```java
/*
 * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.util;

import java.util.function.Consumer;

/**
 * Doubly-linked list implementation of the {@code List} and {@code Deque}
 * interfaces.  Implements all optional list operations, and permits all
 * elements (including {@code null}).
 *
 * <p>All of the operations perform as could be expected for a doubly-linked
 * list.  Operations that index into the list will traverse the list from
 * the beginning or the end, whichever is closer to the specified index.
 *
 * <p><strong>Note that this implementation is not synchronized.</strong>
 * If multiple threads access a linked list concurrently, and at least
 * one of the threads modifies the list structurally, it <i>must</i> be
 * synchronized externally.  (A structural modification is any operation
 * that adds or deletes one or more elements; merely setting the value of
 * an element is not a structural modification.)  This is typically
 * accomplished by synchronizing on some object that naturally
 * encapsulates the list.
 *
 * If no such object exists, the list should be "wrapped" using the
 * {@link Collections#synchronizedList Collections.synchronizedList}
 * method.  This is best done at creation time, to prevent accidental
 * unsynchronized access to the list:<pre>
 *   List list = Collections.synchronizedList(new LinkedList(...));</pre>
 *
 * <p>The iterators returned by this class's {@code iterator} and
 * {@code listIterator} methods are <i>fail-fast</i>: if the list is
 * structurally modified at any time after the iterator is created, in
 * any way except through the Iterator's own {@code remove} or
 * {@code add} methods, the iterator will throw a {@link
 * ConcurrentModificationException}.  Thus, in the face of concurrent
 * modification, the iterator fails quickly and cleanly, rather than
 * risking arbitrary, non-deterministic behavior at an undetermined
 * time in the future.
 *
 * <p>Note that the fail-fast behavior of an iterator cannot be guaranteed
 * as it is, generally speaking, impossible to make any hard guarantees in the
 * presence of unsynchronized concurrent modification.  Fail-fast iterators
 * throw {@code ConcurrentModificationException} on a best-effort basis.
```

```
68   * Therefore, it would be wrong to write a program that depended on this
69   * exception for its correctness:   <i>the fail-fast behavior of iterators
70   * should be used only to detect bugs.</i>
71   *
72   * <p>This class is a member of the
73   * <a href="{@docRoot}/../technotes/guides/collections/index.html">
74   * Java Collections Framework</a>.
75   *
76   * @author  Josh Bloch
77   * @see     List
78   * @see     ArrayList
79   * @since 1.2
80   * @param <E> the type of elements held in this collection
81   */
82
83  public class LinkedList<E>
84      extends AbstractSequentialList<E>
85      implements List<E>, Deque<E>, Cloneable, java.io.Serializable
86  {
87      transient int size = 0;
88
89      /**
90       * Pointer to first node.
91       * Invariant: (first == null && last == null) ||
92       *            (first.prev == null && first.item != null)
93       */
94      transient Node<E> first;
95
96      /**
97       * Pointer to last node.
98       * Invariant: (first == null && last == null) ||
99       *            (last.next == null && last.item != null)
100      */
101     transient Node<E> last;
102
103     /**
104      * Constructs an empty list.
105      */
106     public LinkedList() {
107     }
108
109     /**
110      * Constructs a list containing the elements of the specified
111      * collection, in the order they are returned by the collection's
112      * iterator.
113      *
114      * @param  c the collection whose elements are to be placed into this list
115      * @throws NullPointerException if the specified collection is null
116      */
117     public LinkedList(Collection<? extends E> c) {
118         this();
119         addAll(c);
120     }
121
122     /**
123      * Links e as first element.
124      */
125     private void linkFirst(E e) {
126         final Node<E> f = first;
127         final Node<E> newNode = new Node<>(null, e, f);
128         first = newNode;
129         if (f == null)
130             last = newNode;
131         else
132             f.prev = newNode;
133         size++;
134         modCount++;
```

```
135     }
136
137     /**
138      * Links e as last element.
139      */
140     void linkLast(E e) {
141         final Node<E> l = last;
142         final Node<E> newNode = new Node<>(l, e, null);
143         last = newNode;
144         if (l == null)
145             first = newNode;
146         else
147             l.next = newNode;
148         size++;
149         modCount++;
150     }
151
152     /**
153      * Inserts element e before non-null Node succ.
154      */
155     void linkBefore(E e, Node<E> succ) {
156         // assert succ != null;
157         final Node<E> pred = succ.prev;
158         final Node<E> newNode = new Node<>(pred, e, succ);
159         succ.prev = newNode;
160         if (pred == null)
161             first = newNode;
162         else
163             pred.next = newNode;
164         size++;
165         modCount++;
166     }
167
168     /**
169      * Unlinks non-null first node f.
170      */
171     private E unlinkFirst(Node<E> f) {
172         // assert f == first && f != null;
173         final E element = f.item;
174         final Node<E> next = f.next;
175         f.item = null;
176         f.next = null; // help GC
177         first = next;
178         if (next == null)
179             last = null;
180         else
181             next.prev = null;
182         size--;
183         modCount++;
184         return element;
185     }
186
187     /**
188      * Unlinks non-null last node l.
189      */
190     private E unlinkLast(Node<E> l) {
191         // assert l == last && l != null;
192         final E element = l.item;
193         final Node<E> prev = l.prev;
194         l.item = null;
195         l.prev = null; // help GC
196         last = prev;
197         if (prev == null)
198             first = null;
199         else
200             prev.next = null;
201         size--;
```

```
202         modCount++;
203         return element;
204     }
205
206     /**
207      * Unlinks non-null node x.
208      */
209     E unlink(Node<E> x) {
210         // assert x != null;
211         final E element = x.item;
212         final Node<E> next = x.next;
213         final Node<E> prev = x.prev;
214
215         if (prev == null) {
216             first = next;
217         } else {
218             prev.next = next;
219             x.prev = null;
220         }
221
222         if (next == null) {
223             last = prev;
224         } else {
225             next.prev = prev;
226             x.next = null;
227         }
228
229         x.item = null;
230         size--;
231         modCount++;
232         return element;
233     }
234
235     /**
236      * Returns the first element in this list.
237      *
238      * @return the first element in this list
239      * @throws NoSuchElementException if this list is empty
240      */
241     public E getFirst() {
242         final Node<E> f = first;
243         if (f == null)
244             throw new NoSuchElementException();
245         return f.item;
246     }
247
248     /**
249      * Returns the last element in this list.
250      *
251      * @return the last element in this list
252      * @throws NoSuchElementException if this list is empty
253      */
254     public E getLast() {
255         final Node<E> l = last;
256         if (l == null)
257             throw new NoSuchElementException();
258         return l.item;
259     }
260
261     /**
262      * Removes and returns the first element from this list.
263      *
264      * @return the first element from this list
265      * @throws NoSuchElementException if this list is empty
266      */
267     public E removeFirst() {
268         final Node<E> f = first;
```

```
269         if (f == null)
270             throw new NoSuchElementException();
271         return unlinkFirst(f);
272     }
273 
274     /**
275      * Removes and returns the last element from this list.
276      *
277      * @return the last element from this list
278      * @throws NoSuchElementException if this list is empty
279      */
280     public E removeLast() {
281         final Node<E> l = last;
282         if (l == null)
283             throw new NoSuchElementException();
284         return unlinkLast(l);
285     }
286 
287     /**
288      * Inserts the specified element at the beginning of this list.
289      *
290      * @param e the element to add
291      */
292     public void addFirst(E e) {
293         linkFirst(e);
294     }
295 
296     /**
297      * Appends the specified element to the end of this list.
298      *
299      * <p>This method is equivalent to {@link #add}.
300      *
301      * @param e the element to add
302      */
303     public void addLast(E e) {
304         linkLast(e);
305     }
306 
307     /**
308      * Returns {@code true} if this list contains the specified element.
309      * More formally, returns {@code true} if and only if this list contains
310      * at least one element {@code e} such that
311      * <tt>(o==null ? e==null : o.equals(e))</tt>.
312      *
313      * @param o element whose presence in this list is to be tested
314      * @return {@code true} if this list contains the specified element
315      */
316     public boolean contains(Object o) {
317         return indexOf(o) != -1;
318     }
319 
320     /**
321      * Returns the number of elements in this list.
322      *
323      * @return the number of elements in this list
324      */
325     public int size() {
326         return size;
327     }
328 
329     /**
330      * Appends the specified element to the end of this list.
331      *
332      * <p>This method is equivalent to {@link #addLast}.
333      *
334      * @param e element to be appended to this list
335      * @return {@code true} (as specified by {@link Collection#add})
```

```
336      */
337     public boolean add(E e) {
338         linkLast(e);
339         return true;
340     }
341 
342     /**
343      * Removes the first occurrence of the specified element from this list,
344      * if it is present.  If this list does not contain the element, it is
345      * unchanged.  More formally, removes the element with the lowest index
346      * {@code i} such that
347      * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>
348      * (if such an element exists).  Returns {@code true} if this list
349      * contained the specified element (or equivalently, if this list
350      * changed as a result of the call).
351      *
352      * @param o element to be removed from this list, if present
353      * @return {@code true} if this list contained the specified element
354      */
355     public boolean remove(Object o) {
356         if (o == null) {
357             for (Node<E> x = first; x != null; x = x.next) {
358                 if (x.item == null) {
359                     unlink(x);
360                     return true;
361                 }
362             }
363         } else {
364             for (Node<E> x = first; x != null; x = x.next) {
365                 if (o.equals(x.item)) {
366                     unlink(x);
367                     return true;
368                 }
369             }
370         }
371         return false;
372     }
373 
374     /**
375      * Appends all of the elements in the specified collection to the end of
376      * this list, in the order that they are returned by the specified
377      * collection's iterator.  The behavior of this operation is undefined if
378      * the specified collection is modified while the operation is in
379      * progress.  (Note that this will occur if the specified collection is
380      * this list, and it's nonempty.)
381      *
382      * @param c collection containing elements to be added to this list
383      * @return {@code true} if this list changed as a result of the call
384      * @throws NullPointerException if the specified collection is null
385      */
386     public boolean addAll(Collection<? extends E> c) {
387         return addAll(size, c);
388     }
389 
390     /**
391      * Inserts all of the elements in the specified collection into this
392      * list, starting at the specified position.  Shifts the element
393      * currently at that position (if any) and any subsequent elements to
394      * the right (increases their indices).  The new elements will appear
395      * in the list in the order that they are returned by the
396      * specified collection's iterator.
397      *
398      * @param index index at which to insert the first element
399      *              from the specified collection
400      * @param c collection containing elements to be added to this list
401      * @return {@code true} if this list changed as a result of the call
402      * @throws IndexOutOfBoundsException {@inheritDoc}
```

```
403         * @throws NullPointerException if the specified collection is null
404         */
405        public boolean addAll(int index, Collection<? extends E> c) {
406            checkPositionIndex(index);
407
408            Object[] a = c.toArray();
409            int numNew = a.length;
410            if (numNew == 0)
411                return false;
412
413            Node<E> pred, succ;
414            if (index == size) {
415                succ = null;
416                pred = last;
417            } else {
418                succ = node(index);
419                pred = succ.prev;
420            }
421
422            for (Object o : a) {
423                @SuppressWarnings("unchecked") E e = (E) o;
424                Node<E> newNode = new Node<>(pred, e, null);
425                if (pred == null)
426                    first = newNode;
427                else
428                    pred.next = newNode;
429                pred = newNode;
430            }
431
432            if (succ == null) {
433                last = pred;
434            } else {
435                pred.next = succ;
436                succ.prev = pred;
437            }
438
439            size += numNew;
440            modCount++;
441            return true;
442        }
443
444        /**
445         * Removes all of the elements from this list.
446         * The list will be empty after this call returns.
447         */
448        public void clear() {
449            // Clearing all of the links between nodes is "unnecessary", but:
450            // - helps a generational GC if the discarded nodes inhabit
451            //   more than one generation
452            // - is sure to free memory even if there is a reachable Iterator
453            for (Node<E> x = first; x != null; ) {
454                Node<E> next = x.next;
455                x.item = null;
456                x.next = null;
457                x.prev = null;
458                x = next;
459            }
460            first = last = null;
461            size = 0;
462            modCount++;
463        }
464
465
466        // Positional Access Operations
467
468        /**
469         * Returns the element at the specified position in this list.
```

```
470         *
471         * @param index index of the element to return
472         * @return the element at the specified position in this list
473         * @throws IndexOutOfBoundsException {@inheritDoc}
474         */
475        public E get(int index) {
476            checkElementIndex(index);
477            return node(index).item;
478        }
479
480        /**
481         * Replaces the element at the specified position in this list with the
482         * specified element.
483         *
484         * @param index index of the element to replace
485         * @param element element to be stored at the specified position
486         * @return the element previously at the specified position
487         * @throws IndexOutOfBoundsException {@inheritDoc}
488         */
489        public E set(int index, E element) {
490            checkElementIndex(index);
491            Node<E> x = node(index);
492            E oldVal = x.item;
493            x.item = element;
494            return oldVal;
495        }
496
497        /**
498         * Inserts the specified element at the specified position in this list.
499         * Shifts the element currently at that position (if any) and any
500         * subsequent elements to the right (adds one to their indices).
501         *
502         * @param index index at which the specified element is to be inserted
503         * @param element element to be inserted
504         * @throws IndexOutOfBoundsException {@inheritDoc}
505         */
506        public void add(int index, E element) {
507            checkPositionIndex(index);
508
509            if (index == size)
510                linkLast(element);
511            else
512                linkBefore(element, node(index));
513        }
514
515        /**
516         * Removes the element at the specified position in this list.  Shifts any
517         * subsequent elements to the left (subtracts one from their indices).
518         * Returns the element that was removed from the list.
519         *
520         * @param index the index of the element to be removed
521         * @return the element previously at the specified position
522         * @throws IndexOutOfBoundsException {@inheritDoc}
523         */
524        public E remove(int index) {
525            checkElementIndex(index);
526            return unlink(node(index));
527        }
528
529        /**
530         * Tells if the argument is the index of an existing element.
531         */
532        private boolean isElementIndex(int index) {
533            return index >= 0 && index < size;
534        }
535
536        /**
```

```
537         * Tells if the argument is the index of a valid position for an
538         * iterator or an add operation.
539         */
540        private boolean isPositionIndex(int index) {
541            return index >= 0 && index <= size;
542        }
543
544        /**
545         * Constructs an IndexOutOfBoundsException detail message.
546         * Of the many possible refactorings of the error handling code,
547         * this "outlining" performs best with both server and client VMs.
548         */
549        private String outOfBoundsMsg(int index) {
550            return "Index: "+index+", Size: "+size;
551        }
552
553        private void checkElementIndex(int index) {
554            if (!isElementIndex(index))
555                throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
556        }
557
558        private void checkPositionIndex(int index) {
559            if (!isPositionIndex(index))
560                throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
561        }
562
563        /**
564         * Returns the (non-null) Node at the specified element index.
565         */
566        Node<E> node(int index) {
567            // assert isElementIndex(index);
568
569            if (index < (size >> 1)) {
570                Node<E> x = first;
571                for (int i = 0; i < index; i++)
572                    x = x.next;
573                return x;
574            } else {
575                Node<E> x = last;
576                for (int i = size - 1; i > index; i--)
577                    x = x.prev;
578                return x;
579            }
580        }
581
582        // Search Operations
583
584        /**
585         * Returns the index of the first occurrence of the specified element
586         * in this list, or -1 if this list does not contain the element.
587         * More formally, returns the lowest index {@code i} such that
588         * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
589         * or -1 if there is no such index.
590         *
591         * @param o element to search for
592         * @return the index of the first occurrence of the specified element in
593         *         this list, or -1 if this list does not contain the element
594         */
595        public int indexOf(Object o) {
596            int index = 0;
597            if (o == null) {
598                for (Node<E> x = first; x != null; x = x.next) {
599                    if (x.item == null)
600                        return index;
601                    index++;
602                }
603            } else {
```

```
604             for (Node<E> x = first; x != null; x = x.next) {
605                 if (o.equals(x.item))
606                     return index;
607                 index++;
608             }
609         }
610         return -1;
611     }
612 
613     /**
614      * Returns the index of the last occurrence of the specified element
615      * in this list, or -1 if this list does not contain the element.
616      * More formally, returns the highest index {@code i} such that
617      * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
618      * or -1 if there is no such index.
619      *
620      * @param o element to search for
621      * @return the index of the last occurrence of the specified element in
622      *         this list, or -1 if this list does not contain the element
623      */
624     public int lastIndexOf(Object o) {
625         int index = size;
626         if (o == null) {
627             for (Node<E> x = last; x != null; x = x.prev) {
628                 index--;
629                 if (x.item == null)
630                     return index;
631             }
632         } else {
633             for (Node<E> x = last; x != null; x = x.prev) {
634                 index--;
635                 if (o.equals(x.item))
636                     return index;
637             }
638         }
639         return -1;
640     }
641 
642     // Queue operations.
643 
644     /**
645      * Retrieves, but does not remove, the head (first element) of this list.
646      *
647      * @return the head of this list, or {@code null} if this list is empty
648      * @since 1.5
649      */
650     public E peek() {
651         final Node<E> f = first;
652         return (f == null) ? null : f.item;
653     }
654 
655     /**
656      * Retrieves, but does not remove, the head (first element) of this list.
657      *
658      * @return the head of this list
659      * @throws NoSuchElementException if this list is empty
660      * @since 1.5
661      */
662     public E element() {
663         return getFirst();
664     }
665 
666     /**
667      * Retrieves and removes the head (first element) of this list.
668      *
669      * @return the head of this list, or {@code null} if this list is empty
670      * @since 1.5
```

```
671         */
672        public E poll() {
673            final Node<E> f = first;
674            return (f == null) ? null : unlinkFirst(f);
675        }
676
677        /**
678         * Retrieves and removes the head (first element) of this list.
679         *
680         * @return the head of this list
681         * @throws NoSuchElementException if this list is empty
682         * @since 1.5
683         */
684        public E remove() {
685            return removeFirst();
686        }
687
688        /**
689         * Adds the specified element as the tail (last element) of this list.
690         *
691         * @param e the element to add
692         * @return {@code true} (as specified by {@link Queue#offer})
693         * @since 1.5
694         */
695        public boolean offer(E e) {
696            return add(e);
697        }
698
699        // Deque operations
700        /**
701         * Inserts the specified element at the front of this list.
702         *
703         * @param e the element to insert
704         * @return {@code true} (as specified by {@link Deque#offerFirst})
705         * @since 1.6
706         */
707        public boolean offerFirst(E e) {
708            addFirst(e);
709            return true;
710        }
711
712        /**
713         * Inserts the specified element at the end of this list.
714         *
715         * @param e the element to insert
716         * @return {@code true} (as specified by {@link Deque#offerLast})
717         * @since 1.6
718         */
719        public boolean offerLast(E e) {
720            addLast(e);
721            return true;
722        }
723
724        /**
725         * Retrieves, but does not remove, the first element of this list,
726         * or returns {@code null} if this list is empty.
727         *
728         * @return the first element of this list, or {@code null}
729         *         if this list is empty
730         * @since 1.6
731         */
732        public E peekFirst() {
733            final Node<E> f = first;
734            return (f == null) ? null : f.item;
735         }
736
737        /**
```

```
738         * Retrieves, but does not remove, the last element of this list,
739         * or returns {@code null} if this list is empty.
740         *
741         * @return the last element of this list, or {@code null}
742         *         if this list is empty
743         * @since 1.6
744         */
745        public E peekLast() {
746            final Node<E> l = last;
747            return (l == null) ? null : l.item;
748        }
749
750        /**
751         * Retrieves and removes the first element of this list,
752         * or returns {@code null} if this list is empty.
753         *
754         * @return the first element of this list, or {@code null} if
755         *     this list is empty
756         * @since 1.6
757         */
758        public E pollFirst() {
759            final Node<E> f = first;
760            return (f == null) ? null : unlinkFirst(f);
761        }
762
763        /**
764         * Retrieves and removes the last element of this list,
765         * or returns {@code null} if this list is empty.
766         *
767         * @return the last element of this list, or {@code null} if
768         *     this list is empty
769         * @since 1.6
770         */
771        public E pollLast() {
772            final Node<E> l = last;
773            return (l == null) ? null : unlinkLast(l);
774        }
775
776        /**
777         * Pushes an element onto the stack represented by this list.  In other
778         * words, inserts the element at the front of this list.
779         *
780         * <p>This method is equivalent to {@link #addFirst}.
781         *
782         * @param e the element to push
783         * @since 1.6
784         */
785        public void push(E e) {
786            addFirst(e);
787        }
788
789        /**
790         * Pops an element from the stack represented by this list.  In other
791         * words, removes and returns the first element of this list.
792         *
793         * <p>This method is equivalent to {@link #removeFirst()}.
794         *
795         * @return the element at the front of this list (which is the top
796         *         of the stack represented by this list)
797         * @throws NoSuchElementException if this list is empty
798         * @since 1.6
799         */
800        public E pop() {
801            return removeFirst();
802        }
803
804        /**
```

```
805         * Removes the first occurrence of the specified element in this
806         * list (when traversing the list from head to tail).  If the list
807         * does not contain the element, it is unchanged.
808         *
809         * @param o element to be removed from this list, if present
810         * @return {@code true} if the list contained the specified element
811         * @since 1.6
812         */
813        public boolean removeFirstOccurrence(Object o) {
814            return remove(o);
815        }
816
817        /**
818         * Removes the last occurrence of the specified element in this
819         * list (when traversing the list from head to tail).  If the list
820         * does not contain the element, it is unchanged.
821         *
822         * @param o element to be removed from this list, if present
823         * @return {@code true} if the list contained the specified element
824         * @since 1.6
825         */
826        public boolean removeLastOccurrence(Object o) {
827            if (o == null) {
828                for (Node<E> x = last; x != null; x = x.prev) {
829                    if (x.item == null) {
830                        unlink(x);
831                        return true;
832                    }
833                }
834            } else {
835                for (Node<E> x = last; x != null; x = x.prev) {
836                    if (o.equals(x.item)) {
837                        unlink(x);
838                        return true;
839                    }
840                }
841            }
842            return false;
843        }
844
845        /**
846         * Returns a list-iterator of the elements in this list (in proper
847         * sequence), starting at the specified position in the list.
848         * Obeys the general contract of {@code List.listIterator(int)}.<p>
849         *
850         * The list-iterator is <i>fail-fast</i>: if the list is structurally
851         * modified at any time after the Iterator is created, in any way except
852         * through the list-iterator's own {@code remove} or {@code add}
853         * methods, the list-iterator will throw a
854         * {@code ConcurrentModificationException}.  Thus, in the face of
855         * concurrent modification, the iterator fails quickly and cleanly, rather
856         * than risking arbitrary, non-deterministic behavior at an undetermined
857         * time in the future.
858         *
859         * @param index index of the first element to be returned from the
860         *              list-iterator (by a call to {@code next})
861         * @return a ListIterator of the elements in this list (in proper
862         *         sequence), starting at the specified position in the list
863         * @throws IndexOutOfBoundsException {@inheritDoc}
864         * @see List#listIterator(int)
865         */
866        public ListIterator<E> listIterator(int index) {
867            checkPositionIndex(index);
868            return new ListItr(index);
869        }
870
871        private class ListItr implements ListIterator<E> {
```

```
872         private Node<E> lastReturned;
873         private Node<E> next;
874         private int nextIndex;
875         private int expectedModCount = modCount;
876
877         ListItr(int index) {
878             // assert isPositionIndex(index);
879             next = (index == size) ? null : node(index);
880             nextIndex = index;
881         }
882
883         public boolean hasNext() {
884             return nextIndex < size;
885         }
886
887         public E next() {
888             checkForComodification();
889             if (!hasNext())
890                 throw new NoSuchElementException();
891
892             lastReturned = next;
893             next = next.next;
894             nextIndex++;
895             return lastReturned.item;
896         }
897
898         public boolean hasPrevious() {
899             return nextIndex > 0;
900         }
901
902         public E previous() {
903             checkForComodification();
904             if (!hasPrevious())
905                 throw new NoSuchElementException();
906
907             lastReturned = next = (next == null) ? last : next.prev;
908             nextIndex--;
909             return lastReturned.item;
910         }
911
912         public int nextIndex() {
913             return nextIndex;
914         }
915
916         public int previousIndex() {
917             return nextIndex - 1;
918         }
919
920         public void remove() {
921             checkForComodification();
922             if (lastReturned == null)
923                 throw new IllegalStateException();
924
925             Node<E> lastNext = lastReturned.next;
926             unlink(lastReturned);
927             if (next == lastReturned)
928                 next = lastNext;
929             else
930                 nextIndex--;
931             lastReturned = null;
932             expectedModCount++;
933         }
934
935         public void set(E e) {
936             if (lastReturned == null)
937                 throw new IllegalStateException();
938             checkForComodification();
```

```
939             lastReturned.item = e;
940         }
941
942         public void add(E e) {
943             checkForComodification();
944             lastReturned = null;
945             if (next == null)
946                 linkLast(e);
947             else
948                 linkBefore(e, next);
949             nextIndex++;
950             expectedModCount++;
951         }
952
953         public void forEachRemaining(Consumer<? super E> action) {
954             Objects.requireNonNull(action);
955             while (modCount == expectedModCount && nextIndex < size) {
956                 action.accept(next.item);
957                 lastReturned = next;
958                 next = next.next;
959                 nextIndex++;
960             }
961             checkForComodification();
962         }
963
964         final void checkForComodification() {
965             if (modCount != expectedModCount)
966                 throw new ConcurrentModificationException();
967         }
968     }
969
970     private static class Node<E> {
971         E item;
972         Node<E> next;
973         Node<E> prev;
974
975         Node(Node<E> prev, E element, Node<E> next) {
976             this.item = element;
977             this.next = next;
978             this.prev = prev;
979         }
980     }
981
982     /**
983      * @since 1.6
984      */
985     public Iterator<E> descendingIterator() {
986         return new DescendingIterator();
987     }
988
989     /**
990      * Adapter to provide descending iterators via ListItr.previous
991      */
992     private class DescendingIterator implements Iterator<E> {
993         private final ListItr itr = new ListItr(size());
994         public boolean hasNext() {
995             return itr.hasPrevious();
996         }
997         public E next() {
998             return itr.previous();
999         }
1000        public void remove() {
1001            itr.remove();
1002        }
1003    }
1004
1005    @SuppressWarnings("unchecked")
```

```
1006    private LinkedList<E> superClone() {
1007        try {
1008            return (LinkedList<E>) super.clone();
1009        } catch (CloneNotSupportedException e) {
1010            throw new InternalError(e);
1011        }
1012    }
1013
1014    /**
1015     * Returns a shallow copy of this {@code LinkedList}. (The elements
1016     * themselves are not cloned.)
1017     *
1018     * @return a shallow copy of this {@code LinkedList} instance
1019     */
1020    public Object clone() {
1021        LinkedList<E> clone = superClone();
1022
1023        // Put clone into "virgin" state
1024        clone.first = clone.last = null;
1025        clone.size = 0;
1026        clone.modCount = 0;
1027
1028        // Initialize clone with our elements
1029        for (Node<E> x = first; x != null; x = x.next)
1030            clone.add(x.item);
1031
1032        return clone;
1033    }
1034
1035    /**
1036     * Returns an array containing all of the elements in this list
1037     * in proper sequence (from first to last element).
1038     *
1039     * <p>The returned array will be "safe" in that no references to it are
1040     * maintained by this list.  (In other words, this method must allocate
1041     * a new array).  The caller is thus free to modify the returned array.
1042     *
1043     * <p>This method acts as bridge between array-based and collection-based
1044     * APIs.
1045     *
1046     * @return an array containing all of the elements in this list
1047     *         in proper sequence
1048     */
1049    public Object[] toArray() {
1050        Object[] result = new Object[size];
1051        int i = 0;
1052        for (Node<E> x = first; x != null; x = x.next)
1053            result[i++] = x.item;
1054        return result;
1055    }
1056
1057    /**
1058     * Returns an array containing all of the elements in this list in
1059     * proper sequence (from first to last element); the runtime type of
1060     * the returned array is that of the specified array.  If the list fits
1061     * in the specified array, it is returned therein.  Otherwise, a new
1062     * array is allocated with the runtime type of the specified array and
1063     * the size of this list.
1064     *
1065     * <p>If the list fits in the specified array with room to spare (i.e.,
1066     * the array has more elements than the list), the element in the array
1067     * immediately following the end of the list is set to {@code null}.
1068     * (This is useful in determining the length of the list <i>only</i> if
1069     * the caller knows that the list does not contain any null elements.)
1070     *
1071     * <p>Like the {@link #toArray()} method, this method acts as bridge between
1072     * array-based and collection-based APIs.  Further, this method allows
```

```
1073      * precise control over the runtime type of the output array, and may,
1074      * under certain circumstances, be used to save allocation costs.
1075      *
1076      * <p>Suppose {@code x} is a list known to contain only strings.
1077      * The following code can be used to dump the list into a newly
1078      * allocated array of {@code String}:
1079      *
1080      * <pre>
1081      *     String[] y = x.toArray(new String[0]);</pre>
1082      *
1083      * Note that {@code toArray(new Object[0])} is identical in function to
1084      * {@code toArray()}.
1085      *
1086      * @param a the array into which the elements of the list are to
1087      *          be stored, if it is big enough; otherwise, a new array of the
1088      *          same runtime type is allocated for this purpose.
1089      * @return an array containing the elements of the list
1090      * @throws ArrayStoreException if the runtime type of the specified array
1091      *         is not a supertype of the runtime type of every element in
1092      *         this list
1093      * @throws NullPointerException if the specified array is null
1094      */
1095     @SuppressWarnings("unchecked")
1096     public <T> T[] toArray(T[] a) {
1097         if (a.length < size)
1098             a = (T[])java.lang.reflect.Array.newInstance(
1099                                 a.getClass().getComponentType(), size);
1100         int i = 0;
1101         Object[] result = a;
1102         for (Node<E> x = first; x != null; x = x.next)
1103             result[i++] = x.item;
1104
1105         if (a.length > size)
1106             a[size] = null;
1107
1108         return a;
1109     }
1110
1111     private static final long serialVersionUID = 876323262645176354L;
1112
1113     /**
1114      * Saves the state of this {@code LinkedList} instance to a stream
1115      * (that is, serializes it).
1116      *
1117      * @serialData The size of the list (the number of elements it
1118      *             contains) is emitted (int), followed by all of its
1119      *             elements (each an Object) in the proper order.
1120      */
1121     private void writeObject(java.io.ObjectOutputStream s)
1122         throws java.io.IOException {
1123         // Write out any hidden serialization magic
1124         s.defaultWriteObject();
1125
1126         // Write out size
1127         s.writeInt(size);
1128
1129         // Write out all elements in the proper order.
1130         for (Node<E> x = first; x != null; x = x.next)
1131             s.writeObject(x.item);
1132     }
1133
1134     /**
1135      * Reconstitutes this {@code LinkedList} instance from a stream
1136      * (that is, deserializes it).
1137      */
1138     @SuppressWarnings("unchecked")
1139     private void readObject(java.io.ObjectInputStream s)
```

```
1140        throws java.io.IOException, ClassNotFoundException {
1141        // Read in any hidden serialization magic
1142        s.defaultReadObject();
1143
1144        // Read in size
1145        int size = s.readInt();
1146
1147        // Read in all elements in the proper order.
1148        for (int i = 0; i < size; i++)
1149            linkLast((E)s.readObject());
1150    }
1151
1152    /**
1153     * Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>
1154     * and <em>fail-fast</em> {@link Spliterator} over the elements in this
1155     * list.
1156     *
1157     * <p>The {@code Spliterator} reports {@link Spliterator#SIZED} and
1158     * {@link Spliterator#ORDERED}.  Overriding implementations should document
1159     * the reporting of additional characteristic values.
1160     *
1161     * @implNote
1162     * The {@code Spliterator} additionally reports {@link Spliterator#SUBSIZED}
1163     * and implements {@code trySplit} to permit limited parallelism..
1164     *
1165     * @return a {@code Spliterator} over the elements in this list
1166     * @since 1.8
1167     */
1168    @Override
1169    public Spliterator<E> spliterator() {
1170        return new LLSpliterator<E>(this, -1, 0);
1171    }
1172
1173    /** A customized variant of Spliterators.IteratorSpliterator */
1174    static final class LLSpliterator<E> implements Spliterator<E> {
1175        static final int BATCH_UNIT = 1 << 10;  // batch array size increment
1176        static final int MAX_BATCH = 1 << 25;  // max batch array size;
1177        final LinkedList<E> list; // null OK unless traversed
1178        Node<E> current;      // current node; null until initialized
1179        int est;              // size estimate; -1 until first needed
1180        int expectedModCount; // initialized when est set
1181        int batch;            // batch size for splits
1182
1183        LLSpliterator(LinkedList<E> list, int est, int expectedModCount) {
1184            this.list = list;
1185            this.est = est;
1186            this.expectedModCount = expectedModCount;
1187        }
1188
1189        final int getEst() {
1190            int s; // force initialization
1191            final LinkedList<E> lst;
1192            if ((s = est) < 0) {
1193                if ((lst = list) == null)
1194                    s = est = 0;
1195                else {
1196                    expectedModCount = lst.modCount;
1197                    current = lst.first;
1198                    s = est = lst.size;
1199                }
1200            }
1201            return s;
1202        }
1203
1204        public long estimateSize() { return (long) getEst(); }
1205
1206        public Spliterator<E> trySplit() {
```

```
            Node<E> p;
            int s = getEst();
            if (s > 1 && (p = current) != null) {
                int n = batch + BATCH_UNIT;
                if (n > s)
                    n = s;
                if (n > MAX_BATCH)
                    n = MAX_BATCH;
                Object[] a = new Object[n];
                int j = 0;
                do { a[j++] = p.item; } while ((p = p.next) != null && j < n);
                current = p;
                batch = j;
                est = s - j;
                return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
            }
            return null;
        }

        public void forEachRemaining(Consumer<? super E> action) {
            Node<E> p; int n;
            if (action == null) throw new NullPointerException();
            if ((n = getEst()) > 0 && (p = current) != null) {
                current = null;
                est = 0;
                do {
                    E e = p.item;
                    p = p.next;
                    action.accept(e);
                } while (p != null && --n > 0);
            }
            if (list.modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }

        public boolean tryAdvance(Consumer<? super E> action) {
            Node<E> p;
            if (action == null) throw new NullPointerException();
            if (getEst() > 0 && (p = current) != null) {
                --est;
                E e = p.item;
                current = p.next;
                action.accept(e);
                if (list.modCount != expectedModCount)
                    throw new ConcurrentModificationException();
                return true;
            }
            return false;
        }

        public int characteristics() {
            return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
        }
    }

}
```