Data Structures & Algorithms Problem Sheet 3

Worth 7% of your final grade
Due Friday 4/12, 7pm in Moodle
Submit the following files <u>in a single ZIP archive</u> to Moodle: `Contact.java`, `Sorter.java`, `AVLTree.java`, `AVLTreeNode.java`

Motivation: Sorting is a very popular topic in coding interviews. When applying for a job at a tech company, it definitely helps if you can say "yes, I know that sorting algorithm – and I have implemented it, too." Sorting is also used everywhere: whenever you do an Internet search or look at online bank transactions there is likely some sorting happening in the background – because sorting data makes many algorithms faster (e.g. search). Similarly, trees are among the most popular and most widely used data structures and many problems can be solved with them. Implementing the AVL tree balancing is a challenge, but it makes your trees much faster.

Can you help Gabriel sort his toys?

Note:

- If you have questions about this coursework please ask them in our Moodle forum. You will get answers more quickly as the whole teaching team can answer, and you are helping your classmates.
- Marks will be given based on correctly working code and correct answers submitted. You do not need to comment your code, unless there is a problem with it (then comments may give you partial marks).
- Do not copy code or answers from others. More about plagiarism and how to avoid it here: http://www.bath.ac.uk/library/help/infoguides/plagiarism.html
- If you submit code that does not compile, you can only get a maximum of 60% of the marks for it. If your code does not work, it is much better to make it compile and leave comments why you think it does not work in the code.
- For this problem sheet you should implement your own algorithms and data structures and not use pre-defined ones such as the ones from the Java API. You may use any type of array, e.g. `Object[]`. You may not use predefined data structures such as `ArrayList` or API methods such as in `java.util.Arrays`.
- Do not change the method signature (i.e. parameters, name, types) when implementing a given method. Changing the method signature may make it impossible for the marker to test it correctly. **Change only the parts in the given files that are marked for your code.**
- You can assume that your methods will only be used with correct inputs, i.e. your code does not need to handle errors caused by incorrect inputs.
- Please respect the Java conventions for naming of methods and classes, e.g. the use of upper/lower case: http://www.oracle.com/technetwork/java/codeconventions-135099.html
- Please ensure that the Java files you submit are recognized as correct text files (beware of copy-pasting special characters from the lecture slides as they can confuse the Java compiler).
- If you are struggling with Java or programming tools such as IntelliJ, have a look at the Topic 0 resources such as the video tutorials on YouTube. Many students found them useful.
- After submitting, please **check your submission on Moodle to make sure it is there and you have submitted the right files!!!** Also, have you submitted them to the right assignment?

- We aim to provide feedback no later than three weeks after the submission date.
- If you need a deadline extension, please apply to your Director of Studies. Please email submissions with extended deadline to the lecturer instead of submitting them on Moodle.

Question 1: Comparing Contacts (10% of marks)
Download the file `Contact.java` from Moodle. It implements a Java class which stores a contact with a first and last name. Implement the method `compareTo(Contact c)` which compares a Contact object with another given Contact object `c`. It returns an integer number which indicates whether the contact is before, equal to, or after the contact `c`, similar to the `compareTo` method of class String (have a look at the description below to understand what it should do):
http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo(java.lang.String)
However, the `compareTo` method of class Contact should be implemented so that it compares the two objects as follows: for contacts with different last names, it should compare their last names using lexicographic order (for that you may use the `compareTo` method of class String); for contacts with the same last name it should compare their first names using lexicographic order (again you may use `compareTo` of class String).

Question 2: Selection Sort (10% of marks)
Implement method `selectionSort` of class `Sorter`, so that it sorts the given array of contacts using selection sort. You may use the code from the lecture slides.

Question 3: Insertion Sort (10% of marks)
Implement method `insertionSort` of class `Sorter`, so that it sorts the given array of contacts using insertion sort. You may use the code from the lecture slides.

Question 4: Quicksort (10% of marks)
Implement method `quickSort` of class `Sorter`, so that it sorts the given array of contacts using quicksort. You may use the code from the lecture slides. Note that you may need to add another internal quicksort method with more parameters to `Sort`, which is called by the given method.

Question 5: Mergesort (10% of marks)
Implement method `mergeSort` of class `Sorter`, so that it sorts the given array of contacts using mergesort. You may use the code from the lecture slides. Note that you may need to add another internal mergesort method with more parameters to `Sort`, which is called by the given method.

Submit your files `Contact.java` and `Sorter.java`. No other files need to be submitted for Questions 1-5.

Question 6 on next page…

Question 6: Binary Trees (50% of marks)
Given the following incomplete Java class AVLTree, which can be downloaded from Moodle:

```
class AVLTree {
  AVLTreeNode root;
  // Note: you may define other variables here

  public AVLTree() {
    // TODO implement this
  }

  public void createTestTree() {
    // TODO implement this
  }

  public void print() {
    // TODO implement this
  }

  public boolean inTree(String e) {
    // TODO implement this
  }

  public void insert(String e) {
    // TODO implement this
  }
}
```

a) (10% of marks) Based on the information given in the lecture, implement the AVLTreeNode class and the AVLTree constructor, which creates a new binary tree. Implement also the `createTestTree` method, which should construct a perfectly balanced binary tree for testing, using the strings "1", "2", "3", "4", "5", "6", "7" as elements. Finally, implement the `print` method, which prints out the current AVL tree based on one of the tree traversals described in the lecture, using one line per element and indentation to signify the depth of an element in the tree. The printed output for the tree created by `createTestTree` should look like this:

```
4
  2
    1
    3
  6
    5
    7
```

Question 6 continued on next page…

b) (15% of marks) Implement the `inTree` method, which checks whether the given string element `e` is in the current tree. If it is in the tree, the method should return true, and false otherwise. Implement also the `insert` method, which can use the usual insertion algorithm for binary search trees as described in the lecture, without re-balancing. You may use the `String compareTo` method. You may assume the tree can never contain any duplicates.

c) (25% of marks) Modify and extend your code so that the tree is balanced after each insertion, based on the AVL tree algorithm described in the lecture.

Submit your files `AVLTree.java` and `AVLTreeNode.java`. No other files need to be submitted for Question 6.