Homework 7

GWU CSCI 1112 - Spring 2020

March 12, 2020

1 Introduction

This homework is intended to reinforce your understanding of Linked Lists and Nodes. Your task is to implement a doubly circular linked list as well as some functions that will give that list a purpose within a Netfilx scroller context.

1.1 Deadline

March 27, 2020 at 11:59pm

1.2 Submission

You must create a `.zip` file containing all of the files that you develop and work with and you must submit only the `.zip` file to blackboard before the deadline.

Your `.zip` file must be named using the following naming convention: `<yourNetId>-hw-07.zip`
For example, my NetId is `jrt`. If I submitted the homework, I would name my `.zip` file: `jrt-hw-07.zip`

Do not submit the compiled `.class` files. You must submit any `.java` files and other supporting files.

1.3 Grading Rubric

- 40% For successful compilation
- 15% For sufficient comments
- 10% For consistent coding style
- 20% For base implementation
- 15% For unit testing

1.4 Unit Testing

In this homework, you continue to take responsibility for unit testing your implementation and you must implement your own unit test file modeled on those provided in the previous homeworks. Your unit tests will be evaluated in terms of comprehensiveness and correctness. We are also withholding a set of unit tests that will be used to independently evaluate your implementation.

Your class implementations must therefore fulfill the prescribed interfaces so that our unit tests can use your classes. You may add additional functions as needed; however, for each new function you add, you should provide a unit test and evaluate it in your test application. You must unit test all of the prescribed interfaces.

1.5 Comments

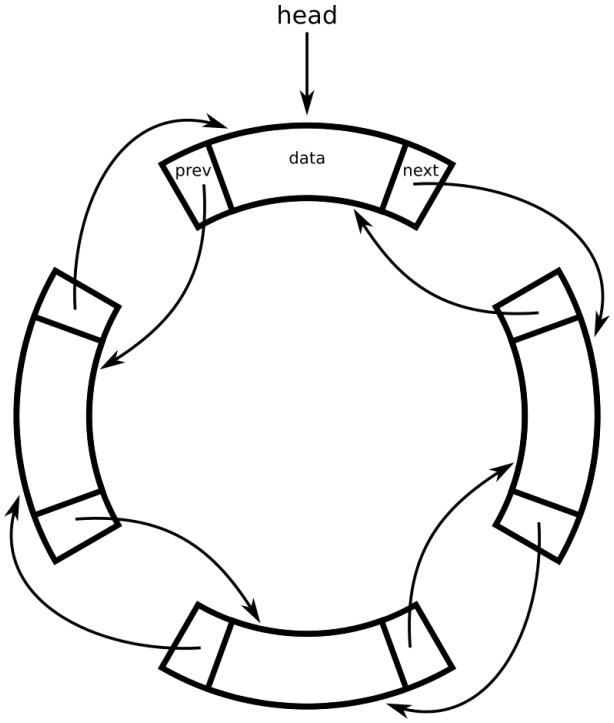
Again, the comments in the assignment are your responsibility. You must provide a file header for any files that you submit that documents the author and what is defined in the file. You must also document all functions that you implement with a function block header which must include information on what the function does, what the inputs to the function represent, and what the function returns. Finally, you must also provide relevant line comments inside any functions.

1.6 Plagarism

We will use a set of automated tools specifically designed to analyze code for plagarism. If you copy code from another source, classmate or website, there is a very high probability that these tools will flag your work as plagarized. You are permitted to discuss the problems at a high level; however, you must code your own solution. If you do not share code or outright borrow code from a website, you will have no problem with the plagarism filter.

2 Movie Catelog Navigation

In order to practice on Linked Lists, you will implement one from scratch which will allow you to examine how they work and explore their capabilities and limitations. You will build a doubly-circular linked list. In doubly linked lists, each node in the list has a pointer to the next node and the previous node. In circular linked lists, the tail node's next pointer refers to the head and the head node's previous pointer refers to the tail rather than null. For this linked list, you will not explicitly define a tail pointer; however, you should see that you can reach the tail simply by following the head's previous pointer.



For this homework, we will model navigating through a show catelog called ShowCat and base our system's behaviors on behaviors we see in similar existing catelog based services. A user might browse through a catelog moving back and forth between shows in a catelog. If the user scrolls far enough, the catelog returns back to the first show in the catelog. You should see that this catelog behavior closely resembles the list behavior described above. To track which show the user has currently selected in the catelog, your list will also track the "current" show which you will change

using simple forward and backward controls.

A framework for this project has been provided to you. Your implementation will run inside of the `UnitTests.java` class that you implement. All of your implementation will go inside of the `Catelog.java` class. The extension, as well as an implementation example, takes place in the class defined by `ShowCat.java`. In order to compile and test your work, compile and run `UnitTests`, or you can run the example and extension by compiling and running `ShowCat`.

2.1 Classes

The framework works consists 6 different classes: `Catelog`, `Data`, `Node`, `ShowCat`, `ShowFactory`, `UnitTests`. Classes `Data`, `Node`, and `Catelog` encapsulate all aspects of the linked list. Class `ShowFactory` is a factory class that generates sample data you may use with the list. Class `ShowCat` is a demo program that shows a typical use case of the list. Class `UnitTests` will contain all unit testing.

2.1.1 `Data`

This class is fully developed and encapsulates the data stored within a linked list node. It primarily maintains a string that represents show data. In a real application, this class may contain many more fields. It also provides methods that can be used when you must determine the lexicographical order of two data elements.

2.1.2 `Node`

This class represents a list element inside the linked list maintained by the `Catelog` class. A node is composed of a `Data` reference and references to the`next` and `prev` nodes.

2.1.3 `Catelog`

This class encapsulates a catelog of show data which uses a doubly-circular linked list to maintian the data in the catelog. This class contains the node `head` which is the start of the linked list and the reference that allows access to nodes in the list and maintains the reference to the list in memory. This class also contains another node `current` which represents the which show is currently selected in the catelog. `current` must always either be `null` or point to a node in the linked list. This class has a number of methods that must be implemented which include methods to manipulate the list, methods to view and validate the list, and methods to manage manipulation of the current show. You also may find it useful to add a number of helper methods that may simplify operations frequently used in other methods.

2.1.4 `ShowCat`

This class represents an example use case of the catelog class and an example of navigating through the list. You can run this class just like you run your unit test class. Running ShowCat will go through a quick example that does several linked list and catelog navigation activities.

2.1.5 `UnitTests`

This class is responsible testing the functions inside of your doubly-circular linked list and should explicitly test the methods that you implement in the `Catelog` class.

2.2 Base

You must implement the following `Catelog` class members using the following function signatures:

```
public boolean addToBack(String show)
public boolean addToFront(String show)
public boolean removeShow(String show)
public void clear()
```

```
public boolean isEmpty()
public String getCurrentShow()
public String stepForward()
public String stepBackward()
```

2.2.1 Base Functions

This section gives a general explanation of the requirements for each function. More information may be available in the function headers and in how the functions are unit tested.

```
public boolean addToBack(String show)
```

This function is responsible for placing a new show into a catelog. It places this show at the tail of the list. Note that you do not have a tail pointer for this list, but because it is circular and doubly-linked, the head can give quick access to the tail. If this show is already in the list, then return false. Any invalid inputs or failures must return false. This function must return true if the show is successfully added. Adding the first element to an empty list should set `current` to the `head`. If the list is non-empty, `current` must and can only refer to an element in the list or null.

```
public boolean addToFront(String show)
```

This function is responsible for placing a new show in your a catelog. It places this show at the front (1st element) of the list. If this show is already in the list, then return false. Any invalid inputs or failures must return false. This function must return true if the show is successfully added to the list. Adding the first element to an empty list should set the `current` to the `head`. `current` must and can only refer to an element in the list or null.

```
public boolean removeShow(String show)
```

This function attempts to remove a show by name from the linked list if the show exists in the list. If the show does not exist in the list or there are any other errors, this function must return false. If this function is successful in removing the specified show, it must return true. Reminder, if you are removing the "current" show, the current show must step forward. If you are removing the only show in the list, then both the head and current nodes must be null.

```
public void clear()
```

This function must remove all elements from the list and sets the current node to null.

```
public boolean isEmpty()
```

This function must indicate whether or not the list is empty. It must return false if the list is empty and must return true if the list is not empty.

```
public String getCurrentShow()
```

This function must return the show attached to the `current` node; otherwise, it must return null.

```
public String stepForward()
```

This function will change what the show that is referenced by `current` by advancing to the next show in the list. If `current` is not null, it must return the show attached to the `current` node; otherwise, it must return null.

```
public String stepBackward()
```

This function will change what the show that is referenced by `current` by moving to the previous show in the list. If `current` is not null, it must return the show attached to the `current` node; otherwise, it must return null.

2.3 Extension

Each extension is worth a total of 10% toward the cumulative semester total. 5% of the credit will be awarded for your implementation and 5% will be awarded for unit testing unless otherwise noted. You will receive no credit for an extension if you do not provide a set of unit tests for the implementation.

2.3.1 Extension 1 - Implement one of Selection, Bubble, or Insertion sort

Add sorts: 5% one of Selection, Bubble, or Insertion Sort 5% Quicksort 5% Mergesort 5% Unittests on each
You must implement one of the following functions and that function must use the prescribed function signature:

```
public static void sortSelection()
public static void sortBubble()
public static void sortInsertion()
```

Requirements

```
public static void sortSelection()
```

`sortSelection` must sort the linked list using the selection sort algorithm. The list must be sorted in ascending alphabetical order.

```
public static void sortBubble()
```

`sortBubble` must sort the linked list using the bubble sort algorithm. The list must be sorted in ascending alphabetical order.

```
public static void sortInsertion()
```

`sortInsertion` must sort the linked list using the insertion sort algorithm. The list must be sorted in ascending alphabetical order.

2.3.2 Extension 2 - Implement Quicksort

You must implement the following function and that function must use the prescribed function signature:

```
public static void sortQuick()
```

Requirements

```
public static void sortQuick()
```

`sortQuick` sorts the linked list using the quicksort algorithm. The list must be sorted in ascending alphabetical order.

Notes
You might be wondering "how the hell" do you quick sort on a linked list. The short and easy answer is that copying the list data to an array is less expensive in terms of computation time than the sort itself. This should put the data in a structure that you are comfortable with and can implement quick sort on without too much thought. Once you have finished the sort, you can then knit the list back together again as a list at less cost than the computational cost of sorting the list. This technique will also greatly help with the other two extensions. You should write two helper functions: `toArray` and `fromArray`. `toArray` is a private function in the `Catelog` class that reads through the list, puts all elements in the list into an array, and returns the array. `fromArray` is a private function in the `Catelog` class that reads through an array provided as a parameter and reconstructs the list for this catelog's instance from the array.

Yes, these hints are offered here after extension 1. If you get this far and realize that you labored over implementing the other sorts in the list rather than an array, then this is a pretty important lesson to read the document first before jumping in and writing the code. So "sorry, not sorry".

2.3.3 Extension 3 - Implement Mergesort

You must implement the following function and that function must use the prescribed function signature:

```
public static void sortMerge()
```

If you have reached this point, you should note that you could almost entirely reuse the unit tests from extension 1 to test extension 2 with only minor the change of calling the correct sort. You can probably do the same for this extension. So, in order to receive the credit for this extension, you must of course not only provide unit tests but you must also provide a detailed analysis of how merge sort works. You may not plagarize, so the analysis must be provided in your own words and using your own figures. Please submit this information in `.pdf` format named `extension3.pdf` in your code folder. For the algorithm, refer to https://en.wikipedia.org/wiki/Merge_sort.

Requirements

```
public static void sortMerge()
```

`sortMerge` sorts the linked list using the merge sort algorithm. The list must be sorted in ascending alphabetical order.