Homework 09 – Linked List

Problem Description

For this homework, you will be implementing a LinkedList. The instructions for this assignment provide you with a framework for the methods to implement when creating **LinkedList.java**. The Node class to use is provided and will not be turned in. To complete this assignment, you will use your knowledge of lists and generics.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

`Provided Files`

`Node.java` – Each element of the LinkedList should be a Node with a data pointer to the next Node. The only elements in a Node should be `T data` and `Node<T> next`. **Do NOT change this class at all, and do NOT submit this file to Gradescope.**

`LinkedList.java`

Variables

Variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this! **Do not add any more instance variables**. You will lose points if you add any more.

- `Node<T> head`- this represents the first node in the LinkedList
- `int size` - this represents the number of nodes in the LinkedList

Constructors

- A constructor that takes in nothing that sets the `head` to null and the `size` to 0 to instantiate a `LinkedList.`

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions.

- `addAtIndex(T data, int index)`
  - Method that adds another Node at the specified index in the LinkedList
    - If the index is out of bounds (outside of [0, size]), then throw an `IllegalArgumentException` with an appropriate message
  - Does not return anything
- `removeFromIndex(int index)`
  - This method removes the Node at the index that is specified.
    - If the index is out of bounds (outside of [0, size - 1]), then throw an `IllegalArgumentException` with an appropriate message
  - Returns the data that was removed. (Note: the data, not the Node)

- `clear()`
  - This method will clear out the entire linked list
    - If the LinkedList is already clear, throw a NoSuchElementException with an appropriate message
  - Returns nothing
  - Hint: You can do this in 2 lines of code
- `get(int index)`
  - Find the data at the specified index in the LinkedList
    - If the index is out of bounds (outside of [0, size - 1]), then throw an `IllegalArgumentException` with an appropriate message
  - Returns the data that is in the specified index
- `isEmpty()`
  - This method will return true or false based on if the LinkedList is currently empty.
  - Returns a boolean indicating that the LinkedList is empty
  - Hint: You can do this with one line of code
- `toArrayList()`
  - Convert the LinkedList to an ArrayList
  - Traverse through your LinkedList and add all the data contained in each Node to an ArrayList
    - Data should appear in the same exact order and the ArrayList should be the same size
  - Returns the ArrayList that was created
    - The ArrayList should be parameterized so that it is an ArrayList<T>.
- `fizzBuzzLinkedList`()
  - Returns a new LinkedList that stores elements of type String, following a similar idea to the wellknown FizzBuzz coding problem
  - The returned LinkedList will be of the same size as the one this method is called on. Each element will be mapped into the new LinkedList as follows:
    - We define position as starting from 1. That is, the head of the linked list is at position 1, the element next to the head is a position 2, and a LinkedList of size s will have its last element at position s.
    - If the element is in a position that is multiple of 3 but not of 5, the element in the new LinkedList will be “Fizz”
    - If the element is in a position that is multiple of 5 but not of 3, the element in the new LinkedList will be “Buzz”
    - If the element is in a position that is both multiple of 3 and 5, the element in the new LinkedList will be “FizzBuzz”
    - Otherwise, the element in the new LinkedList will be the toString of the element in the original position, prefixed with “[# of position]: [toString]” (number colon space toString)
  - Example:
    - Original Linked List (showing toString of each element, LinkedList could be of any type): “A” -> “B” -> “C” -> … -> “O” -> “P”
    - New LinkedList (always parameterized with type String): “1: A” -> “2: B” -> “Fizz” -> “4: D” -> “Buzz” -> “Fizz” -> “7: G” -> “8: H” -> “Fizz” -> “Buzz” -> “11: K” -> “Fizz” -> “13: M” -> “14: N” -> “FizzBuzz” -> “16: P”

- **HINT:** Make sure that head, size, and the next pointers are updates when necessary. If head is not updated when it needs to be, data will be lost and the LinkedList will not be updated correctly!

These tests and the ones on Gradescope are by no means comprehensive so be sure to create your own!

Rubric

[100] `LinkedList.java`

- [5] `LinkedList()`
  - Constructs the LinkedList correctly
- [20] `addAtIndex(T data, int index)`
  - [5] `IllegalArgumentException` thrown when the index is out of bounds
  - [10] A Node containing the correct data is added to the LinkedList at the correct location. All data is preserved and in the correct order.
  - [5] `head` and `size` are changed appropriately
- [20] `removeFromIndex(int index)` ○ [5] `IllegalArgumentException` thrown when the index is out of bounds
  - [5] The data in the Node at the index is returned
  - [5] The Node at the index is removed. All other data must be preserved
  - [5] `head` and `size` are changed appropriately
- [15] `clear()`
  - [5] `NoSuchElementException` thrown when the List is already empty
  - [5] The LinkedList is cleared
  - [5] `head` and `size` are changed appropriately
- [5] `isEmpty()`
  - [5] Correctly returns whether the LinkedList is empty or not
- [8] `get(int index)`
  - [3] `IllegalArgumentException` thrown when the index is out of bounds
  - [5] The correct data of the Node at that index is returned
- [13] `toArrayList()`
  - [3] An ArrayList is returned
  - [10] All data is preserved and in the same order as the LinkedList. ArrayList is the same size.
- [14] `fizzBuzzLinkedList`()
  - [3] Works for LinkedList with less than 5 elements
  - [3] Works for LinkedList with less than 15 elements
  - [8] Works for LinkedList with more than 15 elements

We reserve the right to adjust the rubric, but this is typically only done for correcting mistakes.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import the following classes. You are *not* allowed to import any other classes or packages.

- `java.util.ArrayList`
- `java.util.NoSuchElementException`

Feature Restriction

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`

Javadoc

For this assignment, you will be commenting your code with Javadoc. Javadoc comments are a clean and useful way to document your code's functionality. For more information on what Javadoc comments are and why they are awesome, the online overview for them is extremely detailed and helpful.

You can generate the Javadoc overview for your code using the command below, which will put all the files into a folder named "javadoc". Note you should execute this after adding Javadoc comments to your code:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to include are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 *This class represents a Dog object.
 *@author George P.Burdell
 *@version 1.0
 */
 public class Dog {

    /**
     *Creates an awesome dog(NOT a dawg!)
     */      public Dog() {
     ...
     }

     /**
      *This method takes in two ints and returns their sum
      *@param a first number
      *@param b second number
      *@return sum of a and b
      */
      public int add(int a,int b) {
      ...
```

```
        }
}
```

A more thorough tutorial for Javadoc Comments can be found here.

Take note of a few things:

1. Javadoc comments begin with `/**` and end with `*/`.
2. Every class you write must be Javadoc'd and the @author and @verion tag included. The comments for a class should start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an @param tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

Checkstyle can check for Javadoc comments using the -a flag, as described in the next section.

Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style. Checkstyle can be downloaded here.

To run Checkstyle, put the jar file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **35 points**. This means that up to 35 points can be lost from Checkstyle errors.

Collaboration

Collaboration Statement

To ensure that you acknowledge a 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].

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation

- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- **approved**: "Hey, I'm really confused on how we are supposed to implement this part of the homework. What strategies/resources did you use to solve it?"
- **disapproved**: "Hey, it's 10:40 on Thursday... Can I see your code? I won't copy it directly I promise"

In addition to the above rules, 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.

Turn-In Procedure

Submission

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

- 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. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. 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:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

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 a note containing all official clarifications