Object Orientation – Spring 2018 **Assignment 7**

1 Quad Trees

A *quad tree* is a *recursive data structure* that can be used to represent particular types of trees. A tree just as a list, consists of nodes that can be *self-referential*. This means that every class representing a node has to contain one or several instance variables that point to an object of the same class (or to a super class). In the case of quad trees, each internal node has exactly 4 self-references pointing to its successor nodes.

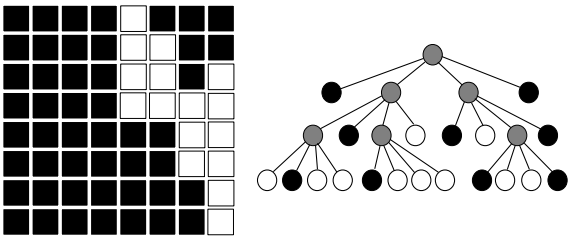
2 Learning Goals

For this assignment, you will be asked to design and implement a quad tree representation in Java. After complementing this exercise, you should be able to:

- Create recursive data structures based on self-referencing classes,
- implement recursion-based conversions from and to such a recursive data type,
- introduce a suitable interface /abstract class as a base for the representation of different kinds of nodes,
- define each concrete node as an extension of this base class.

3 Problem sketch

Quad trees can be used to store two dimensional images efficiently. In our case, we consider an image of $N \times N$ black and white pixels. For convenience, we assume that N is a power of two (such as 4,8,16,...), and the pixels are exclusively black or white without in-between gray scale values. The more compact recursive storing method is based on the following idea: First, the $N \times N$ figure is split up into 4 quadrants of size $\frac{N}{2} \times \frac{N}{2}$. These subimages are then themselves recursively packed into four quad trees. These four subtrees form the four children of the (internal) node - so the structure eventually represents the entire image. Of course, the recursive process stops once you reach a subimage that consists of a single pixel only. We use two kinds of leaf nodes: One for the white and one for the black pixels. The actual compression happens when four subtrees are packed into a new tree. Namely, when all subtree nodes have the same color, the entire subtree is represented as a single color pixel node. This way, you can represent entire quadratic surfaces (i.e., adjacent pixels of the same color) by a single node. The following example illustrates this procedure:



On the left, we see a black-and-white figure of size $N = 8$, on the right the quad-tree representation. As described above, the figure is initially split up into four subfigures, each of size 4. Clockwise, beginning with the upper left subfigure, we assign the ciphers 1 through 4 to them. Each of these subfigures is itself split up into four subfigures -and so forth- until we have reached the individual pixels. After the subtrees are converted, you should check whether you can replace an internal node by a black or white leaf - such as the 4-by-4 squares on the top left and

bottom left in this example, which are black throughout. The same is true for the right half of the figure: On the top right, we find one black and one white 4-by-4 square; on the bottom right, there is a black 2-by-2 square. In the tree, you find the black 4-by-4 squares represented as two black leaf nodes connected directly to the root. Try to comprehend how the rest of the tree comes about!

4 Implementation

We will use the `Bitmap` class to represent the images:

```
public class Bitmap {
    // each bit is stored in a two dimensional array named raster
    private final boolean[][] raster;
    private final int bmWidth, bmHeight;

    /**
     * Creates an empty bitmap of size width * height
     * @param width
     * @param height
     */
    public Bitmap( int width, int height ) {
        raster   = new boolean[width][height];
        bmWidth  = width;
        bmHeight = height;
    }
}
```

This class also contains methods to request and manipulate bits, a method called `fillArea` that fills a given section with a given color and a method to convert the `Bitmap` to a `String`. These methods are not displayed in the example above to save space.

We use the following interface to represent the nodes:

```
public interface QTNode {
    public void fillBitmap( int x, int y, int width, Bitmap bitmap );
    public void writeNode( Writer out );
}
```

For the actual nodes, we need to create three implementations of this interface: `GrayNode` for internal nodes, `BlackLeaf` for black leaves and `WhiteLeaf` for white leaves. An abstract method `fillBitmap` fills a given bitmap according to the tree structure.

Next to these three extensions you should introduce a *wrapper* class called `QTree` to access the trees 'from the outside'. The class keeps the internal representation (of connected nodes) of such a tree hidden.

```
public class QTree {
    private QTNode root;

    public QTree( Reader input ) {
        root = readQTree( input );
    }

    public QTree( Bitmap bitmap ) {
        root = bitmap2QTree( 0, 0,  bitmap.getWidth(), bitmap );
    }

    public void writeQTree( Writer out ) {
        root.writeNode( out );
    }
```

```
    public void fillBitmap ( Bitmap bitmap ) {
        root.fillBitmap(0, 0, bitmap.getWidth(), bitmap);
    }

}
```

As you can see, this class contains two constructors: The first one builds a tree from an input image provided by a `Reader`. The second constructor builds the tree from the bitmap argument as described earlier. The two remaining methods do the opposite: `writeQTree` writes the tree to a `Writer` and `fillBitmap` fills a given bitmap according to the tree structure. `Reader` and `Writer` are predefined Java classes used to read from/write to character streams. Look into the standard Java API for further information.

Writing out and reading in

The method `writeNode` writes the node structure as a sequence of bits. This is done by *pre-order traversal*. Each internal node is rendered as a '1' and each leaf as a '0'. In the case of an internal node, the subsequent characters are representations of the four subtrees. For a leaf, the following character is a '0' for a black leaf and '1' for a white leaf. The above-given tree is represented by the following series of bits:

"10011010001010010001010101100011000101000000"

The reading happens the same way: A '1' is always followed by the content representation of the internal nodes. A '0' is interpreted as a leaf, the color of which is indicated by the following bit.

Conversions

Altogether, we have three different options to represent an image: (1) as a bitmap, (2) as a quad tree and (3) as a series of ones and zeros. You should implement a total of 4 direct conversions between these representations, namely from (1) to (2) and back and from (2) to (3) and back.

5 Pre-assembled Content

All code snippets shown in this exercise are found on Blackboard in the form of class definitions. You may use these for you own implementation.

6 Products

You should hand in all `.java` files. Make sure that your code compiles - otherwise we will not consider it a serious attempt.

7 Handing In

Before Sunday 25 March, 23:59 uur, via Blackboard.