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.map;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.AbstractList;  
024import java.util.AbstractSet;  
025import java.util.ArrayList;  
026import java.util.Collection;  
027import java.util.HashMap;  
028import java.util.Iterator;  
029import java.util.List;  
030import java.util.ListIterator;  
031import java.util.Map;  
032import java.util.NoSuchElementException;  
033import java.util.Set;  
034  
035import org.apache.commons.collections4.OrderedMap;  
036import org.apache.commons.collections4.OrderedMapIterator;  
037import org.apache.commons.collections4.ResettableIterator;  
038import org.apache.commons.collections4.iterators.AbstractUntypedIteratorDecorator;  
039import org.apache.commons.collections4.keyvalue.AbstractMapEntry;  
040import org.apache.commons.collections4.list.UnmodifiableList;  
041  
042/\*\*  
043 \* Decorates a <code>Map</code> to ensure that the order of addition is retained  
044 \* using a <code>List</code> to maintain order.  
045 \* <p>  
046 \* The order will be used via the iterators and toArray methods on the views.  
047 \* The order is also returned by the <code>MapIterator</code>.  
048 \* The <code>orderedMapIterator()</code> method accesses an iterator that can  
049 \* iterate both forwards and backwards through the map.  
050 \* In addition, non-interface methods are provided to access the map by index.  
051 \* </p>  
052 \* <p>  
053 \* If an object is added to the Map for a second time, it will remain in the  
054 \* original position in the iteration.  
055 \* </p>  
056 \* <p>  
057 \* <strong>Note that ListOrderedMap is not synchronized and is not thread-safe.</strong>  
058 \* If you wish to use this map from multiple threads concurrently, you must use  
059 \* appropriate synchronization. The simplest approach is to wrap this map  
060 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
061 \* exceptions when accessed by concurrent threads without synchronization.  
062 \* </p>  
063 \* <p>  
064 \* <strong>Note that ListOrderedMap doesn't work with  
065 \* {@link java.util.IdentityHashMap IdentityHashMap}, {@link CaseInsensitiveMap},  
066 \* or similar maps that violate the general contract of {@link java.util.Map}.</strong>  
067 \* The <code>ListOrderedMap</code> (or, more precisely, the underlying <code>List</code>)  
068 \* is relying on {@link Object#equals(Object) equals()}. This is fine, as long as the  
069 \* decorated <code>Map</code> is also based on {@link Object#equals(Object) equals()},  
070 \* and {@link Object#hashCode() hashCode()}, which  
071 \* {@link java.util.IdentityHashMap IdentityHashMap}, and  
072 \* {@link CaseInsensitiveMap} don't: The former uses <code>==</code>, and  
073 \* the latter uses {@link Object#equals(Object) equals()} on a lower-cased  
074 \* key.  
075 \* </p>  
076 \* <p>  
077 \* This class is {@link Serializable} starting with Commons Collections 3.1.  
078 \* </p>  
079 \*  
080 \* @param <K> the type of the keys in this map  
081 \* @param <V> the type of the values in this map  
082 \* @since 3.0  
083 \*/  
084public class ListOrderedMap<K, V>  
085 extends AbstractMapDecorator<K, V>  
086 implements OrderedMap<K, V>, Serializable {  
087  
088 /\*\* Serialization version \*/  
089 private static final long serialVersionUID = 2728177751851003750L;  
090  
091 /\*\* Internal list to hold the sequence of objects \*/  
092 private final List<K> insertOrder = new ArrayList<>();  
093  
094 /\*\*  
095 \* Factory method to create an ordered map.  
096 \* <p>  
097 \* An <code>ArrayList</code> is used to retain order.  
098 \*  
099 \* @param <K> the key type  
100 \* @param <V> the value type  
101 \* @param map the map to decorate, must not be null  
102 \* @return a new list ordered map  
103 \* @throws NullPointerException if map is null  
104 \* @since 4.0  
105 \*/  
106 public static <K, V> ListOrderedMap<K, V> listOrderedMap(final Map<K, V> map) {  
107 return new ListOrderedMap<>(map);  
108 }  
109  
110 //-----------------------------------------------------------------------  
111 /\*\*  
112 \* Constructs a new empty <code>ListOrderedMap</code> that decorates  
113 \* a <code>HashMap</code>.  
114 \*  
115 \* @since 3.1  
116 \*/  
117 public ListOrderedMap() {  
118 this(new HashMap<K, V>());  
119 }  
120  
121 /\*\*  
122 \* Constructor that wraps (not copies).  
123 \*  
124 \* @param map the map to decorate, must not be null  
125 \* @throws NullPointerException if map is null  
126 \*/  
127 protected ListOrderedMap(final Map<K, V> map) {  
128 super(map);  
129 insertOrder.addAll(decorated().keySet());  
130 }  
131  
132 //-----------------------------------------------------------------------  
133 /\*\*  
134 \* Write the map out using a custom routine.  
135 \*  
136 \* @param out the output stream  
137 \* @throws IOException if an error occurs while writing to the stream  
138 \* @since 3.1  
139 \*/  
140 private void writeObject(final ObjectOutputStream out) throws IOException {  
141 out.defaultWriteObject();  
142 out.writeObject(map);  
143 }  
144  
145 /\*\*  
146 \* Read the map in using a custom routine.  
147 \*  
148 \* @param in the input stream  
149 \* @throws IOException if an error occurs while reading from the stream  
150 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
151 \* @since 3.1  
152 \*/  
153 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
154 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
155 in.defaultReadObject();  
156 map = (Map<K, V>) in.readObject(); // (1)  
157 }  
158  
159 // Implement OrderedMap  
160 //-----------------------------------------------------------------------  
161 @Override  
162 public OrderedMapIterator<K, V> mapIterator() {  
163 return new ListOrderedMapIterator<>(this);  
164 }  
165  
166 /\*\*  
167 \* Gets the first key in this map by insert order.  
168 \*  
169 \* @return the first key currently in this map  
170 \* @throws NoSuchElementException if this map is empty  
171 \*/  
172 @Override  
173 public K firstKey() {  
174 if (size() == 0) {  
175 throw new NoSuchElementException("Map is empty");  
176 }  
177 return insertOrder.get(0);  
178 }  
179  
180 /\*\*  
181 \* Gets the last key in this map by insert order.  
182 \*  
183 \* @return the last key currently in this map  
184 \* @throws NoSuchElementException if this map is empty  
185 \*/  
186 @Override  
187 public K lastKey() {  
188 if (size() == 0) {  
189 throw new NoSuchElementException("Map is empty");  
190 }  
191 return insertOrder.get(size() - 1);  
192 }  
193  
194 /\*\*  
195 \* Gets the next key to the one specified using insert order.  
196 \* This method performs a list search to find the key and is O(n).  
197 \*  
198 \* @param key the key to find previous for  
199 \* @return the next key, null if no match or at start  
200 \*/  
201 @Override  
202 public K nextKey(final Object key) {  
203 final int index = insertOrder.indexOf(key);  
204 if (index >= 0 && index < size() - 1) {  
205 return insertOrder.get(index + 1);  
206 }  
207 return null;  
208 }  
209  
210 /\*\*  
211 \* Gets the previous key to the one specified using insert order.  
212 \* This method performs a list search to find the key and is O(n).  
213 \*  
214 \* @param key the key to find previous for  
215 \* @return the previous key, null if no match or at start  
216 \*/  
217 @Override  
218 public K previousKey(final Object key) {  
219 final int index = insertOrder.indexOf(key);  
220 if (index > 0) {  
221 return insertOrder.get(index - 1);  
222 }  
223 return null;  
224 }  
225  
226 //-----------------------------------------------------------------------  
227 @Override  
228 public V put(final K key, final V value) {  
229 if (decorated().containsKey(key)) {  
230 // re-adding doesn't change order  
231 return decorated().put(key, value);  
232 }  
233 // first add, so add to both map and list  
234 final V result = decorated().put(key, value);  
235 insertOrder.add(key);  
236 return result;  
237 }  
238  
239 @Override  
240 public void putAll(final Map<? extends K, ? extends V> map) {  
241 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
242 put(entry.getKey(), entry.getValue());  
243 }  
244 }  
245  
246 /\*\*  
247 \* Puts the values contained in a supplied Map into the Map starting at  
248 \* the specified index.  
249 \*  
250 \* @param index the index in the Map to start at.  
251 \* @param map the Map containing the entries to be added.  
252 \* @throws IndexOutOfBoundsException if the index is out of range [0, size]  
253 \*/  
254 public void putAll(int index, final Map<? extends K, ? extends V> map) {  
255 if (index < 0 || index > insertOrder.size()) {  
256 throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + insertOrder.size());  
257 }  
258 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
259 final K key = entry.getKey();  
260 final boolean contains = containsKey(key);  
261 // The return value of put is null if the key did not exist OR the value was null  
262 // so it cannot be used to determine whether the key was added  
263 put(index, entry.getKey(), entry.getValue());  
264 if (!contains) {  
265 // if no key was replaced, increment the index  
266 index++;  
267 } else {  
268 // otherwise put the next item after the currently inserted key  
269 index = indexOf(entry.getKey()) + 1;  
270 }  
271 }  
272 }  
273  
274 @Override  
275 public V remove(final Object key) {  
276 V result = null;  
277 if (decorated().containsKey(key)) {  
278 result = decorated().remove(key);  
279 insertOrder.remove(key);  
280 }  
281 return result;  
282 }  
283  
284 @Override  
285 public void clear() {  
286 decorated().clear();  
287 insertOrder.clear();  
288 }  
289  
290 //-----------------------------------------------------------------------  
291 /\*\*  
292 \* Gets a view over the keys in the map.  
293 \* <p>  
294 \* The Collection will be ordered by object insertion into the map.  
295 \*  
296 \* @see #keyList()  
297 \* @return the fully modifiable collection view over the keys  
298 \*/  
299 @Override  
300 public Set<K> keySet() {  
301 return new KeySetView<>(this);  
302 }  
303  
304 /\*\*  
305 \* Gets a view over the keys in the map as a List.  
306 \* <p>  
307 \* The List will be ordered by object insertion into the map.  
308 \* The List is unmodifiable.  
309 \*  
310 \* @see #keySet()  
311 \* @return the unmodifiable list view over the keys  
312 \* @since 3.2  
313 \*/  
314 public List<K> keyList() {  
315 return UnmodifiableList.unmodifiableList(insertOrder);  
316 }  
317  
318 /\*\*  
319 \* Gets a view over the values in the map.  
320 \* <p>  
321 \* The Collection will be ordered by object insertion into the map.  
322 \* <p>  
323 \* From Commons Collections 3.2, this Collection can be cast  
324 \* to a list, see {@link #valueList()}  
325 \*  
326 \* @see #valueList()  
327 \* @return the fully modifiable collection view over the values  
328 \*/  
329 @Override  
330 public Collection<V> values() {  
331 return new ValuesView<>(this);  
332 }  
333  
334 /\*\*  
335 \* Gets a view over the values in the map as a List.  
336 \* <p>  
337 \* The List will be ordered by object insertion into the map.  
338 \* The List supports remove and set, but does not support add.  
339 \*  
340 \* @see #values()  
341 \* @return the partially modifiable list view over the values  
342 \* @since 3.2  
343 \*/  
344 public List<V> valueList() {  
345 return new ValuesView<>(this);  
346 }  
347  
348 /\*\*  
349 \* Gets a view over the entries in the map.  
350 \* <p>  
351 \* The Set will be ordered by object insertion into the map.  
352 \*  
353 \* @return the fully modifiable set view over the entries  
354 \*/  
355 @Override  
356 public Set<Map.Entry<K, V>> entrySet() {  
357 return new EntrySetView<>(this, this.insertOrder);  
358 }  
359  
360 //-----------------------------------------------------------------------  
361 /\*\*  
362 \* Returns the Map as a string.  
363 \*  
364 \* @return the Map as a String  
365 \*/  
366 @Override  
367 public String toString() {  
368 if (isEmpty()) {  
369 return "{}";  
370 }  
371 final StringBuilder buf = new StringBuilder();  
372 buf.append('{');  
373 boolean first = true;  
374 for (final Map.Entry<K, V> entry : entrySet()) {  
375 final K key = entry.getKey();  
376 final V value = entry.getValue();  
377 if (first) {  
378 first = false;  
379 } else {  
380 buf.append(", ");  
381 }  
382 buf.append(key == this ? "(this Map)" : key);  
383 buf.append('=');  
384 buf.append(value == this ? "(this Map)" : value);  
385 }  
386 buf.append('}');  
387 return buf.toString();  
388 }  
389  
390 //-----------------------------------------------------------------------  
391 /\*\*  
392 \* Gets the key at the specified index.  
393 \*  
394 \* @param index the index to retrieve  
395 \* @return the key at the specified index  
396 \* @throws IndexOutOfBoundsException if the index is invalid  
397 \*/  
398 public K get(final int index) {  
399 return insertOrder.get(index);  
400 }  
401  
402 /\*\*  
403 \* Gets the value at the specified index.  
404 \*  
405 \* @param index the index to retrieve  
406 \* @return the key at the specified index  
407 \* @throws IndexOutOfBoundsException if the index is invalid  
408 \*/  
409 public V getValue(final int index) {  
410 return get(insertOrder.get(index));  
411 }  
412  
413 /\*\*  
414 \* Gets the index of the specified key.  
415 \*  
416 \* @param key the key to find the index of  
417 \* @return the index, or -1 if not found  
418 \*/  
419 public int indexOf(final Object key) {  
420 return insertOrder.indexOf(key);  
421 }  
422  
423 /\*\*  
424 \* Sets the value at the specified index.  
425 \*  
426 \* @param index the index of the value to set  
427 \* @param value the new value to set  
428 \* @return the previous value at that index  
429 \* @throws IndexOutOfBoundsException if the index is invalid  
430 \* @since 3.2  
431 \*/  
432 public V setValue(final int index, final V value) {  
433 final K key = insertOrder.get(index);  
434 return put(key, value);  
435 }  
436  
437 /\*\*  
438 \* Puts a key-value mapping into the map at the specified index.  
439 \* <p>  
440 \* If the map already contains the key, then the original mapping  
441 \* is removed and the new mapping added at the specified index.  
442 \* The remove may change the effect of the index. The index is  
443 \* always calculated relative to the original state of the map.  
444 \* <p>  
445 \* Thus the steps are: (1) remove the existing key-value mapping,  
446 \* then (2) insert the new key-value mapping at the position it  
447 \* would have been inserted had the remove not occurred.  
448 \*  
449 \* @param index the index at which the mapping should be inserted  
450 \* @param key the key  
451 \* @param value the value  
452 \* @return the value previously mapped to the key  
453 \* @throws IndexOutOfBoundsException if the index is out of range [0, size]  
454 \* @since 3.2  
455 \*/  
456 public V put(int index, final K key, final V value) {  
457 if (index < 0 || index > insertOrder.size()) {  
458 throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + insertOrder.size());  
459 }  
460  
461 final Map<K, V> m = decorated();  
462 if (m.containsKey(key)) {  
463 final V result = m.remove(key);  
464 final int pos = insertOrder.indexOf(key);  
465 insertOrder.remove(pos);  
466 if (pos < index) {  
467 index--;  
468 }  
469 insertOrder.add(index, key);  
470 m.put(key, value);  
471 return result;  
472 }  
473 insertOrder.add(index, key);  
474 m.put(key, value);  
475 return null;  
476 }  
477  
478 /\*\*  
479 \* Removes the element at the specified index.  
480 \*  
481 \* @param index the index of the object to remove  
482 \* @return the removed value, or <code>null</code> if none existed  
483 \* @throws IndexOutOfBoundsException if the index is invalid  
484 \*/  
485 public V remove(final int index) {  
486 return remove(get(index));  
487 }  
488  
489 /\*\*  
490 \* Gets an unmodifiable List view of the keys which changes as the map changes.  
491 \* <p>  
492 \* The returned list is unmodifiable because changes to the values of  
493 \* the list (using {@link java.util.ListIterator#set(Object)}) will  
494 \* effectively remove the value from the list and reinsert that value at  
495 \* the end of the list, which is an unexpected side effect of changing the  
496 \* value of a list. This occurs because changing the key, changes when the  
497 \* mapping is added to the map and thus where it appears in the list.  
498 \* <p>  
499 \* An alternative to this method is to use the better named  
500 \* {@link #keyList()} or {@link #keySet()}.  
501 \*  
502 \* @see #keyList()  
503 \* @see #keySet()  
504 \* @return The ordered list of keys.  
505 \*/  
506 public List<K> asList() {  
507 return keyList();  
508 }  
509  
510 //-----------------------------------------------------------------------  
511 static class ValuesView<V> extends AbstractList<V> {  
512 private final ListOrderedMap<Object, V> parent;  
513  
514 @SuppressWarnings("unchecked")  
515 ValuesView(final ListOrderedMap<?, V> parent) {  
516 super();  
517 this.parent = (ListOrderedMap<Object, V>) parent;  
518 }  
519  
520 @Override  
521 public int size() {  
522 return this.parent.size();  
523 }  
524  
525 @Override  
526 public boolean contains(final Object value) {  
527 return this.parent.containsValue(value);  
528 }  
529  
530 @Override  
531 public void clear() {  
532 this.parent.clear();  
533 }  
534  
535 @Override  
536 public Iterator<V> iterator() {  
537 return new AbstractUntypedIteratorDecorator<Map.Entry<Object, V>, V>(parent.entrySet().iterator()) {  
538 @Override  
539 public V next() {  
540 return getIterator().next().getValue();  
541 }  
542 };  
543 }  
544  
545 @Override  
546 public V get(final int index) {  
547 return this.parent.getValue(index);  
548 }  
549  
550 @Override  
551 public V set(final int index, final V value) {  
552 return this.parent.setValue(index, value);  
553 }  
554  
555 @Override  
556 public V remove(final int index) {  
557 return this.parent.remove(index);  
558 }  
559 }  
560  
561 //-----------------------------------------------------------------------  
562 static class KeySetView<K> extends AbstractSet<K> {  
563 private final ListOrderedMap<K, Object> parent;  
564  
565 @SuppressWarnings("unchecked")  
566 KeySetView(final ListOrderedMap<K, ?> parent) {  
567 super();  
568 this.parent = (ListOrderedMap<K, Object>) parent;  
569 }  
570  
571 @Override  
572 public int size() {  
573 return this.parent.size();  
574 }  
575  
576 @Override  
577 public boolean contains(final Object value) {  
578 return this.parent.containsKey(value);  
579 }  
580  
581 @Override  
582 public void clear() {  
583 this.parent.clear();  
584 }  
585  
586 @Override  
587 public Iterator<K> iterator() {  
588 return new AbstractUntypedIteratorDecorator<Map.Entry<K, Object>, K>(parent.entrySet().iterator()) {  
589 @Override  
590 public K next() {  
591 return getIterator().next().getKey();  
592 }  
593 };  
594 }  
595 }  
596  
597 //-----------------------------------------------------------------------  
598 static class EntrySetView<K, V> extends AbstractSet<Map.Entry<K, V>> {  
599 private final ListOrderedMap<K, V> parent;  
600 private final List<K> insertOrder;  
601 private Set<Map.Entry<K, V>> entrySet;  
602  
603 public EntrySetView(final ListOrderedMap<K, V> parent, final List<K> insertOrder) {  
604 super();  
605 this.parent = parent;  
606 this.insertOrder = insertOrder;  
607 }  
608  
609 private Set<Map.Entry<K, V>> getEntrySet() {  
610 if (entrySet == null) {  
611 entrySet = parent.decorated().entrySet();  
612 }  
613 return entrySet;  
614 }  
615  
616 @Override  
617 public int size() {  
618 return this.parent.size();  
619 }  
620 @Override  
621 public boolean isEmpty() {  
622 return this.parent.isEmpty();  
623 }  
624  
625 @Override  
626 public boolean contains(final Object obj) {  
627 return getEntrySet().contains(obj);  
628 }  
629  
630 @Override  
631 public boolean containsAll(final Collection<?> coll) {  
632 return getEntrySet().containsAll(coll);  
633 }  
634  
635 @Override  
636 @SuppressWarnings("unchecked")  
637 public boolean remove(final Object obj) {  
638 if (obj instanceof Map.Entry == false) {  
639 return false;  
640 }  
641 if (getEntrySet().contains(obj)) {  
642 final Object key = ((Map.Entry<K, V>) obj).getKey();  
643 parent.remove(key);  
644 return true;  
645 }  
646 return false;  
647 }  
648  
649 @Override  
650 public void clear() {  
651 this.parent.clear();  
652 }  
653  
654 @Override  
655 public boolean equals(final Object obj) {  
656 if (obj == this) {  
657 return true;  
658 }  
659 return getEntrySet().equals(obj);  
660 }  
661  
662 @Override  
663 public int hashCode() {  
664 return getEntrySet().hashCode();  
665 }  
666  
667 @Override  
668 public String toString() {  
669 return getEntrySet().toString();  
670 }  
671  
672 @Override  
673 public Iterator<Map.Entry<K, V>> iterator() {  
674 return new ListOrderedIterator<>(parent, insertOrder);  
675 }  
676 }  
677  
678 //-----------------------------------------------------------------------  
679 static class ListOrderedIterator<K, V> extends AbstractUntypedIteratorDecorator<K, Map.Entry<K, V>> {  
680 private final ListOrderedMap<K, V> parent;  
681 private K last = null;  
682  
683 ListOrderedIterator(final ListOrderedMap<K, V> parent, final List<K> insertOrder) {  
684 super(insertOrder.iterator());  
685 this.parent = parent;  
686 }  
687  
688 @Override  
689 public Map.Entry<K, V> next() {  
690 last = getIterator().next();  
691 return new ListOrderedMapEntry<>(parent, last);  
692 }  
693  
694 @Override  
695 public void remove() {  
696 super.remove();  
697 parent.decorated().remove(last);  
698 }  
699 }  
700  
701 //-----------------------------------------------------------------------  
702 static class ListOrderedMapEntry<K, V> extends AbstractMapEntry<K, V> {  
703 private final ListOrderedMap<K, V> parent;  
704  
705 ListOrderedMapEntry(final ListOrderedMap<K, V> parent, final K key) {  
706 super(key, null);  
707 this.parent = parent;  
708 }  
709  
710 @Override  
711 public V getValue() {  
712 return parent.get(getKey());  
713 }  
714  
715 @Override  
716 public V setValue(final V value) {  
717 return parent.decorated().put(getKey(), value);  
718 }  
719 }  
720  
721 //-----------------------------------------------------------------------  
722 static class ListOrderedMapIterator<K, V> implements OrderedMapIterator<K, V>, ResettableIterator<K> {  
723 private final ListOrderedMap<K, V> parent;  
724 private ListIterator<K> iterator;  
725 private K last = null;  
726 private boolean readable = false;  
727  
728 ListOrderedMapIterator(final ListOrderedMap<K, V> parent) {  
729 super();  
730 this.parent = parent;  
731 this.iterator = parent.insertOrder.listIterator();  
732 }  
733  
734 @Override  
735 public boolean hasNext() {  
736 return iterator.hasNext();  
737 }  
738  
739 @Override  
740 public K next() {  
741 last = iterator.next();  
742 readable = true;  
743 return last;  
744 }  
745  
746 @Override  
747 public boolean hasPrevious() {  
748 return iterator.hasPrevious();  
749 }  
750  
751 @Override  
752 public K previous() {  
753 last = iterator.previous();  
754 readable = true;  
755 return last;  
756 }  
757  
758 @Override  
759 public void remove() {  
760 if (readable == false) {  
761 throw new IllegalStateException(AbstractHashedMap.REMOVE\_INVALID);  
762 }  
763 iterator.remove();  
764 parent.map.remove(last);  
765 readable = false;  
766 }  
767  
768 @Override  
769 public K getKey() {  
770 if (readable == false) {  
771 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
772 }  
773 return last;  
774 }  
775  
776 @Override  
777 public V getValue() {  
778 if (readable == false) {  
779 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
780 }  
781 return parent.get(last);  
782 }  
783  
784 @Override  
785 public V setValue(final V value) {  
786 if (readable == false) {  
787 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
788 }  
789 return parent.map.put(last, value);  
790 }  
791  
792 @Override  
793 public void reset() {  
794 iterator = parent.insertOrder.listIterator();  
795 last = null;  
796 readable = false;  
797 }  
798  
799 @Override  
800 public String toString() {  
801 if (readable == true) {  
802 return "Iterator[" + getKey() + "=" + getValue() + "]";  
803 }  
804 return "Iterator[]";  
805 }  
806 }  
807  
808}