Experiment 4

AA Experiment 4A DATE:

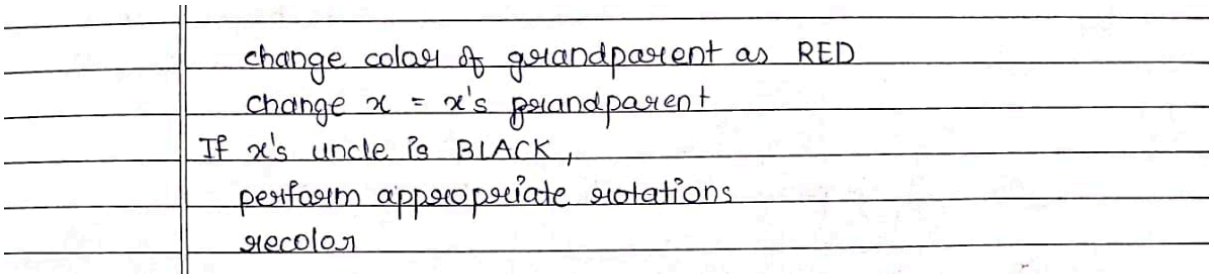
Kreena Shah
60004210243
C'32

Aim : To implement insertion in Red Black Tree

Theory :
Red Black Tree is a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced.
This balance guarantees that the time complexity for operation such as insertion, deletion & searching is always O(log n), regardless of the initial shape of tree
Red Black trees are self balancing, means that the tree adjust itself automatically after each insertion / deletion operation.

Properties :
(1) Root is black
(2) Every leaf is black in RB tree
(3) The child of a red node are black
(4) All leaves have same black depth
(5) Every path from root to descendant leaf contains same no. of black nodes
(6) self balancing

Insertion in Red Black Tree
(1) Perform standard BST insertion & make the color of newly inserted node as RED
(2) If x is root, change color of x as BLACK
(3) If x' uncle is RED
change color of parent & uncle as BLACK




```
from queue import Queue

class COLOR:
	RED = 'RED'
	BLACK = 'BLACK'

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

	def uncle(self):
		if self.parent is None or self.parent.parent is None:
			return None
		if self.parent.isOnLeft():
			return self.parent.parent.right
		else:
			return self.parent.parent.left

	def isOnLeft(self):
		return self == self.parent.left

	def sibling(self):
		if self.parent is None:
			return None
		if self.isOnLeft():
			return self.parent.right
		else:
			return self.parent.left

	def moveDown(self, new_parent):
		if self.parent is not None:
			if self.isOnLeft():
				self.parent.left = new_parent
			else:
				self.parent.right = new_parent
		new_parent.parent = self.parent
		self.parent = new_parent

	def hasRedChild(self):
```

```
                return (self.left is not None and self.left.color == COLOR.RED) or \
                        (self.right is not None and self.right.color == COLOR.RED)

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

        def leftRotate(self, x):
                new_parent = x.right
                if x == self.root:
                        self.root = new_parent
                x.moveDown(new_parent)
                x.right = new_parent.left
                if new_parent.left is not None:
                        new_parent.left.parent = x
                new_parent.left = x

        def rightRotate(self, x):
                new_parent = x.left
                if x == self.root:
                        self.root = new_parent
                x.moveDown(new_parent)
                x.left = new_parent.right
                if new_parent.right is not None:
                        new_parent.right.parent = x
                new_parent.right = x

        def swapColors(self, x1, x2):
                temp = x1.color
                x1.color = x2.color
                x2.color = temp

        def swapValues(self, u, v):
                temp = u.val
                u.val = v.val
                v.val = temp

        def fixRedRed(self, x):
                if x == self.root:
                        x.color = COLOR.BLACK
                        return
                parent = x.parent
                grandparent = parent.parent
                uncle = x.uncle()
                if parent.color != COLOR.BLACK:
                        if uncle is not None and uncle.color == COLOR.RED:
                                parent.color = COLOR.BLACK
                                uncle.color = COLOR.BLACK
                                grandparent.color = COLOR.RED
                                self.fixRedRed(grandparent)
                        else:
```

```
                    if parent.isOnLeft():
                        if x.isOnLeft():
                            self.swapColors(parent, grandparent)
                        else:
                            self.leftRotate(parent)
                            self.swapColors(x, grandparent)
                        self.rightRotate(grandparent)
                    else:
                        if x.isOnLeft():
                            self.rightRotate(parent)
                            self.swapColors(x, grandparent)
                        else:
                            self.swapColors(parent, grandparent)
                        self.leftRotate(grandparent)

    def BSTreplace(self, x):
        if x.left is not None and x.right is not None:
            return self.successor(x.right)

        if x.left is None and x.right is None:
            return None

        if x.left is not None:
            return x.left
        else:
            return x.right


    def levelOrder(self, x):
        if x is None:
            return
        q = Queue()
        q.put(x)
        while not q.empty():
            curr = q.get()
            print(curr.val, end=" ")
            if curr.left is not None:
                q.put(curr.left)
            if curr.right is not None:
                q.put(curr.right)

    def inorder(self, x):
        if x is None:
            return
        self.inorder(x.left)
        print(x.val, end=" ")
        self.inorder(x.right)

    def getRoot(self):
        return self.root
```

```
def search(self, n):
        temp = self.root
        while temp is not None:
                if n < temp.val:
                        if temp.left is None:
                                break
                        else:
                                temp = temp.left
                elif n == temp.val:
                        break
                else:
                        if temp.right is None:
                                break
                        else:
                                temp = temp.right

        return temp


def insert(self, n):
        newNode = Node(n)

        if self.root is None:
                newNode.color = COLOR.BLACK
                self.root = newNode
        else:
                temp = self.search(n)

                if temp.val == n:
                        return

                newNode.parent = temp

                if n < temp.val:
                        temp.left = newNode
                else:
                        temp.right = newNode


                self.fixRedRed(newNode)


def printInOrder(self):
        print("Inorder:")
        if self.root is None:
                print("Tree is empty")
        else:
                self.inorder(self.root)
        print()
```

```
    def printLevelOrder(self):
        print("Level order:")
        if self.root is None:
            print("Tree is empty")
        else:
            self.levelOrder(self.root)
        print()


tree = RBTree()

tree.insert(7)
tree.insert(3)
tree.insert(18)
tree.insert(10)
tree.insert(22)
tree.insert(8)
tree.insert(11)
tree.insert(26)
tree.insert(2)
tree.insert(6)
tree.insert(13)

tree.printInOrder()
tree.printLevelOrder()
```

```
PS C:\Users\Admin\OneDrive\Desktop\sem6\AA\Pracs> py .\rbInsertion.py
Inorder:
2 3 6 7 8 10 11 13 18 22 26
Level order:
10 7 18 3 8 11 22 2 6 13 26
```

Conclusion : Thus, we understood & implemented insertion in Red Black Tree.