```java
import java.util.Iterator;

/**
 * {@code List} represented as a doubly linked list, done "bare-handed", with
 * implementations of primary methods and {@code retreat} secondary method.
 *
 * <p>
 * Execution-time performance of all methods implemented in this class is O(1).
 * </p>
 *
 * @param <T>
 *            type of {@code List} entries
 * @convention <pre>
 * $this.leftLength >= 0  and
 * [$this.rightLength >= 0] and
 * [$this.preStart is not null]  and
 * [$this.lastLeft is not null]  and
 * [$this.postFinish is not null]  and
 * [$this.preStart points to the first node of a doubly linked list
 *  containing ($this.leftLength + $this.rightLength + 2) nodes] and
 * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
 *  that doubly linked list]  and
 * [$this.postFinish points to the last node in that doubly linked list]  and
 * [for every node n in the doubly linked list of nodes, except the one
 *  pointed to by $this.preStart, n.previous.next = n]  and
 * [for every node n in the doubly linked list of nodes, except the one
 *  pointed to by $this.postFinish, n.next.previous = n]
 * </pre>
 * @correspondence <pre>
 * this =
 *  ([data in nodes starting at $this.preStart.next and running through
 *    $this.lastLeft],
 *   [data in nodes starting at $this.lastLeft.next and running through
 *    $this.postFinish.previous])
 * </pre>
 *
```

```
41  * @author Evan Frisbie, Charan Nanduri, Sarina Mathis
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
179         // TODO - fill in body
180
181     }
182
183     /**
184      * No-argument constructor.
185      */
186     public List3() {
187
188         // TODO - fill in body
189
190         assert this.conventionHolds();
```

```
191     }
192
193     @SuppressWarnings("unchecked")
194     @Override
195     public final List3<T> newInstance() {
196         try {
197             return this.getClass().getConstructor().newInstance();
198         } catch (ReflectiveOperationException e) {
199             throw new AssertionError(
200                     "Cannot construct object of type " +
    this.getClass());
201         }
202     }
203
204     @Override
205     public final void clear() {
206         this.createNewRep();
207         assert this.conventionHolds();
208     }
209
210     @Override
211     public final void transferFrom(List<T> source) {
212         assert source instanceof List3<?> : ""
213                 + "Violation of: source is of dynamic type List3<?
    >";
214         /*
215          * This cast cannot fail since the assert above would have
    stopped
216          * execution in that case: source must be of dynamic type
    List3<?>, and
217          * the ? must be T or the call would not have compiled.
218          */
219         List3<T> localSource = (List3<T>) source;
220         this.preStart = localSource.preStart;
221         this.lastLeft = localSource.lastLeft;
222         this.postFinish = localSource.postFinish;
223         this.leftLength = localSource.leftLength;
224         this.rightLength = localSource.rightLength;
225         localSource.createNewRep();
226         assert this.conventionHolds();
227         assert localSource.conventionHolds();
228     }
229
230     @Override
231     public final void addRightFront(T x) {
```

```
232         assert x != null : "Violation of: x is not null";
233
234         // TODO - fill in body
235         Node newNode = new Node();
236         Node n2 = this.lastLeft;
237
238         // Assign the data
239         newNode.data = x;
240
241         // Adjust the pointers
242         newNode.next = n2.next;
243         n2.next = newNode;
244         newNode.previous = n2;
245         newNode.next.previous = newNode;
246
247         // Update 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         // TODO - fill in body
258         Node lastleft = this.lastLeft;
259         Node n2 = lastleft.next;
260         T val = n2.data;
261         // remove node and relink
262         Node n3 = n2.next;
263         n3.previous = n2.previous;
264         lastleft.next = n3;
265         // update length
266         this.rightLength--;
267
268         assert this.conventionHolds();
269         // Fix this line to return the result after checking the
    convention.
270         return null;
271     }
272
273     @Override
274     public final void advance() {
```

```
275         assert this.rightLength() > 0 : "Violation of: this.right /
    = <>";
276
277         // TODO - fill in body
278         // LastLeft is like our focus node. So we want to move our
    focus to the
279         // next value which we get from our right side list. So
    pull one from
280         // the right and add to left
281
282         // Redirect the pointer
283         this.lastLeft = this.lastLeft.next;
284
285         // Adjust the lengths of both lists
286         this.rightLength--;
287         this.leftLength++;
288
289         assert this.conventionHolds();
290     }
291
292     @Override
293     public final void moveToStart() {
294
295         // TODO - fill in body
296         // Here we are moving our focus value, lastLeft, to the
    front of the
297         // list
298
299         // Move our pointer to the front
300         this.lastLeft = this.preStart;
301
302         // The right length is all the values
303         this.rightLength = this.leftLength + this.rightLength;
304
305         // Now the left Length is 0
306         this.leftLength = 0;
307
308         assert this.conventionHolds();
309     }
310
311     @Override
312     public final int leftLength() {
313
314         // TODO - fill in body
315
```

```
316         assert this.conventionHolds();
317         // Fix this line to return the result after checking the
    convention.
318         return this.leftLength;
319     }
320 
321     @Override
322     public final int rightLength() {
323 
324         // TODO - fill in body
325 
326         assert this.conventionHolds();
327         // Fix this line to return the result after checking the
    convention.
328         return this.rightLength;
329     }
330 
331     @Override
332     public final Iterator<T> iterator() {
333         assert this.conventionHolds();
334         return new List3Iterator();
335     }
336 
337     /**
338      * Implementation of {@code Iterator} interface for {@code
    List3}.
339      */
340     private final class List3Iterator implements Iterator<T> {
341 
342         /**
343          * Current node in the linked list.
344          */
345         private Node current;
346 
347         /**
348          * No-argument constructor.
349          */
350         private List3Iterator() {
351             this.current = List3.this.preStart.next;
352             assert List3.this.conventionHolds();
353         }
354 
355         @Override
356         public boolean hasNext() {
357             return this.current != List3.this.postFinish;
```

```
358         }
359
360         @Override
361         public T next() {
362             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
363             if (!this.hasNext()) {
364                 /*
365                  * Exception is supposed to be thrown in this case,
    but with
366                  * assertion-checking enabled it cannot happen
    because of assert
367                  * above.
368                  */
369                 throw new NoSuchElementException();
370             }
371             T x = this.current.data;
372             this.current = this.current.next;
373             assert List3.this.conventionHolds();
374             return x;
375         }
376
377         @Override
378         public void remove() {
379             throw new UnsupportedOperationException(
380                     "remove operation not supported");
381         }
382
383     }
384
385     /*
386      * Other methods (overridden for performance reasons)
    ----------------------
387      */
388
389     @Override
390     public final void moveToFinish() {
391
392         // TODO - fill in body
393         // Here we are moving our focus to the end and putting all
    values in the
394         // left list
395
396         // Move the pointer to the one before the last one since we
    are still in
```

```
397         // the left list and there will still be a value, lastLeft
    has to have a
398         // Value
399
400         // Adjust the left length
401         this.leftLength = this.rightLength + this.leftLength;
402
403         // The right length is now the one that is zero
404         this.rightLength = 0;
405         this.lastLeft = this.postFinish.previous;
406
407         assert this.conventionHolds();
408     }
409
410     @Override
411     public final void retreat() {
412         assert this.leftLength() > 0 : "Violation of: this.left /=
    <>";
413
414         // TODO - fill in body
415         this.lastLeft.next = this.lastLeft;
416
417         // Here, we are going right to left. So LastLeft ->
    lastLeft.previous
418         this.lastLeft = this.lastLeft.previous;
419
420         //Adjust the lengths accordingly
421         this.leftLength--;
422         this.rightLength++;
423
424         assert this.conventionHolds();
425     }
426
427 }
428
```