```java
/**
package dataStructures;

/**
 * @author Ricardo Gaspar Nr. 35277
 * @author Hugo António Nr. 34334 Turno P2 Docente Vasco Amaral
 */
public class OrderedDoublyLL<K extends Comparable<K>, V> implements
        OrderedDictionary<K, V> {

    private static final long serialVersionUID = 1L;

    // Node at the head of the list.
    protected DListNode<Entry<K, V>> head;

    // Node at the tail of the list.
    protected DListNode<Entry<K, V>> tail;

    // Number of elements in the list.
    protected int currentSize;

    /**
     * Construtor da OrderedDoublyLL. Inicializa a cabeca, cauda e o tamanho da
     * lista.
     */
    public OrderedDoublyLL() {
        head = null;
        tail = null;
        currentSize = 0;
    }

    @Override
    public boolean isEmpty() {
        return currentSize == 0;
    }

    @Override
    public int size() {
        return currentSize;
    }
```

```
43
44      @Override
45      public V find(K key) {
46          V value = null;
47          if (!this.isEmpty()) {
48              DListNode<Entry<K, V>> node = findNode(key);
49              if (node != null &&
  node.getElement().getKey().compareTo(key) == 0)
50                  value = node.getElement().getValue();
51          }
52          return value;
53      }
54
55      @Override
56      public V insert(K key, V value) {
57
58          Entry<K, V> newEntry = new EntryClass<K, V>(key, value);
59          if (this.isEmpty()) {
60              addFirst(newEntry);
61              return null;
62          }
63
64          DListNode<Entry<K, V>> nodeFound = findNode(key);
65          if (nodeFound != null) {
66              Entry<K, V> entryFound = nodeFound.getElement();
67
68              if (entryFound.getKey().compareTo(key) == 0) {
69                  V resultValue = entryFound.getValue();
70                  nodeFound.setElement(newEntry);
71                  return resultValue;
72
73              } else if (entryFound.getKey().compareTo(key) > 0) {
74                  if (nodeFound == head)
75                      addFirst(newEntry);
76                  else
77                      addMiddle(newEntry, nodeFound);
78                  return null;
79              }
80          }
81          addLast(newEntry);
82          return null;
```

```
 83     }
 84
 85     @Override
 86     public V remove(K key) {
 87
 88         if (this.isEmpty())
 89             return null;
 90
 91         DListNode<Entry<K, V>> nodeFound = findNode(key);
 92
 93         if (nodeFound != null) {
 94             Entry<K, V> entryFound = nodeFound.getElement();
 95             if (entryFound.getKey().compareTo(key) == 0) {
 96                 V resultValue = entryFound.getValue();
 97                 if (nodeFound == head)
 98                     removeFirstNode();
 99                 else if (nodeFound == tail)
100                     removeLastNode();
101                 else
102                     removeMiddleNode(nodeFound);
103
104                 return resultValue;
105             }
106         }
107
108         return null;
109     }
110
111     @Override
112     public Iterator<Entry<K, V>> iterator() {
113
114         return new DoublyLLIterator<Entry<K, V>>(head, tail);
115     }
116
117     @Override
118     public Entry<K, V> minEntry() throws EmptyDictionaryException {
119
120         if (this.isEmpty())
121             return null;
122         return head.getElement();
123     }
```

```
124
125     @Override
126     public Entry<K, V> maxEntry() throws EmptyDictionaryException {
127
128         if (this.isEmpty())
129             return null;
130         return tail.getElement();
131     }
132
133     /**
134      * Métodos Auxiliares
135      */
136
137     /**
138      * Devolve o um nó da lista que contenha a chave dada. Caso não
    exista
139      * devolve o primeiro cuja chave é maior. Devolve <code>null</
    code> se
140      * chegar ao fim da lista sem encontrar o nó pretendido.
141      *
142      * @param key
143      *             Chave do elemento a procurar.
144      * @return Devolve um <code>DListNode< Entry< K,V > ></code>
    com a chave (
145      *         <code>key</code>) dada, caso exista. Caso não
    exista: -
146      *         <code>DListNode< Entry< K,V > ></code> do primeiro
    nó cuja chave
147      *         é maior que <code>key</code>. - <code>null</code>
    quando
148      *         <code>key</code> É maior que todas as chaves
    existentes.
149      */
150     protected DListNode<Entry<K, V>> findNode(K key) {
151         DListNode<Entry<K, V>> node = head;
152         while (node != null &&
    node.getElement().getKey().compareTo(key) < 0)
153             node = node.getNext();
154         return node;
155     }
156
```

```
157     /**
158      * Adiciona um elemento à cabeça da lista.
159      *
160      * @param element
161      *            Elemento a inserir.
162      */
163     protected void addFirst(Entry<K, V> element) {
164         DListNode<Entry<K, V>> newNode = new DListNode<Entry<K,
    V>>(element,
165                 null, head);
166         if (this.isEmpty())
167             tail = newNode;
168         else
169             head.setPrevious(newNode);
170         head = newNode;
171         currentSize++;
172     }
173 
174     /**
175      * Adiciona um elemento à cauda da lista.
176      *
177      * @param element
178      *            Elemento a inserir.
179      */
180     protected void addLast(Entry<K, V> element) {
181 
182         DListNode<Entry<K, V>> newNode = new DListNode<Entry<K,
    V>>(element,
183                 tail, null);
184         if (this.isEmpty())
185             head = newNode;
186         else
187             tail.setNext(newNode);
188         tail = newNode;
189         currentSize++;
190     }
191 
192     /**
193      * Adiciona um elemento no meio da lista (entre dois nós). O
    elemento é
194      * adicionado antes de um dado nó.
```

```
195      *
196      * @param element
197      *            Elemento a inserir.
198      * @param node
199      *            Nó seguinte ao elemento.
200      */
201     protected void addMiddle(Entry<K, V> element,
    DListNode<Entry<K, V>> node) {
202 
203         DListNode<Entry<K, V>> PrevNode = node.getPrevious();
204         DListNode<Entry<K, V>> newNode = new DListNode<Entry<K,
    V>>(element,
205                 PrevNode, node);
206         PrevNode.setNext(newNode);
207         node.setPrevious(newNode);
208         currentSize++;
209     }
210 
211     /**
212      * Remove a cabeça da lista.
213      *
214      * @Pre: A lista não está vazia
215      */
216     protected void removeFirstNode() {
217 
218         head = head.getNext();
219         if (head == null)
220             tail = null;
221         else
222             head.setPrevious(null);
223         currentSize--;
224     }
225 
226     /**
227      * Remove a cauda da lista.
228      *
229      * @Pre: A lista não está vazia
230      */
231     protected void removeLastNode() {
232 
233         tail = tail.getPrevious();
```

```
234         if (tail == null)
235             head = null;
236         else
237             tail.setNext(null);
238         currentSize--;
239     }
240 
241     /**
242      * Remove um dado nó que se encontra a meio da lista.
243      *
244      * @Pre: O nó não é cabeça nem cauda da lista.
245      */
246     protected void removeMiddleNode(DListNode<Entry<K, V>> node) {
247 
248         DListNode<Entry<K, V>> prevNode = node.getPrevious();
249         DListNode<Entry<K, V>> nextNode = node.getNext();
250         prevNode.setNext(nextNode);
251         nextNode.setPrevious(prevNode);
252         currentSize--;
253     }
254 
255 }
256 
```