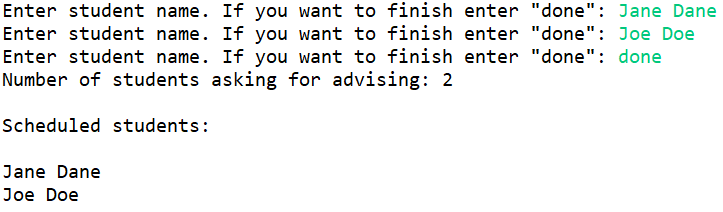
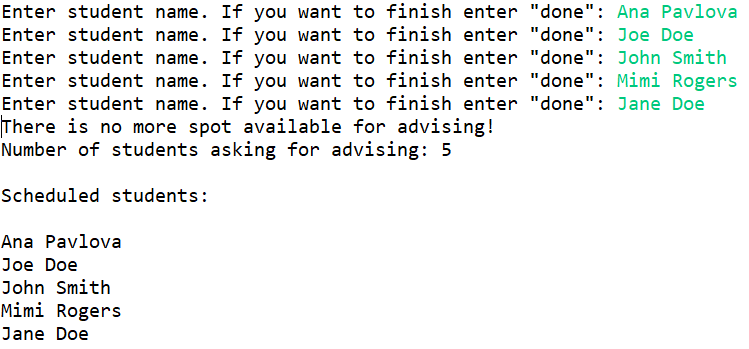
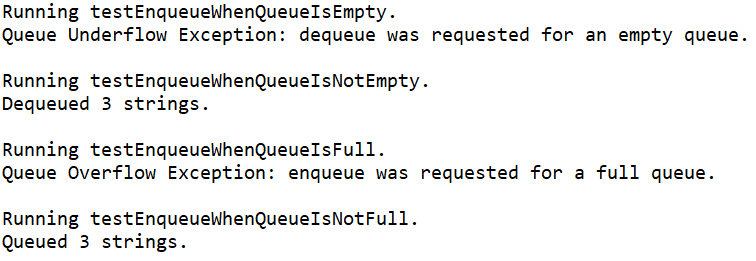
**1**

Main Output





Test Output



Code

**package** Com.TSL.AdvisingScheduleUtilities;

**import** java.util.Scanner;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\*

\* Implements the application for AdvisingSchedule\_Array class, using queue.

\* Outputs strings in same order of entry.

\*/

**public** **class** AdvisingSchedule\_ArrayDriver

{

/\*\*

\* main is the entry point of this program, which repeatedly prompts a user to enter a student name until "done" is

\* entered. The program proceeds to enqueue each name in a queue of names. If a name has been entered but the queue

\* is full, the program overwrites the name with "done" and outputs a warning. The program prints information about

\* the number of names in the queue and displays the names in the queue in the order in which they were entered into

\* the queue.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args)

{

Scanner scan = **new** Scanner(System.***in***);

//\*\*\* Task #1: defime a queue with elements of type String using QueueInterface

QueueInterface<String> nameQueue;

//\*\*\* Task #2: instantiate the queue using the constructor that provides the size, using a smaller number for test

nameQueue = **new** AdvisingSchedule\_Array<String>(5);

String name="";

//\*\*\* Task #3: enter the names of students that ask for advising.

//\* allow the user to terminate the entry

//\* terminate the loop when the queue is full, and display appropriate message to signal it

**while**(!name.equalsIgnoreCase("done"))

{

System.***out***.print("Enter student name. If you want to finish enter \"done\": ");

name = scan.nextLine();

**if**(!name.equalsIgnoreCase("done"))

nameQueue.enqueue(name);

// test if the queue is full

**if**(nameQueue.isFull())

{

name="done";

System.***out***.println("There is no more spot available for advising!");

}

}

//\*\*\* Task #4: identify the number of students in the advising queue and display it

**int** noStudents = nameQueue.size();

System.***out***.println("Number of students asking for advising: " + noStudents);

//\*\*\* Task #5: display the list of students presents in the advising queue

System.***out***.println("\nScheduled students:\n");

**while** (!nameQueue.isEmpty())

{

name = nameQueue.dequeue();

System.***out***.println(name);

}

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\*

\* AdvisingSchedule\_Array.java

\*

\* Implements QueueInterface with an array to hold the queue elements.

\*

\* Two constructors are provided: one that creates a queue of a default

\* capacity and one that allows the calling program to specify the capacity.

\*/

**public** **class** AdvisingSchedule\_Array<T> **implements** QueueInterface<T>

{

**protected** **final** **int** DEFCAP = 10; // default capacity

**protected** T[] elements; // array that holds queue elements

**protected** **int** numElements = 0; // number of elements in this queue

**protected** **int** front = 0; // index of front of queue

**protected** **int** rear; // index of rear of queue

/\*\*

\* AdvisingSchedule\_Array() is a zero-parameter constructor for AdvisingSchedule\_Array, which defines this advising

\* schedule's array of elements as a new array of <default capacity> elements of type T, and sets this advising

\* schedule's index for the rear-most element in the array as the default capacity of this advising schedule minus 1.

\*/

**public** AdvisingSchedule\_Array()

{

**this**.elements = (T[]) **new** Object[DEFCAP];

**this**.rear = DEFCAP - 1;

}

/\*\*

\* AdvisingSchedule\_Array(int maxSize) is a one-parameter constructor for AdvisingSchedule\_Array, which defines this

\* advising schedule's array of elements as a new array of <argument> elements of type T, and sets this advising

\* schedule's index for the rear-most element in the array as the argument minus 1.

\* **@param** maxSize

\*/

**public** AdvisingSchedule\_Array(**int** maxSize)

{

**this**.elements = (T[]) **new** Object[maxSize];

**this**.rear = maxSize - 1;

}

**public** **void** enqueue(T element)

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

{

**if** (isFull()) {

**throw** **new** QueueOverflowException("Queue Overflow Exception: enqueue was requested for a full queue.");

}

**this**.rear = (**this**.rear + 1) % **this**.elements.length;

**this**.elements[**this**.rear] = element;

**this**.numElements++;

}

**public** T dequeue()

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

{

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: dequeue was requested for an empty queue.");

}

T theStorage = **this**.elements[**this**.front];

**this**.elements[**this**.front] = **null**;

**this**.front = (**this**.front + 1) % **this**.elements.length;

**this**.numElements--;

**return** theStorage;

}

**public** **boolean** isEmpty()

// Returns true if this queue is empty; otherwise, returns false.

{

**return** (**this**.numElements == 0);

}

**public** **boolean** isFull()

// Returns true if this queue is full; otherwise, returns false.

{

**return** (**this**.numElements == **this**.elements.length);

}

**public** **int** size()

// Returns the number of elements in this queue.

{

**return** **this**.numElements;

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

//----------------------------------------------------------------------------

//QueueInterface.java

//

//Interface for a class that implements a queue of T.

//A queue is a "first in, first out" structure.

//

// Author: Emilia Butu

// Version: 1.0

// Since: 06/09/21

//----------------------------------------------------------------------------

**public** **interface** QueueInterface<T>

{

**void** enqueue(T element) **throws** QueueOverflowException;

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

T dequeue() **throws** QueueUnderflowException;

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

**boolean** isFull();

// Returns true if this queue is full; otherwise, returns false.

**boolean** isEmpty();

// Returns true if this queue is empty; otherwise, returns false.

**int** size();

// Returns the number of elements in this queue.

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* QueueOverflowException represents an exception that is thrown when an enqueue operation is requested for a full queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueOverflowException **extends** RuntimeException

{

/\*\*

\* QueueOverflowException() is a conventional zero-parameter constructor for QueueOverflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueOverflowException()

{

**super**();

}

/\*\*

\* QueueOverflowException(String message) is a one-parameter constructor for QueueOverflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueOverflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* QueueUnderflowException represents an exception that is thrown when an dequeue operation is requested for an empty

\* queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueUnderflowException **extends** RuntimeException

{

/\*\*

\* QueueUnderflowException() is a conventional zero-parameter constructor for QueueUnderflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueUnderflowException()

{

**super**();

}

/\*\*

\* QueueUnderflowException(String message) is a one-parameter constructor for QueueUnderflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueUnderflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

**import** org.junit.jupiter.api.Test;

/\*\*

\* AdvisingScheduleTest encapsulates JUnit tests of the enqueue and dequeue methods of AdvisingSchedule\_Array.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** AdvisingScheduleTest {

/\*\*

\* testEnqueueWhenQueueIsNotFull tests enqueue by adding three names to an advising schedule with a capacity of

\* more than 3 names.

\*/

@Test

**public** **void** testEnqueueWhenQueueIsNotFull() {

System.***out***.println("Running testEnqueueWhenQueueIsNotFull.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**try** {

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

System.***out***.println("Queued 3 strings.");

}

**catch** (QueueOverflowException theQueueOverflowException) {

System.***out***.println(theQueueOverflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsFull tests enqueue by attempting to add 1000 names to an advising schedule with a capacity

\* significantly less than 1000.

\*/

@Test

**public** **void** testEnqueueWhenQueueIsFull() {

System.***out***.println("Running testEnqueueWhenQueueIsFull.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**try** {

**for** (**int** i = 0; i < 1000; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

System.***out***.println("Queued 1000 strings.");

}

**catch** (QueueOverflowException theQueueOverflowException) {

System.***out***.println(theQueueOverflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by removing three names from an advising schedule with three names

\* in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsNotEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsNotEmpty.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 3 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by attempting to remove 1000 names from an advising schedule with

\* significantly fewer elements in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsEmpty.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 1000; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 1000 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

}

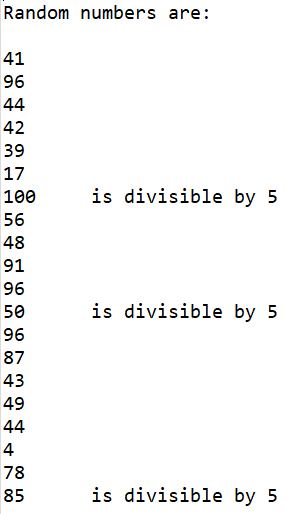
System.***out***.println();

}

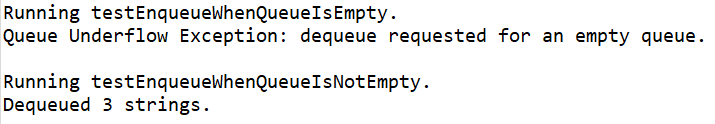
}

**2**

Main Output



Test Output



**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\*

\* Implements the application for RandomNumbers\_Linked class, using queue.

\* Buids a queue of random numbers and displays the content of the queue,

\* adding a message for the values divisible by 5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

**public** **class** RandomNumbers\_LinkedDriver

{

/\*\* -----------------------------------------------------------------------------------------------------------------

\* main is the entry point of this program, which creates a linked list based queue, generates twenty random integers

\* between 1 and 100 inclusive, enqueues those integers, and displays those integers in the order in which they were

\* queued, along with their divisibility by 5.

\*

\* **@param** args

----------------------------------------------------------------------------------------------------------------- \*/

**public** **static** **void** main(String[] args)

{

//\*\*\* Task #1: define a queue with elements of type Integer using QueueInterface

QueueInterface<Integer> randomQueue;

//\*\*\* Task #2: instantiate the queue as RandomNumbers\_Linked object

randomQueue = **new** RandomNumbers\_Linked<Integer>();

**int** number;

//\*\*\* Task #3: fill the queue with 20 integer values randomly generated in a range from 1 to 100

**for** (**int** i = 1; i <= 20; i++)

{

number = (**int**)(Math.*random*()\*100+1);

randomQueue.enqueue(number);

}

//\*\*\* Task #4: displays the content of the queue,

// for numbers divisible by 5, add the message "is divisible by 5"

System.***out***.println("\nRandom numbers are:\n");

**while** (!randomQueue.isEmpty())

{

number = randomQueue.dequeue();

System.***out***.println(number+((number%5==0)?"\tis divisible by 5":""));

}

}

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

//----------------------------------------------------------------------------

// QueueInterface.java

//

// Interface for a class that implements a queue of T.

// A queue is a "first in, first out" structure.

//

// Author: Emilia Butu

// Version: 1.0

// Since: 06/09/21

//----------------------------------------------------------------------------

**public** **interface** QueueInterface<T>

{

**void** enqueue(T element) **throws** QueueOverflowException;

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

T dequeue() **throws** QueueUnderflowException;

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

**boolean** isFull();

// Returns true if this queue is full; otherwise, returns false.

**boolean** isEmpty();

// Returns true if this queue is empty; otherwise, returns false.

**int** size();

// Returns the number of elements in this queue.

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\* Implements QueueInterface using a linked list.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

**public** **class** RandomNumbers\_Linked<T> **implements** QueueInterface<T>

{

**protected** LLNode<T> front; // reference to the front of this queue

**protected** LLNode<T> rear; // reference to the rear of this queue

**protected** **int** numElements = 0; // number of elements in this queue

/\*\*

\* RandomNumbers\_Linked is the zero-parameter constructor for RandomNumbers\_Linked, which sets this linked list's

\* front and rear node variables to reference null.

\*/

**public** RandomNumbers\_Linked()

{

**this**.front = **null**;

**this**.rear = **null**;

}

**public** **void** enqueue(T element)

// Adds element to the rear of this queue.

{

LLNode<T> theLinkedListNodeForTheElement = **new** LLNode<T>(element);

**if** (**this**.rear == **null**) {

**this**.front = theLinkedListNodeForTheElement;

}

**else** {

**this**.rear.setLink(theLinkedListNodeForTheElement);

}

**this**.rear = theLinkedListNodeForTheElement;

**this**.numElements++;

}

**public** T dequeue()

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

{

//\*\*\* Task #2: implement the method, using LLNode type objects defined with LLNode class

//\* if the queue is empty, throw the appropriate exception

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: dequeue requested for an empty queue.");

}

T theInformationOfTheFrontLinkedListNode = **this**.front.getInfo();

**this**.front = **this**.front.getLink();

**if** (**this**.front == **null**) {

**this**.rear = **null**;

}

**this**.numElements--;

**return** theInformationOfTheFrontLinkedListNode;

}

**public** **boolean** isEmpty()

// Returns true if this queue is empty; otherwise, returns false.

{

**if** (**this**.front == **null**) {

**return** **true**;

}

**return** **false**;

}

**public** **boolean** isFull()

// Returns false - a linked queue is never full.

{

**return** **false**;

}

**public** **int** size()

// Returns the number of elements in this queue.

{

**return** **this**.numElements;

}

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Implements <T> nodes for a Linked List.

\*

\* **@author** Emilia Butu

\* **@version** 1.0

\* **@since** 06/09/21

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

**public** **class** LLNode<T>

{

**protected** LLNode<T> link;

**protected** T info;

/\*\* -------------------------------------------------------------------------------------------------------------

\* LLNode is the one-parameter constructor for LLNode, which sets the info of this LLNode to the argument and the

\* link of this LLNode to null.

\*

\* **@param** info

------------------------------------------------------------------------------------------------------------- \*/

**public** LLNode(T info)

{

**this**.info = info;

link = **null**;

}

/\*\* -----------------------------------------------------

\* setInfo sets the info of this LLNode to provided info.

\* **@param** info

----------------------------------------------------- \*/

**public** **void** setInfo(T info) {

**this**.info = info;

}

/\*\* -----------------------------------

\* getInfo provides this LLNode's info.

\* **@return**

----------------------------------- \*/

**public** T getInfo() {

**return** info;

}

/\*\* -----------------------------------------------------------------------------------------

\* setLink sets the link of this LLNode to reference the provided LLNode with info of type T.

\* **@param** link

----------------------------------------------------------------------------------------- \*/

**public** **void** setLink(LLNode<T> link) {

**this**.link = link;

}

/\*\* -----------------------------------

\* getLink provides this LLNode's link.

\* **@return**

----------------------------------- \*/

**public** LLNode<T> getLink() {

**return** link;

}

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

/\*\*

\* QueueOverflowException represents an exception that is thrown when an enqueue operation is requested for a full queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueOverflowException **extends** RuntimeException

{

/\*\*

\* QueueOverflowException() is a conventional zero-parameter constructor for QueueOverflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueOverflowException()

{

**super**();

}

/\*\*

\* QueueOverflowException(String message) is a one-parameter constructor for QueueOverflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueOverflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

/\*\*

\* QueueUnderflowException represents an exception that is thrown when an dequeue operation is requested for an empty

\* queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueUnderflowException **extends** RuntimeException

{

/\*\*

\* QueueUnderflowException() is a conventional zero-parameter constructor for QueueUnderflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueUnderflowException()

{

**super**();

}

/\*\*

\* QueueUnderflowException(String message) is a one-parameter constructor for QueueUnderflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueUnderflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.UtilitiesForQueuingAndEvaluatingRandomNumbers;

**import** org.junit.jupiter.api.Test;

/\*\*

\* AdvisingScheduleTest encapsulates JUnit tests of the enqueue and dequeue methods of AdvisingSchedule\_Array.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** LinkedListOfRandomNumbersTest {

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by removing three names from an advising schedule with three names

\* in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsNotEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsNotEmpty.");

RandomNumbers\_Linked<String> theAdvisingSchedule = **new** RandomNumbers\_Linked<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 3 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by attempting to remove 1000 names from an advising schedule with

\* significantly fewer elements in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsEmpty.");

RandomNumbers\_Linked<String> theAdvisingSchedule = **new** RandomNumbers\_Linked<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 1000; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 1000 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

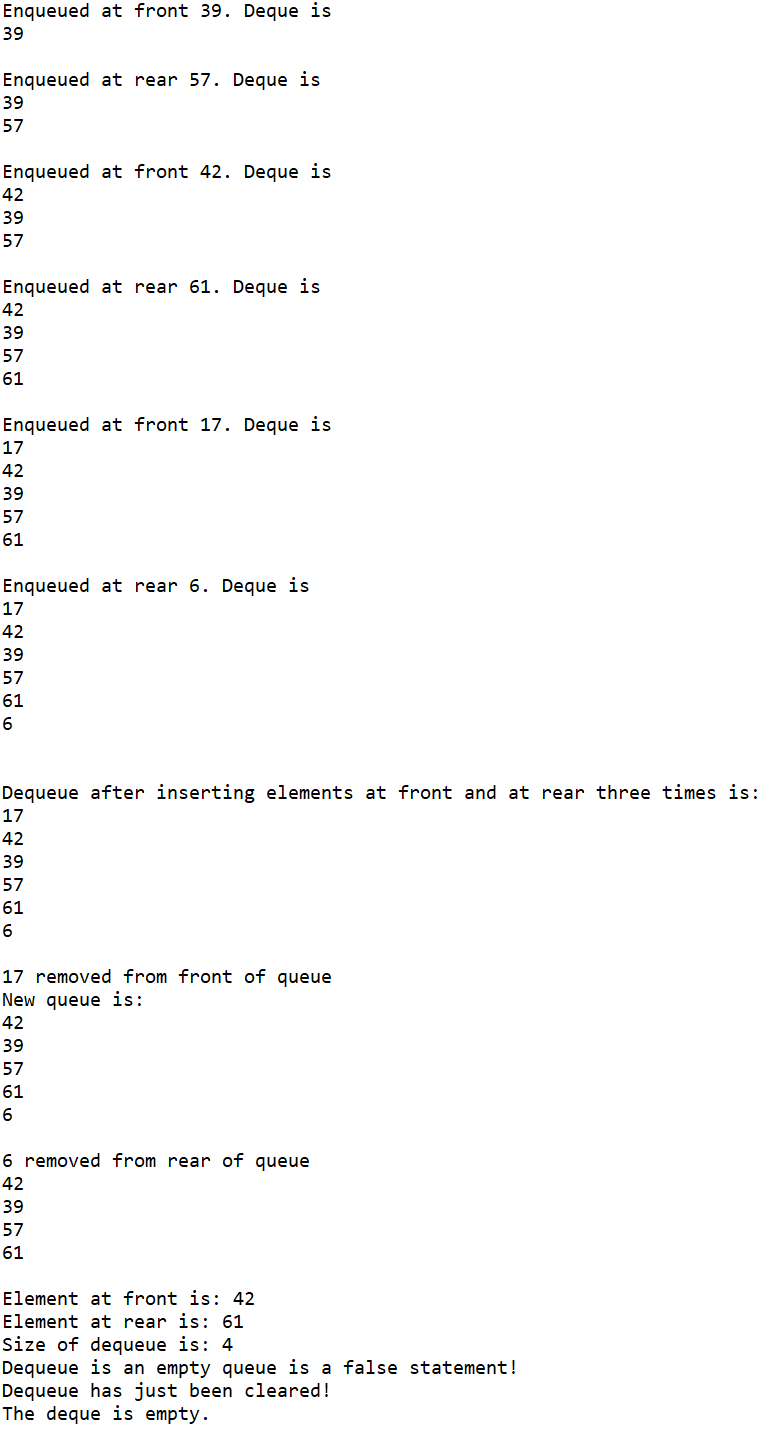
}

System.***out***.println();

}

}

**3**



**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/10/21

\*

\* DequeueDriver.java

\*

\* Implements the application for Dequeue class.

\*/

**public** **class** DequeDriver

{

/\*\*

\* main is the entry point of this program, which creates a double-ended queue, enqueues random integers between 1

\* and 100 at the front and rear, displays the double-ended queue, dequeues and displays, displays information about

\* the double-ended queue, and clears and displays the double-ended queue.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args)

{

//\*\*\* Task #1: object of type Dequeue and instantiate it with integers

Deque dq = **new** Deque<Integer>();

**int** value;

// work with random numbers between 1 to 100

// perform a number of dequeue operations

//\*\*\* Task #2: perform insert at front followed by insert at rear three times

**for**(**int** i=0;i<3; i++)

{

value=(**int**)(Math.*random*()\*100+1);

dq.enqueueFront(value);

System.***out***.println("Enqueued at front " + value + ". Deque is ");

dq.display();

value=(**int**)(Math.*random*()\*100+1);

dq.enqueueRear(value);

System.***out***.println("Enqueued at rear " + value + ". Deque is ");

dq.display();

}

System.***out***.println();

System.***out***.println("Dequeue after inserting elements at front and at rear three times is: ");

//\*\*\* Task #2: display dequeue

dq.display();

//\*\*\* Task #3: delete at front

System.***out***.println(dq.dequeueFront() + " removed from front of queue");

System.***out***.println("New queue is: ");

//\*\*\* Task #4: display dequeue

dq.display();

//\*\*\* Task #5: delete at rear

System.***out***.println(dq.dequeueRear() + " removed from rear of queue");

//\*\*\* Task #6: display dequeue

dq.display();

//\*\*\* Task #7: display value at front

System.***out***.println("Element at front is: " + dq.peekAtFront());

//\*\*\* Task #8: display value at rear

System.***out***.println("Element at rear is: " + dq.peekAtRear());

//\*\*\* Task #9: display size

System.***out***.println("Size of dequeue is: " + dq.size());

//\*\*\* Task #10: check if queue is empty

System.***out***.println("Dequeue is an empty queue is a " + dq.isEmpty() + " statement!");

//\*\*\* Task #11: clear dequeue and display message to announcing the operation

dq.clear();

System.***out***.println("Dequeue has just been cleared!");

//\*\*\* Task #12: display resulting dequeue

dq.display();

}

}

**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/10/21

\*

\* Implements DequeInterface using a linked list.

\*/

**public** **class** Deque<T> **implements** DequeInterface<T>

{

**protected** LLNode<T> front; // reference to the front of this queue

**protected** LLNode<T> rear; // reference to the rear of this queue

**protected** **int** size = 0; // number of elements in this queue

// constructor

**public** Deque()

{

**this**.front = **null**;

**this**.rear = **null**;

}

**public** **boolean** isEmpty()

// Returns true if this queue is empty; otherwise, returns false.

{

**return** (**this**.front == **null**);

}

**public** **boolean** isFull()

// Returns false - a linked queue is never full.

{

**return** **false**;

}

**public** **int** size()

// Returns the number of elements in this queue.

{

**return** **this**.size;

}

/\*\* -------------------------------------

\* clear trashes all nodes in this deque.

------------------------------------- \*/

**public** **void** clear()

{

**this**.front = **null**;

**this**.rear = **null**;

}

// insert an element in the beginning

**public** **void** enqueueFront(T element)

// Adds element to the front of this queue.

{

LLNode<T> theLinkedListNodeForTheElement = **new** LLNode<T>(element);

**if** (**this**.rear == **null**) {

**this**.rear = theLinkedListNodeForTheElement;

}

**else** {

theLinkedListNodeForTheElement.setLink(**this**.front);

}

**this**.front = theLinkedListNodeForTheElement;

**this**.size++;

}

// insert an element at the end

**public** **void** enqueueRear(T element)

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

{

**if** (isFull()) {

**throw** **new** QueueOverflowException("Queue Overflow Exception: enqueueRear requested for a full deque.");

}

LLNode<T> theLinkedListNodeForTheElement = **new** LLNode<T>(element);

**if** (**this**.rear == **null**) {

**this**.front = theLinkedListNodeForTheElement;

}

**else** {

**this**.rear.setLink(theLinkedListNodeForTheElement);

}

**this**.rear = theLinkedListNodeForTheElement;

**this**.size++;

}

**public** T dequeueFront() **throws** QueueUnderflowException

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

{

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: dequeueFront requested for an empty deque.");

}

T theDataOfTheFrontLinkedListNode = **this**.front.getData();

**this**.front = **this**.front.getLink();

**if** (**this**.front == **null**) {

**this**.rear = **null**;

}

**this**.size--;

**return** theDataOfTheFrontLinkedListNode;

}

**public** T dequeueRear() **throws** QueueUnderflowException

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes rear element from this queue and returns it.

{

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: dequeueFront requested for an empty deque.");

}

T theDataOfTheRearLinkedListNode;

**if** (**this**.front.getLink() == **null**) {

theDataOfTheRearLinkedListNode = **this**.front.getData();

**this**.front = **null**;

**this**.rear = **null**;

**return** theDataOfTheRearLinkedListNode;

}

LLNode<T> theCurrentLinkedListNode = **this**.front;

**while** (theCurrentLinkedListNode.getLink() != **this**.rear) {

theCurrentLinkedListNode = theCurrentLinkedListNode.getLink();

}

theDataOfTheRearLinkedListNode = theCurrentLinkedListNode.getLink().getData();

theCurrentLinkedListNode.setLink(**null**);

**this**.rear = theCurrentLinkedListNode;

**this**.size--;

**return** theDataOfTheRearLinkedListNode;

}

// method to check the front element of the queue

**public** T peekAtFront()

{

//\*\*\* Task #10: implement method

**if**(isEmpty()) {

**throw** **new** QueueUnderflowException("underflow exception");

}

**return** **this**.front.getData();

}

// method to check the end element of the queue

**public** T peekAtRear()

{

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: peekAtRear requested for an empty deque.");

}

**return** **this**.rear.getData();

}

// method to display the queue, giving a specific message if the queue is empty

**public** **void** display()

{

**if** (isEmpty()) {

System.***out***.println("The deque is empty.");

**return**;

}

**if** (**this**.front.getLink() == **null**) {

System.***out***.println(**this**.front.getData() + "\n");

**return**;

}

String theRepresentationOfTheDeque = "";

LLNode<T> theCurrentLinkedListNode = **this**.front;

**while** (theCurrentLinkedListNode != **null**) {

theRepresentationOfTheDeque += theCurrentLinkedListNode.getData() + "\n";

theCurrentLinkedListNode = theCurrentLinkedListNode.getLink();

}

System.***out***.println(theRepresentationOfTheDeque);

}

}

**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\* DequeInterface.java

\* Interface for a class that implements a deque of T.

\* A deque is a linear structure allowing insertion/removal at both ends.

\*/

**public** **interface** DequeInterface<T>

{

**void** enqueueFront(T element) **throws** QueueOverflowException;

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the front of this queue.

**void** enqueueRear(T element) **throws** QueueOverflowException;

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

T dequeueFront() **throws** QueueUnderflowException;

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

T dequeueRear() **throws** QueueUnderflowException;

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes rear element from this queue and returns it.

**boolean** isFull();

// Returns true if this queue is full; otherwise, returns false.

**boolean** isEmpty();

// Returns true if this queue is empty; otherwise, returns false.

**int** size();

// Returns the number of elements in this queue.

}

**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/10/21

\*

\* LLNode.java: implements <T> nodes for a Linked List.

\*/

**public** **class** LLNode<T>

{

**protected** LLNode<T> link;

**protected** T data;

/\*\*

\* LLNode() is a zero-parameter constructor for LLNode, which sets this node's data and link variables to

\* reference null.

\*/

**public** LLNode()

{

data=**null**;

link=**null**;

}

/\*\*

\* LLNode(T d) is a one-parameter constructor for LLNode, which sets this node's data to d and sets this node's

\* link variable to reference null.

\*

\* **@param** d

\*/

**public** LLNode(T d)

{

data = d;

link = **null**;

}

/\*\*

\* LLNode(T d, LLNode n) is a two-parameter constructor for LLNode, which sets this node's data to d and sets this

\* node's link to n.

\*

\* **@param** d

\* **@param** n

\*/

**public** LLNode(T d, LLNode n)

{

data=d;

link=n;

}

/\*\*

\* setLink sets this node's link to link.

\*

\* **@param** link

\*/

**public** **void** setLink(LLNode<T> link)

{

**this**.link = link;

}

/\*\*

\* setData sets this node's data to d.

\*

\* **@param** d

\*/

**public** **void** setData(T d)

{

data = d;

}

/\*\*

\* getData gets this node's data.

\*

\* **@return**

\*/

**public** T getData()

{

**return** data;

}

/\*\*

\* getLink gets this node's link.

\*

\* **@return**

\*/

**public** LLNode<T> getLink()

{

**return** link;

}

}

**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\*

\* QueueOverflowException represents an exception that is thrown when an enqueue operation is requested for a full queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueOverflowException **extends** RuntimeException

{

/\*\*

\* QueueOverflowException() is a conventional zero-parameter constructor for QueueOverflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueOverflowException()

{

**super**();

}

/\*\*

\* QueueOverflowException(String message) is a one-parameter constructor for QueueOverflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueOverflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.UtilitiesForWorkingWithDoubleEndedQueues;

/\*\*

\* QueueUnderflowException represents an exception that is thrown when an dequeue operation is requested for an empty

\* queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueUnderflowException **extends** RuntimeException

{

/\*\*

\* QueueUnderflowException() is a conventional zero-parameter constructor for QueueUnderflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueUnderflowException()

{

**super**();

}

/\*\*

\* QueueUnderflowException(String message) is a one-parameter constructor for QueueUnderflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueUnderflowException(String message)

{

**super**(message);

}

}