The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algoirthms

Assignment 1

Date Due: January 17, 2017, 10:00pm

Total Marks: 33

1 Submission Instructions

- Assignments must be submitted using Moodle.
- Programs must be written in Java.
- 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 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.

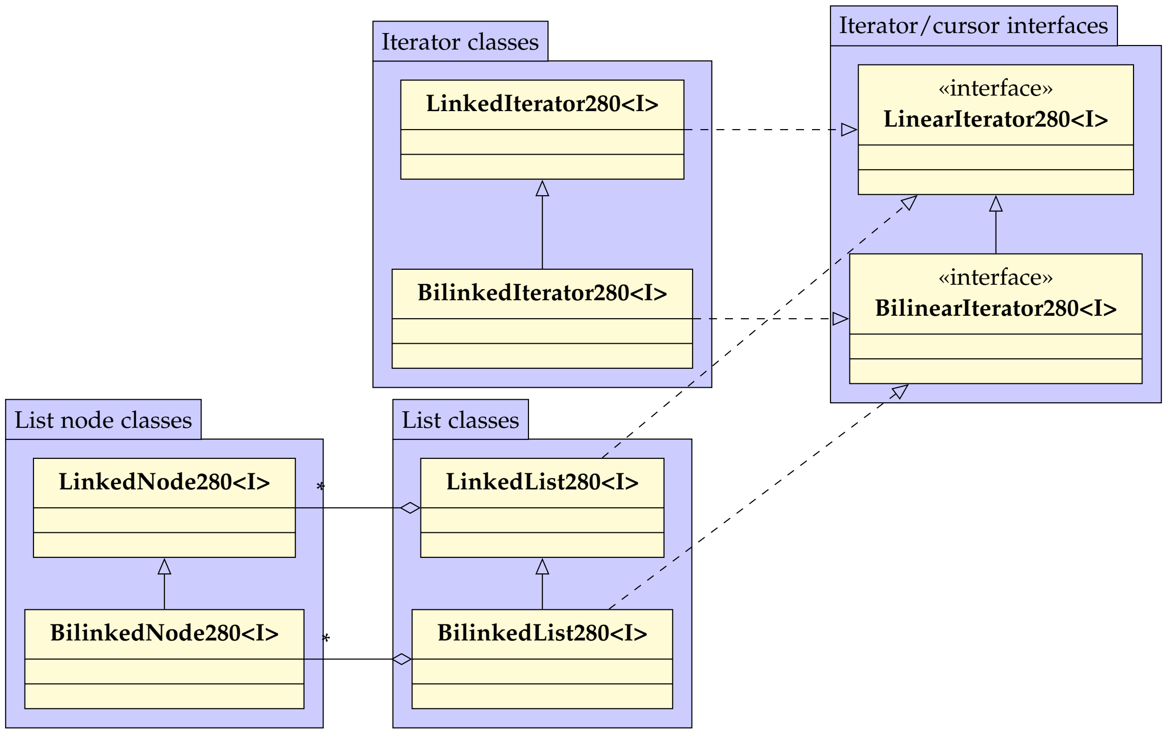
The class website provides a self-guided tutorial that explains how to import lib280 into an IntelliJ project once you have downloaded it; it is located under the "Laboratory/Tutorial Resources" heading. First complete part 1 of the tutorial to create an empty IntelliJ project. Then complete part 2 of 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 `ArrayedList280` and `LinkedList280`.

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. In essence, there are four pairs of classes/interfaces (surrounded by light blue boxes[1]). and 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()`.

`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>`.

[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`.

`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 (8 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 111. At the end of each day, Jack enters into his computer a log of each sack of grain he has plundered, what kind of grain it is, and how much it weighs. You will help Jack write a program to organize this data, and calculate much of each type of grain he plundered.[2]

Enumerations

In this question we're going to use a data type in Java called an enumeration. 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
```

Here are some other thing we can do with enumerations that you will need:

- Enumeration types have a static method called `values()` which returns an array of the values it defines. For example:

  ```
  // This returns the array:  { WHEAT, BARLEY, OATS, RYE, OTHER }
  Grain.values()
  ```

- Each value defined by the enumeration is associated with an integer value called an *ordinal* between 0 and $N-1$ (where N is the number of values in the enumeration). These ordinals are a convenient way to map values in an enumeration to offsets of an N-element array.

  ```
  Grain g = Grain.OATS;
  g.ordinal();      // returns the integer 2,
                    // because OATS is the 3rd value in the enum.
  g = Grain.WHEAT;
  g.ordinal();      // returns the integer 0,
                    // because WHEAT is the 1st value in the enum.

  myArray[g.ordinal()]  // Use a value from the Grain enumeration to
                        // select a corresponding element of an array.
  ```

- You can find out how many values are in an enumeration asking for the length of the array returned by the `values()` method:

  ```
  Grain.values().length  // this is 5 because there are five values
                         // in the Grain enumeration.
  ```

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

- Because enumerations have `toString()` methods, we can print enumerated types, and concatenate them with strings. For example:

```
Grain g = Grain.RYE;
System.out.println(g)  // prints: RYE
System.out.println("The␣grain␣is:␣"+g) // prints:  The grain is RYE
```

The Problem

Your task is to write a program to sort the sacks of grain that Jack plundered by the type of grain in each sack, and calculate how much of each type of grain he has. This will be done by adding all the sacks of grain containing one kind of grain to its own list.

Create a new project that references lib280-asn1 as described in the self-guided tutorial on the class website. Add to it a class called `A1Q1`. Add the `generatePlunder` method (given below) to this class. Add the provided `Sack.java` to your project (just copy it into the same folder as `A1Q1.java`).

Add a `main()` method to your `A1Q1` class. Inside it, write a program that does the following:

1. Add the line `import java.util.Random` to the top of your `A1Q1` class.
2. Call the following function to generate some data that represents Captain Jack's plunder for the day:

   ```
   public static Sack[] generatePlunder(int howMany) {
       Random generator = new Random();
       Sack grain[] = new Sack[howMany];

       for(int i=0; i < howMany; i++) {
           grain[i] = new Sack(
           Grain.values()[generator.nextInt(Grain.values().length)],
           generator.nextDouble() * 100 );
       }

       return grain;
   }
   ```

   This will return a randomly generated array of `Sack` objects with length `howMany`. `Sack` is a simple class that stores the type of grain in a sack (as the enumerated type `Grain`), and how much it weighs. The `Sack` class is provided for you to use. You'll that see its quite straightforward.
3. Create an array of linked lists of `Sack` objects; there should be one list for each type of grain, including `OTHER`. This should be an array of `LinkedList280<Sack>` objects. Each list in the array will store `Sack` objects for one and only one type of grain. All of the sacks containing `OTHER` grain should go on the same list. Remember to create not only the array, but also instantiate a list for each element of the array.
4. Put each `Sack` object in the array you created in step 1 onto one of the lists you created in step 2. Use the ordinal of the grain type of the sack object to index the array of linked lists to find the correct list for the type of grain, and add the sack object to that list.
5. Go through the items each list using its cursor and compute the total weight of each type of grain that Jack plundered. Remember that since the data is randomly generated, it is possible for a list to be empty!
6. Print a report of Jack's plundering to the console. Below is a sample of what that should look like.[3]

[3]Jack seems to have a very precise scale for weighing his sacks of grain!

```
Jack plundered 0.0 pounds of WHEAT
Jack plundered 75.8422984943735 pounds of BARLEY
Jack plundered 74.01721574496484 pounds of OATS
Jack plundered 48.82389493369351 pounds of RYE
Jack plundered 44.962951754620065 pounds of OTHER
```

Implementation Hints

Make use of the examples with enumerations provided above. They are there because you should be using these features of enumerations in your solution! You can't use the examples as is, but everything they show you how to do should be used in your solution.

Question 2 (15 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[4]. 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.

Marks for this question are earned by implementing the methods correctly.

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 (10 points):

Write a regression tests 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.

4 Files Provided

lib280-asn1: A copy of lib280.

Sack.java: A copy of Captain Jack's `Sack` class.

5 What to Hand In

A1Q1.java: Your program for question 1. (You do not need to submit `Sack.java`).

A1Q1output.txt: The output from the program you wrote for question 1 cut and paste from the IntelliJ console.

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.

[4]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.