

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 5

Выполнил:
Егоров Алексей Алексеевич
К3141

Проверила:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №5. Простое двоичное дерево поиска [1 баллов]	3
Задача №10. Проверка корректности [2 баллов]	12
Задача №16. К-й максимум [3 баллов]	17
Дополнительные задачи	24
Задача №7. Оpozнание двоичного дерева поиска (усложненная версия) [2.5 баллов]	24
Задача 12. Проверка сбалансированности [2 баллов]	28
Задача №13. Делаю я левый поворот... [3 баллов]	32
Задача №14. Вставка в AVL-дерево [3 баллов]	39
Задача №15. Удаление из AVL-дерева [3 баллов]	46
Вывод	54

Задачи по варианту

Задача №5. Простое двоичное дерево поиска [1 баллов]

Реализуйте простое двоичное дерево поиска.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание операций с деревом, их количество N не превышает 100. В каждой строке находится одна из следующих операций:
 - insert x – добавить в дерево ключ x . Если ключ x есть в дереве, то ничего делать не надо;
 - delete x – удалить из дерева ключ x . Если ключа x в дереве нет, то ничего делать не надо;
 - exists x – если ключ x есть в дереве выведите «true», если нет – «false»;
 - next x – выведите минимальный элемент в дереве, строго больший x , или «none», если такого нет;
 - prev x – выведите максимальный элемент в дереве, строго меньший x , или «none», если такого нет.

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

- **Ограничения на входные данные.** $0 \leq N \leq 100$, $|x_i| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 512 мб.
- **Пример:**

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

Листинг кода.

```
import time
import tracemalloc

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

```

class BinTree:
    def __init__(self):
        self.root = None
        self.pred = None

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

    def delete(self, x, root):
        if root is None:
            return None
        if x < root.key:
            root.left = self.delete(x, root.left)
        elif x > root.key:
            root.right = self.delete(x, root.right)
        else:
            if root.left is None and root.right is None:
                return None
            elif root.left is None:
                return root.right
            elif root.right is None:
                return root.left
            else:
                next_root = self.find_min(root.right)
                root.key = next_root.key
                root.right = self.delete(next_root.key, root.right)
        return root

```

```

def exist(self, x, root):
    if root is None:
        return False
    if x < root.key:
        return self.exist(x, root.left)
    elif x > root.key:
        return self.exist(x, root.right)
    else:
        return True

def find_min(self, root):
    while root and root.left:
        root = root.left
    return root

def find_max(self, root):
    while root and root.right:
        root = root.right
    return root

def next(self, x, root):
    if root is None:
        return None
    successor = None
    while root:
        if root.key > x:
            successor = root
            root = root.left
        else:
            root = root.right
    return successor.key if successor else None

def prev(self, x, root):
    if root is None:
        return None
    predecessor = None

```

```

while root:
    if root.key < x:
        predecessor = root
        root = root.right
    else:
        root = root.left
return predecessor.key if predecessor else None

def insert(self, x):
    self.root = self.ins(x, self.root)

def delete_key(self, x):
    self.root = self.delete(x, self.root)

def exists(self, x):
    return self.exist(x, self.root)

def next_key(self, x):
    return self.next(x, self.root)

def prev_key(self, x):
    return self.prev(x, self.root)

def main():
    with open("input.txt", "r") as input_file, open("output.txt", "w") as
output_file:
        bin_tree = BinTree()
        for line in input_file:
            command, x = line.strip().split()
            x = int(x)
            if command == "insert":
                bin_tree.insert(x)
            elif command == "delete":
                bin_tree.delete_key(x)
            elif command == "exists":
                output_file.write("true\n" if bin_tree.exists(x) else "false\n")

```

```

elif command == "next":
    result = bin_tree.next_key(x)
    output_file.write(f"{result}\n" if result is not None else "none\n")
elif command == "prev":
    result = bin_tree.prev_key(x)
    output_file.write(f"{result}\n" if result is not None else "none\n")

if __name__ == "__main__":
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f"Время работы: {t_end - t_start} секунд")
    print(f"Затраченная память: {tracemalloc.get_traced_memory()}")
    tracemalloc.stop()

```

Текстовое объяснение решения.

В задаче происходит построчное считывание команд, пока они не заканчиваются, которые запускают соответствующие им команды.

В данной задаче реализован класс дерева поиска со следующими командами:

- команда *insert* запускает добавление элемента в дерево. Эта команда принимает только значение, которое требуется добавить, чтобы потом запустить команду с передачей в неё дополнительно переменной *root*, которая или является пустой, из-за чего мы создаем *Node*, являющейся нашим листом в дереве, или сравниваем её значение с поступившим *x*, чтобы выбрать в правое или левое поддереву должна добавиться переменная, а потом продолжаем рекурсию добавления от правого или левого поддерева соответственно, что закончится только когда оно дойдет до пустого места, где и появится *Node* со значением *x*.
- команда *delete_key* запускает удаление элемента в дереве по заданному ключу *x*. В этом методе запускается рекурсивный метод *delete* с передачей в неё *x* и *root*. Уже в методе *delete* идет проверка на пустоту переданной ноды, чтобы вернуть *None*, если *нода* пустая.

Дальше идет сравнение x со значением переданной в неё `root`. Если элемент x меньше, то запускается удаление от левого поддерева, если больше, то запускается удаление от правого поддерева. В случае, если x равен значению `root`, то начинается удаление нынешней ноды `root`. Здесь есть 3 сценария удаления. Если нода является листом, то просто идет возвращение `None`. Если у ноды есть только одно правое поддерево или левое, то происходит возвращение только правого или левого поддерева, чтобы связать разорвавшееся дерево. Если у ноды есть и правое и левое поддерево, то запускается поиск минимального элемента от правого поддерева, что даст нам следующий элемент в дереве по значению, чтобы записать в нашу нынешнюю позицию значение следующего элемента и не нарушить структуру дерева. Теперь остается удалить ноду, чье значение мы скопировали. Итогом является удаление ноды по ключу

- Функция поиска минимального элемента проста, мы идем по левым поддеревам пока не уткнемся в отсутствие левого поддерева, что означает, что мы находимся в минимальном элементе для изначально запущенной позиции поиска.
- Функция поиска максимального элемента проста, мы идем по правым поддеревам пока не уткнемся в отсутствие правого поддерева, что означает, что мы находимся в максимальном элементе для изначально запущенной позиции поиска.
- Команда *exists* запускает проверку на наличие элемента x в дереве. В ней рекурсивно запускается *exist* с передачей в неё `root` и x . В ней идет проверка на пустоту переданной ноды, чтобы вернуть в этом случае `false`. Дальше идет сравнение x со значением ноды, чтобы либо запустить поиск по левому или правому поддереву, если x не равно значению, либо возвращение `true`, если значения совпали.
- Команда *next_key* запускает поиск следующего значения в дереве после заданного ключа x . Для этого запускается метод *next* с передачей в неё x и `root`. В нем проходит проверка на пустоту ноды, если пустая, то возвращается `None`. Дальше создаётся переменная `successor` равная `None`, чтобы после выполнения блока с циклом поиска, потом вернуть только значение `successor`, если в него запишется следующий элемент. В цикле мы сравниваем значение текущей ноды с x , если оно больше то мы записываем в `successor`

текущую ноду и переходим левое поддерево, если иначе, то просто переходим в правое поддерево.

- Команда *prev_key* запускает поиск предыдущего значения в дереве перед заданным ключем *x*. Для этого запускается метод *prev* с передачей в неё *x* и *root*. В нем проходит проверка на пустоту ноды, если пустая, то возвращается *None*. Далее создаётся переменная *successor* равная *None*, чтобы после выполнения блока с циклом поиска, потом вернуть только значение *successor*, если в него запишется предыдущий элемент. В цикле мы сравниваем значение текущей ноды с *x*, если оно меньше, то мы записываем в *successor* текущую ноду и переходим правое поддерево, если иначе, то просто переходим в левое поддерево.

Результат работы кода на примерах из текста задачи:

generate_tests.py × 5.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			

Результат работы кода на максимальных и минимальных значениях:

```

generate_tests_5.py × input.txt × 5.py × output.txt ×
78 insert 999999921 ✓ 1 true
79 insert 999999924 2 false
80 insert 999999923 3 false
81 insert 999999918 4 999999926
82 insert 999999917 5 true
83 insert 999999920 6
84 insert 999999919
85 insert 999999914
86 insert 999999913
87 insert 999999916
88 insert -999999915
89 insert -999999910
90 insert 999999909
91 insert 999999912
92 insert 999999911
93 insert 999999906
94 insert 999999905
95 insert 999999908
96 insert 999999907
97 insert -999999900
98 insert -1000000000
99 exists -1000000000
100

```

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.0002579999854788184	27,50
Пример из задачи	0.0003996999002993107	33,35
Верхняя граница диапазона значений входных данных из текста задачи	0.001783600077033043	66,56

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №10. Проверка корректности [2 баллов]

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

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

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

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

В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i + 1)$ -ой строке файла ($1 \leq i \leq N$) 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).

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

Примеры:	input.txt	output.txt	input.txt	output.txt
	6	YES	0	YES
	-2 0 2			
	8 4 3		input.txt	output.txt
	9 0 0		3	NO
	3 6 5		5 2 3	
	6 0 0		6 0 0	
	0 0 0		4 0 0	

- Проверить можно по [ссылке](#), OpenEdu, курс "Алгоритмы программирования и структуры данных 6 неделя, наблюдаемая задача.

Листинг кода.

```
import sys
sys.setrecursionlimit(200000)
import time
import tracemalloc

class Node:
    def __init__(self, key, left, right):
        self.key = key
        self.left = left
        self.right = right
        self.left_bord = -sys.maxsize
```

```

self.right_bord = sys.maxsize

def checkTree(bin_tree, i):
    # print(f'chT Key: {bin_tree[i].key} left {bin_tree[i].left} right
    {bin_tree[i].right} l_bord {bin_tree[i].left_bord} r_bord
    {bin_tree[i].right_bord}')
    if bin_tree[i].left != -1 and bin_tree[i].key < bin_tree[bin_tree[i].left].key:
        return False
    if bin_tree[i].right != -1 and bin_tree[i].key > bin_tree[bin_tree[i].right].key:
        return False

    if bin_tree[i].key <= bin_tree[i].left_bord:
        return False
    if bin_tree[i].key >= bin_tree[i].right_bord:
        return False

    if bin_tree[i].left != -1:
        # print(" l_b_m ")
        bin_tree[bin_tree[i].left].left_bord = bin_tree[i].left_bord
        bin_tree[bin_tree[i].left].right_bord = bin_tree[i].key

    if bin_tree[i].right != -1:
        # print(" r_b_m ")
        bin_tree[bin_tree[i].right].left_bord = bin_tree[i].key
        bin_tree[bin_tree[i].right].right_bord = bin_tree[i].right_bord

    if bin_tree[i].left != -1 and not checkTree(bin_tree, bin_tree[i].left):
        print("l_fal")
        return False
    if bin_tree[i].right != -1 and not checkTree(bin_tree, bin_tree[i].right):
        # print("r_fal")
        return False

    return True

def main():
    with open("input.txt", "r") as input_file:

```

```

n = int(input_file.readline().strip())
if n == 0:
    with open("output.txt", "w") as output_file:
        output_file.write("YES")
    return

bin_tree = []
for _ in range(n):
    k, l, r = map(int, input_file.readline().strip().split())
    bin_tree.append(Node(k, l-1, r-1))

if checkTree(bin_tree, 0):
    with open("output.txt", "w") as output_file:
        output_file.write("YES")
else:
    with open("output.txt", "w") as output_file:
        output_file.write("NO")

if __name__ == "__main__":
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f"Время работы: {t_end - t_start} секунд")
    print(f"Затраченная память: {tracemalloc.get_traced_memory()}")
    tracemalloc.stop()

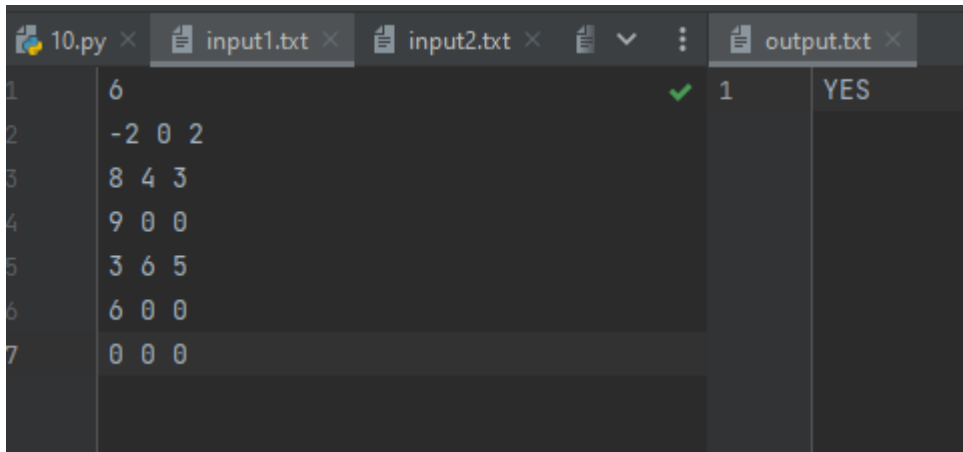
```

Текстовое объяснение решения.

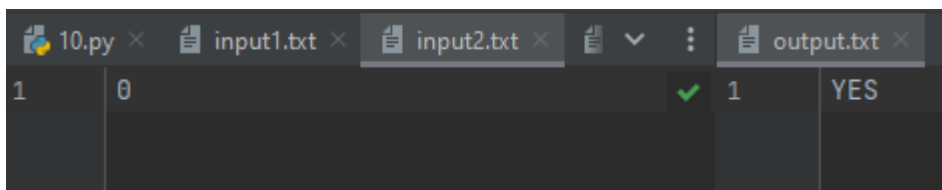
При считывании данных мы записываем их в массив состоящий из различных Node. Каждая Node имеет в себе следующие параметры: *key* – ключ узла, *left* – левый ребёнок, *right* – правый ребёнок, *left_bord* - левая граница(по умолчанию равно минимальному значению), *right_bord* - правая граница(по умолчанию равно максимальному значению).

После этого программа начинает проверку чтобы каждый левый ребёнок был меньше своего родителя каждый правый был больше. Затем рекурсивно запускается проверка от левого и правого ребёнка.

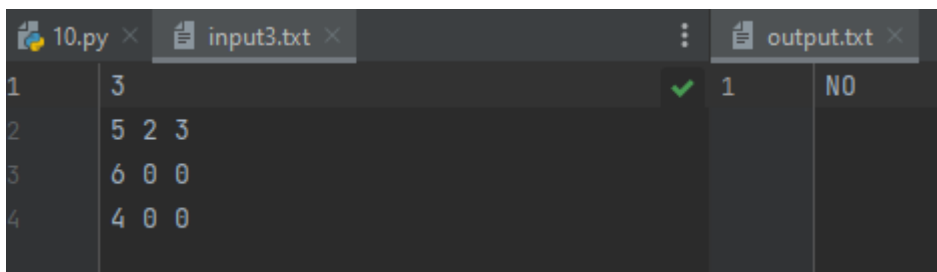
Результат работы кода на примерах из текста задачи:



```
10.py x input1.txt x input2.txt x v ... output.txt x
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
```



```
10.py x input1.txt x input2.txt x v ... output.txt x
1 0 ✓ 1 YES
```



```
10.py x input3.txt x ... output.txt x
1 3 ✓ 1 NO
2 5 2 3
3 6 0 0
4 4 0 0
```

Результат работы кода на максимальных и минимальных значениях:

Проверка минимальных значений уже выполнена в примерах самой задачи

Проверка на максимальных:

```

10.py × input.txt × output.txt ×
The file size (3,89 MB) exceeds the configured limit (2,56 MB)....
1 200000 ✓
2 1000000000 2 0
3 999999999 3 0
4 999999998 4 0
5 999999997 5 0
6 999999996 6 0
7 999999995 7 0
8 999999994 8 0
9 999999993 9 0
10 999999992 10 0
11 999999991 11 0

```

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.00037880009040236473	17,40
Пример из задачи	0.00047289999201893806	17,97
Пример из задачи	0.00037880009040236473	17,40
Пример из задачи	0.0004029000410810113	17,71
Верхняя граница диапазона значений входных данных из текста задачи	2.024702	66793,56

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №16. К-й максимум [3 баллов]

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

- **Формат ввода / входного файла (input.txt).** Первая строка входного файла содержит натуральное число n – количество команд. Последующие n строк содержат по одной команде каждая. Команда записывается в виде двух чисел c_i и k_i – тип и аргумент команды соответственно. Поддерживаемые команды:

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

Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе k_i -го максимума, он существует.

- **Ограничения на входные данные.** $n \leq 100000$, $|k_i| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число – k_i -й максимум.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 512 мб.
- Пример:

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

Листинг кода

```
import time
```

```
import tracemalloc
```

```
result = []
```

```
class Node:
```

```
    def __init__(self, key):
```

```
        self.key = key
```

```
        self.left = None
```

```
self.right = None
```

```
class Tree:
```

```
    def __init__(self):
```

```
        self.root = None
```

```
    def node_find(self, node, parent, key):
```

```
        if node is None:
```

```
            return None, parent, False
```

```
        if key == node.key:
```

```
            return node, parent, True
```

```
        if key < node.key:
```

```
            if node.left:
```

```
                return self.node_find(node.left, node, key)
```

```
        if key > node.key:
```

```
            if node.right:
```

```
                return self.node_find(node.right, node, key)
```

```
        return node, parent, False
```

```
    def node_find_min(self, node, p):
```

```
        if node.left:
```

```
            return self.node_find_min(node.left, node)
```

```
        return node, p
```

```
# =====
```

```

def append(self, node):
    if self.root is None:
        self.root = node
        return node

    root, parent, key_find = self.node_find(self.root, None, node.key)
    if not key_find and root:
        if node.key < root.key:
            root.left = node
        else:
            root.right = node

    return node

```

=====

```

def delete_child(self, s, p):
    if p.left == s:
        if s.left is None:
            p.left = s.right
        elif s.right is None:
            p.left = s.left
    elif p.right == s:
        if s.left is None:
            p.right = s.right
        elif s.right is None:
            p.right = s.left

def delete_node(self, key):
    s, p, fl_find = self.node_find(self.root, None, key)

```

```

if not fl_find:
    return None

if s.left is None and s.right is None:
    if p:
        if p.left == s:
            p.left = None
        elif p.right == s:
            p.right = None
    else:
        self.root = None
elif s.left is None or s.right is None:
    self.delete_child(s, p)
else:
    sr, pr = self.node_find_min(s.right, s)
    s.key = sr.key
    self.delete_child(sr, pr)

# =====

def get_ki_max(self, key):
    self.get_ki(self.root, key, 0)

def get_ki(self, node, key, index):
    if node is None:
        return index
    index = self.get_ki(node.right, key, index)
    index += 1
    if index == key:
        result.append(str(node.key))

```

```

        return index
    return self.get_ki(node.left, key, index)

def main():
    with open("input.txt", "r") as f:
        data = f.readlines()
    n = int(data[0])
    tree = Tree()
    for i in range(n):
        s = data[i + 1].split()
        if s[0] == "+1" or s[0] == "1":
            tree.append(Node(int(s[1])))
        elif s[0] == "0":
            tree.get_ki_max(int(s[1]))
        else:
            tree.delete_node(int(s[1]))

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

if __name__ == '__main__':
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f'Время: {t_end - t_start} секунд')
    print(f'Память: {tracemalloc.get_traced_memory()}')
    tracemalloc.stop()

```

Текстовое объяснение решения

Во время добавления элементов в дерево, они также добавляются и в дерево с максимальными значениями. Что позволяет нам при запуске вывода Ki максимума спокойно находить его по этому дополнительному дереву.

При удалении элемента из дерева выполняется стандартная процедура удаления элемента. Если это лист то он просто удаляется, если он имеет только одного наследника то он вырезается, а если это элемент у которого есть и левый, и правый, то запускается поиск элемента на его место в котором он меняется местами и становится листом, а лист можно спокойно удалить.

Результат работы кода на примерах из текста задачи:

input.txt			2-lab\...\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				
8	-1 5				
9	+1 10				
10	0 1				
11	0 2				
12	0 3				

Результат работы кода на максимальных и минимальных значениях:

input.txt			2-lab\...\output.txt	
1	1	✓	1	
2	0 1			

input.txt			2-lab\...\output.txt	
1	100000	✓	1	-978437567
2	+1 -995697359		2	-983657345
3	+1 -992545355		3	2363456
4	+1 -986456335		4	2234624
5	+1 -983657345		5	2031543
6	+1 -978437567		6	1934653
7	0 1		7	1736346
8	0 2		8	1513463

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.0003538999008014798	18,19
Пример из задачи	0.005854399991221726	18,79
Верхняя граница диапазона значений входных данных из текста задачи	1.7103000999195501	24425,63

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Дополнительные задачи

Задача №7. Опознание двоичного дерева поиска (усложненная версия) [2.5 баллов]

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

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

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

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

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

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

- **Ограничения на входные данные.** $0 \leq n \leq 10^5$, $-2^{31} \leq K_i \leq 2^{31} - 1$, $-1 \leq L_i, R_i \leq n - 1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \neq -1$ и $R_i \neq -1$, то $L_i \neq R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла. Обратите внимание, что минимальное и максимальное возможные значения 32-битного целочисленного типа могут быть ключами в дереве.
- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

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

- Примеры:

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

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

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

Листинг кода

```
import time
import tracemalloc
```



```

def is_bst(tree):
    if not tree:
        return True

    def is_bst_node(index, min_key, max_key):
        if index == -1:
            return True

        key, left_index, right_index = tree[index]
        if min_key > key or key >= max_key:
            return False
        return is_bst_node(left_index, min_key, key) and is_bst_node(right_index,
key, max_key)

    return is_bst_node(0, float('-inf'), float('inf'))

def main():
    with open("input.txt", "r") as f:
        data = f.readlines()
        n = int(data[0])

    tree = []
    for i in range(n):
        node = list(map(int, data[i + 1].split()))
        tree.append((node[0], node[1], node[2]))

    with open("output.txt", "w") as f:
        if is_bst(tree):
            f.write("CORRECT")
        else:
            f.write("INCORRECT")

if __name__ == "__main__":
    tracemalloc.start()

```

```

t_start = time.perf_counter()
main()
t_end = time.perf_counter()
print(f"Время: {t_end - t_start} секунд")
print(f"Память: {tracemalloc.get_traced_memory()}")
tracemalloc.stop()

```

Текстовое объяснение решения

Данная программа проверяет все ли вершины в левом поддереве меньше корневого узла, и все ли вершины в правом поддереве больше корневого узла или равны ему.

Во время использования рекурсивного метода мы будем нести за собой как минимальное значение, так и максимальное для следующей ноды. Таким образом, в случае, если элемент ноды выходит за границы, то это означает что это не двоичное дерево.

Результат работы кода на примерах из текста задачи:

<div>input1.txt ×</div> <div>output.txt ×</div> <div>1 3 ✓ 1 CORRECT</div> <div>2 2 1 2</div> <div>3 1 -1 -1</div> <div>4 3 -1 -1</div>	<div>input2.txt ×</div> <div>output.txt ×</div> <div>1 3 ✓ 1 INCORRECT</div> <div>2 1 1 2</div> <div>3 2 -1 -1</div> <div>4 3 -1 -1</div>
<div>input3.txt ×</div> <div>output.txt ×</div> <div>1 3 ✓ 1 CORRECT</div> <div>2 2 1 2</div> <div>3 1 -1 -1</div> <div>4 2 -1 -1</div>	<div>input4.txt ×</div> <div>output.txt ×</div> <div>1 3 ✓ 1 INCORRECT</div> <div>2 2 1 2</div> <div>3 2 -1 -1</div> <div>4 3 -1 -1</div>

input5.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	

input6.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	

input7.txt	output.txt
1 1	1 CORRECT
2 2147483647 -1 -1	

Результат работы кода на максимальных и минимальных значениях:

inputmin.txt	output.txt
1 0	1 CORRECT

inputM.txt	output.txt
1 100000	1 CORRECT
2 -979227428 1 58422	
3 -921861124 2 58958	
4 -868644615 3 63373	
5 -595000748 4 59501	
6 -545943357 5 68703	
7 -522016866 6 63983	
8 -466691543 7 61569	

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.0004453000146895647	18,50
Пример из задачи	0.0005330999847501516	18,80
Пример из задачи	0.00043210003059357405	18,80

Пример из задачи	0.000401900033466517 9	18,80
Пример из задачи	0.000557500054128468	18,80
Пример из задачи	0.000420999946072697 64	18,91
Пример из задачи	0.000425799982622265 8	19,02
Пример из задачи	0.000339700025506317 6	18,69
Верхняя граница диапазона значений входных данных из текста задачи	0.5882835000520572	23,60

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача 12. Проверка сбалансированности [2 баллов]

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.

- **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.

- **Ограничения на входные данные.** $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$.

- **Формат вывода / выходного файла (output.txt).** Для i -ой вершины в i -ой строке выведите одно число – баланс данной вершины.

- **Ограничение по времени.** 2 сек.

- **Ограничение по памяти.** 256 мб.

- **Пример:**

input.txt	output.txt
6	3
-2 0 2	-1
8 4 3	0
9 0 0	0
3 6 5	0
6 0 0	0
0 0 0	

- Проверить можно по [ссылке](#), OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 1 задача.

Листинг кода.

```
import time
```

```
import tracemalloc
```

```
def height(nodes, index_node):
```

```
    if index_node == 0:
```

```
        return 0
```

```
    return nodes[index_node][3]
```

```
def check_balance(nodes, index_node):
```

```

if index_node == 0:
    return

check_balance(nodes, nodes[index_node][1]) #check left
check_balance(nodes, nodes[index_node][2]) #check right

nodes[index_node][3] = 1 + max(height(nodes, nodes[index_node][1]),
height(nodes, nodes[index_node][2])) #получение высоты
nodes[index_node][4] = height(nodes, nodes[index_node][2]) - height(nodes,
nodes[index_node][1]) #получение разницы

def main():
    with open("input.txt") as f:
        n = int(f.readline())
        data = f.readlines()

        Nodes = [None] * (n + 1)
        for i, node in enumerate(data, 1):
            Nodes[i] = list(map(int, node.split())) + [1, 0] #[1, 0] для высот

    if n != 0:
        check_balance(Nodes, 1)

    with open("output.txt", "w") as f:
        for node in Nodes[1:]:
            f.write(str(node[4]) + '\n')

if __name__ == '__main__':
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f'Время: {t_end - t_start} секунд')
    print(f'Память: {tracemalloc.get_traced_memory()}')
    tracemalloc.stop()

```

Текстовое объяснение решения.

Список Nodes состоит из списков, соответствующих вершинам дерева. В каждом таком списке хранятся в следующей последовательности: ключ вершины, индекс её левого ребёнка, индекс её правого ребёнка, высота поддерева с корнем в этой вершине, баланс вершины.

За определения баланса каждого узла дерева отвечает функция `check_balance`, в которую передаются массив вершин дерева и индекс текущей вершины. Сначала она вызывает саму себя для левого и правого дочерних элементов текущего узла, затем вычисляет высоту текущего узла как максимальную высоту среди его дочерних элементов плюс один, после она вычисляет баланс вершины как разницу высоты правого и левого поддеревьев. Все значения заносятся в список Nodes.

Результат работы кода на примерах из текста задачи:

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

Результат работы кода на максимальных и минимальных значениях:

input.txt			2-lab\...\output.txt
1	0	✓	1

input.txt			2-lab\...\output.txt
The file size (4,76 MB) exceeds the config...			
1	200000	✓	49999
2	-945839445 2 51438		49998
3	-773094202 3 59567		49997
4	-772326334 4 68198		49996
5	-750897776 5 55057		49995
6	-709201560 6 53962		49994
7	-669189168 7 53964		49993
			49992
			49991

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.0003645999822765589	18,50
Пример из задачи	0.0005763999652117491	18,54
Верхняя граница диапазона значений входных данных из текста задачи	1,9893845678924987444	47634,80

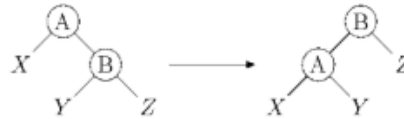
Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №13. Делаю я левый поворот... [3 баллов]

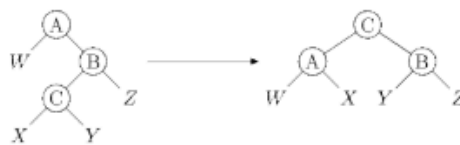
Для балансировки AVL-дерева при операциях вставки и удаления производятся *левые* и *правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем -1.

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

- Проверить можно по [ссылке](#), OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 2 задача.

Листинг кода

```
import time
import tracemalloc
```

```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
        self.output_index = 0

class AVLTree():
    def __init__(self, Nodes, index=1):
        self.root = self.create_tree(Nodes, index)

    def create_tree(self, Nodes, index):
        if len(Nodes) == 1:
            return

        if index == 0:
            return

        root = Node(Nodes[index][0])
        root.left = self.create_tree(Nodes, Nodes[index][1])
        root.right = self.create_tree(Nodes, Nodes[index][2])
        self.fix_height(root)
        return root

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

```

```
return node.height
```

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

```
def rotate_left(self, node):  
    p = node.right  
    node.right = p.left  
    p.left = node  
    return p
```

```
def rotate_right(self, node):  
    q = node.left  
    node.left = q.right  
    q.right = node  
    return q
```

```
def big_left_rotate(self, node):  
    node.right = self.rotate_right(node.right)  
    return self.rotate_left(node)
```

```
def get_balance(self, node):  
    return self.height(node.right) - self.height(node.left)
```

```
def print_tree(self):
```

```

def tree_queue(root):
    if root is None:
        return
    nonlocal index
    root.output_index = index
    index += 1
    tree_queue(root.left)
    tree_queue(root.right)

```

```

def print_tree_queue(root):
    if root is None:
        return
    nonlocal Nodes
    Nodes.append(map(str, (root.key,
root.left.output_index if not root.left is None else '0',
root.right.output_index if not root.right is None else '0')))
    print_tree_queue(root.left)
    print_tree_queue(root.right)

```

```

index = 1
Nodes = []
tree_queue(self.root)
print_tree_queue(self.root)
return Nodes

```

```

def main():
    with open("input.txt") as f:
        n = int(f.readline())
        Nodes = [None] + [tuple(map(int, node.split())) for node in f.readlines()]

```

```

tree = AVLTree(Nodes)

if tree.get_balance(tree.root.right) == -1: #финальный поворот
    tree.root = tree.big_left_rotate(tree.root)
else:
    tree.root = tree.rotate_left(tree.root)

with open("output.txt", "w") as f:
    f.write(str(n) + '\n')
    for node in tree.print_tree():
        f.write(' '.join(node) + '\n')

if __name__ == '__main__':
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f'Время: {t_end - t_start} секунд')
    print(f'Память: {tracemalloc.get_traced_memory()}')
    tracemalloc.stop()

```

Текстовое объяснение решения

Реализован класс Node, который представляет узел в двоичном дереве поиска, со следующими атрибутами: key – ключ узла, left – левый ребёнок, right – правый ребёнок, height – высота поддерева, output_index – индекс элемента при печати дерева. Функция create_tree рекурсивно запускает создание двоичного дерева. Входные данные занесены в список Nodes,

состоящий из кортежей, где в каждом кортеже хранятся ключ узла и индексы его детей. Функция запускается с корневого узла и рекурсивно создает левое и правое поддеревья.

Вычисляется баланс правого дочернего узла. Если он равен -1, то вызывается функция `big_left_rotate`, которая делает большой левый поворот, представляющий из себя комбинацию малого правого поворота (функция `rotate_right`) только правого поддерева и последующего левого поворота (функция `rotate_left`). В противном случае производится просто `rotate_left`.

Функция `rotate_left` делает следующее: она устанавливает текущий корень в качестве левого ребёнка правого узла (левый ребёнок правого узла становится правым ребёнком текущего корня), а затем возвращает этот правый узел в качестве нового корня.

Функция `rotate_right` аналогично делает следующее: она устанавливает текущий корень в качестве правого ребенка левого узла (правый ребёнок левого узла становится левым ребёнком текущего корня), а затем возвращает этот левый узел в качестве нового корня.

Печать дерева в требуемом по условию задачи формате осуществляет функция `print_tree`. Внутри неё есть две вложенные функции: `tree_queue` определяет индексы всех узлов в новом списке и присваивает их `output_index`; `print_tree_queue` возвращает список кортежей, хранящих ключ узла, индексы его левого и правого ребёнка.

Результат работы кода на примерах из текста задачи:

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

Результат работы кода на максимальных и минимальных значениях:

input.txt		2-lab\...\output.txt	
1	3	1	3
2	5 0 2	2	7 2 3
3	7 0 3	3	5 0 0
4	10 0 0	4	10 0 0

input.txt			2-lab\...\output.txt		
1	200000	✓	1	200000	
2	142574835 2 109234		2	535854632 2 185572	
3	-253257855 3 68092		3	142574835 3 109235	
4	-457496364 4 35351		4	-253257855 4 68093	
5	-647364263 5 36732		5	-457496364 5 35352	
6	-753567354 6 2654		6	-647364263 6 36733	

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.000466400058940053	18,46
Пример из задачи	0.005782300024293363	21,62
Верхняя граница диапазона значений входных данных из текста задачи	3,20534723	76746,39

Вывод по задаче: Программа корректно работает на всех приведенных тестах. Однако на максимальных входных данных программа выходит за ограничения по времени. Изучив работу программы я сделал вывод, что на работоспособность сильно влияет чтение входных данных и вывод готовых, что приводит к выходу за границы ограничения по времени

Задача №14. Вставка в AVL-дерево [3 баллов]

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.

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

В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i + 1)$ -ой строке файла ($1 \leq i \leq N$) 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 корректным AVL-деревом.

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

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

Пример:

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

- Проверить можно по [ссылке](#), OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 3 задача.

Листинг кода

```
import time
import tracemalloc
```



```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
        self.output_index = 0

class AVLTree():
    def __init__(self, n, Nodes, index=1):
        self.root = self.create_tree(Nodes, index)
        self.count_of_nodes = n

    def create_tree(self, Nodes, index):
        if len(Nodes) == 1:
            return
        if index == 0:
            return
        root = Node(Nodes[index][0])

        root.left = self.create_tree(Nodes, Nodes[index][1])
        root.right = self.create_tree(Nodes, Nodes[index][2])

        self.fix_height(root)
        return root

# =====

    def height(self, node):

```

```
if node is None:
    return 0
return node.height
```

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

```
# =====
```

```
def rotate_left(self, node):
    p = node.right
    node.right = p.left
    p.left = node
    return p
```

```
def rotate_right(self, node):
    q = node.left
    node.left = q.right
    q.right = node
    return q
```

```
# =====
```

```
def get_balance(self, node):
    return self.height(node.right) - self.height(node.left)
```

```
def balance_node(self, node):
    self.fix_height(node)
```

```

if self.get_balance(node) == 2:
    if self.get_balance(node.right) < 0:
        node.right = self.rotate_right(node.right)
    return self.rotate_left(node)
if self.get_balance(node) == -2:
    if self.get_balance(node.left) > 0:
        node.left = self.rotate_left(node.left)
    return self.rotate_right(node)
return node

```

=====

```

def insert(self, node, key):
    if node is None:
        self.count_of_nodes += 1
        return Node(key)
    if key < node.key:
        node.left = self.insert(node.left, key)
    else:
        node.right = self.insert(node.right, key)
    return self.balance_node(node)

```

=====

```

def print_tree(self):
    def tree_queue(root):
        if root is None:
            return
        nonlocal index
        root.index_for_print = index
        index += 1

```

```

        tree_queue(root.left)
        tree_queue(root.right)

def print_tree_queue(root):
    if root is None:
        return
    nonlocal Nodes
    Nodes.append(tuple(map(str, (root.key,
                                root.left.output_index if not root.left is None else '0',
                                root.right.output_index if not root.right is None else
'0')))))
    print_tree_queue(root.left)
    print_tree_queue(root.right)

index = 1
Nodes = []
tree_queue(self.root)
print_tree_queue(self.root)
return Nodes

def main():
    with open("input.txt") as f:
        n = int(f.readline())
        Nodes = [None] * (n + 1)
        for i in range(1, n + 1):
            Nodes[i] = tuple(map(int, f.readline().split()))
        x = int(f.readline())

    tree = AVLTree(n, Nodes)

```

```

tree.root = tree.insert(tree.root, x)

with open("output.txt", "w") as f:
    f.write(str(tree.count_of_nodes) + '\n')
    for node in tree.print_tree():
        f.write(' '.join(node) + '\n')

if __name__ == '__main__':
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f"Время: {t_end - t_start} секунд")
    print(f"Память: {tracemalloc.get_traced_memory()}")
    tracemalloc.stop()

```

Текстовое объяснение решения

Аналогично предыдущей задаче реализована класс Node, теми же параметрами.

А также реализована функция вставки новый элементов существующее дерево. Задействуется функция insert, которая рекурсивно проходит по дереву, сравнивая ключи его узлов с ключом нового узла, и тем самым определяет в какое поддерево поместить новый элемент. Как только у текущей вершины нет левого/правого ребенка, то она останавливается и возвращает новый экземпляр класса Node. Текущая вершина будет родителем новой вершины. В дальнейшем производится перебалансировка дерева, которую регулирует функция balance_node. Сначала обновляется высота поддерева с корнем в текущем узле. Затем баланс каждого узла вычисляется как разность высот левого и правого

поддеревьев. Если баланс равен 2 (высота правого поддерева на 2 больше высоты левого поддерева), то выполняется малый левый поворот (функция `rotate_left`). Если же при этом баланс правого узла меньше 0, то производится большой левый поворот, являющийся комбинацией малого правого поворота (функция `rotate_right`) только правого поддерева и последующего левого поворота. Аналогично, когда баланс равен -2, выполняется малый правый поворот. Если же при этом баланс левого узла больше 0, то производится большой правый поворот, являющийся комбинацией малого левого поворота левого поддерева и последующего малого правого поворота.

Результат работы кода на примерах из текста задачи:

input.txt		2-lab\...\output.txt
1	2	1 3
2	3 0 2	2 4 0 0
3	4 0 0	3 3 0 0
4	5	4 5 0 0

Результат работы кода на максимальных и минимальных значениях:

input.txt		2-lab\...\output.txt
1	0	1 1
2	1	2 1 0 0

input.txt		2-lab\...\output.txt
1	200000	330 -37920 0 0
2	250917917 2 12447	331 -34892 0 0
3	-595084365 3 74143	332 -31691 0 0
4	-790671650 4 3269	333 -31624 333 334
5	-965193047 5 5481	334 -32179 0 0
6	-9740465290 6 7794	335 -31575 335
7	-9773653456 7 3375	336 -31250 0 0

	Время выполнения, с	Затраты памяти, КБ
--	---------------------	--------------------

Нижняя граница диапазона значений входных данных из текста задачи	0.000655200099572539 3	18,56
Пример из задачи	0.005266899941489100 5	19,98
Верхняя граница диапазона значений входных данных из текста задачи	3,434986738969847533 6	94423,56

Вывод по задаче: Программа корректно работает на всех приведенных тестах. Однако на максимальных входных данных программа выходит за ограничения по времени. Изучив работу программы я сделал вывод, что на работоспособность сильно влияет чтение входных данных и вывод готовых, что приводит к выходу за границы ограничения по времени

Задача №15. Удаление из AVL-дерева [3 баллов]

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.

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

В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i + 1)$ -ой строке файла ($1 \leq i \leq N$) 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.

В последней строке содержится число X – ключ вершины, которую требуется удалить из дерева. Гарантируется, что такая вершина в дереве существует.

- **Ограничения на входные данные.** $1 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$, $|X| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции удаления. Нумерация вершин может быть произвольной при условии соблюдения формата.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 мб.

• Пример:

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

- Проверить можно по [ссылке](#), OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 4 задача.

Листинг кода

```
import time
import tracemalloc
```



```

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
        self.output_index = 0

class AVLTree():
    def __init__(self, n, Nodes, index=1):
        self.root = self.create_tree(Nodes, index)
        self.number_of_nodes = n

    def create_tree(self, Nodes, index):
        if len(Nodes) == 1:
            return
        if index == 0:
            return
        root = Node(Nodes[index][0])
        root.left = self.create_tree(Nodes, Nodes[index][1])
        root.right = self.create_tree(Nodes, Nodes[index][2])
        self.fix_height(root)
        return root

# =====

```

```
def height(self, node):
```

```
    if node is None:
```

```
        return 0
```

```
    return node.height
```

```
def fix_height(self, node):
```

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

```
#=====
```

```
def rotate_left(self, node):
```

```
    p = node.right
```

```
    node.right = p.left
```

```
    p.left = node
```

```
    return p
```

```
def rotate_right(self, node):
```

```
    q = node.left
```

```
    node.left = q.right
```

```
    q.right = node
```

```
    return q
```

```
#=====
```

```
def get_balance(self, node):
```

```
    return self.height(node.right) - self.height(node.left)
```

```
def balance_node(self, node):
```

```

if self.get_balance(node) == 2:
    if self.get_balance(node.right) < 0:
        node.right = self.rotate_right(node.right)
    return self.rotate_left(node)
if self.get_balance(node) == -2:
    if self.get_balance(node.left) > 0:
        node.left = self.rotate_left(node.left)
    return self.rotate_right(node)
return node

```

#=====

```

def delete(self, root, key, is_delete_key=True):
    if is_delete_key:
        self.number_of_nodes -= 1

    if root is None:
        return root

    if key == root.key:
        if not (root.left or root.right): #если лист
            root = None
            return None

        elif root.right is None:
            temp = root.left
            root = None
            return temp

        elif root.left is None:
            temp = root.right
            root = None
            return temp

```

```

else:
    left_tree = root.left
    while left_tree.right is not None:
        left_tree = left_tree.right
    root.key = left_tree.key

    root.left = self.delete(root.left, left_tree.key, False)

```

```

else:
    if key < root.key:
        root.left = self.delete(root.left, key, False)
    elif key > root.key:
        root.right = self.delete(root.right, key, False)

```

```

self.fix_height(root)
balance = self.get_balance(root)
if balance not in [-1, 0, 1]:
    root = self.balance_node(root)
return root

```

```

#=====

```

```

def print_tree(self):
    def tree_queue(root):
        if root is None:
            return
        nonlocal index
        root.output_index = index
        index += 1
        tree_queue(root.left)

```

```
tree_queue(root.right)
```

```
def print_tree_queue(root):  
    if root is None:  
        return  
    nonlocal Nodes  
    Nodes.append(map(str, (root.key,  
        root.left.output_index if not root.left is None else '0',  
        root.right.output_index if not root.right is None else '0')))  
    print_tree_queue(root.left)  
    print_tree_queue(root.right)  
index = 1  
Nodes = []  
tree_queue(self.root)  
print_tree_queue(self.root)  
return Nodes
```

```
def main():  
    with open("input.txt") as f:  
        n = int(f.readline())  
        Nodes = [None] * (n + 1)  
        for i in range(1, n + 1):  
            Nodes[i] = tuple(map(int, f.readline().split()))  
        x = int(f.readline())
```

```
tree = AVLTree(n, Nodes)  
tree.root = tree.delete(tree.root, x)
```

```

with open("output.txt", "w") as f:
    f.write(str(tree.number_of_nodes) + '\n')
    for node in tree.print_tree():
        f.write(' '.join(node) + '\n')

if __name__ == '__main__':
    tracemalloc.start()
    t_start = time.perf_counter()
    main()
    t_end = time.perf_counter()
    print(f"Время: {t_end - t_start} секунд")
    print(f"Память: {tracemalloc.get_traced_memory()}")
    tracemalloc.stop()

```

Текстовое объяснение решения

Аналогично предыдущей задаче реализована класс Node, теми же параметрами.

Также реализовано удаление необходимой вершины при условии её наличия. Это осуществляется функцией delete, согласно алгоритму из условия задачи. Пока вершина не 52 найдена происходит спуск по левому и правому поддеревьям. Когда вершина найдена, то выполняется один из трёх сценариев. Если вершина является листом, то она просто удаляется. Если у вершины нет левого ребёнка, то вершина заменяется её правым ребёнком. В противном случае происходит поиск самой правой вершины в левом поддереве, она переносится на место удаляемой вершины, и сама удаляется. В любой ситуации после удаления происходит при необходимости перебалансировка дерева, которую регулирует функция balance_node. Сначала обновляется высота поддерева с корнем в текущем узле, затем баланс каждого узла (реализация аналогична прошлой задаче).

Результат работы кода на примерах из текста задачи:

input.txt		2-lab\...\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	

Результат работы кода на максимальных и минимальных значениях:

input.txt		2-lab\...\output.txt
1	1	1 0
2	10 0 0	2
3	10	

input.txt		2-lab\...\output.txt
1	200000	1 199999
2	-190617682 2 66552	2 -190617682 2 66552
3	-637899278 3 21729	3 -637899278 3 21729
4	-781059996 4 14710	4 -781059996 4 14710
5	-843255340 5 12557	5 -843255340 5 12557
6	-845966558 6 5872	6 -845966558 6 5872
7	-830584173 7 3419	7 -830584173 7 3419
8	-848341882 8 1626	8 -848341882 8 1626
9	-858240443 9 647	9 -858240443 9 647
10	-861343620 10 462	10 -861343620 10 462

	Время выполнения, с	Затраты памяти, КБ
Нижняя граница диапазона значений входных данных из текста задачи	0.00038700003642588854	18,73
Пример из задачи	0.008123099920339882	18,85
Верхняя граница диапазона значений входных данных из текста задачи	4,3426254632344434354	95245,43

Вывод по задаче: Программа корректно работает на всех приведенных тестах. Однако на максимальных входных данных программа выходит за ограничения по времени. Изучив работу программы я сделал вывод, что на работоспособность сильно влияет чтение входных данных и вывод готовых, что приводит к выходу за границы ограничения по времени

Вывод

В ходе данной лабораторной работы я научился решать задачи. Написанные программы были протестированы, а также были измерены потребляемый ими объём памяти и время работы. Все программы работают корректно и укладываются в установленные ограничения по времени и памяти на примерах из задач.