**READ ME**

Regarding this problem, we put explication also in the code for us in order to try not to get lost in trying to solve it and we kept them for a better understanding.

For this exercise we created a class called “QuadTree”. The concept is similar to the binary trees. The only difference is that if a node has children, it must have exactly 4.

The class contains the following variables:

* T pinfo; // it receives the values ‘p’, ‘b’ or ‘w’
* QuadTree<T> \*\*sons; // a vector of pointers to the QuadTrees’
* QuadTree<T> \*parent, \*root; // they represent the pointers to the current parent, respectively to the root of the parent
* Int nr\_sons; // the number of the sons that are created
* Int incomplete\_sons; // it is like an ok; it tells me if the vector of the sons of a certain QuadTree is full or not
* Int level; // it represents the level of a node
* Int number; // we used this variable for printing purposes
* Int pixels; // it store the number of pixels here, but we change the value only for the root node in order to keep everything organized

For the constructor, we allocated memory for the vector of pointers and anything else is initialized with NULL or 0.

The deconstructor calls a function ‘delete\_quaf()’ that deletes everything. It is a recursive function that deletes from the “younger” nodes to the root.

The function ‘print\_quad()’ works almost exactly as the delete function, the only difference being the fact that we print the levels from the first level to the last level.

The function ‘insert\_element(T info)’ is a recursive function.

Firstly, we check if we have a root node (we apply the instructions only for the first node; we make itself the root, the parent, we attribute it the info value and we allow it to have children by initializing the variable incomplete\_sons with 1).

Then, we check if we add a son, the first son of a parent. If so, we create a node of type QuadTree, we attribute it the info value, and we set the parent and the root. If the new node is a parent, we give the okay to it to have 4 children. Then, we set the level of the node, which number is the son and then we increase the number of kids of the current node.

Then, we check if the node must be added as a son or not. If the number of created sons is less than 4, we check if the previous node (son) has available space for a new son, and if so we call again the function. If not, we continue. We create a node of type QuadTree, we attribute it the info value, and we set the parent and the root, we set the level of the node, which number is the son and then we increase the number of kids of the current node. Else if the node is equal to 4, there are no empty spots for a son of the current node. We check if the last node (son) has available space for a new son, and if so we call again the function. If not, we set the parent nodes’ incomplete\_sons = 0 (the okay of having another son from the respective parent), and then we call the function again.

The function get pixels sets the number of pixels of the image. If the image is represented only by a root node, a parent with no sons, if it is a ‘b’ node, the number of pixels is 1024. If it is a ‘w’ node, the number of pixels is 0. If it is not the case, it calls the function ‘calculate\_pixels’.

The function ‘calculate pixels’ is similar to the other two functions (delete and print). It has the structure almost the same as delete, but when you get to a node with no sons created, you add to roots’ pixels the number of pixels of that the node occupies. Another difference is that we don’t need the last block of instructions if because we don’t actually operate in this situation with parent nodes when it comes to computing the sum of pixels.

Outside the class, we made another function ‘create\_quad’ which creates the QuadTrees. For each letter that we read from the vector input (which stores the array of letters read from the text file “date.txt”) we call the function ‘insert\_elem(input[i])’.

Regarding the function QuadTree<char>\* combine(QuadTree<char> \*first\_tree, QuadTree<char> \*second\_tree), we tried to implement it, but we could not succeed. The plan for it was to compare node by node from the two QuadTrees, transform the first QuadTree and respect the following rules in doing so:

* If we have a black node vs a white node or a parent, we keep the black node in the QuadTree that we build
* If we have a white node vs a parent, we must keep the parent with its children
* If we have white vs white we keep one of them in the QuadTree that we build
* If we have parent vs parent we must go down to their children in order to compare, and ultimately we will resume to the three cases presented above

In the main block, we read two strings of letters, input1 and imput2, from a file and declared two QuadTree trees which are created using the function create\_quad.

Then we printed them one by one in order to show the way the trees look, and after that we printed the number of pixels that they have.

Next, we combined the two pictures, resulting the one picture, printed in the same manner as before.