Mastery I — Data Structures (T. III/22–23)

Directions:

- This mastery examination starts at 2pm on Thursday June 1, 2023. You have until 4pm of the same day to complete the following *three* problems. Each problem has been broken down into multiple goals. Each goal will be graded on an all-or-nothing basis by a machine grader. No collaboration of any kind whatsoever is permitted during the exam.
- You are expected to use an IDE. Your code will be compiled and tested using Java 11 and Gradle 7.6.
- **WHAT IS PERMITTED:** The exam is open- book, notes, Internet, Google, Stack Overflow, etc. The Internet can only be used in *read-only* mode.
- **WHAT IS *NOT* PERMITTED:** Communication/collaboration of any kind. Obviously, asking a question online is strictly *not* allowed. Using an AI-powered bot is *not* allowed.
- We're providing a starter package. The fact that you're reading this PDF means you have successfully downloaded the starter pack.
- Your Java code must *not* be in any package. That is to say, your files must not contain the statement `package ....`.
- To hand in your work, zip all the relevant `.java` files as `m1handin.zip` and upload it to Canvas before it closes.
- Based on your answers and other (random) factors, you may be called in for an interview to explain your code and write some code on the spot.

Problem 1: Triple Vowels (10 points)

The letters `a`, `e`, `i`, `o`, and `u` are vowels in English. In this problem, you'll implement a function that detects the presence of three consecutive vowels—that is to say, three consecutive letters that are vowels.

Inside a public class `TripleVowels`, write a static method

```
public static boolean hasTripleVowels(String st)
```

that takes as input a `String` and returns a boolean value indicating whether the input string contains three *consecutive* letters that are vowels. The input string may have both upper- and lower- cased letters. For example:

```
hasTripleVowels("OoO") // => true
hasTripleVowels("baZaa") // => false
hasTripleVowels("fooA") // => true
hasTripleVowels("moraiene") // => true
```

Goals: There are 4 goals:

1. The method correctly detects the condition when the occurrence of triple vowels is neither at the start nor the end—and the input string contains only lower-cased letters.
2. The method correctly detects the condition when the occurrence of triple vowels is neither at the start nor the end—and the input string contains mixed-case letters.
3. The method correctly detects the condition when the input contains only lower-cased letters.
4. The method correctly detects the condition on all possible inputs.

Problem 2: Enhancing `SLList` (10 points)

The starter code for this problem contains a rudimentary implementation of the singly-linked list (`SLList<T>`). The code uses a front sentinel. You will work to enhance it in this problem while retaining the front sentinel.

Your Task: You will implement the following methods in the `SLList` class:

- `public T[] toArray()` returns an array of all the elements of the list in list order. The length of the array must be exactly the size of the list (it doesn't include the sentinel node). This method should take time proportional to the length of the list.
- `public void deleteIf(Predicate<T> p)` takes as an input parameter a predicate `p` and deletes all items in the list for which the predicate `p` returns true. Other than deleting items matching the criteria, everything else must remain untouched. This method should take time proportional to the length of the list.

  The predicate `p`, expressed using the `Predicate` interface, represents a higher-order function that takes in a type `T` parameter and returns a boolean (i.e., true or false). This is similar to other higher-order function (HoF) code we have done in the past. More specifically to this task, the interface has a method `public boolean test(T t)`. Hence, to test if an element `x` satisfies the predicate, we will look at the outcome of `p.test(x)`. This means inside your `deleteIf` method, you can call `p.test(x)`, where `x` is of type `T`, and it will return true or false indicating the outcome of the predicate.
- `public SLList<T> reversed()` returns a new `SLList<T>` that is the reverse of this list. For example, reversing `[3, 2, 7]` will give us `[7, 2, 3]`. This method should take time proportional to the length of the list.

Here are some examples:

```
class BanA implements Predicate<String> {
   public boolean test(String t) { return t.equals("a"); }
}
SLList<String> list1 = new SLList<>(List.of("J", "a", "v", "a", "S", "E"));
SLList<String> list2 = new SLList<>(List.of("J", "a", "v", "a", "S", "E"));

list1.deleteIf((String e) -> e.equals(e.toUpperCase()));
// list1 should become [a, v, a]
list2.deleteIf(new BanA());
// list2 should become [J, v, S, E]
SLList<String> list3 = list2.reversed();
// list3 is [E, S, v, J]
```

Ground Rules:

1. The only file you can modify here (aside from creating/writing tests) is `SLList.java`.
2. You are free to modify the `Node` class inside `SLList`.
3. You must *not* replicate the entire list using another collection. This means that you cannot, for example, make a temporary `ArrayList` or `LinkedList`. In other words, your logic must work directly on the `SLList`.

Grading: There are 4 goals:

1. The code has some legit implementation and compiles clean.
2. Correctly implement the constructors and `toArray` as measured by what `.toArray` returns.
3. Correctly implement `deleteIf` as measured by what `.toArray` returns.
4. Correctly implement `reversed` as measured by what `.toArray` returns.

Problem 3: Row-by-Row Iterator (10 points)

Given a two-dimensional structure (example below), a row-by-row iterator (aka. natural Pusheen walker) yields elements from the first row from left to right, then proceed to the second row, so on so forth. As an example, the 2d structure on the left results in the sequence on the right. Observe that the empty rows—which can appear anywhere, including at the start and/or the end—are skipped.

```
List<List<Integer>> twoD = List.of(
    List.of(3, 7),
    List.of(),
    List.of(1),
    List.of(5, 0, 2)
)
```

⇒ 3, 7, 1, 5, 0, 2

You will implement a public class

```
public class PusheenWalker<T> implements Iterable<T> { ... }
```

with the following specifications:

- The only constructor takes as input a `List<List<T>>` representing the said 2D structure. As an example, this lets us write `new PusheenWalker<>(twoD)` and the result has type `PusheenWalker<Integer>`.
- The iterator given by this class—as described earlier—yields elements from left to right, then onto the following row, skipping each empty row, until all the elements have been returned. There may be many consecutive empty rows.

Ground Rules: Your code cannot store a collection of any kind, aside from a reference to the input structure. This means, for example, that you *cannot* make an output list ahead of time and simply return from that list. Moreover, do *not* use Java streams.

Grading: There are 5 goals:

1. The code has some legit implementation and compiles clean.
2. The iterator works correctly on inputs involving just one non-empty row of data.
3. The iterator works correctly on inputs involving multiple rows, each with the same non-zero length.
4. The iterator works correctly on inputs involving multiple rows of varying non-empty lengths
5. The iterator works correctly on all possible inputs.