Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №4

по курсу «Логика и основы алгоритмизации

в инженерных задачах»

на тему «Бинарное дерево поиска»

Выполнил: ст. гр. 24ВВВ1

Цыбузин Д.В.

Тусков А.А

Принял:

к.т.н, доцент Юрова О. В.

к.т.н. Деев М.В.

Пенза 2025

**Цель работы:**

Реализовать бинарное дерево поиска.

**Лабораторное задание:**

1. Реализовать алгоритм поиска вводимого с клавиатуры значения в уже созданном дереве.
2. Реализовать функцию подсчёта числа вхождений заданного элемента в дерево.
3. \*Изменить функцию добавления элементов для исключения добавления одинаковых символов.
4. \*Оценить сложность процедуры поиска по значению в бинарном дереве.

**Ход работы:**

1. Сlass Treenode – это класс для узла бинарного дерева

2.def insert(root, value)– функция вставки элемента в дерево

3.def in\_order\_traversal(root, result) — функция обхода дерева от самого левого элемента к самому правом

4.def search() - функция для поиска определённого элемента в дереве

5.def count\_occurences(root, value) – подсчёт числа вхождений элемента в дерево

6.def generate\_random\_numbers(count, min\_val, max\_val) – функция для генерации count случайных чисел в пределах от min\_val до max\_val

7.def manual\_input() - функция для ручного ввода чисел

8.def fill\_data() - функция для выбора заполнения дерева случайными числами или вручную

9.print\_tree(root, level, prefix) – функция для вывода дерева в терминал

**Вывод:** в ходе выполнения лабораторной работы по реализации бинарного дерева поиска были успешно решены поставленные задачи и получены практические навыки работы с древовидными структурами данных. Была разработана программа, реализующая основные операции с бинарным деревом поиска, включая добавление элементов, поиск значений и подсчет вхождений. Основным достижением работы стало создание эффективного алгоритма поиска в бинарном дереве, который продемонстрировал логарифмическую сложность *O(log n)* в среднем случае для сбалансированного дерева. Однако было выявлено, что в худшем случае, при вырождении дерева в линейный список, сложность поиска возрастает до *O(n)*, что подчеркивает важность поддержания сбалансированности структуры.

**Листинг программы:**

**Файл *lab4.py***

import random

from typing import List, Optional, Tuple

class TreeNode:

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

        self.value = value

        self.left: Optional[TreeNode] = None

        self.right: Optional[TreeNode] = None

def insert(root: Optional[TreeNode], value: int) -> TreeNode:

    if root is None:

        return TreeNode(value)

    if value < root.value:

        root.left = insert(root.left, value)

    else:  # Дубликаты идут в правое поддерево

        root.right = insert(root.right, value)

    return root

def in\_order\_traversal(root: Optional[TreeNode], result: List[int]) -> None:

    if root is not None:

        in\_order\_traversal(root.left, result)

        result.append(root.value)

        in\_order\_traversal(root.right, result)

def search\_all\_occurrences(root: Optional[TreeNode], value: int, level: int = 0) -> List[Tuple[bool, int]]:

    if root is None:

        return []

    results = []

    # Если нашли совпадение - добавляем в результаты

    if root.value == value:

        results.append((True, level))

    # Поиск в левом поддереве (только если значение МЕНЬШЕ или РАВНО)

    # Для дубликатов тоже ищем слева, если это нужно

    if value <= root.value:

        results.extend(search\_all\_occurrences(root.left, value, level + 1))

    # Поиск в правом поддереве (только если значение БОЛЬШЕ или РАВНО)

    if value >= root.value:

        results.extend(search\_all\_occurrences(root.right, value, level + 1))

    return results

def search\_first\_occurrence(root: Optional[TreeNode], value: int, level: int = 0) -> Tuple[bool, int]:

    if root is None:

        return False, -1

    if root.value == value:

        return True, level

    # Если значение меньше - ищем только слева

    if value < root.value:

        return search\_first\_occurrence(root.left, value, level + 1)

    # Если значение больше или равно - ищем только справа

    return search\_first\_occurrence(root.right, value, level + 1)

def count\_occurrences(root: Optional[TreeNode], value: int) -> int:

    if root is None:

        return 0

    count = 0

    if root.value == value:

        count += 1

    count += count\_occurrences(root.left, value)

    count += count\_occurrences(root.right, value)

    return count

def generate\_random\_numbers(count: int, min\_val: int, max\_val: int) -> List[int]:

    return [random.randint(min\_val, max\_val) for \_ in range(count)]

def manual\_input() -> List[int]:

    while True:

        try:

            print("Введите числа через пробел:")

            numbers = input().split()

            numbers = [int(num) for num in numbers]

            return numbers

        except ValueError:

            print("Ошибка: введите только целые числа!")

def get\_fill\_method() -> str:

    print("Выберите способ заполнения:")

    print("1 - Вручную")

    print("2 - Случайными числами")

    while True:

        choice = input("Ваш выбор (1/2): ")

        if choice in ['1', '2']:

            return choice

        print("Неверный ввод! Попробуйте снова.")

def fill\_data() -> List[int]:

    choice = get\_fill\_method()

    if choice == '1':

        return manual\_input()

    else:

        while True:

            try:

                count = int(input("Введите количество чисел в массиве: "))

                min\_val = int(input("Введите нижнюю границу диапазона: "))

                max\_val = int(input("Введите верхнюю границу диапазона: "))

                return generate\_random\_numbers(count, min\_val, max\_val)

            except ValueError:

                print("Ошибка: введите целые числа!")

def print\_tree(root: Optional[TreeNode], level: int = 0, prefix: str = ""):

    if root is not None:

        # Печатаем правое поддерево

        print\_tree(root.right, level + 1, "    " + "┌─── ")

        # Печатаем текущий узел

        print("    " \* level + prefix + str(root.value))

        # Печатаем левое поддерево

        print\_tree(root.left, level + 1, "    " + "└─── ")

def main():

    print("\n--- Создание бинарного дерева поиска ---\n")

    # Выбор способа заполнения

    data = fill\_data()

    print(f"\nИсходные данные: {data}\n")

    # Создание дерева

    root = None

    print("--- Процесс создания дерева ---")

    for num in data:

        root = insert(root, num)

        print(f"Добавлено значение: {num}")

    # Визуализация дерева

    print("\n--- Визуализация дерева ---")

    print\_tree(root)

    # Демонстрация поиска

    while True:

        try:

            search\_value = int(input("\nВведите значение для поиска (или нечисло для выхода): "))

            # Поиск всех вхождений

            all\_occurrences = search\_all\_occurrences(root, search\_value)

            count = count\_occurrences(root, search\_value)

            if all\_occurrences:

                print(f"Значение {search\_value} найдено {count} раз(а)!")

                print("Уровни нахождения элементов:")

                for i, (found, level) in enumerate(all\_occurrences, 1):

                    print(f"  Вхождение {i}: уровень {level}")

                # Первое вхождение

                first\_found, first\_level = search\_first\_occurrence(root, search\_value)

                print(f"Первое вхождение: уровень {first\_level}")

            else:

                print(f"Значение {search\_value} не найдено")

        except ValueError:

            break

    print("\nПрограмма завершена")

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

    main()