Write a generic data type for a deque and a randomized queue. The goal of this assignment is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

**Dequeue.** A *double-ended queue* or *deque* (pronounced “deck”) is a generalization of a stack and a queue that supports adding and removing items from either the front or the back of the data structure. Create a generic data type Deque that implements the following API:

**public class Deque<Item> implements Iterable<Item> {**

// construct an empty deque

**public Deque()**

// is the deque empty?

**public boolean isEmpty()**

// return the number of items on the deque

**public int size()**

// add the item to the front

**public void addFirst(Item item)**

// add the item to the back

**public void addLast(Item item)**

// remove and return the item from the front

**public Item removeFirst()**

// remove and return the item from the back

**public Item removeLast()**

// return an iterator over items in order from front to back

**public Iterator<Item> iterator()**

// unit testing (required)

**public static void main(String[] args)**

**}**

*Corner cases.* Throw the specified exception for the following corner cases:

* Throw an IllegalArgumentException if the client calls either addFirst() or addLast() with a null argument.
* Throw a java.util.NoSuchElementException if the client calls either removeFirst() or removeLast when the deque is empty.
* Throw a java.util.NoSuchElementException if the client calls the next() method in the iterator when there are no more items to return.
* Throw an UnsupportedOperationException if the client calls the remove() method in the iterator.

*Unit testing.* Your main() method must call directly every public constructor and method to help verify that they work as prescribed (e.g., by printing results to standard output).

*Performance requirements.* Your deque implementation must support each deque operation (including construction) in *constant worst-case time*. A deque containing *n* items must use at most 48*n* + 192 bytes of memory. Additionally, your iterator implementation must support each operation (including construction) in *constant worst-case time*.

**Randomized queue.** A *randomized queue* is similar to a stack or queue, except that the item removed is chosen uniformly at random among items in the data structure. Create a generic data type RandomizedQueue that implements the following API:

**public class RandomizedQueue<Item> implements Iterable<Item> {**

// construct an empty randomized queue

**public RandomizedQueue()**

// is the randomized queue empty?

**public boolean isEmpty()**

// return the number of items on the randomized queue

**public int size()**

// add the item

**public void enqueue(Item item)**

// remove and return a random item

**public Item dequeue()**

// return a random item (but do not remove it)

**public Item sample()**

// return an independent iterator over items in random order

**public Iterator<Item> iterator()**

// unit testing (required)

**public static void main(String[] args)**

**}**

*Iterator*.  Each iterator must return the items in uniformly random order. The order of two or more iterators to the same randomized queue must be *mutually independent*; each iterator must maintain its own random order.

*Corner cases.* Throw the specified exception for the following corner cases:

* Throw an IllegalArgumentException if the client calls enqueue() with a null argument.
* Throw a java.util.NoSuchElementException if the client calls either sample() or dequeue() when the randomized queue is empty.
* Throw a java.util.NoSuchElementException if the client calls the next() method in the iterator when there are no more items to return.
* Throw an UnsupportedOperationException if the client calls the remove() method in the iterator.

*Unit testing.* Your main() method must call directly every public constructor and method to verify that they work as prescribed (e.g., by printing results to standard output).

*Performance requirements.* Your randomized queue implementation must support each randomized queue operation (besides creating an iterator) in *constant amortized time*. That is, any intermixed sequence of *m* randomized queue operations (starting from an empty queue) must take at most *cm* steps in the worst case, for some constant *c*. A randomized queue containing *n* items must use at most 48*n* + 192 bytes of memory. Additionally, your iterator implementation must support operations next() and hasNext() in *constant worst-case time*; and construction in *linear time*; you may (and will need to) use a linear amount of extra memory per iterator.

**Client.** Write a client program Permutation.java that takes an integer *k* as a command-line argument; reads a sequence of strings from standard input using StdIn.readString(); and prints exactly *k* of them, uniformly at random. Print each item from the sequence at most once.

|  |  |
| --- | --- |
| ~/Desktop/queues> cat distinct.txt  A B C D E F G H I  ~/Desktop/queues> java Permutation 3 < distinct.txt  C  G  A  ~/Desktop/queues> java Permutation 3 < distinct.txt  E  F  G | ~/Desktop/queues> cat duplicates.txt  AA BB BB BB BB BB CC CC  ~/Desktop/queues> java Permutation 8 < duplicates.txt  BB  AA  BB  CC  BB  BB  CC  BB |

Your program must implement the following API:

**public class Permutation {**

**public static void main(String[] args)**

**}**

*Command-line argument.* You may assume that 0 ≤ *k* ≤ *n*, where *n* is the number of string on standard input. Note that you are not given *n*.

*Performance requirements.* The running time of Permutation must be linear in the size of the input. You may use only a constant amount of memory plus either one Deque or RandomizedQueue object of maximum size at most *n*. (For an extra challenge and a small amount of extra credit, use only one Deque or RandomizedQueue object of maximum size at most *k*.)

**Web submission.** Submit a .zip file containing only RandomizedQueue.java, Deque.java, and Permutation.java. Your submission may not call library functions except those in [StdIn](https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdIn.html), [StdOut](https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdOut.html), [StdRandom](https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/StdRandom.html), [java.lang](http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html), [java.util.Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html), and [java.util.NoSuchElementException](http://docs.oracle.com/javase/8/docs/api/java/util/NoSuchElementException.html). In particular, do not use either [java.util.LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) or [java.util.ArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html).

Moreover, submit screenshoots after running your program