Lab: Circular Lists
CSC 207, “Algorithms and Object-Oriented Design”
Department of Computer Science
Grinnell College
May 4, 2021

Setting Up

The goal of this lab is to see how far we can get with Exercise 17.16 in the textbook, “Implement a circularly and doubly linked list.”

1. Open a new project in Eclipse and create two new classes: a 
CircularList
class and a 
CircularListTester
class for testing the methods that you implement in 
CircularList. Write the header for the definition of the 
CircularList
class, making it a generic class that extends the 
AbstractCollection
class. Note that our implementation of the 
CircularList
does not require sentinel nodes.

Designing the Implementation

Figure 17.18 of the textbook is a box-and-pointer diagram for a circular, doubly-linked list with four elements. It indicates that the class will need at least one field, called first, which will be a reference to one of the nodes in the list.

2. In order to declare the first field, you’ll need to provide a definition of the 
Node
class. Inside the definition of the 
CircularList
class, define a private, static class 
Node
with a data field and (for double linking) prev and next fields that will contain references to adjacent nodes. Provide a three-argument constructor that simply initializes the three fields from the arguments supplied. This class should be generic in the same way as the 
CircularList
class itself. Note that the fields of the 
Node
class are generic.

Nesting the definition of 
Node
inside the definition of 
CircularList
ensures that objects of the 
Node
class cannot be created or used outside of the definition of the 
CircularList
class. As a result, it’s kind of pointless to make the fields of the 
Node
class private and to supply accessor and mutator methods, but you can do so if you like.

3. Add a declaration of the first field to the definition of the 
CircularList
class.

4. How will an empty 
CircularList
be implemented? Define an isEmpty method that tests the appropriate condition and returns the appropriate boolean value. Remember to initialize first field in a zero-argument constructor of the 
CircularList
class.

Implementing the Methods

In order to have 
CircularList
extend the 
AbstractCollection
class, as promised in the header, we have to define the abstract methods from that class, which are size and iterator.

The Java 11 API recommends that we also supply two constructors, one with no arguments that creates an empty 
CircularList
and one that takes as argument any object that implements the 
Collection
interface and creates a 
CircularList
that contains the same elements as the given 
Collection.

Because we want to be able to add elements to instances of 
CircularList, we’ll also want to override the add method, which in the 
AbstractCollection
class itself is implemented as a method that always throws an 
UnsupportedOperationException.

5. Add a field to the definition of the 
CircularList
class that keeps track of the number of elements that the list currently stores. Add a size method that acts as an accessor to this field.

6. Initialize the field inside the zero-argument constructor of 
CircularList.
7. **(2 points)** Implement the `add` method. It would be a good idea to insert the new element so that it immediately precedes the `first` node, if there is one. Because of the circularity of the list, the `add` method can be implemented by insertion at the beginning of the list (if you adjust the value of `first` after the insertion) or insertion at the end (if you don’t change `first`). Note that you will have to treat an empty list as a special case. An insertion into an empty list should result in the `first` field containing a reference to a single node in which the `prev` and the `next` fields both contain references to that very same node.

8. With the help of the `add` method, define the one-argument constructor for `CircularList`. (Hint: The hardest part is probably figuring out how to declare the parameter. The textbook’s implementation of the corresponding constructor for `LinkedList` provides a useful example. You can find that implementation on page 637.)

**Extracting and Removing Elements**

We also want to be able to remove elements from instances of `CircularList`, so we’ll need a `remove` method.

In the `AbstractCollection` class itself, `remove` is implemented as a method that takes one argument, an `Object`, and uses the collection’s iterator to search through the elements of the collection. When and if an element that is equal to the argument is encountered, the iterator’s `remove` method is used to delete it from the collection, and the `AbstractCollection`’s `remove` method returns a boolean that indicates whether the search was successful.

One straightforward way for us to handle this situation is for the `CircularList` class to support two `remove` methods: (a) a private method that takes a `Node` as argument, splices it out of the `CircularList` that contains it, and returns the element in the node’s `data` field, and (b) a public method that takes an `Object` as argument, overriding the implementation that `AbstractCollection` provides, and conducts its own search for the element to be removed (you may use a for-loop or a while-loop for implementing the search). If the search succeeds, method (b) uses method (a) to splice out the node and returns `true`; otherwise, method (b) leaves the list unchanged and returns `false`. Note that `Object` is the element in the `data` field of a `Node` that is removed.

9. **(3 points)** Implement the two `remove` methods.

**Implementing an Iterator**

Extending the `AbstractCollection` class requires us to provide an `iterator` method that, when invoked, constructs and returns an object that can be used to iterate over the elements of a circular list. For this purpose, we can define an inner class called `CircularListIterator` inside the definition of the `CircularList` class (an inner class is a nested non-static class). Our `CircularListIterator` objects will implement the `Iterator` interface in a straightforward way (`Iterator` is a generic interface).

For this class, we need, at the very least, a zero-argument constructor and methods called `hasNext` and `next`. The simplest possible iterator starts with the node to which `first` refers and follows the `next` reference in each node to advance to the node that follows it. Let’s implement this design.

10. Write the header for the definition of the `CircularListIterator` class.

11. The iterator will keep track of its progress through the list by updating a field called `current` containing a reference to the node containing the element that will be returned on the next call to the iterator’s `next` method. Add a declaration for this field to the definition of the `CircularListIterator` class, and write a zero-argument constructor for this class that appropriately initializes this field. To initialize this field, the `CircularListIterator` class should access a field of the `CircularList` class.
12. In a circular list, which has no end markers, how will the iterator know when it is finished? Define the `hasNext` method so that it detects this condition and returns `false` when it is met. `hasNext` also returns false if the circular list is empty.

If a `remove` operation is performed in the middle of the iterator’s traversal of a circular list, it can invalidate the iteration. (Specifically, the iterator’s `current` field may contain a reference to the deleted node.) Weiss’s implementation of the `LinkedList` class manages this problem by maintaining a `modCount` field, initialized to 0 and incremented each time a node is removed from the list. When the iterator is created, it stores the current value of `modCount` in a field of its own (`expectedModCount`). The `hasNext` method can then compare `modCount` to `expectedModCount` to determine whether a deletion has occurred since the iterator was created, in which case it throws a `ConcurrentModificationException`.

13. Add this check, along with the necessary infrastructure, to the `CircularList` and `CircularListIterator` classes.


15. Add an `iterator` method to the definition of `CircularList`. It should simply construct and return an instance of `CircularListIterator`.

Tests

At this point, our implementation is sufficiently far along that we can start adding tests to the `CircularListTester` class.

16. Write and run a test to confirm that the zero-argument constructor for `CircularList` creates an empty circular list.

17. (1 point) Create an empty circular list of `Integer` objects and add 5, 7, and 13 to it. Confirm that its size is 0 before the insertions and 3 afterwards.

18. (1 point) Write a static method that prints out all of the elements in a given `CircularList`. Use it to confirm that the list you created in the previous exercise does indeed contain 5, 7, and 13.

19. (1 point) Remove 7 from the circular list and print it to confirm that 7 has been removed.

20. (1 point) Call the `remove` method, giving it an argument that is not an element of the circular list. Print the circular list afterward and confirm that it has not changed.

21. Create an `ArrayList` containing the elements 16, 21, and 42, then call the one-argument constructor to create a `CircularList` containing the same elements. Print the resulting circular list to make sure that its contents are correct.

22. Call the `iterator` method to obtain an iterator for the circular list you created in the last exercise. Call the iterator’s `next` procedure once—it should return 16—and then remove 21 from the circular list and call the iterator’s `next` procedure again. (This should cause an exception to be thrown, so you may want to enclose the second call to `next` in a `try`-block with a `catch`-clause that handles the expected exception.)