Compressed files are of the form "compressedX_Y.out" where X is the index of the example file, and Y is the compression factor applied.

Example: "compressed1_3000.out" is the resulting file from applying the compression algorithm with a factor of 3000 to "example1.ppm".

Decompressed files have names like "decompressedX_Y.ppm" where X is the index of the compressed file, and Y is the compression factor applied.

Example: "decompressed1_3000.ppm" is the decompression of the file "compressed1_3000.out".

##

//.....TASK1.....

Title: Implementation of the compression tree.

I defined two structures: Grid and Tree. I created a matrix using the Grid structure to store the image pixels, and allocated it dynamically. To build the quaternary compression tree, I created the insert function, which recursively inserts nodes into the tree. For this, the color score calculated using a formula needs to be checked and taking into account the toleration, the tree will have more leaves or not. The score will be calculated for each part of the matrix. If the node is not a leaf (a node can be a leaf only if the score is lower than maximum score accepted), the matrix will be divided into four parts, and for each smaller matrix, a new score will be calculated to determine the node type. I iterate through the large matrix without modifying it. I use the dimension for this purpose, and I have an X and Y that will traverse within the limits of this dimension, which halves for each quarter of the previous matrix part.

To read the information from the given ppm file, I used a trick because some images had a newline before the binary data, and others did not. I used fseek to skip the header and read the information in binary.

For initializing the tree, I considered the following default settings: the type is 0, the pixels are 0, and the dimension is always received as an argument. Each node has 4 children, which are initialized as NULL. I created the countLevel function, which returns the number of levels in the tree. It is a recursive function with the base case of the root being different from NULL, returning 1. The maximum between this value and the value returned for the children is taken. When the last pointer is null, the function returns 0.

I created the countLeaves function, which recursively calculates the number of leaves, and the maxLeafSize function, which calculates the maximum size in a leaf node. Since the maximum was required multiple times, I created a macro to assist with the calculations.

//.....TASK2.....

Title: Image compression

I implemented the compression tree. For this task, I created the writeOnLevel function and implemented a queue. The working principle is as follows: the root node of the tree is inserted into the queue, whose size corresponds to the number of nodes at that level. Initially, the size is 1. I created a gSize function to quickly extract the current size of the queue. I check the type of the queue head, and based on the type, I write the necessary information to the file. This node is then removed from the queue; if it has children, they will be added to the queue, thus obtaining the size of the next level. I also created the other functions specific to a queue: createQueue, enqueue, dequeue, destroyQueue.

//.....TASK3.....

Title: Construction of the decompression compression tree

For decompression, we need to reconstruct the tree read level by level from the file. I used a queue and followed a similar principle as in the previous exercise. I created a read function that reads the data from the file and creates the tree. I created an empty queue to track the levels of the tree. I read the image dimension from the file and create the root node of the tree. The root node is then added to the queue. While the queue is not empty, we iterate through all the elements of the current level. For each element, we extract the first node from the queue and read the node type from the file. If the node type is a leaf, i.e., the pixel value from the file, read the pixel values and store them in the current node. If the node type is internal, i.e., a node that contains four children, create four child nodes and assign them in the required order within the square tree. The children are added to the queue. The root is returned, and the queue is destroyed. After obtaining the tree, I built the image matrix. The function allocates memory for the 2D matrix and initializes it with default values. Then, the function traverses the tree and fills the corresponding pixels in the grid matrix. Finally, the function returns the matrix. The getGridAdd() function is an auxiliary function called within this function, which fills the grid matrix with the appropriate values from the tree. In addition, if the current node is a leaf, the function sets the corresponding pixels in the grid matrix to the color value. If the current node is not a leaf, the function recursively traverses the four sub-matrices and fills the grid matrix with the corresponding values from the tree. Finally, the function returns the 2D matrix. The last function, which is also called in the main function, is toPpm. This function takes a file (fp), a 2D matrix (grid), and the size of the matrix (size) as parameters. The function writes the data from the grid matrix to a PPM format file. The function starts by writing the PPM file header, which includes the matrix size and the maximum color value. Then, the function traverses the matrix and writes the color values of each pixel to the PPM file. Finally, the function closes the file. This function is useful for writing the data from the grid matrix to a PPM format file, which can be opened and visualized to see the corresponding image of the tree.