Lab 7: Tries & Sorting

F28SG – Introduction to Data Structures and Algorithms (**12 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 topics of this lab are **tries** and **sorting**. This lab is larger than the other labs and is worth 12 marks.

The project is organized as follows:

- The `src` directory contains the following source files:
  - **Trie.java** implements the Trie discussed in the lecture and is where you should implement Q1 and the optional task
  - **ArraySort.java** where you should implement methods to sort arrays for Q3 and Q5
  - **DLinkedList.java** where you should implement methods to sort doubly linked lists for Q4
- The `test` directory contains the unit tests for the project:
  - **TrieTest.java** where you should write two tests for the Trie implementation (Q1)
  - **ArraySortTest.java** where you should write tests for bubble sort and quick sort (Q2)
  - **DlinkedListTest.java** where you should write tests for insertion sort (Q2)

Q1) Counting the number of Words in a Trie (2 points)

Complete the JUnit tests for the `countAllWords` method in **Trie.java**. This method will count the number of words that have been inserted into the Trie. Your tests should be completed in **TrieTest.java** and should test:

- Operations on an empty trie
- A tries when words that contains multiple "prefix" words (e.g. "ball" and "balloon").

Then complete the following method in the `TrieNode` class in **Trie.java**:

```
public int countAllWords(){
	// your code
}
```

This method should count the number of words stored in a Trie. Your implementation should be **recursive.** The `returnAllWords` method cannot be used in the implementation of `countAllWords`, although `returnAllWords` performs a full recursive traversal of the Trie, so you could use this programming structure as a template to work from.

Q2) Write Unit Tests for Sorting Algorithms (2 points)

In Q3, Q4 and Q5 you will implement algorithms to sort integers. These are:

- Bubble-sort of an **array** of integers (ArraySort.java, Q3)
- Insertion-sort of a **Doubly Linked List** of integers (DLinkedList.java, Q4)
- Quick-sort of an **ArrayList** of integers (ArraySort.java, Q5)

The bubble sort and insertion sort methods will sort a parameter `int[] arr` and sort it in-place, I.e. it will re-order values *in that array*.

In contrast, the quick sort method is different: it will take a parameter `S` of type `ArrayList<Integer>` and return a sorted version of the `S` array. See the method declaration in **ArraySort.java** to understand what the method's argument is and the type of the return value:

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> S)
```

In **ArraySortTest.java** and **DlinkedListTest.java** there are empty test methods that have been provided for you to complete. For each sorting method you should test the sorting of:

- an empty collection
- an already sorted (ordered) collection
- an un-ordered collection, i.e. elements in a random, unsorted order

The tests should check that the array/list is ordered and has the same size as before - suitable `isSorted()` methods have been provided in **ArraySort.java** and **DLinkedList.java** which you can use to test these with assertions.

Q3) Implement Bubble Sort (2 points)

Bubble sort is a sorting algorithm. **ArraySort.java** has an empty method skeleton:

```
public static void bubbleSort(int[] arr){
      // your code
}
```

Your task is to implement this method to sort the given array of integers.

You need to do the following in the method:

- Introduce a boolean variable `swaps` initially set to `true`
- While `swaps` is `true`
    - set `swaps` is `false`
    - step through the array from beginning to end (minus the last element)
        - for each step `i`

- if `arr[i+1]` is smaller than `arr[i]`
  - swap the values of `arr[i+1]` and `arr[i]`
  - set `swaps` to `true`

The `main` method contains a test case so you can see the program running. To run it with Eclipse, right click the project then click "Run As" then "Java application".

The bubble sort JUnit tests from Q2 should succeed when your bubble sort implementation is correct.

Q4) Insertion-Sort of Doubly Linked List (2 points)

The **DLinkedList.java** file contains an implementation of a Doubly Linked List of integers. This class contains an empty method:

```
public void insertionSort(){
      // your code
}
```

This method should sort the list using the insertion-sort algorithm discussed in the lecture. A `main` method has been provided for this class so that you can run your code.

When you have implemented insertion sort, your JUnit tests for insertion sort from Q2 should now succeed. Two additional methods have been provided in this class to support you in your implementation, in case you find them helpful:

- `int delete(Node n)` deletes node `n` from the doubly linked list and returns its value.
- `insertAfter(Node n,int val)` inserts a new node containing value `v` after node `n`.

Q5) Implement Quick-Sort (3 points)

In **ArraySort.java** there is an empty method

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> S){
     // your code
}
```

Your task is to implement the quick sort algorithm in this method. Unlike the other array sorting methods, this one will ***return a sorted version*** of the input array, rather than sorting it in-place.

The quick sort algorithm works as follows:

- If the size of (input) `S` is less than or equal to one, then `S` is sorted so you can return (i.e. base case)
- Select an element of `S` to be the `pivot`. You can choose which element this is, e.g. the first element of `S`, the middle element of `S`, or the last element of `S`.
- Create 3 new `ArrayLists` (holding values of type `Integer`):
  - L which should store elements of S that are less than the `pivot`
  - E which should store elements of S that equal to the `pivot`
  - G which should store elements of S that are greater than the `pivot`

- While S is not empty
    - **get and delete** the first element and add it to one of L, E and G, according to how it compares with the `pivot`
- After the while loop, recursively perform quick sort on L and G and assign those sorted lists into new `ArrayLists` called `sortedL` and `sortedG`.
- Create a new empty `ArrayList`, then combine all three sorted `ArrayLists` into this `ArrayList` in the order:
    - elements of the sorted version of L
    - elements of E (which don't need sorting since they all have the same value as the pivot)
    - elements of the sorted version of G
- Return that `ArrayList` from the method.

`The ArrayLists` class has an `addAll` method that will be useful for combining those arrays into S.

The `main` method contains a test case you can use to run your code.

All your JUnit tests for your lab 7 project should now pass.

Q6) 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:
- **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***

Additional Challenges

Return all words with a given prefix

Given a prefix string, return all strings in a trie that have that given string as a prefix. For example, if a trie contains the word "balloon", then searching for words with the prefix "ba" should return a list that at least includes the string "balloon".

You should complete the following method in **Trie.java** :

```
public ArrayList<String> wordsWithPrefix(String str) {

}
```

JUnit tests `wordsWithPrefixTestTrue` and `wordsWithPrefixTestFalse` are provided in **TrieTest.java**.

Is there a word with a given prefix?

Given a prefix string, return true if and only if there is a string in the Trie that has that string as a prefix. For example, `true` should be returned for the prefix "zeb" if a tree containing the word "zebra". Again, it may be best to implement the helper method in TrieNode **recursively.** You should complete the following method (from **Trie.java**):

```
public boolean areWordsWithPrefix(String str){

}
```

JUnit tests `areWordsWithPrefixTestTrue()` and `areWordsWithPrefixTestFalse()` are provided in **TrieTest.java**.