

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 24

Выполнил:  
Чан Тхи Лиен  
К3140

Проверил:  
Петросян.А.М

Санкт-Петербург  
2025 г.

## Содержание отчета

<b>Содержание отчета</b>	<b>2</b>
<b>Задачи по варианту</b>	
Задание 4. Простейший неявный ключ	3
Задание 10. Проверка корректности	7
Задание 14. Вставка в АВЛ-дерево	9
<b>Дополнительные задачи</b>	
Задание 5. Простое двоичное дерево поиска	14
Задание 12. Проверка сбалансированности	19
Задание 16. К-й максимум	22
<b>Вывод</b>	<b>24</b>

## Цель работы

В данной лабораторной работе изучается новая структура данных, двоичные (бинарные) деревья поиска и сбалансированные деревья поиска: AVL и Splay.

## Задачи по варианту

### Задание 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.

1. Класс **Node**(узел декартова дерева):

```
class Node: 1 usage
    def __init__(self, key):
        self.key = key
        self.priority = random.randint(1, 10**9)
        self.left = None
        self.right = None
        self.size = 1

    def update_size(self): 4 usages (4 dynamic)
        self.size = 1 + (self.left.size if self.left else 0) + (self.right.size if self.right else 0)
```

- Каждый узел имеет:
  - Ключ (**key**) — значение, по которому узлы хранятся в дереве.
  - Приоритет (**priority**) — случайное число, что делает структуру похожей на кучу.
  - Левый и правый потомки (**left, right**).
  - Размер поддерева (**size**), который обновляется при изменениях в дереве.
2. Функция **update\_size**:
- пересчитывает размер поддерева.
3. Функция **get\_size**:

```
def get_size(node): 2 usages
    return node.size if node else 0
```

- возвращает размер поддерева (или 0, если **node** — **None**).
4. Функция **split**:

```
def split(root, key):
    if not root:
        return None, None
    if root.key < key:
        root.right, right = split(root.right, key)
        root.update_size()
        return root, right
    else:
        left, root.left = split(root.left, key)
        root.update_size()
        return left, root
```

- Функция разделяет дерево **root** на два:
  - Левое поддерево содержит ключи  $< \text{key}$ .
  - Правое поддерево содержит ключи  $\geq \text{key}$ .

5. Функция **merge**:

```
def merge(left, right): 6 usages
    if not left or not right:
        return left or right
    if left.priority > right.priority:
        left.right = merge(left.right, right)
        left.update_size()
        return left
    else:
        right.left = merge(left, right.left)
        right.update_size()
        return right
```

- Узел с наивысшим приоритетом становится новым корнем.
- Деревья соединяются, сохраняя свойства кучи по **priority** и свойства бинарного дерева поиска.

6. Функция **insert**:

```
def insert(root, key): 9 usages
    if find(root, key):
        return root
    new_node = Node(key)
    left, right = split(root, key)
    return merge(merge(left, new_node), right)
```

- Проверяет, есть ли уже **key** в дереве.
- Создаёт новый узел.
- Разбивает дерево по **key** и вставляет новый узел.
- Объединяет обратно.

7. Функция **find**:

```
def find(root, key): 8 usages
    while root:
        if root.key == key:
            return True
        root = root.right if key > root.key else root.left
    return False
```

- Просто ищет **key** в дереве.

8. Функция **kth\_element**:

```
def kth_element(root, k): 5 usages
    left_size = get_size(root.left)
    if k == left_size + 1:
        return root.key
    elif k <= left_size:
        return kth_element(root.left, k)
    else:
        return kth_element(root.right, k - left_size - 1)
```

- Определяет k-й по порядку элемент:
    - Если  $k == \text{left\_size} + 1$ , текущий узел — ответ.
    - Если  $k \leq \text{left\_size}$ , ищем в левом поддереве.
    - Иначе, ищем в правом поддереве, но теперь  $k$  уменьшается на  $\text{left\_size} + 1$ .
9. Функция `task4()` для выполнения задания:

```
def task4(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    queries = read_file(PATH_INPUT)

    results = []
    treap = None

    for query in queries:
        if not query:
            continue
        if query[0] == '+':
            x = int(query[2:])
            treap = insert(treap, x)
        elif query[0] == '?':
            k = int(query[2:])
            results.append(str(kth_element(treap, k)) + '\n')

    write_file(PATH_OUTPUT, str(''.join(results)))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task4()
```

- **treap** — изначально пустое дерево.
- **+ x** — добавляет число **x** в **treap**.
- **? k** — ищет **k**-й по порядку элемент в дереве.

#### 10. Проведенные тесты.

```
+ 1
+ 4
+ 3
+ 3
? 1
? 2
? 3
+ 2
? 3|
```

input.txt:

```
1
3
4
3
```

output.txt:

```
Время работы: 0.00050360000075051561 секунд
память использовать: 0.011719 МВ
```

#### 11. Запись теста.

Этот код реализует декартово дерево, используя его для эффективного вставки ( $O(\log n)$ ) и поиска  $k$ -го по порядку элемента ( $O(\log n)$ ).

## Задание 10. Проверка корректности

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

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

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

### 1. Функция `find_tree`:

```
def find_tree(data): 3 usages
    tree = []
    for line in data:
        k, l, r = map(int, line.split())
        tree.append((k, l - 1 if l else -1, r - 1 if r else -1))
    return tree
```

- Эта функция принимает список строк **data** (каждая строка содержит информацию о вершине в дереве).
- Для каждой строки создается кортеж с:
  - ключом узла (**k**)
  - индексом левого потомка (**l**), если он есть, или **-1**, если его нет,
  - индексом правого потомка (**r**), если он есть, или **-1**, если его нет.
- Строит и возвращает список из этих кортежей.

### 2. Функция `is_bst`:

```
def is_bst(tree, index=0, min_val=float('-inf'), max_val=float('inf')): 8 usages
    if not tree or index == -1:
        return True
    key, left, right = tree[index]
    if not (min_val < key < max_val):
        return False
    return is_bst(tree, left, min_val, key) and is_bst(tree, right, key, max_val)
```

- Эта рекурсивная функция проверяет, является ли поддерево, начиная с индекса **index**, бинарным деревом поиска (BST).
- Для каждого узла проверяется:
  - Если его ключ меньше минимального значения (**min\_val**) или больше максимального значения (**max\_val**), то это не дерево поиска, и возвращается **False**.
  - Затем рекурсивно проверяются левое поддерево (с новыми пределами: **min\_val** и **key**) и правое поддерево (с новыми пределами: **key** и **max\_val**).

3. Функция **task10()** для выполнения задания:

```
def task10(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    array = data[1:]

    if n == 0:
        write_file(PATH_OUTPUT, str('YES'))
    else:
        tree = find_tree(array)
        if is_bst(tree):
            write_file(PATH_OUTPUT, str('YES'))
        else:
            write_file(PATH_OUTPUT, str('NO'))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task10()
```

4. Проведенные тесты.

input.txt:

5	2	3
6	0	0
4	0	0

output.txt:

NO
----

```
Время работы: 0.0004611000040313229 секунд
память использовать: 0.023438 MB
```

5. Запись теста.



#### Задание 14. Вставка в AVL-дерево

Вставка в AVL-дерево вершины  $V$  с ключом  $X$  при условии, что такой вершины в этом дереве нет, осуществляется следующим образом:

- находится вершина  $W$ , ребенком которой должна стать вершина  $V$  ;
- вершина  $V$  делается ребенком вершины  $W$ ;
- производится подъем от вершины  $W$  к корню, при этом, если какая-то из вершин несбалансирована, производится, в зависимости от значения баланса, левый или правый поворот.

Первый этап нуждается в пояснении. Спуск до будущего родителя вершины  $V$  осуществляется, начиная от корня, следующим образом:

- Пусть ключ текущей вершины равен  $Y$  .
- Если  $X < Y$  и у текущей вершины есть левый ребенок, переходим к левому ребенку.
- Если  $X < Y$  и у текущей вершины нет левого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.
- Если  $X > Y$  и у текущей вершины есть правый ребенок, переходим к правому ребенку.
- Если  $X > Y$  и у текущей вершины нет правого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.

Отдельно рассматривается следующий крайний случай – если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

##### 1. Класс **Node** (узел дерева):

```
class Node: 7 usages
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right
        self.height = 1
```

- Каждый узел дерева хранит:
  - Ключ (**key**), который используется для поиска и сортировки.
  - Указатели на левого и правого потомков (**left**, **right**).
  - Высоту поддерева с этим узлом (начальная высота равна 1).

##### 2. Класс **AVLTree**:

```
class AVLTree: 3 usages
    def __init__(self):
        self.root = None
        self.nodes = {}
        self.index_map = {}
        self.index_counter = 1
```

- **root** — корень дерева.

- **nodes** — словарь для хранения узлов дерева, где ключ — это уникальный индекс, а значение — сам узел.
- **index\_map** — отображение узлов на их индексы.
- **index\_counter** — счетчик индексов для узлов.

❖ Функция **height**:

```
def height(self, node): 8 usages (3 dynamic)
    return node.height if node else 0
```

- возвращает высоту поддерева, корнем которого является узел.
- если узел отсутствует, возвращает 0.

❖ Функция **balance\_factor**:

```
def balance_factor(self, node): 5 usages
    return self.height(node.left) - self.height(node.right) if node else 0
```

- вычисляет **баланс** узла как разницу высот левого и правого поддеревьев.
- в AVL-дереве баланс должен быть в пределах  $[-1, 1]$ .

❖ Функция **fix\_height**:

```
def fix_height(self, node): 5 usages
    node.height = max(self.height(node.left), self.height(node.right)) + 1
```

- после изменений в поддеревьях высота узла должна быть обновлена.

❖ Функция **rotate\_right**:

```
def rotate_right(self, node): 3
    new_root = node.left
    node.left = new_root.right
    new_root.right = node
    self.fix_height(node)
    self.fix_height(new_root)
    return new_root
```

- правый поворот, используется для исправления случая, когда левое поддерево имеет большую высоту.

❖ Функция **rotate\_left**:

```
def rotate_left(self, node): 3
    new_root = node.right
    node.right = new_root.left
    new_root.left = node
    self.fix_height(node)
    self.fix_height(new_root)
    return new_root
```

- левый поворот, используется для исправления случая, когда правое поддерево имеет большую высоту.

❖ Функция **balance**:

```
def balance(self, node): 1 usage
    self.fix_height(node)
    if self.balance_factor(node) == 2:
        if self.balance_factor(node.left) < 0:
            node.left = self.rotate_left(node.left)
        return self.rotate_right(node)
    if self.balance_factor(node) == -2:
        if self.balance_factor(node.right) > 0:
            node.right = self.rotate_right(node.right)
        return self.rotate_left(node)
    return node
```

- если баланс узла больше 1, это означает, что дерево "провисло" слева, и выполняются соответствующие повороты.
- если баланс узла меньше -1, дерево "провисло" справа, и выполняются противоположные повороты.

❖ Функция **insert**:

```
def insert(self, node, key): 8 usages
    if not node:
        new_node = Node(key)
        self.nodes[len(self.nodes) + 1] = new_node
        return new_node
    if key < node.key:
        node.left = self.insert(node.left, key)
    else:
        node.right = self.insert(node.right, key)
    return self.balance(node)
```

- вставка нового ключа происходит рекурсивно в левое или правое поддерево.
- после вставки выполняется балансировка дерева.

❖ Функция **build\_tree**:

```
def build_tree(self, nodes_info): 3 usages
    self.nodes = {i + 1: Node(k) for i, (k, _, _) in enumerate(nodes_info)}
    for i, (k, l, r) in enumerate(nodes_info):
        if l:
            self.nodes[i + 1].left = self.nodes[l]
        if r:
            self.nodes[i + 1].right = self.nodes[r]
    self.root = self.nodes[1] if self.nodes else None
```

- Эта функция строит дерево из списка кортежей с информацией о каждом узле.
- Каждый кортеж включает ключ узла и индексы левого и правого потомков.
- Строится словарь **nodes**, в котором каждый узел имеет уникальный индекс.

❖ Функция **update\_nodes**:

```
def update_nodes(self, node): 3 usages
    if not node:
        return
    if node not in self.index_map:
        self.index_map[node] = self.index_counter
        self.nodes[self.index_counter] = node # 0
        self.index_counter += 1
    self.update_nodes(node.left)
    self.update_nodes(node.right)
```

- обход дерева.
- создание отображения (или карты) узлов с их уникальными индексами
- вывод дерева в определенном формате.

❖ Функция **generate\_tree\_output**:

```
def generate_tree_output(self): 2 usages
    self.index_map.clear()
    self.index_counter = 1
    self.nodes.clear()
    self.update_nodes(self.root)
    output = [f"{len(self.nodes)}"]
    for i in sorted(self.nodes):
        node = self.nodes[i]
        left_index = self.index_map.get(node.left, 0)
        right_index = self.index_map.get(node.right, 0)
        output.append(f"{node.key} {left_index} {right_index}")
    return output
```

- Эта функция генерирует вывод в формате, который ожидает задача:
- Число узлов.
- Для каждого узла: его ключ, индекс левого потомка и индекс правого потомка.

### 3. Функция **task14()** для выполнения задания:

```
def task14(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    nodes_info = [tuple(map(int, line.split())) for line in data[1:n+1]]
    x = int(data[n + 1])

    avl = AVLTree()
    avl.build_tree(nodes_info)
    avl.root = avl.insert(avl.root, x)
    result = avl.generate_tree_output()

    write_file(PATH_OUTPUT, str("\n".join(result) + "\n"))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task14()
```

### 4. Проведенные тесты.

input.txt:

2
3 0 2
4 0 0
5

output.txt:

3
4 2 3
3 0 0
5 0 0

```
Время работы: 0.0005265999934636056 секунд
память использовать: 0.011719 MB
```

### 5. Запись теста.

## Дополнительные задачи

### Задание 5. Простое двоичное дерево поиска

Реализуйте простое двоичное дерево поиска.

- `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.

#### 1. Класс **TreeNode** (Узел дерева):

```
class TreeNode: 3 usages
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

- Создает узел бинарного дерева.
- **key** – значение узла.
- **left** и **right** – ссылки на левого и правого потомков.

#### 2. Класс **BST** (Бинарное дерево поиска):

```
class BST: 3 usages
    def __init__(self):
        self.root = None

    def insert(self, key): 13 usages
        if not self.root:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)
```

##### ❖ Функция **insert**:

- Добавляет новый узел с ключом **key** в дерево.
- Если корня нет – создает его.
- Иначе вызывает вспомогательную функцию **\_insert**.

##### ❖ Функция **\_insert**:

```
def _insert(self, node, key): 3 usages
    if key < node.key:
        if node.left:
            self._insert(node.left, key)
        else:
            node.left = TreeNode(key)
    elif key > node.key:
        if node.right:
            self._insert(node.right, key)
        else:
            node.right = TreeNode(key)
```

- Вставляет **key** в правильное место, двигаясь по дереву.
- Если **key < node.key** → идем влево.
- Если **key > node.key** → идем вправо.
- Если место найдено, создаем новый узел.

❖ Функция **delete**:

- Удаляет узел с **key**, вызывая **\_delete**.

❖ Функция **\_delete**:

```
def delete(self, key): 2 usages
    self.root = self._delete(self.root, key)

def _delete(self, node, key): 4 usages
    if not node:
        return node
    if key < node.key:
        node.left = self._delete(node.left, key)
    elif key > node.key:
        node.right = self._delete(node.right, key)
    else:
        if not node.left:
            return node.right
        elif not node.right:
            return node.left
        min_node = self._min(node.right)
        node.key = min_node.key
        node.right = self._delete(node.right, min_node.key)
    return node
```

- Если **key < node.key**, ищем в левом поддереве.
- Если **key > node.key**, ищем в правом поддереве.
- Если нашли:
  - Нет детей → просто удаляем.
  - Один ребенок → заменяем этим ребенком.
  - Два ребенка → заменяем на минимальный элемент из правого поддерева.

❖ Функция **exists**:

- Проверяет, существует ли узел с **key**, вызывая **\_exists**.

❖ Функция **\_exists**:

```
def exists(self, key): 8 usages
    return self._exists(self.root, key)

def _exists(self, node, key): 3 usages
    if not node:
        return False
    if key < node.key:
        return self._exists(node.left, key)
    elif key > node.key:
        return self._exists(node.right, key)
    return True
```

- Ищет **key** в дереве:

- Если **key < node.key**, идем влево.
- Если **key > node.key**, идем вправо.
- Если **key == node.key**, возвращаем **True**.

❖ Функция **next**:

- Находит минимальный элемент, **больше** чем **key**.
- Если нет такого элемента – возвращает **"none"**.

❖ Функция **\_next**:

```
def next(self, key): 4 usages
    result = self._next(self.root, key, succ: None)
    return result if result is not None else "none"

def _next(self, node, key, succ): 3 usages
    if not node:
        return succ
    if node.key > key:
        return self._next(node.left, key, node.key)
    return self._next(node.right, key, succ)
```

- Если **node.key > key**, возможный следующий элемент — **node.key**, идем влево.
- Если **node.key ≤ key**, идем вправо.



❖ Функция **prev**:

- Находит максимальный элемент, меньше чем **key**.
- Если нет такого элемента – возвращает **"none"**.

❖ Функция **\_prev**:

```
def prev(self, key): 4 usages
    result = self._prev(self.root, key, pred: None)
    return result if result is not None else "none"

def _prev(self, node, key, pred): 3 usages
    if not node:
        return pred
    if node.key < key:
        return self._prev(node.right, key, node.key)
    return self._prev(node.left, key, pred)
```

- Если **node.key < key**, возможный предыдущий элемент — **node.key**, идем вправо.
- Если **node.key ≥ key**, идем влево.

❖ Функция **\_min**:

```
def _min(self, node): 1 usage
    while node.left:
        node = node.left
    return node
```

- Находит минимальный узел в поддереве.

❖ Функция **process\_operations**:

```
def process_operations(operations): 3 usages
    tree = BST()
    result = []
    for operation in operations:
        op = operation.split()
        command = op[0]
        if command == "insert":
            tree.insert(int(op[1]))
        elif command == "delete":
            tree.delete(int(op[1]))
        elif command == "exists":
            result.append("true" if tree.exists(int(op[1])) else "false")
        elif command == "next":
            result.append(str(tree.next(int(op[1]))))
        elif command == "prev":
            result.append(str(tree.prev(int(op[1]))))
    return result
```

- Создает дерево **BST**.
- Выполняет команды:

- "insert X" → вставляет X.
- "delete X" → удаляет X.
- "exists X" → проверяет, есть ли X в дереве.
- "next X" → находит следующий элемент после X.
- "prev X" → находит предыдущий элемент перед X.

3. Функция **task5()** для выполнения задания:

```
def task5(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    operations = read_file(PATH_INPUT)

    result = process_operations(operations)

    write_file(PATH_OUTPUT, "\n".join(result) + "\n")

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f'память использовать: {end_memory - start_memory:.6f} MB')

if __name__ == '__main__':
    task5()
```

4. Проведенные тесты.

input.txt:

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

output.txt:

```
false
none
3
3
```

```
Время работы: 0.0004791000101249665 секунд
память использовать: 0.011719 MB
```

5. Запись теста.

Этот код реализует **бинарное дерево поиска (BST)** и поддерживает:

- Вставку
- Удаление
- Проверку существования
- Поиск следующего и предыдущего элемента

## Задание 12. Проверка сбалансированности

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:  $-1 \leq B(V) \leq 1$

Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

```
sys.setrecursionlimit(300000)
```

### 1. Функция `dfs`:

```
def dfs(node, tree, height, balance): 6 usages
    # Если вершина пустая (ноль), высота 0
    if node == 0:
        return 0

    # Если высота для этой вершины уже вычислена, возвращаем её
    if height[node] != -1:
        return height[node]

    # Рекурсивный вызов для левого и правого поддерева
    left_height = dfs(tree[node][0], tree, height, balance)
    right_height = dfs(tree[node][1], tree, height, balance)

    # Вычисление высоты текущей вершины
    height[node] = 1 + max(left_height, right_height)

    # Баланс текущей вершины
    balance[node] = right_height - left_height

    return height[node]
```

- Обход в глубину для вычисления высоты и баланса узлов.
- Рекурсивно вычисляет высоту каждого узла.
- Рассчитывает баланс узла (**right\_height - left\_height**).
- Используется для проверки сбалансированности дерева.

## 2. Функция `slove_data`:

```
def slove_data(n, data): 6 usages

    if n == 0:
        return n, [], [], []

    tree = [(0, 0)] * (n + 1)
    height = [-1] * (n + 1)
    balance = [0] * (n + 1)

    for i in range(1, n+1):
        k, l, r = map(int, data[i-1].split())
        tree[i] = (l, r)

    return n, tree, height, balance
```

- Создает массив **tree**, хранящий пары (левый сын, правый сын) для каждого узла.
  - Создает массивы **height** и **balance** для хранения высоты и баланса узлов.
  - Заполняет **tree** на основе входных данных.
- ## 3. Функция `task12()` для выполнения задания:

```
def task12(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    data1 = data[1:]

    n, tree, height, balance = slove_data(n, data1)

    if n == 0:
        return

    dfs(node: 1, tree, height, balance)

    write_file(PATH_OUTPUT, "\n".join(map(str, balance[1:])))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task12()
```

#### 4. Проведенные тесты.

input.txt:

```
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
```

output.txt:

```
3
-1
0
0
0
0
```

```
Время работы: 0.00046329999167937785 секунд
память использовать: 0.015625 MB
```

#### 5. Запись теста.

- Этот код решает задачу нахождения баланс-факторов узлов в бинарном дереве.
- Использует обход в глубину (**DFS**) для вычисления высот и баланс-факторов.
- Эффективен благодаря кэшированию высот (**height[node]**).
- Использует рекурсию и массивы для хранения информации о дереве.

## Задание 16. К-й максимум

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также находить  $k$ -й максимум.

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

### 1. Функция `process_commands`:

```
def process_commands(n, commands): 7 usages
    """Основная функция для обработки команд."""
    elements = []
    results = []

    for command, k in commands:
        if command == +1: # Добавить элемент
            bisect.insort(elements, k) # Вставка с сохранением сортировки

        elif command == 0: # Найти k-й максимум
            if 1 <= k <= len(elements): # Проверяем, что k в допустимом диапазоне
                results.append(elements[-k]) # k-й максимум будет на индексе -k
            else:
                results.append(None) # k-й максимум будет на индексе -k в отсортированном списке

        elif command == -1: # Удалить элемент
            elements.remove(k) # Удаляем элемент из списка

    return results
```

- Хранение данных:
  - Все элементы хранятся в отсортированном списке **elements**.
  - Вставка (+1  $k$ ) выполняется с сохранением сортировки с помощью **bisect.insort()**.
- Поиск  $k$ -го максимального (0  $k$ )
  - Если  $k$  входит в допустимый диапазон (от 1 до длины списка), мы берем **- $k$** -й элемент (например, **-1** — максимум).
  - Если  $k$  выходит за границы, добавляем **None** в **results**.
- Удаление (-1  $k$ )
  - Просто удаляет элемент **k** из списка с помощью **elements.remove(k)**.

2. Функция **task16()** для выполнения задания:

```
def task16(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    n = int(data[0])
    commands = [tuple(map(int, line.split())) for line in data[1:n+1]]

    results = process_commands(n, commands)

    write_file(PATH_OUTPUT, "\n".join(map(str, results)))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task16()
```

3. Проведенные тесты.

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
```

```
Время работы: 0.0006083999905968085 секунд  
память использовать: 0.015625 МВ
```

#### 4. Запись теста.

- Работает корректно, но не оптимально для больших входных данных
- Использует **bisect** для поддержки отсортированного списка
- Простая реализация, но требует оптимизации (например, с **SortedList**)

#### Вывод:

- Выполнение заданий.
- Знакомство с двоичными деревьями поиска.
- Использование класса и **bisect.insort()**, .....