|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-14 «Цифровые технологии обработки данных»

**ОТЧЕТ   
о выполнении лабораторной работы №3**

**«Деревья двоичного поиска»**

**по дисциплине   
«Алгоритмы и структуры данных»**

**Вариант № 45**

Выполнил: студент 2 курса

группы **\_\_\_\_\_\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(фио студента)*

Проверил: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2021 г.

**Задание на лабораторную работу № 3.**

Реализовать в виде программы абстрактный тип данных «Дерево» согласно варианту (Номер варианта – две последние цифры шифра студента, номера зачетной книжки) с учетом заданного представления дерева.

Пусть А, В, С – деревья соответствующего типа, узлы которых могут содержать целочисленные значения. Требуется реализовать начальное формирование деревьев А и В, путем добавления некоторой последовательности значений (узлов) в пустое дерево.

После чего требуется по варианту реализовать заданную операцию над деревьями без использования каких-либо вспомогательных структур (списков, массивов и т.п.), работая только с узлами деревьев А и В.

Операция А=A ⋃прB означает, что элементы дерева В будут добавлены в дерево А в прямом порядке обхода дерева В, соответственно А=A ⋃обрB – в обратном, а А=A ⋃симB – симметричном обходе дерева В.

Операция А = A ⋂ B означает, что из дерева А исключаются узлы, отсутствующие в дереве В.

Операция А = A \ B означает, что из дерева А исключаются узлы, присутствующие в дереве В.

**Вариант № 45**

**Тип дерева: Оптимальное дерево двоичного поиска**

**Реализация дерева: Левый сын, правый брат (Указатели)**

**Операция: С=A \ B**

**Порядок обхода: A – Обратный, B - Симметричный**

**Теория о Деревьях двоичного поиска.**

Дерево двоичного поиска — это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла. Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева. Узел дерева, не имеющий потомков, называется листом.



Рисунок 1. – Иллюстрация дерева A.

Симметричный обход дерева идет в следующем порядке: левый потомок, корень, правый потомок. Пример обхода согласно дереву на рисунке 1: 40, 50, 60, 70, 80, 90, 100.

Прямой обход - корень, левый потомок, правый потомок. Пример обхода согласно дереву на рисунке 1: 70, 50, 40, 60, 90, 80,100.

Обратный обход - левый потомок, правый потомок, корень. Пример обхода согласно дереву на рисунке 1: 40, 60, 50, 80, 100, 90,70.



Рисунок 2. – Иллюстрация дерева B.



Рисунок 3. – Иллюстрация дерева C.

Иллюстрация дерева С, полученного добавлением элементов из дерева B на рисунке 2 в дерево A на рисунке 1 в прямом обходе дерева B, представлена на рисунке 3. Прямой обход полученного дерева C: 70, 50, 40, 35, 45, 60, 55, 65, 90, 80, 75, 85, 100, 95, 105.

**В качестве оптимального дерева выбрано АВЛ-дерево.**

**АВЛ-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. **Балансировка**

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

Используются 4 типа вращений:

[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_LR.GIF)1.**Малое левое вращение** Данное вращение используется тогда, когда высота b-поддерева — высота L = 2 и высота С <= высота R.

[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_BR.GIF)2.**Большое левое вращение** Данное вращение используется тогда, когда высота b-поддерева — высота L = 2 и высота c-поддерева > высота R.

[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_LL.GIF)3.**Малое правое вращение** Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота С <= высота L.

[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_BL.GIF)4.**Большое правое вращение** Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота c-поддерева > высота L.

В каждом случае достаточно просто доказать то, что операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться. Также можно заметить, что большое вращение это комбинация правого и левого малого вращения. Из-за условия балансированности высота дерева О(log(N)), где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций

**Листинг программы.**

from typing import Generic, TypeVar, Optional, List

VT = TypeVar("VT")

class Node(Generic[VT]):

    value: VT

    left: Optional['Node[VT]']

    right: Optional['Node[VT]']

    height: int

    def \_\_init\_\_(self, value: VT):

        self.value = value

        self.left = None

        self.right = None

        self.height = 1

    def \_\_repr\_\_(self):

        return str(self.value)

class AVL\_BST(Generic[VT]):

    \_root: Node[VT]

    def \_\_init\_\_(self):

        self.\_root = None

    def \_find(self, root: Node[VT], value: VT) -> Optional[Node[VT]]:

        if root is None or root.value == value:

            return root

        elif value < root.value:

            return self.\_find(root.left, value)

        else:

            return self.\_find(root.right, value)

    def find(self, value: VT) -> Optional[Node[VT]]:

        return self.\_find(self.\_root, value)

    def \_fix\_height(self, root: Node[VT]):

        root.height = 1 + max(self.\_height(root.left), self.\_height(root.right))

    def \_insert(self, root: Node[VT], value: VT) -> Node[VT]:

        if not root:

            return Node(value)

        elif value < root.value:

            root.left = self.\_insert(root.left, value)

        elif value > root.value:

            root.right = self.\_insert(root.right, value)

        else:

            return root

        self.\_fix\_height(root)

        balance\_factor = self.\_balance\_factor(root)

        if balance\_factor > 1:

            if value < root.left.value:

                return self.\_right\_rotate(root)

            else:

                root.left = self.\_left\_rotate(root.left)

                return self.\_right\_rotate(root)

        if balance\_factor < -1:

            if value > root.right.value:

                return self.\_left\_rotate(root)

            else:

                root.right = self.\_right\_rotate(root.right)

                return self.\_left\_rotate(root)

        return root

    def \_remove(self, root: Node[VT], value: VT) -> Node[VT]:

        if not root:

            return root

        elif value < root.value:

            root.left = self.\_remove(root.left, value)

        elif value > root.value:

            root.right = self.\_remove(root.right, value)

        else:

            if root.left is None:

                temp = root.right

                root = None

                return temp

            elif root.right is None:

                temp = root.left

                root = None

                return temp

            temp = self.\_min(root.right)

            root.value = temp.value

            root.right = self.\_remove(root.right, temp.value)

        if root is None:

            return root

        self.\_fix\_height(root)

        balance\_factor = self.\_balance\_factor(root)

        if balance\_factor > 1:

            if self.\_balance\_factor(root.left) >= 0:

                return self.\_right\_rotate(root)

            else:

                root.left = self.\_left\_rotate(root.left)

                return self.\_right\_rotate(root)

        if balance\_factor < -1:

            if self.\_balance\_factor(root.right) <= 0:

                return self.\_left\_rotate(root)

            else:

                root.right = self.\_right\_rotate(root.right)

                return self.\_left\_rotate(root)

        return root

    def \_left\_rotate(self, z: Node[VT]) -> Optional[Node[VT]]:

        y = z.right

        z.right = y.left

        y.left = z

        self.\_fix\_height(z)

        self.\_fix\_height(y)

        return y

    def \_right\_rotate(self, z: Node[VT]) -> Optional[Node[VT]]:

        y = z.left

        z.left = y.right

        y.right = z

        self.\_fix\_height(z)

        self.\_fix\_height(y)

        return y

    def \_height(self, root: Node[VT]) -> int:

        if not root:

            return 0

        return root.height

    def \_balance\_factor(self, root) -> int:

        if not root:

            return 0

        return self.\_height(root.left) - self.\_height(root.right)

    def \_min(self, root: Node[VT]) -> Node[VT]:

        if root is None or root.left is None:

            return root

        return self.\_min(root.left)

    def height(self) -> int:

        return self.\_height(self.\_root)

    def insert(self, value: VT):

        self.\_root = self.\_insert(self.\_root, value)

    def remove(self, value: VT):

        self.\_root = self.\_remove(self.\_root, value)

    def \_traverse\_inorder(self, root: Node[VT], nodes\_list: List[Node[VT]]):

        if root is not None:

            self.\_traverse\_inorder(root.left, nodes\_list)

            nodes\_list.append(root)

            self.\_traverse\_inorder(root.right, nodes\_list)

        return nodes\_list

    def traverse\_inorder(self) -> List[Node[VT]]:

        return self.\_traverse\_inorder(self.\_root, [])

    def \_traverse\_postorder(self, root: Node[VT], nodes\_list: List[Node[VT]]):

        if root is not None:

            self.\_traverse\_postorder(root.left, nodes\_list)

            self.\_traverse\_postorder(root.right, nodes\_list)

            nodes\_list.append(root)

        return nodes\_list

    def traverse\_postorder(self) -> List[Node[VT]]:

        return self.\_traverse\_postorder(self.\_root, [])

    def \_traverse\_preorder(self, root: Node[VT], nodes\_list: List[Node[VT]]):

        if root is not None:

            nodes\_list.append(root)

            self.\_traverse\_preorder(root.left, nodes\_list)

            self.\_traverse\_preorder(root.left, nodes\_list)

        return nodes\_list

    def traverse\_preorder(self) -> List[Node[VT]]:

        return self.\_traverse\_preorder(self.\_root, [])

if \_\_name\_\_ == '\_\_main\_\_':

    from random import randint

    tree\_a = AVL\_BST[int]()

    tree\_b = AVL\_BST[int]()

    tree\_c = AVL\_BST[int]()

    for \_ in range(20):

        value = randint(-99, 99)

        tree\_a.insert(value)

        if not randint(1, 29) % 4:

            tree\_b.insert(value)

    print(f"Дерево А в обратном порядке: {tree\_a.traverse\_postorder()}\nВысота: {tree\_a.height()}")

    print(f"Дерево B в симметричном порядке: {tree\_b.traverse\_inorder()}\nВысота: {tree\_b.height()}")

    for elem in tree\_a.traverse\_postorder():

        if not tree\_b.find(elem.value):

            tree\_c.insert(elem.value)

    print(f"Дерево С созданное из дерева А исключая элементы дерева B: {tree\_c.traverse\_inorder()}\nВысота: {tree\_c.height()}")

**Скриншот работы программы.**

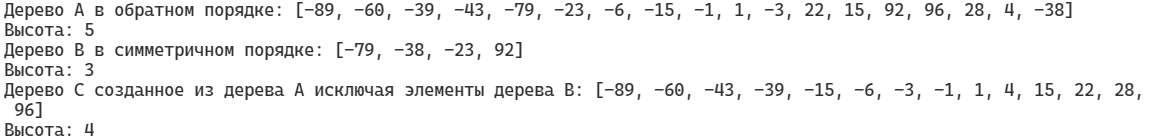
****

Рисунок 4. Вывод обхода деревьев

**Выводы.**

В результате выполнения данной работы была реализована структура АВЛ-дерева с операциями вставки, удаления и обхода дерева в прямом, симметричном и обратных порядках.

**Литература.**

1. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989.
2. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР. — 1962.