```python
#

from binary_search_tree import TreeMap

class RedBlackTreeMap(TreeMap):
  """Sorted map implementation using a red-black tree."""

  #------------------------- nested _Node class -------------------------
  class _Node(TreeMap._Node):
    """Node class for red-black tree maintains bit that denotes color."""
    __slots__ = '_red'     # add additional data member to the Node class

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._red = True     # new node red by default

  #------------------------ positional-based utility methods ------------------
-------

  # we consider a nonexistent child to be trivially black

  def _set_red(self, p): p._node._red = True
  def _set_black(self, p): p._node._red = False
  def _set_color(self, p, make_red): p._node._red = make_red
  def _is_red(self, p): return p is not None and p._node._red
  def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p)

  def _get_red_child(self, p):
    """Return a red child of p (or None if no such child)."""
    for child in (self.left(p), self.right(p)):
      if self._is_red(child):
        return child
    return None

  #------------------------ support for insertions -------------------------
  def _rebalance_insert(self, p):
    self._resolve_red(p)                         # new node is always red

  def _resolve_red(self, p):
    if self.is_root(p):
      self._set_black(p)                         # make root black
    else:
      parent = self.parent(p)
      if self._is_red(parent):                   # double red problem
        uncle = self.sibling(parent)
        if not self._is_red(uncle):              # Case 1: misshapen 4-node
          middle = self._restructure(p)          # do trinode restructuring
          self._set_black(middle)                # and then fix colors
          self._set_red(self.left(middle))
          self._set_red(self.right(middle))
        else:                                    # Case 2: overfull 5-node
          grand = self.parent(parent)
          self._set_red(grand)                   # grandparent becomes red
          self._set_black(self.left(grand))      # its children become black
          self._set_black(self.right(grand))
```

```
        self._resolve_red(grand)              # recur at red grandparent

  #------------------------- support for deletions -------------------------
  def _rebalance_delete(self, p):
    if len(self) == 1:
      self._set_black(self.root())  # special case: ensure that root is black
    elif p is not None:
      n = self.num_children(p)
      if n == 1:                    # deficit exists unless child is a red leaf
        c = next(self.children(p))
        if not self._is_red_leaf(c):
          self._fix_deficit(p, c)
      elif n == 2:                  # removed black node with red child
        if self._is_red_leaf(self.left(p)):
          self._set_black(self.left(p))
        else:
          self._set_black(self.right(p))

  def _fix_deficit(self, z, y):
    """Resolve black deficit at z, where y is the root of z's heavier
subtree."""
    if not self._is_red(y): # y is black; will apply Case 1 or 2
      x = self._get_red_child(y)
      if x is not None: # Case 1: y is black and has red child x; do "transfer"
        old_color = self._is_red(z)
        middle = self._restructure(x)
        self._set_color(middle, old_color)   # middle gets old color of z
        self._set_black(self.left(middle))   # children become black
        self._set_black(self.right(middle))
      else: # Case 2: y is black, but no red children; recolor as "fusion"
        self._set_red(y)
        if self._is_red(z):
          self._set_black(z)                 # this resolves the problem
        elif not self.is_root(z):
          self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
    else: # Case 3: y is red; rotate misaligned 3-node and repeat
      self._rotate(y)
      self._set_black(y)
      self._set_red(z)
      if z == self.right(y):
        self._fix_deficit(z, self.left(z))
      else:
        self._fix_deficit(z, self.right(z))
```