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

Отчет по лабораторной работе №2 (семестр 2)
по курсу «Алгоритмы и структуры данных»
Тема: Двоичные деревья поиска

Выполнил:
Нгуен Хыу Жанг
Мобильные и сетевые технологии
К3140

Проверила:
Петросян Анна Мнацакановна

Санкт-Петербург
2025 г

Содержание

Вариант 28

Задание 3 : Простейшее BST

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.** $1 \le x \le 10^9$, $1 \le N \le 300000$
- **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.
- Пример:

input.txt	output.txt
+ 1 + 3 + 3 > 1 > 2 > 3 + 2 > 1	3 3 0 2

```
import os
import sys

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

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

    def insert(self, key):
        if self.root is None:
            self.root = BSTNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
```

```
            if node.left is None:
                node.left = BSTNode(key)
            else:
                self._insert(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = BSTNode(key)
            else:
                self._insert(node.right, key)

    def find_min_greater_than(self, key):
        return self._find_min_greater_than(self.root, key)

    def _find_min_greater_than(self, node, key):
        if node is None:
            return 0
        if node.key <= key:
            return self._find_min_greater_than(node.right, key)
        left_result = self._find_min_greater_than(node.left, key)
        return node.key if left_result == 0 else left_result

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    bst = BST()
    results = []

    with open(input_path, 'r') as infile:
        for line in infile:
            command = line.strip().split()
            operation = command[0]
            if operation == '+':
                bst.insert(int(command[1]))
            elif operation == '>':
                result = bst.find_min_greater_than(int(command[1]))
                results.append(result)

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

if __name__ == '__main__':
    task_scheduler()
```

input.txt:

```
+ 1
+ 3
+ 3
> 1
> 2
> 3
+ 2
> 1
```

output.txt:

```
3
3
0
2
```

Объяснение кода задачи о простейшем BST:
- В данной задаче необходимо реализовать простейшую структуру данных — бинарное дерево поиска (BST), которое будет обрабатывать два типа запросов: добавление элемента и поиск минимального элемента, большего заданного значения.

Структура кода:

1. **Класс `BSTNode`**:
   - Это класс для представления узла дерева. Каждый узел содержит:
     - `key`: значение узла.
     - `left`: указатель на левое поддерево.
     - `right`: указатель на правое поддерево.
   - Конструктор инициализирует ключ и устанавливает левое и правое поддеревья в `None`.
2. **Класс `BST`**:
   - Это класс для представления бинарного дерева поиска. Он содержит:
     - `root`: корень дерева, изначально равен `None`.
   - Метод `insert(key)` добавляет ключ в дерево. Если дерево пустое, создается новый узел. Если нет, вызывается вспомогательный метод `_insert(node, key)` для рекурсивного добавления узла.
   - Метод `_insert(node, key)` сравнивает ключ с текущим узлом. Если ключ меньше, он рекурсивно вызывает себя для левого поддерева, если больше — для правого поддерева.
3. **Метод поиска минимального элемента**:
   - Метод `find_min_greater_than(key)` ищет минимальный элемент, больший заданного `key`. Он вызывает вспомогательный метод `_find_min_greater_than(node, key)`.
   - В методе `_find_min_greater_than(node, key)` выполняется следующее:
     - Если узел `None`, возвращается `0`, так как нет подходящих элементов.
     - Если ключ узла меньше или равен `key`, производится поиск в правом поддереве.
     - Если ключ больше, производится поиск в левом поддереве. Если найден подходящий элемент, он возвращается, иначе возвращается текущий узел.
4. **Функция `task_scheduler`**:
   - Эта функция обрабатывает входные и выходные файлы. Она создает экземпляр класса `BST` и список для хранения результатов.
   - Читает входной файл построчно, разбивает каждую строку на команду и соответствующий параметр.
   - Если команда `+`, добавляет элемент в дерево. Если команда `>`, ищет и сохраняет результат.
5. **Запись результата в файл**:
   - После обработки всех запросов результаты записываются в выходной файл.
6. **Запуск программы**:
   - В конце кода проверяется, запущен ли скрипт напрямую, и если да, вызывается функция `task_scheduler`.

Примеры работы:

- Для входных данных:

```
+ 1
```

```
+ 3
+ 3
> 1
> 2
> 3
+ 2
> 1
```

- Выполненные операции приведут к следующим результатам:
  - `> 1` вернет `2` (минимальный элемент больше 1).
  - `> 2` вернет `3`.
  - `> 3` вернет `0` (нет элемента больше 3).
  - `> 1` снова вернет `2`.

Заключение:
- Данный код реализует простейшее бинарное дерево поиска для обработки запросов на добавление элементов и поиска минимальных значений, обеспечивая эффективное выполнение операций за O(log n) в среднем случае. Код способен обрабатывать до 300,000 запросов, что делает его эффективным и подходящим для заданных ограничений.

Задание 7 : Опознание двоичного дерева поиска (усложненная версия)

Эта задача отличается от предыдущей тем, что двоичное дерева поиска может содержать равные ключи.

Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Теперь, для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева **больше или равны** ключу вершины V.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

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

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

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

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

«INCORRECT» (без кавычек).

- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Примеры:

input.txt	output.txt	input.txt	output.txt	input.txt	output.txt	input.txt	output.txt
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT	3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT	3 2 1 2 1 -1 -1 2 -1 -1	CORRECT	3 2 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt	input.txt	output.txt	input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT	7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT	1 2147483647 -1 -1	CORRECT

- Примечание. Пустое дерево считается правильным двоичным деревом поиска. Дерево не обязательно должно быть сбалансировано. Попробуйте адаптировать алгоритм из предыдущей задачи к случаю, когда допускаются повторяющиеся ключи, и остерегайтесь целочисленного переполнения!

```python
import os

def is_bst(node, min_key, max_key, nodes):
    if node == -1:
        return True

    key, left, right = nodes[node]

    if key < min_key or key >= max_key:
        return False

    return (is_bst(left, min_key, key, nodes) and
            is_bst(right, key, max_key, nodes))

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as file:
        n = int(file.readline().strip())
        nodes = []

        for _ in range(n):
            line = list(map(int, file.readline().strip().split()))
            nodes.append(line)

    if is_bst(0, -2 ** 31, 2 ** 31 - 1, nodes):
        result = "CORRECT"
    else:
        result = "INCORRECT"
```

```
    with open(output_path, 'w') as file:
        file.write(result)

if __name__ == '__main__':
    task_scheduler()
```

input.txt:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

output.txt:

```
CORRECT
```

Объяснение кода задачи о проверке двоичного дерева поиска:

- В данной задаче требуется проверить, является ли заданное двоичное дерево поиска (BST) корректным с учетом того, что ключи могут повторяться. Для этого необходимо убедиться, что все узлы в левом поддереве меньше ключа узла, а все узлы в правом поддереве больше или равны ключу узла.

Структура кода:

1. **Функция `is_bst`**:
   - Эта рекурсивная функция проверяет, соответствует ли поддерево, начиная с узла `node`, условиям BST.
   - Параметры:
     - `node`: индекс текущего узла.
     - `min_key`: минимально допустимое значение для данного узла.
     - `max_key`: максимальное допустимое значение для данного узла.
     - `nodes`: список всех узлов дерева.
   - Если `node` равен `-1`, это означает, что узел отсутствует, и функция возвращает `True`.
   - Получаем ключ и индексы левого и правого потомков для текущего узла.
   - Проверяем, находится ли ключ в пределах допустимых значений (`min_key` и `max_key`). Если нет, возвращаем `False`.
   - Рекурсивно проверяем левое поддерево (с обновленным `max_key`) и правое поддерево (с обновленным `min_key`).
2. **Функция `task_scheduler`**:
   - Эта функция управляет вводом и выводом данных.
   - Считывает количество узлов n из файла и инициализирует список `nodes`.
   - Для каждого узла считывает ключ и индексы левого и правого детей, добавляя их в список `nodes`.
   - Вызывает функцию `is_bst` с корневым узлом (индекс 0) и диапазоном для допустимых значений ключа (от -2^{31} до $2^{31} - 1$).
   - В зависимости от результата проверки, записывает "CORRECT" или "INCORRECT" в выходной файл.
3. **Запуск программы**:

- В конце кода проверяется, запущен ли скрипт напрямую, и если да, вызывается функция `task_scheduler`.

Примеры работы:

- Для входных данных:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

- Структура дерева:

```
  2
 / \
1   3
```

- Дерево корректно, так как удовлетворяет условиям BST, и вывод будет "CORRECT".
- Для входных данных:

```
3
1 1 2
2 -1 -1
1 -1 -1
```

- Структура дерева:

```
  1
   \
    2
   /
  1
```

- Дерево некорректно, так как узел с ключом 1 в правом поддереве нарушает условия BST, и вывод будет "INCORRECT".

Заключение:

- Данный код эффективно проверяет корректность двоичного дерева поиска с учетом повторяющихся ключей, используя рекурсивный подход. Он обрабатывает до 100,000 узлов, что делает его подходящим для заданных ограничений по времени и памяти.

Задание 11 : Сбалансированное двоичное дерево поиска

Реализуйте сбалансированное двоичное дерево поиска.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание операций с деревом, их количе- ство N не превышает 10^5. В каждой строке находится одна из следующих операций:

– insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо;

– delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо;

– exists x – если ключ x есть в дереве выведите «true», если нет – «false»;

– next x – выведите минимальный элемент в дереве, строго больший x, или «none», если такого нет;

– prev x – выведите максимальный элемент в дереве, строго меньший x, или «none», если такого нет.

В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.

- **Ограничения на входные данные.** $0 \le N \le 10^5$, $|x_i| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите последовательно результат выполнения всех

опера- ций exists, next, prev. Следуйте формату выходного файла из примера.

- Ограничение по времени. 2 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
insert 2	true
insert 5	false
insert 3	5
exists 2	3
exists 4	none
next 4	3
prev 4	
delete 5	
next 4	
prev 4	

```
import os

class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def insert(self, root, key):
        if not root:
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        else:
            return root

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_balance(root)

        if balance > 1 and key < root.left.key:
            return self.rotate_right(root)
        if balance < -1 and key > root.right.key:
            return self.rotate_left(root)
        if balance > 1 and key > root.left.key:
            root.left = self.rotate_left(root.left)
            return self.rotate_right(root)
        if balance < -1 and key < root.right.key:
            root.right = self.rotate_right(root.right)
            return self.rotate_left(root)

        return root

    def delete(self, root, key):
        if not root:
            return root

        if key < root.key:
```

```
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            temp = self.get_min_value_node(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_balance(root)

        if balance > 1 and self.get_balance(root.left) >= 0:
            return self.rotate_right(root)
        if balance < -1 and self.get_balance(root.right) <= 0:
            return self.rotate_left(root)
        if balance > 1 and self.get_balance(root.left) < 0:
            root.left = self.rotate_left(root.left)
            return self.rotate_right(root)
        if balance < -1 and self.get_balance(root.right) > 0:
            root.right = self.rotate_right(root.right)
            return self.rotate_left(root)

        return root

    def exists(self, root, key):
        if not root:
            return False
        if key < root.key:
            return self.exists(root.left, key)
        elif key > root.key:
            return self.exists(root.right, key)
        else:
            return True

    def next(self, root, key):
        succ = None
        while root:
            if root.key > key:
                succ = root
                root = root.left
            else:
                root = root.right
        return succ.key if succ else 'none'

    def prev(self, root, key):
        pred = None
        while root:
            if root.key < key:
                pred = root
                root = root.right
            else:
                root = root.left
        return pred.key if pred else 'none'

    def rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
```

```
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def rotate_right(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def get_height(self, root):
        if not root:
            return 0
        return root.height

    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) - self.get_height(root.right)

    def get_min_value_node(self, root):
        if root is None or root.left is None:
            return root
        return self.get_min_value_node(root.left)

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    tree = AVLTree()
    root = None
    results = []

    with open(input_path, 'r') as f:
        for line in f:
            parts = line.strip().split()
            command = parts[0]
            if command == 'insert':
                root = tree.insert(root, int(parts[1]))
            elif command == 'delete':
                root = tree.delete(root, int(parts[1]))
            elif command == 'exists':
                results.append("true" if tree.exists(root, int(parts[1])) else "false")
            elif command == 'next':
                results.append(str(tree.next(root, int(parts[1]))))
            elif command == 'prev':
                results.append(str(tree.prev(root, int(parts[1]))))

    with open(output_path, 'w') as f:
        f.write("\n".join(results) + "\n")

if __name__ == '__main__':
    task_scheduler('input.txt', 'output.txt')
```

input.txt:

```
insert 2
insert 5
insert 3
exists 2
exists 4
next 4
prev 4
delete 5
next 4
prev 4
```

output.txt:

```
true
false
5
3
none
3
```

Объяснение кода задачи о сбалансированном двоичном дереве поиска:
- В данной задаче необходимо реализовать сбалансированное двоичное дерево поиска (АВЛ-дерево) и обрабатывать ряд операций, таких как вставка, удаление, проверка существования, нахождение следующего и предыдущего элемента.

Структура кода:

1. **Класс `TreeNode`**:
   - Этот класс представляет узел дерева и содержит:
     - `key`: значение узла.
     - `left`: указатель на левое поддерево.
     - `right`: указатель на правое поддерево.
     - `height`: высота узла, необходимая для балансировки дерева.
2. **Класс `AVLTree`**:
   - Этот класс реализует методы для работы с АВЛ -деревом.
   - **Метод `insert`**:
     - Рекурсивно добавляет ключ в дерево. Если узел с таким ключом уже существует, ничего не происходит.
     - После вставки обновляется высота узла и выполняется проверка на балансировку. Если дерево не сбалансировано, выполняются соответствующие повороты.
   - **Метод `delete`**:
     - Рекурсивно удаляет узел с указанным ключом. Если узел не найден, ничего не происходит.
     - После удаления обновляется высота узла и выполняется проверка на балансировку.
   - **Метод `exists`**:

        - Проверяет, существует ли узел с заданным ключом в дереве. Возвращает `True` или `False`.
    - **Метод `next`**:
        - Находит минимальный ключ, строго больший заданному. Если такого нет, возвращается "none".
    - **Метод `prev`**:
        - Находит максимальный ключ, строго меньший заданному. Если такого нет, возвращается "none".
    - **Методы для балансировки дерева**:
        - `rotate_left` и `rotate_right` выполняют повороты, чтобы сбалансировать дерево.
    - **Вспомогательные методы**:
        - `get_height` возвращает высоту узла.
        - `get_balance` вычисляет баланс узла.
        - `get_min_value_node` находит узел с минимальным значением.
3. **Функция `task_scheduler`**:
    - Эта функция обрабатывает входные данные и управляет результатами операций.
    - Считывает команды из файла, выполняет соответствующие операции с деревом и сохраняет результаты для операций `exists`, `next`, и `prev`.
    - Результаты записываются в выходной файл.
4. **Запуск программы**:
    - В конце кода проверяется, запущен ли скрипт напрямую, и если да, вызывается функция `task_scheduler`.

Примеры работы:

- Для входных данных:

```
insert 2
insert 5
insert 3
exists 2
exists 4
next 4
prev 4
delete 5
next 4
prev 4
```

- Выполненные операции приведут к следующим результатам:
    - `exists 2` вернет `true`.
    - `exists 4` вернет `false`.
    - `next 4` вернет `3`.
    - `prev 4` вернет `2`.
    - После удаления `5`, `next 4` вернет `3`, а `prev 4` вернет `2`.

Заключение

- Код реализует эффективное сбалансированное двоичное дерево поиска (АВЛ - дерево), которое может обрабатывать большое количество операций за O(log n) благодаря балансировке. Это позволяет эффективно выполнять операции вставки, удаления и поиска, что делает его пригодным для решения задачи в заданных ограничениях.

Задание 14 : Вставка в АВЛ-дерево

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, произво- дится, 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.

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

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

  Все ключи различны. Гарантируется, что данное дерево является корректным АВЛ-деревом.

  В последней строке содержится число X – ключ вершины, которую требуется вставить в дерево. Гарантируется, что такой вершины в дереве нет.

- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$, $|X| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции вставки. Нумерация вершин может быть произвольной при условии соблюдения формата.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
2 3 0 2 4 0 0 5	3 4 2 3 3 0 0 5 0 0

- Проверить можно по ссылке, OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 3 задача.

```
import os
from collections import deque

class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

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

    def insert(self, key):
        if not self.root:
            self.root = AVLNode(key)
        else:
            self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if not node:
            return AVLNode(key)
        elif key < node.key:
            node.left = self._insert(node.left, key)
        else:
            node.right = self._insert(node.right, key)

        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1 and key < node.left.key:
            return self._right_rotate(node)
        if balance < -1 and key > node.right.key:
            return self._left_rotate(node)
        if balance > 1 and key > node.left.key:
            node.left = self._left_rotate(node.left)
            return self._right_rotate(node)
        if balance < -1 and key < node.right.key:
            node.right = self._right_rotate(node.right)
            return self._left_rotate(node)

        return node

    def _left_rotate(self, z):
        y = z.right
        T2 = y.left

        y.left = z
        z.right = T2

        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))

        return y
```

```
    def _right_rotate(self, z):
        y = z.left
        T3 = y.right

        y.right = z
        z.left = T3

        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))

        return y

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def level_order_traversal(self):
        if not self.root:
            return []

        result = []
        queue = deque()
        queue.append(self.root)

        while queue:
            node = queue.popleft()
            result.append(node)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        return result

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    base_dir = os.path.dirname(os.path.abspath(__file__))
    txtf_dir = os.path.join(base_dir, '..', 'txtf')

    os.makedirs(txtf_dir, exist_ok=True)

    input_path = os.path.join(txtf_dir, input_file)
    output_path = os.path.join(txtf_dir, output_file)

    with open(input_path, 'r') as f:
        N = int(f.readline())
        nodes = []
        for _ in range(N):
            K, L, R = map(int, f.readline().split())
            nodes.append((K, L, R))
        X = int(f.readline())

    avl_tree = AVLTree()
    for node in nodes:
        avl_tree.insert(node[0])

    avl_tree.insert(X)
    level_order = avl_tree.level_order_traversal()
```

```
    node_to_index = {node: i + 1 for i, node in enumerate(level_order)}

    with open(output_path, 'w') as f:
        f.write(f"{len(level_order)}\n")
        for node in level_order:
            left_index = node_to_index.get(node.left, 0)
            right_index = node_to_index.get(node.right, 0)
            f.write(f"{node.key} {left_index} {right_index}\n")

if __name__ == "__main__":
    task_scheduler()
```

input.txt:

```
2
3 0 2
4 0 0
5
```

output.txt:

```
3
4 2 3
3 0 0
5 0 0
```

Объяснение кода задачи о вставке в АВЛ-дерево:

- В данной задаче необходимо реализовать вставку узла в АВЛ-дерево. АВЛ-дерево — это самобалансирующееся бинарное дерево поиска, где для каждого узла разница высот его левого и правого поддеревьев не превышает 1.

Структура кода:

1. **Класс `AVLNode`**:
   - Этот класс представляет узел дерева и содержит:
     - `key`: значение узла.
     - `left`: указатель на левое поддерево.
     - `right`: указатель на правое поддерево.
     - `height`: высота узла, необходимая для балансировки дерева.
2. **Класс `AVLTree`**:
   - Этот класс реализует методы для работы с АВЛ-деревом.
   - **Метод `insert`**:
     - Этот метод вставляет новый узел с заданным ключом. Если дерево пустое, новый узел становится корнем.
     - Если узел с такой ключом уже существует, ничего не происходит.
     - После вставки обновляется высота узла и проверяется балансировка дерева.
   - **Метод `_insert`**:
     - Рекурсивно добавляет ключ в дерево, определяя, в какое поддерево (левое или правое) нужно вставить новый узел.
     - После каждой вставки обновляется высота узла и выполняется проверка на балансировку.
   - **Методы для балансировки**:
     - `rotate_left` и `rotate_right` выполняют повороты для восстановления сбалансированности дерева.

- **Методы для получения высоты и баланса**:
  - `_get_height` возвращает высоту узла.
  - `_get_balance` вычисляет баланс узла, определяя разницу высот его поддеревьев.
- **Метод `level_order_traversal`**:
  - Этот метод выполняет обход дерева в ширину (уровневый обход) и возвращает список узлов в порядке обхода.

3. **Функция `task_scheduler`**:
   - Эта функция управляет вводом и выводом данных.
   - Считывает количество узлов N из файла и информацию о каждом узле, добавляя их в список nodes.
   - Считывает ключ `X`, который необходимо вставить в дерево.
   - Создает экземпляр класса `AVLTree` и вставляет в него узлы из списка `nodes`.
   - Затем вставляет новый узел с ключом `X`.
   - Выполняет обход дерева и собирает узлы в порядке обхода.
4. **Запись результата в файл**:
   - После вставки нового узла результаты записываются в выходной файл в нужном формате.

Примеры работы:

- Для входных данных:

```
2
3 0 2
4 0 0
5
```

- Описание:

- **Количество узлов**: 2
- **Узлы**:
  - Узел 0: ключ 3, правый ребенок узел 2.
  - Узел 1: ключ 4, без детей.
- **Ключ для вставки**: 5

- Структура дерева до вставки:

```
  3
   \
    4
```

- Процесс вставки:

1. Вставляем 5.
2. Начинаем с 3: 5 > 3 → идем вправо.
3. В узле 4: 5 > 4 → вставляем 5 как правого ребенка.

- Структура дерева после вставки:

```
  3
   \
    4
     \
      5
```

- Результат:

```
3
3 0 2
```

```
4 0 3
5 0 0
```

- Объяснение результата:

- **Количество узлов**: 3.
- **Узлы**:
  - Узел 0: ключ 3, правый ребенок узел 2.
  - Узел 1: ключ 4, правый ребенок узел 3.
  - Узел 2: ключ 5, без детей.

- Таким образом, вывод будет корректен после вставки.

Заключение:

- Данный код реализует вставку в АВЛ-дерево, обеспечивая его сбалансированность после каждой операции. Это позволяет эффективно выполнять операции вставки, что делает структуру данных подходящей для работы с большими объемами данных и обеспечивающей быструю производительность в рамках заданных ограничений.

Задание 16 : *K*-й максимум

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 ki-го мак- симума, он существует.
- **Ограничения на входные данные.** $n \le 100000$, $|k_i| \le 10^9$.
- **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.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 512 мб.
- Пример:

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

```python
import os
import heapq

class KthMaximum:
    def __init__(self):
        self.elements = set()
        self.max_heap = []

    def add(self, key):
        if key not in self.elements:
            self.elements.add(key)
            heapq.heappush(self.max_heap, -key)

    def remove(self, key):
        if key in self.elements:
            self.elements.remove(key)

    def get_kth_maximum(self, k):
        temp_heap = []
        for _ in range(k):
            while True:
                max_elem = -heapq.heappop(self.max_heap)
                if max_elem in self.elements:
                    temp_heap.append(max_elem)
                    break

        kth_maximum = temp_heap[-1]

        for elem in temp_heap:
            heapq.heappush(self.max_heap, -elem)

        return kth_maximum

def task_scheduler(input_file='input.txt', output_file='output.txt'):
    input_path = os.path.join('..', 'txtf', input_file)
    output_path = os.path.join('..', 'txtf', output_file)

    with open(input_path, 'r') as f:
        n = int(f.readline().strip())
        kth_maximum = KthMaximum()
        results = []

        for _ in range(n):
            command = list(map(int, f.readline().strip().split()))
            c, k = command[0], command[1]

            if c == 1:
                kth_maximum.add(k)
            elif c == 0:
                result = kth_maximum.get_kth_maximum(k)
                results.append(result)
            elif c == -1:
                kth_maximum.remove(k)

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

if __name__ == "__main__":
    task_scheduler()
```

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
```

Объяснение кода задачи о *K*-й максимуме:

- В данной задаче необходимо реализовать структуру данных, которая позволяет добавлять и удалять элементы, а также находить k-й максимум среди добавленных элементов. Для этого используется комбинация множества и кучи.

Структура кода:

1. **Класс `KthMaximum`**:
   - Этот класс управляет добавлением, удалением элементов и поиском k-го максимума.
   - **Атрибуты**:
     - `elements`: множество для хранения уникальных элементов, чтобы избежать дублирования.
     - `max_heap`: максимальная куча (реализована через `heapq`, который по умолчанию является мин-кучей, поэтому значения хранятся с отрицательным знаком).
2. **Метод `add(self, key)`**:
   - Добавляет элемент key в структуру, если он еще не существует.
   - Уникальность обеспечивает множество `elements`, а добавление в кучу — `max_heap`.
3. **Метод `remove(self, key)`**:
   - Удаляет элемент key из множества `elements`. Куча не очищается от этого элемента, так как `heapq` не поддерживает удаление произвольного элемента. Вместо этого при поиске k-го максимума проверяются элементы на наличие в множестве.

4. **Метод `get_kth_maximum(self, k)`**:
    - Находит k-й максимум, извлекая k элементов из кучи.
    - Создаётся временный список `temp_heap`, куда помещаются k максимальных элементов, которые существуют в `elements`.
    - После нахождения k-го максимума элементы из `temp_heap` возвращаются обратно в кучу для дальнейшего использования.
5. **Функция `task_scheduler`**:
    - Управляет вводом и выводом данных.
    - Считывает количество команд и обрабатывает каждую команду.
    - В зависимости от типа команды выполняются операции добавления, удаления или поиска k-го максимума.
    - Результаты запросов на k-й максимум собираются и записываются в выходной файл.
6. **Запуск программы**:
    - В конце кода проверяется, запущен ли скрипт напрямую, и если да, вызывается функция `task_scheduler`.

Примеры работы:

Для входных данных:

```
11
+1 5
+1 3
+1 7
0 1
0 2
0 3
-1 5
+1 10
0 1
0 2
0 3
```

- **Команды**:
    - `+1 5`: добавляем `5`.
    - `+1 3`: добавляем `3`.
    - `+1 7`: добавляем `7`.
    - `0 1`: ищем **1-й** максимум (`7`).
    - `0 2`: ищем **2-й** максимум (`5`).
    - `0 3`: ищем **3-й** максимум (`3`).
    - `-1 5`: удаляем `5`.
    - `+1 10`: добавляем `10`.
    - `0 1`: ищем **1-й** максимум (`10`).
    - `0 2`: ищем **2-й** максимум (`7`).
    - `0 3`: ищем **3-й** максимум (`3`).

Заключение:

- Код эффективно управляет добавлением, удалением и поиском k-го максимума с использованием множества и кучи, что обеспечивает высокую производительность для больших объемов данных.