9 Direct Access Data Structures

Practice Problems

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Finish Lab 8.

Working with the StringTokenizer

Set up a simple project, designing your solutions in the `Algorithms` class. Add to your project the class **Words.java** from the assignment web site.

1. Look up the `StringTokenizer` class in JavaDocs. The methods there allow you to traverse over a `String` and produce one word at a time delimited by the selected characters. Read the examples. Then write the method `makeWords` that consumes one `String` (that represents a sentence with several words, commas, and other *delimiters* and produces an `ArrayList<String>` of words (`String`s that contain only letters — we ignore the possibility of words like "don't"). The delimiters you should recognize are the comma, the semicolon, and the question mark.

2. The text in the`ArrayList<String> words` in the class `Words` is a secret encoding. It represents verses from a poem - if you read only the first words. Design the method `firstWord` that produces the first word from a given `String`. Use it to decode the poem.

Pair Programming Assignment

9.1 Insertion Sort

We have seen the recursively defined *insertion sort* algorithm both in the first semester and also recently, using the recursively defined lists in Java.

The main idea behind the insertion sort was that each new item has been inserted into the already sorted list. We can modify this as follows:

1. Design the method `sortedInsert` that consumes a sorted `ArrayList<T>` a `Comparator<T>` that has been used to define the sorted order for the given list, and an item of the type `T`. It modifies the given `ArrayList<T>` by adding the given item to the `ArrayList<T>`, preserving the ordering.

   Note: Be careful to make sure it works correctly when the given `ArrayList` is empty, and when the item is inserted at the end of the `ArrayList`.

2. Design the method `insertionSort` that consumes an arbitrary (unsorted) `ArrayList<T>` and a `Comparator<T>` and produces a new sorted `ArrayList<T>` as follows:

   It starts with an empty sorted list and inserts into it one by one all the elements of the given `ArrayList<T>`.

   Note: It is a bit more difficult to define the insertion sort algorithm so that it mutates the existing `ArrayList` *in place*.

3. **Extra Credit**

   Design an in-place `insertionSort` method. You will get the credit only if the design is neat and clearly organized.

9.2 Mars Images: Image Processing

We ask you to work with with image data, and learn two simple techniques for enhancement of images defined by pixel shades. Additionally, you will learn how secret images can be encoded in an image, and explore the power of colorization of images.

You will read images data files of NASA images of the planet Mars, display the images as received from the Viking Explorer, and by manipulating this data generate enhanced images.

Read the following tutorial/explanation of the techniques you will use. **The detailed description of the classes and methods you should design is at the end of the tutorial.**

The Images

The data in the files **mg20s002**, **mg20s007**, etc. came from a NASA jukebox of planetary images. Each file starts with several lines of text (a label) that identifies the image - the location on Mars, the resolution (how large an area is represented by one pixel), what spacecraft took the image, and the information about the size of the image data (number of lines and the number of pixels per line). After the label is histogram data - specifying how many pixels there are of each shade (gray shade, just like the color shades, has values in the range from 0 to 255). The last part of the file contains the image data: each pixel is represented as one byte.

Your program that manipulates the images should use the given library class `MarsReader`. You will use the following functionality of the class `MarsReader`:

- The constructor for `MarsReader` looks for the original Mars image file, reads the file labels and stores them in the field `labels`.
- The constructor then initializes the field `BufferedInputStream bytestream` to deliver the bytes of the selected image.

To create new images and save then as *.png* files, use the given class `ImageBuilder`. It works as follows:

- The constructor also initializes the field `BufferedImage image` that is ready to receive the data needed to represent the resulting image. You need to supply the height and the width of the image.
- The method `setColorPixel (x, y, r, g, b)` sets the color of the given pixel in the `image` to the specified RGB shade.
- The method `public void saveImage(String filename)` saves the image you have created in the *.png* format — it adds the *.png* to the filename you specify.

Image Processing

Each pixel shade is represented as one byte. You can read one byte of data from the `bytestream` using the method

```
int read()
```

The integer will be in the range from 0 to 255.

All images in this collection have the same size: 320 lines of 306 pixels in each line. You can set the color of each individual pixel in the `BufferedImage image` using the method

```
void setColorPixel(int x, int y, int r, int g, int b)
```

A 'black and white' image is represented by pixels of different shade of gray. By choosing `setColorPixel (x, y, s, s, s)` with values of `s` ranging from 0 to 255 we can represent 256 different shades of gray.

In pictures of low quality the range of the shades is often much smaller than 256. For example, in the images from Mars most of the shades are in the range between about 70 and 170, leaving more than half of the shades unused. Image enhancement methods take advantage of this deficiency.

Linear scaling.

The first method uses linear scaling to modify the shade of each pixel. It starts with computing the minimum and maximum of the existing shades. It then scales each shade so that the range of shades is expanded to 256 values. The scaling formula is:

```
newshade = (oldshade - min) *  (255 / (max - min));
```

That means that in our example (the range between 70 and 170), `oldshade=70` would be represented as `newshade=0`, similarly, `oldshade=170` would be represented as `255`, and finally, `oldshade=100` would be represented as `(100 - 70) * (255 / (170 - 70)) = 30 * 2.55 = 76.5`,or `newshade=76`:

old shade	new shade
70	0
170	255
100	76.5

We do not want to do this computation for every pixel over and over again. Instead, we should save the computed values in a table indexed by the `oldshade` values with the `newshade` values in the table.

Implement the linear scaling image processing and observe the impact on the original image.

Histogram equalization.

The second method is called histogram equalization. Histogram equalization is simply a transformation of the original distribution of pixels such that the resulting histogram is more evenly distributed from black to white.

We start by computing the distribution of the pixel shades (a frequency array or a histogram H). Histogram is a simple count of the number of occurrences of each pixel shade (a frequency chart). (For example a histogram of rolling a die 100 times may tell us that we rolled 1 15 times, 2 18 times, 3 17 times, 4 12 times, 5 15 times and 6 13 times.)

We start by reading all pixel data and building the histogram. Let us assume that h_i is the number of pixels of the shade i and that h is the count of all pixels in the image.

We compute the *scaling factor* s_i of each pixel initially at gray level i as:

$s_i = (1/h) * sum(h_0, h_1, h_2, ..., h_i)$

where h is the total number of pixels and hi is the number of pixels at gray level i (i.e. the histogram data).

Once we have the *scaling factor*s, we compute $newshade_i = 255 * s_i$

Of course, again we do not want to keep recomputing these values and store them in a lookup table instead.

Include in your program a visual display of the histogram you have computed to verify that our assumptions about the color distribution are correct.

Note: You will need to read the Mars data file twice - first just to compute the histogram and set up the mapping of old shades to new ones, the second time, reading the old shades and writing the new shades into the output file.

Colorization

Explore what happens to your image when you add a bit of coloring to it. One way to do it is by replacing the gray shade color

```
new Color(shade, shade, shade);
```

by

```
new Color(shade, 255 - shade, 255 - shade);
```

Color processing

The class `ImageReader` allows you to read any *.bmp* or *.png* file and analyze the individual pixels. The constructor expects the name of the file to be read. It reads the file and initializes the value of the `width` and `height` fields for the given image.

The method `Color getColorPixel(int x, int y)` returns the color value of the pixel at the given location. You can extract the red, blue, and green component of the color as integers using the methods

```
c.getRed()
c.getGreen()
c.getBlue()
```

Create a negative of the given *Flowers.png* image. Explore other ways of manipulating the images and document your exploration.

1. Create a project *MarsImages* and include the files `MarsReader.java`, `ImageBuilder.java`. Include in your project a class `MarsAlgorithms` where you will implement several image processing algorithms.
2. Design the method `setMinMax` that will read a *Mars* image data and determine the minimum (greater than 0) and the maximum shade in the image, i.e. ignore the shade 0 and save the values.
3. Design the method `buildLinearScaleMap` needed to implement the *linear scaling* algorithm described earlier. The method consumes the minimum and maximum shade you have computed and produce an `ArrayList pixelMap` that can be used as a **lookup table** as follows. If the linear scaling algorithm changes the shade $p - old$ to shade $p - new$, then the value of the `pixelMap.get(p-old)` will be `p-new`.
4. Design the method `enhanceMars` that reads a *Mars* image, processes the pixel data and displays (and saves) the image enhanced by deploying the linear scaling algorithm.

   Your method should read the *Mars* image file and create a new image using the `ImageBuilder` or `ImageBuilder2`.
5. Design the method `computeHistogram` that consumes the *Mars* image data and produces an `ArrayList` of frequencies for each color of the pixel (i.e. an `ArrayList` of size 256.
6. Design the method that will display the histogram as a bar chart in a Canvas.
7. Design the method that will consume a histogram for 256 pixel shades and produce an `ArrayList pixelMap` that can be used as a **lookup table** as follows. If the histogram equalization algorithm changes the

shade $p - old$ to shade $p - new$, then the value of the `pixelMap.get(p-old)` will be `p-new`.

8. Add the code needed to produce an even better *Mars* image file by processing the byte data using the *histogram equalization* algorithm: at this point all you need to do is replace each shade by the shade defined by the `pixelMap`.

   Note that you will need to read the Mars data twice.

9. Design a method `colorize` that will colorize the Mars image, i.e. use `MarsReader` to read the *Mars* data and display a colorized image. You may want to experiment with different ways of changing the colors.

10. Design a method `separate` that will read the data for a regular color image using the `ImageReader` and separate the image into its three basic color component, displaying each color in one of three *Canvases* simultaneously.

References

The idea for this lab came from the book by Robert S. Wolff and Larry Yaeger, *Visualization of Natural Phenomena*, Springer Verlag 1993 (TELOS Series)

Thanks also to Peter Ford from MIT who helped us locate the original image data file.

In 1999, The Viking Orbiter and other planetary data files could be found at

```
ftp://pdsimage.wr.usgs.gov/cdroms/
```

We suggest using the files in vo_2002 that start with *mg*. For example:

```
ftp://pdsimage.wr.usgs.gov/cdroms/vo_2002/mg25sxxx/mg25s022.img
```

These files are relatively small, about 100K and contain images that are approximately 300 by 300 pixels. See

```
ftp://pdsimage.wr.usgs.gov/cdroms/vo_2002/volinfo.txt
```

for a description of the file format.