

• 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 3

Выполнила:
Еремеева А.В.
К3244(13.2)

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №3. Простейшее BST (1 балл)	3
Задача №8. Высота дерева возвращается (2 балла)	6
Задача №16. Задача К-й максимум (3 балла)	9
Дополнительные задачи	16
Задача №12. Проверка сбалансированности(2 балла)	16
Задача №9. Удаление поддеревьев(2 балла)	20
Задача №7. Оpozнание двоичного дерева поиска(2.5 балла)	23
Задача №4. Простейший неявный ключ(1 балл)	26
Вывод	29

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.

Листинг кода:

```
class Node:
    def __init__(self, key):
        self.value = key # val – значение узла (ключ)
        self.left = None #left – ссылка на левое поддерево
        self.right = None #right – ссылка на правое поддерево

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

    def insert(self, key):
        if self.root is None: #Если дерево пустое, создается новый узел с
ключом key и он становится корнем
            self.root = Node(key)
        else: #Если корень уже существует, рекурсивно вызываем метод
insert_node, чтобы найти место для нового узла
            self.insert_node(self.root, key)

    def insert_node(self, root, key): #Ищем правильное место для вставки
нового узла в дерево
        if key < root.value: #Если ключ меньше значения текущего узла,
проверяем левое поддерево
            if root.left is None: #Если в левом поддереве нет узла, создается
новый узел
                root.left = Node(key)
            else:
                self.insert_node(root.left, key)
        elif key > root.value: #Если ключ больше значения узла, поиск
продолжается в правом поддереве
            if root.right is None:
                root.right = Node(key)
            else:
                self.insert_node(root.right, key)
```

```

def find(self, root, key, min_greater):
    if root is None:
        return 0 if min_greater == float('inf') else min_greater
    if root.value > key:
        min_greater = min(min_greater, root.value)
        return self.find(root.left, key, min_greater)
    else:
        return self.find(root.right, key, min_greater)

def find_min(self, key):
    return self.find(self.root, key, float('inf'))

with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    commands = input_file.readlines()
    result = []
    bst = BST() #Создается новое дерево BST
    for command in commands:
        if command.startswith('+'): #добавляем элемент в дерево с помощью
        метода insert
            _, x = command.split()
            bst.insert(int(x))
        elif command.startswith('>'): #поиск минимального элемента, большего
        указанного числа, с помощью метода find_min
            _, x = command.split()
            result.append(bst.find_min(int(x)))
    for elem in result:
        output_file.write(f'{elem}\n')

```

Текстовое объяснение решения:

Создаем класс Node который описывает узел дерева, по умолчанию ссылки на левого и правого потомков изначально равны None. Создаем класс Bst – конструктор бинарного дерева поиска. Изначально дерево пустое, и его корневой узел равен None. Метод insert вставляет новый элемент в дерево: если дерево пустое (корень none), создается новый узел с ключом key и становится корнем. Если корень уже существует, вызывается вспомогательный метод insert_node, чтобы рекурсивно найти место для нового узла. Метод insert_node ищет нужное/подходящее место для вставки нового узла в дерево: если ключ меньше значения текущего узла, проверяется левое поддерево, если в левом поддереве нет узла, создается новый узел, если ключ больше значения узла, поиск продолжается в правом поддереве. Таким образом, соблюдается правило того, что узлы в левом поддереве всегда меньше, а узлы в правом — больше. Метод find —

рекурсивно ищет минимального элемента больше заданного. Этот метод ищет минимальный элемент, который больше указанного числа `key`: если текущий узел равен `none` (значит мы достигли конца дерева), тогда возвращаем `0`, если подходящих элементов не найдено, либо минимальный найденный элемент. Если значение текущего узла больше, чем `key`, то это потенциальный минимальный элемент, больший `key`. Продолжаем искать в левом поддереве. Если значение узла меньше или равно `key`, то продолжаем поиск в правом поддереве. Метод `find_min` вызывает рекурсивно метод `find`, чтобы найти минимальный элемент, большего заданного числа, начиная с корня дерева. Если не найдено ни одного подходящего элемента, то он возвращает `0`.

Результат работы кода на значениях из задачи:

+ 1	1	3
+ 3	2	3
+ 3	3	0
> 1	4	2
> 2	5	
> 3		
+ 2		
> 1		

	Время выполнения	Затраты памяти
Пример из задачи	0.00 сек.	23.3 Mib
Верхняя граница диапазона значений входных данных из текста задачи(1000000)	2.1 сек.	77 Mib

Вывод по задаче: программа создаёт дерево поиска, в которое добавляются элементы и ищет минимальный элемент, больший числа `x`

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 никакой вершину дважды.

Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю

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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i -ой вершины ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет).

Листинг кода:

```
class Node: #Класс, который представляет вершину дерева
    def __init__(self, key, left, right):
        self.key = key #key – значение ключа узла
        self.left = left #left – индекс левого ребенка в массиве
        self.right = right #right – индекс правого ребенка в массиве

def tree_height(tree, node_index):
    if node_index == 0:
        return 0 # Если узла нет, высота 0
    node = tree[node_index]
    left_height = tree_height(tree, node.left)
    right_height = tree_height(tree, node.right)
    return max(left_height, right_height) + 1

with open('input.txt', 'r') as f:
    n = int(f.readline()) # Чтение числа вершин
    tree = [None] * (n + 1) # Массив для хранения вершин (на 1 больше, чем кол-во вершин, чтобы
# индексация начиналась с 1, так как вершины нумеруются с 1).
```

```

# Считываем вершины
for i in range(1, n + 1):
    key, left, right = map(int, f.readline().split())
    tree[i] = Node(key, left, right)
#Создаём новый объект класса Node для каждой вершины с ключом, индексом левого и правого ребенка и сохраняем
его в массив tree на соответствующей позиции.

#Записываем ответ в файл
end = open('output.txt', 'x')
if n == 0: #Проверяем, есть ли в дереве вершины
    end.write('0')
else:
    end.write(str(tree_height(tree, 1)))
end.close()

```

Текстовое объяснение решения:

Создаем класс, который будет представлять вершину дерева. У каждой вершины будет `key` — значение ключа узла, `left` — индекс левого ребенка в массиве (или 0, если у узла нет этого ребенка), `right` — индекс правого ребенка в массиве.

Определяем функцию, которая принимает два аргумента: `tree` — список, представляющий дерево, где каждый элемент — это объект класса `node`, `node_index` — индекс текущего узла в массиве `tree`. Проверяем `if node_index == 0` (существует ли текущий узел, если индекс равен 0, то узла нет (т.е. пустой ребенок), и значит высота этого поддеревья равна 0. Если узел существует, то получаем текущий узел из массива `tree` по индексу `node_index`.

Далее `left_height = tree_height(tree, node.left)` с помощью рекурсии вычисляем высоту левого поддеревья. (`node.left` — это индекс левого ребенка текущего узла) Если у текущего узла нет левого ребенка, `node.left` будет равно 0, что приведет к возврату значения 0 для высоты этого поддеревья. Если ребенок есть, вызывается функция `tree_height` для левого ребенка, и так продолжается до самого низа дерева. Аналогично вычисляется высота для правого поддеревья.

`return max(left_height, right_height) + 1` возвращаем максимальную высоту между левым и правым поддеревом, прибавляя 1 для текущего узла (так как текущий узел тоже считается). Это даёт высоту текущего узла как максимальную глубину до листьев (конечных узлов).

Результат работы кода на примерах из задачи:

```

1 6
2 -2 0 2
3 8 4 3
4 9 0 0
5 3 6 5
6 6 0 0
7 0 0 0|

```

	Время выполнения	Затраты памяти
Примеры из задачи()	0.00сек.	22.8 Mib
Верхняя граница диапазона значений входных данных из текста задачи(1000000)	1.91 сек.	55.2Mib

Вывод по задаче: программа рекурсивно вычисляет высоту бинарного дерева, начиная с заданного узла и записывает результат в файл output.txt.

Задача №16. Задача. К-й максимум (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 c_i и k_i – тип и аргумент команды соответственно. Поддерживаемые команды:

- +1 (или просто 1): Добавить элемент с ключом k_i
- 0 : Найти и вывести k_i -й максимум.
- -1 : Удалить элемент с ключом k_i

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

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

Листинг кода:

```
class Node:
    def __init__(self, key):
        self.key = key # ключ, который хранит узел
        self.left = None #ссылки на левого и правого ребёнка соответственно
        self.right = None #ссылки на левого и правого ребёнка соответственно
        self.height = 1 #высота узла (максимальная высота поддерева).
        self.size = 1 # Размер поддерева корнем которого является текущий узел (с учетом этого узла)

class AVLTree:
    def get_height(self, node): #возвращаем высоту узла. Если узел пустой, возвращаем 0
        return node.height if node else 0
    def get_size(self, node):# возвращаем размер поддерева для узла.
        return node.size if node else 0
    def update(self, node):#обновляем высоту и размер текущего узла, используя высоты и размеры его потомков
        if node:
            node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))
            node.size = 1 + self.get_size(node.left) + self.get_size(node.right)
    def rotate_right(self, y): # правый поворот, если левая ветвь дерева перевешена
        x = y.left
        T2 = x.right
        # Поворот
        x.right = y
        y.left = T2
        # Обновляем высоту и размер поддеревьев
        self.update(y)
        self.update(x)
        return x
    def rotate_left(self, x):# левый поворот, если правая ветвь дерева перевешена
        y = x.right
        T2 = y.left
        # Поворот
        y.left = x
        x.right = T2
        # Обновляем высоту и размер поддеревьев
        self.update(x)
        self.update(y)
        return y

    def get_balance(self, node): #если результат больше 0, значит левое поддерево выше правого,если результат меньше 0, значит
        # что правое поддерево выше левого
        if not node:
            return 0
```

```

        return self.get_height(node.left) - self.get_height(node.right) #если результат 0, значит, что поддеревья имеют
одинаковую высоту, и узел сбалансирован.

def insert(self, node, key):
    if not node: #Если текущий узел None, это значит, что мы достигли места, где следует вставить новый узел
        return Node(key) #Мы создаем новый узел с ключом key и возвращаем его.
    if key < node.key: #Если key меньше, чем node.key, то рекурсивно вызываем метод insert для левого поддерева
        node.left = self.insert(node.left, key)
    else:
        node.right = self.insert(node.right, key) #Если key больше или равен, то вызываем метод для правого поддерева
    # Обновляем текущий узел
    self.update(node)
    # Балансировка
    balance = self.get_balance(node)
    # Левый поворот
    if balance > 1 and key < node.left.key:
        return self.rotate_right(node)
    # Правый поворот
    if balance < -1 and key > node.right.key:
        return self.rotate_left(node)
    # Левый-правый
    if balance > 1 and key > node.left.key:
        node.left = self.rotate_left(node.left)
        return self.rotate_right(node)
    # Правый-левый
    if balance < -1 and key < node.right.key:
        node.right = self.rotate_right(node.right)
        return self.rotate_left(node)
    return node

def min_value_node(self, node):
    if node is None or node.left is None:
        return node
    return self.min_value_node(node.left)

def delete(self, node, key):
    if not node: #остановить рекурсию в случае, если ключ не найден в дереве
        return node
    if key < node.key: #Если key меньше, чем ключ текущего узла, то рекурсивно вызываем метод delete для левого дочернего
узла
        node.left = self.delete(node.left, key)
    elif key > node.key: #Если ключ key больше, чем ключ текущего узла, продолжаем поиск в правом поддереве
        node.right = self.delete(node.right, key)
    else: #Когда ключ совпадает с node.key, значит мы дошли рекурсивно и его нужно удалить.
        # Узел с одним или без детей
        if node.left is None:
            return node.right
        elif node.right is None:
            return node.left
        # Узел с двумя детьми, получаем следующий по возрастанию узел
        temp = self.min_value_node(node.right)
        node.key = temp.key
        node.right = self.delete(node.right, temp.key)
    # Обновляем текущий узел
    self.update(node)
    # Балансировка
    balance = self.get_balance(node)
    # Левый поворот
    if balance > 1 and self.get_balance(node.left) >= 0:
        return self.rotate_right(node)
    # Правый поворот
    if balance < -1 and self.get_balance(node.right) <= 0:
        return self.rotate_left(node)
    # Левый-правый
    if balance > 1 and self.get_balance(node.left) < 0:
        node.left = self.rotate_left(node.left)
        return self.rotate_right(node)
    # Правый-левый
    if balance < -1 and self.get_balance(node.right) > 0:
        node.right = self.rotate_right(node.right)
        return self.rotate_left(node)
    return node

def find_kth_max(self, node, k):
    if not node: #достигли конца дерева, и k-й максимальный элемент не найден.
        return None
    right_size = self.get_size(node.right) # кол-во узлов в правом поддереве текущего узла (используется для определения
позиции текущего узла относительно k-го максимума)
    # Если в правом поддереве ровно (k-1) элементов, то текущий узел - это k-й максимум
    if right_size == k - 1:
        return node.key
    elif right_size >= k:
        return self.find_kth_max(node.right, k)
    else:
        return self.find_kth_max(node.left, k - right_size - 1)

class AVLTreeManager:
    def __init__(self): #Создаем атрибут tree для хранения корня AVL-дерева
        self.tree = None

```

```

self.avl = AVLTree()

def add(self, key): #вставку элемента с ключом key в дерево self.tree
    self.tree = self.avl.insert(self.tree, key)

def remove(self, key): #Удаляем элемент с ключом key из дерева self.tree
    self.tree = self.avl.delete(self.tree, key)

def find_kth_max(self, k):#ищем k-й максимальный элемент в AVL-дереве
    return self.avl.find_kth_max(self.tree, k)

with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    n = int(input_file.readline().strip())
    commands = [tuple(map(int, input_file.readline().split())) for _ in range(n)]
    avl_manager = AVLTreeManager()
    result = []
    for command in commands:
        c_type, k = command

        if c_type == 1:
            avl_manager.add(k)
        elif c_type == -1:
            avl_manager.remove(k)
        elif c_type == 0:
            result.append(avl_manager.find_kth_max(k))

    for res in result:
        output_file.write(f"{res}\n")

```

Текстовое объяснение решения:

Каждая вершина AVL-дерева — это объект класса `node`, у которого есть атрибуты: ключ, который хранит узел, ссылки на левого и правого ребёнка, высота узла (максимальная высота поддеревы), размер поддеревы, корнем которого является текущий узел (включая сам узел). Сам класс AVL будет реализовывать операции вставки, удаления и поиска по AVL-дереву. Метод `get_height` возвращает высоту узла. Если узел пустой (`None`), возвращаем 0, метод `get_size` возвращает размер поддеревы для узла, метод `update` обновляет высоту и размер текущего узла, используя высоты и размеры его потомков. Методы поворотов используются для восстановления баланса AVL-дерева после операций вставки или удаления. Эти методы позволяют поддерживать свойства AVL-дерева, обеспечивая, чтобы высота левого и правого поддеревьев каждого узла различалась не более чем на 1, что гарантирует эффективность операций поиска, вставки и удаления.

Метод `rotate_right`: используется, когда левое поддерево выше, чем правое.

$x = y.left$ Узел x назначается как левый ребенок узла y . Это будет узел, который поднимается вверх после поворота.

$T2 = x.right$ $T2$ назначается как правый ребенок узла x . Это поддерево, которое будет временно отсоединено во время поворота.

Сам поворот происходит так:

`x.right=y` Узел `y` теперь становится правым ребенком узла `x`. Таким образом, `y` перемещается вниз на одну позицию.

`y.left=T2` Правое поддерево `x` (т.е. `T2`) присоединяется к `y` как его левый ребенок.

Потом происходит обновление узлов:

`self.update(y)` `self.update(y)` обновляют высоту и баланс для узлов `y` и `x`. В конце возвращаем новый корень поддерева : `return x`

Метод `rotate_left` используется, когда правое поддерево выше, чем левое

`y=x.right` : Узел `y` назначается как правый ребенок узла `x`. Это будет узел, который поднимается вверх после поворота.

`T2=y.left` `T2` назначается как левый ребенок узла `y`. Это поддерево, которое будет временно отсоединено во время поворота.

Поворот:

`y.left=x` Узел `x` теперь становится левым ребенком узла `y`. Таким образом, `x` перемещается вниз на одну позицию.

`x.right=T2` `T2` Левое поддерево `y` (т.е. `T2`) присоединяется к `x` как его правый ребенок.

Метод `insert` для вставки нового ключа в AVL-дерево. Этот метод не только добавляет новый узел, но и обеспечивает сбалансированность дерева с помощью поворотов, если это необходимо. Если текущий узел (`node`) равен `None`, это означает, что мы достигли места, где следует вставить новый узел. Мы создаем новый узел с ключом `key` и возвращаем его. Далее проверяем если `key` меньше, чем `node.key`, мы рекурсивно вызываем метод `insert` для левого поддерева. Если `key` больше или равен, то вызываем метод для правого поддерева.

После вставки нового узла мы обновляем параметры текущего узла `node` (высота и баланс). Потом вычисляем баланс текущего узла, используя метод `get_balance`. Баланс узла вычисляется как разница высоты левого и правого поддеревьев. Если баланс больше 1 (левое поддерево выше) и `key` меньше, чем ключ левого узла, это левый левый случай и выполняем

правый поворот. Если баланс меньше -1 (правое поддереве выше) и key больше, чем ключ правого узла, это правый правый случай. Мы выполняем левый поворот. Если баланс больше 1 (левое поддереве выше) и key больше, чем ключ левого узла, это левый правый случай. Мы сначала выполняем левый поворот на левом поддереве, а затем правый поворот на текущем узле. Если баланс меньше -1 (правое поддереве выше) и key меньше, чем ключ правого узла, это правый левый случай. Мы сначала выполняем правый поворот на правом поддереве, а затем левый поворот на текущем узле. В конце метода мы возвращаем текущий узел. Это может быть обновленный узел (если произошли повороты), который будет использован для связи с родительским узлом.

Метод delete рекурсивно ищет узел для удаления, выполняет удаление, а затем проверяет, требуется ли балансировка дерева. Делаем сначала проверку остановить рекурсию в случае, если ключ не найден в дереве. Если key меньше, чем ключ текущего узла, то рекурсивно вызываем метод delete для левого дочернего узла. Если ключ key больше, чем ключ текущего узла, продолжаем поиск в правом поддереве. Когда ключ совпадает с node.key то значит мы дошли рекурсивно до нужного и его нужно удалить. Если у узла нет левого ребенка, мы просто возвращаем правое поддереве, т.е. правый узел становится на место удаленного узла. Если у узла нет правого ребенка, возвращаем левое поддереве. Если у узла есть оба ребенка, мы находим узел с минимальным значением в правом поддереве (следующий по возрастанию узел). Этот узел гарантированно не имеет левого ребенка. Мы копируем ключ из этого узла в текущий узел и рекурсивно удаляем этот узел из правого поддереве. После удаления узла и изменения поддеревьев обновляем информацию о текущем узле. Потом вычисляем баланс текущего узла.

Если баланс больше 1 и баланс левого дочернего узла неотрицателен (т.е. его поддереве тоже сбалансировано или левый перекося), выполняем правый поворот (левый левый). Если баланс меньше -1 (правый перекося) и баланс правого дочернего узла не положителен, выполняем левый поворот (правый правый). Если баланс больше 1 и левое поддереве имеет правый перекося (баланс левого узла отрицателен), выполняем левый поворот на левом поддереве, затем правый поворот на текущем узле (левый правый). Если баланс меньше -1 и правое поддереве имеет левый перекося,

выполняем правый поворот на правом поддереве, затем левый поворот на текущем узле(правый левый).Потом возвращаем текущий узел.

метод `find_kth_max` реализует поиск k-го максимального элемента.Проверяем в начале достигли ли мы конца дерева, и следовательно k-й максимальный элемент не найден.`right_size = self.get_size(node.right)` # кол-во узлов в правом поддереве текущего узла(используется для определения позиции текущего узла относительно k-го максимума).Если размер правого поддерева равен k-1, значит текущий узел `node` является k-м максимальным элементом. (Т.к. все узлы в правом поддереве меньше, чем текущий узел, и если в правом поддереве ровно k-1 узлов, значит, текущий узел и есть искомый элемент), поэтому возвращаем `node.key`.Если размер правого поддерева больше или равен k, это означает, что k-й максимальный элемент находится в правом поддереве. Мы продолжаем поиск, вызывая `find_kth_max` для правого дочернего узла с тем же значением k.Если размер правого поддерева меньше k, это означает, что k-й максимальный элемент находится в левом поддереве. В этом случае мы вызываем этот метод для левого дочернего узла, передавая `k-right_size-1` в качестве нового значения k(Поскольку теперь мы ищем k-й максимум в левом поддереве, где мы уже исключили `right_size` узлов и текущий узел)

Результат работы кода на значениях из задачи:

1	11	
2	+1 5	
3	+1 3	
4	+1 7	
5	0 1	
6	0 2	
7	0 3	
8	-1 5	
9	+1 10	
10	0 1	
11	0 2	
12	0 3	

1	7
2	5
3	3
4	10
5	7
6	3
7	

	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	22Mib
Верхняя граница диапазона значений входных данных из текста задачи(1000)	1.99сек.	67.4Mib

Вывод по задаче: программа позволяет добавлять и удалять элементы, а также находить k-й максимальный элемент

Дополнительные задачи

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:

$-1 \leq B(V) \leq 1$ 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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i -ой вершины ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

Листинг кода:

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

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

    def add_node(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._add_node(self.root, key)
```



```

def _add_node(self, node, key):
    if key < node.key:
        if node.left is None:
            node.left = Node(key)
        else:
            self._add_node(node.left, key)
    else:
        if node.right is None:
            node.right = Node(key)
        else:
            self._add_node(node.right, key)

def height(self, node):
    if node is None:
        return 0
    return 1 + max(self.height(node.left), self.height(node.right))

def balance(self, node):
    if node is None:
        return 0
    left_height = self.height(node.left)
    right_height = self.height(node.right)
    return right_height - left_height

def get_balances(self):
    balances = []
    self._get_balances(self.root, balances)
    return balances

def _get_balances(self, node, balances):
    if node is None:
        return
    balances.append(self.balance(node))
    self._get_balances(node.left, balances)
    self._get_balances(node.right, balances)

def main():
    with open("input.txt", "r") as in_file, open("output.txt", "w") as out_file:
        n = int(in_file.readline().strip())

        if n == 0:
            out_file.write(str(0))
            return

        # Считываем данные и строим дерево
        nodes = [tuple(map(int, in_file.readline().strip().split())) for _ in range(n)]

        tree = BST() #Создаем экземпляр BST, и в него добавляем узлы
        for key, left, right in nodes:
            tree.add_node(key)

        # Получаем и записываем балансы
        balances = tree.get_balances()
        for balance in balances:
            out_file.write(f"{balance}\n")

if __name__ == '__main__':
    main()

```

Текстовое объяснение решения:

Создается класс Node, представляющий узел в двоичном дереве с параметрами: key–значение, хранящееся в узле, left–указатель на левое поддерево (или None, если его нет), right– указатель на правое поддерево (или None, если его нет).

Создается класс BST (Binary Search Tree), представляющий двоичное дерево поиска с параметром:root–указатель на корень дерева (изначально равен None, так как дерево пустое).

Метод add_node добавляет новый узел с ключом key в дерево.Если root равен none, создается новый узел, который становится корнем.Если дерево не пустое, вызывается вспомогательный метод _add_node, который рекурсивно находит правильное место для нового узла.

Вспомогательный метод _add_node добавляет узел с ключом key в поддерево, начиная с узла node .Если key меньше ключа текущего узла, проверяется левое поддерево.Если левого поддерева нет, создается новый узел.Если оно есть, вызывается _add_node рекурсивно для левого поддерева.

Метод height вычисляет высоту поддерева, начиная с узла node.Если node равен none, высота равна 0.В противном случае, высота равна 1 плюс максимальная высота левого и правого поддеревьев.Аналогичная логика для правого поддерева, если key больше или равен ключу текущего узла.

Метод balance вычисляет баланс узла node .Если node равен none, баланс равен 0.Высчитывается высота левого и правого поддеревьев, затем возвращается разность: высота правого минус высота левого поддерева.

Метод get_balances собирает балансы всех узлов дерева.Создается пустой список balances.Вызывается вспомогательный метод _get_balances, который рекурсивно собирает балансы узлов, начиная с корня.

Вспомогательный метод _get_balancesдобавляет балансы узлов в список balances. Если node равен none, просто возвращает/Добавляет баланс текущего узла в список.Рекурсивно вызывает себя для левого и правого поддеревьев.

Результат работы кода на значениях из задачи:

1	6	
2	-2	0 2
3	8	4 3
4	9	0 0
5	3	6 5
6	6	0 0
7	0	0 0

1	3	
2	-1	
3	0	
4	0	
5	0	
6	0	
7		

	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	22Mib
Верхняя граница диапазона значений входных данных из текста задачи(100000)	1.67сек.	77.7Mib

Вывод по задаче: программа создает двоичное дерево поиска из заданного списка узлов, вычисляет высоту и баланс каждого узла и записывает результаты в файл

Задача №9. Удаление поддеревьев (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 10^9 . 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.

Формат ввода / входного файла (input.txt). Входной файл содержит описание двоичного дерева и описание запросов на удаление. 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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i -ой вершины ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. В следующей строке находится число M – число запросов на удаление. В следующей строке находятся M чисел, разделенных пробелами – ключи, вершины с которыми (вместе с их поддеревьями) необходимо удалить. Все эти числа не превосходят 10^9 по абсолютному значению. Вершина с таким ключом не обязана существовать в дереве – в этом случае дерево изменять не требуется. Гарантируется, что корень дерева никогда не будет удален.

- Ограничения на входные данные.

$$1 \leq N \leq 2 \cdot 10^5, |K_i| \leq 10^9, 1 \leq M \leq 2 \cdot 10^5$$

- Формат вывода / выходного файла (output.txt). Выведите M строк. На i -ой строке требуется вывести число вершин, оставшихся в дереве после выполнения i -го запроса на удаление.

Листинг кода:

```
#Удаляем поддерево с корнем в узле с заданным ключом и возвращаем кол-во оставшихся узлов
def delete_subtree_node(bst, key):
    # Удаляем узел с заданным ключом
    if key in bst:
        # Получаем ключи левого и правого поддеревьев
        left_k, right_k = bst[key]
        # Удаляем узел
        bst.pop(key)

        # Рекурсивно удаляем поддеревья, если они существуют
        if left_k is not None:
            delete_subtree_node(bst, left_k)
        if right_k is not None:
            delete_subtree_node(bst, right_k)

    return len(bst) # Возвращаем количество оставшихся узлов

def build_bst(nodes): #Создаем BST из списка узлов в виде (ключ, левый_ребенок,
#правый_ребенок)
    bst = {}
    for key in nodes:
        k, left_idx, right_idx = key
        # Преобразуем индексы детей в ключи
        left_k = nodes[left_idx - 1][0] if left_idx > 0 else None
        right_k = nodes[right_idx - 1][0] if right_idx > 0 else None
        bst[k] = (left_k, right_k) # Добавляем узел в BST
    return bst

with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    n = int(input_file.readline().strip()) # Читаем количество узлов
    nodes = [list(map(int, input_file.readline().strip().split())) for _ in range(n)] #
#Читаем узлы

    # Строим BST
    bst = build_bst(nodes)

    m = int(input_file.readline().strip()) # Читаем количество запросов на удаление
    removes = list(map(int, input_file.readline().strip().split())) # Читаем ключи для
#удаления

    # Обрабатываем каждый запрос на удаление
    for key in removes:
        remaining_nodes = delete_subtree_node(bst, key) # Удаляем поддерево
        output_file.write(f'{remaining_nodes}\n') # Записываем количество оставшихся
#узлов
```

Текстовое объяснение решения:

Функция `delete_subtree_node(bst, key)` удаляет поддерево с корнем в узле с заданным ключом и возвращает количество оставшихся узлов. Сначала проверяем, существует ли узел с заданным ключом в словаре `bst`. Если узел существует, то получаем ключи его левого (`left_k`) и правого (`right_k`) поддеревьев. Затем удаляем этот узел из `bst` с помощью метода `pop()`. Рекурсивное удаление поддеревьев: удаление левого поддерева если

left_k не none, то вызываем рекурсивную функцию для удаления левого поддерева. Удаление правого поддерева: Аналогично, если right_k не none, то вызываем рекурсивную функцию для удаления правого поддерева.

Инициализируем пустой словарь bst, который будет хранить узлы и их связи. Запускаем цикл по узлам: проходимся по всем узлам в списке nodes, извлекаем ключ узла k, индекс левого ребенка left_idx, и индекс правого ребенка right_idx.

Преобразование индексов: если индекс больше 0, извлекаем соответствующий ключ ребенка из nodes. Если индекс равен 0, устанавливаем значение left_k или right_k в none, так как ребенка нет.

Заполнение словаря: Добавляем узел в словарь bst, где ключ — это ключ узла, а значение — кортеж, содержащий ключи его детей.

Результат работы кода на значениях из задачи:

1	6	
2	-2 0 2	
3	8 4 3	
4	9 0 0	
5	3 6 5	
6	6 0 0	
7	0 0 0	
8	4	
9	6 9 7 8	
10		

1	5
2	4
3	4
4	1
5	

	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	22Mib
Верхняя граница диапазона значений входных данных из текста задачи(500000)	1.88сек.	98.7Mib

Вывод по задаче: программа считывает двоичное дерево поиска и запросы на удаление, обрабатывает эти запросы, удаляя соответствующие поддеревья, и записывает количество оставшихся узлов в выходной файл

Задача №7. Оpoznание 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 .

Листинг кода:

```
def BST(tree_nodes, i_node, low_lim, upper_lim): #функция проверяет, находится ли каждое
значение в своем допустимом диапазоне, соблюдая правила BST
    if i_node == -1: #Если текущий узел не существует, функция возвращает True, т.к
отсутствие узла не нарушает свойства BST
        return True
    if tree_nodes[i_node][0] < low_lim: #Если ключ текущего узла меньше low, это означает,
что он не может находиться в допустимом диапазоне
        return False
    if tree_nodes[i_node][0] >= upper_lim: #Если ключ текущего узла больше или равен high,
это также нарушает свойства BST
        return False
    return (BST(tree_nodes, tree_nodes[i_node][1], low_lim, tree_nodes[i_node][0]) and
#Вызываем BST для левого поддерева с обновленным верхним пределом и для правого с
обновленным нижним пределом
        BST(tree_nodes, tree_nodes[i_node][2], tree_nodes[i_node][0], upper_lim))

def check_if_BST(nodes):
    if len(nodes) == 0 or BST(nodes, 0, -2 ** 32, 2 ** 32): #пустое дерево считается
корректным BST.
        return 'CORRECT'
```

```

else:
    return 'INCORRECT'

with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    n = int(input_file.readline())#Считывается количество узлов n.
    nodes = [list(map(int, input_file.readline().split())) for _ in range(n)]#добавляем
все узлы, считанные из файла, разбивая каждый на три числа (ключ, левый и правый индекс)
    output_file.write(check_if_BST(nodes))

```

Текстовое объяснение:

BST функция проверяет, находится ли каждое значение в своем допустимом диапазоне, соблюдая правила BST. В нее поступают аргументы: nodes–список, где каждый элемент представляет собой узел дерева. Каждый узел представлен в виде списка из трех значений: [K, L, R], где K — ключ узла, L — индекс левого ребенка (или -1, если отсутствует), R — индекс правого ребенка (или -1, если отсутствует); i_node–индекс текущего узла, который проверяется; low и high–пределы, в которых должен находиться ключ текущего узла.

Если текущий узел не существует (i_node == -1), функция возвращает True, так как отсутствие узла не нарушает свойства BST.

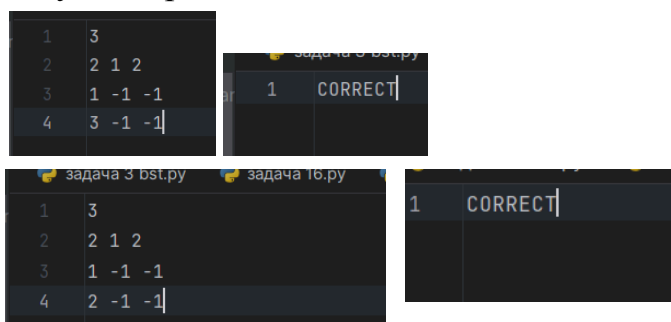
Если ключ текущего узла меньше low, это означает, что он не может находиться в допустимом диапазоне, и функция возвращает False.

Если ключ текущего узла больше или равен high, это также нарушает свойства BST, и функция возвращает False. Рекурсивные проверки:

Вызывается BST для левого поддерева с обновленным верхним пределом (nodes[i_node][0]). Вызывается BST для правого поддерева с обновленным нижним пределом (nodes[i_node][0]).

Функция check_if_BST : если список узлов пуст (len(nodes) == 0), возвращает 'CORRECT', поскольку пустое дерево считается корректным BST. В противном случае вызывает функцию is_BST, начиная с корневого узла (индекс 0) и задает допустимый диапазон для 32-битных целых чисел. Возвращает 'CORRECT', если дерево является корректным BST, иначе 'INCORRECT'.

Результат работы кода на значениях из задачи:



```

1 3
2 2 1 2
3 1 -1 -1
4 3 -1 -1

```

```

1 CORRECT

```


	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	21Mib
Верхняя граница диапазона значений входных данных из текста задачи(500000)	1.78сек.	98.7Mib

Вывод по задаче: программа проверяет, является ли заданное двоичное дерево поиска корректным, ключи могут повторяться, но для повторяющихся ключей должен соблюдаться порядок (ключи в правом поддереве должны быть больше или равны).

Задача №4. Простейший неявный ключ (1 балл)

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

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

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

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. Формат вывода / выходного файла (output.txt). Для каждого запроса вида «? k» выведите в отдельной строке ответ.

Листинг кода:

```
class TreeNode: #класс для представления узлов дерева.
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.size = 1 # размер поддерева

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

    def _insert(self, node, key): #Рекурсивный метод для вставки нового ключа в дерево
        if node is None: #Если текущий узел None, создаем новый узел с ключом и возвращаем его
            return TreeNode(key)
        if key < node.key: #Если key меньше node.key, рекурсивно вставляем ключ в левое поддерево
            node.left = self._insert(node.left, key)
        elif key > node.key: #Если key больше node.key, вставляем в правое поддерево
            node.right = self._insert(node.right, key)
        else: #Если ключ уже существует, просто возвращаем текущий узел (ничего не добавляем)
            return node

        # Обновляем размер поддерева
        node.size = 1 + self._get_size(node.left) + self._get_size(node.right)
        return node

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _get_size(self, node): #Возвращает размер поддерева узла
        if node is None:
            return 0
        return node.size

    def _kth_smallest(self, node, k):
        if node is None:
            return None

        left_size = self._get_size(node.left) #Получаем размер левого поддерева текущего узла с помощью _get_size
```

```

        if k <= left_size:#Если k меньше или равно left_size, рекурсивно ищем k-й элемент в
        левом поддереве
            return self._kth_smallest(node.left, k)
        elif k == left_size + 1:#Если k равно left_size + 1, это значит, что текущий узел –
        k-й элемент, и мы возвращаем его ключ
            return node.key
        else:#Если k больше left_size + 1, продолжаем искать в правом поддереве, при этом
        меняя значение k
            return self._kth_smallest(node.right, k - left_size - 1)

    def kth_smallest(self, k):
        return self._kth_smallest(self.root, k)

bst = BST()
results = []
with open('input.txt', 'r') as input_file:
    for line in input_file:
        command = line.strip().split()
        if command[0] == '+':
            x = int(command[1])
            bst.insert(x)
        elif command[0] == '?':
            k = int(command[1])
            result = bst.kth_smallest(k)
            results.append(result)

with open('output.txt', 'w') as output_file:
    for result in results:
        output_file.write(f"{result}\n")

```

Текстовое объяснение:

TreeNode– класс для представления узлов дерева, key хранит значение, которое хранится в узле; left и right– указатели на левого и правого ребенка соответственно. Изначально они равны None, поскольку узел не имеет детей; size хранит размер поддерева, корнем которого является этот узел. Изначально установлен в 1, так как сам узел считается частью своего поддерева. BST– класс представляет бинарное дерево поиска, где root–указатель на корень дерева. Изначально он равен None, так как дерево еще не создано.

Метод _insert–рекурсивный метод для вставки нового ключа в дерево.

Если текущий узел None, создаем новый узел с ключом и возвращаем его.

Если key меньше node.key, рекурсивно вставляем ключ в левое поддерево.

Если key больше node.key, вставляем в правое поддерево.

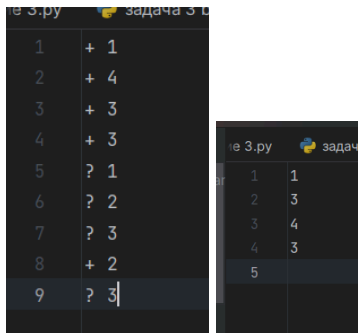
Если ключ уже существует, просто возвращаем текущий узел (ничего не добавляем).

Обновление размера поддерева: После вставки нового узла мы обновляем размер текущего узла, суммируя размеры левого и правого поддеревьев, плюс один для самого узла. Рекурсивный метод kth_smallest для поиска k-го по возрастанию элемента. Если узел None, возвращаем None.

Получаем размер левого поддерева текущего узла с помощью `_get_size`.

Сравниваем `k` с `left_size`:

Если `k` меньше или равно `left_size`, рекурсивно ищем `k`-й элемент в левом поддереве. Если `k` равно `left_size + 1`, это значит, что текущий узел — `k`-й элемент, и мы возвращаем его ключ. Если `k` больше `left_size + 1`, продолжаем искать в правом поддереве, при этом корректируем значение `k`.
Результат работы кода на значениях из задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	20Mib
Верхняя граница диапазона значений входных данных из текста задачи(500000)	1.68сек.	95.7Mib

Вывод по задаче: программа осуществляет простую и эффективную реализацию бинарного дерева поиска для решения задач с неявными ключами.

Вывод: Решая эту лабораторную, я научилась работать со бинарными деревьями поиска и сбалансированные деревья поиска AVL