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
}
```

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*. A deque containing *N* items must use at most 48*N* + 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 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*. 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*. 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.

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 ≤ *k* ≤ *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                                              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
```

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 `algs4.jar`. Your submission not call library functions except those in `StdIn`, `StdOut`, `StdRandom`, `java.lang,java.util.Iterator`, and `java.util.NoSuchElementException`. In particular, you may not use either `java.util.LinkedList` or `java.util.ArrayList`.

Programming Assignment 2 Checklist: Randomized Queues and Dequeues

Frequently Asked Questions

Should I use arrays or linked lists in my implementations? In general we don't tell you *how* to implement your data structures—you can use arrays, linked lists, or maybe even invent your own new structure provide you abide by the specified time and space requirements. So, before you begin to write the code, make sure that your data structure will achieve the required resource bounds.

How serious are you about not calling any external library function other than those in `stdlib.jar`? You will receive a substantial deduction. The goal of this assignment is to implement data types from first principles, using resizing arrays and linked lists—feel free to use java.util.LinkedList and java.util.ArrayList on future programming assignments. We also require you to use `StdIn` (instead of java.util.Scanner) because we will intercept the calls to `StdIn` in our testing.

Can I add extra public methods to the `Deque` or `RandomizedQueue` APIs? Can I use different names for the methods? No, you must implement the API exactly as specified. The only exception is the `main()` method, which you should use for unit testing.

What is meant by uniformly at random? If there are *N* items in the randomized queue, then you should choose each one with probability 1/*N*, up to the randomness of `StdRandom.uniform()`, independent of past decisions. You can generate a pseudo-random integer between 0 and *N*-1 using `StdRandom.uniform(N)` from `StdRandom`.

Given an array, how can I rearrange the entries in random order? Use `StdRandom.shuffle()`—it implements the Knuth shuffle discussed in lecture and runs in linear time. Note that depending on your implementation, you may not need to call this method.

What should my deque (or randomized queue) iterator do if the deque (or randomized queue) is structurally modified at any time after the iterator is created (but before it is done iterating)? You don't need to worry about this in your solution. An industrial-strength solution (used in the Java libraries) is to make the iterator *fail-fast*: throw a `java.lang.ConcurrentModificationException` as soon as this is detected.

Why does the following code lead to a `generic array creation` compile-time error when `Item` is a generic type parameter?

```
Item[] a = new Item[1];
```

Java prohibits the creation of arrays of generic types. See the Q+A in Section 1.3 for a brief discussion. Instead, use a cast.

```
Item[] a = (Item[]) new Object[1];
```

Unfortunately, this leads to an unavoidable compiler warning.

The compiler says that my program uses unchecked or unsafe operations and to recompile with -Xlint:unchecked for details. Usually this means you did a potentially unsafe cast. When implementing a generic stack with an array, this is unavoidable since Java does not allow generic array creation. For example, the compiler outputs the following warning with ResizingArrayStack.java:

```
% javac ResizingArrayStack.java
Note: ResizingArrayStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked ResizingArrayStack.java
ResizingArrayStack.java:25: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: Item[]
        a = (Item[]) new Object[2];
                     ^
ResizingArrayStack.java:36: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: Item[]
        Item[] temp = (Item[]) new Object[capacity];
                               ^
2 warnings
```

You should not make any other casts.

Checkstyle complains that my nested class' instance variables must be private and have accessor methods that are not private. Do I need to make them private? No, but there's no harm in doing so. The access modifier of a nested class' instance variable is irrelevant—regardless of its access modifiers, it can be accessed anywhere in the file. (Of course, the enclosing class' instance variables should be private.)

Can a nested class have a constructor? Yes.

What assumptions can I make about the input to `Subset`? Standard input can contain any sequence of strings. You may assume that there is one integer command-line argument *k* and it is between 0 and the number of strings on standard input.

Will I lose points for loitering? Yes. Loitering is maintaining a useless reference to an object that could otherwise be garbage collected.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps. These same steps apply to each of the two data types that you will be implementing.

1. **Make sure you understand the performance requirements for both `Deque` and `RandomizedQueue`.** They are summarized in the table below. *Every detail in these performance requirements is important. Do not proceed until you understand them.*

	Deque	**Randomized Queue**
Non-iterator operations	Constant worst-case time	Constant amortized time
Iterator constructor	Constant worst-case time	linear in current # of items
Other iterator operations	Constant worst-case time	Constant worst-case time
Non-iterator memory use	Linear in current # of items	Linear in current # of items
Memory per iterator	Constant	Linear in current # of items

2. **Decide whether you want to use an array, linked list, or your own class.** This choice should be made based on the performance requirements discussed above. You may make different choices for `Deque` and`RandomizedQueue`. You might start by considering why a resizing array does not support *constant worst-case* time operations in a stack.
3. **Use our example programs as a guide when implementing your methods.** There are many new ideas in this programming assignment, including resizing arrays, linked lists, iterators, the *foreach* keyword, and generics. If you are not familiar with these topics, our example code should make things much easier. ResizingArrayStack.java uses a resizing array; LinkedStack.java uses a singly-linked list. Both examples use iterators, foreach, and generics.
4. **We strongly recommend that you develop unit tests for your code as soon as you've written enough methods to allow for testing.** As an example for `Deque`, you know that if you call `addFirst()` with the numbers 1 through N in ascending order, then call `removeLast()` N times, you should see the numbers 1 through N in ascending order. As soon as you have those two methods written, you can write a unit test for these methods. Arguably even better are randomized unit tests (which we employ heavily in our correctness testing). We recommend that you create a client class with a name like `TestDeque`, where each unit test is a method in this class. Don't forget to test your iterator.

Programming Tricks and Common Pitfalls

1. **It is very important that you carefully plan your implementation before you begin.** In particular, for each data structure that you're implementing (`RandomizedQueue` and `Deque`), you must decide whether to use a linked list, an array, or something else. If you make the wrong choice, you will not achieve the performance requirements and you will have to abandon your code and start over.
2. **Make sure that your memory use is linear in the current number of items, as opposed to the greatest number of items that has ever been in the data structure since its instantiation.** If you're using a resizing array, you must resize the array when it becomes sufficiently empty. You must also take care to avoid loitering anytime you remove an item.
3. **Make sure to test what happens when your data structures are emptied.** One very common bug is for something to go wrong when your data structure goes from non-empty to empty and then back to non-empty. Make sure to include this in your tests.
4. **Make sure to test that multiple iterators can be used simultaneously.** You can test this with a nested *foreach* loop. The iterators should operate independently of one another.
5. **Don't rely on our automated tests for debugging.** You don't have access to the source code of our testing suite, so the *Assessment Details* may be hard to utilize for debugging. As suggested above, write your own unit tests; it's good practice.
6. **If you use a linked list, consider using a sentinel node (or nodes).** Sentinel nodes can simplify your code and prevent bugs. However, they are not required (and we have not provided examples that use sentinel nodes).