Homework # 7
due Monday, October 28, 2019

Due after exam week, but we *strongly* recommend you finish this before the exam.

In this homework you will implement a queue using a circular array. The textbook describes the technique on pages 383–393 (3rd ed. pp. 372–382), but we change the data structure somewhat. You will also complete the implementation of a Tetris block supplier that uses your ADT.

1 Queue ADT

For this assignment, a Queue will provide the following operations:

boolean isEmpty() Return true if the queue is empty.

int size() Return the number of elements in the queue.

E front() Return the element at the front of the queue.

void enqueue(E) Add something at the back of the queue.

E dequeue() Remove the front element from the queue and return it.

If one tries to access or remove the front element when the queue is empty, a `NoSuchElementException` exception should be thrown. We provide a skeleton file for Queue, which gives the precise signatures to use for the methods.

The data structure for this assignment is a circular buffer built using dynamic arrays. This data structure retains the efficiency of an array (hardly any dynamic allocation, and direct addressing), and gains some limited additional benefit: there are now two places we can add or remove in constant time without changing the order of the data. An array is still efficient here because, even though we need to preserve order, because we never add or remove from the middle of the data; there is no need for copying elements within the array.

The implementation makes use of three fields:

data An array of values.

front The index of the frontmost element.

manyItems The number of elements stored in the array.

It does not (and *must* not) include a back or tail field (as in the book's implementation). This means that, when we need to find the back of the queue, we must compute the relevant index using `front` and `manyItems`.

We also reverse the direction of the queue in the array. If the front element of the queue is at index `front`, the successive elements are encountered as you *decrease* the index, rather than increase as in the textbook. This is done as an exercise, and not for any useful design reason.

2 Tetris Block Supplier

We will use the queue you design in an application that can supply blocks for a game of Tetris. In a typical game of Tetris, the player is given a current piece, and shown a preview of the next piece that they will receive. But there are different ways that the blocks can be supplied. We will explore four of these.

random Each piece is generated uniformly at random from the set of pieces.

in order There is a set order through the 7 pieces, and this order is repeated indefinitely.

7 supply Pieces are supplied in successive series of 7. Each series contains one of each of the 7 pieces, in a random order.

14 supply Pieces are supplied in successive series of 14. Each series contains two of each of the 7 pieces, in a random order.

In order to achieve this, we first need a way to generate a random piece. The `Block` class has a static `RBG_TABLE`, which is an array of pieces containing an equal number of each piece in a random order. Our program will copy all of these pieces into a queue called `random`. Whenever we want to generate a random piece, we can pull a piece from the queue to use. But we should put that piece back into the queue so that our random queue doesn't run out. This is not truly random in a statistical sense, but we often call this pseudorandom generation.

Likewise, we will create a queue called `fixed` that we can use for our `in order` supply mode. This queue will contain the 7 blocks in a fixed order.

Finally, we will have a third queue called `pending`. This is the queue that holds the next pieces that the player will receive. This queue should always contain at least one element, so that the player can see the next piece that they will receive. Whenever we are about to remove the last piece, we should add new pieces according to the supply mode. If using `in order` or `7 supply`, we should add 7 pieces to the pending queue. If using `14 supply` we should add 14, and if using `random`, we only need to add one new piece.

`BlockQueue.java` is a runnable application. It will allow you to switch supply modes, view the current and next pieces, and request a new piece. When you change supply modes, you should leave the pending queue as it currently is, but you should expect newly added pieces to follow the newly selected supply behavior.

You should test your code thoroughly using the application. Try to verify that the pieces appear in the same order, randomly, or in random groupings of the 7 pieces, as appropriate. Remember that if you switch to `random`, you will have to empty out your pending queue before you start seeing randomly generated pieces. The application uses your `toString` method to display the contents of the pending queue. This is where you can verify that your `toString` is working. If it is working, it will be much easier to verify that your supply methods are working apprpriately.

Also remember that you can debug your running application by putting a breakpoint into your code, and looking at the variables view. You might have to turn on "Show Static Variables" in your variables view to see the queues in `BlockQueue`.

You may wish to review how to use an `enum` in Java, especially regarding things like `values()` and `ordinal()`.

3 What You Need To Do

You need to implement the `Queue` ADT with the data structure specified (one array, two int variables). We do a few things differently from the book. Do not copy code blindly from the book! Your code should use `if` *sparingly*: follow the directions in the comments. The methods all should be efficient. The skeleton file for Queue.java includes comments where `if` and loops are permitted.

You also need to complete the tetris block supplier program by completing BlockQueue.java.

4 Files

We provide the following files (among others) in your `homework7.git` repository:

src/edu/uwm/cs351/util/Queue.java A skeleton for the Queue ADT.

src/edu/uwm/cs351/BlockQueue.java A partly completed implementation of a Tetris block supplier. This is runnable.

src/edu/uwm/cs351/Block.java The enum class for the Tetris blocks.

src/*.java Tests for the Queue ADT.