The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algoirthms

Assignment 1

Date Due: January 25, 2021, 6:00pm

Total Marks: 28

1 Submission Instructions

- Assignments must be submitted using Canvas.
- Programs must be written in Java.
- VERY IMPORTANT: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a ZIP archive file. This can be done with a feature built into the Windows explorer (Windows), or with the zip terminal command (LINUX and Mac). We cannot accept any other archive formats. This means no tar, no gzip, no 7zip. Non-zip archives will not be graded. We will not grade assignments if these submission instructions are not followed.
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

For this assignment you'll be working with linked list classes from the data structure library `lib280`. `lib280` is a library of data structures that we will build up over the duration of the course. We will start with a version of `lib280` that has very few data structures in it and add more with each assignment. Each assignment will come with a new version of `lib280` which contains the correct implementations of ADTs that were the subject of the previous assignment.

Obtaining and Setting Up `lib280`

For this assignment the first thing you'll need to do is to obtain a copy of `lib280-asn1`. It is provided along with this assignment description on the class webpage. Download the lib280-asn1.zip file and expand its contents somewhere in your filesystem.

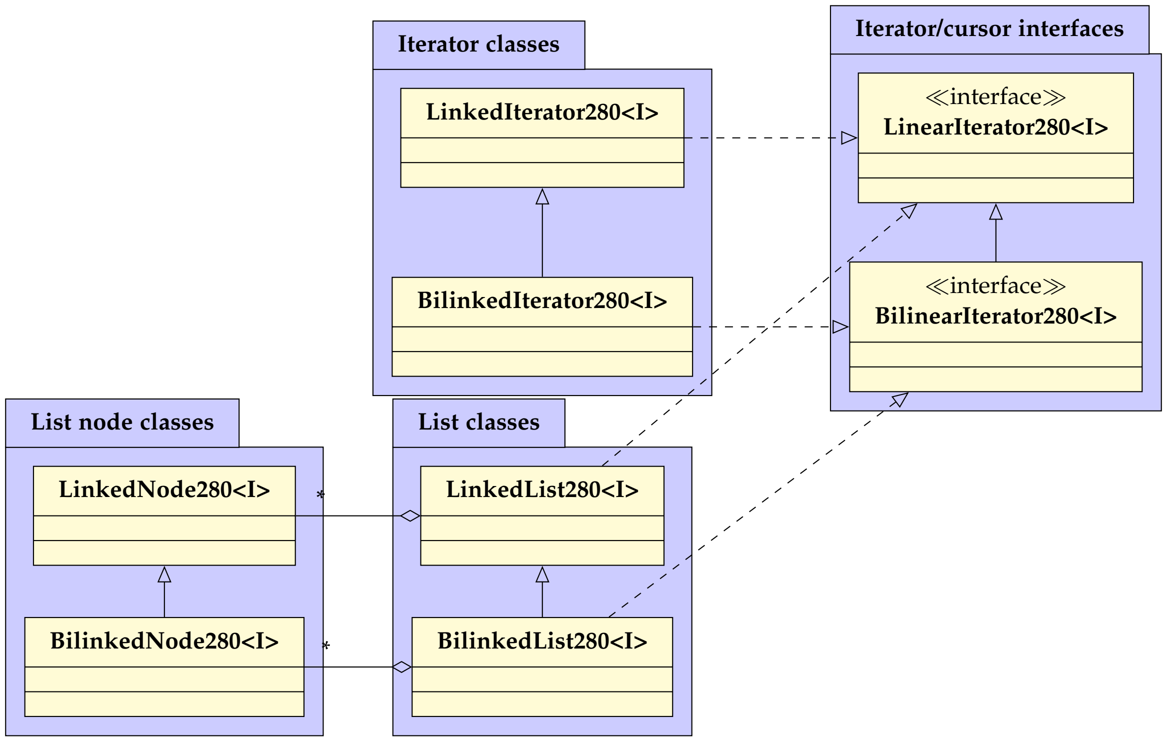
The class website provides a self-guided tutorial that explains how to import lib280 into an IntelliJ project once you have downloaded it; it is located under the "Assignments" heading and "Self-Guided Tutorials" subheading. First complete part 1 of the tutorial to create an empty IntelliJ project. Then complete part 2 of the tutorial to import `lib280-asn1` into your project. For question 1 complete part 3 of the tutorial to create a module for your program that can use the classes from `lib280-asn1`. For questions 2 and 3 you don't need to complete part 3 again because you'll just be working within the `lib280-asn1` module.

Navigating `lib280-asn1` Within IntelliJ

The `lib280-asn1` module contains several packages. The classes of interest to use for this assignment are in the `lib280.list` package. Find the `lib280-asn1` module in your "project" tab, normally located on the left side of the IntelliJ window. Expand it by clicking the little triangle beside it. This should reveal a folder called "src". Expand that as well. Now you will see a list of java packages that contain the various classes in the `lib280-asn1` library. For this assignment, the classes we are interested in are in the `lib280.list` package, so click the triangle to expand it. You should now see classes like `LinkedList280` and `BilinkedList280`.

Singly- and Doubly- Linked List Classes in `lib280`

The UML diagram below shows the class hierarchy you'll be working with in this assignment. It may look a bit daunting at first, but you'll soon see it's not that complicated. There are four pairs of classes/interfaces (surrounded by light blue boxes[1]). In each pair, there is one class for a singly-linked list and one for a doubly-linked list. The class/interface of each pair that pertains to doubly linked lists extends the class/interface related to singly linked lists.



`LinkedNode280<I>`: The node class used for a singly-linked list.

`BilinkedNode280<I>`: An extension of `LinkedNode280<I>` that adds the "previous node" reference required for nodes in a doubly linked list.

`LinearIterator280<I>`: An interface that defines the methods that must be supported by cursors and iterators that can step forwards over a linear structure, such as `goFirst()`, `goForth()`, `after()`, etc.

`BilinearIterator280<I>`: An interface that extends `LinearIterator280<I>` by adding methods that allow stepping backwards, such as `goBack()` and `goLast()`.

[1]The light blue boxes in the UML diagram are only to show the pairs of classes that serve the same roles for singly-/doubly-linked lists and do not represent any actual grouping within `lib280`. All of the pictured classes are in the same package within `lib280`.

`LinkedIterator<I>`: An implementation of `LinearIterator280<I>` which is an iterator object for a singly-linked list. It is used by the `LinkedList280<I>` class to provide iterators.

`BiinkedIterator<I>`: An implementation of `BilinearIterator280<I>`, and an extension of `LinkedIterator280<I>`, which is an iterator object for a doubly-linked list. It is used by the `BilinkedList280<I>` class to provide iterators.

`LinkedList280<I>`: A singly-linked list class. It provides a cursor by implementing the LinearIterator280<I> interface. The Nodes of the list are `LinkedNode280<I>` objects, and it can provide iterators of of type `LinkedIterator<I>`.

`BiLinkedList280<I>`: A doubly-linked list class. It provides a cursor that can move both forwards and backwards by implementing the BilinearIterator280<I> interface. The nodes of the list are `BilinkedNode280<I>` objects, and it can provide iterators of of type `BilinkedIterator<I>`.

Take a moment to familiarize yourself with these classes and their methods, particularly the `LinkedList280<I>` and `LinkedIterator280<I>` classes as you will be working on coding extensions of these classes.

Iterators

This section describes a bit more about how iterators work. Iterators provide the same functionality as a container ADT that has a cursor, but they are separate objects from the container. This allows us to record a cursor position that is different and independent from the position recorded by the container's internal cursor.

The list objects, `LinkedList280<I>` and `BilinkedList280<I>` both have methods called `iterator`. The `iterator` method in the `LinkedList280<I>` class returns a new cursor position encapsulated in an instance of the `LinkedIterator280<I>` class. This instance will have references directly to the nodes of the `LinkedList280<I>` instance that created it. In essence, the `LinkedIterator280<I>` contains its own copies of the `position` and `prevPosition` fields that appear in `LinkedList280<I>` – i.e. another cursor that is external to the list! This cursor can be manipulated in exactly the same was as the internal cursor of the list. If you compare the methods in `LinkedIterator280<I>` to the methods of the same name in `LinkedList280<I>`, you'll see that they are almost identical.

Thus, each time we want a new cursor that is independent of the list's internal cursor, we can call the `iterator` method and get a new one. This adds additional flexibility. If we can get away with just using the lists internal cursor for our purposes, then we can do so, but we have the option to create more cursors in the form of iterators should we so desire.

3 Your Tasks

Question 1 (4 points):

Tractor Jack is a notorious pirate captain who sails the Saskatchewan River plundering farms for wheat, barley, and all the other grains. You may remember him from his exploits in CMPT 141. Jack wants to simulate the loading of cargo onto his ships so he can track how much of one type of grain is on a given ship, and to make sure that his ships are not overloaded. [2]

In this problem you will be given a list-of-lists data structure. It will be a list of `Ship` objects, each of which contains a list of `Sack` objects. Each `Sack` object represents one sack of a particular type of grain.

The type of grain in a sack is represented by a value of type `Grain`, where `Grain` is an *enumeration*.

Enumerations

In this question use a data type in Java called an enumeration to represent the type of grain in a sack. Enumerations define a fixed set of named constant values. The grains Jack most commonly plunders are wheat, barley, oats and rye so he wants to count the amount of those four grains separately. Any other types of grain he wants to count together. We can us an enumeration to define five constants to denote what type of grain is in a sack:

```
enum Grain {
   WHEAT, BARLEY, OATS, RYE, OTHER
}
```

This declaration defines a data type called `Grain` and five values which we can assign to variables of type `Grain`. You can find it at the top of `Sack.java`. Now we can then write in Java:

```
Grain g = Grain.WHEAT; // Assign value WHEAT to the variable g
```

You'll need to use one of the values from the `Grain` enumeration in task 3, below.

The Problem

You are provided with three Java files:

Sack.java: Contains the class `Sack` which is an object that represents a sack of grain. A sack of grain has a grain type, and a weight (in pounds). This object is complete and you will not need to edit this file, but you should familiarize yourself with its data and methods. This file also contains the definition of the enumerated type `Grain`.

Ship.java: Contains the class `Ship` which represents a ship in Jack's fleet. Each ship has a name, a capacity (weight in pounds), and contains a list of `Sack` objects which represent the ship's cargo. This class contains two unfinished methods that you will write (see below). Familiarize yourself with the other methods and instance variables of this class.

CargoSimulator.java This file contains the `CargoSimulator` class. It doesn't do much. Its constructor generates a list of ships and fills them with sacks of grain. This is the data structure you'll be working with. You'll be writing some code in the `main()` method (see below).

Complete the following tasks:

[2]This question was inspired by this song (click to link). Arrrr!

1. In `Ship.java`, complete the `isOverloaded` method. This method must return a boolean value `true` if the ship is overloaded, and `false` otherwise. The ship is overloaded if the total weight of all sacks of grain in its cargo exceeds the ship's capacity.
2. In `Ship.java`, complete the `sacksOfGrainType` method. This method must return the number of sacks of grain of the grain type indicated by its parameter that are in the ship's cargo. That is, if the parameter `type` is `Grain.WHEAT` and the ship's cargo contains 42 sacks of wheat, the method should return 42.
3. In the `main()` method of `CargoSimulator.java`, there is an instance of a `CargoSimulator` object called `sim`. As described above, this object contains a list of ships, and each ship contains its list of cargo. At the location indicated, print out how many sacks of **barley** each ship in `sim` is carrying.
4. In the `main()` method of `CargoSimulator.java`, print out a message for each ship in the `CargoSimulator` instance `sim` that is overloaded. If a ship is not overloaded, print nothing.

Note: Upon inspection of the constructor for `CargoSimulator`, it may appear that the data is being randomly generated. The data **is** "randomly" generated, but a fixed random seed is used to ensure that the same random instance of the data is generated every time the program is run. Thus, you should expect that the data will always be the same, and that you will get the same answer every time. That said, it is possible that different computers and/or operating systems will generate different data, but on the same machine within the same operating system, the same data will be generated every time.

Sample Output

Here is what the output might look like. This demonstrates the form of the output only, and not the expected numbers.

```
The Icebreaker is carrying 47 sacks of barley.
The Salty Farmer is carrying 47 sacks of barley.
The Bunnyhug is carrying 35 sacks of barley.
The Blackstrap is carrying 31 sacks of barley.
The Prairie Onion is carrying 42 sacks of barley.
The Bunnyhug is overloaded!
```

Question 2 (12 points):

The `BilinkedList280<I>` and `BilinkedIterator280<I>` classes in `lib280-asn1` are incomplete. There are missing method bodies in each class. Each missing method body is tagged with a `// TODO` comment. The javadoc headers for each method explain what each method is supposed to do[3]. Many of the methods you must implement override methods of the `LinkedList280<I>` superclass. Add your code right into the existing files within the `lib280-asn1` module.

When implementing the methods, consider carefully any special cases that might cause need to update the cursor position, or ensure that it remains in a valid state.

You are not permitted to modify any existing code in the .java files given. You may only fill in the missing method bodies.

Question 3 (12 points):

Write a regression test for the `BilinkedList280<I>` class. You only need to test the methods that *you* had to write. You may generate test cases using white-box, black-box, or a combination of both methods. Again, write this code in the existing `BiLinkedList280.java` within the `lib280-asn1` project. A function header for the regression test (`main()` function) has already been provided.

Marks for this question will be earned for generating and coding good tests, not whether or not the methods being tested actually work. This means that you can still get full marks on this question even if the methods you were supposed to code in Question 2 don't work.

The regression test should be commented so that the purpose of each test is clearly indicated. Indicate which methods you are testing and the conditions under which you are testing it.

4 Files Provided

lib280-asn1: A copy of lib280.

Sack.java Object representing a sack of grain for question 1.

Ship.java Object representing one of Tractor Jack's pirate ships for question 1.

CargoSimulator.java Object for simulating the loading of cargo onto Tractor Jack's ships for question 1.

5 What to Hand In

Ship.java Your completed `Ship` object for question 1.

CargoSimulator.java Your completed `CargoSimulator` object for question 1.

BilinkedList280.java: Your completed doubly linked list class from question 2 and its regression test that you wrote for question 3.

BilinkedIterator280.java: Your completed iterator class from question 2.

6 Grading Rubric

The grading rubric can be found on Canvas.

[3]The javadoc comments in these files are also good examples of how we will expect you to document methods that you write yourself in future assignments.