This project will help you to understand how to implement elementary data structures using arrays and linked lists and will introduce you to generics and iterators.

Submitting Information:

- Use the code I provided for each problem. DON'T DELETE ANY METHOD
- You should write your code in (LinkedDeque.java, ResizingArrayRandomQueue.java, Subsets.java, TestCases.java)
- Put all files in one .zip folder. Name of the folder should be: P2_LastNameFirstInitial.zip
- If you are working in group of two, include partner.txt file with your partner's name
- Only person in group should submit the Project
- Submit your work on Canvas.
- The deadline is Wednesday, March 4th at 11:55PM
- Do not use any fancy libraries. We should be able to compile it under standard installs.

Programming Standards :

- Your header comment must describe what your program does.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something tricky.
- You must use indentation and blank lines to make control structures more readable.

Problem1. *(LinkedDeque.java)*

Introduction. 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()`	is the dequeue empty?
`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 queue in order from front to end
`String toString()`	returns a string representation of this deque

Corner Cases.

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

Performance Requirement. 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.

```
Output:
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
```

Problem2. *(ResizingArrayRandomQueue.java)*

Introduction. 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()`	is the dequeue empty?
`int size()`	returns the number of items on the deque
`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
`StringtoString()`	returns a string representation of this 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.

1. Throw a java.lang.NullPointerException if the client attempts to add a null item
2. Throw a java.util.NoSuchElementException if the client attempts to sample or dequeue an item from an empty randomized queue
3. Throw a java.lang.UnsupportedOperationException if the client calls the remove() method in the iterator
4. 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.

To run the code, include input.txt as "Redirect input from" in edit configuration (Similar to Union find program in HW1)

```
$ Expected output:
5050
0
5050
true
```

Problem3. *(Subset.java)*

Introduction. Write a client program Subset.java that takes a command-line integer k , reads in a sequence of 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 < N , where N is the number of string on standard input. The running time of the program must be linear in the size of the input. You may use only a constant amount of memory plus either one LinkedDeque or ResizingArrayRandomQueue object of maximum size at most N. For an extra challenge, limit the maximum size to k.

```
$ java Subset 8 (Note: When running in IntelliJ '8' is argument)
AA BB BB BB BB BB CC CC  (Note: This line is standard Input when you run program)
<ctrl-d>
BB
CC
AA
BB
BB
BB
CC
BB
```

About Test Cases (TestCases.java and TestRunner.java):
I have added few testcases in TestCases.java which should give idea about how to check basic functionality of code. You can run TestRunner.java to execute those test cases. You should add more test cases in TestCases.java to check functionality of your code. Read about how to add Junit test cases on Internet if you don't know about Junit. I will have Test Cases against which I will check submitted code for grading. So, make sure you test your code thoroughly.
[Note: You have to add JUnit jar as library in your project to compile test cases successfully]