

## 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.

**Deque.** 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
}
```

*Corner cases.* 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.

*Performance requirements.* Your deque implementation must 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 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 queue empty?
    public int size()                // return the number of items on the queue
    public void enqueue(Item item)    // add the item
    public Item dequeue()             // remove and return a random item
    public Item sample()             // return (but do not remove) a random item
    public Iterator<Item> iterator()  // return an independent iterator over items in random order
    public static void main(String[] args) // unit testing
}
```

*Corner cases.* The order of two or more iterators to the same randomized queue must be *mutually independent*; each iterator must maintain its own 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.

*Performance requirements.* Your randomized queue implementation must 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 must support operations `next()` and `hasNext()` in *constant worst-case time*; and construction in *linear time*; you may use a linear amount of extra memory per iterator.

**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  $0 \leq k \leq N$ , where  $N$  is 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
G
A
                                         BB
                                         AA
                                         BB
                                         CC
% echo A B C D E F G H I | java Subset 3      BB
E
F
G                                         BB
                                         CC
                                         BB
```

The running time of `Subset` 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  $N$ , where  $N$  is the number of strings on standard input. (For an extra challenge, use only one `Deque` or `RandomizedQueue` object of maximum size at most  $k$ .) 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 `stdlib.jar`, `java.lang`, `java.util.Iterator`, and `java.util.NoSuchElementException`.