```
 1 import java.util.Iterator;
 6
 7 /**
 8  * {@code List} represented as a doubly linked list, done "bare-handed", with
 9  * implementations of primary methods and {@code retreat} secondary method.
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  * [$this.postFinish is not null]  and
23  * [$this.preStart points to the first node of a doubly linked list
24  *  containing ($this.leftLength + $this.rightLength + 2) nodes]  and
25  * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
26  *  that doubly linked list]  and
27  * [$this.postFinish points to the last node in that doubly linked list]  and
28  * [for every node n in the doubly linked list of nodes, except the one
29  *  pointed to by $this.preStart, n.previous.next = n]  and
30  * [for every node n in the doubly linked list of nodes, except the one
31  *  pointed to by $this.postFinish, n.next.previous = n]
32  * </pre>
33  * @correspondence <pre>
34  * this =
35  *  ([data in nodes starting at $this.preStart.next and running through
36  *    $this.lastLeft],
37  *   [data in nodes starting at $this.lastLeft.next and running through
38  *    $this.postFinish.previous])
39  * </pre>
40  *
41  * @author Put your name here
42  *
43  */
44 public class List3<T> extends ListSecondary<T> {
45
46     /**
47      * Node class for doubly linked list nodes.
48      */
49     private final class Node {
50
51         /**
52          * Data in node, or, if this is a "smart" Node, irrelevant.
53          */
54         private T data;
55
56         /**
57          * Next node in doubly linked list, or, if this is a trailing "smart"
58          * Node, irrelevant.
59          */
60         private Node next;
61
62         /**
63          * Previous node in doubly linked list, or, if this is a leading "smart"
```

```
64          * Node, irrelevant.
65          */
66         private Node previous;
67
68     }
69
70     /**
71      * "Smart node" before start node of doubly linked list.
72      */
73     private Node preStart;
74
75     /**
76      * Last node of doubly linked list in this.left.
77      */
78     private Node lastLeft;
79
80     /**
81      * "Smart node" after finish node of linked list.
82      */
83     private Node postFinish;
84
85     /**
86      * Length of this.left.
87      */
88     private int leftLength;
89
90     /**
91      * Length of this.right.
92      */
93     private int rightLength;
94
95     /**
96      * Checks that the part of the convention repeated below holds for the
97      * current representation.
98      *
99      * @return true if the convention holds (or if assertion checking is off);
100     *         otherwise reports a violated assertion
101     * @convention <pre>
102     * $this.leftLength >= 0  and
103     * [$this.rightLength >= 0] and
104     * [$this.preStart is not null]  and
105     * [$this.lastLeft is not null]  and
106     * [$this.postFinish is not null]  and
107     * [$this.preStart points to the first node of a doubly linked list
108     *  containing ($this.leftLength + $this.rightLength + 2) nodes]  and
109     * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
110     *  that doubly linked list]  and
111     * [$this.postFinish points to the last node in that doubly linked list]  and
112     * [for every node n in the doubly linked list of nodes, except the one
113     *  pointed to by $this.preStart, n.previous.next = n]  and
114     * [for every node n in the doubly linked list of nodes, except the one
115     *  pointed to by $this.postFinish, n.next.previous = n]
116     * </pre>
117     */
118    private boolean conventionHolds() {
119        assert this.leftLength >= 0 : "Violation of: $this.leftLength >= 0";
120        assert this.rightLength >= 0 : "Violation of: $this.rightLength >= 0";
121        assert this.preStart != null : "Violation of: $this.preStart is not null";
122        assert this.lastLeft != null : "Violation of: $this.lastLeft is not null";
```

```
123         assert this.postFinish != null : "Violation of: $this.postFinish is not null";
124
125         int count = 0;
126         boolean lastLeftFound = false;
127         Node n = this.preStart;
128         while ((count < this.leftLength + this.rightLength + 1)
129                 && (n != this.postFinish)) {
130             count++;
131             if (n == this.lastLeft) {
132                 /*
133                  * Check $this.lastLeft points to the ($this.leftLength + 1)-th
134                  * node in that doubly linked list
135                  */
136                 assert count == this.leftLength + 1 : ""
137                         + "Violation of: [$this.lastLeft points to the"
138                         + " ($this.leftLength + 1)-th node in that doubly linked
    list]";
139                 lastLeftFound = true;
140             }
141             /*
142              * Check for every node n in the doubly linked list of nodes, except
143              * the one pointed to by $this.postFinish, n.next.previous = n
144              */
145             assert (n.next != null) && (n.next.previous == n) : ""
146                     + "Violation of: [for every node n in the doubly linked"
147                     + " list of nodes, except the one pointed to by"
148                     + " $this.postFinish, n.next.previous = n]";
149             n = n.next;
150             /*
151              * Check for every node n in the doubly linked list of nodes, except
152              * the one pointed to by $this.preStart, n.previous.next = n
153              */
154             assert n.previous.next == n : ""
155                     + "Violation of: [for every node n in the doubly linked"
156                     + " list of nodes, except the one pointed to by"
157                     + " $this.preStart, n.previous.next = n]";
158         }
159         count++;
160         assert count == this.leftLength + this.rightLength + 2 : ""
161                 + "Violation of: [$this.preStart points to the first node of"
162                 + " a doubly linked list containing"
163                 + " ($this.leftLength + $this.rightLength + 2) nodes]";
164         assert lastLeftFound : ""
165                 + "Violation of: [$this.lastLeft points to the"
166                 + " ($this.leftLength + 1)-th node in that doubly linked list]";
167         assert n == this.postFinish : ""
168                 + "Violation of: [$this.postFinish points to the last"
169                 + " node in that doubly linked list]";
170
171         return true;
172     }
173
174     /**
175      * Creator of initial representation.
176      */
177     private void createNewRep() {
178         this.preStart = new Node();
179         this.postFinish = new Node();
180         this.lastLeft = this.preStart;
```

```
181         this.preStart.next = this.postFinish;
182         this.postFinish.previous = this.lastLeft;
183
184         this.leftLength = 0;
185         this.rightLength = 0;
186     }
187
188     /**
189      * No-argument constructor.
190      */
191     public List3() {
192         this.createNewRep();
193         assert this.conventionHolds();
194     }
195
196     @SuppressWarnings("unchecked")
197     @Override
198     public final List3<T> newInstance() {
199         try {
200             return this.getClass().getConstructor().newInstance();
201         } catch (ReflectiveOperationException e) {
202             throw new AssertionError(
203                     "Cannot construct object of type " + this.getClass());
204         }
205     }
206
207     @Override
208     public final void clear() {
209         this.createNewRep();
210         assert this.conventionHolds();
211     }
212
213     @Override
214     public final void transferFrom(List<T> source) {
215         assert source instanceof List3<?> : ""
216                 + "Violation of: source is of dynamic type List3<?>";
217         /*
218          * This cast cannot fail since the assert above would have stopped
219          * execution in that case: source must be of dynamic type List3<?>, and
220          * the ? must be T or the call would not have compiled.
221          */
222         List3<T> localSource = (List3<T>) source;
223         this.preStart = localSource.preStart;
224         this.lastLeft = localSource.lastLeft;
225         this.postFinish = localSource.postFinish;
226         this.leftLength = localSource.leftLength;
227         this.rightLength = localSource.rightLength;
228         localSource.createNewRep();
229         assert this.conventionHolds();
230         assert localSource.conventionHolds();
231     }
232
233     @Override
234     public final void addRightFront(T x) {
235         assert x != null : "Violation of: x is not null";
236
237         // create and initialize new node
238         Node p = new Node();
239         p.data = x;
```

```
240
241         // insert p in between lastLeft and lastLeft.next
242         p.next = this.lastLeft.next;
243         p.previous = this.lastLeft;
244         p.next.previous = p;
245         this.lastLeft.next = p;
246
247         // increment length
248         this.rightLength++;
249
250         assert this.conventionHolds();
251     }
252
253     @Override
254     public final T removeRightFront() {
255         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
256
257         // remove this.lastLeft.next and correct pointers
258         Node p = this.lastLeft.next;
259         this.lastLeft.next = p.next;
260         p.next.previous = this.lastLeft;
261
262         // decrement length
263         this.rightLength--;
264
265         assert this.conventionHolds();
266         return p.data;
267     }
268
269     @Override
270     public final void advance() {
271         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
272
273         // move "forward" one node
274         this.lastLeft = this.lastLeft.next;
275
276         // update lengths
277         this.rightLength--;
278         this.leftLength++;
279
280         assert this.conventionHolds();
281     }
282
283     @Override
284     public final void moveToStart() {
285
286         // point lastLeft node to first smart node
287         this.lastLeft = this.preStart;
288
289         //update lengths
290         this.rightLength += this.leftLength;
291         this.leftLength = 0;
292
293         assert this.conventionHolds();
294     }
295
296     @Override
297     public final int leftLength() {
298         assert this.conventionHolds();
```

```
299
300         return this.leftLength;
301     }
302
303     @Override
304     public final int rightLength() {
305         assert this.conventionHolds();
306
307         return this.rightLength;
308     }
309
310     @Override
311     public final Iterator<T> iterator() {
312         assert this.conventionHolds();
313
314         return new List3Iterator();
315     }
316
317     /**
318      * Implementation of {@code Iterator} interface for {@code List3}.
319      */
320     private final class List3Iterator implements Iterator<T> {
321
322         /**
323          * Current node in the linked list.
324          */
325         private Node current;
326
327         /**
328          * No-argument constructor.
329          */
330         private List3Iterator() {
331             this.current = List3.this.preStart.next;
332             assert List3.this.conventionHolds();
333         }
334
335         @Override
336         public boolean hasNext() {
337             return this.current != List3.this.postFinish;
338         }
339
340         @Override
341         public T next() {
342             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
343             if (!this.hasNext()) {
344                 /*
345                  * Exception is supposed to be thrown in this case, but with
346                  * assertion-checking enabled it cannot happen because of assert
347                  * above.
348                  */
349                 throw new NoSuchElementException();
350             }
351             T x = this.current.data;
352             this.current = this.current.next;
353             assert List3.this.conventionHolds();
354             return x;
355         }
356
357         @Override
```

```
358         public void remove() {
359             throw new UnsupportedOperationException(
360                     "remove operation not supported");
361         }
362 
363     }
364 
365     /*
366      * Other methods (overridden for performance reasons) ----------------------
367      */
368 
369     @Override
370     public final void moveToFinish() {
371 
372         // make lastLeft the last node in list
373         this.lastLeft = this.postFinish.previous;
374 
375         // update lengths
376         this.leftLength += this.rightLength;
377         this.rightLength = 0;
378 
379         assert this.conventionHolds();
380     }
381 
382     @Override
383     public final void retreat() {
384         assert this.leftLength() > 0 : "Violation of: this.left /= <>";
385 
386         // move "backward" one node
387         this.lastLeft = this.lastLeft.previous;
388 
389         // update lengths
390         this.rightLength++;
391         this.leftLength--;
392 
393         assert this.conventionHolds();
394     }
395 
396 }
397 
```