# Solution: Quadtrees

*created by Lisa Wang*

***Pseudocode for convertToQuadtree:***

Quadtree * convertGridToQuadtree(Grid<bool> & image, int lowx, int highx, int lowy, int highy):
    Create a new Quadtree node qt.
    Initialize the coordinates of the square that this tree covers.
    Check whether all pixels in the square have the same color.
        If yes: New node is a leaf node.
            Set its color to the color of the square and set all child node pointers to NULL.
        If no: New node has children, each corresponding to one of the four sub-squares.
            Call this function recursively once for each child.
    Return the pointer to the current node qt.

```
// Recursive Function that converts a grid image to a quadtree representing the same image
Quadtree *convertGridToQuadtree(Grid<bool>& image, int lowx, int highx, int lowy, int highy) {
    Quadtree * qt = new Quadtree;
    qt->lowx = lowx;
    qt->highx = highx - 1;
    qt->lowy = lowy;
    qt->highy = highy - 1;
    if (allPixelsSameColor(image, lowx, highx, lowy, highy)) {
        if (image[lowy][lowx]) qt->isBlack = true;
        else qt->isBlack = false;
        for (int i = 0; i < 4; i++) {
            qt->children[i] = NULL;
        }
    } else {
        int midx = (highx + lowx) / 2;
        int midy = (highy + lowy) / 2;
        qt->children[NW] = convertGridToQuadtree(image, lowx, midx, lowy, midy);
        qt->children[NE] = convertGridToQuadtree(image, midx, highx, lowy, midy);
        qt->children[SE] = convertGridToQuadtree(image, midx, highx, midy, highy);
        qt->children[SW] = convertGridToQuadtree(image, lowx, midx, midy, highy);
    }
    return qt;
}

// Wrapper Function
Quadtree *convertGridToQuadtree(Grid<bool>& image) {
    return convertGridToQuadtree(image, 0, image.numCols(), 0, image.numRows());
}

/* Function: allPixelsSameColor
 * Usage: if(allPixelsSameColor(image, lowx, highx, lowy, highy)) { ... }
 * ----------------------------------------------------------------------
 * Returns true if all pixels in the given portion of the grid have the same color,
 * and false otherwise. Note that the upper bounds are exclusive.
 */
bool allPixelsSameColor(Grid<bool>& image, int lowx, int highx, int lowy, int highy) {
    bool color = image[lowy][lowx];
    for (int i = lowy; i < highy; i++) {
        for (int j = lowx; j < highx; j++) {
            if (image[i][j] != color) return false;
        }
    }
    return true;
```

***Pseudocode for compressImage:***

Algorithmically, compressing an image simply means removing nodes from the bottom of the quadtree until the desired tree height is reached. Parent nodes whose children are removed are set to be the average color of its children.

int compressImage(Quadtree * qt, int targetWidth):

compute width of the square corresponding to qt.
compute compression factor by dividing the squareWidth by the targetWidth.
E.g. if squareWidth is 16 and targetWidth is 4, then the compression factor is 4.

Handle special cases for robustness:

If compressionFactor is 1, we don't have to do anything, so return 0 to indicate success.
If compressionFactor is < 1, we cannot compress, return -1 to indicate an error.

Compute the new coordinates of the square that qt corresponds to, since the coordinates change with compression. Off-by-one error warning. Drawing pictures helps!

Base Case 1: If current node is a leaf node, we do not have to go any further and can return 0.
To see whether a node is a leaf node, we check whether its child pointers are NULL.

Base Case 2: The width of the square corresponding to qt has width 1 after compression, so it has to become a leaf node. Set the color of qt to the average of the its children's colors and delete all children recursively. Set qt's child pointers to NULL and return 0.

Recursive cases:
Call compressImage on each child with targetWidth/2.

```
int compressImage(Quadtree * qt, int targetWidth) {
    if(qt == NULL) return 0;
    int curWidth = qt->highx - qt->lowx + 1;
    int compressionFactor = curWidth/targetWidth;
    // We cannot compress the image if the targetWidth is larger than the current width.
    if (targetWidth > curWidth) return -1;
    // if the width of the current square is equal to the target width, we are done.
    if (targetWidth == curWidth) return 0;

    qt->lowx = qt->lowx/compressionFactor;
    qt->highx = ((qt->highx + 1)/compressionFactor) - 1;
    qt->lowy = qt->lowy/compressionFactor;
    qt->highy = ((qt->highy + 1)/compressionFactor) - 1;

    if (qt->children[0] == NULL) return 0;
    // if the current square is a single pixel after compression
    if (targetWidth == 1) {
        int numChildrenBlack = 0;
        for (int i = 0; i < 4; i++) {
            if (deleteQuadtree(qt->children[i])) numChildrenBlack++;
            qt->children[i] = NULL;
        }
        if (numChildrenBlack >= 2) qt->isBlack = true;
        else qt->isBlack = false;
        return 0;
    }
    // recursive cases: compress child trees
    for (int i = 0; i < 4; i++) {
        compressImage(qt->children[i], targetWidth/2);
    }
    return 0;
}
```

```
/*
 * This function deletes the tree and all its children. It returns the
 * average color of the entire tree.
 */
bool deleteQuadtree(Quadtree * qt) {
    bool deletedTreeIsBlack;
    if(qt == NULL) return false;
    if(qt->children[0] == NULL) { // qt is leaf node
        deletedTreeIsBlack = qt->isBlack;
    } else {
        int numChildrenBlack = 0;
        for(int i = 0; i < 4; i++) {
            if(deleteQuadtree(qt->children[i])) numChildrenBlack++;
        }
        if(numChildrenBlack >= 2) deletedTreeIsBlack = true;
        else deletedTreeIsBlack = false;
    }
    delete qt;
    return deletedTreeIsBlack;
}
```