This document only contains the description of the project and the project problems. For the programming exercises on concepts needed for the project, please refer to the project checklist .

Goal The purpose of this project is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

Problem 1. (*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 iterable data type `LinkedDeque<Item>` in `LinkedDeque.java` that uses a linked list to implement the following deque API:

Method	Description
`LinkedDeque()`	constructs an empty deque
`boolean isEmpty()`	returns `true` if the deque empty, and `false` otherwise
`int size()`	returns the number of items on the deque
`void addFirst(Item item)`	adds *item* to the front of the deque
`void addLast(Item item)`	adds *item* to the end of the deque
`Item removeFirst()`	removes and returns the item from the front of the deque
`Item removeLast()`	removes and returns the item from the end of the deque
`Iterator<Item> iterator()`	returns an iterator over items in the deque in order from front to end
`String toString()`	returns a string representation of the deque

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 (including construction) in constant worst-case time and use space proportional to linear in the number of items currently in the deque. Additionally, your iterator implementation must support each operation (including construction) in constant worst-case time.

```
>_ ~/workspace/project2

$ java edu.umb.cs210.p2.LinkedDeque
false
(364 characters) There is grandeur in this view of life, with its several powers, having been originally
breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to
the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful
have been, and are being, evolved. ~ Charles Darwin, The Origin of Species
true
```

Problem 2. (*Random Queue*) A random 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 iterable data type `ResizingArrayRandomQueue<Item>` in `ResizingArrayRandomQueue.java` that uses a resizing array to implement the following random queue API:

Method	Description
`ResizingArrayRandomQueue()`	constructs an empty queue
`boolean isEmpty()`	returns `true` if the queue empty and `false` otherwise
`int size()`	returns the number of items on the queue
`void enqueue(Item item)`	adds *item* to the queue
`Item dequeue()`	removes and returns a random item from the queue
`Item sample()`	returns a random item from the queue, but does not remove it
`Iterator<Item> iterator()`	returns an independent iterator over items in the queue in random order
`String toString()`	returns a string representation of the queue

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 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 linear in the number of items currently in the queue. 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. Additionally, your iterator implementation must support `next()` and `hasNext()` in constant worst-case time and construction in linear time; you may use a linear amount of extra memory per iterator.

```
>_ ~/workspace/project2

$ java edu.umb.cs210.p2.ResizingArrayRandomQueue
1 2 3 4 5 6 7 8 9 10
<ctrl-d>
55
0
55
true
```

Problem 3. (*Subset*) The client program `Subset.java` takes a command-line integer k and a filename, reads in a sequence of strings from the file, and prints out exactly k of them, uniformly at random. Each item from the sequence is printed out at most once. Implement the `subset()` method in the program that reads strings from the file whose name is `args[1]`, stores the strings in a `ResizingArrayRandomQueue` object, and returns the object.

```
>_ ~/workspace/project2

$ java edu.umb.cs210.p2.Subset 3 data/input10.txt
B
J
A
$ java edu.umb.cs210.p2.Subset 10 data/input10.txt
D
G
B
I
C
A
E
F
J
H
```

Acknowledgements This project is an adaptation of the Deques and Randomized Queues assignment developed at Princeton University by Kevin Wayne.