9 Direct Access Data Structures

Portfolio Problems

Finish the part 9.4 of Lab 9 that deals with stacks and queues.

9.1 Eliza

Our goal is to train our computer to be a mock psychiatrist, carrying on a conversation with a patient. The patient (the user) asks a series of questions. The computer-psychiatrist replies to each question as follows. If the question starts with one of the following (key)words: Why, Who, How, Where, When, and What, the computer selects one of the three (or more) possible answers appropriate for that question. If the first word is none of these words the computer replies 'I do not know' or something like that.

1. Start by designing the class `Reply` that holds a keyword for a question, and an `ArrayList` of answers to a the question that starts with this keyword.

2. Design the method `randomAnswer` for the class `Reply` that produces one of the possible answers each time it is invoked. Make sure it works fine even if you add new answers to your database later. Make at least three answers to each question.

3. Design the class `Eliza` that contains an `ArrayList` of `Reply`s.

4. In the class `Eliza` design the helper method `firstWord` that consumes a `String` and produces the first word in the `String`.

   The following code reads the next input line from the user. You will need to find out what was the first word in the patient's question. Look up the documentation for the `String` class (and we gently hint that the methods `trim`, `toLowerCase`, and `startsWith` may be relevant).

   ```
   System.out.println("Type in a question: ");
   s = input.nextLine();
   ```

   Make sure your program works if the user uses all uppercase letters, all lower case letter, mixes them up, etc.

5. In the class `Eliza` design the method `answerQuestion` that consumes the question `String` and produces the (random) answer. If the first word of the question does not match any of the replies, produce an answer *Don't ask me that.* — or something similar. If no first word exists, i.e., the user either did not type any letters, or just hit the return, throw an `EndOfSessionException`.

   Of course, you need to define the class `EndOfSessionException`.

6. In the `Interactions` class design the method that repeats asking questions and providing answers until it catches the `EndOfSessionException` — at which time it ends the game.

9.2 Binary Search

Binary Search allows you to find quickly a piece of data in a sorted collection of data that can be accessed directly at a specific location. You check the item in the middle, if that is not the one you were looking for, you continue the search either in the upper half, or in the lower half — and recur till there you either succeed, or have nowhere else to look.

In the `Algorithms` class design the method `binarySearch` that consumes an `ArrayList<T>` that contains data sorted by using the given instance of a class that implements the `Comparator<T>` interface. The method also consumes another item of the type `<T>`, the item we are searching for.

The method produces the index in the `ArrayList<T>` where the given item has been found. If the item does not appear in the list, the method throws an `ItemNotFoundException`.

Of course, you need to define the class `ItemNotFoundException`.

Tests

Of course, you need to test your methods. Make a simple class of data, such as a `Book` or `Balloon` we have used in the past — or come up with something different — and define two different `Comparator`s for this class. Then make examples of lists of these data items and make sure your tests use both of the `Comparator`s.

Organize your tests so that the reader can readily see what is the purpose of each test and what data is used in computing the result and in providing the expected value.

9.3 Game Project

Your task for this week is to design the classes that represent the objects in your game as well as the whole game world.

You should also design all `draw` methods for these objects and verify that they display the various scenes in the game correctly.

We give you a sample of the code that shows side-by-side the applicative version of the world (our style till now when we produce a new instance of the world in response to either a key event or a tick), as well as the imperative version (where the sate of the world mutates in response to the key events and ticks).

The samples also show how to run visual tests of the drawings (and provide simple helper methods you can use to display the new `Canvas` for each scene you wish to show.

Finally, the samples show how to set up the test scenarios for the imperative games — this will be relevant next week when you add functionality to your game.