Queues Assignment

Web Clip

Write a generic data type for a deque and a randomized queue. The goal of this assignment is to implement elementary data structures using *resizing 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 implementation must achieve the following *worst-case* performance requirements:

A deque containing n items must use at most $48n + 192$ bytes of memory, not including the memory for the items themselves.
Each deque operation (including construction) must take *constant time*.
Each iterator operation (including construction) must take *constant 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 implementation must achieve the following *worst-case* performance requirements:

- A randomized queue containing n items must use at most $48n + 192$ bytes of memory, not including the memory for the items themselves.
- Each randomized queue operation (besides creating an iterator) must take *constant amortized time*. That is, starting from an empty randomized queue, any intermixed sequence of m such operations must take $\Theta(m)$ time in the worst case.
- An iterator over n items must at use at most $8n + 72$ bytes of memory.
- Constructing an iterator must take $\Theta(n)$ time; the `next()` and `hasNext()` operations must take *constant time*.

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> more distinct.txt
A B C D E F G H I

~/Desktop/queues> java-algs4 Permutation 3 < distinct.txt
```

```
~/Desktop/queues> more duplicates.txt
AA BB BB BB BB BB CC CC

~/Desktop/queues> java-algs4 Permutation 8 < duplicates.txt
```

```
C
G
A

~/Desktop/queues> java-algs4 Permutation 3 < distinct.txt
E
F
G
```

```
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 $0 \le k \le n$, where n is the number of string on standard input. Note that you are not given n.

Performance requirements. Your implementation must achieve the following *worst-case* performance requirements:

- The running time of `Permutation` must be 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 a constant amount of memory plus either one `Deque` or `RandomizedQueue` object of maximum size at most k.

Deliverables. Submit the programs `RandomizedQueue.java`, `Deque.java`, and `Permutation.java`, along with a `readme.txt` file, answering all questions. Your submission may not call library functions except those in `StdIn`, `StdOut`, `StdRandom`, `java.lang`, `java.util.Iterator`, and `java.util.NoSuchElementException`. In particular, do not use either `java.util.LinkedList` or `java.util.ArrayList`.

Grading.

file	*points*
Deque.java	15
RandomizedQueue.java	15
Permutation.java	5
readme.txt	5
	40

Reminder: You can lose up to 4 points for poor style and up to 4 points for inadequate unit testing.

Extra credit: You can earn 2 points of extra credit for conserving memory in `Permutation.java`.

This assignment was developed by Kevin Wayne

This assignment was developed by Kevin Wayne.