P07 Inventory Storage System

Overview

This assignment involves the implementation of a simple inventory storage system. An inventory storage system helps businesses to store, manage, and track inventory. There are several different ways to store inventory and properly manage it. In this programming assignment, we will store and manage inventory using a singly linked list defined by its head, its tail, its size, and different count fields to keep track of some statistics. This inventory storage list stores boxes of three colors: YELLOW, BLUE, and BROWN.

- YELLOW boxes contain fragile things. They should be always added to the head of the list.
- BROWN boxes store heavy things. They must be always added to the tail of the list.
- BLUE boxes store non-fragile less heavy things. They must be added below any YELLOW boxes currently in the list, and above any BROWN boxes in the list.

Every box has a unique inventory number to keep track of it. Boxes stored in the list must be easily removed from the inventory list when needed.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P07 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 04/04/2021**. Access to the pre-assignment quiz will be unavailable passing its deadline.
15 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they detected.
20 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include:

- Implement a storage unit as dynamic sorted Linked List from scratch.
- Further developing your experience working with object oriented design code and graphic applications.
- Improve your experience with developing unit tests.

Additional Assignment Requirements and Notes

- You are NOT allowed to add any constant or variable data field outside of any method, or any additional public method, or additional constructor not defined or provided in this write-up. You can only add private helper methods that you can call from the predefined public methods if needed.
- You CAN define local variables that you may need to implement the methods defined in this program.
- DO NOT submit the provided `Box.java` and `Color.java` files on gradescope.
- You ARE NOT allowed to use a dummy node at the head of the `InventoryList` linked list.
- You ARE NOT allowed to import any class to your source files, except relevant exceptions.
- In addition to the required test methods, we HIGHLY recommend (not require) that you develop your own unit tests (public static methods that return a boolean) to convince yourself of the correctness of every behavior implemented in your `LinkedBox` and `InventoryList` classes with respect to these javadocs and the additional details provided in this write-up. Make sure to design the test scenarios for every method before starting its implementation. Make sure also to test all the special cases.
- Feel free to reuse the javadoc method headers provided in these javadocs in your class and method javadoc headers.
- All implemented methods including the overridden methods MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide. All classes MUST include a complete javadoc class header. A complete file header MUST be added at the top of every submitted source file.
- As always, you MUST adhere to the Academic Conduct Expectations and Advice. We are going to rely on the MOSS integrated within gradescope to detect any integrity issue.

1 Getting Started

1. Start by creating a new Java Project in eclipse called P07 Inventory Storage System, for instance. .
   (a) You must ensure this project uses Java 11. Select "JavaSE-11" under "Use an execution environment JRE" in the New Java Project dialog box.
   (b) Do NOT create a project-specific package; use the default package! Your source files MUST NOT include any package statement.
2. Download the following source files and add them to your project's src folder. You MUST NOT make any changes to these files.
   (a) Color.java, and
   (b) Box.java
3. Create three classes and add them to the default src package of your project:
   (a) `LinkedBox` (instantiable class; does not include a main method);
   (b) `InventoryList` (instantiable class; does NOT include a main method);
   (c) `InventorySystemTester` (NOT instantiable class; includes a main method); You HAVE TO implement all the P07's test methods (the required ones and the ones that you may define in your own) in this class.

The provided `Box` class represents a box which can be stored and managed by our Inventory Storage System. Read carefully through the provided code and its javadoc method headers and try to familiarize yourself with it. You can notice that every Box object has a unique inventory number and a color. Three color constants are defined in the `enum Color` to serve as possible colors for the box objects. These colors are Color.BLUE, Color.BROWN, and Color.YELLOW. In the next steps, you are going to create 3 classes and add them to this project.

2 Implement the LinkedBox class

The `LinkedBox` represents a linked box node. Every instance of LinkedBox must define the TWO following private instance fields, only. You are NOT allowed to add any additional instance or static field to the `LinkedBox` class.

- **box**: private instance field of type Box. It represents the data carried by this linked box node.
- **next**: private instance field of type LinkedBox. It represents the link to the next linkedBox.

Now, implement the constructor and the public methods defined for the `LinkedBox` class according to their detailed javadocs description provided within these javadocs. We provide in the following additional details about the `LinkedBox.toString()` method. A LinkedBox object should be represented by the following String:

```
box.toString() + "-> "    // if next field is not null
box.toString() + "-> END" // if next field is null
```

Next, add a test method to your `InventorySystemTester` class with exactly the following signature. This method should test and make use of at least one LinkedBox constructor, an accessor (getter) method, and a mutator (setter) method. Note that NO setter method is defined for the LinkedBox.box instance field.

```
public static boolean testLinkedBox()
```

3 Implement the InventoryList class

The `InventoryList` This class models the singly linked list data structure that stores elements of type `Box`. This class MUST define the following instance fields.

- **head**: private instance field of type `LinkedBox` which represents the head of this list.
- **tail**: private instance field of type `LinkedBox` which represents tail of this list
- **size**: private instance field of type **int** which keeps track of the total number of boxes stored in this list.
- **yellowCount**: private instance field of type **int** which keeps track of the total number of YELLOW boxes stored in this list.
- **blueCount**: private instance field of type **int** which keeps track of the total number of BLUE boxes stored in this list.
- **brownCount**: private instance field of type **int** which keeps track of the total number of BROWN boxes stored in this list.

You are not allowed to add any additional instance or static field to this class. Recall also that you ARE NOT allowed to use a dummy node at the head of the InventoryList singly linked list.

Note that you DO NOT need to define an explicit constructor to this class. The compiler will automatically add a no-argument default constructor `InventoryList()` which creates a new empty inventory list and sets all its instance fields to their default values.

Now, make sure to implement all the public methods defined in this class with respect to the specification provided in these javadocs. We note that our inventory list will be implemented

as a chain of linked boxes nodes. Our linked list can store only boxes having one of three colors defined in the `Color` enum only (Color.BLUE, Color.BROWN, and Color.YELLOW). Adding and removing boxes to/from the list must obey to the following rules. Further details are provided in these javadocs.

- YELLOW boxes can be added only at the head of the list.
- YELLOW boxes can be removed only from the head of the list.
- BROWN boxes can be added only at the end (tail) of the list.
- BROWN boxes can be removed only from the end (tail) of the list.
- BLUE boxes can be added ABOVE all blue boxes already added to the list, but AFTER all the already added YELLOW boxes, and BEFORE all the already added BROWN boxes. This means that a BLUE box MUST be added at **position yellowCount** of the inventory list.
- Any box stored in the inventory list can be removed given their inventory number.

We note also the `InventoryList.toString()` method should traverse the contents of the list and return the following String. Refer to the String representation of the linked boxes to help you implement this method.

```
(a) An empty String ""; if the list is empty.
(b) A string containing an String representation of each box stored in the list
    separated by " -> "s, and ending with " -> END"; if the list is not empty.
```

3.1 HINTS

Here are some hints which can help you implement the different behaviors related to adding and removing boxes to and from our inventory list. We highly recommend that you try to make a problem decomposition and try to consider all the edge cases:

1. Adding a box to an empty list,
2. Adding a box to the head of the list,
3. Adding a box at the tail of the list, and
4. Adding a box at the middle of the list.

For remove operations, we recommend that you distinguish the cases of :

1. Trying to remove from an empty list, or specific box to be removed not found.

2. Removing a box from a list which contains only one box,

3. Removing a box from the middle and the tail of a non-empty the list. You will need a reference to the previous node!

Make sure to DRAW out the behavior for each of these cases ON PAPER before you begin coding. Write down the steps you need to take (and all the variables you'll need to update) as you draw.

Adding BLUE and BROWN boxes:

BLUE boxes can be added at the head of the list if no YELLOW boxes are present in the list. BROWN boxes can be added to an empty list.

Removing Boxes:

Make sure to consider the special cases for removeYELLOW() and removeBROWN() operations. As for `removeBox(int inventoryNumber)`, we recommend the following problem decomposition. Break this method down into four cases:

1. The box you're looking for is the first one in the list.

2. The box you're looking for is in the middle of the list (there are boxes both before AND after it).

3. The box you're looking for is at the end of the list.

4. The box you're looking for is not in the list. This includes the case of an empty list.

Draw out the behavior for each of these cases ON PAPER before you begin coding. Write down the steps you need to take (and variables you'll need to update) as you draw. Review the questions in the pre-assignment quiz. They can give you some hints on how to keep references to the current and previous nodes as you traverse the list looking for a match with the box to be removed given its inventory number.

3.2 Add more test methods!

You are HIGHLY encouraged to add additional test methods to your `InventorySystemTester` class to gain confidence that the methods defined in your `InventoryList` class are working correctly with respect to these javadocs the details provided above. All your test methods MUST return a boolean (true if the expected behavior is satisfied and false otherwise). In particular, create and add the following **five** test methods with exactly the following signatures to your `InventorySystemTester` class.

```
// checks for the correctness of the InventoryList.clear() method
public static boolean testClear()

// checks for the correctness of the InventoryList.addYellow(),
// InventoryList.addBlue(), and InventoryList.addBrown() methods
public static boolean testAddBoxes()

// checks for the correctness of the InventoryList.removeBox()
// InventoryList.removeYellow(), and InventoryList.remove Brown()
// methods
public static boolean testRemoveBoxes()

// checks for the correctness of the InventoryList.get() method
public static boolean testGetBoxes()

// a test suite method to run all your test methods
public static boolean runAllTests()
```

Make sure to call `Box.restartNextInventoryNumber()` at the beginning of every test method. Your `runAllTests()` test method should run all your public test methods. It **should return false if any of its component tests fail, and true if they all succeed**. Note that we will run this specific test method against several `LinkedBox` or `InventoryList` implementations (some working, some broken) in our automated tests on Gradescope.

Demo: Illustrative Example

In order to provide you with a better understanding on how to use the implemented classes, we provide in the following an example of source code and its expected output, when the methods are called with correct input arguments. You can add these methods to your tester class and class the demo() method in its main method. The following text file contains a copy of the source code of this demo.

```
public static void main(String[] args) {
  // call your test methods here
  demo();
}
/**
 * Helper method to display the size and the count of different
 * boxes stored in a list of boxes
 * @param list a reference to an InventoryList object
 * @throws NullPointerException if list is null
 */
private static void displaySizeCounts(InventoryList list) {
  System.out.println(" Size: " + list.size() +
```

```
      ", yellowCount: " + list.getYellowCount() +
      ", blueCount: " + list.getBlueCount() +
      ", brownCount: " + list.getBrownCount());
}

/**
 * Demo method showing how to use the implemented classes in P07 Inventory Storage System
 * @param args input arguments
 */
public static void demo() {
  // Create a new empty InventoryList object
  InventoryList list = new InventoryList();
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  // Add a blue box to an empty list
  list.addBlue(new Box(Color.BLUE)); // adds BLUE 1
  System.out.println(list); // prints list's content
  list.addYellow(new Box(Color.YELLOW)); // adds YELLOW 2 at the head of the list
  System.out.println(list); // prints list's content
  list.addBlue(new Box(Color.BLUE)); // adds BLUE 3
  System.out.println(list); // prints list's content
  list.addYellow(new Box(Color.YELLOW)); // adds YELLOW 4
  System.out.println(list); // prints list's content
  list.addYellow(new Box(Color.YELLOW)); // adds YELLOW 5 at the head of the list
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  // Add more boxes to list and display its contents
  list.addBrown(new Box(Color.BROWN)); // adds BROWN 6 at the end of the list
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  list.addBrown(new Box(Color.BROWN)); // adds BROWN 7 at the end of the list
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  list.removeBrown(); // removes BROWN 7 from the list
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  list.addBlue(new Box(Color.BLUE)); // adds BLUE 8
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  list.removeBrown(); // removes BROWN 6 from the list
  System.out.println(list); // prints list's content
  displaySizeCounts(list); // displays list's size and counts
  list.removeYellow(); // removes YELLOW 5
  System.out.println(list); // prints list's content
  list.removeBox(3); // removes BLUE 3 from the list
  System.out.println(list); // prints list's content
```

```
    displaySizeCounts(list); // displays list's size and counts
    try {
      list.removeBox(25); // tries to remove box #25
    }
    catch(NoSuchElementException e) {
      System.out.println(e.getMessage());
    }
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    // remove all yellow boxes
    while(list.getYellowCount() != 0) {
      list.removeYellow();
    }
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.removeBox(1); // removes BLUE 1 from the list -> empty list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.addBrown(new Box(Color.BROWN)); // adds BROWN 9 to the list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.removeBox(8); // removes BLUE 8 from the list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.removeBrown(); // removes BROWN 9 from the list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.addYellow(new Box(Color.YELLOW)); // adds YELLOW 10 to the list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
    list.removeBox(10); // removes YELLOW 10 from the list
    System.out.println(list); // prints list's content
    displaySizeCounts(list); // displays list's size and counts
  }
```

Expected output:

```
  Size: 0, yellowCount: 0, blueCount: 0, brownCount: 0
BLUE 1 -> END
YELLOW 2 -> BLUE 1 -> END
YELLOW 2 -> BLUE 3 -> BLUE 1 -> END
YELLOW 4 -> YELLOW 2 -> BLUE 3 -> BLUE 1 -> END
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 3 -> BLUE 1 -> END
  Size: 5, yellowCount: 3, blueCount: 2, brownCount: 0
```

```
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 3 -> BLUE 1 -> BROWN 6 -> END
  Size: 6, yellowCount: 3, blueCount: 2, brownCount: 1
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 3 -> BLUE 1 -> BROWN 6 -> BROWN 7 -> END
  Size: 7, yellowCount: 3, blueCount: 2, brownCount: 2
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 3 -> BLUE 1 -> BROWN 6 -> END
  Size: 6, yellowCount: 3, blueCount: 2, brownCount: 1
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 8 -> BLUE 3 -> BLUE 1 -> BROWN 6 -> END
  Size: 7, yellowCount: 3, blueCount: 3, brownCount: 1
YELLOW 5 -> YELLOW 4 -> YELLOW 2 -> BLUE 8 -> BLUE 3 -> BLUE 1 -> END
  Size: 6, yellowCount: 3, blueCount: 3, brownCount: 0
YELLOW 4 -> YELLOW 2 -> BLUE 8 -> BLUE 3 -> BLUE 1 -> END
YELLOW 4 -> YELLOW 2 -> BLUE 8 -> BLUE 1 -> END
  Size: 4, yellowCount: 2, blueCount: 2, brownCount: 0
Error to remove box. Box #25 not found!
YELLOW 4 -> YELLOW 2 -> BLUE 8 -> BLUE 1 -> END
  Size: 4, yellowCount: 2, blueCount: 2, brownCount: 0
BLUE 8 -> BLUE 1 -> END
  Size: 2, yellowCount: 0, blueCount: 2, brownCount: 0
BLUE 8 -> END
  Size: 1, yellowCount: 0, blueCount: 1, brownCount: 0
BLUE 8 -> BROWN 9 -> END
  Size: 2, yellowCount: 0, blueCount: 1, brownCount: 1
BROWN 9 -> END
  Size: 1, yellowCount: 0, blueCount: 0, brownCount: 1

  Size: 0, yellowCount: 0, blueCount: 0, brownCount: 0
YELLOW 10 -> END
  Size: 1, yellowCount: 1, blueCount: 0, brownCount: 0

  Size: 0, yellowCount: 0, blueCount: 0, brownCount: 0
```

4 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only THREE files that you must submit is: `LinkedBox.java`, `InventoryList.java`, and `InventorySystemTester.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **11:59PM on April 7**th. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.