001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.bidimap;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.ArrayList;  
024import java.util.Comparator;  
025import java.util.Iterator;  
026import java.util.ListIterator;  
027import java.util.Map;  
028import java.util.SortedMap;  
029import java.util.TreeMap;  
030  
031import org.apache.commons.collections4.BidiMap;  
032import org.apache.commons.collections4.OrderedBidiMap;  
033import org.apache.commons.collections4.OrderedMap;  
034import org.apache.commons.collections4.OrderedMapIterator;  
035import org.apache.commons.collections4.ResettableIterator;  
036import org.apache.commons.collections4.SortedBidiMap;  
037import org.apache.commons.collections4.map.AbstractSortedMapDecorator;  
038  
039/\*\*  
040 \* Implementation of {@link BidiMap} that uses two {@link TreeMap} instances.  
041 \* <p>  
042 \* The setValue() method on iterators will succeed only if the new value being set is  
043 \* not already in the bidimap.  
044 \* </p>  
045 \* <p>  
046 \* When considering whether to use this class, the {@link TreeBidiMap} class should  
047 \* also be considered. It implements the interface using a dedicated design, and does  
048 \* not store each object twice, which can save on memory use.  
049 \* </p>  
050 \* <p>  
051 \* NOTE: From Commons Collections 3.1, all subclasses will use {@link TreeMap}  
052 \* and the flawed <code>createMap</code> method is ignored.  
053 \* </p>  
054 \*  
055 \* @param <K> the type of the keys in this map  
056 \* @param <V> the type of the values in this map  
057 \* @since 3.0  
058 \*/  
059public class DualTreeBidiMap<K, V> extends AbstractDualBidiMap<K, V>  
060 implements SortedBidiMap<K, V>, Serializable {  
061  
062 /\*\* Ensure serialization compatibility \*/  
063 private static final long serialVersionUID = 721969328361809L;  
064  
065 /\*\* The key comparator to use \*/  
066 private final Comparator<? super K> comparator;  
067  
068 /\*\* The value comparator to use \*/  
069 private final Comparator<? super V> valueComparator;  
070  
071 /\*\*  
072 \* Creates an empty <code>DualTreeBidiMap</code>  
073 \*/  
074 public DualTreeBidiMap() {  
075 super(new TreeMap<K, V>(), new TreeMap<V, K>());  
076 this.comparator = null;  
077 this.valueComparator = null;  
078 }  
079  
080 /\*\*  
081 \* Constructs a <code>DualTreeBidiMap</code> and copies the mappings from  
082 \* specified <code>Map</code>.  
083 \*  
084 \* @param map the map whose mappings are to be placed in this map  
085 \*/  
086 public DualTreeBidiMap(final Map<? extends K, ? extends V> map) {  
087 super(new TreeMap<K, V>(), new TreeMap<V, K>());  
088 putAll(map);  
089 this.comparator = null;  
090 this.valueComparator = null;  
091 }  
092  
093 /\*\*  
094 \* Constructs a {@link DualTreeBidiMap} using the specified {@link Comparator}.  
095 \*  
096 \* @param keyComparator the comparator  
097 \* @param valueComparator the values comparator to use  
098 \*/  
099 public DualTreeBidiMap(final Comparator<? super K> keyComparator, final Comparator<? super V> valueComparator) {  
100 super(new TreeMap<K, V>(keyComparator), new TreeMap<V, K>(valueComparator));  
101 this.comparator = keyComparator;  
102 this.valueComparator = valueComparator;  
103 }  
104  
105 /\*\*  
106 \* Constructs a {@link DualTreeBidiMap} that decorates the specified maps.  
107 \*  
108 \* @param normalMap the normal direction map  
109 \* @param reverseMap the reverse direction map  
110 \* @param inverseBidiMap the inverse BidiMap  
111 \*/  
112 protected DualTreeBidiMap(final Map<K, V> normalMap, final Map<V, K> reverseMap,  
113 final BidiMap<V, K> inverseBidiMap) {  
114 super(normalMap, reverseMap, inverseBidiMap);  
115 this.comparator = ((SortedMap<K, V>) normalMap).comparator();  
116 this.valueComparator = ((SortedMap<V, K>) reverseMap).comparator();  
117 }  
118  
119 /\*\*  
120 \* Creates a new instance of this object.  
121 \*  
122 \* @param normalMap the normal direction map  
123 \* @param reverseMap the reverse direction map  
124 \* @param inverseMap the inverse BidiMap  
125 \* @return new bidi map  
126 \*/  
127 @Override  
128 protected DualTreeBidiMap<V, K> createBidiMap(final Map<V, K> normalMap, final Map<K, V> reverseMap,  
129 final BidiMap<K, V> inverseMap) {  
130 return new DualTreeBidiMap<>(normalMap, reverseMap, inverseMap);  
131 }  
132  
133 //-----------------------------------------------------------------------  
134  
135 @Override  
136 public Comparator<? super K> comparator() {  
137 return ((SortedMap<K, V>) normalMap).comparator();  
138 }  
139  
140 @Override  
141 public Comparator<? super V> valueComparator() {  
142 return ((SortedMap<V, K>) reverseMap).comparator();  
143 }  
144  
145 @Override  
146 public K firstKey() {  
147 return ((SortedMap<K, V>) normalMap).firstKey();  
148 }  
149  
150 @Override  
151 public K lastKey() {  
152 return ((SortedMap<K, V>) normalMap).lastKey();  
153 }  
154  
155 @Override  
156 public K nextKey(final K key) {  
157 if (isEmpty()) {  
158 return null;  
159 }  
160 if (normalMap instanceof OrderedMap) {  
161 return ((OrderedMap<K, ?>) normalMap).nextKey(key);  
162 }  
163 final SortedMap<K, V> sm = (SortedMap<K, V>) normalMap;  
164 final Iterator<K> it = sm.tailMap(key).keySet().iterator();  
165 it.next();  
166 if (it.hasNext()) {  
167 return it.next();  
168 }  
169 return null;  
170 }  
171  
172 @Override  
173 public K previousKey(final K key) {  
174 if (isEmpty()) {  
175 return null;  
176 }  
177 if (normalMap instanceof OrderedMap) {  
178 return ((OrderedMap<K, V>) normalMap).previousKey(key);  
179 }  
180 final SortedMap<K, V> sm = (SortedMap<K, V>) normalMap;  
181 final SortedMap<K, V> hm = sm.headMap(key);  
182 if (hm.isEmpty()) {  
183 return null;  
184 }  
185 return hm.lastKey();  
186 }  
187  
188 //-----------------------------------------------------------------------  
189 /\*\*  
190 \* Obtains an ordered map iterator.  
191 \* <p>  
192 \* This implementation copies the elements to an ArrayList in order to  
193 \* provide the forward/backward behaviour.  
194 \*  
195 \* @return a new ordered map iterator  
196 \*/  
197 @Override  
198 public OrderedMapIterator<K, V> mapIterator() {  
199 return new BidiOrderedMapIterator<>(this);  
200 }  
201  
202 public SortedBidiMap<V, K> inverseSortedBidiMap() {  
203 return inverseBidiMap();  
204 }  
205  
206 public OrderedBidiMap<V, K> inverseOrderedBidiMap() {  
207 return inverseBidiMap();  
208 }  
209  
210 //-----------------------------------------------------------------------  
211  
212 @Override  
213 public SortedMap<K, V> headMap(final K toKey) {  
214 final SortedMap<K, V> sub = ((SortedMap<K, V>) normalMap).headMap(toKey);  
215 return new ViewMap<>(this, sub);  
216 }  
217  
218 @Override  
219 public SortedMap<K, V> tailMap(final K fromKey) {  
220 final SortedMap<K, V> sub = ((SortedMap<K, V>) normalMap).tailMap(fromKey);  
221 return new ViewMap<>(this, sub);  
222 }  
223  
224 @Override  
225 public SortedMap<K, V> subMap(final K fromKey, final K toKey) {  
226 final SortedMap<K, V> sub = ((SortedMap<K, V>) normalMap).subMap(fromKey, toKey);  
227 return new ViewMap<>(this, sub);  
228 }  
229  
230 @Override  
231 public SortedBidiMap<V, K> inverseBidiMap() {  
232 return (SortedBidiMap<V, K>) super.inverseBidiMap();  
233 }  
234  
235 //-----------------------------------------------------------------------  
236 /\*\*  
237 \* Internal sorted map view.  
238 \*/  
239 protected static class ViewMap<K, V> extends AbstractSortedMapDecorator<K, V> {  
240 /\*\*  
241 \* Constructor.  
242 \* @param bidi the parent bidi map  
243 \* @param sm the subMap sorted map  
244 \*/  
245 protected ViewMap(final DualTreeBidiMap<K, V> bidi, final SortedMap<K, V> sm) {  
246 // the implementation is not great here...  
247 // use the normalMap as the filtered map, but reverseMap as the full map  
248 // this forces containsValue and clear to be overridden  
249 super(new DualTreeBidiMap<>(sm, bidi.reverseMap, bidi.inverseBidiMap));  
250 }  
251  
252 @Override  
253 public boolean containsValue(final Object value) {  
254 // override as default implementation uses reverseMap  
255 return decorated().normalMap.containsValue(value);  
256 }  
257  
258 @Override  
259 public void clear() {  
260 // override as default implementation uses reverseMap  
261 for (final Iterator<K> it = keySet().iterator(); it.hasNext();) {  
262 it.next();  
263 it.remove();  
264 }  
265 }  
266  
267 @Override  
268 public SortedMap<K, V> headMap(final K toKey) {  
269 return new ViewMap<>(decorated(), super.headMap(toKey));  
270 }  
271  
272 @Override  
273 public SortedMap<K, V> tailMap(final K fromKey) {  
274 return new ViewMap<>(decorated(), super.tailMap(fromKey));  
275 }  
276  
277 @Override  
278 public SortedMap<K, V> subMap(final K fromKey, final K toKey) {  
279 return new ViewMap<>(decorated(), super.subMap(fromKey, toKey));  
280 }  
281  
282 @Override  
283 protected DualTreeBidiMap<K, V> decorated() {  
284 return (DualTreeBidiMap<K, V>) super.decorated();  
285 }  
286  
287 @Override  
288 public K previousKey(final K key) {  
289 return decorated().previousKey(key);  
290 }  
291  
292 @Override  
293 public K nextKey(final K key) {  
294 return decorated().nextKey(key);  
295 }  
296 }  
297  
298 //-----------------------------------------------------------------------  
299 /\*\*  
300 \* Inner class MapIterator.  
301 \*/  
302 protected static class BidiOrderedMapIterator<K, V> implements OrderedMapIterator<K, V>, ResettableIterator<K> {  
303  
304 /\*\* The parent map \*/  
305 private final AbstractDualBidiMap<K, V> parent;  
306  
307 /\*\* The iterator being decorated \*/  
308 private ListIterator<Map.Entry<K, V>> iterator;  
309  
310 /\*\* The last returned entry \*/  
311 private Map.Entry<K, V> last = null;  
312  
313 /\*\*  
314 \* Constructor.  
315 \* @param parent the parent map  
316 \*/  
317 protected BidiOrderedMapIterator(final AbstractDualBidiMap<K, V> parent) {  
318 super();  
319 this.parent = parent;  
320 iterator = new ArrayList<>(parent.entrySet()).listIterator();  
321 }  
322  
323 @Override  
324 public boolean hasNext() {  
325 return iterator.hasNext();  
326 }  
327  
328 @Override  
329 public K next() {  
330 last = iterator.next();  
331 return last.getKey();  
332 }  
333  
334 @Override  
335 public boolean hasPrevious() {  
336 return iterator.hasPrevious();  
337 }  
338  
339 @Override  
340 public K previous() {  
341 last = iterator.previous();  
342 return last.getKey();  
343 }  
344  
345 @Override  
346 public void remove() {  
347 iterator.remove();  
348 parent.remove(last.getKey());  
349 last = null;  
350 }  
351  
352 @Override  
353 public K getKey() {  
354 if (last == null) {  
355 throw new IllegalStateException(  
356 "Iterator getKey() can only be called after next() and before remove()");  
357 }  
358 return last.getKey();  
359 }  
360  
361 @Override  
362 public V getValue() {  
363 if (last == null) {  
364 throw new IllegalStateException(  
365 "Iterator getValue() can only be called after next() and before remove()");  
366 }  
367 return last.getValue();  
368 }  
369  
370 @Override  
371 public V setValue(final V value) {  
372 if (last == null) {  
373 throw new IllegalStateException(  
374 "Iterator setValue() can only be called after next() and before remove()");  
375 }  
376 if (parent.reverseMap.containsKey(value) &&  
377 parent.reverseMap.get(value) != last.getKey()) {  
378 throw new IllegalArgumentException(  
379 "Cannot use setValue() when the object being set is already in the map");  
380 }  
381 final V oldValue = parent.put(last.getKey(), value);  
382 // Map.Entry specifies that the behavior is undefined when the backing map  
383 // has been modified (as we did with the put), so we also set the value  
384 last.setValue(value);  
385 return oldValue;  
386 }  
387  
388 @Override  
389 public void reset() {  
390 iterator = new ArrayList<>(parent.entrySet()).listIterator();  
391 last = null;  
392 }  
393  
394 @Override  
395 public String toString() {  
396 if (last != null) {  
397 return "MapIterator[" + getKey() + "=" + getValue() + "]";  
398 }  
399 return "MapIterator[]";  
400 }  
401 }  
402  
403 // Serialization  
404 //-----------------------------------------------------------------------  
405 private void writeObject(final ObjectOutputStream out) throws IOException {  
406 out.defaultWriteObject();  
407 out.writeObject(normalMap);  
408 }  
409  
410 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
411 in.defaultReadObject();  
412 normalMap = new TreeMap<>(comparator);  
413 reverseMap = new TreeMap<>(valueComparator);  
414 @SuppressWarnings("unchecked") // will fail at runtime if the stream is incorrect  
415 final Map<K, V> map = (Map<K, V>) in.readObject();  
416 putAll(map);  
417 }  
418  
419}