**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

Солтисюк Д.А.

**Прийняв:**

асист. каф. ПЗ

Франко А.В.

« \_\_\_\_ » \_\_\_\_\_\_\_\_ 2022 р.

∑= \_\_\_\_\_ .

Львів – 2022

**Тема роботи:** НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ЧЕРВОНО-ЧОРНІ ДЕРЕВА

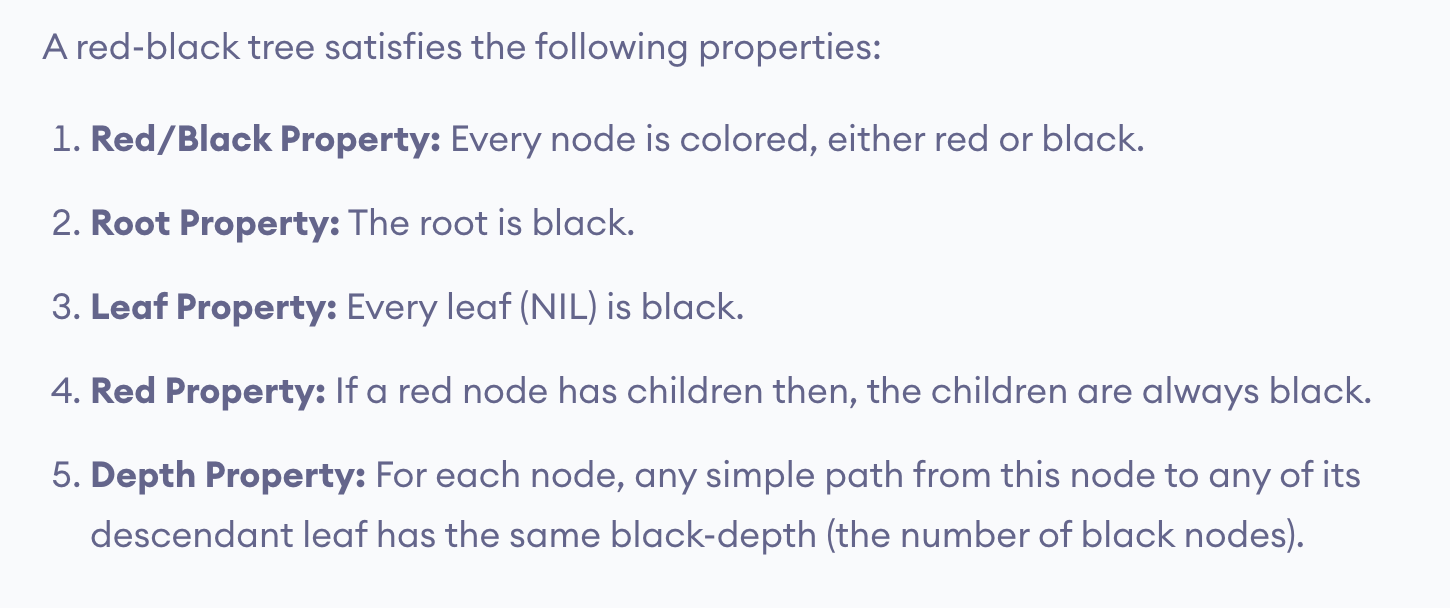
**Мета роботи:** ознайомитися з червоно-чорними деревами та отримати навички програмування алгоритмів, що їх обробляють.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

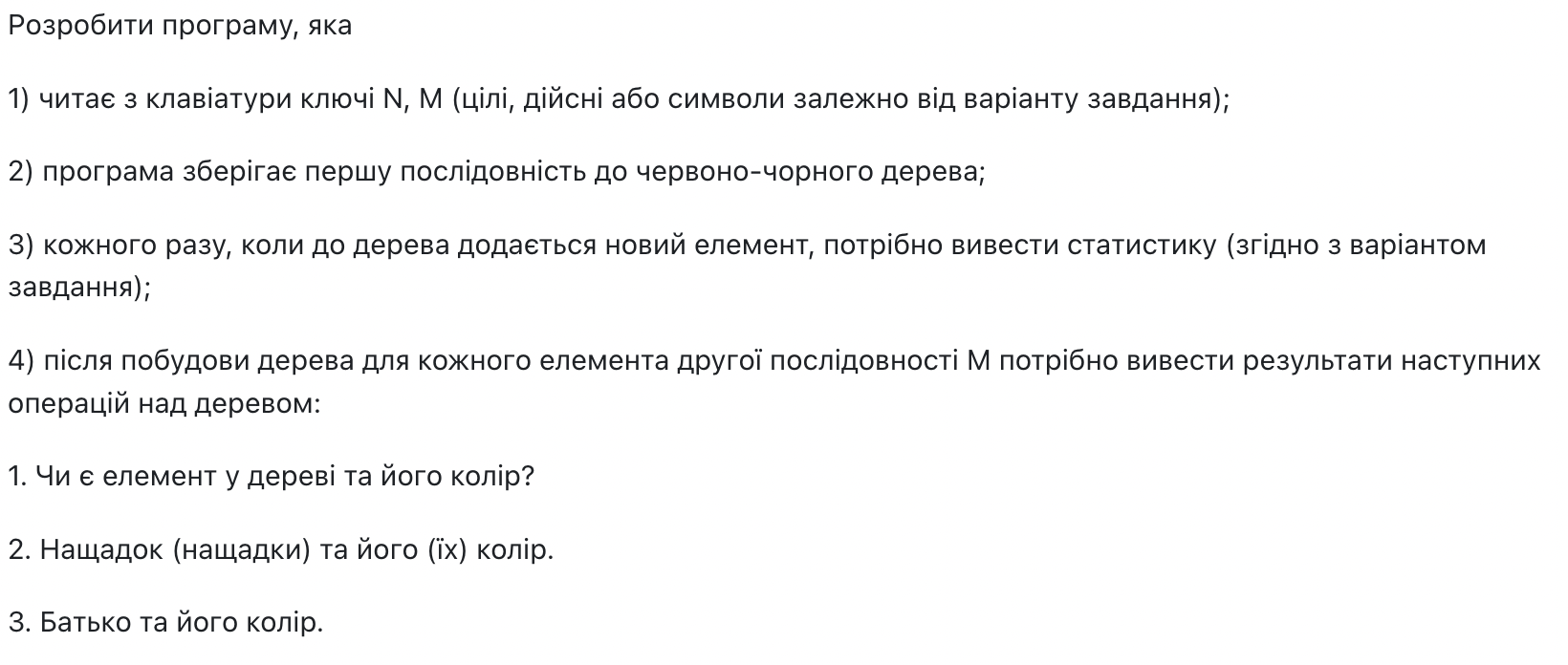
Дерева як засіб реалізації словників ефективні, якщо їх висота мала, але мала висота не гарантується, і в гіршому випадку дерева не більш ефективні, ніж списки. Червоно-чорні дерева – це один з типів збалансованих дерев пошуку, в яких передбачені операції балансування гарантують, що висота дерева не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

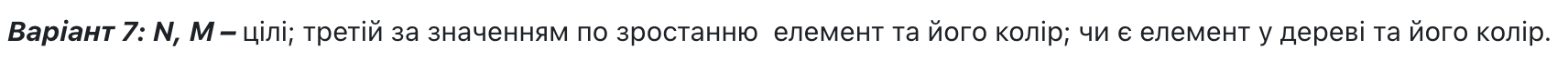
При цьому повинні виконуватися певні вимоги, які гарантують, що глибина будь-яких двох листків дерева відрізняється не більше, ніж у два рази, тому дерево можна назвати збалансованим (balanced).

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.

**ІНДИВІДУАЛЬНЕ ЗАВДАННЯ**

Загальна умова завдання:

Персональний варіант:22 (7):

****

**ВИКОНАННЯ РОБОТИ**

**Код програми:**

Файл main.py:

**import** random

**from** lab9.rb\_tree **import** RedBlackTree

**def** gen\_random\_real\_array(n):

**return** random.sample(range(-1000, 1000), n)

**if** \_\_name\_\_ == "\_\_main\_\_":

**print**()

N = int(input("Enter number N: ") **or** 10)

**print**()

bst = RedBlackTree()

arr = gen\_random\_real\_array(N)

**for** x **in** range(N):

bst.insert(x)

**print**(f"Inserted: {x}")

**print**()

**print**("Resulting tree")

bst.print\_tree()

**print**()

**while True**:

M = float(input("Please enter value to search: ") **or** 10.0)

v = bst.search\_tree(M)

**if** v == bst.TNULL:

**print**("Not found")

**print**()

**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**()

Файл rb\_tree.py:

"""

General information:

1) Red-Black tree should be a binary search tree:

- left is lesser values, while right - bigger

- subtrees should be also binary search trees

2) Root and Leafs are black

3) The child node is always black if the parent node is red in color,

therefore, there should not be two consecutive red nodes

4) Every path from the root node to any leaf node should have the same

number of black-colored nodes

Quick facts:

1) Removes the con of binary search tree, when it grows linearly:

time complexity reduces from O(n) to O(n\*log(n))

2) Red-black tree makes fewer structural changes, hence is faster than AVL tree

"""

**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, end=" ")

**if** last:

**print**("R----", end=" ")

indent += " "

**else**:

**print**("L----", end=" ")

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**)

**def** third\_from\_min(self):

min\_node = self.minimum(self.root)

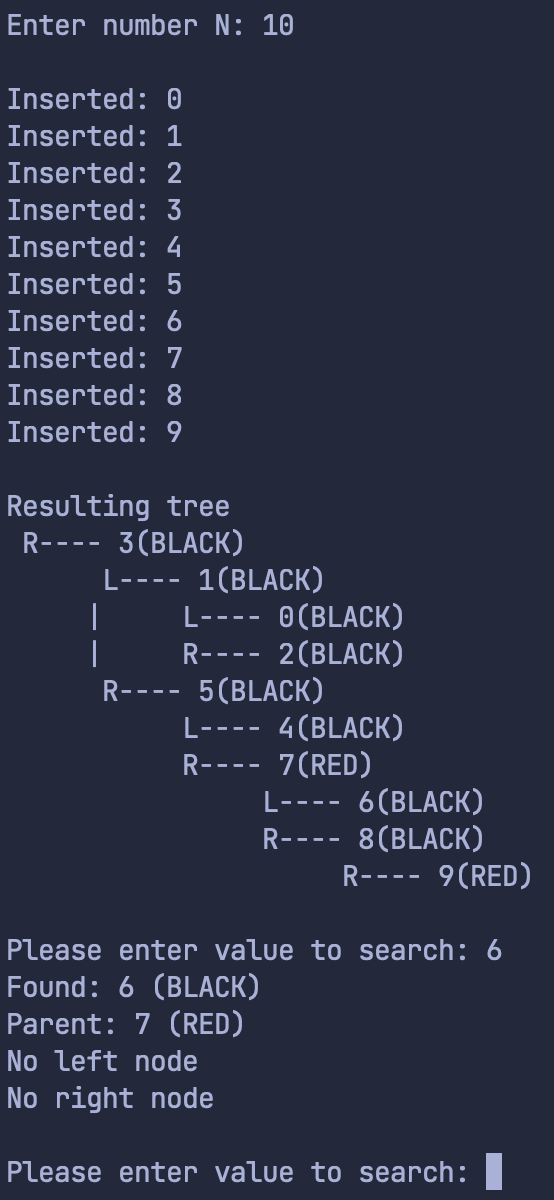
**if** min\_node.parent.right != self.TNULL:

**return** min\_node.parent.right

**return** min\_node.parent.parent

**ПРОТОКОЛ РОБОТИ**

Результатом роботи є командна утиліта, яка генерує рандомний N-значний масив та записує його в червоно-чорне дерево. Під час кожної ітерації по масиву виводяться значення, які були занесені у дерево, а також загальний вигляд самого дерева. Після виконаних операцій, програма очікує ввід користувача для пошуку елементу у дереві та виводу статистичної інформації.

****

**ВИСНОВКИ**

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