```
from binary_search_tree 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.

    We use convention that a "None" child has height 0, thus a leaf has height
1.
    """
    __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):
    p._node._height = 1 + max(p._node.left_height(), p._node.right_height())

  def _isbalanced(self, p):
    return abs(p._node.left_height() - p._node.right_height()) <= 1

  def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
    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                          # trivially 0 if new
node
      if not self._isbalanced(p):                           # imbalance
detected!
        # perform trinode restructuring, setting p to resulting root,
        # and recompute new local heights after the restructuring
        p = self._restructure(self._tall_grandchild(p))
        self._recompute_height(self.left(p))
        self._recompute_height(self.right(p))
```

```
      self._recompute_height(p)                                # adjust for recent
changes
      if p._node._height == old_height:                        # has height
changed?
        p = None                                               # no further changes
needed
      else:
        p = self.parent(p)                                     # repeat with parent

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

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