Project 4: Recursive Linked Lists

A Recursive List Abstract Data Type.

Overview **1**

Learning Goals **2**

Project Specification: **2**

Project Code 2

Development Notes 3

Tests **4**

Style Guide **4**

Export and Submit **5**

Academic Honesty Policy Reminder **6**

Dev Tips **6**

Overview

Imagine you are an engineer of DroneDrop, a company that deploys drones to deliver life essentials in a post-apocalypse world, where everyone stays home because of biohazards outside. The drones are assigned to different districts, each with a pre-calculated route to visit all households. In order to provide fast and responsive customer services, the drone will update the route to skip those with no active order. Your goal is to provide a linear data structure that represents the route, and allows fast update, so we can beat our competitor. Luckily, some insider told us they use inefficient ArrayLists for the same job.

In this project, you will write a linked-based implementation for the List abstract data type, as specified in **`ListInterface.java`.** You'll be building a singly linked list that can act as a drone route. The requirement is all methods you implement **must** use recursion and you are **NOT** allowed to use any loop (such as `for`, `while`, `do-while`). You must also stay within Big-O runtime bounds so that the drones are efficient in delivering life essentials.

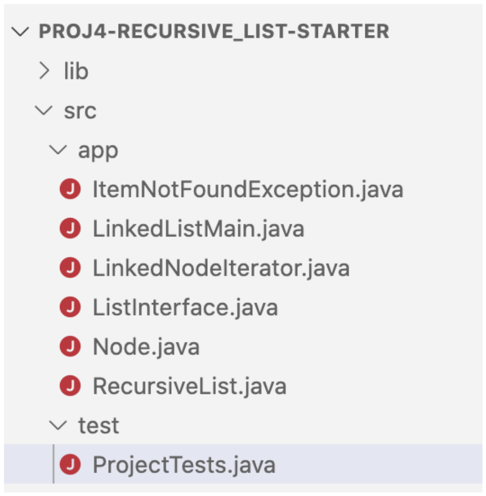
Learning Goals

- Show ability to write a generic class that implements a given interface, fulfilling the contract it specifies (including Big O() behavior and an `Iterable` implementation).
- Gain practice writing recursive methods and List ADT.
- Gain practice writing appropriate unit tests.

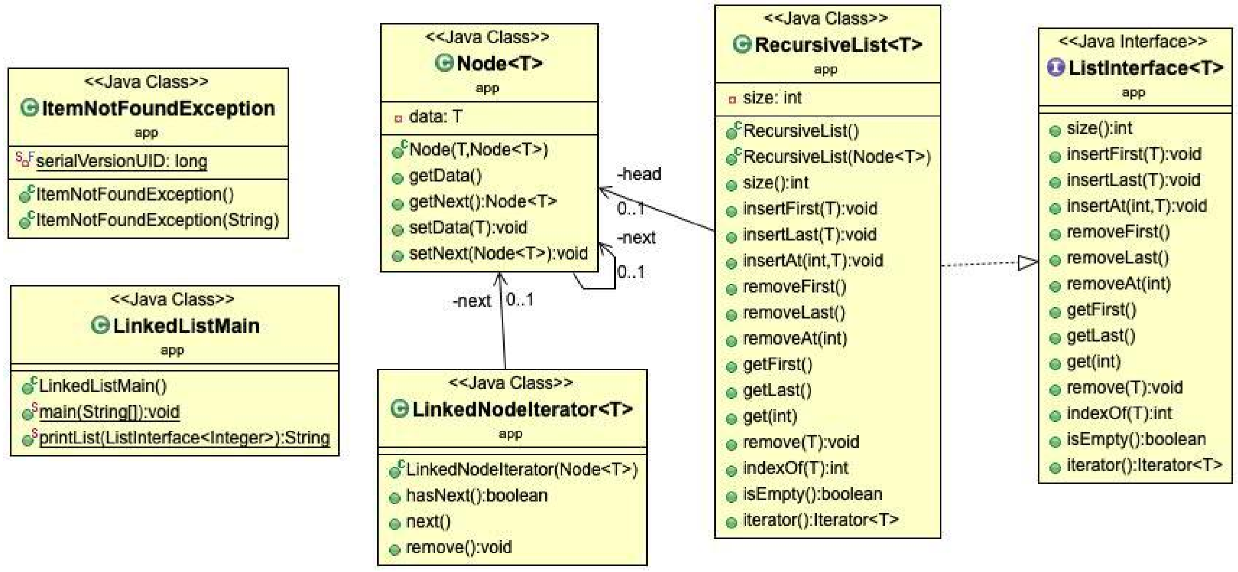
Project Specification:

Project Code

The starter code contains the following files. Note that **`LinkedListMain.java`** does not compile initially.


```
PROJ4-RECURSIVE_LIST-STARTER
  lib
  src
    app
      ItemNotFoundException.java
      LinkedListMain.java
      LinkedNodeIterator.java
      ListInterface.java
      Node.java
      RecursiveList.java
    test
      ProjectTests.java
```

The following diagram shows the file structure using Unified Modeling Language (UML) notation. It gives you an overview of the files and the parameters and return types of methods.



Development Notes

1. Figure out the order of ToDos

Your main task is to implement `RecursiveList.java`. You'll find a bunch of TODOs in the starter code. Spend some time tracing what should happen when a new `RecursiveList` is created in `LinkedListMain` and the `insertLast` method in `RecursiveList.java`. is called. You'll find it helpful to read the comments in `ListInterface.java`. You'll see that all the methods in the interface are implemented in `RecursiveList.java` (with `@Override`).

2. Use Recursion

In your implementation, you must use recursion and you may not use explicit iteration (i.e. for, while, or do while loops) of any kind. You will not receive any points if you submit an implementation using explicit iteration.

Hints: Recursion is all about reducing some problems slightly to make it easier to solve. For some methods, to implement them using recursion, you will find it necessary to create private, "helper" methods that are recursive, and then have your public methods call these helper methods to jump start the recursive call. You have to decide what arguments these helper methods need in order to implement them recursively.

Review zyBooks chapters 12.19 and 12.20 for an overview of recursion. You'll also find extra resources in Moodle for understanding how recursion works. In particular, see the Echo 360 links to 2 videos: "Optional: Recursion Intro" and "Optional: recursion on Lists". (Note: chapter names and numbers in the videos do not match chapters in the current zyBooks).

3. Use Helper Methods

Let's look at a specific example of a helper method for this project: take a look at the `indexOf` method – this public method is written in such a way that does not lend itself to recursion on a linked list node. So you should create a private helper method to use internally. Perhaps something like:

```
private int indexOfHelper(T elem, int index, Node<T> node)
```

We can then simply ask, does the current node contain the element you are looking for? If it does, we can return the current index. Otherwise, we recurse on the next node in the list, at the next index, with the same element to search for. Think about what the base cases are.

Another suggested helper method would be:

```
private final Node<T> findNode(int distance, Node<T> curr)
```

This method finds the node that is an (integer) distance nodes after curr. That is, if distance is 0, then curr is returned. Otherwise, recur down the list.

4. Stay within Big-O runtime bounds

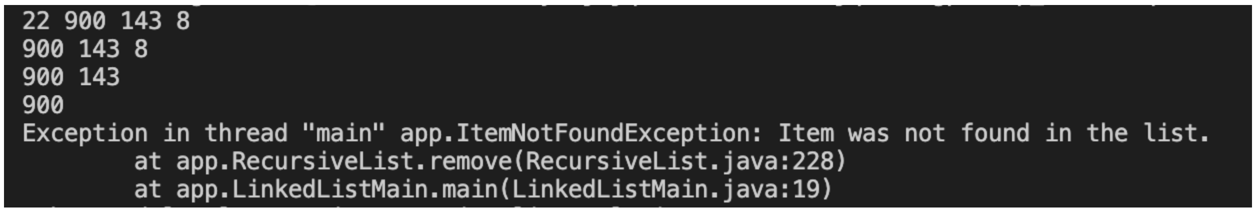
Be sure that your implementation complies with the required Big-O runtime bounds in the comments of each method as described in the `ListInterface` interface. Your code will not pass all of the autograder tests if your methods are not within these bounds. For e.g. the `size()` method must run in O(1) time.

5. Implement the Iterator method

The `ListInterface<T> extends Iterable<T>` (see API documentation). You will need to implement the `iterator()` method that returns an `Iterator<T>`. Your implementation of the `iterator()` method can simply create a new `Iterator<T>` each time it is called.

6. Throw Exceptions

The use of Exceptions is an important part of programming. They are the best way to signal to the caller that a method encountered an unexpected, or abnormal condition. Review zyBooks chapters 12.17 and 12.18 for Exception Basics and Exceptions with Methods.
You will see several kinds of Exceptions in this code. One example is the `remove` method (called in the main method with the sample input data). It throws several types of Exceptions including an `ItemNotFoundException`. This Exception class is part of the project code in the `app` package.



```
22 900 143 8
900 143 8
900 143
900
Exception in thread "main" app.ItemNotFoundException: Item was not found in the list.
        at app.RecursiveList.remove(RecursiveList.java:228)
        at app.LinkedListMain.main(LinkedListMain.java:19)
```

Tests

We have provided you with some public tests and some private tests in Gradescope. You will need to come up with some of your own. Try to think of all of the edge cases that could occur, and write tests to check for each of them. You will not be graded on your tests, but writing them and passing them is the only way that you can be reasonably sure that your code works.
Remember: Do not modify any existing instance variables, method signatures or any of the starter code provided. Do not import any code libraries.

Style Guide

Follow these style guidelines:

1. All unnecessary lines of code have been removed.
2. Proper indenting must be used.
3. Optimal use of blank lines and spaces for operators.

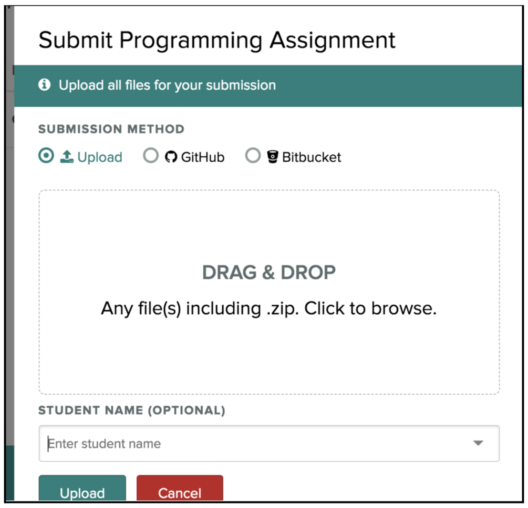
4. Use of camelCase for variable/parameter and method names.
5. Variables declared early (not within code), and initialized where appropriate and practical.
6. Descriptive variable and method names.

Export and Submit

Step 1: Export your project

Step 1: Export your project
Within VSCode click on the “*View > Command Palette…*” menu option. Then type into the Command Palette: “*Archive Folder*” and hit enter. This will produce a Zip file of your project folder. You can then upload that zip file to the corresponding project assignment in Gradescope. You can add the Archive extension to VSCode if you don’t have it.

Step 2: Submit the zip file to Gradescope
Log into Gradescope, select the assignment, and submit the zip file for grading.



Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

[✓ Upload] GitHub Bitbucket

DRAG & DROP

Any file(s) including .zip. Click to browse.

STUDENT NAME (OPTIONAL)

Enter student name

Upload Cancel

The autograder will run successfully only if you submit a **correctly formatted zip file**- it has to have the same **names and directory structure as shown on page 2**. The autograder will also not run if your code **does not compile**, or if you **import libraries** that were not specifically allowed in the instructions.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does.

Academic Honesty Policy Reminder

Copying partial or whole solutions, obtained from other students or elsewhere, is **academic dishonesty**. Do not share your code with your classmates, and do not use your classmates' code. Posting solutions to the project on public sites is not allowed. If you are confused about what constitutes academic dishonesty, you should re-read the course policies.

Dev Tips

- Use your debugger! It is very useful in finding exactly where your program stops doing what you want it to do.
- Debug your code in your development environment- that is what it is designed for. Do not use Gradescope as a way to develop your code- it will not be helpful.
- Start early! Projects are starting to get a little longer, so if you get stuck you need to make sure you have enough time to seek help.
- Seek help when you get stuck. We have office hours and Piazza chat specifically for you to ask questions when you need assistance. Use public posts as much as possible so we don't have to answer the same question multiple times. Upload your code in Gradescope for us to check - don't make private posts with snippets of code,.
- Submit to gradescope at least once, even if you aren't completely done. There is a huge difference between a 50% and a 0%. That said, aim for 100%.