Programming Assignment 2: Deques and Randomized Queues

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> {
    public Deque()                          // construct an empty deque
    public boolean isEmpty()                // is the deque empty?
    public int size()                       // return the number of items on the deque
    public void addFirst(Item item)         // add the item to the front
    public void addLast(Item item)          // add the item to the end
    public Item removeFirst()               // remove and return the item from the front
    public Item removeLast()                // remove and return the item from the end
    public Iterator<Item> iterator()        // return an iterator over items in order from front to end
    public static void main(String[] args)  // unit testing (optional)
}
```

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

- Throw a `java.lang.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 a `java.lang.UnsupportedOperationException` if the client calls the `remove()` method in the iterator.

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 $48n + 192$ bytes of memory and use space proportional to the number of items *currently* in the deque. 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 from items in the data structure. Create a generic data type `RandomizedQueue` that implements the following API:

```
public class RandomizedQueue<Item> implements Iterable<Item> {
    public RandomizedQueue()                // construct an empty randomized queue
    public boolean isEmpty()                // is the randomized queue empty?
    public int size()                       // return the number of items on the randomized queue
    public void enqueue(Item item)          // add the item
    public Item dequeue()                   // remove and return a random item
    public Item sample()                    // return a random item (but do not remove it)
    public Iterator<Item> iterator()        // return an independent iterator over items in random order
    public static void main(String[] args)  // unit testing (optional)
}
```

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 a `java.lang.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 a `java.lang.UnsupportedOperationException` if the client calls the `remove()` method in the iterator.

Performance requirements. Your randomized queue implementation must support each randomized queue operation (besides creating an iterator) in *constant amortized time*. That is, any 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 $48n + 192$ 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 in 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.

```
% more distinct.txt                       % more duplicates.txt
A B C D E F G H I                         AA BB BB BB BB BB CC CC

% java-algs4 Permutation 3 < distinct.txt  % java-algs4 Permutation 8 < duplicates.txt
C                                                    BB
G                                                    AA
A                                                    BB
                                                     CC
% java-algs4 Permutation 3 < distinct.txt            BB
E                                                    BB
F                                                    CC
G                                                    BB
```

Your program must implement the following API:

```
public class Permutation {
    public static void main(String[] args)
}
```

Command-line input. You may assume that $0 \le k \le n$, where n is the number of string on standard input.

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, use only one `Deque` or `RandomizedQueue` object of maximum size at most k.)

Deliverables. Submit the programs `RandomizedQueue.java`, `Deque.java`, and `Permutation.java`. 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.

This assignment was developed by Kevin Wayne.
Copyright © 2005.