Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
Final Examination: Part 2 of 2

Guy-Vincent Jourdan and Mehrdad Sabetzadeh

April 2021, duration: 110 minutes (exam) + 10 minutes (submission) = 120 minutes

Instructions

1. This is an open book examination and the only authorized resources are:
   - Course lecture notes, laboratories, and assignments;
   - Java Development Kit (JDK) on your local computer.
2. By submitting this examination, you agree to the following terms:
   - You understand the importance of professional integrity in your education and future career in engineering or computer science.
   - You hereby certify that you have done and will do all work on this examination entirely by yourself, without outside assistance or the use of unauthorized information sources.
   - Furthermore, you will not provide assistance to others.
3. Anyone who fails to comply with these terms will be charged with academic fraud.

Marking scheme

Question	Maximum
1	**60**
2	**40**
Total	**100**

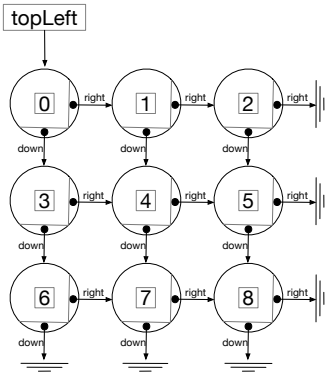
Question 1 (60 marks)

For this question, you will write a program that transforms a given two-dimensional array of objects (of a generic type `E`) into a **grid-like linked structure**. You will then implement certain operations as well as an iterator over the resulting linked structure. Below, we illustrate the transformation for a two-dimenstional array of integer objects (i.e., `Integer[][]`) and a two-dimenstional array of string objects (i.e., `String[][]`), respectively.

`Integer[3][3]`

0	1	2
3	4	5
6	7	8

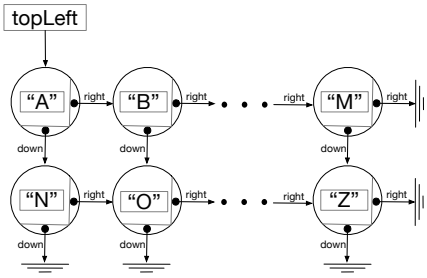
Linked Grid Representation



`String[2][13]`

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"	"I"	"J"	"K"	"L"	"M"
"N"	"O"	"P"	"Q"	"R"	"S"	"T"	"U"	"V"	"W"	"X"	"Y"	"Z"

Linked Grid Representation



Each node in the grid is an instance of the (nested) `Node` class, shown below. As seen from the declaration of `Node`, each instance of `Node` points to two other `Node` instances: the node immediately to its right and the node immediately below it. The rows in the grid are terminated with a null `right` pointer; the columns are terminated with a null `down` pointer, as illustrated in the above figure. The grid has a **perfectly rectangular shape**; there are **no missing nodes** in the grid structure.

```
public static class Node<T> {
        private T data;
        private Node<T> right, down;
        ... // Details removed due to space; see the Node class in LinkedGrid.java
}
```

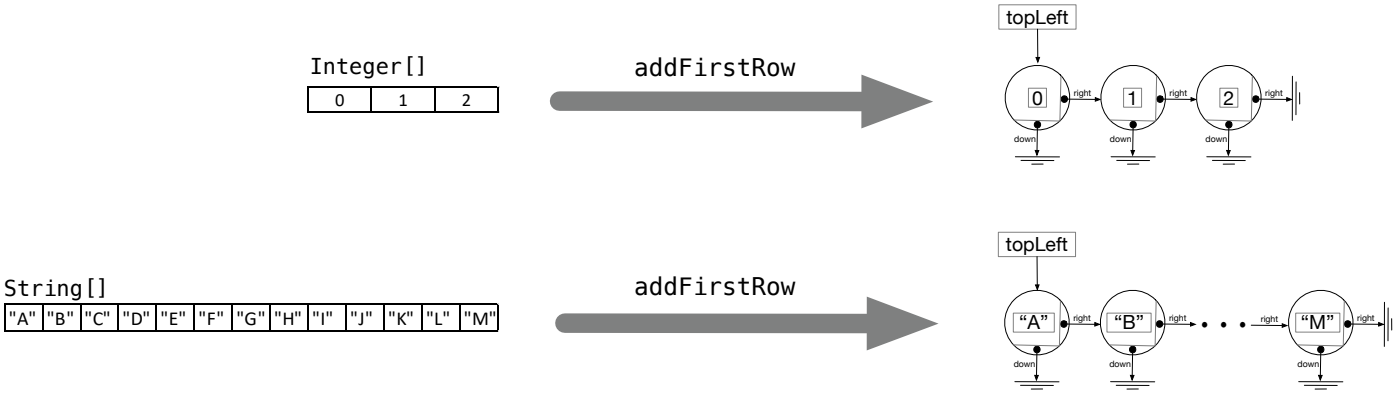
Question 1 is decomposed into five subquestions: Q(uestions) 1.1 to 1.5. **The implementation of Question 1 will all be done in `LinkedGrid.java`.** We have provided you with a shell implementation that includes: (1) the declaration of the `Node` class (shown above), (2) the signatures of the methods you need to implement in Q1.1 to Q1.4, (3) the skeleton code (including the `Iterator` interface) for the `LinkedGridIterator` class in Q1.5, (4) the complete implentation of the `toString()` method for `LinkedGrid`[1], and (5) a `Q1Test` class that allows you to test your implementation.

Important! Question 1 does *not* require any exception handling beyond what is provided to you in the shell implementation.

[1] `toString()` depends on `getElementAt(...)` in Q1.4.

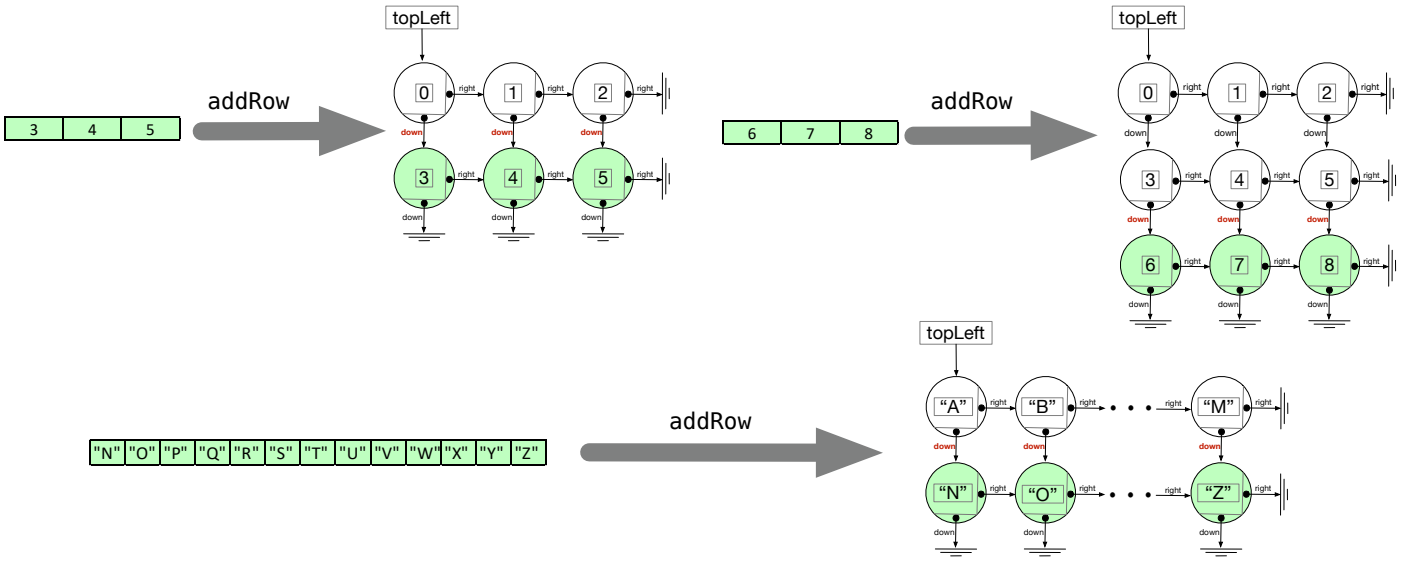
Question 1.1 Method addFirstRow(E[] array) and Constructor LinkedGrid(E[] array)

Complete the (private) `addFirstRow(E[] array)` method. When this method is called (by the `LinkedGrid` constructors), the first row of the grid is created and the `topLeft` instance variable in `LinkedGrid` is made to point to the first node in this row. The expected behaviour of the method is illustrated in the figure below. **In addition,** the method should set the `rowCount` and `columnCount` instance variables in `LinkedGrid` (hint: `rowCount` should be set to 1 and `columnCount` should be set to `array.length`). After completing `addFirstRow(E[] array)`, proceed to complete `LinkedGrid(E[] array)` (hint: you need to add only a single line of code to `LinkedGrid(E[] array)`).



Question 1.2 Method addRow(E[] array)

Complete the `addRow(E[] array)` method. When called, this method appends a new row to the bottom of an existing grid. This behaviour is illustrated in the figure below. Notice that the `addRow(E[] array)` method **can be called iteratively**, with each call adding one row at the bottom of the grid. To build the full integer grid in our illustration, `addRow` needs to be called twice, once with $[3, 4, 5]$ as parameter and then with $[6, 7, 8]$ as parameter. Notice that the first row (that is, $[0, 1, 2]$) needs to have been added a priori by calling `addFirstRow(E[] array)` developed in Q1.1. Calling `addRow(E[] array)` should further increase the `rowCount` instance variable by one.



Question 1.3 Constructor LinkedGrid(E[][] array)

Equipped with `addFirstRow(E[] array)` in Q1.1 and `addRow(E[] array)` in Q1.2, complete the `LinkedGrid(E[][] array)` constructor. The constructor takes as input a two-dimensional array and builds a linked-grid representation of that array. If this constructor is implemented properly using `addFirstRow(E[] array)` and `addRow(E[] array)`, the `rowCount` and `columnCount` variables should already be dealt with without additional code in the constructor.

Question 1.4 Method getElementAt(int i, int j)

Complete the `getElementAt(int i, int j)` method. This method returns the object stored at row `i` and column `j`. For example, calling `getElementAt(1, 2)` over our illustrative integer grid will return an Integer instance with value 5; and, calling `getElementAt(0, 12)` over our illustrative string grid will return "M".

Question 1.5 Class LinkedGridIterator

Complete the `LinkedGridIterator()` constructor and the `hasNext()` and `next()` methods in `LinkedGridIterator`. As you do so, also define any necessary instance variable(s) (in `LinkedGridIterator`) that are used by `hasNext()` and `next()`. The iterator will return objects starting at `topLeft`, progressing through the nodes in the first row, then moving on to the nodes in the second row and so on. For instance, the sequence returned via iterating over our example integer grid will be $0, 1, 2, 3, 4, 5, 6, 7, 8$, and the sequence returned via iterating over our example string grid will be "A", "B", "C", ..., "X", "Y", "Z".

Example Output

To test your implementation, run `Q1Test`. If your answers to Q1.1–Q1.5 are correct, the output produced will be as shown below:

```
$ javac Q1Test.java
$ java Q1Test
==== Integer grid with 3 rows x 3 columns ====
[Test LinkedGrid(E[] array) / addFirstRow(...) and addRow(...)]
getTopLeft().getData(): 0
getTopLeft().getRight().getRight().getDown().getData(): 5
getTopLeft().getDown().getDown().getData(): 6
getTopLeft().getDown().getRight().getDown().getRight().getData(): 8

[Test getElementAt(...)] Print integer grid via toString() which uses getElementAt(...)
0, 1, 2
3, 4, 5
6, 7, 8

[Test LinkedGridIterator] Print integer grid via an iterator
0, 1, 2, 3, 4, 5, 6, 7, 8

==== String grid with 2 rows x 13 columns ====
[Test LinkedGrid(E[] array) / addFirstRow(...) and addRow(...)]
getTopLeft().getData(): A
getTopLeft().getRight().getRight().getRight().getRight().getDown().getData(): R
getTopLeft().getDown().getRight().getData(): O
getTopLeft().getDown().getRight().getRight().getRight().getRight().getRight().getRight().getData(): T

[Test getElementAt(...)] Print string grid via toString() which uses getElementAt(...)
A, B, C, D, E, F, G, H, I, J, K, L, M
N, O, P, Q, R, S, T, U, V, W, X, Y, Z

[Test LinkedGridIterator] Print string grid via an iterator
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
```

```
==== Integer grid with 3 rows x 3 columns ====
[Test LinkedGrid(E[][] array)]
getTopLeft().getData(): 0
getTopLeft().getRight().getRight().getDown().getData(): 5
getTopLeft().getDown().getDown().getData(): 6
getTopLeft().getDown().getRight().getDown().getRight().getData(): 8

[Test getElementAt(...)] Print integer grid via toString() which uses getElementAt(...)
0, 1, 2
3, 4, 5
6, 7, 8

[Test LinkedGridIterator] Print integer grid via an iterator
0, 1, 2, 3, 4, 5, 6, 7, 8

==== String grid with 2 rows x 13 columns ====
[Test LinkedGrid(E[][] array)]
getTopLeft().getData(): A
getTopLeft().getRight().getRight().getRight().getDown().getData(): R
getTopLeft().getDown().getRight().getData(): O
getTopLeft().getDown().getRight().getRight().getRight().getRight().getRight().getRight().getData(): T

[Test getElementAt(...)] Print string grid via toString() which uses getElementAt(...)
A, B, C, D, E, F, G, H, I, J, K, L, M
N, O, P, Q, R, S, T, U, V, W, X, Y, Z

[Test LinkedGridIterator] Print string grid via an iterator
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
$
```

Important Restrictions for Question 1

- You **cannot** define any new class variable or instance variable in `LinkedGrid`.[2]
- You **cannot** import anything at all, except what has been already imported.[3]
- You **cannot** use any other class than the ones we provide.
- You submit only one version of **LinkedGrid.java**, containing your answer to all five questions.
- You can change `Q1Test.java` if you want to perform additional testing, but for your submission, leave `Q1Test.java` just as you found it in the template code.

Files

- `LinkedGrid.java` (you need to update this file).
- `Iterator.java`
- `Q1Test.java`

[2] You need and are thus pemitted to define instance variables in the **iterator** class, i.e., `LinkedGridIterator`.

[3] `java.util.NoSuchElementException`

Question 2 (40 marks)

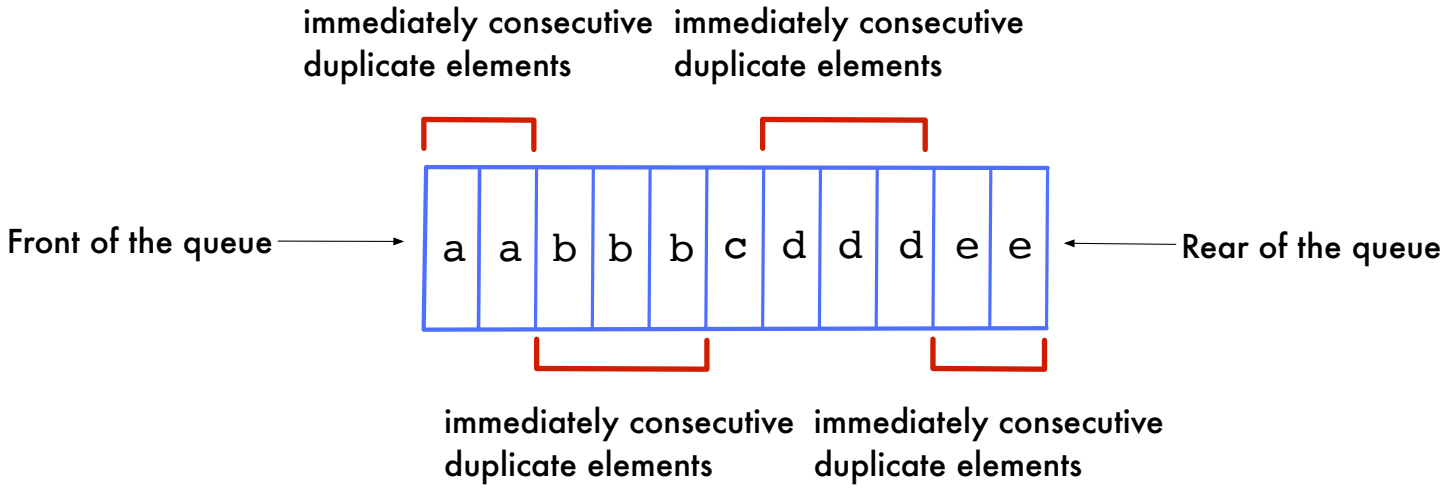
For this question, you are going to develop a **specialization (subclass) of LinkedQueue**. LinkedQueue has been already covered in the course lectures. The subclass, named **UniquifiableLinkedQueue**, provides a single additional method, named **uniquify()**.

```
public class UniquifiableLinkedQueue<E> extends LinkedQueue<E> {
        public Queue<E> uniquify() {
          // All the code that you write for Question 2 goes here!
        }
}
```

The method **uniquify()** does the following:

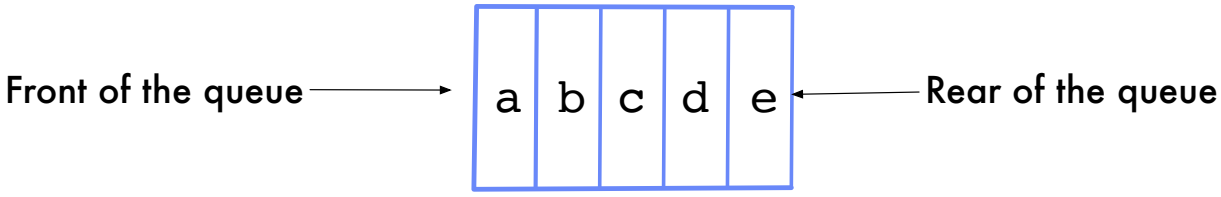
It returns a queue which is the same as the UniquifiableLinkedQueue instance that the method is called on, except that the returned queue has no immediately consecutive duplicate elements.

To illustrate what "immediately consecutive duplicate elements" are, consider the example queue below:



More precisely, two queue elements are immediately consecutive duplicate elements, when they are adjacent and have identical content. Having identical content for (non-null) queue elements `elem1` and `elem2` means that `elem1.equals(elem2)` is true. In other words, the concept of immediately consecutive duplicate elements is exactly the same as that in Q2 of the midterm. However, whereas Q2 in the midterm was concerned with stacks, here, we are working with queues. **You can assume that all queue elements are non-null.**

In Question 2, you complete the method **uniquify()** in the **UniquifiableLinkedQueue** class, so that only one instance of immediately consecutive duplicate elements would be retained. For example, if **uniquify()** is called over the example queue above (an instance of `UniquifiableLinkedQueue<String>`), the result should be as shown in the figure below.



Note that **uniquify() should not have any side effect** on the **UniquifiableLinkedQueue** instance that the method is called on. Furthermore, **uniquify()** is **not meant to deal with non-consecutive duplicate elements**. In other words, the queue returned by **uniquify()** can have non-consecutive duplicate elements. The absence of side effects on the original queue as well as the carry-over of non-consecutive duplicate elements to the result of **uniquify()** are illustrated in the example output by **Q2Test**, presented next.

Example Output

To test your implementation of **uniquify()**, you can use the **Q2Test.java** in the template code provided to you. The output from running **Q2Test** should be as follows:

```
Original Integer Queue: Front -> [0, 1, 1, 1, 2, 2, 3, 3, 3, 4] <- Rear
Integer Queue without immediately consecutive duplicates: Front -> [0, 1, 2, 3, 4] <- Rear
Original Integer Queue (after uniquify): Front -> [0, 1, 1, 1, 2, 2, 3, 3, 3, 4] <- Rear

Original String Queue: Front -> [a, a, b, b, b, c, d, d, d, e, e] <- Rear
String Queue without immediately consecutive duplicates: Front -> [a, b, c, d, e] <- Rear
Original String Queue (after uniquify): Front -> [a, a, b, b, b, c, d, d, d, e, e] <- Rear

---- Now, testing with some non-consecutive duplicates -----

Original String Queue: Front -> [a, b, b, c, a, d, d, e, e, d, d, d, d, b] <- Rear
String Queue without immediately consecutive duplicates: Front -> [a, b, c, a, d, e, d, b] <- Rear
Original String Queue (after uniquify): Front -> [a, b, b, c, a, d, d, e, e, d, d, d, d, b] <- Rear
```

Important Restrictions for Question 2

- You **cannot** change either `LinkedQueue.java` or `Queue.java`. You can change `Q2Test.java` if you would like to perform additional testing, but for your submission, please leave `Q2Test.java` just as you found it in the template code.
- All your variables should be **local variables**. You **cannot** declare any class or instance variables in **UniquifiableLinkedQueue** (or anywhere else for that matter).
- The local reference variables in **uniquify()** as well as in any helper private methods that you may implement can only be of type **E** (that is, the generic type parameter) or of type **Queue<E>**. Stated otherwise, you **cannot** declare any reference variable that has a different type than either **E** or **Queue<E>**.
- You are allowed to use local primitive variables if you find them necessary, but please note that it is feasible to implement **uniquify()** *without* using any local primitive variables.

Files

- `UniquifiableLinkedQueue.java` (you need to update this one).
- `LinkedQueue.java`
- `Queue.java`
- `Q2Test.java`

Rules and regulation

- Submit your examination through **Brightspace**.

- You must do this examination individually.
- You must use the provided template classes.
- It is your responsibility to make sure that Brightspace has received your examination.
- Late submissions will not be graded.

Files

- Download the archive **f2_3000000.zip**;
- Unzip the file and rename the directory, replacing **3000000** by **your student id**;
- Add your **name** and **student id** in a comment in **LinkedGrid.java** and **UniquifiableLinkedQueue.java**;

You must submit a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **f2_3000000**, where 3000000 is your student number. The name of the folder starts with the letter "f" (lowercase), followed by 2, since this is part 2 of the final examination. The segments are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. Your submission must contain the following files, and nothing else. In particular, do not submit the byte-code (.class) files.

- README.txt
  - A text file that contains your name and student id
- `LinkedGrid.java` (you need to update this file)
- `Iterator.java`
- `Q1Test.java`
- `UniquifiableLinkedQueue.java` (you need to update this file)
- `LinkedQueue.java`
- `Queue.java`
- `Q2Test.java`