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.AbstractCollection;  
024import java.util.AbstractSet;  
025import java.util.Collection;  
026import java.util.Iterator;  
027import java.util.Map;  
028import java.util.NoSuchElementException;  
029import java.util.Set;  
030  
031import org.apache.commons.collections4.IterableMap;  
032import org.apache.commons.collections4.MapIterator;  
033import org.apache.commons.collections4.ResettableIterator;  
034import org.apache.commons.collections4.iterators.EmptyIterator;  
035import org.apache.commons.collections4.iterators.EmptyMapIterator;  
036  
037/\*\*  
038 \* A <code>Map</code> implementation that stores data in simple fields until  
039 \* the size is greater than 3.  
040 \* <p>  
041 \* This map is designed for performance and can outstrip HashMap.  
042 \* It also has good garbage collection characteristics.  
043 \* </p>  
044 \* <ul>  
045 \* <li>Optimised for operation at size 3 or less.  
046 \* <li>Still works well once size 3 exceeded.  
047 \* <li>Gets at size 3 or less are about 0-10% faster than HashMap,  
048 \* <li>Puts at size 3 or less are over 4 times faster than HashMap.  
049 \* <li>Performance 5% slower than HashMap once size 3 exceeded once.  
050 \* </ul>  
051 \* <p>  
052 \* The design uses two distinct modes of operation - flat and delegate.  
053 \* While the map is size 3 or less, operations map straight onto fields using  
054 \* switch statements. Once size 4 is reached, the map switches to delegate mode  
055 \* and only switches back when cleared. In delegate mode, all operations are  
056 \* forwarded straight to a HashMap resulting in the 5% performance loss.  
057 \* </p>  
058 \* <p>  
059 \* The performance gains on puts are due to not needing to create a Map Entry  
060 \* object. This is a large saving not only in performance but in garbage collection.  
061 \* </p>  
062 \* <p>  
063 \* Whilst in flat mode this map is also easy for the garbage collector to dispatch.  
064 \* This is because it contains no complex objects or arrays which slow the progress.  
065 \* </p>  
066 \* <p>  
067 \* Do not use <code>Flat3Map</code> if the size is likely to grow beyond 3.  
068 \* </p>  
069 \* <p>  
070 \* <strong>Note that Flat3Map is not synchronized and is not thread-safe.</strong>  
071 \* If you wish to use this map from multiple threads concurrently, you must use  
072 \* appropriate synchronization. The simplest approach is to wrap this map  
073 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
074 \* exceptions when accessed by concurrent threads without synchronization.  
075 \* </p>  
076 \*  
077 \* @param <K> the type of the keys in this map  
078 \* @param <V> the type of the values in this map  
079 \* @since 3.0  
080 \*/  
081public class Flat3Map<K, V> implements IterableMap<K, V>, Serializable, Cloneable {  
082  
083 /\*\* Serialization version \*/  
084 private static final long serialVersionUID = -6701087419741928296L;  
085  
086 /\*\* The size of the map, used while in flat mode \*/  
087 private transient int size;  
088 /\*\* Hash, used while in flat mode \*/  
089 private transient int hash1;  
090 /\*\* Hash, used while in flat mode \*/  
091 private transient int hash2;  
092 /\*\* Hash, used while in flat mode \*/  
093 private transient int hash3;  
094 /\*\* Key, used while in flat mode \*/  
095 private transient K key1;  
096 /\*\* Key, used while in flat mode \*/  
097 private transient K key2;  
098 /\*\* Key, used while in flat mode \*/  
099 private transient K key3;  
100 /\*\* Value, used while in flat mode \*/  
101 private transient V value1;  
102 /\*\* Value, used while in flat mode \*/  
103 private transient V value2;  
104 /\*\* Value, used while in flat mode \*/  
105 private transient V value3;  
106 /\*\* Map, used while in delegate mode \*/  
107 private transient AbstractHashedMap<K, V> delegateMap;  
108  
109 /\*\*  
110 \* Constructor.  
111 \*/  
112 public Flat3Map() {  
113 super();  
114 }  
115  
116 /\*\*  
117 \* Constructor copying elements from another map.  
118 \*  
119 \* @param map the map to copy  
120 \* @throws NullPointerException if the map is null  
121 \*/  
122 public Flat3Map(final Map<? extends K, ? extends V> map) {  
123 super();  
124 putAll(map);  
125 }  
126  
127 //-----------------------------------------------------------------------  
128 /\*\*  
129 \* Gets the value mapped to the key specified.  
130 \*  
131 \* @param key the key  
132 \* @return the mapped value, null if no match  
133 \*/  
134 @Override  
135 public V get(final Object key) {  
136 if (delegateMap != null) {  
137 return delegateMap.get(key);  
138 }  
139 if (key == null) {  
140 switch (size) {  
141 // drop through  
142 case 3:  
143 if (key3 == null) {  
144 return value3;  
145 }  
146 case 2:  
147 if (key2 == null) {  
148 return value2;  
149 }  
150 case 1:  
151 if (key1 == null) {  
152 return value1;  
153 }  
154 }  
155 } else {  
156 if (size > 0) {  
157 final int hashCode = key.hashCode();  
158 switch (size) {  
159 // drop through  
160 case 3:  
161 if (hash3 == hashCode && key.equals(key3)) {  
162 return value3;  
163 }  
164 case 2:  
165 if (hash2 == hashCode && key.equals(key2)) {  
166 return value2;  
167 }  
168 case 1:  
169 if (hash1 == hashCode && key.equals(key1)) {  
170 return value1;  
171 }  
172 }  
173 }  
174 }  
175 return null;  
176 }  
177  
178 /\*\*  
179 \* Gets the size of the map.  
180 \*  
181 \* @return the size  
182 \*/  
183 @Override  
184 public int size() {  
185 if (delegateMap != null) {  
186 return delegateMap.size();  
187 }  
188 return size;  
189 }  
190  
191 /\*\*  
192 \* Checks whether the map is currently empty.  
193 \*  
194 \* @return true if the map is currently size zero  
195 \*/  
196 @Override  
197 public boolean isEmpty() {  
198 return size() == 0;  
199 }  
200  
201 //-----------------------------------------------------------------------  
202 /\*\*  
203 \* Checks whether the map contains the specified key.  
204 \*  
205 \* @param key the key to search for  
206 \* @return true if the map contains the key  
207 \*/  
208 @Override  
209 public boolean containsKey(final Object key) {  
210 if (delegateMap != null) {  
211 return delegateMap.containsKey(key);  
212 }  
213 if (key == null) {  
214 switch (size) { // drop through  
215 case 3:  
216 if (key3 == null) {  
217 return true;  
218 }  
219 case 2:  
220 if (key2 == null) {  
221 return true;  
222 }  
223 case 1:  
224 if (key1 == null) {  
225 return true;  
226 }  
227 }  
228 } else {  
229 if (size > 0) {  
230 final int hashCode = key.hashCode();  
231 switch (size) { // drop through  
232 case 3:  
233 if (hash3 == hashCode && key.equals(key3)) {  
234 return true;  
235 }  
236 case 2:  
237 if (hash2 == hashCode && key.equals(key2)) {  
238 return true;  
239 }  
240 case 1:  
241 if (hash1 == hashCode && key.equals(key1)) {  
242 return true;  
243 }  
244 }  
245 }  
246 }  
247 return false;  
248 }  
249  
250 /\*\*  
251 \* Checks whether the map contains the specified value.  
252 \*  
253 \* @param value the value to search for  
254 \* @return true if the map contains the key  
255 \*/  
256 @Override  
257 public boolean containsValue(final Object value) {  
258 if (delegateMap != null) {  
259 return delegateMap.containsValue(value);  
260 }  
261 if (value == null) { // drop through  
262 switch (size) {  
263 case 3:  
264 if (value3 == null) {  
265 return true;  
266 }  
267 case 2:  
268 if (value2 == null) {  
269 return true;  
270 }  
271 case 1:  
272 if (value1 == null) {  
273 return true;  
274 }  
275 }  
276 } else {  
277 switch (size) { // drop through  
278 case 3:  
279 if (value.equals(value3)) {  
280 return true;  
281 }  
282 case 2:  
283 if (value.equals(value2)) {  
284 return true;  
285 }  
286 case 1:  
287 if (value.equals(value1)) {  
288 return true;  
289 }  
290 }  
291 }  
292 return false;  
293 }  
294  
295 //-----------------------------------------------------------------------  
296 /\*\*  
297 \* Puts a key-value mapping into this map.  
298 \*  
299 \* @param key the key to add  
300 \* @param value the value to add  
301 \* @return the value previously mapped to this key, null if none  
302 \*/  
303 @Override  
304 public V put(final K key, final V value) {  
305 if (delegateMap != null) {  
306 return delegateMap.put(key, value);  
307 }  
308 // change existing mapping  
309 if (key == null) {  
310 switch (size) { // drop through  
311 case 3:  
312 if (key3 == null) {  
313 final V old = value3;  
314 value3 = value;  
315 return old;  
316 }  
317 case 2:  
318 if (key2 == null) {  
319 final V old = value2;  
320 value2 = value;  
321 return old;  
322 }  
323 case 1:  
324 if (key1 == null) {  
325 final V old = value1;  
326 value1 = value;  
327 return old;  
328 }  
329 }  
330 } else {  
331 if (size > 0) {  
332 final int hashCode = key.hashCode();  
333 switch (size) { // drop through  
334 case 3:  
335 if (hash3 == hashCode && key.equals(key3)) {  
336 final V old = value3;  
337 value3 = value;  
338 return old;  
339 }  
340 case 2:  
341 if (hash2 == hashCode && key.equals(key2)) {  
342 final V old = value2;  
343 value2 = value;  
344 return old;  
345 }  
346 case 1:  
347 if (hash1 == hashCode && key.equals(key1)) {  
348 final V old = value1;  
349 value1 = value;  
350 return old;  
351 }  
352 }  
353 }  
354 }  
355  
356 // add new mapping  
357 switch (size) {  
358 default:  
359 convertToMap();  
360 delegateMap.put(key, value);  
361 return null;  
362 case 2:  
363 hash3 = key == null ? 0 : key.hashCode();  
364 key3 = key;  
365 value3 = value;  
366 break;  
367 case 1:  
368 hash2 = key == null ? 0 : key.hashCode();  
369 key2 = key;  
370 value2 = value;  
371 break;  
372 case 0:  
373 hash1 = key == null ? 0 : key.hashCode();  
374 key1 = key;  
375 value1 = value;  
376 break;  
377 }  
378 size++;  
379 return null;  
380 }  
381  
382 /\*\*  
383 \* Puts all the values from the specified map into this map.  
384 \*  
385 \* @param map the map to add  
386 \* @throws NullPointerException if the map is null  
387 \*/  
388 @Override  
389 public void putAll(final Map<? extends K, ? extends V> map) {  
390 final int size = map.size();  
391 if (size == 0) {  
392 return;  
393 }  
394 if (delegateMap != null) {  
395 delegateMap.putAll(map);  
396 return;  
397 }  
398 if (size < 4) {  
399 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
400 put(entry.getKey(), entry.getValue());  
401 }  
402 } else {  
403 convertToMap();  
404 delegateMap.putAll(map);  
405 }  
406 }  
407  
408 /\*\*  
409 \* Converts the flat map data to a map.  
410 \*/  
411 private void convertToMap() {  
412 delegateMap = createDelegateMap();  
413 switch (size) { // drop through  
414 case 3:  
415 delegateMap.put(key3, value3);  
416 case 2:  
417 delegateMap.put(key2, value2);  
418 case 1:  
419 delegateMap.put(key1, value1);  
420 case 0:  
421 break;  
422 default:  
423 throw new IllegalStateException("Invalid map index: " + size);  
424 }  
425  
426 size = 0;  
427 hash1 = hash2 = hash3 = 0;  
428 key1 = key2 = key3 = null;  
429 value1 = value2 = value3 = null;  
430 }  
431  
432 /\*\*  
433 \* Create an instance of the map used for storage when in delegation mode.  
434 \* <p>  
435 \* This can be overridden by subclasses to provide a different map implementation.  
436 \* Not every AbstractHashedMap is suitable, identity and reference based maps  
437 \* would be poor choices.  
438 \*  
439 \* @return a new AbstractHashedMap or subclass  
440 \* @since 3.1  
441 \*/  
442 protected AbstractHashedMap<K, V> createDelegateMap() {  
443 return new HashedMap<>();  
444 }  
445  
446 /\*\*  
447 \* Removes the specified mapping from this map.  
448 \*  
449 \* @param key the mapping to remove  
450 \* @return the value mapped to the removed key, null if key not in map  
451 \*/  
452 @Override  
453 public V remove(final Object key) {  
454 if (delegateMap != null) {  
455 return delegateMap.remove(key);  
456 }  
457 if (size == 0) {  
458 return null;  
459 }  
460 if (key == null) {  
461 switch (size) { // drop through  
462 case 3:  
463 if (key3 == null) {  
464 final V old = value3;  
465 hash3 = 0;  
466 key3 = null;  
467 value3 = null;  
468 size = 2;  
469 return old;  
470 }  
471 if (key2 == null) {  
472 final V old = value2;  
473 hash2 = hash3;  
474 key2 = key3;  
475 value2 = value3;  
476 hash3 = 0;  
477 key3 = null;  
478 value3 = null;  
479 size = 2;  
480 return old;  
481 }  
482 if (key1 == null) {  
483 final V old = value1;  
484 hash1 = hash3;  
485 key1 = key3;  
486 value1 = value3;  
487 hash3 = 0;  
488 key3 = null;  
489 value3 = null;  
490 size = 2;  
491 return old;  
492 }  
493 return null;  
494 case 2:  
495 if (key2 == null) {  
496 final V old = value2;  
497 hash2 = 0;  
498 key2 = null;  
499 value2 = null;  
500 size = 1;  
501 return old;  
502 }  
503 if (key1 == null) {  
504 final V old = value1;  
505 hash1 = hash2;  
506 key1 = key2;  
507 value1 = value2;  
508 hash2 = 0;  
509 key2 = null;  
510 value2 = null;  
511 size = 1;  
512 return old;  
513 }  
514 return null;  
515 case 1:  
516 if (key1 == null) {  
517 final V old = value1;  
518 hash1 = 0;  
519 key1 = null;  
520 value1 = null;  
521 size = 0;  
522 return old;  
523 }  
524 }  
525 } else {  
526 if (size > 0) {  
527 final int hashCode = key.hashCode();  
528 switch (size) { // drop through  
529 case 3:  
530 if (hash3 == hashCode && key.equals(key3)) {  
531 final V old = value3;  
532 hash3 = 0;  
533 key3 = null;  
534 value3 = null;  
535 size = 2;  
536 return old;  
537 }  
538 if (hash2 == hashCode && key.equals(key2)) {  
539 final V old = value2;  
540 hash2 = hash3;  
541 key2 = key3;  
542 value2 = value3;  
543 hash3 = 0;  
544 key3 = null;  
545 value3 = null;  
546 size = 2;  
547 return old;  
548 }  
549 if (hash1 == hashCode && key.equals(key1)) {  
550 final V old = value1;  
551 hash1 = hash3;  
552 key1 = key3;  
553 value1 = value3;  
554 hash3 = 0;  
555 key3 = null;  
556 value3 = null;  
557 size = 2;  
558 return old;  
559 }  
560 return null;  
561 case 2:  
562 if (hash2 == hashCode && key.equals(key2)) {  
563 final V old = value2;  
564 hash2 = 0;  
565 key2 = null;  
566 value2 = null;  
567 size = 1;  
568 return old;  
569 }  
570 if (hash1 == hashCode && key.equals(key1)) {  
571 final V old = value1;  
572 hash1 = hash2;  
573 key1 = key2;  
574 value1 = value2;  
575 hash2 = 0;  
576 key2 = null;  
577 value2 = null;  
578 size = 1;  
579 return old;  
580 }  
581 return null;  
582 case 1:  
583 if (hash1 == hashCode && key.equals(key1)) {  
584 final V old = value1;  
585 hash1 = 0;  
586 key1 = null;  
587 value1 = null;  
588 size = 0;  
589 return old;  
590 }  
591 }  
592 }  
593 }  
594 return null;  
595 }  
596  
597 /\*\*  
598 \* Clears the map, resetting the size to zero and nullifying references  
599 \* to avoid garbage collection issues.  
600 \*/  
601 @Override  
602 public void clear() {  
603 if (delegateMap != null) {  
604 delegateMap.clear(); // should aid gc  
605 delegateMap = null; // switch back to flat mode  
606 } else {  
607 size = 0;  
608 hash1 = hash2 = hash3 = 0;  
609 key1 = key2 = key3 = null;  
610 value1 = value2 = value3 = null;  
611 }  
612 }  
613  
614 //-----------------------------------------------------------------------  
615 /\*\*  
616 \* Gets an iterator over the map.  
617 \* Changes made to the iterator affect this map.  
618 \* <p>  
619 \* A MapIterator returns the keys in the map. It also provides convenient  
620 \* methods to get the key and value, and set the value.  
621 \* It avoids the need to create an entrySet/keySet/values object.  
622 \* It also avoids creating the Map Entry object.  
623 \*  
624 \* @return the map iterator  
625 \*/  
626 @Override  
627 public MapIterator<K, V> mapIterator() {  
628 if (delegateMap != null) {  
629 return delegateMap.mapIterator();  
630 }  
631 if (size == 0) {  
632 return EmptyMapIterator.<K, V>emptyMapIterator();  
633 }  
634 return new FlatMapIterator<>(this);  
635 }  
636  
637 /\*\*  
638 \* FlatMapIterator  
639 \*/  
640 static class FlatMapIterator<K, V> implements MapIterator<K, V>, ResettableIterator<K> {  
641 private final Flat3Map<K, V> parent;  
642 private int nextIndex = 0;  
643 private boolean canRemove = false;  
644  
645 FlatMapIterator(final Flat3Map<K, V> parent) {  
646 super();  
647 this.parent = parent;  
648 }  
649  
650 @Override  
651 public boolean hasNext() {  
652 return nextIndex < parent.size;  
653 }  
654  
655 @Override  
656 public K next() {  
657 if (hasNext() == false) {  
658 throw new NoSuchElementException(AbstractHashedMap.NO\_NEXT\_ENTRY);  
659 }  
660 canRemove = true;  
661 nextIndex++;  
662 return getKey();  
663 }  
664  
665 @Override  
666 public void remove() {  
667 if (canRemove == false) {  
668 throw new IllegalStateException(AbstractHashedMap.REMOVE\_INVALID);  
669 }  
670 parent.remove(getKey());  
671 nextIndex--;  
672 canRemove = false;  
673 }  
674  
675 @Override  
676 public K getKey() {  
677 if (canRemove == false) {  
678 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
679 }  
680 switch (nextIndex) {  
681 case 3:  
682 return parent.key3;  
683 case 2:  
684 return parent.key2;  
685 case 1:  
686 return parent.key1;  
687 }  
688 throw new IllegalStateException("Invalid map index: " + nextIndex);  
689 }  
690  
691 @Override  
692 public V getValue() {  
693 if (canRemove == false) {  
694 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
695 }  
696 switch (nextIndex) {  
697 case 3:  
698 return parent.value3;  
699 case 2:  
700 return parent.value2;  
701 case 1:  
702 return parent.value1;  
703 }  
704 throw new IllegalStateException("Invalid map index: " + nextIndex);  
705 }  
706  
707 @Override  
708 public V setValue(final V value) {  
709 if (canRemove == false) {  
710 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
711 }  
712 final V old = getValue();  
713 switch (nextIndex) {  
714 case 3:  
715 parent.value3 = value;  
716 break;  
717 case 2:  
718 parent.value2 = value;  
719 break;  
720 case 1:  
721 parent.value1 = value;  
722 break;  
723 default:  
724 throw new IllegalStateException("Invalid map index: " + nextIndex);  
725 }  
726 return old;  
727 }  
728  
729 @Override  
730 public void reset() {  
731 nextIndex = 0;  
732 canRemove = false;  
733 }  
734  
735 @Override  
736 public String toString() {  
737 if (canRemove) {  
738 return "Iterator[" + getKey() + "=" + getValue() + "]";  
739 }  
740 return "Iterator[]";  
741 }  
742 }  
743  
744 /\*\*  
745 \* Gets the entrySet view of the map.  
746 \* Changes made to the view affect this map.  
747 \* <p>  
748 \* NOTE: from 4.0, the returned Map Entry will be an independent object and will  
749 \* not change anymore as the iterator progresses. To avoid this additional object  
750 \* creation and simply iterate through the entries, use {@link #mapIterator()}.  
751 \*  
752 \* @return the entrySet view  
753 \*/  
754 @Override  
755 public Set<Map.Entry<K, V>> entrySet() {  
756 if (delegateMap != null) {  
757 return delegateMap.entrySet();  
758 }  
759 return new EntrySet<>(this);  
760 }  
761  
762 /\*\*  
763 \* EntrySet  
764 \*/  
765 static class EntrySet<K, V> extends AbstractSet<Map.Entry<K, V>> {  
766 private final Flat3Map<K, V> parent;  
767  
768 EntrySet(final Flat3Map<K, V> parent) {  
769 super();  
770 this.parent = parent;  
771 }  
772  
773 @Override  
774 public int size() {  
775 return parent.size();  
776 }  
777  
778 @Override  
779 public void clear() {  
780 parent.clear();  
781 }  
782  
783 @Override  
784 public boolean remove(final Object obj) {  
785 if (obj instanceof Map.Entry == false) {  
786 return false;  
787 }  
788 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
789 final Object key = entry.getKey();  
790 final boolean result = parent.containsKey(key);  
791 parent.remove(key);  
792 return result;  
793 }  
794  
795 @Override  
796 public Iterator<Map.Entry<K, V>> iterator() {  
797 if (parent.delegateMap != null) {  
798 return parent.delegateMap.entrySet().iterator();  
799 }  
800 if (parent.size() == 0) {  
801 return EmptyIterator.<Map.Entry<K, V>>emptyIterator();  
802 }  
803 return new EntrySetIterator<>(parent);  
804 }  
805 }  
806  
807 static class FlatMapEntry<K, V> implements Map.Entry<K, V> {  
808 private final Flat3Map<K, V> parent;  
809 private final int index;  
810 private volatile boolean removed;  
811  
812 public FlatMapEntry(final Flat3Map<K, V> parent, final int index) {  
813 this.parent = parent;  
814 this.index = index;  
815 this.removed = false;  
816 }  
817  
818 /\*\*  
819 \* Used by the iterator that created this entry to indicate that  
820 \* {@link java.util.Iterator#remove()} has been called.  
821 \* <p>  
822 \* As a consequence, all subsequent call to {@link #getKey()},  
823 \* {@link #setValue(Object)} and {@link #getValue()} will fail.  
824 \*  
825 \* @param flag the new value of the removed flag  
826 \*/  
827 void setRemoved(final boolean flag) {  
828 this.removed = flag;  
829 }  
830  
831 @Override  
832 public K getKey() {  
833 if (removed) {  
834 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
835 }  
836 switch (index) {  
837 case 3:  
838 return parent.key3;  
839 case 2:  
840 return parent.key2;  
841 case 1:  
842 return parent.key1;  
843 }  
844 throw new IllegalStateException("Invalid map index: " + index);  
845 }  
846  
847 @Override  
848 public V getValue() {  
849 if (removed) {  
850 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
851 }  
852 switch (index) {  
853 case 3:  
854 return parent.value3;  
855 case 2:  
856 return parent.value2;  
857 case 1:  
858 return parent.value1;  
859 }  
860 throw new IllegalStateException("Invalid map index: " + index);  
861 }  
862  
863 @Override  
864 public V setValue(final V value) {  
865 if (removed) {  
866 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
867 }  
868 final V old = getValue();  
869 switch (index) {  
870 case 3:  
871 parent.value3 = value;  
872 break;  
873 case 2:  
874 parent.value2 = value;  
875 break;  
876 case 1:  
877 parent.value1 = value;  
878 break;  
879 default:  
880 throw new IllegalStateException("Invalid map index: " + index);  
881 }  
882 return old;  
883 }  
884  
885 @Override  
886 public boolean equals(final Object obj) {  
887 if (removed) {  
888 return false;  
889 }  
890 if (obj instanceof Map.Entry == false) {  
891 return false;  
892 }  
893 final Map.Entry<?, ?> other = (Map.Entry<?, ?>) obj;  
894 final Object key = getKey();  
895 final Object value = getValue();  
896 return (key == null ? other.getKey() == null : key.equals(other.getKey())) &&  
897 (value == null ? other.getValue() == null : value.equals(other.getValue()));  
898 }  
899  
900 @Override  
901 public int hashCode() {  
902 if (removed) {  
903 return 0;  
904 }  
905 final Object key = getKey();  
906 final Object value = getValue();  
907 return (key == null ? 0 : key.hashCode()) ^  
908 (value == null ? 0 : value.hashCode());  
909 }  
910  
911 @Override  
912 public String toString() {  
913 if (!removed) {  
914 return getKey() + "=" + getValue();  
915 }  
916 return "";  
917 }  
918  
919 }  
920  
921 static abstract class EntryIterator<K, V> {  
922 private final Flat3Map<K, V> parent;  
923 private int nextIndex = 0;  
924 private FlatMapEntry<K, V> currentEntry = null;  
925  
926 /\*\*  
927 \* Create a new Flat3Map.EntryIterator.  
928 \*/  
929 public EntryIterator(final Flat3Map<K, V> parent) {  
930 this.parent = parent;  
931 }  
932  
933 public boolean hasNext() {  
934 return nextIndex < parent.size;  
935 }  
936  
937 public Map.Entry<K, V> nextEntry() {  
938 if (!hasNext()) {  
939 throw new NoSuchElementException(AbstractHashedMap.NO\_NEXT\_ENTRY);  
940 }  
941 currentEntry = new FlatMapEntry<>(parent, ++nextIndex);  
942 return currentEntry;  
943 }  
944  
945 public void remove() {  
946 if (currentEntry == null) {  
947 throw new IllegalStateException(AbstractHashedMap.REMOVE\_INVALID);  
948 }  
949 currentEntry.setRemoved(true);  
950 parent.remove(currentEntry.getKey());  
951 nextIndex--;  
952 currentEntry = null;  
953 }  
954  
955 }  
956  
957 /\*\*  
958 \* EntrySetIterator and MapEntry  
959 \*/  
960 static class EntrySetIterator<K, V> extends EntryIterator<K, V> implements Iterator<Map.Entry<K, V>> {  
961 EntrySetIterator(final Flat3Map<K, V> parent) {  
962 super(parent);  
963 }  
964  
965 @Override  
966 public Map.Entry<K, V> next() {  
967 return nextEntry();  
968 }  
969 }  
970  
971 /\*\*  
972 \* Gets the keySet view of the map.  
973 \* Changes made to the view affect this map.  
974 \* To simply iterate through the keys, use {@link #mapIterator()}.  
975 \*  
976 \* @return the keySet view  
977 \*/  
978 @Override  
979 public Set<K> keySet() {  
980 if (delegateMap != null) {  
981 return delegateMap.keySet();  
982 }  
983 return new KeySet<>(this);  
984 }  
985  
986 /\*\*  
987 \* KeySet  
988 \*/  
989 static class KeySet<K> extends AbstractSet<K> {  
990 private final Flat3Map<K, ?> parent;  
991  
992 KeySet(final Flat3Map<K, ?> parent) {  
993 super();  
994 this.parent = parent;  
995 }  
996  
997 @Override  
998 public int size() {  
999 return parent.size();  
1000 }  
1001  
1002 @Override  
1003 public void clear() {  
1004 parent.clear();  
1005 }  
1006  
1007 @Override  
1008 public boolean contains(final Object key) {  
1009 return parent.containsKey(key);  
1010 }  
1011  
1012 @Override  
1013 public boolean remove(final Object key) {  
1014 final boolean result = parent.containsKey(key);  
1015 parent.remove(key);  
1016 return result;  
1017 }  
1018  
1019 @Override  
1020 public Iterator<K> iterator() {  
1021 if (parent.delegateMap != null) {  
1022 return parent.delegateMap.keySet().iterator();  
1023 }  
1024 if (parent.size() == 0) {  
1025 return EmptyIterator.<K>emptyIterator();  
1026 }  
1027 return new KeySetIterator<>(parent);  
1028 }  
1029 }  
1030  
1031 /\*\*  
1032 \* KeySetIterator  
1033 \*/  
1034 static class KeySetIterator<K> extends EntryIterator<K, Object> implements Iterator<K>{  
1035  
1036 @SuppressWarnings("unchecked")  
1037 KeySetIterator(final Flat3Map<K, ?> parent) {  
1038 super((Flat3Map<K, Object>) parent);  
1039 }  
1040  
1041 @Override  
1042 public K next() {  
1043 return nextEntry().getKey();  
1044 }  
1045 }  
1046  
1047 /\*\*  
1048 \* Gets the values view of the map.  
1049 \* Changes made to the view affect this map.  
1050 \* To simply iterate through the values, use {@link #mapIterator()}.  
1051 \*  
1052 \* @return the values view  
1053 \*/  
1054 @Override  
1055 public Collection<V> values() {  
1056 if (delegateMap != null) {  
1057 return delegateMap.values();  
1058 }  
1059 return new Values<>(this);  
1060 }  
1061  
1062 /\*\*  
1063 \* Values  
1064 \*/  
1065 static class Values<V> extends AbstractCollection<V> {  
1066 private final Flat3Map<?, V> parent;  
1067  
1068 Values(final Flat3Map<?, V> parent) {  
1069 super();  
1070 this.parent = parent;  
1071 }  
1072  
1073 @Override  
1074 public int size() {  
1075 return parent.size();  
1076 }  
1077  
1078 @Override  
1079 public void clear() {  
1080 parent.clear();  
1081 }  
1082  
1083 @Override  
1084 public boolean contains(final Object value) {  
1085 return parent.containsValue(value);  
1086 }  
1087  
1088 @Override  
1089 public Iterator<V> iterator() {  
1090 if (parent.delegateMap != null) {  
1091 return parent.delegateMap.values().iterator();  
1092 }  
1093 if (parent.size() == 0) {  
1094 return EmptyIterator.<V>emptyIterator();  
1095 }  
1096 return new ValuesIterator<>(parent);  
1097 }  
1098 }  
1099  
1100 /\*\*  
1101 \* ValuesIterator  
1102 \*/  
1103 static class ValuesIterator<V> extends EntryIterator<Object, V> implements Iterator<V> {  
1104  
1105 @SuppressWarnings("unchecked")  
1106 ValuesIterator(final Flat3Map<?, V> parent) {  
1107 super((Flat3Map<Object, V>) parent);  
1108 }  
1109  
1110 @Override  
1111 public V next() {  
1112 return nextEntry().getValue();  
1113 }  
1114 }  
1115  
1116 //-----------------------------------------------------------------------  
1117 /\*\*  
1118 \* Write the map out using a custom routine.  
1119 \*  
1120 \* @param out the output stream  
1121 \* @throws IOException if an error occurs while writing to the stream  
1122 \*/  
1123 private void writeObject(final ObjectOutputStream out) throws IOException {  
1124 out.defaultWriteObject();  
1125 out.writeInt(size());  
1126 for (final MapIterator<?, ?> it = mapIterator(); it.hasNext();) {  
1127 out.writeObject(it.next()); // key  
1128 out.writeObject(it.getValue()); // value  
1129 }  
1130 }  
1131  
1132 /\*\*  
1133 \* Read the map in using a custom routine.  
1134 \*  
1135 \* @param in the input stream  
1136 \* @throws IOException if an error occurs while reading from the stream  
1137 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
1138 \*/  
1139 @SuppressWarnings("unchecked")  
1140 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
1141 in.defaultReadObject();  
1142 final int count = in.readInt();  
1143 if (count > 3) {  
1144 delegateMap = createDelegateMap();  
1145 }  
1146 for (int i = count; i > 0; i--) {  
1147 put((K) in.readObject(), (V) in.readObject());  
1148 }  
1149 }  
1150  
1151 //-----------------------------------------------------------------------  
1152 /\*\*  
1153 \* Clones the map without cloning the keys or values.  
1154 \*  
1155 \* @return a shallow clone  
1156 \* @since 3.1  
1157 \*/  
1158 @Override  
1159 @SuppressWarnings("unchecked")  
1160 public Flat3Map<K, V> clone() {  
1161 try {  
1162 final Flat3Map<K, V> cloned = (Flat3Map<K, V>) super.clone();  
1163 if (cloned.delegateMap != null) {  
1164 cloned.delegateMap = cloned.delegateMap.clone();  
1165 }  
1166 return cloned;  
1167 } catch (final CloneNotSupportedException ex) {  
1168 throw new InternalError();  
1169 }  
1170 }  
1171  
1172 /\*\*  
1173 \* Compares this map with another.  
1174 \*  
1175 \* @param obj the object to compare to  
1176 \* @return true if equal  
1177 \*/  
1178 @Override  
1179 public boolean equals(final Object obj) {  
1180 if (obj == this) {  
1181 return true;  
1182 }  
1183 if (delegateMap != null) {  
1184 return delegateMap.equals(obj);  
1185 }  
1186 if (obj instanceof Map == false) {  
1187 return false;  
1188 }  
1189 final Map<?, ?> other = (Map<?, ?>) obj;  
1190 if (size != other.size()) {  
1191 return false;  
1192 }  
1193 if (size > 0) {  
1194 Object otherValue = null;  
1195 switch (size) { // drop through  
1196 case 3:  
1197 if (other.containsKey(key3) == false) {  
1198 return false;  
1199 }  
1200 otherValue = other.get(key3);  
1201 if (value3 == null ? otherValue != null : !value3.equals(otherValue)) {  
1202 return false;  
1203 }  
1204 case 2:  
1205 if (other.containsKey(key2) == false) {  
1206 return false;  
1207 }  
1208 otherValue = other.get(key2);  
1209 if (value2 == null ? otherValue != null : !value2.equals(otherValue)) {  
1210 return false;  
1211 }  
1212 case 1:  
1213 if (other.containsKey(key1) == false) {  
1214 return false;  
1215 }  
1216 otherValue = other.get(key1);  
1217 if (value1 == null ? otherValue != null : !value1.equals(otherValue)) {  
1218 return false;  
1219 }  
1220 }  
1221 }  
1222 return true;  
1223 }  
1224  
1225 /\*\*  
1226 \* Gets the standard Map hashCode.  
1227 \*  
1228 \* @return the hash code defined in the Map interface  
1229 \*/  
1230 @Override  
1231 public int hashCode() {  
1232 if (delegateMap != null) {  
1233 return delegateMap.hashCode();  
1234 }  
1235 int total = 0;  
1236 switch (size) { // drop through  
1237 case 3:  
1238 total += hash3 ^ (value3 == null ? 0 : value3.hashCode());  
1239 case 2:  
1240 total += hash2 ^ (value2 == null ? 0 : value2.hashCode());  
1241 case 1:  
1242 total += hash1 ^ (value1 == null ? 0 : value1.hashCode());  
1243 case 0:  
1244 break;  
1245 default:  
1246 throw new IllegalStateException("Invalid map index: " + size);  
1247 }  
1248 return total;  
1249 }  
1250  
1251 /\*\*  
1252 \* Gets the map as a String.  
1253 \*  
1254 \* @return a string version of the map  
1255 \*/  
1256 @Override  
1257 public String toString() {  
1258 if (delegateMap != null) {  
1259 return delegateMap.toString();  
1260 }  
1261 if (size == 0) {  
1262 return "{}";  
1263 }  
1264 final StringBuilder buf = new StringBuilder(128);  
1265 buf.append('{');  
1266 switch (size) { // drop through  
1267 case 3:  
1268 buf.append(key3 == this ? "(this Map)" : key3);  
1269 buf.append('=');  
1270 buf.append(value3 == this ? "(this Map)" : value3);  
1271 buf.append(',');  
1272 case 2:  
1273 buf.append(key2 == this ? "(this Map)" : key2);  
1274 buf.append('=');  
1275 buf.append(value2 == this ? "(this Map)" : value2);  
1276 buf.append(',');  
1277 case 1:  
1278 buf.append(key1 == this ? "(this Map)" : key1);  
1279 buf.append('=');  
1280 buf.append(value1 == this ? "(this Map)" : value1);  
1281 break;  
1282 // case 0: has already been dealt with  
1283 default:  
1284 throw new IllegalStateException("Invalid map index: " + size);  
1285 }  
1286 buf.append('}');  
1287 return buf.toString();  
1288 }  
1289  
1290}