1. (a) We use two AVL trees with keys sorted in different ways. Each node corresponds to one position (x,y), and each nodes stores the set of colours (e.g., $\{R,G\}$) that are assigned to this position.

   (b) The keys for both trees are the position pairs (x,y). One AVL tree is row-first ordered ($\leqslant_R$), and the other is column-first ordered ($\leqslant_C$). In the row-first ordered tree, two keys are compared as follows:

   $$(x,y) \leqslant_R (x',y') \quad \text{iff} \quad x < x' \text{ or } ((x = x') \text{ and } (y \leqslant y'))$$

   This way of ordering makes sure that when the pixels are sorted, they are ordered in such a way that all rows are sorted from low to high, and within each row the pixels are sorted by their column number from low to high.

   Symmetrically, in the column-first ordered tree, two keys are compared as follows:

   $$(x,y) \leqslant_C (x',y') \quad \text{iff} \quad y < y' \text{ or } ((y = y') \text{ and } (x \leqslant x'))$$

   (c) Below is the description of how each operation works.

   - READCOLOUR(S,x,y): Perform a AVLSEARCH for key (x,y) in either of the AVL trees and return the colour set stored in the found node. If the node is not found, return an empty set. AVLSEARCH takes $\mathcal{O}(\log n)$ time.
   - WRITECOLOUR(S,x,y,c): Perform a AVLSEARCH for (x,y) first. If the node is found, then add colour c to the colour set stored in the found node (if c is not in the set yet). If the node is not found, then call AVLINSERT to insert a new node with key (x,y) and with $\{c\}$ stored as its colour set. This needs to be done to **both** AVL trees. This operation takes $\mathcal{O}(\log n)$ time because because both AVLSEARCH and AVLINSERT take $\mathcal{O}(\log n)$ time.
   - NEXTINROW(x,y): First, in the row-first ordered tree, use AVLSEARCH to locate the node p with key (x,y) (note that we assumed that the key must exist in the tree). Then call TREESUCCESSOR to obtained the successor of p with key (x',y'). If the successor does not exist, return $(0,0)$; if $x' > x$, return $(0,0)$; otherwise return (x',y'). Because of how row-first order ($\leqslant_R$) is defined in Part (b), this returned successor is exactly the next pixel after (x,y) in the same row. The running time is $\mathcal{O}(\log n)$ because both AVLSEARCH and TREESUCCESSOR take $\mathcal{O}(\log n)$ time in an AVL tree.
   - NEXTINCOLUMN(x,y): Exactly the same as NEXTINROW except that we do it in the column-first ordered AVL tree.
   - ROWEMPTY(x): First, insert a node with key $(x,0)$ to the row-first ordered AVL tree, then call NEXTINROW$(x,0)$, and return **True** if and only if NEXTINROW$(x,0)$ returns $(0,0)$, i.e., Row x is empty if and only if there is no next pixel in Row x after $(x,0)$. Then don't forget to delete node $(x,0)$ from the row-first ordered AVL tree. This operation runs in $\mathcal{O}(\log n)$ time because INSERT, DELETE and NEXTINROW are all in $\mathcal{O}(\log n)$. We can also avoid the INSERT and DELETE by carefully doing a TREESEARCH for $(x,0)$, and choose the appropriate node to run NEXTINROW on.
   - COLUMNEMPTY(y): Exactly the same idea as ROWEMPTY except that we do it in the column-first ordered AVL tree. Insert $(0,y)$, check NEXTINCOLUMN$(0,y) = (0,0)$, then delete $(0,y)$.

2. (a) At each node x in the AVL tree we store the sum of the value of the keys in the subtree rooted at x, which we will denote sum and the number of nodes in the subtree rooted at x, which we denote n. Computing $Average(x)$ can then be done in $\theta(1)$ time, by dividing the sum of the keys in the subtree rooted at x by the number of nodes in the subtree rooted at x.

(b) First, we modify the rotations which used to balance the AVL tree in order to preserve the *sum* and n values for each node. Observe that under any of the rotations, the ancestors of the parent node being rotated will all have the same *sum* and n values, because the nodes in the subtrees of this ancestor remain the same, they are just rearranged. Therefore, whenever a child is removed form a node in a rotation, subtract its *sum* and n values from its parent. Whenever a child (or subtree) is attached to a node in a rotation, add the *sum* and n values of that child to the parent.

Insert(x): Use TreeInsert(x), and at every visited node along the path to the location where you insert the node x, increment that node's n and add the key of x to that node's *sum*. Finally balance the tree as usual, using the modified rotations mentioned above to ensure the AVL property holds.

(c) Delete(x):

- Case 1: Use TreeDelete(x). If x had no, or only a single child, subtract one from the n and the key of x from the *sum* of each ancestor node of the deleted node, by following the path from the location x deleted from was up back up to the root in $O(\log n)$ time.
- Case 2: If the node had two children, then:
  Case a: If the right child of x is the successor: replace x with its right child, and append the left child (and its subtree) as the successor's left child (just as we do in the regular TreeDelete(x)). Add the values of n and *sum* of the left child of x to the successor's n and *sum*. Finally, subtract 1 from *num* and the key value of x from all ancestors of x in $O(logn)$ time.
  Case b: If the right child of x is not the successor y: promote y's right child to y's position. Traverse from y's right child's ancestor's until a node z is found which is not the left child of a node, subtracting 1 from the n and y's key from the *sum* of the nodes along this path. Place z (and its subtrees) as y's right child, and replace x with y. Let y's n be the ns of it's new left and right trees plus 1, and it's *sum* be the *sum*s of it's left and right trees plus one. Finally, subtract 1 from the n and the key of x from the *sum* of all of x's ancestors in $O(logn)$ time.

TreeDelete is the same as it was in regular BSTs, except that now it modifies the *sum* and n values. Finally balance the tree as usual, using the modified rotations mentioned above to ensure the AVL property holds.