Name: Prerna Sunil Jadhav
Sap Id: 60004220127
Batch: C2-2
Course: Advance Algorithm.

EXP - 4A

AIM: Implement Red-Black Tree Operation.
4A) Insertion.

THEORY: Red Black tree are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation.
It is a binary search tree in which every node is colored with either red or black.
It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

Algorithm:
Let x be the newly inserted node.
1) Perform standard BST insertion and make the color of newly inserted node as RED.
2) If x is the root, change the color to Black.
3) Do the following if the color of x's parent is not black and x is not the root.
   a) If x's uncle in Red
      (i) change color of parent & uncle as Black
      (ii) color of grandparent RED
      (iii) Change x = x's grandparent, repeat step 2 & 3 for new x.

b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to AVL)

i) Left Left case (p is left child of g and x is left child of p)

ii) Left right case (p is left child of g and x is right child of p)

iii) Right Right case (Mirror of case i)

iv) Right left case (mirror of case ii)

Recoloring after rotations:

For left left case [3.b(i)] and Right Right case [3.b(iii)], swap colors of grandparent and parent after rotations.

For left Right case [3.b(ii)] and Right left case [3.b(iv)], swap colors of grandparent and inserted node after rotations.

CONCLUSION: The time complexity of insertion is O(log N), here N is the total no. of nodes in the red-black tree.

Hence we studied about red black tree and insertion of nodes in it.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	T. Y. B. Tech (Computer Engineering)
Course:	Advance Algorithm Laboratory
Course Code:	DJ19CEL602
Experiment No.:	04-A

AIM: Implement Red-black Tree Operations.
04-A) INSERTION

CODE:

```
# RB tree insertion
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):
        if node:
            self.inorder_traversal(node.left)
            print(f"{node.val} ({node.color})", end=" ")
            self.inorder_traversal(node.right)

# 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 C:\Users\Jadhav\Documents\BTech\Docs\6th Sem\AA\Code> & C:/msys64/mingw64/bin/python.exe "c:/Users/Jadhav/Documents
/BTech/Docs/6th Sem/AA/Code/Red-Black_Insert.py"
Inorder traversal of Red Black Tree:
5 (BLACK) 8 (RED) 15 (BLACK) 17 (BLACK) 18 (BLACK) 25 (RED) 40 (BLACK) 80 (RED)
PS C:\Users\Jadhav\Documents\BTech\Docs\6th Sem\AA\Code>
```