

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 1

Выполнила:

Гайдук А. С.

К3241

Проверила:

Ромакина О. М.

Санкт-Петербург

2025 г.

Содержание отчета

Содержание отчета.....	2
Задачи по варианту	3
Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл]	3
Задача №12. Проверка сбалансированности [2s, 256 Mb, 2 балла]	6
Задача №17. Множество с суммой [120s, 512 Mb, 3 балла]	10
Дополнительные задачи	19
Задача №3. Простейшее BST [2 s, 256 Mb, 1 балл]	19
Задача №4. Простейший неявный ключ [2s, 256 Mb, 1 балл]	22
Задача №6. Оpozнание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]	26
Задача №7. Оpozнание двоичного дерева поиска (усложненная версия) [10s, 512 Mb, 2.5 балла]	30
Задача №13. Делаю я левый поворот... [2 s, 256 Mb, 3 балла]	35
Вывод	41

Задачи по варианту

Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл]

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), прямой (pre-order) и обратный (post-order). Очень полезно попрактиковаться в их реализации, чтобы лучше понять бинарные деревья поиска.

Вам дано корневое двоичное дерево. Выведите центрированный (in-order), прямой (pre-order) и обратный (post-order) обходы в глубину.

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

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

def inorder(root, result):
    if root is None:
        return
    inorder(root.left, result)
    result.append(str(root.key))
    inorder(root.right, result)

def preorder(root, result):
    if root is None:
        return
    result.append(str(root.key))
    preorder(root.left, result)
    preorder(root.right, result)

def postorder(root, result):
    if root is None:
        return
    postorder(root.left, result)
    postorder(root.right, result)
    result.append(str(root.key))

with open('input.txt', 'r') as infile:
    n = int(infile.readline().strip())
    node_data = [list(map(int, line.split())) for line in
infile.readlines()]

start_time = time.perf_counter()
```

```

nodes = [Node(key) for key, _, _ in node_data]

for i, (key, left_index, right_index) in enumerate(node_data):
    if left_index != -1:
        nodes[i].left = nodes[left_index]
    if right_index != -1:
        nodes[i].right = nodes[right_index]

inorder_result = []
preorder_result = []
postorder_result = []

inorder(nodes[0], inorder_result)
preorder(nodes[0], preorder_result)
postorder(nodes[0], postorder_result)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as outfile:
    outfile.write(' '.join(inorder_result) + '\n')
    outfile.write(' '.join(preorder_result) + '\n')
    outfile.write(' '.join(postorder_result) + '\n')

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала класс Node, который описывает узел бинарного дерева (значение узла, ссылки на левый и правый дочерние узлы). Далее я реализовала центрированный подход: рекурсивно обхожу сначала левое поддерево, затем текущий узел, потом правое поддерево. Прямой подход: сначала текущий узел, затем левое поддерево, правое поддерево. Обратный подход: сначала левое поддерево, затем правое поддерево, потом текущий узел. После чтения входных данных (количество узлов и их описание), я для каждого ключа создаю объект класса Node, а затем через цикл связываю узлы: если индекс дочернего узла не равен -1, устанавливается соответствующая ссылка (left/right). После этого выполняются три обхода дерева, начиная с корня, и результаты записываются в списки.

Результат работы кода на примерах из текста задачи:

```
task1.py  input.txt  output.txt
1      5
2      4 1 2
3      2 3 4
4      5 -1 -1
5      1 -1 -1
6      3 -1 -1
```

```
task1.py  input.txt  output.txt
1      1 2 3 4 5
2      4 2 1 3 5
3      1 3 2 5 4
```

```
task1.py  input.txt  output.txt
1      10
2      0 7 2
3      10 -1 -1
4      20 -1 6
5      30 8 9
6      40 3 -1
7      50 -1 -1
8      60 1 -1
9      70 5 4
10     80 -1 -1
11     90 -1 -1
```

```
task1.py  input.txt  output.txt
1      50 70 80 30 90 40 0 20 10 60
2      0 70 50 40 30 80 90 20 60 10
3      50 80 90 30 40 70 10 60 20 0
```

Результат работы кода на максимальных и минимальных значениях:

```
task1.py  input.txt  output.txt
1 1
2 0 -1 -1
```

```
task1.py  input.txt  output.txt
1 0
2 0
3 0
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000033 сек	0.040547 МВ
Пример из задачи	0.000044 сек	0.040771 МВ
Пример из задачи	0.000058 сек	0.041134 МВ

Вывод по задаче:

В данной задаче я реализовала алгоритм обхода дерева в глубину используя три подхода: центрированный, прямой и обратный. В этом мне помогли материалы лекций.

Задача №12. Проверка сбалансированности [2s, 256 Мб, 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.

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

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

def build_tree(nodes_data):
    nodes = {i: Node(key) for i, (key, left, right) in
nodes_data.items()}

    for i, (key, left, right) in nodes_data.items():
        if left != 0:
            nodes[i].left = nodes[left]
        if right != 0:
            nodes[i].right = nodes[right]

    return nodes[1], nodes

def tree_balance(node, balances, node_id_map):

    if not node:
        return 0

    left_height = tree_balance(node.left, balances, node_id_map)
    right_height = tree_balance(node.right, balances, node_id_map)
    current_height = 1 + max(left_height, right_height)
    balance = right_height - left_height
    balances[node_id_map[node]] = balance

    return current_height

with open('input.txt', 'r') as file:
    n = int(file.readline().strip())
    data = {}
    for i in range(1, n + 1):
        k, l, r = map(int, file.readline().strip().split())
        data[i] = (k, l, r)

start_time = time.perf_counter()
```

```

root, nodes = build_tree(data)

node_id_map = {node: node_id for node_id, node in nodes.items()}

balances = {}

tree_balance(root, balances, node_id_map)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

with open('output.txt', 'w') as file:
    for i in range(1, n + 1):
        file.write(f"{balances[i]}\n")

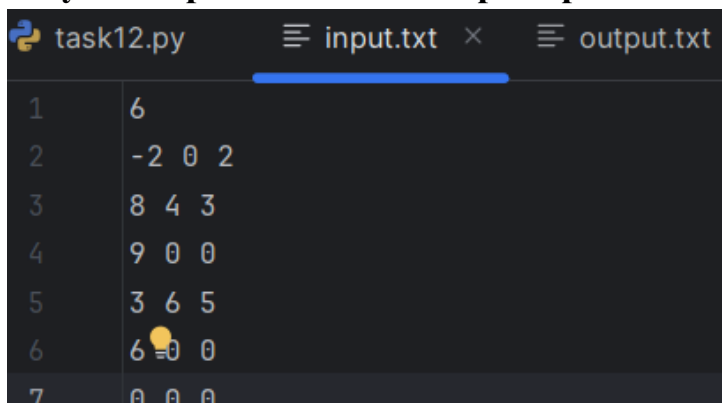
print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

На вход из файла я принимаю количество узлов и описание каждого узла. Я создала класс Node, а также метод для построения дерева. На вход он принимает словарь nodes_data, где ключ – номер узла, а значение – key, left, right. Для каждого узла создается объект класса, а также устанавливаются связи с детьми (если они есть). Возвращается корневой узел, а также словарь всех узлов. Функция tree_balance принимает на вход текущий узел, словарь для хранения балансов, а также сопоставление объекта узла с его номером в исходных данных. Если узла нет, возвращается 0, в ином случае рекурсивно вычисляются высоты левого и правого поддеревьев, а также высота текущего узла – это $1 + \max(\text{высота левого}, \text{высота правого})$. Баланс текущего узла – это разность высот правого и левого поддеревьев, его сохраняю в словарь balances с ключом, соответствующим номеру узла. Возвращается высота текущего узла.

Результат работы кода на примерах из текста задачи:



task12.py	input.txt	output.txt
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	


```
task12.py  input.txt  output.txt x
1 3
2 -1
3 0
4 0
5 0
6 0
```

Результат работы кода на максимальных и минимальных значениях:

```
task12.py  input.txt x  output.txt
⚠ The file size (3,18 MB) exceeds the configured limit (2,56 MB)
1 200000
2 1 2 3
3 2 4 5
```

```
task12.py  input.txt  output.txt x
1 0
2 0
3 -
4 0
5 0
```

```
task12.py  input.txt x  output.txt
1 1
2 1 0 0
```

```
task12.py  input.txt  output.txt x
1 0
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000027 сек	0.040351 МВ
Пример из задачи	0.000051 сек	0.040787 МВ
Верхняя граница диапазона значений входных данных из текста задачи	0.552134 сек	95.204036 МВ

Вывод по задаче:

Я реализовала алгоритм, вычисляющий баланс вершины, узнала, что такое АВЛ дерево.

Задача №17. Множество с суммой [120s, 512 Мб, 3 балла]

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

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

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

• Формат ввода / входного файла (input.txt). Изначально множество S пусто. Первая строка содержит n – количество операций. Следующие n строк содержат операции. Однако, чтобы убедиться, что ваше решение может работать в режиме онлайн, каждый запрос фактически будет зависеть от результата последнего запроса суммы. Обозначим $M = 1\,000\,000\,001$. В любой момент пусть x будет результатом последней операции суммирования или просто 0, если до этого операций суммирования не было. Тогда каждая операция будет являться одной из следующих:

- «+ i» – добавить некоторое число в множество S. Но не само число i, а число $((i + x) \bmod M)$.
- «- i» – удалить из множества S, т. е. $\text{del}((i + x) \bmod M)$.
- «? i» – $\text{find}((i + x) \bmod M)$.
- «s l r» – вывести сумму всех элементов множества S из определенного диапазона, т. е. $\text{sum}((l+x) \bmod M, (r+x) \bmod M)$.

Листинг кода:

```
import time
import tracemalloc
from collections import deque

tracemalloc.start()

class Node:
    def __init__(self, value, left=None, right=None, parent=None):
        self.value = value
        self.left = left
        self.right = right
        self.parent = parent
    def __str__(self):
        return f'{{self.value}}({self.parent}) -> [{{self.left}}, {{self.right}}]'
```

```
class Splay:
    def __init__(self, root=None):
        self.root = root
        self.x = 0
        self.M = 1000000001
        self.sum = []

    def splay(self, node):
        while node.parent != None:
            if node.parent.parent == None:
                if node == node.parent.left:
                    self.right_turn(node.parent)
                else:
                    self.left_turn(node.parent)
            elif node == node.parent.left and node.parent == node.parent.parent.left:
                self.right_turn(node.parent.parent)
                self.right_turn(node.parent)
            elif node == node.parent.right and node.parent == node.parent.parent.right:
                self.left_turn(node.parent.parent)
                self.left_turn(node.parent)
            elif node == node.parent.right and node.parent == node.parent.parent.left:
                self.left_turn(node.parent)
                self.right_turn(node.parent)
            else:
                self.right_turn(node.parent)
                self.left_turn(node.parent)

    def join(self, lt, rt):
        if lt == None:
```

```

        return rt
    if rt == None:
        return lt

    x = self.mx(lt)
    self.splay(x)
    x.right = rt
    rt.parent = x
    return x

def mx(self, tree):
    while tree.right != None:
        tree = tree.right
    return tree

def left_turn(self, node):
    y = node.right
    node.right = y.left
    if y.left != None:
        y.left.parent = node

    y.parent = node.parent
    if node.parent == None:
        self.root = y
    elif node == node.parent.left:
        node.parent.left = y
    else:
        node.parent.right = y

    y.left = node
    node.parent = y

def right_turn(self, node):
    y = node.left
    node.left = y.right
    if y.right != None:
        y.right.parent = node

    y.parent = node.parent
    if node.parent == None:
        self.root = y
    elif node == node.parent.right:
        node.parent.right = y
    else:
        node.parent.left = y

    y.right = node
    node.parent = y

def add(self, value):
    value = (value + self.x) % self.M
    if self.find_h(self.root, value) != None:
        return

    node = Node(value)
    y = None
    x = self.root
    while x != None:
        y = x
        if node.value < x.value:
            x = x.left

```

```

        else:
            x = x.right
        node.parent = y
        if y == None:
            self.root = node
        elif node.value < y.value:
            y.left = node
        else:
            y.right = node

        self.splay(node)

    def find_h(self, node, value):
        if node == None or value == node.value:
            return node
        elif value < node.value:
            return self.find_h(node.left, value)
        else:
            return self.find_h(node.right, value)

    def find(self, value):
        value = (value + self.x) % self.M
        el = self.find_h(self.root, value)
        if el != None:
            self.splay(el)
            return 'Found'
        return 'Not found'

    def delete(self, value):
        value = (value + self.x) % self.M
        node = self.root
        necessary_element = None
        lt = None
        rt = None

        while node != None:
            if node.value == value:
                necessary_element = node

            if node.value < value:
                node = node.right
            else:
                node = node.left

        if necessary_element == None:
            return 'Not found'

        self.splay(necessary_element)
        if necessary_element.right != None:
            rt = necessary_element.right
            rt.parent = None
        else:
            rt = None
        lt = necessary_element
        lt.right = None
        necessary_element = None

        if lt.left != None:
            lt.left.parent = None

        self.root = self.join(lt.left, rt)

```

```

lt = None

def summa(self, l, r):
    if self.root == None:
        return 0
    l = (l + self.x) % self.M
    r = (r + self.x) % self.M
    que = deque([])
    current = self.root
    que.append(current)
    s = 0
    while que:
        el = que.popleft()
        if l <= el.value <= r:
            s += el.value
        if current.left != None and current.left.value >= l:
            que.append(current.left)
        if current.right != None and current.right.value >= r:
            que.append(current.right)

        if len(que) > 0:
            current = que[0]
        else:
            self.x = s
            return s

with open('input.txt', 'r') as file:
    n = int(file.readline().strip())
    operations = [line.strip() for line in file]

start_time = time.perf_counter()

Splay_Tree = Splay()
p = open('output.txt', 'w')
for op in operations:
    if op.startswith('+'):
        _, x = op.split()
        result = Splay_Tree.add(int(x))
    elif op.startswith('-'):
        _, x = op.split()
        result = Splay_Tree.delete(int(x))
    elif op.startswith('?'):
        _, x = op.split()
        p.write(Splay_Tree.find(int(x)) + '\n')
    elif op.startswith('s'):
        _, l, r = op.split()
        p.write(str(Splay_Tree.summa(int(l), int(r))) + '\n')
p.close()

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое  
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я реализовала структуру данных на основе Splay Tree. Я создала класс Node и класс Splay, который содержит корневой узел дерева, последнюю вычисленную сумму x , константу M для взятия остатка по модулю и массив для хранения промежуточных сумм sum . Далее я реализовала методы вращения – `left_turn`, `right_turn`, которые выполняют повороты относительно узла. Левый поворот переносит правого потомка узла на его место и обновляет связи между родителями и детьми. Правый – аналогично. Эти методы нужны для балансировки дерева в процессе операций.

Метод `splay` перемещает указанный узел к корню дерева: если узел является левым/правым ребенком – выполняются простые повороты – `zig/zag`. Если узел на более глубоком уровне, выполняются двойные повороты – `zig-zig`, `zag-zag`, `zig-zag`, `zag-zig`.

Метод `join` объединяет два поддерева: максимальный элемент из левого поддерева становится корнем, правое поддерево становится правым ребенком корня. Если какого-то из поддеревьев нет, то просто возвращается второе.

Метод `add` добавляет новое значение в дерево: модифицирует значение числа с учетом $(x + \text{last_sum}) \% M$, если узел с таким значением уже существует, то операция завершается, если нет – создает узел и находит подходящее место для него в соответствии с его значением, создает ссылки ребенок-родитель. После перемещает узел в корень с помощью функции `splay`.

Поиск узла `find` работает так: если узел найден, он перемещается к корню. Поиск элемента осуществляется за счет перемещений влево и вправо. Возвращается Found или Not Found.

Метод `delete` удаляет узел из дерева: если узел найден, он перемещается к корню (`splay`) и дерево разбивается на два поддерева – левое и правое. Эти поддерева объединяются с помощью метода `join`.

Метод `summa` находит сумму значений в диапазоне $[l, r]$: модифицирует границы диапазона с учетом $(l + x) \% M$, $(r + x) \% M$, обходит дерево в ширину и проверяет, попадает ли значение узла в диапазон. Затем суммирует подходящие значения и обновляет x .

Результат работы кода на примерах из текста задачи:

```
task17.py  input.txt  output.txt
1      15
2      ? 1
3      + 1
4      ? 1
5      + 2
6      s 1 2
7      + 1000000000
8      ? 1000000000
9      - 1000000000
10     ? 1000000000
11     s 999999999 1000000000
12     - 2
13     ? 2
14     - 0
15     + 9
16     s 0 9
```

```
task17.py  input.txt  output.txt
1      Not found
2      Found
3      3
4      Found
5      Not found
6      1
7      Not found
8      10
```



```
task17.py  input.txt  output.txt
1      5
2      ? 0
3      + 0
4      ? 0
5      - 0
6      ? 0
```

```
task17.py  input.txt  output.txt
1      Not found
2      Found
3      Not found
```

```
task17.py  input.txt  output.txt
1      5
2      + 491572259
3      ? 491572259
4      ? 899375874
5      s 310971296 877523306
6      + 352411209
```

```
task17.py  input.txt  output.txt
1      Found
2      Not found
3      491572259
```

Результат работы кода на максимальных и минимальных значениях:

```
task17.py  input.txt  output.txt
1      1
2      ? 1
```

```
task17.py  input.txt  output.txt
1      Not found
```

```
task17.py  input.txt  output.txt
1 100000
2 + 1
3 - 2
4 ? 3
5 s 4 1004
```

```
task17.py  input.txt  output.txt
1 Not found
2 p
3 Not found
4 0
5 Not found
6 0
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000345 сек	0.044411 MB
Пример из задачи	0.000348 сек	0.045389 MB
Пример из задачи	0.000450 сек	0.044684 MB
Пример из задачи	0.000486 сек	0.044802 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.665316 сек	9.021780 MB

Вывод по задаче:

Я реализовала структуру данных для хранения набора чисел и быстрого вычисления суммы элементов в заданном диапазоне, используя splay дерево.

Дополнительные задачи

Задача №3. Простейшее BST [2 s, 256 Mb, 1 балл]

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

- «+ x» – добавить в дерево x (если x уже есть, ничего не делать).
- «> x» – вернуть минимальный элемент больше x или 0, если таких нет.

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

class Node:
    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 = Node(key)
            return
        node = self.root
        while True:
            if key < node.key:
                if node.left is None:
                    node.left = Node(key)
                    return
                node = node.left
            elif key > node.key:
                if node.right is None:
                    node.right = Node(key)
                    return
                node = node.right
            else:
                return

    def find_min_gt(self, key):
        node = self.root
        min_gt = 0
        while node is not None:
            if node.key > key:
                min_gt = node.key
                node = node.left
            else:
                node = node.right
        return min_gt
```

```

with open('input.txt', 'r') as f:
    queries = [line.strip() for line in f.readlines()]

start_time = time.perf_counter()

bst = BST()
results = []

for query in queries:
    if query.startswith("+"):
        x = int(query[1:])
        bst.insert(x)
    elif query.startswith(">"):
        x = int(query[1:])
        min_gt = bst.find_min_gt(x)
        results.append(str(min_gt))

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

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

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала класс Node, который описывает узел BST, а также класс BST с указателем на корень дерева.

Метод insert осуществляет вставку в дерево: если дерево пустое, создаем новый узел и делаем его корнем. Иначе, начинается поиск места для вставки:

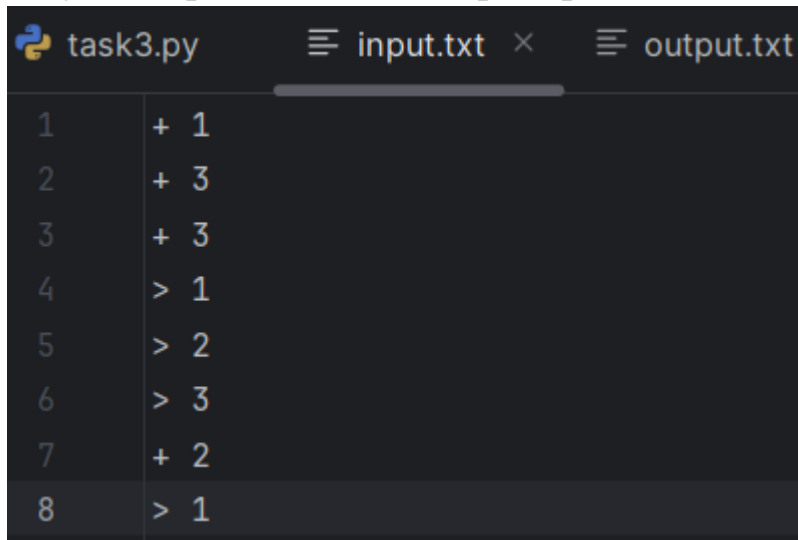
- 1) если ключ меньше текущего узла, переходим в левое поддерево;
- 2) если ключ больше текущего узла, переходим в правое поддерево;
- 3) если ключ равен текущему значению, ничего не делаем, т. к. это будет дублированием.

Когда находим подходящее место, вставляем новый узел.

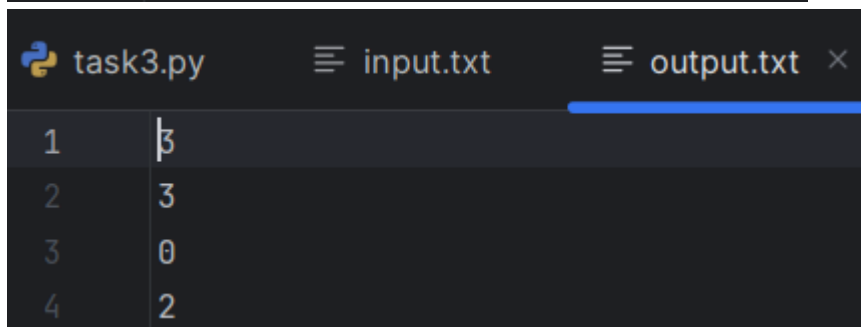
Метод find_min_gt находит минимальный элемент, больше заданного x. Поиск начинается с корня, инициализируем переменную min_gt = 0, которая хранит минимальное значение больше x. Если значение текущего узла больше ключа, обновляем min_gt и переходим в левое поддерево (там меньшие значения, которые тоже могут быть больше x). Если значение текущего узла меньше или равно ключу, переходим в правое поддерево (только там могут быть элементы больше x).

При чтении исходного файла все запросы я сохраняю в список `queries`, а затем прохожусь циклом по этим запросам: если запрос начинается с «+», я вызываю метод `insert`, если с «>», я вызываю метод `find_min_gt` и результат добавляю в список `results`.

Результат работы кода на примерах из текста задачи:

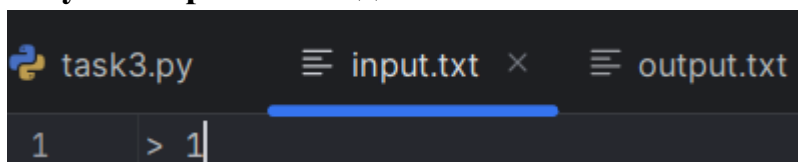


```
task3.py  input.txt  output.txt
1      + 1
2      + 3
3      + 3
4      > 1
5      > 2
6      > 3
7      + 2
8      > 1
```

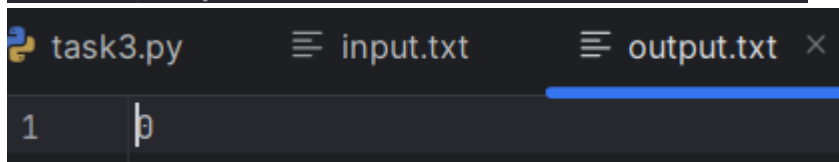


```
task3.py  input.txt  output.txt
1      |
2      3
3      0
4      2
```

Результат работы кода на максимальных и минимальных значениях:



```
task3.py  input.txt  output.txt
1      > 1
```



```
task3.py  input.txt  output.txt
1      |
```

```
task3.py  input.txt  output.txt
! The file size (3,87 MB) exceeds the configured l
1      + 691510318
2      > 152109163
3      + 434475956
4      > 679054580
```

```
task3.py  input.txt  output.txt
1      691510318
2      691510318
3      691510318
4      691510318
5      0
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000021 сек	0.042924 MB
Пример из задачи	0.000043 сек	0.043239 MB
Верхняя граница диапазона значений входных данных из текста задачи	1.423218 сек	44.362136 MB

Вывод по задаче:

Я реализовала простейшее BST с операциями вставки в дерево и поиска минимального значения, больше заданного.

Задача №4. Простейший неявный ключ [2s, 256 Mb, 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.

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.size = 1

def update_size(node):
    if node:
        node.size = 1 + (node.left.size if node.left else 0) +
        (node.right.size if node.right else 0)

def insert(root, value):
    if not root:
        return Node(value)
    if value < root.value:
        root.left = insert(root.left, value)
    elif value > root.value:
        root.right = insert(root.right, value)
    update_size(root)
    return root

def find_k(root, k):
    if not root:
        return None
    left_size = root.left.size if root.left else 0
    if k <= left_size:
        return find_k(root.left, k)
    elif k == left_size + 1:
        return root.value
    else:
        return find_k(root.right, k - left_size - 1)

with open('input.txt', 'r') as f:
    queries = [line.strip() for line in f.readlines()]

start_time = time.perf_counter()
root = None
results = []

for query in queries:
    if query.startswith("+"):
        x = int(query[2:])
        root = insert(root, x)
    elif query.startswith("?"):
        k = int(query[2:])
        result = find_k(root, k)
        results.append(str(result))
```

```
end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

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

print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое
использование: {peak / 10 ** 6:.6f} МВ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

Текстовое объяснение решения:

Я создала класс Node, представляющий узел дерева, он хранит значение, ссылки на потомков и размер поддерева с этим узлом в качестве корня (изначально единица).

Далее я реализовала метод update_size: если узел существует, его размер обновляется как сам узел + размер левого поддерева + размер правого поддерева.

Метод insert осуществляет вставку узла. Если корень пуст, создается новый узел, если значение меньше текущего узла, рекурсивно добавляем его в левое поддерево (иначе, в правое). После вставки обновляется размер текущего узла с помощью метода update_size.

Метод find_k ищет k-ый элемент: если узел отсутствует, возвращается None, иначе рассчитывается размер левого поддерева:

- 1) если k меньше или равно размеру левого поддерева, ищем в левом поддереве;
- 2) если k совпадает с текущим узлом, возвращаем значение этого узла;
- 3) если k больше, ищем в правом поддереве, учитывая, что уже пропустили left_size + 1 элементов.

Изначально дерево инициализируется как пустое. Для каждого запроса, если запрос начинается с «+», значение x добавляется в дерево, если запрос начинается с «?», извлекается номер k и ищется k-ый элемент.

Результат работы кода на примерах из текста задачи:

```
task4.py  input.txt  output.txt
1      + 1
2      + 4
3      + 3
4      + 3
5      ? 1
6      ? 2
7      ? 3
8      + 2
9      ? 3
```

```
task4.py  input.txt  output.txt
1      1
2      3
3      4
4      3
```

Результат работы кода на максимальных и минимальных значениях:

```
task4.py  input.txt  output.txt
1      ? 3
```

```
task4.py  input.txt  output.txt
1      None
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000016 сек	0.040500 MB
Пример из задачи	0.000055 сек	0.040892 MB

Вывод по задаче:

Я реализовала структуру данных с операциями для вставки элемента в BST и поиска элемента по неявному ключу.

Задача №6. Оpozнание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]

В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска. Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет.

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

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

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

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

import sys

sys.setrecursionlimit(10 ** 6)

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

def build_tree(tree, index):
    if index == -1:
        return None
    key, left_idx, right_idx = tree[index]
    node = Node(key)
    node.left = build_tree(tree, left_idx)
    node.right = build_tree(tree, right_idx)
    return node

def is_bst(node, min_val, max_val):
    if node is None:
```

```

        return True
    if node.key <= min_val or node.key >= max_val:
        return False
    return is_bst(node.left, min_val, node.key) and is_bst(node.right,
node.key, max_val)

with open('input.txt', 'r') as f:
    n = int(f.readline())
    tree = []
    for _ in range(n):
        K, L, R = map(int, f.readline().split())
        tree.append((K, L, R))

start_time = time.perf_counter()

if n == 0:
    with open('output.txt', 'w') as f:
        f.write("CORRECT")

else:
    root = build_tree(tree, 0)
    if is_bst(root, -2 ** 31 - 1, 2 ** 31):
        with open('output.txt', 'w') as f:
            f.write("CORRECT")
    else:
        with open('output.txt', 'w') as f:
            f.write("INCORRECT")

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала класс Node для представления узла дерева. Далее я создала метод is_bst, который проверяет на соответствие свойствам бинарного дерева поиска. На вход функция принимает узел, минимальное и максимальное значения для ключа текущего узла. Для корня min_val принимает значение -2^{31} , max_val принимает $2^{31} - 1$. При переходе к левому поддереву min_val остается неизменным, а max_val обновляется до значения текущего узла, т. к. все значения в левом поддереве должны быть меньше, чем ключ текущего узла. При переходе к правому – аналогично, но обновляется min_val.

Если узел отсутствует, то возвращается True (пустое дерево = BST). Если значение ключа узла меньше или равно min_val или больше или равно max_val, возвращается False. Рекурсивно проверяются левое и правое поддерева, возвращается результат проверки для обоих поддеревьев.

После чтения файла циклом обрабатываются n строк, и в массив tree добавляются K, L, R.

Далее строится само дерево, начиная с корневого узла. Если индекс текущего элемента в массиве равен -1, то узел отсутствует. Рекурсивно создаются правый и левый потомки. Возвращается объект класса Node. Далее проверяется, является ли дерево BST.

Результат работы кода на примерах из текста задачи:

```
task6.py  input.txt  output.txt
1      3
2      2 1 2
3      1 -1 -1
4      3 -1 -1
```

```
task6.py  input.txt  output.txt
1      CORRECT
```

```
task6.py  input.txt  output.txt
1      3
2      1 1 2
3      2 -1 -1
4      3 -1 -1
```

```
task6.py  input.txt  output.txt
1      INCORRECT
```

```
task6.py  input.txt  output.txt
1      5
2      1 -1 1
3      2 -1 2
4      3 -1 3
5      4 -1 4
6      5 -1 -1
```

```
task6.py  input.txt  output.txt x
1  CORRECT
```

```
task6.py  input.txt x  output.txt
1  7
2  4 1 2
3  2 3 4
4  6 5 6
5  1 -1 -1
6  3 -1 -1
7  5 -1 -1
8  7 -1 -1
```

```
task6.py  input.txt  output.txt x
1  CORRECT
```

```
task6.py  input.txt x  output.txt
1  4
2  4 1 -1
3  2 2 3
4  1 -1 -1
5  5 -1 -1
```

```
task6.py  input.txt  output.txt x
1  INCORRECT
```

Результат работы кода на максимальных и минимальных значениях:

```
task6.py  input.txt x  output.txt
1  0
```

```
task6.py  input.txt  output.txt x
1  CORRECT
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000484 сек	0.040266 МВ
Пример из задачи	0.000443 сек	0.041219 МВ
Пример из задачи	0.000441 сек	0.041219 МВ
Пример из задачи	0.000345 сек	0.041363 МВ
Пример из задачи	0.000541 сек	0.041539 МВ
Пример из задачи	0.000618 сек	0.041291 МВ

Вывод по задаче:

Я создала алгоритм для проверки на правильность реализации структуры данных бинарного дерева поиска.

Задача №7. Оpoznание двоичного дерева поиска (усложненная версия) [10s, 512 Mb, 2.5 балла]

Эта задача отличается от предыдущей тем, что двоичное дерево 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 поддеревья меньше ключа вершины V ;
- все ключи вершин из правого поддеревья 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.

Листинг кода:

```
import time
import tracemalloc

tracemalloc.start()

import sys

sys.setrecursionlimit(10 ** 6)

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

def build_tree(tree, index):
    if index == -1:
        return None
    key, left_idx, right_idx = tree[index]
    node = Node(key)
    node.left = build_tree(tree, left_idx)
    node.right = build_tree(tree, right_idx)
    return node

def is_bst(node, min_val, max_val):
    if node is None:
        return True
    if not (min_val < node.key <= max_val):
        return False
    return is_bst(node.left, min_val, node.key - 1) and
is_bst(node.right, node.key, max_val)

with open('input.txt', 'r') as f:
    n = int(f.readline())
    tree = []
    for _ in range(n):
        K, L, R = map(int, f.readline().split())
        tree.append((K, L, R))

start_time = time.perf_counter()

if n == 0:
    with open('output.txt', 'w') as f:
        f.write("CORRECT")
else:
    root = build_tree(tree, 0)
    if is_bst(root, -2 ** 31 - 1, 2 ** 31):
        with open('output.txt', 'w') as f:
            f.write("CORRECT")
    else:
        with open('output.txt', 'w') as f:
            f.write("INCORRECT")

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()
```

```
print(f"Затраты памяти: {current / 10 ** 6:.6f} МВ; Пиковое  
использование: {peak / 10 ** 6:.6f} МВ")  
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")
```

Текстовое объяснение решения:

Я адаптировала код из задачи №6, немного изменив логику функции `is_bst`. Теперь левый потомок должен быть строго меньше ключа текущего узла, а правый потомок может быть больше или равен ключу текущего узла.

Также для левого поддерева граница уменьшается на единицу, чтобы гарантировать, что все ключи в левом поддереве строго меньше текущего узла. В правом поддереве нижняя граница равна ключу текущего узла.

Результат работы кода на примерах из текста задачи:

```
task7.py  input.txt  output.txt  
1 3  
2 2 1 2  
3 1 -1 -1  
4 3 -1 -1
```

```
task7.py  input.txt  output.txt  
1 CORRECT
```

```
task7.py  input.txt  output.txt  
1 3  
2 1 1 2  
3 2 -1 -1  
4 3 -1 -1
```

```
task7.py  input.txt  output.txt  
1 INCORRECT
```

```
task7.py  input.txt  output.txt  
1 3  
2 2 1 2  
3 1 -1 -1  
4 2 -1 -1
```



```
task7.py  input.txt  output.txt x
1 INCORRECT
```

```
task7.py  input.txt x  output.txt
1 3
2 2 1 2
3 2 -1 -1
4 3 -1 -1
```

```
task7.py  input.txt  output.txt x
1 INCORRECT
```

```
task7.py  input.txt x  output.txt
1 5
2 1 -1 1
3 2 -1 2
4 3 -1 3
5 4 -1 4
6 5 -1 -1
```

```
task7.py  input.txt  output.txt x
1 CORRECT
```

```
task7.py  input.txt x  output.txt
1 7
2 4 1 2
3 2 3 4
4 6 5 6
5 1 -1 -1
6 3 -1 -1
7 5 -1 -1
8 7 -1 -1
```

```
task7.py  input.txt  output.txt x
1 CORRECT
```

task7.py input.txt output.txt

```
1 1
2 2147483647 -1 -1
```

task7.py input.txt output.txt

```
1 CORRECT
```

Результат работы кода на максимальных и минимальных значениях:

task7.py input.txt output.txt

```
1 0
```

task7.py input.txt output.txt

```
1 CORRECT
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.000325 сек	0.040266 MB
Пример из задачи	0.000398 сек	0.041219 MB
Пример из задачи	0.000651 сек	0.041219 MB
Пример из задачи	0.000420 сек	0.041219 MB
Пример из задачи	0.000425 сек	0.041219 MB
Пример из задачи	0.000452 сек	0.041363 MB
Пример из задачи	0.000771 сек	0.041539 MB
Пример из задачи	0.000419 сек	0.040373 MB

Вывод по задаче:

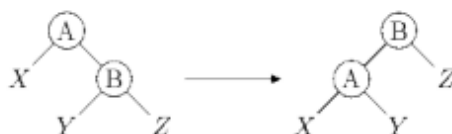
Я реализовала алгоритм для проверки правильности реализации структуры данных бинарного дерева поиска, которое может содержать равные ключи.

Задача №13. Делаю я левый поворот... [2 s, 256 Mb, 3 балла]

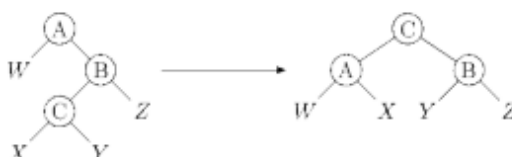
Для балансировки AVL-дерева при операциях вставки и удаления производятся *левые* и *правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем -1.

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

Малый левый поворот осуществляется следующим образом:



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



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

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

В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i + 1)$ -ой строке файла ($1 \leq i \leq N$) 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. Баланс корня дерева (вершины с номером 1) равен 2, баланс всех остальных вершин находится в пределах от -1 до 1.

- **Ограничения на входные данные.** $3 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления левого поворота. Нумерация вершин может быть произвольной при условии соблюдения формата. Так, номер вершины должен быть меньше номера ее детей.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 мб.
- **Пример:**

input.txt	output.txt
7	7
-2 7 2	3 2 3
8 4 3	-2 4 5
9 0 0	8 6 7
3 6 5	-7 0 0
6 0 0	0 0 0
0 0 0	6 0 0
-7 0 0	9 0 0

Листинг кода:

```
from collections import deque
import time
import tracemalloc

tracemalloc.start()

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

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

    def set_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
                parent.height = node.height + 1
            else:
                break
            node = parent

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

    def left_turn(self, A):
        B = A.right
        if B.balance == -1:
            C = B.left
            X = C.left
            Y = C.right
            if X:
                X.parent = A
            A.right = X
            if Y:
                Y.parent = B
            B.left = Y
            if A.key != self.root.key:
                C.parent = A.parent
                if A.parent.left == A:
```

```

        A.parent.left = C
    else:
        A.parent.right = C
    else:
        self.root = C
        A.parent = C
        C.left = A
        B.parent = C
        C.right = B
    else:
        Y = B.left
        if Y:
            Y.parent = A
        A.right = Y
        if A != self.root:
            B.parent = A.parent
            if A.parent.left == A:
                A.parent.left = B
            else:
                A.parent.right = B
        else:
            self.root = B
        A.parent = B
        B.left = A

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

with open('input.txt', 'r') as file:
    n = int(file.readline())
    tree = BinTree()
    data = []
    leaves = []
    for i in range(1, n + 1):
        data.append(list(map(int, file.readline().split())))
        tree.nodes[i] = Node()
        tree.nodes[i].key = data[i - 1][0]
        if data[i - 1][1] == 0 and data[i - 1][2] == 0:
            leaves.append(i)

start_time = time.perf_counter()

```

```

for i in range(1, n + 1):
    if data[i - 1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i - 1][1]]
        tree.nodes[data[i - 1][1]].parent = tree.nodes[i]
    if data[i - 1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i - 1][2]]
        tree.nodes[data[i - 1][2]].parent = tree.nodes[i]
    if i == 1:
        tree.root = tree.nodes[i]

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

tree.set_balance()
tree.left_turn(tree.root)
tree_output(tree, n)

end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()

print(f"Затраты памяти: {current / 10 ** 6:.6f} МБ; Пиковое
использование: {peak / 10 ** 6:.6f} МБ")
print(f"Время выполнения программы: {end_time - start_time:.6f} секунд")

```

Текстовое объяснение решения:

Я создала класс Node, представляющий узел бинарного дерева. Каждый узел имеет значение, ссылки на потомков и родителя, высоту и баланс.

Класс BinTree – это бинарное дерево. В атрибутах хранится корень дерева и все словарь для хранения всех узлов.

Метод set_height обновляет высоты всех узлов по дереву, начиная с листа (устанавливаем как единицу). Переходим к родителям узла, и если высота узла больше либо равна высоте родителя, то высота родителя принимает значение node.height + 1. В ином случае ничего обновлять не нужно. Переходим к следующему родителю.

Метод set_balance вычисляет баланс для каждого узла дерева. Обходим все узлы дерева и получаем узел с номером i:

- 1) если у узла есть и левый, и правый потомки, то баланс вычисляется как разница между высотой правого и левого потомков;
- 2) если есть только левый потомок, то баланс отрицателен (равен значению высоты левого потомка с минусом);
- 3) если есть только правый потомок, то баланс равен высоте правого потомка;
- 4) если потомков нет (т. е. узел является листом), то баланс равен 0.

Метод `left_turn` осуществляет левый поворот. Он принимает на вход узел *A* – корень поддерева, где выполняется левый поворот. Узел *B* – его правый ребенок, который станет новым корнем.

Если баланс *B* равен -1, необходимо выполнить большой левый поворот:

- 1) узел *C* (левый ребенок узла *B*) становится новым корнем поддерева;
- 2) узлы *X* и *Y* – левый и правый потомки узла *C* соответственно.

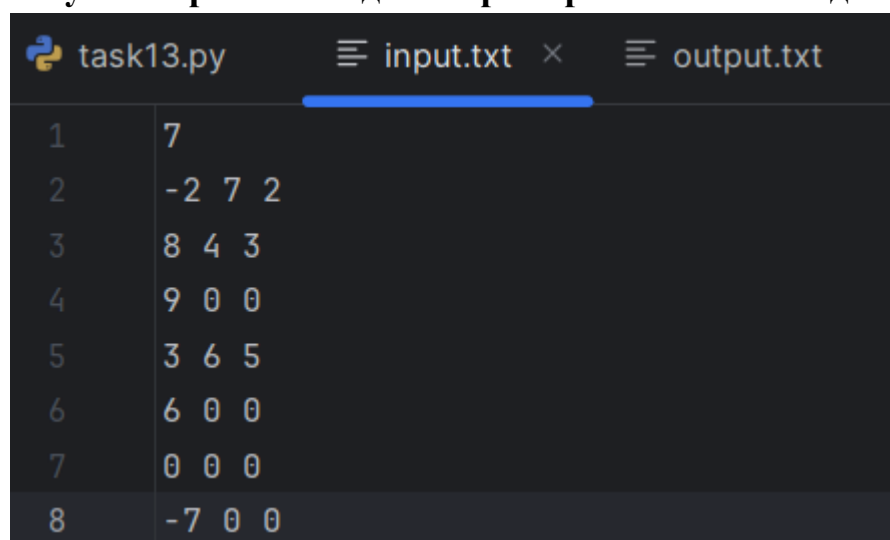
Узел *X* становится правым потомком узла *A*, т. е. поддерево *X* переносится вправо от узла *A*. Узел *Y* становится левым ребенком *B*, т. е. поддерево *Y* переносится влево от узла *B*.

Если *A* не является корнем всего дерева, узел *C* связываем с родителями узла *A*, иначе *C* становится новым корнем дерева. Узел *A* становится левым ребенком узла *C*, узел *B* становится правым ребенком узла *C*.

Если баланс *B* не равен -1, делаем малый левый поворот: левый ребенок узла *B* становится правым ребенком узла *A*. Узел *B* занимает место узла *A*, узел *A* становится левым потомком узла *B*.

Функция `tree_output` выводит результат. Сначала я создала очередь для обхода дерева в ширину (BFS) и добавила туда корень дерева. Затем я запустила счетчик для нумерации узлов, открыла файл и записала количество узлов. Пока очередь не пуста, извлекаем оттуда узел и записываем значение узла в файл. Если у узла есть левый ребенок, записываем номер левого ребенка, добавляем его в очередь. Аналогично с правым ребенком. Если потомков нет – записываем 0.

Результат работы кода на примерах из текста задачи:



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

```
task13.py  input.txt  output.txt x
1      7
2      3 2 3
3      -2 4 5
4      8 6 7
5      -7 0 0
6      0 0 0
7      6 0 0
8      9 0 0
```

	Время выполнения	Затраты памяти
Пример из задачи	0.000518 сек	0.046396 МВ

Вывод по задаче:

Я реализовала алгоритм, осуществляющий левые повороты АВЛ дерева.

Вывод

В ходе выполнения данной лабораторной работы я изучила двоичные деревья поиска, AVL-деревья и Splay-деревья. Я смогла реализовать различные алгоритмы по работе с этими структурами данных.