Programming Assignment 2: Mini-Git [LinkedRepository]

Specification

Background

Version control systems are software features or programs designed to track changes to documents or sets of documents over time. In most systems, each time changes are made to the documents being tracked, a new *version* or *revision* is logged. Usually some additional information, called *metadata*, is also tracked along with each revision. This metadata can include a timestamp for when the changes were made, one or more authors of the changes, comments or notes about the changes, and/or many other types of information. Version control systems also typically provide a way to review the *history* of the documents being tracked, along with operations to revert to previous points in history if necessary. The history tracking features of Google Docs are an example of a version control system.

Version control systems that are designed specifically for tracking source code for computer programs are often called *source control systems* and may include additional features useful for tracking source code. These features may include associating certain types of files with particular programming languages or running automated tests each time a new revision is created. One popular source control system in wide use today is Git, which was developed by Linus Torvalds (who also created the Linux operating system) and initially released in 2005.

In this assignment, we will implement our own, simplified version of a version control system similar to Git, using linked lists.

> **NOTE:** Version control systems typically need to address at least two significant problems: how to track and manage the metadata for the revisions that make up the version history, and how to represent and track the actual changes to the documents themselves. We will focus only on the first problem (tracking metadata and history); for more information on how Git handles tracking the changes, see the free, online book Pro Git.

System Structure

In our system, as in Git, a set of documents and their histories are referred to as a *repository.* Each revision within a repository is referred to as a *commit.* You will implement a class called `LinkedRepository` that supports a subset of the operations supported by real Git repositories. (We will not be dealing with features such as branching or remote repositories. We will assume histories are fairly linear and mostly take place in a single, local repository.)

We will represent commits with following provided class. **You must not modify this class in any way.**

▶ Expand

Each commit consists of a unique identifier, a message describing the changes, the general time the commit was made, and a reference to the immediately previous commit. In our representation, identifiers will be strings.

> **NOTE:** You may see some code you're unfamiliar with (namely the `SimpleDateFormat` and `Date` classes), but that's okay! You are not required to understand these, just know that `SimpleDateFormat` and `Date` allow us to print out the current date. Feel free to explore these classes or ask the course staff if you'd like to learn more about them!

As we eventually did with our `LinkedIntList` and `ListNode` classes, we will implement the `Commit` class as a public static inner class within the `LinkedRepository` class. (Ideally, we would make this class private, but we leave it public for ease of testing.)

Notice that the `id` and `message` fields of the `Commit` class are all `final`, meaning that you will not be able to modify them. If you attempt to change the value of these fields after they have been initialized, you will get a compiler error such as the following:

`error: cannot assign a value to final variable message`

Required Operations

Your `LinkedRepository` class must include the following methods:

```
public LinkedRepository(String name)
```

- Create a new, empty repository with the specified name
  - If the name is null or empty, throw an `IllegalArgumentException`

```
public String getRepoHead()
```

- Return the ID of the current head of this repository.
  - If the head is `null`, return `null`

```
public int getRepoSize()
```

- Return the number of commits in the repository

```
public String toString()
```

- Return a string representation of this repository in the following format:

- `<name> - Current head: <head>`
    - `<head>` should be the result of calling `toString()` on the head commit.
- If there are no commits in this repository, instead return `<name> - No commits`

```
public boolean contains(String targetId)
```

- Return true if the commit with ID `targetId` is in the repository, false if not.
- Throws an `IllegalArgumentException` if `targetId` is `null`
- Note that all elements are unique. Therefore, it should not continue looping unnecessarily once the element of interest is found.

```
public String getHistory(int n)
```

- Return a string consisting of the String representations of the most recent `n` commits in this repository, with the most recent first. Commits should be separated by a newline (`\n`) character.
    - If there are fewer than `n` commits in this repository, return them all.
    - If there are no commits in this repository, return the empty string.
    - If n is non-positive, throw an `IllegalArgumentException`.

```
public String commit(String message)
```

- Create a new commit with the given message, add it to this repository.
    - The new commit should become the new head of this repository, preserving the history behind it.
- Throws an `IllegalArgumentException` if `message` is `null`
- Return the ID of the new commit.

```
public boolean drop(String targetId)
```

- Remove the commit with ID `targetId` from this repository, maintaining the rest of the history.
- Throws an `IllegalArgumentException` if `targetId` is `null`
- Returns `true` if the commit was successfully dropped, and `false` if there is no commit that matches the given ID in the repository.
- Note that all elements are unique. Therefore, it should not continue looping unnecessarily once the element of interest is found.

```
public void synchronize(LinkedRepository other)
```

- Takes all the commits in the `other` repository and moves them into `this` repository, combining the two repository histories such that chronological order is preserved. That is, after executing this method, `this` repository should contain all commits that were from `this` and `other`, and the commits should be ordered in timestamp order from most recent to least recent.

- If the `other` repository is null, throw an `IllegalArgumentException`
- If the `other` repository is empty, `this` repository should remain unchanged.
- If `this` repository is empty, all commits in the `other` repository should be moved into `this` repository.
- At the end of this method's execution, `other` should be an empty repository in all cases.
- You should not construct any new `Commit` objects to implement this method. You may however create as many references as you like.

Run Time Requirement

- Unless otherwise stated, all methods must run in `O(n)` time where `n` is the size of the repository.
    - In the case of `synchronize`, the method should run in `O(n+m)` time where `n` is the size of this repository and `m` is the size of the other repository. In other words, the method should run in linear time.
- `getRepoHead`, `toString`, and `commit` must run in `O(1)` time.

`synchronize` Explained

▶ Expand

Client Program & Visualization

We have provided a client program that will allow you to test your `Repository` implementation by creating and manipulating repositories. The client program will directly call the methods you implement in your `Repository` class and will show you the resulting changes to the repositories. Click "Expand" below to see a sample execution of the client (user input is **bold and underlined**).

▶ Expand

In addition to this, you may (and are *encouraged to*) create your own client programs to test out your implementation on various cases. You may also modify the provided client if you find it helpful. However, **your `LinkedRepository` class must work with the provided client without modification and must meet all requirements above**. To better understand what is happening, you can reference these slides which visualize the repositories changing throughout the operations.

Testing

On this assignment, you are required to write **4 of your own test cases for synchronize** covering the different LinkedList cases we've introduced in Section 6 and 7: Front, Middle, Empty, and End. **Each of these test cases should be contained within their own method in your Testing class.** We've provided you two additional helper methods to help in this process, as well as the

`ExampleTesting.java` to give you an idea of how to use them. You are not required to write test cases for any of the other instance methods for your `LinkedRepository` implementation, but we'd encourage you do so to get more practice writing JUnit tests.

Each of the tests you write must include a `throws InterruptedException` in the method declaration (similar to how we write `throws FileNotFoundException` when doing File I/O) as we must temporarily interrupt execution via `Thread.sleep` to ensure individual commits have unique timestamps. Examples of this can be seen in ExampleTesting.java

> ⚠ **WARNING:** If you choose to not use the provided helper methods, you must make sure to call `Thread.sleep(1)` between each of your commits. This will ensure each individual Commit node has a unique timestamp

> ⚠ **WARNING:** We have provided a test case that simply tests to see if you've uploaded Testing.java and it fails no tests. It does not check whether or not you've met the testing requirements for this assignment.

Implementation Guidelines

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements and hints:

> ⚠ **WARNING: You must use an iterative approach to this assignment**. While recursion is a powerful tool that we'll explore later in the course, we're specifically assessing your ability to reason about LinkedLists and the cases they generate.

- The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.
- You should not construct any unnecessary `Commit` objects. Specifically, you should only construct a `Commit` object when an entirely new commit is being created. If commits are being removed or rearranged, you should manipulate the existing `Commit` objects. (You may create as many *references* to `Commit` objects as you like.)
  - You should only need to construct `Commit` objects in the `commit()` method.
- Your `LinkedRepository` class should have the following fields as specified below and they should be declared `private`. You are not allowed to have any other fields.
  - A reference to the head of the repository.
  - A field to keep track of the repository's name.
  - (Optional) A size field to keep track of the size of the repository.
- These methods can be quite challenging! It is valuable to take a look at resources from class, particularly the `LinkedIntList` pre-class work, in-class activities, and section problems. Notably, the `weave` problem in Section 7 will be a helpful starting point for implementing `synchronize`.

- **You should not modify the `id, message`, or `timeStamp`** fields directly. In particular if you run into the issue `error: cannot assign a value to final variable message`, it likely means that you are attempting to modify a `Commit` object's data, instead of rearranging the commits.
- Some notes on `synchronize`:
  - Note that you will have to compare the time stamps to determine which order they should appear in and that the `timeStamp` field is of type `long`. This is another primitive that you haven't seen before, but you can essentially treat it as an `int` when doing your comparisons. So, if you're trying to check if `commit1` is chronologically earlier than `commit2`, you can check if `commit1.timeStamp < commit2.timeStamp`.