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

Отчет по лабораторной работе №2

по курсу «Алгоритмы и структуры данных»

Тема: Двоичные деревья поиска

Вариант 4

Выполнила:

Буй Тхук Хуен - К3139

Проверила:

Петросян А.М.

Санкт-Петербург

2024 г.

Содержание отчета

I. Задачи по варианту

Задача №4. Простейший неявный ключ

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.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \le x \le 10^9$, $1 \le N \le 300000$, в запросах «? k», число k от 1 до количества элементов в дереве.
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «? k» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

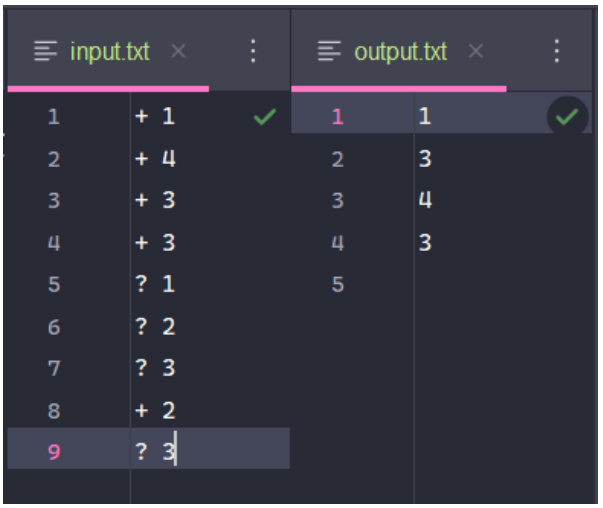
- Листинг кода

```
from Lab2.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'


def create_node(value):
    return {'value': value, 'size': 1, 'left': None, 'right': None}


def update_size(node):
    if node:
        node['size'] = 1
        if node['left']:
            node['size'] += node['left']['size']
        if node['right']:
            node['size'] += node['right']['size']


def insert(root, x):
    if not root:
        return create_node(x)

    current = root
    while True:
        if x < current['value']:
            if current['left'] is None:
                current['left'] = create_node(x)
                break
            current = current['left']
```

```
        elif x > current['value']:
            if current['right'] is None:
                current['right'] = create_node(x)
                break
            current = current['right']
        else:
            break  # x already exists in the tree

    # Update sizes
    current = root
    while current:
        update_size(current)
        if x < current['value']:
            current = current['left']
        elif x > current['value']:
            current = current['right']
        else:
            break

    return root


def kth_smallest(root, k):
    stack = []
    current = root
    while current or stack:
        while current:
            stack.append(current)
            current = current['left']

        current = stack.pop()
        k -= 1
        if k == 0:
            return current['value']

        current = current['right']

    return None

def process_queries(queries):
    root = []
    results = []
    for operation, value in queries:
        if operation == '+':
            root = insert(root, value)
        elif operation == '?':
            results.append(kth_smallest(root, value))
    return results

def main():
    queries = read(PATH_INPUT, 4)
    results = process_queries(queries)
    write(PATH_OUTPUT, results, 4)
```

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

- Текстовое объяснение решения
  + класса TreeNode: Создать узел двоичного дерева поиска (BST). Характеристики:
    value: значение узла.
    size: Количество узлов в поддереве, корнем которого является данный узел.
    left: Левый дочерний элемент узла.
    справа: Правый дочерний элемент узла.
  + функция update_size() обновляет размер поддерева для данного узла.
  + функция insert() вставляет новый элемент `x` в бинарное дерево поиска (BST). Если дерево пустое, создается новый узел. Если элемент уже существует, он не добавляется. После вставки обновляются размеры всех затронутых поддеревьев.
  + функция kth_smallest() находит k-й наименьший элемент в дереве. Она использует обход дерева в порядке in-order (левый, корень, правый) с помощью стека.
    Рассчитайте количество элементов в левом поддереве (left_size).
    Если k == left_size + 1, то текущим узлом является k-й элемент.
    Если k <= left_size, продолжаем поиск в левом поддереве.
    Если k > left_size + 1, выполнить поиск в правом поддереве с индексом k, измененным до k - left_size - 1.
  + функция process_queries() обрабатывает список запросов. Каждый запрос — это пара `(операция, значение)`:
    - Если операция "+", значение добавляется в дерево.
    - Если операция "?", функция ищет k-й наименьший элемент (где `k = value`) и добавляет результат в список `results`.
- Результат работы код на примерах из текста задачи:



- Test:

```
Test Case: test_process_queries1
Execution Time = 0.00001350 s, Memory Usage = 0.2158 KB

Test Case: test_process_queries2
Execution Time = 0.00000600 s, Memory Usage = 0.1377 KB


Test Case: test_process_queries3
Ran 3 tests in 0.375s

OK

Execution Time = 0.18770710 s, Memory Usage = 211.7500 KB

Process finished with exit code 0
```

Тест	Время выполнения (s)	Затраты памяти (KB)
+5 +3 +7 ?1 ?2 ?3	0.00001350	0.2158
+5 +3 +7	0.00000600	0.1377

[('+', i) for i in range(1000)] +[('?', 500)]	0.18770710	211.7500

- Вывод
  - + Вставка элемента в BST выполняется за время O(h), где `h` высота дерева. В сбалансированном дереве это O(log n).
  - + Поиск k-го наименьшего элемента выполняется за O(h) с использованием in-order обхода.
  - + Код может быть легко расширен для поддержки других операций, таких как удаление элемента или поиск k-го наибольшего элемента.

Задача №9. Удаление поддеревьев

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 $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).

  Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

  В 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.** $1 \le N \le 2 \cdot 10^5, |K_i| \le 10^9, 1 \le M \le 2 \cdot 10^5$
- **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.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.

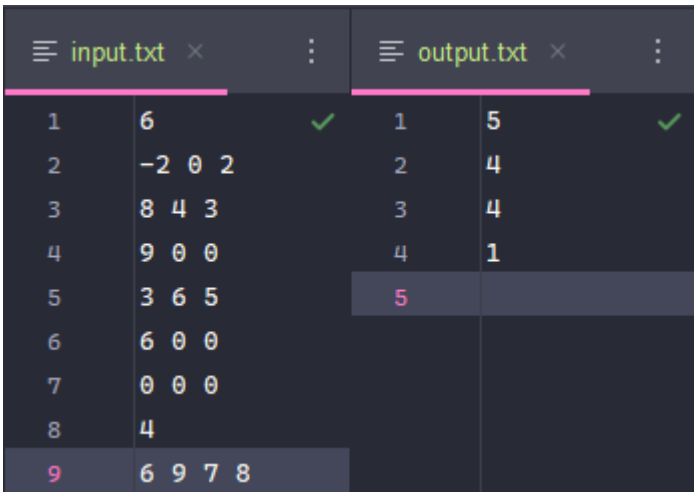
- Листинг кода

```
from Lab2.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def build_tree(N, nodes_data):
```

```
    nodes = [{'key': 0, 'left': None, 'right': None, 'size': 0}]

    for key, left, right in nodes_data:
        nodes.append({'key': key, 'left': None, 'right': None, 'size': 1})

    for i, (key, left, right) in enumerate(nodes_data, 1):
        if left != 0:
            nodes[i]['left'] = nodes[left]
        if right != 0:
            nodes[i]['right'] = nodes[right]

    update_sizes(nodes[1])
    return nodes[1]

def update_sizes(node):
    if not node:
        return 0
    node['size'] = 1 + update_sizes(node['left']) + update_sizes(node['right'])
    return node['size']

def delete_subtree(root, key):
    if not root:
        return root, 0

    if key < root['key']:
        root['left'], deleted_size = delete_subtree(root['left'], key)
        root['size'] -= deleted_size
        return root, deleted_size

    elif key > root['key']:
        root['right'], deleted_size = delete_subtree(root['right'], key)
        root['size'] -= deleted_size
        return root, deleted_size

    else:
        deleted_size = root['size']
        return None, deleted_size

def process_queries(root, queries):
    results = []
    currennt_size = root['size']

    for key in queries :
        _, delete_size = delete_subtree(root, key)
        currennt_size -= delete_size
        results.append(currennt_size)

    return results

def main():
    N, nodes_data, queries = read(PATH_INPUT, 9)
    root = build_tree(N, nodes_data)
    results = process_queries(root, queries)
```

```
    write(PATH_OUTPUT, results, 9)

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

- Текстовое объяснение решения:
  + Код реализует бинарное дерево поиска. Каждый узел (TreeNode) содержит:
    - key: значение узла
    - left и right: ссылки на левого и правого потомков
    - size: количество узлов в поддереве, включая текущий узел
  + Построение дерева (build_tree):
    - Создается словарь nodes, где ключ - это значение узла, а значение - объект TreeNode
    - Затем устанавливаются связи между узлами (left и right)
    - Корнем считается первый узел в списке nodes_data
    - После построения вызывается update_sizes для пересчета размеров поддеревьев
  + Обновление размеров (update_sizes):
    - Рекурсивно обходит дерево
    - Размер каждого узла = 1 (сам узел) + размер левого поддерева + размер правого поддерева
  + Удаление поддерева (delete_subtree):
    - Ищет узел с заданным ключом
    - Если найден, удаляет все поддерево, возвращая None и размер удаленного поддерева
    - Если не найден, возвращает исходное дерево и 0
    - При возврате обновляет размеры узлов на пути поиска
  + Обработка запросов (process_queries):
    - Для каждого запроса вызывает delete_subtree
    - Сохраняет размер оставшегося дерева после каждого удаления
- Результат работы код на примерах из текста задачи:



- Test

```
Test Case: test_case_1
Execution Time = 0.00002100 s, Memory Usage = 0.3672 KB

Test Case: test_case_2
Execution Time = 0.00001790 s, Memory Usage = 0.5391 KB

Test Case: test_case_3
Execution Time = 0.00001880 s, Memory Usage = 0.5859 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
3 5 2 3 3 0 0 7 0 0 2 3 7	0.00002100	0.3672
7 10 2 3 5 4 5 15 6 7 3 0 0 7 0 12 0 0	0.00001790	0.5391

17 0 0 4 5 15 3 17		
7 8 2 3 4 4 5 12 6 7 2 0 0 6 0 0 10 0 0 14 0 0 5 1 4 13 12 8	0.00001880	0.5859

- Вывод

  + Этот код эффективно решает задачу удаления поддеревьев из бинарного дерева поиска и отслеживания размера оставшегося дерева после каждой операции удаления.
  + Сложность:
    - Построение дерева: O(N)
    - Удаление поддерева: O(log N) в среднем, O(N) в худшем случае
    - Общая сложность: O(N + Q*log N), где Q — количество запросов.

Задача №15. Удаление из АВЛ-дерева

Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом:

- путем спуска от корня и проверки ключей находится V – удаляемая вершина;
- если вершина V – лист (то есть, у нее нет детей):
  - удаляем вершину;
  - поднимаемся к корню, начиная с бывшего родителя вершины V, при этом если встречается несбалансированная вершина, то производим поворот.
- если у вершины V не существует левого ребенка:
  - следовательно, баланс вершины равен единице и ее правый ребенок – лист;
  - заменяем вершину V ее правым ребенком;
  - поднимаемся к корню, производя, где необходимо, балансировку.
- иначе:
  - находим R – самую правую вершину в левом поддереве;
  - переносим ключ вершины R в вершину V;
  - удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являющегося листом);
  - поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку.

Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины - корня. Результатом удаления в этом случае будет пустое дерево.

Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как тестирующая система проверяет точное равенство получающихся деревьев.

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

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

  В последней строке содержится число X – ключ вершины, которую требуется удалить из дерева. Гарантируется, что такая вершина в дереве существует.
- **Ограничения на входные данные.** $1 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$, $|X| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции удаления. Нумерация вершин может быть произвольной при условии соблюдения формата.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.

- Листинг кода

```
from Lab2.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

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

```
        self.height = 1

def height(node):
    if not node:
        return 0
    return node.height

def balance_factor(node):
    if not node:
        return 0
    return height(node.right) - height(node.left)

def update_height(node):
    if not node:
        return
    node.height = max(height(node.left), height(node.right)) + 1

def rotate_right(node):
    y = node.left
    node.left = y.right
    y.right = node
    update_height(node)
    update_height(y)
    return y

def rotate_left(node):
    y = node.right
    node.right = y.left
    y.left = node
    update_height(node)
    update_height(y)
    return y

def balance(node):
    if not node:
        return node
    bf = balance_factor(node)
    if bf > 1:
        if balance_factor(node.right) < 0:
            node.right = rotate_right(node.right)
        return rotate_left(node)
    if bf < -1:
        if balance_factor(node.left) > 0:
            node.left = rotate_left(node.left)
        return rotate_right(node)
    return node

def find_max(node):
    current = node
    while current.right:
        current = current.right
    return current

def delete_node(root, key):
```

```
    # Удаление вершины
    if not root:
        return None
    if key < root.key:
        root.left = delete_node(root.left, key)
    elif key > root.key:
        root.right = delete_node(root.right, key)
    else:
        # Нашли вершину для удаления
        if not root.left and not root.right:  # Лист
            return None
        if not root.left:  # Нет левого ребёнка
            return root.right
        else:  # Есть оба ребёнка
            max_node = find_max(root.left)
            root.key = max_node.key
            root.left = delete_node(root.left, max_node.key)
    update_height(root)
    return balance(root)

def build_tree(nodes, index):
    # Построение дерева
    if index == 0 or index > len(nodes):
        return None
    key, left, right = nodes[index - 1]
    node = Node(key)
    node.left = build_tree(nodes, left)
    node.right = build_tree(nodes, right)
    update_height(node)
    return node

def collect_nodes(root, nodes_list):
    # Сбор оставшихся вершин
    if not root:
        return
    nodes_list.append(root)
    collect_nodes(root.left, nodes_list)
    collect_nodes(root.right, nodes_list)

def main():
    N, nodes, X = read(PATH_INPUT, 15)

    root = build_tree(nodes, 1)
    root = delete_node(root, X)
    nodes_list = []
    collect_nodes(root, nodes_list)

    # Присвоение новых индексов и подготовка вывода
    if not nodes_list:
        output = "0\n"
    else:
        node_to_index = {node: i + 1 for i, node in enumerate(nodes_list)}
        output = f"{len(nodes_list)}\n"
        for node in nodes_list:
```

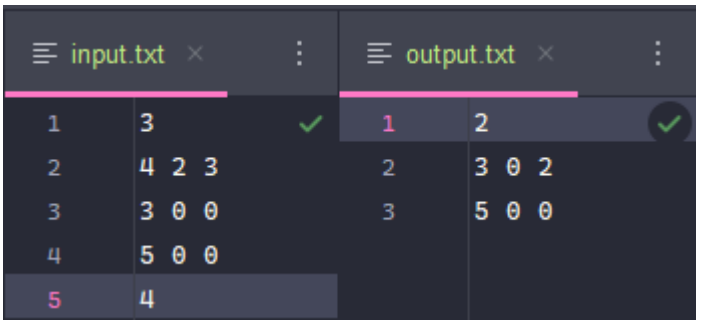
```
            left_idx = node_to_index.get(node.left, 0)
            right_idx = node_to_index.get(node.right, 0)
            output += f"{node.key} {left_idx} {right_idx}\n"

    # Запись результата

    write(PATH_OUTPUT, output, 15)

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

- Текстовое объяснение решения
  + Узел класса: Каждый узел имеет ключ (значение), левого потомка, правого потомка и высоту. Начальная высота устанавливается равной 1 для каждого нового узла.
  + Функция height(): Возвращает высоту кнопки, если кнопка пустая (None), то возвращает 0.
  + Функция update_height(): Обновить высоту узла на основе максимальной высоты двух дочерних элементов плюс 1.
  + Функция balance_factor() расчета коэффициента равновесия: Коэффициент баланса = высота правого поддерева - высота левого поддерева. Если этот коэффициент выходит за пределы диапазона [-1, 1], дерево будет несбалансированным.
  + Функция rotate_right(): Повернуть узел вправо, если левое поддерево имеет отрицательный коэффициент баланса (< -1). После поворота обновите высоту кнопок.
  + Функция rotate_left(): Повернуть узел влево, когда поддерево имеет положительный коэффициент баланса (> 1). После поворота обновите высоту.
  + Функция balance(): Перебалансируйте узел, если он несбалансирован (bf > 1 или bf < -1). При необходимости используйте одинарный поворот (rotate_left, rotate_right) или двойной поворот.
  + Функция find_max(): Используется для поиска наибольшего элемента в левом поддереве узла при выполнении удаления.
  + Функция delete_node():
    - Если удаляемый узел является конечным узлом → Удалить напрямую.

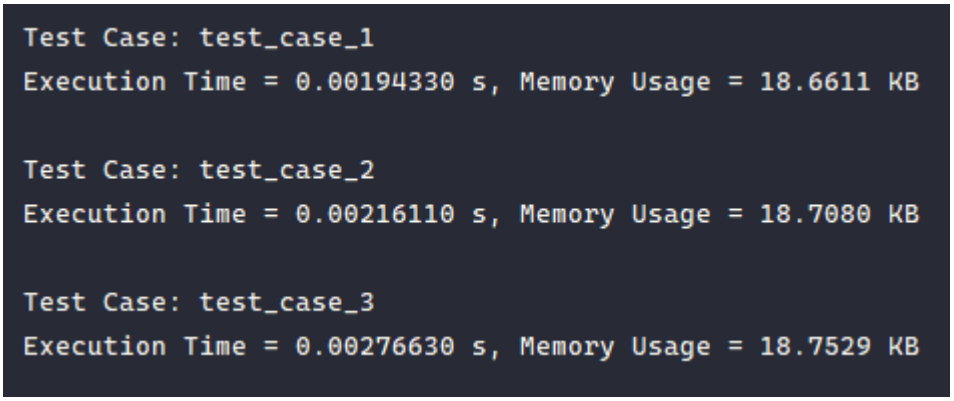
- Если у удаляемой кнопки есть дочерний элемент → Замените ее этим дочерним элементом.
- Если удаляемый узел имеет двух дочерних элементов → Найдите наибольший элемент в левом поддереве для замены, затем удалите этот элемент.

+ Функция build_tree():Построить дерево из списка узлов в соответствии со структурой ‘‘‘ key left_child_index right_child_index ’’’. Вызовите рекурсивно, чтобы создать дерево из заданного списка узлов.
+ Функция collect_nodes(): Предварительный заказ обхода для сбора узлов.

- Результат работы код на примерах из текста задачи



- Test



Тест	Время выполнения (s)	Затраты памяти (KB)
3 10 2 3 5 0 0 15 0 0 10	0.00194330	18.6611
5 20 2 3	0.00216110	18.7080

10 4 5 30 0 0 5 0 0 15 0 0 10		
7 50 2 3 30 4 5 70 6 7 20 0 0 40 0 0 60 0 0 80 0 0 30	0.00276630	18.7529

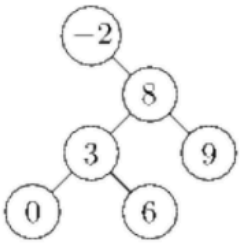
- Вывод
  + Эта программа обрабатывает AVL-дерево, удаляет узлы и обеспечивает сбалансированность дерева. Он реализует важные алгоритмы, такие как поворот дерева, балансировка дерева и поиск наибольшего элемента при удалении узла с двумя дочерними элементами.
  + Сложность:
    - В худшем случае весь алгоритм имеет сложность O(n).
    - Удаление элемента имеет сложность O(log n), но поскольку для этого требуется повторно собрать узлы и записать файл, общая сложность все еще составляет O(n)

II. Дополнительные задачи

Задача №8. Высота дерева возвращается

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды.

Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырем.



Дано **двоичное дерево поиска**. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Для каждой вершины дерева V выполняется следующее условие:

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

Найдите высоту данного дерева.

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

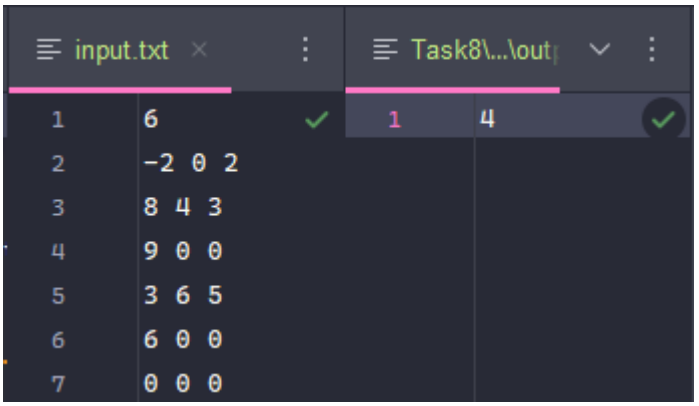
- Листинг кода

```python
from Lab2.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def build_tree(nodes, index):
    if index == 0:
        return None
    key, left, right = nodes[index - 1]
    return (key, build_tree(nodes, left), build_tree(nodes, right))

def tree_height(root):
    if not root:
        return 0
    _, left, right = root
    return 1 + max(tree_height(left), tree_height(right))
```

```
def main():
    n, nodes = read(PATH_INPUT, 8)
    if n == 0:
        height = 0
    else:
        root = build_tree(nodes, 1)
        height = tree_height(root)

    write(PATH_OUTPUT, height, 8)

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

- Текстовое объяснение решения:
  + Функция построения дерева build_tree:
    - nodes: Список узлов, каждый элемент имеет вид (key, left_index, right_index).
    - index: Положение корневого узла в списке.
    - Если index == 0, это означает, что узел не существует → Вернет None.
    - Получить значение key, левый подстрочный индекс left, правый подстрочный индекс `right` из списка.
    - Вызвать рекурсивно для построения левого поддерева и правого поддерева.
    - Возвращает кортеж, содержащий (key, left_subtree, right_subtree), то есть каждый узел будет представлен кортежем, а не классом Node.
  + Функция высоты дерева (tree_height):
    - Если дерево пустое (root == None), вернуть 0.
    - Получить левое поддерево и правое поддерево из root.
    - Использую рекурсию для вычисления высоты каждого поддерева.
    - Высота текущего узла = 1 + максимальная высота двух поддеревьев

- Результат работы кода:



- Test

```
Test Case: test_case_1
Execution Time = 0.00158080 s, Memory Usage = 18.3740 KB

Test Case: test_case_2
Execution Time = 0.00135390 s, Memory Usage = 18.3271 KB


Ran 3 tests in 0.013s

OK

Test Case: test_case_3
Execution Time = 0.00152100 s, Memory Usage = 18.2725 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
3 10 2 3 5 0 0 15 0 0	0.00158080	18.3740
5 20 2 3 10 4 5 30 0 0 5 0 0 15 0 0	0.00135390	18.3271
7 50 2 3	0.00152100	18.2725

Тест	Время выполнения (s)	Затраты памяти (KB)
30 4 5 70 6 7 20 0 0 40 0 0 60 0 0 80 0 0		

- Вывод
  + Алгоритм может эффективно работать на больших деревьях, если для хранения данных достаточно памяти.
  + Для простого расширения в более сложных приложениях можно использовать класс Node вместо кортежа.
  + Сложность времени: O(n)

Задача №10. Проверка корректности

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

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

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

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

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

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

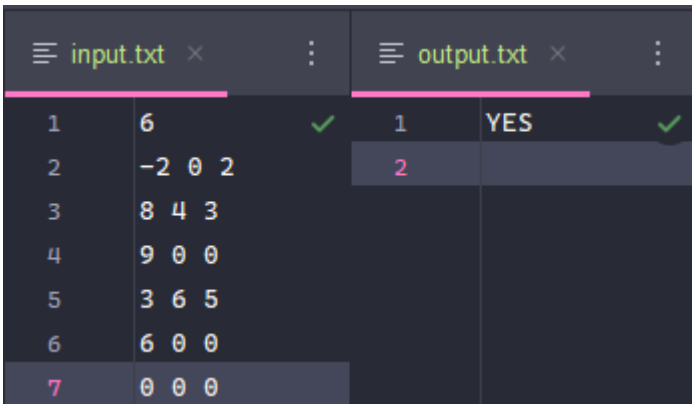
- Листинг кода

```
from Lab2.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def is_bst_util(nodes, index, min_val, max_val):
    if index == 0:
```

```
        return True
    key, left, right = nodes[index - 1]
    if not (min_val < key < max_val):
        return False
    return is_bst_util(nodes, left, min_val, key) and is_bst_util(nodes, right,
key, max_val)

def is_bst(n, nodes):
    if n == 0:
        return "YES"
    return "YES" if is_bst_util(nodes, 1, float('-inf'), float('inf')) else "NO"

def main():
    n, nodes = read(PATH_INPUT, 10)
    result = is_bst(n, nodes)

    write(PATH_OUTPUT, result, 10)

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

- Текстовое объяснение решения:
  + Функция is_bst_util(): Эта рекурсивная функция проверяет, является ли дерево BST или нет, следуя правилу:
    - Значение каждого узла должно находиться в допустимом диапазоне (min_val, max_val).
    - Значение левого дочернего узла должно быть меньше значения текущего узла.
    - Значение дочернего узла должно быть больше значения текущего узла.
    - Продолжает рекурсивную проверку обеих ветвей дерева.
  + Функция is_bst(): Эта функция вызывает is_bst_util для проверки BST. Если дерево пустое (n == 0), мы считаем его допустимым BST.

- Результат работы кода:



- Test

```
Test Case: test_case_1
Execution Time = 0.00000670 s, Memory Usage = 0.0625 KB

Test Case: test_case_2
Execution Time = 0.00000400 s, Memory Usage = 0.0938 KB

Test Case: test_case_3
Execution Time = 0.00000380 s, Memory Usage = 0.1094 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
0	0.00000670	0.0625
3 5 2 3 6 0 0 4 0 0	0.00000400	0.0938
5 20 2 3 10 4 5 30 0 0 5 0 0 25 0 0	0.00000380	0.1094

- Вывод
  + Программа проверяет, является ли дерево BST, обходя его в соответствии с правилами BST.

+ Временная сложность составляет O(n), что весьма оптимально, поскольку каждый узел посещается только один раз.
+ Может возникнуть ошибка переполнения стека, если дерево слишком глубокое (n очень большое и несбалансированное).

Задача №11. Сбалансированное двоичное дерево поиска

Реализуйте сбалансированное двоичное дерево поиска.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание операций с деревом, их количество 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 операций exists, next, prev. Следуйте формату выходного файла из примера.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 512 мб.

- Листинг кода

```
import sys
from Lab2.utils import read, write


PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'


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


def height(node):
```

```
    return node.height if node else 0


def update_height(node):
    if node:
        node.height = max(height(node.left), height(node.right)) + 1


def balance_factor(node):
    return height(node.right) - height(node.left) if node else 0


def rotate_right(y):
    x = y.left
    y.left = x.right
    x.right = y
    update_height(y)
    update_height(x)
    return x


def rotate_left(x):
    y = x.right
    x.right = y.left
    y.left = x
    update_height(x)
    update_height(y)
    return y


def balance(node):
    update_height(node)
    bf = balance_factor(node)
    if bf > 1:
        if balance_factor(node.right) < 0:
            node.right = rotate_right(node.right)
        return rotate_left(node)
    if bf < -1:
        if balance_factor(node.left) > 0:
```

```
            node.left = rotate_left(node.left)
        return rotate_right(node)
    return node

def insert(node, key):
    if not node:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    elif key > node.key:
        node.right = insert(node.right, key)
    return balance(node)

def find_min(node):
    while node.left:
        node = node.left
    return node

def delete(node, key):
    if not node:
        return None
    if key < node.key:
        node.left = delete(node.left, key)
    elif key > node.key:
        node.right = delete(node.right, key)
    else:
        if not node.left:
            return node.right
        if not node.right:
            return node.left
        min_larger_node = find_min(node.right)
        node.key = min_larger_node.key
        node.right = delete(node.right, min_larger_node.key)
    return balance(node)
```

```
def exists(node, key):
    if not node:
        return "false"
    if key == node.key:
        return "true"
    elif key < node.key:
        return exists(node.left, key)
    else:
        return exists(node.right, key)

def next(node, key):
    successor = None
    while node:
        if node.key > key:
            successor = node
            node = node.left
        else:
            node = node.right
    return successor.key if successor else "none"

def prev(node, key):
    predecessor = None
    while node:
        if node.key < key:
            predecessor = node
            node = node.right
        else:
            node = node.left
    return predecessor.key if predecessor else "none"

def main():
    root = None
    input_data = read(PATH_INPUT, 11)
```

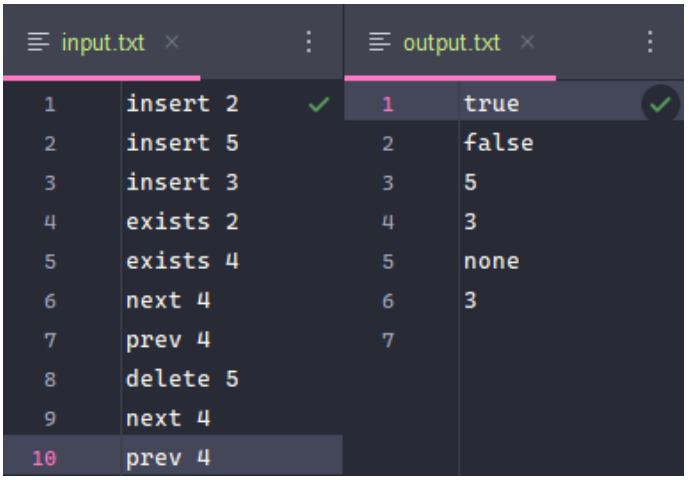
```
    output = []
    for line in input_data:
        parts = line.split()
        command, value = parts[0], int(parts[1])
        if command == "insert":
            root = insert(root, value)
        elif command == "delete":
            root = delete(root, value)
        elif command == "exists":
            output.append(exists(root, value))
        elif command == "next":
            output.append(str(next(root, value)))
        elif command == "prev":
            output.append(str(prev(root, value)))

    write(PATH_OUTPUT, output, 11)

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

- Текстовое объяснение решения
  + Класс Node, представляющий каждый узел в дереве. Каждый узел имеет следующие свойства:
    - key: Значение кнопки.
    - left: Указатель на левый дочерний узел.
    - справа: Указатель на правый дочерний узел.
    - высота: Высота кнопки (используется для расчета баланса).
  + Функции, связанные с высотой и равновесием дерева
    - height(node): Возвращает высоту узла. Если узел пуст, вернуть 0.
    - update_height(node): Обновить высоту узла, высота равна 1 + max(height(left), height(right)).
    - balance_factor(node): Возвращает коэффициент баланса узла, который представляет собой разницу между высотами правого поддерева и левого поддерева.

- rotate_right(y): Повернуть вправо, чтобы сохранить сбалансированность дерева, когда коэффициент баланса больше 1.
- rotate_left(x): Повернуть влево, чтобы сохранить сбалансированность дерева, когда коэффициент баланса меньше -1.
- balance(node): балансирует дерево, если коэффициент баланса превышает пороговое значение (т. е. возникает дисбаланс).

+ Основные операции:
  - insert(node, key): вставляет значение в дерево. После вставки функция вызовет balance, чтобы гарантировать, что дерево останется сбалансированным.
  - delete(node, key): удаляет узел из дерева. Если удаляемый узел имеет двух дочерних узлов, замените его узлом с наименьшим значением в правом поддереве (или наибольшим в левом поддереве), а затем удалите этот замененный узел. После удаления снова вызовите balance, чтобы сбалансировать дерево.
  - exists(узел, ключ): проверяет, существует ли значение в дереве. Возвращает «true», если существует, и «false» в противном случае.
  - next(node, key): находит наименьшее значение в дереве, большее значения ключа. Это операция по поиску последующего значения в бинарном дереве поиска.
  - prev(node, key): найти наибольшее значение в дереве, которое меньше значения ключа. Это операция по поиску предыдущего значения (предшественника) в бинарном дереве поиска.

- Результат работы кода:



- Test:

```
Test Case: test_case_1
Execution Time = 0.00304180 s, Memory Usage = 11.7207 KB

Test Case: test_case_2
Execution Time = 0.00192750 s, Memory Usage = 11.3135 KB


Ran 3 tests in 0.018s

OK

Test Case: test_case_3
Execution Time = 0.00194860 s, Memory Usage = 11.4043 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
insert 20 insert 30 exists 20 prev 25 next 15	0.00304180	11.7207
insert 30 insert 70 delete 30 exists 30 prev 50 next 50	0.00192750	11.3135
insert 200 insert 50 insert 150 delete 100 exists 100 prev 200 next 50	0.00194860	11.4043

- Вывод
  - \+ Алгоритм довольно прост в реализации и использует базовые концепции из теории деревьев поиска (двойное вращение для балансировки, минимизация количества уровней).
  - \+ Временная сложность: каждая операция (вставка, удаление, поиск) имеет сложность $O(\log n)$, где n — количество узлов в дереве.

III. Вывод

Основная цель лабораторной работы — изучение и отработка на практике структур данных двоичного дерева поиска (BST) и его сбалансированных вариантов, включая AVL-дерево и Splay-дерево.

Критерии	Характеристика	Преимущество	Недостатки
BST	- Каждый узел имеет максимум двух потомков: левого и правого. - Значение левого дочернего узла < значения родительского узла < значение правого дочернего узла	- Простота установки и понимания. - Поддерживает операции поиска, добавления, удаления со средней сложностью O(log n)	В худшем случае дерево может превратиться в связанный список со сложностью O(n)
AVL Tree	-Самобалансирующийся BST. - Поддерживает баланс, чтобы коэффициент баланса каждого узла находился в [-1, 1]. - При необходимости использует повороты для балансировки дерева.	- Всегда гарантирует сложность O(log n) для операций поиска, добавления и удаления. - Подходит для приложений, требующих быстрого и стабильного времени выполнения запросов.	- Установка сложнее, чем у обычного BST. - Стоимость балансировки дерева может увеличить время выполнения операций.
Splay Tree	-Саморегулирующийся BST. - После каждой операции поиска, добавления или удаления последний посещенный узел будет «splay»	- Самонастройка для оптимизации доступа к часто используемым элементам. - Средняя сложность операций составляет O(log n).	- В худшем случае сложность может достигать O(n). - Не гарантирует строгого баланса, как AVL Tree.

	(перемещаться в корень дерева). - Использует повороты, чтобы переместить узел в корень.		

- BST является основой для более сложных древовидных структур, таких как AVL и Splay.
- AVL Tree и Splay Tree — это два разных подхода к решению проблемы балансировки деревьев.
- AVL Tree фокусируется на поддержании строгого баланса, в то время как Splay Tree фокусируется на оптимизации доступа к часто используемым элементам.