---

1. (a) 2 AVL trees are needed and every node correspond to a "coloured pixels" represented by triples (x, y, c). x coordinate, y coordinate and colour $c \in (R, B, G)$ are stored in every node.

   (b) We sort the AVL tree based on two keys x, y.

   First tree:
   For every node $A(x, y, c)$ in the tree, every node $B(x_b, y_b, c_b)$ in the left subtree has either $x_b < x$ or $(x_b = x) \wedge (y_b < y)$, and every node $C(x_b, y_b, c_b)$ in the right subtree has either $x_b > x$ or $(x_b = x) \wedge (y_b > y)$.
   Second tree:
   For every node $A(x, y, c)$ in the tree, every node $B(x_b, y_b, c_b)$ in the left subtree has either $y_b < y$ or $(y_b = y) \wedge (x_b < x)$, and every node $C(x_b, y_b, c_b)$ in the right subtree has either $y_b > y$ or $(y_b = y) \wedge (x_b > x)$.

   (c) Clearly, both trees order the set of pixels so that no two pixels have same rank.

   - ReadColour(S, x, y):
     First Tree: First search based on x coordinate, for a root $A(x', y', c')$ with $x' > x$, then we go to left subtree, with $x' < x$ we go to right subtree, then based on y coordinate for root with $(x' = x) \wedge (y' < y)$ we go to left subtree and with $(x' = x) \wedge (y' > y)$ we go to right subtree. Else, with $(x' = x) \wedge (y' = y)$ we just read the colour. Clearly, it satisfied BST property which mean we could return colours in worst case $\mathcal{O}(log(n))$.

     Second Tree: First search based on y coordinate, for a root $A(x', y', c')$ with $y' > y$, then we go to left subtree, with $y' < y$ we go to right subtree, then based on x coordinate for root with $(y' = y) \wedge (x' < x)$ we go to left subtree and with $(y' = y) \wedge (x' > x)$ we go to right subtree. Else, with $(x' = x) \wedge (y' = y)$ we just read the colour. Clearly, it satisfied BST property which mean we could return colours in worst case $\mathcal{O}(log(n))$.
   - WriteColour(S, x, y, c):
     First Tree: First based on x coordinate, if a root $A(x', y', c')$ with $x' > x$, then we go to left subtree or with $x' < x$ we go to right subtree. Then based on y coordinate for root with $(x' = x) \wedge (y' < y)$ we go to left subtree and with $(x' = x) \wedge (y' > y)$ we go to right subtree. Once we reach a empty spot we add this triple (x, y, c), or if we reach a node with $(x' = x) \wedge (y' = y)$, we replace the node with this triple (x, y, c). Clearly, it satisfied BST property which mean we could write colour in worst case $\mathcal{O}(log(n))$.

     Second Tree: First based on y coordinate, if a root $A(x', y', c')$ with $y' > y$, then we go to left subtree or with $y' < y$ we go to right subtree. Then based on x coordinate for root with $(y' = y) \wedge (x' < x)$ we go to left subtree and with $(y' = y) \wedge (x' > x)$ we go to right subtree. Once we reach a empty spot we add this new triple (x, y, c), or if we reach a node with $(x' = x) \wedge (y' = y)$, we replace the node with this new triple (x, y, c). Clearly, it satisfied BST property which mean we could write colour in worst case $\mathcal{O}(log(n))$.
   - NextInRow(S, x, y): Using the first tree, first we using ReadColour(S, x, y) for $First\ Tree$ find the pixel A, then if the right child of A is non-empty, get the minimal in the right child, else get the maximal in its parent's left tree. Clearly, it just like the find successor method for BST, since now it is a AVL tree the worst case $\mathcal{O}(log(n))$.
   - NextInColumn(S, x, y): Similarly, using the second tree, first we ReadColour(S, x, y) for for $SeondTree$ find the pixel A, then if the right child of A is non-empty, get the minimal

in the right child, else get the maximal in its parent's left tree. Clearly, it just like the find successor method for BST, since now it is a AVL tree the worst case $\mathcal{O}(log(n))$.

- RowEmpty(S,x): Using the first tree, for a root $A(x',y',c')$ with $x' > x$, we go to left subtree, with $x' < x$ we go to right subtree, with $(x' = x)$ we return $Nonempty$, or return $empty$ if we reach the end of the tree. Cleary, it satisfied BST property which means we could return colours in worst case $\mathcal{O}(log(n))$.
- ColumnEmpty(S,y): Using the second tree, for a root $A(x',y',c')$ with $y' > y$, we go to left subtree, with $y' < y$ we go to right subtree, with $(y' = y)$ we return $Nonempty$, or return $empty$ if we reach the end of the tree. Since, it satisfied BST property and it is a AVL tree which means we could return colours in worst case $\mathcal{O}(log(n))$.

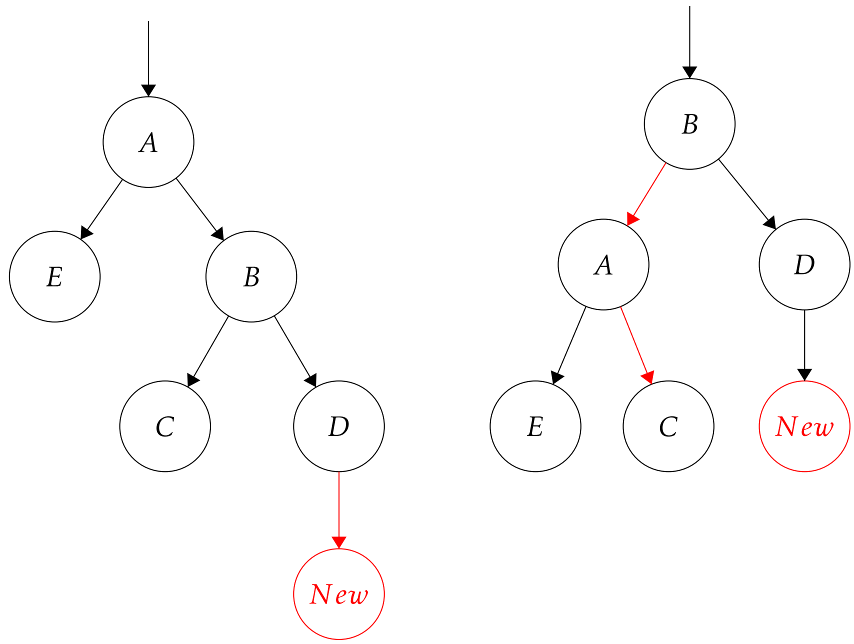
2. (a) For every node A we store 2 extra information, one is the total number of nodes in subtrees, the other is the sum of the key-value of subtrees.
Every node A is represented by a triple $(Key, NumNodes_{subtrees}, Sum_{subtrees})$.

(b) For insertion, suppose we insert a node x, and $x = (x.key, 1, x.key)$

```
Insert(root, x):
  if root = None:
    root ← x
  elif x.key < root.key:
    root.numNodes += 1
    root.sum += x.key
    root.left ← TreeInsert(root.left, x)
  elif x.key > root.key:
    root.numNodes += 1
    root.sum += x.key
    root.right ← TreeInsert(root.right, x)
  else: # x.key = root.key:
    replace root with x # update x.left, x.right, x.sum x.numNodes
  return root
```

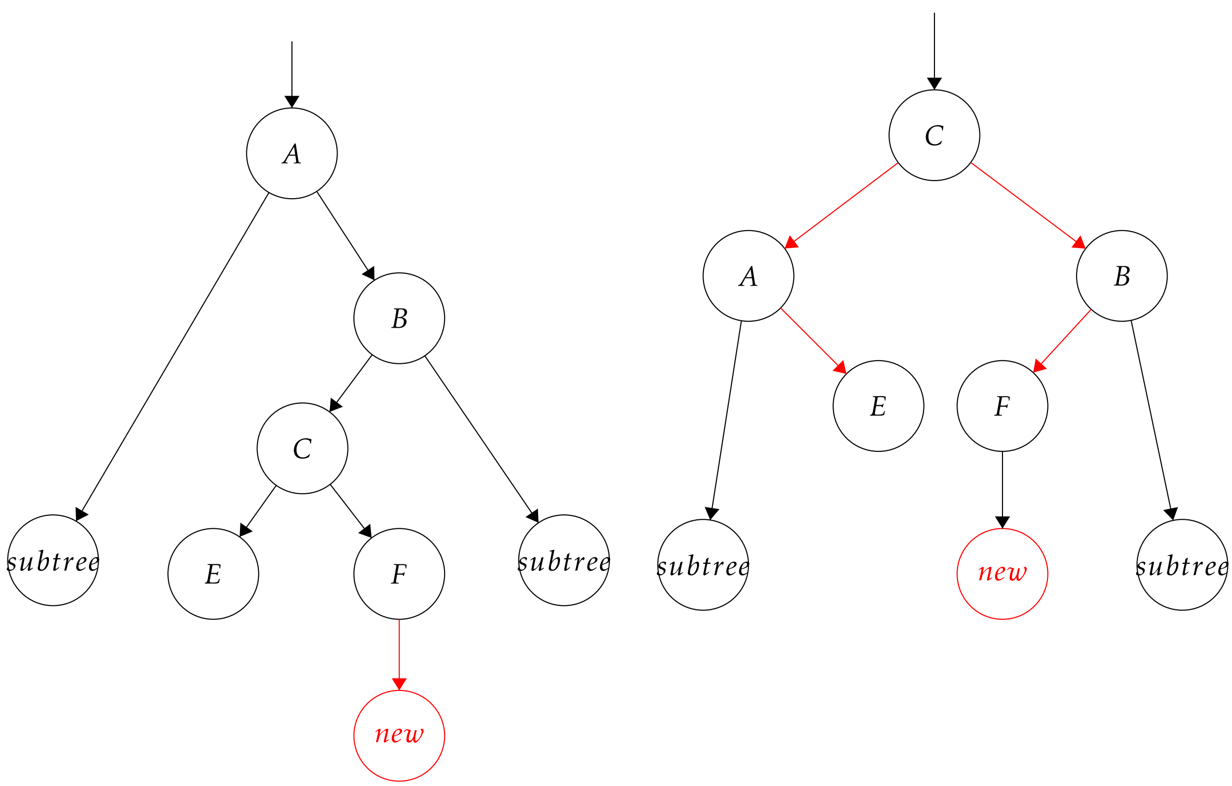
After we done the insertion, when we do $Rotation$ to maintain AVL properties:
For left rotation around A:

Case 1:



Hence, we only need to update node B and A.
$temp.sum = A.sum$, $temp.numNodes = A.numNodes$;
$A.sum = A.sum - B.sum + C.sum$, $A.numNodes = A.numNodes - B.numNodes + C.numNodes$;
$B.sum = temp.sum$, $B.numNodes = temp.numNodes$.

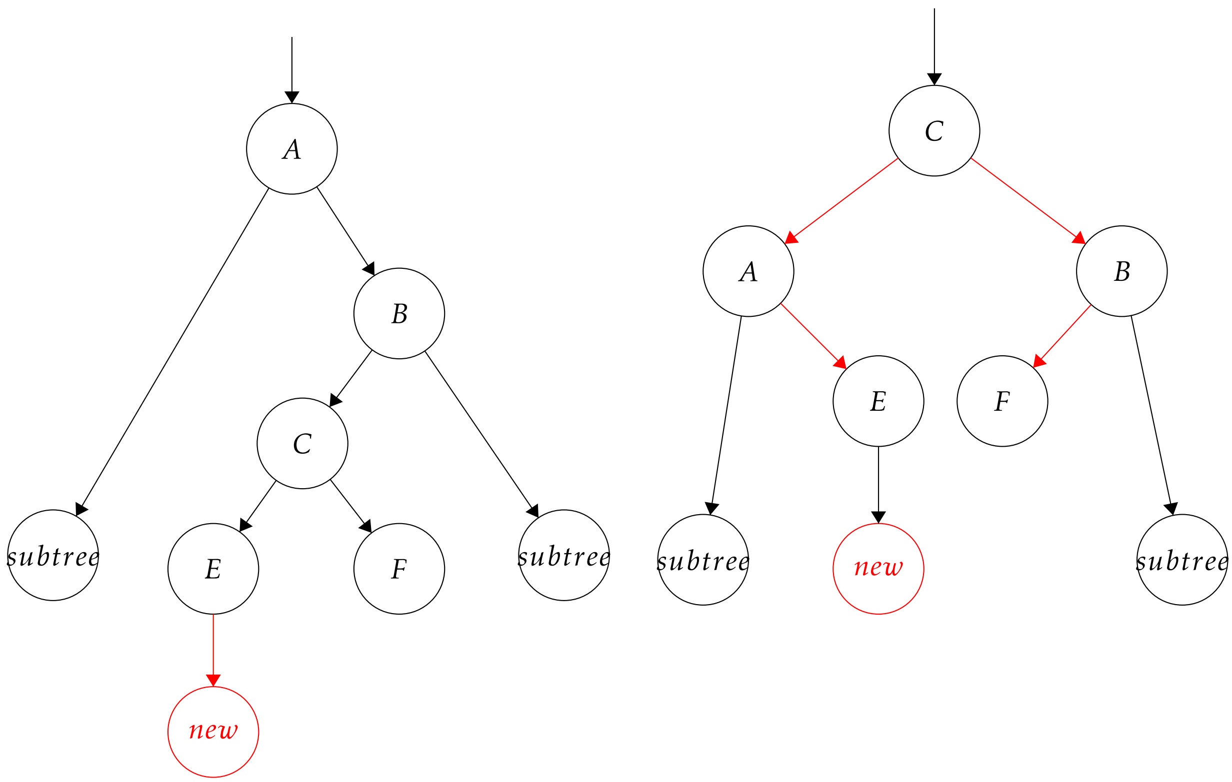
Case 2:



Clearly, we only need to update nodes A, B and C.

$temp.sum = A.sum,\ \ temp.numNodes = A.numNodes;$
$A.sum = A.sum - B.sum + E.sum,\ \ A.numNodes = A.numNodes - B.numNodes + E.numNodes;$
$B.sum = B.sum - C.sum + F.sum,\ \ B.numNodes = B.numNodes - C.numNodes + F.numNodes;$
$C.sum = temp.sum,\ C.numNodes = temp.numNodes.$

Case 3:



Also, we only need to update nodes A, B and C.
$temp.sum = A.sum,\ \ temp.numNodes = A.numNodes;$
$A.sum = A.sum - B.sum + E.sum,\ \ A.numNodes = A.numNodes - B.numNodes + E.numNodes;$
$B.sum = B.sum - C.sum + F.sum,\ \ B.numNodes = B.numNodes - C.numNodes + F.numNodes;$
$C.sum = temp.sum,\ \ C.numNodes = temp.numNodes.$

The right rotation around A is the same.
Therefore, it takes constant time to update all nodes which are changed in the rotation, which means rotation is still in order $\mathcal{O}(1)$. Hence, worst-case running time is still $\mathcal{O}(log(n))$

(c) For deletion, suppose we delete a node x, and $x = (x.key, x.numNode, x.sum)$

```
Delete(root, x):
  For all parents P of x:
    P.sum -= x.key # update all x's parents' sum
    P.numNodes -= 1 # update all x's parents' numNodes
  if x.left = Node:
    Transplant(root, x, x.right)
```

```
if x.right = Node:
   Transplant(root, x, x.left)
else:
   y ← TreeMinimum(x.right)
   For all parents P of y in subtrees of x:
      P.sum -= x.key # update all y's parents' sum (which are in x's subtrees)
if x.left = Node:
      P.numNodes -= 1 # update all y's parents' numNodes (which are in x's subtrees)
    y.sum = x.sum − x.key + y.key # update y.sum
    y.numNodes = x.numNodes − 1 # update y.numNodes
   if y.p ≠ x:
      Transplant(root, y, y.right)
       y.right ← x.right y
       y.right.p ← y
   Transplant(root, x, y)
   y.left ← x.left
   y.left.p ← y
return root
```

Clearly, in a AVL tree to get all parents of a node x, takes $\mathcal{O}(log(n))$ in the worst case, and updates their take constant time; therefore deletion is still $\mathcal{O}(log(n))$. Also, because rotation likes previous (a), still takes constant time; hence, the worst time run time is still in order $\mathcal{O}(log(n))$.