```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>
```

```
 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 Zhuoyang Li + Xinci Ma
 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         // initialize the "smart" nodes
180         this.preStart = new Node();
181         this.postFinish = new Node();
182
183         // link the "smart" nodes
184         this.preStart.next = this.postFinish;
185         this.postFinish.previous = this.preStart;
186
187         // initial lengths to 0
188         this.leftLength = 0;
189         this.rightLength = 0;
190
191         // lastLeft is preStart since no elements in
    left part
192         this.lastLeft = this.preStart;
193
194     }
195
196     /**
197      * No-argument constructor.
198      */
199     public List3() {
200
201         this.createNewRep();
202
203         assert this.conventionHolds();
204     }
205
```

```
206     @SuppressWarnings("unchecked")
207     @Override
208     public final List3<T> newInstance() {
209         try {
210             return
    this.getClass().getConstructor().newInstance();
211         } catch (ReflectiveOperationException e) {
212             throw new AssertionError(
213                     "Cannot construct object of type " +
    this.getClass());
214         }
215     }
216
217     @Override
218     public final void clear() {
219         this.createNewRep();
220         assert this.conventionHolds();
221     }
222
223     @Override
224     public final void transferFrom(List<T> source) {
225         assert source instanceof List3<?> : ""
226                 + "Violation of: source is of dynamic
    type List3<?>";
227         /*
228          * This cast cannot fail since the assert above
    would have stopped
229          * execution in that case: source must be of
    dynamic type List3<?>, and
230          * the ? must be T or the call would not have
    compiled.
231          */
232         List3<T> localSource = (List3<T>) source;
233         this.preStart = localSource.preStart;
234         this.lastLeft = localSource.lastLeft;
235         this.postFinish = localSource.postFinish;
236         this.leftLength = localSource.leftLength;
237         this.rightLength = localSource.rightLength;
238         localSource.createNewRep();
239         assert this.conventionHolds();
240         assert localSource.conventionHolds();
241     }
242
```

```
243     @Override
244     public final void addRightFront(T x) {
245         assert x != null : "Violation of: x is not
    null";
246 
247         Node newNode = new Node();
248         newNode.data = x;
249 
250         newNode.next = this.lastLeft.next;
251         // point to first node in right or postFinish if
    right is empty
252         newNode.previous = this.lastLeft; // point back
    to lastLeft
253 
254         // adjust the links of surrounding nodes
255         this.lastLeft.next.previous = newNode;
256         this.lastLeft.next = newNode;
257 
258         // increase rightLength since we add element to
    right
259         this.rightLength++;
260 
261         assert this.conventionHolds();
262     }
263 
264     @Override
265     public final T removeRightFront() {
266         assert this.rightLength() > 0 : "Violation of:
    this.right /= <>";
267 
268         Node nodeToRemove = this.lastLeft.next;
269         T data = nodeToRemove.data;
270 
271         // adjust links to bypass the nodeToRemove
272         this.lastLeft.next = nodeToRemove.next;
273         nodeToRemove.next.previous = this.lastLeft;
274 
275         // decrease rightLength after removal
276         this.rightLength--;
277 
278         assert this.conventionHolds();
279         return data;
280     }
```

```
281
282     @Override
283     public final void advance() {
284         assert this.rightLength() > 0 : "Violation of:
    this.right /= <>";
285
286         this.lastLeft = this.lastLeft.next; // move
    divider one node to the right
287         this.leftLength++;
288         this.rightLength--;
289
290         assert this.conventionHolds();
291     }
292
293     @Override
294     public final void moveToStart() {
295
296         // move all elements to the right
297         this.rightLength += this.leftLength;
298         this.leftLength = 0;
299
300         // reset lastLeft to preStart
301         this.lastLeft = this.preStart;
302
303         assert this.conventionHolds();
304     }
305
306     @Override
307     public final int leftLength() {
308
309         // TODO - fill in body
310
311         assert this.conventionHolds();
312         return this.leftLength;
313     }
314
315     @Override
316     public final int rightLength() {
317
318         // TODO - fill in body
319
320         assert this.conventionHolds();
321         return this.rightLength;
```

```
322     }
323
324     @Override
325     public final Iterator<T> iterator() {
326         assert this.conventionHolds();
327         return new List3Iterator();
328     }
329
330     /**
331      * Implementation of {@code Iterator} interface for
    {@code List3}.
332      */
333     private final class List3Iterator implements
    Iterator<T> {
334
335         /**
336          * Current node in the linked list.
337          */
338         private Node current;
339
340         /**
341          * No-argument constructor.
342          */
343         private List3Iterator() {
344             this.current = List3.this.preStart.next;
345             assert List3.this.conventionHolds();
346         }
347
348         @Override
349         public boolean hasNext() {
350             return this.current !=
    List3.this.postFinish;
351         }
352
353         @Override
354         public T next() {
355             assert this.hasNext() : "Violation of:
    ~this.unseen /= <>";
356             if (!this.hasNext()) {
357                 /*
358                  * Exception is supposed to be thrown in
    this case, but with
359                  * assertion-checking enabled it cannot
```

```
    happen because of assert
360                  * above.
361                  */
362                 throw new NoSuchElementException();
363             }
364             T x = this.current.data;
365             this.current = this.current.next;
366             assert List3.this.conventionHolds();
367             return x;
368         }
369
370         @Override
371         public void remove() {
372             throw new UnsupportedOperationException(
373                     "remove operation not supported");
374         }
375
376     }
377
378     /*
379      * Other methods (overridden for performance
    reasons) ----------------------
380      */
381
382     @Override
383     public final void moveToFinish() {
384
385         // move all to the left
386         this.leftLength += this.rightLength;
387         this.rightLength = 0;
388
389         // set lastLeft to node before postFinish
390         this.lastLeft = this.postFinish.previous;
391
392         assert this.conventionHolds();
393     }
394
395     @Override
396     public final void retreat() {
397         assert this.leftLength() > 0 : "Violation of:
    this.left /= <>";
398
399         // move divider one node to the left
```

```
400         this.lastLeft = this.lastLeft.previous;
401         this.leftLength--;
402         this.rightLength++;
403 
404         assert this.conventionHolds();
405     }
406 
407 }
408 
```