Data Structures (CS 102), Professor Yap – Fall 2016

HOMEWORK

September 30, 2016

Homework 3: Due on Friday Oct 7, 2016

-
- This homework is worth 100+5 Points. In each homework, you get the extra 5 points by following all our instructions.
- READ CAREFULLY the homework instructions and policy in `http://cs.nyu.edu/~yap/wiki/class/index.php/DataStruc/Hw-page`. In particular, if your name is Yap, the zip file is named `hw3_Yap.zip` and it contains only one folder named `hw3_Yap` which has these four files: `README`, `Makefile`, `hw3_Yap.pdf`, `src` (a folder). The `Makefile` should be modified from our Makefile in hw2.

PART A: WRITTEN ASSIGNMENT (30 Points)

Question A.1 (3+3+3 Points) Java

(a) In p.82, it talks about a String literal and a String variable. In analogy to this, can also speak of **double literal** and **double variable**. Please give examples of a double literal and double variable, and discuss their differences. How would you describe those tokens that represent double literals?

(b) In parsing the command line arguments, we often write something like this:

```
int n = Integer.parseInt(args[0]); // first argument
```

Explain each aspect of this construction (e.g., what is `Integer` and `parseInt`?)

(c) Continuing the previous question: suppose the second command line argument is is a `double`. What would the analogous Java code look like?

```
double d = Double.parseDouble(args[0]); // 2nd argument
```

Question A.2 (3+4 Points) Java String Patterns

Read up on pattern matching in Java (e.g., our Programming Page in Wiki (click →JavaLinks, →Java FAQs, →12. Java Strings and Pattern Matching).

(a) Which of the following words match the pattern "[^a].[bc]+"?

(1) abc
(2) 1bbbbbbbb
(3) abbbbbbbb
(4) 1zc
(5) abcbcbcbc
(6) 1c
(7) asccbbbbcbcccc

(b) Write a pattern that will match each of the words in list A, and not match any words in list B:

A: pit, spot, spate, slaptwo, respite

B: pt, Pot, peat, part

Question A.3 (14 Points) ArrayStringLog

In Chapter 2, we read about the `ArrayStringLog` class that has these methods:

(a) insert()

(b) isFull()

(c) size()

(d) contains()

(e) clear()

(f) getName()

(g) toString()

If an instance of `ArrayStringLog` has N strings stored in it, what is the big-Oh complexity of each of these operations?

PART B: PROGRAMMING ASSIGNMENT (70 Points)

All needed files are found in `hw3_Yap.zip` from Piazza/Resources. They must be included in your `src` folder but unmodified. Our provided Makefile should also work without any changes.

Question B.1 (30 Points) Extending ArrayStringLog

For this problem, we provide an interface called `MyStringLogInterface`. It is a variant of the `StringLogInterface` found in p.74 of the text. The class `ArrayStringLog` (p.78 and following) is an implementation of `StringLogInterface`. Write a class called `MyArrayStringLog` to implements the `StringLogInterface`. Your implementation should imitate `ArrayStringLog` in the text unless we specify otherwise here. Specifically,

(a) You need to implement a new method called `resize()`. Here is how `resize()` works: if the array is not full, do nothing; otherwise, it will create a new array with double the current array size, and copy the contents of the old array into the new array.

(b) You must modify the implementation of `insert(String)` (p.80) so that it always call `resize()` before inserting a new item. Therefore, the precondition of `insert` will always hold, and it will succeed in the insertion.

(c) Finally modify the implementation of `toString()` (p.87). so that it now takes an `int m` argument. This argument says that you should only print the first `m` strings in the Log, and print the last `m` strings in the Log. (NOTE: this only makes sense if `m` is less that half of the number of strings in the Log, but we do not care to enforce this.)

You must put all the java files of this problem in a package called `stringLog`, which is also the name of the folder under `src`. Write a main method in `MyArrayStringLog` class which initially calls the default constructor of MyArrayStringLog; this makes the `maxSize` of the string array is 20. Then, insert a sequence of n random strings into an instance of `ArrayStringLog`. Here n is the optional first command line argument (default is $n = 100$). Finally, print the first m strings and the last m strings of the array to the terminal. Here m is the optional second command line argument (default is $m = 3$).

How to get "random strings" for insertion? First generate a random integer between 0 and n and convert it into a string. Your random number generator takes a seed, given by int variable s. This is the optional third command line argument (default is $s = 0$). As usual, when $ss = 0$, it means that we DO NOT give any seed to the random number generator.

NOTE: our Makefile is set up so that

≫ make run1

will run this program with the default arguments. But

≫ make run1 s=111 m=7 n=500

will insert 500 random strings into the `MyArrayStringLog` object, using 111 as random seed. Finally, it prints the first 7 strings and the last 7 strings in the array.

Question B.2 (40 Points) Zoombinis!!!

There is a popular children's computer game[1] from 1996 called *Logical Journey of the Zoombinis*. For more information, see Lecture 4 (or the wikipedia entry http://en.wikipedia.org/wiki/Zoombinis#Logical_Journey_of_the_Zoombinis. But you don't need it for doing this problem.

Each Zoombini has a name – our goal is to generate random names for Zoombinis. Here is a sample list of names that our program has generated:

Kejuka, He, Mojejiwijehefa, Mibojafudufe, Felule, Zasefacaga,
Rezuragobi, Cepi, Nidi, Ci, Borewi, Puxecaco,
Julenorazinicituzaxa, Voco, Zi, Laqezetaki, Habape, Fa.

Modes: Notice that each name is pronounceable: it is a sequence of syllables where each syllable consists of a consonant followed by a vowel. The number of syllables varies randomly. It is easy to produce such names, so we call this the **Simple mode** or **mode 0**. But they don't sound authentic at all. We would like to generate names that are recognizably ethnic (Chinese, Indian, Italian, etc). The variable `mm` is the mode indicator, and only the values 0 to 10 are allowed:

Simple `mm = 0`, American `mm = 1`, Chinese `mm = 2`, Hebrew `mm = 3`,

Indian `mm = 4`, Italian `mm = 5`, Japanese `mm = 6`, Korean `mm = 7`,

Spanish `mm = 8`, Hawaiian `mm = 9`, Other `mm = 10`.

You might observe some characteristics of names in these modes:

- American Zoombinis like short names (typically one or two syllables, and the last syllable should have consonant).
- Italian Zoombinis prefer a final vowel (like `Zoombini` or `Pelligrino`) and typically three or four syllables.
- Chinese Zoombinis should never more than three syllables, usually 2 syllable, and there is a preponderance of initial digraphs such as `Ch-`, `Sh-` and t-consonants like `-n` or `-ng`.

Syntax Files To generate random names in different modes, we use statistics. The statistical information for mode `mm` is stored in the folder `src/syntax-mm`. Inside this folder, we must have these 7 files:

consonants, digraphs, diphthongs, info, LDT, tconsonants, vowels The info is just some text file in which you explain your mode in more detail. We will explain the meaning of the other files next.

[1] Update for fans: the original creator of Zoombinis, TERC, came out with a version of the game for iPads and Android tablets in August 2015.

LDT: One statistical parameter is the number of syllables in a name: Americans like short (one or two syllables). Chinese and Koreans have almost exclusively two syllable names (not counting last name), but Italian and Indian names may prefer three or four syllables. We approximate this information using an array of `int`'s. For instance, consider this array of length 10:

LDT: (1 2 2 2 3 3 3 4 4 5)

If we pick a random number N from this array, we see that N has 30% chance of being 2 or 3, 20% chance of being 4, and 10% chance of being 1 or 5. After getting a random N from this array, we can generate a name with N syllables! The length of this array is arbitrary; their entries do not have to be sorted, though it is easier to understand when sorted. If you want Chinese or Korean names, you might use LDT:(1 2 2 2 2 3). If you want a higher probability of getting a name with one syllable (a trend in Chinese names) then you insert another 1 to get use LDT:(1 1 2 2 2 2 3). Call such an array a **Length Distribution Table** (or LDT) stored in an file named LDT.

Generalized vowels and consonants. Naturally, we need a list of vowels and one of consonants; these are stored in the files **vowels** and **consonants**. You might think that there are just 5 vowels `(a e i o u)` and 21 consonants; indeed, in mode 0, that is just what are stored in these files. In other modes, some vowels or consonants might be omitted. Just as in LDT, a vowel or consonant can be duplicated as many times as you like, to increase their probability.

So far, vowels or consonants are single letters. But they can be generalized to a pair of letters, called a **digraph**. A digraph consisting of two vowels is called a **diphthong**: E.g., `ai`, `oi`, `ia`, `oe`, and `oo`. Since there is no special name for a pair of consonants, therefore we will exclusively use the term "digraph" to refer to a "consonant digraph". E.g., `tr`, `sh`, `st`, `ch`, and `gl`. Therefore we need files named `diphthongs` and `digraphs` to store such lists (as usual, with as much duplication as desired to skew the probability distribution).

A **generalized vowel** means either a vowel or diphthong, and a **generalized consonant** means either a consonant or a digraph. Unfortunately, the world is more complicated: there are trigraphs such as `thr` or `chl`. For simplicity, we put trigraphs in the `digraphs` file. Letters like `y` (and sometimes `u`) seems to be able to function as vowel as well as consonant. You can throw these in both categories if you like.

Terminal Consonants and Initial Vowels. We will define a **syllable** as a generalized consonant followed by a generalized vowel. This is clearly a simplification since a syllable could end in a consonant. We have sing-let but is it `tab-let` or `ta-blet`)? For simplicity, we will only consider the probabilities of the **terminal consonant** and an **initial consonant**, i.e., a consonant at the end of a name, or a consonant at the beginning of a name. Thus, the name `Obama` has neither a terminal consonant nor an initial consonant. These probabilities are captured by two `int` arrays of length 11 (corresponding to the 11 modes we allow):

static int[ ] terminalProbability = { 5, 8, 7, 7, 7, 2, 3, 7, 5, 6, 5};
static int[ ] initialProbability = { 5, 8, 9, 4, 4, 4, 9, 4, 4, 4, 3};

For instance, the fact that `terminalProbability[5]=2` says that Italian names (mode 5), has only 20% chance of having terminal consonants. But American names (mode 1) has 80% chance. Similarly, `initialProbability[2]=9` says that Chinese names has 90% chance of an initial consonant (very unlikely to get `Obama` as a Chinese name).

Command Line Arguments for Zoombinis. We provide an abstract class in `zoombinis/ZoombinisAbstract.java` which must not be changed. Note the package name of `zoombinis` (small `z`). You are to extend this class in your file `zoombinis/Zoombinis.java` which has a main method to print random Zoombini names. The command line argument of

this program takes four optional arguments, in this order: integers `ss`, `nn` and `mm`, and string `pp`. Argument `ss` is the random seed (as in previous question), `nn` is the number of Zoombini names to generate, `mm` is the "mode" of your names, and `pp` is a relative path to some folder where data files may be found. The default values are `ss=0`, `nn=16`, `mm=0` and `pp="src"`. Note that this path is a relative path (relative to the location of the Makefile). The target `run2` in the provided Makefile will run the main method: E.g.,

≫ make run2
≫ make run2 ss=0
≫ make run2 nn=16 ss=0 pp=src mm=0

are all equivalent ways to call the program with the default values! They all print a list of 16 random Zoombini names in mode 0, using random seed 0. Of course, you are free to change any of these arguments. Study the provided Makefile to see how the argument $(args2) is constructed for this target.

What you need to implement and hand in: We will provide sample syntax files for the Simple mode (`mm=0`) and the American mode (`mm=1`). Your program should read these syntax files to produce names in these two modes. In addition, you are to write syntax files for ONE other mode of your choice. E.g., if you choose to do Spanish names (`mm=8`), be sure to provide the folder `src/syntax-8` with the needed 7 files. Please choose modes whose culture you understand well, so that you can be as realistic as possible. If you choose mode 10, you must explain this mode to us in the **Introductory Comments** section of your Zoombinis.java. We like to see interesting cultures!

Sample Outputs. The provided Makefile for hw3 should work if you set up your folders correctly: namely, your `src` folder should have these subfolders: `zoombinis`, `stringLog`, and at least three syntax folders. We have targets for automatic testing called test1, test2, etc (read the comments in Makefile). The outputs of these tests are output-test1, output-test2, etc. Try to emulate our output.