CMPSCI 187 / Spring 2015
Implementing Sets Using Linked Lists

Due on Tuesday February 24, 2015, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

In this assignment you will build a class to represent a *set*; as in the mathematical concept of a set, a collection that has no duplicates. The class will be *immutable* (more later), *generic*, and will use `Iterator`s. It will be implemented with linked lists. We will give you (in the `support` directory) the `Set` interface, and the `LinkedNode` class, and (in the `src` directory) skeletons for the `LinkedSet` and `LinkedNodeIterator` classes which you will complete. In this assignment you will be starting from initial code provided by us, and adding to and modifying it. You should not have to create any new classes or interfaces.

Learning Goals

- Continue to exercise your understanding of linked lists.
- Introduce the concept of immutable classes.
- Introduce the concept of generics.
- Show ability to write a class that implements a given interface.
- Read a specification in text and as a Java interface, and demonstrate understanding by implementing it.
- Use existing JUnit test cases.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Problem 1

Review of the Concept of Sets

(This is just a simple review. One more thorough explanation may be found on Wikipedia, at `http://en.wikipedia.org/wiki/Set_(mathematics)`.)

We are referring to sets in the sense of the mathematical construct of set. A set is an unordered collection of elements without duplicates. They are usually expressed in the following fashion:

$$Q = \{a, b, c, d, e\}$$

says that Q is a set with five elements, a, b, c, d, and e. Note that the collections are unordered, so:

$$\{a, b, c, d, e\} = \{a, e, c, b, d\} = \{c, b, e, d, a\} = \dots$$

There are various standard operation on sets, tests for membership, tests for inclusion, test for equality, and the union,

Problem 1 continued on next page. . .

intersection, and set difference operations. These are illustrated below.

Given:
$A = \{a, b, c, d, e\}$
$B = \{c, a\}$
$C = \{3, 1, 4, 5, 9\}$
then

$a \in A$	membership (read as a is an element of A)
$B \subset A$	subset (read as B is a subset of A)
$A \supset B$	A is a superset of B
$B \cup C = \{3, 1, c, a, 4, 9, 5\}$	set union, include all elements from both sets
$A \cup A = A$	sets don't have duplicates
$A \cap B = B \cap A = B$	set intersection, those elements that are elements of both sets
$A \cap C = \emptyset$	set intersection with no common elements gives the empty set
$A - B = \{b, d, e\}$	set difference, those elements in A but not in B

We will be implementing all of these operations on our sets.

Immutability

Some classes represent *immutable* objects, i.e., objects whose state cannot be changed. You are already familiar with at least one such class, the `String` class. Instances of that class cannot be changed in any way. If you look carefully you will notice that any `String` method that could be construed as changing the `String` actually returns a new `String`; the original is unchanged. Other immutable classes include `Boolean`, `Integer`, and `Float`.

The `Set` interface described in this project specifies an immutable class. Note that none of the methods allows one to change an instance, they only return new instances.

Immutable objects have some nice properties, most of which we will not go into here, but we can say that they are easier to reason about than mutable objects, and also that they are much safer to use in a multi-threaded environment.

Your implementation of the `Set` interface must maintain that immutability. Once an instance is created it never changes.

Generics

The textbook defined the `LLStringNode` class (DJW page 103), and you have also seen the `LLDogNode` class, which differed from the former only in that the `info` field was of type `Dog` instead of type `String`. In future exercises we might need node classes to build linked lists of `Locomotive`, `Integer`, or pretty much anything else. We would copy and paste and then modify (slightly) the basic class, but that is wasteful and prone to errors; if we discover a bug in one version we have to go back and fix that bug in every version. A better answer is *parametric polymorphism*. The idea is to replace specific types in a class definition with type variables, and declare those type variables in the class header. Java calls this concept *generics*, and generic classes and interfaces were introduced in Java 5 (JDK 1.5).

Figure 1 shows three different classes, `LLStringNode`, `LLDogNode`, and `LLNode`, the last being a generic version of the first two. This is designed to illustrate how to define generic classes. We have simplified these definitions slightly

```
 1  class LLStringNode {               class LLDogNode {              class LLNode<T> {
 2    String info;                       Dog info;                      T info;
 3    LLStringNode link;                 LLDogNode link;                LLNode<T> link;
 4
 5    LLStringNode(String info) {        LLDogNode(Dog info) {          LLNode(T info) {
 6      this.info = info;                  this.info = info;              this.info = info;
 7      this.link = null;                  this.link = null;              this.link = null;
 8    }                                  }                              }
 9
10    String getInfo() {                 Dog getInfo() {                T getInfo() {
11      return info;                       return info;                   return info;
12    }                                  }                              }
13
14    LLStringNode getLink() {           LLDogNode getLink() {          LLNode<T> getLink() {
15      return link;                       return link;                   return link;
16    }                                  }                              }
17
18    void setInfo(String info) {        void setInfo(Dog info) {       void setInfo(T info) {
19      this.info = info;                  this.info = info;              this.info = info;
20    }                                  }                              }
21
22    void setLink(                      void setLink(                  void setLink(
23        LLStringNode link)                 LLDogNode link)                LLNode<T> link)
24   {                                   {                              {
25      this.link = link;                  this.link = link;              this.link = link;
26    }                                  }                              }
27  }                                  }                              }
```

Figure 1: Definitions of `LLStringNode`, `LLDogNode`, and `LLNode` classes.

```
LLNode<Dog> dogNode = new LLNode<Dog>(dog1);
dogNode.setInfo(dog2);
dogNode.setLink(new LLNode<Dog>(dog3));
Dog dog = dogNode.getInfo(); // dog == dog2

LLNode<String> sNode = new LLNode<String>("abc");
sNode.setInfo("xyz");
sNode.setLink(new LLNode<String>("xyz"));
String s = sNode.getInfo(); // s == "xyz"
```

Figure 2: Using the `LLNode` class.

to fit them on the page (we removed comments and the **`public`** and **`private`** modifiers). Figure 2 shows typical uses of the `LLNode` class.

The major changes are the inclusion of `<T>` on line 1 of the `LLNode` definition. This declares that `LLNode` is generic, and has a type variable named `T`. When you use the `LLNode` class, for instance by using `LLNode<String>`, you are specifying that you want a version of `LLNode` created to handle `String`s. It is as though you created a new specialized class with all of the instances of the type variable `T` replaced by the type `String`. It is perfectly possible, as Figure 2 shows, to have several different instantiations of the generic class.

DJW has a discussion of generic classes and interfaces on page 166. See that for additional enlightenment.

Iterators and the `Iterable` Interface

Iterators and the `Iterator` and `Iterable` interfaces are given short shrift in the DJW, being mentioned briefly in a box on pages 396 and 397, but they are actually very important, and you have probably, all unknowingly, used them

Problem 1 continued on next page. . .

before. If you have ever written code like:

```java
for (Type variable : array) {
  ...
}
```

then you have used an `Iterator` and the `Iterable` interface. To be specific `array` must have been an instance of a class that implemented the `Iterable` interface. In fact, the code that you wrote is exactly equivalent to:

```java
Iterator iter = array.iterator();
while (iter.hasNext()) {
  Type variable = iter.next();
  ...
}
```

(except that the variable `iter` is hidden from you).

An iterator is a object that allows you to examine the elements of some collection one-by-one.

The `Iterable<T>` interface specifies only one method, the `Iterator<T> iterator()` method, which specifies that the implementing class will happily return an iterator for the collection.

The `Iterator<T>` interface specifies three methods, `boolean hasNext()`, `T next()`, and `void remove()`. For our purposes in this assignment we are going to ignore the `remove()` method; the definition you are given will simply throw an `UnsupportedOperationException` to indicate that this operation is not supported, which it cannot be, because we cannot remove elements from an immutable collection.

Note: You may not use any of the classes from the Java Platform API that implement the Collection interface (or its sub-interfaces, etc.). The list of these classes is included in the API documentation, available at: `http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`. Doing so will be viewed as an attempt to cheat.

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the

Problem 1 continued on next page. . .

menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Starter Code and Tests

In this assignment we are not giving you any "driver," no class with a `main` method. All of your testing will be done using the supplied tests (or those you write yourself).

In the `support/sets` directory you will find `Set.java`. This file defines the `Set` interface. Read that file closely, as it defines exactly what your implementation must do. You will also find `LinkedNode.java`, which defines the `LinkedNode` class, an immutable, generic, generalization of the `LL...Node` classes you have seen before. As with all files in the `support` directory you must not modify either of these files.

In the `src` directory you will find two files that you must complete, `LinkedSet.java` and `LinkedNodeIterator.java`. `LinkedSet.java` will have a single error on line 34; this error will be resolved by your work on `LinkedNodeIterator.java`.

The version of `LinkedSet<E>` we have given you has one attribute, `head`, which is intended to be the head of a linked list of elements in the set. It also has three constructors; two public, one which creates an empty set, and the other of which creates a set containing a single element; and one private which creates a set from a linked list of `LinkedNode <E>`s. This last constructor is the one that you will call from your methods to create the new `LinkedSet` object to be returned. There are ten methods that you will need to supply implementations for. We suggest that you start by implementing the `iterator` method; because many of the other methods can use it to good effect; for an example see the `hashCode()` method.

To implement the `iterator` method you will need to finish and use the `LinkedNodeIterator<T>` class. This class has two methods you will need to complete, `hasNext()` and `next()`. You will also need to choose appropriate attributes and build an appropriate constructor. If you remember that you are iterating over a linked list of `LinkedNode` s you should not have too much difficulty with this.

A few hints:

- An appropriate parameter for the constructor of the `LinkedNodeIterator` class is a reference to the first `LinkedNode` in the list.
- `isEmpty()` should be really trivial.
- Use the iterator in implementing the `size`, `contains`, `isSubset`, `union`, `intersect`, `subtract`, and `remove` methods.

Problem 1 continued on next page...

- Use the `contains` method everywhere it would be useful.
- Under certain circumstances `adjoin` and `remove` can both just return **`this`**.
- Implement `isSuperset` using `isSubset`.
- This one won't make any sense until you try it, but, when implementing `isSubset` iterate over the other set, not **`this`**.

The starter code you receive is *not a working implementation*. However, you can still run the tests, most of which will fail.

Testing

We provide one public JUnit test classes you should use to test your implementation, `LinkedSetPublicTest.java`.

You should run these tests to test your implementations. Your grade for this assignment will depend on the results of these tests as well as private tests (that are not visible to you) that we have constructed to ensure that your implementation has not been tailored to the public tests.

For full credit, you must correctly implement each method in `LinkedSet.java` and `LinkedNodeIterator.java`. Make sure that you have done everything that is specified in the TODO comments in those files.

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `sets-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`sets-student`**. Save the exported file with the `zip` extension (any name is fine). Log into Moodle and submit the exported zip file.