Homework 10

1 Introduction

The goal of this homework assignment is twofold: first, to give you an opportunity to practice writing a basic implementation of the linked list data structure, and second, to get your feet wet with graphical user interface (GUI) programming using the JavaFX libraries. For this homework, you will be required to implement an interface which defines some standard operations associated with linked lists and then use your implementation to complete the code for a GUI application that we have provided.

The GUI is just a simple application that allows the user to add red, blue, or green dots to a black screen by clicking on it and then later remove the dots by clicking on them. Note that you should complete the linked list portion of this assignment before moving on to the GUI part, as the GUI requires a functional Linked List.

2 Problem Description

You have been provided with the following files:

- LinkedList.java The interface that you will be implementing. All of the methods that you need to write for your linked list are defined here. Note that LinkedList is generic, and so any class that implements LinkedList should specify a type parameter.
- Dots.java The code for a JavaFX application which allows users to add and remove colorful Dots from a screen. Much of this file has been written for you; you only have to fill in the start method.

To get a full understanding of what is expected of you, be sure to read the Javadocs in the provided code.

3 Solution Description

You will be creating two new classes, SinglyLinkedList, which implements the LinkedList interface as well as a simple inner Node class to be used by the SinglyLinkedList. You will also be completing the start method in Dots.java. The comments in the provided files specify whats expected of you, so be sure to read them.

3.1 SinglyLinkedList and Node

SinglyLinkedList.java is an implementation of the LinkedList interface which uses nodes that are singly-linked which has a field called head which always points to the first node in the list and

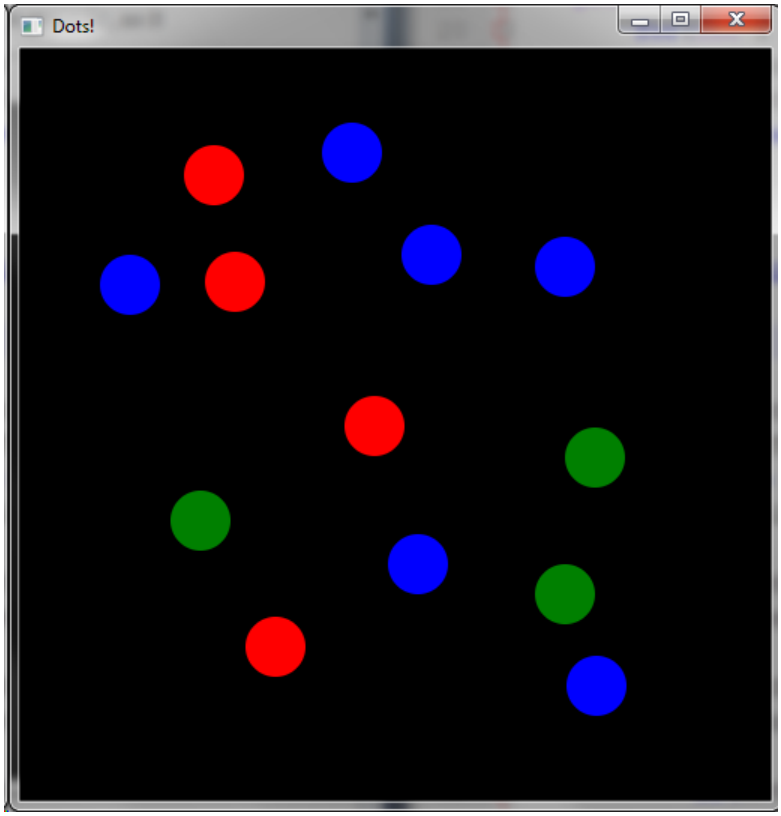
a field called tail which always points to the last node in the list. SinglyLinkedList needs to be generic, so that it can hold any type of object that someone using the linked list might need.

To successfully make a singly-linked list, you will have to define a private inner class called Node. Each instance of Node should hold a single piece of data (the data should be of the same generic type as the linked list itself), and should have a field called next which points to the next Node in the linked list. Nodes for this linked list should NOT have a previous field (hence the name SinglyLinkedList). Note that the Node class itself doesnt need to be generic, as it can just use the same type parameter that the list uses.

Some of the methods defined in LinkedList.java require that you execute them in O(1) (constant) time. Generally speaking, this means that you should not have to move through the entire list in order to accomplish the goal of the method. This is where the head and tail variables become useful. For the size method, you should keep a size variable so that the method doesnt have to count the number of elements in the list each time. It is also important to note that after adding and removing, the head, tail, and size variables may need to be updated.

3.2 Dots

Dots.java is a JavaFX application that presents the user with a black screen. The user can select a color at any time by pressing one of three keys: the 1 key for red, the 2 key for blue, and the 3 key for green. When the user clicks a blank region in the screen, a dot of whatever color is currently selected should appear at the same location that the user clicked. If the user clicks a dot on the screen, it should disappear. When executing correctly, the program should look something like this:



Much of Dots.java has already been written for you. The only thing that you have to do is fill in the start method. This method is responsible for adding the appropriate graphical components to the stage (Group, Scene) and adding the logic necessary for allowing the user to add and remove dots of different colors. This can be done by passing lambda expressions into Scenes setOnMouseClicked and setOnKeyPressed methods.

The Dot class has already been written for you. You should familiarize yourself with its methods before continuing.

As far as the actual logic goes, you will need to make use of the private inner class Dot and the two variables which have been declared for you: dotList (which has type LinkedList¡Dot¿) and currentColor (which has type Color). When a key is pressed, you need to figure out which key it was, and if it was "1", "2", or "3", then you will need to change currentColor to red, blue, or green accordingly. At the beginning of the start method, currentColor should be initialized to red.

The dotList variable should be initialized at the beginning of the start method and should be used to store all of the Dots that are currently on the screen. When the user clicks the screen, you must look through dotList to see if any of the Dots were clicked. If a Dot was clicked, it should be removed from both dotList and the screen. If no Dot was clicked, a new Dot should be created at the location that user clicked and should be added to both dotList and the screen.

4 Tips

- There are a few moving parts to this homework, especially the GUI part. Please, please, please dont wait until the last minute to start. Talk to your TAs and get help. If you know what youre doing, the assignment becomes much easier.
- Be sure to update the head, tail, and size variables when adding or removing to your linked list. Also, for all of the O(1) methods, use these variables. Thats why we make you have them.
- GUI programming is all about knowing the libraries! Look at the API for Group, Scene, KeyEvent, MouseEvent, and any of the other classes that you see imported at the top of Dots.java. Knowing methods like Scenes setOnKeyPressed and setOnMouseClicked are crucial for this homework.
- To make a Scene, you need to specify a Group. The Group contains all of the objects that are meant to be shown on the screen. Be sure to add the Scene to the Stage using the setScene method just before stage.show() is called.
- Take a look at Dots contains method. If you can determine what point the user clicked, the logic for telling whether or not a Dot contains that point has already been implemented for you.
- Use helper methods! Sometimes more complicated methods can get a little overwhelming. Its a good idea in these situations to write a helper method to factor out some of the work.

5 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

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

The relevant tags that you need to have 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 13.31
 */
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){
      ...
   }
}
```

Take note of a few things:

1. Javadocs are begin with `/**` and ended with `*/`.

2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class 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.

5.1 Javadoc and Checkstyle

Be sure you download the updated Checkstyle file (6.2.1) from T-Square along with this assignment. It will be used to verify both your Checkstyle and Javadocs. Javadocs will count towards your grade on this assignment.

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

6 Checkstyle

Checkstyle counts for this homework. You may be deducted up to 100 points for having Checkstyle errors in your assignment. Each error found by Checkstyle is worth one point. This cap will be raised next assignment. Again, the full style guide for this course that you must adhere to can be found by clicking **here**.

Come to us in office hours or post on Piazza if you have specific questions about what Checkstyle is looking for and how to fix Checkstyle errors.

First, make sure you download the `checkstyle-6.2.1-all.jar` from the T-Square assignment page. Then, make sure you put this file in the same directory (folder) as the `.java` files you want to run Checkstyle on. Finally, to run Checkstyle, type the first line into your terminal while in the directory of your Java files and press enter.

```
$ java -jar checkstyle-6.2.1.jar *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-6.2.1.jar *.java | wc -l
      2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-6.2.1.jar *.java | findstr /v "Starting audit..." | findstr /v "Audit
    done" | find /c /v "hashcode()"
0
```

7 Turn-in Procedure

Submit all `.java` files you wrote/changed on T-Square as an attachment. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

8 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Thursday. Do not wait until the last minute!