```java
package components.waitingLine;

import java.util.Iterator;

/**
 * Layered implementations of secondary methods for {@code Queue}.
 *
 * <p>
 * Assuming execution-time performance of O(1) for method {@code iterator} and
 * its return value's method {@code next}, execution-time performance of
 * {@code front} as implemented in this class is O(1). Execution-time
 * performance of {@code replaceFront} and {@code flip} as implemented in this
 * class is O(|{@code this}|). Execution-time performance of {@code append} as
 * implemented in this class is O(|{@code q}|). Execution-time performance of
 * {@code sort} as implemented in this class is O(|{@code this}| log
 * |{@code this}|) expected, O(|{@code this}|^2) worst case. Execution-time
 * performance of {@code rotate} as implemented in this class is
 * O({@code distance} mod |{@code this}|).
 *
 * @param <T>
 *            type of {@code Queue} entries
 */
public abstract class WaitingLineSecondary<T> implements WaitingLine<T> {

    /*
     * Private members ---------------------------------------------------------
     */

    /*
     * 2221/2231 assignment code deleted.
     */

    /*
     * Public members ----------------------------------------------------------
     */

    /*
     * Common methods (from Object) --------------------------------------------
     */

    @Override
    public final boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof WaitingLine<?>)) {
            return false;
        }
        WaitingLine<?> q = (WaitingLine<?>) obj;
        if (this.length() != q.length()) {
            return false;
        }
        Iterator<T> it1 = this.iterator();
        Iterator<?> it2 = q.iterator();
```

```
58          while (it1.hasNext()) {
59              T x1 = it1.next();
60              Object x2 = it2.next();
61              if (!x1.equals(x2)) {
62                  return false;
63              }
64          }
65          return true;
66      }
67
68      // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
69      @Override
70      public int hashCode() {
71          final int samples = 2;
72          final int a = 37;
73          final int b = 17;
74          int result = 0;
75          /*
76           * This code makes hashCode run in O(1) time. It works because of the
77           * iterator order string specification, which guarantees that the (at
78           * most) samples entries returned by the it.next() calls are the same
79           * when the two Queues are equal.
80           */
81          int n = 0;
82          Iterator<T> it = this.iterator();
83          while (n < samples && it.hasNext()) {
84              n++;
85              T x = it.next();
86              result = a * result + b * x.hashCode();
87          }
88          return result;
89      }
90
91      // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
92      @Override
93      public String toString() {
94          StringBuilder result = new StringBuilder("<");
95          Iterator<T> it = this.iterator();
96          while (it.hasNext()) {
97              result.append(it.next());
98              if (it.hasNext()) {
99                  result.append(",");
100             }
101         }
102         result.append(">");
103         return result.toString();
104     }
105
106     /*
107      * Other non-kernel methods ------------------------------------------
108      */
109
110     // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
111     @Override
112     public T front() {
113         assert this.length() > 0 : "Violation of: this /= <>";
114
```

```
115         T front = this.dequeue();
116         T next = this.dequeue();
117         this.enqueue(front);
118 
119         while (!front.equals(next)) {
120             this.enqueue(next);
121             next = this.dequeue();
122         }
123 
124         this.enqueue(next);
125 
126         return front;
127     }
128 
129     // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
130     @Override
131     public T replaceFront(T x) {
132         assert this.length() > 0 : "Violation of: this /= <>";
133 
134         T front = this.dequeue();
135         T next = this.dequeue();
136         this.enqueue(x);
137 
138         while (!x.equals(next)) {
139             this.enqueue(next);
140             next = this.dequeue();
141         }
142 
143         this.enqueue(next);
144 
145         return front;
146     }
147 
148     // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
149     @Override
150     public void merge(WaitingLine<T> w) {
151         assert w != null : "Violation of: q is not null";
152         assert w != this : "Violation of: q is not this";
153 
154         WaitingLine<T> thisNew = this.newInstance();
155         thisNew.clear();
156 
157         T next1 = this.dequeue();
158         T next2 = w.dequeue();
159 
160         while (!next1.equals(null) && !next2.equals(null)) {
161             thisNew.enqueue(next1);
162             thisNew.enqueue(next2);
163 
164             next1 = this.dequeue();
165             next2 = w.dequeue();
166         }
167         while (!next1.equals(null)) {
168             thisNew.enqueue(next1);
169 
170             next1 = this.dequeue();
171         }
```

```
172         while (!next2.equals(null)) {
173             thisNew.enqueue(next2);
174 
175             next2 = w.dequeue();
176         }
177 
178         this.transferFrom(thisNew);
179 
180     }
181 
182     // CHECKSTYLE: ALLOW THIS METHOD TO BE OVERRIDDEN
183     @Override
184     public WaitingLine<T> split(int x) {
185         assert x < this.length() : "Violation of: x is less than this.length";
186         assert x < 0 : "Violation of: x is positive";
187 
188         WaitingLine<T> frontLine = this.newInstance();
189         WaitingLine<T> rearLine = this.newInstance();
190 
191         for (int i = 0; i < x; i++) {
192             frontLine.enqueue(this.dequeue());
193         }
194         while (this.length() >= 1) {
195             rearLine.enqueue(this.dequeue());
196         }
197 
198         this.transferFrom(frontLine);
199 
200         return rearLine;
201     }
202 
203 }
204 
```