Java Final Test

Tuesday 3rd May 2022

09:00 – 12:00

THREE HOURS

(including 10 minutes planning time)

Please monitor COMP40009 on EdStem for announcements

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 for help on how to use an IDE.**
- Before your final commit/push, 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. Please make sure that your folder structure for each Section is:

  ```
  ~/javafinaltestpartX_username/.git/
  ~/javafinaltestpartX_username/lib/
  ~/javafinaltestpartX_username/src/
  ~/javafinaltestpartX_username/test/
  ```

  where X is "a" or "b" and username is your login.
- Before the 12:00 deadline, you will need to **commit/push** your code to GitLab and **submit** it via LabTS to CATe. Code that is pushed to GitLab after the deadline will be capped at **zero**.

Useful commands

```
cd ~/javafinaltestparta_username/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore aeroplane.Question#Tests

java
-ea
-cp "./lib/*:out"
aeroplane.TestSuiteRunner



cd ~/javafinaltestpartb_username/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore alphatree.TestQuestion#

java
-ea
-cp "./lib/*:out"
alphatree.TestSuiteRunner
```

Section A

Problem Description

Your job is to write a program that will allocate seats on an aeroplane. The aeroplane has 50 rows, numbered 1–50. Each row has six seats, labelled 'A'-'F'. Row 1 is reserved for crew. Rows 2–15 are reserved for business class passengers. The remaining rows are for economy class passengers. Emergency exits are located at rows 1, 10 and 30. We consider the aeroplane seats to be ordered as follows: 1A, 1B, ..., 1F, 2A, 2B, ..., 2F, etc. The first and last seats are 1A and 50F, respectively. In the instructions below, the specific ranges 1–50 and 'A'-'F' are used for rows and seat letters. However, your solution should be designed so that minimal code modifications would be required to change the number of rows and seats per row, which rows are reserved for which types of passengers, and which rows are emergency exit rows. You can assume that each reserved section will always be a contiguous sequence of rows.

Getting Started

The skeleton files are located in the `aeroplane` package. This is located in your Lexis home directory at:

- `~/javafinaltestparta_username/src/aeroplane`

You are provided with the following files:

- `Seat.java`: a skeleton class that you will fill out to represent an aeroplane seat
- `Passenger.java`: a skeleton class that you will fill out and subclass to represent different types of aeroplane passenger
- `SeatAllocator.java`: an incomplete class which you will complete to represent and perform the allocation of passengers to seats
- `Luxury.java`: an enumeration representing various luxuries to which business class passengers are entitled (do not modify this file)
- `AllocationNotPossibleException.java`: a checked exception used to handle the case where it is not possible to allocate a seat to a passenger (do not modify this file)
- `MalformedDataException.java`: a checked exception used to handle badly formed input data (do not modify this file)

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 `aeroplane` package.

Testing

You are provided with a set of test classes under:

- `~/javafinaltestparta_username/test/aeroplane`

There is a test class `Question`i`Tests.java` for each question i. 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 class associated with each question in order to test your work.

These tests are not exhaustive and are merely intended to guide you: while it is good if your solution passes these tests, your work will be assessed with respect to a larger, more thorough test suite. You should thus think carefully about whether your solution is complete, even if you pass all of the given tests.

What to do

1. **Fill out the `Seat` class.** Complete the `Seat` class, to meet the following requirements:
   - A seat should be represented by a row and letter, in the appropriate ranges.
   - A seat should be constructed from a row and letter. An `IllegalArgumentException` should be thrown if the given row or letter are not appropriate.
   - A seat should be immutable and it should not be possible to create subclasses of `Seat`.
   - The string representation of a seat should be the row (with a leading zero if the row is less than 10) followed by the letter, e.g. a seat with row 5 and letter E should be represented by the string "05E".
   - `Seat` should provide an instance method, `boolean isEmergencyExit()`, that returns true if and only if the seat is located in an emergency exit row.
   - `Seat` should provide an instance method, `boolean hasNext()`, which returns true for all but the last seat in the aeroplane, according to the ordering of seats given above.
   - `Seat` should provide an instance method, `Seat next()`, which should return a `Seat` object corresponding to the next seat in the ordering of seats given above. If there is no next seat, a `NoSuchElementException` should be thrown. **Hint:** if you add an `int` to a `char` the result has type `int`; you may find that you need to cast this back to `char`.

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

   [10 marks]
2. **The `Passenger` class hierarchy.** Write a hierarchy of classes to represent the three types of passenger: crew members, business class passengers and economy class passengers. The hierarchy should include at least the new classes `CrewMember`, `BusinessClassPassenger` and `EconomyClassPassenger`.
   - The abstract class `Passenger` (the skeleton of which you are given) should be at the top of this hierarchy. All other passenger classes should be derived (directly or indirectly) from this class. Where possible you should avoid duplicating state or functionality that is common to multiple classes. You should also maximise encapsulation in the design of these classes.
   - Every passenger should have a first name and surname that should be represented as strings.
   - Every non-crew passenger should have a non-negative integer field representing their age.

- All passengers should have an instance method, `bool isAdult()`, that returns true if and only if the passenger is at least 18 years of age. You should assume that all crew are adults.
- Business class passengers should have an associated *luxury*. This can be either *champagne*, *truffles* or *strawberries*, and should be represented using the `Luxury` enumeration with which you are provided.
- The string representation of a passenger should consist of the passenger's type (crew, business class or economy class), followed by details of the fields associated with the passenger. The unit tests for Question2 dictate constraints on the precise format this string should take: study these tests and ensure that your solution passes them all.

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

[16 marks]

3. **Equality on `Seats`.** Further edit the `Seat` class so that two `Seat` objects are regarded as equal if and only if they have identical fields. Test your solution using (at least) the tests in `Question3Tests`.

   [5 marks]

4. **Allocating seats.** Look at the skeleton class `SeatAllocator`, which represents an allocation of seats to passengers. `SeatAllocator` has a single field, `allocation`, which maps seats to passengers. If a seat is not mapped to any passenger, the seat is free. The `toString()` method of `SeatAllocator` provides details of allocated seats in seat order, based on the `toString()` methods of `Seat` and the `Passenger` class hierarchy.

   `SeatAllocator` provides an instance method, `void allocate(String filename)`, whose job is to read passenger details from the specified input file and allocate seats for these passengers. The functionality for seat allocation is missing: it is your job to complete this functionality.

   - A stub instance method, `allocateInRange(...)`, is provided. Fill out this method so that it allocates the given passenger to the first suitable seat in the range `first`–`last` (inclusive). A seat is suitable for a passenger if a) the seat is free, and b) if the seat is in an emergency row then the passenger is an adult. If there is no suitable seat, the method should throw an `AllocationNotPossibleException`.
   - Look at the incomplete methods `allocateCrew(...)`, `allocateBusiness(...)` and `allocateEconomy(...)`. These methods read passenger fields from a buffered reader object. Complete each of these methods by a) using the fields which have been read to construct an appropriate passenger object, and b) calling `allocateInRange` with this passenger object and an appropriate range of seats.

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

   [15 marks]

5. **Counting adults.** In class `SeatAllocator`, write a static method, `int countAdults(...)`, that takes a set of passengers and returns the number of adults in the set. It should be possible to call the method with *any* set of passenger objects, not just with a set of the form `Set<Passenger>`. Test your solution using (at least) the tests in `Question5Tests`.

   [4 marks]

Total for Section A: 50 marks

Section B

Problem Description

This question is about a word guessing game we will call *bangman*. Your opponent randomly chooses a secret word, consisting of n lowercase characters, from a given dictionary and they tell you its length. Your goal is to guess the secret word in the minimum number of turns. Each turn you can send the opponent a message:

1. You can send the message `ask(xs)` to ask if the secret is a given string `xs`.
2. You can send the message `ask(i, xs)` to ask if the letter at index `i` in the secret is any letter in a given string `xs`, where $0 \leq$ `i` $<$ `n`.

Your opponent will either answer `true` or `false`. The game is finished when the answer to your guess `ask(xs)` is `true`, since it is the same word as the secret.

For example, your opponent could choose the word `"imperial"`, and your play might be as follows:

1. `ask(0, "aeiou") = true`
2. `ask(0, "ae") = false`
3. `ask(0, "i") = true`
4. `ask(3, "abcdefghijkl") = false`
5. `ask(3, "mnopqrs") = true`
6. `ask(3, "m") = true`
7. `ask("imperium") = false`
8. `ask("imperial") = true`

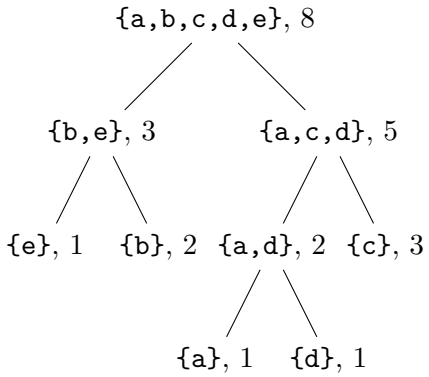
The game has ended after 8 turns because you have guessed the correct word. This was a remarkably short game, and it would usually take many more guesses to finish the game.

If you like, you can even play this game on the command line, though it is not recommend you spend too much time on it right now, exciting as it is. To help you cheat, the game has been set so that it actually tells you the secret as a hint.

Your task is to write some code that will help a program to play this game for you by implementing the algorithm described below. The algorithm does not implement an optimal strategy, but performs relatively well.

The overall strategy will be to guess letters that occur with high frequency first. The *frequency* of a letter is calculated by counting the number of occurrences of each letter in the whole dictionary. Each letter in the secret is guessed in turn by guessing a string where the sum of its letter frequencies is high. This allows the secret character to be narrowed down with each guess until a single letter remains. Then, the next letter is guessed using the same frequencies. When all the letters have been guessed, the word is known.

To build the strategy that works with frequencies, a binary tree called an `AlphaTree` is constructed. For instance, here is the `AlphaTree` for the characters in "abcbdecc":



Notice that the nodes of the tree consist of a set of letters together with an integer that contains the sum of their frequencies. Also note that each parent node is formed from two children by taking the union of their letters and the sum of their frequencies.

The *tree building algorithm* to build such a tree is as follows:

1. The dictionary is used to construct an `AlphaFreq` structure, which contains a mapping from each alphabetical character to its frequency.
2. Each letter and its associated frequency in the `AlphaFreq` is used to construct a new singleton `AlphaTree` that is added to a priority queue of type `AlphaTreeQueue`. The queue is ordered so that the trees with least frequency are extracted first. If two trees have the same frequencies, the one most recently added is first.
3. While the queue contains more than one tree, the first two trees are removed from the queue and a new parent is created with them as children, and is added into the queue.
4. When the queue contains exactly one tree, this is returned as the desired frequency tree.

Testing

You can play a game of bangman by executing the `bangman.Bangman` class. This is an interactive game you can play in the console. You will also see various bots implemented in the `bangman.player` package, such as `PlayerCheat`, `PlayerBruteAlpha`, and `PlayerBruteFreq`. The `PlayerBruteFreq` class makes use of the classes you need to implement.

You are furthermore provided with a set of test classes under:

- `~/javafinaltestpartb_username/test/alphatree`

There is a test class `TestQuestion`i`.java` where i corresponds to questions 1, 2, 3, and 4.

These tests are not exhaustive and are merely intended to guide you. You should think carefully about whether your solution is complete, even if you pass all of the given tests.

Overview

The skeleton files are provided in the `bangman` package and the `alphatree` package. These are located at:

- `~/javainterimtestpartb_username/src/bangman`
- `~/javainterimtestpartb_username/src/alphatree`

You may only modify the following files, or those you create yourself:

- `alphatree/AlphaFreq.java`, where you will implement a class that associates frequencies to the letters of the alphabet.
- `alphatree/AlphaTree.java`, where you will implement a class that represents binary trees whose nodes contain sets of characters and their frequencies.
- `alphatree/AlphaTreeQueue.java`, where you will implement a class that stores a priority queue of trees.
- `alphatree/Utils.java`, where you will implement the tree building algorithm.

You must not change type signatures of existing methods or constructors of these classes. You may add any attributes and methods as desired, as well as any new classes you may require.

Tasks

1. **Implement the `AlphaTree` class**

   The `AlphaTree` class represents trees that store sets of characters and their frequencies in nodes. You may implement this class using any collection types from `java.util`. You will have to implement:

   - `AlphaTree(...)`: constructors for empty, singleton, and binary nodes.
   - `isEmpty()`: return `true` only when the tree is empty.
   - `isSingleton()`: return `true` only when the tree contains one node.
   - `left()` and `right()`: return the left and right subtree if they exist, otherwise `null`.
   - `chars()`: return the set of characters in the current node.
   - `freq()`: return the frequency of the current node.

   [10 marks]

2. **Implement the `AlphaFreq` class**

   The `AlphaFreq` class is a structure that associates a frequency to each of the 26 (lowercase) characters, one for each of the letters in the alphabet. You must implement this from first principles where any new attributes you define must not be collections from `java.util`. You will have to implement:

   - `AlphaFreq()`: constructs a structure where the frequency of every letter is `0`.
   - `AlphaFreq(List<Character> cs)`: constructs a structure where the frequency of every letter is the number of occurrences it has in the given list `cs`.
   - `isEmpty()`: return `true` only when the frequencies for all the characters are `0`.
   - `size()`: return the sum of all the frequencies.
   - `get(char c)`: return the frequency corresponding to the character `c`.
   - `add(char c)`: increment the frequency corresponding to the character `c`.
   - `reset()`: reset the frequencies of all characters to `0`.

   Note that not all values of type `char` are lowercase alphabetical letters, and these methods should be well defined even for bad input.

   [10 marks]

3. **Implement the `AlphaTreeQueue` class**

   The `AlphaTreeQueue` class is a priority queue of trees. You must implement this from first principles where any new attributes you define must not be collections from `java.util`. You will have to implement:

   - `AlphaTreeQueue()`: constructs an empty priority queue.
   - `AlphaTreeQueue(AlphaFreq freqs)`: constructs a priority queue that contains singleton `AlphaTree` values with the given frequencies.
   - `add(AlphaTree t)`: adds the tree `t` to the priority queue when the tree is not empty.
   - `addAll(AlphaFreq freqs)`: creates a singleton tree for each letter in `freq` with its assoicated frequency, and adds it to the queue.
   - `peek()`: returns the tree with lowest frequency from the queue, but does not remove it from the queue.
   - `poll()`: removes and returns the tree with lowest frequency from the queue.
   - `size()`: returns the number of trees stored in the queue.

   If two trees have the same frequency, the tree that was added last is returned first.

   [10 marks]

4. **Build an `AlphaTree` from an `AlphaTreeQueue`**

   The `alphatree.Utils` class contains `newAlphaTree(List<Character> chars)`, a method which is used to construct an `AlphaTree`, using the *tree building algorithm* described in the Problem Description. You should implement this method.

   [10 marks]

5. **Implement the `AlphaTreeQueueCoarse` and `AlphaTreeQueueFine` classes**

   Implement a new class in the `alphatree` package called `AlphaTreeQueueCoarse`, which extends the `AlphaTreeQueue` class to work in a concurrent environment using a coarse-grained locking strategy. Also do this for a new class called `AlphaTreeQueueFine` that implements fine-grained locking.

   Explain, using at most 100 words in a comment at the top of each file, how your implementation works.

   [10 marks]