Project 2

DUE: September 24th at 11:59pm

Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided).
- Have a style (indentation, good variable names, etc.) and pass the provided style checker (see P0).
- Comment your code well in JavaDoc style and pass the provided JavaDoc checker (see P0).
- Have code that compiles in your user directory without errors or warnings.
- For methods that come with a big-O requirement, make sure your implementation meets the requirement.
- Have code that runs (see detailed commands below).

You may:

- Add additional methods and class/instance variables, however they **must be private**.
- Import these classes only: `File`, `FileNotFoundException`, `Iterator`, `PrintWriter`, `Scanner`

You may NOT:

- Make your program part of a package
- Add additional public/protected/package-private methods or variables
- Declare/use arrays of any type anywhere in your program
- Use any Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, etc.)
- Alter any class/method signatures defined in this document or the template code
- Add any additional import statements (or use the “fully qualified name” to get around adding import statements)
- Add any additional libraries/packages
- Create any additional classes or interfaces
- Use nested classes or lambda expressions

Setup

- Download the **`p2.zip`** and unzip it. It contains a template for all the files you must implement
- Edit the **`readme.txt`** file to reflect your own information

Submission Instructions

- Make a new temporary folder and copy there your **`.java`** files and your readme.txt (do not copy the test files, jar files, class files, etc.)
- Upload the temporary folder to OneDrive as a backup (this is not your submission, just a backup!)
- Follow the Gradescope link provided in Blackboard>Projects and upload the files from your temporary folder onto Gradescope’s submission site (do not zip the files or the folder)

Grading

Grading will be divided into two portions:

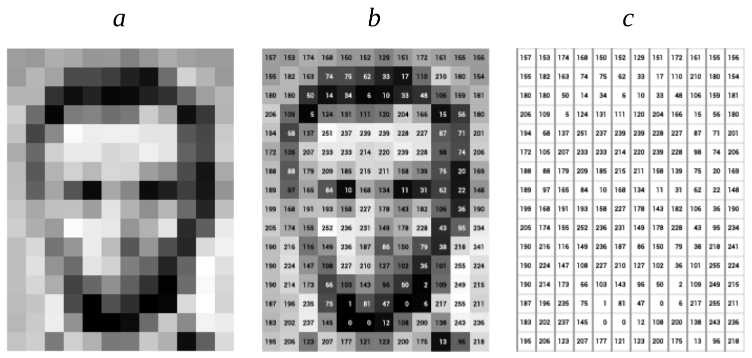
- Unit Testing (80%): To assess the correctness of programs.
- Manual Inspection (20%): A checklist of features your programs should exhibit that cannot be easily checked via unit tests (e.g. good variable name selection, proper decomposition of a problem into multiple functions or cooperating objects, overall design elegance, and proper asymptotic complexity)

Extra credit for early submissions

- 1% extra credit rewarded for every 24 hours your submission made before the due time. Max reward is 5%
- Your latest submission will be used for grading and extra credit checking. You can’t choose which one counts

Overview

A digital image with dimensions *W*x*H* is represented with a **matrix of pixels** that has *W* columns and *H* rows. In grayscale images, each pixel is an integer that takes values from 0-255 (i.e. 1 byte). A value of zero corresponds to the black color and a value of 255 corresponds to the white color. Everything in between is a tone of gray; the darker the gray the closer to 0, the lighter the gray the closer to 255. Figure 1a depicts a zoom-in on a 12x16 grayscale image. In Figure 1b you can see the values of each pixel superimposed on the image. And in Figure 1c you can see just the matrix representation of this grayscale image. In this project we will not work with color images but everything we discuss here could equally well apply to color images too.



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Figure 1: Grayscale image

Your main task is to create a **two-dimensional doubly-linked list** to store the pixels of an image and then do some operations on it. Each node in a regular doubly-linked list has two pointers (next, previous), but since this is a two-dimensional doubly-linked list, each node has two extra pointers (up, down), so the total number of pointers is 4. Thus, the data structure will look like the one in Figure 2.

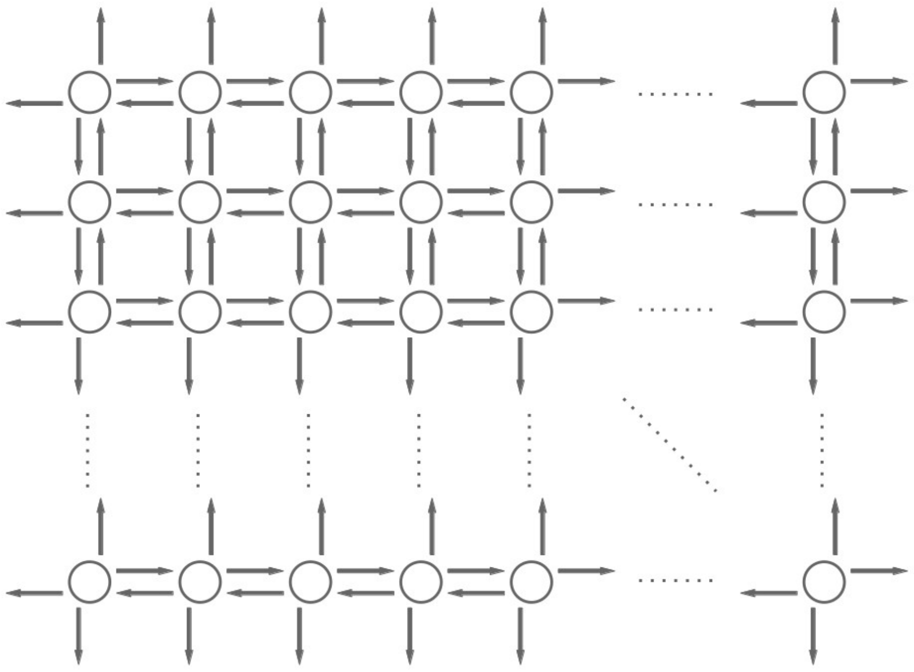


Figure 2: two-dimensional doubly-linked list

Images come in many different formats, but to keep things simple for this project, we will use the PGM format (portable graymap format). PGM images can be stored in either a binary or a text format. We will use the text format to make the reading/editing of the images and the debugging of your code much simpler (*Note*: Real applications usually use binary files because they're more efficient). All major photo viewing/editing applications should support this format (if yours doesn't, you can download GIMP for free). So, it will be easy to visually inspect the input you use and the output you generate. In Figure 3 you can see an example of an actual image stored in a PGM text format. The first row is the code **P2** that indicates a grayscale image in text format (it doesn't change in this project). The second row contains the width and the height of the image (**6** and **4** respectively in this example, but it can be any value in this project). The third row defines the maximum grayscale value which is **255** in this case (always the same in this project). And what follows is simply a series of *width*height* pixel values (either in a single or multiple lines, it doesn't matter).

```
P2

6 4

255

10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240
```

Figure 3: Example of a PGM image content in text format

Open your favorite **plain text** editor (not MS Word), write in it the above values, and save the file as **image.pgm** (it is still in plain text; changing the file extension doesn't change the format of the content). Then, open it with your favorite photo viewing/editing application to verify that it looks like the one in Figure 4. You will need to zoom-in a lot because this image is tiny, just 6x4 pixels.

You can find more information about the PGM format at https://en.wikipedia.org/wiki/Netpbm (but really, you don't need to know more).

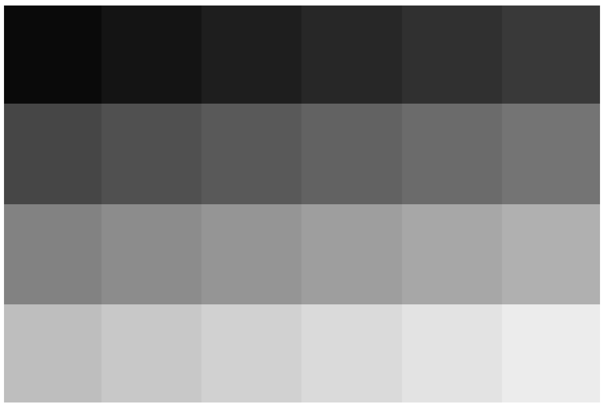


Figure 4: Image rendering of the PGM snippet in Figure 3

A typical scenario of your program execution begins with a call to the `Utilities.loadImage` method which opens a PGM file and reads the image data. When the method reads the dimensions of the image in the second line, it knows what size the data structure should be, and it proceeds to create an `Image` object before reading any pixel values from the subsequent lines of the PGM file. This is because the data structure (i.e. the two-dimensional doubly-linked list) is stored inside the `Image` class and we need it to be ready when we proceed with the reading of the pixel values and the assigning of them to the nodes of the list. Thus, when the Image constructor has completed its job, the Image object does not hold any pixel values yet, but it holds the complete **two-dimensional doubly-linked list** created in memory. In other words, the Image constructor creates a two-dimensional doubly-linked list "container" without setting any values in its nodes. And later, when the `loadImage` continues with the reading of the pixel values from the PGM file, it must start traversing the two-dimensional doubly-linked list and start setting the values of the nodes one by one. This traversal of the two-dimensional doubly-linked list can be done with an enhanced-for loop, provided of course that the Image class has implemented the Iterable interface.

Classes to be implemented

`public final class Node<T extends Comparable<T>> implements Comparable<Node<T>>`

This class represent a single node in the list. It is generic of course and, because we want to be able to compare any two nodes with each other, it implements the `Comparable` interface. This is a functional interface which means that it has one method only. For more details see https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

The comparison of any two nodes returns the result of the comparison of their `data` field. This means that the generic type `T` must implement the Comparable interface as well. As a result, the declaration of a simple node becomes quite long and ugly. Each node has the following members:

`private T data`
Holds the data of the node

`private Node<T> up`
The pointer to the node above

`private Node<T> down`
The pointer to the node below

`private Node<T> right`
The pointer to the node on the right

`private Node<T> left`
The pointer to the node on the left

`public void setValue(T value)`
Setter method for the `data` field

`public T getValue()`
Getter method for the `data` field

`public Node<T> getUp()`
Getter method for the `up` field

`public Node<T> getDown()`
Getter method for the **`down`** field

`public Node<T> getRight()`
Getter method for the **`right`** field

`public Node<T> getLeft()`
Getter method for the **`left`** field

`public void setUp(Node<T> p)`
Setter method for the **`up`** field

`public void setDown(Node<T> p)`
Setter method for the **`down`** field

`public void setRight(Node<T> p)`
Setter method for the **`right`** field

`public void setLeft(Node<T> p)`
Setter method for the **`left`** field

`public Node()`
constructor that sets everything to null

`public Node(T value)`
constructor that sets the data field only and everything else to null

`public final class Utilities`
It contains two utility methods for reading/writing images from/to files. This class is not generic. It can only load/save images that have pixel values of type **`Short`**.

`public static Image<Short> loadImage(String pgmFile)`
It reads a PGB image from a file and creates an `Image<Short>` object. After reading the second line in the file, it calls the constructor of the `Image` to instantiate an `Image` object. Once the data structure of the image (i.e. the two-dimensional doubly-linked list) is created in memory, it continues the reading of the file and uses an `ImageIterator` to traverse the image row by row and store the value of each pixel in the respective node.

Time-compexity: `O(height * width)`

Throws a `RuntimeException` if the file doesn't exist

`public static void saveImage(Image<Short> image, String pgmFile)`
It writes an `Image<Short>` object into a PGM file. It uses an `ImageIterator` to traverse the image row by row, and writes the value of each pixel into a PGM-formatted file.

Time-compexity: `O(height * width)`

Throws a `RuntimeException` if the file doesn't exist

`public class Image<T extends Comparable<T>> implements Iterable<Node<T>>`

It implements a generic two-dimensional doubly-linked list. The class must implement the `Iterable` interface in order to allow iterations on its data.

`private Node<T> head`

It points to the top-left corner of the two-dimensional doubly-linked list

`public Image(int width, int height)`

The constructor builds a two-dimensional data structure that has a rectangular shape. All the rows must have the same number of nodes. The constructor does not assign values to the nodes.

Time-compexity: `O(height * width)`

Throws a `RuntimeException` if `height` or `width` are invalid

`public int getHeight()`

Getter method for the height of the Image

`public int getWidth()`

Getter method for the width of the Image

`public Node<T> getHead()`

Getter method for the head of the Image

`public void insertRow(int index, T value)`

Inserts a single row that has the same value for all its nodes. The maximum value of index is k where k is the number of rows (i.e. insert is allowed to behave like append).

Time-compexity: `O(height + width)`

Throws a `RuntimeException` if `index` is invalid

`public void removeColumn(int index)`

Removes a single column from the data structure based on its index

Time-compexity: `O(height + width)`

Throws a `RuntimeException` if `index` is invalid

`public int compress()`

If two adjacent rows or columns have the same values, it removes one of the two from the image. The whole process repeats until no adjacent rows or columns have the same values.

Returns the total number of nodes removed from the image

Time-compexity: `O(height * width)`

`public void addBorder()`

Adds a border to the perimeter of the image. The border has a width of 1 pixel and, thus, it increases the height and the width of the image by 2 pixels in each dimension. The value of each pixel in the border is the same with its adjacent pixel in the original image. For example (see Figure 5), if yellow is the original image, the border will be the one depicted in green.

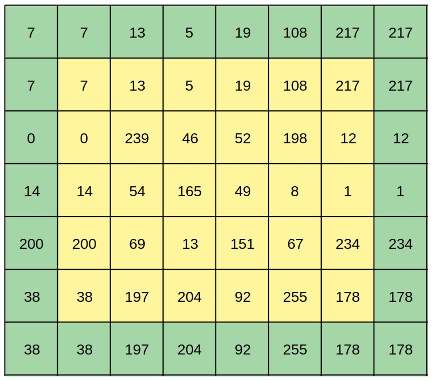
Time-compexity: `O(height + width)`

`public void removeBorder()`

Removes the borderline pixels from the image. The border has a width of 1 pixel and, thus, it decreases the height and the width of the image by 2 pixels in each dimension.

Time-compexity: `O(height + width)`

Throws a `RuntimeException` if the image `height` or `width` is too small to support this operation

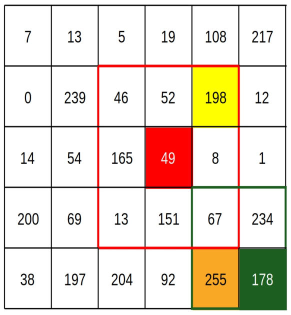


7	7	13	5	19	108	217	217
7	7	13	5	19	108	217	217
0	0	239	46	52	198	12	12
14	14	54	165	49	8	1	1
200	200	69	13	151	67	234	234
38	38	197	204	92	255	178	178
38	38	197	204	92	255	178	178

Figure 5: Border (green) added to image (yellow)

`public Image<T> maxFilter()`

It creates a new image that has the same size with the original, but the value of each pixel is replaced by the maximum value in its neighborhood. The default neighborhood is a 3x3 region centered on the pixel examined, but it will be smaller for pixels close to the border and the corners. See Figure 6 for an example. In the new/returned image, the value of the red pixel (49) will be replaced by the maximum value in its 3x3 neighborhood which is the yellow pixel (198). But the value of the green pixel (178) will be replaced by the orange pixel (255) because the 3x3 neighborhood that is centered on the green pixel has only 4 pixels inside the image.



7	13	5	19	108	217
0	239	46	52	198	12
14	54	165	49	8	1
200	69	13	151	67	234
38	197	204	92	255	178

Figure 6: Max filter example

Time-compexity: `O(height*width)`

`public Iterator<Node<T>> iterator()`

The default **`iterator`** required by the implementation of the **`Iterable`** interface. When this method is called, it creates and returns an **`ImageIterator`** object that traverses the image horizontally as depicted in Figure 7.

`public Iterator<Node<T>> iterator(Direction dir)`

An overloaded **`iterator`** that can set the **`Direction`** of the iteration to one of two options, either **`HORIZONTAL`** (depicted in Figure 7) or **`VERTICAL`** (depicted in Figure 8). When this method is called, it creates and returns the same **`ImageIterator`** object as before. The only difference is the arguments it passes to its constructor.

`public String toString()`

Useful for debugging purposes but not required

`public class ImageIterator<T extends Comparable<T>> implements Iterator<Node<T>>`

Implements an iterator that can traverse the two-dimensional doubly-linked list in different directions. The default direction is `HORIZONTAL`, as depicted in Figure 7, and an alternative direction is `VERTICAL`, as depicted in Figure 8. When the **`Image.iterator`** method is called, it creates an instance of this class. This instance handles all the logic of the iteration. In other words, the **`Image.iterator`** method is nothing but just a gateway to this class. Of course, you need to add constructor(s) here so that the instance is created properly and has access to the Image object that invoked the constructor(s).

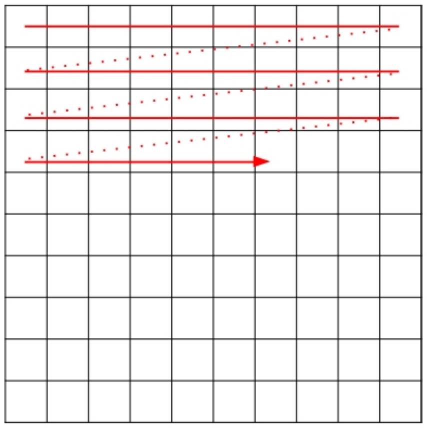


Figure 7: Horizontal iteration

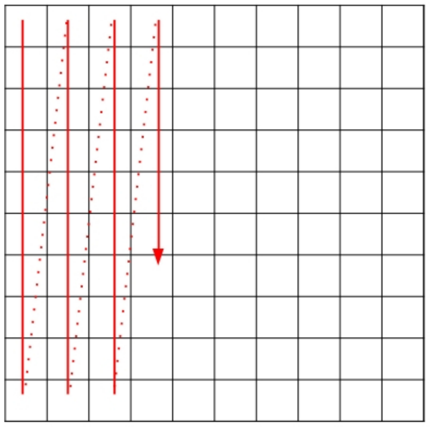


Figure 8: Vertical iteration

Additionally, you must implement the logic of the iteration by implementing the following two methods:

`public boolean hasNext()`

Returns true if the iterator has more elements to consume, or false if the iteration is over.

Time-compexity: `O(1)`

`public Node<T> next()`

Returns the next node of the data structure based on the traversal, horizontal or vertical, that the iterator is doing. Keep in mind that a typical implementation of a data structure would return the **`data`** field only and not the whole node. But we will violate the encapsulation principle here in order to get better practice with linked lists.

Time-compexity: `O(m)` where **`m`** is the number of rows or columns depending on whether we do a **`VERTICAL`** versus a **`HORIZONTAL`** traversal respectively.

`public enum Direction`

An enumeration that has two constants only: `HORIZONTAL` and `VERTICAL`