Doubly Linked List Lab

CS 331 — Fall 2010

Revision 1.0

Assigned
Due

1 Objectives

In this lab you will create a doubly linked list, with four iterators.

- Create a doubly linked list.
- Have more experience using the `Interface` feature of Java.
- Use inheritance to enable you to reuse code.

2 Doubly-Linked Lists

As we discussed in lecture, a doubly linked list node has two pointers: `next` and `prev`. Instead of just going forward through the list, you can also go backwards. This is very convenient, because we can back up as well as go forward, but it also means that there are two pointers per node that must be maintained, and it is easy to cause inconsistencies.

3 Given Files

You are given 4 files for this lab. They are in the `doubly-linked-lists` directory of your repository. Do an `git pull` to download them. Here are the files you will find:

- `DList.java` — A skeletal doubly-linked list implementation. Please review this code carefully before using it, it's not meant to be complete.
- `Iterator.java` — An interface file for the `Iterator`. This is the same one you had from last time.
- `Makefile` — the scripts needed to compile your files and run the tests. It supports the following targets:

  compile Compiles everything.

  tests Tests everything.

- `TestDList.java` — another skeleton test file.

4 Your Work

First you will need to implement the doubly-linked list methods. After that, you will get to write four iterators.

4.1 Doubly Linked Lists

You will need to implement at least the following methods. You may add more if you feel they are necessary.

constructor This just creates an empty list.

int size() Return the number of elements stored in the list. This should run in $\mathcal{O}(1)$ time.

void insertFront(E data) Insert an element at the front of the list. This should run in $\mathcal{O}(1)$ time.

void insertEnd(E data) Insert an element at the back of the list. This should run in $\mathcal{O}(1)$ time.

void deleteFront() Delete the first element, if one exists. This should run in $\mathcal{O}(1)$ time.

void deleteEnd() Delete the last element, if one exists. This should run in $\mathcal{O}(1)$ time.

Iterator makeFwdIterator() Create a forward iterator.

Iterator makeRevIterator() Create a reverse iterator. It starts from the last element and works its way to the first element.

Iterator makeFwdFindIterator(E data) Create a find iterator that moves forward.

Iterator makeRevFindIterator(E data) Create a find iterator that moves backward.

Suggested order: write the constructor and `size` method first[1]. After that, write `insertFront`, `makeFwdIterator`, and `makeRevIterator` to test insertions. It will be best if you test these three at once—each time you test a new method that updates the list, make a forward iterator and a reverse iterator and make sure they both return the proper elements. This will ensure your list's consistency. Next add the rest of the methods, and finally the find iterators.

Oh, one other thing. Use Sentinels!! Of course, it's totally up to you whether you use them or not. But it is much easier to use them than not to use them.

4.2 The Iterators

In this lab you will write four iterators. The methods are the same as last time.

constructor we need to create them, after all!

E get() Return the current element. It does *not* advance the cursor! If there is no current element, return `null`.

void next() Advance the iterator's cursor. If the iterator is currently invalid, do nothing.

boolean isValid() Return `true` if the iterator still has data.

[1] I will assume that when I say "write method `x`," that you actually write the test case for `x` first, then write the method.

void delete() We've added this one to the usual mix. It will delete the current element from the original list, and advance the iterator's cursor. Do nothing if the cursor is invalid.

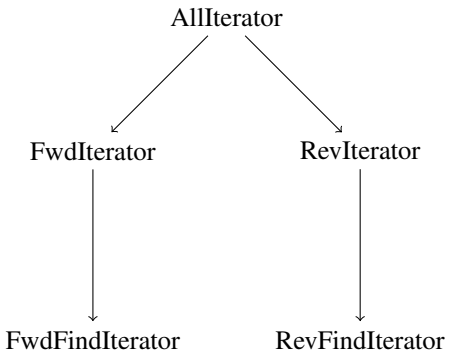
One note about delete: when delete is called on an iterator, all other iterators are considered "undefined". You don't have to remove them, but do **NOT** write any tests against other iterators that were not part of the delete. For example, suppose you have three iterators i_1, i_2, and i_3. If you call delete on i_2, then your tests should not require anything of i_1 or i_3. What happens to them is up to you.

Now, before you go making a bunch of iterators, you should think about it for a minute. Look at your iterators from the Iterator lab. You will notice that `isValid`, `delete`, and `get` are the same. At least, they *should* be. It's only `next` and the constructors that are different.

In this lab, you *must use inheritance* to take advantage of this situation. First, write a member class called `AllIterator`. This class should define the methods we mentioned above, and simply declare the `next` method.

After that, define the class `FwdIterator` which `extends AllIterator`. It will inherit all those methods from `AllIterator`, and all you have to do is write the `next` method for it. Later, you can write `FwdFindIterator`, and override the `next` method to skip ahead until it finds matches.

Similarly, `RevIterator` will inherit from `AllIterator`, and `RevFindIterator` will inherit from `RevIterator`. Here's a class diagram.



Confused? This time we did much of the work for you. The `AllIterator` is defined, and the `FwdIterator` is defined as well, inherited from `AllIterator`. We'll let you do the other three.

You'll notice a few interesting things about the `AllIterator` class. First, the header uses the keyword `abstract`:

```
private abstract class AllIterator implements Iterator<E> {
```

This tells Java that we are going to declare some of the methods, but not all of them. Sure enough, later in the definition we find:

```
public abstract void next();
```

Again the `abstract` keyword. You'll notice the `;` after `next()`. This tells Java that we are not going to give a definition for this method, we just want to say that it needs to exist. The `abstract` keyword here tells Java that, yes, we really meant to leave it undefined, we didn't forget.

Another interesting thing is that `cursor` is `protected`. This allows it to be accessed by inherited classes, but not by `DList` or other outsiders. If we made it `public`, anyone could get it. If we made it `private`, then the `FwdIterator` class wouldn't be able to use it either.

You'll also notice that the `delete()` method is defined, it is *not* abstract, but the code is empty. You get to write this one. There are several conditions you need to be sure to handle. You are *highly encouraged* to draw a memory diagram before you start coding to make sure your thinking is clear.

4.3 Find Iterators

You'll notice the header to the `DList` class contains a strange thing:

```
public class DList<E extends Comparable> { ... }
```

This says that the type `E` must have the `compareTo` method defined for it. It works like this: given two variables `x` and `y`, you can call `x.compareTo(y)` and get one of the following results:

< 0 if $\texttt{x} < \texttt{y}$.

0 if $\texttt{x} = \texttt{y}$.

> 0 if $\texttt{x} > \texttt{y}$.

Use this instead of memory equality for the `Find` iterators.

5 Handing In

To hand in your code, commit your final changes to the repository. We will test your code with our own `JUnit` tests. It is due in one week, once the grading script starts. We will announce when that has happened.