МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Інститут комп'ютерних наук та інформаційних технологій
Кафедра програмного забезпечення

ЗВІТ
до лабораторної роботи №9
на тему: «Нелінійні структури даних:
червоно-чорні дерева»
з дисципліни: «Алгоритми і структури даних»

Лектор:
доц. кафедри ПЗ
Коротєєва Т. О.

Виконав:
ст. гр. ПЗ-22
Чаус О. М.

Прийняв:
асист. кафедри 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.

Двійкове дерево пошуку називається червоно-чорним деревом, якщо воно має такі властивості (будемо називати їх RB-властивостями, red-black properties):

1) кожна вершина або червона, або чорна;
2) Правило червоного: Якщо елемент червоний, його батько повинен бути чорним.
3) Правило шляху: Кількість чорних елементів повинна бути однаковою для всіх шляхів від кореневого елемента до елемента, що не має дочірніх або має один дочірній елемент.

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.

Використовувати готові реалізації структур даних (наприклад, STL) заборонено.

Варіант 13: N, M – матриці цілих чисел розміром 3*3 (використовувати суму елементів для впорядкування) ; матриця з максимальною сумою елементів та її колір; нащадки та їх колір.

Код програми

Файл **rbtrees.py**

```
import numpy as np

class Node:
    def __init__(self, matrix: np.ndarray):
        if matrix.shape != (3, 3):
            raise TypeError("Parameter has to be a matrix of 3x3 size")
        self.color = 1
        self.parent = None
        self.matrix = matrix
        self.right = None
        self.left = None
        self.value = np.sum(self.matrix)

class RBTree:
    def __init__(self):
        self.nil = Node(np.ndarray(shape=(3,3), dtype=int))
        self.nil.color = 0
        self.root = self.nil


    def insert_node(self, key: np.ndarray):
        node = Node(key)
        node.parent = self.nil
        node.right = self.nil
        node.left = self.nil
        y = self.nil
        x = self.root

        while x != self.nil:
            y = x
            if node.value == x.value:
                print("Node with the same sum already exists!")
                return
            if node.value < x.value:
                x = x.left
            else: x = x.right

        node.parent = y
        if y == self.nil:
            self.root = node
            node.color = 0
        elif node.value < y.value:
            y.left = node
        else: y.right = node
        self.insert_fixup(node)

    def insert_fixup(self, node):
        while node.parent.color == 1:
```

```
            if node.parent == node.parent.parent.left:
                uncle = node.parent.parent.right
                if uncle.color == 1:
                # case 1: node uncle color is red
                # set parent's and uncle's color to black, grandparent's
color to red
                # set node as parent to continue balancing
                    node.parent.color = 0
                    uncle.color = 0
                    node.parent.parent.color = 1
                    node = node.parent.parent
                else:
                    if node == node.parent.right:
                # case 2: uncle's color is black and we have a "triangle" -
                # parent is left child, node is right child
                # rotate parent to right
                        node = node.parent
                        self.rotate_left(node)
                # case 3: we have a line (both parent and child are left
children)
                # rotate grandparent to the right & recolor parent and
grandparent
                    node.parent.color = 0
                    node.parent.parent.color = 1
                    self.rotate_right(node.parent.parent)
            else:
                # the same as above, but mirrored
                uncle = node.parent.parent.left
                if uncle.color == 1:
                    node.parent.color = 0
                    uncle.color = 0
                    node.parent.parent.color = 1
                    node = node.parent.parent
                else:
                    if node == node.parent.left:
                        node = node.parent
                        self.rotate_right(node)
                    node.parent.color = 0
                    node.parent.parent.color = 1
                    self.rotate_left(node.parent.parent)
            self.root.color = 0
            if node == self.root: break

    def rotate_right(self, node):
        y = node.left
        node.left = y.right
        y.parent = node.parent
        if node.parent == self.nil:
            self.root = y
        elif node == node.parent.right:
            node.parent.right = y
        else: node.parent.left = y
        y.right = node
```

```
        node.parent = y

    def rotate_left(self, node):
        y = node.right
        node.right = y.left
        y.parent = node.parent
        if node.parent == self.nil:
            self.root = y
        elif node == node.parent.right:
            node.parent.right = y
        else: node.parent.left = y
        y.left = node
        node.parent = y

    def print(self, node, level=0):
        if node != self.nil:
            self.print(node.right, level + 1)
            print((" " * 8 * level) + str(node.value) + ": " + ("Red" if
node.color else "Black"))
            self.print(node.left, level + 1)

    def print_tree(self):
        print("")
        self.print(self.root, 0)
        print("")

    def _print_node(self, node):
        print(node.matrix)
        print(f"Sum is {node.value}, color: " + ("Red" if node.color else
"Black") +
            "\n\nChildren:\n" + "-" * 20)
        print("Left child: ")
        if node.left != self.nil:
            print(node.left.matrix)
        else: print("NIL")
        print((f"Sum is {node.left.value}, " if node.left != self.nil else
"") + "color: "
              + ("Red" if node.left.color else "Black"))
        print("Right child: ")
        if node.right != self.nil:
            print(node.right.matrix)
        else: print("NIL")
        print((f"Sum is {node.right.value}, " if node.right != self.nil else
"") + "color: "
              + ("Red" if node.right.color else "Black"))
        print(("-" * 20) + "\nParent:")
        if node.parent != self.nil:
            print(node.parent.matrix)
        else: print("NIL")
        print((f"Sum is {node.parent.value}, " if node.parent != self.nil
else "") + "color: "
              + ("Red" if node.parent.color else "Black"))
```

```
    def print_max(self):
        if self.root != self.nil:
            node = self.root
            while node.right != self.nil:
                node = node.right
            print(f"Matrix with max sum:")
            self._print_node(node)
        else: print("The tree is empty!")

    def find_node(self, value):
        x = self.root
        while x != self.nil:
            if value == x.value:
                print(f"Element {value} found")
                self._print_node(x)
                return
            elif value < x.value:
                x = x.left
            else:
                x = x.right
        print(f"Element with sum {value} not found")

    def find_node_matrix(self, matrix):
        value = np.sum(matrix)
        x = self.root
        while x != self.nil:
            if value == x.value:
                print(f"Element {value} found")
                self._print_node(x)
                return
            elif value < x.value:
                x = x.left
            else:
                x = x.right
        print(f"Element with sum {value} not found")
```

Файл **main.py**

```
import numpy as np
import rbtrees as rb

def main():
    tree = rb.RBTree()
    while True:
        print("\nIf you want to add a new node, type '1'.")
        print("If you want to find nodes in the tree, type '2'.")
        print("If you want to close the program, type '3'.\n")
        user_input = int(input())
        if user_input == 1:
            input_list = list()
            while True:
                try:
```

```
                    input_list = list((int(i) for i in input("Enter 9 values
divided by spaces:\n").split()))
                    if len(input_list) != 9:
                        raise Exception("Incorrect length was given! Please
try again.")
                except ValueError:
                    print("Incorrect value given! Please try again.")
                    continue
                except Exception as err:
                    print(err)
                    continue
                break
            input_matrix = np.array(input_list)
            input_matrix = np.reshape(input_list, (3, 3))
            tree.insert_node(input_matrix)
            print("-" * 20)
            tree.print_tree()
            print("-" * 20)
            tree.print_max()
        elif user_input == 2:
            list_size = int(input("How many nodes do you want to find?\n"))
            pass_by_sum = (input("If you want to find elements by their sum,
type 1.\n"
                "If you want to find elements by matrices, type 0\n") ==
"1")
            if pass_by_sum:
                sum_list = list()
                for i in range(list_size):
                    sum_list.append(int(input("Enter sum of element you want
to find:\n")))
                for sum in sum_list:
                    tree.find_node(sum)
            else:
                matrix_list = list()
                for i in range(list_size):
                    while True:
                        try:
                            input_list = list((int(i) for i in input("Enter
9 values divided by spaces:\n").split()))
                            if len(input_list) != 9:
                                raise Exception("Incorrect length was given!
Please try again.")
                        except ValueError:
                            print("Incorrect value given! Please try
again.")
                            continue
                        except Exception as err:
                            print(err)
                            continue
                        break
                    input_matrix = np.array(input_list)
                    input_matrix = np.reshape(input_list, (3, 3))
                    matrix_list.append(input_matrix)
```

```
                for matrix in matrix_list:
                    tree.find_node_matrix(matrix)
                    print("-" * 20)
        else:
            break


if __name__ == "__main__":
    main()

}
```

Зображення програми

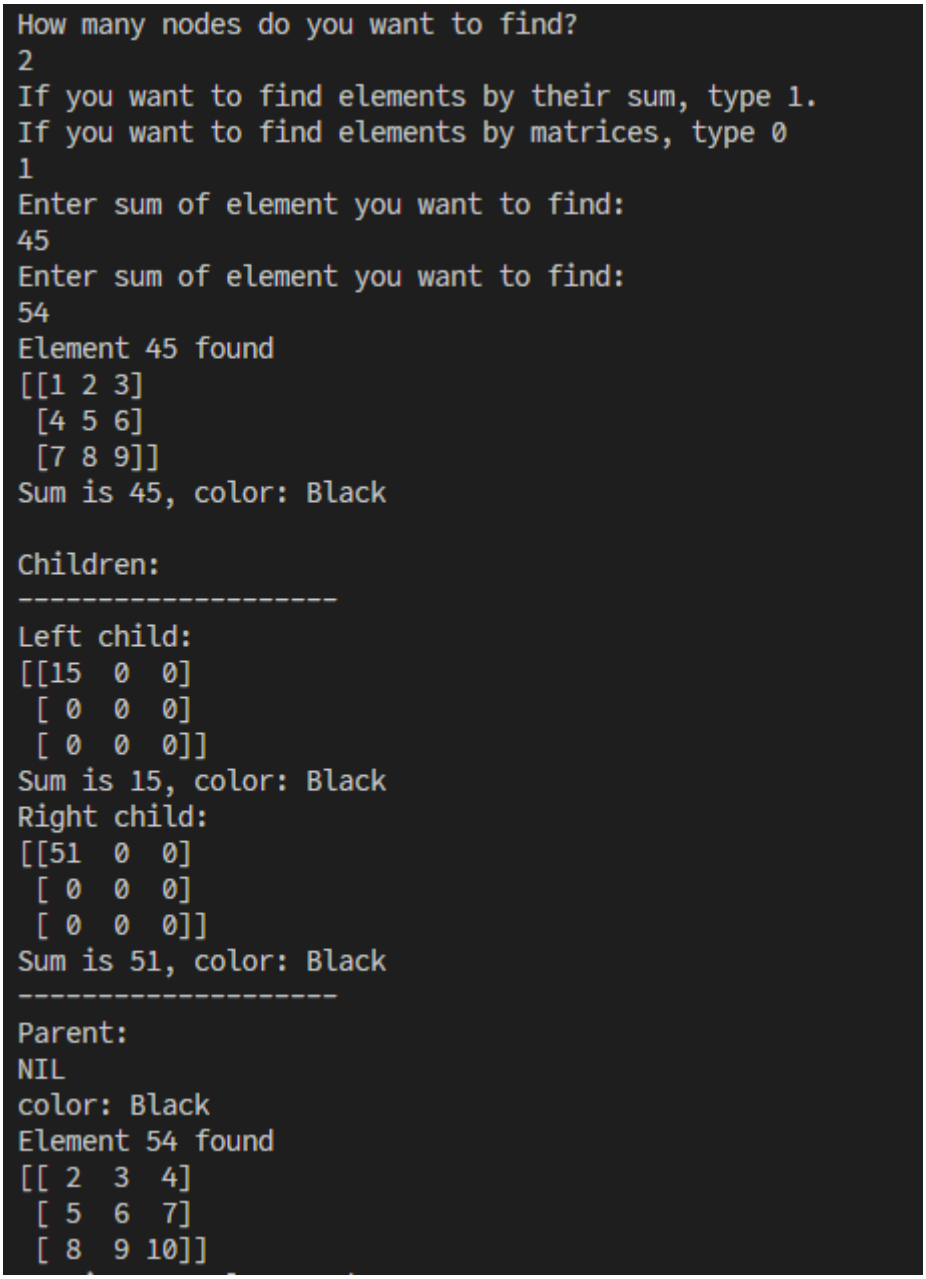
```
If you want to add a new node, type '1'.
If you want to find nodes in the tree, type '2'.
If you want to close the program, type '3'.

1
Enter 9 values divided by spaces:
1 2 3 4 5 6 7 8 9
--------------------

45: Black

--------------------
Matrix with max sum:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Sum is 45, color: Black

Children:
--------------------
Left child:
NIL
color: Black
Right child:
NIL
color: Black
--------------------
Parent:
NIL
color: Black
```



Висновок:

познайовся з червоно-чорними деревами та отримав навички програмування алгоритмів, що їх обробляють.