Knoten in einem AVL-Baum, als Sub-Klasse von BinaryTreeNode.

```
# Kleiner Trick (wie bei DoublyLinkedList): "left" <-> "right".
OTHER_SIDE = {
    "left" : "right",
    "right" : "left"
}

# Eine Sub-Klasse in Python - "erbt" von BinaryTreeNode:
class AVLTreeNode(BinaryTreeNode):
    """Class implementing (nodes of) AVL-trees, as a subclass of
    BinaryTreeNode: The key should be numeric (it is used for sorting).
    Recall: Since an AVLNode is a BinaryTreeNode, it serves also as root
    of the subtree pending from it!"""
    def __init__(self, key, data=None, parent = None, node_type = "root"):
        """Initialize:"""
        # Initialize the parent class (BinaryTreeNode), obtained by super():
        super().__init__(key, None, parent, node_type)

        # Hier ist es einfacher, die Daten in einer Liste _im_ Dictionary
        # contents zu speichern:
        if data is None:
            self.contents["data"] = []
        else:
            self.contents["data"] = [data]
        # Additionally, we need the lengths (levels) of left/right subtrees:
        # (Denn diese Information brauchen wir für das "rebalancing".)
        self.own_length = 0
        self.subtree_length = {
            'left': -1,
            'right': -1
        }

    def __str__(self):
        """Return string representation:"""
        # Aufruf der Funktion __str__ der super-Klasse:
        l,r = self.subtree_length["left"], self.subtree_length["right"]
        return super().__str__()+f', (l,r) = {(l,r)}.'

    # Auxiliary functions:
    def update_own_length(self):
        """Compute own length:"""
        # Wenn die Längen der an self hängenden Teilbäumen richtig
        # gespeichert wurden, ist das ganz leicht:
        self.own_length = max(
            self.subtree_length["left"],
            self.subtree_length["right"]
        ) + 1
        return self.own_length

# Wichtige Hilfsfunktion für die "Rotation": Ersetze den Teilbaum
# side durch den neuen Teilbaum new_subtree (noch _ohne_ "rebalancing"),
# aber mit Buchführung über eventuell geänderte Längen von Teilbäumen:
def replace_subtree(self, new_subtree, side):
    """Replace the side (left or right) subtree by new_subtree,
    return the subtree thus replaced."""
    if new_subtree == None:
        # new_subtree might by None: In this case, there are, of course,
        # no members parent and node_type.
        new_subtree_length = -1
    else:
        new_subtree.parent = self
        new_subtree.node_type = side
        new_subtree_length = new_subtree.own_length

    # Wurzel des Teilbaums, der ersetzt wird:
    replaced_subtree = self.child[side]

    # Ersetzen, und Buchführen über die Länge des neuen Teilbaums:
    self.child[side] = new_subtree
    self.subtree_length[side] = new_subtree_length
    self.update_own_length()

    # Eigentlich müssten wir auch für alle Teilbäume "oberhalb" von
    # self die Längen anpassen - aber die Funktion replace_subtree
    # soll NUR verwendet werden, wenn diese Anpassung gleich danach
    # erfolgt. Die Länge von self ist nun aber jedenfalls richtig,
    # wenn die Länge von new_subtree richtig war.

    # Nothing has changed "locally" in the replaced subtree!
    # Diesen "abgehängten" Teilbaum müssen wir in den AVL-Rotationen
    # woanders anhängen!
    return replaced_subtree
```

```
def move_up_subtree(self, side):
    """Move up subtree side (assumed to be not None!) and return it
    (this is the "rotation" according to Adelson-Velski and Landis)"""

    # Die andere Seite:
    otherside = OTHER_SIDE[side]

    # Store information for later use:
    # (in der Graphik aus der Vorlesung: self ist Knoten r)
    node_parent = self.parent
    node_type = self.node_type

    # This subtree should "move up":
    # (in der Graphik aus der Vorlesung: Knoten u)
    up_mover = self.child[side]

    # This subtree (might be None!) should "change sides":
    # (in der Graphik aus der Vorlesung: Knoten s)
    side_changer = up_mover.child[otherside]

    # Replace subtrees:

    # Wir hängen dreimal Teilbäume an, von denen wir
    # wissen, dass ihre Längen richtig sind:

    # Fügen Sie hier den passenden Code ein:

    # Längen von Teilbäumen "oberhalb von u" (die u = up_mover
    # enthalten) sind möglicherweise nicht richtig erfasst:
    # Für das "Rebalancing" (und die richtige Erfassung aller
    # Teilbaum-Längen "oberhalb") müssen wir mit dem Knoten u
    # weitermachen.
    return up_mover

def rebalance_subtrees(self):
    """Rebalance subtrees: Recall that for _every_ node we want
    to maintain the condition
        (ALL keys in the left subtree)
        <= node.key
        < (ALL keys in the right subtree),
    and balancing means maintaining the condition
        -1
        <= (self.subtree_length["left"] - self.subtree_length["right"])
        <= 1.
    """
    # self.update_own_length()
    # "Bias" of the left subtree:
    left_bias=self.subtree_length['left']-self.subtree_length['right']
    if left_bias < -1:
        side = 'right'
    elif left_bias > 1:
        side = 'left'
    else:
        # Bias in [-1,1]: No rebalancing required:
        return self

    biased_subtree = self.child[side]

    otherside = OTHER_SIDE[side]

    # Fügen Sie hier den passenden Code ein (einfache Rotation,
    # eventuell mit Vorbereitung):


    # Geben Sie den Knoten zurück, bei dem
    # Sie das "Rebalancing" fortsetzen müssen.
    return # Code einfügen!

def rebalance(self):
    """Rebalance all nodes, ascending up to the root:
    """

    # Fügen Sie hier den passenden Code ein: Sie müssen
    # im Baum "nach oben" wandern und immer wieder
    # "rebalancieren"; beachten Sie: Sie müssen die Längen
    # der an self hängenden Teilbäume in self.subtree_length
    # updaten!

    # Return the root (node with no parent), but update its
    # length first:
    # Verwenden Sie update_own_length() für die (möglicherweise
```

```
    # geänderte) Wurzel und geben Sie die Wurzel zurück:
    return # Code einfügen

def binary_search(self, key):
    """Find the position where to append key _as a leaf_:"""
    # Fügen Sie hier den passenden Code ein:
```