THE OHIO STATE UNIVERSITY

PROJECT 6: DOUBLY-LINKED LIST IMPLEMENTATION OF LIST WITH RETREAT

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

October 23, 2023

```
 1 import java.util.Iterator;
 2 import java.util.NoSuchElementException;
 3
 4 import components.list.List;
 5 import components.list.ListSecondary;
 6
 7 /**
 8  * {@code List} represented as a doubly linked list, done "bare-handed",
   with
 9  * implementations of primary methods and {@code retreat} secondary method.
10  *
11  * <p>
12  * Execution-time performance of all methods implemented in this class is O
   (1).
13  * </p>
14  *
15  * @param <T>
16  *            type of {@code List} entries
17  * @convention <pre>
18  * $this.leftLength >= 0  and
19  * [$this.rightLength >= 0] and
20  * [$this.preStart is not null]  and
21  * [$this.lastLeft is not null]  and
22  * [$this.postFinish is not null]  and
23  * [$this.preStart points to the first node of a doubly linked list
24  *  containing ($this.leftLength + $this.rightLength + 2) nodes]  and
25  * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
26  *  that doubly linked list]  and
27  * [$this.postFinish points to the last node in that doubly linked list]
   and
28  * [for every node n in the doubly linked list of nodes, except the one
29  *  pointed to by $this.preStart, n.previous.next = n]  and
30  * [for every node n in the doubly linked list of nodes, except the one
31  *  pointed to by $this.postFinish, n.next.previous = n]
32  * </pre>
33  * @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 Daniil Gofman, Ansh Pachauri
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         // Create a node for the starting point of the list
179         this.preStart = new Node();
180
```

```
181         // Create a node for the ending point of the list
182         this.postFinish = new Node();
183
184         // Establish connections between the starting and ending nodes
185         this.preStart.next = this.postFinish;
186         this.postFinish.previous = this.preStart;
187
188         // Initialize the reference to the last left node, left length, and
    right length
189         this.lastLeft = this.preStart;
190         this.leftLength = 0;
191         this.rightLength = 0;
192     }
193
194     /**
195      * No-argument constructor.
196      */
197     public List3() {
198         // Create the initial representation of the data structure
199         this.createNewRep();
200
201         // Assert that the data structure's convention is maintained
202         assert this.conventionHolds();
203     }
204
205     @SuppressWarnings("unchecked")
206     @Override
207     public final List3<T> newInstance() {
208         try {
209             return this.getClass().getConstructor().newInstance();
210         } catch (ReflectiveOperationException e) {
211             throw new AssertionError(
212                     "Cannot construct object of type " + this.getClass());
213         }
214     }
215
216     @Override
217     public final void clear() {
218         this.createNewRep();
219         assert this.conventionHolds();
220     }
221
222     @Override
223     public final void transferFrom(List<T> source) {
224         assert source instanceof List3<?> : ""
225                 + "Violation of: source is of dynamic type List3<?>";
226         /*
227          * This cast cannot fail since the assert above would have stopped
228          * execution in that case: source must be of dynamic type List3<?>,
    and
```

```
229          * the ? must be T or the call would not have compiled.
230          */
231         List3<T> localSource = (List3<T>) source;
232         this.preStart = localSource.preStart;
233         this.lastLeft = localSource.lastLeft;
234         this.postFinish = localSource.postFinish;
235         this.leftLength = localSource.leftLength;
236         this.rightLength = localSource.rightLength;
237         localSource.createNewRep();
238         assert this.conventionHolds();
239         assert localSource.conventionHolds();
240     }
241
242     @Override
243     public final void addRightFront(T x) {
244         // Ensure that the input value 'x' is not null
245         assert x != null : "Violation of: x is not null";
246
247         // Create a new node to hold the data 'x'
248         Node newNode = new Node();
249
250         // Get the current last left node
251         Node newNodeLast = this.lastLeft;
252
253         // Set the data of the new node
254         newNode.data = x;
255
256         // Update references to insert the new node in front of the last
    left node
257         newNode.previous = this.lastLeft;
258         newNode.next = newNodeLast.next;
259         newNodeLast.next = newNode;
260         newNode.next.previous = newNode;
261
262         // Increase the length of the right side
263         this.rightLength++;
264
265         // Assert that the data structure's convention is maintained
266         assert this.conventionHolds();
267     }
268
269     @Override
270     public final T removeRightFront() {
271         // Ensure that there are elements on the right side to remove
272         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
273
274         // Get the current last left node
275         Node currentNode = this.lastLeft;
276
277         // Get the next node in the right side, which we want to remove
```

```
278         Node currentNodeNext = currentNode.next;
279
280         // Update references to remove the next node from the right side
281         currentNode.next = currentNodeNext.next;
282
283         // Retrieve the data from the removed node
284         T x = currentNodeNext.data;
285
286         // Update references to maintain the structure
287         Node tempNode = this.lastLeft.next;
288         tempNode.previous = this.lastLeft;
289
290         // Decrease the length of the right side
291         this.rightLength--;
292
293         // Assert that the data structure's convention is maintained
294         assert this.conventionHolds();
295
296         // Return the removed data
297         return x;
298     }
299
300     @Override
301     public final void advance() {
302         // Ensure that there are elements on the right side to advance to
303         assert this.rightLength() > 0 : "Violation of: this.right /= <>";
304
305         // Get the node to move to, which is the next node on the right
    side
306         Node nodeToMove = this.lastLeft.next;
307
308         // Update the reference to the last left node
309         this.lastLeft = nodeToMove;
310
311         // Increase the left side length
312         this.leftLength++;
313
314         // Decrease the right side length
315         this.rightLength--;
316
317         // Assert that the data structure's convention is maintained
318         assert this.conventionHolds();
319     }
320
321     @Override
322     public final void moveToStart() {
323         // Move to the start of the data structure
324         // Update the reference to the last left node
325         this.lastLeft = this.preStart;
326
```

```
327         // Add the length of the left side to the right side
328         this.rightLength += this.leftLength;
329 
330         // Reset the left side length to 0
331         this.leftLength = 0;
332 
333         // Assert that the data structure's convention is maintained
334         assert this.conventionHolds();
335     }
336 
337     @Override
338     public final int leftLength() {
339 
340         assert this.conventionHolds();
341         // Fix this line to return the result after checking the
    convention.
342         return this.leftLength;
343     }
344 
345     @Override
346     public final int rightLength() {
347 
348         assert this.conventionHolds();
349         // Fix this line to return the result after checking the
    convention.
350         return this.rightLength;
351     }
352 
353     @Override
354     public final Iterator<T> iterator() {
355         assert this.conventionHolds();
356         return new List3Iterator();
357     }
358 
359     /**
360      * Implementation of {@code Iterator} interface for {@code List3}.
361      */
362     private final class List3Iterator implements Iterator<T> {
363 
364         /**
365          * Current node in the linked list.
366          */
367         private Node current;
368 
369         /**
370          * No-argument constructor.
371          */
372         private List3Iterator() {
373             this.current = List3.this.preStart.next;
374             assert List3.this.conventionHolds();
```

```
375         }
376
377         @Override
378         public boolean hasNext() {
379             return this.current != List3.this.postFinish;
380         }
381
382         @Override
383         public T next() {
384             assert this.hasNext() : "Violation of: ~this.unseen /= <>";
385             if (!this.hasNext()) {
386                 /*
387                  * Exception is supposed to be thrown in this case, but
    with
388                  * assertion-checking enabled it cannot happen because of
    assert
389                  * above.
390                  */
391                 throw new NoSuchElementException();
392             }
393             T x = this.current.data;
394             this.current = this.current.next;
395             assert List3.this.conventionHolds();
396             return x;
397         }
398
399         @Override
400         public void remove() {
401             throw new UnsupportedOperationException(
402                     "remove operation not supported");
403         }
404
405     }
406
407     /*
408      * Other methods (overridden for performance reasons)
    ----------------------
409      */
410
411     @Override
412     public final void moveToFinish() {
413         // Move to the end of the data structure
414         // Update the reference to the last left node
415         this.lastLeft = this.postFinish.previous;
416
417         // Add the length of the right side to the left side
418         this.leftLength += this.rightLength;
419
420         // Reset the right side length to 0
421         this.rightLength = 0;
```

```
422
423         // Assert that the data structure's convention is maintained
424         assert this.conventionHolds();
425     }
426
427     @Override
428     public final void retreat() {
429         // Ensure that there are elements on the left side to retreat from
430         assert this.leftLength() > 0 : "Violation of: this.left /= <>";
431
432         // Get the node to move, which is the current last left node
433         Node nodeToMove = this.lastLeft;
434
435         // Update the reference to the last left node to the previous node
436         this.lastLeft = nodeToMove.previous;
437
438         // Decrease the left side length by 1
439         this.leftLength--;
440
441         // Increase the right side length by 1
442         this.rightLength++;
443
444         // Assert that the data structure's convention is maintained
445         assert this.conventionHolds();
446     }
447
448 }
449
```