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 static org.apache.commons.collections4.bidimap.TreeBidiMap.DataElement.KEY;  
020import static org.apache.commons.collections4.bidimap.TreeBidiMap.DataElement.VALUE;  
021  
022import java.io.IOException;  
023import java.io.ObjectInputStream;  
024import java.io.ObjectOutputStream;  
025import java.io.Serializable;  
026import java.util.AbstractSet;  
027import java.util.ConcurrentModificationException;  
028import java.util.Iterator;  
029import java.util.Map;  
030import java.util.NoSuchElementException;  
031import java.util.Set;  
032  
033import org.apache.commons.collections4.KeyValue;  
034import org.apache.commons.collections4.MapIterator;  
035import org.apache.commons.collections4.OrderedBidiMap;  
036import org.apache.commons.collections4.OrderedIterator;  
037import org.apache.commons.collections4.OrderedMapIterator;  
038import org.apache.commons.collections4.iterators.EmptyOrderedMapIterator;  
039import org.apache.commons.collections4.keyvalue.UnmodifiableMapEntry;  
040  
041/\*\*  
042 \* Red-Black tree-based implementation of BidiMap where all objects added  
043 \* implement the <code>Comparable</code> interface.  
044 \* <p>  
045 \* This class guarantees that the map will be in both ascending key order  
046 \* and ascending value order, sorted according to the natural order for  
047 \* the key's and value's classes.  
048 \* </p>  
049 \* <p>  
050 \* This Map is intended for applications that need to be able to look  
051 \* up a key-value pairing by either key or value, and need to do so  
052 \* with equal efficiency.  
053 \* </p>  
054 \* <p>  
055 \* While that goal could be accomplished by taking a pair of TreeMaps  
056 \* and redirecting requests to the appropriate TreeMap (e.g.,  
057 \* containsKey would be directed to the TreeMap that maps values to  
058 \* keys, containsValue would be directed to the TreeMap that maps keys  
059 \* to values), there are problems with that implementation.  
060 \* If the data contained in the TreeMaps is large, the cost of redundant  
061 \* storage becomes significant. The {@link DualTreeBidiMap} and  
062 \* {@link DualHashBidiMap} implementations use this approach.  
063 \* </p>  
064 \* <p>  
065 \* This solution keeps minimizes the data storage by holding data only once.  
066 \* The red-black algorithm is based on {@link java.util.TreeMap}, but has been modified  
067 \* to simultaneously map a tree node by key and by value. This doubles the  
068 \* cost of put operations (but so does using two TreeMaps), and nearly doubles  
069 \* the cost of remove operations (there is a savings in that the lookup of the  
070 \* node to be removed only has to be performed once). And since only one node  
071 \* contains the key and value, storage is significantly less than that  
072 \* required by two TreeMaps.  
073 \* </p>  
074 \* <p>  
075 \* The Map.Entry instances returned by the appropriate methods will  
076 \* not allow setValue() and will throw an  
077 \* UnsupportedOperationException on attempts to call that method.  
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 \*  
083 \* @since 3.0 (previously DoubleOrderedMap v2.0)  
084 \*/  
085public class TreeBidiMap<K extends Comparable<K>, V extends Comparable<V>>  
086 implements OrderedBidiMap<K, V>, Serializable {  
087  
088 enum DataElement {  
089 KEY("key"), VALUE("value");  
090  
091 private final String description;  
092  
093 /\*\*  
094 \* Create a new TreeBidiMap.DataElement.  
095 \*  
096 \* @param description the description for the element  
097 \*/  
098 DataElement(final String description) {  
099 this.description = description;  
100 }  
101  
102 @Override  
103 public String toString() {  
104 return description;  
105 }  
106 }  
107  
108 private static final long serialVersionUID = 721969328361807L;  
109  
110 private transient Node<K, V>[] rootNode;  
111 private transient int nodeCount = 0;  
112 private transient int modifications = 0;  
113 private transient Set<K> keySet;  
114 private transient Set<V> valuesSet;  
115 private transient Set<Map.Entry<K, V>> entrySet;  
116 private transient Inverse inverse = null;  
117  
118 //-----------------------------------------------------------------------  
119 /\*\*  
120 \* Constructs a new empty TreeBidiMap.  
121 \*/  
122 @SuppressWarnings("unchecked")  
123 public TreeBidiMap() {  
124 super();  
125 rootNode = new Node[2];  
126 }  
127  
128 /\*\*  
129 \* Constructs a new TreeBidiMap by copying an existing Map.  
130 \*  
131 \* @param map the map to copy  
132 \* @throws ClassCastException if the keys/values in the map are  
133 \* not Comparable or are not mutually comparable  
134 \* @throws NullPointerException if any key or value in the map is null  
135 \*/  
136 public TreeBidiMap(final Map<? extends K, ? extends V> map) {  
137 this();  
138 putAll(map);  
139 }  
140  
141 //-----------------------------------------------------------------------  
142 /\*\*  
143 \* Returns the number of key-value mappings in this map.  
144 \*  
145 \* @return the number of key-value mappings in this map  
146 \*/  
147 @Override  
148 public int size() {  
149 return nodeCount;  
150 }  
151  
152 /\*\*  
153 \* Checks whether the map is empty or not.  
154 \*  
155 \* @return true if the map is empty  
156 \*/  
157 @Override  
158 public boolean isEmpty() {  
159 return nodeCount == 0;  
160 }  
161  
162 /\*\*  
163 \* Checks whether this map contains the a mapping for the specified key.  
164 \* <p>  
165 \* The key must implement <code>Comparable</code>.  
166 \*  
167 \* @param key key whose presence in this map is to be tested  
168 \* @return true if this map contains a mapping for the specified key  
169 \* @throws ClassCastException if the key is of an inappropriate type  
170 \* @throws NullPointerException if the key is null  
171 \*/  
172 @Override  
173 public boolean containsKey(final Object key) {  
174 checkKey(key);  
175 return lookupKey(key) != null;  
176 }  
177  
178 /\*\*  
179 \* Checks whether this map contains the a mapping for the specified value.  
180 \* <p>  
181 \* The value must implement <code>Comparable</code>.  
182 \*  
183 \* @param value value whose presence in this map is to be tested  
184 \* @return true if this map contains a mapping for the specified value  
185 \* @throws ClassCastException if the value is of an inappropriate type  
186 \* @throws NullPointerException if the value is null  
187 \*/  
188 @Override  
189 public boolean containsValue(final Object value) {  
190 checkValue(value);  
191 return lookupValue(value) != null;  
192 }  
193  
194 /\*\*  
195 \* Gets the value to which this map maps the specified key.  
196 \* Returns null if the map contains no mapping for this key.  
197 \* <p>  
198 \* The key must implement <code>Comparable</code>.  
199 \*  
200 \* @param key key whose associated value is to be returned  
201 \* @return the value to which this map maps the specified key,  
202 \* or null if the map contains no mapping for this key  
203 \* @throws ClassCastException if the key is of an inappropriate type  
204 \* @throws NullPointerException if the key is null  
205 \*/  
206 @Override  
207 public V get(final Object key) {  
208 checkKey(key);  
209 final Node<K, V> node = lookupKey(key);  
210 return node == null ? null : node.getValue();  
211 }  
212  
213 /\*\*  
214 \* Puts the key-value pair into the map, replacing any previous pair.  
215 \* <p>  
216 \* When adding a key-value pair, the value may already exist in the map  
217 \* against a different key. That mapping is removed, to ensure that the  
218 \* value only occurs once in the inverse map.  
219 \* <pre>  
220 \* BidiMap map1 = new TreeBidiMap();  
221 \* map.put("A","B"); // contains A mapped to B, as per Map  
222 \* map.put("A","C"); // contains A mapped to C, as per Map  
223 \*  
224 \* BidiMap map2 = new TreeBidiMap();  
225 \* map.put("A","B"); // contains A mapped to B, as per Map  
226 \* map.put("C","B"); // contains C mapped to B, key A is removed  
227 \* </pre>  
228 \* <p>  
229 \* Both key and value must implement <code>Comparable</code>.  
230 \*  
231 \* @param key key with which the specified value is to be associated  
232 \* @param value value to be associated with the specified key  
233 \* @return the previous value for the key  
234 \* @throws ClassCastException if the key is of an inappropriate type  
235 \* @throws NullPointerException if the key is null  
236 \*/  
237 @Override  
238 public V put(final K key, final V value) {  
239 final V result = get(key);  
240 doPut(key, value);  
241 return result;  
242 }  
243  
244 /\*\*  
245 \* Puts all the mappings from the specified map into this map.  
246 \* <p>  
247 \* All keys and values must implement <code>Comparable</code>.  
248 \*  
249 \* @param map the map to copy from  
250 \*/  
251 @Override  
252 public void putAll(final Map<? extends K, ? extends V> map) {  
253 for (final Map.Entry<? extends K, ? extends V> e : map.entrySet()) {  
254 put(e.getKey(), e.getValue());  
255 }  
256 }  
257  
258 /\*\*  
259 \* Removes the mapping for this key from this map if present.  
260 \* <p>  
261 \* The key must implement <code>Comparable</code>.  
262 \*  
263 \* @param key key whose mapping is to be removed from the map.  
264 \* @return previous value associated with specified key,  
265 \* or null if there was no mapping for key.  
266 \* @throws ClassCastException if the key is of an inappropriate type  
267 \* @throws NullPointerException if the key is null  
268 \*/  
269 @Override  
270 public V remove(final Object key) {  
271 return doRemoveKey(key);  
272 }  
273  
274 /\*\*  
275 \* Removes all mappings from this map.  
276 \*/  
277 @Override  
278 public void clear() {  
279 modify();  
280  
281 nodeCount = 0;  
282 rootNode[KEY.ordinal()] = null;  
283 rootNode[VALUE.ordinal()] = null;  
284 }  
285  
286 //-----------------------------------------------------------------------  
287 /\*\*  
288 \* Returns the key to which this map maps the specified value.  
289 \* Returns null if the map contains no mapping for this value.  
290 \* <p>  
291 \* The value must implement <code>Comparable</code>.  
292 \*  
293 \* @param value value whose associated key is to be returned.  
294 \* @return the key to which this map maps the specified value,  
295 \* or null if the map contains no mapping for this value.  
296 \* @throws ClassCastException if the value is of an inappropriate type  
297 \* @throws NullPointerException if the value is null  
298 \*/  
299 @Override  
300 public K getKey(final Object value) {  
301 checkValue(value);  
302 final Node<K, V> node = lookupValue(value);  
303 return node == null ? null : node.getKey();  
304 }  
305  
306 /\*\*  
307 \* Removes the mapping for this value from this map if present.  
308 \* <p>  
309 \* The value must implement <code>Comparable</code>.  
310 \*  
311 \* @param value value whose mapping is to be removed from the map  
312 \* @return previous key associated with specified value,  
313 \* or null if there was no mapping for value.  
314 \* @throws ClassCastException if the value is of an inappropriate type  
315 \* @throws NullPointerException if the value is null  
316 \*/  
317 @Override  
318 public K removeValue(final Object value) {  
319 return doRemoveValue(value);  
320 }  
321  
322 //-----------------------------------------------------------------------  
323 /\*\*  
324 \* Gets the first (lowest) key currently in this map.  
325 \*  
326 \* @return the first (lowest) key currently in this sorted map  
327 \* @throws NoSuchElementException if this map is empty  
328 \*/  
329 @Override  
330 public K firstKey() {  
331 if (nodeCount == 0) {  
332 throw new NoSuchElementException("Map is empty");  
333 }  
334 return leastNode(rootNode[KEY.ordinal()], KEY).getKey();  
335 }  
336  
337 /\*\*  
338 \* Gets the last (highest) key currently in this map.  
339 \*  
340 \* @return the last (highest) key currently in this sorted map  
341 \* @throws NoSuchElementException if this map is empty  
342 \*/  
343 @Override  
344 public K lastKey() {  
345 if (nodeCount == 0) {  
346 throw new NoSuchElementException("Map is empty");  
347 }  
348 return greatestNode(rootNode[KEY.ordinal()], KEY).getKey();  
349 }  
350  
351 /\*\*  
352 \* Gets the next key after the one specified.  
353 \* <p>  
354 \* The key must implement <code>Comparable</code>.  
355 \*  
356 \* @param key the key to search for next from  
357 \* @return the next key, null if no match or at end  
358 \*/  
359 @Override  
360 public K nextKey(final K key) {  
361 checkKey(key);  
362 final Node<K, V> node = nextGreater(lookupKey(key), KEY);  
363 return node == null ? null : node.getKey();  
364 }  
365  
366 /\*\*  
367 \* Gets the previous key before the one specified.  
368 \* <p>  
369 \* The key must implement <code>Comparable</code>.  
370 \*  
371 \* @param key the key to search for previous from  
372 \* @return the previous key, null if no match or at start  
373 \*/  
374 @Override  
375 public K previousKey(final K key) {  
376 checkKey(key);  
377 final Node<K, V> node = nextSmaller(lookupKey(key), KEY);  
378 return node == null ? null : node.getKey();  
379 }  
380  
381 //-----------------------------------------------------------------------  
382 /\*\*  
383 \* Returns a set view of the keys contained in this map in key order.  
384 \* <p>  
385 \* The set is backed by the map, so changes to the map are reflected in  
386 \* the set, and vice-versa. If the map is modified while an iteration over  
387 \* the set is in progress, the results of the iteration are undefined.  
388 \* <p>  
389 \* The set supports element removal, which removes the corresponding mapping  
390 \* from the map. It does not support the add or addAll operations.  
391 \*  
392 \* @return a set view of the keys contained in this map.  
393 \*/  
394 @Override  
395 public Set<K> keySet() {  
396 if (keySet == null) {  
397 keySet = new KeyView(KEY);  
398 }  
399 return keySet;  
400 }  
401  
402 //-----------------------------------------------------------------------  
403 /\*\*  
404 \* Returns a set view of the values contained in this map in key order.  
405 \* The returned object can be cast to a Set.  
406 \* <p>  
407 \* The set is backed by the map, so changes to the map are reflected in  
408 \* the set, and vice-versa. If the map is modified while an iteration over  
409 \* the set is in progress, the results of the iteration are undefined.  
410 \* <p>  
411 \* The set supports element removal, which removes the corresponding mapping  
412 \* from the map. It does not support the add or addAll operations.  
413 \*  
414 \* @return a set view of the values contained in this map.  
415 \*/  
416 @Override  
417 public Set<V> values() {  
418 if (valuesSet == null) {  
419 valuesSet = new ValueView(KEY);  
420 }  
421 return valuesSet;  
422 }  
423  
424 //-----------------------------------------------------------------------  
425 /\*\*  
426 \* Returns a set view of the entries contained in this map in key order.  
427 \* For simple iteration through the map, the MapIterator is quicker.  
428 \* <p>  
429 \* The set is backed by the map, so changes to the map are reflected in  
430 \* the set, and vice-versa. If the map is modified while an iteration over  
431 \* the set is in progress, the results of the iteration are undefined.  
432 \* <p>  
433 \* The set supports element removal, which removes the corresponding mapping  
434 \* from the map. It does not support the add or addAll operations.  
435 \* The returned MapEntry objects do not support setValue.  
436 \*  
437 \* @return a set view of the values contained in this map.  
438 \*/  
439 @Override  
440 public Set<Map.Entry<K, V>> entrySet() {  
441 if (entrySet == null) {  
442 entrySet = new EntryView();  
443 }  
444 return entrySet;  
445 }  
446  
447 //-----------------------------------------------------------------------  
448 @Override  
449 public OrderedMapIterator<K, V> mapIterator() {  
450 if (isEmpty()) {  
451 return EmptyOrderedMapIterator.<K, V>emptyOrderedMapIterator();  
452 }  
453 return new ViewMapIterator(KEY);  
454 }  
455  
456 //-----------------------------------------------------------------------  
457 /\*\*  
458 \* Gets the inverse map for comparison.  
459 \*  
460 \* @return the inverse map  
461 \*/  
462 @Override  
463 public OrderedBidiMap<V, K> inverseBidiMap() {  
464 if (inverse == null) {  
465 inverse = new Inverse();  
466 }  
467 return inverse;  
468 }  
469  
470 //-----------------------------------------------------------------------  
471 /\*\*  
472 \* Compares for equals as per the API.  
473 \*  
474 \* @param obj the object to compare to  
475 \* @return true if equal  
476 \*/  
477 @Override  
478 public boolean equals(final Object obj) {  
479 return this.doEquals(obj, KEY);  
480 }  
481  
482 /\*\*  
483 \* Gets the hash code value for this map as per the API.  
484 \*  
485 \* @return the hash code value for this map  
486 \*/  
487 @Override  
488 public int hashCode() {  
489 return this.doHashCode(KEY);  
490 }  
491  
492 /\*\*  
493 \* Returns a string version of this Map in standard format.  
494 \*  
495 \* @return a standard format string version of the map  
496 \*/  
497 @Override  
498 public String toString() {  
499 return this.doToString(KEY);  
500 }  
501  
502 //-----------------------------------------------------------------------  
503 /\*\*  
504 \* Put logic.  
505 \*  
506 \* @param key the key, always the main map key  
507 \* @param value the value, always the main map value  
508 \*/  
509 private void doPut(final K key, final V value) {  
510 checkKeyAndValue(key, value);  
511  
512 // store previous and remove previous mappings  
513 doRemoveKey(key);  
514 doRemoveValue(value);  
515  
516 Node<K, V> node = rootNode[KEY.ordinal()];  
517 if (node == null) {  
518 // map is empty  
519 final Node<K, V> root = new Node<>(key, value);  
520 rootNode[KEY.ordinal()] = root;  
521 rootNode[VALUE.ordinal()] = root;  
522 grow();  
523  
524 } else {  
525 // add new mapping  
526 while (true) {  
527 final int cmp = compare(key, node.getKey());  
528  
529 if (cmp == 0) {  
530 // shouldn't happen  
531 throw new IllegalArgumentException("Cannot store a duplicate key (\"" + key + "\") in this Map");  
532 } else if (cmp < 0) {  
533 if (node.getLeft(KEY) != null) {  
534 node = node.getLeft(KEY);  
535 } else {  
536 final Node<K, V> newNode = new Node<>(key, value);  
537  
538 insertValue(newNode);  
539 node.setLeft(newNode, KEY);  
540 newNode.setParent(node, KEY);  
541 doRedBlackInsert(newNode, KEY);  
542 grow();  
543  
544 break;  
545 }  
546 } else { // cmp > 0  
547 if (node.getRight(KEY) != null) {  
548 node = node.getRight(KEY);  
549 } else {  
550 final Node<K, V> newNode = new Node<>(key, value);  
551  
552 insertValue(newNode);  
553 node.setRight(newNode, KEY);  
554 newNode.setParent(node, KEY);  
555 doRedBlackInsert(newNode, KEY);  
556 grow();  
557  
558 break;  
559 }  
560 }  
561 }  
562 }  
563 }  
564  
565 private V doRemoveKey(final Object key) {  
566 final Node<K, V> node = lookupKey(key);  
567 if (node == null) {  
568 return null;  
569 }  
570 doRedBlackDelete(node);  
571 return node.getValue();  
572 }  
573  
574 private K doRemoveValue(final Object value) {  
575 final Node<K, V> node = lookupValue(value);  
576 if (node == null) {  
577 return null;  
578 }  
579 doRedBlackDelete(node);  
580 return node.getKey();  
581 }  
582  
583 /\*\*  
584 \* do the actual lookup of a piece of data  
585 \*  
586 \* @param data the key or value to be looked up  
587 \* @param dataElement either {@link DataElement#KEY} key}  
588 \* or the {@link DataElement#VALUE value}.  
589 \* @return the desired Node, or null if there is no mapping of the  
590 \* specified data  
591 \*/  
592 @SuppressWarnings("unchecked")  
593 private <T extends Comparable<T>> Node<K, V> lookup(final Object data, final DataElement dataElement) {  
594 Node<K, V> rval = null;  
595 Node<K, V> node = rootNode[dataElement.ordinal()];  
596  
597 while (node != null) {  
598 final int cmp = compare((T) data, (T) node.getData(dataElement));  
599 if (cmp == 0) {  
600 rval = node;  
601 break;  
602 }  
603 node = cmp < 0 ? node.getLeft(dataElement) : node.getRight(dataElement);  
604 }  
605  
606 return rval;  
607 }  
608  
609 private Node<K, V> lookupKey(final Object key) {  
610 return this.<K>lookup(key, KEY);  
611 }  
612  
613 private Node<K, V> lookupValue(final Object value) {  
614 return this.<V>lookup(value, VALUE);  
615 }  
616  
617 /\*\*  
618 \* get the next larger node from the specified node  
619 \*  
620 \* @param node the node to be searched from  
621 \* @param dataElement either {@link DataElement#KEY} key}  
622 \* or the {@link DataElement#VALUE value}.  
623 \* @return the specified node  
624 \*/  
625 private Node<K, V> nextGreater(final Node<K, V> node, final DataElement dataElement) {  
626 Node<K, V> rval;  
627 if (node == null) {  
628 rval = null;  
629 } else if (node.getRight(dataElement) != null) {  
630 // everything to the node's right is larger. The least of  
631 // the right node's descendants is the next larger node  
632 rval = leastNode(node.getRight(dataElement), dataElement);  
633 } else {  
634 // traverse up our ancestry until we find an ancestor that  
635 // is null or one whose left child is our ancestor. If we  
636 // find a null, then this node IS the largest node in the  
637 // tree, and there is no greater node. Otherwise, we are  
638 // the largest node in the subtree on that ancestor's left  
639 // ... and that ancestor is the next greatest node  
640 Node<K, V> parent = node.getParent(dataElement);  
641 Node<K, V> child = node;  
642  
643 while (parent != null && child == parent.getRight(dataElement)) {  
644 child = parent;  
645 parent = parent.getParent(dataElement);  
646 }  
647 rval = parent;  
648 }  
649 return rval;  
650 }  
651  
652 /\*\*  
653 \* get the next larger node from the specified node  
654 \*  
655 \* @param node the node to be searched from  
656 \* @param dataElement either {@link DataElement#KEY} key}  
657 \* or the {@link DataElement#VALUE value}.  
658 \* @return the specified node  
659 \*/  
660 private Node<K, V> nextSmaller(final Node<K, V> node, final DataElement dataElement) {  
661 Node<K, V> rval;  
662 if (node == null) {  
663 rval = null;  
664 } else if (node.getLeft(dataElement) != null) {  
665 // everything to the node's left is smaller. The greatest of  
666 // the left node's descendants is the next smaller node  
667 rval = greatestNode(node.getLeft(dataElement), dataElement);  
668 } else {  
669 // traverse up our ancestry until we find an ancestor that  
670 // is null or one whose right child is our ancestor. If we  
671 // find a null, then this node IS the largest node in the  
672 // tree, and there is no greater node. Otherwise, we are  
673 // the largest node in the subtree on that ancestor's right  
674 // ... and that ancestor is the next greatest node  
675 Node<K, V> parent = node.getParent(dataElement);  
676 Node<K, V> child = node;  
677  
678 while (parent != null && child == parent.getLeft(dataElement)) {  
679 child = parent;  
680 parent = parent.getParent(dataElement);  
681 }  
682 rval = parent;  
683 }  
684 return rval;  
685 }  
686  
687 //-----------------------------------------------------------------------  
688  
689 /\*\*  
690 \* Compare two objects  
691 \*  
692 \* @param o1 the first object  
693 \* @param o2 the second object  
694 \*  
695 \* @return negative value if o1 < o2; 0 if o1 == o2; positive  
696 \* value if o1 > o2  
697 \*/  
698 private static <T extends Comparable<T>> int compare(final T o1, final T o2) {  
699 return o1.compareTo(o2);  
700 }  
701  
702 /\*\*  
703 \* Find the least node from a given node.  
704 \*  
705 \* @param node the node from which we will start searching  
706 \* @param dataElement either {@link DataElement#KEY} key}  
707 \* or the {@link DataElement#VALUE value}.  
708 \* @return the smallest node, from the specified node, in the  
709 \* specified mapping  
710 \*/  
711 private Node<K, V> leastNode(final Node<K, V> node, final DataElement dataElement) {  
712 Node<K, V> rval = node;  
713 if (rval != null) {  
714 while (rval.getLeft(dataElement) != null) {  
715 rval = rval.getLeft(dataElement);  
716 }  
717 }  
718 return rval;  
719 }  
720  
721 /\*\*  
722 \* Find the greatest node from a given node.  
723 \*  
724 \* @param node the node from which we will start searching  
725 \* @param dataElement either {@link DataElement#KEY} key}  
726 \* or the {@link DataElement#VALUE value}.  
727 \* @return the greatest node, from the specified node  
728 \*/  
729 private Node<K, V> greatestNode(final Node<K, V> node, final DataElement dataElement) {  
730 Node<K, V> rval = node;  
731 if (rval != null) {  
732 while (rval.getRight(dataElement) != null) {  
733 rval = rval.getRight(dataElement);  
734 }  
735 }  
736 return rval;  
737 }  
738  
739 /\*\*  
740 \* copy the color from one node to another, dealing with the fact  
741 \* that one or both nodes may, in fact, be null  
742 \*  
743 \* @param from the node whose color we're copying; may be null  
744 \* @param to the node whose color we're changing; may be null  
745 \* @param dataElement either {@link DataElement#KEY} key}  
746 \* or the {@link DataElement#VALUE value}.  
747 \*/  
748 private void copyColor(final Node<K, V> from, final Node<K, V> to, final DataElement dataElement) {  
749 if (to != null) {  
750 if (from == null) {  
751 // by default, make it black  
752 to.setBlack(dataElement);  
753 } else {  
754 to.copyColor(from, dataElement);  
755 }  
756 }  
757 }  
758  
759 /\*\*  
760 \* is the specified node red? if the node does not exist, no, it's  
761 \* black, thank you  
762 \*  
763 \* @param node the node (may be null) in question  
764 \* @param dataElement either {@link DataElement#KEY} key}  
765 \* or the {@link DataElement#VALUE value}.  
766 \*/  
767 private static boolean isRed(final Node<?, ?> node, final DataElement dataElement) {  
768 return node != null && node.isRed(dataElement);  
769 }  
770  
771 /\*\*  
772 \* is the specified black red? if the node does not exist, sure,  
773 \* it's black, thank you  
774 \*  
775 \* @param node the node (may be null) in question  
776 \* @param dataElement either {@link DataElement#KEY} key}  
777 \* or the {@link DataElement#VALUE value}.  
778 \*/  
779 private static boolean isBlack(final Node<?, ?> node, final DataElement dataElement) {  
780 return node == null || node.isBlack(dataElement);  
781 }  
782  
783 /\*\*  
784 \* force a node (if it exists) red  
785 \*  
786 \* @param node the node (may be null) in question  
787 \* @param dataElement either {@link DataElement#KEY} key}  
788 \* or the {@link DataElement#VALUE value}.  
789 \*/  
790 private static void makeRed(final Node<?, ?> node, final DataElement dataElement) {  
791 if (node != null) {  
792 node.setRed(dataElement);  
793 }  
794 }  
795  
796 /\*\*  
797 \* force a node (if it exists) black  
798 \*  
799 \* @param node the node (may be null) in question  
800 \* @param dataElement either {@link DataElement#KEY} key}  
801 \* or the {@link DataElement#VALUE value}.  
802 \*/  
803 private static void makeBlack(final Node<?, ?> node, final DataElement dataElement) {  
804 if (node != null) {  
805 node.setBlack(dataElement);  
806 }  
807 }  
808  
809 /\*\*  
810 \* get a node's grandparent. mind you, the node, its parent, or  
811 \* its grandparent may not exist. no problem  
812 \*  
813 \* @param node the node (may be null) in question  
814 \* @param dataElement either {@link DataElement#KEY} key}  
815 \* or the {@link DataElement#VALUE value}.  
816 \*/  
817 private Node<K, V> getGrandParent(final Node<K, V> node, final DataElement dataElement) {  
818 return getParent(getParent(node, dataElement), dataElement);  
819 }  
820  
821 /\*\*  
822 \* get a node's parent. mind you, the node, or its parent, may not  
823 \* exist. no problem  
824 \*  
825 \* @param node the node (may be null) in question  
826 \* @param dataElement either {@link DataElement#KEY} key}  
827 \* or the {@link DataElement#VALUE value}.  
828 \*/  
829 private Node<K, V> getParent(final Node<K, V> node, final DataElement dataElement) {  
830 return node == null ? null : node.getParent(dataElement);  
831 }  
832  
833 /\*\*  
834 \* get a node's right child. mind you, the node may not exist. no  
835 \* problem  
836 \*  
837 \* @param node the node (may be null) in question  
838 \* @param dataElement either {@link DataElement#KEY} key}  
839 \* or the {@link DataElement#VALUE value}.  
840 \*/  
841 private Node<K, V> getRightChild(final Node<K, V> node, final DataElement dataElement) {  
842 return node == null ? null : node.getRight(dataElement);  
843 }  
844  
845 /\*\*  
846 \* get a node's left child. mind you, the node may not exist. no  
847 \* problem  
848 \*  
849 \* @param node the node (may be null) in question  
850 \* @param dataElement either {@link DataElement#KEY} key}  
851 \* or the {@link DataElement#VALUE value}.  
852 \*/  
853 private Node<K, V> getLeftChild(final Node<K, V> node, final DataElement dataElement) {  
854 return node == null ? null : node.getLeft(dataElement);  
855 }  
856  
857 /\*\*  
858 \* do a rotate left. standard fare in the world of balanced trees  
859 \*  
860 \* @param node the node to be rotated  
861 \* @param dataElement either {@link DataElement#KEY} key}  
862 \* or the {@link DataElement#VALUE value}.  
863 \*/  
864 private void rotateLeft(final Node<K, V> node, final DataElement dataElement) {  
865 final Node<K, V> rightChild = node.getRight(dataElement);  
866 node.setRight(rightChild.getLeft(dataElement), dataElement);  
867  
868 if (rightChild.getLeft(dataElement) != null) {  
869 rightChild.getLeft(dataElement).setParent(node, dataElement);  
870 }  
871 rightChild.setParent(node.getParent(dataElement), dataElement);  
872  
873 if (node.getParent(dataElement) == null) {  
874 // node was the root ... now its right child is the root  
875 rootNode[dataElement.ordinal()] = rightChild;  
876 } else if (node.getParent(dataElement).getLeft(dataElement) == node) {  
877 node.getParent(dataElement).setLeft(rightChild, dataElement);  
878 } else {  
879 node.getParent(dataElement).setRight(rightChild, dataElement);  
880 }  
881  
882 rightChild.setLeft(node, dataElement);  
883 node.setParent(rightChild, dataElement);  
884 }  
885  
886 /\*\*  
887 \* do a rotate right. standard fare in the world of balanced trees  
888 \*  
889 \* @param node the node to be rotated  
890 \* @param dataElement either {@link DataElement#KEY} key}  
891 \* or the {@link DataElement#VALUE value}.  
892 \*/  
893 private void rotateRight(final Node<K, V> node, final DataElement dataElement) {  
894 final Node<K, V> leftChild = node.getLeft(dataElement);  
895 node.setLeft(leftChild.getRight(dataElement), dataElement);  
896 if (leftChild.getRight(dataElement) != null) {  
897 leftChild.getRight(dataElement).setParent(node, dataElement);  
898 }  
899 leftChild.setParent(node.getParent(dataElement), dataElement);  
900  
901 if (node.getParent(dataElement) == null) {  
902 // node was the root ... now its left child is the root  
903 rootNode[dataElement.ordinal()] = leftChild;  
904 } else if (node.getParent(dataElement).getRight(dataElement) == node) {  
905 node.getParent(dataElement).setRight(leftChild, dataElement);  
906 } else {  
907 node.getParent(dataElement).setLeft(leftChild, dataElement);  
908 }  
909  
910 leftChild.setRight(node, dataElement);  
911 node.setParent(leftChild, dataElement);  
912 }  
913  
914 /\*\*  
915 \* complicated red-black insert stuff. Based on Sun's TreeMap  
916 \* implementation, though it's barely recognizable any more  
917 \*  
918 \* @param insertedNode the node to be inserted  
919 \* @param dataElement the KEY or VALUE int  
920 \*/  
921 private void doRedBlackInsert(final Node<K, V> insertedNode, final DataElement dataElement) {  
922 Node<K, V> currentNode = insertedNode;  
923 makeRed(currentNode, dataElement);  
924  
925 while (currentNode != null  
926 && currentNode != rootNode[dataElement.ordinal()]  
927 && isRed(currentNode.getParent(dataElement), dataElement)) {  
928 if (currentNode.isLeftChild(dataElement)) {  
929 final Node<K, V> y = getRightChild(getGrandParent(currentNode, dataElement), dataElement);  
930  
931 if (isRed(y, dataElement)) {  
932 makeBlack(getParent(currentNode, dataElement), dataElement);  
933 makeBlack(y, dataElement);  
934 makeRed(getGrandParent(currentNode, dataElement), dataElement);  
935  
936 currentNode = getGrandParent(currentNode, dataElement);  
937 } else {  
938 //dead code?  
939 if (currentNode.isRightChild(dataElement)) {  
940 currentNode = getParent(currentNode, dataElement);  
941  
942 rotateLeft(currentNode, dataElement);  
943 }  
944  
945 makeBlack(getParent(currentNode, dataElement), dataElement);  
946 makeRed(getGrandParent(currentNode, dataElement), dataElement);  
947  
948 if (getGrandParent(currentNode, dataElement) != null) {  
949 rotateRight(getGrandParent(currentNode, dataElement), dataElement);  
950 }  
951 }  
952 } else {  
953  
954 // just like clause above, except swap left for right  
955 final Node<K, V> y = getLeftChild(getGrandParent(currentNode, dataElement), dataElement);  
956  
957 if (isRed(y, dataElement)) {  
958 makeBlack(getParent(currentNode, dataElement), dataElement);  
959 makeBlack(y, dataElement);  
960 makeRed(getGrandParent(currentNode, dataElement), dataElement);  
961  
962 currentNode = getGrandParent(currentNode, dataElement);  
963 } else {  
964 //dead code?  
965 if (currentNode.isLeftChild(dataElement)) {  
966 currentNode = getParent(currentNode, dataElement);  
967  
968 rotateRight(currentNode, dataElement);  
969 }  
970  
971 makeBlack(getParent(currentNode, dataElement), dataElement);  
972 makeRed(getGrandParent(currentNode, dataElement), dataElement);  
973  
974 if (getGrandParent(currentNode, dataElement) != null) {  
975 rotateLeft(getGrandParent(currentNode, dataElement), dataElement);  
976 }  
977 }  
978 }  
979 }  
980  
981 makeBlack(rootNode[dataElement.ordinal()], dataElement);  
982 }  
983  
984 /\*\*  
985 \* complicated red-black delete stuff. Based on Sun's TreeMap  
986 \* implementation, though it's barely recognizable any more  
987 \*  
988 \* @param deletedNode the node to be deleted  
989 \*/  
990 private void doRedBlackDelete(final Node<K, V> deletedNode) {  
991 for (final DataElement dataElement : DataElement.values()) {  
992 // if deleted node has both left and children, swap with  
993 // the next greater node  
994 if (deletedNode.getLeft(dataElement) != null && deletedNode.getRight(dataElement) != null) {  
995 swapPosition(nextGreater(deletedNode, dataElement), deletedNode, dataElement);  
996 }  
997  
998 final Node<K, V> replacement = deletedNode.getLeft(dataElement) != null ?  
999 deletedNode.getLeft(dataElement) : deletedNode.getRight(dataElement);  
1000  
1001 if (replacement != null) {  
1002 replacement.setParent(deletedNode.getParent(dataElement), dataElement);  
1003  
1004 if (deletedNode.getParent(dataElement) == null) {  
1005 rootNode[dataElement.ordinal()] = replacement;  
1006 } else if (deletedNode == deletedNode.getParent(dataElement).getLeft(dataElement)) {  
1007 deletedNode.getParent(dataElement).setLeft(replacement, dataElement);  
1008 } else {  
1009 deletedNode.getParent(dataElement).setRight(replacement, dataElement);  
1010 }  
1011  
1012 deletedNode.setLeft(null, dataElement);  
1013 deletedNode.setRight(null, dataElement);  
1014 deletedNode.setParent(null, dataElement);  
1015  
1016 if (isBlack(deletedNode, dataElement)) {  
1017 doRedBlackDeleteFixup(replacement, dataElement);  
1018 }  
1019 } else {  
1020  
1021 // replacement is null  
1022 if (deletedNode.getParent(dataElement) == null) {  
1023  
1024 // empty tree  
1025 rootNode[dataElement.ordinal()] = null;  
1026 } else {  
1027  
1028 // deleted node had no children  
1029 if (isBlack(deletedNode, dataElement)) {  
1030 doRedBlackDeleteFixup(deletedNode, dataElement);  
1031 }  
1032  
1033 if (deletedNode.getParent(dataElement) != null) {  
1034 if (deletedNode == deletedNode.getParent(dataElement).getLeft(dataElement)) {  
1035 deletedNode.getParent(dataElement).setLeft(null, dataElement);  
1036 } else {  
1037 deletedNode.getParent(dataElement).setRight(null, dataElement);  
1038 }  
1039  
1040 deletedNode.setParent(null, dataElement);  
1041 }  
1042 }  
1043 }  
1044 }  
1045 shrink();  
1046 }  
1047  
1048 /\*\*  
1049 \* complicated red-black delete stuff. Based on Sun's TreeMap  
1050 \* implementation, though it's barely recognizable any more. This  
1051 \* rebalances the tree (somewhat, as red-black trees are not  
1052 \* perfectly balanced -- perfect balancing takes longer)  
1053 \*  
1054 \* @param replacementNode the node being replaced  
1055 \* @param dataElement the KEY or VALUE int  
1056 \*/  
1057 private void doRedBlackDeleteFixup(final Node<K, V> replacementNode, final DataElement dataElement) {  
1058 Node<K, V> currentNode = replacementNode;  
1059  
1060 while (currentNode != rootNode[dataElement.ordinal()] && isBlack(currentNode, dataElement)) {  
1061 if (currentNode.isLeftChild(dataElement)) {  
1062 Node<K, V> siblingNode = getRightChild(getParent(currentNode, dataElement), dataElement);  
1063  
1064 if (isRed(siblingNode, dataElement)) {  
1065 makeBlack(siblingNode, dataElement);  
1066 makeRed(getParent(currentNode, dataElement), dataElement);  
1067 rotateLeft(getParent(currentNode, dataElement), dataElement);  
1068  
1069 siblingNode = getRightChild(getParent(currentNode, dataElement), dataElement);  
1070 }  
1071  
1072 if (isBlack(getLeftChild(siblingNode, dataElement), dataElement)  
1073 && isBlack(getRightChild(siblingNode, dataElement), dataElement)) {  
1074 makeRed(siblingNode, dataElement);  
1075  
1076 currentNode = getParent(currentNode, dataElement);  
1077 } else {  
1078 if (isBlack(getRightChild(siblingNode, dataElement), dataElement)) {  
1079 makeBlack(getLeftChild(siblingNode, dataElement), dataElement);  
1080 makeRed(siblingNode, dataElement);  
1081 rotateRight(siblingNode, dataElement);  
1082  
1083 siblingNode = getRightChild(getParent(currentNode, dataElement), dataElement);  
1084 }  
1085  
1086 copyColor(getParent(currentNode, dataElement), siblingNode, dataElement);  
1087 makeBlack(getParent(currentNode, dataElement), dataElement);  
1088 makeBlack(getRightChild(siblingNode, dataElement), dataElement);  
1089 rotateLeft(getParent(currentNode, dataElement), dataElement);  
1090  
1091 currentNode = rootNode[dataElement.ordinal()];  
1092 }  
1093 } else {  
1094 Node<K, V> siblingNode = getLeftChild(getParent(currentNode, dataElement), dataElement);  
1095  
1096 if (isRed(siblingNode, dataElement)) {  
1097 makeBlack(siblingNode, dataElement);  
1098 makeRed(getParent(currentNode, dataElement), dataElement);  
1099 rotateRight(getParent(currentNode, dataElement), dataElement);  
1100  
1101 siblingNode = getLeftChild(getParent(currentNode, dataElement), dataElement);  
1102 }  
1103  
1104 if (isBlack(getRightChild(siblingNode, dataElement), dataElement)  
1105 && isBlack(getLeftChild(siblingNode, dataElement), dataElement)) {  
1106 makeRed(siblingNode, dataElement);  
1107  
1108 currentNode = getParent(currentNode, dataElement);  
1109 } else {  
1110 if (isBlack(getLeftChild(siblingNode, dataElement), dataElement)) {  
1111 makeBlack(getRightChild(siblingNode, dataElement), dataElement);  
1112 makeRed(siblingNode, dataElement);  
1113 rotateLeft(siblingNode, dataElement);  
1114  
1115 siblingNode = getLeftChild(getParent(currentNode, dataElement), dataElement);  
1116 }  
1117  
1118 copyColor(getParent(currentNode, dataElement), siblingNode, dataElement);  
1119 makeBlack(getParent(currentNode, dataElement), dataElement);  
1120 makeBlack(getLeftChild(siblingNode, dataElement), dataElement);  
1121 rotateRight(getParent(currentNode, dataElement), dataElement);  
1122  
1123 currentNode = rootNode[dataElement.ordinal()];  
1124 }  
1125 }  
1126 }  
1127  
1128 makeBlack(currentNode, dataElement);  
1129 }  
1130  
1131 /\*\*  
1132 \* swap two nodes (except for their content), taking care of  
1133 \* special cases where one is the other's parent ... hey, it  
1134 \* happens.  
1135 \*  
1136 \* @param x one node  
1137 \* @param y another node  
1138 \* @param dataElement the KEY or VALUE int  
1139 \*/  
1140 private void swapPosition(final Node<K, V> x, final Node<K, V> y, final DataElement dataElement) {  
1141 // Save initial values.  
1142 final Node<K, V> xFormerParent = x.getParent(dataElement);  
1143 final Node<K, V> xFormerLeftChild = x.getLeft(dataElement);  
1144 final Node<K, V> xFormerRightChild = x.getRight(dataElement);  
1145 final Node<K, V> yFormerParent = y.getParent(dataElement);  
1146 final Node<K, V> yFormerLeftChild = y.getLeft(dataElement);  
1147 final Node<K, V> yFormerRightChild = y.getRight(dataElement);  
1148 final boolean xWasLeftChild =  
1149 x.getParent(dataElement) != null && x == x.getParent(dataElement).getLeft(dataElement);  
1150 final boolean yWasLeftChild =  
1151 y.getParent(dataElement) != null && y == y.getParent(dataElement).getLeft(dataElement);  
1152  
1153 // Swap, handling special cases of one being the other's parent.  
1154 if (x == yFormerParent) { // x was y's parent  
1155 x.setParent(y, dataElement);  
1156  
1157 if (yWasLeftChild) {  
1158 y.setLeft(x, dataElement);  
1159 y.setRight(xFormerRightChild, dataElement);  
1160 } else {  
1161 y.setRight(x, dataElement);  
1162 y.setLeft(xFormerLeftChild, dataElement);  
1163 }  
1164 } else {  
1165 x.setParent(yFormerParent, dataElement);  
1166  
1167 if (yFormerParent != null) {  
1168 if (yWasLeftChild) {  
1169 yFormerParent.setLeft(x, dataElement);  
1170 } else {  
1171 yFormerParent.setRight(x, dataElement);  
1172 }  
1173 }  
1174  
1175 y.setLeft(xFormerLeftChild, dataElement);  
1176 y.setRight(xFormerRightChild, dataElement);  
1177 }  
1178  
1179 if (y == xFormerParent) { // y was x's parent  
1180 y.setParent(x, dataElement);  
1181  
1182 if (xWasLeftChild) {  
1183 x.setLeft(y, dataElement);  
1184 x.setRight(yFormerRightChild, dataElement);  
1185 } else {  
1186 x.setRight(y, dataElement);  
1187 x.setLeft(yFormerLeftChild, dataElement);  
1188 }  
1189 } else {  
1190 y.setParent(xFormerParent, dataElement);  
1191  
1192 if (xFormerParent != null) {  
1193 if (xWasLeftChild) {  
1194 xFormerParent.setLeft(y, dataElement);  
1195 } else {  
1196 xFormerParent.setRight(y, dataElement);  
1197 }  
1198 }  
1199  
1200 x.setLeft(yFormerLeftChild, dataElement);  
1201 x.setRight(yFormerRightChild, dataElement);  
1202 }  
1203  
1204 // Fix children's parent pointers  
1205 if (x.getLeft(dataElement) != null) {  
1206 x.getLeft(dataElement).setParent(x, dataElement);  
1207 }  
1208  
1209 if (x.getRight(dataElement) != null) {  
1210 x.getRight(dataElement).setParent(x, dataElement);  
1211 }  
1212  
1213 if (y.getLeft(dataElement) != null) {  
1214 y.getLeft(dataElement).setParent(y, dataElement);  
1215 }  
1216  
1217 if (y.getRight(dataElement) != null) {  
1218 y.getRight(dataElement).setParent(y, dataElement);  
1219 }  
1220  
1221 x.swapColors(y, dataElement);  
1222  
1223 // Check if root changed  
1224 if (rootNode[dataElement.ordinal()] == x) {  
1225 rootNode[dataElement.ordinal()] = y;  
1226 } else if (rootNode[dataElement.ordinal()] == y) {  
1227 rootNode[dataElement.ordinal()] = x;  
1228 }  
1229 }  
1230  
1231 /\*\*  
1232 \* check if an object is fit to be proper input ... has to be  
1233 \* Comparable and non-null  
1234 \*  
1235 \* @param o the object being checked  
1236 \* @param dataElement either {@link DataElement#KEY} key}  
1237 \* or the {@link DataElement#VALUE value}.  
1238 \*  
1239 \* @throws NullPointerException if o is null  
1240 \* @throws ClassCastException if o is not Comparable  
1241 \*/  
1242 private static void checkNonNullComparable(final Object o, final DataElement dataElement) {  
1243 if (o == null) {  
1244 throw new NullPointerException(dataElement + " cannot be null");  
1245 }  
1246 if (!(o instanceof Comparable)) {  
1247 throw new ClassCastException(dataElement + " must be Comparable");  
1248 }  
1249 }  
1250  
1251 /\*\*  
1252 \* check a key for validity (non-null and implements Comparable)  
1253 \*  
1254 \* @param key the key to be checked  
1255 \*  
1256 \* @throws NullPointerException if key is null  
1257 \* @throws ClassCastException if key is not Comparable  
1258 \*/  
1259 private static void checkKey(final Object key) {  
1260 checkNonNullComparable(key, KEY);  
1261 }  
1262  
1263 /\*\*  
1264 \* check a value for validity (non-null and implements Comparable)  
1265 \*  
1266 \* @param value the value to be checked  
1267 \*  
1268 \* @throws NullPointerException if value is null  
1269 \* @throws ClassCastException if value is not Comparable  
1270 \*/  
1271 private static void checkValue(final Object value) {  
1272 checkNonNullComparable(value, VALUE);  
1273 }  
1274  
1275 /\*\*  
1276 \* check a key and a value for validity (non-null and implements  
1277 \* Comparable)  
1278 \*  
1279 \* @param key the key to be checked  
1280 \* @param value the value to be checked  
1281 \*  
1282 \* @throws NullPointerException if key or value is null  
1283 \* @throws ClassCastException if key or value is not Comparable  
1284 \*/  
1285 private static void checkKeyAndValue(final Object key, final Object value) {  
1286 checkKey(key);  
1287 checkValue(value);  
1288 }  
1289  
1290 /\*\*  
1291 \* increment the modification count -- used to check for  
1292 \* concurrent modification of the map through the map and through  
1293 \* an Iterator from one of its Set or Collection views  
1294 \*/  
1295 private void modify() {  
1296 modifications++;  
1297 }  
1298  
1299 /\*\*  
1300 \* bump up the size and note that the map has changed  
1301 \*/  
1302 private void grow() {  
1303 modify();  
1304 nodeCount++;  
1305 }  
1306  
1307 /\*\*  
1308 \* decrement the size and note that the map has changed  
1309 \*/  
1310 private void shrink() {  
1311 modify();  
1312 nodeCount--;  
1313 }  
1314  
1315 /\*\*  
1316 \* insert a node by its value  
1317 \*  
1318 \* @param newNode the node to be inserted  
1319 \*  
1320 \* @throws IllegalArgumentException if the node already exists  
1321 \* in the value mapping  
1322 \*/  
1323 private void insertValue(final Node<K, V> newNode) throws IllegalArgumentException {  
1324 Node<K, V> node = rootNode[VALUE.ordinal()];  
1325  
1326 while (true) {  
1327 final int cmp = compare(newNode.getValue(), node.getValue());  
1328  
1329 if (cmp == 0) {  
1330 throw new IllegalArgumentException(  
1331 "Cannot store a duplicate value (\"" + newNode.getData(VALUE) + "\") in this Map");  
1332 } else if (cmp < 0) {  
1333 if (node.getLeft(VALUE) != null) {  
1334 node = node.getLeft(VALUE);  
1335 } else {  
1336 node.setLeft(newNode, VALUE);  
1337 newNode.setParent(node, VALUE);  
1338 doRedBlackInsert(newNode, VALUE);  
1339  
1340 break;  
1341 }  
1342 } else { // cmp > 0  
1343 if (node.getRight(VALUE) != null) {  
1344 node = node.getRight(VALUE);  
1345 } else {  
1346 node.setRight(newNode, VALUE);  
1347 newNode.setParent(node, VALUE);  
1348 doRedBlackInsert(newNode, VALUE);  
1349  
1350 break;  
1351 }  
1352 }  
1353 }  
1354 }  
1355  
1356 //-----------------------------------------------------------------------  
1357 /\*\*  
1358 \* Compares for equals as per the API.  
1359 \*  
1360 \* @param obj the object to compare to  
1361 \* @param dataElement either {@link DataElement#KEY} key}  
1362 \* or the {@link DataElement#VALUE value}.  
1363 \* @return true if equal  
1364 \*/  
1365 private boolean doEquals(final Object obj, final DataElement dataElement) {  
1366 if (obj == this) {  
1367 return true;  
1368 }  
1369 if (obj instanceof Map == false) {  
1370 return false;  
1371 }  
1372 final Map<?, ?> other = (Map<?, ?>) obj;  
1373 if (other.size() != size()) {  
1374 return false;  
1375 }  
1376  
1377 if (nodeCount > 0) {  
1378 try {  
1379 for (final MapIterator<?, ?> it = getMapIterator(dataElement); it.hasNext(); ) {  
1380 final Object key = it.next();  
1381 final Object value = it.getValue();  
1382 if (value.equals(other.get(key)) == false) {  
1383 return false;  
1384 }  
1385 }  
1386 } catch (final ClassCastException ex) {  
1387 return false;  
1388 } catch (final NullPointerException ex) {  
1389 return false;  
1390 }  
1391 }  
1392 return true;  
1393 }  
1394  
1395 /\*\*  
1396 \* Gets the hash code value for this map as per the API.  
1397 \*  
1398 \* @param dataElement either {@link DataElement#KEY} key}  
1399 \* or the {@link DataElement#VALUE value}.  
1400 \* @return the hash code value for this map  
1401 \*/  
1402 private int doHashCode(final DataElement dataElement) {  
1403 int total = 0;  
1404 if (nodeCount > 0) {  
1405 for (final MapIterator<?, ?> it = getMapIterator(dataElement); it.hasNext(); ) {  
1406 final Object key = it.next();  
1407 final Object value = it.getValue();  
1408 total += key.hashCode() ^ value.hashCode();  
1409 }  
1410 }  
1411 return total;  
1412 }  
1413  
1414 /\*\*  
1415 \* Gets the string form of this map as per AbstractMap.  
1416 \*  
1417 \* @param dataElement either {@link DataElement#KEY} key}  
1418 \* or the {@link DataElement#VALUE value}.  
1419 \* @return the string form of this map  
1420 \*/  
1421 private String doToString(final DataElement dataElement) {  
1422 if (nodeCount == 0) {  
1423 return "{}";  
1424 }  
1425 final StringBuilder buf = new StringBuilder(nodeCount \* 32);  
1426 buf.append('{');  
1427 final MapIterator<?, ?> it = getMapIterator(dataElement);  
1428 boolean hasNext = it.hasNext();  
1429 while (hasNext) {  
1430 final Object key = it.next();  
1431 final Object value = it.getValue();  
1432 buf.append(key == this ? "(this Map)" : key)  
1433 .append('=')  
1434 .append(value == this ? "(this Map)" : value);  
1435  
1436 hasNext = it.hasNext();  
1437 if (hasNext) {  
1438 buf.append(", ");  
1439 }  
1440 }  
1441  
1442 buf.append('}');  
1443 return buf.toString();  
1444 }  
1445  
1446 private MapIterator<?, ?> getMapIterator(final DataElement dataElement) {  
1447 switch (dataElement) {  
1448 case KEY:  
1449 return new ViewMapIterator(KEY);  
1450 case VALUE:  
1451 return new InverseViewMapIterator(VALUE);  
1452 default:  
1453 throw new IllegalArgumentException();  
1454 }  
1455 }  
1456  
1457 /\*\*  
1458 \* Reads the content of the stream.  
1459 \*  
1460 \* @param stream the input stream  
1461 \* @throws IOException if an error occurs while reading from the stream  
1462 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
1463 \*/  
1464 @SuppressWarnings("unchecked") // This will fail at runtime if the stream is incorrect  
1465 private void readObject(final ObjectInputStream stream) throws IOException, ClassNotFoundException{  
1466 stream.defaultReadObject();  
1467 rootNode = new Node[2];  
1468 final int size = stream.readInt();  
1469 for(int i = 0; i < size; i++){  
1470 final K k =(K) stream.readObject();  
1471 final V v =(V) stream.readObject();  
1472 put(k, v);  
1473 }  
1474 }  
1475  
1476 /\*\*  
1477 \* Writes the content to the stream for serialization.  
1478 \*  
1479 \* @param stream the output stream  
1480 \* @throws IOException if an error occurs while writing to the stream  
1481 \*/  
1482 private void writeObject(final ObjectOutputStream stream) throws IOException{  
1483 stream.defaultWriteObject();  
1484 stream.writeInt(this.size());  
1485 for (final Entry<K, V> entry : entrySet()) {  
1486 stream.writeObject(entry.getKey());  
1487 stream.writeObject(entry.getValue());  
1488 }  
1489 }  
1490  
1491 //-----------------------------------------------------------------------  
1492 /\*\*  
1493 \* A view of this map.  
1494 \*/  
1495 abstract class View<E> extends AbstractSet<E> {  
1496  
1497 /\*\* Whether to return KEY or VALUE order. \*/  
1498 final DataElement orderType;  
1499  
1500 /\*\*  
1501 \* Constructor.  
1502 \* @param orderType the KEY or VALUE int for the order  
1503 \*/  
1504 View(final DataElement orderType) {  
1505 super();  
1506 this.orderType = orderType;  
1507 }  
1508  
1509 @Override  
1510 public int size() {  
1511 return TreeBidiMap.this.size();  
1512 }  
1513  
1514 @Override  
1515 public void clear() {  
1516 TreeBidiMap.this.clear();  
1517 }  
1518 }  
1519  
1520 class KeyView extends View<K> {  
1521  
1522 /\*\*  
1523 \* Create a new TreeBidiMap.KeyView.  
1524 \*/  
1525 public KeyView(final DataElement orderType) {  
1526 super(orderType);  
1527 }  
1528  
1529 @Override  
1530 public Iterator<K> iterator() {  
1531 return new ViewMapIterator(orderType);  
1532 }  
1533  
1534 @Override  
1535 public boolean contains(final Object obj) {  
1536 checkNonNullComparable(obj, KEY);  
1537 return lookupKey(obj) != null;  
1538 }  
1539  
1540 @Override  
1541 public boolean remove(final Object o) {  
1542 return doRemoveKey(o) != null;  
1543 }  
1544  
1545 }  
1546  
1547 class ValueView extends View<V> {  
1548  
1549 /\*\*  
1550 \* Create a new TreeBidiMap.ValueView.  
1551 \*/  
1552 public ValueView(final DataElement orderType) {  
1553 super(orderType);  
1554 }  
1555  
1556 @Override  
1557 public Iterator<V> iterator() {  
1558 return new InverseViewMapIterator(orderType);  
1559 }  
1560  
1561 @Override  
1562 public boolean contains(final Object obj) {  
1563 checkNonNullComparable(obj, VALUE);  
1564 return lookupValue(obj) != null;  
1565 }  
1566  
1567 @Override  
1568 public boolean remove(final Object o) {  
1569 return doRemoveValue(o) != null;  
1570 }  
1571  
1572 }  
1573  
1574 /\*\*  
1575 \* A view of this map.  
1576 \*/  
1577 class EntryView extends View<Map.Entry<K, V>> {  
1578  
1579 EntryView() {  
1580 super(KEY);  
1581 }  
1582  
1583 @Override  
1584 public boolean contains(final Object obj) {  
1585 if (obj instanceof Map.Entry == false) {  
1586 return false;  
1587 }  
1588 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
1589 final Object value = entry.getValue();  
1590 final Node<K, V> node = lookupKey(entry.getKey());  
1591 return node != null && node.getValue().equals(value);  
1592 }  
1593  
1594 @Override  
1595 public boolean remove(final Object obj) {  
1596 if (obj instanceof Map.Entry == false) {  
1597 return false;  
1598 }  
1599 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
1600 final Object value = entry.getValue();  
1601 final Node<K, V> node = lookupKey(entry.getKey());  
1602 if (node != null && node.getValue().equals(value)) {  
1603 doRedBlackDelete(node);  
1604 return true;  
1605 }  
1606 return false;  
1607 }  
1608  
1609 @Override  
1610 public Iterator<Map.Entry<K, V>> iterator() {  
1611 return new ViewMapEntryIterator();  
1612 }  
1613 }  
1614  
1615 /\*\*  
1616 \* A view of this map.  
1617 \*/  
1618 class InverseEntryView extends View<Map.Entry<V, K>> {  
1619  
1620 InverseEntryView() {  
1621 super(VALUE);  
1622 }  
1623  
1624 @Override  
1625 public boolean contains(final Object obj) {  
1626 if (obj instanceof Map.Entry == false) {  
1627 return false;  
1628 }  
1629 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
1630 final Object value = entry.getValue();  
1631 final Node<K, V> node = lookupValue(entry.getKey());  
1632 return node != null && node.getKey().equals(value);  
1633 }  
1634  
1635 @Override  
1636 public boolean remove(final Object obj) {  
1637 if (obj instanceof Map.Entry == false) {  
1638 return false;  
1639 }  
1640 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
1641 final Object value = entry.getValue();  
1642 final Node<K, V> node = lookupValue(entry.getKey());  
1643 if (node != null && node.getKey().equals(value)) {  
1644 doRedBlackDelete(node);  
1645 return true;  
1646 }  
1647 return false;  
1648 }  
1649  
1650 @Override  
1651 public Iterator<Map.Entry<V, K>> iterator() {  
1652 return new InverseViewMapEntryIterator();  
1653 }  
1654 }  
1655  
1656 //-----------------------------------------------------------------------  
1657 /\*\*  
1658 \* An iterator over the map.  
1659 \*/  
1660 abstract class ViewIterator {  
1661  
1662 /\*\* Whether to return KEY or VALUE order. \*/  
1663 private final DataElement orderType;  
1664 /\*\* The last node returned by the iterator. \*/  
1665 Node<K, V> lastReturnedNode;  
1666 /\*\* The next node to be returned by the iterator. \*/  
1667 private Node<K, V> nextNode;  
1668 /\*\* The previous node in the sequence returned by the iterator. \*/  
1669 private Node<K, V> previousNode;  
1670 /\*\* The modification count. \*/  
1671 private int expectedModifications;  
1672  
1673 /\*\*  
1674 \* Constructor.  
1675 \* @param orderType the KEY or VALUE int for the order  
1676 \*/  
1677 ViewIterator(final DataElement orderType) {  
1678 super();  
1679 this.orderType = orderType;  
1680 expectedModifications = modifications;  
1681 nextNode = leastNode(rootNode[orderType.ordinal()], orderType);  
1682 lastReturnedNode = null;  
1683 previousNode = null;  
1684 }  
1685  
1686 public final boolean hasNext() {  
1687 return nextNode != null;  
1688 }  
1689  
1690 protected Node<K, V> navigateNext() {  
1691 if (nextNode == null) {  
1692 throw new NoSuchElementException();  
1693 }  
1694 if (modifications != expectedModifications) {  
1695 throw new ConcurrentModificationException();  
1696 }  
1697 lastReturnedNode = nextNode;  
1698 previousNode = nextNode;  
1699 nextNode = nextGreater(nextNode, orderType);  
1700 return lastReturnedNode;  
1701 }  
1702  
1703 public boolean hasPrevious() {  
1704 return previousNode != null;  
1705 }  
1706  
1707 protected Node<K, V> navigatePrevious() {  
1708 if (previousNode == null) {  
1709 throw new NoSuchElementException();  
1710 }  
1711 if (modifications != expectedModifications) {  
1712 throw new ConcurrentModificationException();  
1713 }  
1714 nextNode = lastReturnedNode;  
1715 if (nextNode == null) {  
1716 nextNode = nextGreater(previousNode, orderType);  
1717 }  
1718 lastReturnedNode = previousNode;  
1719 previousNode = nextSmaller(previousNode, orderType);  
1720 return lastReturnedNode;  
1721 }  
1722  
1723 public final void remove() {  
1724 if (lastReturnedNode == null) {  
1725 throw new IllegalStateException();  
1726 }  
1727 if (modifications != expectedModifications) {  
1728 throw new ConcurrentModificationException();  
1729 }  
1730 doRedBlackDelete(lastReturnedNode);  
1731 expectedModifications++;  
1732 lastReturnedNode = null;  
1733 if (nextNode == null) {  
1734 previousNode = greatestNode(rootNode[orderType.ordinal()], orderType);  
1735 } else {  
1736 previousNode = nextSmaller(nextNode, orderType);  
1737 }  
1738 }  
1739 }  
1740  
1741 //-----------------------------------------------------------------------  
1742 /\*\*  
1743 \* An iterator over the map.  
1744 \*/  
1745 class ViewMapIterator extends ViewIterator implements OrderedMapIterator<K, V> {  
1746  
1747 /\*\*  
1748 \* Constructor.  
1749 \*/  
1750 ViewMapIterator(final DataElement orderType) {  
1751 super(orderType);  
1752 }  
1753  
1754 @Override  
1755 public K getKey() {  
1756 if (lastReturnedNode == null) {  
1757 throw new IllegalStateException(  
1758 "Iterator getKey() can only be called after next() and before remove()");  
1759 }  
1760 return lastReturnedNode.getKey();  
1761 }  
1762  
1763 @Override  
1764 public V getValue() {  
1765 if (lastReturnedNode == null) {  
1766 throw new IllegalStateException(  
1767 "Iterator getValue() can only be called after next() and before remove()");  
1768 }  
1769 return lastReturnedNode.getValue();  
1770 }  
1771  
1772 @Override  
1773 public V setValue(final V obj) {  
1774 throw new UnsupportedOperationException();  
1775 }  
1776  
1777 @Override  
1778 public K next() {  
1779 return navigateNext().getKey();  
1780 }  
1781  
1782 @Override  
1783 public K previous() {  
1784 return navigatePrevious().getKey();  
1785 }  
1786 }  
1787  
1788 /\*\*  
1789 \* An iterator over the map.  
1790 \*/  
1791 class InverseViewMapIterator extends ViewIterator implements OrderedMapIterator<V, K> {  
1792  
1793 /\*\*  
1794 \* Create a new TreeBidiMap.InverseViewMapIterator.  
1795 \*/  
1796 public InverseViewMapIterator(final DataElement orderType) {  
1797 super(orderType);  
1798 }  
1799  
1800 @Override  
1801 public V getKey() {  
1802 if (lastReturnedNode == null) {  
1803 throw new IllegalStateException(  
1804 "Iterator getKey() can only be called after next() and before remove()");  
1805 }  
1806 return lastReturnedNode.getValue();  
1807 }  
1808  
1809 @Override  
1810 public K getValue() {  
1811 if (lastReturnedNode == null) {  
1812 throw new IllegalStateException(  
1813 "Iterator getValue() can only be called after next() and before remove()");  
1814 }  
1815 return lastReturnedNode.getKey();  
1816 }  
1817  
1818 @Override  
1819 public K setValue(final K obj) {  
1820 throw new UnsupportedOperationException();  
1821 }  
1822  
1823 @Override  
1824 public V next() {  
1825 return navigateNext().getValue();  
1826 }  
1827  
1828 @Override  
1829 public V previous() {  
1830 return navigatePrevious().getValue();  
1831 }  
1832 }  
1833  
1834 /\*\*  
1835 \* An iterator over the map entries.  
1836 \*/  
1837 class ViewMapEntryIterator extends ViewIterator implements OrderedIterator<Map.Entry<K, V>> {  
1838  
1839 /\*\*  
1840 \* Constructor.  
1841 \*/  
1842 ViewMapEntryIterator() {  
1843 super(KEY);  
1844 }  
1845  
1846 @Override  
1847 public Map.Entry<K, V> next() {  
1848 return navigateNext();  
1849 }  
1850  
1851 @Override  
1852 public Map.Entry<K, V> previous() {  
1853 return navigatePrevious();  
1854 }  
1855 }  
1856  
1857 /\*\*  
1858 \* An iterator over the inverse map entries.  
1859 \*/  
1860 class InverseViewMapEntryIterator extends ViewIterator implements OrderedIterator<Map.Entry<V, K>> {  
1861  
1862 /\*\*  
1863 \* Constructor.  
1864 \*/  
1865 InverseViewMapEntryIterator() {  
1866 super(VALUE);  
1867 }  
1868  
1869 @Override  
1870 public Map.Entry<V, K> next() {  
1871 return createEntry(navigateNext());  
1872 }  
1873  
1874 @Override  
1875 public Map.Entry<V, K> previous() {  
1876 return createEntry(navigatePrevious());  
1877 }  
1878  
1879 private Map.Entry<V, K> createEntry(final Node<K, V> node) {  
1880 return new UnmodifiableMapEntry<>(node.getValue(), node.getKey());  
1881 }  
1882 }  
1883  
1884 //-----------------------------------------------------------------------  
1885 //-----------------------------------------------------------------------  
1886 /\*\*  
1887 \* A node used to store the data.  
1888 \*/  
1889 static class Node<K extends Comparable<K>, V extends Comparable<V>> implements Map.Entry<K, V>, KeyValue<K, V> {  
1890  
1891 private final K key;  
1892 private final V value;  
1893 private final Node<K, V>[] leftNode;  
1894 private final Node<K, V>[] rightNode;  
1895 private final Node<K, V>[] parentNode;  
1896 private final boolean[] blackColor;  
1897 private int hashcodeValue;  
1898 private boolean calculatedHashCode;  
1899  
1900 /\*\*  
1901 \* Make a new cell with given key and value, and with null  
1902 \* links, and black (true) colors.  
1903 \*  
1904 \* @param key the key of this node  
1905 \* @param value the value of this node  
1906 \*/  
1907 @SuppressWarnings("unchecked")  
1908 Node(final K key, final V value) {  
1909 super();  
1910 this.key = key;  
1911 this.value = value;  
1912 leftNode = new Node[2];  
1913 rightNode = new Node[2];  
1914 parentNode = new Node[2];  
1915 blackColor = new boolean[] { true, true };  
1916 calculatedHashCode = false;  
1917 }  
1918  
1919 private Object getData(final DataElement dataElement) {  
1920 switch (dataElement) {  
1921 case KEY:  
1922 return getKey();  
1923 case VALUE:  
1924 return getValue();  
1925 default:  
1926 throw new IllegalArgumentException();  
1927 }  
1928 }  
1929  
1930 private void setLeft(final Node<K, V> node, final DataElement dataElement) {  
1931 leftNode[dataElement.ordinal()] = node;  
1932 }  
1933  
1934 private Node<K, V> getLeft(final DataElement dataElement) {  
1935 return leftNode[dataElement.ordinal()];  
1936 }  
1937  
1938 private void setRight(final Node<K, V> node, final DataElement dataElement) {  
1939 rightNode[dataElement.ordinal()] = node;  
1940 }  
1941  
1942 private Node<K, V> getRight(final DataElement dataElement) {  
1943 return rightNode[dataElement.ordinal()];  
1944 }  
1945  
1946 /\*\*  
1947 \* Set this node's parent node.  
1948 \*  
1949 \* @param node the new parent node  
1950 \* @param dataElement either {@link DataElement#KEY} key}  
1951 \* or the {@link DataElement#VALUE value}.  
1952 \*/  
1953 private void setParent(final Node<K, V> node, final DataElement dataElement) {  
1954 parentNode[dataElement.ordinal()] = node;  
1955 }  
1956  
1957 /\*\*  
1958 \* Get the parent node.  
1959 \*  
1960 \* @param dataElement either {@link DataElement#KEY} key}  
1961 \* or the {@link DataElement#VALUE value}.  
1962 \* @return the parent node, may be null  
1963 \*/  
1964 private Node<K, V> getParent(final DataElement dataElement) {  
1965 return parentNode[dataElement.ordinal()];  
1966 }  
1967  
1968 /\*\*  
1969 \* Exchange colors with another node.  
1970 \*  
1971 \* @param node the node to swap with  
1972 \* @param dataElement either {@link DataElement#KEY} key}  
1973 \* or the {@link DataElement#VALUE value}.  
1974 \*/  
1975 private void swapColors(final Node<K, V> node, final DataElement dataElement) {  
1976 // Swap colors -- old hacker's trick  
1977 blackColor[dataElement.ordinal()] ^= node.blackColor[dataElement.ordinal()];  
1978 node.blackColor[dataElement.ordinal()] ^= blackColor[dataElement.ordinal()];  
1979 blackColor[dataElement.ordinal()] ^= node.blackColor[dataElement.ordinal()];  
1980 }  
1981  
1982 /\*\*  
1983 \* Is this node black?  
1984 \*  
1985 \* @param dataElement either {@link DataElement#KEY} key}  
1986 \* or the {@link DataElement#VALUE value}.  
1987 \* @return true if black (which is represented as a true boolean)  
1988 \*/  
1989 private boolean isBlack(final DataElement dataElement) {  
1990 return blackColor[dataElement.ordinal()];  
1991 }  
1992  
1993 /\*\*  
1994 \* Is this node red?  
1995 \*  
1996 \* @param dataElement either {@link DataElement#KEY} key}  
1997 \* or the {@link DataElement#VALUE value}.  
1998 \* @return true if non-black  
1999 \*/  
2000 private boolean isRed(final DataElement dataElement) {  
2001 return !blackColor[dataElement.ordinal()];  
2002 }  
2003  
2004 /\*\*  
2005 \* Make this node black.  
2006 \*  
2007 \* @param dataElement either {@link DataElement#KEY} key}  
2008 \* or the {@link DataElement#VALUE value}.  
2009 \*/  
2010 private void setBlack(final DataElement dataElement) {  
2011 blackColor[dataElement.ordinal()] = true;  
2012 }  
2013  
2014 /\*\*  
2015 \* Make this node red.  
2016 \*  
2017 \* @param dataElement either {@link DataElement#KEY} key}  
2018 \* or the {@link DataElement#VALUE value}.  
2019 \*/  
2020 private void setRed(final DataElement dataElement) {  
2021 blackColor[dataElement.ordinal()] = false;  
2022 }  
2023  
2024 /\*\*  
2025 \* Make this node the same color as another  
2026 \*  
2027 \* @param node the node whose color we're adopting  
2028 \* @param dataElement either {@link DataElement#KEY} key}  
2029 \* or the {@link DataElement#VALUE value}.  
2030 \*/  
2031 private void copyColor(final Node<K, V> node, final DataElement dataElement) {  
2032 blackColor[dataElement.ordinal()] = node.blackColor[dataElement.ordinal()];  
2033 }  
2034  
2035 private boolean isLeftChild(final DataElement dataElement) {  
2036 return parentNode[dataElement.ordinal()] != null  
2037 && parentNode[dataElement.ordinal()].leftNode[dataElement.ordinal()] == this;  
2038 }  
2039  
2040 private boolean isRightChild(final DataElement dataElement) {  
2041 return parentNode[dataElement.ordinal()] != null  
2042 && parentNode[dataElement.ordinal()].rightNode[dataElement.ordinal()] == this;  
2043 }  
2044  
2045 //-------------------------------------------------------------------  
2046 /\*\*  
2047 \* Gets the key.  
2048 \*  
2049 \* @return the key corresponding to this entry.  
2050 \*/  
2051 @Override  
2052 public K getKey() {  
2053 return key;  
2054 }  
2055  
2056 /\*\*  
2057 \* Gets the value.  
2058 \*  
2059 \* @return the value corresponding to this entry.  
2060 \*/  
2061 @Override  
2062 public V getValue() {  
2063 return value;  
2064 }  
2065  
2066 /\*\*  
2067 \* Optional operation that is not permitted in this implementation  
2068 \*  
2069 \* @param ignored this parameter is ignored.  
2070 \* @return does not return  
2071 \* @throws UnsupportedOperationException always  
2072 \*/  
2073 @Override  
2074 public V setValue(final V ignored) throws UnsupportedOperationException {  
2075 throw new UnsupportedOperationException("Map.Entry.setValue is not supported");  
2076 }  
2077  
2078 /\*\*  
2079 \* Compares the specified object with this entry for equality.  
2080 \* Returns true if the given object is also a map entry and  
2081 \* the two entries represent the same mapping.  
2082 \*  
2083 \* @param obj the object to be compared for equality with this entry.  
2084 \* @return true if the specified object is equal to this entry.  
2085 \*/  
2086 @Override  
2087 public boolean equals(final Object obj) {  
2088 if (obj == this) {  
2089 return true;  
2090 }  
2091 if (!(obj instanceof Map.Entry)) {  
2092 return false;  
2093 }  
2094 final Map.Entry<?, ?> e = (Map.Entry<?, ?>) obj;  
2095 return getKey().equals(e.getKey()) && getValue().equals(e.getValue());  
2096 }  
2097  
2098 /\*\*  
2099 \* @return the hash code value for this map entry.  
2100 \*/  
2101 @Override  
2102 public int hashCode() {  
2103 if (!calculatedHashCode) {  
2104 hashcodeValue = getKey().hashCode() ^ getValue().hashCode();  
2105 calculatedHashCode = true;  
2106 }  
2107 return hashcodeValue;  
2108 }  
2109 }  
2110  
2111 //-----------------------------------------------------------------------  
2112 /\*\*  
2113 \* The inverse map implementation.  
2114 \*/  
2115 class Inverse implements OrderedBidiMap<V, K> {  
2116  
2117 /\*\* Store the keySet once created. \*/  
2118 private Set<V> inverseKeySet;  
2119 /\*\* Store the valuesSet once created. \*/  
2120 private Set<K> inverseValuesSet;  
2121 /\*\* Store the entrySet once created. \*/  
2122 private Set<Map.Entry<V, K>> inverseEntrySet;  
2123  
2124 @Override  
2125 public int size() {  
2126 return TreeBidiMap.this.size();  
2127 }  
2128  
2129 @Override  
2130 public boolean isEmpty() {  
2131 return TreeBidiMap.this.isEmpty();  
2132 }  
2133  
2134 @Override  
2135 public K get(final Object key) {  
2136 return TreeBidiMap.this.getKey(key);  
2137 }  
2138  
2139 @Override  
2140 public V getKey(final Object value) {  
2141 return TreeBidiMap.this.get(value);  
2142 }  
2143  
2144 @Override  
2145 public boolean containsKey(final Object key) {  
2146 return TreeBidiMap.this.containsValue(key);  
2147 }  
2148  
2149 @Override  
2150 public boolean containsValue(final Object value) {  
2151 return TreeBidiMap.this.containsKey(value);  
2152 }  
2153  
2154 @Override  
2155 public V firstKey() {  
2156 if (TreeBidiMap.this.nodeCount == 0) {  
2157 throw new NoSuchElementException("Map is empty");  
2158 }  
2159 return leastNode(TreeBidiMap.this.rootNode[VALUE.ordinal()], VALUE).getValue();  
2160 }  
2161  
2162 @Override  
2163 public V lastKey() {  
2164 if (TreeBidiMap.this.nodeCount == 0) {  
2165 throw new NoSuchElementException("Map is empty");  
2166 }  
2167 return greatestNode(TreeBidiMap.this.rootNode[VALUE.ordinal()], VALUE).getValue();  
2168 }  
2169  
2170 @Override  
2171 public V nextKey(final V key) {  
2172 checkKey(key);  
2173 final Node<K, V> node = nextGreater(TreeBidiMap.this.<V>lookup(key, VALUE), VALUE);  
2174 return node == null ? null : node.getValue();  
2175 }  
2176  
2177 @Override  
2178 public V previousKey(final V key) {  
2179 checkKey(key);  
2180 final Node<K, V> node = TreeBidiMap.this.nextSmaller(TreeBidiMap.this.<V>lookup(key, VALUE), VALUE);  
2181 return node == null ? null : node.getValue();  
2182 }  
2183  
2184 @Override  
2185 public K put(final V key, final K value) {  
2186 final K result = get(key);  
2187 TreeBidiMap.this.doPut(value, key);  
2188 return result;  
2189 }  
2190  
2191 @Override  
2192 public void putAll(final Map<? extends V, ? extends K> map) {  
2193 for (final Map.Entry<? extends V, ? extends K> e : map.entrySet()) {  
2194 put(e.getKey(), e.getValue());  
2195 }  
2196 }  
2197  
2198 @Override  
2199 public K remove(final Object key) {  
2200 return TreeBidiMap.this.removeValue(key);  
2201 }  
2202  
2203 @Override  
2204 public V removeValue(final Object value) {  
2205 return TreeBidiMap.this.remove(value);  
2206 }  
2207  
2208 @Override  
2209 public void clear() {  
2210 TreeBidiMap.this.clear();  
2211 }  
2212  
2213 @Override  
2214 public Set<V> keySet() {  
2215 if (inverseKeySet == null) {  
2216 inverseKeySet = new ValueView(VALUE);  
2217 }  
2218 return inverseKeySet;  
2219 }  
2220  
2221 @Override  
2222 public Set<K> values() {  
2223 if (inverseValuesSet == null) {  
2224 inverseValuesSet = new KeyView(VALUE);  
2225 }  
2226 return inverseValuesSet;  
2227 }  
2228  
2229 @Override  
2230 public Set<Map.Entry<V, K>> entrySet() {  
2231 if (inverseEntrySet == null) {  
2232 inverseEntrySet = new InverseEntryView();  
2233 }  
2234 return inverseEntrySet;  
2235 }  
2236  
2237 @Override  
2238 public OrderedMapIterator<V, K> mapIterator() {  
2239 if (isEmpty()) {  
2240 return EmptyOrderedMapIterator.<V, K>emptyOrderedMapIterator();  
2241 }  
2242 return new InverseViewMapIterator(VALUE);  
2243 }  
2244  
2245 @Override  
2246 public OrderedBidiMap<K, V> inverseBidiMap() {  
2247 return TreeBidiMap.this;  
2248 }  
2249  
2250 @Override  
2251 public boolean equals(final Object obj) {  
2252 return TreeBidiMap.this.doEquals(obj, DataElement.VALUE);  
2253 }  
2254  
2255 @Override  
2256 public int hashCode() {  
2257 return TreeBidiMap.this.doHashCode(DataElement.VALUE);  
2258 }  
2259  
2260 @Override  
2261 public String toString() {  
2262 return TreeBidiMap.this.doToString(DataElement.VALUE);  
2263 }  
2264 }  
2265  
2266}