|  |
| --- |
| 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 № 30**

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 № 30**

**Тип дерева: Рандомизированное дерево двоичного поиска**

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

**Операция: А = 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.

**Листинг программы.**

from typing import Generic, TypeVar, Optional, List

import random

VT = TypeVar("VT")

class Node(Generic[VT]):

    size: int

    value: VT

    left: Optional['Node[VT]']

    right: Optional['Node[VT]']

    def \_\_init\_\_(self, key: VT):

        self.value = key

        self.size = 1

        self.left = None

        self.right = None

    def \_\_repr\_\_(self):

        return str(self.value)

class RandomizedBST(Generic[VT]):

    \_root: Optional[Node[VT]]

    def \_\_init\_\_(self):

        self.\_root = None

    def empty(self) -> bool:

        return self.root is 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)

        elif value > root.value:

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

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

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

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

        return root.size if root is not None else 0

    def \_fix\_size(self, root):

        root.size = 1 + self.\_size(root.left) + self.\_size(root.right)

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

        b = d.left

        d.left = b.right

        b.right = d

        self.\_fix\_size(d)

        self.\_fix\_size(b)

        return b

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

        d = b.right

        b.right = d.left

        d.left = b

        self.\_fix\_size(b)

        self.\_fix\_size(d)

        return d

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

        if root is None:

            return Node(value)

        elif value < root.value:

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

            return self.\_rotate\_right(root)

        elif value > root.value:

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

            return self.\_rotate\_left(root)

        else:

            return root

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

        if root is None:

            return Node(value)

        elif random.randint(0, self.\_size(root) + 1) == 0:

            return self.\_root\_insert(root, value)

        elif value < root.value:

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

            self.\_fix\_size(root)

            return root

        elif value > root.value:

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

            self.\_fix\_size(root)

            return root

        else:

            return root

    def insert(self, value: VT):

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

    def \_join(self, tree\_min: Node[VT], tree\_max: Node[VT]) -> Node[VT]:

        if tree\_min is None:

            return tree\_max

        elif tree\_max is None:

            return tree\_min

        elif random.randint(0, self.\_size(tree\_min) + self.\_size(tree\_max)) < self.\_size(tree\_min):

            tree\_min.right = self.\_join(tree\_min.right, tree\_max)

            self.\_fix\_size(tree\_min)

            return tree\_min

        else:

            tree\_max.left = self.\_join(tree\_min, tree\_max.left)

            self.\_fix\_size(tree\_max)

            return tree\_max

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

        if root is None:

            return root

        elif value < root.value:

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

            self.\_fix\_size(root)

            return root

        elif value > root.value:

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

            self.\_fix\_size(root)

            return root

        else:

            return self.\_join(root.left, root.right)

    def remove(self, value: VT):

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

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

        if root is None:

            return 0

        else:

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

    def height(self):

        return self.\_height(self.\_root)

    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.right, 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 = RandomizedBST[int]()

    tree\_b = RandomizedBST[int]()

    for \_ in range(20):

        value = randint(-99, 99)

        tree\_a.insert(value)

        if randint(7, 53) % 6 >= 4:

            tree\_b.insert(value)

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

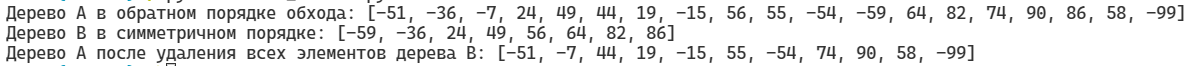
    print(f"Дерево B в симметричном порядке: {tree\_b.traverse\_inorder()}")

    for node in tree\_b.traverse\_inorder():

        tree\_a.remove(node.value)

    print(f"Дерево А после удаления всех элементов дерева В: {tree\_a.traverse\_postorder()}")

**Скриншот работы программы.**

Рисунок 4. Вывод обхода деревьев

**Выводы.**

В результате выполнения данной работы была реализована структура данных – рандомизированное двоичное дерево поиска, а также алгоритм, способный строить и выводить их в симметричном, прямом и обратном порядках, а также удалять элементы одного дерева из другого в трех вышеперечисленных порядках.

**Литература.**

* Martinez, Conrado; Roura, Salvador (1997), "Randomized binary search trees", Journal of the ACM 45
* Seidel, Raimund; Aragon, Cecilia R. «Randomized Search Trees», 1996 г.