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 8

Выполнил:
Макунина А.А.
К32421

Проверила:
Артамонова В.Е.

Санкт-Петербург
2022 г.

Содержание отчета

Задачи по варианту

Задача №3. Простейшее BST (1 балл)

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.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \leq x \leq 10^9$, $1 \leq N \leq 300000$
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «> x» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

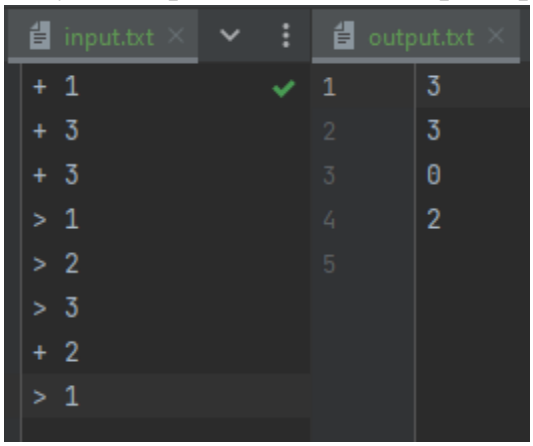
Я использую два класса – Node и Tree. Класс Node хранит свой собственный ключ, а также ключи левого и правого наследников. В классе Tree корневое значение root — это ключ корневого узла дерева. Функция добавления add добавляет новое значение в дерево в соответствии с правилом: левый дочерний элемент меньше родительского, правый - больше. Функция find_min_element выполняет поиск минимального элемента, размер которого больше указанного. Если ключ вершины меньше указанного, мы переключаемся на правый дочерний элемент, в противном случае – на левый. Если предыдущий ключ был меньше x, а текущий больше, то мы возвращаем его. Если ключ больше x и код выполняется до конца, то мы возвращаем последний узел, больший x.

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


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

    def add(self, key):
```

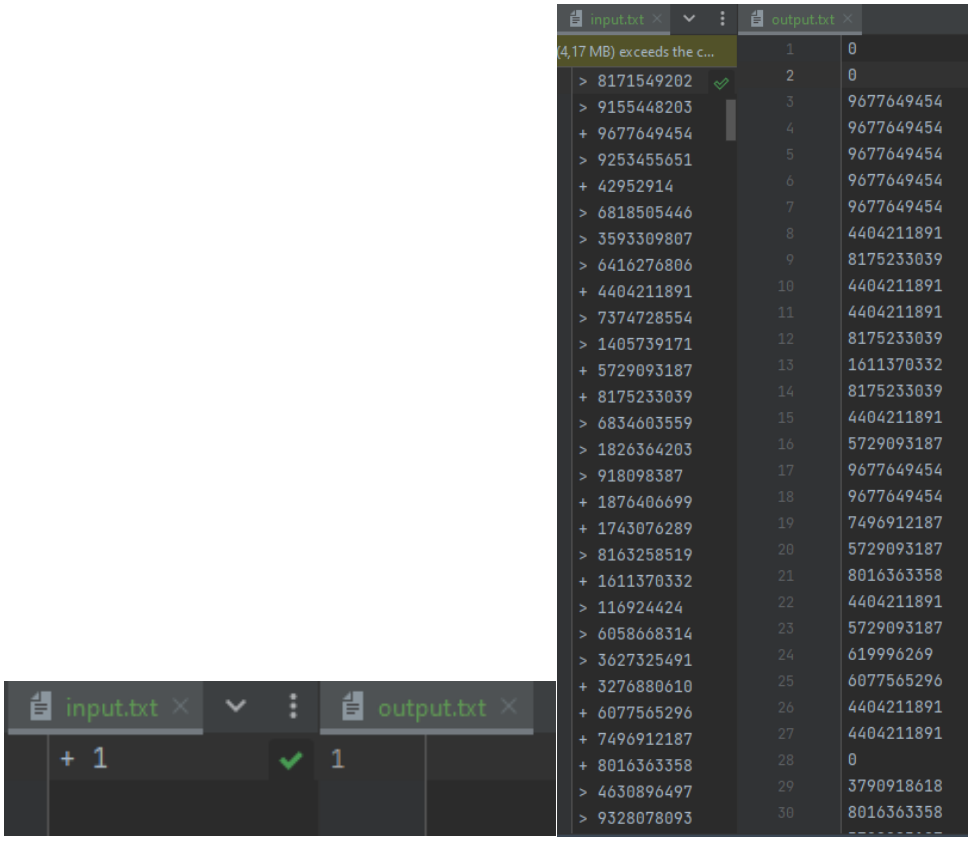
```
        parent = None
        top = self.root
        while top is not None:
            parent = top
            if key < top.key:
                top = top.left
            elif key > top.key:
                top = top.right
            else:
                return
        nd = Node(key)
        if parent is None:
            self.root = nd
        elif key < parent.key:
            parent.left = nd
        elif key > parent.key:
            parent.right = nd

    def find_min_element(self, x):
        if self.root is None:
            return 0
        arr = []
        node = self.root
        while True:
            arr.append(node)
            if x > node.key:
                if node.right == None:
                    break
                node = node.right
            elif x < node.key:
                if node.left == None:
                    return node.key
                node = node.left
            else:
                if node.right == None:
                    break
                node = node.right
                while node.left != None:
                    node = node.left
                return node.key

        for i in range(len(arr) - 1, -1, -1):
            if arr[i].key > x:
```

```
                return arr[i].key
        return 0

fi = open('input.txt', 'r')
fo = open('output.txt', 'w')
t_start = time.perf_counter()
tracemalloc.start()
tree = Tree()
s = fi.readline()
while s != "":
    operand, num = s.split()
    num = int(num)
    if operand == '+':
        tree.add(num)
    else:
        fo.write(f"{tree.find_min_element(num)}\n")
    s = fi.readline()

fi.close()
fo.close()
```

Результат работы кода на примере из текста задачи:



Результат работы кода при минимальном и максимальном значениях:



	Время выполнения, с	Затраты памяти, мб
Нижняя граница диапазона значений входных данных из текста задачи	0.0011908999999999947	0.019178390502929688
Пример из задачи	0.003729199999999988	0.019532203674316406
Верхняя граница диапазона значений входных данных из текста задачи	1.544868863	26.2221622467041

Вывод по задаче: начала использовать классы для алгоритмов, реализовала двоичное дерево поиска как раз с помощью класса, поиск элемента в таком дереве выполняется за $O(\log n)$.

Задача №10. Проверка корректности (2 балла)

Свойство двоичного дерева поиска можно сформулировать следующим образом: для каждой вершины дерева выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева больше ключа вершины V.

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

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева.

  В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).

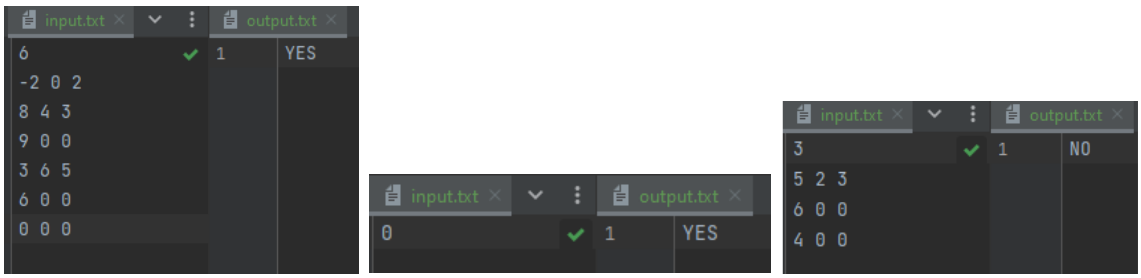
- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$.
- На 60% от при $0 \le N \le 2000$.
- **Формат вывода / выходного файла (output.txt).** Выведите «YES», если данное во входном файле дерево является двоичным деревом поиска, и «NO», если не является.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.

Свойство двоичного дерева поиска можно сформулировать следующим образом. Для каждой вершины дерева выполняется следующее условие:
• все ключи вершин из левого поддерева меньше ключа вершины V;
• все ключи вершин из правого поддерева больше ключа вершины V.
Как это работает – рекурсивно проверяем условие соответствия бинарному дереву поиска у каждого листка и его родителя.

```
class Node:
    def __init__(self, data=0, left=0, right=0):
        self.data = data
        self.left = left
        self.right = right


def get_result(a, i, min, max):
    if i == -1:
        return True
    if a[i].data <= min or a[i].data >= max:
        return False
    return get_result(a, a[i].left, min, a[i].data)
and get_result(a,

a[i].right, a[i].data, max)
```

```
t_start = time.perf_counter()
tracemalloc.start()
f = open('input.txt')
n = int(f.readline())
if n == 0:
    result = 'YES'
else:
    array = []
    i = 0
    while i < n:
        a = list(f.readline().split())
        array.append(Node(int(a[0]), int(a[1]) - 1,
int(a[2]) - 1))
        i += 1
    result = get_result(array, 0, -math.inf,
math.inf)
    if result:
        result = 'YES'
    else:
        result = 'NO'
f.close()
w = open('output.txt', 'w')
w.write(str(result))
w.close()
```

Результат работы кода на примере из текста задачи:



	Время выполнения, с	Затраты памяти, мб
Пример 1 из задачи	0.0011975999999999931	0.01804065704345703
Пример 2 из задачи	0.0012735999999999997	0.016994476318359375
Пример 3 из задачи	0.001920099999999994	0.017394065856933594

Вывод по задаче: у меня нет возможности провести тесты на OpenEdu, так как курс закончился. На примерах из задачи код работает корректно.

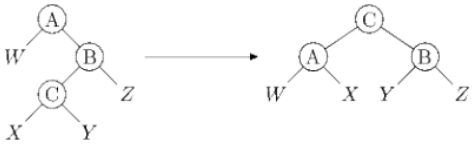
Задача №13. Делаю я левый поворот... (3 балла)

Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые* и *правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем -1.

Существует два разных левых (как, разумеется, и правых) поворота: *большой* и *малый* левый поворот.

Малый левый поворот осуществляется следующим образом:

Заметим, что если до выполнения малого левого поворота был нарушен баланс только корня дерева, то после его выполнения все вершины становятся сбалансированными, за исключением случая, когда у правого ребенка корня баланс до поворота равен -1. В этом случае вместо малого левого поворота выполняется большой левый поворот, который осуществляется так:



Дано дерево, в котором баланс корня равен 2. Сделайте левый поворот.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева.

  В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. Баланс корня дерева (вершины с номером 1) равен 2, баланс всех остальных вершин находится в пределах от -1 до 1.

- **Ограничения на входные данные.** $3 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$.

- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления левого поворота. Нумерация вершин может быть произвольной при условии соблюдения формата. Так, номер вершины должен быть меньше номера ее детей.

- **Ограничение по времени. 2 сек.**

- Ограничение по памяти. 256 мб.

Для хранения информации о каждом узле дерева используется класс Node, атрибутами которого являются данные о значении узла, левом и правом дочернем элементе, родительском элементе, высоте узла и его балансе.
Кроме того, для построения самого дерева создается класс, атрибутами которого являются корень дерева и все остальные узлы. Тремя основными методами класса являются get_height, get_balance и left_rotation. Используя метод get_balance, вычисляется баланс каждого узла, используя метод get_height, вычисляется высота каждого узла, а используя метод left_rotation, левый вращение зависит от баланса правого дочернего

элемента корня дерева. Для записи сбалансированного дерева в файл используется deque из модуля collections.

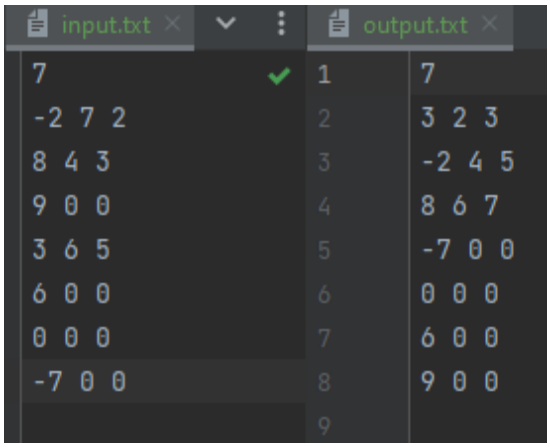
```
from collections import deque


class Node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0
        self.balance = None


class Tree:

    def __init__(self):
        self.root = None
        self.nodes = {}

    def get_balance(self):
        for i in range(1, n + 1):
            node = self.nodes[i]
            if node.left:
                if node.right:
                    node.balance = node.right.height
- node.left.height
                else:
                    node.balance = -node.left.height
            else:
                if node.right:
                    node.balance = node.right.height
                else:
                    node.balance = 0

    def get_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
```

```
                parent.height = node.height + 1
            else:
                break
            node = parent

    def left_rotation(self, node):
        node_1 = node.right
        if node_1.balance == -1:
            node_2 = node_1.left
            node_3 = node_2.left
            node_4 = node_2.right
            if node_3:
                node_3.parent = node
            node.right = node_3
            if node_4:
                node_4.parent = node_1
            node_1.left = node_4
            if node.key != self.root.key:
                node_2.parent = node.parent
                if node.parent.left == node:
                    node.parent.left = node_2
                else:
                    node.parent.right = node_2
            else:
                self.root = node_2
            node.parent = node_2
            node_2.left = node
            node_1.parent = node_2
            node_2.right = node_1
        else:
            node_4 = node_1.left
            if node_4:
                node_4.parent = node
            node.right = node_4
            if node != self.root:
                node_1.parent = node.parent
                if node.parent.left == node:
                    node.parent.left = node_1
                else:
                    node.parent.right = node_1
            else:
                self.root = node_1
            node.parent = node_1
```

```
            node_1.left = node

    def write_tree_in_file(self):
        q = deque()
        q.append(self.root)
        i = 1
        with open('output.txt', 'w') as f:
            global n
            f.write(str(n) + '\n')
            while len(q) != 0:
                node = q.popleft()
                f.write(str(node.key))
                if node.left:
                    i += 1
                    f.write(' ' + str(i))
                    q.append(node.left)
                else:
                    f.write(' 0')
                if node.right:
                    i += 1
                    f.write(' ' + str(i) + '\n')
                    q.append(node.right)
                else:
                    f.write(' 0\n')


f = open('input.txt')
n = int(f.readline())
t_start = time.perf_counter()
tracemalloc.start()
tree, data, leaves = Tree(), [], []
for i in range(1, n + 1):
    data.append(list(map(int, f.readline().split())))
    tree.nodes[i] = Node()
    tree.nodes[i].key = data[i - 1][0]
    if data[i - 1][1] == 0 and data[i - 1][2] == 0:
        leaves.append(i)

for i in range(1, n + 1):
    if data[i - 1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i -
1][1]]
        tree.nodes[data[i - 1][1]].parent =
```

```
tree.nodes[i]
    if data[i - 1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i -
1][2]]
        tree.nodes[data[i - 1][2]].parent =
tree.nodes[i]
    if i == 1:
        tree.root = tree.nodes[i]

for i in leaves:
    tree.get_height(tree.nodes[i])

tree.get_balance()
tree.left_rotation(tree.root)
tree.write_tree_in_file()
```

Результат работы кода на примере из текста задачи:



	Время выполнения, с	Затраты памяти, мб
Пример из задачи	0.0016624999999999973	0.016794204711914062

Вывод по задаче: и снова OpenEdu не позволил сделать тесты. Задача сложная. Приведённый пример выполнен успешно.

Дополнительные задачи

Задача №6. Опознание двоичного дерева поиска (2.5 балла)

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.

Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска. 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 ключа вершины V.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию. Вам гарантируется, что входные данные содержат допустимое двоичное дерево. То есть это дерево, и каждый узел имеет не более двух ребенков.

- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.

- **Ограничения на входные данные.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

  Все ключи во входных данных различны.

- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

- Ограничение по времени. 10 сек.

- Ограничение по памяти. 512 мб.

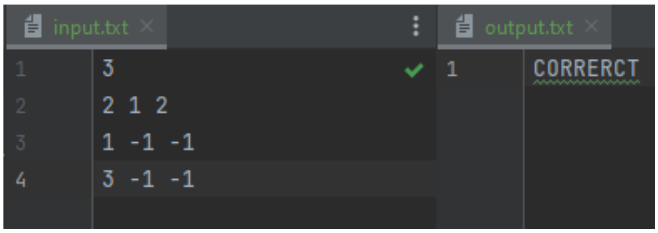
Для решения создается класс узла Node, атрибутами которого являются значения узла, правый и левый дочерние элементы, которые принимают либо значение None (если дочернего элемента нет), либо объект класса узла. Далее, в функции tree_check, которая принимает граничные значения для значения узла в качестве входных данных. Функция проверяет, существует ли узел, если нет, возвращается значение True, в противном случае она проверяет, выходит ли значение узла за переданные границы, если нет, функция вызывает себя с новыми границами для правого и левого дочернего элемента, если да, возвращается булеаново значение False. Новые границы - для левого дочернего элемента - минимальное значение для родительского элемента в качестве нижней границы и значение родительского элемента в качестве верхней границы, для правого дочернего элемента - значение родительского элемента в качестве нижней границы и максимальное значение для родительского элемента в качестве верхней границы. При вызове функции и передаче значения корня всего дерева

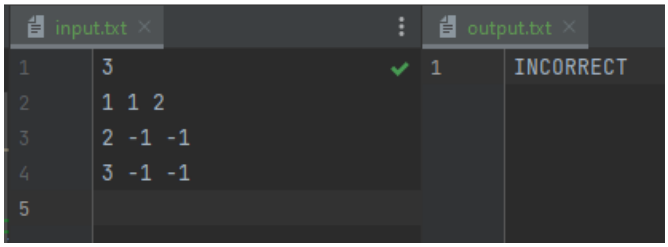
значения -10^9 и 10^9 передаются в качестве начальных границ в качестве максимальных значений для значения узла в соответствии с условием задачи.

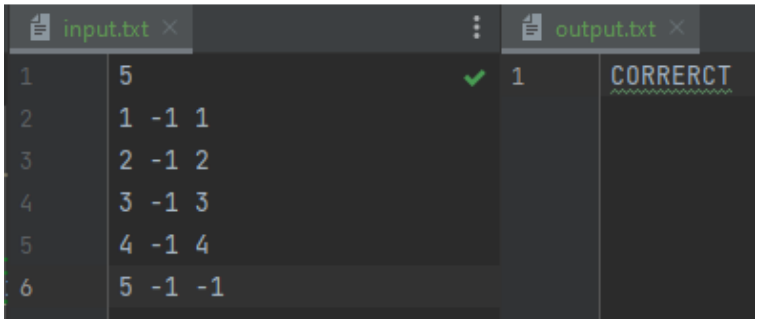
```
class Node:
    def __init__(self, line):
        self.key = int(line[0])
        self.left = None
        self.right = None


def tree_check(node, mini, maxi):
    if node == None:
        return True
    if node.key <= mini or node.key >= maxi:
        return False
    return tree_check(node.left, mini, node.key) and
tree_check(node.right, node.key, maxi)


f = open('input.txt')
w = open('output.txt', 'w')
t_start = time.perf_counter()
tracemalloc.start()
n = int(f.readline())
array = f.readlines()
f.close()
tree = []
if n == 0:
    res = True
else:
    for i in range(n):
        line = list(map(int, array[i].split()))
        node = Node(line)
        tree.append(node)

    for i in range(n):
        line = list(map(int, array[i].split()))
        if line[1] != -1:
            tree[i].left = tree[line[1]]
        if line[2] != -1:
            tree[i].right = tree[line[2]]
```

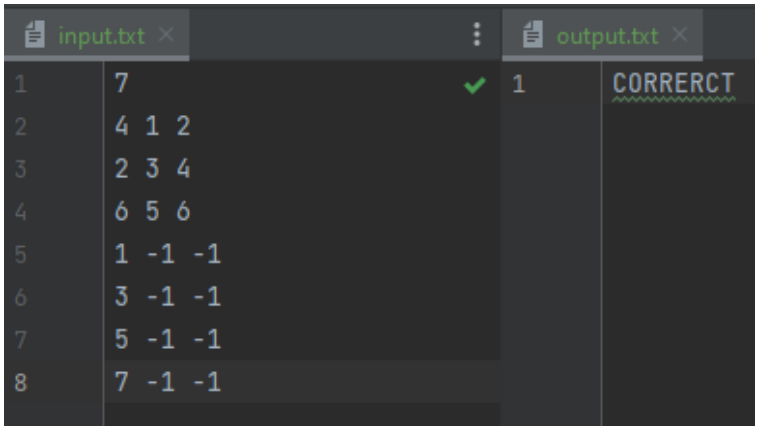
```
    res = tree_check(tree[0], -(2 ** 31), 2 ** 31)
if res:
    w.write('CORRERCT')
else:
    w.write('INCORRECT')
w.close()
```

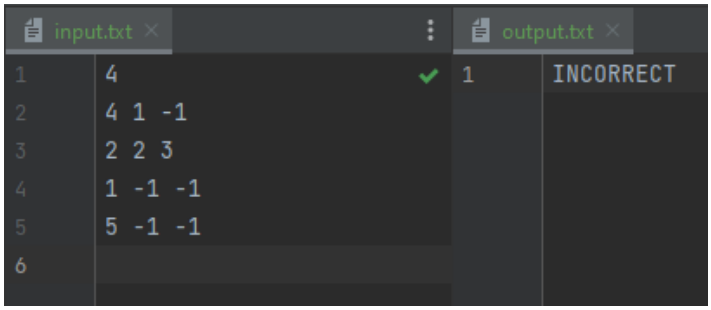
Результат работы кода на примере из текста задачи:











	Время выполнения, с	Затраты памяти, мб
Пример 1 из задачи	0.0004432000000000047	0.00847625732421875
Пример 2 из задачи	0.0004195000000000032	0.008448600769042969
Пример 3 из задачи	0.0003610000000000002	0.008263587951660156
Пример 4 из задачи	0.0004757000000000945	0.00859832763671875
Пример 5 из задачи	0.0005181999999999964	0.00872039794921875
Пример 6 из задачи	0.0004182999999999645	0.008509635925292969

Вывод по задаче: вроде все просто, просто берешь и проверяешь является ли данное дерево бинарным деревом поиска или нет.

Задача №7. Опознание двоичного дерева поиска (2.5 балла)

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.

- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.

- **Ограничения на входные данные.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла. Обратите внимание, что минимальное и максимальное возможные значения 32-битного целочисленного типа могут быть ключами в дереве.

- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

- Ограничение по времени. 10 сек.

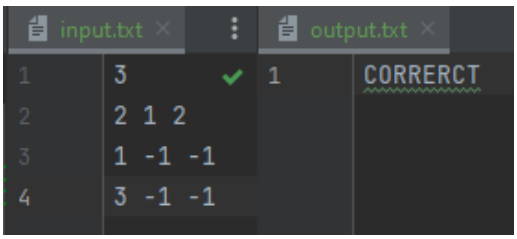
- Ограничение по памяти. 512 мб.

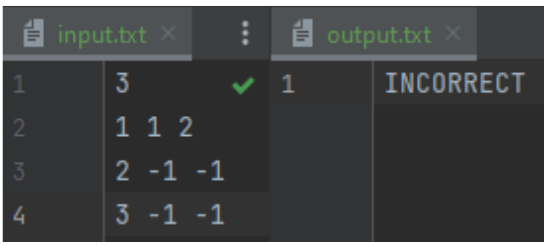
Решение аналогично заданию №6, изменилось лишь условие if node.key < mini и res = None перед заходом в цикл.

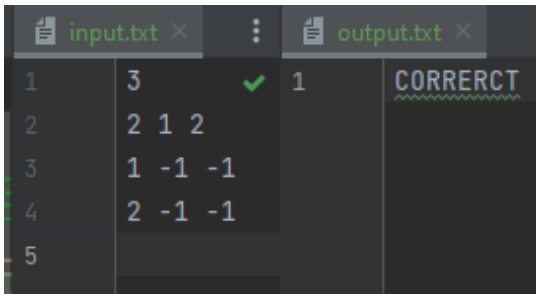
```
class Node:
    def __init__(self, line):
        self.key = int(line[0])
        self.left = None
        self.right = None


def tree_check(node, mini, maxi):
    if node == None:
        return True
    if node.key < mini or node.key >= maxi:
        return False
    return tree_check(node.left, mini, node.key) and
tree_check(node.right, node.key, maxi)


f = open('input.txt')
w = open('output.txt', 'w')
t_start = time.perf_counter()
```

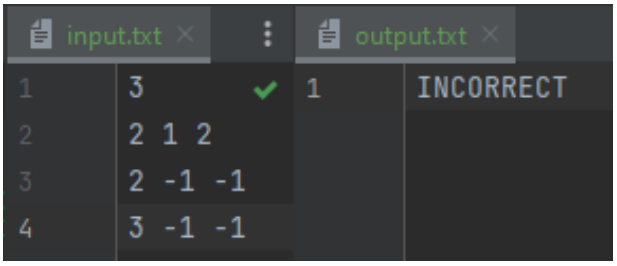
```
tracemalloc.start()
res = None
n = int(f.readline())
array = f.readlines()
f.close()
tree = []
if n == 0:
    res = True
else:
    for i in range(n):
        line = list(map(int, array[i].split()))
        node = Node(line)
        tree.append(node)

    for i in range(n):
        line = list(map(int, array[i].split()))
        if line[1] != -1:
            tree[i].left = tree[line[1]]
        if line[2] != -1:
            tree[i].right = tree[line[2]]

    res = tree_check(tree[0], -(2 ** 31), 2 ** 31)
if res:
    w.write('CORRERCT')
else:
    w.write('INCORRECT')
w.close()
```

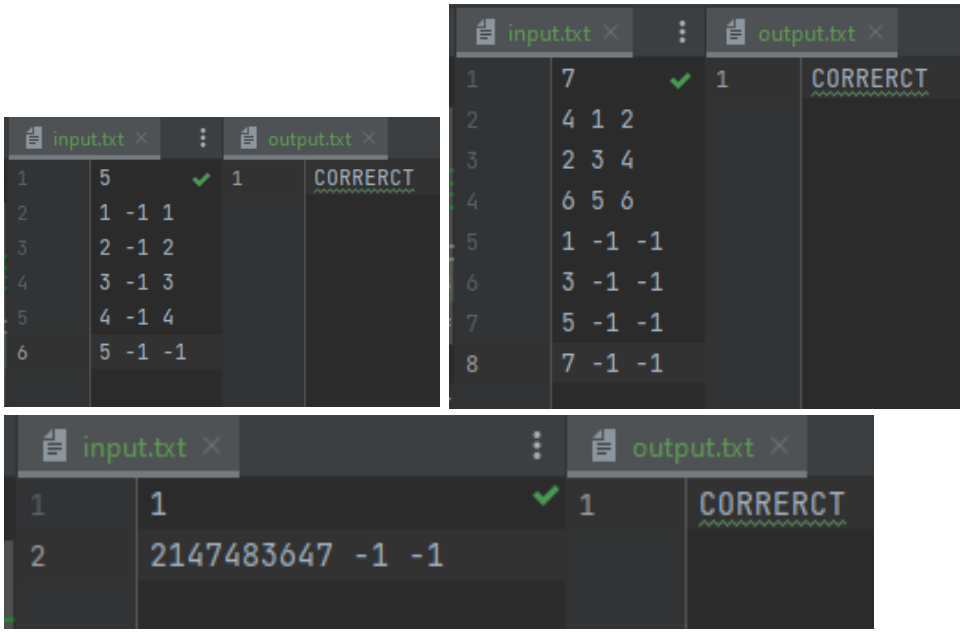
Результат работы кода на примере из текста задачи:











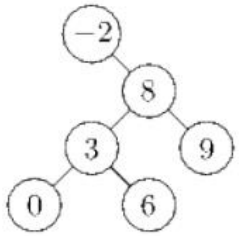
Пример 1 из задачи	0.00042680000000000495	0.00847625732421875
Пример 2 из задачи	0.00040579999999999783	0.00847625732421875
Пример 3 из задачи	0.00039979999999999877	0.008448600769042969
Пример 4 из задачи	0.0003939999999999916	0.00847625732421875
Пример 5 из задачи	0.0005281999999999995	0.00859832763671875
Пример 6 из задачи	0.0004829999999999973	0.00872039794921875
Пример 7 из задачи	0.00038390000000000646	0.008371353149414062

Вывод по задаче: наверное, задача сложная, если 6 была бы реализована как-то иначе, но тут мне было нужно было изменить пару строк.

Задача №8. Высота дерева возвращается (2 балла)

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.



Дано **двоичное дерево поиска**. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 10^9. 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 ключа вершины V.

Найдите высоту данного дерева.

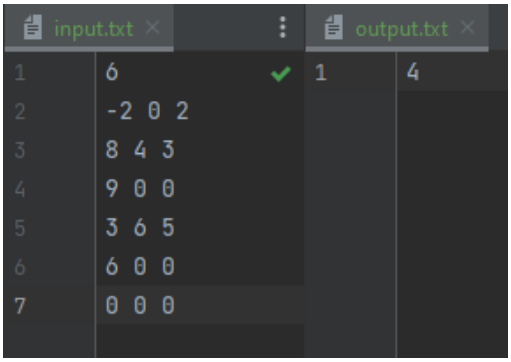
- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева. В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).
- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$. Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
- **Формат вывода / выходного файла (output.txt).** Выведите одно целое число – высоту дерева.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.

Создаётся класс узла Node, атрибутами которого являются значения узла, высота каждого узла (по умолчанию 0), правый и левый дочерние элементы, которые принимают либо значение None (если нет дочернего элемента) или объект класса Node. Далее создается массив tree, элементами которого являются объекты класса Node, не относящиеся к правым/левым потомкам, при повторном прохождении массива по индексам, заданным в условии задачи, объектам из древовидного массива присваиваются атрибуты left и right. Функция height принимает массив с узлами в качестве входных данных с конца, она проходит через все узлы дерева, записывая последовательно высоту каждого следующего дерева как сумму единицы и максимальную высоту дочерних элементов. В ответ записывается значение атрибута первого узла в массиве, поскольку он является корнем дерева, а его высота равна высоте дерева.

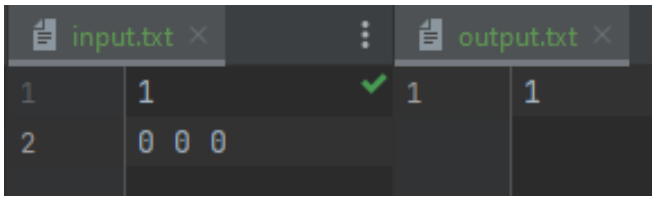
```
class Node:

    def __init__(self, line):
        self.height = 0
        self.key = int(line[0])
        self.left = None
        self.right = None


def height(array):
    for i in range(n - 1, -1, -1):
        if array[i].right == None and array[i].left
== None:
            array[i].height = 1
        elif array[i].right != None and array[i].left
== None:
            array[i].height = array[i].right.height +
1
        elif array[i].right == None and array[i].left
!= None:
            array[i].height = array[i].left.height +
1
        else:
            array[i].height =
max(array[i].left.height, array[i].right.height) + 1


f = open('input.txt')
w = open('output.txt', 'w')
n = int(f.readline())
array = f.readlines()
t_start = time.perf_counter()
tracemalloc.start()
tree = []
for i in range(n):
    line = list(map(int, array[i].split()))
    node = Node(line)
    tree.append(node)

for i in range(n):
    line = list(map(int, array[i].split()))
    if line[1] != 0:
        tree[i].left = tree[line[1] - 1]
```

```
    if line[2] != 0:
        tree[i].right = tree[line[2] - 1]
if n == 0:
    res = 0
elif n == 1:
    res = 1
else:
    height(tree)
    res = tree[0].height
w.write(str(res))
```

Результат работы кода на примере из текста задачи:



Результат работы кода на максимальном и минимальном значениях:



	Время выполнения, с	Затраты памяти, мб
Нижняя граница диапазона значений входных данных из текста задачи	6.350000000000106e-05	0.0017461776733398438
Пример из задачи	0.00012370000000000436	0.0034637451171875

Вывод по задаче: доработка задачи №7, здорово!

Задача №12. Проверка сбалансированности (2 балла)

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.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева.

  В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5, |K_i| \le 10^9$.

- **Формат вывода / выходного файла (output.txt).** Для i-ой вершины в i-ой строке выведите одно число – баланс данной вершины.

- **Ограничение по времени. 2 сек.**

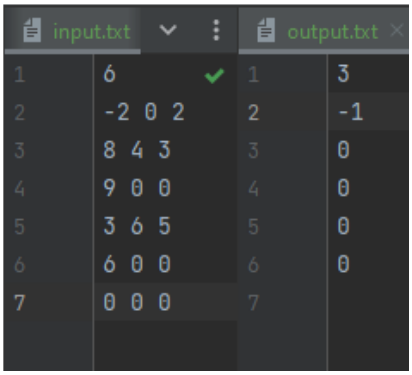
- Ограничение по памяти. 256 мб.

В этой задаче мне нужно было проверить дерево на сбалансированность. Дерево считается сбалансированным, если для любой вершины высота его левого поддерева отличается от высоты его правого поддерева не более чем на 1. Для начала я поискал родителей, рассчитал рост и непосредственно проверил равновесие.

```
def find_parents(array):
    for i in range(len(array)):
        if array[i][1]: array[array[i][1] - 1] += [i,
None]
        if array[i][2]: array[array[i][2] - 1] += [i,
None]
    array[0] += [None, None]
    return array


def count_height(array, i):
    height = 1
    while i:
        if (not array[i][4]) or (array[i][4] and
array[i][4] < height):
```

```
            array[i][4] = height
            height += 1
            i = array[i][3]
        else:
            break

    return array


def ballance_tree(array):
    for i in range(len(array)):
        if not array[i][1] and not array[i][2]:
            array = count_height(array, i)
    for i in array:
        left = 0
        right = 0
        if i[1]: left = array[i[1] - 1][4]
        if i[2]: right = array[i[2] - 1][4]
        w.write(str(right - left) + "\n")


f = open("input.txt")
w = open("output.txt", "w")
data = f.readlines()
t_start = time.perf_counter()
tracemalloc.start()
data = data[1:]
res = []
for i in data: res.append(list(map(int, i.split())))
ballance_tree(find_parents(res))
f.close()
w.close()
```

Результат работы кода на примере из текста задачи:



	Время выполнения, с	Затраты памяти, мб
Пример из задачи	0.00041940000000000033	0.0012884140014648438

Вывод по задаче: пример из текста задачи выполнен корректно.

Задача №15. Удаление из АВЛ-дерева (3 балла)

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 несбалансированная 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, являющегося 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.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется удалить из дерева.

  В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

  В последней строке содержится число X – ключ вершины, которую требуется удалить из дерева. Гарантируется, что такая вершина в дереве существует.

- **Ограничения на входные данные.** $1 \le N \le 2 \cdot 10^5, |K_i| \le 10^9, |X| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции удаления. Нумерация вершин может быть произвольной при условии соблюдения формата.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.

Решение данной задачи состоит из 8 функций. Функция count_height_for_every_node вычисляет высоту для каждой вершины дерева, функция height_difference возвращает разницу в высоте между

правым и левым дочерними элементами. Update_height – вычисляет высоту для вершины. Delete – удаляет узел. Right_turn – правый поворот дерева, left_turn – влево. Balance – уравновешивание дерева. Correct_tree – возвращает ответ в желаемом формате.

```
def count_height_for_every_node(arr):
    for i in range(n, 0, -1):
        arr[i].height = max(arr[arr[i].left].height,
arr[arr[i].right].height) + 1


def height_difference(arr, j):
    return arr[arr[j].left].height -
arr[arr[j].right].height


def update_height(arr, j):
    left = arr[j].left
    right = arr[j].right
    if left != 0:
        arr[left].height =
max(arr[arr[left].left].height,
arr[arr[left].right].height) + 1
    if right != 0:
        arr[right].height =
max(arr[arr[right].left].height,
arr[arr[right].right].height) + 1
    arr[j].height = max(arr[left].height,
arr[right].height) + 1


def right_turn(arr, i):
    j = arr[i].left
    p = arr[i].parent
    if i == arr[p].left:
        arr[p].left = j
    else:
        arr[p].right = j
    arr[i].left = arr[j].right
    arr[arr[i].left].parent = i
    arr[i].parent = j
    arr[j].right = i
    arr[j].parent = p
```

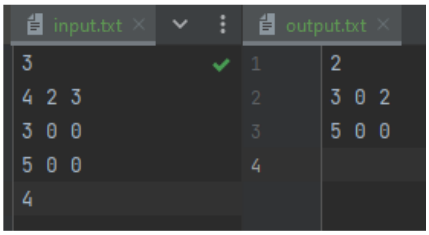
```
    update_height(arr, i)
    update_height(arr, j)
    return j


def left_turn(arr, i):
    j = arr[i].right
    p = arr[i].parent
    if i == arr[p].left:
        arr[p].left = j
    else:
        arr[p].right = j
    arr[i].right = arr[j].left
    arr[arr[i].right].parent = i
    arr[i].parent = j
    arr[j].left = i
    arr[j].parent = p

    update_height(arr, i)
    update_height(arr, j)
    return j


def balance(arr, i):
    j = i
    while j != 0:
        update_height(arr, j)
        if height_difference(arr, j) == 2:
            if height_difference(arr, arr[j].left) <
0:
                left_turn(arr, arr[j].left)
            j = right_turn(arr, j)
        elif height_difference(arr, j) == -2:
            if height_difference(arr, arr[j].right) >
0:
                right_turn(arr, arr[j].right)
            j = left_turn(arr, j)
        c = j
        j = arr[j].parent
    return c
```

```
def correct_tree(arr, a):
    k = 2
    r = [a]
    res = [0]
    while (len(r)) > 0:
        c = r
        r = []
        for i in range(len(c)):
            if arr[c[i]].left > 0:
                r.append(arr[c[i]].left)
                l = k
                k += 1
            else:
                l = 0
            if arr[c[i]].right > 0:
                r.append(arr[c[i]].right)
                r1 = k
                k += 1
            else:
                r1 = 0
            res.append(Node(arr[c[i]].key, l, r1, 0))

    return res


def delete(a, x, m):
    if len(a) != 2:
        i = m
        while True:
            if a[i].key == x:
                if a[i].left == a[i].right == 0:
                    if a[a[i].parent].left == i:
                        a[a[i].parent].left = 0
                    else:
                        a[a[i].parent].right = 0
                    m = a[i].parent
                elif a[i].left == 0:
                    if a[a[i].parent].left == i:
                        a[a[i].parent].left =
a[i].right
                        a[a[i].right].parent =
a[i].parent
                        m = a[i].right
```

```
                    else:
                        a[a[i].parent].right =
a[i].right
                        a[a[i].right].parent =
a[i].parent
                        m = a[i].right
                else:
                    j = a[i].left
                    while a[j].right > 0:
                        j = a[j].right
                    a[i].key = a[j].key
                    if a[a[j].parent].left == j:
                        a[a[j].parent].left =
a[j].left
                        a[a[j].left].parent =
a[j].parent
                    else:
                        a[a[j].parent].right =
a[j].left
                        a[a[j].right].parent =
a[j].parent
                    m = a[j].parent
                break
            elif x < a[i].key and a[i].left > 0:
                i = a[i].left
            elif x > a[i].key and a[i].right > 0:
                i = a[i].right
    return balance(a, m)


class Node:
    def __init__(self, key, left, right, parent,
height=-1):
        self.key = key
        self.left = left
        self.right = right
        self.parent = parent
        self.height = height


f = open('input.txt')
n = int(f.readline())
t_start = time.perf_counter()
```

```
tracemalloc.start()
arr = []
for i in range(n + 1):
    arr.append(Node(0, 0, 0, 0))
for i in range(1, n + 1):
    arr[i].key, arr[i].left, arr[i].right = map(int,
f.readline().split())
    if arr[i].left > 0:
        arr[arr[i].left].parent = i
    if arr[i].right > 0:
        arr[arr[i].right].parent = i
count_height_for_every_node(arr)
x = int(f.readline())
a = delete(arr, x, 1)
w = open('output.txt', 'w')
w.write(str(n - 1) + '\n')
if n - 1 > 0:
    arr = correct_tree(arr, a)
    for i in range(1, n):
        w.write(str(arr[i].key) + ' ' +
str(arr[i].left) + ' ' + str(arr[i].right) + '\n')
w.close()
f.close()
```

Результат работы кода на примере из текста задачи:



	Время выполнения, с	Затраты памяти, мб
Пример из задачи	0.001042300000000003	0.012637138366699219

Вывод по задаче: мне очень жаль, что задачу нельзя протестить задачу на OpenEdu, потому что про оптимизацию ничего толком сказать не могу. Пример из текста задачи выполнен корректно.

Задача №17. Множество с суммой (3 балла)

В этой задаче ваша цель – реализовать структуру данных для хранения набора целых чисел и быстрого вычисления суммы элементов в заданном диапазоне.

Реализуйте такую структуру данных, в которой хранится набор целых чисел S и доступны следующие операции:

- add(i) – добавить число i в множество S. Если i уже есть в S, то ничего делать не надо;
- del(i) – удалить число i из множества S. Если i нет в S, то ничего делать не надо;
- find(i) – проверить, есть ли i во множестве S или нет;
- sum(l, r) – вывести сумму всех элементов v из S таких, что $l \le v \le r$.

- **Формат ввода / входного файла (input.txt).** Изначально множество S пусто. Первая строка содержит n – количество операций. Следующие n строк содержат операции. Однако, чтобы убедиться, что ваше решение может работать в режиме онлайн, каждый запрос фактически будет зависеть от результата последнего запроса суммы. Обозначим $M = 1\,000\,000\,001$. В любой момент пусть x будет результатом последней операции суммирования или просто 0, если до этого операций суммирования не было. Тогда каждая операция будет являться одной из следующих:
  - «+ i» – добавить некое число в множество S. Но не само число i, а число $((i + x) \bmod M)$.
  - «– i» – удалить из множества S, т.е. del$((i + x) \bmod M)$.
  - «? i» – find$((i + x) \bmod M)$.
  - «s l r» – вывести сумму всех элементов множества S из определенного диапазона, т.е. sum$((l+x) \bmod M, (r+x) \bmod M)$.
- **Ограничения на входные данные.** $1 \le n \le 100000$, $1 \le i \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса «find», выведите только «Found» или «Not found» (без кавычек, первая буква заглавная) в зависимости от того, есть ли число $((i + x) \bmod M)$ в S или нет.

  Для каждого запроса суммы «sum» выведите сумму всех значений v из S из диапазона $(l + x) \bmod M \le v \le (r + x) \bmod M$, где x – результат подсчета прошлой суммы «sum», или 0, если еще не было таких операций.
- **Ограничение по времени. 120 сек. Python**
- Ограничение по памяти. 512 мб.

Реализована структура данных, которая сохраняет набор целых чисел и проводит с ними некоторые операции. give_parent – передача родителю, keep_parent – сохраняет родителя, rotation – поворот, splay – сращивание, find – поиск, split – разделение, insert – вставка, merge – объединение, remove – удаление, sum – суммирование.

```
from collections import deque


class sNode:
    def __init__(self, key, left=None, right=None,
parent=None):
        self.left = left
        self.right = right
        self.parent = parent
        self.key = key


class sTree:
```

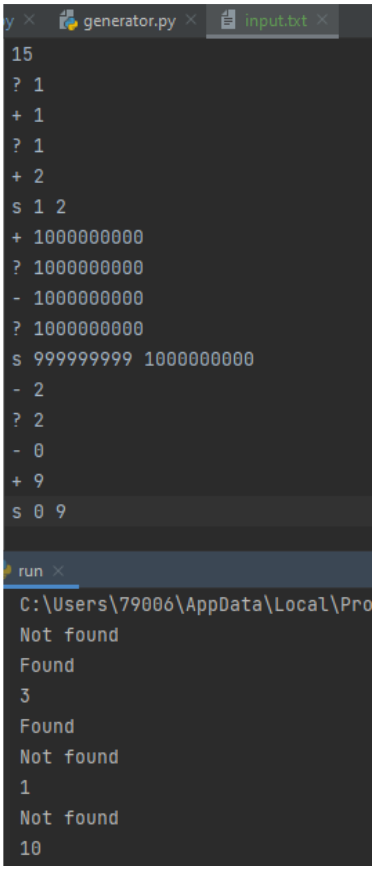
```
    def give_parent(self, child, parent):
        if child != None:
            child.parent = parent

    def keep_parent(self, node):
        self.give_parent(node.left, node)
        self.give_parent(node.right, node)

    def rotation(self, parent, child):
        gparent = parent.parent
        if gparent != None:
            if gparent.left == parent:
                gparent.left = child
            else:
                gparent.right = child
        if parent.left == child:
            parent.left, child.right = child.right,
parent
        else:
            parent.right, child.left = child.left,
parent
        self.keep_parent(child)
        self.keep_parent(parent)
        child.parent = gparent

    def splay(self, node):
        if node.parent == None:
            return node
        parent = node.parent
        gparent = parent.parent
        if gparent == None:  # если нет прародителя
делаем поворот
            self.rotation(parent, node)
            return node
        else:
            z = (gparent.left == parent) ==
(parent.left == node)
            if z:
                self.rotation(gparent, parent)  #
поворот прародителя относительно родителя
                self.rotation(parent, node)  #
поворот родителя относительно вершины
            else:
```

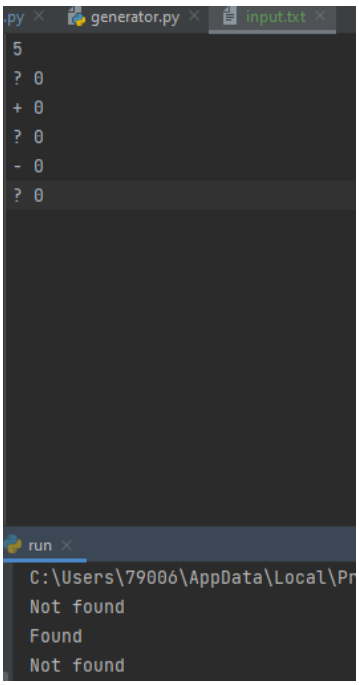
```
                self.rotation(parent, node)
                self.rotation(gparent, node)
        return self.splay(node)

    def find(self, node, key):
        if node == None:
            return None
        if key == node.key:  # node.key - число i
            return self.splay(node)
        if key < node.key and node.left != None:
            return self.find(node.left, key)
        if key > node.key and node.right != None:
            return self.find(node.right, key)
        return self.splay(node)

    def split(self, root, key):  # разделение для
вставки, чтобы освободить место
        if root == None:
            return None, None
        root = self.find(root, key)
        if root.key == key:
            self.give_parent(root.left, None)
            self.give_parent(root.right, None)
            return root.left, root.right
        if root.key < key:
            right, root.right = root.right, None
            self.give_parent(right, None)
            return root, right
        else:
            left, root.left = root.left, None
            self.give_parent(left, None)
            return left, root

    def insert(self, root, key):
        left, right = self.split(root, key)
        root = sNode(key, left, right)
        self.keep_parent(root)
        return root

    def merge(self, left, right):
        if right == None:
            return left
        if left == None:
```

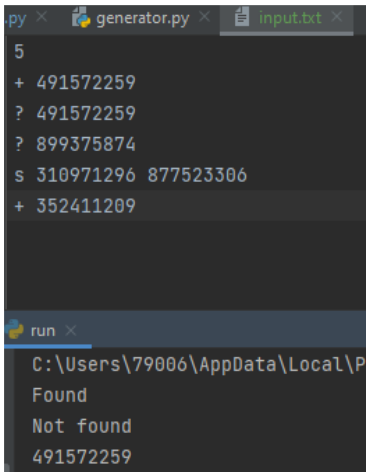
```
            return right
        right = self.find(right, left.key)
        right.left, left.parent = left, right
        return right

    def remove(self, root, key):
        root = self.find(root, key)
        self.give_parent(root.left, None)
        self.give_parent(root.right, None)
        return self.merge(root.left, root.right)

    def sum(self, root, l, r):
        stack = deque()
        list_of_num = []
        if (root == None):
            return 0
        while (True):
            stack.append(root)
            if (root.left == None) or (root.key <=
l):
                break
            root = root.left
        while (True):
            if (len(stack) != 0):
                list_of_num.append(stack[-1].key)
                if (stack[-1].right != None) and
(stack[-1].key <= r):
                    root = stack.pop().right
                else:
                    stack.pop()
                    continue
                while (True):
                    stack.append(root)
                    if (root.left == None) or
(root.key <= l):
                        break
                    root = root.left
            if (len(stack) == 0):
                summa = 0
                for num in list_of_num:
                    if ((num >= l) and (num <= r)):
                        summa += num
                return summa
```

```
k = 1000000001
postop = 0
f = open("input.txt")
n = int(f.readline())
t_start = time.perf_counter()
tracemalloc.start()
root = None
tree = sTree()
for i in range(n):
    com = f.readline().split()
    num = (int(com[1]) + postop) % k
    if (com[0] == "?"):
        root = tree.find(root, num)
        if (root != None) and (root.key == num):
            print("Found")
        else:
            print("Not found")
    elif (com[0] == "+"):
        if (root == None):
            root = sNode(num)
        else:
            root = tree.insert(root, num)
    elif (com[0] == "-"):
        if (root != None):
            root = tree.remove(root, num)
    elif (com[0] == "s"):
        num_s = (int(com[2]) + postop) % k
        postop = tree.sum(root, num, num_s)
        print(f"{postop}")
f.close()
```

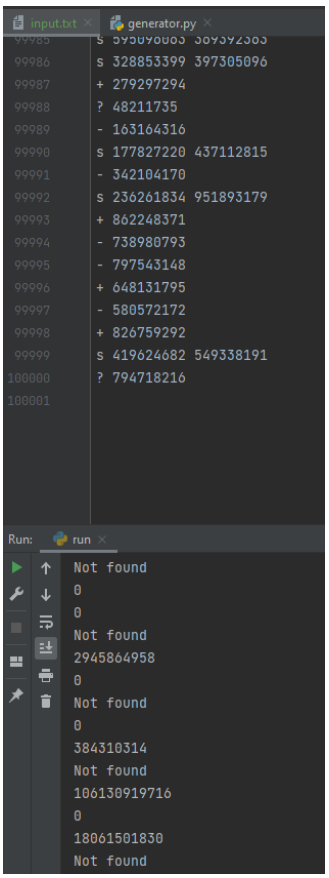
Результат работы кода на примере из текста задачи:

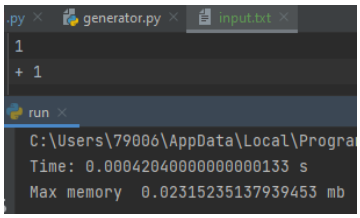






Результат работы кода на максимальном и минимальном значениях:





	Время выполнения, с	Затраты памяти, мб
Нижняя граница диапазона значений входных данных из текста задачи	0.00010499999999999399	0.009385108947753906
Пример 1 из задачи	0.0011897000000000019	0.02752971649169922
Пример 2 из задачи	0.00041439999999999533	0.024941444396972656
Пример 3 из задачи	0.0005582000000000018	0.025015830993652344
Верхняя граница диапазона значений входных данных из текста задачи	4.3100127359	1.1649169932

Вывод по задаче: очень сложная задача, с которой я просидела несколько часов, но в итоге тесты выполнены успешно.

Вывод

Через меня пройдены новые структуры данных, бинарные деревья поиска и сбалансированные деревья поиска. Забавно, что задачи в прошлых лабораторных работах мне казались сложными, но каждый раз лабораторные всё круче и круче – а я справилась.