```
 1 import java.util.Iterator;
 6
 7 /**
 8  * {@code List} represented as a singly linked list, done "bare-handed", with
 9  * implementations of primary methods.
10  *
11  * <p>
12  * Execution-time performance of all methods implemented in this class is O(1).
13  * </p>
14  *
15  * @param <T>
16  *            type of {@code List} entries
17  * @convention <pre>
18  * $this.leftLength >= 0  and
19  * [$this.rightLength >= 0] and
20  * [$this.preStart is not null]  and
21  * [$this.lastLeft is not null]  and
22  * [$this.finish is not null]  and
23  * [$this.preStart points to the first node of a singly linked list
24  *  containing $this.leftLength + $this.rightLength + 1 nodes]  and
25  * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
26  *  that singly linked list]  and
27  * [$this.finish points to the last node in that singly linked list]  and
28  * [$this.finish.next is null]
29  * </pre>
30  * @correspondence <pre>
31  * this =
32  *  ([data in nodes starting at $this.preStart.next and running through
33  *    $this.lastLeft],
34  *   [data in nodes starting at $this.lastLeft.next and running through
35  *    $this.finish])
36  * </pre>
37  */
38 public class List2a<T> extends ListSecondary<T> {
39
40     /**
41      * Node class for singly linked list nodes.
42      */
43     private final class Node {
44
45         /**
46          * Data in node.
47          */
48         private T data;
49
50         /**
51          * Next node in singly linked list, or null.
52          */
53         private Node next;
54
55     }
56
57     /**
58      * "Smart node" before front node of singly linked list.
59      */
60     private Node preStart;
61
62     /**
63      * Last left node of singly linked list in this.left.
```

```
64      */
65     private Node lastLeft;
66
67     /**
68      * Finish node of linked list.
69      */
70     private Node postFinish;
71
72     /**
73      * Length of this.left.
74      */
75     private int leftLength;
76
77     /**
78      * Length of this.right.
79      */
80     private int rightLength;
81
82     /**
83      * Creator of initial representation.
84      */
85     private void createNewRep() {
86         this.preStart = new Node();
87         this.preStart.next = null;
88         this.postFinish = this.preStart.next;
89         this.lastLeft = this.preStart;
90         this.leftLength = 0;
91         this.rightLength = 0;
92     }
93
94     /**
95      * No-argument constructor.
96      */
97     public List2a() {
98         this.createNewRep();
99     }
100
101    @SuppressWarnings("unchecked")
102    @Override
103    public final List2a<T> newInstance() {
104        try {
105            return this.getClass().getConstructor().newInstance();
106        } catch (ReflectiveOperationException e) {
107            throw new AssertionError(
108                    "Cannot construct object of type " + this.getClass());
109        }
110    }
111
112    @Override
113    public final void clear() {
114        this.createNewRep();
115    }
116
117    @Override
118    public final void transferFrom(List<T> source) {
119        assert source instanceof List2a<?> : ""
120                + "Violation of: source is of dynamic type List2<?>";
121        /*
122         * This cast cannot fail since the assert above would have stopped
```

```
123          * execution in that case: source must be of dynamic type List2<?>, and
124          * the ? must be T or the call would not have compiled.
125          */
126         List2a<T> localSource = (List2a<T>) source;
127         this.preStart = localSource.preStart;
128         this.lastLeft = localSource.lastLeft;
129         this.postFinish = localSource.postFinish;
130         this.rightLength = localSource.rightLength;
131         this.leftLength = localSource.leftLength;
132         localSource.createNewRep();
133     }
134
135     @Override
136     public final void addRightFront(T x) {
137         assert x != null : "Violation of: x is not null";
138         Node p = new Node();
139         Node q = this.lastLeft;
140         p.data = x;
141         p.next = q.next;
142         q.next = p;
143         this.rightLength++;
144     }
145
146     @Override
147     public final T removeRightFront() {
148         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
149         Node p = this.lastLeft;
150         Node q = p.next;
151         p.next = q.next;
152         T x = q.data;
153         this.rightLength--;
154         return x;
155     }
156
157     @Override
158     public final void advance() {
159         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
160         Node p = this.lastLeft;
161         this.lastLeft = p.next;
162         this.leftLength++;
163         this.rightLength--;
164     }
165
166     @Override
167     public final void moveToStart() {
168         this.lastLeft = this.preStart;
169         this.rightLength += this.leftLength;
170         this.leftLength = 0;
171     }
172
173     @Override
174     public final int rightLength() {
175         return this.rightLength;
176     }
177
178     @Override
179     public final int leftLength() {
180         return this.leftLength;
181     }
```

```
182
183     @Override
184     public final Iterator<T> iterator() {
185         return new List2aIterator();
186     }
187
188     /**
189      * Implementation of {@code Iterator} interface for {@code List2}.
190      */
191     private final class List2aIterator implements Iterator<T> {
192
193         /**
194          * Current node in the linked list.
195          */
196         private Node current;
197
198         /**
199          * No-argument constructor.
200          */
201         private List2aIterator() {
202             this.current = List2a.this.preStart.next;
203         }
204
205         @Override
206         public boolean hasNext() {
207             return this.current != null;
208         }
209
210         @Override
211         public T next() {
212             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
213             if (!this.hasNext()) {
214                 /*
215                  * Exception is supposed to be thrown in this case, but with
216                  * assertion-checking enabled it cannot happen because of assert
217                  * above.
218                  */
219                 throw new NoSuchElementException();
220             }
221             T x = this.current.data;
222             this.current = this.current.next;
223             return x;
224         }
225
226         @Override
227         public void remove() {
228             throw new UnsupportedOperationException(
229                     "remove operation not supported");
230         }
231
232     }
233
234     /*
235      * Other methods (overridden for performance reasons) ----------------------
236      */
237
238     @Override
239     public final void moveToFinish() {
240         while (this.lastLeft.next != this.postFinish) {
```

```
241             this.lastLeft = this.lastLeft.next;
242         }
243         this.leftLength += this.rightLength;
244         this.rightLength = 0;
245     }
246 
247 }
248 
```