



Assume that the root node is at level 0. At each level  $i$  of the tree (with the exception of the leaf level), each node has eight children, numbered 0 through 7. Think of these numbers as binary numbers: 000, 001, 010, 011, 100, 101, 110, and 111. (In the implementation, it isn't actually necessary to create *all* these nodes at each level. That would make a very big tree –  $8^8$  nodes, which is more than 16 million. Just think of these nodes as "available." They are created as needed, as you'll see below.)

Now consider a pixel from the image file – for example, a pixel that has the color value, in binary, of 001001111110100001111111. Assuming that the original file is in RGB mode, then this pixel can be broken into its R, G, and B components as follows:

```
R: 00100111
G: 11101000
B: 01111111
```

Each pixel in the image file is processed through the tree, down to the leaf nodes (level 8), as follows: Beginning with the least significant (leftmost) bit, take the bits corresponding to the R, G, and B components, in that order, and put them together to form a binary number. In our example, taking the first bit from R, G, and B, we get the number 010. This is the decimal value 2. Thus, we mark node 2 (node 010 in binary) as "visited" and we move to that node, ready to determine which of its children to mark visited as we move down the tree.

In general for an arbitrary pixel, we have bit sequences  $R = r_1r_2r_3\dots r_8$ ,  $G = g_1g_2g_3\dots g_8$ , and  $B = b_1b_2b_3\dots b_8$ , where  $r_i, g_i, b_i \in [0, 1]$  for  $1 \leq i \leq 8$ . As we insert a pixel into the octree, at each level  $i$  of the tree if the bit sequence  $r_i, g_i, b_i$  has value  $j$ , we mark the  $j^{\text{th}}$  node as visited. Marking a node as visited entails incrementing a variable, so that in the end we know how many pixels have passed through each visited node. At each node, we also keep a running total of all the R, G, and B component values that have passed through the node. This makes it possible to compute an average of the R, G, and B components that have passed through a node. This will be useful later in the algorithm.

All the pixels in the image are processed through the tree in the manner just described. In the end, the leaf nodes of the tree represent all the colors that exist in the image file. You may note at this point that unless we put constraints on this process, we could end up with  $8^8$  leaf nodes. But the idea is to reduce the number of colors. Thus, we need to add another feature to the procedure described above: Assume that the only nodes that are actually created are those that are visited. (The other positions are available but never actually created.) If at any point it becomes necessary to create a 257<sup>th</sup> leaf node, you don't let this happen. Instead, you choose a node at the lowest level possible that has at least two children, and you "collapse" the children nodes into their parent – the implication being that the children

nodes will be assigned the "average" of the colors that have passed through their parent. Different strategies can be employed for choosing the node to collapse (in cases where there exists more than one level 7 node with two or more children).

After the octree is generated, the colors at the leaf nodes are placed in a color table, to be stored with the indexed color file. The position in the color table is stored with each leaf node. After that, each pixel is walked through the tree again, and the leaf node that it reaches then points the pixel to its color replacement in the color table. The RGB pixel value is replaced by this index.

## **Instructions:**

### **The Assignment**

Using the programming language of your choice, implement the octree algorithm.

Your program should take as input a raw RGB image. Prompt the user for the name of the input file, and the height and width of the image.

We are mainly interested in how the octree algorithm picks its 256 colors. You could do just a simulation of the octree algorithm, producing as output a 256-element table with R,G,B values and a 2D array of values from this table that represents the indexed image. You could then just print out your data to see if it seems reasonable. Alternatively, you can create a new raw image file that has three-bytes per image, using just the 256 colors that you created in your color table. You haven't really reduced the file size, but you'll be able to see how the image looks with a reduced palette of colors.

If you want to do a full implementation of the program, do a little web research in order to determine the format for a standard indexed file type – e.g., BMP. You need to know what is expected in the header, where to put the color table, and where to put the image data. Then write your data out to the file in the proper format and name your file with the appropriate suffix (e.g., *.bmp*), and you should be able to open your image with any standard image viewer.

### **Before Writing the Program**

To run your program, you will need some raw RGB image files as input, so create these first or ask your instructor if he or she plans to create them for you. The image files should have three bytes per pixel. The input image files can be created by saving an RGB image (BMP, JPG, etc.) in raw format in a standard image processing program.

### **Tips for C/C++ Programmers**



You can use *unsigned chars* for your pixel data. Eight-bit *unsigned chars* are just the right size to store red, green, and blue pixel values. Instead of using the `>>` file-input operator to read your pixel data, use the *read* function associated with random access files. The `>>` operator skips non-printable characters, which will cause you problems if one of your pixel values happens to be the ascii code for a non-printable character. You can use the *read* function to read exactly one byte at a time from the original image file.



You can take advantage of the fact that *chars* can be treated like integers in C/C++. For *rTotal*, *gTotal*, and *bTotal*, you can use *unsigned char* variables, and add to them as you would for integers.

### **Ideas for Further Experimentation and Analysis**

- Explore different options for the node reduction technique. The method described above employs a minimum-damage concept in which fewer color nodes are combined as one. However, this means that the tree must be reduced more often. Try replacing the node with the most children and see how the colors change in the final image.
- Add an implementation of dithering to this algorithm.