4.5 PROGRAMMING PROJECT: THE SEQUENCE ADT WITH A LINKED LIST

In Section 3.3 on page 142 we gave a specification for a sequence ADT that was implemented using an array. Now you can reimplement this ADT using a *linked list* as the data structure rather than an array. Start by rereading the ADT's specification on page 148 and then return here for some implementation suggestions.

The Revised Sequence ADT—Design Suggestions

Using a linked list to implement the sequence ADT seems natural. We'll keep the elements stored on a linked list in their sequence order. The "current" element on the list can be maintained by an instance variable that refers to the node that contains the current element. When the `start` method is activated, we set this cursor to refer to the first node of the linked list. When `advance` is activated, we move the cursor to the next node on the linked list.

With this in mind, we propose five private instance variables for the new sequence class:

- The first variable, `manyNodes`, keeps track of the number of nodes in the list.
- `head` and `tail`—These are references to the head and tail nodes of the linked list. If the list has no elements, then these references are both `null`. The reason for the tail reference is the `addAfter` method. Normally this method adds a new element immediately after the current element. But if there is no current element, then `addAfter` places its new element at the tail of the list, so it makes sense to maintain a connection with the list's tail.
- `cursor`—Refers to the node with the current element (or `null` if there is no current element).
- `precursor`—Refers to the node before the current element (or `null` if there is no current element or if the current element is the first node). Can you figure out why we propose a *pre*cursor? The answer is the `addBefore` method, which normally adds a new element immediately *before* the current element. But there is no node method to add a new node before a specified node. We can only add new nodes after a specified node. Therefore, the `addBefore` method will work by adding the new element *after* the precursor node—which is also just *before* the cursor node.

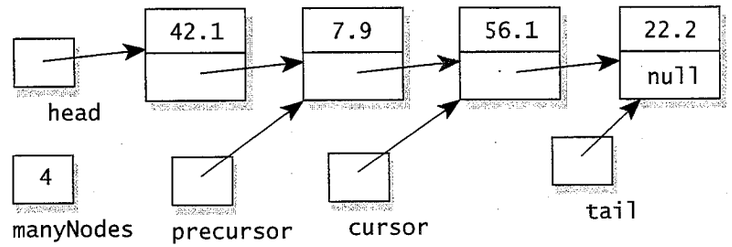
The sequence class you implement could have integer elements, double number elements, or several other possibilities. The choice of element type will determine which kind of node you use for the linked list. If you choose integer elements, then you will use `edu.colorado.nodes.IntNode`. All the node classes, including `IntNode`, are available for you to view at `http://www.cs.colorado.edu/~main/edu/colorado/nodes`.

For this programming project, you should use double numbers for the elements and follow these guidelines:

- The name of the class is `DoubleLinkedSeq`.
- You'll use `DoubleNode` for your node class.
- Put your class in a package `edu.colorado.collections`.
- Follow the specification from Figure 4.18 on page 228. This specification is also available at the `DoubleLinkedSeq` link in
  `http//www.cs.colorado.edu/~main/docs/`

Notice that the specification states a limitation that, beyond `Int.MAX_VALUE` elements, the `size` method does not work (though all other methods should be okay).

Here's an example of the five instance variables for a sequence with four elements and the current element at the third location:



Notice that `cursor` and `precursor` are *references* to two nodes—one right after the other.

what is the invariant of the new list ADT?

Start your implementation by writing the invariant for the new sequence ADT. You might even write the invariant in large letters on a sheet of paper and pin it up in front of you as you work. Each of the methods counts on that invariant being true when the method begins, and each method is responsible for ensuring that the invariant is true when the method finishes.

As you implement each modification method, you might use the following matrix to increase your confidence that the method works correctly.

	`manyNodes`	`head`	`tail`	`cursor`	`precursor`
An empty list					
A nonempty list with no current element					
Current element at the head					
Current element at the tail					
Current element not at head or tail					

Here's how to use the matrix: Suppose you have just implemented one of the modification methods such as `addAfter`. Go through the matrix one row at a time, executing your method with pencil and paper. For example, with the first row of the matrix, you would try `addAfter` to see its behavior for an empty list. As you execute each method by hand, keep track of the five instance variables and put five check marks in the row to indicate that the final values correctly satisfy the invariant.

The Revised Sequence ADT—Clone Method

The sequence class has a `clone` method to make a new copy of a sequence. The sequence you are copying activates the `clone` method, and we'll call it the "original sequence." As with all `clone` methods, you should start with the pattern from page 82. After activating `super.clone`, the extra work must make a separate copy of the linked list for the clone and correctly set the clone's head, tail, cursor, and precursor. We suggest that you handle the work with the following three cases:

- If the original sequence has no current element, then simply copy the original's linked list with `listCopyWithTail`. Then set both `precursor` and `cursor` to null.
- If the current element of the original sequence is its first element, then copy the original's linked list with `listCopyWithTail`. Then set the `precursor` to null and set `cursor` to refer to the head node of the newly created linked list.
- If the current element of the original sequence is after its first element, then copy the original's linked list in two pieces using `listPart`: The first piece goes from the head node to the precursor; the second piece goes from the cursor to the tail. Put these two pieces together by making the link part of the precursor node refer to the cursor node. The reason for copying in two separate pieces is to easily set the precursor and cursor of the newly created list.

FIGURE 4.18 Specification for the Second Version of the `DoubleLinkedSeq` Class

Class DoubleLinkedSeq

❖ **public class DoubleLinkedSeq from the package edu.colorado.collections**

A `DoubleLinkedSeq` is a sequence of double numbers. The sequence can have a special "current element," which is specified and accessed through four methods that are not available in the bag class (`start`, `getCurrent`, `advance`, and `isCurrent`).

Limitations:
Beyond `Int.MAX_VALUE` elements, the `size` method does not work.

Specification

♦ **Constructor for the DoubleLinkedSeq**

`public DoubleLinkedSeq( )`

Initialize an empty sequence.

Postcondition:

(continued)

(FIGURE 4.18 continued)

♦ **addAfter and addBefore**

`public void addAfter(double element)`
`public void addBefore(double element)`

Adds a new element to this sequence either before or after the current element.

Parameters:
`element` – the new element that is being added

Postcondition:
A new copy of the element has been added to this sequence. If there was a current element, `addAfter` places the new element after the current element, and `addBefore` places the new element before the current element. If there was no current element, `addAfter` places the new element at the end of the sequence, and `addBefore` places the new element at the front of the sequence. The new element always becomes the new current element of the sequence.

Throws: `OutOfMemoryError`
Indicates insufficient memory for a new node.

♦ **addAll**

`public void addAll(DoubleLinkedSeq addend)`

Place the contents of another sequence at the end of this sequence.

Parameters:
`addend` – a sequence whose contents will be placed at the end of this sequence

Precondition:
The parameter, `addend`, is not null.

Postcondition:
The elements from `addend` have been placed at the end of this sequence. The current element of this sequence remains where it was, and the `addend` is also unchanged.

Throws: `NullPointerException`
Indicates that `addend` is null.

Throws: `OutOfMemoryError`
Indicates insufficient memory to increase the size of the sequence.

♦ **advance**

`public void advance( )`

Move forward so that the current element is now the next element in the sequence.

Precondition:
`isCurrent( )` returns `true`.

Postcondition:
If the current element was already the end element of the sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

Throws: `IllegalStateException`
Indicates that there is no current element, so `advance` may not be called.

(continued)

(FIGURE 4.18 continued)

♦ **clone**
`public Object clone( )`
Generate a copy of this sequence.
Returns:
The return value is a copy of this sequence. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to a `DoubleLinkedSeq` before it is used.
Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

♦ **concatenation**
`public static DoubleLinkedSeq concatenation (DoubleLinkedSeq s1, DoubleLinkedSeq s2)`
Create a new sequence that contains all the elements from one sequence followed by another.
Parameters:
`s1` – the first of two sequences
`s2` – the second of two sequences
Precondition:
Neither `s1` nor `s2` is null.
Returns:
a new sequence that has the elements of `s1` followed by `s2` (with no current element)
Throws: `IllegalArgumentException`
Indicates that one of the arguments is null.
Throws: `OutOfMemoryError`
Indicates insufficient memory for the new sequence.

♦ **getCurrent**
`public double getCurrent( )`
Accessor method to determine the current element of the sequence.
Precondition:
`isCurrent( )` returns `true`.
Returns:
the current element of the sequence
Throws: `IllegalStateException`
Indicates that there is no current element.

♦ **isCurrent**
`public boolean isCurrent( )`
Accessor method to determine whether this sequence has a specified current element that can be retrieved with the `getCurrent` method.
Returns:
`true` (there is a current element) or `false` (there is no current element at the moment)
(continued)

(FIGURE 4.18 continued)

♦ **removeCurrent**
`public void removeCurrent( )`
Remove the current element from this sequence.
Precondition:
`isCurrent( )` returns `true`.
Postcondition:
The current element has been removed from the sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.
Throws: `IllegalStateException`
Indicates that there is no current element, so `removeCurrent` may not be called.

♦ **size**
`public int size( )`
Accessor method to determine the number of elements in this sequence.
Returns:
the number of elements in this sequence

♦ **start**
`public void start( )`
Set the current element at the front of the sequence.
Postcondition:
The front element of this sequence is now the current element (but if the sequence has no elements at all, then there is no current element).

Self-Test Exercises for Section 4.5

34. Suppose a sequence contains your three favorite numbers, and the current element is the first element. Draw the instance variables of this sequence using our implementation.

35. Write a new method to remove a specified element from a sequence of double numbers. The method has one parameter (the element to remove). After the removal, the current element should be the element after the removed element (if there is one).

36. Which of the sequence methods use the `new` operator to allocate at least one new node?

37. Which of the sequence methods use `DoubleNode.listPart`?