

```

1 import java.util.Iterator;
2
3 /**
4  * {@code List} represented as a doubly linked list, done "bare-handed", with
5  * implementations of primary methods and {@code retreat} secondary method.
6  *
7  * <p>
8  * Execution-time performance of all methods implemented in this class is O(1).
9  * </p>
10 *
11 * @param <T>
12 *     type of {@code List} entries
13 * @convention <pre>
14 * $this.leftLength >= 0 and
15 * [$this.rightLength >= 0] and
16 * [$this.preStart is not null] and
17 * [$this.lastLeft is not null] and
18 * [$this.postFinish is not null] and
19 * [$this.preStart points to the first node of a doubly linked list
20 * containing ($this.leftLength + $this.rightLength + 2) nodes] and
21 * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
22 * that doubly linked list] and
23 * [$this.postFinish points to the last node in that doubly linked list] and
24 * [for every node n in the doubly linked list of nodes, except the one
25 * pointed to by $this.preStart, n.previous.next = n] and
26 * [for every node n in the doubly linked list of nodes, except the one
27 * pointed to by $this.postFinish, n.next.previous = n]
28 * </pre>
29 * @correspondence <pre>
30 * this =
31 * ([data in nodes starting at $this.preStart.next and running through
32 *  $this.lastLeft],
33 *  [data in nodes starting at $this.lastLeft.next and running through
34 *  $this.postFinish.previous])
35 * </pre>
36 *
37 * @author David P & Zach
38 */
39
40 public class List3<T> extends ListSecondary<T> {
41
42     /**
43      * Node class for doubly linked list nodes.
44      */
45     private final class Node {
46
47         /**
48          * Data in node, or, if this is a "smart" Node, irrelevant.
49          */
50         private T data;
51
52         /**
53          * Next node in doubly linked list, or, if this is a trailing "smart"
54          * Node, irrelevant.
55          */
56         private Node next;
57     }
58 }

```

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

```

```
233     public final void addRightFront(T x) {
234         assert x != null : "Violation of: x is not null";
235
236         // Create a new node
237         Node newNode = new Node();
238         newNode.data = x;
239         // Insert the new node at the front of the right side
240         newNode.next = this.lastLeft.next;
241         newNode.previous = this.lastLeft;
242         this.lastLeft.next.previous = newNode;
243         this.lastLeft.next = newNode;
244         // Increment the right length
245         this.rightLength++;
246
247         assert this.conventionHolds();
248     }
249
250     @Override
251     public final T removeRightFront() {
252         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
253
254         // Get the node to be remove
255         Node removeNode = this.lastLeft.next;
256         T end = removeNode.data;
257         // Remove node from the list
258         this.lastLeft.next = removeNode.next;
259         removeNode.next.previous = this.lastLeft;
260         // Decrement the right length
261         this.rightLength--;
262
263         assert this.conventionHolds();
264         return end;
265     }
266
267     @Override
268     public final void advance() {
269         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
270
271         // set last left to next last left
272         this.lastLeft = this.lastLeft.next;
273         //set left length to be +1
274         this.leftLength++;
275         // set right length to be -1
276         this.rightLength--;
277
278         assert this.conventionHolds();
279     }
280
281     @Override
282     public final void moveToStart() {
283         // add the left length to the right length
284         this.rightLength += this.leftLength;
285         // reset the left length to zero
286         this.leftLength = 0;
287         // Set the last left pointer to the node
288         this.lastLeft = this.preStart;
289     }
```

```
290     assert this.conventionHolds();
291 }
292
293 @Override
294 public final int leftLength() {
295     assert this.conventionHolds();
296     return this.leftLength;
297 }
298
299 @Override
300 public final int rightLength() {
301     assert this.conventionHolds();
302     return this.rightLength;
303 }
304
305 @Override
306 public final Iterator<T> iterator() {
307     assert this.conventionHolds();
308     return new List3Iterator();
309 }
310
311 /**
312  * Implementation of {@code Iterator} interface for {@code List3}.
313  */
314 private final class List3Iterator implements Iterator<T> {
315
316     /**
317      * Current node in the linked list.
318      */
319     private Node current;
320
321     /**
322      * No-argument constructor.
323      */
324     private List3Iterator() {
325         this.current = List3.this.preStart.next;
326         assert List3.this.conventionHolds();
327     }
328
329     @Override
330     public boolean hasNext() {
331         return this.current != List3.this.postFinish;
332     }
333
334     @Override
335     public T next() {
336         assert this.hasNext() : "Violation of: ~this.unseen != <>";
337         if (!this.hasNext()) {
338             /**
339              * Exception is supposed to be thrown in this case, but with
340              * assertion-checking enabled it cannot happen because of assert
341              * above.
342              */
343             throw new NoSuchElementException();
344         }
345         T x = this.current.data;
346         this.current = this.current.next;
```

```
347         assert List3.this.conventionHolds();
348         return x;
349     }
350
351     @Override
352     public void remove() {
353         throw new UnsupportedOperationException(
354             "remove operation not supported");
355     }
356
357 }
358
359 /*
360  * Other methods (overridden for performance reasons) -----
361  */
362
363 @Override
364 public final void moveToFinish() {
365     // add the right length to the left length
366     this.leftLength += this.rightLength;
367     // reset the right length to zero
368     this.rightLength = 0;
369     // set the last left pointer to the node
370     this.lastLeft = this.postFinish.previous;
371
372     assert this.conventionHolds();
373 }
374
375 @Override
376 public final void retreat() {
377     assert this.leftLength() > 0 : "Violation of: this.left != <>";
378
379     // move to previous node on the left side
380     this.lastLeft = this.lastLeft.previous;
381     // dec the length of the left side
382     this.leftLength--;
383     // inc the length of the right side
384     this.rightLength++;
385
386     assert this.conventionHolds();
387 }
388
389 }
390
```