Tutorial 5: Circular Lists

Computer Science 214: Data Structures and Algorithms

6 March 2009 Due: 20 March 2009

Instructions

Your completed tutorial must be submitted as a `jar`, containing your source code, via WebCT by Friday, 20 March 2009 at 14:00. Remember that you are writing a class test on Friday, 13 March 2009, so you do not really have two weeks to complete the tutorial. The usual terms and conditions apply: No late submissions are possible. Should you like to leave the tutorial session before 16:50, the tutorial must be completed in full and demonstrated to the lecturer or one of the assistants. *Note that this course is assessed by continuous evaluation and that all tutorials to be submitted count directly towards your final mark.*

Overview

In this tutorial, you will

- implement a circular list class, adhering to a given interface;
- write code to test this class, paying attention to coverage; and
- start building the GUI that will eventually become your web browser, using the circular list as underlying data structure for keeping tabs.

Tutorial

1. Download the source bundle `cs214-tut05.jar` from WebCT, create a new Java project in Eclipse, and import the source code from the `jar`.
2. Use the same package structure as was used in the source bundle. If you do not follow this structure, you will be penalised severely as it will impede standardised testing of your work.
3. Write a parameterised class `cs214.lists.CircularList⟨E⟩`[1] that implements the parameterised interface `cs214.lists.CircularListADT⟨E⟩`. The defining characteristic of a circular list is that it has a single access point, called the cursor, which may be moved around to point to various list nodes so as to access the data element stored in any particular node.
   (a) Use a variation of the doubly-linked list as the underlying data structure. This is not to say that you have to create a separate class implementing a doubly-linked list. Rather, use the structure of a

[1]For clarity we shall use the fully qualified package path when first mentioning a class.

doubly-linked list directly in your implementation. Since the list is circular, there should be neither header nor trailer nodes—only the cursor keeps a handle on the underlying node list.

   (b) For the nodes of the circular list, create a parameterised class `cs214.lists.CNode⟨E⟩` that implements `cs214.common.Position⟨E⟩`. Base your design on that of a node in a doubly-linked list.

   (c) Implement the iterator over the nodes in the circular list as a local class or as an anonymous local class. The iterator need neither make a snapshot of the list, nor worry about the possibility of changes made to the list while the iterator iterates over the list. That is to say, the iterator does not need to fail fast. Your iterator may not change the state of the list in any way, so you may not change any values, including the cursor. You should handle the `remove()` method of the iterator by throwing an `UnsupportedOperationException()` as only statement in its body.

   (d) When implementing any removal operations, null out all relevant fields in the removed structures so that these may be picked up by the garbage collector.

   (e) Be sure to override the `toString()` method in both `CNode⟨E⟩` and `CircularList⟨E⟩`. In the former case, return something to the effect that it is a node, as well as the string representation of the element stored at the node. In the latter case, return a string representation of the list as a vector. Do not change the state of the list in any way when computing its string representation.

4. Write a class in the package `cs214.tests` to test your implementation of `CircularList⟨E⟩`. You should test each method separately to verify that it works according to the specification. Assuming that your `toString()` methods work correctly, use these to test and debug. Also, be sure to test boundary cases—here, test that your implementation performs as expected, possibly throwing the indicated exceptions, when passed `null` values, when operating on an empty list, or when performing an operation that results in an empty list. In all cases, ensure that your list is in a coherent state, i.e., that there are no references to unlinked nodes, etc.

5. Using your `CircularList`, write a minimal GUI that implements tabbing as found in web browsers like Firefox. You need not render actual HTML pages; use a `JPanel` with a distinguishing `JLabel`—or something similar—to represent a page tab. You may not use `JTabbedPane`; rather, provide four buttons in your window to handle the following operations—note how these buttons correspond to functionality provided by your circular list:

   (a) New to create a new tab;

   (b) Close to close the current tab;

   (c) Next to display the next tab in the list; and

   (d) Previous to display the previous tab in the list.

6. Try to plan your GUI well: Although you will change your GUI considerably as you add functionality, the better you plan now, the less your work

will be later on. Think, always, of what the web browsers you use every day—okay, you may ignore Internet Explorer—look like. For example, it is an excellent idea to have some kind of status area for displaying error and other messages. It is a less excellent idea to use dialog boxes for error conditions—just think how frustrated you will be when your web browser shows a dialog for every single error or broken piece of HTML it encounters. Put the code for your GUI in the package `cs214.gui`, and make it the application entry point of your `jar` file to be submitted.

Background Information and Reading

Refer to §3.3 in your text book for a discussion and implementation of a doubly-linked list. In particular, look at Code Fragment 3.17 for a class representing a node of a doubly-linked list. You may reuse this code as far as possible, but remember that your circular list node must be parameterised, as well as implement the `Position⟨E⟩` interface. Code Fragment 3.23 shows how to null out the references of removed nodes.

Nested and inner classes are discussed in the Java Tutorial. For an introduction, read at least http://star.sun.ac.za/java/docs/tutorial/java/javaOO/nested.html, and consider the example in http://star.sun.ac.za/java/docs/tutorial/java/javaOO/innerclasses.html. Remember: These are introductory remarks, and in any event, we want to use local classes or anonymous local classes. Local classes are discussed at http://docstore.mik.ua/orelly/java-ent/jnut/ch03_11.htm, and anonymous classes in http://docstore.mik.ua/orelly/java-ent/jnut/ch03_12.htm (use `sinetkey`). In particular, note that, if `I` is an interface, the expression syntax

```
new I() {
    class-body
}
```

creates an anonymous local class, with the given *class-body*, that must implement the interface `I`, and also creates an object of that anonymous class. Note that there can be no constructor and that the parameter list after `I` must be empty. In the Swing tutorial of last week, you have already seen such use:

```
public static void main(String[] args) {
    // Schedule a job for the event-dispatching thread:
    // creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
```

The `invokeLater()` has single parameter that must implement the `Runnable` interface. `Runnable` mandates a single method `run()`, which the code above implements in an anonymous local class.

`UnsupportedOperationException` is detailed in the Java API documentation. The Swing classes mentioned are described in both in the Java Tutorial and API documentation.