Project 3: Image Processing

Topic: QuadTree & Circular Queue

DUE: Sunday, March 31, 11:59pm

Extra Credit for Early Submission!

Introduction

The QuadTree data structure is a special type of tree where the inner nodes have exactly **four** children. The goal of this tree is to divide a 2D space into four quadrants and have each child represent one of them. Each quadrant can further divide into four smaller quadrants and so forth according to the level of detail we want to achieve. But not all the partitions need to have the same level of detail, i.e. some leafs can be deeper than others. So, in practice, the quadtree is usually not a perfect tree. Figure 1 depicts an example of a quadtree that does the minimum partitioning required in order for each point in space (A, B, C, D, E, F) to be represented by a distinct leaf.

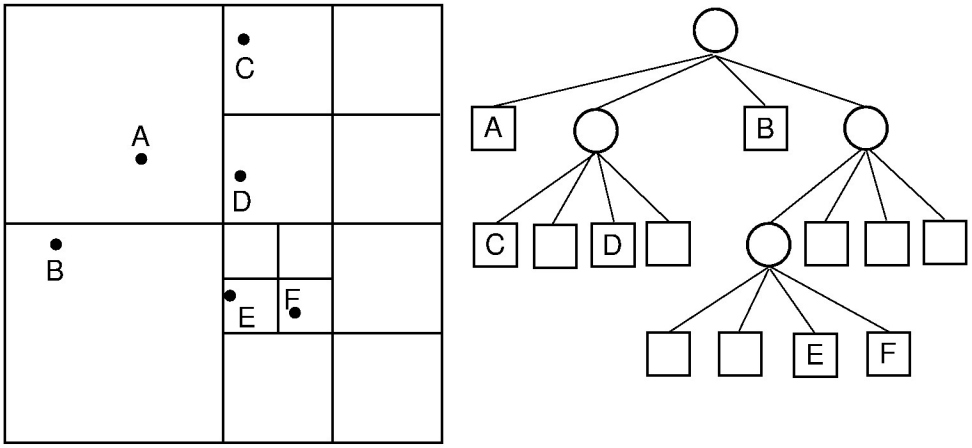


Figure 1: QuadTree partioning of a 2D space

Quadtrees have various applications in domains like image processing, spatial indexing, collision detection, storing of sparse data, and others. This project is inspired by the use of quadtrees in image processing but you don't need to have a background on this topic. All you need to know is described in this document.

Where should I start?

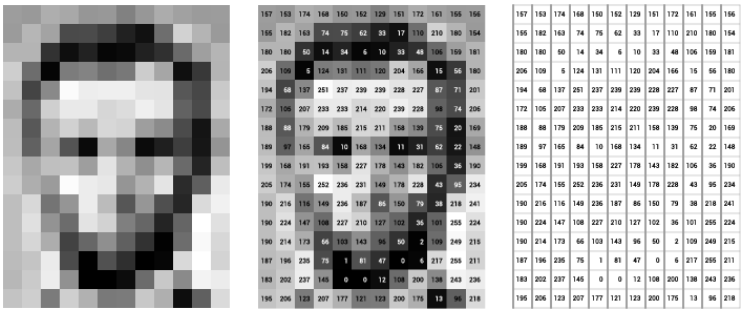
Its recommended to follow these steps in this specific order:

1. Read the Background section to fully understand what kind of data stuctures you're called to implement. Do **not** start coding before you do that!
2. Check to see if you have a tool to view PGM files. Open the provided **`image1.pgm`** with your favorite image viewing application. Most applications support the PGM format. If not, download the popular GIMP which is free and opensource. *This is not a required step to complete this project but it will help you a lot if you can visualize the images that you use as input / output.*
3. Read the "Rules" section to avoid violations that can cost you many points.
4. We recommend you work on the code in the following order:
    1. `Utilities.loadData` – you need to read an image file before you can do anything else
    2. class `TreeNode` – a simple node for the tree; requires minimal work
    3. class `Queue` – it's a generic queue that doesn't depend on quadtrees; it's not hard to implement and you will need it later for the tree traversal anyway
    4. `QuadTreeImage` constructor – it creates the quadtree structure which is the basis for the other tree operations
    5. `QuadTreeImage.countNodes` – it's a low-hanging fruit and will quickly get you into the tree-traversal mood
    6. `QuadTreeImage.brightness` – it's a good practice of recursion before you attempt the `setColor` method
    7. `QuadTreeImage.getColor` – it's a good practice of recursion before you attempt the `setColor` method
    8. `Utilities.exportImage` – it's easy to implement with the use of `getColor`
    9. `QuadTreeImage.compareTo` – another low-hanging fruit
    10. `QuadTreeImage.toString` – although the level-order traversal is straight forward once you implement the queue, this method has a small challenge; you must figure out how to include some extra data in the returned string
    11. The `setColor` is the most challenging method because it can trigger the restructuring of the quadtree. Work on it after you're done with everything else.

Background

A digital image with dimensions *H*x*W* is represented with a **matrix of pixels** that has *H* rows and *W* columns. 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 2a depicts a zoom-in on a 16x12 grayscale image. In Figure 2b you can see the values of each pixel superimposed on the image. And in Figure 2c you can see 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.



a *b* *c*

Figure 2: Example of a grayscale image representation

The reason we want to use quadtrees to store images is because certain image processing operations (e.g. intersection, union) can be done very efficiently with this representation. If you consider the fact that a typical image nowadays contains about 20-50 millions pixels, it's not a surprise that efficiency is very important for image processing. Let's see how an image can be stored efficiently with a quadtree. For illustration purposes only, we will demonstrate the process for a special type of grayscale image, called bitmap. Pixels in a bitmap can only have one of two possible values: black or white. In Figure 3 we have an 8x8 bitmap on the left, and the respective quadtree on the right. The leafs of the tree represent regions of pixels that have the same color. These nodes don't need any children because they can efficiently represent the whole region by using a single pixel value. Inner nodes, on the other hand, represent regions of pixels that do not have a unique color. If a region has more than one colors, it needs to be further partitioned. The smallest region a leaf can represent is 1x1; this means that the smallest region an inner node can represent is 2x2. In Figure 3, leafs are depicted with the color of their region (either black or white), while inner nodes with gray. The root of the tree represents the whole image and can be either a leaf (if all the pixels have the same color) or an inner node. *Note*: the four children/quadrants of an inner node must always be in the same fixed order; in this figure it's a clockwise order ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4$) but, in general, it can be any fixed order you chose.

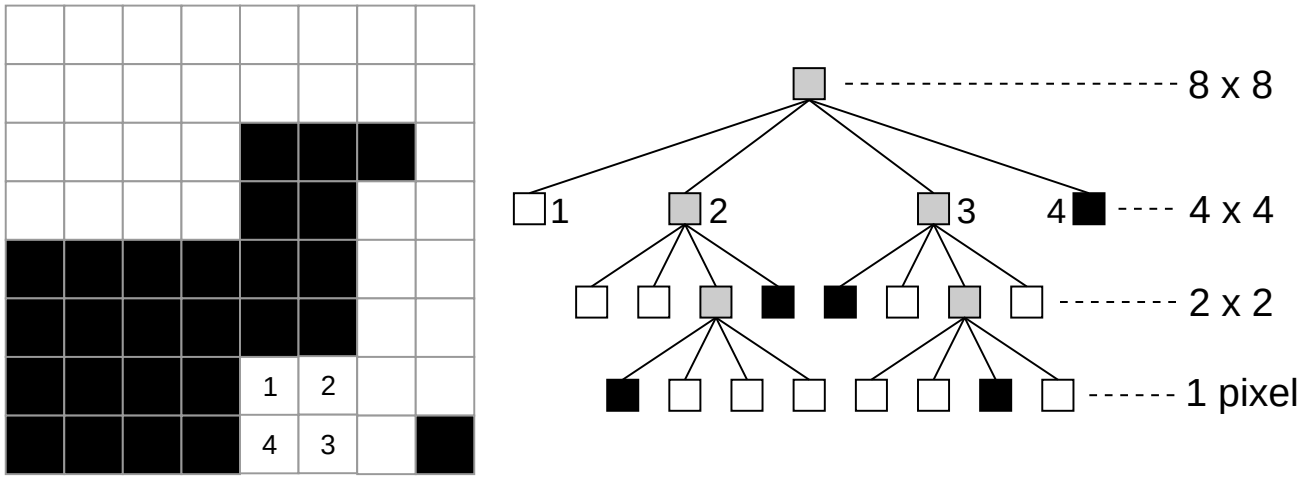


Figure 3: Quadtree for a bitmap (source: wikipedia)

This partitioning of the image works best for images that have large regions with the same color (e.g. in Figure 3 the top-left and the bottom-left quadrants became leafs in the very first partition at depth 1). This storage efficiency is more obvious in a natural image as the one in Figure 4. As you can see, there are many regions of the image that have the same color and therefore we can use just a few nodes of the quadtree to represent them en masse instead of storing all their individual pixels.

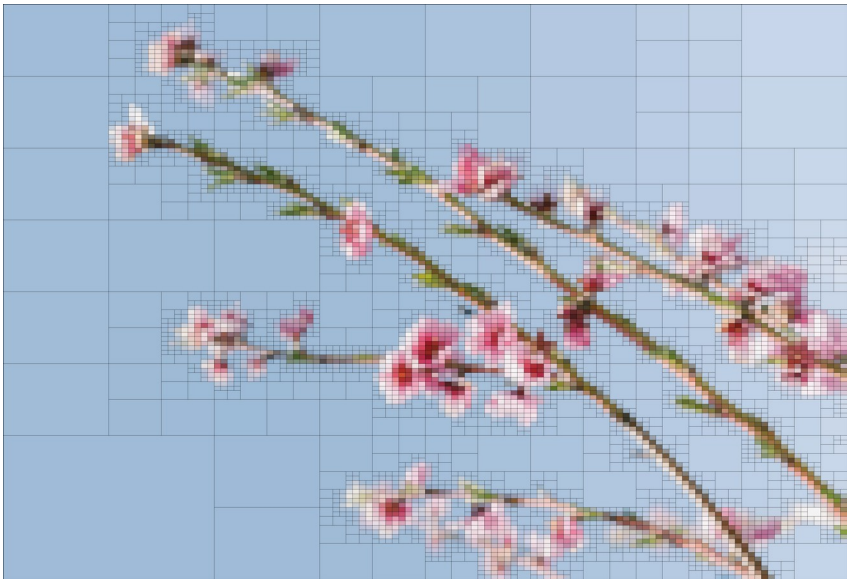


Figure 4: Example of quadtree partition (source: medium.com)

If you're looking for more insight regarding the quad tree structure and how we construct a tree from an image, watch this video: https://youtu.be/wppL6rp-HNU

Working with image files

Image files come in many different formats (JPEG, PNG, BMP, etc.), but to keep things simple for this project we will use the PGM format that supports storage of the pixels in plain text. This makes the reading/writing of image files, as well as the debugging, much easier. Moreover, you can visually inspect the images you use and the ones that your code generates, with an image viewer application (if you don't have one, download GIMP which is free and opensource). In the snippet below, you can see an example of an actual image stored as a PGM file.

```
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
```

- The first line is the string **`P2`** which indicates that the image is grayscale (not color or bitmap) and the storage in the file is in plain text (not binary). This line is necessary because PGM supports other formats too, so do not change this line.
- The second line contains the *width* and the *height* of the image separated by a single space. In this case the width is **`6`** and the height is **`4`**. Note that the width (not the height) comes first.
- The third line defines the maximum value of a pixel (**`255`** in this case). Do not change this line either; all the images we work with in this project are grayscale images with pixel values in the range 0-255.
- The last part of the file is the pixel values. It consists of a series of *`width*height`* pixel values that are separated by a single space. Whether you put these values in a single line or in multiple lines, doesn't make any difference when using a photo viewing/editing application. So, make sure your code behaves the same. *Hint*: the simplest thing to do is have the `Scanner` object read single tokens repeatedly (i.e. don't use the `nextLine` method).

For practice, open your favorite **plain text** editor, copy+paste the above snippet, and save the file as **`image.pgm`** (make sure the encoding is plain text). Then, open it with your favorite photo viewing/editing application (not a text editor) to verify that it looks like the image in Figure 5. You will have to zoom-in a lot because this image is 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.

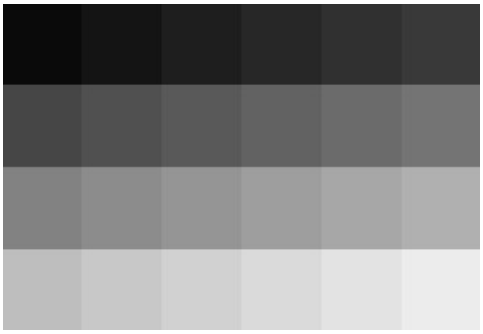


Figure 5: Image rendering of the above code snippet

Submission Instructions

1. Make a backup of all your project files and upload it to OneDrive (this is not your submission, just a backup!)
2. Follow the Gradescope link provided in Blackboard>Projects and upload the .java files only (not the .class files or the images). Do not put them in a folder or zip them
3. Verify your submission. Download the submitted files from Gradescope and check if they compile and run as expected.

If you don't follow all three steps, there will be no excuse if you submitted the wrong files

Rules

You must:

- Have code that compiles with **`javac *.java`** in the terminal without errors or **warnings**.

You may:

- Add additional methods and variables, however these **must be private**.
- Add additional nested, local, or anonymous classes, but any nested classes must be private.
- Use the provided **`Project3.java`** to get started and see if your code works. Keep in mind that **this is not a tester**, it is only provided to give you an idea of how your code should work. Do not include this file in your submission.
- Import the following: `File`, `Scanner`, `PrintWriter`, `FileNotFoundException`, `Iterator`

You may **NOT**:

- Add additional *public* methods, variables, or classes.
- Use any built in Java Collections Framework classes (e.g. **`LinkedList`**, **`ArrayList`**, etc.) or any third-party libraries/packages.
- Add any additional import statements or use the *fully qualified name* to get around adding import statements.
- Store anywhere in your program the **`Pixel[][] array`** that was passed to the **`QuadTreeImage`** constructor. This array is used only for constructing the quadtree. Violation of this rule will result in a 0 for the entire **`QuadTreeImage`** class.
- Make your program part of a package.
- Add `@SuppressWarnings` to avoid fixing warnings. There are a couple of places that are exempted from this rule and they're clearly stated in the description.

Assumptions

- We will assume that images have a **square size** and the **width of an image is always a power of 2** (i.e. 1, 2, 4, 8, 16, 32, etc). This will simplify the creation of the quadtree as you won't have to deal with rectangular regions. It will also make memory usage more efficient since **there is no need to store the coordinates of the image region inside the tree node**.

Tasks

You must implement the following classes and methods

`class Utilities`

It's a helper class that contains two utility methods only:

`loadData` opens a PGM file and loads the image data into a 2D array. You will need this array when you call the constructor of the quadtree.

`exportImage` creates a new PGM file and writes in it an image that is represented by a quadtree

`class TreeNode`

This class represents a single node in the quadtree structure.

The node has four children. To avoid any issues from numbering/ordering these children, we use the names `NW` (northwest), `NE` (northeast), `SW` (southwest), `SE` (southeast) so that it's clear which quadrant each child points to. The provided file `cs310code.xml` that you use to check the style of your code, has been modified to accept these two-letter variable names. Make sure you use this customized version instead of the one we provided for previous projects.

The node has also a `value` which is of generic type. Only the leafs make use of this value because the inner nodes represent regions of pixels that do not have a unique value.

`class Queue`

It's a helper class that is needed for a level-order tree traversal. It implements a **circular FIFO queue** with the use of a **dynamic array** (you must implement it with a dynamic array, you're not allowed to use a linked list implementation). For this class specifically, you *are* allowed to use the annotation `@SuppressWarnings`. The queue has an initial capacity of 10. When it expands, the capacity doubles. It never shrinks.

`class QuadTreeImage`

This class is the most important component of this project. It uses a quadtree data structure to represent a grayscale image. It also provides various methods for processing the image. It implements the `Comparable` and the `Iterable` interfaces.

`QuadTreeImage(array[][])`

The constructor builds the quadtree from an array that holds the pixel values you have read from the PGM file. You **may not store the `array`** or a pointer to the `array` anywhere outside the constructor, i.e. the array must be discarded after the construction of the quadtree.

`countNodes()`

Returns the total number of nodes in the quadtree including the root and the leafs.

`brightness()`

Returns the sum of all the pixel values. You may **not** call the `getColor` method, you must work directly on the quadtree. Be reminded that a leaf of the quadtree may represent more than one pixel.

`compareTo(QuadTreeImage other)`

The `QuadTreeImage` class implements the `Comparable` interface and, therefore, it must provide this method. Comparison between two images is based on their brightness. A brighter image is considered a larger image. The method returns the difference in the brightness between the two images.

`toString()`

Overrides the default behavior. It returns a representation of the quadtree that lists the nodes of the quadtree in a **level order** (breadth first traversal). Within each level we use a **clockwise** order for the children starting from the northwest child, i.e. `NW`→`NE`→`SE`→`SW`. **Only leafs** should be included in the string representation. For each leaf, you report: the top row coordinate, followed by the left column coordinate, followed by the width of the image region it represents, followed by the pixel value. The following is an example of the string returned for Figure 3 assuming it's a proper PGM where the black is 0 and the white is 255:

```
{0 0 4 255},{4 0 4 0},{0 4 2 255},{0 6 2 255},{2 4 2 0},{4 4 2 0},{4 6 2 255},{6 4 2 255},{2 6 1 0},{2 7 1 255},{3 7 1 255},{3 6 1 255},{6 6 1 255},{6 7 1 255},{7 7 1 0},{7 6 1 255}
```

`getColor(int w, int h)`

Returns the value of the pixel at location *w*, *h* where *w* is the column index and *h* is the row index in the image (indexes start from 0). If the coordinates *w* and *h* are invalid, it throws an `IndexOutOfBoundsException`. You may **not** call this method anywhere in your code except in the `Utilities.exportImage`. It will result in 0 points for any method that violates this restriction. The same is true if you try to bypass this restriction by building your own helper method that has the same or a similar functionality with getColor.

`setColor(w, h, newValue)`

Sets a new value for the pixel at location *w*, *h* where *w* and *h* are the column and row indexes in the image. If the new value is different from the previous one, the tree will require a restructuring. See Figure 6 for an example. By setting the pixel at location {6,2} to value 255, the four leafs of the respective subtree become white and the subtree must now be removed (its parent is not anymore an inner node but becomes a white leaf). Keep in mind that such an action can trigger a **recursive restructuring** that goes multiple levels up the tree (imagine for instance what would happen if the black leaf on the right of the one that was just changed to white was a white instead of a black leaf). But the restructuring doesn't always go up, it can also go down by adding more levels to a subtree (for example, if we change the black pixel at location {0,5} to a white one, then node 4 is not anymore a leaf but requires partitioning).

A fundamental concept in this method is that you're not recreating the image array. You must work directly on the quadtree to take advantage of its efficient storage. It will result in 0 points if you violate this concept. Moreover, you may not use the `getColor` or any kind of helper method with similar functionality to indirectly recreate the image array.

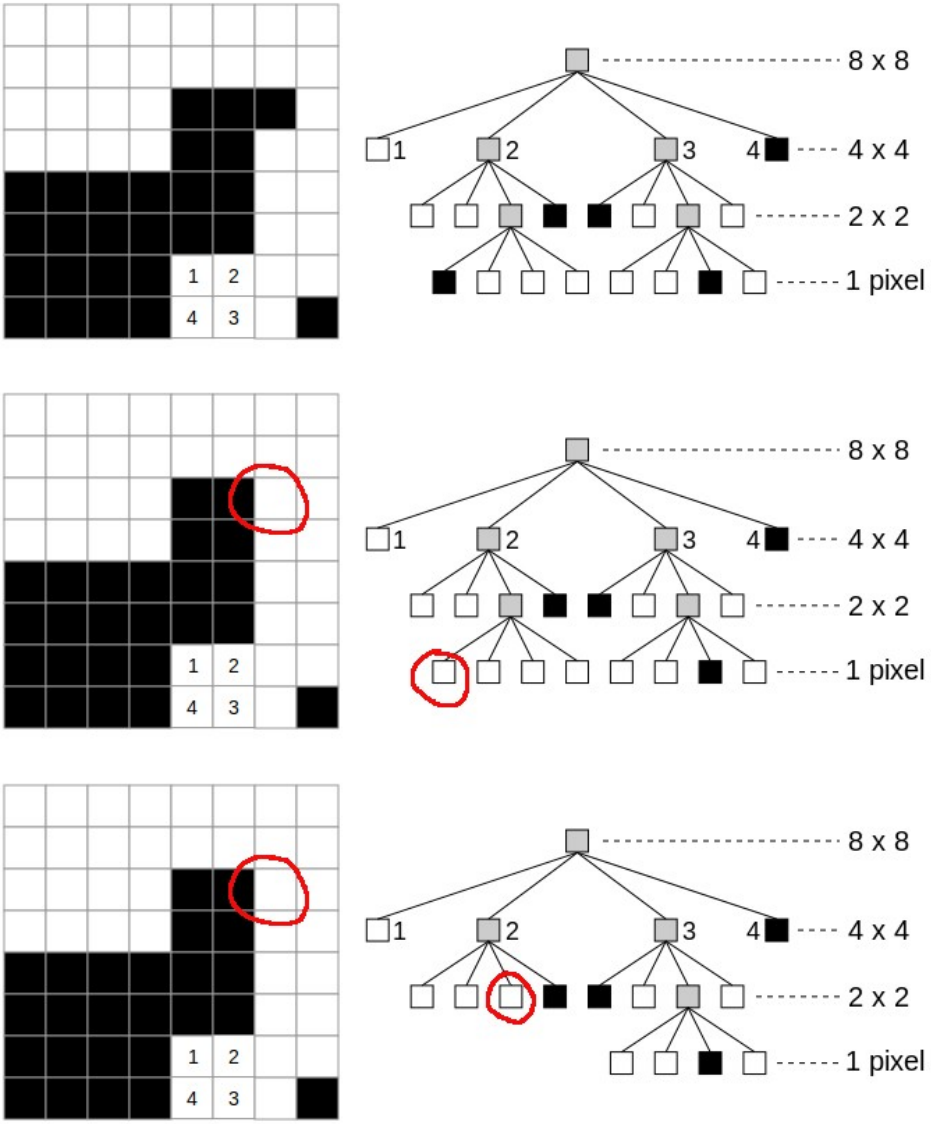


Figure 6: Effect of calling setColor(6, 2, 255)

`class QuadTreeImageIterator`

It creates a quadtree iterator that traverses all the nodes (both inner and leafs) in a **level order** (BFS). Within each level, the order of traversal of the four children should be: `NW → NE → SE → SW`

The following output is an example of a for-each loop running on the image in Figure 3 (assuming it's a proper PGM where the black is 0 and the white is 255).

```
null 255 null null 0 255 255 null 0 0 255 null 255 0 255 255 255 255 255 0 255
```

You don't need to implement the `remove` method, just the methods `hasNext` and `next`

Grading Rubric

Disclaimer: The professors reserve the right to adjust this rubric due to unforeseen circumstances.

How will my assignment be graded?

- Grading will be divided into two portions:
  - Automatic Testing: To assess the correctness of programs.
  - Manual Inspection: For features your programs should exhibit that are not easily tested with code such as Big-O, violations, etc. You **cannot** get points (even style/manual-inspection points) for code that doesn't compile or for submitting just the files given to you.
- Extra credit for early submissions:
  - 1pt extra credit rewarded for every 24 hours your submission made before the due time
  - Up to 5pts extra credit will be rewarded
  - Your latest submission before the due time will be used for grading and extra credit checking. You **cannot** choose which one counts.
- No credit (a 0 for the entire assignment) will be given for any of the following:
  - Non submitted assignments
  - Assignments late by more than 24 hours
  - Non compiling assignments

A breakdown of the point allocations is given below:

5 pts	Utilities.loadData
5 pts	Utilities.exportImage
20 pts	QuadTreeImage constructor
5 pts	QuadTreeImage.countNodes
5 pts	QuadTreeImage.brightness
3 pts	QuadTreeImage.compareTo
5 pts	QuadTreeImage.toString
5 pts	QuadTreeImage.getColor
7 pts	QuadTreeImage.setColor
5 pts	TreeNode
15 pts	Queue
10 pts	QuadTreeImageIterator
5 pts	Big-O requirements
2 pts	Code style
3 pts	JavaDoc documentation
up to +5 pts	Extra credit for early submission