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.list;  
018  
019import java.util.AbstractList;  
020import java.util.ArrayDeque;  
021import java.util.Collection;  
022import java.util.ConcurrentModificationException;  
023import java.util.Deque;  
024import java.util.Iterator;  
025import java.util.ListIterator;  
026import java.util.NoSuchElementException;  
027  
028import org.apache.commons.collections4.OrderedIterator;  
029  
030/\*\*  
031 \* A <code>List</code> implementation that is optimised for fast insertions and  
032 \* removals at any index in the list.  
033 \* <p>  
034 \* This list implementation utilises a tree structure internally to ensure that  
035 \* all insertions and removals are O(log n). This provides much faster performance  
036 \* than both an <code>ArrayList</code> and a <code>LinkedList</code> where elements  
037 \* are inserted and removed repeatedly from anywhere in the list.  
038 \* </p>  
039 \* <p>  
040 \* The following relative performance statistics are indicative of this class:  
041 \* </p>  
042 \* <pre>  
043 \* get add insert iterate remove  
044 \* TreeList 3 5 1 2 1  
045 \* ArrayList 1 1 40 1 40  
046 \* LinkedList 5800 1 350 2 325  
047 \* </pre>  
048 \* <p>  
049 \* <code>ArrayList</code> is a good general purpose list implementation.  
050 \* It is faster than <code>TreeList</code> for most operations except inserting  
051 \* and removing in the middle of the list. <code>ArrayList</code> also uses less  
052 \* memory as <code>TreeList</code> uses one object per entry.  
053 \* </p>  
054 \* <p>  
055 \* <code>LinkedList</code> is rarely a good choice of implementation.  
056 \* <code>TreeList</code> is almost always a good replacement for it, although it  
057 \* does use slightly more memory.  
058 \* </p>  
059 \*  
060 \* @since 3.1  
061 \*/  
062public class TreeList<E> extends AbstractList<E> {  
063// add; toArray; iterator; insert; get; indexOf; remove  
064// TreeList = 1260;7360;3080; 160; 170;3400; 170;  
065// ArrayList = 220;1480;1760; 6870; 50;1540; 7200;  
066// LinkedList = 270;7360;3350;55860;290720;2910;55200;  
067  
068 /\*\* The root node in the AVL tree \*/  
069 private AVLNode<E> root;  
070  
071 /\*\* The current size of the list \*/  
072 private int size;  
073  
074 //-----------------------------------------------------------------------  
075 /\*\*  
076 \* Constructs a new empty list.  
077 \*/  
078 public TreeList() {  
079 super();  
080 }  
081  
082 /\*\*  
083 \* Constructs a new empty list that copies the specified collection.  
084 \*  
085 \* @param coll the collection to copy  
086 \* @throws NullPointerException if the collection is null  
087 \*/  
088 public TreeList(final Collection<? extends E> coll) {  
089 super();  
090 if (!coll.isEmpty()) {  
091 root = new AVLNode<>(coll);  
092 size = coll.size();  
093 }  
094 }  
095  
096 //-----------------------------------------------------------------------  
097 /\*\*  
098 \* Gets the element at the specified index.  
099 \*  
100 \* @param index the index to retrieve  
101 \* @return the element at the specified index  
102 \*/  
103 @Override  
104 public E get(final int index) {  
105 checkInterval(index, 0, size() - 1);  
106 return root.get(index).getValue();  
107 }  
108  
109 /\*\*  
110 \* Gets the current size of the list.  
111 \*  
112 \* @return the current size  
113 \*/  
114 @Override  
115 public int size() {  
116 return size;  
117 }  
118  
119 /\*\*  
120 \* Gets an iterator over the list.  
121 \*  
122 \* @return an iterator over the list  
123 \*/  
124 @Override  
125 public Iterator<E> iterator() {  
126 // override to go 75% faster  
127 return listIterator(0);  
128 }  
129  
130 /\*\*  
131 \* Gets a ListIterator over the list.  
132 \*  
133 \* @return the new iterator  
134 \*/  
135 @Override  
136 public ListIterator<E> listIterator() {  
137 // override to go 75% faster  
138 return listIterator(0);  
139 }  
140  
141 /\*\*  
142 \* Gets a ListIterator over the list.  
143 \*  
144 \* @param fromIndex the index to start from  
145 \* @return the new iterator  
146 \*/  
147 @Override  
148 public ListIterator<E> listIterator(final int fromIndex) {  
149 // override to go 75% faster  
150 // cannot use EmptyIterator as iterator.add() must work  
151 checkInterval(fromIndex, 0, size());  
152 return new TreeListIterator<>(this, fromIndex);  
153 }  
154  
155 /\*\*  
156 \* Searches for the index of an object in the list.  
157 \*  
158 \* @param object the object to search  
159 \* @return the index of the object, -1 if not found  
160 \*/  
161 @Override  
162 public int indexOf(final Object object) {  
163 // override to go 75% faster  
164 if (root == null) {  
165 return -1;  
166 }  
167 return root.indexOf(object, root.relativePosition);  
168 }  
169  
170 /\*\*  
171 \* Searches for the presence of an object in the list.  
172 \*  
173 \* @param object the object to check  
174 \* @return true if the object is found  
175 \*/  
176 @Override  
177 public boolean contains(final Object object) {  
178 return indexOf(object) >= 0;  
179 }  
180  
181 /\*\*  
182 \* Converts the list into an array.  
183 \*  
184 \* @return the list as an array  
185 \*/  
186 @Override  
187 public Object[] toArray() {  
188 // override to go 20% faster  
189 final Object[] array = new Object[size()];  
190 if (root != null) {  
191 root.toArray(array, root.relativePosition);  
192 }  
193 return array;  
194 }  
195  
196 //-----------------------------------------------------------------------  
197 /\*\*  
198 \* Adds a new element to the list.  
199 \*  
200 \* @param index the index to add before  
201 \* @param obj the element to add  
202 \*/  
203 @Override  
204 public void add(final int index, final E obj) {  
205 modCount++;  
206 checkInterval(index, 0, size());  
207 if (root == null) {  
208 root = new AVLNode<>(index, obj, null, null);  
209 } else {  
210 root = root.insert(index, obj);  
211 }  
212 size++;  
213 }  
214  
215 /\*\*  
216 \* Appends all of the elements in the specified collection to the end of this list,  
217 \* in the order that they are returned by the specified collection's Iterator.  
218 \* <p>  
219 \* This method runs in O(n + log m) time, where m is  
220 \* the size of this list and n is the size of {@code c}.  
221 \*  
222 \* @param c the collection to be added to this list  
223 \* @return {@code true} if this list changed as a result of the call  
224 \* @throws NullPointerException {@inheritDoc}  
225 \*/  
226 @Override  
227 public boolean addAll(final Collection<? extends E> c) {  
228 if (c.isEmpty()) {  
229 return false;  
230 }  
231 modCount += c.size();  
232 final AVLNode<E> cTree = new AVLNode<>(c);  
233 root = root == null ? cTree : root.addAll(cTree, size);  
234 size += c.size();  
235 return true;  
236 }  
237  
238 /\*\*  
239 \* Sets the element at the specified index.  
240 \*  
241 \* @param index the index to set  
242 \* @param obj the object to store at the specified index  
243 \* @return the previous object at that index  
244 \* @throws IndexOutOfBoundsException if the index is invalid  
245 \*/  
246 @Override  
247 public E set(final int index, final E obj) {  
248 checkInterval(index, 0, size() - 1);  
249 final AVLNode<E> node = root.get(index);  
250 final E result = node.value;  
251 node.setValue(obj);  
252 return result;  
253 }  
254  
255 /\*\*  
256 \* Removes the element at the specified index.  
257 \*  
258 \* @param index the index to remove  
259 \* @return the previous object at that index  
260 \*/  
261 @Override  
262 public E remove(final int index) {  
263 modCount++;  
264 checkInterval(index, 0, size() - 1);  
265 final E result = get(index);  
266 root = root.remove(index);  
267 size--;  
268 return result;  
269 }  
270  
271 /\*\*  
272 \* Clears the list, removing all entries.  
273 \*/  
274 @Override  
275 public void clear() {  
276 modCount++;  
277 root = null;  
278 size = 0;  
279 }  
280  
281 //-----------------------------------------------------------------------  
282 /\*\*  
283 \* Checks whether the index is valid.  
284 \*  
285 \* @param index the index to check  
286 \* @param startIndex the first allowed index  
287 \* @param endIndex the last allowed index  
288 \* @throws IndexOutOfBoundsException if the index is invalid  
289 \*/  
290 private void checkInterval(final int index, final int startIndex, final int endIndex) {  
291 if (index < startIndex || index > endIndex) {  
292 throw new IndexOutOfBoundsException("Invalid index:" + index + ", size=" + size());  
293 }  
294 }  
295  
296 //-----------------------------------------------------------------------  
297 /\*\*  
298 \* Implements an AVLNode which keeps the offset updated.  
299 \* <p>  
300 \* This node contains the real work.  
301 \* TreeList is just there to implement {@link java.util.List}.  
302 \* The nodes don't know the index of the object they are holding. They  
303 \* do know however their position relative to their parent node.  
304 \* This allows to calculate the index of a node while traversing the tree.  
305 \* <p>  
306 \* The Faedelung calculation stores a flag for both the left and right child  
307 \* to indicate if they are a child (false) or a link as in linked list (true).  
308 \*/  
309 static class AVLNode<E> {  
310 /\*\* The left child node or the predecessor if {@link #leftIsPrevious}.\*/  
311 private AVLNode<E> left;  
312 /\*\* Flag indicating that left reference is not a subtree but the predecessor. \*/  
313 private boolean leftIsPrevious;  
314 /\*\* The right child node or the successor if {@link #rightIsNext}. \*/  
315 private AVLNode<E> right;  
316 /\*\* Flag indicating that right reference is not a subtree but the successor. \*/  
317 private boolean rightIsNext;  
318 /\*\* How many levels of left/right are below this one. \*/  
319 private int height;  
320 /\*\* The relative position, root holds absolute position. \*/  
321 private int relativePosition;  
322 /\*\* The stored element. \*/  
323 private E value;  
324  
325 /\*\*  
326 \* Constructs a new node with a relative position.  
327 \*  
328 \* @param relativePosition the relative position of the node  
329 \* @param obj the value for the node  
330 \* @param rightFollower the node with the value following this one  
331 \* @param leftFollower the node with the value leading this one  
332 \*/  
333 private AVLNode(final int relativePosition, final E obj,  
334 final AVLNode<E> rightFollower, final AVLNode<E> leftFollower) {  
335 this.relativePosition = relativePosition;  
336 value = obj;  
337 rightIsNext = true;  
338 leftIsPrevious = true;  
339 right = rightFollower;  
340 left = leftFollower;  
341 }  
342  
343 /\*\*  
344 \* Constructs a new AVL tree from a collection.  
345 \* <p>  
346 \* The collection must be nonempty.  
347 \*  
348 \* @param coll a nonempty collection  
349 \*/  
350 private AVLNode(final Collection<? extends E> coll) {  
351 this(coll.iterator(), 0, coll.size() - 1, 0, null, null);  
352 }  
353  
354 /\*\*  
355 \* Constructs a new AVL tree from a collection.  
356 \* <p>  
357 \* This is a recursive helper for {@link #AVLNode(Collection)}. A call  
358 \* to this method will construct the subtree for elements {@code start}  
359 \* through {@code end} of the collection, assuming the iterator  
360 \* {@code e} already points at element {@code start}.  
361 \*  
362 \* @param iterator an iterator over the collection, which should already point  
363 \* to the element at index {@code start} within the collection  
364 \* @param start the index of the first element in the collection that  
365 \* should be in this subtree  
366 \* @param end the index of the last element in the collection that  
367 \* should be in this subtree  
368 \* @param absolutePositionOfParent absolute position of this node's  
369 \* parent, or 0 if this node is the root  
370 \* @param prev the {@code AVLNode} corresponding to element (start - 1)  
371 \* of the collection, or null if start is 0  
372 \* @param next the {@code AVLNode} corresponding to element (end + 1)  
373 \* of the collection, or null if end is the last element of the collection  
374 \*/  
375 private AVLNode(final Iterator<? extends E> iterator, final int start, final int end,  
376 final int absolutePositionOfParent, final AVLNode<E> prev, final AVLNode<E> next) {  
377 final int mid = start + (end - start) / 2;  
378 if (start < mid) {  
379 left = new AVLNode<>(iterator, start, mid - 1, mid, prev, this);  
380 } else {  
381 leftIsPrevious = true;  
382 left = prev;  
383 }  
384 value = iterator.next();  
385 relativePosition = mid - absolutePositionOfParent;  
386 if (mid < end) {  
387 right = new AVLNode<>(iterator, mid + 1, end, mid, this, next);  
388 } else {  
389 rightIsNext = true;  
390 right = next;  
391 }  
392 recalcHeight();  
393 }  
394  
395 /\*\*  
396 \* Gets the value.  
397 \*  
398 \* @return the value of this node  
399 \*/  
400 E getValue() {  
401 return value;  
402 }  
403  
404 /\*\*  
405 \* Sets the value.  
406 \*  
407 \* @param obj the value to store  
408 \*/  
409 void setValue(final E obj) {  
410 this.value = obj;  
411 }  
412  
413 /\*\*  
414 \* Locate the element with the given index relative to the  
415 \* offset of the parent of this node.  
416 \*/  
417 AVLNode<E> get(final int index) {  
418 final int indexRelativeToMe = index - relativePosition;  
419  
420 if (indexRelativeToMe == 0) {  
421 return this;  
422 }  
423  
424 final AVLNode<E> nextNode = indexRelativeToMe < 0 ? getLeftSubTree() : getRightSubTree();  
425 if (nextNode == null) {  
426 return null;  
427 }  
428 return nextNode.get(indexRelativeToMe);  
429 }  
430  
431 /\*\*  
432 \* Locate the index that contains the specified object.  
433 \*/  
434 int indexOf(final Object object, final int index) {  
435 if (getLeftSubTree() != null) {  
436 final int result = left.indexOf(object, index + left.relativePosition);  
437 if (result != -1) {  
438 return result;  
439 }  
440 }  
441 if (value == null ? value == object : value.equals(object)) {  
442 return index;  
443 }  
444 if (getRightSubTree() != null) {  
445 return right.indexOf(object, index + right.relativePosition);  
446 }  
447 return -1;  
448 }  
449  
450 /\*\*  
451 \* Stores the node and its children into the array specified.  
452 \*  
453 \* @param array the array to be filled  
454 \* @param index the index of this node  
455 \*/  
456 void toArray(final Object[] array, final int index) {  
457 array[index] = value;  
458 if (getLeftSubTree() != null) {  
459 left.toArray(array, index + left.relativePosition);  
460 }  
461 if (getRightSubTree() != null) {  
462 right.toArray(array, index + right.relativePosition);  
463 }  
464 }  
465  
466 /\*\*  
467 \* Gets the next node in the list after this one.  
468 \*  
469 \* @return the next node  
470 \*/  
471 AVLNode<E> next() {  
472 if (rightIsNext || right == null) {  
473 return right;  
474 }  
475 return right.min();  
476 }  
477  
478 /\*\*  
479 \* Gets the node in the list before this one.  
480 \*  
481 \* @return the previous node  
482 \*/  
483 AVLNode<E> previous() {  
484 if (leftIsPrevious || left == null) {  
485 return left;  
486 }  
487 return left.max();  
488 }  
489  
490 /\*\*  
491 \* Inserts a node at the position index.  
492 \*  
493 \* @param index is the index of the position relative to the position of  
494 \* the parent node.  
495 \* @param obj is the object to be stored in the position.  
496 \*/  
497 AVLNode<E> insert(final int index, final E obj) {  
498 final int indexRelativeToMe = index - relativePosition;  
499  
500 if (indexRelativeToMe <= 0) {  
501 return insertOnLeft(indexRelativeToMe, obj);  
502 }  
503 return insertOnRight(indexRelativeToMe, obj);  
504 }  
505  
506 private AVLNode<E> insertOnLeft(final int indexRelativeToMe, final E obj) {  
507 if (getLeftSubTree() == null) {  
508 setLeft(new AVLNode<>(-1, obj, this, left), null);  
509 } else {  
510 setLeft(left.insert(indexRelativeToMe, obj), null);  
511 }  
512  
513 if (relativePosition >= 0) {  
514 relativePosition++;  
515 }  
516 final AVLNode<E> ret = balance();  
517 recalcHeight();  
518 return ret;  
519 }  
520  
521 private AVLNode<E> insertOnRight(final int indexRelativeToMe, final E obj) {  
522 if (getRightSubTree() == null) {  
523 setRight(new AVLNode<>(+1, obj, right, this), null);  
524 } else {  
525 setRight(right.insert(indexRelativeToMe, obj), null);  
526 }  
527 if (relativePosition < 0) {  
528 relativePosition--;  
529 }  
530 final AVLNode<E> ret = balance();  
531 recalcHeight();  
532 return ret;  
533 }  
534  
535 //-----------------------------------------------------------------------  
536 /\*\*  
537 \* Gets the left node, returning null if its a faedelung.  
538 \*/  
539 private AVLNode<E> getLeftSubTree() {  
540 return leftIsPrevious ? null : left;  
541 }  
542  
543 /\*\*  
544 \* Gets the right node, returning null if its a faedelung.  
545 \*/  
546 private AVLNode<E> getRightSubTree() {  
547 return rightIsNext ? null : right;  
548 }  
549  
550 /\*\*  
551 \* Gets the rightmost child of this node.  
552 \*  
553 \* @return the rightmost child (greatest index)  
554 \*/  
555 private AVLNode<E> max() {  
556 return getRightSubTree() == null ? this : right.max();  
557 }  
558  
559 /\*\*  
560 \* Gets the leftmost child of this node.  
561 \*  
562 \* @return the leftmost child (smallest index)  
563 \*/  
564 private AVLNode<E> min() {  
565 return getLeftSubTree() == null ? this : left.min();  
566 }  
567  
568 /\*\*  
569 \* Removes the node at a given position.  
570 \*  
571 \* @param index is the index of the element to be removed relative to the position of  
572 \* the parent node of the current node.  
573 \*/  
574 AVLNode<E> remove(final int index) {  
575 final int indexRelativeToMe = index - relativePosition;  
576  
577 if (indexRelativeToMe == 0) {  
578 return removeSelf();  
579 }  
580 if (indexRelativeToMe > 0) {  
581 setRight(right.remove(indexRelativeToMe), right.right);  
582 if (relativePosition < 0) {  
583 relativePosition++;  
584 }  
585 } else {  
586 setLeft(left.remove(indexRelativeToMe), left.left);  
587 if (relativePosition > 0) {  
588 relativePosition--;  
589 }  
590 }  
591 recalcHeight();  
592 return balance();  
593 }  
594  
595 private AVLNode<E> removeMax() {  
596 if (getRightSubTree() == null) {  
597 return removeSelf();  
598 }  
599 setRight(right.removeMax(), right.right);  
600 if (relativePosition < 0) {  
601 relativePosition++;  
602 }  
603 recalcHeight();  
604 return balance();  
605 }  
606  
607 private AVLNode<E> removeMin() {  
608 if (getLeftSubTree() == null) {  
609 return removeSelf();  
610 }  
611 setLeft(left.removeMin(), left.left);  
612 if (relativePosition > 0) {  
613 relativePosition--;  
614 }  
615 recalcHeight();  
616 return balance();  
617 }  
618  
619 /\*\*  
620 \* Removes this node from the tree.  
621 \*  
622 \* @return the node that replaces this one in the parent  
623 \*/  
624 private AVLNode<E> removeSelf() {  
625 if (getRightSubTree() == null && getLeftSubTree() == null) {  
626 return null;  
627 }  
628 if (getRightSubTree() == null) {  
629 if (relativePosition > 0) {  
630 left.relativePosition += relativePosition;  
631 }  
632 left.max().setRight(null, right);  
633 return left;  
634 }  
635 if (getLeftSubTree() == null) {  
636 right.relativePosition += relativePosition - (relativePosition < 0 ? 0 : 1);  
637 right.min().setLeft(null, left);  
638 return right;  
639 }  
640  
641 if (heightRightMinusLeft() > 0) {  
642 // more on the right, so delete from the right  
643 final AVLNode<E> rightMin = right.min();  
644 value = rightMin.value;  
645 if (leftIsPrevious) {  
646 left = rightMin.left;  
647 }  
648 right = right.removeMin();  
649 if (relativePosition < 0) {  
650 relativePosition++;  
651 }  
652 } else {  
653 // more on the left or equal, so delete from the left  
654 final AVLNode<E> leftMax = left.max();  
655 value = leftMax.value;  
656 if (rightIsNext) {  
657 right = leftMax.right;  
658 }  
659 final AVLNode<E> leftPrevious = left.left;  
660 left = left.removeMax();  
661 if (left == null) {  
662 // special case where left that was deleted was a double link  
663 // only occurs when height difference is equal  
664 left = leftPrevious;  
665 leftIsPrevious = true;  
666 }  
667 if (relativePosition > 0) {  
668 relativePosition--;  
669 }  
670 }  
671 recalcHeight();  
672 return this;  
673 }  
674  
675 //-----------------------------------------------------------------------  
676 /\*\*  
677 \* Balances according to the AVL algorithm.  
678 \*/  
679 private AVLNode<E> balance() {  
680 switch (heightRightMinusLeft()) {  
681 case 1 :  
682 case 0 :  
683 case -1 :  
684 return this;  
685 case -2 :  
686 if (left.heightRightMinusLeft() > 0) {  
687 setLeft(left.rotateLeft(), null);  
688 }  
689 return rotateRight();  
690 case 2 :  
691 if (right.heightRightMinusLeft() < 0) {  
692 setRight(right.rotateRight(), null);  
693 }  
694 return rotateLeft();  
695 default :  
696 throw new RuntimeException("tree inconsistent!");  
697 }  
698 }  
699  
700 /\*\*  
701 \* Gets the relative position.  
702 \*/  
703 private int getOffset(final AVLNode<E> node) {  
704 if (node == null) {  
705 return 0;  
706 }  
707 return node.relativePosition;  
708 }  
709  
710 /\*\*  
711 \* Sets the relative position.  
712 \*/  
713 private int setOffset(final AVLNode<E> node, final int newOffest) {  
714 if (node == null) {  
715 return 0;  
716 }  
717 final int oldOffset = getOffset(node);  
718 node.relativePosition = newOffest;  
719 return oldOffset;  
720 }  
721  
722 /\*\*  
723 \* Sets the height by calculation.  
724 \*/  
725 private void recalcHeight() {  
726 height = Math.max(  
727 getLeftSubTree() == null ? -1 : getLeftSubTree().height,  
728 getRightSubTree() == null ? -1 : getRightSubTree().height) + 1;  
729 }  
730  
731 /\*\*  
732 \* Returns the height of the node or -1 if the node is null.  
733 \*/  
734 private int getHeight(final AVLNode<E> node) {  
735 return node == null ? -1 : node.height;  
736 }  
737  
738 /\*\*  
739 \* Returns the height difference right - left  
740 \*/  
741 private int heightRightMinusLeft() {  
742 return getHeight(getRightSubTree()) - getHeight(getLeftSubTree());  
743 }  
744  
745 private AVLNode<E> rotateLeft() {  
746 final AVLNode<E> newTop = right; // can't be faedelung!  
747 final AVLNode<E> movedNode = getRightSubTree().getLeftSubTree();  
748  
749 final int newTopPosition = relativePosition + getOffset(newTop);  
750 final int myNewPosition = -newTop.relativePosition;  
751 final int movedPosition = getOffset(newTop) + getOffset(movedNode);  
752  
753 setRight(movedNode, newTop);  
754 newTop.setLeft(this, null);  
755  
756 setOffset(newTop, newTopPosition);  
757 setOffset(this, myNewPosition);  
758 setOffset(movedNode, movedPosition);  
759 return newTop;  
760 }  
761  
762 private AVLNode<E> rotateRight() {  
763 final AVLNode<E> newTop = left; // can't be faedelung  
764 final AVLNode<E> movedNode = getLeftSubTree().getRightSubTree();  
765  
766 final int newTopPosition = relativePosition + getOffset(newTop);  
767 final int myNewPosition = -newTop.relativePosition;  
768 final int movedPosition = getOffset(newTop) + getOffset(movedNode);  
769  
770 setLeft(movedNode, newTop);  
771 newTop.setRight(this, null);  
772  
773 setOffset(newTop, newTopPosition);  
774 setOffset(this, myNewPosition);  
775 setOffset(movedNode, movedPosition);  
776 return newTop;  
777 }  
778  
779 /\*\*  
780 \* Sets the left field to the node, or the previous node if that is null  
781 \*  
782 \* @param node the new left subtree node  
783 \* @param previous the previous node in the linked list  
784 \*/  
785 private void setLeft(final AVLNode<E> node, final AVLNode<E> previous) {  
786 leftIsPrevious = node == null;  
787 left = leftIsPrevious ? previous : node;  
788 recalcHeight();  
789 }  
790  
791 /\*\*  
792 \* Sets the right field to the node, or the next node if that is null  
793 \*  
794 \* @param node the new left subtree node  
795 \* @param next the next node in the linked list  
796 \*/  
797 private void setRight(final AVLNode<E> node, final AVLNode<E> next) {  
798 rightIsNext = node == null;  
799 right = rightIsNext ? next : node;  
800 recalcHeight();  
801 }  
802  
803 /\*\*  
804 \* Appends the elements of another tree list to this tree list by efficiently  
805 \* merging the two AVL trees. This operation is destructive to both trees and  
806 \* runs in O(log(m + n)) time.  
807 \*  
808 \* @param otherTree  
809 \* the root of the AVL tree to merge with this one  
810 \* @param currentSize  
811 \* the number of elements in this AVL tree  
812 \* @return the root of the new, merged AVL tree  
813 \*/  
814 private AVLNode<E> addAll(AVLNode<E> otherTree, final int currentSize) {  
815 final AVLNode<E> maxNode = max();  
816 final AVLNode<E> otherTreeMin = otherTree.min();  
817  
818 // We need to efficiently merge the two AVL trees while keeping them  
819 // balanced (or nearly balanced). To do this, we take the shorter  
820 // tree and combine it with a similar-height subtree of the taller  
821 // tree. There are two symmetric cases:  
822 // \* this tree is taller, or  
823 // \* otherTree is taller.  
824 if (otherTree.height > height) {  
825 // CASE 1: The other tree is taller than this one. We will thus  
826 // merge this tree into otherTree.  
827  
828 // STEP 1: Remove the maximum element from this tree.  
829 final AVLNode<E> leftSubTree = removeMax();  
830  
831 // STEP 2: Navigate left from the root of otherTree until we  
832 // find a subtree, s, that is no taller than me. (While we are  
833 // navigating left, we store the nodes we encounter in a stack  
834 // so that we can re-balance them in step 4.)  
835 final Deque<AVLNode<E>> sAncestors = new ArrayDeque<>();  
836 AVLNode<E> s = otherTree;  
837 int sAbsolutePosition = s.relativePosition + currentSize;  
838 int sParentAbsolutePosition = 0;  
839 while (s != null && s.height > getHeight(leftSubTree)) {  
840 sParentAbsolutePosition = sAbsolutePosition;  
841 sAncestors.push(s);  
842 s = s.left;  
843 if (s != null) {  
844 sAbsolutePosition += s.relativePosition;  
845 }  
846 }  
847  
848 // STEP 3: Replace s with a newly constructed subtree whose root  
849 // is maxNode, whose left subtree is leftSubTree, and whose right  
850 // subtree is s.  
851 maxNode.setLeft(leftSubTree, null);  
852 maxNode.setRight(s, otherTreeMin);  
853 if (leftSubTree != null) {  
854 leftSubTree.max().setRight(null, maxNode);  
855 leftSubTree.relativePosition -= currentSize - 1;  
856 }  
857 if (s != null) {  
858 s.min().setLeft(null, maxNode);  
859 s.relativePosition = sAbsolutePosition - currentSize + 1;  
860 }  
861 maxNode.relativePosition = currentSize - 1 - sParentAbsolutePosition;  
862 otherTree.relativePosition += currentSize;  
863  
864 // STEP 4: Re-balance the tree and recalculate the heights of s's ancestors.  
865 s = maxNode;  
866 while (!sAncestors.isEmpty()) {  
867 final AVLNode<E> sAncestor = sAncestors.pop();  
868 sAncestor.setLeft(s, null);  
869 s = sAncestor.balance();  
870 }  
871 return s;  
872 }  
873 otherTree = otherTree.removeMin();  
874  
875 final Deque<AVLNode<E>> sAncestors = new ArrayDeque<>();  
876 AVLNode<E> s = this;  
877 int sAbsolutePosition = s.relativePosition;  
878 int sParentAbsolutePosition = 0;  
879 while (s != null && s.height > getHeight(otherTree)) {  
880 sParentAbsolutePosition = sAbsolutePosition;  
881 sAncestors.push(s);  
882 s = s.right;  
883 if (s != null) {  
884 sAbsolutePosition += s.relativePosition;  
885 }  
886 }  
887  
888 otherTreeMin.setRight(otherTree, null);  
889 otherTreeMin.setLeft(s, maxNode);  
890 if (otherTree != null) {  
891 otherTree.min().setLeft(null, otherTreeMin);  
892 otherTree.relativePosition++;  
893 }  
894 if (s != null) {  
895 s.max().setRight(null, otherTreeMin);  
896 s.relativePosition = sAbsolutePosition - currentSize;  
897 }  
898 otherTreeMin.relativePosition = currentSize - sParentAbsolutePosition;  
899  
900 s = otherTreeMin;  
901 while (!sAncestors.isEmpty()) {  
902 final AVLNode<E> sAncestor = sAncestors.pop();  
903 sAncestor.setRight(s, null);  
904 s = sAncestor.balance();  
905 }  
906 return s;  
907 }  
908  
909// private void checkFaedelung() {  
910// AVLNode maxNode = left.max();  
911// if (!maxNode.rightIsFaedelung || maxNode.right != this) {  
912// throw new RuntimeException(maxNode + " should right-faedel to " + this);  
913// }  
914// AVLNode minNode = right.min();  
915// if (!minNode.leftIsFaedelung || minNode.left != this) {  
916// throw new RuntimeException(maxNode + " should left-faedel to " + this);  
917// }  
918// }  
919//  
920// private int checkTreeDepth() {  
921// int hright = (getRightSubTree() == null ? -1 : getRightSubTree().checkTreeDepth());  
922// // System.out.print("checkTreeDepth");  
923// // System.out.print(this);  
924// // System.out.print(" left: ");  
925// // System.out.print(\_left);  
926// // System.out.print(" right: ");  
927// // System.out.println(\_right);  
928//  
929// int hleft = (left == null ? -1 : left.checkTreeDepth());  
930// if (height != Math.max(hright, hleft) + 1) {  
931// throw new RuntimeException(  
932// "height should be max" + hleft + "," + hright + " but is " + height);  
933// }  
934// return height;  
935// }  
936//  
937// private int checkLeftSubNode() {  
938// if (getLeftSubTree() == null) {  
939// return 0;  
940// }  
941// int count = 1 + left.checkRightSubNode();  
942// if (left.relativePosition != -count) {  
943// throw new RuntimeException();  
944// }  
945// return count + left.checkLeftSubNode();  
946// }  
947//  
948// private int checkRightSubNode() {  
949// AVLNode right = getRightSubTree();  
950// if (right == null) {  
951// return 0;  
952// }  
953// int count = 1;  
954// count += right.checkLeftSubNode();  
955// if (right.relativePosition != count) {  
956// throw new RuntimeException();  
957// }  
958// return count + right.checkRightSubNode();  
959// }  
960  
961 /\*\*  
962 \* Used for debugging.  
963 \*/  
964 @Override  
965 public String toString() {  
966 return new StringBuilder()  
967 .append("AVLNode(")  
968 .append(relativePosition)  
969 .append(',')  
970 .append(left != null)  
971 .append(',')  
972 .append(value)  
973 .append(',')  
974 .append(getRightSubTree() != null)  
975 .append(", faedelung ")  
976 .append(rightIsNext)  
977 .append(" )")  
978 .toString();  
979 }  
980 }  
981  
982 /\*\*  
983 \* A list iterator over the linked list.  
984 \*/  
985 static class TreeListIterator<E> implements ListIterator<E>, OrderedIterator<E> {  
986 /\*\* The parent list \*/  
987 private final TreeList<E> parent;  
988 /\*\*  
989 \* Cache of the next node that will be returned by {@link #next()}.  
990 \*/  
991 private AVLNode<E> next;  
992 /\*\*  
993 \* The index of the next node to be returned.  
994 \*/  
995 private int nextIndex;  
996 /\*\*  
997 \* Cache of the last node that was returned by {@link #next()}  
998 \* or {@link #previous()}.  
999 \*/  
1000 private AVLNode<E> current;  
1001 /\*\*  
1002 \* The index of the last node that was returned.  
1003 \*/  
1004 private int currentIndex;  
1005 /\*\*  
1006 \* The modification count that the list is expected to have. If the list  
1007 \* doesn't have this count, then a  
1008 \* {@link java.util.ConcurrentModificationException} may be thrown by  
1009 \* the operations.  
1010 \*/  
1011 private int expectedModCount;  
1012  
1013 /\*\*  
1014 \* Create a ListIterator for a list.  
1015 \*  
1016 \* @param parent the parent list  
1017 \* @param fromIndex the index to start at  
1018 \*/  
1019 protected TreeListIterator(final TreeList<E> parent, final int fromIndex) throws IndexOutOfBoundsException {  
1020 super();  
1021 this.parent = parent;  
1022 this.expectedModCount = parent.modCount;  
1023 this.next = parent.root == null ? null : parent.root.get(fromIndex);  
1024 this.nextIndex = fromIndex;  
1025 this.currentIndex = -1;  
1026 }  
1027  
1028 /\*\*  
1029 \* Checks the modification count of the list is the value that this  
1030 \* object expects.  
1031 \*  
1032 \* @throws ConcurrentModificationException If the list's modification  
1033 \* count isn't the value that was expected.  
1034 \*/  
1035 protected void checkModCount() {  
1036 if (parent.modCount != expectedModCount) {  
1037 throw new ConcurrentModificationException();  
1038 }  
1039 }  
1040  
1041 @Override  
1042 public boolean hasNext() {  
1043 return nextIndex < parent.size();  
1044 }  
1045  
1046 @Override  
1047 public E next() {  
1048 checkModCount();  
1049 if (!hasNext()) {  
1050 throw new NoSuchElementException("No element at index " + nextIndex + ".");  
1051 }  
1052 if (next == null) {  
1053 next = parent.root.get(nextIndex);  
1054 }  
1055 final E value = next.getValue();  
1056 current = next;  
1057 currentIndex = nextIndex++;  
1058 next = next.next();  
1059 return value;  
1060 }  
1061  
1062 @Override  
1063 public boolean hasPrevious() {  
1064 return nextIndex > 0;  
1065 }  
1066  
1067 @Override  
1068 public E previous() {  
1069 checkModCount();  
1070 if (!hasPrevious()) {  
1071 throw new NoSuchElementException("Already at start of list.");  
1072 }  
1073 if (next == null) {  
1074 next = parent.root.get(nextIndex - 1);  
1075 } else {  
1076 next = next.previous();  
1077 }  
1078 final E value = next.getValue();  
1079 current = next;  
1080 currentIndex = --nextIndex;  
1081 return value;  
1082 }  
1083  
1084 @Override  
1085 public int nextIndex() {  
1086 return nextIndex;  
1087 }  
1088  
1089 @Override  
1090 public int previousIndex() {  
1091 return nextIndex() - 1;  
1092 }  
1093  
1094 @Override  
1095 public void remove() {  
1096 checkModCount();  
1097 if (currentIndex == -1) {  
1098 throw new IllegalStateException();  
1099 }  
1100 parent.remove(currentIndex);  
1101 if (nextIndex != currentIndex) {  
1102 // remove() following next()  
1103 nextIndex--;  
1104 }  
1105 // the AVL node referenced by next may have become stale after a remove  
1106 // reset it now: will be retrieved by next call to next()/previous() via nextIndex  
1107 next = null;  
1108 current = null;  
1109 currentIndex = -1;  
1110 expectedModCount++;  
1111 }  
1112  
1113 @Override  
1114 public void set(final E obj) {  
1115 checkModCount();  
1116 if (current == null) {  
1117 throw new IllegalStateException();  
1118 }  
1119 current.setValue(obj);  
1120 }  
1121  
1122 @Override  
1123 public void add(final E obj) {  
1124 checkModCount();  
1125 parent.add(nextIndex, obj);  
1126 current = null;  
1127 currentIndex = -1;  
1128 nextIndex++;  
1129 expectedModCount++;  
1130 }  
1131 }  
1132  
1133}