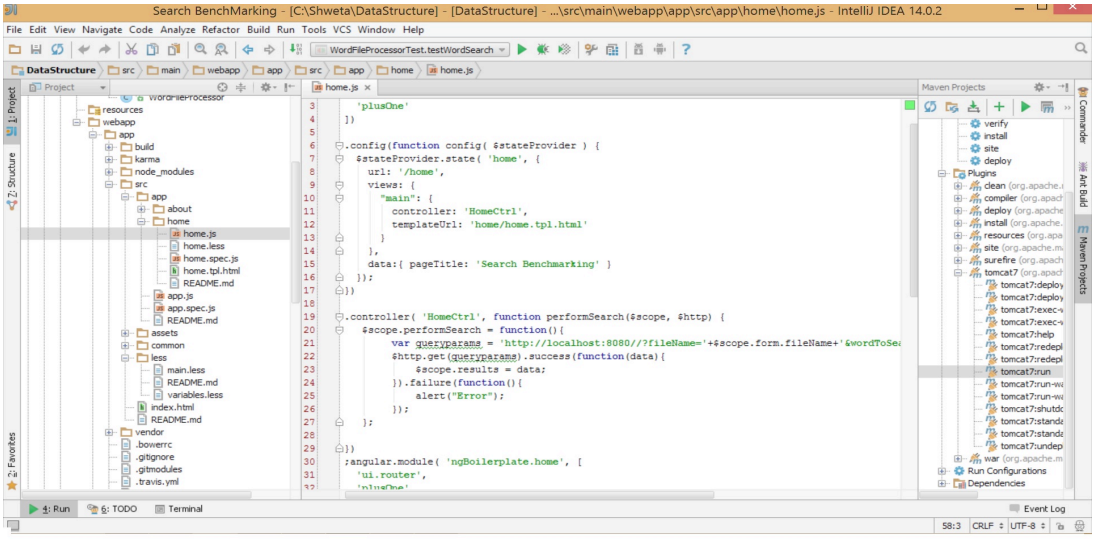

Figure 15 home.js

6. Search BenchMarking Application's Graphical User Interface

The Graphical user interface (GUI) is everything designed into an information device with which a human being may interact.

6.1 Components of the Webpage

1. URL: This specifies the URL of the web page from which the browser sends a HTTP request to the server and then the server then sends back HTML response.

2. File Path: User has to specify the path of the file where the input file is present in the computer.

3. Word to be search: User has to specify the word which he/she wants to find whether that specific word exist in the file or not. Once done user clicks on Search button.

4. Sorting Results: After the form is submitted. Java Application will search for the word and it will display the result in terms of time taken by merge sort, binary search and sequential search along with the line number on which word exit and its Complexity.

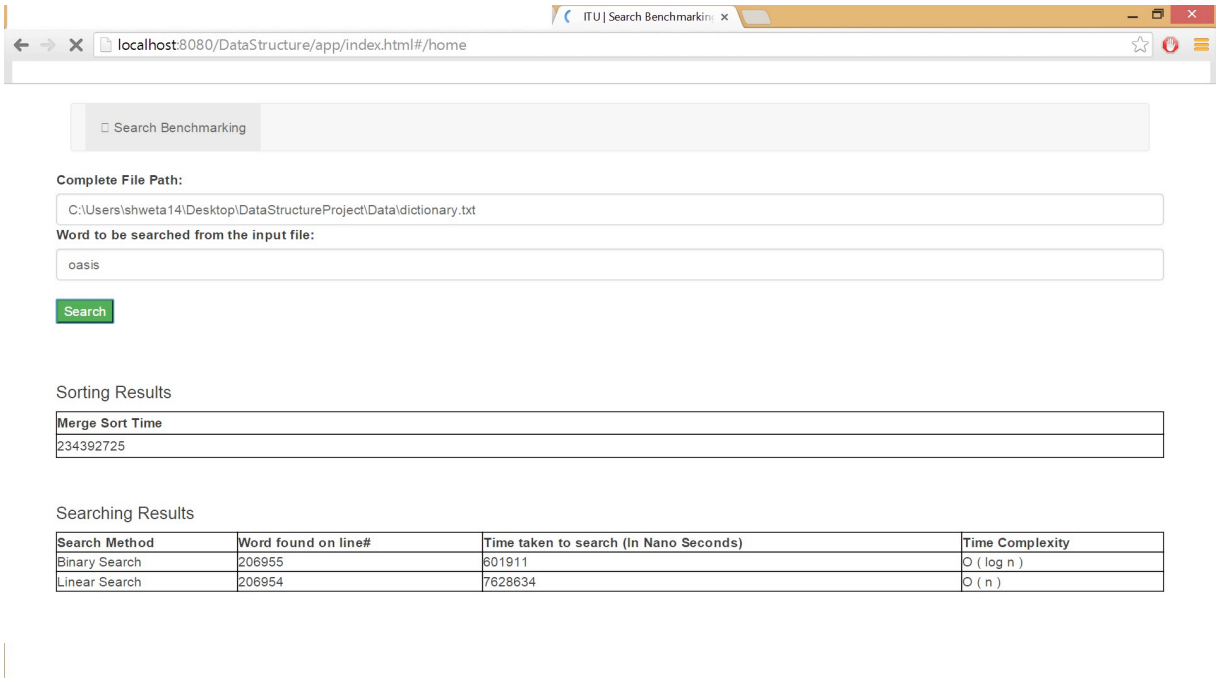

Figure 16 Search BenchMarking GUI

7. Performance Analysis

The complexity of an algorithm is the amount of a resource, such as time, that the algorithm requires. It is a measure of how 'good' the algorithm is at solving the problem. The complexity of a problem is defined as the best algorithm that solves a problem.

Sequential Search searches every element in a list one at a time and in sequence starting from the first element. In complexity term it is O(n), where n is the number of elements in the list. It is useful and fast when we have small number of elements in the list. Since, the amount of effort needed is in proportion with the count of the list being searched, it takes more time to perform sequential search when the count is large.

Binary Search on the other hand starts from the middle of a sorted list and decides on which half of the list to proceed in order to find the element to be searched. By doing it this way it eliminates half of the list on every routine until it finds an exact match. In complexity term it takes O(log n)time to complete.

7.1 Sequential Search Vs Binary Search Performance Analysis w.r.t time

In this project I tried several different files which contain different size of data to analyze the time taken by both search algorithms. Below are the different snapshot which demonstrate real time analysis.

Scenario 1: Text File with 60 words

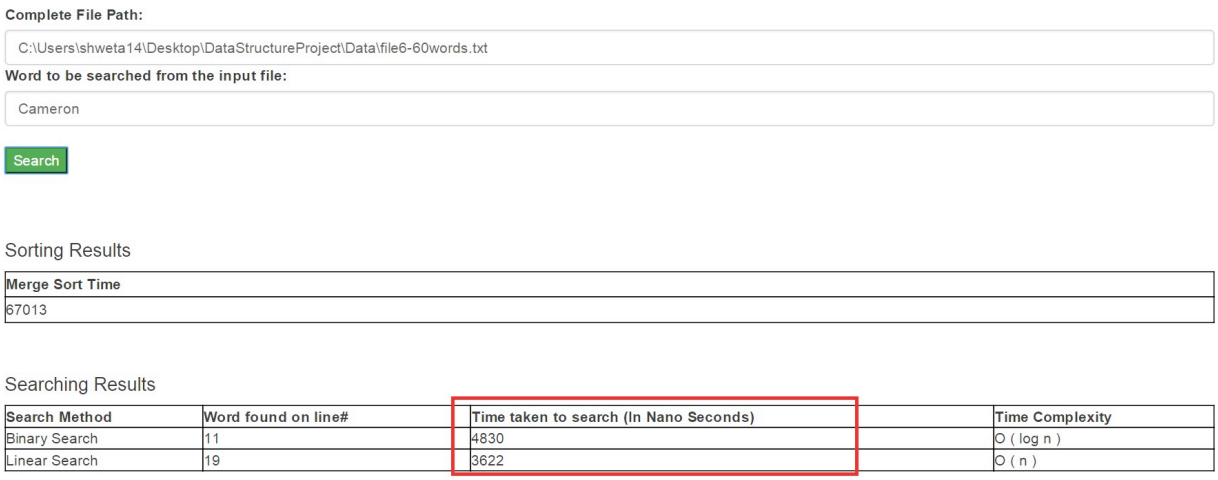

Figure 17 Performance Analysis 1

Scenario 2: Text file with 100 words.

Complete File Path:

C:\Users\shweta14\Desktop\DataStructureProject\Data\file10-100words.txt

Word to be searched from the input file:

Jacob

Search

Sorting Results

Merge Sort Time
572327

Searching Results

Search Method	Word found on line#	Time taken to search (In Nano Seconds)	Time Complexity
Binary Search	52	35015	O (log n)
Linear Search	4	18716	O (n)

Figure 18 Performance Analysis 2

Scenario 3: Text file with 345000 words.

Complete File Path:

C:\Users\shweta14\Desktop\DataStructureProject\Data\dictionary.txt

Word to be searched from the input file:

rath

Search

Sorting Results

Merge Sort Time
108831397

Searching Results

Search Method	Word found on line#	Time taken to search (In Nano Seconds)	Time Complexity
Binary Search	245798	602513	O (log n)
Linear Search	245797	3924180	O (n)

Figure 19 Performance Analysis 3

Scenario 4: Text file with 680000 words.

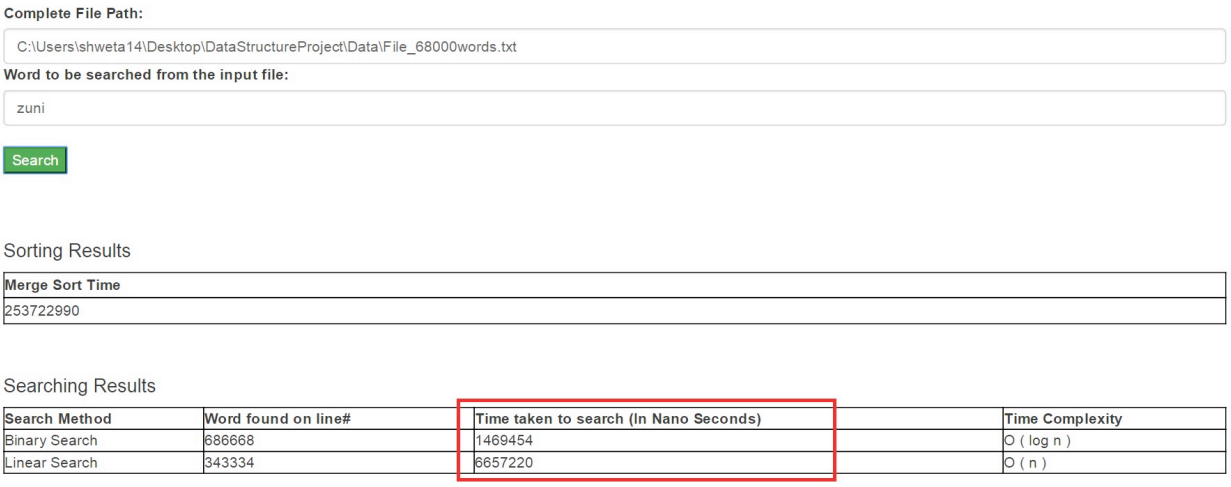

Figure 20 Performance Analysis 4

Above scenarios and their result within application describes that for small data file, sequential search performs faster as compared to binary search. On the other side when file size is huge or contains more data than binary search performs efficiently and faster as compared to sequential search. Below table demonstrates real time statistics:

S.No.	Word Count (in txt file)	Time Taken (Nano Second) by Sequential Search	Time Taken (Nano Second) by Binary Search
1	60 words	3622	4830
2	100 words	18716	35015
3	345000 words	3924180	602513
4	680000 words	6657220	1469454

7.2 Graphical Analysis

- **Sequential Search Time Complexity Graph:** Given a list of length n for sequential search algorithm we find that the number of comparison is n.

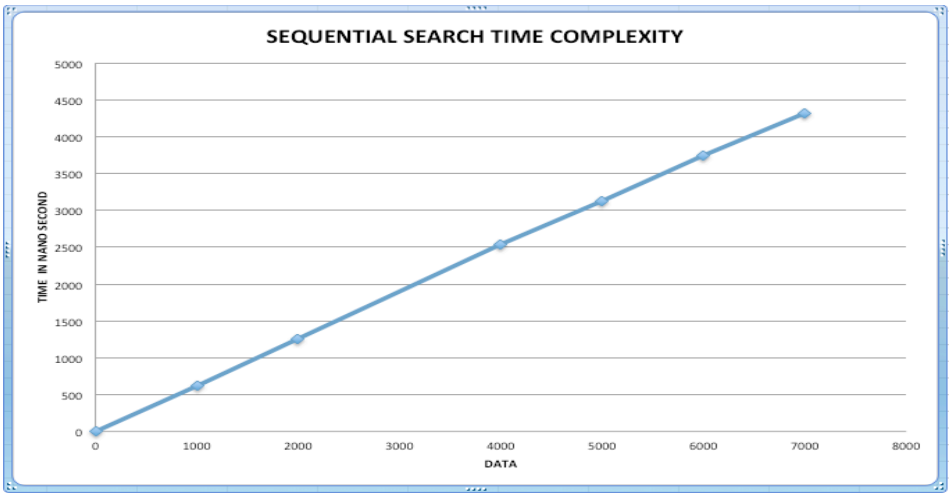

Figure 21 Sequential Search Time Complexity

- **Binary Search Time Complexity Graph:** Given a list of length n for binary search algorithm we find that the number of comparison is log n.

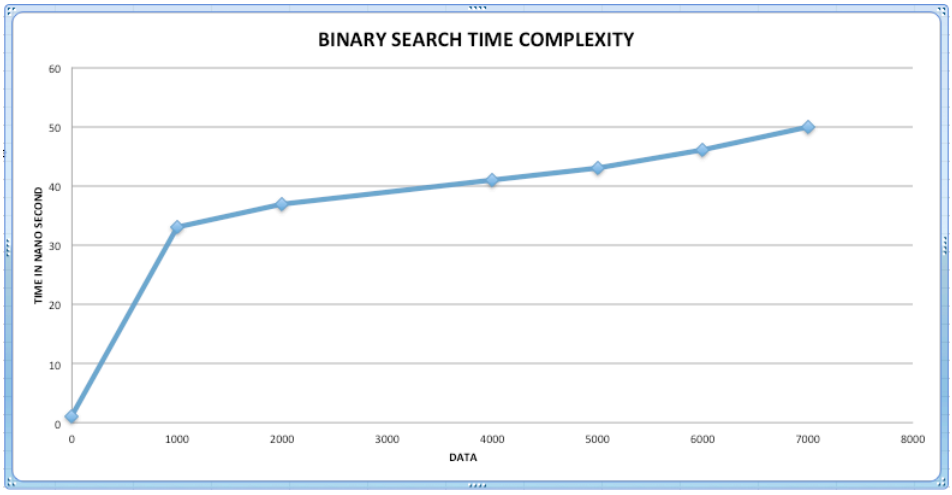

Figure 22 2 Binary Search Time Complexity

8. Conclusion and Future Improvements

The overall goal of the project was to develop an application which searches through a text file using binary and sequential search and also provide enough data to compare time complexity of both the search algorithm. Also the goal was to understand the functionality of major Data Structure and their practical implementation. In addition to this, test cases were written in JUnit to make sure that code is bug free and code execution is smooth.

Sequential search is good for list with a smaller number of elements and binary search is good for lis with a larger number of elements. When choosing between binary and sequential search, one must take into consideration the requirement that Binary search needs a sorted list as input.

Future Improvement

There is still several improvements that could be implemented in the project, following is brief description of extensions that could improve it.

- Case Sensitive: Application could search the word ignoring the case.
- Real-Time Graph: Integration of real time graph generation could be in done in future with respect to time and data size.
- Regex based Search: Application could also support regular expression.

9. References

[1] Web MVC Framework

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[2] Create Maven Application

http://www.mkyong.com/maven/how-to-create-a-web-application-project-with-maven/

[3] Map Data Structure

http://math.hws.edu/javanotes/c10/s3.html

[4] ngBoilerplate, node js set up

https://github.com/ngbp/ngbp

[5] Spring MVC

http://www.javaworld.com/article/2078034/spring-framework/mastering-spring-mvc.html

[6] Implement Binary and Sequential Search

http://cs-fundamentals.com/data-structures/sequential-and-binary-search-in-java.php

[7] Spring Framework

http://docs.spring.io/docs/Spring-MVC-step-by-step/part1.html#step1.2

[8] Spring Tutorial

http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

[9] Binary Search Algorithm

http://javatechig.com/c/binary-search-algorithm-data-structure

10. Appendix Source Code (in Java)

10.1 String Binary Search

```
import java.util.List;

/**
 * Created by shweta14 on 12/12/2014.
 */
public class StringBinarySearch {
    /**
     *<pre>
     * @see <a
href="http://en.wikipedia.org/wiki/Binary_search_algorithm">Binary
Search On Wiki</a>
     * @param sortedStringList
     * @param key
     * @return index of the found key
     * </pre>
     */
    public int getIndexOfTheKey(List<String> sortedStringList, String
key){
        String[] source = new String[sortedStringList.size()];// list
to array conversion
        source = sortedStringList.toArray(source);

        int low = 0;
        int high = source.length - 1;
        int firstOccurrence = Integer.MIN_VALUE;

        while (low <= high) {
            int middle = low + ((high - low) >>> 1);

            if (source[middle].equals(key)) {
                // key found and we want to search an earlier
occurrence
                firstOccurrence = middle;
                high = middle - 1;
            } else if (source[middle].compareTo(key) < 0) {
                low = middle + 1;
            } else {
                high = middle - 1;
            }
        }

        if (firstOccurrence != Integer.MIN_VALUE) {
            return firstOccurrence;
        }
        return -(low + 1);  // key not found
    }
}
```

10.2 String Linear Search

```
package ds;

import java.util.List;

/**
 * Created by shweta14 on 12/12/2014.
 */
public class StringLinearSearch {
    /**
     * <pre>
     * <a href="http://en.wikipedia.org/wiki/Linear_search">Linear
Search On Wiki</a>
     * @param sortedStringList
     * @param key
     * @return return the index of the word if found, -1 otherwise
     * </pre>
     */
    public int getIndexOfTheKey(List<String> sortedStringList, String
key){
        String[] source = new String[sortedStringList.size()];
        source = sortedStringList.toArray(source);

        int index = -1;

        for(int i = 0 ; i < source.length ; i++){
            if(source[i].equals(key)){
                return i;
            }
        }

        return index;
    }
}
```

10.3 String Merge Sort

```
package ds;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * Created by shweta14 on 12/12/2014.
 */
public class StringMergeSort {
     public static List<String> mergeSort(List<String> list) {
        String[] a = new String[list.size()];
        a = list.toArray(a);
        Comparable[] tmp = new Comparable[a.length];
        mergeSort(a, tmp, 0, a.length - 1);

        return Arrays.asList(a);
    }
    private static void mergeSort(Comparable[] a, Comparable[] tmp,
int left, int right) {
        if (left < right) {
            int center = (left + right) / 2;
            mergeSort(a, tmp, left, center);
            mergeSort(a, tmp, center + 1, right);
            merge(a, tmp, left, center + 1, right);
        }
    }
    private static void merge(Comparable[] a, Comparable[] tmp, int
left, int right, int rightEnd) {
        int leftEnd = right - 1;
        int k = left;
        int num = rightEnd - left + 1;

        while (left <= leftEnd && right <= rightEnd)
            if (a[left].compareTo(a[right]) <= 0)
                tmp[k++] = a[left++];
            else
                tmp[k++] = a[right++];

        while (left <= leftEnd)    // Copy rest of first half
            tmp[k++] = a[left++];

        while (right <= rightEnd)  // Copy rest of right half
            tmp[k++] = a[right++];

        // Copy tmp back
        for (int i = 0; i < num; i++, rightEnd--)
            a[rightEnd] = tmp[rightEnd];
    }
}
```

10.4 Word Search Controller

```
package controller;

import model.SearchResult;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import processor.WordFileProcessor;


import java.util.List;

/**
 * Created by shweta14 on 12/12/2014.
 */
@Controller
public class WordSearchController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public @ResponseBody List<SearchResult> searchWord(@RequestParam
String fileName, @RequestParam String wordToSearch){
        WordFileProcessor processor = new WordFileProcessor();
        return processor.getSearchResult(fileName, wordToSearch);
    }

}
```

10.5 Search Result

```
package model;
public class SearchResult {
    private Integer lineNumber;
    private Long timeTaken;
    private String searchMethod;
    private Long mergeSortTime;
    private String timeComplexity;


    public Integer getLineNumber() {
        return lineNumber;
    }

    public void setLineNumber(Integer index) {
        this.lineNumber = index;
    }

    public Long getTimeTaken() {
        return timeTaken;
    }

    public void setTimeTaken(Long timeTaken) {
        this.timeTaken = timeTaken;
    }

    public String getSearchMethod() {
        return searchMethod;
    }

    public void setSearchMethod(String searchMethod) {
        this.searchMethod = searchMethod;
    }

    public Long getMergeSortTime() {
        return mergeSortTime;
    }

    public void setMergeSortTime(Long mergeSortTime) {
        this.mergeSortTime = mergeSortTime;
    }

    public String getTimeComplexity() {
        return timeComplexity;
    }

    public void setTimeComplexity(String timeComplexity) {
        this.timeComplexity = timeComplexity;
    }
}
```

10.6 Word File Processor

```
package processor;

import ds.StringBinarySearch;
import ds.StringLinearSearch;
import ds.StringMergeSort;
import model.SearchResult;

import java.io.*;
import java.util.ArrayList;
import java.util.List;

/**
 * Created by shweta14 on 12/12/2014.
 */
public class WordFileProcessor {
       /**<pre>
     * Description:
     * Read the given file and populate the list out of it.
     * @param fileName
     * @return List of words in the file (Pre condition: One word per
line)
     * </pre>
     */
    private List<String> getLinesIfFileExists(String fileName) {
        List<String> list = new ArrayList<>();
        File file = new File(fileName);
        BufferedReader reader = null;

        try {
            reader = new BufferedReader(new FileReader(file));
            String text = null;

            while ((text = reader.readLine()) != null) {
                list.add(text);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
            }
        }
        return list;
    }

    /**
```

```
     * <pre>
     * Does all the tasks in order to populate the response.
     * Following are the tasks:
     * 1. Read file
     * 2. If file has words, processing happens, nothing returned
otherwise.
     * 3. For Binary Search, sorted input is needed.
     *    - The file is sorted first and then handed of to search the
word from sorted list to the search method.
     * 4. For Linear Search, no such requirement is there,
     *    It can perform based on the input file y'ou have entered and
also depends on whether the entries are sorted.
     * @param fileName
     * @param key
     * @return List of search results
     * </pre>
     */
    public List<SearchResult> getSearchResult(String fileName, String
key) {
        List<SearchResult> results = new ArrayList<>();

        long start = 0L;
        SearchResult binarySearchResult = new SearchResult();
        SearchResult linearSearchResult = new SearchResult();
        //1
        List<String> fileLines = getLinesIfFileExists(fileName);

        if(fileLines.size() > 0){
            //2
            // Sort the file
            start = System.nanoTime();// clock starts
            List<String> sortedFile =
StringMergeSort.mergeSort(fileLines);// merge sort happens
            long mergeSortTime = System.nanoTime() - start;
//calculate total time


            //Binary search result
            StringBinarySearch stringBinarySearch = new
StringBinarySearch();
            start = System.nanoTime();

binarySearchResult.setLineNumber(stringBinarySearch.getIndexOfTheKey(s
ortedFile, key));
            binarySearchResult.setTimeTaken(System.nanoTime() -
start);
            binarySearchResult.setSearchMethod("Binary Search");
            binarySearchResult.setTimeComplexity("O ( log n )");
            binarySearchResult.setMergeSortTime(mergeSortTime);
            results.add(binarySearchResult);

            //Linear search result
```

```
            StringLinearSearch stringLinearSearch = new
StringLinearSearch();
            start = System.nanoTime();

linearSearchResult.setLineNumber(stringLinearSearch.getIndexOfTheKey(f
ileLines, key));
            linearSearchResult.setTimeTaken(System.nanoTime() -
start);
            linearSearchResult.setSearchMethod("Linear Search");
            linearSearchResult.setTimeComplexity("O ( n )");
            linearSearchResult.setMergeSortTime(mergeSortTime);
            results.add(linearSearchResult);
        }

        return results;
    }
}
```