CMPSC 413 - Lab 8

Red Black or AVL Tree

Exercise: Develop python codes to create a Red Black or AVL Tree. This tree should perform the following:

- Traverse nodes i.e. print all nodes

```
def in_order_helper(self, node):
    if node != self.TNULL:
        self.in_order_helper(node.left)
        print(node.key, end=" ")
        self.in_order_helper(node.right)
```

- Insert nodes

```
def insert(self, key):
    node = Node(key)
    node.parent = None
    node.key = key
    node.left = self.TNULL
    node.right = self.TNULL
    node.color = 1  # new node must be red

    y = None
    x = self.root

    while x != self.TNULL:
        y = x
        if node.key < x.key:
            x = x.left
        else:
            x = x.right

    node.parent = y
    if y is None:
        self.root = node
    elif node.key < y.key:
        y.left = node
    else:
        y.right = node

    if node.parent is None:
        node.color = 0
        return

    if node.parent.parent is None:
        return

    self.balance(node)
```

- Delete nodes

```
def delete_node_helper(self, node, key):
    z = self.TNULL
    while node != self.TNULL:
        if node.key == key:
            z = node

        if node.key <= key:
            node = node.right
        else:
            node = node.left

    if z == self.TNULL:
        print("Couldn't find key in the tree")
        return

    y = z
    y_original_color = y.color
    if z.left == self.TNULL:
        x = z.right
        self.rb_transplant(z, z.right)
    elif z.right == self.TNULL:
        x = z.left
        self.rb_transplant(z, z.left)
    else:
        y = self.minimum(z.right)
        y_original_color = y.color
        x = y.right
        if y.parent == z:
            x.parent = y
        else:
            self.rb_transplant(y, y.right)
            y.right = z.right
            y.right.parent = y

        self.rb_transplant(z, y)
        y.left = z.left
        y.left.parent = y
        y.color = z.color
    if y_original_color == 0:
        self.delete_fix(x)
```

Note: every time you add a node or delete node, the tree shouldn't violate the Red Black or AVL tree properties. create a tree with minimum 10 values and demonstrate all the functions.Attach the screenshots of the results.

Deliverables: Report, codes and the demonstration video (~3 minutes) For video demonstration, answer the following questions: 1. For the chosen self-balancing tree, explain how self-balancing tree works for keeping the tree balanced? Explain the algorithm?

Code

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.color = 1  # 1 for red, 0 for black


class RedBlackTree:
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0  # 0 for black
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    def pre_order_helper(self, node):
        if node != self.TNULL:
            print(node.key, end=" ")
            self.pre_order_helper(node.left)
            self.pre_order_helper(node.right)

    def in_order_helper(self, node):
        if node != self.TNULL:
            self.in_order_helper(node.left)
            print(node.key, end=" ")
            self.in_order_helper(node.right)

    def post_order_helper(self, node):
        if node != self.TNULL:
            self.post_order_helper(node.left)
            self.post_order_helper(node.right)
            print(node.key, end=" ")

    def search_tree_helper(self, node, key):
        if node == self.TNULL or key == node.key:
            return node

        if key < node.key:
            return self.search_tree_helper(node.left, key)
        return self.search_tree_helper(node.right, key)

    def balance(self, node):
        while node.parent.color == 1:
            if node.parent == node.parent.parent.right:
                uncle = node.parent.parent.left
                if uncle.color == 1:
                    uncle.color = 0
                    node.parent.color = 0
```

```
                node.parent.parent.color = 1
                node = node.parent.parent
            else:
                if node == node.parent.left:
                    node = node.parent
                    self.right_rotate(node)
                node.parent.color = 0
                node.parent.parent.color = 1
                self.left_rotate(node.parent.parent)
        else:
            uncle = node.parent.parent.right

            if uncle.color == 1:
                uncle.color = 0
                node.parent.color = 0
                node.parent.parent.color = 1
                node = node.parent.parent
            else:
                if node == node.parent.right:
                    node = node.parent
                    self.left_rotate(node)
                node.parent.color = 0
                node.parent.parent.color = 1
                self.right_rotate(node.parent.parent)

        if node == self.root:
            break

    self.root.color = 0

def insert(self, key):
    node = Node(key)
    node.parent = None
    node.key = key
    node.left = self.TNULL
    node.right = self.TNULL
    node.color = 1  # new node must be red

    y = None
    x = self.root

    while x != self.TNULL:
        y = x
        if node.key < x.key:
            x = x.left
        else:
            x = x.right

    node.parent = y
    if y is None:
        self.root = node
    elif node.key < y.key:
```

```
            y.left = node
        else:
            y.right = node

        if node.parent is None:
            node.color = 0
            return

        if node.parent.parent is None:
            return

        self.balance(node)

    def delete_node_helper(self, node, key):
        z = self.TNULL
        while node != self.TNULL:
            if node.key == key:
                z = node

            if node.key <= key:
                node = node.right
            else:
                node = node.left

        if z == self.TNULL:
            print("Couldn't find key in the tree")
            return

        y = z
        y_original_color = y.color
        if z.left == self.TNULL:
            x = z.right
            self.rb_transplant(z, z.right)
        elif z.right == self.TNULL:
            x = z.left
            self.rb_transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.rb_transplant(y, y.right)
                y.right = z.right
                y.right.parent = y

            self.rb_transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == 0:
```

```
        self.delete_fix(x)

    def delete_fix(self, x):
        while x != self.root and x.color == 0:
            if x == x.parent.left:
                s = x.parent.right
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.left_rotate(x.parent)
                    s = x.parent.right

                if s.left.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.right.color == 0:
                        s.left.color = 0
                        s.color = 1
                        self.right_rotate(s)
                        s = x.parent.right

                    s.color = x.parent.color
                    x.parent.color = 0
                    s.right.color = 0
                    self.left_rotate(x.parent)
                    x = self.root
            else:
                s = x.parent.left
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.right_rotate(x.parent)
                    s = x.parent.left

                if s.right.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.left.color == 0:
                        s.right.color = 0
                        s.color = 1
                        self.left_rotate(s)
                        s = x.parent.left

                    s.color = x.parent.color
                    x.parent.color = 0
                    s.left.color = 0
                    self.right_rotate(x.parent)
                    x = self.root
        x.color = 0
```

```
def rb_transplant(self, u, v):
    if u.parent is None:
        self.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    v.parent = u.parent

def minimum(self, node):
    while node.left != self.TNULL:
        node = node.left
    return node

def left_rotate(self, x):
    y = x.right
    x.right = y.left
    if y.left != self.TNULL:
        y.left.parent = x

    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y
    y.left = x
    x.parent = y

def right_rotate(self, x):
    y = x.left
    x.left = y.right
    if y.right != self.TNULL:
        y.right.parent = x

    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.right:
        x.parent.right = y
    else:
        x.parent.left = y
    y.right = x
    x.parent = y

def search_tree(self, k):
    return self.search_tree_helper(self.root, k)

def delete_node(self, data):
    self.delete_node_helper(self.root, data)
```

```
    def pre_order(self):
        self.pre_order_helper(self.root)

    def in_order(self):
        self.in_order_helper(self.root)

    def post_order(self):
        self.post_order_helper(self.root)


if __name__ == "__main__":
    bst = RedBlackTree()

    bst.insert(55)
    bst.insert(40)
    bst.insert(65)
    bst.insert(60)
    bst.insert(75)
    bst.insert(57)
    bst.insert(58)
    bst.insert(56)
    bst.insert(59)
    bst.insert(54)

    print("In order traversal of the tree:")
    bst.in_order()
    print("\n")

    bst.delete_node(55)

    print("In order traversal of the tree after deleting 55:")
    bst.in_order()
```