САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №2
по курсу «Алгоритмы и структуры данных»
Тема: Двоичные деревья поиска
Вариант 16

Выполнила:
Бархатова Н.А.
К3139

Проверила:
Артамонова В.Е.

Санкт-Петербург
2023 г.

Содержание отчета

Оглавление

Задачи по варианту

Задача №1. Обход двоичного дерева [1 балл]

Текст задачи:

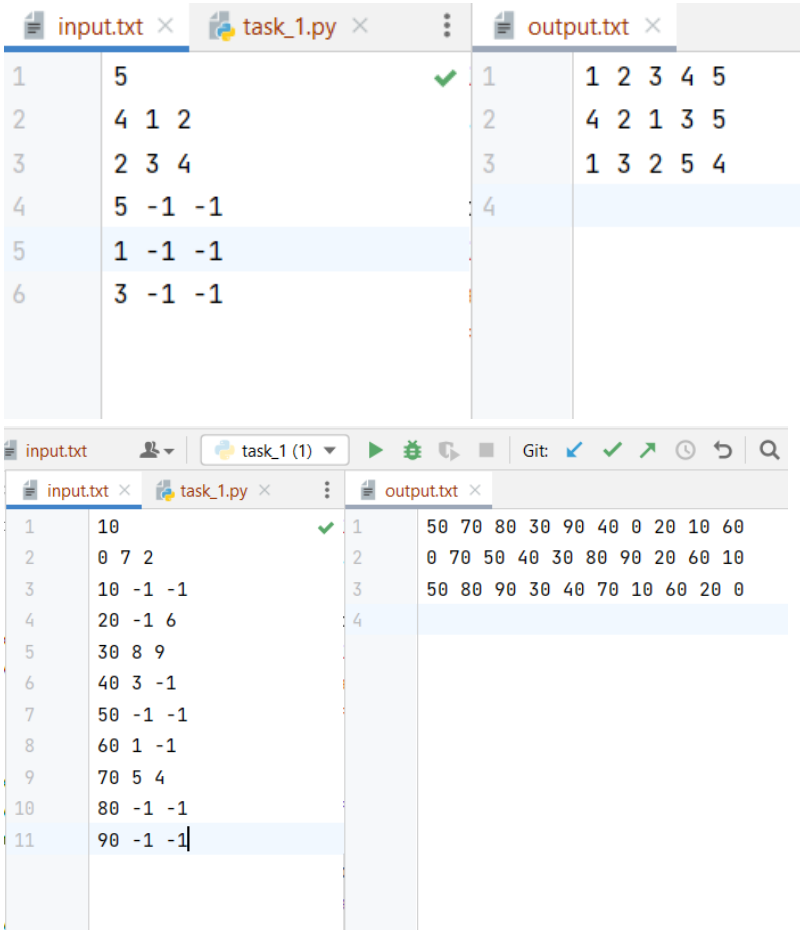
В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (inorder), прямой (pre-order) и обратный (post-order). Очень полезно попрактиковаться в их реализации, чтобы лучше понять бинарные деревья поиска. Вам дано корневое двоичное дерево. Выведите центрированный (in-order), прямой (pre-order) и обратный (postorder) обходы в глубину

Листинг кода:

```
import time
import tracemalloc


class TreeNode:
    def __init__(self, data=None):
        self.left = None
        self.right = None
        self.data = data

    @staticmethod
    def insert_left(data, parent):
        if parent.left is None:
            parent.left = TreeNode(data)

    @staticmethod
    def insert_right(data, parent):
        if parent.right is None:
            parent.right = TreeNode(data)

    def find(self, data):
        answer = None
        if self.data == data:
            answer = self
        else:
            if self.left is not None:
                answer = self.left.find(data)
            if answer is None and self.right is not None:
                answer = self.right.find(data)
        return answer

    def preorder(self, list_of_values):
        if self.data is not None:
            list_of_values.append(self.data)
        if self.left is not None:
            self.left.preorder(list_of_values)
        if self.right is not None:
            self.right.preorder(list_of_values)
        return list_of_values
```

```
    def inorder(self, list_of_values):
        if self.left is not None:
            self.left.inorder(list_of_values)
        if self.data is not None:
            list_of_values.append(self.data)
        if self.right is not None:
            self.right.inorder(list_of_values)
        return list_of_values

    def postorder(self, list_of_values):
        if self.left is not None:
            self.left.postorder(list_of_values)
        if self.right is not None:
            self.right.postorder(list_of_values)
        if self.data is not None:
            list_of_values.append(self.data)
        return list_of_values


start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = []
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
tree = TreeNode(info[0][0])
left = 1
right = 2


def create_tree(node_index):
    parent = tree.find(info[node_index][0])
    if parent:
        left_index = info[node_index][left]
        right_index = info[node_index][right]
        if left_index >= 0:
            tree.insert_left(info[left_index][0], parent)
            create_tree(left_index)
        else:
            tree.insert_left(None, parent)
        if right_index >= 0:
            tree.insert_right(info[right_index][0], parent)
            create_tree(right_index)
        else:
            tree.insert_right(None, parent)


create_tree(0)
with open('output.txt', 'w') as output_file:
    print(*tree.inorder([]), file=output_file)
    print(*tree.preorder([]), file=output_file)
    print(*tree.postorder([]), file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Напишем класс TreeNode. У него есть несколько методов: статические методы insert_left и insert_right для добавления нового узла, find для поиска узла по значению и методы preorder, postorder, inorder для разных видов обхода дерева. Далее считываем данные из файла в массив info. Зададим корень дерева. С помощью рекурсивной функции create_tree заполним дерево данными из массива. Сделаем 3 обхода.



	Время выполнения	Затраты памяти
Пример из задачи	0.0013	0.0091
Пример из задачи	0.0016	0.0135

Вывод по задаче: мне не понравился инпут файл

Задача №11. Сбалансированное двоичное дерево поиска [2 балла]

Текст задачи:
Реализуйте сбалансированное двоичное дерево поиска.

Входной файл содержит описание операций с деревом, их количество N не превышает 105. В каждой строке находится одна из следующих операций:
– insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо;
– delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо;
– exists x – если ключ x есть в дереве выведите .true., если нет – .false.;
– next x – выведите минимальный элемент в дереве, строго больший x, или .none., если такого нет;
– prev x – выведите максимальный элемент в дереве, строго меньший x, или .none., если такого нет.
В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.
Листинг кода:

```
import time
import tracemalloc


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def preorder(self, list_of_values, node):
        if node.data is not None:
            list_of_values.append(node.data)
        if node.left is not None:
            self.preorder(list_of_values, node.left)
        if node.right is not None:
            self.preorder(list_of_values, node.right)
        return list_of_values

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def exists(self, x):
        return bool(self.find(x, self.root))

    def insert(self, data):
        found_parent = None
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
```

```
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent
        if not self.is_balanced(self.root, True)[1]:
            self.balancing(self.root)

    def tree_min(self, node):
        while node.left is not None:
            node = node.left
        return node

    def find_closest_greater_number(self, x):
        if self.root is None:
            return 0
        full_way = []
        c_node = self.root
        while True:
            full_way.append(c_node)
            if x > c_node.data:
                if c_node.right is None:
                    break
                c_node = c_node.right
            elif x < c_node.data:
                if c_node.left is None:
                    return c_node.data
                c_node = c_node.left
            else:
                if c_node.right is None:
                    break
                c_node = c_node.right
                while c_node.left is not None:
                    c_node = c_node.left
                return c_node.data
        for i in range(len(full_way) - 1, -1, -1):
            if full_way[i].data > x:
                return full_way[i].data
        return "none"

    def find_closest_less_number(self, x):
        if self.root is None:
            return 0
        full_way = []
        c_node = self.root
        while True:
            full_way.append(c_node)
            if x < c_node.data:
                if c_node.left is None:
                    break
```

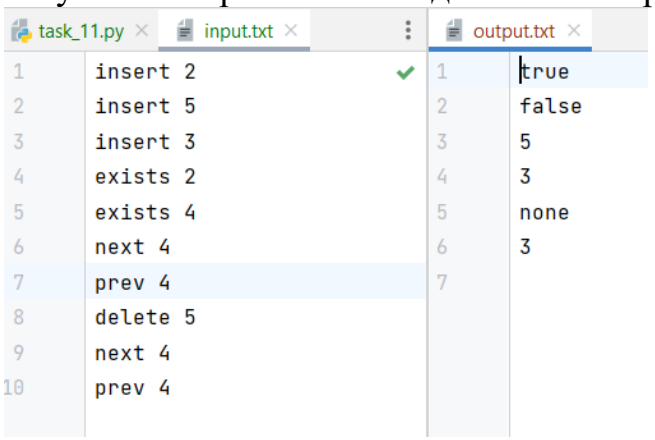
```
                c_node = c_node.left
            elif x > c_node.data:
                if c_node.right is None:
                    return c_node.data
                c_node = c_node.right
            else:
                if c_node.left is None:
                    break
                c_node = c_node.left
                while c_node.right is not None:
                    c_node = c_node.right
                return c_node.data
        for i in range(len(full_way) - 1, -1, -1):
            if full_way[i].data < x:
                return full_way[i].data
        return "none"

    def __del_leaf(self, node):
        if node.parent.left == node:
            node.parent.left = None
        elif node.parent.right == node:
            node.parent.right = None

    def __del_one_child(self, node):
        if node.parent.left == node:
            if node.left is None:
                node.parent.left = node.right
            elif node.right is None:
                node.parent.left = node.left
        elif node.parent.right == node:
            if node.left is None:
                node.parent.right = node.right
            elif node.right is None:
                node.parent.right = node.left

    def __del_two_children(self, node):
        new = self.tree_min(node.right)
        node.data = new.data
        self.__del_one_child(new)

    def delete(self, x):
        node = self.find(x, self.root)
        if node:
            if node.right is None and node.left is None:
                self.__del_leaf(node)
            elif node.left is None or node.right is None:
                self.__del_one_child(node)
            else:
                self.__del_two_children(node)
        if not self.is_balanced(self.root, True)[1]:
            self.balancing(self.root)

    def is_balanced(self, node, isBalanced=True):
        if node is None or not isBalanced:
            return 0, isBalanced
        left_height, isBalanced = self.is_balanced(node.left, isBalanced)
        right_height, isBalanced = self.is_balanced(node.right, isBalanced)
        if abs(left_height - right_height) > 1:
            isBalanced = False
        return max(left_height, right_height) + 1, isBalanced
```

```
    def rotation_left(self, node):
        if node is not None:
            new_node = node.right
            node.right = new_node.left
            if new_node.left:
                new_node.left.parent = node
            new_node.left = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    def rotation_right(self, node):
        if node is not None:
            new_node = node.left
            node.left = new_node.right
            if new_node.right:
                new_node.right.parent = node
            new_node.right = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    def balancing(self, node):
        if node is None:
            return
        h1 = tree.is_balanced(node.left, True)[0]
        h2 = tree.is_balanced(node.right, True)[0]
        if h1 - h2 > 1:
            if tree.is_balanced(node.left.left, True)[0] >
tree.is_balanced(node.left.right, True)[0]:
                tree.rotation_right(node)
            else:
                tree.rotation_left(node.left)
                tree.rotation_right(node)
        elif h2 - h1 > 1:
            if tree.is_balanced(node.right.right, True)[0] >
tree.is_balanced(node.right.left, True)[0]:
                tree.rotation_left(node)
            else:
```

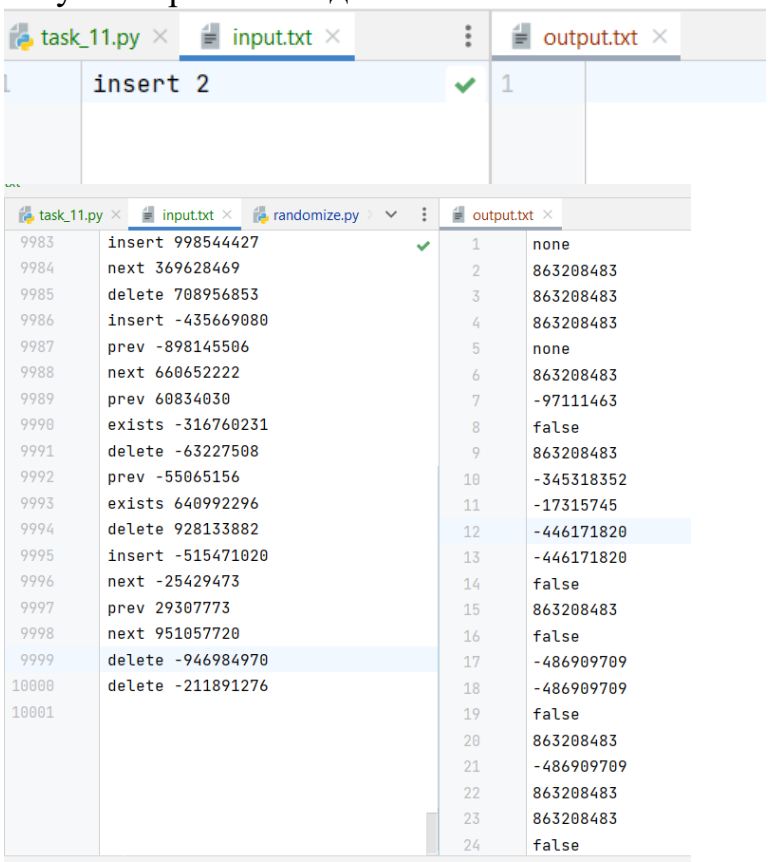
```
                tree.rotation_right(node.right)
                tree.rotation_left(node)
        else:
            self.balancing(node.right)
            self.balancing(node.left)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
commands = input_file.readlines()
with open('output.txt', 'w') as output_file:
    for command in commands:
        action, value = command.split()
        value = int(value)
        match action:
            case "insert":
                tree.insert(value)
            case "exists":
                if tree.exists(value):
                    print("true", file=output_file)
                else:
                    print("false", file=output_file)
            case "next":
                print(tree.find_closest_greater_number(value),
file=output_file)
            case "prev":
                print(tree.find_closest_less_number(value),
file=output_file)
            case "delete":
                tree.delete(value)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Функция is_balanced возвращает высоту узла и булевое значение (сбалансированно или нет). Функции rotation_left и rotation_right отвечают за простые повороты дерева. Например, левый простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: $h(s) \leq h(D)$. Большой левый поворот применяется при условии $h(s) > h(D)$ и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p. Аналогично с правым большим поворотом. После каждой вставке или удалении элемента проверяем дерево на сбалансированность. Если она отсутствует, то применяем функцию балансировки.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0006	0.0037
Пример из задачи	0.0020	0.0068
Пример из задачи		

Верхняя граница диапазона значений входных данных из текста задачи	23.4184	1.0613

Вывод по задаче: долго выполняется при большом количестве значений

Задача №16. K-й максимум [3 балла]

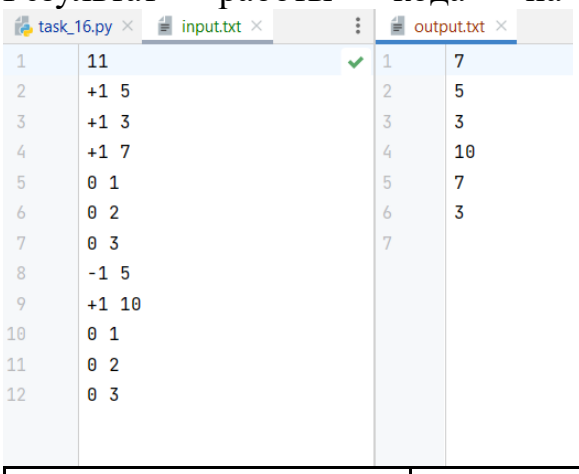
Текст задачи:
Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также
находить k-й максимум.
• Формат ввода / входного файла (input.txt). Первая строка входного файла содержит натуральное число n –
количество команд. Последующие n строк содержат по одной команде каждая. Команда записывается в виде
двух чисел ci и ki – тип и аргумент команды соответственно.
Поддерживаемые команды:
– +1 (или просто 1): Добавить элемент с ключом ki.
– 0 : Найти и вывести ki-й максимум.
– -1 : Удалить элемент с ключом ki.
Листинг кода:

```
import time
import tracemalloc


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def exists(self, x):
        return bool(self.find(x, self.root))

    def insert(self, data):
        found_parent = None
```

```
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def tree_min(self, node):
        while node.left is not None:
            node = node.left
        return node

    def inorder(self, node, list_of_values):
        if node.left is not None:
            self.inorder(node.left, list_of_values)
        if node.data is not None:
            list_of_values.append(node.data)
        if node.right is not None:
            self.inorder(node.right, list_of_values)
        return list_of_values

    def __del_leaf(self, node):
        if node.parent.left == node:
            node.parent.left = None
        elif node.parent.right == node:
            node.parent.right = None

    def __del_one_child(self, node):
        if node.parent.left == node:
            if node.left is None:
                node.parent.left = node.right
            elif node.right is None:
                node.parent.left = node.left
        elif node.parent.right == node:
            if node.left is None:
                node.parent.right = node.right
            elif node.right is None:
                node.parent.right = node.left

    def __del_two_children(self, node):
        new = self.tree_min(node.right)
        node.data = new.data
        self.__del_one_child(new)

    def delete(self, x):
        node = self.find(x, self.root)
        if node:
            if node.right is None and node.left is None:
```

```
                self.__del_leaf(node)
            elif node.left is None or node.right is None:
                self.__del_one_child(node)
            else:
                self.__del_two_children(node)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
N = int(input_file.readline())
commands = input_file.readlines()
with open('output.txt', 'w') as output_file:
    for command in commands:
        action, value = command.split()
        value = int(value)
        match action:
            case "+1":
                tree.insert(value)
            case "0":
                all_numbers = tree.inorder(tree.root, [])
                print(all_numbers[value * (-1)], file=output_file)
            case "-1":
                tree.delete(value)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Бинарное дерево поиска с реализованной функцией inorder.
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0008	0.0055

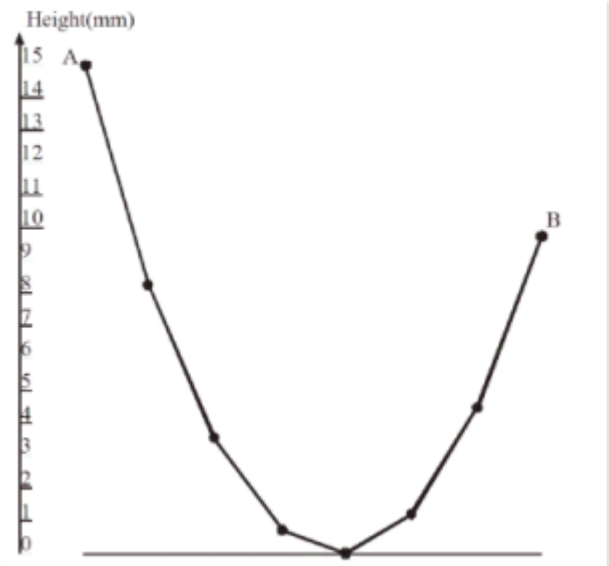
Вывод по задаче: inorder полезная функция для BST

Дополнительные задачи

Задача №2. Гирлянда [1 балл]

Текст задачи:

Гирлянда состоит из n лампочек на общем проводе. Один её конец закреплён на заданной высоте A мм (h1 = A). Благодаря силе тяжести гирлянда прогибается: высота каждой неконцевой лампы на 1 мм меньше, чем средняя высота ближайших соседей (hi = hi−1 + hi+1 2 − 1 для 1 < i < N). Требуется найти минимальное значение высоты второго конца B (B = hn), такое что для любого ϵ > 0 при высоте второго конца B + ϵ для всех лампочек выполняется условие hi > 0. Обратите внимание на то, что при данном значении высоты либо ровно одна, либо две соседних лампочки будут иметь нулевую высоту.



Листинг кода:

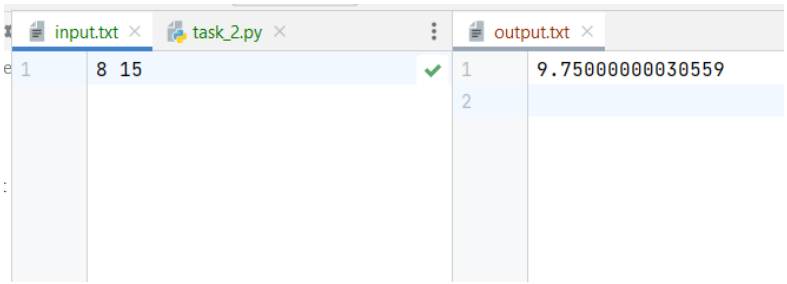
```
import time
import tracemalloc

input_file = open('input.txt')
n, A = map(float, input_file.readline().split())

accuracy = 10 ** (-10)


def correct(x, y):
    return abs(x - y) <= accuracy


def less(x, y):
    return x < y and not correct(x, y)
```

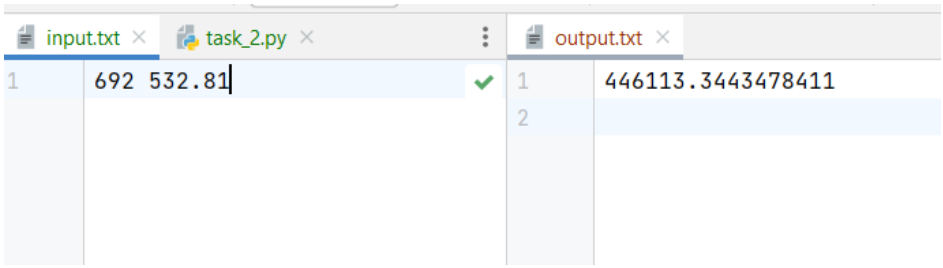
```
def greater(x, y):
    return x > y and not correct(x, y)


start_time = time.perf_counter()
tracemalloc.start()
heights = [0] * int(n)
heights[0] = A
answer = float('inf')
left = 0
right = heights[0]

while less(left, right):
    heights[1] = (left + right) / 2
    isNotUp = False
    for i in range(2, int(n)):
        heights[i] = 2 * heights[i - 1] - heights[i - 2] + 2
        if not greater(heights[i], 0):
            isNotUp = True
            break
    if greater(heights[-1], 0):
        answer = min(answer, heights[-1])
    if isNotUp:
        left = heights[1]
    else:
        right = heights[1]
with open('output.txt', 'w') as output_file:
    print(answer, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:

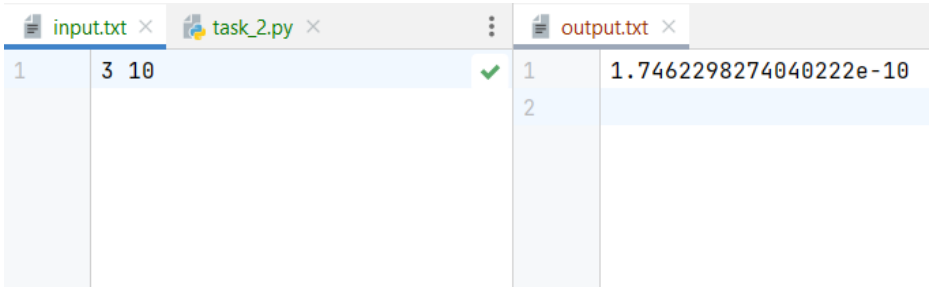
Перебираем высоту для 2-й лампочки (далее из её положения можно вычислить все остальные по формуле в дано). Сначала берем её как среднее между 0 и A, а потом, если гирлянда лежит или последняя лампочка слишком высоко, то берем значение 2 лампочки как среднее между A и предыдущим значением 2-й лампочки (то есть поднимаем её выше).

Результат работы кода на примерах из текста задачи:(скрины input output файлов)

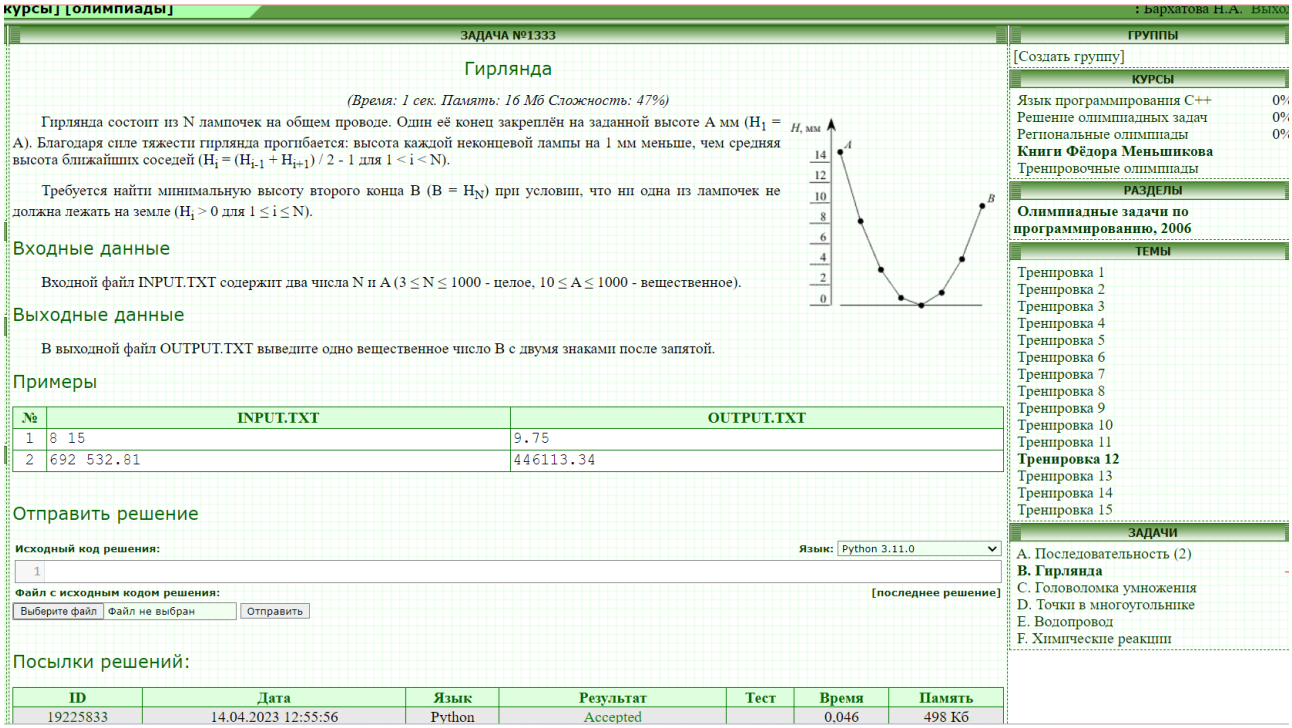




Результат работы кода на максимальных и минимальных значениях:(скрины input output файлов)



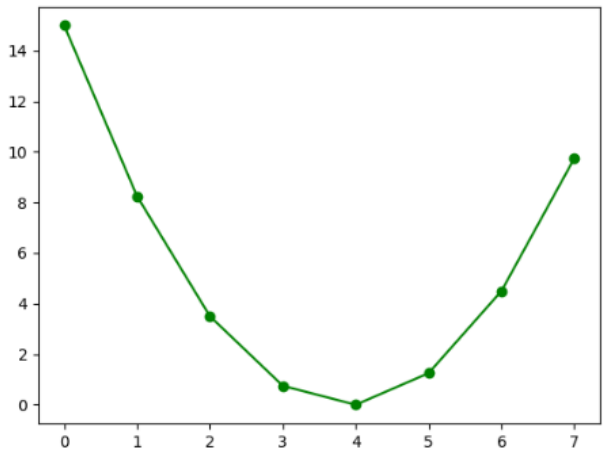
Проверка задачи на (openedu, acmp и тд при наличии в задаче). (скрин)

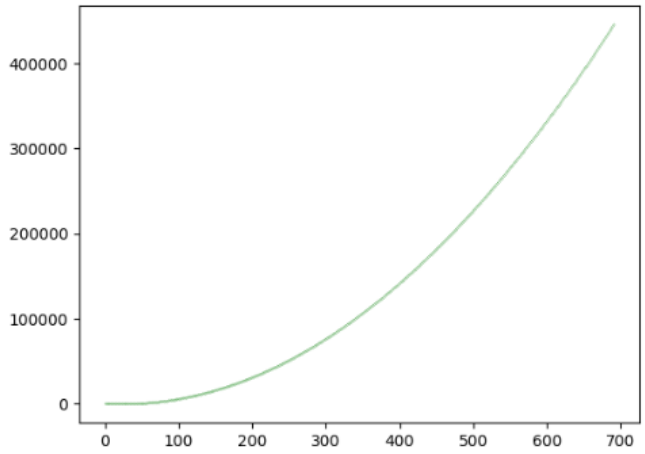


	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0016	0.0022

Пример из задачи	0.0027	0.0028
Пример из задачи	0.0418	0.0385

Вывод по задаче: мне понравился график, и я захотела сделать свои с помощью mathplotlib





Задача №3. Простейшее BST [1 балл]

Текст задачи:

В этой задаче вам нужно написать простейшее BST по явному ключу и отвечать им на запросы:

«+ x» – добавить в дерево x (если x уже есть, ничего не делать).

«> x» – вернуть минимальный элемент больше x или 0, если таких нет.

Листинг кода:

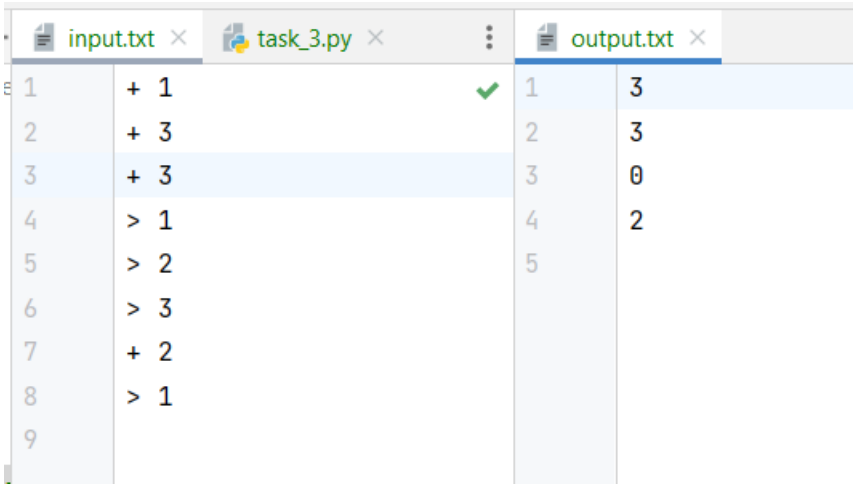
```
import time
import tracemalloc


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class Tree:
    def   init  (self):
        self.root = None

    def insert(self, data):
        parent = None
        c_node = self.root
        while c_node is not None:
            parent = c_node
```

```
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if parent is None:
            self.root = new
        elif data < parent.data:
            parent.left = new
        elif data > parent.data:
            parent.right = new

    def find_closest_greater_number(self, x):
        if self.root is None:
            return 0
        full_way = []
        c_node = self.root
        while True:
            full_way.append(c_node)
            if x > c_node.data:
                if c_node.right is None:
                    break
                c_node = c_node.right
            elif x < c_node.data:
                if c_node.left is None:
                    return c_node.data
                c_node = c_node.left
            else:
                if c_node.right is None:
                    break
                c_node = c_node.right
                while c_node.left is not None:
                    c_node = c_node.left
                return c_node.data
        for i in range(len(full_way) - 1, -1, -1):
            if full_way[i].data > x:
                return full_way[i].data
        return 0


start_time = time.perf_counter()
tracemalloc.start()

input_file = open('input.txt')
content = input_file.readlines()
tree = Tree()
for request in content:
    operation = request[0]
    data = int(request[2:-1])
    with open('output.txt', 'a') as output_file:
        if operation == "+":
            tree.insert(data)
        elif operation == ">":
            print(tree.find_closest_greater_number(data), file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

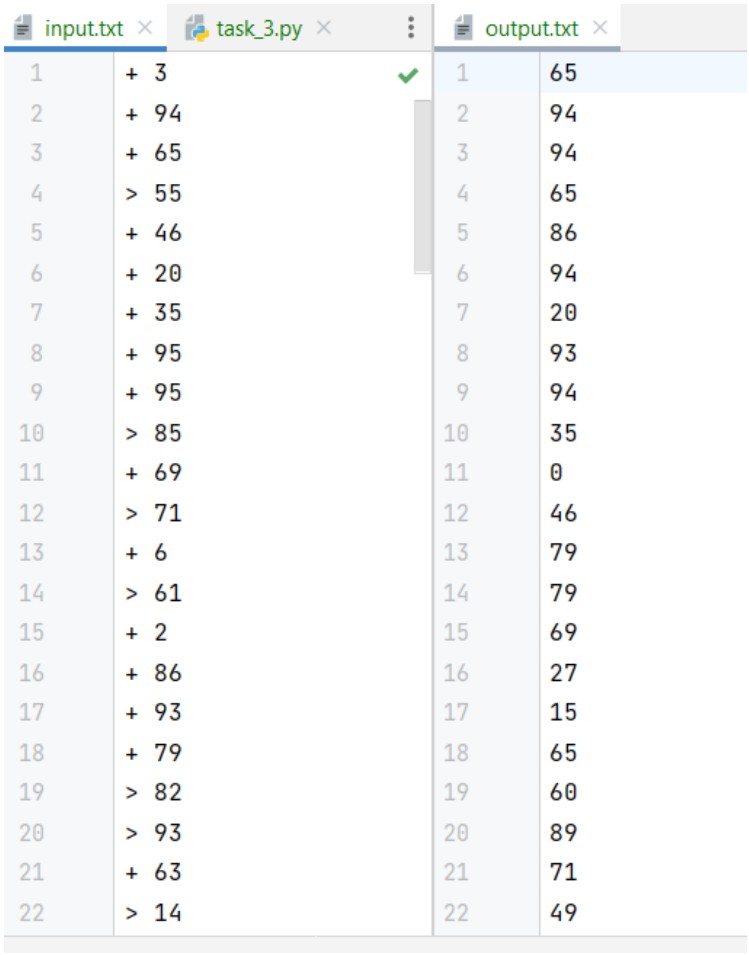
Текстовое объяснение решения:

Есть два класса: Node и Tree. Первый отвечает за узлы и включает в себя несколько параметров, в то время как второй включает в себя функции. Напишем функцию insert – вставка нового узла. Сначала определяется узел, к которому необходимо прикрепить новый узел. Если значение больше текущего, то мы идет вправо, если меньше, то влево. Затем просто вставляем значение основываясь на значении листа, который мы только что нашли. Далее создадим функцию find_closest_greater_number(). Мы должны пройтись по всем возможным вариантам ответа и затем выбрать из них наименьший. Итак, начинаем проверку с узла. Если значение больше узла, то переходим к правому ребенку, если он есть (если нет, то break). Если значение меньше узла, то переходим к левому ребенку (если его нет, то ответом является значение в корне). Если мы нашли в дереве то же значение, что нам дано, то мы идем в правого ребенка (если его нет, то break), а потом идем в самый низ по левым детям. То есть таким образом находим минимум в этой ветке. Потом через массив full_way находим ответ.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0006	0.0034
Пример из задачи	0.0026	0.0052
Верхняя граница диапазона значений входных данных из текста задачи	0.0267	0.0179

Вывод по задаче: моя любимая задача (не считая гирлянду)

Задача №4. Простейший неявный ключ [1 балл]

Текст задачи:

В этой задаче вам нужно написать BST по **неявному** ключу и отвечать им на запросы:

. .+ x. – добавить в дерево x (если x уже есть, ничего не делать).

. .? k. – вернуть k-й по возрастанию элемент.

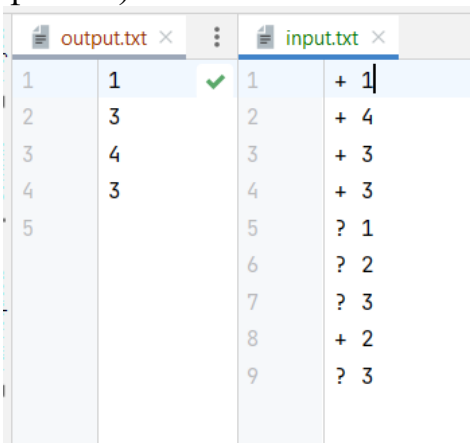
Листинг кода:

```
import time
import tracemalloc


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def insert(self, data):
        found_parent = None
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def inorder(self, node, list_of_values):
        if node.left is not None:
            self.inorder(node.left, list_of_values)
        if node.data is not None:
            list_of_values.append(node.data)
        if node.right is not None:
            self.inorder(node.right, list_of_values)
        return list_of_values


start_time = time.perf_counter()
```

```
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
commands = input_file.readlines()
with open('output.txt', 'w') as output_file:
    for command in commands:
        action, value = command.split()
        value = int(value)
        match action:
            case "+":
                tree.insert(value)
            case "?":
                all_numbers = tree.inorder(tree.root, [])
                print(all_numbers[value - 1], file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Бинарное дерево поиска с реализованной функцией inorder.

Результат работы кода на примерах из текста задачи:(скрины input output файлов)



Результат работы кода на максимальных и минимальных значениях:(скрины input output файлов)
Проверка задачи на (openedu, acmp и тд при наличии в задаче). (скрин)

	Время выполнения	Затраты памяти
Пример из задачи	0.0011	0.0053

Вывод по задаче: inorder классная функция

Задача №5. Простое двоичное дерево поиска [1 балл]

Текст задачи:

Реализуйте простое двоичное дерево поиска. • Формат ввода / входного файла (input.txt). Входной файл содержит описание операций с деревом, их количество N не превышает 100. В каждой строке находится одна из следующих операций: – insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо; – delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо; – exists x – если ключ x есть в дереве выведите «true», если нет – «false»; – next x – выведите минимальный элемент в дереве, строго больший x, или «none», если такого нет; – prev x – выведите максимальный элемент в дереве, строго меньший x, или «none», если такого нет.

Листинг кода:

```
import time
import tracemalloc


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def preorder(self, list_of_values, node):
        if node.data is not None:
            list_of_values.append(node.data)
        if node.left is not None:
            self.preorder(list_of_values, node.left)
        if node.right is not None:
            self.preorder(list_of_values, node.right)
        return list_of_values

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def exists(self, x):
        return bool(self.find(x, self.root))

    def insert(self, data):
        found_parent = None
        c_node = self.root
```

```
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def tree_min(self, node):
        while node.left is not None:
            node = node.left
        return node

    def find_closest_greater_number(self, x):
        if self.root is None:
            return 0
        full_way = []
        c_node = self.root
        while True:
            full_way.append(c_node)
            if x > c_node.data:
                if c_node.right is None:
                    break
                c_node = c_node.right
            elif x < c_node.data:
                if c_node.left is None:
                    return c_node.data
                c_node = c_node.left
            else:
                if c_node.right is None:
                    break
                c_node = c_node.right
                while c_node.left is not None:
                    c_node = c_node.left
                return c_node.data
        for i in range(len(full_way) - 1, -1, -1):
            if full_way[i].data > x:
                return full_way[i].data
        return "none"

    def find_closest_less_number(self, x):
        if self.root is None:
            return 0
        full_way = []
        c_node = self.root
        while True:
            full_way.append(c_node)
            if x < c_node.data:
                if c_node.left is None:
                    break
```

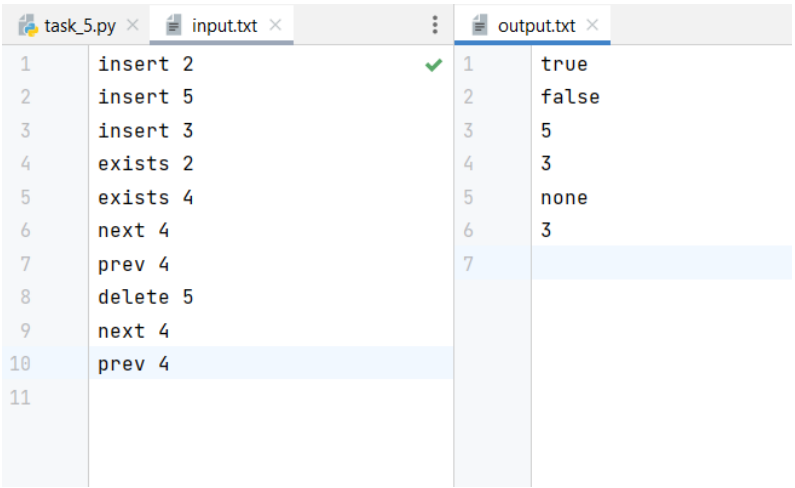
```
                c_node = c_node.left
            elif x > c_node.data:
                if c_node.right is None:
                    return c_node.data
                c_node = c_node.right
            else:
                if c_node.left is None:
                    break
                c_node = c_node.left
                while c_node.right is not None:
                    c_node = c_node.right
                return c_node.data
        for i in range(len(full_way) - 1, -1, -1):
            if full_way[i].data < x:
                return full_way[i].data
        return "none"

    def __del_leaf(self, node):
        if node.parent.left == node:
            node.parent.left = None
        elif node.parent.right == node:
            node.parent.right = None

    def __del_one_child(self, node):
        if node.parent.left == node:
            if node.left is None:
                node.parent.left = node.right
            elif node.right is None:
                node.parent.left = node.left
        elif node.parent.right == node:
            if node.left is None:
                node.parent.right = node.right
            elif node.right is None:
                node.parent.right = node.left

    def __del_two_children(self, node):
        new = self.tree_min(node.right)
        node.data = new.data
        self.__del_one_child(new)

    def delete(self, x):
        node = self.find(x, self.root)
        if node.right is None and node.left is None:
            self.__del_leaf(node)
        elif node.left is None or node.right is None:
            self.__del_one_child(node)
        else:
            self.__del_two_children(node)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
commands = input_file.readlines()
with open('output.txt', 'w') as output_file:
    for command in commands:
        action, value = command.split()
        value = int(value)
        match action:
            case "insert":
```

```
                tree.insert(value)
            case "exists":
                if tree.exists(value):
                    print("true", file=output_file)
                else:
                    print("false", file=output_file)
            case "next":
                print(tree.find_closest_greater_number(value),
file=output_file)
            case "prev":
                print(tree.find_closest_less_number(value),
file=output_file)
            case "delete":
                tree.delete(value)

print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

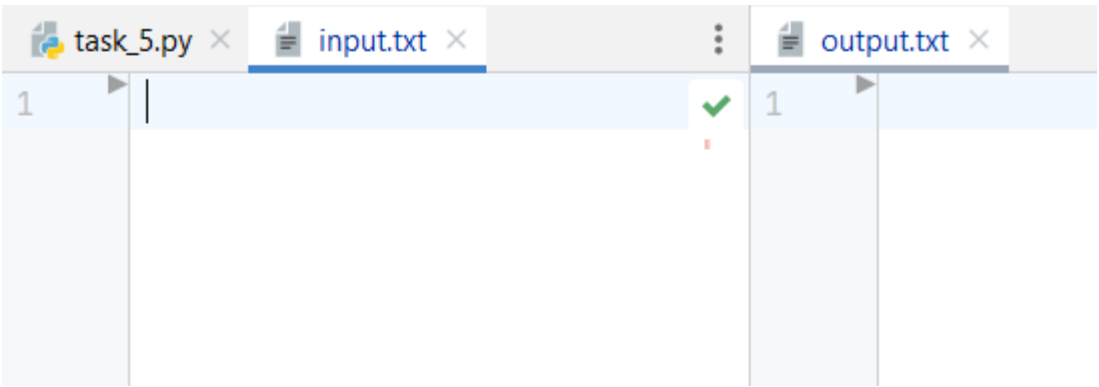
Текстовое объяснение решения:

Реализованы функции find, insert, find_closest_greater_number, find_closest_less_number, delete.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:(скрины input output файлов)



task_5.py | input.txt | randomize.py

```
next 561255854
prev 850901573
prev 552149099
next -698905380
exists -359641139
insert -90812196
prev 623516314
delete -955783273
insert -347783850
next 359097253
delete -161655196
prev 924883658
insert -357007078
prev 807615771
prev -439528363
prev 62895328
insert 320547035
next -172780720
insert -611928728
exists -509245970
insert -206230522
next 62675486
prev 845605787
insert -277873529
exists -638960290
exists 259636811
insert 663150131
next 928706982
next -354024947
delete 776162140
exists -577071189
```

output.txt

```
0
0
0
0
false
-90812196
none
-90812196
-90812196
none
-90812196
-90812196
false
320547035
320547035
false
false
none
-347783850
false
663150131
false
false
false
663150131
false
false
663150131
663150131
663150131
-611928728
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0006	0.0029
Пример из задачи	0.0011	0.0056
Верхняя граница диапазона значений входных данных из текста задачи	0.0022	0.0172

Вывод по задаче: нормально

Задача №6. Опознание двоичного дерева поиска [1.5 балла]

Текст задачи:

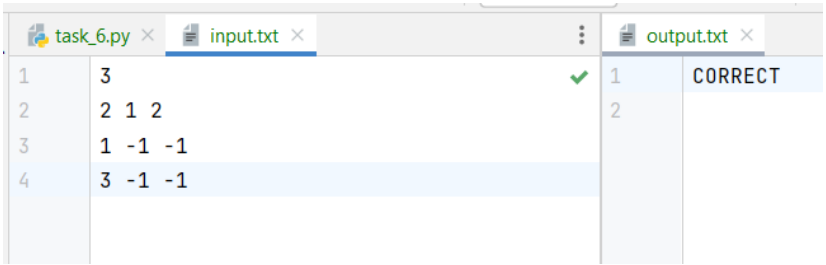
В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска. Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет.

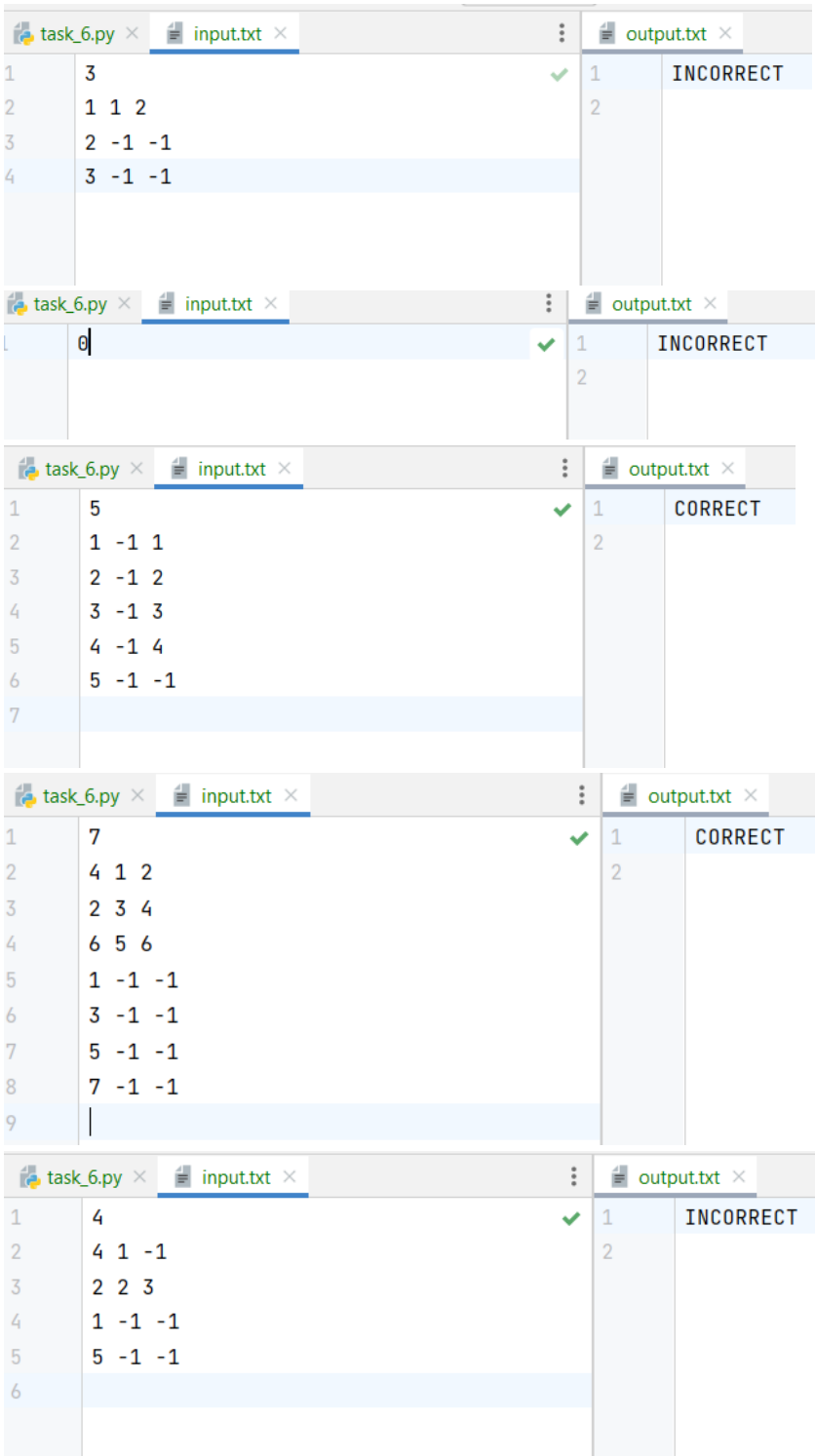
Листинг кода:

```
import time
import tracemalloc


class TreeNode:
    def __init__(self, value=None):
        self.left_child = None
        self.right_child = None
        self.value = value

    def insert_left(self, value, parent):
        if parent.left_child is None:
            parent.left_child = TreeNode(value)

    def insert_right(self, value, parent):
        if parent.right_child is None:
            parent.right child = TreeNode(value)

    def find(self, value):
        answer = None
        if self.value == value:
            answer = self
        else:
            if self.left_child is not None:
                answer = self.left_child.find(value)
            if answer is None and self.right_child is not None:
                answer = self.right_child.find(value)
        return answer

    def inorder(self, list_of_values):
        if self.left_child is not None:
            self.left_child.inorder(list_of_values)
        if self.value is not None:
            list_of_values.append(self.value)
        if self.right_child is not None:
            self.right_child.inorder(list_of_values)
        return list_of_values

start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = []
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
if info:
```

```
    tree = TreeNode(info[0][0])
    left = 1
    right = 2


    def create_tree(i):
        parent = tree.find(info[i][0])
        if parent:
            left_i = info[i][left]
            right_i = info[i][right]
            if left_i >= 0:
                tree.insert_left(info[left_i][0], parent)
                create_tree(left_i)
            else:
                tree.insert_left(None, parent)
            if right_i >= 0:
                tree.insert_right(info[right_i][0], parent)
                create_tree(right_i)
            else:
                tree.insert_right(None, parent)


    create_tree(0)
    inorder = tree.inorder([])
    for i in range(len(inorder) - 1):
        if inorder[i] < inorder[i + 1]:
            pass
        else:
            answer = "INCORRECT"
            break
    else:
        answer = "CORRECT"
else:
    answer = "INCORRECT"
with open('output.txt', 'w') as output_file:
    print(answer, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Читаю дерево. Заметим, что массив, полученный функцией inorder в бинарном дереве поиска, отсортирован по возрастанию.
Результат работы кода на примерах из текста задачи:(скрины input output файлов)





	Время выполнения	Затраты памяти
Пример из задачи	0.0014	0.0063
Пример из задачи	0.0022	0.0078

Вывод по задаче: использовала код из задачи №1

Задача №7. Опознание двоичного дерева поиска (усложненная версия) [2.5 балла]

Текст задачи:

Эта задача отличается от предыдущей тем, что двоичное дерева поиска может содержать равные ключи. Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Теперь, для каждой вершины дерева V выполняется следующее условие: • все ключи вершин из левого поддерева меньше ключа вершины V ; • все ключи вершин из правого поддерева больше или равны ключу вершины V . Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

Листинг кода:

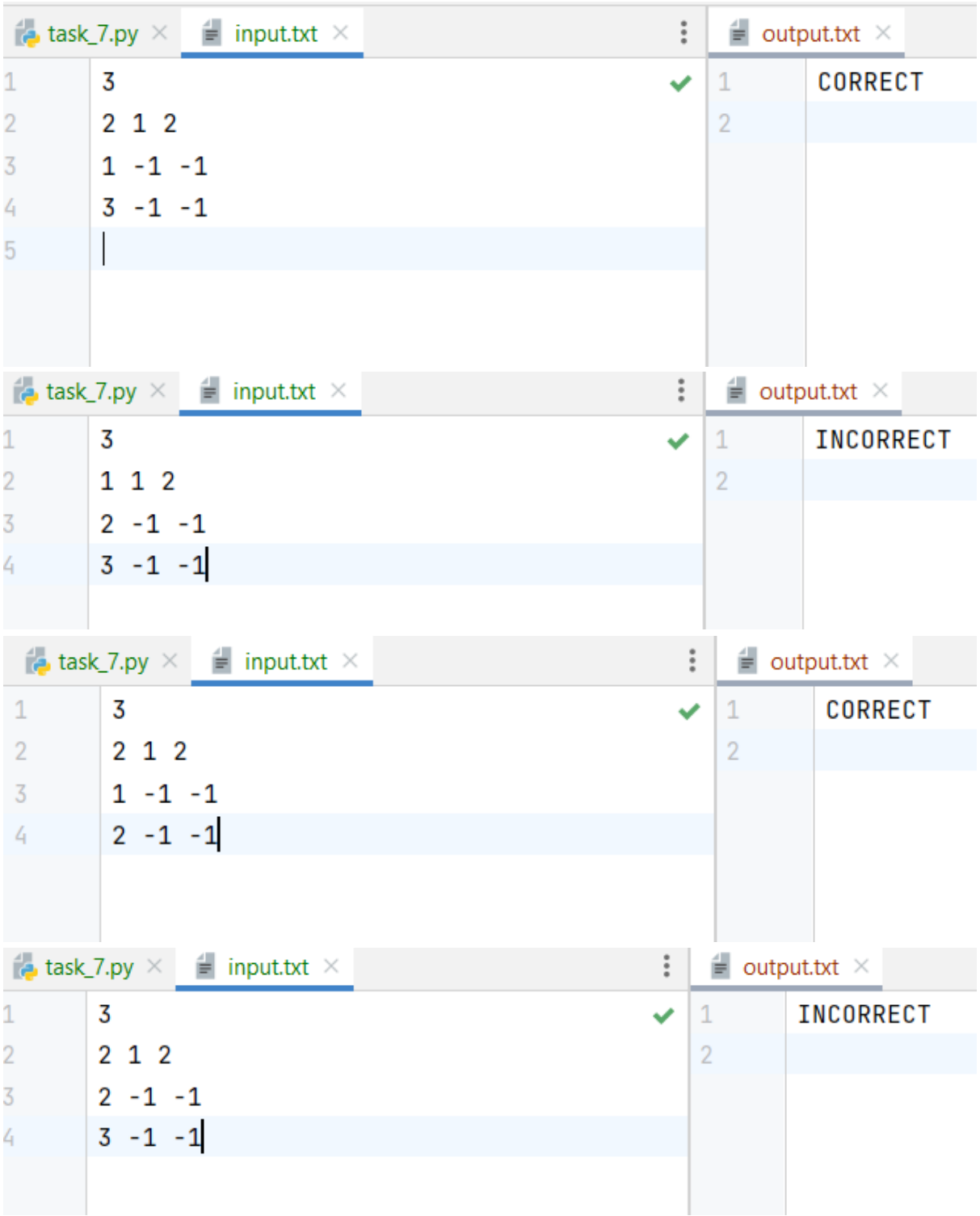
```
import time
import tracemalloc


class TreeNode:
    def __init__(self, value=None):
        self.left_child = None
        self.right_child = None
        self.value = value
        self.isFound = False

    def insert_left(self, value, parent):
        if parent.left_child is None:
            parent.left_child = TreeNode(value)

    def insert_right(self, value, parent):
        if parent.right_child is None:
            parent.right_child = TreeNode(value)

    def find(self, value):
        answer = None
        if self.value == value and self.isFound == False:
            answer = self
            self.isFound = True
        else:
            if self.left_child is not None:
                answer = self.left_child.find(value)
            if answer is None and self.right_child is not None:
                answer = self.right_child.find(value)
        return answer

    def check_tree(self):
        if self is None:
            return True
        if self.left_child.value is not None:
            if self.value > self.left_child.value:
                left_branch = self.left_child.check_tree()
            else:
```

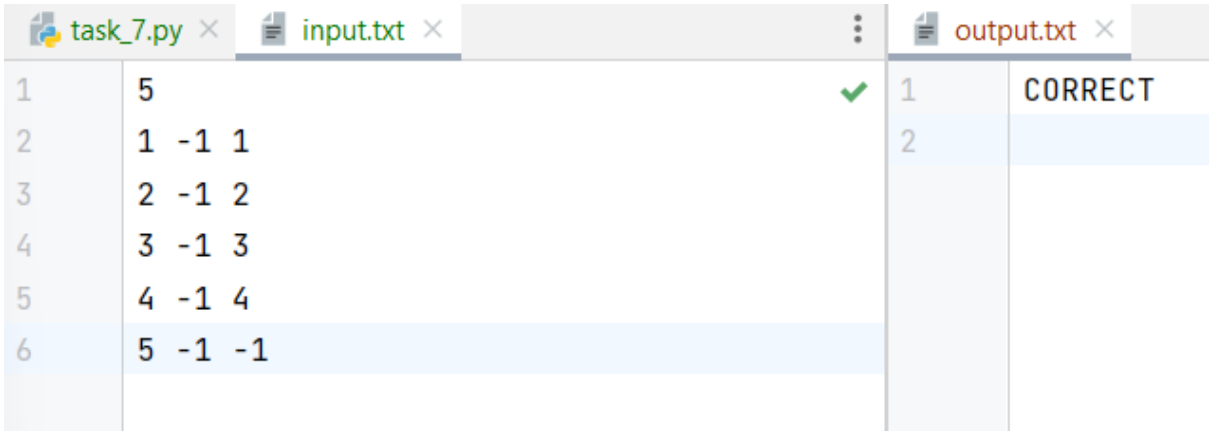
```
                return False
        else:
            left_branch = True
        if self.right_child.value is not None:
            if self.value <= self.right_child.value:
                right_branch = self.right_child.check_tree()
            else:
                return False
        else:
            right_branch = True
        return left_branch and right_branch


def create_tree(i):
    parent = tree.find(info[i][0])
    if parent:
        left_i = info[i][left]
        right_i = info[i][right]
        if left_i >= 0:
            tree.insert_left(info[left_i][0], parent)
            create_tree(left_i)
        else:
            tree.insert_left(None, parent)
        if right_i >= 0:
            tree.insert_right(info[right_i][0], parent)
            create_tree(right_i)
        else:
            tree.insert_right(None, parent)


start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = []
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
if info:
    tree = TreeNode(info[0][0])
    left = 1
    right = 2
    create_tree(0)
    if tree.check_tree():
        answer = "CORRECT"
    else:
        answer = "INCORRECT"
else:
    answer = "INCORRECT"
with open('output.txt', 'w') as output_file:
    print(answer, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

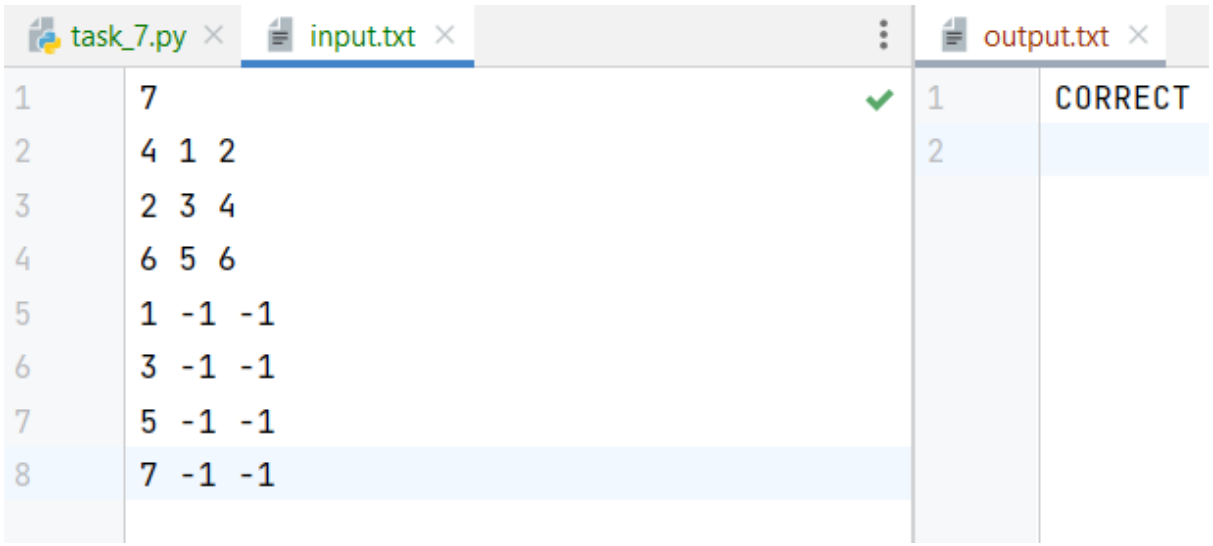
Текстовое объяснение решения:

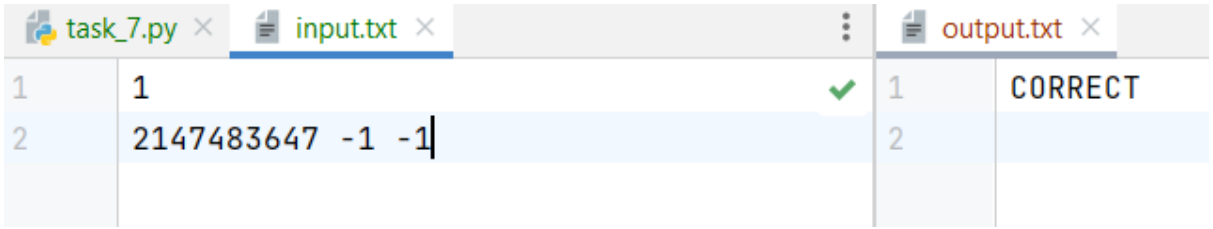
Читаю дерево. По сравнению с кодом из предыдущего задания тут присутствует функция check_tree. Она поочередно проверяет каждый узел

на выполнение условия бинарного дерева с повторами. Также из-за наличия повторов я добавила атрибут isFound для корректного составления дерева. Результат работы кода на примерах из текста задачи:









	Время выполнения	Затраты памяти
Пример из задачи	0.0011	0.0055
Пример из задачи	0.0011	0.0077

Вывод по задаче: неприятная задача, не получилось через inorder, как в предыдущей

Задача №8. Высота дерева возвращается [2 балла]

Текст задачи:

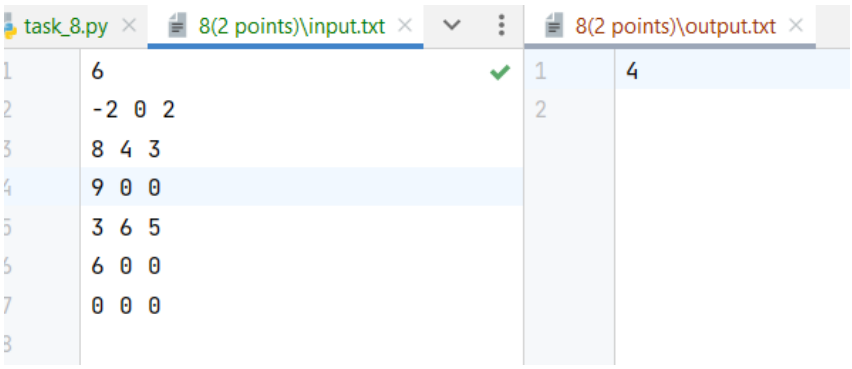
Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды. Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырем.

Листинг кода:

```
import time
import tracemalloc


class TreeNode:
    def __init__(self, value=None):
        self.left_child = None
        self.right_child = None
        self.value = value

    def insert_left(self, value, parent):
        if parent.left_child is None:
            parent.left_child = TreeNode(value)

    def insert_right(self, value, parent):
        if parent.right_child is None:
            parent.right child = TreeNode(value)

    def find(self, value):
        answer = None
        if self.value == value:
            answer = self
        else:
            if self.left_child is not None:
                answer = self.left_child.find(value)
            if answer is None and self.right_child is not None:
                answer = self.right_child.find(value)
        return answer

    def height(self):
        if self.value is None:
            return 0
        return 1 + max(self.left_child.height(), self.right_child.height())


def create_tree(i):
    parent = tree.find(info[i][0])
    if parent:
        left_i = info[i][left]
        right_i = info[i][right]
        if left_i != 0:
            tree.insert_left(info[left_i][0], parent)
            create_tree(left_i)
        else:
            tree.insert_left(None, parent)
        if right_i != 0:
            tree.insert_right(info[right_i][0], parent)
```

```
            create_tree(right_i)
        else:
            tree.insert_right(None, parent)


start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = [N]
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
tree = TreeNode(info[1][0])
left = 1
right = 2

create_tree(1)
with open('output.txt', 'w') as output_file:
    print(tree.height(), file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Написала рекурсивную функцию height(). Базовый случай: дерево со значением None имеет высоту 0. Повторяется для левого и правого поддерева и учитывает максимальную глубину
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0012	0.0077

Вывод по задаче: теперь умею находить высоту дерева

Задача №10. Проверка корректности [2 балла]

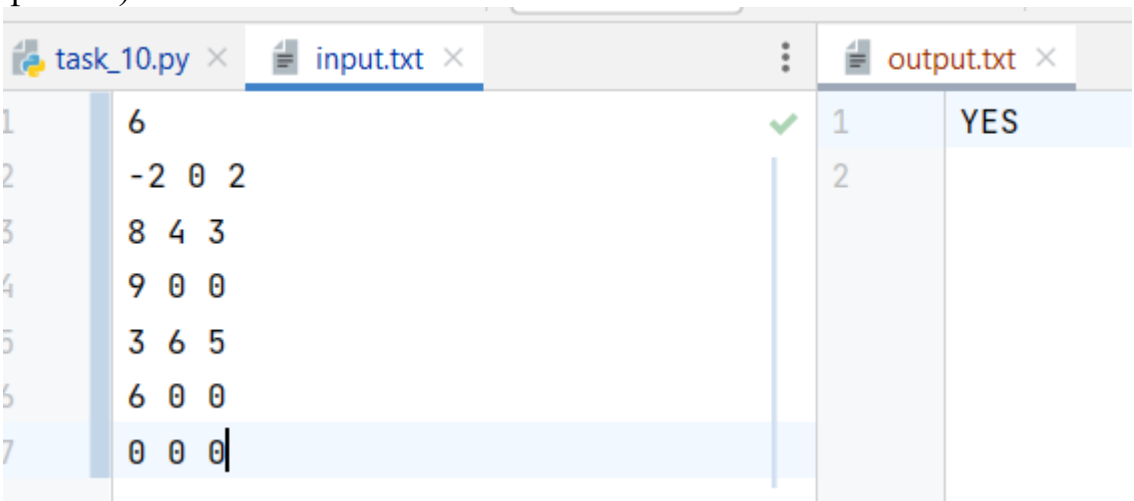
Текст задачи:
Дано двоичное дерево. Проверьте, выполняется ли для него свойство двоичного дерева поиска.

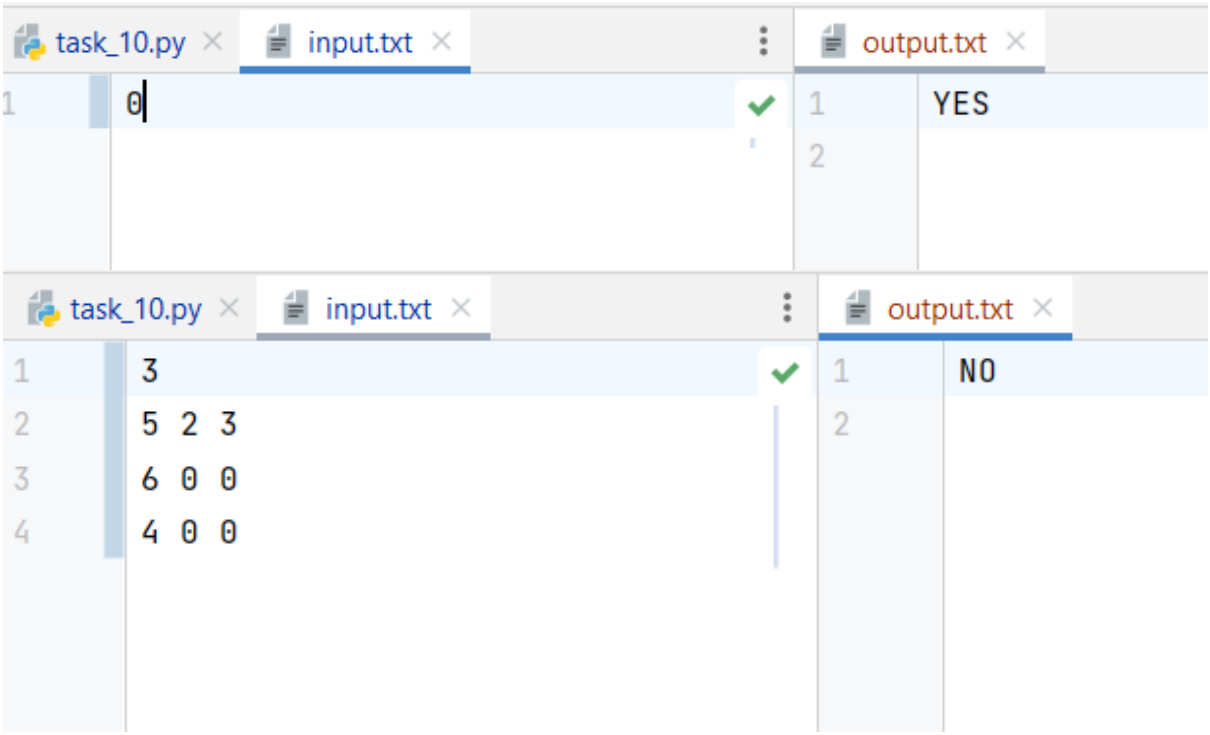
Листинг кода:

```
import time
import tracemalloc


class TreeNode:
    def __init__(self, value=None):
        self.left_child = None
        self.right_child = None
        self.value = value
        self.isFound = False

    def insert_left(self, value, parent):
        if parent.left_child is None:
            parent.left_child = TreeNode(value)

    def insert_right(self, value, parent):
        if parent.right_child is None:
            parent.right_child = TreeNode(value)

    def find(self, value):
        answer = None
        if self.value == value and self.isFound == False:
            answer = self
            self.isFound = True
        else:
            if self.left_child is not None:
                answer = self.left_child.find(value)
            if answer is None and self.right_child is not None:
                answer = self.right_child.find(value)
        return answer

    def check_tree(self):
        if self is None:
            return True
        if self.left_child.value is not None:
            if self.value > self.left_child.value:
                left_branch = self.left_child.check_tree()
            else:
                return False
        else:
            left_branch = True
        if self.right_child.value is not None:
            if self.value <= self.right_child.value:
                right_branch = self.right_child.check_tree()
            else:
                return False
        else:
            right_branch = True
        return left_branch and right_branch


def create_tree(i):
    parent = tree.find(info[i][0])
    if parent:
        left_i = info[i][left]
        right_i = info[i][right]
        if left_i != 0:
            tree.insert_left(info[left_i][0], parent)
            create_tree(left_i)
        else:
```

```
            tree.insert_left(None, parent)
        if right_i != 0:
            tree.insert_right(info[right_i][0], parent)
            create_tree(right_i)
        else:
            tree.insert_right(None, parent)


start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = [N]
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
if len(info) > 1:
    tree = TreeNode(info[1][0])
    left = 1
    right = 2
    create_tree(1)
    if tree.check_tree():
        answer = "YES"
    else:
        answer = "NO"
else:
    answer = "YES"
with open('output.txt', 'w') as output_file:
    print(answer, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
С помощью функции check_tree () проходим по всем узлам и проверяем, выполняется ли условие бинарного дерева поиска.
Результат работы кода на примерах из текста задачи:(скрины input output файлов)





	Время выполнения	Затраты памяти
Пример из задачи	0.0013	0.0078
Пример из задачи	0.0009	0.0028
Пример из задачи	0.0010	0.0055

Вывод по задаче:
Умею проверять произвольное бинарное дерево на корректность (является ли BST)

Задача №12. Проверка сбалансированности [2 балла]

Текст задачи:
АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева
отличается от высоты ее правого поддерева не больше, чем на единицу.
Введем понятие баланса вершины: для вершины дерева V ее баланс B(V) равен разности высоты правого поддерева и высоты левого поддерева.
Таким образом, свойство АВЛ-дерева, приведенное выше, можно сформулировать
следующим образом: для любой ее вершины V выполняется следующее неравенство:

$-1 \le B(V) \le 1$

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой
недели. зеркально отражено. по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит
Вам неудобств. В литературе по алгоритмам – как российской, так и мировой – ситуация, как правило, примерно та
же.

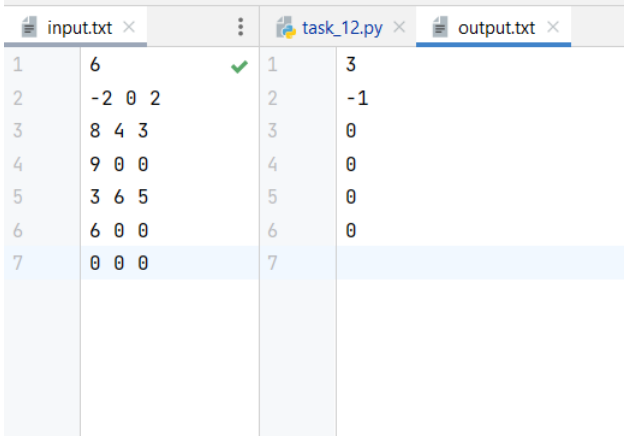
Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

Листинг кода:

```
import time
import tracemalloc


class TreeNode:
    def __init__(self, value=None):
        self.left_child = None
        self.right_child = None
        self.value = value

    def insert_left(self, value, parent):
        if parent.left_child is None:
            parent.left_child = TreeNode(value)

    def insert_right(self, value, parent):
        if parent.right_child is None:
            parent.right_child = TreeNode(value)

    def find(self, value):
        answer = None
        if self.value == value:
            answer = self
        else:
            if self.left_child is not None:
                answer = self.left_child.find(value)
            if answer is None and self.right_child is not None:
                answer = self.right_child.find(value)
        return answer

    def is_balanced(self, node, isBalanced=True):
        if node.value is None or not isBalanced:
            return 0, isBalanced, 0
        left_height, isBalanced, balance =
self.is_balanced(node.left_child, isBalanced)
        right_height, isBalanced, balance =
self.is_balanced(node.right_child, isBalanced)
        if abs(left_height - right_height) > 1:
            isBalanced = False
        return max(left_height, right_height) + 1, isBalanced, right_height
- left_height


def create_tree(i):
    parent = tree.find(info[i][0])
    if parent:
```

```
        left_i = info[i][left]
        right_i = info[i][right]
        if left_i != 0:
            tree.insert_left(info[left_i][0], parent)
            create_tree(left_i)
        else:
            tree.insert_left(None, parent)
        if right_i != 0:
            tree.insert_right(info[right_i][0], parent)
            create_tree(right_i)
        else:
            tree.insert_right(None, parent)


start_time = time.perf_counter()
tracemalloc.start()
input_file = open('input.txt')
N = int(input_file.readline())
info = [N]
for node in range(N):
    K, L, R = map(int, input_file.readline().split())
    info.append([K, L, R])
tree = TreeNode(info[1][0])
left = 1
right = 2

create_tree(1)
with open('output.txt', 'w') as output_file:
    for i in range(1, N + 1):
        print(tree.is_balanced(tree.find(info[i][0]))[2], file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Написана функция isBalanced, которая выдает высоту узла, сбалансированность дерева и значение баланса.
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0012	0.0092

Вывод по задаче: пришлось адаптировать функцию баланса под другой инпут

Задача №13. Делаю я левый поворот... [3 балла]

Текст задачи:
Дано дерево, в котором баланс корня равен 2. Сделайте левый поворот.
Листинг кода:

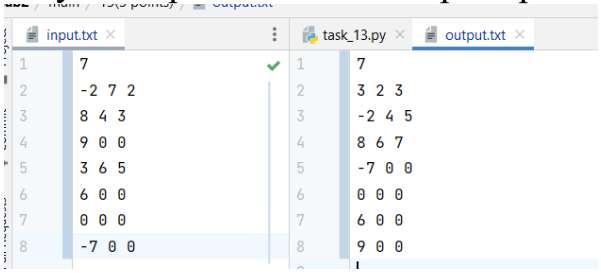
```
import time
import tracemalloc
import queue


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def __nodes_of_level(self, node, level, list_of_values):
        if node is None:
            return False, list_of_values
        if level == 1:
            list_of_values.append(node.data)
            return True, list_of_values
        left, list of values = self.  nodes of level(node.left, level - 1,
list_of_values)
        right, list_of_values = self.__nodes_of_level(node.right, level -
1, list_of_values)
        return left or right, list_of_values

    def bsf(self, node, list_of_values):
        level = 1
        while self.__nodes_of_level(node, level, list_of_values)[0]:
            level = level + 1
        return list_of_values

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def insert(self, data):
        found_parent = None
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
```

```
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def isHeightBalanced(self, node, isBalanced=True):
        if node is None or not isBalanced:
            return 0, isBalanced
        left_height, isBalanced = self.isHeightBalanced(node.left,
isBalanced)
        right_height, isBalanced = self.isHeightBalanced(node.right,
isBalanced)
        if abs(left_height - right_height) > 1:
            isBalanced = False
        return max(left_height, right_height) + 1, isBalanced

    def rotation_left(self, node):
        if node is not None:
            new_node = node.right
            node.right = new_node.left
            if new_node.left:
                new_node.left.parent = node
            new_node.left = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    def rotation_right(self, node):
        if node is not None:
            new_node = node.left
            node.left = new_node.right
            if new_node.right:
                new_node.right.parent = node
            new_node.right = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
```

```
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    #
    def rotation(self, node):
        if node is None:
            return
        h1 = tree.isHeightBalanced(node.left, True)[0]
        h2 = tree.isHeightBalanced(node.right, True)[0]
        if h1 - h2 > 1:
            if tree.isHeightBalanced(node.left.left, True)[0] >
tree.isHeightBalanced(node.left.right, True)[0]:
                # print("small right")
                tree.rotation_right(node)
            else:
                # print("big right")
                tree.rotation_left(node.left)
                tree.rotation_right(node)
        elif h2 - h1 > 1:
            if tree.isHeightBalanced(node.right.right, True)[0] >
tree.isHeightBalanced(node.right.left, True)[0]:
                # print("small left")
                tree.rotation_left(node)
            else:
                # print("big left")
                tree.rotation_right(node.right)
                tree.rotation_left(node)
        else:
            self.rotation(node.right)
            self.rotation(node.left)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
N = int(input_file.readline())
values = [None]
for i in range(N):
    K, L, R = map(int, input_file.readline().split())
    tree.insert(K)
    values.append(K)
tree.rotation(tree.root)
bsf = tree.bsf(tree.root, [])
with open('output.txt', 'w') as output_file:
    print(N, file=output_file)
    for i in bsf:
        val = tree.find(i, tree.root)
        if val.left:
            l_ch = val.left.data
            l_ch = bsf.index(l_ch)
        else:
            l_ch = -1
        if val.right:
```

```
            r_ch = val.right.data
            r_ch = bsf.index(r_ch)
        else:
            r_ch = -1
        print(val.data, l_ch + 1, r_ch + 1, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Читаю входные данные как дерево поиска. Использую функцию balancing для балансировки. Написала функцию обхода в ширину, чтобы сделать корректный вывод.
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0013	0.0069

Вывод по задаче: мой код делает и правый поворот 😊

Задача №14. Вставка в АВЛ-дерево [3 балла]

Текст задачи:
Вставка в АВЛ-дерево вершины V с ключом X при условии, что такой вершины в этом дереве нет, осуществляется
следующим образом:
• находится вершина W, ребенком которой должна стать вершина V ;
• вершина V делается ребенком вершины W;
• производится подъем от вершины W к корню, при этом, если какая-то из вершин несбалансирована, производится, в зависимости от значения баланса, левый или правый поворот.
Первый этап нуждается в пояснении. Спуск до будущего родителя вершины V осуществляется, начиная от корня,
следующим образом:
• Пусть ключ текущей вершины равен Y .

• Если X < Y и у текущей вершины есть левый ребенок, переходим к левому ребенку.
• Если X < Y и у текущей вершины нет левого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.
• Если X > Y и у текущей вершины есть правый ребенок, переходим к правому ребенку.
• Если X > Y и у текущей вершины нет правого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.
Отдельно рассматривается следующий крайний случай – если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

Листинг кода:

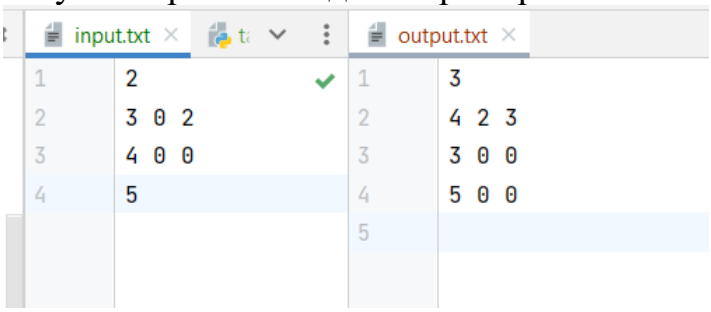
```
import time
import tracemalloc
import queue


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def __nodes_of_level(self, node, level, list_of_values):
        if node is None:
            return False, list_of_values
        if level == 1:
            list_of_values.append(node.data)
            return True, list_of_values
        left, list_of_values = self.__nodes_of_level(node.left, level - 1,
list_of_values)
        right, list_of_values = self.__nodes_of_level(node.right, level -
1, list_of_values)
        return left or right, list_of_values

    def bsf(self, node, list_of_values):
        level = 1
        while self.__nodes_of_level(node, level, list_of_values)[0]:
            level = level + 1
        return list_of_values

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
```

```
        else:
            return self.find(x, node.right)

    def insert(self, data):
        found_parent = None
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def isHeightBalanced(self, node, isBalanced=True):
        if node is None or not isBalanced:
            return 0, isBalanced
        left_height, isBalanced = self.isHeightBalanced(node.left,
isBalanced)
        right_height, isBalanced = self.isHeightBalanced(node.right,
isBalanced)
        if abs(left_height - right_height) > 1:
            isBalanced = False
        return max(left_height, right_height) + 1, isBalanced

    def rotation_left(self, node):
        if node is not None:
            new_node = node.right
            node.right = new_node.left
            if new_node.left:
                new_node.left.parent = node
            new_node.left = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    def rotation_right(self, node):
        if node is not None:
            new_node = node.left
```

```
            node.left = new_node.right
            if new_node.right:
                new_node.right.parent = node
            new_node.right = node
            if node.parent is not None:
                if node.parent.data > node.data:
                    node.parent.left = new_node
                    new_node.parent = node.parent
                else:
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    #
    def rotation(self, node):
        if node is None:
            return
        h1 = tree.isHeightBalanced(node.left, True)[0]
        h2 = tree.isHeightBalanced(node.right, True)[0]
        if h1 - h2 > 1:
            if tree.isHeightBalanced(node.left.left, True)[0] >
tree.isHeightBalanced(node.left.right, True)[0]:
                # print("small right")
                tree.rotation_right(node)
            else:
                # print("big right")
                tree.rotation_left(node.left)
                tree.rotation_right(node)
        elif h2 - h1 > 1:
            if tree.isHeightBalanced(node.right.right, True)[0] >
tree.isHeightBalanced(node.right.left, True)[0]:
                # print("small left")
                tree.rotation_left(node)
            else:
                # print("big left")
                tree.rotation_right(node.right)
                tree.rotation_left(node)
        else:
            self.rotation(node.right)
            self.rotation(node.left)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
N = int(input_file.readline())
values = [None]
for i in range(N):
    K, L, R = map(int, input_file.readline().split())
    tree.insert(K)
    values.append(K)
tree.insert(int(input_file.readline()))
tree.rotation(tree.root)
bsf = tree.bsf(tree.root, [])
```

```
with open('output.txt', 'w') as output_file:
    print(N+1, file=output_file)
    for i in bsf:
        val = tree.find(i, tree.root)
        if val.left:
            l_ch = val.left.data
            l_ch = bsf.index(l_ch)
        else:
            l_ch = -1
        if val.right:
            r_ch = val.right.data
            r_ch = bsf.index(r_ch)
        else:
            r_ch = -1
        print(val.data, l_ch + 1, r_ch + 1, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
После вставки нового элемента дерево балансируется заново.
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0010	0.0062

Вывод по задаче: дерево балансируется само, удобно

Задача №15. Удаление из АВЛ-дерева [3 балла]

Текст задачи:
Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом:
• путем спуска от корня и проверки ключей находится V – удаляемая вершина;
• если вершина V – лист (то есть, у нее нет детей):
– удаляем вершину;
– поднимаемся к корню, начиная с бывшего родителя вершины V , при этом если встречается несбалансиро-

ванная вершина, то производим поворот.
• если у вершины V не существует левого ребенка:
– следовательно, баланс вершины равен единице и ее правый ребенок – лист;
– заменяем вершину V ее правым ребенком;
– поднимаемся к корню, производя, где необходимо, балансировку.
• иначе:
– находим R – самую правую вершину в левом поддереве;
– переносим ключ вершины R в вершину V ;
– удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являю-
щегося листом);
– поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку.
Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины - корня.
Результатом удаления в этом случае будет пустое дерево.
Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как
тестирующая система проверяет точное равенство получающихся деревьев.
Листинг кода:

```
import time
import tracemalloc
import queue


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None


class Tree:
    def __init__(self):
        self.root = None

    def __nodes_of_level(self, node, level, list_of_values):
        if node is None:
            return False, list_of_values
        if level == 1:
            list_of_values.append(node.data)
            return True, list_of_values
        left, list_of_values = self.__nodes_of_level(node.left, level - 1,
list_of_values)
        right, list_of_values = self.__nodes_of_level(node.right, level -
1, list of values)
        return left or right, list_of_values

    def bsf(self, node, list_of_values):
        level = 1
```

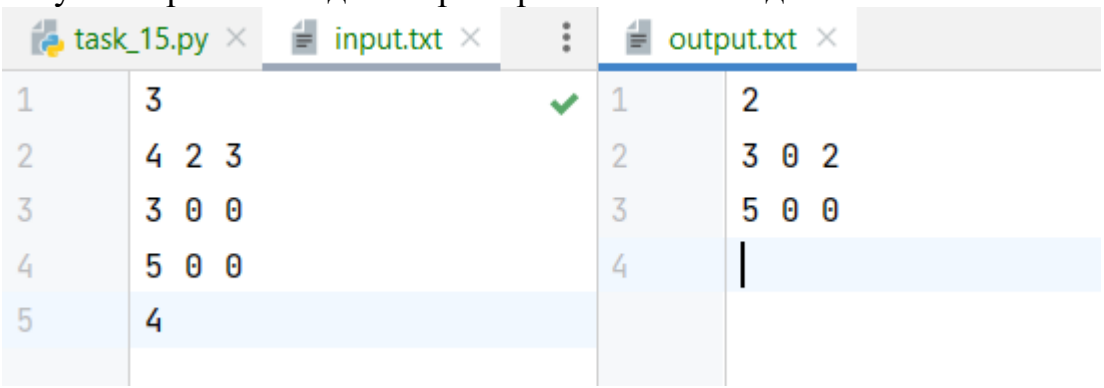
```
        while self.__nodes_of_level(node, level, list_of_values)[0]:
            level = level + 1
        return list_of_values

    def tree_min(self, node):
        while node.left is not None:
            node = node.left
        return node

    def tree_max(self, node):
        while node.right is not None:
            node = node.right
        return node

    def find(self, x, node):
        if node is None or node.data == x:
            return node
        elif x < node.data:
            return self.find(x, node.left)
        else:
            return self.find(x, node.right)

    def insert(self, data):
        found_parent = None
        c_node = self.root
        while c_node is not None:
            found_parent = c_node
            if data < c_node.data:
                c_node = c_node.left
            elif data > c_node.data:
                c_node = c_node.right
            else:
                return
        new = Node(data)
        if found_parent is None:
            self.root = new
            new.parent = None
        elif data < found_parent.data:
            found_parent.left = new
            new.parent = found_parent
        elif data > found_parent.data:
            found_parent.right = new
            new.parent = found_parent

    def __del_leaf(self, node):
        if node.parent.left == node:
            node.parent.left = None
        elif node.parent.right == node:
            node.parent.right = None

    def __del_one_child(self, node):
        if node.parent.right == node:
            if node.left is None:
                node.parent.right = node.right
            elif node.right is None:
                node.parent.right = node.left
        elif node.parent.left == node:
            if node.left is None:
                node.parent.left = node.right
            elif node.right is None:
                node.parent.left = node.left
```

```
def __del_two_children(self, node):
    new = self.tree_max(node.left)
    node.data = new.data
    self.__del_one_child(new)

def delete(self, x):
    node = self.find(x, self.root)
    if node:
        if node.right is None and node.left is None:
            self.__del_leaf(node)
        elif node.left is None or node.right is None:
            self.__del_one_child(node)
        else:
            self.__del_two_children(node)
    if not self.is_balanced(self.root, True)[1]:
        self.balancing(self.root)

def is_balanced(self, node, isBalanced=True):
    if node is None or not isBalanced:
        return 0, isBalanced
    left_height, isBalanced = self.is_balanced(node.left, isBalanced)
    right_height, isBalanced = self.is_balanced(node.right, isBalanced)
    if abs(left_height - right_height) > 1:
        isBalanced = False
    return max(left_height, right_height) + 1, isBalanced

def rotation_left(self, node):
    if node is not None:
        new_node = node.right
        node.right = new_node.left
        if new_node.left:
            new_node.left.parent = node
        new_node.left = node
        if node.parent is not None:
            if node.parent.data > node.data:
                node.parent.left = new_node
                new_node.parent = node.parent
            else:
                node.parent.right = new_node
                new_node.parent = node.parent
        else:
            self.root = new_node
            new_node.parent = None
        node.parent = new_node
        return new_node
    else:
        return node

def rotation_right(self, node):
    if node is not None:
        new_node = node.left
        node.left = new_node.right
        if new_node.right:
            new_node.right.parent = node
        new_node.right = node
        if node.parent is not None:
            if node.parent.data > node.data:
                node.parent.left = new_node
                new_node.parent = node.parent
            else:
```

```
                    node.parent.right = new_node
                    new_node.parent = node.parent
            else:
                self.root = new_node
                new_node.parent = None
            node.parent = new_node
            return new_node
        else:
            return node

    #
    def balancing(self, node):
        if node is None:
            return
        h1 = tree.is_balanced(node.left, True)[0]
        h2 = tree.is_balanced(node.right, True)[0]
        if h1 - h2 > 1:
            if tree.is_balanced(node.left.left, True)[0] >
tree.is_balanced(node.left.right, True)[0]:
                # print("small right")
                tree.rotation_right(node)
            else:
                # print("big right")
                tree.rotation_left(node.left)
                tree.rotation_right(node)
        elif h2 - h1 > 1:
            if tree.is_balanced(node.right.right, True)[0] >
tree.is_balanced(node.right.left, True)[0]:
                # print("small left")
                tree.rotation_left(node)
            else:
                # print("big left")
                tree.rotation_right(node.right)
                tree.rotation_left(node)
        else:
            self.balancing(node.right)
            self.balancing(node.left)


start_time = time.perf_counter()
tracemalloc.start()
tree = Tree()
input_file = open('input.txt')
N = int(input_file.readline())
values = [None]
for i in range(N):
    K, L, R = map(int, input_file.readline().split())
    tree.insert(K)
    values.append(K)
tree.delete(int(input_file.readline()))
tree.balancing(tree.root)
bsf = tree.bsf(tree.root, [])
with open('output.txt', 'w') as output_file:
    print(N - 1, file=output_file)
    for i in bsf:
        val = tree.find(i, tree.root)
        if val.left:
            l_ch = val.left.data
            l_ch = bsf.index(l_ch)
        else:
            l_ch = -1
```

```
        if val.right:
            r_ch = val.right.data
            r_ch = bsf.index(r_ch)
        else:
            r_ch = -1
        print(val.data, l_ch + 1, r_ch + 1, file=output_file)
print("Время: ", time.perf_counter() - start_time)
print("Память: ", float(tracemalloc.get_tracemalloc_memory()) / (2 ** 20))
tracemalloc.stop()
```

Текстовое объяснение решения:
Добавила функцию удаления узла из дерева. После удаления происходит балансировка.
Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0033	0.0172

Вывод по задаче: теперь умею и удалять

Вывод

В ходе данной лабораторной работы было изучено понятие бинарных деревьев поиска и их реализация на языке программирования Python. Были рассмотрены следующие операции с бинарными деревьями: вставка узла, удаление узла, поиск узла и обход дерева в прямом, обратном и симметричном порядке. В результате выполнения работы были получены навыки работы с бинарными деревьями поиска и использования их в практических задачах.