```python
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)
```

```python
from .binary_search_tree import TreeMap


class SplayTreeMap(TreeMap):
    """Sorted map implementation using a splay tree."""

    # ------------- splay operation -------------
    def _splay(self, p):
        while p != self.root():
            parent = self.parent(p)
            grand = self.parent(parent)
            if grand is None:
                # zig case
                self._rotate(p)
            elif (parent == self.left(grand)) == (p == self.left(parent)):
                # zig-zig case
                self._rotate(parent)  # move PARENT up
                self._rotate(p)  # then move p up
            else:
                # zig-zag case
                self._rotate(p)  # move p up
                self._rotate(p)  # move p up again

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

    def _rebalance_delete(self, p):
        if p is not None:
            self._splay(p)

    def _rebalance_access(self, p):
        self._splay(p)
```

```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))
```

```python
from ..tree.linked_binary_tree import LinkedBinaryTree
from ..maps.map_base import MapBase


class TreeMap(LinkedBinaryTree, MapBase):
    """Sorted map implementation using a binary search tree."""

    # ------------- override Position class -------------
    class Position(LinkedBinaryTree.Position):
        def key(self):
            """Return key of map's key-value pair."""
            return self.element()._key

        def value(self):
            """Return value of map's key-value pair."""
            return self.element()._value

    # ------------- nonpublic utilities -------------
    def _subtree_search(self, p, k):
        """Return Position of p's subtree having key k, or last node searched."""
        if k == p.key():  # found match
            return p
        elif k < p.key():  # search left subtree
            if self.left(p) is not None:
                return self._subtree_search(self.left(p), k)
        else:  # search right subtree
            if self.right(p) is not None:
                return self._subtree_search(self.right(p), k)
        return p  # unsucessful search

    def _subtree_first_position(self, p):
        """Return Position of first item in subtree rooted at p."""
        walk = p
        while self.left(walk) is not None:  # keep walking left
            walk = self.left(walk)
        return walk

    def _subtree_last_position(self, p):
        """Return Position of last item in subtree rooted at p."""
        walk = p
        while self.right(walk) is not None:  # keep walking right
            walk = self.right(walk)
        return walk

    # ------------- public methods providing "positional" support -------------
    def first(self):
        """Return the first Position in the tree (or None if empty)."""
        return self._subtree_first_position(self.root()) if len(self) > 0 else None

    def last(self):
        """Return the last Position in the tree (or None if empty)."""
        return self._subtree_last_position(self.root()) if len(self) > 0 else None

    def before(self, p):
        """Return the Position just before p in the natural order.

        Return None if p is the first position.
        """
        self._validate(p)  # inherited from LinkedBinaryTree
        if self.left(p):
            return self._subtree_last_position(self.left(p))
        else:
            # walk upward
            walk = p
            above = self.parent(walk)
            while above is not None and walk == self.left(above):
                walk = above
                above = self.parent(walk)
            return above

    def after(self, p):
        """Return the Position just after p in the natural order.

        Return None if p is the last position.
        """
        self._validate(p)  # inherited from LinkedBinaryTree
```

```
        if self.right(p):
            return self._subtree_first_position(self.right(p))
        else:
            walk = p
            above = self.parent(walk)
            while above is not None and walk == self.right(above):
                walk = above
                above = self.parent(walk)
            return above

    def find_position(self, k):
        """Return position with key k, or else neighbor (or None if empty)."""
        if self.is_empty():
            return None
        else:
            p = self._subtree_search(self.root(), k)
            self._rebalance_access(p)  # hook for balanced tree subclasses
            return p

    def delete(self, p):
        """Remove the item at given Position."""
        self._validate(p)  # inherited from LinkedBinaryTree
        if self.left(p) and self.right(p):  # p has two children
            replacement = self._subtree_last_position(self.left(p))
            self._replace(p, replacement.element())  # from LinkedBinaryTree
            p = replacement
        # now p has at most one child
        parent = self.parent(p)
        self._delete(p)  # inherited from LinkedBinaryTree
        self._rebalance_delete(parent)  # if root deleted, parent is None

    # ------------- public methods for (standard) map interface -------------
    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not found)."""
        if self.is_empty():
            raise KeyError('Key Error: ' + repr(k))
        else:
            p = self._subtree_search(self.root(), k)
            self._rebalance_access(p)  # hook for balanced tree subclasses
            if k != p.key():
                raise KeyError('Key Error: ' + repr(k))
            return p.value()

    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        if self.is_empty():
            leaf = self._add_root(self._Item(k, v))  # from LinkedBinaryTree
        else:
            p = self._subtree_search(self.root(), k)
            if p.key() == k:
                p.element()._value = v  # replace existing item's value
                self._rebalance_access(p)  # hook for balanced tree subclasses
                return
            else:
                item = self._Item(k, v)
                if p.key() < k:
                    leaf = self._add_right(p, item)  # inherited from LinkedBinaryTree
                else:
                    leaf = self._add_left(p, item)  # inherited from LinkedBinaryTree
        self._rebalance_insert(leaf)  # hook for balanced tree subclasses

    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not found)."""
        if not self.is_empty():
            p = self._subtree_search(self.root(), k)
            if k == p.key():
                self.delete(p)  # rely on positional version
                return  # successful deletion complete
            self._rebalance_access(p)  # hook for balanced tree subclasses
        raise KeyError('Key Error: ' + repr(k))

    def __iter__(self):
        """Generate an iteration of all keys in the map in order."""
        p = self.first()
        while p is not None:
            yield p.key()
```

```
            p = self.after(p)

    # ------------- public methods for sorted map interface -------------
    def __reversed__(self):
        """Generate an iteration of all keys in the map in reverse order."""
        p = self.last()
        while p is not None:
            yield p.key()
            p = self.before(p)

    def find_min(self):
        """Return (key,value) pair with minimum key (or None if empty)."""
        if self.is_empty():
            return None
        else:
            p = self.first()
            return (p.key(), p.value())

    def find_max(self):
        """Return (key,value) pair with maximum key (or None if empty)."""
        if self.is_empty():
            return None
        else:
            p = self.last()
            return (p.key(), p.value())

    def find_le(self, k):
        """Return (key,value) pair with greatest key less than or equal to k.

        Return None if there does not exist such a key.
        """
        if self.is_empty():
            return None
        else:
            p = self.find_position(k)
            if k < p.key():
                p = self.before(p)
            return (p.key(), p.value()) if p is not None else None

    def find_lt(self, k):
        """Return (key,value) pair with greatest key strictly less than k.

        Return None if there does not exist such a key.
        """
        if self.is_empty():
            return None
        else:
            p = self.find_position(k)
            if not p.key() < k:
                p = self.before(p)
            return (p.key(), p.value()) if p is not None else None

    def find_ge(self, k):
        """Return (key,value) pair with least key greater than or equal to k.

        Return None if there does not exist such a key.
        """
        if self.is_empty():
            return None
        else:
            p = self.find_position(k)  # may not find exact match
            if p.key() < k:  # p's key is too small
                p = self.after(p)
            return (p.key(), p.value()) if p is not None else None

    def find_gt(self, k):
        """Return (key,value) pair with least key strictly greater than k.

        Return None if there does not exist such a key.
        """
        if self.is_empty():
            return None
        else:
            p = self.find_position(k)
            if not k < p.key():
                p = self.after(p)
```

```
            return (p.key(), p.value()) if p is not None else None

    def find_range(self, start, stop):
        """Iterate all (key,value) pairs such that start <= key < stop.

        If start is None, iteration begins with minimum key of map.
        If stop is None, iteration continues through the maximum key of map.
        """
        if not self.is_empty():
            if start is None:
                p = self.first()
            else:
                # we initialize p with logic similar to find_ge
                p = self.find_position(start)
                if p.key() < start:
                    p = self.after(p)
            while p is not None and (stop is None or p.key() < stop):
                yield (p.key(), p.value())
                p = self.after(p)

    # ------------- hooks used by subclasses to balance a tree -------------
    def _rebalance_insert(self, p):
        """Call to indicate that position p is newly added."""
        pass

    def _rebalance_delete(self, p):
        """Call to indicate that a child of p has been removed."""
        pass

    def _rebalance_access(self, p):
        """Call to indicate that position p was recently accessed."""
        pass

    # ------------- nonpublic methods to support tree balancing -------------

    def _relink(self, parent, child, make_left_child):
        """Relink parent node with child node (we allow child to be None)."""
        if make_left_child:  # make it a left child
            parent._left = child
        else:  # make it a right child
            parent._right = child
        if child is not None:  # make child point to parent
            child._parent = parent

    def _rotate(self, p):
        """Rotate Position p above its parent.

        Switches between these configurations, depending on whether p==a or p==b.

              b                     a
             / \                   /  \
            a  t2               t0    b
           / \                        / \
          t0  t1                     t1  t2

        Caller should ensure that p is not the root.
        """
        """Rotate Position p above its parent."""
        x = p._node
        y = x._parent  # we assume this exists
        z = y._parent  # grandparent (possibly None)
        if z is None:
            self._root = x  # x becomes root
            x._parent = None
        else:
            self._relink(z, x, y == z._left)  # x becomes a direct child of z
        # now rotate x and y, including transfer of middle subtree
        if x == y._left:
            self._relink(y, x._right, True)  # x._right becomes left child of y
            self._relink(x, y, False)  # y becomes right child of x
        else:
            self._relink(y, x._left, False)  # x._left becomes right child of y
            self._relink(x, y, True)  # y becomes left child of x

    def _restructure(self, x):
        """Perform a trinode restructure among Position x, its parent, and its
```

```
grandparent.

        Return the Position that becomes root of the restructured subtree.

        Assumes the nodes are in one of the following configurations:

            z=a                 z=c           z=a               z=c
           /  \                 /  \          /  \              /  \
          t0  y=b             y=b  t3       t0   y=c          y=a  t3
             /  \             /  \              /  \          /  \
            t1  x=c          x=a  t2           x=b  t3       t0   x=b
               /  \         /  \              /  \              /  \
              t2  t3       t0  t1            t1  t2            t1  t2

        The subtree will be restructured so that the node with key b becomes its root.

                  b
                /   \
              a       c
             / \     / \
            t0  t1  t2  t3

        Caller should ensure that x has a grandparent.
        """
        """Perform trinode restructure of Position x with parent/grandparent."""
        y = self.parent(x)
        z = self.parent(y)
        if (x == self.right(y)) == (y == self.right(z)):  # matching alignments
            self._rotate(y)  # single rotation (of y)
            return y  # y is new subtree root
        else:  # opposite alignments
            self._rotate(x)  # double rotation (of x)
            self._rotate(x)
            return x  # x is new subtree root
```