```
package java.util;

import java.io.Serializable;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.UnaryOperator;
import jdk.internal.misc.SharedSecrets;

/**
 * Resizable-array implementation of the {@link Deque} interface.  Array
 * deques have no capacity restrictions; they grow as necessary to support
 * usage.  They are not thread-safe; in the absence of external
 * synchronization, they do not support concurrent access by multiple threads.
 * Null elements are prohibited.  This class is likely to be faster than
 * {@link Stack} when used as a stack, and faster than {@link LinkedList}
 * when used as a queue.
 *
 * <p>Most {@code ArrayDeque} operations run in amortized constant time.
 * Exceptions include
 * {@link #remove(Object) remove},
 * {@link #removeFirstOccurrence removeFirstOccurrence},
 * {@link #removeLastOccurrence removeLastOccurrence},
 * {@link #contains contains},
 * {@link #iterator iterator.remove()},
 * and the bulk operations, all of which run in linear time.
 *
 * <p>The iterators returned by this class's {@link #iterator() iterator}
 * method are <em>fail-fast</em>: If the deque is modified at any time after
 * the iterator is created, in any way except through the iterator's own
 * {@code remove} method, the iterator will generally throw a {@link
 * ConcurrentModificationException}.  Thus, in the face of concurrent
 * modification, the iterator fails quickly and cleanly, rather than risking
 * arbitrary, non-deterministic behavior at an undetermined time in the
 * future.
 *
 * <p>Note that the fail-fast behavior of an iterator cannot be guaranteed
 * as it is, generally speaking, impossible to make any hard guarantees in the
 * presence of unsynchronized concurrent modification.  Fail-fast iterators
 * throw {@code ConcurrentModificationException} on a best-effort basis.
 * Therefore, it would be wrong to write a program that depended on this
 * exception for its correctness: <i>the fail-fast behavior of iterators
 * should be used only to detect bugs.</i>
 *
 * <p>This class and its iterator implement all of the
```

```
 * <em>optional</em> methods of the {@link Collection} and {@link
 * Iterator} interfaces.
 *
 * <p>This class is a member of the
 * <a href="{@docRoot}/java/util/package-summary.html#CollectionsFramework">
 * Java Collections Framework</a>.
 *
 * @author  Josh Bloch and Doug Lea
 * @param <E> the type of elements held in this deque
 * @since   1.6
 */
public class ArrayDeque<E> extends AbstractCollection<E>
                           implements Deque<E>, Cloneable, Serializable
{
    /*
     * VMs excel at optimizing simple array loops where indices are
     * incrementing or decrementing over a valid slice, e.g.
     *
     * for (int i = start; i < end; i++) ... elements[i]
     *
     * Because in a circular array, elements are in general stored in
     * two disjoint such slices, we help the VM by writing unusual
     * nested loops for all traversals over the elements.  Having only
     * one hot inner loop body instead of two or three eases human
     * maintenance and encourages VM loop inlining into the caller.
     */

    /**
     * The array in which the elements of the deque are stored.
     * All array cells not holding deque elements are always null.
     * The array always has at least one null slot (at tail).
     */
    transient Object[] elements;

    /**
     * The index of the element at the head of the deque (which is the
     * element that would be removed by remove() or pop()); or an
     * arbitrary number 0 <= head < elements.length equal to tail if
     * the deque is empty.
     */
    transient int head;

    /**
     * The index at which the next element would be added to the tail
     * of the deque (via addLast(E), add(E), or push(E));
     * elements[tail] is always null.
     */
    transient int tail;

    /**
     * The maximum size of array to allocate.
     * Some VMs reserve some header words in an array.
     * Attempts to allocate larger arrays may result in
     * OutOfMemoryError: Requested array size exceeds VM limit
     */
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

    /**
     * Increases the capacity of this deque by at least the given amount.
     *
     * @param needed the required minimum extra capacity; must be positive
     */
    private void grow(int needed) {
        // overflow-conscious code
        final int oldCapacity = elements.length;
        int newCapacity;
        // Double capacity if small; else grow by 50%
        int jump = (oldCapacity < 64) ? (oldCapacity + 2) : (oldCapacity >> 1);
        if (jump < needed
            || (newCapacity = (oldCapacity + jump)) - MAX_ARRAY_SIZE > 0)
            newCapacity = newCapacity(needed, jump);
        final Object[] es = elements = Arrays.copyOf(elements, newCapacity);
        // Exceptionally, here tail == head needs to be disambiguated
        if (tail < head || (tail == head && es[head] != null)) {
            // wrap around; slide first leg forward to end of array
            int newSpace = newCapacity - oldCapacity;
            System.arraycopy(es, head,
                             es, head + newSpace,
```

```
                        oldCapacity - head);
            for (int i = head, to = (head += newSpace); i < to; i++)
                es[i] = null;
        }
    }

    /** Capacity calculation for edge conditions, especially overflow. */
    private int newCapacity(int needed, int jump) {
        final int oldCapacity = elements.length, minCapacity;
        if ((minCapacity = oldCapacity + needed) - MAX_ARRAY_SIZE > 0) {
            if (minCapacity < 0)
                throw new IllegalStateException("Sorry, deque too big");
            return Integer.MAX_VALUE;
        }
        if (needed > jump)
            return minCapacity;
        return (oldCapacity + jump - MAX_ARRAY_SIZE < 0)
            ? oldCapacity + jump
            : MAX_ARRAY_SIZE;
    }

    /**
     * Constructs an empty array deque with an initial capacity
     * sufficient to hold 16 elements.
     */
    public ArrayDeque() {
        elements = new Object[16];
    }

    /**
     * Constructs an empty array deque with an initial capacity
     * sufficient to hold the specified number of elements.
     *
     * @param numElements lower bound on initial capacity of the deque
     */
    public ArrayDeque(int numElements) {
        elements =
            new Object[(numElements < 1) ? 1 :
                       (numElements == Integer.MAX_VALUE) ? Integer.MAX_VALUE :
                       numElements + 1];
    }

    /**
     * Constructs a deque containing the elements of the specified
     * collection, in the order they are returned by the collection's
     * iterator.  (The first element returned by the collection's
     * iterator becomes the first element, or <i>front</i> of the
     * deque.)
     *
     * @param c the collection whose elements are to be placed into the deque
     * @throws NullPointerException if the specified collection is null
     */
    public ArrayDeque(Collection<? extends E> c) {
        this(c.size());
        addAll(c);
    }

    /**
     * Circularly increments i, mod modulus.
     * Precondition and postcondition: 0 <= i < modulus.
     */
    static final int inc(int i, int modulus) {
        if (++i >= modulus) i = 0;
        return i;
    }

    /**
     * Circularly decrements i, mod modulus.
     * Precondition and postcondition: 0 <= i < modulus.
     */
    static final int dec(int i, int modulus) {
        if (--i < 0) i = modulus - 1;
        return i;
    }

    /**
     * Circularly adds the given distance to index i, mod modulus.
     * Precondition: 0 <= i < modulus, 0 <= distance <= modulus.
```

```
     * @return index 0 <= i < modulus
     */
    static final int inc(int i, int distance, int modulus) {
        if ((i += distance) - modulus >= 0) i -= modulus;
        return i;
    }

    /**
     * Subtracts j from i, mod modulus.
     * Index i must be logically ahead of index j.
     * Precondition: 0 <= i < modulus, 0 <= j < modulus.
     * @return the "circular distance" from j to i; corner case i == j
     * is disambiguated to "empty", returning 0.
     */
    static final int sub(int i, int j, int modulus) {
        if ((i -= j) < 0) i += modulus;
        return i;
    }

    /**
     * Returns element at array index i.
     * This is a slight abuse of generics, accepted by javac.
     */
    @SuppressWarnings("unchecked")
    static final <E> E elementAt(Object[] es, int i) {
        return (E) es[i];
    }

    /**
     * A version of elementAt that checks for null elements.
     * This check doesn't catch all possible comodifications,
     * but does catch ones that corrupt traversal.
     */
    static final <E> E nonNullElementAt(Object[] es, int i) {
        @SuppressWarnings("unchecked") E e = (E) es[i];
        if (e == null)
            throw new ConcurrentModificationException();
        return e;
    }

    // The main insertion and extraction methods are addFirst,
    // addLast, pollFirst, pollLast. The other methods are defined in
    // terms of these.

    /**
     * Inserts the specified element at the front of this deque.
     *
     * @param e the element to add
     * @throws NullPointerException if the specified element is null
     */
    public void addFirst(E e) {
        if (e == null)
            throw new NullPointerException();
        final Object[] es = elements;
        es[head = dec(head, es.length)] = e;
        if (head == tail)
            grow(1);
    }

    /**
     * Inserts the specified element at the end of this deque.
     *
     * <p>This method is equivalent to {@link #add}.
     *
     * @param e the element to add
     * @throws NullPointerException if the specified element is null
     */
    public void addLast(E e) {
        if (e == null)
            throw new NullPointerException();
        final Object[] es = elements;
        es[tail] = e;
        if (head == (tail = inc(tail, es.length)))
            grow(1);
    }

    /**
     * Adds all of the elements in the specified collection at the end
```

```
     * of this deque, as if by calling {@link #addLast} on each one,
     * in the order that they are returned by the collection's iterator.
     *
     * @param c the elements to be inserted into this deque
     * @return {@code true} if this deque changed as a result of the call
     * @throws NullPointerException if the specified collection or any
     *         of its elements are null
     */
    public boolean addAll(Collection<? extends E> c) {
        final int s, needed;
        if ((needed = (s = size()) + c.size() + 1 - elements.length) > 0)
            grow(needed);
        c.forEach(this::addLast);
        return size() > s;
    }

    /**
     * Inserts the specified element at the front of this deque.
     *
     * @param e the element to add
     * @return {@code true} (as specified by {@link Deque#offerFirst})
     * @throws NullPointerException if the specified element is null
     */
    public boolean offerFirst(E e) {
        addFirst(e);
        return true;
    }

    /**
     * Inserts the specified element at the end of this deque.
     *
     * @param e the element to add
     * @return {@code true} (as specified by {@link Deque#offerLast})
     * @throws NullPointerException if the specified element is null
     */
    public boolean offerLast(E e) {
        addLast(e);
        return true;
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E removeFirst() {
        E e = pollFirst();
        if (e == null)
            throw new NoSuchElementException();
        return e;
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E removeLast() {
        E e = pollLast();
        if (e == null)
            throw new NoSuchElementException();
        return e;
    }

    public E pollFirst() {
        final Object[] es;
        final int h;
        E e = elementAt(es = elements, h = head);
        if (e != null) {
            es[h] = null;
            head = inc(h, es.length);
        }
        return e;
    }

    public E pollLast() {
        final Object[] es;
        final int t;
        E e = elementAt(es = elements, t = dec(tail, es.length));
        if (e != null)
            es[tail = t] = null;
        return e;
```

```
}

/**
 * @throws NoSuchElementException {@inheritDoc}
 */
public E getFirst() {
    E e = elementAt(elements, head);
    if (e == null)
        throw new NoSuchElementException();
    return e;
}

/**
 * @throws NoSuchElementException {@inheritDoc}
 */
public E getLast() {
    final Object[] es = elements;
    E e = elementAt(es, dec(tail, es.length));
    if (e == null)
        throw new NoSuchElementException();
    return e;
}

public E peekFirst() {
    return elementAt(elements, head);
}

public E peekLast() {
    final Object[] es;
    return elementAt(es = elements, dec(tail, es.length));
}

/**
 * Removes the first occurrence of the specified element in this
 * deque (when traversing the deque from head to tail).
 * If the deque does not contain the element, it is unchanged.
 * More formally, removes the first element {@code e} such that
 * {@code o.equals(e)} (if such an element exists).
 * Returns {@code true} if this deque contained the specified element
 * (or equivalently, if this deque changed as a result of the call).
 *
 * @param o element to be removed from this deque, if present
 * @return {@code true} if the deque contained the specified element
 */
public boolean removeFirstOccurrence(Object o) {
    if (o != null) {
        final Object[] es = elements;
        for (int i = head, end = tail, to = (i <= end) ? end : es.length;
             ; i = 0, to = end) {
            for (; i < to; i++)
                if (o.equals(es[i])) {
                    delete(i);
                    return true;
                }
            if (to == end) break;
        }
    }
    return false;
}

/**
 * Removes the last occurrence of the specified element in this
 * deque (when traversing the deque from head to tail).
 * If the deque does not contain the element, it is unchanged.
 * More formally, removes the last element {@code e} such that
 * {@code o.equals(e)} (if such an element exists).
 * Returns {@code true} if this deque contained the specified element
 * (or equivalently, if this deque changed as a result of the call).
 *
 * @param o element to be removed from this deque, if present
 * @return {@code true} if the deque contained the specified element
 */
public boolean removeLastOccurrence(Object o) {
    if (o != null) {
        final Object[] es = elements;
        for (int i = tail, end = head, to = (i >= end) ? end : 0;
             ; i = es.length, to = end) {
            for (i--; i > to - 1; i--)
```

```
                if (o.equals(es[i])) {
                    delete(i);
                    return true;
                }
            if (to == end) break;
        }
    }
    return false;
}

// *** Queue methods ***

/**
 * Inserts the specified element at the end of this deque.
 *
 * <p>This method is equivalent to {@link #addLast}.
 *
 * @param e the element to add
 * @return {@code true} (as specified by {@link Collection#add})
 * @throws NullPointerException if the specified element is null
 */
public boolean add(E e) {
    addLast(e);
    return true;
}

/**
 * Inserts the specified element at the end of this deque.
 *
 * <p>This method is equivalent to {@link #offerLast}.
 *
 * @param e the element to add
 * @return {@code true} (as specified by {@link Queue#offer})
 * @throws NullPointerException if the specified element is null
 */
public boolean offer(E e) {
    return offerLast(e);
}

/**
 * Retrieves and removes the head of the queue represented by this deque.
 *
 * This method differs from {@link #poll() poll()} only in that it
 * throws an exception if this deque is empty.
 *
 * <p>This method is equivalent to {@link #removeFirst}.
 *
 * @return the head of the queue represented by this deque
 * @throws NoSuchElementException {@inheritDoc}
 */
public E remove() {
    return removeFirst();
}

/**
 * Retrieves and removes the head of the queue represented by this deque
 * (in other words, the first element of this deque), or returns
 * {@code null} if this deque is empty.
 *
 * <p>This method is equivalent to {@link #pollFirst}.
 *
 * @return the head of the queue represented by this deque, or
 *         {@code null} if this deque is empty
 */
public E poll() {
    return pollFirst();
}

/**
 * Retrieves, but does not remove, the head of the queue represented by
 * this deque.  This method differs from {@link #peek peek} only in
 * that it throws an exception if this deque is empty.
 *
 * <p>This method is equivalent to {@link #getFirst}.
 *
 * @return the head of the queue represented by this deque
 * @throws NoSuchElementException {@inheritDoc}
 */
```

```
public E element() {
    return getFirst();
}

/**
 * Retrieves, but does not remove, the head of the queue represented by
 * this deque, or returns {@code null} if this deque is empty.
 *
 * <p>This method is equivalent to {@link #peekFirst}.
 *
 * @return the head of the queue represented by this deque, or
 *         {@code null} if this deque is empty
 */
public E peek() {
    return peekFirst();
}

// *** Stack methods ***

/**
 * Pushes an element onto the stack represented by this deque.  In other
 * words, inserts the element at the front of this deque.
 *
 * <p>This method is equivalent to {@link #addFirst}.
 *
 * @param e the element to push
 * @throws NullPointerException if the specified element is null
 */
public void push(E e) {
    addFirst(e);
}

/**
 * Pops an element from the stack represented by this deque.  In other
 * words, removes and returns the first element of this deque.
 *
 * <p>This method is equivalent to {@link #removeFirst()}.
 *
 * @return the element at the front of this deque (which is the top
 *         of the stack represented by this deque)
 * @throws NoSuchElementException {@inheritDoc}
 */
public E pop() {
    return removeFirst();
}

/**
 * Removes the element at the specified position in the elements array.
 * This can result in forward or backwards motion of array elements.
 * We optimize for least element motion.
 *
 * <p>This method is called delete rather than remove to emphasize
 * that its semantics differ from those of {@link List#remove(int)}.
 *
 * @return true if elements near tail moved backwards
 */
boolean delete(int i) {
    final Object[] es = elements;
    final int capacity = es.length;
    final int h, t;
    // number of elements before to-be-deleted elt
    final int front = sub(i, h = head, capacity);
    // number of elements after to-be-deleted elt
    final int back = sub(t = tail, i, capacity) - 1;
    if (front < back) {
        // move front elements forwards
        if (h <= i) {
            System.arraycopy(es, h, es, h + 1, front);
        } else { // Wrap around
            System.arraycopy(es, 0, es, 1, i);
            es[0] = es[capacity - 1];
            System.arraycopy(es, h, es, h + 1, front - (i + 1));
        }
        es[h] = null;
        head = inc(h, capacity);
        return false;
    } else {
        // move back elements backwards
```

```
        tail = dec(t, capacity);
        if (i <= tail) {
            System.arraycopy(es, i + 1, es, i, back);
        } else { // Wrap around
            System.arraycopy(es, i + 1, es, i, capacity - (i + 1));
            es[capacity - 1] = es[0];
            System.arraycopy(es, 1, es, 0, t - 1);
        }
        es[tail] = null;
        return true;
    }
}

// *** Collection Methods ***

/**
 * Returns the number of elements in this deque.
 *
 * @return the number of elements in this deque
 */
public int size() {
    return sub(tail, head, elements.length);
}

/**
 * Returns {@code true} if this deque contains no elements.
 *
 * @return {@code true} if this deque contains no elements
 */
public boolean isEmpty() {
    return head == tail;
}

/**
 * Returns an iterator over the elements in this deque.  The elements
 * will be ordered from first (head) to last (tail).  This is the same
 * order that elements would be dequeued (via successive calls to
 * {@link #remove} or popped (via successive calls to {@link #pop}).
 *
 * @return an iterator over the elements in this deque
 */
public Iterator<E> iterator() {
    return new DeqIterator();
}

public Iterator<E> descendingIterator() {
    return new DescendingIterator();
}

private class DeqIterator implements Iterator<E> {
    /** Index of element to be returned by subsequent call to next. */
    int cursor;

    /** Number of elements yet to be returned. */
    int remaining = size();

    /**
     * Index of element returned by most recent call to next.
     * Reset to -1 if element is deleted by a call to remove.
     */
    int lastRet = -1;

    DeqIterator() { cursor = head; }

    public final boolean hasNext() {
        return remaining > 0;
    }

    public E next() {
        if (remaining <= 0)
            throw new NoSuchElementException();
        final Object[] es = elements;
        E e = nonNullElementAt(es, cursor);
        cursor = inc(lastRet = cursor, es.length);
        remaining--;
        return e;
    }
```

```
    void postDelete(boolean leftShifted) {
        if (leftShifted)
            cursor = dec(cursor, elements.length);
    }

    public final void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        postDelete(delete(lastRet));
        lastRet = -1;
    }

    public void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        int r;
        if ((r = remaining) <= 0)
            return;
        remaining = 0;
        final Object[] es = elements;
        if (es[cursor] == null || sub(tail, cursor, es.length) != r)
            throw new ConcurrentModificationException();
        for (int i = cursor, end = tail, to = (i <= end) ? end : es.length;
             ; i = 0, to = end) {
            for (; i < to; i++)
                action.accept(elementAt(es, i));
            if (to == end) {
                if (end != tail)
                    throw new ConcurrentModificationException();
                lastRet = dec(end, es.length);
                break;
            }
        }
    }
}

private class DescendingIterator extends DeqIterator {
    DescendingIterator() { cursor = dec(tail, elements.length); }

    public final E next() {
        if (remaining <= 0)
            throw new NoSuchElementException();
        final Object[] es = elements;
        E e = nonNullElementAt(es, cursor);
        cursor = dec(lastRet = cursor, es.length);
        remaining--;
        return e;
    }

    void postDelete(boolean leftShifted) {
        if (!leftShifted)
            cursor = inc(cursor, elements.length);
    }

    public final void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        int r;
        if ((r = remaining) <= 0)
            return;
        remaining = 0;
        final Object[] es = elements;
        if (es[cursor] == null || sub(cursor, head, es.length) + 1 != r)
            throw new ConcurrentModificationException();
        for (int i = cursor, end = head, to = (i >= end) ? end : 0;
             ; i = es.length - 1, to = end) {
            // hotspot generates faster code than for: i >= to !
            for (; i > to - 1; i--)
                action.accept(elementAt(es, i));
            if (to == end) {
                if (end != head)
                    throw new ConcurrentModificationException();
                lastRet = end;
                break;
            }
        }
    }
}

/**
```

```
     * Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>
     * and <em>fail-fast</em> {@link Spliterator} over the elements in this
     * deque.
     *
     * <p>The {@code Spliterator} reports {@link Spliterator#SIZED},
     * {@link Spliterator#SUBSIZED}, {@link Spliterator#ORDERED}, and
     * {@link Spliterator#NONNULL}.  Overriding implementations should document
     * the reporting of additional characteristic values.
     *
     * @return a {@code Spliterator} over the elements in this deque
     * @since 1.8
     */
    public Spliterator<E> spliterator() {
        return new DeqSpliterator();
    }

    final class DeqSpliterator implements Spliterator<E> {
        private int fence;      // -1 until first use
        private int cursor;     // current index, modified on traverse/split

        /** Constructs late-binding spliterator over all elements. */
        DeqSpliterator() {
            this.fence = -1;
        }

        /** Constructs spliterator over the given range. */
        DeqSpliterator(int origin, int fence) {
            // assert 0 <= origin && origin < elements.length;
            // assert 0 <= fence && fence < elements.length;
            this.cursor = origin;
            this.fence = fence;
        }

        /** Ensures late-binding initialization; then returns fence. */
        private int getFence() { // force initialization
            int t;
            if ((t = fence) < 0) {
                t = fence = tail;
                cursor = head;
            }
            return t;
        }

        public DeqSpliterator trySplit() {
            final Object[] es = elements;
            final int i, n;
            return ((n = sub(getFence(), i = cursor, es.length) >> 1) <= 0)
                ? null
                : new DeqSpliterator(i, cursor = inc(i, n, es.length));
        }

        public void forEachRemaining(Consumer<? super E> action) {
            if (action == null)
                throw new NullPointerException();
            final int end = getFence(), cursor = this.cursor;
            final Object[] es = elements;
            if (cursor != end) {
                this.cursor = end;
                // null check at both ends of range is sufficient
                if (es[cursor] == null || es[dec(end, es.length)] == null)
                    throw new ConcurrentModificationException();
                for (int i = cursor, to = (i <= end) ? end : es.length;
                     ; i = 0, to = end) {
                    for (; i < to; i++)
                        action.accept(elementAt(es, i));
                    if (to == end) break;
                }
            }
        }

        public boolean tryAdvance(Consumer<? super E> action) {
            Objects.requireNonNull(action);
            final Object[] es = elements;
            if (fence < 0) { fence = tail; cursor = head; } // late-binding
            final int i;
            if ((i = cursor) == fence)
                return false;
            E e = nonNullElementAt(es, i);
```

```
        cursor = inc(i, es.length);
        action.accept(e);
        return true;
    }

    public long estimateSize() {
        return sub(getFence(), cursor, elements.length);
    }

    public int characteristics() {
        return Spliterator.NONNULL
            | Spliterator.ORDERED
            | Spliterator.SIZED
            | Spliterator.SUBSIZED;
    }
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
public void forEach(Consumer<? super E> action) {
    Objects.requireNonNull(action);
    final Object[] es = elements;
    for (int i = head, end = tail, to = (i <= end) ? end : es.length;
         ; i = 0, to = end) {
        for (; i < to; i++)
            action.accept(elementAt(es, i));
        if (to == end) {
            if (end != tail) throw new ConcurrentModificationException();
            break;
        }
    }
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
public boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    return bulkRemove(filter);
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
public boolean removeAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return bulkRemove(e -> c.contains(e));
}

/**
 * @throws NullPointerException {@inheritDoc}
 */
public boolean retainAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return bulkRemove(e -> !c.contains(e));
}

/** Implementation of bulk remove methods. */
private boolean bulkRemove(Predicate<? super E> filter) {
    final Object[] es = elements;
    // Optimize for initial run of survivors
    for (int i = head, end = tail, to = (i <= end) ? end : es.length;
         ; i = 0, to = end) {
        for (; i < to; i++)
            if (filter.test(elementAt(es, i)))
                return bulkRemoveModified(filter, i);
        if (to == end) {
            if (end != tail) throw new ConcurrentModificationException();
            break;
        }
    }
    return false;
}

// A tiny bit set implementation

private static long[] nBits(int n) {
```

```
    return new long[((n - 1) >> 6) + 1];
}
private static void setBit(long[] bits, int i) {
    bits[i >> 6] |= 1L << i;
}
private static boolean isClear(long[] bits, int i) {
    return (bits[i >> 6] & (1L << i)) == 0;
}

/**
 * Helper for bulkRemove, in case of at least one deletion.
 * Tolerate predicates that reentrantly access the collection for
 * read (but writers still get CME), so traverse once to find
 * elements to delete, a second pass to physically expunge.
 *
 * @param beg valid index of first element to be deleted
 */
private boolean bulkRemoveModified(
    Predicate<? super E> filter, final int beg) {
    final Object[] es = elements;
    final int capacity = es.length;
    final int end = tail;
    final long[] deathRow = nBits(sub(end, beg, capacity));
    deathRow[0] = 1L;   // set bit 0
    for (int i = beg + 1, to = (i <= end) ? end : es.length, k = beg;
         ; i = 0, to = end, k -= capacity) {
        for (; i < to; i++)
            if (filter.test(elementAt(es, i)))
                setBit(deathRow, i - k);
        if (to == end) break;
    }
    // a two-finger traversal, with hare i reading, tortoise w writing
    int w = beg;
    for (int i = beg + 1, to = (i <= end) ? end : es.length, k = beg;
         ; w = 0) { // w rejoins i on second leg
        // In this loop, i and w are on the same leg, with i > w
        for (; i < to; i++)
            if (isClear(deathRow, i - k))
                es[w++] = es[i];
        if (to == end) break;
        // In this loop, w is on the first leg, i on the second
        for (i = 0, to = end, k -= capacity; i < to && w < capacity; i++)
            if (isClear(deathRow, i - k))
                es[w++] = es[i];
        if (i >= to) {
            if (w == capacity) w = 0; // "corner" case
            break;
        }
    }
    if (end != tail) throw new ConcurrentModificationException();
    circularClear(es, tail = w, end);
    return true;
}

/**
 * Returns {@code true} if this deque contains the specified element.
 * More formally, returns {@code true} if and only if this deque contains
 * at least one element {@code e} such that {@code o.equals(e)}.
 *
 * @param o object to be checked for containment in this deque
 * @return {@code true} if this deque contains the specified element
 */
public boolean contains(Object o) {
    if (o != null) {
        final Object[] es = elements;
        for (int i = head, end = tail, to = (i <= end) ? end : es.length;
             ; i = 0, to = end) {
            for (; i < to; i++)
                if (o.equals(es[i]))
                    return true;
            if (to == end) break;
        }
    }
    return false;
}

/**
 * Removes a single instance of the specified element from this deque.
```

```
     * If the deque does not contain the element, it is unchanged.
     * More formally, removes the first element {@code e} such that
     * {@code o.equals(e)} (if such an element exists).
     * Returns {@code true} if this deque contained the specified element
     * (or equivalently, if this deque changed as a result of the call).
     *
     * <p>This method is equivalent to {@link #removeFirstOccurrence(Object)}.
     *
     * @param o element to be removed from this deque, if present
     * @return {@code true} if this deque contained the specified element
     */
    public boolean remove(Object o) {
        return removeFirstOccurrence(o);
    }

    /**
     * Removes all of the elements from this deque.
     * The deque will be empty after this call returns.
     */
    public void clear() {
        circularClear(elements, head, tail);
        head = tail = 0;
    }

    /**
     * Nulls out slots starting at array index i, upto index end.
     * Condition i == end means "empty" - nothing to do.
     */
    private static void circularClear(Object[] es, int i, int end) {
        // assert 0 <= i && i < es.length;
        // assert 0 <= end && end < es.length;
        for (int to = (i <= end) ? end : es.length;
             ; i = 0, to = end) {
            for (; i < to; i++) es[i] = null;
            if (to == end) break;
        }
    }

    /**
     * Returns an array containing all of the elements in this deque
     * in proper sequence (from first to last element).
     *
     * <p>The returned array will be "safe" in that no references to it are
     * maintained by this deque.  (In other words, this method must allocate
     * a new array).  The caller is thus free to modify the returned array.
     *
     * <p>This method acts as bridge between array-based and collection-based
     * APIs.
     *
     * @return an array containing all of the elements in this deque
     */
    public Object[] toArray() {
        return toArray(Object[].class);
    }

    private <T> T[] toArray(Class<T[]> klazz) {
        final Object[] es = elements;
        final T[] a;
        final int head = this.head, tail = this.tail, end;
        if ((end = tail + ((head <= tail) ? 0 : es.length)) >= 0) {
            // Uses null extension feature of copyOfRange
            a = Arrays.copyOfRange(es, head, end, klazz);
        } else {
            // integer overflow!
            a = Arrays.copyOfRange(es, 0, end - head, klazz);
            System.arraycopy(es, head, a, 0, es.length - head);
        }
        if (end != tail)
            System.arraycopy(es, 0, a, es.length - head, tail);
        return a;
    }

    /**
     * Returns an array containing all of the elements in this deque in
     * proper sequence (from first to last element); the runtime type of the
     * returned array is that of the specified array.  If the deque fits in
     * the specified array, it is returned therein.  Otherwise, a new array
     * is allocated with the runtime type of the specified array and the
```

```
     * size of this deque.
     *
     * <p>If this deque fits in the specified array with room to spare
     * (i.e., the array has more elements than this deque), the element in
     * the array immediately following the end of the deque is set to
     * {@code null}.
     *
     * <p>Like the {@link #toArray()} method, this method acts as bridge between
     * array-based and collection-based APIs.  Further, this method allows
     * precise control over the runtime type of the output array, and may,
     * under certain circumstances, be used to save allocation costs.
     *
     * <p>Suppose {@code x} is a deque known to contain only strings.
     * The following code can be used to dump the deque into a newly
     * allocated array of {@code String}:
     *
     * <pre> {@code String[] y = x.toArray(new String[0]);}</pre>
     *
     * Note that {@code toArray(new Object[0])} is identical in function to
     * {@code toArray()}.
     *
     * @param a the array into which the elements of the deque are to
     *          be stored, if it is big enough; otherwise, a new array of the
     *          same runtime type is allocated for this purpose
     * @return an array containing all of the elements in this deque
     * @throws ArrayStoreException if the runtime type of the specified array
     *         is not a supertype of the runtime type of every element in
     *         this deque
     * @throws NullPointerException if the specified array is null
     */
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(T[] a) {
        final int size;
        if ((size = size()) > a.length)
            return toArray((Class<T[]>) a.getClass());
        final Object[] es = elements;
        for (int i = head, j = 0, len = Math.min(size, es.length - i);
             ; i = 0, len = tail) {
            System.arraycopy(es, i, a, j, len);
            if ((j += len) == size) break;
        }
        if (size < a.length)
            a[size] = null;
        return a;
    }

    // *** Object methods ***

    /**
     * Returns a copy of this deque.
     *
     * @return a copy of this deque
     */
    public ArrayDeque<E> clone() {
        try {
            @SuppressWarnings("unchecked")
            ArrayDeque<E> result = (ArrayDeque<E>) super.clone();
            result.elements = Arrays.copyOf(elements, elements.length);
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }

    private static final long serialVersionUID = 2340985798034038923L;

    /**
     * Saves this deque to a stream (that is, serializes it).
     *
     * @param s the stream
     * @throws java.io.IOException if an I/O error occurs
     * @serialData The current size ({@code int}) of the deque,
     * followed by all of its elements (each an object reference) in
     * first-to-last order.
     */
    private void writeObject(java.io.ObjectOutputStream s)
            throws java.io.IOException {
        s.defaultWriteObject();
```

```
        // Write out size
        s.writeInt(size());

        // Write out elements in order.
        final Object[] es = elements;
        for (int i = head, end = tail, to = (i <= end) ? end : es.length;
             ; i = 0, to = end) {
            for (; i < to; i++)
                s.writeObject(es[i]);
            if (to == end) break;
        }
    }

    /**
     * Reconstitutes this deque from a stream (that is, deserializes it).
     * @param s the stream
     * @throws ClassNotFoundException if the class of a serialized object
     *         could not be found
     * @throws java.io.IOException if an I/O error occurs
     */
    private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();

        // Read in size and allocate array
        int size = s.readInt();
        SharedSecrets.getJavaObjectInputStreamAccess().checkArray(s, Object[].class, size + 1);
        elements = new Object[size + 1];
        this.tail = size;

        // Read in all elements in the proper order.
        for (int i = 0; i < size; i++)
            elements[i] = s.readObject();
    }

    /** debugging */
    void checkInvariants() {
        // Use head and tail fields with empty slot at tail strategy.
        // head == tail disambiguates to "empty".
        try {
            int capacity = elements.length;
            // assert 0 <= head && head < capacity;
            // assert 0 <= tail && tail < capacity;
            // assert capacity > 0;
            // assert size() < capacity;
            // assert head == tail || elements[head] != null;
            // assert elements[tail] == null;
            // assert head == tail || elements[dec(tail, capacity)] != null;
        } catch (Throwable t) {
            System.err.printf("head=%d tail=%d capacity=%d%n",
                              head, tail, elements.length);
            System.err.printf("elements=%s%n",
                              Arrays.toString(elements));
            throw t;
        }
    }

}
```