Programming Assignment 2: Randomized Queues and Deques

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 inserting 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)       // insert the item at the front
    public void addLast(Item item)        // insert the item at the end
    public Item removeFirst()             // delete and return the item at the front
    public Item removeLast()              // delete and return the item at the end
    public Iterator<Item> iterator()      // return an iterator over items in order from
front to end
}
```

Throw a `java.lang.NullPointerException` if the client attempts to add a null item; throw a `java.util.NoSuchElementException` if the client attempts to remove an item from an empty deque; throw a `java.lang.UnsupportedOperationException` if the client calls the `remove()` method in the iterator; throw a `java.util.NoSuchElementException` if the client calls the `next()` method in the iterator and there are no more items to return.

Your deque implementation should support each deque operation in *constant worst-case time* and use space proportional to the number of items *currently* in the deque. Additionally, your iterator implementation should support the operations `next()` and `hasNext()` (plus construction) in constant worst-case time and use a constant amount of extra space per iterator.

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 queue empty?
    public int size()                     // return the number of items on the queue
    public void enqueue(Item item)        // add the item
    public Item dequeue()                 // delete and return a random item
    public Item sample()                  // return (but do not delete) a random item
    public Iterator<Item> iterator()      // return an independent iterator over items in
random order
}
```

Throw a `java.lang.NullPointerException` if the client attempts to add a null item; throw a `java.util.NoSuchElementException` if the client attempts to sample or dequeue an item from an empty randomized queue; throw a `java.lang.UnsupportedOperationException` if the client calls the `remove()` method in the iterator; throw a `java.util.NoSuchElementException` if the client calls the `next()` method in the iterator and there are no more items to return.

Your randomized queue implementation should support each randomized queue operation (besides creating an iterator) in *constant amortized time* and use space proportional to the number of items *currently* in the queue. That is, any sequence of *M* randomized queue operations (starting from an empty queue) should take at most *cM* steps in the worst case, for some constant *c*. Additionally, your iterator implementation should support construction in time linear in the number of items and it should support the operations `next()` and `hasNext()` in constant worst-case time; you may use a linear amount of extra memory per iterator. The order of two or more iterators to the same randomized queue should be *mutually independent*; each iterator must maintain its own random order.

Subset client. Write a client program `Subset.java` that takes a command-line integer k, reads in a sequence of N strings from standard input using `StdIn.readString()`, and prints out exactly k of them, uniformly at random. Each item from the sequence can be printed out at most once. You may assume that $k \geq 0$ and no greater than the number of string on standard input.

```
% echo A B C D E F G H I | java Subset 3        % echo AA BB BB BB BB BB CC CC | java
Subset 8
C                                               BB
G                                               AA
A                                               BB
                                                CC
% echo A B C D E F G H I | java Subset 3        BB
E                                               BB
F                                               CC
G                                               BB
```

Your client should use only constant space plus one object either of type `Deque` or of type `RandomizedQueue`; use generics properly to avoid casting and compiler warnings. It should also use time and space proportional to at most N in the worst case, where N is the number of strings on standard input. (For an extra challenge, use space proportional to k.) It should have the following API.

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

Deliverables. Submit only `Deque.java`, `RandomizedQueue.java`, and `Subset.java`. We will supply `stdlib.jar`. You may not call any library functions other than those in `java.lang` and `stdlib.jar`.