Homework 10 - Linked List

Author: Andrew Chafos

Description and Purpose

In this homework assignment, you will be developing the Singly-Linked `LinkedList` data structure, which will implement the provided `List` and `Queue` interfaces. In doing so, you will write a variety of methods from each. You will also be creating your own generic `Node` class that your `LinkedList` will use. Both the provided interfaces and the classes you create will be written using generics such that they could work for any class parameter - you will be facing many of the challenges that the developers who wrote the `java.util.LinkedList` class had to confront, as that class uses generics, too!

Directions

We have provided `List.java` and `Queue.java`, which contain interfaces that you will be implementing in `LinkedList.java`. However, you will need to create 2 new files, `Node.java` and `LinkedList.java`, which are extensively described below.

Write a class called `Node` that has the following:

- Generic Type
  - This class should be based on some generic type with no restrictions
- Fields
  - A variable of the generic type that holds the data associated with this class
    - **You must name this variable `data`**
  - A variable of type `Node` (with the generic type attached on) that acts as a "next" pointer - represents the Node that is "next" in the list
    - **You must name this variable `next`**
  - **You should not have any other fields.** No "previous" pointer variable or other variables are necessary.
- Constructors
  - A constructor that takes in 2 arguments: the first being some `data` of the generic type, the other being a `Node` (with the generic type attached on) that acts as a `next` pointer as described above
    - This constructor should assign the fields accordingly
  - A constructor that takes in 1 argument: the data of the generic type
    - Should assign the data instance variable accordingly and should assign the "next" Node to `null`
    - Should use constructor chaining
- Methods
  - Getters for each variable
  - A setter for the `Node` variable

Write a class called `LinkedList` that has the following:

- Generic Type
  - This class should be based on a generic parameter with no restrictions
- Interfaces
  - This class should implement the provided `List<T>` and `Queue<T>` interfaces
- Fields
  - A variable of type `Node` (with the generic type attached on) that represents the **head** of this `LinkedList`. When the list is empty, this should be set to `null`, and should always contain the `Node` representing the first item of the list otherwise.
    - **You must name this variable `head`**

  - A variable of type `Node` (with the generic type attached on) that represents the **tail** of this `LinkedList`. When the list is empty, this should be set to `null`, and should always contain the `Node` representing the last item of the list otherwise.
  - **You should not have any other fields.** No size variable is necessary.
    - **You must name this variable `tail`**
- Constructors
  - A no-arg constructor that sets each variable equal to `null`
- Methods
  - This class should implement every method in the `List<T>` and `Queue<T>` interfaces.
    - All of the following should be `public`.
    - **Important**: The `Queue` and `List` methods are supposed to be entirely separate from each other. This is accomplished in practice by changing the *static* type to reflect the methods that should be used. For example:

```
List<String> myList = new LinkedList<>();
myList.add(0, "First");
myList.add(1, "Second");
myList.add(1, "In between 1 and 2");
System.out.println(myList.remove(2));
Queue<String> myQueue = new LinkedList<>();
myQueue.enqueue("First");
myQueue.enqueue("Second");
myQueue.enqueue("Third");
System.out.println(myQueue.dequeue());
```

      - Note in the above code how we *don't* mix and match `List` and `Queue` methods. For example, we don't call both `add` and `enqueue`; we pick one to treat our `LinkedList` as either a `List` or a `Queue`, not both.
  - `Queue<T>` methods:
    - `void enqueue(T data)`: adds `data` to the Queue (note our `LinkedList` is also a `Queue`!). Think about where in the list the data should be added in order to make the ordering compatible.
      - Should throw an `IllegalArgumentException` if `data` is null
    - `T dequeue()`: removes the oldest piece of `data` from our Queue; that is, the first thing we added, or the item that has been waiting the longest to be removed. Think about from where in the list we should remove to make this work.
      - Should throw an `IllegalArgumentException` if the list is empty
  - `List<T>` methods:
    - `int size()`: returns the number of items in this list
      - This is *not* a getter for a "size" variable - you should not have a "size" variable in your LinkedList class. Instead, you should iterate over the list, starting from the head, to determine its size.
    - `void add(int index, T data)`: adds the given data to the list at the specified index. For example, `add(0, "Hello")` for a `List<String>` would add `"Hello"` to the beginning of the list.
      - Should throw an `IllegalArgumentException` when `index < 0` or `index > size()` or `data` is null.
      - When the `LinkedList` has size 1, both the head and tail `Node`s should point to the only Node in the list.
    - `T remove(int index)`: removes the data at the given index from the list, and then returns it.
      - Should throw an `IllegalArgumentException` when `index < 0` or `index >= size()` or the list is empty.
    - `T get(int index)`: returns the data at the specified index in the list
      - Should throw an `IllegalArgumentException` when `index < 0` or `index >= size()` or

the list is empty

* `List<T> subList(int n)`: returns **a new List object** containing *the first* ***n*** *elements in this list.*
  - For example, if our list was `[1, 2, 3, 4, 5, 6]`, `subList(2)` should return a new list containing `[1, 2]`, and `subList(3)` should contain a new list containing `[1, 2, 3]`.
  - Note that if `n` is greater than or equal to the length of the list, you should *not simply return the current list.* Instead, you should add each item individually to the new list, and then return the new list.
  - Note also that `n = 0` is a valid case; this should just return a new, empty list.
  - This method should not change current list; its job is simply to copy over some number of elements from the current list to a new one.
  - Should throw an `IllegalArgumentException` if `n` is negative.

Testing

We have provided a testing file, `LinkedListTests.java`, which has a `main` method that tests very basic functionality of your methods. However, note that these tests are not comprehensive, so feel free to add your own tests!

The code in that file acts as the "sample input/output", so we will not repeat it here. If you reach the end of the `main` method without any tests failing, it means you have passed the *limited* provided tests we have given you - otherwise, it will stop the program and inform you of the expected behavior.

Note: Make sure your classes are all in separate files and that all of your files are in the same directory.

Allowed Imports

To prevent trivialization of the assignment, **no imports** are allowed for this homework.

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points. If you don't have checkstyle yet, download it from Canvas -> Files/Resources. Place it in the same folder as the files you want checkstyled. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must javadoc your code.

Run the following to only check your javadocs.

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both javadocs and checkstyle.

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or breaks our auto grader. For that reason, do not use any of the following in your final submission: * var (the reserved keyword) * System.exit * Runtime.getRuntime.halt * Runtime.getRuntime.exit

Collaboration

Collaboration Statement

To ensure that you acknowledge collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Recall that comments are special lines in Java that begin with `//`.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `Node.java`
- `LinkedList.java`

Make sure you see the message stating "HW09 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

1. Prevent upload mistakes (e.g. forgetting checkstyle, non-compiling code)
2. Provide usage validation (e.g. forbidden imports, reserved keywords, etc)

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files.
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for all official clarifications
