CIS 121—Data Structures and Algorithms with Java—Spring 2018

Homework 2—Thursday, January 18

Due Monday, January 29 by 11:59 PM

7 Required Problems (85 points), and Style and Tests (15 points)

DO NOT modify methods or method headers that were already provided to you.

DO NOT add public or protected methods.

DO introduce helper methods that are **package-private** (i.e. do not have a privacy modifier) as you deem necessary.

Logistics

You will notice that all of the stub files for this homework do not belong to any package. **DO NOT** place any source files into a package, and ensure that no file has a package declaration e.g. `package src;`. When you submit, be sure that all files are located at the root level i.e. do not submit a folder that contains the source files. The easiest way to do this is to drag the files individually onto Gradescope.

Important: Be sure that your own test cases pass and do not stall. Otherwise, your submission will not be processed correctly.

You have unlimited submissions for this homework. Upon submission, our infrastructure will compile your code and run some basic test cases against the interface. These are here to test that your code compiles (i.e. you didn't forget to add a class etc.) when we run real tests. They should not be used to test your own code's correctness since the functionality they test is very very basic.

The compilation tests will be worth 1 point of the final score.

Important: Before asking any questions on Piazza, be sure to check and keep updated with the pinned clarifications post, where important clarifications will be posted.

Files to Submit: There are **6** files to submit in total.

`QuadNodeImpl.java, QuadTreeImpl.java, QuadTreeFactoryImpl.java`
`QuadNodeImplTest.java, QuadTreeImplTest.java, QuadTreeFactoryImplTest.java`

Motivation—Image Compression

Since the dawn of time, mankind has sought to make things smaller...

Erlich Bachman

Today's computers are constantly communicating over the network, transmitting both raw application information and user created data in the form of large files. We are already familiar with the notion that faster programs are better. This also holds true over the network; high-latency networks affect everything from multiplayer video games to standard Internet access to mission- and life-critical applications.

Faster network access is always better. However, we cannot always make an Internet link faster (the networked equivalent of throwing more hardware at the problem). Therefore, we must use intelligent algorithms to reduce our load on the network. Some of these are built directly into the internals of our computers, like TCP congestion control[1] (less is more, sometimes). Others are optionally implemented at the application level, such as compression algorithms.

The goal of a lossless compression algorithm is to represent existing data through a succinct model in a way such that the original can be recovered. For example, common image compression algorithms make use of methods like segmenting the image into areas, inferring colors from adjacent pixels, or building shorter color representations based on color frequency to achieve a smaller file size that can be recovered into the original image.

In this assignment, you will implement a relatively simple lossless image compression algorithm based on quadtrees to decompose the image. You will generalize your knowledge of recursing on binary search trees to quadtrees, adapting known algorithms to perform various operations on the quadtree representation of a compressed image.

Overview

The simplest way to represent an image is to have a structure that maps every pixel in the image to a color. This is commonly known as a 'bitmap image'. However, for large images this quickly becomes very inefficient. Take the example of a document scanned at a high resolution: Most of the page would be white pixels, and clumps of text would be black and grey pixels. Recording each white pixel of the page separately is very wasteful compared to telling the computer some region of an image should be white.

One solution is to extend this uniform region representation recursively. Instead of defining an image as a collection of arbitrary regions, we define an image as a collection of chunks. Each chunk, in turn, can either be a single color region (blank area) or collection of some other chunks. The end result of this is, of course, a tree like structure, where the entire image is represented by the root chunk and any internal nodes represent chunks that were subdivided. In this model, leaf nodes would represent uniform areas of color.

In general, splitting up chunks is not a very easy task since images come in all sorts of dimensions. We will make the following simplifications and clarifications for this assignment:

- We will only consider square images. Moreover, images will always have row and column sizes that are non-negative powers of 2; that way they can always be divided into *four quadrants of equal size* (excluding the case of a 1x1 region).

[1] http://en.wikipedia.org/wiki/TCP_congestion-avoidance_algorithm

- A region will be considered uniform only if every pixel in it has the exact same color. As such, our compression will be *lossless*, meaning you will be able to exactly decompress the original image.
- Since we are recursively splitting our image into four quadrants of equal size, this kind of tree is called a *quadtree*. You should use the supplied enum `QuadNode.QuadName` to refer to each of the four quadrants (upper left, upper right, lower left, lower right).

To recap, we have defined a quadtree structure with leaves and non-leaves. Leaves are defined to be nodes with no children and a color. Non-leaves have four child nodes and no notion of a color. Notice that this definition of a quadtree guarantees a full tree; leaves have zero children, and parents have four children.

Now, how do we represent color? The Java `BufferedImage` class, which we look to for inspiration, stores RGB values as a single combined integer. We use that representation here. You don't have to worry about how to generate that representation, but just know this is the reason why colors are represented as integers in this assignment. Note that 0 is a valid color in this representation.

Part Zero: Read and understand the requirements

Files to submit: None

Understanding the problem is the first step to solving it. Take time to read through this writeup and the Javadocs for the provided interfaces completely before starting. Get familiar with Java syntax and style conventions outlined in the course website. Go over the requirements and expectations of each class in your head and plan things out.

Part One: Nodes and Trees (13 points)

Files to submit: `QuadNodeImpl.java`, `QuadNodeImplTest.java`

Implement the interface defined in `QuadNode.java` with your own concrete implementation called `QuadNodeImpl`. The `QuadNode` interface defines the fields and methods that a single node in the tree should have, namely:

```
public interface QuadNode {
    int getColor();
    QuadNode getQuadrant(QuadName quadrant);
    boolean isLeaf();
    int getDimension();
    int getSize();
    void setColor(int color);
    void setQuadrant(QuadName quadrant, QuadNode value);
}
```

Please refer to the Javadoc comments for these methods for their specifications. Keep in mind that your implementation of `QuadNode` should play nicely with other potential implementations of `QuadNode`. For example, it must accept any generic `QuadNode` implementation for `setQuadrant`.

Consider how you would go about building a compressed tree representation of an image using your own `QuadNode` instances. Think about how the quadtree model would work to compress the image as well as possible.

Part Two: Tree abstraction / Implementing compression

Files to submit: `QuadTreeImpl.java`, `QuadTreeImplTest.java`

As can be seen from the `QuadNode` specifications, `QuadNode` implementations don't necessarily obey the invariants outlined for a quad tree. One could imagine that users of the interface would be able to manipulate the `QuadNode` (by setting quadrants) leaving it in an inconsistent state (e.g. has some children, has both children and color etc.). Instead we want to package a quad tree constructed through `QuadNode`s so that it only exposes methods and operations that would still keep the tree state consistent.

For this section, you only need to implement the constructor for the quad tree. This constructor should be able to take in an array of image data (integers) and, using your implementation of `QuadNode`, build a tree that satisfies the quad tree invariants. You *can* use package-private helper methods here (and we encourage you to do so here and for the rest of the assignment). We're leaving this part of the design up to you, so make sure to spend some time planning out how you want to build things. There are many ways to structure your implementations - some will make your life much easier. An additional hint: Take a look at the structure of the input the tree factory takes in before starting.

Part Three: Tree Factories (6 points)

Files to submit: `QuadTreeFactoryImpl.java`, `QuadTreeFactoryImplTest.java`

Now that you've implemented concrete classes for the quad tree interfaces, we need a way for you to offer your implementations for external use. Now, one certainly *could* just provide the constructors of the implementation classes to the public. However, doing that makes the external code dependent on your internal implementation details and thus hinders future refactoring of the code.

Instead, we can use the factory method pattern in Java. A factory is a class that creates objects through its namesake "factory" methods. Generally, a factory creates instances of a particular interface. Why is this useful? This allows us to receive an instance of an interface without knowing what the concrete class is. In our case, we have an interface, `QuadTreeFactory` for a factory that builds `QuadTree`s. In our concrete implementation, `QuadTreeFactoryImpl`, we might choose to return `MyFancyQuadTree` or `MySeventeenthAttemptAtAQuadTree`. This is how we were able to write the `SimplePaint` program without knowing about your concrete `QuadTree` class.

In this assignment, you are asked to implement the `buildFromIntArray` method of the factory. Traditionally, computer monitors render the image from top to bottom by filling in each line, so in computer graphics, a flipped coordinate system with the y-axis pointing downwards is used to coordinate points in the image. This means that $(0, 0)$ will be referring to the **top left** corner of the image. We will be using a row-major order coordinate addressing scheme, which means that when getting a value, we first index into the correct row, and then index into the column. For example, with the following array

```
int[][] image = {
        {0, 1, 2, 3},
        {4, 5, 6, 7},
        {8, 9, 0, 1},
        {2, 3, 4, 5}
    };
```

we can access the value `7` with `image[1][3]`. You can also have a look at `buildFromImage()` in `SimplePaint` to figure out how this works.

Part Four: Finishing up the tree abstraction (23 points)

Files to submit: `QuadTreeImpl.java`, `QuadTreeImplTest.java`

Now that we've done the lower-level implementation of the compression tree, we can move onto implementing more interesting methods on the compression tree itself. You will notice that `QuadTree.java` has a `getCompressionRatio()` stub. Let us now focus on that. The *compression ratio* is $\frac{N}{P}$, where N is the total number of nodes in the tree and P is the number of pixels in the image.

For example: Given an image with n pixels...

- ...that is purely one color: An optimal tree will have one node, with compression ratio $\frac{1}{n}$. See the first example in Figure 1.
- ...that is half black, half white: An optimal tree would limit itself to five nodes, one for the root and four quadrants as leaves. In this case, the optimal compression ratio is $\frac{5}{n}$. See the second example in Figure 1. Note what this implies about counting nodes: When there is a non-leaf, it, as well as its children, are included in the count.

You should be able to get the node count from `getSize()`, and you should be able to easily calculate the number of pixels in the image from `getDimension()` (both are methods of the `QuadNode` interface).

The compression ratio is a good way for programs to determine when to use compression and when not to. For images without any uniform colored regions, quad trees not only cannot effectively compress them, a quad tree will incur extra overhead which can bring the number of nodes over the number of pixels. Programs in real life scenarios would look at this value and decide whether compressing was worth it.

Part Five: Decompression (13 points)

Files to submit: `QuadTreeImpl.java`, `QuadTreeImplTest.java`

Because this compression algorithm is lossless, it is possible to recover the original array of RGB combined values from the compressed quad tree with no loss of information. The `decompress()` method on the `QuadTree` class is meant to provide this functionality. Implement the method stub.

As a result of implementing this method correctly, we would expect that the original pixel array and the decompression array of the compression tree of the original pixel array should exactly equal each other for all coordinate pairs.

Part Six: Retrieving pixel color and editing the image (29 points)

Files to submit: `QuadTreeImpl.java`, `QuadTreeImplTest.java`

Implement the method `getColor` in the in your `QuadTree`. This method receives integer coordinates `x` and `y` as input (refer to the previously outlined coordinate system). The method returns the color of the pixel at coordinate (x, y) of the image represented by the `QuadTree`. Please implement this method *in-place*: in other words, do not convert the quadtree into an array and then index into the array. Traverse the quadtree instead; it's asymptotically faster and more space-efficient.

Next, write the method `setColor` in the class `QuadTree`. This method has three inputs: a color, and two integer coordinates `x` and `y`. This method is trickier than the previous one in that it performs an in-place modification of the quad tree. You should always keep the quad tree optimal (i.e. no node should have four children of the same color) which may require compressing nodes or expanding them. You should *not* have to decompress the whole quad tree in the process of editing the image. As a hint, you may find some overlap with the logic of building the tree from an array source. These similarities may be great candidates to abstract out as helpers. Remember that you're in charge of designing your implementation.

After implementing this method, you can arbitrarily edit the image without needing to decompress it! That's pretty cool.

Style & Tests (15 points)

The above parts together are worth a total of 85 points. The remaining 15 points are awarded for code style, documentation, and sensible tests. Style is worth 5 of the 15 points, and you will be graded according to the 121 style guide.

You will need to write comprehensive unit test cases for each of the classes and methods you implement. Make sure you consider edge cases and exceptional cases, i.e. the more "interesting" inputs and situations.

Each unit test should test a single simple aspect of a class, i.e. how one specific method behaves. Prefer many small tests over cramming a bunch of asserts into a single test method. Be sure to demonstrate that you considered "bad" inputs such as `null` inputs. Your test cases should cover all the code you have written, including any helper methods you have decided to include.

> Unit tests are **extremely important** for this and future homeworks. These tests guarantee that individual parts of your program function correctly. Writing good tests are to your own benefit. Our automated tests, like any other application that uses the interfaces, will only test compliance to the interface documentation and cannot test your implementation details. As such, they will not give partial credit for any cascading errors (i.e. an error that didn't get caught in your `QuadNode` unit tests may manifest as multiple failures and point deductions when testing the tree). You are responsible for testing each component in your implementation.

Note: You will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier.

Simple paint!

Files to submit: None

You will notice that we have included a `SimplePaint` class. This is a self-contained simple paint application that uses a `QuadTree` to store an image and allow you to make edits to it by drawing with your mouse. The paint program is built on top of the interfaces provided by quad trees.

At this point, you should be able to run the program flawlessly. Play around with opening some test images or images of your own. You can also turn on node borders look at your quad tree structure through this application.

This part is simply meant to be for fun, and there's no need to include this in unit tests (nor is there a very obvious way to do so).

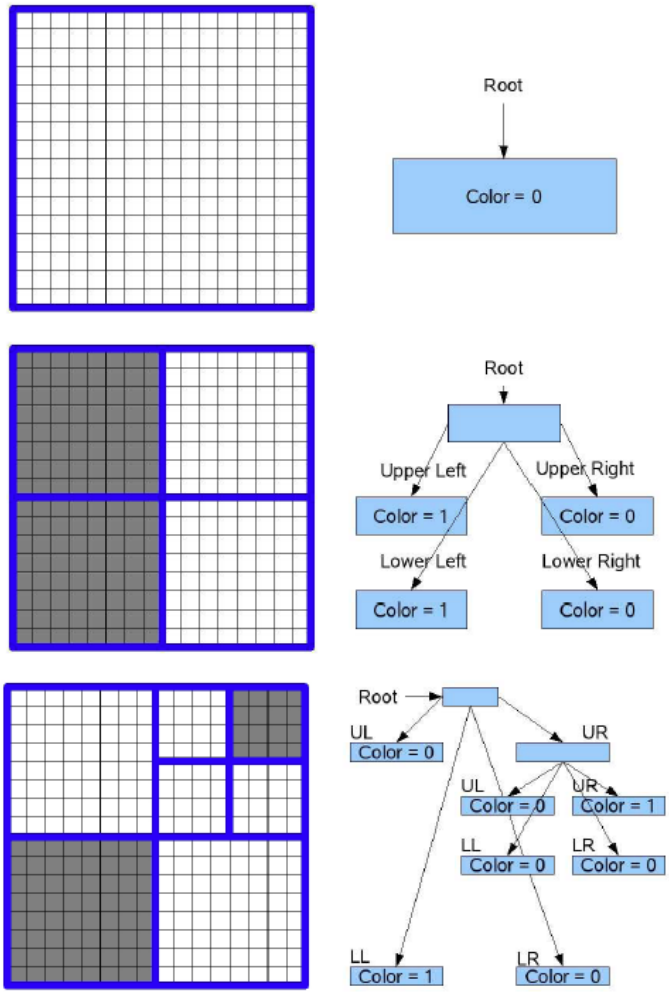

Figure 1: Examples of binary images and their quad trees.