|  |
| --- |
| 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 №2**

**«Деревья двоичного поиска»**

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

**Вариант № 1**

Выполнил: студент 2 курса

группы

шифр

*(фио студента)*

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

Москва 2020 г.

**Задание на лабораторную работу № 2.**

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. (1 балл из 3)

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 (1 балл из 3) 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. Защита оформленной работы (1 балл из 3)

**Вариант № 1.**

**Тип дерева: Дерево двоичного поиска; A – обратный, B - симметричный**

**Реализация дерева: Указатель (курсор) на родителя**

**Операция: C = A Uпр B**

**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.

**Листинг программы.**

import graphviz

class Node:

    def \_\_init\_\_(self, item):

        self.key = item

        self.left = self.right = None

        self.parent = None

class BinaryTree:

    def \_\_init\_\_(self):

        self.root = None

        self.have\_root = False

        self.vis\_buff = []

    def visualize(self,name):

        dot = graphviz.Digraph(comment='Binary Tree')

        self.vis\_buff\_filling(self.root)

        dot.edges(self.vis\_buff)

        dot.render(f'output/{name}.gv', view=True)

    def vis\_buff\_filling(self,root)-> None:

        if not root is None:

            try:

                self.vis\_buff.append([str(root.parent.key),str(root.key)])

            except:pass

            self.vis\_buff\_filling(root.left)

            self.vis\_buff\_filling(root.right)

    def direct\_traversal(self,root)-> None:

        if not root is None:

            print("(",root.key, ") ", end = "")

            self.direct\_traversal(root.left)

            self.direct\_traversal(root.right)

    def symmetric\_traversal(self,root)-> None:

        if not root is None:

            self.symmetric\_traversal(root.left)

            print("(",root.key, ") ", end = "")

            self.symmetric\_traversal(root.right)

    def reverse\_traversal(self,root)-> None:

        if not root is None:

            self.reverse\_traversal(root.left)

            self.reverse\_traversal(root.right)

            print("(",root.key, ") ", end = "")

    def direct\_tree\_insertion(self,root\_from)-> None:

        if not root\_from is None:

            self.insert(self.root, root\_from.key)

            self.direct\_tree\_insertion(root\_from.left)

            self.direct\_tree\_insertion(root\_from.right)

    def symmetric\_tree\_insertion(self,root\_from)-> None:

        if not root\_from is None:

            self.symmetric\_tree\_insertion(root\_from.left)

            self.insert(self.root, root\_from.key)

            self.symmetric\_tree\_insertion(root\_from.right)

    def reverse\_tree\_insertion(self,root\_from)-> None:

        if not root\_from is None:

            self.reverse\_tree\_insertion(root\_from.left)

            self.reverse\_tree\_insertion(root\_from.right)

            self.insert(self.root, root\_from.key)

    def insert(self,node, key):

        if self.have\_root == False:

            self.have\_root = True

            self.root = self.insert(self.root, key)

        if node is None:

            return Node(key)

        if key < node.key:

            left\_son = self.insert(node.left, key)

            node.left = left\_son

            left\_son.parent = node

        elif key > node.key:

            right\_brother = self.insert(node.right, key)

            node.right = right\_brother

            right\_brother.parent = node

        return node

tree\_A = BinaryTree()

tree\_A.insert(tree\_A.root, 70)

tree\_A.insert(tree\_A.root, 50)

tree\_A.insert(tree\_A.root, 40)

tree\_A.insert(tree\_A.root, 60)

tree\_A.insert(tree\_A.root, 90)

tree\_A.insert(tree\_A.root, 80)

tree\_A.insert(tree\_A.root, 100)

tree\_B = BinaryTree()

tree\_B.insert(tree\_B.root, 75)

tree\_B.insert(tree\_B.root, 55)

tree\_B.insert(tree\_B.root, 45)

tree\_B.insert(tree\_B.root, 65)

tree\_B.insert(tree\_B.root, 35)

tree\_B.insert(tree\_B.root, 95)

tree\_B.insert(tree\_B.root, 85)

tree\_B.insert(tree\_B.root, 105)

tree\_C=BinaryTree()

print("\nСимметричный обход:")

tree\_A.symmetric\_traversal(tree\_A.root)

print("\nПрямой обход :")

tree\_A.direct\_traversal(tree\_A.root)

print("\nОбратный обход: ")

tree\_A.reverse\_traversal(tree\_A.root)

print("\nДобавление элементов одного дерева в другое в прямом обходе: ")

tree\_C.direct\_tree\_insertion(tree\_A.root)

tree\_C.direct\_tree\_insertion(tree\_B.root)

tree\_C.direct\_traversal(tree\_C.root)

print("\nВизуализация дерева: ")

tree\_A.visualize("A\_binary\_tree")

tree\_B.visualize("B\_binary\_tree")

tree\_C.visualize("C\_binary\_tree")

**Скриншот работы программы.**

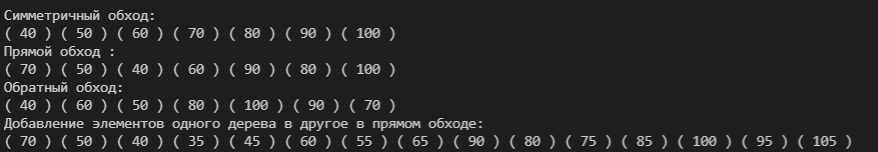


Рисунок 4. – вывод обходов.

**Выводы.**

В результате выполнения данной работы был реализован алгоритм, способный строить деревья и выводить их в симметричном, прямом и обратном порядках, а также добавлять элементы одного дерева в другое в одном из трех вышеперечисленных порядках.

**Литература:**

1. Алгоритмы: построение и анализ. Кормен Т.Х., Лейзерсон Ч.И.

2. https://en.wikipedia.org/wiki/Binary\_tree