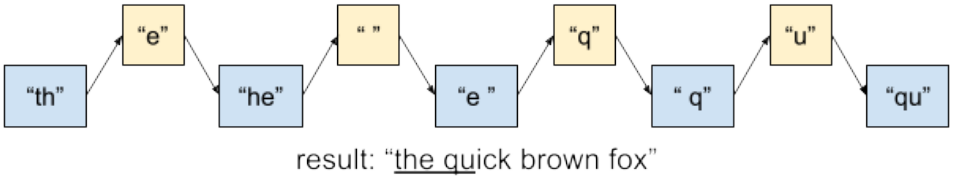
P08 Stacks, Queues, and Mr. Markov

Overview

Whether you realize it or not, Natural Language Processing (NLP) is quickly becoming more prevalent in our lives – you probably have used ChatGPT at some point in the last year.[1] The inner workings of the Large Language Models that power ChatGPT is out of scope for this class, but generating text with a basic algorithm is certainly not!

In this project we will make use of a nifty model called the *order-k Markov model*. It has a fancy name but its function is pretty simple: it reads in a piece of text and generates similar text.



This version of the model uses two-character chunks to predict the next letter which should appear. The steps taken in the model so far create the underlined part of the output, one letter at a time.

Learning Objectives

The goals of this assignment include:

- **Describe** the functionality of stack/queue data structures, and **explain** their operational details and complexities when implemented using a linked list
- **Predict** edge case situations for stack and queue usage and verify that your implementation handles them correctly
- Use Java documentation to **implement** code that uses a HashMap data structure to manage a series of stacks
- **Implement** a simple Markov Chain Model for text generation using your stack and queue data structures

[1] No judgement – the tool is value-neutral, but the question is how you use it. If you've been using it a lot, well, you tell *us* how that in-person exam went.

Grading Rubric

5 points	**Pre-Assignment Quiz:** Accessible through Canvas until **11:59PM CT on Sunday, April 14**.
+2.5 points	**Bonus Points:** Students whose **final** submission to Gradescope is before **4:59PM on Wednesday, April 17** *and* who pass ALL immediate tests will receive an additional 2.5 points toward this assignment, up to a maximum total of **50** points.
15 points	**Immediate Automated Tests:** accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
20 points	**Additional Automated Tests:** these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.
10 points	**Manual Grading:** Human graders will review the commenting, style, and organization of your final submission. You will NOT be able to resubmit corrections for extra points, and should therefore carefully review the readability of your code with respect to the CS300 Course Style Guide.
50 points	**MAXIMUM Total Score**

Assignment Requirements and Notes

(Please read carefully!)

Pair Programming and Use of External Libraries and Sources

- Pair programming is **NOT ALLOWED** for this assignment.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- All other sources must be cited explicitly in your program comments, in accordance with the Appropriate Academic Conduct guidelines.
- Any use of **ChatGPT** or other large language models (LLM) must be cited AND your submission MUST include a file called `log.txt` containing the full transcript of your usage of the tool. Failure to cite or include your logs is considered academic misconduct and will be handled accordingly.

- You are only allowed to **import** or **use** the libraries listed below in their respective files **only**.
- **The ONLY external libraries** you may use in your submitted files are:
  - MyStack and MyQueue: `java.util.ArrayList`, `java.util.Collections`
  - MarkovTester: `java.util.ArrayList`
  - MarkovModel: `java.util.HashMap`
  - MarkovTestGenerator: any `java.io` libraries to facilitate file IO

P08 Assignment Requirements

- Any helper methods you write must be private.

CS300 Assignment Requirements

This section is VALID for ALL the CS300 assignments

- If you need assistance, please check the list of our Resources.
- You MUST NOT add any additional fields either instance or static, and any public methods either static or instance to your program, other than those defined in this write-up.
- You CAN define local variables (declared inside a method's body) that you may need to implement the methods defined in this program.
- You CAN define **private** methods to help implement the different public methods defined in this program, if needed.
- Your assignment submission must conform to the CS300 Course Style Guide. Please review ALL the commenting, naming, and style requirements.
  - Every submitted file MUST contain a complete file header, with accordance to the CS300 Course Style Guide.
  - All your classes MUST have a javadoc-style class header.
  - All implemented methods including the main method MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.
  - Indentation should be 2 spaces and NOT 4 spaces.
  - The maximum width line should be 100.
- If starter code files to download are provided, be sure to remove the comments including the `TODO` tags from your last submission to gradescope.

- Avoid submitting code which does not compile. Make sure that ALL of your submitted files ALWAYS compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- Run your program locally before you submit to Gradescope. If it doesn't work on your computer, it will not work on Gradescope.
- You are responsible for maintaining secure back-ups of your progress as you work. The OneDrive and GoogleDrive accounts associated with your UW NetID are often convenient and secure places to store such backups. Aspiring students may try their hands at version control.
- Be sure to submit your code (work in progress) of this assignment on Gradescope both early and often. This will 1) give you time before the deadline to fix any defects that are detected by the tests, 2) provide you with an additional backup of your work, and 3) help track your progress through the implementation of the assignment. These tests are designed to detect and provide feedback about only very specific kinds of defects. **It is your responsibility to implement additional testing to verify that the rest of your code is functioning in accordance with this write-up.**
- You can submit your **work in progress (incomplete work) multiple times** on gradescope. Your submission may include methods not implemented or with partial implementation or with a default return statement.

1 Getting Started

1.1 Create the Project

1. Create a new project in Eclipse, and name it something like "P08 Text Generation".
   a. Ensure this project uses Java 17. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **NOT** create a project-specific package; use the default package.
2. Download the following Java source files from the P08 assignment page on Canvas:
   - `QueueADT.java`
   - `StackADT.java`
   - `LinkedNode.java`
   - `MarkovTextGenerator.java`
   - `sampleText.txt`
3. Create four (4) Java source files within your project's src folder:

- `MarkovModel.java`
- `MarkovTester.java`
- `MyQueue.java`
- `MyStack.java`

1.2 Provided Code: LinkedNode

Familiarize yourself with the `LinkedNode` class, which conforms to the description in the javadocs. This is a singly-linked node (unlike P07, where you had references to both the previous AND next nodes), but otherwise it should look very familiar.

The objects you create from this class will form the foundation of both your stack and your queue in the next step, so make sure you know how it works.

2 Data Structures

The primary component of this assignment is data structure creation. Once you've completed these successfully, we'll have some fun using them to generate text, but in order to get to that part you've got to create the data structures correctly first.

2.1 Expected Data Structure Behavior: MarkovTester

BEFORE you begin the data structure implementation, open up the `MarkovTester` file and add in tester methods corresponding to the data structure you'll be implementing. Each of the data structures for this program will be **generic**, meaning you can write your tester class to test a MyStack or MyQueue containing *any* type of value.

We recommend a basic object like a wrapper class object (Integer, Double, etc) or a String, for simplicity.

All tester methods are static boolean methods which expect NO parameters. All methods specified here must be **public**; if you write additional methods, they must be **private**.

- Both stacks and queues have a `getList()` method, which returns the current state of the stack/queue from top to bottom or front to back as an ArrayList. This will be useful for verifying that everything not at an end of the data structure is in the correct position!
- There are SIX (6) tests described in this section; make sure your tester class implements ALL of them before you start adding new tests.

Testing Stacks

For testing your stack behavior, you MUST implement the following methods:

- `testStackAdd` – verify that adding things to the stack correctly increases the stack's size, and that the ordering of all elements is correct.
- `testStackRemove` – verify that removing things from the stack (after adding them) correctly *decreases* the stack's size, and that the ordering of all remaining elements is correct. Additionally, verify that a stack that has had elements added to it can become empty again later.
- `testStackShuffle` – verify that calling `shuffle()` on the stack results in a stack that still contains all of the same elements, but in **any** order that is different from the original order.

Testing Queues

For testing your queue, you MUST implement the following methods:

- `testQueueAdd` – verify that adding things to the queue correctly increases the queue's size, and that the ordering of all elements is correct.
- `testQueueRemove` – verify that removing things from the queue (after adding them) correctly *decreases* the queue's size, and that the ordering of all remaining elements is correct. This test should also verify that the custom method `maintainSize(int)` works as described.

Common Tests

As both stacks and queues have a `peek()` method, we'll combine the tests for that into one tester method:

- `testPeek` – verify that calling `peek()` on both a stack and a queue returns the correct element AND does not make any modifications to the data structure.

2.2 Stacks and Queues

Now it's time to implement the stack and queue data structures. We'll call these MyQueue and MyStack, since they've got some nonstandard methods outside of just those defined in their abstract data types.

- Implement the MyStack and MyQueue classes as outlined in the javadocs.
- Make sure you verify that you are implementing the correct interfaces!
- Remember both data structures are *generic*. There must be no restrictions on the type of value stored in either data structure.
- The ADT interfaces and javadocs contain the descriptions of the standard methods that ALL data structures of these types should have; be aware that you are also required to implement the additional methods in the MyStack/MyQueue javadocs.

3 Markov Model

If your stacks and queues pass all of your tests, you are now ready to build the actual text-generating MarkovModel. Make sure that your data structures pass those tests first, otherwise this step will be MUCH more difficult.

- Two of the model's parameters, `windowWidth` and `currentQueue`, store the length of the substring we'll use to generate text (more on that later) and the queue used to keep track of the current substring value. (Note `windowWidth` is just a more descriptive name for the k from the Markov Model definition.)
  - That substring should always have length `windowWidth`, hence the `maintainSize()` method in MyQueue.
- The third parameter in the model uses a *HashMap*. A HashMap is a data structure similar to a Python dictionary, which stores mappings of **keys** to **values**. Simply put, a *key* is a string that has an associated *value*, like {"Name" : "Paul Atreides"}.
  - W3Schools has a useful resource for HashMaps; take a few minutes to familiarize yourself with it before proceeding.
- The last parameter, `shuffleStacks`, is a boolean value indicating whether the stacks should be shuffled between generating characters for the text. If it's false, you'll just use the learned values in the reverse order that you saw them in the text; if it's true, well, we'll shake things up a little, I guess...

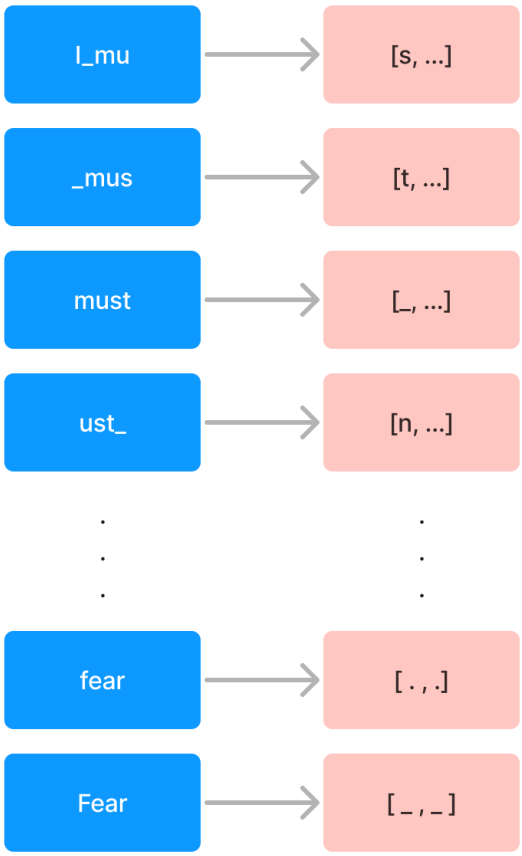
3.1 Implementing the Model

This model has two phases: **training** and **running**. In the *training* phase, you'll create the mapping of substrings of a given length to a stack containing each of the characters which follow that substring in the training data. Then when you run that model, you'll generate text using a queue of characters that always has the same length, and get the next character to add from the stack that it maps to.

- Start with `processText()`:
  - Go through the input String and find *each* substring of length k (`windowWidth`). For example, the string `"hello"` contains 4 substrings of length 2: he, el, ll, lo.
  - **This is your key**.
  - The character *following* that substring is your predicted value – this will be added to the stack containing predicted values for the given substring.

– To illustrate: the figure below shows the state of the model (with its keys and associated stacks of characters) after a few steps of processing the following line of text from Frank Herbert's *Dune* with k=4. Spaces are indicated with an underscore:

```
I must not fear.  Fear is the mind-killer.  Fear is the little-death that
brings total obliteration.  I will face my fear.
```



– Use the provided line below to add the key-value pair to your HashMap model. **Note: you will need to update this line with your OWN values for <HashMap>, <substring>, and <next char>.**

```
// Updates OR creates a new stack for the
// given substring and adds next char to it.
<HashMap>.computeIfAbsent(<substring>, k -> new MyStack<>()).push(<next char>);
```

- Once you have a trained model, you can move to the other two methods.
- Recall that you can get data from the HashMap by calling its `get({key})` instance method. Make sure to replace {key} with an actual String value, though.

- You may be asking yourself, "Why am I using a queue to keep track of a substring?". It's simply because we need to incorporate queues into this lab, don't worry about it :)

3.2 Markov Text Generator

This class is provided for you in its entirety. It will use everything you have written up to this point to generate text!

- If you want to change the input file, you must do so by changing the value of the `filePath` instance variable.
- If you run it with `sampleText.txt` as input, $k = 2$, and textLength $= 50$, you'll see some garbled text like:

  ```
  "This painnintandlessly reveresometwirld is is thers,"
  ```

  This *resembles* English, but not that well. Increase k to 4, what happens? Why? Feel free to deduce this yourself or ask a TA. Change the input file to something longer!

  - We've provided a .txt of Jane Austen's *Pride and Prejudice*, but you can add longer .txt files that you're interested in (see section 3.3 below).
- Feel free to explore this file to learn more about how text is actually generated. Also, **play around with the parameter values,** `k` **and** `textLength`.
- You'll notice that as k is lowered, the text becomes more garbled and chaotic, as k increases, it becomes increasingly more like the original text. In order to generate fun, new text we have to find the sweet-spot value for k.

3.3 Next Steps

- Now you're ready to have some fun, if you choose to. One of the great things about this lab is generating new text that is specifically interesting to you, not just the instructors. You can download lots of full books (as .txt files) from Project Gutenberg. You can also write your own .txt files. All you have to do is load them into Eclipse and update the file path appropriately.
- Note: while we can't provide you the files directly due to copyright laws, no one is checking to see if you're generating new lyrics from your favorite artist, using old State of the Union addresses from here, or searching a bookname/screenplay/etc with ".txt" into Google. If you're getting text from a website directly, just copy and paste it into a .txt file using whicever text editor you're comfortable with.

4 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope.The only files that you must submit are:

- `MarkovModel.java`
- `MarkovTester.java`
- `MyQueue.java`
- `MyStack.java`

Additionally, if you used any generative AI at any point during your development, you must include the full transcript of your interaction in a file called

- `log.txt`

Your score for this assignment will be based on your **"active"** submission made prior to the assignment due date of Due: **9:59PM CT Thu April 18**. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.

Students whose final submission is made before 5pm on the Wednesday before the due date and who pass all immediate tests will receive an additional 5% bonus toward this assignment. Submissions made after this time are NOT eligible for this bonus, but you may continue to make submissions until 10:00PM CT on the due date with no penalty.