

```

from linked_binary_tree import LinkedBinaryTree
from 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
        # unsuccessful 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:
        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.



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

```

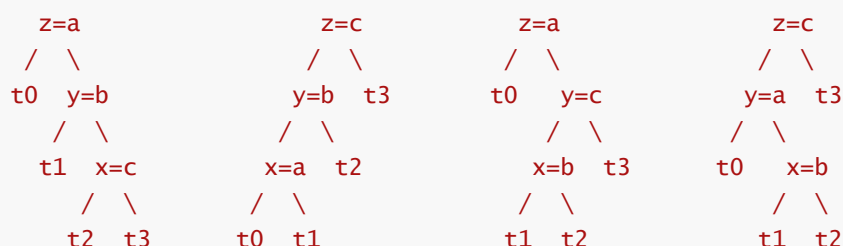
```

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

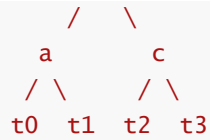
```

Return the Position that becomes root of the restructured subtree.

Assumes the nodes are in one of the following configurations:



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



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