```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
```

```
 36  *    $this.lastLeft],
 37  *   [data in nodes starting at $this.lastLeft.next and running
     through
 38  *    $this.postFinish.previous])
 39  * </pre>
 40  *
 41  * @author Shyam Sai Bethina and Yihone Chu
 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         /*
180          * Both preStart and postFinish are new nodes pointing to
    each other.
181          */
182         this.preStart = new Node();
183         this.postFinish = new Node();
184 
185         /*
186          * preStart's next points to postFinish, and postFinish's
    previous
187          * points to preStart, making them point to each other.
188          */
189         this.preStart.next = this.postFinish;
190 
191         this.postFinish.previous = this.preStart;
192 
193         /*
194          * lastLeft is equal to preStart since there are no nodes
    in the left
195          * string.
196          */
197         this.lastLeft = this.preStart;
198 
199         /*
200          * The leftLength and rightLength equals 0 since there are
    no elements
201          * in the left and right strings.
202          */
203         this.leftLength = 0;
204         this.rightLength = 0;
205     }
206 
207     /**
208      * No-argument constructor.
209      */
210     public List3() {
211 
212         /*
213          * Creates a new representation using createNewRep().
214          */
215         this.createNewRep();
216 
217         assert this.conventionHolds();
218     }
```

```
219
220     @SuppressWarnings("unchecked")
221     @Override
222     public final List3<T> newInstance() {
223         try {
224             return this.getClass().getConstructor().newInstance();
225         } catch (ReflectiveOperationException e) {
226             throw new AssertionError(
227                     "Cannot construct object of type " +
    this.getClass());
228         }
229     }
230
231     @Override
232     public final void clear() {
233         this.createNewRep();
234         assert this.conventionHolds();
235     }
236
237     @Override
238     public final void transferFrom(List<T> source) {
239         assert source instanceof List3<?> : ""
240                 + "Violation of: source is of dynamic type List3<?
    >";
241         /*
242          * This cast cannot fail since the assert above would have
    stopped
243          * execution in that case: source must be of dynamic type
    List3<?>, and
244          * the ? must be T or the call would not have compiled.
245          */
246         List3<T> localSource = (List3<T>) source;
247         this.preStart = localSource.preStart;
248         this.lastLeft = localSource.lastLeft;
249         this.postFinish = localSource.postFinish;
250         this.leftLength = localSource.leftLength;
251         this.rightLength = localSource.rightLength;
252
253         localSource.createNewRep();
254
255         assert this.conventionHolds();
256         assert localSource.conventionHolds();
257     }
258
```

```
259     @Override
260     public final void addRightFront(T x) {
261         assert x != null : "Violation of: x is not null";
262 
263         /*
264          * Creates a new node to add into the right string. The
    data of the new
265          * node is equal to x, and the oldRight node is equal to
    the node next
266          * to lastLeft, which is the current front of the right
    string.
267          */
268         Node newRight = new Node();
269         newRight.data = x;
270         Node oldRight = this.lastLeft.next;
271 
272         /*
273          * The newRight's previous points towards lastLeft, and
    the oldRight's
274          * previous points to the newRight.
275          */
276         newRight.previous = this.lastLeft;
277         oldRight.previous = newRight;
278 
279         /*
280          * newRight's next is the oldRight, and the lastLeft
    node's next points
281          * to newRight node.
282          */
283         newRight.next = oldRight;
284         this.lastLeft.next = newRight;
285 
286         /*
287          * Increments rightLength to reflect the added node.
288          */
289         this.rightLength++;
290         assert this.conventionHolds();
291     }
292 
293     @Override
294     public final T removeRightFront() {
295         assert this.rightLength() > 0 : "Violation of:
    this.right /= <>";
296 
```

```
297         /*
298          * Creates a new node to remove from the right string. The
    data of the
299          * new node is equal to answer, and the firstRight node is
    equal to the
300          * node next to lastLeft. newRight node is equal to the
    node after
301          * firstRight.
302          */
303         Node firstRight = this.lastLeft.next;
304         Node newRight = firstRight.next;
305 
306         T answer = firstRight.data;
307 
308         /*
309          * newRight's prevous points to lastLeft now.
310          */
311         newRight.previous = this.lastLeft;
312 
313         /*
314          * lastLeft.next points to the newRight, after firstRight
    is removed.
315          */
316         this.lastLeft.next = newRight;
317 
318         /*
319          * rightLength is decreased to reflect the removed node.
320          */
321         this.rightLength--;
322 
323         assert this.conventionHolds();
324         // Fix this line to return the result after checking the
    convention.
325         return answer;
326     }
327 
328     @Override
329     public final void advance() {
330         assert this.rightLength() > 0 : "Violation of:
    this.right /= <>";
331 
332         /*
333          * lastLeft points to the next node to move the
    "partition" to the right
```

```
334          * by one space.
335          */
336         this.lastLeft = this.lastLeft.next;
337 
338         /*
339          * increments leftLength and decrements rightLength to
    reflect the new
340          * changes.
341          */
342         this.leftLength++;
343         this.rightLength--;
344         assert this.conventionHolds();
345     }
346 
347     @Override
348     public final void moveToStart() {
349 
350         /*
351          * lastLeft is now equal to preStart to make all the nodes
    in the left
352          * string move to the right string.
353          */
354         this.lastLeft = this.preStart;
355 
356         /*
357          * Changes rightLength to reflect the changes, and sets
    leftLength to
358          * nothing.
359          */
360         this.rightLength += this.leftLength;
361         this.leftLength = 0;
362 
363         assert this.conventionHolds();
364     }
365 
366     @Override
367     public final int leftLength() {
368 
369         /*
370          * Sets answer to the length of the left string and
    returns answer.
371          */
372         int answer = this.leftLength;
373 
```

```
374         assert this.conventionHolds();
375         // Fix this line to return the result after checking the
    convention.
376         return answer;
377     }
378 
379     @Override
380     public final int rightLength() {
381         /*
382          * Sets answer to the length of the right string and
    returns answer.
383          */
384         int answer = this.rightLength;
385 
386         assert this.conventionHolds();
387         // Fix this line to return the result after checking the
    convention.
388         return answer;
389     }
390 
391     @Override
392     public final Iterator<T> iterator() {
393         assert this.conventionHolds();
394         return new List3Iterator();
395     }
396 
397     /**
398      * Implementation of {@code Iterator} interface for {@code
    List3}.
399      */
400     private final class List3Iterator implements Iterator<T> {
401 
402         /**
403          * Current node in the linked list.
404          */
405         private Node current;
406 
407         /**
408          * No-argument constructor.
409          */
410         private List3Iterator() {
411             this.current = List3.this.preStart.next;
412             assert List3.this.conventionHolds();
413         }
```

```
414
415         @Override
416         public boolean hasNext() {
417             return this.current != List3.this.postFinish;
418         }
419
420         @Override
421         public T next() {
422             assert this.hasNext() : "Violation of: ~this.unseen /=
    <>";
423             if (!this.hasNext()) {
424                 /*
425                  * Exception is supposed to be thrown in this
    case, but with
426                  * assertion-checking enabled it cannot happen
    because of assert
427                  * above.
428                  */
429                 throw new NoSuchElementException();
430             }
431             T x = this.current.data;
432             this.current = this.current.next;
433             assert List3.this.conventionHolds();
434             return x;
435         }
436
437         @Override
438         public void remove() {
439             throw new UnsupportedOperationException(
440                     "remove operation not supported");
441         }
442
443     }
444
445     /*
446      * Other methods (overridden for performance reasons)
    ----------------------
447      */
448
449     @Override
450     public final void moveToFinish() {
451
452         /*
453          * lastLeft is now equal to node before postFinish to make
```

```
    all the nodes
454          * in the right string move to the left string.
455          */
456         this.lastLeft = this.postFinish.previous;
457 
458         /*
459          * Changes leftLength to reflect the changes, and sets
    rightLength to
460          * nothing.
461          */
462         this.leftLength = this.leftLength + this.rightLength;
463         this.rightLength = 0;
464 
465         assert this.conventionHolds();
466     }
467 
468     @Override
469     public final void retreat() {
470         assert this.leftLength() > 0 : "Violation of: this.left /=
    <>";
471         /*
472          * lastLeft points to the node before itself to move the
    "partition" to
473          * the left by one space.
474          */
475         this.lastLeft = this.lastLeft.previous;
476 
477         /*
478          * Increments rightLength and decrements leftLength to
    reflect the new
479          * changes.
480          */
481         this.leftLength--;
482         this.rightLength++;
483         assert this.conventionHolds();
484     }
485 
486 }
487 
```