Homework 10 - Linked Lists

Problem/Motivation

Welcome to the final homework of CS 1331! Congratulations on making it this far. Up to this point, we've covered a wide breadth of topics - how to use the console, how to create classes, how to use inheritance and polymorphism, generics, and much more.

In this homework, you will gain exposure to another field of computing: *the ways in which we store data.*

For this assignment, we will be storing data in what is known as a *Linked List.*

You will implement a `LinkedList` using the given class and interface we've provided for you. But why are `LinkedLists` necessary, you might ask?

The Linked List

Linked Lists are meant to be an implementation of the abstract idea of a list.

A *list* is known as an *abstract data type,* also known as an *ADT. ADT's* are nothing more than descriptions of a particular data structure - they tell us what the data structure can hold, and what operations we expect to be able to do, They do not provide *implementations* of these operations, nor do they tell us how data is physically held (do we use an array, a map, etc. . .). *ADT's* are almost like a *specific* type of interface, in this regard.

For example, a *list* is an *ADT* whose elements are ordered and can be duplicates, and new data can be added at an *index* within the *list.*

Note that we haven't yet stated *how* the data is being held, or how these addition operations are performed.

We therefore provide *concrete* implementations of *ADT's* - we write classes which implement these storage instructions and operations and then use these concrete classes to store and manipulate our data.

A `LinkedList` is one such implementation of the list idea. `LinkedLists` are composed of *Nodes* which hold one element of data and also a reference to the next Node in the list.

With these `Nodes`, it is possible to retrieve data at any index of the list, as well as add data, remove data, etc. . .

A diagram of this structure is embedded below.

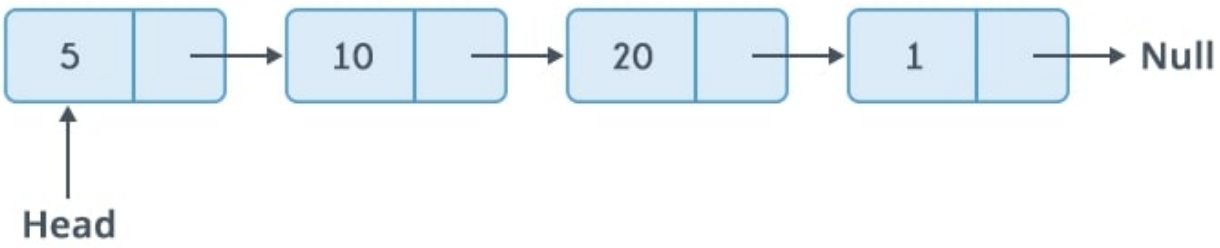

Figure 1: Singly Linked List with Head Reference

The i-th element/piece of data of the list is stored in the i-th node. We keep a reference to the **zero-th** `Node` called *head.*

Solution Description

Now, you must **create a new class** called `LinkedList` which implements the interface `SimpleList`. You **must** use the `Node` class we have given you in order to construct your `LinkedList`.

`LinkedList` should also be able to store elements of **any** reference type using the `Node` class.

Instance Fields

`LinkedList` must have two private instance fields:

- a `Node` called `head` which is **always** the zero-th `Node` in the `LinkedList.`
- an `int` called `size`, which keeps track of how many nodes there are in the List.

Feel free to add any other fields as you find helpful.

Note: please read the interface's and class' javadocs, etc... for more details.

Provided Files

We have provided two files - `SimpleList.java` and `Node.java` - an interface and class, respectively. Let's briefly go through these files.

SimpleList

1. `void addAtIndex(T data, int index)`: takes in an element of `T` type, and an index, and adds the data to the `LinkedList.` Data must be added in the form of a `Node` containing the data.
    - For `addAtIndex`, throw an `IllegalArgumentException` if the index is not valid (outside of [0,size] inclusive).
    - Add the data to the list at the specified index.
    - Before we call this method, consider a list with this data in this order: 0 -> 1 -> 2 -> 3. When we call list.addAtIndex(4, 2), we update our list like so: 0 -> 1 -> 4 -> 2 -> 3.
    - **NOTE:** Make sure that the list is preserved and no data is lost during the adding!
    - Be sure to update `head` and `size` when needed (for example, would adding at the back change `head`?).
2. `T get(int index)`: takes in an index, and returns the appropriate data. Also **swaps data of the i-th node with the data of the (i-1)(th) node.** Read below for more details.
    - If the list is empty, throw a `NoSuchElementException.`
    - Throw an `IllegalArgumentException` if the index is not valid (outside of [0, size-1] inclusive).
        - Please implement these checks in the order they have been presented. If a list is empty, there's no need to worry about an `IllegalArgumentException`, for example.
    - We want to do something very interesting with this method. When we get the data out of the *index-th* node, we want to swap the data of the *index-th* node with the data of the *(index-1)-th* node. That is, let's say there were three nodes (indexed as 0, 1, 2, where 0 is the `head`), and if I got the data located at index 1, I want 0's data and 1's data to be swapped. **Swap the data.**
    - Graphically, this is represented as so. If I have a list with data in this order: 0 -> 1 -> 2, list.get(1) will return 1, and the list is updated as such: 1 -> 0 -> 2.
    - An immediate implication of this *swapping* idea is that data that is more frequently accessed will end up closer to the beginning of the `LinkedList`, and may very well be located at the `head` of the list.
    - What happens if the data is at the `head`, however? Make sure to account for this case (no swaps are needed).
3. `boolean contains(T data)`: takes in an element of data, and returns `true` if the element is contained in the LinkedList, or `false` if it isn't.

4. `boolean isEmpty()`: Returns true the `LinkedList` is empty, false otherwise.
5. `T removeAtIndex(int index)`: takes in an index, and returns the data located at the *index-th* node. Fully removes *index-th* node from the `LinkedList.`
   - If the list is empty, throw a `NoSuchElementException`.
   - If the index is invalid, then throw an `IllegalArgumentException.`
     - Please implement these checks in the order they have been presented. If a list is empty, there's no need to check the index, for example.
   - Given an index that is valid ([0, `size - 1`]), get/save the data of the `Node`.
   - Assuming everything is valid, remove the *index-th* `Node` fully from the list. For example, if I had a list with data in this order: 0 -> 1 -> 2 -> 3, and I invoked `list.removeAtIndex(2)`, my final list should be: 0 -> 1 -> 3, and "2" should be returned.
   - Return the data that you saved.
   - **NOTE:** Make sure that the list is preserved after removal, and no additional data is lost!
   - **NOTE:** Make sure `head` and `size` are changed appropriately (if I remove the zero-th `Node`, for example, from the list, how is `head` affected? What does `head` point to now?).
6. `int size()`: returns the size of the `LinkedList`.
7. `T[] toArray()`: returns an array which is of the same length as the `LinkedList`, but only contains the data of the `LinkedList`, not its nodes. **NOTE:** Make sure the data in the array is in the same order as the `LinkedList`.

Node

`Node` is a generically typed class - it can hold data of **any** reference type. It contains these two fields:

- `data` - some data of type T.
- `next` - another `Node` of type T.

Both of these fields are private, so we've included getter and setter methods for you.

Allowed Imports

To prevent trivialization of the assignment, you can only import `java.util.NoSuchElementException`, as you'll need to throw this exception in certain situations.

No other imports are allowed. If you have questions about what to import or not import, feel free to make a post in Piazza.

Feature Restrictions

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

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

Rubric

[100] **LinkedList**

- [5] `LinkedList()` - the constructor for a `LinkedList.`
  - [5] Constructs the `LinkedList` correctly - `head` is set to `null`, and `size` is set to 0.

- [25] `addAtIndex(T data, int index)`
  - [5] `IllegalArgumentException` is thrown if the `index` provided is invalid.
  - [15] A `Node` containing `data` is added to the `LinkedList` at the correct location, and is now the *index-th* `Node` in the `LinkedList`. All data is preserved correctly after the add. **This is very important.**
    - Seriously, there's a reason this entire method is worth *25 points*. There's a few major cases to handle in this method. Think and code carefully.
  - [5] `head` and `size` are changed appropriately.
- [10] `get(int index)`
  - [1] If the list is empty, throw a `NoSuchElementException.`
  - [1] If `index` is not within bounds, throw an `IllegalArgumentException.`
    - Please implement the above checks in the same order they have been presented.
  - [2] Get/return the data at the *index-th* `Node`.
  - [4] **Successfully swaps data at indexed node with the data of the node right before.** (Handles all cases).
  - [2] All data is preserved; no data is lost.
- [10] `contains(T data)`
  - [5] Returns `true` if data is indeed in the `LinkedList`.
  - [5] Returns `false` if data is indeed **not** in the `LinkedList`.
- [5] `isEmpty()`
  - [5] **5 free points** is nice, right? Return `true` if true, `false` if false.
- [25] `removeAtIndex(int index)`
  - [2.5] If list is empty, then throw a `NoSuchElementException.`
  - [2.5] If `index` is invalid, then throw an `IllegalArgumentException.`
    - Please implement these checks in the order they have been presented.
  - [5] Data of the *index-th* `Node` is returned.
  - [10] The *index-th* `Node` is removed from the `LinkedList`. All data/`Nodes` (except for the removed `Node`) are preserved. **This is very important.**
  - [5] `size` and `head` are changed appropriately.
- [10] `size()` (the method)
  - [10] `size()` (the method), returns the size of the List.
- [10] `toArray()`
  - [3] An array of `T[]` is returned.
  - [3] The array is of the same size as the `LinkedList.`
  - [4] The array contains all the `data` of all the `Nodes` in the **same order** as the `LinkedList.`

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.

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points. Review the style guide and download the checkstyle jar. Run checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -a *.java
Audit done. Errors (potential points off):
0
```

Make sure to include the -a flag included for testing both checkstyle and javadocs

The message above means there were no Checkstyle or javadoc 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!

Depending on your editor, you might be able to change some settings to make it easier to write style-compliant code. See the customization tips page for more information.

Submission

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

- `LinkedList.java`

Make sure you see the message stating "HW## 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:

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

In other words, the test cases on Gradescope are not 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 and run your code; you can do that locally on your machine.

Other portions of your assignment are also autograded once the submission deadline has passed. The autograders used are often dependent on specific output formats, so **make sure that your submission passes all test cases marked "FORMAT:"**.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Ensure you pass all "FORMAT:" tests
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications