WORKED EXAMPLE 16.1 Implementing a Doubly-Linked List

Problem Statement Provide two enhancements to the linked list implementation from Section 16.1 so that it is a doubly-linked list.

In a doubly-linked list, each node has a reference to the node preceding it, so we will add an instance variable `previous`:

```
class Node
{
   public Object data;
   public Node next;
   public Node previous;
}
```

We will also add a reference to the last node, which speeds up adding and removing elements at the end of the list:

```
public class LinkedList
{
   private Node first;
   private Node last;
   . . .
}
```

We need to revisit all methods of the `LinkedList` and `ListIterator` classes to make sure that these instance variables are properly updated. We will also add methods to add, remove, and get the last element.

Changes in the `LinkedList` Class

In the constructor, we simply add an initialization of the `last` instance variable:

```
public LinkedList()
{
   first = null;
   last = null;
}
```

The `getFirst` method is unchanged. However, in the `removeFirst` method, we need to update the `previous` reference of the node following the one that is being removed.

Moreover, we need to take into account the possibility that the list contains a single element before removal. When that element is removed, then the `last` reference needs to be set to `null`:

```
public Object removeFirst()
{
   if (first == null) { throw new NoSuchElementException(); }
   Object element = first.data;
   first = first.next;
   if (first == null) { last = null; } // List is now empty
   else { first.previous = null; }
   return element;
}
```

In the `addFirst` method, we also need to update the `previous` reference of the node following the added node. Moreover, if the list was previously empty, the new node becomes both the first and the last node:

```
public void addFirst(Object element)
{
```

```
   Node newNode = new Node();
   newNode.data = element;
   newNode.next = first;
   newNode.previous = null;
   if (first == null) { last = newNode; }
   else { first.previous = newNode; }
   first = newNode;
}
```

New Methods for Accessing the Last Element of the List

The `getLast`, `removeLast`, and `addLast` methods are the mirror opposites of the `getFirst`, `removeFirst`, and `addFirst` methods, where the roles of `first`/`last` and `next`/`previous` are switched.

```
public Object getLast()
{
   if (last == null) { throw new NoSuchElementException(); }
   return last.data;
}

public Object removeLast()
{
   if (last == null) { throw new NoSuchElementException(); }
   Object element = last.data;
   last = last.previous;
   if (last == null) { first = null; } // List is now empty
   else { last.next = null; }
   return element;
}

public void addLast(Object element)
{
   Node newNode = new Node();
   newNode.data = element;
   newNode.next = null;
   newNode.previous = last;
   if (last == null) { first = newNode; }
   else { last.next = newNode; }
   last = newNode;
}
```

Compare `removeLast`/`addLast` with the `removeFirst`/`addFirst` methods given above and pay attention to the `first`/`last` and `next`/`previous` references!

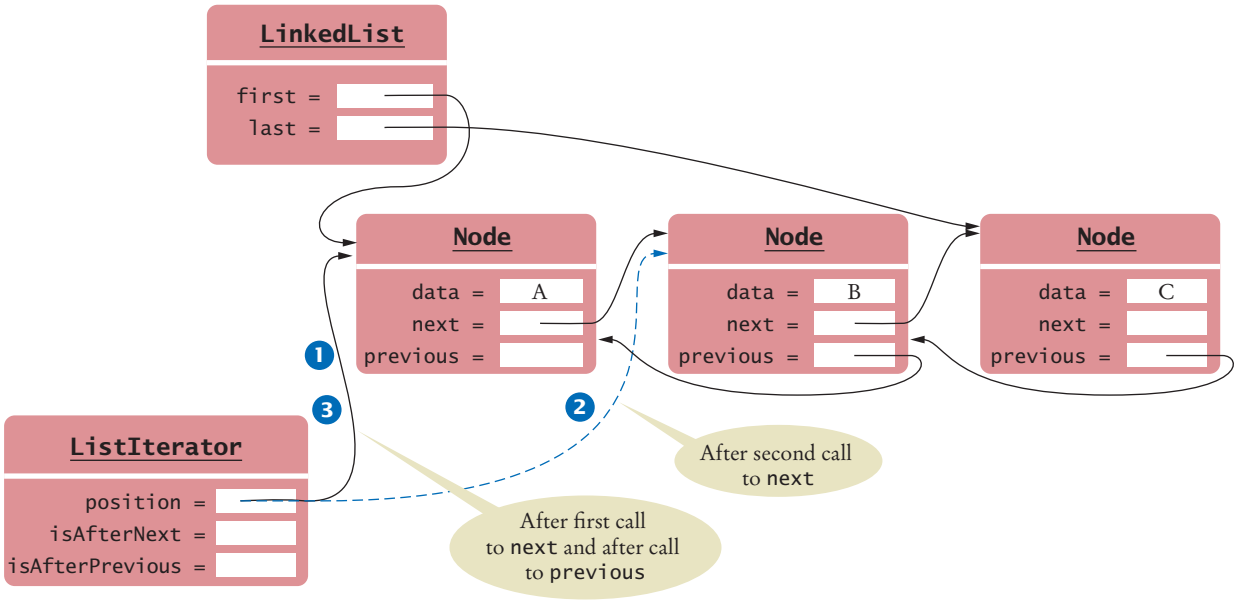
The Bidirectional Iterator

In the `ListIterator` class, we no longer need to store the `previous` reference because we can reach the preceding node as `position.previous`. We can simply remove it from the constructor and the `next` method. (Recall that this reference was required to support the iterator's `remove` operation.)

In a doubly-linked list, the iterator can move forward and backward. For example,

```
LinkedList lst = new LinkedList();
lst.addLast("A");
lst.addLast("B");
lst.addLast("C");
ListIterator iter = lst.listIterator(); // The iterator is before the first element |ABC
iter.next(); // Returns "A"; the iterator is after the first element A|BC  1
iter.next(); // Returns "B"; the iterator is after the second element AB|C  2
iter.previous(); // Returns "B"; the iterator is after the first element A|BC  3
```

The previous method is similar to the next method. However, it returns the value *after* the iterator position. That is perhaps not so intuitive, and it is best to draw a diagram to verify the point. In the figure below, we show two calls to next, followed by a call to previous, as in the code example above. Recall that an iterator conceptually points *between* elements, like the cursor of a word processor, and that the position reference of the iterator points to the element *to the left* (or to null when it is at the beginning of the list).



As you can see, a call to previous moves the iterator backward, and the element that is returned is the one to which it pointed before being moved:

```
public Object previous()
{
   if (!hasPrevious()) { throw new NoSuchElementException(); }
   isAfterNext = false;
   isAfterPrevious = true;

   Object result = position.data;
   position = position.previous;
   return result;
}
```

Removing and Setting Elements Through an Iterator

Note the isAfterNext and isAfterPrevious variables in the previous method. They track whether the iterator just carried out a next or previous call (or neither of the two). This information is needed for implementing the remove and set methods.

These methods remove or set the element that the iterator just traversed, which is position after a call to next or position.next after a call to previous. (If calling previous sets position to null because we reached the front of the list, then we remove or set first.) The following helper method computes this node:

```
private Node lastPosition()
{
   if (isAfterNext)
   {
```

```
         return position;
      }
      else if (isAfterPrevious)
      {
         if (position == null)
         {
            return first;
         }
         else
         {
            return position.next;
         }
      }
      else { throw new IllegalStateException(); }
   }
```

With this helper method, the `set` method is simple:

```
   public void set(Object element)
   {
      Node positionToSet = lastPosition();
      positionToSet.data = element;
   }
```

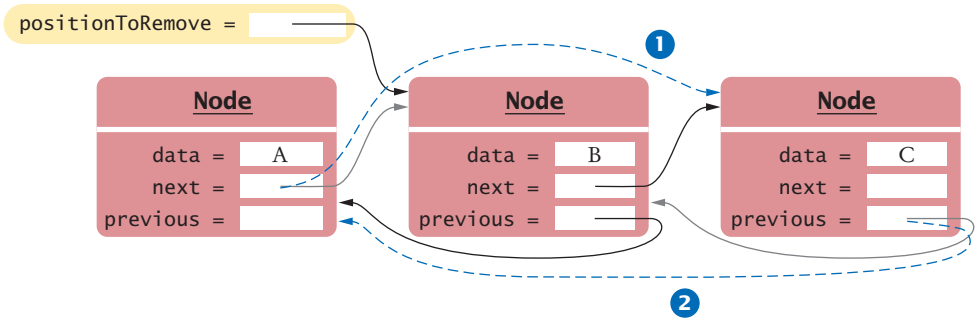
The `remove` method also uses the `lastPosition` helper method. To ensure that the `first` and `last` references are properly updated, we have separate cases for removing the first or last element. Note that the iterator moves one step back when calling `remove` after `next`, and it stays at the same position when calling `remove` after `previous`.

```
   public void remove()
   {
      Node positionToRemove = lastPosition();
      if (positionToRemove == first)
      {
         removeFirst();
      }
      else if (positionToRemove == last)
      {
         removeLast();
      }
      else
      {
         positionToRemove.previous.next = positionToRemove.next; (1)
         positionToRemove.next.previous = positionToRemove.previous; (2)
      }

      if (isAfterNext)
      {
         position = position.previous;
      }

      isAfterNext = false;
      isAfterPrevious = false;
   }
```

The most complex part of this method is the routing of the `next` and `previous` references around the removed elements, which is highlighted above. We know that `positionToRemove.previous` and `positionToRemove.next` are not `null` because we don't remove the first or last element. The following figure shows how the references are updated.



Testing the Implementation

This implementation is so complex that it is unlikely to be implemented correctly at first try. (In fact, I made several errors when I wrote this section.) It is essential to provide a suite of test cases that checks the integrity of all references after every operation, and to test adding and removing elements at either end and in the middle.

Suppose we have a list of strings that should contain nodes for `"A"`, `"B"`, `"C"`, and `"D"`. We can test the `first` and `last` references by verifying that `getFirst` and `getLast` return `"A"` and `"D"`. To check the `next` references of all nodes, we can get an iterator and call the `next` method four times, checking that we get `"A"`, `"B"`, `"C"`, and `"D"`. Then we call `hasNext`, expecting `false`, to check for a `null` in the `next` instance variable of the last node. To check the `previous` references, call `previous` four times on the same iterator and check for `"D"`, `"C"`, `"B"`, and `"A"`. Finally, check that `hasPrevious` returns `false`. These checks ensure that all references are intact.

We provide a test method `check` for this purpose. For example,

```
LinkedList lst = new LinkedList();
check("", lst, "Constructing empty list");
lst.addLast("A");
check("A", lst, "Adding last to empty list");
lst.addLast("B");
check("AB", lst, "Adding last to non-empty list");
```

The `check` method has three arguments: the expected contents (as a string—we assume each node contains a string of length 1), the list, and a string describing the test. The strings are used to print messages such as

```
Passed "Constructing empty list".
Passed "Adding last to empty list".
Passed "Adding last to non-empty list".
```

When implementing the `check` method, we use a helper method `assertEquals` that checks whether an expected value equals an actual one. If it doesn't, an exception is thrown. For example,

```
assertEquals(expected.substring(0, 1), actual.getFirst());
```

You can find the implementation of the `check` and `assertEquals` methods and the provided test cases in the `LinkedListTest` class at the end of this example.

worked_example_1/LinkedList.java

```
1  import java.util.NoSuchElementException;
2
3  /**
4     An implementation of a doubly-linked list.
5  */
```

```
6  public class LinkedList
7  {
8     private Node first;
9     private Node last;
10
11    /**
12       Constructs an empty linked list.
13    */
14    public LinkedList()
15    {
16       first = null;
17       last = null;
18    }
19
20    /**
21       Returns the first element in the linked list.
22       @return the first element in the linked list
23    */
24    public Object getFirst()
25    {
26       if (first == null) { throw new NoSuchElementException(); }
27       return first.data;
28    }
29
30    /**
31       Removes the first element in the linked list.
32       @return the removed element
33    */
34    public Object removeFirst()
35    {
36       if (first == null) { throw new NoSuchElementException(); }
37       Object element = first.data;
38       first = first.next;
39       if (first == null) { last = null; } // List is now empty
40       else { first.previous = null; }
41       return element;
42    }
43
44    /**
45       Adds an element to the front of the linked list.
46       @param element the element to add
47    */
48    public void addFirst(Object element)
49    {
50       Node newNode = new Node();
51       newNode.data = element;
52       newNode.next = first;
53       newNode.previous = null;
54       if (first == null) { last = newNode; }
55       else { first.previous = newNode; }
56       first = newNode;
57    }
58
59    /**
60       Returns the last element in the linked list.
61       @return the last element in the linked list
62    */
63    public Object getLast()
64    {
65       if (last == null) { throw new NoSuchElementException(); }
```

```
66          return last.data;
67       }
68
69       /**
70          Removes the last element in the linked list.
71          @return the removed element
72       */
73       public Object removeLast()
74       {
75          if (last == null) { throw new NoSuchElementException(); }
76          Object element = last.data;
77          last = last.previous;
78          if (last == null) { first = null; } // List is now empty
79          else { last.next = null; }
80          return element;
81       }
82
83       /**
84          Adds an element to the back of the linked list.
85          @param element the element to add
86       */
87       public void addLast(Object element)
88       {
89          Node newNode = new Node();
90          newNode.data = element;
91          newNode.next = null;
92          newNode.previous = last;
93          if (last == null) { first = newNode; }
94          else { last.next = newNode; }
95          last = newNode;
96       }
97
98       /**
99          Returns an iterator for iterating through this list.
100         @return an iterator for iterating through this list
101      */
102      public ListIterator listIterator()
103      {
104         return new LinkedListIterator();
105      }
106
107      class Node
108      {
109         public Object data;
110         public Node next;
111         public Node previous;
112      }
113
114      class LinkedListIterator implements ListIterator
115      {
116         private Node position;
117         private boolean isAfterNext;
118         private boolean isAfterPrevious;
119
120         /**
121            Constructs an iterator that points to the front
122            of the linked list.
123         */
124         public LinkedListIterator()
125         {
```

```
126          position = null;
127          isAfterNext = false;
128          isAfterPrevious = false;
129       }
130
131       /**
132          Moves the iterator past the next element.
133          @return the traversed element
134       */
135       public Object next()
136       {
137          if (!hasNext()) { throw new NoSuchElementException(); }
138          isAfterNext = true;
139          isAfterPrevious = false;
140
141          if (position == null)
142          {
143             position = first;
144          }
145          else
146          {
147             position = position.next;
148          }
149
150          return position.data;
151       }
152
153       /**
154          Tests if there is an element after the iterator position.
155          @return true if there is an element after the iterator position
156       */
157       public boolean hasNext()
158       {
159          if (position == null)
160          {
161             return first != null;
162          }
163          else
164          {
165             return position.next != null;
166          }
167       }
168
169       /**
170          Moves the iterator before the previous element.
171          @return the traversed element
172       */
173       public Object previous()
174       {
175          if (!hasPrevious()) { throw new NoSuchElementException(); }
176          isAfterNext = false;
177          isAfterPrevious = true;
178
179          Object result = position.data;
180          position = position.previous;
181          return result;
182       }
```

```
183
184        /**
185           Tests if there is an element before the iterator position.
186           @return true if there is an element before the iterator position
187        */
188        public boolean hasPrevious()
189        {
190           return position != null;
191        }
192
193        /**
194           Adds an element before the iterator position
195           and moves the iterator past the inserted element.
196           @param element the element to add
197        */
198        public void add(Object element)
199        {
200           if (position == null)
201           {
202              addFirst(element);
203              position = first;
204           }
205           else if (position == last)
206           {
207              addLast(element);
208              position = last;
209           }
210           else
211           {
212              Node newNode = new Node();
213              newNode.data = element;
214              newNode.next = position.next;
215              newNode.next.previous = newNode;
216              position.next = newNode;
217              newNode.previous = position;
218              position = newNode;
219           }
220
221           isAfterNext = false;
222           isAfterPrevious = false;
223        }
224
225        /**
226           Removes the last traversed element. This method may
227           only be called after a call to the next method.
228        */
229        public void remove()
230        {
231           Node positionToRemove = lastPosition();
232
233           if (positionToRemove == first)
234           {
235              removeFirst();
236           }
237           else if (positionToRemove == last)
238           {
239              removeLast();
```

```
240            }
241            else
242            {
243               positionToRemove.previous.next = positionToRemove.next;
244               positionToRemove.next.previous = positionToRemove.previous;
245            }
246
247            if (isAfterNext)
248            {
249               position = position.previous;
250            }
251
252            isAfterNext = false;
253            isAfterPrevious = false;
254         }
255
256         /**
257            Sets the last traversed element to a different value.
258            @param element the element to set
259         */
260         public void set(Object element)
261         {
262            Node positionToSet = lastPosition();
263            positionToSet.data = element;
264         }
265
266         /**
267            Returns the last node traversed by this iterator, or
268            throws an IllegalStateException if there wasn't an immediately
269            preceding call to next or previous.
270            @return the last traversed node
271         */
272         private Node lastPosition()
273         {
274            if (isAfterNext)
275            {
276               return position;
277            }
278            else if (isAfterPrevious)
279            {
280               if (position == null)
281               {
282                  return first;
283               }
284               else
285               {
286                  return position.next;
287               }
288            }
289            else { throw new IllegalStateException(); }
290         }
291      }
292   }
```

worked_example_1/LinkedListTest.java

```
1  import java.util.NoSuchElementException;
2
3  /**
4     This program tests the doubly-linked list implementation.
5  */
6  public class LinkedListTest
7  {
8     public static void main(String[] args)
9     {
10       LinkedList lst = new LinkedList();
11       check("", lst, "Constructing empty list");
12       lst.addLast("A");
13       check("A", lst, "Adding last to empty list");
14       lst.addLast("B");
15       check("AB", lst, "Adding last to non-empty list");
16
17       lst = new LinkedList();
18       lst.addFirst("A");
19       check("A", lst, "Adding first to empty list");
20       lst.addFirst("B");
21       check("BA", lst, "Adding first to non-empty list");
22
23       assertEquals("B", lst.removeFirst());
24       check("A", lst, "Removing first, yielding non-empty list");
25       assertEquals("A", lst.removeFirst());
26       check("", lst, "Removing first, yielding empty list");
27
28       lst = new LinkedList();
29       lst.addLast("A");
30       lst.addLast("B");
31       check("AB", lst, "");
32
33       assertEquals("B", lst.removeLast());
34       check("A", lst, "Removing last, yielding non-empty list");
35       assertEquals("A", lst.removeLast());
36       check("", lst, "Removing last, yielding empty list");
37
38       lst = new LinkedList();
39       lst.addLast("A");
40       lst.addLast("B");
41       lst.addLast("C");
42       check("ABC", lst, "");
43
44       ListIterator iter = lst.listIterator();
45       assertEquals("A", iter.next());
46       iter.set("D");
47       check("DBC", lst, "Set element after next");
48       assertEquals("D", iter.previous());
49       iter.set("E");
50       check("EBC", lst, "Set first element after previous");
51       assertEquals("E", iter.next());
52       assertEquals("B", iter.next());
53       assertEquals("B", iter.previous());
54       iter.set("F");
55       check("EFC", lst, "Set second element after previous");
56       assertEquals("F", iter.next());
57       assertEquals("C", iter.next());
58       assertEquals("C", iter.previous());
```

```
59          iter.set("G");
60          check("EFG", lst, "Set last element after previous");
61
62          lst = new LinkedList();
63          lst.addLast("A");
64          lst.addLast("B");
65          lst.addLast("C");
66          lst.addLast("D");
67          lst.addLast("E");
68          check("ABCDE", lst, "");
69          iter = lst.listIterator();
70          assertEquals("A", iter.next());
71          iter.remove();
72          check("BCDE", lst, "Remove first element after next");
73          assertEquals("B", iter.next());
74          assertEquals("C", iter.next());
75          iter.remove();
76          check("BDE", lst, "Remove middle element after next");
77          assertEquals("D", iter.next());
78          assertEquals("E", iter.next());
79          iter.remove();
80          check("BD", lst, "Remove last element after next");
81
82          lst = new LinkedList();
83          lst.addLast("A");
84          lst.addLast("B");
85          lst.addLast("C");
86          lst.addLast("D");
87          lst.addLast("E");
88          check("ABCDE", lst, "");
89          iter = lst.listIterator();
90          assertEquals("A", iter.next());
91          assertEquals("B", iter.next());
92          assertEquals("C", iter.next());
93          assertEquals("D", iter.next());
94          assertEquals("E", iter.next());
95          assertEquals("E", iter.previous());
96          iter.remove();
97          check("ABCD", lst, "Remove last element after previous");
98          assertEquals("D", iter.previous());
99          assertEquals("C", iter.previous());
100         iter.remove();
101         check("ABD", lst, "Remove middle element after previous");
102         assertEquals("B", iter.previous());
103         assertEquals("A", iter.previous());
104         iter.remove();
105         check("BD", lst, "Remove first element after previous");
106
107         lst = new LinkedList();
108         lst.addLast("B");
109         lst.addLast("C");
110         check("BC", lst, "");
111         iter = lst.listIterator();
112         iter.add("A");
113         check("ABC", lst, "Add first element");
114         assertEquals("B", iter.next());
115         iter.add("D");
116         check("ABDC", lst, "Add middle element");
117         assertEquals("C", iter.next());
```

```
118          iter.add("E");
119          check("ABDCE", lst, "Add last element");
120       }
121
122       /**
123          Checks whether two objects are equal and throws an exception if not.
124          @param expected the expected value
125          @param actual the actual value
126       */
127       public static void assertEquals(Object expected, Object actual)
128       {
129          if (expected == null && actual != null ||
130             !expected.equals(actual))
131          {
132             throw new AssertionError("Expected " + expected + " but found "
133                + actual);
134          }
135       }
136
137       /**
138          Checks whether a linked list has the expected contents, and throws
139          an exception if not.
140          @param expected the letters that are expected in each node
141          @param actual the linked list
142          @param what a string explaining what has been tested. It is included
143          in the message that is displayed when the test passes.
144       */
145       public static void check(String expected, LinkedList actual, String what)
146       {
147          int n = expected.length();
148          if (n > 0)
149          {
150             // Check first and last references
151             assertEquals(expected.substring(0, 1), actual.getFirst());
152             assertEquals(expected.substring(n - 1), actual.getLast());
153
154             // Check next references
155             ListIterator iter = actual.listIterator();
156             for (int i = 0; i < n; i++)
157             {
158                assertEquals(true, iter.hasNext());
159                assertEquals(expected.substring(i, i + 1), iter.next());
160             }
161             assertEquals(false, iter.hasNext());
162
163             // Check previous references
164             for (int i = n - 1 ; i >= 0; i--)
165             {
166                assertEquals(true, iter.hasPrevious());
167                assertEquals(expected.substring(i, i + 1), iter.previous());
168             }
169             assertEquals(false, iter.hasPrevious());
170          }
171          else
172          {
173             // Check that first and last are null
174             try
175             {
176                actual.getFirst();
```

```
177             throw new IllegalStateException("first not null");
178          }
179          catch (NoSuchElementException ex)
180          {
181          }
182
183          try
184          {
185             actual.getLast();
186             throw new IllegalStateException("last not null");
187          }
188          catch (NoSuchElementException ex)
189          {
190          }
191       }
192       if (what.length() > 0)
193       {
194          System.out.println("Passed \"" + what + "\".");
195       }
196    }
197 }
```