Experiment 4B

AA Experiment 4B

DATE:

Kreena Shah
60004210243
C'32

Aim : To implement red black tree deletion

Theory :

(I) While deleting, perform BST deletion

(II) If node to be deleted is red, just delete it

(III) If root is double black (DB), just remove DB

(IV) If DB's sibling is black & both it's children are black
- remove DB
- add black to it's parent (P)
  - if P is red, it becomes black
  - if P is black, it becomes DB
- make sibling red

- If DB still exists, apply other cases

(V) If DB's sibling is Red
- swap colors of parent & it's sibling
- rotate parent in DB's direction
- reapply cases

(VI) If DB's sibling is black, siblings child who is far from DB is black, but near child is red
- swap color of DB's sibling & sibling's child who is near to DB
- rotate sibling in opposite direction to DB
- apply case VII

(VII) DB's sibling is black, far child is red
- swap colors of parent & sibling
- rotate parent in DB's direction
- change color of red child (far) to black
- remove DB


```
from queue import Queue

# Enumeration for colors
class COLOR:
```

```
    RED = 'RED'
    BLACK = 'BLACK'

class Node:
    def __init__(self, val):
        # Initialize node attributes
        self.val = val
        self.color = COLOR.RED # New nodes are always red
        self.left = None
        self.right = None
        self.parent = None

    def uncle(self):
        # Returns the uncle of the node
        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):
        # Checks if the node is the left child of its parent
        return self == self.parent.left

    def sibling(self):
        # Returns the sibling of the node
        if self.parent is None:
            return None

        if self.isOnLeft():
            return self.parent.right
        else:
            return self.parent.left

    def moveDown(self, new_parent):
        # Moves the node down by changing its 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):
        # Checks if the node has a red child
        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):
        # Initialize Red-Black Tree with an empty root
        self.root = None

    def leftRotate(self, x):
        # Performs a left rotation around the given node
        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):
        # Performs a right rotation around the given node
        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):
        # Swaps the colors of two nodes
        temp = x1.color
        x1.color = x2.color
        x2.color = temp

    def swapValues(self, u, v):
        # Swaps the values of two nodes
        temp = u.val
        u.val = v.val
        v.val = temp

    def fixRedRed(self, x):
        # Fixes the red-red violation in the tree
        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:
                # Uncle is red, perform recoloring and recurse
                parent.color = COLOR.BLACK
                uncle.color = COLOR.BLACK
                grandparent.color = COLOR.RED
                self.fixRedRed(grandparent)
            else:
                # Perform rotations based on the cases
                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 successor(self, x):
        # Finds the in-order successor of the given node
        temp = x

        while temp.left is not None:
            temp = temp.left

        return temp

    def BSTreplace(self, x):
        # Finds the replacement node in BST for the given node
        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 deleteNode(self, v):
        # Deletes the given node from the tree
        u = self.BSTreplace(v)
        uvBlack = (u is None or u.color == COLOR.BLACK) and (v.color ==
COLOR.BLACK)
        parent = v.parent

        if u is None:
            # Node to be deleted is a leaf or has only one child
            if v == self.root:
                # If the node is the root
                self.root = None
            else:
                # Detach v from the tree and move u up
                if uvBlack:
                    # u and v both black, fix double black at v
                    self.fixDoubleBlack(v)
```

```
                else:
                    # u or v red, color u black
                    if v.sibling() is not None:
                        v.sibling().color = COLOR.RED

                if v.isOnLeft():
                    parent.left = None
                else:
                    parent.right = None

            del v # Delete the node
            return

        if v.left is None or v.right is None:
            # Node to be deleted has only one child
            if v == self.root:
                # If the node is the root
                v.val = u.val
                v.left = v.right = None
                del u
            else:
                # Detach v from the tree and move u up
                if v.isOnLeft():
                    parent.left = u
                else:
                    parent.right = u

                del v
                u.parent = parent
                if uvBlack:
                    # u and v both black, fix double black at u
                    self.fixDoubleBlack(u)
                else:
                    # u or v red, color u black
                    u.color = COLOR.BLACK
        else:
            # Node to be deleted has two children, swap values with
successor and recurse
            self.swapValues(u, v)
            self.deleteNode(u)

    # Fixes the double black violation in the tree
    def fixDoubleBlack(self, x):
```

```
        # Reached root
        if x == self.root:
            return

        sibling = x.sibling()
        parent = x.parent

        # No sibling, double black pushed up
        if sibling is None:
            self.fixDoubleBlack(parent)
        else:

            # Sibling red
            if sibling.color == COLOR.RED:
                parent.color = COLOR.RED
                sibling.color = COLOR.BLACK

                # Left case
                if sibling.isOnLeft():
                    self.rightRotate(parent)

                # Right case
                else:
                    self.leftRotate(parent)
                self.fixDoubleBlack(x)
            else:

                # Sibling black
                if sibling.hasRedChild():

                    # At least 1 red child
                    if sibling.left is not None and sibling.left.color
== COLOR.RED:

                        # Left Left
                        if sibling.isOnLeft():
                            sibling.left.color = sibling.color
                            sibling.color = parent.color
                            self.rightRotate(parent)

                        # Right Left
                        else:
```

```
                            sibling.left.color = parent.color
                            self.rightRotate(sibling)
                            self.leftRotate(parent)
                    else:

                        # Left Right
                        if sibling.isOnLeft():
                            sibling.right.color = parent.color
                            self.leftRotate(sibling)
                            self.rightRotate(parent)

                        # Right Right
                        else:
                            sibling.right.color = sibling.color
                            sibling.color = parent.color
                            self.leftRotate(parent)

                # 2 black children
                else:
                    sibling.color = COLOR.RED
                    if parent.color == COLOR.BLACK:
                        self.fixDoubleBlack(parent)
                    else:
                        parent.color = COLOR.BLACK

    # Prints the level order traversal of the tree starting from the
given node
    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)
```

```
    # Prints the in-order traversal of the tree starting from the given
node
    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

    # Searches for a given value in the tree and returns the node if
found
    # Otherwise, returns the last node encountered while traversing
    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

    # Inserts the given value into the tree
    def insert(self, n):
        newNode = Node(n)

        # When the tree is empty
        if self.root is None:
            newNode.color = COLOR.BLACK
            self.root = newNode
        else:
            temp = self.search(n)
```

```
            # Value already exists, return
            if temp.val == n:
                return

            # Connect the new node to the correct node
            newNode.parent = temp

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

            # Fix red-red violation if it exists
            self.fixRedRed(newNode)

    # Deletes the node with the given value from the tree
    def deleteByVal(self, n):

        # Tree is empty
        if self.root is None:
            return

        v = self.search(n)

        if v.val != n:
            print(f"No node found to delete with value: {n}")
            return

        self.deleteNode(v)

    # Prints the in-order traversal of the tree
    def printInOrder(self):
        print("Inorder:")
        if self.root is None:
            print("Tree is empty")
        else:
            self.inorder(self.root)
        print()

    # Prints the level order traversal of the tree
    def printLevelOrder(self):
        print("Level order:")
```

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

# Test the RBTree
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()

print("Deleting 18, 11, 3, 10, 22")

tree.deleteByVal(18)
tree.deleteByVal(11)
tree.deleteByVal(3)
tree.deleteByVal(10)
tree.deleteByVal(22)

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

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

Conclusion : Thus, we understood & implemented deletion is Red-Black Tree