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.util.ConcurrentModificationException;  
020import java.util.Iterator;  
021import java.util.Map;  
022import java.util.NoSuchElementException;  
023  
024import org.apache.commons.collections4.OrderedIterator;  
025import org.apache.commons.collections4.OrderedMap;  
026import org.apache.commons.collections4.OrderedMapIterator;  
027import org.apache.commons.collections4.ResettableIterator;  
028import org.apache.commons.collections4.iterators.EmptyOrderedIterator;  
029import org.apache.commons.collections4.iterators.EmptyOrderedMapIterator;  
030  
031/\*\*  
032 \* An abstract implementation of a hash-based map that links entries to create an  
033 \* ordered map and which provides numerous points for subclasses to override.  
034 \* <p>  
035 \* This class implements all the features necessary for a subclass linked  
036 \* hash-based map. Key-value entries are stored in instances of the  
037 \* <code>LinkEntry</code> class which can be overridden and replaced.  
038 \* The iterators can similarly be replaced, without the need to replace the KeySet,  
039 \* EntrySet and Values view classes.  
040 \* </p>  
041 \* <p>  
042 \* Overridable methods are provided to change the default hashing behaviour, and  
043 \* to change how entries are added to and removed from the map. Hopefully, all you  
044 \* need for unusual subclasses is here.  
045 \* </p>  
046 \* <p>  
047 \* This implementation maintains order by original insertion, but subclasses  
048 \* may work differently. The <code>OrderedMap</code> interface is implemented  
049 \* to provide access to bidirectional iteration and extra convenience methods.  
050 \* </p>  
051 \* <p>  
052 \* The <code>orderedMapIterator()</code> method provides direct access to a  
053 \* bidirectional iterator. The iterators from the other views can also be cast  
054 \* to <code>OrderedIterator</code> if required.  
055 \* </p>  
056 \* <p>  
057 \* All the available iterators can be reset back to the start by casting to  
058 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
059 \* </p>  
060 \* <p>  
061 \* The implementation is also designed to be subclassed, with lots of useful  
062 \* methods exposed.  
063 \* </p>  
064 \*  
065 \* @param <K> the type of the keys in this map  
066 \* @param <V> the type of the values in this map  
067 \* @since 3.0  
068 \*/  
069public abstract class AbstractLinkedMap<K, V> extends AbstractHashedMap<K, V> implements OrderedMap<K, V> {  
070  
071 /\*\* Header in the linked list \*/  
072 transient LinkEntry<K, V> header;  
073  
074 /\*\*  
075 \* Constructor only used in deserialization, do not use otherwise.  
076 \*/  
077 protected AbstractLinkedMap() {  
078 super();  
079 }  
080  
081 /\*\*  
082 \* Constructor which performs no validation on the passed in parameters.  
083 \*  
084 \* @param initialCapacity the initial capacity, must be a power of two  
085 \* @param loadFactor the load factor, must be > 0.0f and generally < 1.0f  
086 \* @param threshold the threshold, must be sensible  
087 \*/  
088 protected AbstractLinkedMap(final int initialCapacity, final float loadFactor, final int threshold) {  
089 super(initialCapacity, loadFactor, threshold);  
090 }  
091  
092 /\*\*  
093 \* Constructs a new, empty map with the specified initial capacity.  
094 \*  
095 \* @param initialCapacity the initial capacity  
096 \* @throws IllegalArgumentException if the initial capacity is negative  
097 \*/  
098 protected AbstractLinkedMap(final int initialCapacity) {  
099 super(initialCapacity);  
100 }  
101  
102 /\*\*  
103 \* Constructs a new, empty map with the specified initial capacity and  
104 \* load factor.  
105 \*  
106 \* @param initialCapacity the initial capacity  
107 \* @param loadFactor the load factor  
108 \* @throws IllegalArgumentException if the initial capacity is negative  
109 \* @throws IllegalArgumentException if the load factor is less than zero  
110 \*/  
111 protected AbstractLinkedMap(final int initialCapacity, final float loadFactor) {  
112 super(initialCapacity, loadFactor);  
113 }  
114  
115 /\*\*  
116 \* Constructor copying elements from another map.  
117 \*  
118 \* @param map the map to copy  
119 \* @throws NullPointerException if the map is null  
120 \*/  
121 protected AbstractLinkedMap(final Map<? extends K, ? extends V> map) {  
122 super(map);  
123 }  
124  
125 /\*\*  
126 \* Initialise this subclass during construction.  
127 \* <p>  
128 \* NOTE: As from v3.2 this method calls  
129 \* {@link #createEntry(HashEntry, int, Object, Object)} to create  
130 \* the map entry object.  
131 \*/  
132 @Override  
133 protected void init() {  
134 header = createEntry(null, -1, null, null);  
135 header.before = header.after = header;  
136 }  
137  
138 //-----------------------------------------------------------------------  
139 /\*\*  
140 \* Checks whether the map contains the specified value.  
141 \*  
142 \* @param value the value to search for  
143 \* @return true if the map contains the value  
144 \*/  
145 @Override  
146 public boolean containsValue(final Object value) {  
147 // override uses faster iterator  
148 if (value == null) {  
149 for (LinkEntry<K, V> entry = header.after; entry != header; entry = entry.after) {  
150 if (entry.getValue() == null) {  
151 return true;  
152 }  
153 }  
154 } else {  
155 for (LinkEntry<K, V> entry = header.after; entry != header; entry = entry.after) {  
156 if (isEqualValue(value, entry.getValue())) {  
157 return true;  
158 }  
159 }  
160 }  
161 return false;  
162 }  
163  
164 /\*\*  
165 \* Clears the map, resetting the size to zero and nullifying references  
166 \* to avoid garbage collection issues.  
167 \*/  
168 @Override  
169 public void clear() {  
170 // override to reset the linked list  
171 super.clear();  
172 header.before = header.after = header;  
173 }  
174  
175 //-----------------------------------------------------------------------  
176 /\*\*  
177 \* Gets the first key in the map, which is the first inserted.  
178 \*  
179 \* @return the eldest key  
180 \*/  
181 @Override  
182 public K firstKey() {  
183 if (size == 0) {  
184 throw new NoSuchElementException("Map is empty");  
185 }  
186 return header.after.getKey();  
187 }  
188  
189 /\*\*  
190 \* Gets the last key in the map, which is the most recently inserted.  
191 \*  
192 \* @return the most recently inserted key  
193 \*/  
194 @Override  
195 public K lastKey() {  
196 if (size == 0) {  
197 throw new NoSuchElementException("Map is empty");  
198 }  
199 return header.before.getKey();  
200 }  
201  
202 /\*\*  
203 \* Gets the next key in sequence.  
204 \*  
205 \* @param key the key to get after  
206 \* @return the next key  
207 \*/  
208 @Override  
209 public K nextKey(final Object key) {  
210 final LinkEntry<K, V> entry = getEntry(key);  
211 return entry == null || entry.after == header ? null : entry.after.getKey();  
212 }  
213  
214 @Override  
215 protected LinkEntry<K, V> getEntry(final Object key) {  
216 return (LinkEntry<K, V>) super.getEntry(key);  
217 }  
218  
219 /\*\*  
220 \* Gets the previous key in sequence.  
221 \*  
222 \* @param key the key to get before  
223 \* @return the previous key  
224 \*/  
225 @Override  
226 public K previousKey(final Object key) {  
227 final LinkEntry<K, V> entry = getEntry(key);  
228 return entry == null || entry.before == header ? null : entry.before.getKey();  
229 }  
230  
231 //-----------------------------------------------------------------------  
232 /\*\*  
233 \* Gets the key at the specified index.  
234 \*  
235 \* @param index the index to retrieve  
236 \* @return the key at the specified index  
237 \* @throws IndexOutOfBoundsException if the index is invalid  
238 \*/  
239 protected LinkEntry<K, V> getEntry(final int index) {  
240 if (index < 0) {  
241 throw new IndexOutOfBoundsException("Index " + index + " is less than zero");  
242 }  
243 if (index >= size) {  
244 throw new IndexOutOfBoundsException("Index " + index + " is invalid for size " + size);  
245 }  
246 LinkEntry<K, V> entry;  
247 if (index < size / 2) {  
248 // Search forwards  
249 entry = header.after;  
250 for (int currentIndex = 0; currentIndex < index; currentIndex++) {  
251 entry = entry.after;  
252 }  
253 } else {  
254 // Search backwards  
255 entry = header;  
256 for (int currentIndex = size; currentIndex > index; currentIndex--) {  
257 entry = entry.before;  
258 }  
259 }  
260 return entry;  
261 }  
262  
263 /\*\*  
264 \* Adds an entry into this map, maintaining insertion order.  
265 \* <p>  
266 \* This implementation adds the entry to the data storage table and  
267 \* to the end of the linked list.  
268 \*  
269 \* @param entry the entry to add  
270 \* @param hashIndex the index into the data array to store at  
271 \*/  
272 @Override  
273 protected void addEntry(final HashEntry<K, V> entry, final int hashIndex) {  
274 final LinkEntry<K, V> link = (LinkEntry<K, V>) entry;  
275 link.after = header;  
276 link.before = header.before;  
277 header.before.after = link;  
278 header.before = link;  
279 data[hashIndex] = link;  
280 }  
281  
282 /\*\*  
283 \* Creates an entry to store the data.  
284 \* <p>  
285 \* This implementation creates a new LinkEntry instance.  
286 \*  
287 \* @param next the next entry in sequence  
288 \* @param hashCode the hash code to use  
289 \* @param key the key to store  
290 \* @param value the value to store  
291 \* @return the newly created entry  
292 \*/  
293 @Override  
294 protected LinkEntry<K, V> createEntry(final HashEntry<K, V> next, final int hashCode, final K key, final V value) {  
295 return new LinkEntry<>(next, hashCode, convertKey(key), value);  
296 }  
297  
298 /\*\*  
299 \* Removes an entry from the map and the linked list.  
300 \* <p>  
301 \* This implementation removes the entry from the linked list chain, then  
302 \* calls the superclass implementation.  
303 \*  
304 \* @param entry the entry to remove  
305 \* @param hashIndex the index into the data structure  
306 \* @param previous the previous entry in the chain  
307 \*/  
308 @Override  
309 protected void removeEntry(final HashEntry<K, V> entry, final int hashIndex, final HashEntry<K, V> previous) {  
310 final LinkEntry<K, V> link = (LinkEntry<K, V>) entry;  
311 link.before.after = link.after;  
312 link.after.before = link.before;  
313 link.after = null;  
314 link.before = null;  
315 super.removeEntry(entry, hashIndex, previous);  
316 }  
317  
318 //-----------------------------------------------------------------------  
319 /\*\*  
320 \* Gets the <code>before</code> field from a <code>LinkEntry</code>.  
321 \* Used in subclasses that have no visibility of the field.  
322 \*  
323 \* @param entry the entry to query, must not be null  
324 \* @return the <code>before</code> field of the entry  
325 \* @throws NullPointerException if the entry is null  
326 \* @since 3.1  
327 \*/  
328 protected LinkEntry<K, V> entryBefore(final LinkEntry<K, V> entry) {  
329 return entry.before;  
330 }  
331  
332 /\*\*  
333 \* Gets the <code>after</code> field from a <code>LinkEntry</code>.  
334 \* Used in subclasses that have no visibility of the field.  
335 \*  
336 \* @param entry the entry to query, must not be null  
337 \* @return the <code>after</code> field of the entry  
338 \* @throws NullPointerException if the entry is null  
339 \* @since 3.1  
340 \*/  
341 protected LinkEntry<K, V> entryAfter(final LinkEntry<K, V> entry) {  
342 return entry.after;  
343 }  
344  
345 //-----------------------------------------------------------------------  
346 /\*\*  
347 \* {@inheritDoc}  
348 \*/  
349 @Override  
350 public OrderedMapIterator<K, V> mapIterator() {  
351 if (size == 0) {  
352 return EmptyOrderedMapIterator.<K, V>emptyOrderedMapIterator();  
353 }  
354 return new LinkMapIterator<>(this);  
355 }  
356  
357 /\*\*  
358 \* MapIterator implementation.  
359 \*/  
360 protected static class LinkMapIterator<K, V> extends LinkIterator<K, V> implements  
361 OrderedMapIterator<K, V>, ResettableIterator<K> {  
362  
363 protected LinkMapIterator(final AbstractLinkedMap<K, V> parent) {  
364 super(parent);  
365 }  
366  
367 @Override  
368 public K next() {  
369 return super.nextEntry().getKey();  
370 }  
371  
372 @Override  
373 public K previous() {  
374 return super.previousEntry().getKey();  
375 }  
376  
377 @Override  
378 public K getKey() {  
379 final LinkEntry<K, V> current = currentEntry();  
380 if (current == null) {  
381 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
382 }  
383 return current.getKey();  
384 }  
385  
386 @Override  
387 public V getValue() {  
388 final LinkEntry<K, V> current = currentEntry();  
389 if (current == null) {  
390 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
391 }  
392 return current.getValue();  
393 }  
394  
395 @Override  
396 public V setValue(final V value) {  
397 final LinkEntry<K, V> current = currentEntry();  
398 if (current == null) {  
399 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
400 }  
401 return current.setValue(value);  
402 }  
403 }  
404  
405 //-----------------------------------------------------------------------  
406 /\*\*  
407 \* Creates an entry set iterator.  
408 \* Subclasses can override this to return iterators with different properties.  
409 \*  
410 \* @return the entrySet iterator  
411 \*/  
412 @Override  
413 protected Iterator<Map.Entry<K, V>> createEntrySetIterator() {  
414 if (size() == 0) {  
415 return EmptyOrderedIterator.<Map.Entry<K, V>>emptyOrderedIterator();  
416 }  
417 return new EntrySetIterator<>(this);  
418 }  
419  
420 /\*\*  
421 \* EntrySet iterator.  
422 \*/  
423 protected static class EntrySetIterator<K, V> extends LinkIterator<K, V> implements  
424 OrderedIterator<Map.Entry<K, V>>, ResettableIterator<Map.Entry<K, V>> {  
425  
426 protected EntrySetIterator(final AbstractLinkedMap<K, V> parent) {  
427 super(parent);  
428 }  
429  
430 @Override  
431 public Map.Entry<K, V> next() {  
432 return super.nextEntry();  
433 }  
434  
435 @Override  
436 public Map.Entry<K, V> previous() {  
437 return super.previousEntry();  
438 }  
439 }  
440  
441 //-----------------------------------------------------------------------  
442 /\*\*  
443 \* Creates a key set iterator.  
444 \* Subclasses can override this to return iterators with different properties.  
445 \*  
446 \* @return the keySet iterator  
447 \*/  
448 @Override  
449 protected Iterator<K> createKeySetIterator() {  
450 if (size() == 0) {  
451 return EmptyOrderedIterator.<K>emptyOrderedIterator();  
452 }  
453 return new KeySetIterator<>(this);  
454 }  
455  
456 /\*\*  
457 \* KeySet iterator.  
458 \*/  
459 protected static class KeySetIterator<K> extends LinkIterator<K, Object> implements  
460 OrderedIterator<K>, ResettableIterator<K> {  
461  
462 @SuppressWarnings("unchecked")  
463 protected KeySetIterator(final AbstractLinkedMap<K, ?> parent) {  
464 super((AbstractLinkedMap<K, Object>) parent);  
465 }  
466  
467 @Override  
468 public K next() {  
469 return super.nextEntry().getKey();  
470 }  
471  
472 @Override  
473 public K previous() {  
474 return super.previousEntry().getKey();  
475 }  
476 }  
477  
478 //-----------------------------------------------------------------------  
479 /\*\*  
480 \* Creates a values iterator.  
481 \* Subclasses can override this to return iterators with different properties.  
482 \*  
483 \* @return the values iterator  
484 \*/  
485 @Override  
486 protected Iterator<V> createValuesIterator() {  
487 if (size() == 0) {  
488 return EmptyOrderedIterator.<V>emptyOrderedIterator();  
489 }  
490 return new ValuesIterator<>(this);  
491 }  
492  
493 /\*\*  
494 \* Values iterator.  
495 \*/  
496 protected static class ValuesIterator<V> extends LinkIterator<Object, V> implements  
497 OrderedIterator<V>, ResettableIterator<V> {  
498  
499 @SuppressWarnings("unchecked")  
500 protected ValuesIterator(final AbstractLinkedMap<?, V> parent) {  
501 super((AbstractLinkedMap<Object, V>) parent);  
502 }  
503  
504 @Override  
505 public V next() {  
506 return super.nextEntry().getValue();  
507 }  
508  
509 @Override  
510 public V previous() {  
511 return super.previousEntry().getValue();  
512 }  
513 }  
514  
515 //-----------------------------------------------------------------------  
516 /\*\*  
517 \* LinkEntry that stores the data.  
518 \* <p>  
519 \* If you subclass <code>AbstractLinkedMap</code> but not <code>LinkEntry</code>  
520 \* then you will not be able to access the protected fields.  
521 \* The <code>entryXxx()</code> methods on <code>AbstractLinkedMap</code> exist  
522 \* to provide the necessary access.  
523 \*/  
524 protected static class LinkEntry<K, V> extends HashEntry<K, V> {  
525 /\*\* The entry before this one in the order \*/  
526 protected LinkEntry<K, V> before;  
527 /\*\* The entry after this one in the order \*/  
528 protected LinkEntry<K, V> after;  
529  
530 /\*\*  
531 \* Constructs a new entry.  
532 \*  
533 \* @param next the next entry in the hash bucket sequence  
534 \* @param hashCode the hash code  
535 \* @param key the key  
536 \* @param value the value  
537 \*/  
538 protected LinkEntry(final HashEntry<K, V> next, final int hashCode, final Object key, final V value) {  
539 super(next, hashCode, key, value);  
540 }  
541 }  
542  
543 /\*\*  
544 \* Base Iterator that iterates in link order.  
545 \*/  
546 protected static abstract class LinkIterator<K, V> {  
547  
548 /\*\* The parent map \*/  
549 protected final AbstractLinkedMap<K, V> parent;  
550 /\*\* The current (last returned) entry \*/  
551 protected LinkEntry<K, V> last;  
552 /\*\* The next entry \*/  
553 protected LinkEntry<K, V> next;  
554 /\*\* The modification count expected \*/  
555 protected int expectedModCount;  
556  
557 protected LinkIterator(final AbstractLinkedMap<K, V> parent) {  
558 super();  
559 this.parent = parent;  
560 this.next = parent.header.after;  
561 this.expectedModCount = parent.modCount;  
562 }  
563  
564 public boolean hasNext() {  
565 return next != parent.header;  
566 }  
567  
568 public boolean hasPrevious() {  
569 return next.before != parent.header;  
570 }  
571  
572 protected LinkEntry<K, V> nextEntry() {  
573 if (parent.modCount != expectedModCount) {  
574 throw new ConcurrentModificationException();  
575 }  
576 if (next == parent.header) {  
577 throw new NoSuchElementException(AbstractHashedMap.NO\_NEXT\_ENTRY);  
578 }  
579 last = next;  
580 next = next.after;  
581 return last;  
582 }  
583  
584 protected LinkEntry<K, V> previousEntry() {  
585 if (parent.modCount != expectedModCount) {  
586 throw new ConcurrentModificationException();  
587 }  
588 final LinkEntry<K, V> previous = next.before;  
589 if (previous == parent.header) {  
590 throw new NoSuchElementException(AbstractHashedMap.NO\_PREVIOUS\_ENTRY);  
591 }  
592 next = previous;  
593 last = previous;  
594 return last;  
595 }  
596  
597 protected LinkEntry<K, V> currentEntry() {  
598 return last;  
599 }  
600  
601 public void remove() {  
602 if (last == null) {  
603 throw new IllegalStateException(AbstractHashedMap.REMOVE\_INVALID);  
604 }  
605 if (parent.modCount != expectedModCount) {  
606 throw new ConcurrentModificationException();  
607 }  
608 parent.remove(last.getKey());  
609 last = null;  
610 expectedModCount = parent.modCount;  
611 }  
612  
613 public void reset() {  
614 last = null;  
615 next = parent.header.after;  
616 }  
617  
618 @Override  
619 public String toString() {  
620 if (last != null) {  
621 return "Iterator[" + last.getKey() + "=" + last.getValue() + "]";  
622 }  
623 return "Iterator[]";  
624 }  
625 }  
626  
627}