Java Interim Test

Wednesday 20th March 2019

12:00 – 15:00

THREE HOURS

(including 10 minutes planning time)

- Please make your swipe card visible on your desk throughout the test.
- After the planning time, log in using your username as **both** your username and password.

There are **TWO** sections: Section A and Section B, each worth 50 marks.

Credit will be awarded throughout for code that successfully compiles, which is clear, concise, usefully commented, and has any pre-conditions expressed with appropriate assertions.

Important note:

- In each section, the tasks are in increasing order of difficulty. Manage your time so that you attempt both sections. You may wish to solve the easier tasks in both sections before moving on to the harder tasks
- It is critical that your solution compiles for automated testing to be effective. Up to **TEN MARKS** will be deducted from solutions that do not compile (-5 marks per Section). Comment out any code that does not compile before you log out.
- You can use the terminal or an IDE like IDEA to compile and run your code. **Do not ask an invigilator for help on how to use an IDE.**
- Before you log out at the end of the test, you **must** ensure that your source code is in the correct directory otherwise your marks can suffer heavy penalties. Only code in the original directories provided will be checked.

Section A

Problem Description

In this section you will create Java classes to model files (*data files* and *directories*) in a file system, and implement various operations on files. The aim of this exercise is to assess your object-oriented programming skills—the file system model we consider here is drastically simplified compared with a real file system.

The specification of what you should implement is described via a combination of: this document; a number of JavaDoc comments in the skeleton files; and a set of JUnit tests for each question. Part of the challenge is to judge what you are expected to implement given these materials. If in doubt, write brief but clear source code comments to describe any assumptions you make when implementing your solution.

Getting Started

The skeleton files are located in the `filesystems` package. This is located in your Lexis home ‘fstxt’ directory at:

- `~/fstxt/SectionA/src/filesystems`

During the test you will need to make use of (though should **NOT** modify) the following:

- `filesystems/DocFile`: an abstract class specifying various operations on files. The `Doc` prefix is to avoid confusion with the Java class `File`.

You should not modify this file in any way.

During the test, you will populate the following:

- `filesystems/DocDataFile.java`: an empty class that you will fill to represent data files.
- `filesystems/DocDirectory.java`: an empty class that you will fill to represent directories.
- `filesystems/DocFileUtils.java`: a class containing stubs for a number of static methods that you will implement.

You may feel free to add additional methods and classes beyond those specified in the instructions as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed in the `filesystems` package.

Testing

You are provided with four test classes, named `filesystems/Question`i`Tests.java`, where $1 \leq i \leq 4$. These contain initially commented-out JUnit tests to help you gauge your progress during Section A. **As you progress through the exercise you should un-comment the test associated with each question in order to test your work.**

You can also add extra tests in these files as you see fit, to help you develop your solution, but these will not be marked.

What to do

First, study the `DocFile` abstract class carefully. A `DocFile` is constructed from a string, representing its name. This can be retrieved via the `getName` method. In addition there are a number of abstract methods documented with brief JavaDoc comments:

- The size of a file, in bytes, can be obtained.
- We can ask whether a file is a directory (`DocDirectory`) or a data file (`DocDataFile`).
- We can use `asDirectory` to ask for a file to be reinterpreted as a directory; this will fail if the file is not actually a directory. Analogously, we can ask for a file to be reinterpreted as a data file via `asDataFile`.
- We can ask for a file to be *duplicated*. The meaning of duplication for directories and data files is described below.

1. **Data files.**

   Your first task is to implement support for *data files*.

   You should do this by adapting `DocDataFile` so that it is a subclass of `DocFile`. You should implement the abstract methods specified by `DocFile` appropriately, and provide one additional method, `containsByte`. Use the following notes, as well as the tests for Question 1, to help you decide how to implement these methods. Implementation decisions that should be self-evident are not detailed.

   In addition to a string name, which it inherits from `DocFile`, a data file should have an associated array of type `byte`, representing the byte contents of the file. The constructor of `DocDataFile` should be public, and should construct the data file from a string and an array of bytes.

   The *size* of a data file is the sum of the length of its name and the size of its byte array.

   The `asDataFile` method should return the data file, while the `asDirectory` method should throw an `UnsupportedOperationException`.

   A `DocDataFile` should be duplicated by creating a new `DocDataFile` with the same name and byte array as the original file.

   You should add a new public method specific to `DocDataFile`, called `containsByte`, taking a `byte` as a parameter and returning true if and only if the byte occurs in the byte array associated with the data file.

   Test your solution using (at least) the tests in `Question1Tests`.

   [10 marks]

2. **Equality for data files.**

   Your next task is to implement a specific notion of *object equality* for data files.

   Two data files should be regarded as equal if and only if (1) they have identical names, and (2) the contents of their associated byte arrays match. Add appropriate methods to implement this notion of object equality for `DocDataFile`.

   Because the combination of object equality and inheritance can lead to subtle problems, adapt `DocDataFile` so that sub-classes of `DocDataFile` cannot be created.

Test your solution using (at least) the tests in `Question2Tests`.

[8 marks]

3. **Directories.**

Your task now is to add support for *directories*.

You should do this by adapting `DocDirectory` so that it is a subclass of `DocFile`. You should implement the abstract methods specified by `DocFile` appropriately, and provide several additional methods described below. Use the following details, as well as the tests for Question 3, to help you decide how to implement the abstract methods from `DocFile`. Again, implementation decisions that should be self-evident are not detailed.

In addition to a string name, which it inherits from `DocFile`, a directory contains zero or more files with distinct names, each of which can either be a data file or a directory. The collection of files associated with a directory can be represented in various ways; it is up to you to choose a suitable representation, prioritising code simplicity. The constructor of `DocDirectory` should be public, and should construct an empty directory with a given string name.

The *size* of a directory is simply the length of its name. Note that the contents of the directory do *not* contribute to the size of the directory.

The `asDirectory` method should return the directory, while the `asDataFile` method should throw an `UnsupportedOperationException`.

A `DocDirectory` should be duplicated in a *deep* manner to yield a distinct `DocDirectory` with the same name as the original directory, and containing duplicated versions of each file contained in the original directory.

You should add the following new public methods specific to `DocDirectory`:

- `boolean containsFile(String name);` – returns true if and only if the directory contains a file with the given name
- `Set<DocFile> getAllFiles();` – returns all the files contained in the directory
- `Set<DocDirectory> getDirectories();` – returns all the directories contained in the directory
- `Set<DocDataFile> getDataFiles();` – returns all the data files contained in the directory
- `void addFile(DocFile file);` – throws an `IllegalArgumentException` if the directory already contains a file with the same name as `file`, otherwise adds `file` to the directory
- `boolean removeFile(String filename);` – returns false if the directory does not contain a file named `filename`, otherwise removes the file named `filename` from the directory and returns true
- `DocFile getFile(String filename);` – returns a file named `filename`, which can be assumed to be contained in the directory

Test your solution using (at least) the tests in `Question3Tests`.

[16 marks]

4. **Utility methods.**

   Your final task for Section A is to implement the utility methods specified in `DocFileUtils`. You should use the JavaDoc comments associated with each method, as well as the tests for Question 4, to decide the manner in which these methods should be implemented. If in doubt, write clear source code comments to state any assumptions you make about how these methods are intended to behave.

   Try to implement these methods without using any downcast operations. For maximum credit, use streams and method references or lambdas to express your solution in a concise form. (A working solution that use loops instead of streams will still receive partial credit.)

   Test your solution using (at least) the tests in `Question4Tests`.

   [16 marks]

Total for Section A: 50 marks

Useful commands

```
cd ~/fstxt/SectionA

mkdir out

javac -g -d out
-cp /usr/share/java/junit4-4.12.jar
-sourcepath src:test
src/filesystems/*.java

java
-cp /usr/share/java/junit4-4.12.jar:/usr/share/java/hamcrest-core-1.3.jar:out
org.junit.runner.JUnitCore
filesystems.Question1Tests
filesystems.Question2Tests
filesystems.Question3Tests
filesystems.Question4Tests
```

Section B

You are required to implement some methods of a text compressor that uses a simplified Huffman coding using a tree-based structure.

A Huffman code is an optimal prefix code used for lossless data compression (your zip compressor uses something similar :)). Given a text (which for us is a list of words), our Huffman encoder first counts how many times a word appears in the text and then maps each word into a sequence of bits (encoding) whose length is inversely proportional to its count. The words appearing more frequently in the text will be encoded with fewer bits, while those appearing less frequently use more bits.

Our Huffman encoder provides three main operations: build the encoding, compress, and decompress.

Building the encoding. To build the encoding, we take as input a map associating to each word the number of times it appears in the text. It then builds a binary tree to construct the optimal encoding.

The tree is constructed via the following steps:

1. create as many (leaf) nodes as there are words in the map and record in such nodes the word and its corresponding count
2. place all the node in a priority queue, which keep them ordered in ascending order of the attribute count (so the node with the smallest count get polled first).
3. while there is more than one element in the queue:
   (a) dequeue the two nodes with smallest count
   (b) create a new (internal) node having the two nodes as left and right children, respectively, and assign to this node a count which is equal to sum of the counts of its children, then add it to the queue (where it will be stored according to its count).
4. the single remaining node is the root of the tree.

For example, let's say our text is composed by the list of words:

`shakespeare, dickens, tolkien, shakespeare, shakespeare, dickens`

The corresponding word count map would be:

```
shakespeare: 3
dickens: 2
tolkien: 1
```

The construction of the tree proceeds as follows. One leaf node is created for each word, recording both the word and its count, and added to the priority queue. The priority queue will look like Figure 1, where leaf nodes are represented by boxes with solid borders).

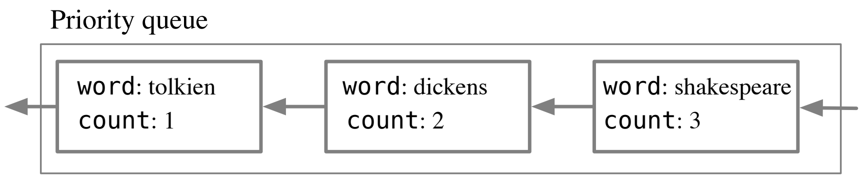

Figure 1: Priority queue: one leaf node per word, ordered by count ascending.

Because the queue contains more than one node, we pull the first two nodes and create a new internal node (`intA`) whose count is the sum of the counts of its children (Figure 2; internal nodes are represented with dashed borders). Notice that internal nodes do not have words associated to them, only counts.

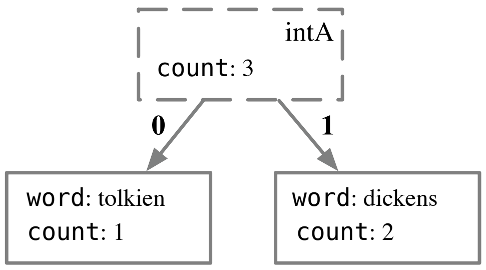

Figure 2: A new internal node (`intA`) is created, with `tolkien` and `dickens` as children.

The new node `intA` is added to the queue, where it will be stored according to its count (Figure 3).

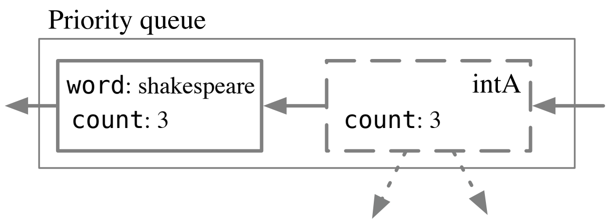

Figure 3: Priority queue after `intA` is added.

Because the queue contains again more than one element, we pull the first two of them and create a new internal node (`intB`) having them as left and right children, respectively, and having as count the sum of the counts of its children (Figure 4).

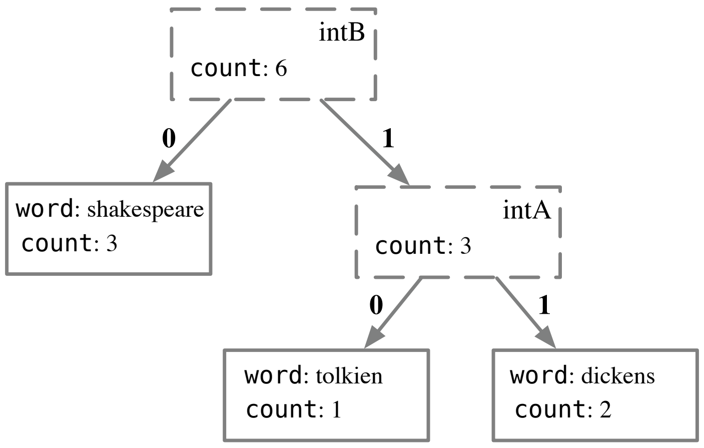

Figure 4: A new internal node (`intB`) is created, with `shakespeare` and `intA` as children.

The new node `intB` is then added to the queue (Figure 5).

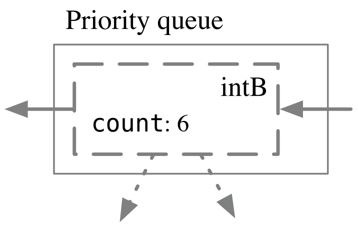

Figure 5: Priority queue containing only `intB`.

At this point, the queue contains only one element, thus `intB` is the root of the Huffman tree.

In the figure, the pointers to the left and right child of an internal node have been labeled **0** and **1**, respectively. For each word in the tree, its corresponding binary encoding is the sequence of **0**s and **1**s encountered along the path from the root to the word (which is always stored in a leaf).

A this point, the tree should be traversed to build a map associating each word to its corresponding encoding. For the example, the resulting map would look like:

```
shakespeare: 0
dickens: 11
tolkien: 10
```

As you can notice even from this small example, the most frequent word is encoded using fewer bits than the others.

Compression. Given the map that associates each word to its binary encoding, compressing a text, which for us is a list of word, is as simple as finding for each word its binary encoding and concatenating such encodings.

In the example:

```
shakespeare -> 0
dickens -> 11
```

```
tolkien -> 10
shakespeare -> 0
shakespeare -> 0
dickens -> 11
```

Which will produce the final sequence of bits 011100011.

Notice that in our simplified compressor, we do not accept any word that is not in the encoding map. Referring to our example, this means that you can compress every sequence of “shakespeare”, “dickens”, and “tolkien”, but cannot compress a text containing, for example, “hemingway”.

Decompression. Given a sequence of bits, the decompressed text can be recovered traversing the tree following the left or right pointer, depending if the next bit is 0 or 1, respectively. The traversal begins from the root and follows the 0-1 path until reaching a leaf. At this point, the word stored in the leaf is appended to the decompressed text and the traversal restarts from the root, following the remaining bits.

Let us decompress the sequence 011100011 from the tree in Figure 4.

Beginning from the root, the first bit is 0 so we go to the left child. The child is the leaf corresponding to `shakespeare`. We append “shakespeare” to the decompressed text and proceed with the remaining sequence 11100011.

We start again from the root. This time the first bit is 1, so we go to the right child (`intA`). `intA` is an internal node, so we need to proceed. Because the next bit is again 1, we go to the right child. The latter is a leaf node, corresponding to `dickens`. We append “dickens” to the decompressed text (which is now [“shakespeare”, “dickens”]), and proceed with the remaining subsequence 100011.

The procedure continues until there are no more bits to process and we recovered the decompressed text.

Notice that if a sequence cannot be processed entirely (because it does not terminate in a leaf node), the compressed text is invalid. Unfortunately, this is a one-way check, as the text can be also altered into another valid one. But it is nonetheless a check.

Now that you understood how your zip compressor works, it’s time for your tasks.

Getting Started

The skeleton files are located in the `src` folder of the project located in your Lexis home ‘fstxt’ directory at:

- `~/fstxt/SectionB/`

The source files are located in `src` which contains a `huffman` package with the source code of the classes you are going to use or implement, and a `test` folder with a set of JUnit test classes you can use to check your implementation and to gain additional clues about what it is expected to do. An additional folder `resources` contains the text of Shakespeare’s Romeo and Juliet in case you want to try your compressor on a larger text. Look at the main file of the class `Compressor` to see how to do it (the expected numeric encoding of the compressed text is in the file `RomeoAndJuliet.clean.compressed.txt`, just for your reference).

During the test you will need to make use of (though should **NOT** modify) the following:

- `Utility`: contains some utility methods to load text from files and report some statistics about its compression. You cannot modify any of its public methods, with the exception of `countWords` as described later.
- `HuffmanEncoderException`: you can add constructors if needed, but cannot remove or rename the class.
- `HuffmanEncoder`: cannot change its name, nor its public methods. Apart from this, you can add any method/field you see fit.

During the implementation, you will populate and/or modify `HuffmanEncoder` and the method `Utility.countWords`.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed (in a suitable package) in the `src` or `test` directories, depending if it is part of the implementation or of its tests, respectively.

Testing

The `test` folder, which will not be marked, contains some JUnit tests to help you check your progress during Section B.

`HuffmanEncoderTest` contains a set of functional tests to help you find problems in the behavior of your `HuffmanEncoder` implementation.

`WordCountTest` contains some test cases to help you identify possible synchronization in your parallel implementation of (`Utility.countWords`).

You can also add extra tests in this file as you see fit, to help you develop your solution.

If you decide you will not implement some methods, you may want to remove the statements and the assertions where these methods are used.

Remember, as usual, that tests can find bugs, not exclude their presence :)

What to do

General observations. Most of the methods you are required to implement can be implemented either iteratively or recursively. You are free to choose your preferred way. As a suggestion, prefer simplicity over time or memory efficiency for these operations. Any correct, complete, well-designed, and non-redundant implementation can achieve the maximum score. Extreme improvements of time or memory performance are not required to achieve the maximum score, so follow good design principles and common sense, and keep your code readable and easy to fix.

1. **Implement the static method `buildEncoder` of the class `HuffmanEncoder`.** This method is composed of two phases: building the tree and then populating the map `word2bitsequence`. The classes to construct both internal nodes and leaves

of the tree are already defined in the code (remember that you can test if a node is a leaf using `node instanceof HuffmanLeaf`; similarly for internal nodes).

A priority queue is already defined in your code and the class `HuffmanNode` representing a node in the tree implements `Comparable`, thus the queue can keep its element ordered.

To add an element to the queue, use the method `queue.offer(<node>)`. To retrieve and remove the node with the smallest count (the first in the priority queue), use the method `queue.poll()`.

Notice that, for simplicity, we always assume that there are at least 2 distinct words in any text (see checks in the method's body). This guarantees that the queue is always initialized with at least two elements.

While the queue contains more than one element, you need to poll the first two, create a new internal node having them as left and right child, respectively, and add this new internal node to the queue using the method `offer`. When the queue will contain a single element, that element will be the root of the tree (see also the description and examples in the previous section).

After the tree is built and the field `root` of the HuffmanEncoder is set, traverse the tree to populate the map `word2bitsequence`. Every leaf contains a word and the encoding of such word is the sequence of 0 and 1 collected along the path from the root to the leaf, where 0 is collected when moving to the left child and 1 when moving to the right child.

The map `word2bitsequence` maps each word to the corresponding sequence of bits.

See also the test suite `HuffmanEncoderTest` for additional examples of expected behaviors.

Note: the private fields `root` and `word2bitsequence` are provided only for your convenience. You may feel free to remove/modify them if you see it fit.

[**20 marks**]

2. **Implement the method `compress` of the class `HuffmanEncoder`.**

   This method takes as argument a list of words, maps each of the to its corresponding encoding, and concatenate the encodings into a single string. The encoding of a word is the value of `word2bitsequence.get(<word>)`, which is a string representing the binary encoding of the word.

   If a word in the input text is not contained in the keys set of the map, the method should throw an `HuffmanEncoderException`.

   [**5 marks**]

3. **Implement the method `decompress` of the class `HuffmanEncoder`.**

   This method takes as argument a string representing a sequence of 0s and 1s and traverses the tree to recover the decompressed text. Starting from the root, at every step, the traversal follows the left or right child when the next bit is 0 or 1, respectively, until a leaf is reached. The word contained in the leaf is added to the result list, the prefix of 0s and 1s used to reach the word is removed, and the procedure is repeated with the remaining subsequence (see also examples in the previous section).

[15 marks]

4. **Modify the static method `Utility.countWords` to use multiple threads to count the words.**

   You have to explicitly create multiple threads (no parallelstream), give each of them a portion of the list of words to count, and wait for their completion (and aggregate the thread-local results, if needed). Besides creating multiple threads explicitly, you have no restrictions on the use of thread-safe Java collections or atomic types.

   Your solution has to work for any (strictly positive) number of words to be counted.

   [10 marks]

Total for Section B: 50 marks

Useful commands

```
cd ~/fstxt/SectionB

mkdir out

javac -g -d out
-cp /usr/share/java/junit4-4.12.jar
-sourcepath src:test
src/huffman/*.java test/huffman/*.java

java -cp /usr/share/java/junit4-4.12.jar:/usr/share/java/hamcrest-core-1.3.jar:out
org.junit.runner.JUnitCore
huffman.HuffmanEncoderTest
huffman.WordCountTest
```

You can use this paper for planning

You can use this paper for planning