ADVANCE ALGORITHM
Experiment 4

Ayush Jain 60004200132 B3

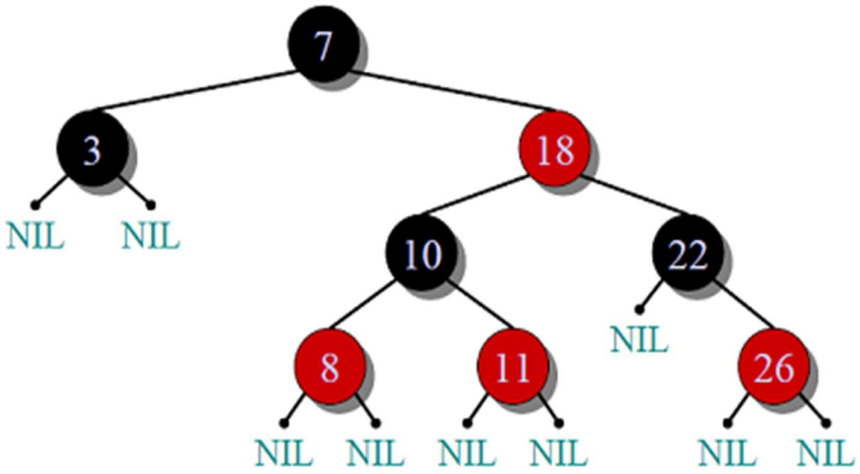
Aim : To implement Red Black Tree Insertion.

Theory:

Red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the colour (red or black) of the node. These colour bits are used to ensure the tree remains approximately balanced during insertions and deletions.
We will explore the insertion operation on a Red Black tree in the session. Inserting a value in Red Black tree takes O (log N) time complexity and O(N) space complexity.

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.



The figure depicts the basic structure of Red Black Tree.

Algorithm

Basic operations associated with Red Black Tree:

To add an element to a Red Black Tree, we must follow this algorithm:

1) Check whether tree is Empty.

2) If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

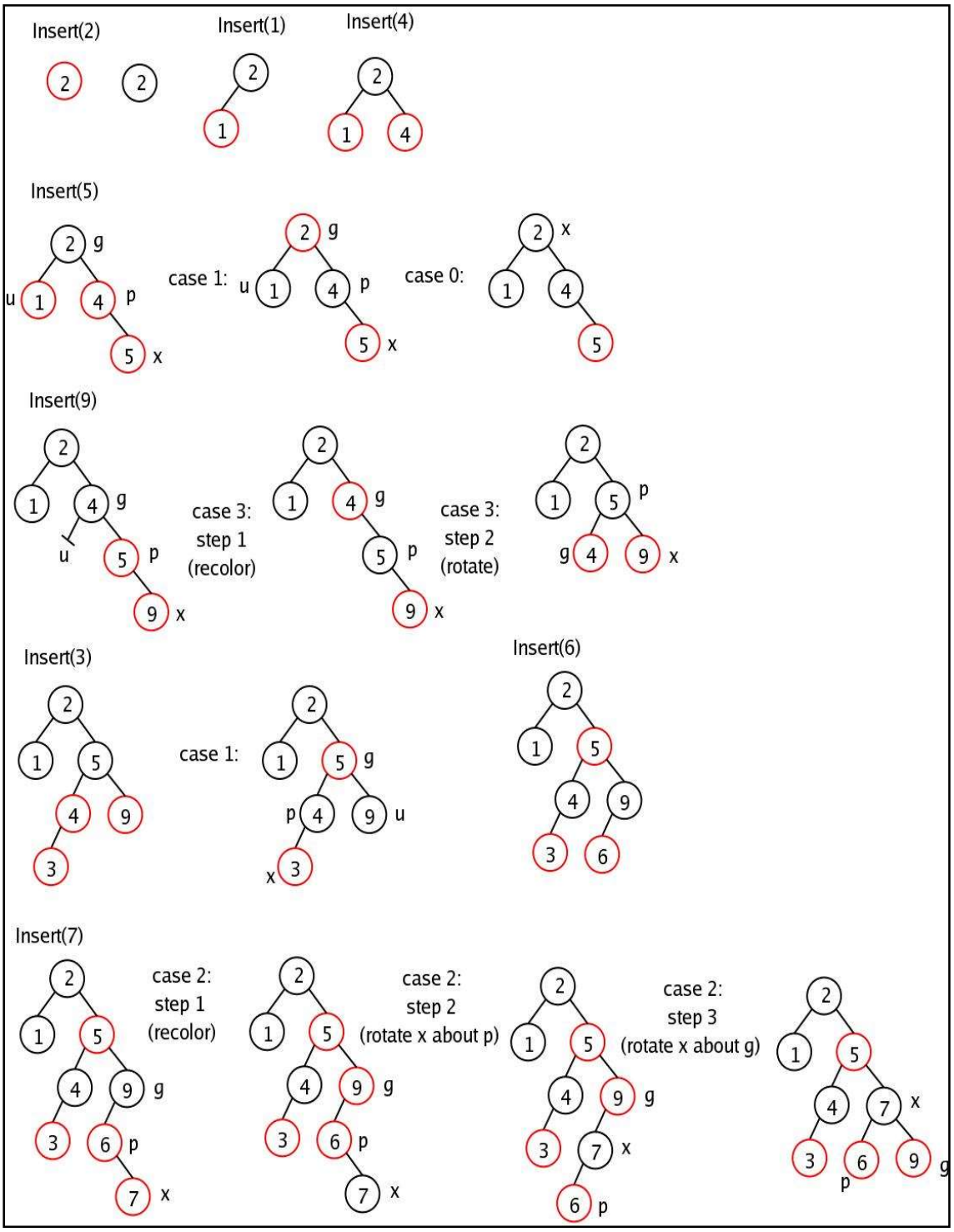
3) If tree is not Empty then insert the newNode as a leaf node with Red color.

4) If the parent of newNode is Black then exit from the operation.

5) If the parent of newNode is Red then check the color of parent node's sibling of newNode.

6) If it is Black or NULL node then make a suitable Rotation and Recolor it.

7) If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.


Insert(2)
Insert(1)
Insert(4)
Insert(5)
case 1:
case 0:
Insert(9)
case 3:
step 1
(recolor)
case 3:
step 2
(rotate)
Insert(3)
case 1:
Insert(6)
Insert(7)
case 2:
step 1
(recolor)
case 2:
step 2
(rotate x about p)
case 2:
step 3
(rotate x about g)

Code:

```
import sys

# Node creation
class Node():
    def __init__(self, item):
        self.item = item
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1


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




    # Balance the tree after insertion
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
                    print("Case 1: Uncle is Red => performing only
Recoloring:")
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    print("Case 3: Uncle is black => perform rotation followed
by recoloring")
                    if k == k.parent.left:
                        print("Rotating Right")
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    print("Rotating left")
```

```
                    self.left_rotate(k.parent.parent)

            else:
                u = k.parent.parent.right

                if u.color == 1:
                    print("Case 1: Uncle is Red => perform only Recoloring")
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent

                else:
                    print("Case 3: Uncle is Black => peroform rotation
followed by recoloring:")
                    if k == k.parent.right:
                        k = k.parent
                        print("Rotating Left")
                        self.left_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    print("Rotating Right")
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0

    # Printing the tree
    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R---> ")
                indent += "     "
            else:
                sys.stdout.write("L---> ")
                indent += "|    "

            s_color = "RED" if node.color == 1 else "BLACK"
            print(str(node.item) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)


    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 == 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 == 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 insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

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

        node.parent = y
        if y == None:
```

```
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

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

        if node.parent.parent == None:
            return

        self.fix_insert(node)

    def get_root(self):
        return self.root

    def print_tree(self):
        self.__print_helper(self.root, "", True)


if __name__ == "__main__":
    bst = RedBlackTree()
    n = int(input("Enter Number of nodes: "))
    for i in range(n):
        value  =  int(input("\nEnter value of Node {} : ".format(i+1)))
        bst.insert(value)

    bst.print_tree()

    while(True):
        print("Menu:\n1.Add new Node\n2.Exit:")
        option = int(input("Enter Option: "))
        if option == 1:
            add_new = int(input("Enter new Node: "))
            bst.insert(add_new)
            bst.print_tree()
        if option == 2:
            print("Exit")
            break
```

Output:

```
Enter Number of nodes: 5

Enter value of Node 1 : 1

Enter value of Node 2 : 2

Enter value of Node 3 : 3
Case 3: Uncle is black => perform rotation followed by recoloring
Rotating left

Enter value of Node 4 : 4
Case 1: Uncle is Red => performing only Recoloring:

Enter value of Node 5 : 5
Case 3: Uncle is black => perform rotation followed by recoloring
Rotating left
R---> 2(BLACK)
     L---> 1(BLACK)
     R---> 4(BLACK)
          L---> 3(RED)
          R---> 5(RED)
Menu:
1.Add new Node
2.Exit:
Enter Option: 2
Exit
```

Conclusion: In conclusion, we have studied how to perform insertion in Red Black Tree.