AVLNode:
info: TComp //*information from the node*
left: ↑ AVLNode //*address of left child*
right: ↑ AVLNode //*address of right child*
h: Integer //*height of the node*

AVLTree:
root: ↑ AVLNode //*root of the tree*

- **Obs:** you might need to keep a relation in the AVLTree structure as well, we will consider the $\leq$ relation by default.

```
subalgorithm recomputeHeight(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: if node ≠ NIL, h of node is set
    if node ≠ NIL then
        if [node].left = NIL and [node].right = NIL then
            [node].h ← 0
        else if [node].left = NIL then
            [node].h ← [[node].right].h + 1
        else if [node].right = NIL then
            [node].h ← [[node].left].h + 1
        else
            [node].h ← max ([[node].left].h, [[node].right].h) + 1
        end-if
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function balanceFactor(node) is:
//pre: node is an ↑ AVLNode. All descendants of node have their height (h) set
//to the correct value
//post: returns the balance factor of the node
    if [node].left = NIL and [node].right = NIL then
        balanceFactor ← 0
    else if [node].left = NIL then
        balanceFactor ← -1 - [[node].right].h //height of empty tree is -1
    else if [node].right = NIL then
        balanceFactor ← [[node].left].h + 1
    else
        balanceFactor ← [[node].left].h - [[node].right].h
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$

```
function DRR(node) is: //pre: node is an ↑ AVLNode on which we perform
the double right rotation
//post: DRR returns the new root after the rotation
    k2 ← node
    k1 ← [node].left
    k3 ← [k1].right
    k3left ← [k3].left
    k3right ← [k3].right
    //reset the links
    newRoot ← k3
    [newRoot].left ← k1
    [newRoot].right ← k2
    [k1].right ← k3left
    [k2].left ← k3right
//continued on the next slide
```

```
//recompute the heights of the modified nodes
    recomputeHeight(k1)
    recomputeHeight(k2)
    recomputeHeight(newRoot)
    DRR ← newRoot
end-function
```

- Complexity: $\Theta(1)$

```
function insertRec(node, elem) is
//pre: node is a ↑ AVLNode, elem is the value we insert in the
(sub)tree that
//has node as root
//post: insertRec returns the new root of the (sub)tree after the
insertion
   if node = NIL then
      insertRec ← createNode(elem)
   else ifelem ≤ [node].info then
       [node].left ← insertRec([node].left, elem)
   else
      [node].right ← insertRec([node].right, elem)
   end-if
   recomputeHeight(node)
   balance ← getBalanceFactor(node)
   if balance = -2 then
   //right subtree has larger height, we will need a rotation to the LEFT
      rightBalance ← getBalanceFactor([node].right)
      if rightBalance < 0 then
      //the right subtree of the right subtree has larger height, SRL
         node ← SRL(node)
      else
         node ← DRL(node)
      end-if
   else if balance = 2 then
   //left subtree has larger height, we will need a RIGHT rotation
      leftBalance ← getBalanceFactor([node].left)
      if leftBalance > 0 then
      //the left subtree of the left subtree has larger height, SRR
         node ← SRR(node)
      else
         node ← DRR(node)
      end-if
   end-if
   insertRec ← node
end-function
```

```
subalgorithm insert(tree, elem) is
//pre: tree is an AVL Tree, elem is the element to be inserted
//post: elem was inserted to tree
   tree.root ← insertRec(tree.root, elem)
end-subalgorithm
```