```python
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 12 18:16:26 2017

@author: admin
"""

"""
A classic (not left-leaning) Red-Black Tree implementation, supporting addition and deletion.
"""

# The possible Node colors
BLACK = 'BLACK'
RED = 'RED'
NIL = 'NIL'



class Node:
    def __init__(self, value, color, parent, left=None, right=None):
        self.value = value
        self.color = color
        self.parent = parent
        self.left = left
        self.right = right

    def __repr__(self):
        return '{color} {val} Node'.format(color=self.color, val=self.value)
```

```
    def __iter__(self):
        if self.left.color != NIL:
            yield from self.left.__iter__()

        yield self.value

        if self.right.color != NIL:
            yield from self.right.__iter__()

    def __eq__(self, other):
        if self.color == NIL and self.color == other.color:
            return True

        if self.parent is None or other.parent is None:
            parents_are_same = self.parent is None and other.parent is None
        else:
            parents_are_same = self.parent.value == other.parent.value and self.parent.color ==
other.parent.color
        return self.value == other.value and self.color == other.color and parents_are_same

    def has_children(self) -> bool:
        """ Returns a boolean indicating if the node has children """
        return bool(self.get_children_count())

    def get_children_count(self) -> int:
        """ Returns the number of NOT NIL children the node has """
        if self.color == NIL:
            return 0
```

```
        return sum([int(self.left.color != NIL), int(self.right.color != NIL)])


class RedBlackTree:
    # every node has null nodes as children initially, create one such object for easy management
    NIL_LEAF = Node(value=None, color=NIL, parent=None)

    def __init__(self):
        self.count = 0
        self.root = None
        self.ROTATIONS = {
            # Used for deletion and uses the sibling's relationship with his parent as a guide to the rotation
            'L': self._right_rotation,
            'R': self._left_rotation
        }

    def __iter__(self):
        if not self.root:
            return list()
        yield from self.root.__iter__()

    def add(self, value):
        if not self.root:
            self.root = Node(value, color=BLACK, parent=None, left=self.NIL_LEAF, right=self.NIL_LEAF)
            self.count += 1
            return
        parent, node_dir = self._find_parent(value)
        if node_dir is None:
```

```
            return  # value is in the tree
        new_node = Node(value=value, color=RED, parent=parent, left=self.NIL_LEAF, right=self.NIL_LEAF)
        if node_dir == 'L':
            parent.left = new_node
        else:
            parent.right = new_node

        self._try_rebalance(new_node)
        self.count += 1

    def remove(self, value):
        """
        Try to get a node with 0 or 1 children.
        Either the node we're given has 0 or 1 children or we get its successor.
        """
        node_to_remove = self.find_node(value)
        if node_to_remove is None:  # node is not in the tree
            return
        if node_to_remove.get_children_count() == 2:
            # find the in-order successor and replace its value.
            # then, remove the successor
            successor = self._find_in_order_successor(node_to_remove)
            node_to_remove.value = successor.value  # switch the value
            node_to_remove = successor

        # has 0 or 1 children!
        self._remove(node_to_remove)
        self.count -= 1
```

```
def contains(self, value) -> bool:
    """ Returns a boolean indicating if the given value is present in the tree """
    return bool(self.find_node(value))

def ceil(self, value) -> int or None:
    """
    Given a value, return the closest value that is equal or bigger than it,
    returning None when no such exists
    """
    if self.root is None: return None
    last_found_val = None if self.root.value < value else self.root.value

    def find_ceil(node):
        nonlocal last_found_val
        if node == self.NIL_LEAF:
            return None
        if node.value == value:
            last_found_val = node.value
            return node.value
        elif node.value < value:
            # go right
            return find_ceil(node.right)
        else:
            # this node is bigger, save its value and go left
            last_found_val = node.value

            return find_ceil(node.left)
```

```
        find_ceil(self.root)
        return last_found_val

    def floor(self, value) -> int or None:
        """
        Given a value, return the closest value that is equal or less than it,
        returning None when no such exists
        """
        if self.root is None: return None
        last_found_val = None if self.root.value < value else self.root.value

        def find_floor(node):
            nonlocal last_found_val
            if node == self.NIL_LEAF:
                return None
            if node.value == value:
                last_found_val = node.value
                return node.value
            elif node.value < value:
                # this node is smaller, save its value and go right, trying to find a cloer one
                last_found_val = node.value

                return find_floor(node.right)
            else:
                return find_floor(node.left)

        find_floor(self.root)
        return last_found_val
```

```
def _remove(self, node):
    """
    Receives a node with 0 or 1 children (typically some sort of successor)
    and removes it according to its color/children
    :param node: Node with 0 or 1 children
    """
    left_child = node.left
    right_child = node.right
    not_nil_child = left_child if left_child != self.NIL_LEAF else right_child
    if node == self.root:
        if not_nil_child != self.NIL_LEAF:
            # if we're removing the root and it has one valid child, simply make that child the root
            self.root = not_nil_child
            self.root.parent = None
            self.root.color = BLACK
        else:
            self.root = None
    elif node.color == RED:
        if not node.has_children():
            # Red node with no children, the simplest remove
            self._remove_leaf(node)
        else:
            """
            Since the node is red he cannot have a child.
            If he had a child, it'd need to be black, but that would mean that
            the black height would be bigger on the one side and that would make our tree invalid
            """
```

```
            raise Exception('Unexpected behavior')
        else:  # node is black!
            if right_child.has_children() or left_child.has_children():  # sanity check
                raise Exception('The red child of a black node with 0 or 1 children'
                                ' cannot have children, otherwise the black height of the tree becomes invalid! ')
            if not_nil_child.color == RED:
                """
                Swap the values with the red child and remove it  (basically un-link it)
                Since we're a node with one child only, we can be sure that there are no nodes below the red
child.
                """
                node.value = not_nil_child.value
                node.left = not_nil_child.left
                node.right = not_nil_child.right
            else:  # BLACK child
                # 6 cases :o
                self._remove_black_node(node)

    def _remove_leaf(self, leaf):
        """ Simply removes a leaf node by making it's parent point to a NIL LEAF"""
        if leaf.value >= leaf.parent.value:
            # in those weird cases where they're equal due to the successor swap
            leaf.parent.right = self.NIL_LEAF
        else:
            leaf.parent.left = self.NIL_LEAF

    def _remove_black_node(self, node):
        """
```

```
        Loop through each case recursively until we reach a terminating case.
        What we're left with is a leaf node which is ready to be deleted without consequences
        """
        self.__case_1(node)
        self._remove_leaf(node)

    def __case_1(self, node):
        """
        Case 1 is when there's a double black node on the root
        Because we're at the root, we can simply remove it
        and reduce the black height of the whole tree.

            __|10B|__                  __10B__
           /         \      ==>       /       \
          9B         20B            9B        20B
        """
        if self.root == node:
            node.color = BLACK
            return
        self.__case_2(node)

    def __case_2(self, node):
        """
        Case 2 applies when
            the parent is BLACK
            the sibling is RED
            the sibling's children are BLACK or NIL
        It takes the sibling and rotates it
```

```
            40B                          60B
           /  \    --CASE 2 ROTATE-->          /  \
         |20B|   60R     LEFT ROTATE          40R   80B
  DBL BLACK IS 20----^   /  \     SIBLING 60R          /  \
            50B   80B                   |20B|  50B
        (if the sibling's direction was left of it's parent, we would RIGHT ROTATE it)
      Now the original node's parent is RED
      and we can apply case 4 or case 6
      """
      parent = node.parent
      sibling, direction = self._get_sibling(node)
      if sibling.color == RED and parent.color == BLACK and sibling.left.color != RED and
sibling.right.color != RED:
        self.ROTATIONS[direction](node=None, parent=sibling, grandfather=parent)
        parent.color = RED
        sibling.color = BLACK
        return self.__case_1(node)
      self.__case_3(node)


  def __case_3(self, node):
      """
      Case 3 deletion is when:
        the parent is BLACK
        the sibling is BLACK
        the sibling's children are BLACK
      Then, we make the sibling red and
      pass the double black node upwards
```

```
                Parent is black
        ___50B___    Sibling is black                    ___50B___
       /         \  Sibling's children are black        /         \
     30B        80B      CASE 3                  30B       |80B|  Continue with other cases
    /  \       /  \      ==>                    / \       /  \
   20B   35R   70B   |90B|<---REMOVE              20B  35R    70R   X
       / \                                     /  \
     34B   37B                               34B   37B
   """
   parent = node.parent
   sibling, _ = self._get_sibling(node)
   if (sibling.color == BLACK and parent.color == BLACK
     and sibling.left.color != RED and sibling.right.color != RED):
      # color the sibling red and forward the double black node upwards
      # (call the cases again for the parent)
      sibling.color = RED
      return self.__case_1(parent)  # start again

   self.__case_4(node)

def __case_4(self, node):
   """
   If the parent is red and the sibling is black with no red children,
   simply swap their colors
   DB-Double Black
        __10R__              __10B__       The black height of the left subtree has been incremented
       /      \             /      \      And the one below stays the same
```

```
    DB       15B     ===>    X       15R     No consequences, we're done!
           /  \                  /  \
         12B   17B             12B   17B
    """
    parent = node.parent
    if parent.color == RED:
        sibling, direction = self._get_sibling(node)
        if sibling.color == BLACK and sibling.left.color != RED and sibling.right.color != RED:
            parent.color, sibling.color = sibling.color, parent.color  # switch colors
            return  # Terminating
    self.__case_5(node)


def __case_5(self, node):
    """
    Case 5 is a rotation that changes the circumstances so that we can do a case 6
    If the closer node is red and the outer BLACK or NIL, we do a left/right rotation, depending on the
orientation
    This will showcase when the CLOSER NODE's direction is RIGHT


      ___50B___                                  __50B__
     /        \                                /      \
   30B       |80B|  <-- Double black            35B      |80B|       Case 6 is now
  / \       /  \     Closer node is red (35R)       /  \     /          applicable here,
20B  35R     70R   X     Outer is black (20B)          30R    37B  70R          so we redirect the node
    /  \              So we do a LEFT ROTATION         /  \                    to it :)
  34B  37B             on 35R (closer node)         20B   34B
    """
    sibling, direction = self._get_sibling(node)
```

```
    closer_node = sibling.right if direction == 'L' else sibling.left
    outer_node = sibling.left if direction == 'L' else sibling.right
    if closer_node.color == RED and outer_node.color != RED and sibling.color == BLACK:
        if direction == 'L':
            self._left_rotation(node=None, parent=closer_node, grandfather=sibling)
        else:
            self._right_rotation(node=None, parent=closer_node, grandfather=sibling)
        closer_node.color = BLACK
        sibling.color = RED

    self.__case_6(node)

def __case_6(self, node):
    """
    Case 6 requires
        SIBLING to be BLACK
        OUTER NODE to be RED
    Then, does a right/left rotation on the sibling
    This will showcase when the SIBLING's direction is LEFT

                    Double Black
              __50B__      |                        __35B__
             /     \     |                        /     \
  SIBLING--> 35B      |80B| <-                       30R      50R
           /  \     /                         /  \    /  \
         30R    37B  70R   Outer node is RED          20B   34B 37B    80B
        /  \           Closer node doesn't                      /
      20B   34B             matter                            70R
```

```
                    Parent doesn't
                      matter
                    So we do a right rotation on 35B!
    """
    sibling, direction = self._get_sibling(node)
    outer_node = sibling.left if direction == 'L' else sibling.right

    def __case_6_rotation(direction):
        parent_color = sibling.parent.color
        self.ROTATIONS[direction](node=None, parent=sibling, grandfather=sibling.parent)
        # new parent is sibling
        sibling.color = parent_color
        sibling.right.color = BLACK
        sibling.left.color = BLACK

    if sibling.color == BLACK and outer_node.color == RED:
        return __case_6_rotation(direction)  # terminating

    raise Exception('We should have ended here, something is wrong')

def _try_rebalance(self, node):
    """
    Given a red child node, determine if there is a need to rebalance (if the parent is red)
    If there is, rebalance it
    """
    parent = node.parent
    value = node.value
    if (parent is None  # what the fuck? (should not happen)
```

```
  or parent.parent is None  # parent is the root
  or (node.color != RED or parent.color != RED)):  # no need to rebalance
   return
grandfather = parent.parent
node_dir = 'L' if parent.value > value else 'R'
parent_dir = 'L' if grandfather.value > parent.value else 'R'
uncle = grandfather.right if parent_dir == 'L' else grandfather.left
general_direction = node_dir + parent_dir

if uncle == self.NIL_LEAF or uncle.color == BLACK:
   # rotate
   if general_direction == 'LL':
      self._right_rotation(node, parent, grandfather, to_recolor=True)
   elif general_direction == 'RR':
      self._left_rotation(node, parent, grandfather, to_recolor=True)
   elif general_direction == 'LR':
      self._right_rotation(node=None, parent=node, grandfather=parent)
      # due to the prev rotation, our node is now the parent
      self._left_rotation(node=parent, parent=node, grandfather=grandfather, to_recolor=True)
   elif general_direction == 'RL':
      self._left_rotation(node=None, parent=node, grandfather=parent)
      # due to the prev rotation, our node is now the parent
      self._right_rotation(node=parent, parent=node, grandfather=grandfather, to_recolor=True)
   else:
      raise Exception("{} is not a valid direction!".format(general_direction))
else:  # uncle is RED
   self._recolor(grandfather)
```

```
    def __update_parent(self, node, parent_old_child, new_parent):
        """
        Our node 'switches' places with the old child
        Assigns a new parent to the node.
        If the new_parent is None, this means that our node becomes the root of the tree
        """
        node.parent = new_parent
        if new_parent:
            # Determine the old child's position in order to put node there
            if new_parent.value > parent_old_child.value:
                new_parent.left = node
            else:
                new_parent.right = node
        else:
            self.root = node

    def _right_rotation(self, node, parent, grandfather, to_recolor=False):
        grand_grandfather = grandfather.parent
        self.__update_parent(node=parent, parent_old_child=grandfather,
new_parent=grand_grandfather)

        old_right = parent.right
        parent.right = grandfather
        grandfather.parent = parent

        grandfather.left = old_right  # save the old right values
        old_right.parent = grandfather
```

```
        if to_recolor:
            parent.color = BLACK
            node.color = RED
            grandfather.color = RED

    def _left_rotation(self, node, parent, grandfather, to_recolor=False):
        grand_grandfather = grandfather.parent
        self.__update_parent(node=parent, parent_old_child=grandfather,
new_parent=grand_grandfather)

        old_left = parent.left
        parent.left = grandfather
        grandfather.parent = parent

        grandfather.right = old_left  # save the old left values
        old_left.parent = grandfather

        if to_recolor:
            parent.color = BLACK
            node.color = RED
            grandfather.color = RED

    def _recolor(self, grandfather):
        grandfather.right.color = BLACK
        grandfather.left.color = BLACK
        if grandfather != self.root:
            grandfather.color = RED
        self._try_rebalance(grandfather)
```

```
def _find_parent(self, value):
    """ Finds a place for the value in our binary tree"""
    def inner_find(parent):
        """
        Return the appropriate parent node for our new node as well as the side it should be on
        """
        if value == parent.value:
            return None, None
        elif parent.value < value:
            if parent.right.color == NIL:  # no more to go
                return parent, 'R'
            return inner_find(parent.right)
        elif value < parent.value:
            if parent.left.color == NIL:  # no more to go
                return parent, 'L'
            return inner_find(parent.left)

    return inner_find(self.root)

def find_node(self, value):
    def inner_find(root):
        if root is None or root == self.NIL_LEAF:
            return None
        if value > root.value:
            return inner_find(root.right)
        elif value < root.value:
            return inner_find(root.left)
```

```
        else:
            return root

    found_node = inner_find(self.root)
    return found_node

def _find_in_order_successor(self, node):
    right_node = node.right
    left_node = right_node.left
    if left_node == self.NIL_LEAF:
        return right_node
    while left_node.left != self.NIL_LEAF:
        left_node = left_node.left
    return left_node

def _get_sibling(self, node):
    """
    Returns the sibling of the node, as well as the side it is on
    e.g

        20 (A)
       /    \
    15(B)    25(C)

    _get_sibling(25(C)) => 15(B), 'R'
    """
    parent = node.parent
    if node.value >= parent.value:
```

```
    sibling = parent.left
    direction = 'L'
else:
    sibling = parent.right
    direction = 'R'
return sibling, direction
```