**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"**

Інститут **ІКНІ**
Кафедра **ПЗ**

# ЗВІТ

До лабораторної роботи № 9
**З дисципліни:** *"Алгоритми та структури даних"*
**На тему:** *"НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ЧЕРВОНО-ЧОРНІ ДЕРЕВА "*

**Лектор:**
доц. каф. ПЗ
Коротєєва Т.О.

**Виконав:**
ст. гр. ПЗ – 22
Ясногородський Н.В.

**Прийняв:**
асист. каф. ПЗ
Франко А.В.

« ____ » _____ 2022 р.
∑= _____ _____.

Львів – 2022

**Тема роботи:** НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ЧЕРВОНО-ЧОРНІ ДЕРЕВА
**Мета роботи:** ознайомитися з червоно-чорними деревами та отримати навички програмування алгоритмів, що їх обробляють.

## ТЕОРЕТИЧНІ ВІДОМОСТІ

Дерева як засіб реалізації словників ефективні, якщо їх висота мала, але мала висота не гарантується, і в гіршому випадку дерева не більш ефективні, ніж списки. Червоно-чорні дерева – це один з типів збалансованих дерев пошуку, в яких передбачені операції балансування гарантують, що висота дерева не перевищить O(log N).

Червоно-чорне дерево (red-black tree) – це двійкове дерево пошуку, вершини якого розділені на червоні (red) і чорні (black). Таким чином, кожна вершина зберігає один додатковий біт – її колір.

При цьому повинні виконуватися певні вимоги, які гарантують, що глибина будь-яких двох листків дерева відрізняється не більше, ніж у два рази, тому дерево можна назвати збалансованим (balanced).

Кожна вершина червоно-чорного дерева має поля color (колір), key (ключ), left (лівий нащадок), right (правий нащадок) і p (предок). Якщо у вершини відсутній нащадок або предок, відповідне поле містить значення nil. Для зручності ми будемо вважати, що значення nil, які зберігаються в полях left і right, є посиланнями на додаткові (фіктивні) листки дерева. При такому заповненні дерева кожна вершина, що містить ключ, має двох нащадків.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

## ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

Розробити програму, яка

1) читає з клавіатури ключі N, M (цілі, дійсні або символи залежно від варіанту завдання);

2) програма зберігає першу послідовність до червоно-чорного дерева;

3) кожного разу, коли до дерева додається новий елемент, потрібно вивести статистику (згідно з варіантом завдання);

4) після побудови дерева для кожного елемента другої послідовності M потрібно вивести результати наступних операцій над деревом:

1. Чи є елемент у дереві та його колір?

2. Нащадок (нащадки) та його (їх) колір.

3. Батько та його колір.

***Варіант 15: N, M –*** матриці цілих чисел розміром 3*3 (використовувати середнє значення елементів для впорядкування); *матриці з середнім значенням елементів менше за 100*

## ВИКОНАННЯ РОБОТИ

Код програми:

```
import random


from lab9.rb_tree import RedBlackTree



def gen_random_real_array(n):
    return random.sample(range(-1000, 1000), n)



# Варіант 15: N, M – матриці цілих чисел розміром 3*3
# (використовувати середнє значення елементів для впорядкування);
# матриці з середнім значенням елементів менше за 100
```

```
class Matrix:
    def __init__(self, fixed_av=None):
        if fixed_av:
            self.data = [[fixed_av]]
        else:
            self.data = [gen_random_real_array(3) for _ in range(3)]

    def _avarage(self):
        sum = 0
        for row in self.data:
            for cell in row:
                sum += cell
        return sum / (len(self.data * len(self.data[0])))

    def __lt__(self, other):
        return self._avarage() < other._avarage()

    def __eq__(self, other):
        return abs(self._avarage() - other._avarage()) < 0.1

    def __repr__(self):
        return "%.1f" % self._avarage()

    def __str__(self):
        return "%.1f" % self._avarage()

    def print_str(self):
        return "\n".join("\t".join("%0.3f" % x for x in y) for y in self.data)


if __name__ == "__main__":
```

```
N = int(input("Welcome!\nPlease enter number N: ") or 10)

bst = RedBlackTree()
for _ in range(N):
    mtrx = Matrix()
    bst.insert(mtrx)
    print(f"Inserting: {mtrx}")
    res = bst.search_less_than(Matrix(100))
    print(f"Values less than 100: {res}")
    print()

while True:
    M = float(input("Please enter value to search: ") or 10.0)
    v = bst.search_tree(Matrix(M))

    if v == bst.TNULL:
        print("Not found")
        continue

    print(f"Found: {v.item} ({bst.get_node_color(v)})")

    if not v.parent or v.parent == bst.TNULL:
        print("No parent node")
    else:
        print(f"Parent: {v.parent.item} ({bst.get_node_color(v.parent)})")

    if not v.left or v.left == bst.TNULL:
        print("No left node")
    else:
        print(f"Left node: {v.left.item} ({bst.get_node_color(v.left)})")
```

```
        if not v.right or v.right == bst.TNULL:
            print("No right node")
        else:
                                print(f"Right node: {v.right.item}
({bst.get_node_color(v.right)})")

        print()


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

    # Preorder
    def pre_order_helper(self, node):
        if node != self.TNULL:
            print(node.item + " ")
            self.pre_order_helper(node.left)
            self.pre_order_helper(node.right)
```

```
# Inorder
def in_order_helper(self, node):
    if node != self.TNULL:
        self.in_order_helper(node.left)
        print(node.item + " ")
        self.in_order_helper(node.right)

# Postorder
def post_order_helper(self, node):
    if node != self.TNULL:
        self.post_order_helper(node.left)
        self.post_order_helper(node.right)
        print(node.item + " ")

# Search the tree
def search_tree_helper(self, node, key):
    if node == self.TNULL or key == node.item:
        return node

    if key < node.item:
        return self.search_tree_helper(node.left, key)
    return self.search_tree_helper(node.right, key)

def search_less_than(self, key, node=None, acc=None):
    if acc is None:
        acc = []
    if node is None:
        node = self.root

    if node == self.TNULL:
```

```
            return acc

        if node.item < key:
            acc.append(node.item)
            self.search_less_than(key, node.left, acc)
            self.search_less_than(key, node.right, acc)
        else:
            self.search_less_than(key, node.left)

        return acc

    # 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:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right
```

```
            if u.color == 1:
                u.color = 0
                k.parent.color = 0
                k.parent.parent.color = 1
                k = k.parent.parent
            else:
                if k == k.parent.right:
                    k = k.parent
                    self.left_rotate(k)
                k.parent.color = 0
                k.parent.parent.color = 1
                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:
        print(indent)
        if last:
            print("R----")
            indent += "     "
        else:
            print("L----")
            indent += "|    "

        s_color = self.get_node_color(node)
        print(str(node.item) + "(" + s_color + ")")
        self._print_helper(node.left, indent, False)
        self._print_helper(node.right, indent, True)
```

```
def get_node_color(self, node):
    return "RED" if node.color == 1 else "BLACK"

def preorder(self):
    self.pre_order_helper(self.root)

def inorder(self):
    self.in_order_helper(self.root)

def postorder(self):
    self.post_order_helper(self.root)

def search_tree(self, k):
    return self.search_tree_helper(self.root, k)

def minimum(self, node):
    while node.left != self.TNULL:
        node = node.left
    return node

def maximum(self, node):
    while node.right != self.TNULL:
        node = node.right
    return node

def successor(self, x):
    if x.right != self.TNULL:
        return self.minimum(x.right)

    y = x.parent
```

```
    while y ≠ self.TNULL and x == y.right:
        x = y
        y = y.parent
    return y

def predecessor(self, x):
    if x.left ≠ self.TNULL:
        return self.maximum(x.left)

    y = x.parent
    while y ≠ self.TNULL and x == y.left:
        x = y
        y = y.parent

    return y

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 is 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 is 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 is None:
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

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

        if node.parent.parent is None:
            return

        self.fix_insert(node)

    def get_root(self):
        return self.root

    def print_tree(self):
        self._print_helper(self.root, "", True)
```

**ПРОТОКОЛ РОБОТИ**

```
❯ poetry run python -m lab9.main
Welcome!
Please enter number N: 3
Inserting: -147.4
Values less than 100: [-147.4]

Inserting: 158.7
Values less than 100: [-147.4]

Inserting: -121.7
Values less than 100: [-121.7, -147.4]

Please enter value to search:
```

```
Please enter value to search: 158.7
Found: 158.7 (RED)
Parent: -121.7 (BLACK)
No left node
No right node

Please enter value to search: -121.7
Found: -121.7 (BLACK)
No parent node
Left node: -147.4 (RED)
Right node: 158.7 (RED)
```

**ВИСНОВКИ**

Під час виконання лабораторної роботи я ознайомився з червоно-чорними деревами та отримав навички програмування алгоритмів, що їх обробляють.