<2T>

# Red-Black Tree Description

A **red-black tree** is a balanced binary search tree. All basic operations take O(log n) time in the worst case.

Properties:

1. Every node is either red or black
2. The root is black
3. Every leaf is black
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Rotations must be done when inserting or deleting to preserve the properties of the tree.

Insertion: we insert a node into the tree like a binary search tree and color it red initially. Then we call an auxiliary procedure to recolor and rotate the nodes in the tree.

Deleting a node from a red-black tree is more complicated, and we need to define a transplant procedure that swaps two nodes. Deleting a node is structurally similar to deleting a node from a binary search tree. We need a dedicated fixup function to handle updating the coloring of the red-black tree.

# Red-Black Tree Pseudocode

LEFT-ROTATE(T, x)

1 y = x.right // set y

2 x.right = y.left // turn y’s left subtree into x’s right subtree

3 if y.left ≠ T.nil

4 y.left.p = x

5 y.p = x.p // link x’s parent to y

6 if x.p = T.nil

7 T.root = y

8 elseif x == x.p.left

9 x.p.left = y

10 else x.p.right = y

11 y.left = x // put x on y’s left

12 x.p = y

RB-INSERT(T, z)

1 y = T.nil

2 x = T.root

3 while x ≠ T.nil

4 y = x

5 if z.key < x.key

6 x = x.left

7 else x = x.right

8 z.p = y

9 if y == T.nil

10 T.root = z

11 elseif z.key < y.key

12 y.left = z

13 else y.right = z

14 z.left = T.nil

15 z.right = T.nil

16 z.color = RED

17 RB-INSERT-FIXUP(T, z)

RB-INSERT-FIXUP(T, z)

1 while z.p.color = RED

2 if z.p == z.p.p.left

3 y = z.p.p.right

4 if y.color == RED

5 z.p.color = BLACK

6 y.color = BLACK

7 z.p.p.color = RED

8 z = z.p.p

9 elseif z == z.p.right

10 z = z.p

11 LEFT-ROTATE(T, z)

12 z.p.color = BLACK

13 z.p.p.color = RED

14 RIGHT-ROTATE(T, z.p.p)

15 else (same as then clause with “right” and “left” exchanged)

16 T.root.color = BLACK

RB-TRANSPLANT(T, u, v)

1 if u.p == T.nil

2 T.root = v

3 elseif u == u.p.left

4 u.p.left = v

5 else u.p.right = v

6 v.p = u.p

RB-DELETE(T, z)

1 if u.p == T.nil

2 T.root = v

3 elseif u == u.p.left

4 u.p.left = v

5 else u.p.right = v

6 v.p = u.p

RB-DELETE (T, z)

1 y = z

2 y-original-color = y.color

3 if z.left == T.nil

4 x = z.right

5 RB-TRANSPLANT (T, z, z.right)

6 elseif z.right == T.nil

7 x = z.left

8 RB-TRANSPLANT(T, z, z.left)

9 else y == TREE-MINIMUM(z.right)

10 y-original-color = y.color

11 x = y/right

12 if y.p == z

13 x.p = y

14 else RB-TRANSPLANT(T, y, y.right)

15 y.right = z.right

16 y.right.p = y

17 RB-TRANSPLANT(T, z, y)

18 y.left = z.left

19 y.left.p = y

20 y.color = z.color

21 if y-original-color == BLACK

22 RB-DELETE-FIXUP(T, x)

RB-DELETE-FIXUP(T, x)

1 while x ≠ T.root and x.color == BLACK

2 if x == x.p. left

3 w = x.p.right

4 if w.color == RED

5 w.color = BLACK // case 1

6 x.p.color = RED // case 1

7 LEFT-ROTATE (T, x.p) // case 1

8 w = x.p.right // case 1

9 if w.left.color == BLACK and w.right.color == BLACK

10 w.color = RED // case 2

11 x = x.p // case 2

12 else if w.right.color == BLACK

13 w.left.color = BLACK // case 3

14 w.color = RED // case 3

15 RIGHT-ROTATE (T, w) // case 3

16 w = x.p.right // case 3

17 w.color = x.p.color // case 4

18 x.p.color = BLACK // case 4

19 w.right.color = BLACK // case 4

20 LEFT-ROTATE (T, x.p) // case 4

21 x = T.root // case 4

22 else (same as then clause with “right” and “left” exchanged)

23 x.color = BLACK

# Red-Black Tree Code

"""

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: