|  |
| --- |
| 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 № 1**

Выполнил: студент 2 курса

группы **\_\_\_\_\_\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_\_\_\_\_

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

*(фио студента)*

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

Москва 2022 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 № 1**

**Тип дерева: Дерево двоичного поиска**

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

**Операция: А=A ⋃пр 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, List, Optional, TypeVar  
  
VT = TypeVar("VT")  
  
  
class Node(Generic[VT]):  
 left: Optional["Node[VT]"]  
 right: Optional["Node[VT]"]  
 parent: Optional["Node[VT]"]  
 value: VT  
  
 def \_\_init\_\_(self, value: VT):  
 self.value = value  
 self.left = None  
 self.right = None  
 self.parent = None  
  
 def \_\_repr\_\_(self):  
 return str(self.value)  
  
  
class BST(Generic[VT]):  
 \_root: Optional[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) -> Optional[Node[VT]]:  
 return self.\_find(self.\_root, value)  
  
 def \_insert(self, root: Node[VT], value: VT) -> Optional[Node[VT]]:  
 if root is None:  
 return Node(value)  
  
 if value root.value:  
 root.left = self.\_insert(root.left, value)  
 root.left.parent = root  
  
 elif value > root.value:  
 root.right = self.\_insert(root.right, value)  
 root.right.parent = root  
  
 return root  
  
 def insert(self, value: VT):  
 self.\_root = self.\_insert(self.\_root, value)  
  
 def \_find\_min(self, root: Node[VT]) -> Node[VT]:  
 if root.left is not None:  
 return self.\_find\_min(root.left)  
 else:  
 return root  
  
 def \_remove(self, root: Node[VT], value: VT) -> Optional[Node[VT]]:  
 if root is None:  
 return None  
  
 if value root.value:  
 root.left = self.\_remove(root.left, value)  
 if root.left:  
 root.left.parent = root  
  
 return root  
 elif value > root.value:  
 root.right = self.\_remove(root.right, value)  
 if root.right:  
 root.right.parent = root  
  
 return root  
  
 if root.left is None:  
 return root.right  
 elif root.right is None:  
 return root.left  
 else:  
 min\_value = self.\_find\_min(root.right).value  
 root.value = min\_value  
 root.right = self.\_remove(root.right, min\_value)  
 return root  
  
 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\_inorder(root.left, nodes\_List)  
 self.\_traverse\_inorder(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\_inorder(root.left, nodes\_List)  
 self.\_traverse\_inorder(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 = BST[int]()  
 tree\_b = BST[int]()  
  
 for \_ in range(20):  
 value = randint(-99, 99)  
  
 if randint(0, 10) % 3 == 1:  
 tree\_a.insert(value)  
 else:  
 tree\_b.insert(value)  
  
 print("Дерево А в обратном порядке:\n" + str(tree\_a.traverse\_postorder()))  
 print("Дерево В в симметричном порядке:\n" + str(tree\_b.traverse\_inorder()))  
  
 for node in tree\_b.traverse\_preorder():  
 tree\_a.insert(node.value)  
  
 print(  
 "Дерево А после добавления элементов дерева В в прямом порядке:\n"  
 + str(tree\_a.traverse\_inorder())  
 )

**Скриншот работы программы.**

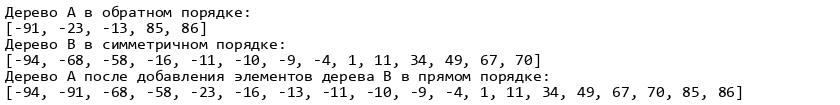


Рисунок 4. Вывод обхода деревьев

**Выводы.**

В результате выполнения данной работы была реализована структура двоичного дерева поиcка 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.