Name: Jigar Siddhpura **SAPID:** 60004200155

DIV: C/C2 **Branch:** Computer Engineering

AA - Experiment 4A - Red Black Tree (Insertion)

60004210155
Jigar Siddhpura
C22

Experiment 4A : RB Insertion

Aim : To implement Red black tree (insertion)

Theory :

1. RB-tree is a self-balancing BST with each node colored red / black, satisfying properties:
   a) Every node is either red / black.
   b) Root is black.
   c) Every leaf which is NIL is black.
   d) If a red node has children, then children are black.
   e) For each node, all simple paths from node to descendant leaves contains same no of black nodes.

2. Insertion begins like a standard BST insertion but may violate red black tree properties. It fixes violations through recolouring or rotations, ensuring the tree remains balanced.

3. In RB-tree, after standard BST insertion, the inserted node is colored red. This may violate RB tree properties, especially, the black height properties. To restore the balance, these are the cases:

   a) If parent of inserted node is black - No Violations.
   b) If the parent is red, violating the property that a red node cannot have a red parent, there are further cases:

i) If uncle is red, ~~set~~ ~~recolor~~ recolor node to balance.
ii) If uncle is black & inserted node is an ainside child, perform a rotation & recolor to balance.
iii) If the uncle is black & inserted node is an outside child, perform double rotation & recolor to balance.

Conclusions: RB Tree insertion maintains balance by recoloring & rotationg the nodes, ensuring efficient search & insertion op. Thus we have implemented it.

CODE :

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

class RedBlackTree:
    def __init__(self):
        self.root = None

    def insert(self, val):
        new_node = Node(val, "RED")
        if not self.root:
            self.root = new_node
            new_node.color = "BLACK"
            return

        curr = self.root
        parent = None
        while curr:
            parent = curr
            if val < curr.val:
                curr = curr.left
            else:
                curr = curr.right

        new_node.parent = parent
        if val < parent.val:
            parent.left = new_node
        else:
            parent.right = new_node

        self._fix_violations(new_node)

    def _fix_violations(self, node):
        while node.parent and node.parent.color == "RED":
            if node.parent == node.parent.parent.left:
                uncle = node.parent.parent.right
```

```
            if uncle and uncle.color == "RED":
                node.parent.color, uncle.color, node.parent.parent.color = "BLACK", "BLACK",
"RED"
                node = node.parent.parent
            else:
                if node == node.parent.right:
                    node = node.parent
                    self._left_rotate(node)
                node.parent.color, node.parent.parent.color = "BLACK", "RED"
                self._right_rotate(node.parent.parent)
        else:
            uncle = node.parent.parent.left
            if uncle and uncle.color == "RED":
                node.parent.color, uncle.color, node.parent.parent.color = "BLACK", "BLACK",
"RED"
                node = node.parent.parent
            else:
                if node == node.parent.left:
                    node = node.parent
                    self._right_rotate(node)
                node.parent.color, node.parent.parent.color = "BLACK", "RED"
                self._left_rotate(node.parent.parent)

    self.root.color = "BLACK"

  def _left_rotate(self, node):
    right_child = node.right
    node.right = right_child.left

    if right_child.left:
        right_child.left.parent = node
    right_child.parent = node.parent

    if not node.parent:
        self.root = right_child
    elif node == node.parent.left:
        node.parent.left = right_child
    else:
        node.parent.right = right_child

    right_child.left = node
    node.parent = right_child

  def _right_rotate(self, node):
    left_child = node.left
```

```
        node.left = left_child.right

        if left_child.right:
            left_child.right.parent = node
        left_child.parent = node.parent

        if not node.parent:
            self.root = left_child
        elif node == node.parent.right:
            node.parent.right = left_child
        else:
            node.parent.left = left_child

        left_child.right = node
        node.parent = left_child

    def inorder_traversal(self, node, depth=0):
        if node:
            self.inorder_traversal(node.left, depth + 1)
            print(" " * depth + f"{node.val} ({node.color})")
            self.inorder_traversal(node.right, depth + 1)

# Example usage
tree = RedBlackTree()
for val in [8, 18, 5, 15, 17, 25, 40, 80]:
    tree.insert(val)

print("Inorder traversal of Red Black Tree:")
tree.inorder_traversal(tree.root)
```

OUTPUT :

```
PS D:\SEM-6\AA\EXPERIMENTS> python -u "d:\SEM-6\AA\EXPERIMENTS\rb_insertion2.py"
Inorder traversal of Red Black Tree:
  5 (BLACK)
 8 (RED)
  15 (BLACK)
17 (BLACK)
  18 (BLACK)
 25 (RED)
  40 (BLACK)
   80 (RED)
PS D:\SEM-6\AA\EXPERIMENTS>
```