```java
import java.util.Iterator;

/**
 * {@code List} represented as a doubly linked list, done "bare-handed", with
 * implementations of primary methods and {@code retreat} secondary method.
 *
 * <p>
 * Execution-time performance of all methods implemented in this class is O(1).
 * </p>
 *
 * @param <T>
 *            type of {@code List} entries
 * @convention <pre>
 * $this.leftLength >= 0  and
 * [$this.rightLength >= 0] and
 * [$this.preStart is not null]  and
 * [$this.lastLeft is not null]  and
 * [$this.postFinish is not null]  and
 * [$this.preStart points to the first node of a doubly linked list
 *  containing ($this.leftLength + $this.rightLength + 2) nodes]  and
 * [$this.lastLeft points to the ($this.leftLength + 1)-th node in
 *  that doubly linked list]  and
 * [$this.postFinish points to the last node in that doubly linked list]  and
 * [for every node n in the doubly linked list of nodes, except the one
 *  pointed to by $this.preStart, n.previous.next = n]  and
 * [for every node n in the doubly linked list of nodes, except the one
 *  pointed to by $this.postFinish, n.next.previous = n]
 * </pre>
 * @correspondence <pre>
 * this =
 *  ([data in nodes starting at $this.preStart.next and running through
 *    $this.lastLeft],
 *   [data in nodes starting at $this.lastLeft.next and running through
 *    $this.postFinish.previous])
 * </pre>
 *
 * @author David P & Zach
 *
 */
public class List3<T> extends ListSecondary<T> {

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

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

        /**
         * Next node in doubly linked list, or, if this is a trailing "smart"
         * Node, irrelevant.
         */
        private Node next;

        /**
         * Previous node in doubly linked list, or, if this is a leading "smart"
         * Node, irrelevant.
         */
        private Node previous;

    }

    /**
     * "Smart node" before start node of doubly linked list.
     */
    private Node preStart;

    /**
     * Last node of doubly linked list in this.left.
     */
    private Node lastLeft;

    /**
     * "Smart node" after finish node of linked list.
     */
    private Node postFinish;

    /**
     * Length of this.left.
     */
    private int leftLength;

    /**
     * Length of this.right.
     */
    private int rightLength;

    /**
     * Checks that the part of the convention repeated below holds for the
     * current representation.
     *
     * @return true if the convention holds (or if assertion checking is off);
     *         otherwise reports a violated assertion
     * @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>
     */
    private boolean conventionHolds() {
```

```java
        assert this.leftLength >= 0 : "Violation of: $this.leftLength >= 0";
        assert this.rightLength >= 0 : "Violation of: $this.rightLength >= 0";
        assert this.preStart != null : "Violation of: $this.preStart is not null";
        assert this.lastLeft != null : "Violation of: $this.lastLeft is not null";
        assert this.postFinish != null : "Violation of: $this.postFinish is not null";

        int count = 0;
        boolean lastLeftFound = false;
        Node n = this.preStart;
        while ((count < this.leftLength + this.rightLength + 1)
                && (n != this.postFinish)) {
            count++;
            if (n == this.lastLeft) {
                /*
                 * Check $this.lastLeft points to the ($this.leftLength + 1)-th
                 * node in that doubly linked list
                 */
                assert count == this.leftLength + 1 : ""
                        + "Violation of: [$this.lastLeft points to the"
                        + " ($this.leftLength + 1)-th node in that doubly linked list]";
                lastLeftFound = true;
            }
            /*
             * Check for every node n in the doubly linked list of nodes, except
             * the one pointed to by $this.postFinish, n.next.previous = n
             */
            assert (n.next != null) && (n.next.previous == n) : ""
                    + "Violation of: [for every node n in the doubly linked"
                    + " list of nodes, except the one pointed to by"
                    + " $this.postFinish, n.next.previous = n]";
            n = n.next;
            /*
             * Check for every node n in the doubly linked list of nodes, except
             * the one pointed to by $this.preStart, n.previous.next = n
             */
            assert n.previous.next == n : ""
                    + "Violation of: [for every node n in the doubly linked"
                    + " list of nodes, except the one pointed to by"
                    + " $this.preStart, n.previous.next = n]";
        }
        count++;
        assert count == this.leftLength + this.rightLength + 2 : ""
                + "Violation of: [$this.preStart points to the first node of"
                + " a doubly linked list containing"
                + " ($this.leftLength + $this.rightLength + 2) nodes]";
        assert lastLeftFound : ""
                + "Violation of: [$this.lastLeft points to the"
                + " ($this.leftLength + 1)-th node in that doubly linked list]";
        assert n == this.postFinish : ""
                + "Violation of: [$this.postFinish points to the last"
                + " node in that doubly linked list]";

        return true;
    }

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {
        this.preStart = new Node();
        this.lastLeft = this.preStart;
        this.postFinish = new Node();
        this.preStart.next = this.postFinish;
        this.postFinish.previous = this.preStart;
        this.leftLength = 0;
        this.rightLength = 0;
    }

    /**
     * No-argument constructor.
     */
    public List3() {
        this.createNewRep();
        assert this.conventionHolds();
    }

    @SuppressWarnings("unchecked")
    @Override
    public final List3<T> newInstance() {
        try {
            return this.getClass().getConstructor().newInstance();
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep();
        assert this.conventionHolds();
    }

    @Override
    public final void transferFrom(List<T> source) {
        assert source instanceof List3<?> : ""
                + "Violation of: source is of dynamic type List3<?>";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type List3<?>, and
         * the ? must be T or the call would not have compiled.
         */
        List3<T> localSource = (List3<T>) source;
        this.preStart = localSource.preStart;
        this.lastLeft = localSource.lastLeft;
        this.postFinish = localSource.postFinish;
        this.leftLength = localSource.leftLength;
        this.rightLength = localSource.rightLength;
        localSource.createNewRep();
        assert this.conventionHolds();
        assert localSource.conventionHolds();
    }

    @Override
```

```java
    public final void addRightFront(T x) {
        assert x != null : "Violation of: x is not null";

        // Create a new node
        Node newNode = new Node();
        newNode.data = x;
        // Insert the new node at the front of the right side
        newNode.next = this.lastLeft.next;
        newNode.previous = this.lastLeft;
        this.lastLeft.next.previous = newNode;
        this.lastLeft.next = newNode;
        // Increment the right length
        this.rightLength++;

        assert this.conventionHolds();
    }

    @Override
    public final T removeRightFront() {
        assert this.rightLength() > 0 : "Violation of: this.right /= <>";

        // Get the node to be remove
        Node removeNode = this.lastLeft.next;
        T end = removeNode.data;
        // Remove node from the list
        this.lastLeft.next = removeNode.next;
        removeNode.next.previous = this.lastLeft;
        // Decrement the right length
        this.rightLength--;

        assert this.conventionHolds();
        return end;
    }

    @Override
    public final void advance() {
        assert this.rightLength() > 0 : "Violation of: this.right /= <>";

        // set last left to next last left
        this.lastLeft = this.lastLeft.next;
        //set left length to be +1
        this.leftLength++;
        // set right length to be -1
        this.rightLength--;

        assert this.conventionHolds();
    }

    @Override
    public final void moveToStart() {
        // add the left length to the right length
        this.rightLength += this.leftLength;
        // reset the left length to zero
        this.leftLength = 0;
        // Set the last left pointer to the node
        this.lastLeft = this.preStart;

        assert this.conventionHolds();
    }

    @Override
    public final int leftLength() {
        assert this.conventionHolds();
        return this.leftLength;
    }

    @Override
    public final int rightLength() {
        assert this.conventionHolds();
        return this.rightLength;
    }

    @Override
    public final Iterator<T> iterator() {
        assert this.conventionHolds();
        return new List3Iterator();
    }

    /**
     * Implementation of {@code Iterator} interface for {@code List3}.
     */
    private final class List3Iterator implements Iterator<T> {

        /**
         * Current node in the linked list.
         */
        private Node current;

        /**
         * No-argument constructor.
         */
        private List3Iterator() {
            this.current = List3.this.preStart.next;
            assert List3.this.conventionHolds();
        }

        @Override
        public boolean hasNext() {
            return this.current != List3.this.postFinish;
        }

        @Override
        public T next() {
            assert this.hasNext() : "Violation of: ~this.unseen /= <>";
            if (!this.hasNext()) {
                /*
                 * Exception is supposed to be thrown in this case, but with
                 * assertion-checking enabled it cannot happen because of assert
                 * above.
                 */
                throw new NoSuchElementException();
            }
            T x = this.current.data;
            this.current = this.current.next;
            assert List3.this.conventionHolds();
            return x;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException(
                    "remove operation not supported");
        }

    }

    /*
     * Other methods (overridden for performance reasons) ---------------------
     */

    @Override
    public final void moveToFinish() {
        // add the right length to the left length
        this.leftLength += this.rightLength;
        // reset the right length to zero
        this.rightLength = 0;
        // set the last left pointer to the node
        this.lastLeft = this.postFinish.previous;

        assert this.conventionHolds();
    }

    @Override
    public final void retreat() {
        assert this.leftLength() > 0 : "Violation of: this.left /= <>";

        // move to previous node on the left side
        this.lastLeft = this.lastLeft.previous;
        // dec the length of the left side
        this.leftLength--;
        // inc the length of the right side
        this.rightLength++;

        assert this.conventionHolds();
    }

}
```