Homework 8: Linked List

Authors: Melanie Su, Sarang Desai, Allan Nguyen, Sumit Choudhury

Problem Description

We are finally going behind the scenes of that mysterious E in ArrayList<E>! Rather than creating a class with specific reference types for every instance variable, method parameter, method return type, etc., generics allow us to make our classes more flexible by allowing users to choose what objects to use within the class. Generics are frequently used in classes designed to store other objects, just like with ArrayLists.

In this homework, you will be creating your own Singly-Linked List data structure that exhibits behaviors defined by two different Abstract Data Types (ADTs), List and Queue. For simplicity, we have provided `List.java` and `Queue.java`, interfaces to ensure your LinkedList implements all the correct methods to have two functional data structures.

Solution Description

Node.java

This class will represent the structure that makes up each item in your LinkedList.

- This class should be based on some generic type with no restrictions
- Should ONLY have the following attributes:
  - `data` of generic type associated with this class
  - An attribute that will act as a pointer to the `next` instance of this class in the LinkedList
    - This type declaration should also have a generic type associated with it
- Should have a two-argument constructor that takes in some `data` of the generic type and the next instance in the list
- Should also have a one-argument constructor that takes in some `data` of the generic type and makes the next instance `null`
- Getters and Setters should be added as needed

LinkedList.java

This is a Singly-Linked list of Nodes. This class will implement the two provided interfaces, `List.java` and `Queue.java`.

- This class should be based on some generic type with no restrictions
- Should ONLY have the following attributes:
  - `Node` of generic type that will represent the `head` of this LinkedList
  - `Node` of generic type that will represent the `tail` of this LinkedList
- Should have a no-argument constructor that sets each attribute described above to `null`
- Methods
  - `toString()` method
    - This will be helpful for you to debug!
    - Returns all the data elements in the LinkedList in the string format:
      "[data 1] --> [data 2] --> [data 3]"
      ex: "[melanie] --> [sarang] --> [allan]"

    - If list is empty, return an empty string.
- `Queue.java`
    - `boolean isEmpty()`
        - Checks to see this Queue is empty
    - `void enqueue(T data)`
        - Adds `data` to the Queue. Think about what kind of object this data should reside in and where it should go to satisfy the FIFO (first in first out) characteristic of a Queue.
        - If `data` is `null`, throw an `NullPointerException` with a message “Data passed in is null.”
    - `T dequeue()`
        - Removes an object from the Queue and returns the data that object holds. Remember the FIFO characteristic of a Queue.
        - If the Queue is empty, throw a `NullPointerException` with a message “Empty queue.”
- `List.java`
    - `boolean isEmpty()`
        - Checks to see this List is empty
    - `int size()`
        - Returns the size of the List
        - Must be implemented by iterating through the nodes. Do not use a private size instance variable
    - `void add(int index, T data)`
        - Adds `data` to the List at the specified index
        - If the index is less than 0 or greater than `size()`, throw an `IllegalArgumentException` with a message “Invalid index!”
        - If the passed in data is `null`, throw a `NullPointerException` with a message “Data passed in is null.”
    - `void addToBack(T data)`
        - Adds `data` to the back of the List
        - If the passed in data is `null`, throw a `NullPointerException` with a message “Data passed in is null.”
    - `T remove(int index)`
        - Removes the object at the specified index and returns the data that object holds.
        - If the List is empty, throw an `IllegalArgumentException` with a message “Invalid index!”
    - `T get(int index)`
        - Retrieves the data from the specified index and returns it
        - If the List is empty or the index is less than 0 or greater than or equal `size()`, throw an `IllegalArgumentException` with a message “Invalid index!”

        - `T set(int index, T data)`
            - Replaces the data of the object at the specified index with the passed in `data` and returns the data that was previously at that index.
            - If the index is less than 0 or greater than or equal `size()`, throw an `IllegalArgumentException` with a message "Invalid index!"
            - If the passed in data is null, throw a `NullPointerException` with a message "Data passed in is null."
        - `boolean contains(T data)`
            - Checks to see if the List contains the passed in `data`
            - If the passed in data is `null`, throw a `NullPointerException` with a message "Null data cannot be in list"
        - `void clear()`
            - removes all elements from the List
        - `List<T> subOddList()`
            - Returns a new List object containing every odd element in this List
            - For example, if our List contains `["a", "b", "c", "d"]`, then `subOddList()` should return a new List containing `["b", "d"]`.
            - This method should not modify the original List in any way and the new List should not have overlapping elements with the original List
            - If the original List is empty, throw an `IllegalArgumentException` with a message "List is empty"
    - Things to keep in mind for Lists and Queues
        - If there is one object in your LinkedList, both the head and tail pointers should be pointing to the same object.
        - Think about all the possible edge cases when you add or remove from a LinkedList. What if you added to the head? What if you removed the tail?

Example Outputs

```
List<String> linkedlist= new LinkedList<>();
linkedlist.add(0, "Prerna");
linkedlist.add(1, "Sumit");
linkedlist.add(2, "Allan");
linkedlist.add(3, "Sarang");
linkedlist.add(4, "Stasko");
//list is: [Prerna] → [Sumit] → [Allan] → [Sarang]→ [Stasko]

System.out.println(linkedlist.remove(3));
//should print "Sarang"
//list is: [Prerna] → [Sumit] → [Allan] → [Stasko]

System.out.println(linkedlist.remove(1)); //prints "Sumit"
System.out.println(linkedlist.remove(0)); //prints "Prerna"
//list is: [Allan] → [Stasko]

int size = linkedlist.size(); //size should equal 2
linkedlist.add(2, "Buzz");
linkedlist.add(2, "Ramblin'");
//list is: [Allan] → [Stasko] → [Ramblin'] → [Buzz]
```

```
linkedlist.clear();
System.out.println(linkedlist.contains("Ramblin'")); //prints false
//list is: []

Queue<String> vaccineLine = new LinkedList<>();
vaccineLine.enqueue("Prerna");
vaccineLine.enqueue("Stasko");
vaccineLine.enqueue("Melanie");
vaccineLine.enqueue("Allan");
//Queue is: [Prerna] → [Stasko] → [Melanie] → [Allan]

System.out.println(vaccineLine.isEmpty()); //prints false
//Queue is: [Prerna] → [Stasko] → [Melanie] → [Allan]

System.out.println(vaccineLine.dequeue()); //prints "Prerna"
System.out.println(vaccineLine.dequeue()); //prints "Stasko"
//Queue is: [Melanie] → [Allan]
```

Submitting

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

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

Make sure you see the message stating, "Homework 8 submitted successfully." We will only grade your last submission. If you choose to create additional files, be sure to **submit *every file* each time you resubmit**.

Import Restrictions

To prevent trivialization of the assignment, you may not import anything.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Checkstyle and Javadoc

You must run Checkstyle on your submission. (To learn more about Checkstyle, examine our course's external website Java Resources page under "CS 1331 Style Guide".) The Checkstyle cap for this assignment is **25 points**. If you don't have Checkstyle yet, download it from Canvas → files → checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. 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
```

Collaboration

No collaboration is allowed on this assignment. See syllabus for more details.

In addition, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due. Only post code on Piazza in a **private** post.

Important Notes (Don't Skip)

- 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
- Make sure to test your program manually based on the assignment instructions and expected outputs
- The Gradescope autograder visible to you is **NOT** comprehensive.