Lab 5: Search

F28SG – Introduction to Data Structures and Algorithms (6 marks)

Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

Project Structure

The project is organized as follows:

- The `src` directory contains all the source files
  - **Entry.java** represents entries of persons in a catalogue that you should search over. Each entry contains the name of a person and their phone number.
  - **ASearch.java** is used to search over arrays. It contains an array called `catalogue` which contains Entry objects.
  - **LSearch.java** is used to search over linked lists. Each node in the linked list is an entry in the catalogue.
- The `test` directory contains the unit tests for the project.

Q1) Write Unit Tests for LSearch and ASearch (1 point)

This is how to think of a catalogue of four person entries:

Entry - Name: "Anna" - Number: 4512291	**Entry** - Name: "Hannah" - Number: 4513329	**Entry** - Name: "Joe" - Number: 4517749	**Entry** - Name: "Tim" - Number: 4513677

The search methods in `ASearch` and `LSearch` should return the corresponding phone number of the Entry in a catalogue of person entries. For example:

```
linearSearch("Hannah")
```

Would return `4513329`

In the `test` directory there are two test files: **ASearchTest.java** and **LSearchTest.java**. These are used to test the catalogue search methods that, under the hood, use an array (ASearch) and a linked list (LSearch).

In **ASearchTest.java** you should write two kinds of tests:

1. Searching for a name that **is** in the entry. The `linearSearch` and `binarySearch` methods should return the phone number for those names, tested by `testLinearSearchOK` and `testBinarySearchOK`.
2. Searching for a name that **is not** in the entry. The `linearSearch` and `binarySearch` methods should return `-1`, tested by `testLinearSearchFail` and `testBinarySearchFail`.

If you wish to write additional tests with more entries, make sure that entries are added to your catalogue in ascending order alphabetically (since your `binarySearch` implementation will rely on this assumption).

In **LSearchTest.java** you should write two tests:

1. One that tests a search for a name that is in the entry. The `linearSearch` method should return the phone number for those names, tested by `testLinearSearchOK`.
2. Another that tests a search for a name that **is not** in the entry. The `linearSearch` method should return `-1`, tested by `testLinearSearchFail`.

Q2) Implement Linear Search over an array (1 point)

In the **ASearch.java** file you are to write a **linear search** method:

```
public int linearSearch(String name){
        // your code
}
```

This method is given a person's name as a String parameter. The method should return the corresponding phone number of the name if it exists in the `catalogue` array of `Entry` objects. **IMPORTANT**: With Java, the `==` operator ***should not*** be used to compare string values. See ***hint1!*** if you are stuck.

Your linear search tests in **ASearchTest.java** from question 1 should now succeed.

State the Big-Oh complexity as a comment above the `linearSearch` method.

Q3) Implement Linear Search over a linked list (1 point)

In the **LSearch.java** file there is another **linear search** method to be implemented:

```
public int linearSearch(String name){
        // your code
}
```

This method should have the same functionality as your code from question 2. However, this time you will perform a linear search ***over a linked list***, rather than over an array. Start from the head node when linearly searching through the linked list.

Your linear search tests in **LSearchTest.java** from question 1 should now succeed.

State the Big-Oh complexity as a comment above the `linearSearch` method.

Q4) implement Binary Search for Arrays (2 points)

For this part you can assume that the given catalogue is sorted (see ***hint2!*** for why this is necessary to assume).

In the **ASearch.java** file there is a **binary search** method:

```
public int binarySearch(String name){
      Return binarySearch(0,current,name);
}
```

This calls the private `binarySearch` method with 3 arguments:

1. The left-most array index 0 to initialise the search.
2. The right-most array index current-1, which is the index position with the last entry in the **catalogue** array.
3. The name to be searched for.

Your task is to complete this private `binarySearch` method. **Your implementation of this method must be recursive**, so you should not use a loop inside the method. As in the other search methods you should return the corresponding number in the Entry object if it exists, or `-1` otherwise. See ***Hint3!*** for help.

Your binary search tests in **ASearchTest.java** should now succeed.

Indeed, all your unit tests should now succeed for your entire project.

State the Big-Oh complexity as a comment above the method:

```
private int binarySearch(int first,int last,String name)
```

Q5) Code Quality (1 point)

Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).

An additional mark is awarded if your code is deemed to be of high quality:

- **Big-Oh**
  - Has Big-Oh complexity been documented for all methods where requested?
- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**

- Are you using comments ***inside your implementation methods and test methods*** to provide an algorithmic commentary about what the code is doing.

Here is a **good** comment:

```
// creates unidirectional connection from the predecessor node to the new node
prevNode.nextNode = newNode;
```

Here is a **less useful** comment:

```
// sets preNode.nextNode to newNode
prevNode.nextNode = newNode;
```

- Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS, the shortcut is: ***⌘ + Alt + j***

- **Conventions**
  - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

Hints

Hint1! You will need to use the `getName()` and `getNumber()` methods of the Entry instances to get the respective name and number of each person. To check if two strings are equal, you need to use the `equals` method for Strings, i.e. `catalogue[i].getName().equals(name)`. The number of `Person` instances in the array is tracked with the `current` counter variable. When iterating over the array, it would be inefficient to traverse the array beyond that counter position.

Hint2! Your binary search implementation will split the search space in half with each recursive call. It will compare the searched name with the middle value. If the searched for name is alphabetically larger than the name of the middle entry, the recursive binary search should focus on the top half of the catalogue. If it is alphabetically smaller, then the recursive binary search will focus on the bottom half of the catalogue. This assumes that the catalogue is ordered alphabetically on the `name` field in each Entry.

Hint3! *The binarySearch method deviates from the one in the binary search lecture, as this lab requires comparison of strings instead of integers.*

To check if one string is smaller than the other (to figure out whether to search on the right or left of middle), use the `compareTo` method, i.e. `(catalogue[middle].getName().compareTo(name) > 0)`