9 Direct Access Data Structures

Portfolio Problems

Finish Lab 9.

Working with the StringTokenizer

Set up a simple project, designing your solutions in the `Algorithms` class. Add to your project the class **Words.java** from the assignment web site.

1. Look up the `StringTokenizer` class in JavaDocs. The methods there allow you to traverse over a `String` and produce one word at a time delimited by the selected characters. Read the examples. Then write the method `makeWords` that consumes one `String` (that represents a sentence with several words, commas, and other *delimiters* and produces an `ArrayList<String>` of words (`String`s that contain only letters — we ignore the possibility of words like "don't"). The delimiters you should recognize are the comma, the semicolon, and the question mark.

2. The text in the`ArrayList<String> words` in the class `Words` is a secret encoding. It represents verses from a poem - if you read only the first words. Design the method `firstWord` that produces the first word from a given `String`. Use it to decode the poem.

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 the **Eliza** project by including the **Interactions.java** file and running the *Configuration* that has the class `Interactions` as its main class.

   The program types a prompt, waits for you to type something in and prints back what you typed.

2. Design the class `ReplyToQuestion` that contains a keyword for a question, and an `ArrayList` of answers to the question that starts with this keyword.

   For example the answers to `"why"` could be `"I don't know why."`, `"Why not!"`, and `"Just because."`.

3. Design the method `randomAnswer` for the class `ReplyToQuestion` 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.

4. Design the class `Eliza` that contains an `ArrayList` of several instances of the class `ReplyToQuestion`s — one for each of the question keywords we recognize.

5. In the class `Eliza` design the helper method `firstWord` that consumes a `String` that represents the patient's question and produces the first word in the `String`. The goal is 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).

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

   So, if the patient's question is any of the following:

   Why do you think so?

   Why, when I ask, you do not answer?

   WHY

   why am I so shy?

   the method will report that the first word was `"why"`

6. In the class `Eliza` design the method `findAnswer` 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.

7. Now add to the method `eliza` in the class `Interactions` the code that uses your class `Eliza` to produce replies to the questions the patient types in.

9.2 Insertion Sort

We have seen the recursively defined *insertion sort* algorithm both in the first semester and also recently, using the recursively defined lists in Java.

The main idea behind the insertion sort was that each new item has been inserted into the already sorted list. We can modify this as follows:

1. Design the method `sortedInsert` that consumes a sorted `ArrayList<T>` a `Comparator<T>` that has been used to define the sorted order for the given list, and an item of the type `T`. It modifies the given `ArrayList<T>` by adding the given item to the `ArrayList<T>`, preserving the ordering.

   Note: Be careful to make sure it works correctly when the given `ArrayList` is empty, and when the item is inserted at the end of the `ArrayList`.

2. Design the method `insertionSort` that consumes an arbitrary (unsorted) `ArrayList<T>` and a `Comparator<T>` and produces a new sorted `ArrayList<T>` as follows:

   It starts with an empty sorted list and inserts into it one by one all the elements of the given `ArrayList<T>`.

   Note: It is a bit more difficult to define the insertion sort algorithm so that it mutates the existing `ArrayList` *in place*.

3. **Extra Credit: Required for Honors Students**

   Design an in-place `insertionSort` method. You will get the credit only if the design is neat and clearly organized.

9.3 Simple Imperative Game

One of the popular early computer games had *Mario* running over moving obstacles, jumping up to avoid them, falling down slowly with the force of gravity.

Design a simple imperative *Mario*-like game using the *idraw* library. Follow the design outlined here:

1. Use a *queue* to represent the current obstacles - just rectangles (maybe of a random color). On each tick, remove the first from the queue and add a new one to the end of the queue, to generate the effect of moving obstacles.

2. *Mario* stays in one place in the middle of the screen. He falls down some small distance at each tick and moves up some larger distance in response to each *up* arrow key press.

3. Design the method that determines if *Mario* has hit an obstacle. Notice that you only have to check one of the moving obstacles - the one at *Mario*'s location. This should be easy if you have used an `ArrayList` to represent the *queue*.

4. Stop the game after some number of hits.

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 state 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.