

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 14

Выполнила:  
Рудникова В.О.  
К3143

Проверила:  
Артамонова В.Е.

Санкт-Петербург  
2022 г.

## **Содержание отчета**

<b>Содержание отчета</b>	<b>2</b>
<b>Задачи по варианту</b>	<b>3</b>
Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл]	3
<b>Дополнительные задачи</b>	<b>10</b>
<b>Вывод</b>	<b>26</b>

## 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 (inorder), 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 (postorder) обходы в глубину

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def in_order_traversal(root, res):
    if root:
        in_order_traversal(root.left, res)
        res.append(str(root.val))
        in_order_traversal(root.right, res)

def pre_order_traversal(root, res):
    if root:
        res.append(str(root.val))
        pre_order_traversal(root.left, res)
        pre_order_traversal(root.right, res)

def post_order_traversal(root, res):
    if root:
```

```

post_order_traversal(root.left, res)
post_order_traversal(root.right, res)
res.append(str(root.val))

start()

with open("input.txt") as f:
n = int(f.readline())
nodes = []
for i in range(n):
k, l, r = map(int, f.readline().split())
node = Node(k)
node.left = l
node.right = r
nodes.append(node)

for i in range(n):
if nodes[i].left != -1:
nodes[i].left = nodes[nodes[i].left]
else:
nodes[i].left = None
if nodes[i].right != -1:
nodes[i].right = nodes[nodes[i].right]
else:
nodes[i].right = None

root = nodes[0]

inorder_res = []
preorder_res = []
postorder_res = []

in_order_traversal(root, inorder_res)
pre_order_traversal(root, preorder_res)
post_order_traversal(root, postorder_res)

with open("output.txt", "w+") as g:

```

```

g.write(" ".join(inorder_res) + "\n")
g.write(" ".join(preorder_res) + "\n")
g.write(" ".join(postorder_res) + "\n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')

```

Текстовое объяснение решения: в задаче выполняется обход двоичного дерева поиска тремя способами: pre-order, in-order, post-order.

Результат работы кода на примерах из текста задачи:

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

task1.py	input.txt	output.txt
1	10	50 70 80 30 90 40 0 20 10 60
2	0 7 2	0 70 50 40 30 80 90 20 60 10
3	10 -1 -1	50 80 90 30 40 70 10 60 20 0
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	

Вывод по задаче:

В задаче я научилась обходить двоичное дерево поиска тремя способами.

#### Задача №4. Простейший неявный ключ [2 s, 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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

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

    def __find(self, node, parent, value):
        if node is None:
            return None, parent, False

        if value == node.data:
            return node, parent, True

        if value < node.data and node.left:
            return self.__find(node.left, node, value)

        if value > node.data and node.right:
            return self.__find(node.right, node, value)

        return node, parent, False

    def append(self, obj):
        if self.root is None:
```

```

self.root = obj
return obj
s, p, fl_find = self.__find(self.root, None,
obj.data)
if not fl_find and s:
if obj.data < s.data:
s.left = obj
else:
s.right = obj
return obj

def in_order_traversal(self, node, res):
if node:
self.in_order_traversal(node.left, res)
res.append(node.data)
self.in_order_traversal(node.right, res)

def kth_smallest(self, k):
res = []
self.in_order_traversal(self.root, res)
if k <= len(res):
return res[k - 1]
else:
return None

start()
t = BinaryTree()

with open("output.txt", "w+") as g:
with open("input.txt") as f:
all_info = f.readlines()
cleared_info = []
for i in all_info:
if "\n" in i:
cleared_info.append(i.replace("\n", ""))
else:
cleared_info.append(i)

```

```

for i in cleared_info:
    if "+" in i:
        t.append(Node(int(i[2])))
    else:
        g.write(str(t.kth_smallest(int(i[2]))) + "\n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Текстовое объяснение решения: в задаче реализован класс двоичного дерева поиска с возможностью отвечать на запросы добавления элемента и его нахождения по неявному ключу.

Результат работы кода на примерах из текста задачи:

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

Вывод по задаче: в задаче я реализовала BST с неявным ключом.



## Задача №7. Опознавание двоичного дерева поиска (усложненная версия)

[10 s, 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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def is_bst(node, min_key, max_key):
    if node is None:
        return True
    if node.key < min_key or node.key >= max_key:
        return False
    return is_bst(node.left, min_key, node.key) and
    is_bst(node.right, node.key, max_key)
```

```

def build_tree(n, nodes):
    tree = [None] * n
    for i in range(n):
        key, left, right = nodes[i]
        left_child = None if left == -1 else tree[left]
        right_child = None if right == -1 else tree[right]
        tree[i] = Node(key, left_child, right_child)
    return tree[0]

start()

with open("input.txt") as f:
    n = int(f.readline().strip())
    nodes = [tuple(map(int, line.strip().split())) for line in f.readlines()]

root = build_tree(n, nodes)

with open("output.txt", "w+") as g:
    if is_bst(root, -float("inf"), float("inf")):
        g.write("CORRECT")
    else:
        g.write("INCORRECT")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')

```

Текстовое объяснение решения: проверяется условие двоичного дерева поиска - слева значения меньше, чем справа.

Результат работы кода на примерах из текста задачи:

task7.py ×		input.txt ×		⋮		output.txt ×	
1	3	✓	1	CORRECT			
2	2 1 2						
3	1 -1 -1						
4	3 -1 -1						

task7.py ×		input.txt ×		⋮		output.txt ×	
1	3	✓	1	INCORRECT			
2	1 1 2						
3	2 -1 -1						
4	3 -1 -1						

task7.py ×		input.txt ×		⋮		output.txt ×	
1	3	✓	1	CORRECT			
2	2 1 2						
3	1 -1 -1						
4	2 -1 -1						

task7.py ×		input.txt ×		⋮		output.txt ×	
1	5	✓	1	CORRECT			
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 ×	
1	7	✓	1	CORRECT
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 ×	
1	1	✓	1	CORRECT
2	2147483647 -1 -1			

Вывод по задаче: я повторила устройство двоичного дерева поиска и реализовала проверку на него.

#### Задача №14. Вставка в AVL-дерево [2 s, 256 Mb, 3 балла]

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.

```
import sys
from time import process_time
from tracemalloc import start, get_traced_memory

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

class AVLTree:
```

```

left_index = 1
right_index = 1
res_left_index = 2
res_right_index = 3

def get_height(self, node):
    if node is None:
        return 0
    else:
        return node.height

def balance(self, node):
    if node is None:
        return 0
    else:
        return self.get_height(node.left) -
               self.get_height(node.right)

def left_rotate(self, node):
    a = node.right
    b = a.left
    a.left = node
    node.right = b
    node.height = 1 + max(self.get_height(node.left),
                           self.get_height(node.right))
    a.height = 1 + max(self.get_height(a.left),
                        self.get_height(a.right))
    return a

def right_rotate(self, node):
    a = node.left
    b = a.right
    a.right = node
    node.left = b
    node.height = 1 + max(self.get_height(node.left),
                           self.get_height(node.right))

```

```

a.height = 1 + max(self.get_height(a.left),
self.get_height(a.right))
return a

def insert(self, value, root):
if root is None:
return Node(value)
elif value <= root.value:
root.left = self.insert(value, root.left)
elif value > root.value:
root.right = self.insert(value, root.right)

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

if balance > 1 and root.left.value > value:
return self.right_rotate(root)
if balance > 1 and root.left.value < value:
root.left = self.left_rotate(root.left)
return self.right_rotate(root)
if balance < -1 and root.right.value < value:
return self.left_rotate(root)
if balance < -1 and root.right.value > value:
root.right = self.right_rotate(root.right)
return self.left_rotate(root)
return root

def preorder(self, root):
if root is None:
return
if root.left:
self.left_index += 1
if self.right_index == self.left_index:
self.left_index += 1
self.res_left_index = self.left_index
else:
self.res_left_index = 0

```

```

if root.right:
self.right_index += 1
if self.right_index == self.right_index:
self.right_index += 1
self.res_right_index = self.right_index
else:
self.res_right_index = 0
print(root.value, self.res_left_index,
self.res_right_index)
self.preorder(root.left)
self.preorder(root.right)

start()

with open("input.txt") as f:
n = int(f.readline())
nodes = [int(f.readline().split()[0]) for i in
range(n + 1)]

tree = AVLTree()
root = None
for node in nodes:
root = tree.insert(node, root)

orig_stdout = sys.stdout
with open("output.txt", "w") as g:
g.write(str(len(nodes)) + "\n")
sys.stdout = g
tree.preorder(root)
sys.stdout = orig_stdout

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```



Текстовое объяснение решения: в задаче производится добавление вершины в AVL-дерево: сначала ищется вершина-родитель, добавляемая вершина делается её ребёнком, а потом, при необходимости, делается поворот дерева для балансировки.

Результат работы кода на примерах из текста задачи:

task14.py ×		input.txt ×		output.txt ×	
1	2	✓	1	3	
2	3 0 2		2	4 2 3	
3	4 0 0		3	3 0 0	
4	5		4	5 0 0	
			5		

Вывод по задаче: я реализовала AVL-дерево и добавление вершины в него.

## Дополнительные задачи

### Задача №2. Гирлянда [2 s, 256 Mb, 1 балл]

Гирлянда состоит из  $n$  лампочек на общем проводе. Один её конец закреплён на заданной высоте  $A$  мм ( $h_1 = A$ ). Благодаря силе тяжести гирлянда прогибается: высота каждой неконцевой лампы на 1 мм меньше, чем средняя высота ближайших соседей ( $h_i = h_{i-1} + h_{i+1} \cdot 2 - 1$  для  $1 < i < N$ ).

Требуется найти минимальное значение высоты второго конца  $B$  ( $B = h_n$ ), такое что для любого  $\epsilon > 0$  при высоте второго конца  $B + \epsilon$  для всех лампочек выполняется условие  $h_i > 0$ . Обратите внимание на то, что при данном значении высоты либо ровно одна, либо две соседних лампочки будут иметь нулевую высоту.

Подсказка: для решения этой задачи можно использовать двоичный поиск.

```
from time import process_time
from tracemalloc import start, get_traced_memory

def binary_search():
    l, r = 0, h[0]
    while r - l > 0.0000000001:
        h[1] = (l + r) / 2
        Up = True
        for i in range(2, n):
            h[i] = 2 * h[i - 1] - h[i - 2] + 2
            if h[i] < 0:
                Up = False
                break
        if Up:
            r = h[1]
        else:
            l = h[1]
    return h[n - 1]
```

```

start()

with open("input.txt") as f:
    n, h = f.readline().split()
    n = int(n)
    h = [float(h)] + [0] * (n - 1)

float_number = binary_search()
float_number = "{:.9f}".format(float_number)
total = [i for i in str(float_number) if i != "0"]
total_string = ""

for i in total:
    total_string += str(i)
with open("output.txt", "w+") as g:
    g.write(total_string)

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')

```

Текстовое объяснение решения: использован алгоритм бинарного поиска, который позволяет найти такую высоту второго конца, чтобы гирлянда не достала до пола.

Результат работы кода на примерах из текста задачи:

task2.py ×	input.txt ×	⋮	output.txt ×
1	8 15	✓	1 9.75

task2.py ×	input.txt ×	⋮	output.txt ×
1	692 532.81	✓	1 446113.344347841

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: в данной задаче я использовала бинарный поиск.

### 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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

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

    def size(self, node):
        if node is None:
            return 0
        return node.size

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

    def _insert(self, node, key):
        if node is None:
            return Node(key)

        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
```

```

node.right = self._insert(node.right, key)

node.size = 1 + self.size(node.left) +
self.size(node.right)
return node

def get_min_gt(self, key):
return self._get_min_gt(self.root, key)

def _get_min_gt(self, node, key):
if node is None:
return 0

if node.key <= key:
return self._get_min_gt(node.right, key)

if node.left is None:
return node.key

if node.left.key <= key:
return self._get_min_gt(node.left.right, key)

return self._get_min_gt(node.left, key)

start()

bst = BinarySearchTree()

with open("output.txt", "w+") as g:
with open("input.txt") as f:
all_info = f.readlines()
for i in all_info:
if i.split()[0] == "+":
bst.insert(i.split()[1])
else:
g.write(str(bst.get_min_gt(i.split()[1])) + "\n")

```

```
print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')
```

Текстовое объяснение решения: тут реализовано двоичное дерево поиска по явному ключу.

Результат работы кода на примерах из текста задачи:

task3.py ×			input.txt ×			⋮			output.txt ×		
1	+	1	✓			1	3				
2	+	3				2	3				
3	+	3				3	0				
4	>	1				4	2				
5	>	2				5					
6	>	3									
7	+	2									
8	>	1									

Вывод по задаче: я реализовала двоичное дерево поиска по явному ключу.

### Задача №5. Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл]

Реализуйте простое двоичное дерево поиска.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val == key:
            return root
        elif root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root

def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def deleteNode(root, key):
    if root is None:
        return root
```

```

if key < root.val:
root.left = deleteNode(root.left, key)
elif key > root.val:
root.right = deleteNode(root.right, key)
else:
if root.left is None:
temp = root.right
root = None
return temp
elif root.right is None:
temp = root.left
root = None
return temp
temp = minValueNode(root.right)
root.val = temp.val
root.right = deleteNode(root.right, temp.val)
return root

```

```

def exists(root, key):
if root is None:
return False
if root.val == key:
return True
elif root.val < key:
return exists(root.right, key)
else:
return exists(root.left, key)

```

```

def nextNode(root, key):
if root is None:
return None
if root.val <= key:
return nextNode(root.right, key)
else:
left = nextNode(root.left, key)
if left is not None:

```



```

return left
else:
return root.val

def prevNode(root, key):
if root is None:
return None
if root.val >= key:
return prevNode(root.left, key)
else:
right = prevNode(root.right, key)
if right is not None:
return right
else:
return root.val

start()

root = None

with open("input.txt", "r") as f, open("output.txt",
"w") as g:
for line in f:
op, val = line.split()
val = int(val)

if op == "insert":
root = insert(root, val)
elif op == "delete":
root = deleteNode(root, val)
elif op == "exists":
g.write(str(exists(root, val)) + "\n")
elif op == "next":
nxt = nextNode(root, val)
g.write("none\n" if nxt is None else str(nxt) +
"\n")

```

```

elif op == "prev":
    prv = prevNode(root, val)
    g.write("none\n" if prv is None else str(prv) +
"\n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task5.py ×			input.txt ×			output.txt ×		
1	insert 2	✓	1	True				
2	insert 5		2	False				
3	insert 3		3	5				
4	exists 2		4	3				
5	exists 4		5	none				
6	next 4		6	3				
7	prev 4		7					
8	delete 5							
9	next 4							
10	prev 4							

Вывод по задаче: реализовано простейшее двоичное дерево поиска.

### Задача №6. Оpoznание 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 поддеревья меньше ключа вершины  $V$  ;
- все ключи вершин из правого поддеревья 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 ребенков.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def is_bst(node, min_val=float("-inf"), max_val=float("inf")):
    if not node:
        return True
    if node.key < min_val or node.key > max_val:
        return False
    return is_bst(node.left, min_val, node.key - 1) and
    is_bst(node.right, node.key + 1, max_val)
```

```

def build_tree(n, nodes):
    tree = {}
    for i in range(n):
        key, left, right = nodes[i]
        if left == -1:
            left_child = None
        else:
            left_child = Node(nodes[left][0])
        if right == -1:
            right_child = None
        else:
            right_child = Node(nodes[right][0])
        node = Node(key, left_child, right_child)
        tree[i] = node
    return tree

start()

with open("input.txt", "r") as f:
    n = int(f.readline().strip())
    nodes = [tuple(map(int, line.strip().split())) for line in f.readlines()]
    tree = build_tree(n, nodes)
    with open("output.txt", "w+") as g:
        if n == 0 or is_bst(tree[0]):
            g.write("CORRECT")
        else:
            g.write("INCORRECT")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task6.py ×		input.txt ×	⋮	output.txt ×
1	3	✓	1	CORRECT
2	2 1 2			
3	1 -1 -1			
4	3 -1 -1			

task6.py ×		input.txt ×	⋮	output.txt ×
1	3	✓	1	INCORRECT
2	1 1 2			
3	2 -1 -1			
4	3 -1 -1			

task6.py ×		input.txt ×	⋮	output.txt ×
1	0	✓	1	CORRECT

task6.py ×		input.txt ×	⋮	output.txt ×
1	5	✓	1	CORRECT
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 ×
1	7	✓	1	CORRECT
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 ×	
1	4	✓	1	INCORRECT	
2	4 1 -1				
3	2 2 3				
4	1 -1 -1				
5	5 -1 -1				

Вывод по задаче: в задаче реализован алгоритм проверки на двоичное дерево поиска.

### Задача №8. Высота дерева возвращается [2 s, 256 Mb, 2 балла]

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

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.key:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def height(root):
    if root is None:
```

```

return 0
else:
    left_height = height(root.left)
    right_height = height(root.right)

    if left_height > right_height:
        return left_height + 1
    else:
        return right_height + 1

start()

root = None

with open("input.txt") as f:
    n = int(f.readline())
    for i in range(n):
        key, left, right = map(int, f.readline().split())
        if i == 0:
            root = Node(key)
        else:
            insert(root, key)
    with open("output.txt", "w+") as g:
        g.write(str(height(root)))

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:



task8.py			input.txt			output.txt		
1	6	✓	1	4				
2	-2 0 2							
3	8 4 3							
4	9 0 0							
5	3 6 5							
6	6 0 0							
7	0 0 0							

Вывод по задаче: реализован алгоритм нахождения высоты двоичного дерева поиска.

### Задача №9. Удаление поддеревьев [2 s, 256 Mb, 2 балла]

Дано некоторое двоичное дерево поиска. Также даны запросы на удаление из него вершин, имеющих заданные ключи, причем вершины удаляются целиком вместе со своими поддеревьями.

После каждого запроса на удаление выведите число оставшихся вершин в дереве.

В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие  $10^9$ . Гарантируется, что данное дерево является двоичным деревом поиска, в частности, для каждой вершины дерева  $V$  выполняется следующее условие:

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

**Высота дерева не превосходит 25**, таким образом, можно считать, что оно сбалансировано.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

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

    def addNode(self, key):
        x = self.root
        y = None
        cmp = 0
        while x is not None:
            cmp = x.key - key
            if cmp == 0:
                return
```

```

else:
    y = x
    if cmp < 0:
        x = x.right
    else:
        x = x.left
    newNode = Node(key)
    if y is None:
        self.root = newNode
    else:
        if cmp > 0:
            y.left = newNode
        else:
            y.right = newNode

def removeSubtree(self, key):
    x = self.root
    y = None
    cmp = 0
    while x is not None:
        cmp = x.key - key
        if cmp == 0:
            break
        else:
            y = x
            if cmp < 0:
                x = x.right
            else:
                x = x.left

    if x is None:
        return 0

    count = self.nodesCount(x)
    if x.key > y.key:
        y.right = None
    else:
        y.left = None

```

```

x = None
return count

def nodesCount(self, node):
    if node.left is None and node.right is None:
        return 1
    left = right = 0
    if node.left is not None:
        left = self.nodesCount(node.left)
    if node.right is not None:
        right = self.nodesCount(node.right)
    return left + right + 1

start()

with open("input.txt") as f, open("output.txt", "w")
as g:
    nodesCount = int(f.readline())
    arrayNodes = []
    for i in range(nodesCount):
        line = list(map(int, f.readline().split()))
        arrayNodes.append(line)

    removesCount = int(f.readline())
    arrayRemove = list(map(int, f.readline().split()))

    tree = BinaryTree()
    for i in range(nodesCount):
        tree.addNode(arrayNodes[i][0])

    for i in range(removesCount):
        nodesCount -= tree.removeSubtree(arrayRemove[i])
    g.write(str(nodesCount) + "\n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task9.py			input.txt			output.txt		
1	6	✓	1	5				
2	-2 0 2		2	4				
3	8 4 3		3	4				
4	9 0 0		4	1				
5	3 6 5		5					
6	6 0 0							
7	0 0 0							
8	4							
9	6 9 7 8							

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: реализован алгоритм удаления поддеревя из сбалансированного дерева.

### Задача №10. Проверка корректности [2 s, 256 Mb, 2 балла]

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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

class BinarySearchTree:
    def __init__(self):
        self.value = 0
        self.left = 0
        self.right = 0

    def isBST(tree, j):
    def check(tree, j, k, c):
        if k == 0:
            if tree[j].value > c:
                return False
        elif k == 1:
            if tree[j].value < c:
                return False
            if tree[j].left != 0 and not check(tree,
            tree[j].left, k, c):
                return False
            if tree[j].right != 0 and not check(tree,
            tree[j].right, k, c):
                return False
            return True
        if tree[j].left == 0 and tree[j].right == 0:
            return True
```

```

if tree[j].left != 0 and tree[j].value <
tree[tree[j].left].value:
return False
if tree[j].right != 0 and tree[j].value >
tree[tree[j].right].value:
return False
if tree[j].left != 0 and not check(tree,
tree[j].left, 0, tree[j].value):
return False
if tree[j].right != 0 and not check(tree,
tree[j].right, 1, tree[j].value):
return False
if tree[j].left != 0 and not isBST(tree,
tree[j].left):
return False
if tree[j].right != 0 and not isBST(tree,
tree[j].right):
return False
return True

start()
with open("input.txt") as f, open("output.txt", "w")
as g:
n = int(f.readline())
if n == 0:
g.write("YES")
else:
tree = [BinarySearchTree() for _ in range(n + 1)]
for i in range(1, n + 1):
tree[i].value, tree[i].left, tree[i].right =
map(int, f.readline().split())
if isBST(tree, 1):
g.write("YES")
else:
g.write("NO")

print('Time:', str(process_time()), 'sec')

```

```
print('Memory usage:', str(get_traced_memory()[1] / 1024), 'KB')
```

Результат работы кода на примерах из текста задачи:

task10.py	input.txt	output.txt
1	6	1 YES
2	-2 0 2	
3	8 4 3	
4	9 0 0	
5	3 6 5	
6	6 0 0	
7	0 0 0	

task10.py	input.txt	output.txt
1	0	1 YES

task10.py	input.txt	output.txt
1	3	1 NO
2	5 2 3	
3	6 0 0	
4	4 0 0	

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: в задаче реализована проверка условия двоичного дерева поиска с помощью обхода его вершин.



### Задача №11. Сбалансированное двоичное дерево поиска [2 s, 512 Mb, 2 балла]

Реализуйте сбалансированное двоичное дерево поиска.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

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

    def height(self, node):
        if node is None:
            return 0
        return node.height

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

    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        y.left = x
        x.right = T2
```

```

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

return y

def right_rotate(self, y):
x = y.left
T2 = x.right

x.right = y
y.left = T2

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

return x

def insert(self, key):
def insert_helper(node, key):
if node is None:
return Node(key)

if key < node.key:
node.left = insert_helper(node.left, key)
elif key > node.key:
node.right = insert_helper(node.right, key)
else:
return node

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

if balance > 1 and key < node.left.key:

```

```

return self.right_rotate(node)

if balance < -1 and key > node.right.key:
return self.left_rotate(node)

if balance > 1 and key > node.left.key:
node.left = self.left_rotate(node.left)
return self.right_rotate(node)

if balance < -1 and key < node.right.key:
node.right = self.right_rotate(node.right)
return self.left_rotate(node)

return node

self.root = insert_helper(self.root, key)

def delete(self, key):
def delete_helper(node, key):
if node is None:
return node

if key < node.key:
node.left = delete_helper(node.left, key)
elif key > node.key:
node.right = delete_helper(node.right, key)
else:
if node.left is None:
temp = node.right
node = None
return temp
elif node.right is None:
temp = node.left
node = None
return temp

temp = self.get_min_value_node(node.right)
node.key = temp.key

```

```

node.right = delete_helper(node.right, temp.key)

if node is None:
    return node

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

if balance > 1 and self.balance(node.left) >= 0:
    return self.right_rotate(node)

if balance < -1 and self.balance(node.right) <= 0:
    return self.left_rotate(node)

if balance > 1 and self.balance(node.left) < 0:
    node.left = self.left_rotate(node.left)
    return self.right_rotate(node)

if balance < -1 and self.balance(node.right) > 0:
    node.right = self.right_rotate(node.right)
    return self.left_rotate(node)

return node

self.root = delete_helper(self.root, key)

def get_min_value_node(self, node):
    current = node

    while current.left is not None:
        current = current.left

    return current

def search(self, key):
def search_helper(node, key):
    if node is None:

```

```

return False

if key == node.key:
    return True
elif key < node.key:
    return search_helper(node.left, key)
else:
    return search_helper(node.right, key)

return search_helper(self.root, key)

def min_value_node(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def max_value_node(self, node):
    current = node
    while current.right is not None:
        current = current.right
    return current

def search(self, key):
    def search_helper(node, key):
        if node is None:
            return "false"

        if key == node.key:
            return "true"
        elif key < node.key:
            return search_helper(node.left, key)
        else:
            return search_helper(node.right, key)

    return search_helper(self.root, key)

def next(self, key):

```

```

def next_helper(node, key):
    if node is None:
        return None

    if node.key <= key:
        return next_helper(node.right, key)

    left = next_helper(node.left, key)
    if left is not None:
        return left

    return node

node = next_helper(self.root, key)
return node.key if node is not None else None

def prev(self, key):
    def prev_helper(node, key):
        if node is None:
            return None

        if node.key >= key:
            return prev_helper(node.left, key)

        right = prev_helper(node.right, key)
        if right is not None:
            return right

        return node

    node = prev_helper(self.root, key)
    return node.key if node is not None else None

start()

avl = AVLTree()

```

```

with open("output.txt", "w+") as g:
with open("input.txt") as f:
all_info = f.readlines()
for i in all_info:
if "\n" in i:
action = i.replace("\n", "")
else:
action = i
if "insert" in action:
avl.insert(int(action[-1]))
elif "exists" in action:
g.write(str(avl.search(int(action[-1]))) + "\n")
elif "delete" in action:
avl.delete(int(action[-1]))
elif "next" in action:
if str(avl.next(int(action[-1]))) == "None":
g.write("none" + "\n")
else:
g.write(str(avl.next(int(action[-1]))) + "\n")
elif "prev" in action:
if str(avl.prev(int(action[-1]))) == "None":
g.write("none" + "\n")
else:
g.write(str(avl.prev(int(action[-1]))) + "\n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task11.py ×			input.txt ×			output.txt ×		
1	insert	2	✓	1	true			
2	insert	5		2	false			
3	insert	3		3	5			
4	exists	2		4	3			
5	exists	4		5	none			
6	next	4		6	3			
7	prev	4		7				
8	delete	5						
9	next	4						
10	prev	4						

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: я узнала, что такое AVL-дерево, и научилась его реализовывать.



### Задача №12. Проверка сбалансированности [2 s, 256 Mb, 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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

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

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.key:
        node = insert(root.left, key)
        root.left = node
        node.parent = root
    else:
```

```

node = insert(root.right, key)
root.right = node
node.parent = root
return root

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

def AVL(root):
    if root is None:
        return True
    lh = height(root.left)
    rh = height(root.right)
    return str(rh - lh)

def print_balance(node, fileo):
    if node is None:
        return
    fileo.write(str((AVL(node))) + "\n")
    print_balance(node.left, fileo)
    print_balance(node.right, fileo)

start()

with open("input.txt") as f:
    n = int(f.readline())
    root = None
    for i in range(n):
        key, left, right = map(int, f.readline().split())
        root = insert(root, key)

```

```

with open("output.txt", "w+") as g:
print_balance(root, g)

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task12.py ×		input.txt ×		⋮		output.txt ×	
1	6	✓	1	3			
2	-2 0 2		2	-1			
3	8 4 3		3	0			
4	9 0 0		4	0			
5	3 6 5		5	0			
6	6 0 0		6	0			
7	0 0 0		7				

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: в задаче реализован алгоритм проверки сбалансированности дерева (то есть для любой вершины проверяется, что высоты её поддеревьев отличаются не более чем на 1)

### 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.

```
from collections import deque
from time import process_time
from tracemalloc import start, get_traced_memory

start()

result = []

class Node:
    def __init__(self, value, parent=None, left=None,
right=None, next=None):
        self.value = value
        self.parent = parent
        self.left = left
        self.right = right
        self.height = 0
        self.key = 0
        self.next = next
```

```

def get_tree(root):
    global result
    stack = deque()
    stack.append((root, (-1, -1)))

    while stack:
        elem, q = stack.popleft()
        if q[0] >= 0 and q[1] >= 0:
            result[q[0]][q[1]] = len(result) + 1
            if elem is None:
                continue

            result.append([elem.value, 0, 0])
            curr = len(result)
            if elem.left is not None:
                stack.append((elem.left, (curr - 1, 1)))
            if elem.right is not None:
                stack.append((elem.right, (curr - 1, 2)))

def rotate_left(node):
    if node is None or node.right is None:
        return node
    parent = node.parent
    right = node.right
    right_left = right.left
    if parent:
        if parent.right == node:
            parent.right = right
        else:
            parent.left = right
            right.parent, right.left = parent, node
            node.parent, node.right = right, right_left
    if right_left:
        right_left.parent = node
    return right

```

```

def rotate_right(node):
    if node is None or node.left is None:
        return node
    parent = node.parent
    left = node.left
    left_right = left.right
    if parent:
        if parent.left == node:
            parent.left = left
        else:
            parent.right = left
            left.parent, left.right = parent, node
            node.parent, node.left = left, left_right
    if left_right:
        left_right.parent = node
    return left

def arr_load(root, n):
    arr = deque()
    for i in range(1, n + 1):
        if root[i].left is None and root[i].right is None:
            root[i].height = 1
            arr.append(root[i])
    while arr:
        elem = arr.pop()
        if elem is None:
            continue
        if elem.parent is not None:
            elem.parent.height = max(elem.parent.height,
                                     elem.height + 1)
            arr.append(elem.parent)

def arr_res(root):
    curr_res = 0
    arr = deque()
    arr.append((root, 0))

```

```

while len(arr):
    elem = arr.pop()
    if elem[0] is None:
        continue
    arr.append((elem[0].left, elem[1] + 1))
    arr.append((elem[0].right, elem[1] + 1))
    curr_res = max(curr_res, elem[1] + 1)
return curr_res

start()

f = open("input.txt")
g = open("output.txt", "w")
n = int(f.readline())

node_arr = []
for _ in range(n + 1):
    node_arr.append(Node(0))
for j in range(n):
    key, left, right = map(int, f.readline().split())
    node_arr[j + 1].value = key
    if left:
        node_arr[j + 1].left, node_arr[left].parent =
        node_arr[left], node_arr[j + 1]
    if right:
        node_arr[j + 1].right, node_arr[right].parent =
        node_arr[right], node_arr[j + 1]

if node_arr[1].right is not None and
arr_res(node_arr[1].right.right)
arr_res(node_arr[1].right.left) == -1:
    rotate_right(node_arr[1].right)
    node_arr[1] = rotate_left(node_arr[1])
else:
    node_arr[1] = rotate_left(node_arr[1])

```

```

get_tree(node_arr[1])
g.write(f"{len(result)} \n")
for m, l, n in result:
    g.write(f"{m} {l} {n} \n")

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

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

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: в задаче реализован алгоритм левого поворота в двоичном дереве поиска.



### Задача №15. Удаление из AVL-дерева [2 s, 256 Mb, 3 балла]

Удаление из AVL-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.

```
from time import process_time
from tracemalloc import start, get_traced_memory

class Node:
    def __init__(self, num):
        self.key = num
```

```

self.left = None
self.right = None
self.height = 1

class Tree:
def height(self, root):
return root.height if root is not None else 0

def balance_factor(self, root):
return self.height(root.right) -
self.height(root.left)

def fix_height(self, root):
left = self.height(root.left)
right = self.height(root.right)
root.height = max(left, right) + 1

def rotateR(self, root):
q = root.left
root.left = q.right
q.right = root
self.fix_height(root)
self.fix_height(q)
return q

def rotateL(self, root):
p = root.right
root.right = p.left
p.left = root
self.fix_height(root)
self.fix_height(p)
return p

def balance(self, root):
self.fix_height(root)
if self.balance_factor(root) == 2:
if self.balance_factor(root.right) < 0:

```

```

root.right = self.rotateR(root.right)
return self.rotateL(root)
if self.balance_factor(root) == -2:
if self.balance_factor(root.left) > 0:
root.left = self.rotateL(root.left)
return self.rotateR(root)
return root

def insert(self, root, key):
if root is None:
return Node(key)
if key < root.key:
root.left = self.insert(root.left, key)
else:
root.right = self.insert(root.right, key)
return self.balance(root)

def find_right(self, root):
if root.right is not None:
return self.find_right(root.right)
return root

def find_right_and_delete(self, root):
if root.right is not None:
if root.right.right is None:
root.right = root.right.left if (root.right.left is
not None) else None
else:
root.right = self.find_right_and_delete(root.right)
return self.balance(root)

def remove(self, root, key):
if root is None:
return None
if key < root.key:
root.left = self.remove(root.left, key)
elif key > root.key:
root.right = self.remove(root.right, key)

```

```

else:
    if (root.left is None) and (root.right is None):
        return None
    elif root.left is None:
        return root.right
    else:
        new_root = self.find_right(root.left)
        root.key = new_root.key
        if root.left.key == new_root.key:
            root.left = None if (root.left.left is None) else
            root.left.left
        else:
            root.left = self.find_right_and_delete(root.left)
        return self.balance(root)

def get_str(self, root):
    if root is None:
        return None
    que = []
    number = 1
    que.append(root)
    ans = []
    while len(que) > 0:
        node = que.pop(0)
        line = f"{node.key} "
        if node.left is not None:
            number += 1
            line += f"{number} "
            que.append(node.left)
        else:
            line += "0 "
        if node.right is not None:
            number += 1
            line += f"{number}\n"
            que.append(node.right)
        else:
            line += "0\n"
        ans.append(line)

```

```

return ans

start()

f = open("input.txt")
n = int(f.readline())
lines = [f.readline() for _ in range(n)]
lines.reverse()
nodes = {}
tree = Tree()

for i in range(n):
    Ki, Li, Ri = map(int, lines[i].split())
    node = Node(int(Ki))
    nodes[n - i] = node

    if Li != 0 and Ri != 0:
        node.left = nodes[Li]
        node.right = nodes[Ri]
        tree.fix_height(node)
    elif Li != 0:
        node.left = nodes[Li]
        tree.fix_height(node)
    elif Ri != 0:
        node.right = nodes[Ri]
        tree.fix_height(node)
    else:
        node.height = 1

if len(nodes) == 0:
    nodes[1] = None

key = int(f.readline())
root = tree.remove(nodes[1], key)
g = open("output.txt", "w")
ans = tree.get_str(root)

if ans is not None:

```

```

g.write(f"{len(ans)}\n")
for i in ans:
    g.write(i)
else:
    g.write("0")

g.close()
f.close()

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task15.py ×		input.txt ×		output.txt ×	
1	3	✓	1	2	
2	4 2 3		2	3 0 2	
3	3 0 0		3	5 0 0	
4	5 0 0		4		
5	4				

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: реализован алгоритм удаления вершины из AVL-дерева с последующей балансировкой (то есть нужным поворотом).

### Задача №16. К-й максимум [2 s, 512 Mb, 3 балла]

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

```
from time import process_time
from tracemalloc import start, get_traced_memory

class Stack:
    def init(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def remove(self, item):
        self.stack.remove(item)

    def maximum(self, item):
        self.stack.sort(reverse=True)
        g.write(str(self.stack[item - 1]) + "\n")

with open("input.txt") as f:
    n = int(f.readline())
    arr1 = []
    for s in f.readlines():
        temp = s.split(" ")
        for i in temp:
            i = i.replace("\n", "")
            arr1.append(i)
    g = open("output.txt", "w+")

arr2 = Stack()
i = 0
while i <= len(arr1) - 1:
    if arr1[i] == "-1":
```

```

arr2.remove(int(arr1[i + 1]))
i += 2
elif arr1[i] == "+1":
arr2.push(int(arr1[i + 1]))
i += 2
elif arr1[i] == "0":
arr2.maximum(int(arr1[i + 1]))
i += 2
g.close()

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task16.py ×		input.txt ×		output.txt ×	
1	11	✓	1	7	
2	+1 5		2	5	
3	+1 3		3	3	
4	+1 7		4	10	
5	0 1		5	7	
6	0 2		6	3	
7	0 3		7		
8	-1 5				
9	+1 10				
10	0 1				
11	0 2				
12	0 3				

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: реализована структура данных по имени Stack, которая умеет добавлять и удалять элементы и искать k-й максимум.



### Задача №17. Множество с суммой [120 s, 512 Mb, 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:

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

```
from time import process_time
from tracemalloc import start, get_traced_memory

M = 1000000001

class HashTable:
    def __init__(self):
        self.table = {}
        self.last_sum = 0

    def add(self, x):
        self.table[(x + self.last_sum) % M] = True

    def remove(self, x):
        self.table.pop((x + self.last_sum) % M, None)

    def find(self, x, fileo):
        if (x + self.last_sum) % M in self.table:
            fileo.write("Found") + "\n"
        else:
            fileo.write("Not found") + "\n"

    def sum(self, l, r, fileo):
```

```

s = 0
for i in range(1 + self.last_sum, r + self.last_sum
+ 1):
    if i % M in self.table:
        s += i % M
    self.last_sum = s % M
fileo.write(str(self.last_sum) + "\n")

start()

hash_table = HashTable()

with open("input.txt") as f, open("output.txt",
"w+") as g:
    n = int(f.readline())
    for i in range(n):
        query = f.readline().split()
        if query[0] == "+":
            hash_table.add(int(query[1]))
        elif query[0] == "-":
            hash_table.remove(int(query[1]))
        elif query[0] == "?":
            hash_table.find(int(query[1]), g)
        elif query[0] == "s":
            hash_table.sum(int(query[1]), int(query[2]), g)

print('Time:', str(process_time()), 'sec')
print('Memory usage:', str(get_traced_memory()[1] /
1024), 'KB')

```

Результат работы кода на примерах из текста задачи:

task17.py	input.txt	output.txt
1	15	1 Not found
2	? 1	2 Found
3	+ 1	3 3
4	? 1	4 Found
5	+ 2	5 Not found
6	s 1 2	6 1
7	+ 1000000000	7 Not found
8	? 1000000000	8 10
9	- 1000000000	9
10	? 1000000000	
11	s 999999999 1000000000	
12	- 2	
13	? 2	
14	- 0	
15	+ 9	
16	s 0 9	
17		

task17.py	input.txt	output.txt
1	5	1 Not found
2	? 0	2 Found
3	+ 0	3 Not found
4	? 0	4
5	- 0	
6	? 0	

task17.py	input.txt	output.txt
1	5	1 Found
2	+ 491572259	2 Not found
3	? 491572259	3 491572259
4	? 899375874	4
5	s 310971296 877523306	
6	+ 352411209	

Проверка задачи на (openedu, астр и тд при наличии в задаче)

Вывод по задаче: реализована структура данных HashTable, которая добавляет, удаляет, находит и складывает элементы (это нужно было делать splay-деревом, но я с ним не очень разобралась).

## **Вывод**

В работе я изучила двоичные деревья поиска, их виды (красно-чёрное, АВЛ, splay) и различные алгоритмы на них (проверки на корректность, добавление и удаление вершин, повороты и различные структуры данных, которые могут работать на их основе).