```java
import java.util.Iterator;
import java.util.NoSuchElementException;

import components.list.List;
import components.list.ListSecondary;

/**
 * {@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>
 *
 * @author Qinuo Shi & Yiming Cheng
 *
 */
public class List3<T> extends ListSecondary<T> {

    /**
     * Node class for doubly linked list nodes.
     */
    private final class Node {

        /**
         * Data in node, or, if this is a "smart" Node, irrelevant.
         */
        private T data;

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

```
229         this.preStart = localSource.preStart;
230         this.lastLeft = localSource.lastLeft;
231         this.postFinish = localSource.postFinish;
232         this.leftLength = localSource.leftLength;
233         this.rightLength = localSource.rightLength;
234         localSource.createNewRep();
235         assert this.conventionHolds();
236         assert localSource.conventionHolds();
237     }
238 
239     @Override
240     public final void addRightFront(T x) {
241         assert x != null : "Violation of: x is not null";
242 
243         // TODO - fill in body
244         Node p = new Node();
245         p.data = x;
246         p.previous = this.lastLeft;
247         p.next = this.lastLeft.next;
248         this.lastLeft.next = p;
249         p.next.previous = p;
250         this.rightLength++;
251 
252         assert this.conventionHolds();
253     }
254 
255     @Override
256     public final T removeRightFront() {
257         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
258 
259         // TODO - fill in body
260         //find the right node to remove
261         Node x = this.lastLeft.next;
262         this.lastLeft.next = x.next;
263         this.lastLeft.next.previous = this.lastLeft;
264         //ensure the correct lengths of the right part of the list
265         this.rightLength--;
266 
267         assert this.conventionHolds();
268         // Fix this line to return the result after checking the convention.
269         return x.data;
270     }
271 
272     @Override
273     public final void advance() {
274         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
275 
276         // TODO - fill in body
277         // find the node that need to move
278         this.lastLeft = this.lastLeft.next;
279         //ensure the correct lengths of the both parts of the list
280         this.rightLength--;
281         this.leftLength++;
282 
283         assert this.conventionHolds();
284     }
285 
```

```
286     @Override
287     public final void moveToStart() {
288 
289         // TODO - fill in body
290         // move the left part to the right
291         this.lastLeft = this.preStart;
292         //ensure the correct lengths of the right part of the list
293         this.rightLength += this.leftLength;
294         this.leftLength = 0;
295 
296         assert this.conventionHolds();
297     }
298 
299     @Override
300     public final int leftLength() {
301 
302         // TODO - fill in body
303 
304         assert this.conventionHolds();
305         // Fix this line to return the result after checking the convention.
306         return this.leftLength;
307     }
308 
309     @Override
310     public final int rightLength() {
311 
312         // TODO - fill in body
313 
314         assert this.conventionHolds();
315         // Fix this line to return the result after checking the convention.
316         return this.rightLength;
317     }
318 
319     @Override
320     public final Iterator<T> iterator() {
321         assert this.conventionHolds();
322         return new List3Iterator();
323     }
324 
325     /**
326      * Implementation of {@code Iterator} interface for {@code List3}.
327      */
328     private final class List3Iterator implements Iterator<T> {
329 
330         /**
331          * Current node in the linked list.
332          */
333         private Node current;
334 
335         /**
336          * No-argument constructor.
337          */
338         private List3Iterator() {
339             this.current = List3.this.preStart.next;
340             assert List3.this.conventionHolds();
341         }
342 
```

```
343         @Override
344         public boolean hasNext() {
345             return this.current != List3.this.postFinish;
346         }
347 
348         @Override
349         public T next() {
350             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
351             if (!this.hasNext()) {
352                 /*
353                  * Exception is supposed to be thrown in this case, but with
354                  * assertion-checking enabled it cannot happen because of assert
355                  * above.
356                  */
357                 throw new NoSuchElementException();
358             }
359             T x = this.current.data;
360             this.current = this.current.next;
361             assert List3.this.conventionHolds();
362             return x;
363         }
364 
365         @Override
366         public void remove() {
367             throw new UnsupportedOperationException(
368                     "remove operation not supported");
369         }
370 
371     }
372 
373     /*
374      * Other methods (overridden for performance reasons) ----------------------
375      */
376 
377     @Override
378     public final void moveToFinish() {
379 
380         // TODO - fill in body
381         this.lastLeft = this.postFinish.previous;
382         this.leftLength += this.rightLength;
383         this.rightLength = 0;
384 
385         assert this.conventionHolds();
386     }
387 
388     @Override
389     public final void retreat() {
390         assert this.leftLength() > 0 : "Violation of: this.left /= <>";
391 
392         // TODO - fill in body
393         this.lastLeft = this.lastLeft.previous;
394         this.rightLength++;
395         this.leftLength--;
396 
397         assert this.conventionHolds();
398     }
399 
```

```
400 }
401
```