```python
# QUESTION 1
class Tree:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def Height(root):
        if root is None:
            return 0
        return max(Tree.Height(root.left), Tree.Height(root.right)) + 1

    def checkHeightBalanced(root):
        if root is None:
            return True
        lh = Tree.Height(root.left)
        rh = Tree.Height(root.right)
        if (abs(lh - rh) <= 1) and Tree.checkHeightBalanced(root.left) is True and Tree.checkHeightBalanced(root.right) is True:
            return True
        return False


root = Tree(1)
root.left = Tree(2)
root.right = Tree(3)
root.left.left = None
root.left.right = None
root.right.left = Tree(6)
root.right.right = Tree(7)
print(Tree.checkHeightBalanced(root))


##############################################

# QUESTION 2
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# Inorder traversal
def inorder(root):
    if root is not None:
        inorder(root.left)
        print(str(root.key) + ", ", end='')
        inorder(root.right)


def insert(node, key):
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    return node


# Find the inorder successor
def minValueNode(node):
    current = node

    # Find the leftmost leaf
    while(current.left is not None):
        current = current.left
    return current

# Deleting a node
def deleteNode(root, key):

    # Return if the tree is empty
    if not root:
        return root

    # Find the node to be deleted
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)

    else:
        # If the node is with only one child or no child
```

```
        if root.left is None and root.right is None:
            return None

        # If the node has two children
        elif root.left and root.right:

            # place the inorder successor in position of the node to be deleted
            successor = minValueNode(root.right)
            root.key = successor.key

            # Delete the inorder successor
            root.right = deleteNode(root.right, successor.key)
        else:
            child = root.left if root.left else root.right
            root = child
    return root


root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)

print("Inorder traversal: ", end=' ')
inorder(root)

print("\nDelete 10")
root = deleteNode(root, 10)
print("Inorder traversal: ", end=' ')
inorder(root)

#####################################################
# QUESTION 3

from libs.tree_map import TreeMap


class AVLTreeMap(TreeMap):
    """Sorted map implementation using an AVL tree."""

    # ---------------- nested _Node class --------------------
    class _Node(TreeMap._Node):
        """Node class for AVL maintains height value for balancing."""
        __slots__ = '_height'  # additional data member to store height

        def __init__(self, element, parent=None, left=None, right=None):
            super().__init__(element, parent, left, right)
            self._height = 0  # will be recomputed during balancing

        def left_height(self):
            return self._left._height if self._left is not None else 0

        def right_height(self):
            return self._right._height if self._right is not None else 0

    # --------------- positional-based utility methods --------------------
    def _recompute_height(self, p):
        if p is None:
            return 0
        return self._isbalanced(p)

    def getHeight(self, p):
        if not p:
            return 0
        return p._node.height()

    def _isbalanced(self, p):
        if not p:
            return 0
        return self.getHeight(p._node.left_height()) - self.getHeight(p._node.right_height()) >= -1


    def _tall_child(self, p, favorleft=False):
        # if two children have same heights, decision depends on the favorleft Boolean value.
        if p._node.left_height() + (1 if favorleft else 0) > p._node.right_height():
            return self.left(p)
        else:
            return self.right(p)
```

```
    def _tall_grandchild(self, p):
        child = self._tall_child(p)
        # if child is on left, favor left grandchild; else favor right grandchild
        alignment = (child == self.left(p))
        return self._tall_child(child, alignment)

    def _rebalance(self, p):
        while p is not None:
            old_height = p._node._height
            if not self._isbalanced(p):
                # perform trinode restructuing, setting p to resulting root,
                # and recompute new local heights after the restructuing
                p = self._restructure(self._tall_grandchild(p))
                self._recompute_height(self.left(p))
                self._recompute_height(self.right(p))
            self._recompute_height(p)
            if p._node._height == old_height:
                p = None
            else:
                p = self.parent(p)

    # ------------------ override balancing hooks -----------------------------
    def _rebalance_insert(self, p):
        self._rebalance(p)

    def _rebalance_delete(self, p):
        self._rebalance(p)


# construct the AVL tree
avl = AVLTreeMap()
input = [84, 13, 33, 91, 19, 16, 96, 26, 34, 27]

for i, n in enumerate(input):
    avl[n] = n
avl.__delitem__(16)
print(avl.__str__())

# TASK 2: After inserting node 27 into the AVL Tree, nodes 19 and 26 changed positions.
# TASK 3: After deleting node 16 into the AVL Tree, node 26 and 27 move up a position, 13 now points to 19 instead.
```