Инфокоммуникационные технологии и системы связи 11.03.02

Программирование в инфокоммуникационных системах

Лабораторная работа #2

Выполнил: Коркина Анна Михайловна

Группа: К3120

Преподаватель: Харьковская Татьяна Александровна

г. Санкт-Петербург

2022

1. Задача. Обход двоичного дерева [5 s, 512 Mb, 1 балл]

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), 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 (post-order) обходы в глубину.

• Формат ввода: стандартный ввод или 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 Ki, Li и Ri. Ki – ключ i-го узла, Li - индекс левого ребенка i-го узла, а Ri - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа Li или Ri (или оба) будут равны −1.

• Ограничения на входные данные. 1 ≤ n ≤ 105, 0 ≤ Ki ≤ 109, −1 ≤ Li, Ri ≤ n−1. Гарантируется, что данное дерево является двоичным деревом. В частности, если Li = −1 и Ri = −1, то Li = Ri. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

• Формат вывода / выходного файла (output.txt). Выведите три строки. Первая строка должна содержать ключи узлов при центрированном обходе дерева (in-order). Вторая строка должна содержать ключи узлов при прямом обходе дерева (pre-order). Третья строка должна содержать ключи узлов при обратном обходе дерева (post-order).

• Ограничение по времени. 5 сек.

• Ограничение по памяти. 512 мб.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []
    K = []
    L = []
    R = []

    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        node = BSTNode(Ki)
        nodes.append(node)
        K.append(int(Ki))
```

```
        L.append(int(Li))
        R.append(int(Ri))

    for i in range(n):
        l = L[i]
        r = R[i]
        if l != -1:
            nodes[i].left = nodes[l]
        if r != -1:
            nodes[i].right = nodes[r]
    fin.close()
    return nodes


def file_output(inorder, preorder, postorder):
    fout = open('output.txt', 'w')
    for num in inorder:
        fout.write(num + ' ')
    fout.write('\n')
    for num in preorder:
        fout.write(num + ' ')
    fout.write('\n')
    for num in postorder:
        fout.write(num + ' ')
    fout.close()

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

    def preorder(self, vals):
        if self.key is not None:
            vals.append(self.key)
        if self.left is not None:
            self.left.preorder(vals)
        if self.right is not None:
            self.right.preorder(vals)
        return vals

    def inorder(self, vals):
        if self.left is not None:
            self.left.inorder(vals)
        if self.key is not None:
            vals.append(self.key)
        if self.right is not None:
            self.right.inorder(vals)
        return vals

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

def main():
```

```
    nodes = file_input()
    preorder = nodes[0].preorder([])
    inorder = nodes[0].inorder([])
    postorder = nodes[0].postorder([])

    file_output(inorder, preorder, postorder)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Метод **insert (self, key: int)** вставляет элемент дерева через объект, являющийся корнем всего дерева.

Метод **inorder (self, vals)** обходит дерево центрировано**.**

Метод **preorder (self, vals)** совершает прямой обход дерева**.**

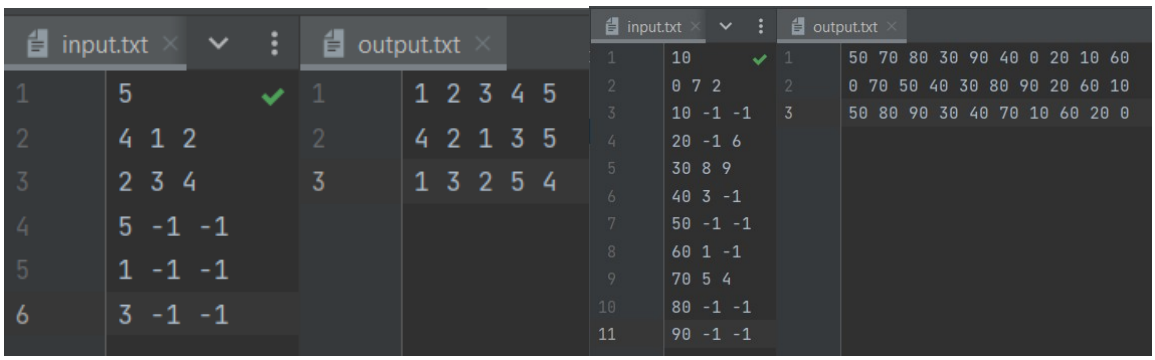
Метод **postorder (self, vals)** совершает обратный обход дерева**.**

Обходы дерева происходят рекурсивно.

Принцип работы:

Программа сначала получает на вход все строки и сохраняет соответствующие значения K, L и R в массивы и параллельно создаёт узлы без ссылок на потомков. Потом, проходя по всем ключам, происходит присваивание узлов потомков каждому узлу соответственно. Далее совершаются рекурсивные проходы по полученному дереву.

Тестовые данные:



Вывод: задача решена с помощью структуры бинарного дерева поиска. Все операции, проводимые с бинарным деревом, осуществлялись посредством методов класса узла, а через методы объекта корня всего дерева эти операции и производятся.

2. Задача. Гирлянда [2 s, 256 Mb, 1 балл]

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.

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. Подсказка: для решения этой задачи можно использовать двоичный поиск.

Код:

```
error = 0.1 ** 10


def file_input():
    fin = open('input.txt', 'r')
    n, A = fin.readline().split()
    fin.close()
    return int(n), float(A)


def file_output(result):
    fout = open('output.txt', 'w')
    fout.write("%.6f" % result)
    fout.close()


def equal(a, b):
    return (abs(a - b) <= error)


def less(a, b):
    return (a < b) and (not equal(a, b))


def more(a, b):
    return (a > b) and (not equal(a, b))
```

```
def search_height(left, right, n):
    dots = [0] * n
    dots[0] = right
    result = 1000000000
    while less(left, right):
        dots[1] = (left + right) / 2
        dots[-1] = 0
        is_up = False
        for i in range(2, n):
            dots[i] = 2 * dots[i - 1] - dots[i - 2] + 2
            if (not more(dots[i], 0)):
                is_up = True
                break
        if more(dots[-1], 0):
            result = min(result, dots[-1])
        if is_up:
            left = dots[1]
        else:
            right = dots[1]
    return result


def main():
    n, H = file_input()
    result = search_height(0, H, n)
    file_output(result)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Функции **equal, less, more** производят сравнения двух чисел с учётом погрешности.

Метод **search_height (left, right, n)** находит минимально возможную высоту второго конца гирлянды.

Тестовые данные:

Примеры

input.txt	output.txt
8 15	9.75
692 532.81	446113.34434782615

Choose Files No file chosen

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.093	9244672	14	13
1	OK	0.015	9195520	9	8
2	OK	0.031	9150464	12	13

Вывод: задача решена с помощью бинарного поиска. При получении отрицательных значений узлов гирлянды левая граница смещается обратно к значению dots[1]. При получении положительных значений края гирлянды они сравниваются с уже полученным минимумом, тем самым находя наименьшее значение высоты.

3. Задача. Простейшее BST [2 s, 256 Mb, 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. $1 \le x \le 109$, $1 \le N \le 300000$

• 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.

Код:

```
def file_input():
    fin = open('input.txt', 'r')
    lines = []

    while True:
        line = fin.readline()
        if line:
            lines.append(line.replace('\n', ''))
        else:
            break
    fin.close()

    return lines


def file_output(results):
    fout = open('output.txt', 'w')
    for num in results:
        fout.write(str(num) + '\n')
    fout.close()









class BSTNode:

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

    def insert(self, key: int):
        if not self.key:
            self.key = key
            return
        if self.key == key:
            return

        if key < self.key:
            if self.left:
                self.left.insert(key)
            self.left = BSTNode(key, self)

        if self.right:
            self.right.insert(key)
            return
        self.right = BSTNode(key, self)


    def inorder(self, results, num):
        if self.left is not None:
            self.left.inorder(results, num)
        if self.key is not None:
```

```
            if self.key > num:
                results.append(self.key)
        if self.right is not None:
            self.right.inorder(results, num)
        return results


def main():
    lines = file_input()
    results = []
    bst = BSTNode()
    for line in lines:
        num = int(line[2:])
        if line[0] == '+':
            bst.insert(num)
        elif line[0] == '>':
            values = bst.inorder([], num)
            if values:
                results.append(min(values))
            else:
                results.append(0)

    file_output(results)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

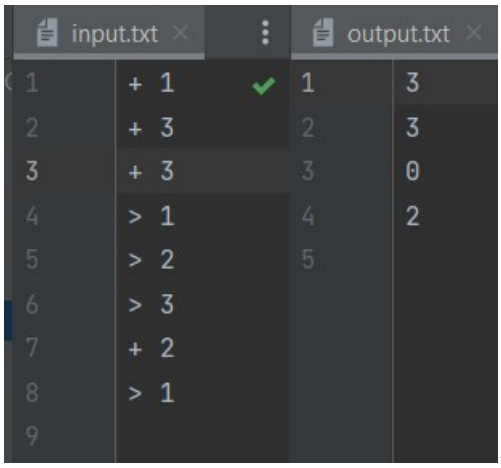
Класс **BSTNode** необходим для хранения узла дерева.

Метод **insert (self, key: int)** вставляет элемент дерева через объект, являющийся корнем всего дерева.

Метод **inorder (self, results, num)** обходит дерево с минимального элемента по максимальный и находит первый элемент, больший **num.**

Обход дерева и вставка элемента в него происходят рекурсивно.

Тестовые данные:



Вывод: задача решена с помощью структуры бинарного дерева поиска. Все операции, проводимые с бинарным деревом, осуществлялись посредством методов класса узла, а через методы объекта корня всего дерева эти операции и производятся.

5. Задача. Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл]

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. В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 109.

• Ограничения на входные данные. 0 ≤ N ≤ 100, |xi| ≤ 109

• Формат вывода / выходного файла (output.txt). Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.

• Ограничение по времени. 2 сек.

• Ограничение по памяти. 512 мб.

```
def file_input():
    fin = open('input.txt', 'r')
    lines = []

    while True:
        line = fin.readline()
        if line:
            lines.append(line.replace('\n', ''))
        else:
            break
    fin.close()

    return lines


def file_output(results):
    fout = open('output.txt', 'w')
    for num in results:
        fout.write(str(num) + '\n')
    fout.close()


class BSTNode:

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

    def insert(self, key: int):
        if not self.key:
            self.key = key
            return
        if self.key == key:
            return

        if key < self.key:
```

```
            if self.left:
                self.left.insert(key)
            self.left = BSTNode(key, self)

        if self.right:
            self.right.insert(key)
            return
        self.right = BSTNode(key, self)

    def delete(self, val):
        if self == None:
            return self
        if val < self.key:
            self.left = self.left.delete(val)
            return self
        if val > self.key:
            self.right = self.right.delete(val)
            return self
        if self.right == None:
            return self.left
        if self.left == None:
            return self.right
        min_larger_node = self.right
        while min_larger_node.left:
            min_larger_node = min_larger_node.left
        self.key = min_larger_node.key
        self.right = self.right.delete(min_larger_node.key)
        return self

    def exists(self, val):
        if val == self.key:
            return True

        if val < self.key:
            if self.left == None:
                return False
            return self.left.exists(val)

        if self.right == None:
            return False
        return self.right.exists(val)

    def get_min(self):
        current = self
        while current.left is not None:
            current = current.left
        return current

    def get_max(self):
        current = self
        while current.right is not None:
            current = current.right
        return current

    def next(self, results, num):
        if self.left is not None:
            self.left.next(results, num)
        if self.key is not None:
            if self.key > num:
                results.append(self.key)
        if self.right is not None:
```

```
            self.right.next(results, num)
        if results:
            return min(results)
        else:
            return None

    def prev(self, results, num):
        if self.left is not None:
            self.left.prev(results, num)
        if self.key is not None:
            if self.key < num:
                results.append(self.key)
        if self.right is not None:
            self.right.prev(results, num)
        if results:
            return max(results)
        else:
            return None


def main():
    lines = file_input()
    results = []
    bst = BSTNode()
    for line in lines:
        index = line.index(' ')
        num = int(line[index + 1:])
        if line[0] == 'i':
            bst.insert(num)
        elif line[0] == 'd':
            bst.delete(num)
        elif line[0] == 'e':
            value = bst.exists(num)
            if value:
                results.append("true")
            else:
                results.append("false")
        elif line[0] == 'n':
            value = bst.next([], num)
            if value is not None:
                results.append(value)
            else:
                results.append('none')
        elif line[0] == 'p':
            value = bst.prev([], num)
            if value is not None:
                results.append(value)
            else:
                results.append('none')

    file_output(results)


main()
```

Описание решения:

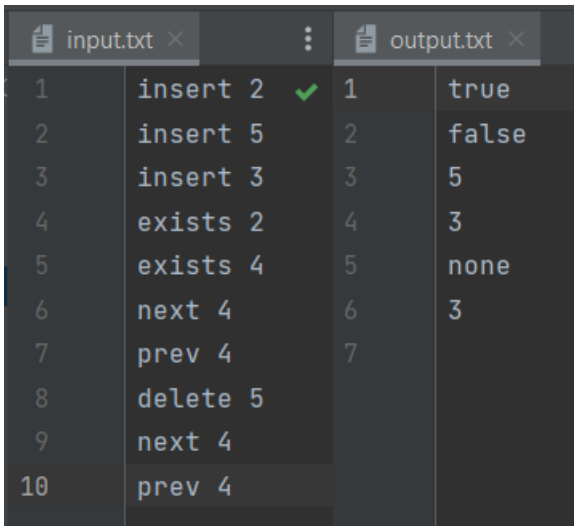
Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Метод **insert –** вставка, **delete** – удаление, **get_min и get_max** – получение минимума и максимума, **next** и **prev** – следующий или предыдущий элемент относительно x, **exists** – проверка на то содержит ли дерево x.

Тестовые данные:



6. Задача. Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 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 .

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []
    K = []
    L = []
    R = []

    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        node = BSTNode(int(Ki))
        nodes.append(node)
        K.append(int(Ki))
        L.append(int(Li))
        R.append(int(Ri))

    for i in range(n):
        l = L[i]
        r = R[i]
        if l != -1:
            nodes[i].left = nodes[l]
        if r != -1:
            nodes[i].right = nodes[r]
    fin.close()
    return nodes


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




def file_output(result):
    fout = open('output.txt', 'w')
    if result:
        fout.write("CORRECT")
    else:
        fout.write("INCORRECT")
    fout.close()


def check(root: BSTNode):
    if root == None:
        return None
    que = []
    que.append((root, -1000000000, 10000000000))
    while len(que) > 0:
        line = que.pop(0)
        node = line[0]
        if node.left != None:
            if node.left.key >= node.key or node.left.key >= line[2] or
```

```
node.left.key <= line[1]:
                return False
            que.append((node.left, line[1], node.key))
        if node.right != None:
            if node.right.key <= node.key or node.right.key >= line[2] or
node.right.key <= line[1]:
                return False
            que.append((node.right, node.key, line[2]))
    return True


def main():
    nodes = file_input()
    if nodes:
        result = check(nodes[0])
    else:
        result = True
    file_output(result)


main()
```

Описание решения:

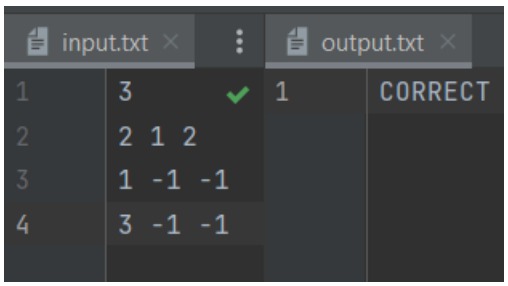
Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

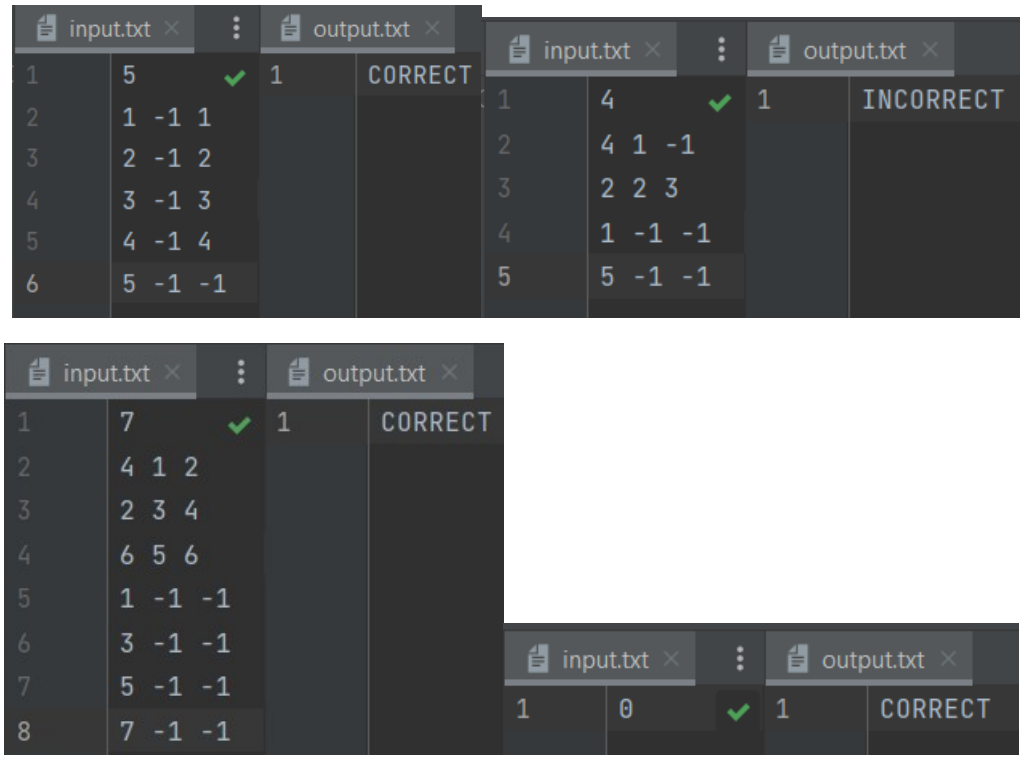
Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Функция **check** проверяет является ли введённое дерево правильным бинарным деревом поиска.

Тестовые данные:





7. Задача. Опознание двоичного дерева поиска (усложненная версия) [10 s,512 Mb, 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.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []
    K = []
    L = []
    R = []

    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        node = BSTNode(int(Ki))
        nodes.append(node)
        K.append(int(Ki))
        L.append(int(Li))
        R.append(int(Ri))

    for i in range(n):
        l = L[i]
        r = R[i]
        if l != -1:
            nodes[i].left = nodes[l]
        if r != -1:
            nodes[i].right = nodes[r]
    fin.close()
    return nodes


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




def file_output(result):
    fout = open('output.txt', 'w')
    if result:
        fout.write("CORRECT")
    else:
        fout.write("INCORRECT")
    fout.close()


def check(root: BSTNode):
    if root == None:
        return None
    que = []
```

```
    que.append((root, -1000000000, 10000000000))
    while len(que) > 0:
        line = que.pop(0)
        node = line[0]
        if node.left != None:
            if node.left.key >= node.key or node.left.key >= line[2] or
node.left.key <= line[1]:
                return False
            que.append((node.left, line[1], node.key))
        if node.right != None:
            if node.right.key <= node.key or node.right.key >= line[2] or
node.right.key <= line[1]:
                return False
            que.append((node.right, node.key, line[2]))
    return True


def main():
    nodes = file_input()
    if nodes:
        result = check(nodes[0])
    else:
        result = True
    file_output(result)


main()
```

Описание решения:

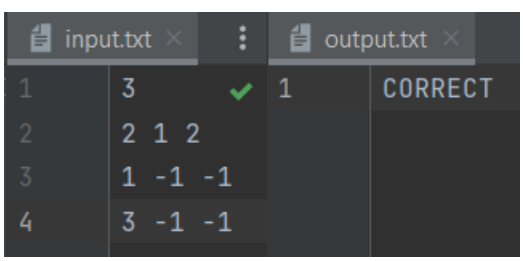
Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

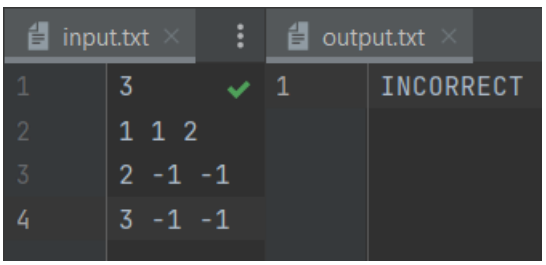
Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

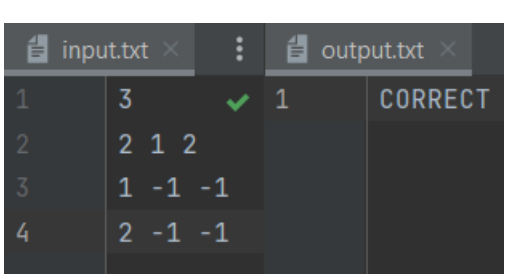
Класс **BSTNode** необходим для хранения узла дерева.

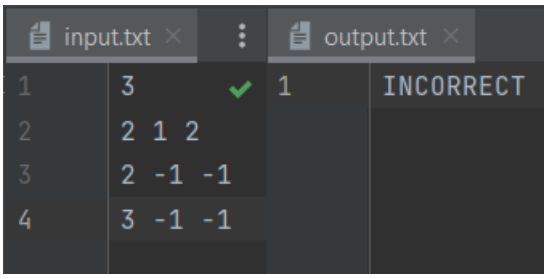
Функция **check** проверяет является ли введённое дерево правильным бинарным деревом поиска (теперь учитывая равные узлы).

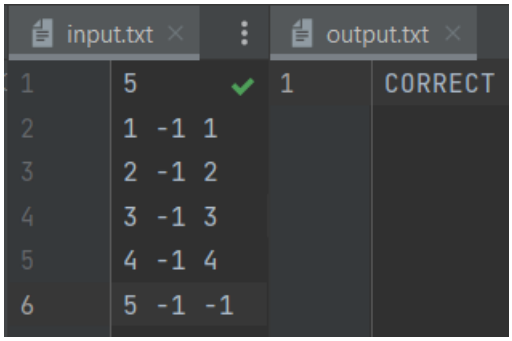
Тестовые данные:

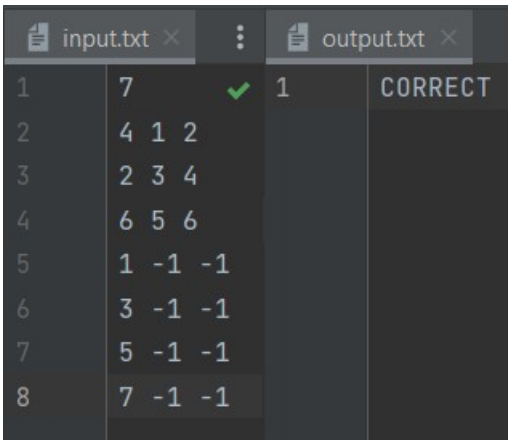


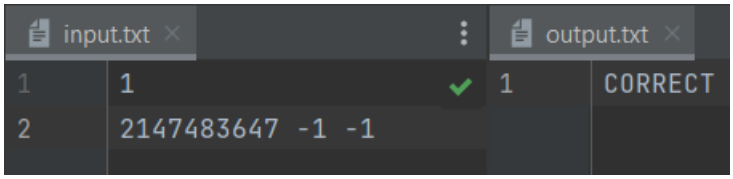












8. Задача. Высота дерева возвращается [2 s, 256 Mb, 2 балла]

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 109. 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.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []

    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        nodes.append((int(Li), int(Ri)))

    fin.close()
    return nodes


def file_output(result):
    fout = open('output.txt', 'w')
    fout.write(str(result))
    fout.close()

def find_height(nodes):
    deeps = [0] * (len(nodes) + 1)
    for i in range(len(nodes) - 1, -1, -1):
        if (nodes[i][0] == 0) and (nodes[i][1] == 0):
            deeps[i + 1] = 1
        else:
            deeps[i + 1] = max(deeps[nodes[i][0]], deeps[nodes[i][1]]) + 1
    return deeps[1]

def main():
    nodes = file_input()
    if nodes:
        result = find_height(nodes)
        file_output(result)
    else:
        file_output(0)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Функция **find_height** предназначена для поиска высоты дерева.

Тестовые данные:

Пример

input.txt	output.txt
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	4

Примечание
Во входном файле задано то же дерево, что и изображено на рисунке.

Choose Files No file chosen

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.343	49967104	3989144	6
1	OK	0.031	9084928	46	1

9. Задача. Удаление поддеревьев [2 s, 256 Mb, 2 балла]

Дано некоторое двоичное дерево поиска. Также даны запросы на удаление из него вершин, имеющих заданные ключи, причем вершины удаляются целиком вместе со своими поддеревьями. После каждого запроса на удаление выведите число оставшихся вершин в дереве. В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие 109. Гарантируется, что данное дерево является двоичным деревом поиска, в частности, 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 .

Высота дерева не превосходит 25, таким образом, можно считать, что оно сбалансировано.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []
    K = []
    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        K.append(int(Ki))
        nodes.append((int(Li), int(Ri)))

    n = int(fin.readline())
    deletes = fin.readline().split()
    deletes = [int(num) for num in deletes]
    fin.close()
    return K, nodes, deletes


def file_output(results):
    fout = open('output.txt', 'w')
    for num in results:
        fout.write(str(num) + '\n')
    fout.close()


def delete(keys, nodes, index):
    if keys[index] is not None:
        keys[index] = None

        if (nodes[index][0] == 0) and (nodes[index][1] == 0):
            return -1
        elif (nodes[index][0] != 0) and (nodes[index][1] == 0):

            if keys[nodes[index][0] - 1] is not None:
                return delete(keys, nodes, nodes[index][0] - 1) - 1
            else:
                return -1

        elif (nodes[index][1] != 0) and (nodes[index][0] == 0):

            if keys[nodes[index][1] - 1] is not None:
```

```
                return delete(keys, nodes, nodes[index][1] - 1) - 1
            else:
                return -1

        else:

            if keys[nodes[index][1] - 1] is not None or keys[nodes[index]
[0] - 1] is not None:
                return delete(keys, nodes, nodes[index][0] - 1) +
delete(keys, nodes, nodes[index][1] - 1) - 1
            else:
                return -1

    else:
        return 0



def main():
    keys, nodes, deletes = file_input()
    results = []
    quantity = len(keys)
    for num in deletes:
        try:
            index = keys.index(num)
            quantity += delete(keys, nodes, index)
            results.append(quantity)
        except ValueError:
            quantity += 0
            results.append(quantity)
    file_output(results)

    file_output(results)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Функция **delete** удаляет необходимые поддеревья. Она делает это рекурсивно (вызывает саму себя для того чтобы удалить всех потомков удаляемого узла, параллельно она подсчитывает количество удалённых узлов)

Тестовые данные:

Превышение ограничения на время работы, тест 14
Подсказка: это полное двоичное дерево с 32767 вершинами, из которого в случайном порядке удаляются все элементы, кроме корня, включая уже несуществующие на момент удаления элементы.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		Превышение	21139456	978616	40297
1	OK	0.031	9273344	58	12
2	OK	0.015	9236480	27	12
3	OK	0.015	9187328	34	15
4	OK	0.015	9195520	211	30
5	OK	0.031	9240576	246	30
6	OK	0.015	9166848	3437	457
7	OK	0.031	9248768	3363	483
8	OK	0.062	9318400	18842	4247
9	OK	0.046	9465856	25683	3739
10	OK	0.140	10043392	69351	14791
11	OK	0.171	10321920	88936	11629
12	OK	1.250	11997184	244892	40297
13	OK	1.359	12222464	255614	37596
14	TL	Превышение	21139456	978616	0

Отправить Вы использовали 20 из 200 попыток

Сохранить

10. Задача. Проверка корректности [2 s, 256 Mb, 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.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    lines = [fin.readline() for i in range(n)]
    nodes = []
    K = []
    L = []
    R = []

    for i in range(n):
        Ki, Li, Ri = lines[i].split()
        node = BSTNode(int(Ki))
        nodes.append(node)
        K.append(int(Ki))
        L.append(int(Li))
        R.append(int(Ri))

    for i in range(n):
        l = L[i] - 1
        r = R[i] - 1
        if l != -1:
            nodes[i].left = nodes[l]
        if r != -1:
            nodes[i].right = nodes[r]
    fin.close()
    return nodes


def file_output(result):
    fout = open('output.txt', 'w')
    if result:
        fout.write("YES")
    else:
        fout.write("NO")
    fout.close()


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

```
    def get_min(self):
        current = self
        while current.left is not None:
            current = current.left
        return current.key

    def get_max(self):
        current = self
        while current.right is not None:
            current = current.right
        return current.key




def check(node, lo, hi, smallest, largest):
    '''if lo > hi:
        return False'''

    if node is None:
        return True
    if node.key <= lo and node.key != smallest or node.key >= hi and
node.key != largest:
        return False

    return check(node.left, lo, node.key, smallest, largest) and
check(node.right, node.key, hi, smallest, largest)


def main():
    nodes = file_input()

    if nodes:
        smalllest = nodes[0].get_min()
        largest = nodes[0].get_max()
        result = check(nodes[0], smalllest, largest, smalllest, largest)
    else:
        result = True

    file_output(result)


main()
```

Описание решения:

Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **BSTNode** необходим для хранения узла дерева.

Методы **get_min** и **get_max** необходимы для получения минимума и максимума дерева. Они необходимы для дальнейшей проверки дерева на корректность (они служат стартовыми границами)

Функция **check** проверяет вводимое бинарное дерево поиска на корректность рекурсивно (погружаясь в глубину каждого поддерева для проверки условия правильности для всех узлов)

Тестовые данные:

Choose Files No file chosen

Частичное решение, 74 баллов из 100

Ваше решение прошло предварительные тесты и набрало 74 баллов из 100 на основных тестах.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла	Группа тестов
Max		0.625	94740480	3988813	3	
1	OK	0.015	9162752	46	3	
2	OK	0.031	9195520	3	3	
3	OK	0.046	9154560	24	2	
4	OK	0.031	9187328	11	3	0

14 Задача. Вставка в АВЛ-дерево [2 s, 256 Mb, 3 балла]

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.

15 Задача. Удаление из АВЛ-дерева [2 s, 256 Mb, 3 балла]Вставка в 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.

```
def file_input():
    fin = open('input.txt', 'r')
    n = int(fin.readline())
    a = []
    for i in range(n + 1):
        a.append(Node(0, 1, 1, 0))
    for i in range(1, n + 1):
        k, l, r = map(int, fin.readline().split())
        a[i].k = k
        a[i].l = l
        a[i].r = r
        if l > 0:
            a[l].p = i
        if r > 0:
            a[r].p = i
    height(a, n)
    f = 1
    x = int(fin.readline())
    fin.close()
    return a, x, f, n
```

```
def file_output(a, n, f):
    fout = open('output.txt', 'w')
    fout.write(str(n + 1) + '\n')
    a = red(a, f)
    for i in range(1, n + 2):
        fout.write(str(a[i].k) + ' ' + str(a[i].l) + ' ' + str(a[i].r) + '\
n')
    fout.close()


class Node():
    def __init__(self, key, left, right, parent):
        self.k = key
        self.l = left
        self.r = right
        self.p = parent
        self.h = -1


def height(a, n):
    for i in range(n, 0, -1):
        a[i].h = max(a[a[i].l].h, a[a[i].r].h) + 1


def newHeight(a, i):
    l = a[i].l
    r = a[i].r
    if l != 0:
        a[l].h = max(a[a[l].l].h, a[a[l].r].h) + 1
    if r != 0:
        a[r].h = max(a[a[r].l].h, a[a[r].r].h) + 1
    a[i].h = max(a[l].h, a[r].h) + 1


def lt(a, i):
    j = a[i].r
    p = a[i].p
    if i == a[p].l:
        a[p].l = j
    else:
        a[p].r = j
    a[i].r = a[j].l
    a[a[i].r].p = i
    a[i].p = j
    a[j].l = i
    a[j].p = p
    newHeight(a, i)
    newHeight(a, j)
    return j


def rt(a, i):
    j = a[i].l
    p = a[i].p
    if i == a[p].l:
        a[p].l = j
    else:
        a[p].r = j
    a[i].l = a[j].r
```

```
    a[a[i].l].p = i
    a[i].p = j
    a[j].r = i
    a[j].p = p

    newHeight(a, i)
    newHeight(a, j)
    return j


def red(a, s):
    k = 2
    rzv = [s, ]
    rez = [0, ]
    while len(rzv) > 0:
        zam = rzv
        rzv = []
        for i in range(len(zam)):
            if a[zam[i]].l > 0:
                rzv.append(a[zam[i]].l)
                l = k
                k += 1
            else:
                l = 0
            if a[zam[i]].r > 0:
                rzv.append(a[zam[i]].r)
                r = k
                k += 1
            else:
                r = 0
            rez.append(Node(a[zam[i]].k, l, r, 0))
    return rez


def pbal(a, i):
    return a[a[i].l].h - a[a[i].r].h


def balance(a, m):
    i = m
    while i != 0:
        newHeight(a, i)
        if pbal(a, i) == 2:
            if pbal(a, a[i].l) < 0:
                lt(a, a[i].l)
            i = rt(a, i)
        elif pbal(a, i) == -2:
            if pbal(a, a[i].r) > 0:
                rt(a, a[i].r)
            i = lt(a, i)
        k = i
        i = a[i].p
    return k


def add(a, n, m):
    if len(a) == 1:
        a.append(Node(n, 0, 0, 0))
        i = 1
    else:
        i = m
```

```
        while True:
            if a[i].k > n and a[i].l == 0:
                a.append(Node(n, 0, 0, i))
                a[i].l = len(a) - 1
                break
            elif a[i].k < n and a[i].r == 0:
                a.append(Node(n, 0, 0, i))
                a[i].r = len(a) - 1
                break
            elif a[i].k > n:
                i = a[i].l
            else:
                i = a[i].r
    return balance(a, i)


def main():
    a, x, f, n = file_input()
    f = add(a, x, f)
    file_output(a, n, f)


main()
```

```
def file_input():
    fi = open('input.txt', 'r')
    n = int(fi.readline())
    a = []
    for i in range(n + 1):
        a.append(Node(0, 1, 1, 0))
    for i in range(1, n + 1):
        k, l, r = map(int, fi.readline().split())
        a[i].k = k
        a[i].l = l
        a[i].r = r
        if l > 0:
            a[l].p = i
        if r > 0:
            a[r].p = i
    height(a, n)
    f = 1
    x = int(fi.readline())
    fi.close()
    return a, x, f, n


def file_output(a, f, n):
    fo = open('output.txt', 'w')
    fo.write(str(n - 1) + '\n')
    if n - 1 > 0:
        a = red(a, f)
        for i in range(1, n):
            fo.write(str(a[i].k) + ' ' + str(a[i].l) + ' ' + str(a[i].r) +
'\n')

    fo.close()


class Node():
    def __init__(self, key, left, right, parent):
```

```
        self.k = key
        self.l = left
        self.r = right
        self.p = parent
        self.h = -1


def height(a, n):
    for i in range(n, 0, -1):
        a[i].h = max(a[a[i].l].h, a[a[i].r].h) + 1


def newH(a, i):
    l = a[i].l
    r = a[i].r
    if l != 0:
        a[l].h = max(a[a[l].l].h, a[a[l].r].h) + 1
    if r != 0:
        a[r].h = max(a[a[r].l].h, a[a[r].r].h) + 1
    a[i].h = max(a[l].h, a[r].h) + 1


def lt(a, i):
    j = a[i].r
    p = a[i].p
    if i == a[p].l:
        a[p].l = j
    else:
        a[p].r = j
    a[i].r = a[j].l
    a[a[i].r].p = i
    a[i].p = j
    a[j].l = i
    a[j].p = p
    newH(a, i)
    newH(a, j)
    return j


def rt(a, i):
    j = a[i].l
    p = a[i].p
    if i == a[p].l:
        a[p].l = j
    else:
        a[p].r = j
    a[i].l = a[j].r
    a[a[i].l].p = i
    a[i].p = j
    a[j].r = i
    a[j].p = p

    newH(a, i)
    newH(a, j)
    return j


def red(a, s):
    k = 2
    rzv = [s, ]
    rez = [0, ]
```

```
    while len(rzv) > 0:
        zam = rzv
        rzv = []
        for i in range(len(zam)):
            if a[zam[i]].l > 0:
                rzv.append(a[zam[i]].l)
                l = k
                k += 1
            else:
                l = 0
            if a[zam[i]].r > 0:
                rzv.append(a[zam[i]].r)
                r = k
                k += 1
            else:
                r = 0
            rez.append(Node(a[zam[i]].k, l, r, 0))
    return rez


def pbal(a, i):
    return a[a[i].l].h - a[a[i].r].h


def balance(a, m):
    i = m
    k = 0
    while i != 0:
        newH(a, i)
        if pbal(a, i) == 2:
            if pbal(a, a[i].l) < 0:
                lt(a, a[i].l)
            i = rt(a, i)
        elif pbal(a, i) == -2:
            if pbal(a, a[i].r) > 0:
                rt(a, a[i].r)
            i = lt(a, i)
        k = i
        i = a[i].p
    return k


def delete(a, n, m):
    if len(a) != 2:
        i = m
        while True:
            if a[i].k == n:
                if a[i].l == a[i].r == 0:
                    if a[a[i].p].l == i:
                        a[a[i].p].l = 0
                    else:
                        a[a[i].p].r = 0
                    m = a[i].p
                elif a[i].l == 0:
                    if a[a[i].p].l == i:
                        a[a[i].p].l = a[i].r
                        a[a[i].r].p = a[i].p
                        m = a[i].r
                    else:
                        a[a[i].p].r = a[i].r
                        a[a[i].r].p = a[i].p
```

```
                        m = a[i].r
                else:
                    j = a[i].l
                    while a[j].r > 0:
                        j = a[j].r
                    a[i].k = a[j].k
                    if a[a[j].p].l == j:
                        a[a[j].p].l = a[j].l
                        a[a[j].l].p = a[j].p
                    else:
                        a[a[j].p].r = a[j].l
                        a[a[j].r].p = a[j].p
                    m = a[j].p
                break
            elif n < a[i].k and a[i].l > 0:
                i = a[i].l
            elif n > a[i].k and a[i].r > 0:
                i = a[i].r
    return balance(a, m)


def main():
    a, x, f, n = file_input()
    f = delete(a, x, f)
    file_output(a, f, n)


main()
```

В данном решение реализованы стандартные операция вставки и удаления из АВЛ-дерева, для операции вставки реализованы дополнительные методы для поворота дерева вокруг элемента – влево и вправо, для удаления реализованы дополнительные методы поиска максимума и поиск, и удаление максимума для того, чтобы удаление выполнялось согласно алгоритму в условии 15 задачи.

Тестовые данные:

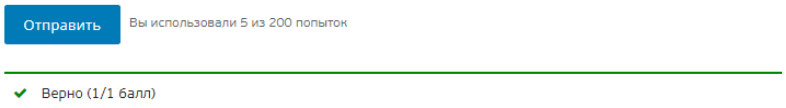
14 Задача:

262	OK	1.046	102854656	3435983	3435992
263	OK	1.015	102907904	3562689	3562698
264	OK	1.031	103354368	3521159	3521168
265	OK	1.031	103186432	3539149	3539158
266	OK	1.109	114118656	3985264	3985273
267	OK	1.171	116629504	3866892	3866901
268	OK	1.156	113831936	3942753	3942762
269	OK	1.140	113680384	3824263	3824272
270	OK	1.156	114020352	4011957	4011966
271	OK	1.156	113762304	3955420	3955429
272	OK	1.125	114024448	3946583	3946592
273	OK	1.171	114212864	3891536	3891545

Отправить Вы использовали 3 из 200 попыток

15 Задача:

227	OK	1.000	103194624	3523352	3523321
228	OK	1.015	102957056	3638077	3638044
229	OK	1.046	103190528	3512844	3512813
230	OK	1.156	114114560	4001320	4001287
231	OK	1.156	113897472	3954282	3954249
232	OK	1.156	113700864	3907814	3907783
233	OK	1.171	113745920	3983014	3982983
234	OK	1.171	113721344	4029895	4029862
235	OK	1.171	113913856	4046188	4046157
236	OK	1.187	114040832	4077288	4077255
237	OK	1.171	114196480	3902092	3902061



Вывод:

Сбалансированные деревья предоставляют огромное преимущество для поиска элементов и работе над деревом.
Также реализация дерева была выполнена быстро, больше пришлось оптимизировать чтение и вывод данных, так что в некоторых задачах необходимо уделять особое внимание не только задаче, но и работе над вводом и выводом.

16 Задача. K-й максимум [2 s, 512 Mb, 3 балла]

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.

Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе ki-го мак- симума, он существует.

• Ограничения на входные данные. n ≤ 100000, |ki

| ≤ 109.

• Формат вывода / выходного файла (output.txt). Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число – ki-й максимум.

• Ограничение по времени. 2 сек.

• Ограничение по памяти. 512 мб.

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


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

    def search(self, data, root):
        if root is None or data == root.key:
            return root
        if data < root.key:
            if root.left is not None:
                return self.search(data, root.left)
            return root
        if root.right is not None:
            return self.search(data, root.right)
        return root

    def insert(self, data):
        if self.root is None:
            self.root = Node()
            self.root.key = data
            self.root.size = 1
        else:
            t = self.search(data, self.root)
            if t.key == data:
                return
            elif data < t.key:
                t.left = Node()
                t.left.key = data
                t.left.parent = t
                t.left.size = 1
            else:
                t.right = Node()
                t.right.key = data
                t.right.parent = t
                t.right.size = 1
            while t is not None:
                t.size += 1
                t = t.parent

    def treeMin(self, root):
        while root.left is not None:
            root = root.left
        return root.key

    def rightAncestor(self, root):
        if root.parent is None:
            return 0
        if root.key < root.parent.key:
            return root.parent.key
        return self.rightAncestor(root.parent)

    def next(self, data):
        if self.root is None:
```

```
            return 0
        t = self.search(data, self.root)
        f = False
        if t.key != data:
            self.insert(data)
            f = True
            t = self.search(data, self.root)
        if t.right is not None:
            res = self.treeMin(t.right)
        else:
            res = self.rightAncestor(t)
        if f:
            self.delete(data)
        return res

    def delete(self, data):
        t = self.search(data, self.root)
        if t.right is None:
            par = t.parent
            if t.left is not None:
                t.left.parent = par
            if par is None:
                self.root = t.left
            elif par.left.key == t.key:
                par.left = t.left
            else:
                par.right = t.left
        else:
            x = self.search(self.next(t.key), self.root)
            y = x.right
            par = x.parent
            t.key = x.key
            if x.key != t.right.key:
                x.parent.left = y
                if y is not None:
                    y.parent = x.parent
            else:
                t.right = y
                if y is not None:
                    y.parent = t
        while par is not None:
            par.size -= 1
            par = par.parent

    def kMin(self, root, k):
        if root.right is None:
            s = 0
        else:
            s = root.right.size
        if k == s + 1:
            return root.key
        if k < s + 1:
            return self.kMin(root.right, k)
        return self.kMin(root.left, k - s - 1)


fin = open('input.txt', 'r')
fout = open('output.txt', 'w')
n = int(fin.readline())
T = binTree()
for i in range(n):
```

```
    com, key = tuple(map(int, fin.readline().split()))
    if com == 1:
        T.insert(key)
    elif com == -1:
        T.delete(key)
    else:
        fout.write(str(T.kMin(T.root, key)) + '\n')

fin.close()
fout.close()
```

Описание решения:

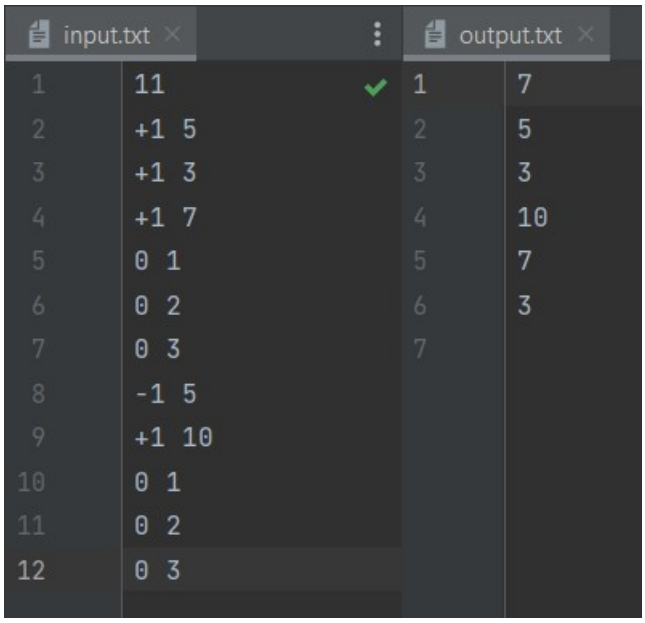
Функции **file_input()** и **file_output(n)** предназначены для ввода и вывода соответственно и непосредственно в решении задачи не участвуют.

Функция **main()** – главная, в ней прописана основная логика программы(ввод, вывод и обработка данных)

Класс **Node** необходим для хранения узла дерева.

Реализованы добавление(**insert**), удаление(**delete**) и поиск(**seacrh**). Для их работы сделаны вспомогательные функции **treeMin**, **rightAncestor**, next и **kMin**

Тестовые данные:


input.txt
11
+1 5
+1 3
+1 7
0 1
0 2
0 3
-1 5
+1 10
0 1
0 2
0 3
output.txt
7
5
3
10
7
3