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: Жадные алгоритмы. Динамическое программирование

Выполнила:
Олейник П.Д.
К3143

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Оглавление

Задачи

Задача №1. Обход двоичного дерева

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

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

- **Формат ввода: стандартный ввод или 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.** $1 \le n \le 10^5, 0 \le K_i \le 10^9, -1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

- **Формат вывода / выходного файла (output.txt).** Выведите три строки. Первая строка должна содержать ключи узлов при центрированном обходе дерева (in-order). Вторая строка должна содержать ключи узлов при прямом обходе дерева (pre-order). Третья строка должна содержать ключи узлов при обратном обходе дерева (post-order).

- Ограничение по времени. 5 сек.

- Ограничение по памяти. 512 мб.

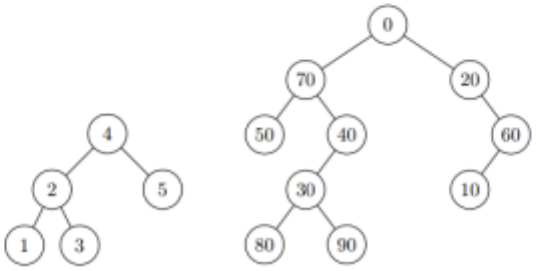
- Примеры:

- Примеры:

input	output.txt
5 4 1 2 2 3 4 5 -1 -1 1 -1 -1 3 -1 -1	1 2 3 4 5 4 2 1 3 5 1 3 2 5 4

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

- Иллюстрации к обоим примерам:



Листинг кода:

```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()
```

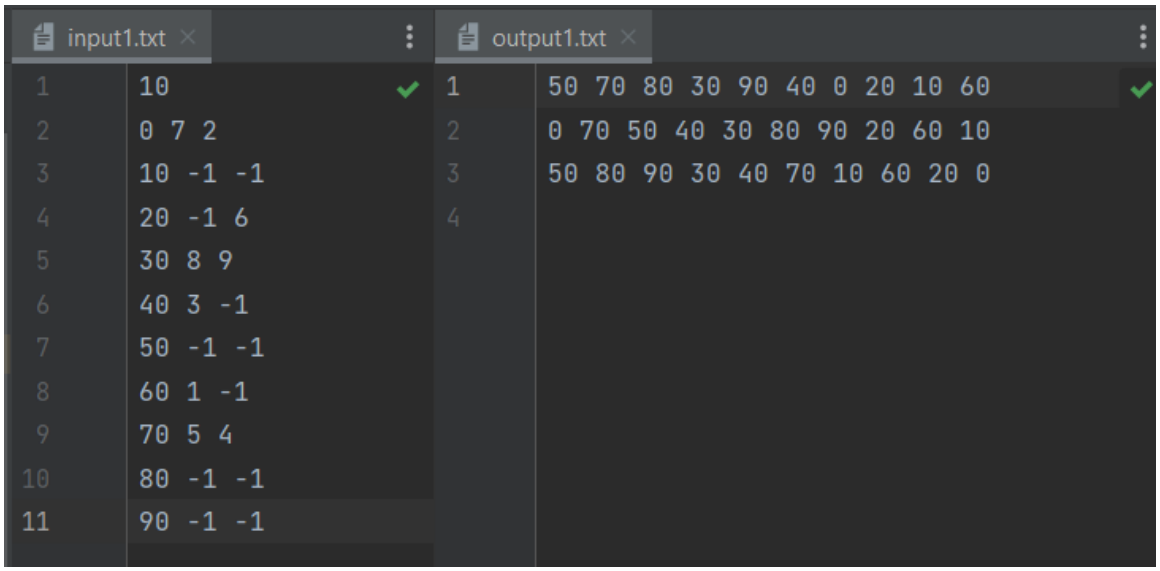
```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None


# центрированный обход
def InOrderTraversal(root):
    if root is None:
        return []
    return [*InOrderTraversal(root.left), root.key,
*InOrderTraversal(root.right)]


# прямой обход
def PreOrderTraversal(root):
    if root is None:
        return []
    return [root.key, *PreOrderTraversal(root.left),
*PreOrderTraversal(root.right)]


# обратный обход
def PostOrderTraversal(root):
    if root is None:
        return []
    return [*PostOrderTraversal(root.left), *PostOrderTraversal(root.right),
root.key]


def solve(n, array):
    dictionary = {}
    for i in range(n):
        dictionary[i] = Node(array[i][0])
    root = dictionary[0]
    for i in range(n):
        node = dictionary[i]
        l_node_index = array[i][1]
        r_node_index = array[i][2]
        if l_node_index != -1:
            node.left = dictionary[l_node_index]
        if r_node_index != -1:
            node.right = dictionary[r_node_index]

    in_order = " ".join([str(i) for i in InOrderTraversal(root)])
    pre_order = " ".join([str(i) for i in PreOrderTraversal(root)])
    post_order = " ".join([str(i) for i in PostOrderTraversal(root)])

    return in_order, pre_order, post_order
```

```
def write_to_file():
    f = open("input1.txt")
    n = int(f.readline())
    array = [[int(i) for i in f.readline().strip().split()] for _ in
range(n)]
    f.close()

    a, b, c = solve(n, array)
    file2 = open("output1.txt", "w+")
    file2.write(a + "\n")
    file2.write(b + "\n")
    file2.write(c + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.

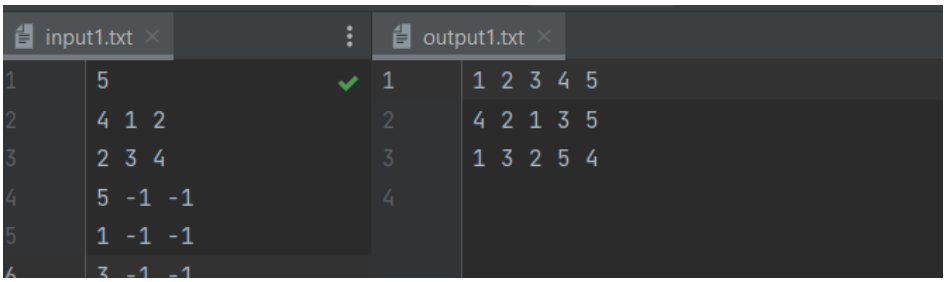
Реализован класс Node(вершина) и алгоритмы обхода в глубину - центрированный обход InOrderTraversal, прямой обход PreOrderTraversal и # обратный обход PostOrderTraversal. В словаре dictionary элемент dictionary[i] это экземпляр класса Node - вершина с индексом i. В двумерной таблице array хранятся входные данные. Проходясь по всем вершинам из dictionary обновляем информацию о левых и правых детях вершин.

Результат работы кода на примерах из текста задачи:



Время выполнения: 0.0080148999986704444 секунд

Затраты памяти: 8.8583984375 КБ



Время выполнения: 0.0070551000009893067 секунд

Затраты памяти: 7.7490234375 КБ

Вывод по задаче: реализованы алгоритмы обхода в глубину.

Задача №3. Простейшее BST

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.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \le x \le 10^9, 1 \le N \le 300000$
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «> x» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
+ 1	3
+ 3	3
+ 3	0
> 1	2
> 2	
> 3	
+ 2	
> 1	

Листинг кода:

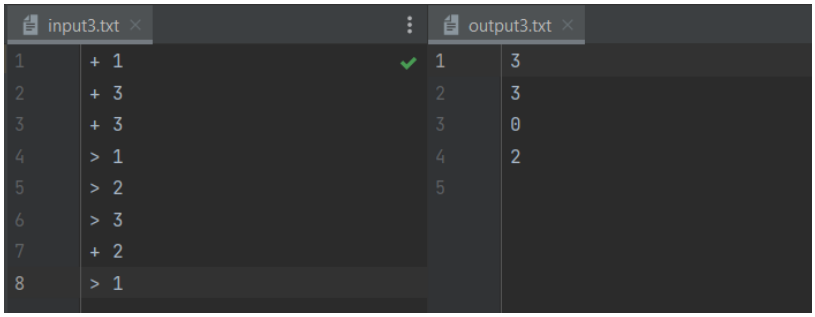
```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()

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


def find(k, root):
    if root is None or root.key == k:
        return root
    elif k < root.key:
        if root.left is not None:
            return find(k, root.left)
        return root
    elif k > root.key:
        if root.right is not None:
            return find(k, root.right)
        return root
```

```
def insert(k, root):
    p = find(k, root)
    if p is not None and p.key != k:
        new_node = Node(k)
        new_node.parent = p
        if k < p.key:
            p.left = new_node
        else:
            p.right = new_node


def tree_min(x):
    while x.left is not None:
        x = x.left
    return x


def tree_max(x):
    while x.right is not None:
        x = x.right
    return x


def next(P):
    if P.right:
        return tree_min(P.right)
    return right_ancestor(P)


def right_ancestor(P):
    if P.key < P.parent.key:
        return P.parent
    return right_ancestor(P.parent)


def find_min_bigger_than_x(x, root):
    biggest = tree_max(root)
    if biggest.key <= x:
        return 0
    N = find(x, root)
    while N.key <= x:
        N = next(N)
    return N.key


def solve(array):
    answer = []
    root = Node(int(array[0].split()[1]))
    for s in array[1:]:
        s = s.split()
        if s[0] == "+":
            insert(int(s[1]), root)
        else:
            answer.append(find_min_bigger_than_x(int(s[1]), root))
```

```
    return answer


def write_to_file():
    f = open("input3.txt")
    array = [s for s in f.readlines()]
    f.close()

    answer = solve(array)

    file2 = open("output3.txt", "w+")
    for el in answer:
        file2.write(str(el) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.
Функция find находит вершину с заданным ключем. Если такой вершины нет, то возвращается вершина, которая может стать ее родителем. Функция insert вставляет новую вершину, зная результат find. tree_max(x) и tree_min(x) находят соответственно вершины с максимальным и минимальным ключом в поддереве с корнем x (те самую правую и самую левую вершину). next(P) находит следующую по возрастанию ключа вершину. Для этого она либо обращается к правому поддереву, либо, если его нет, вызывает функцию right_ancestor(P) которая поднимается вверх по родителям, пока не найдется родитель с ключом большим, чем у ребенка. find_min_bigger_than_x находит минимальное число большее x. Сначала она проверяет, что значение x меньше наибольшего значения в дереве. Далее находит вершину с ключом x и вызывает функцию next.

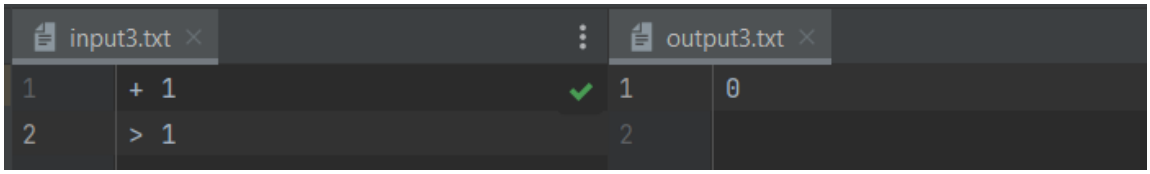
Результат работы кода на примерах из текста задачи:



Время выполнения: 0.0010728000052040443 секунд

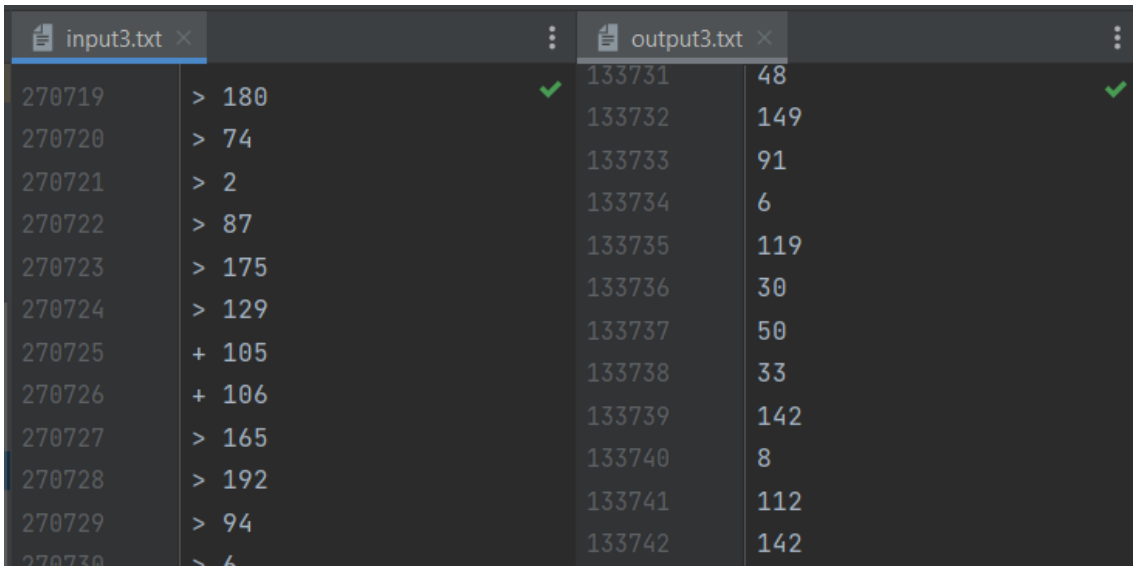
Затраты памяти: 9.2490234375 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.003034400000819005 секунд

Затраты памяти: 7.1943359375 КБ



Время выполнения: 2.015169699995313 секунд

Затраты памяти: 44.24609375 КБ

Вывод: в данной задаче я научилась создавать структуру двоичного дерева и создавать на основе базовых функций новые, необходимые для решения конкретных заданий.

Задача №4. Простейший неявный ключ

В этой задаче вам нужно написать BST по **неявному** ключу и отвечать им на запросы:

- «+ x» – добавить в дерево x (если x уже есть, ничего не делать).
- «? k» – вернуть k-й по возрастанию элемент.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \le x \le 10^9$, $1 \le N \le 300000$, в запросах «? k», число k от 1 до количества элементов в дереве.
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «? k» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
+ 1	1
+ 4	3
+ 3	4
+ 3	3
? 1	
? 2	
? 3	
+ 2	
? 3	

Листинг кода:

```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()

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


def find(k, root):
    if root is None or root.key == k:
        return root
    elif k < root.key:
        if root.left is not None:
            return find(k, root.left)
        return root
    elif k > root.key:
        if root.right is not None:
            return find(k, root.right)
        return root
```

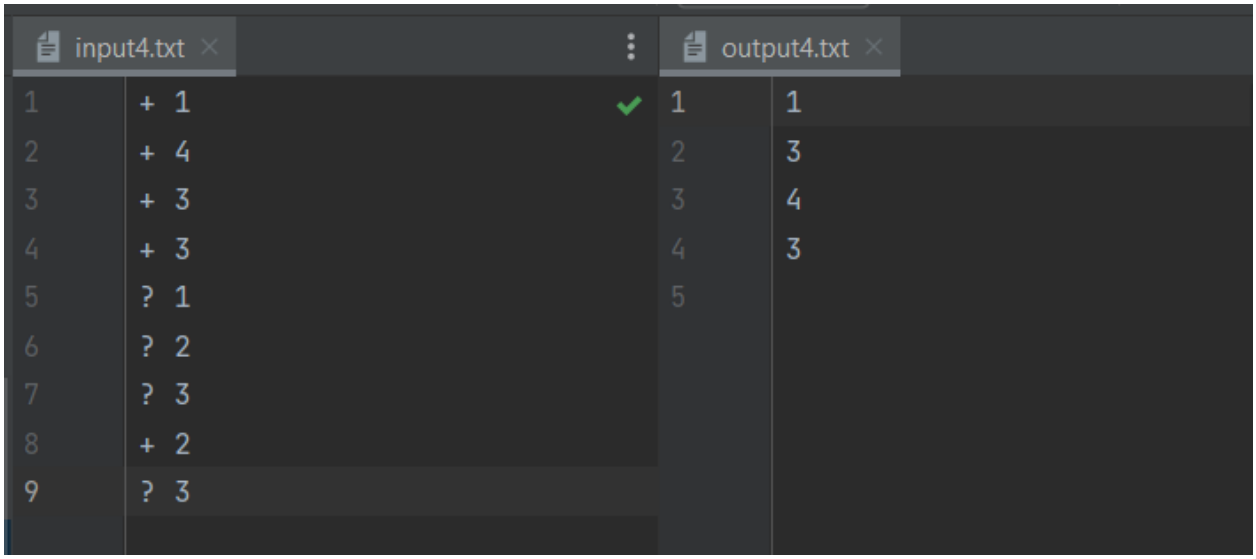
```
def insert(k, root):
    p = find(k, root)
    if p is not None and p.key != k:
        new_node = Node(k)
        new_node.parent = p
        if k < p.key:
            p.left = new_node
        else:
            p.right = new_node


def tree_min(x):
    while x.left is not None:
        x = x.left
    return x


def tree_max(x):
    while x.right is not None:
        x = x.right
    return x


def next(P):
    if P.right:
        return tree_min(P.right)
    return right_ancestor(P)


def right_ancestor(P):
    if P.key < P.parent.key:
        return P.parent
    return right_ancestor(P.parent)


def find_k_min_node(k, root):
    node = tree_min(root)
    for _ in range(k-1):
        node = next(node)
    return node.key


def solve(array):
    answer = []
    root = Node(int(array[0].split()[1]))
    for s in array[1:]:
        s = s.split()
        if s[0] == "+":
            insert(int(s[1]), root)
        else:
            answer.append(find_k_min_node(int(s[1]), root))
    return answer


def write_to_file():
```

```
    f = open("input4.txt")
    array = [s for s in f.readlines()]
    f.close()

    answer = solve(array)

    file2 = open("output4.txt", "w+")
    for el in answer:
        file2.write(str(el) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.

В данной задачи используются те же базовые функции, что и в задаче №3. Появляется функция find_k_min_node, которая сначала находит вершину с наименьшим ключом, а затем вызывает функцию next k-1 раз. Предполагается, что в дереве не менее k вершин.

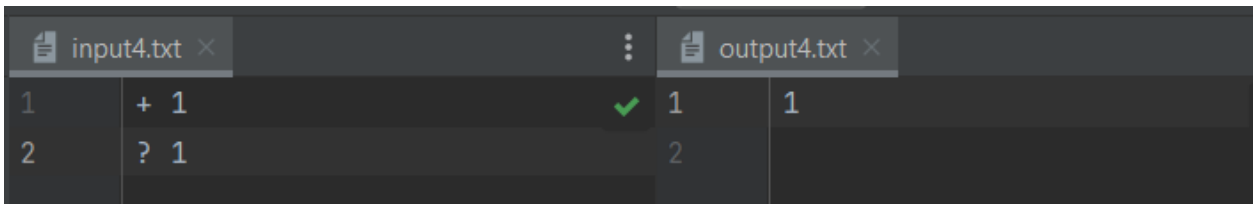
Результат работы кода на примерах из текста задачи:



Время выполнения: 0.005527899993467145 секунд

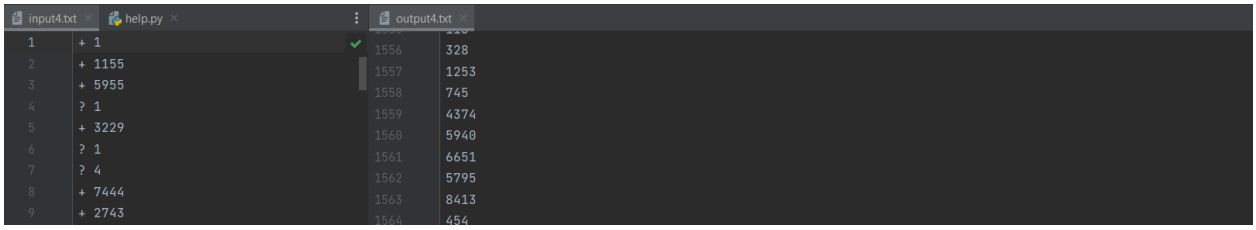
Затраты памяти: 9.4365234375 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.0036159000010229647 секунд

Затраты памяти: 7.1943359375 КБ



Время выполнения: 2.019040200009476 секунд

Затраты памяти: 702.171875 КБ

Вывод: в данной задаче я научилась создавать структуру двоичного дерева и создавать на основе базовых функций новые, необходимые для решения конкретных заданий.

Задача №6. Опознание двоичного дерева поиска

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

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

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

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

- **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.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

  Все ключи во входных данных различны.

- **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.

- Примеры:

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

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

input.txt	output.txt
0	CORRECT

input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT

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

input.txt	output.txt
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

- Примечание. Пустое дерево считается правильным двоичным деревом поиска. Дерево не обязательно должно быть сбалансировано.

Листинг кода:

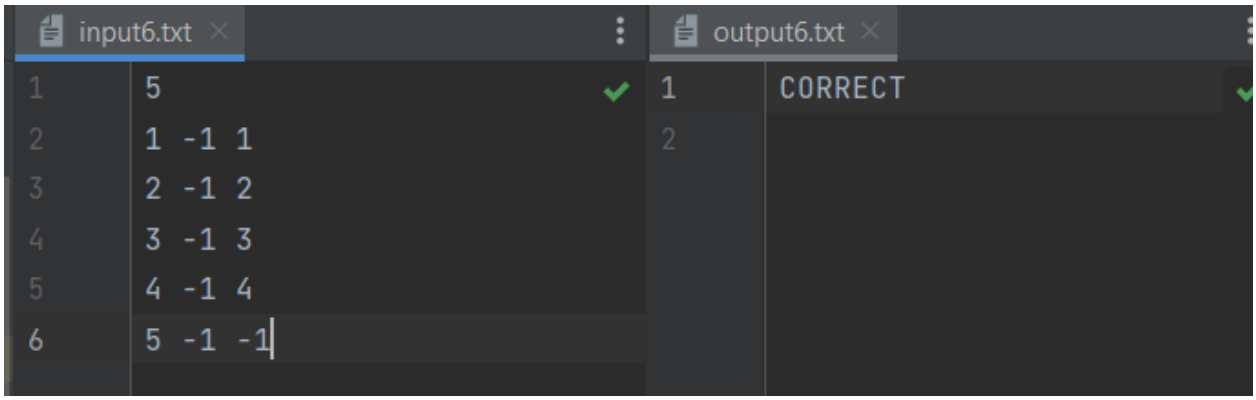
```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()


# центрированный обход
```

```
def InOrderTraversal(id, array):
    if id == -1:
        return []
    left_id = array[id][1]
    right_id = array[id][2]
    key = array[id][0]
    return [*InOrderTraversal(left_id, array), key,
*InOrderTraversal(right_id, array)]


def solve(n, array):
    if not array:
        return "CORRECT"
    values = InOrderTraversal(0, array)
    for i in range(n-1):
        if values[i] >= values[i+1]:
            return "INCORRECT"
    return "CORRECT"


def write_to_file():
    f = open("input6.txt")
    n = int(f.readline())
    array = [[int(i) for i in f.readline().strip().split()] for _ in
range(n)]
    f.close()

    answer = solve(n, array)
    file2 = open("output6.txt", "w+")
    file2.write(answer + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.

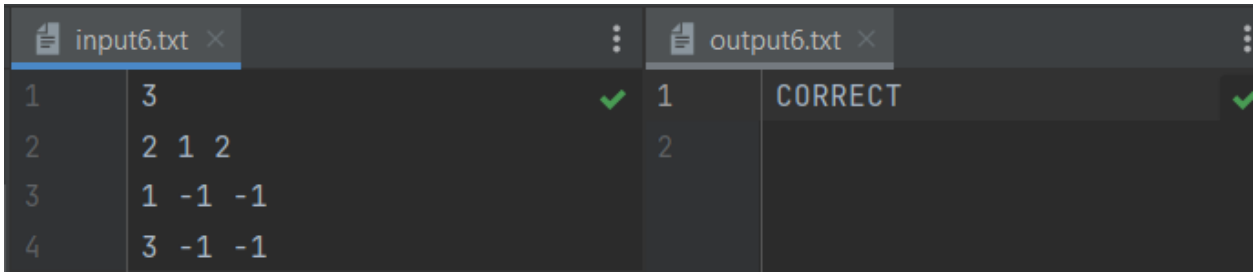
В результате центрированного обхода InOrderTraversal, который выполняется рекурсивно, должен получиться отсортированный список ключей. Функция solve сначала проверяет, пустое ли дерево или нет, далее вызывает InOrderTraversal и проверяет, является ли полученный список отсортированным. Сложность алгоритма линейна по количеству вершин и ребер в дереве.

Результат работы кода на примерах из текста задачи:



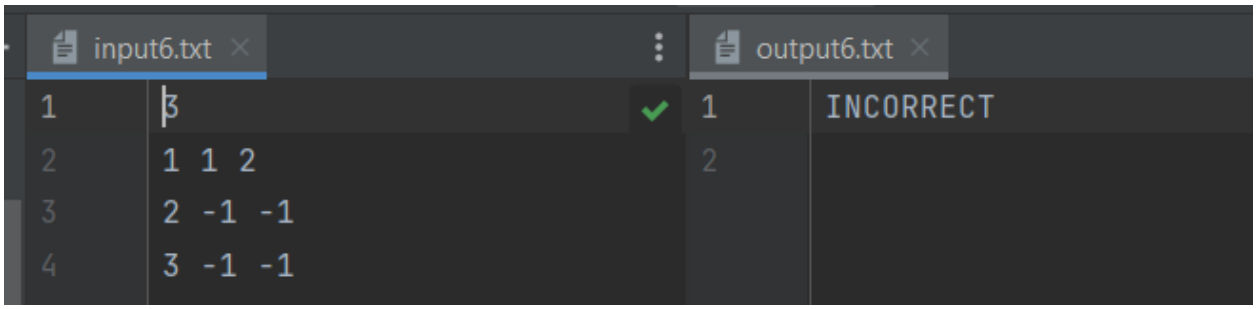
Время выполнения: 0.0070758999936515465 секунд

Затраты памяти: 4.4443359375 КБ



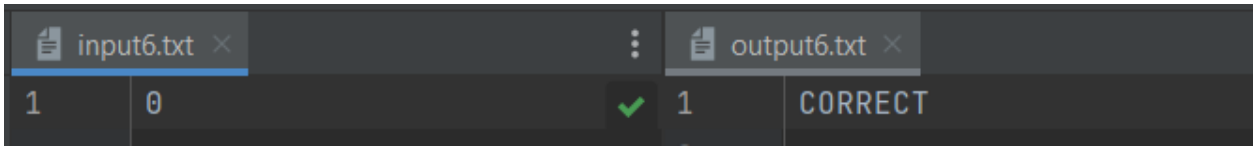
Время выполнения: 0.0038560000102734193 секунд

Затраты памяти: 3.1318359375 КБ



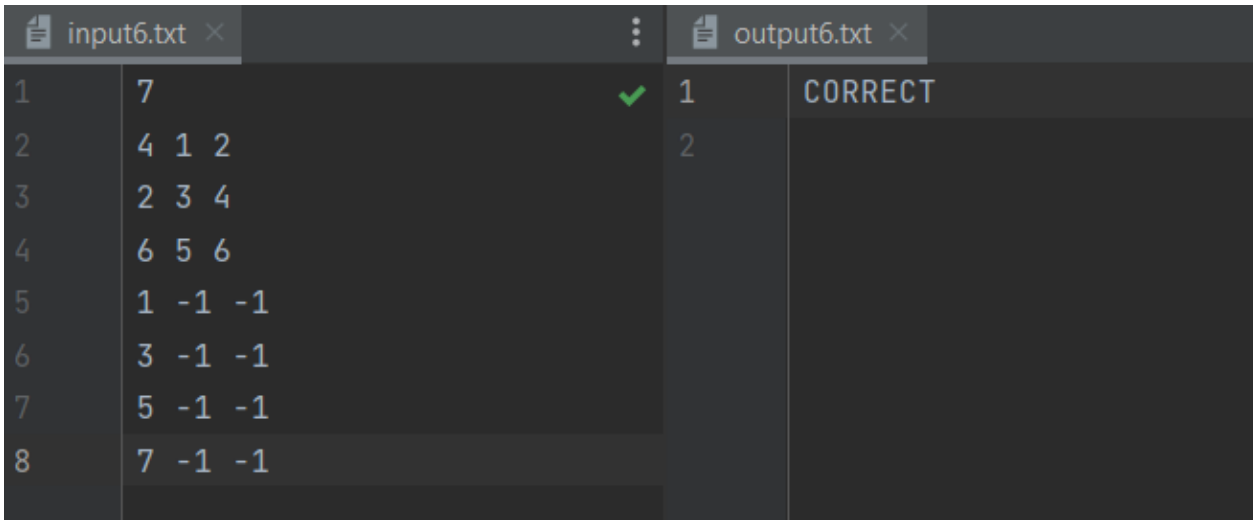
Время выполнения: 0.0007240999984787777 секунд

Затраты памяти: 3.1318359375 КБ



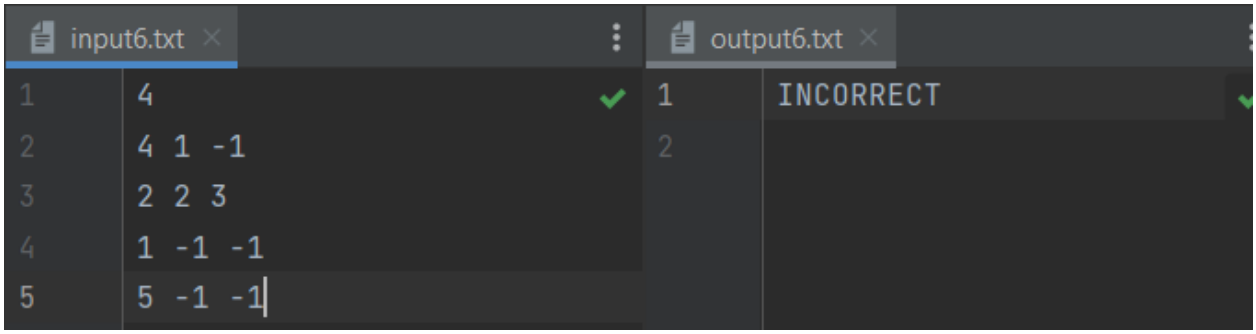
Время выполнения: 0.000632499999483116 секунд

Затраты памяти: 2.6318359375 КБ



Время выполнения: 0.000930599999264814 секунд

Затраты памяти: 3.5693359375 КБ



Время выполнения: 0.004210300001432188 секунд

Затраты памяти: 3.5693359375 КБ

Вывод:

Один из способов проверить, что граф является BST, это обход графа в глубину, который задается рекурсивной функцией.

Задача №7. Опознание двоичного дерева поиска

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.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. 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).

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

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

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

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

input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT

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

input.txt	output.txt
1 2147483647 -1 -1	CORRECT

- 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:

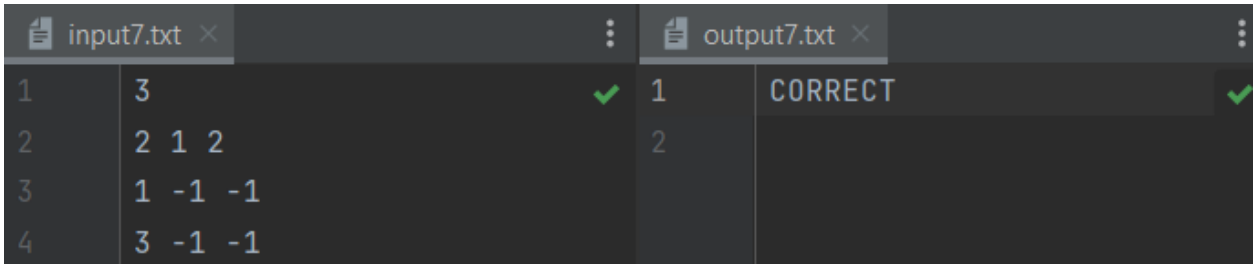
```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()



# центрированный обход
def InOrderTraversal(id, array):
    left_id = array[id][1]
    right_id = array[id][2]
    key = array[id][0]
```

```
    if left_id == -1 and right_id == -1:
        return [array[id][0], True, array[id][0]]
    if left_id == -1:
        return [key, key <= array[right_id][0], InOrderTraversal(right_id,
array)]
    if right_id == -1:
        return [InOrderTraversal(left_id, array), array[left_id][0] < key,
key]
    res1 = InOrderTraversal(left_id, array)
    res2 = InOrderTraversal(right_id, array)
    return [res1[0], (res1[1] and res2[1] and (res1[2] < key <= res2[0])),
res2[2]]


def solve(n, array):
    if not array:
        return "CORRECT"
    res = InOrderTraversal(0, array)
    if res[1]:
        return "CORRECT"
    return "INCORRECT"


def write_to_file():
    f = open("input7.txt")
    n = int(f.readline())
    array = [[int(i) for i in f.readline().strip().split()] for _ in
range(n)]
    f.close()

    answer = solve(n, array)
    file2 = open("output7.txt", "w+")
    file2.write(answer + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.
Как и в задаче №6 применим центрированный обход. На этот раз рекурсивная функция InOrderTraversal(id, array) будет возвращать массив из трех элементов. Нулевой элемент это минимальный ключ в левом поддереве дерева с корнем id. Первый элемент – булево значение. Он будет равен истине, если максимальный элемент левого поддерева строго меньше корня, минимальный элемент правого поддерева нестрого больше корня и если в

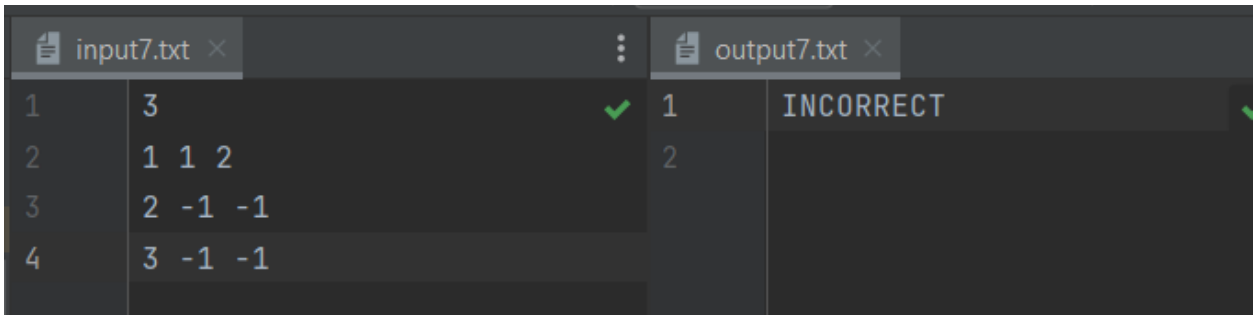
рекурсивная функция от левого и правого поддеревьев так же истинна. Второй элемент – это максимальный ключ в правом поддереве дерева с корнем id. Иначе говоря, первый элемент возвращаемого списка и есть искомый ответ.

Результат работы кода на примерах из текста задачи:



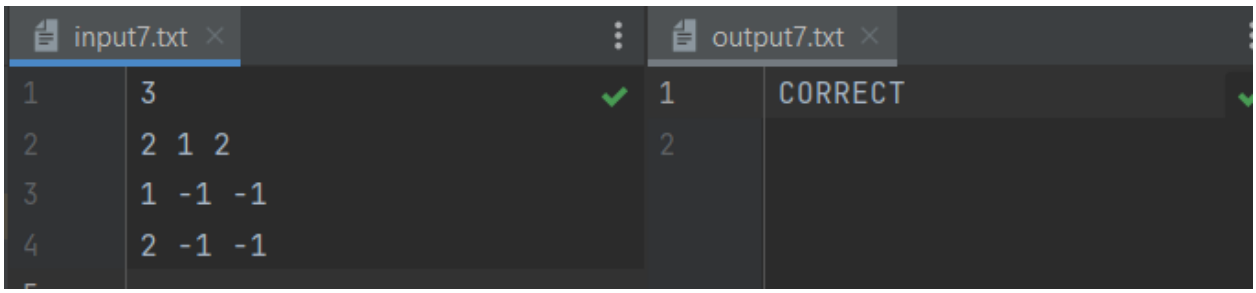
Время выполнения: 0.0071309000049950555 секунд

Затраты памяти: 2.6865234375 КБ



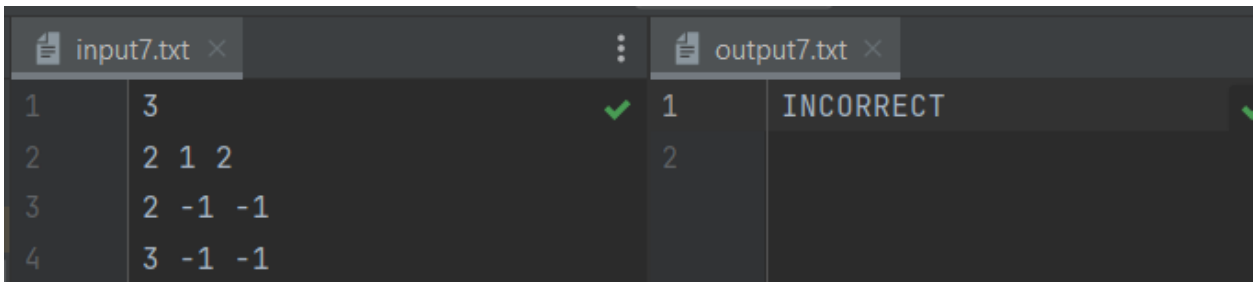
Время выполнения: 0.0070829000032972544 секунд

Затраты памяти: 2.6865234375 КБ



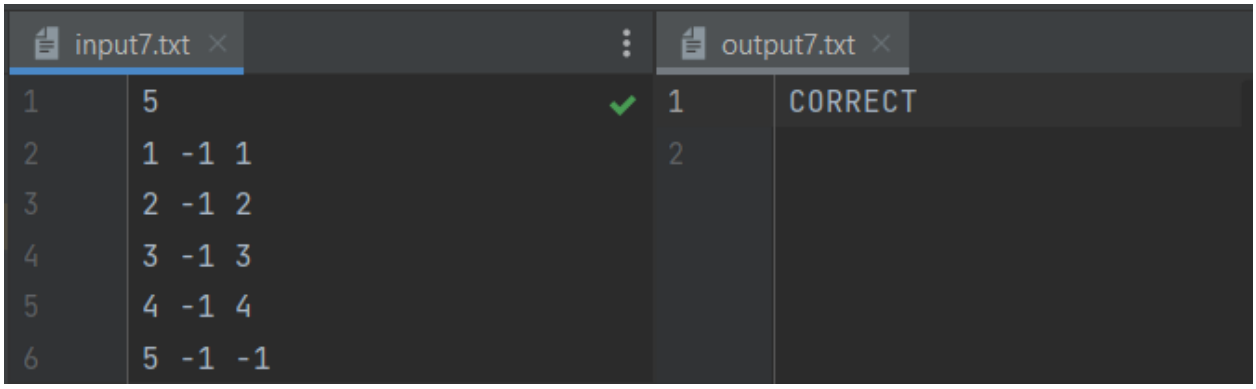
Время выполнения: 0.005825500003993511 секунд

Затраты памяти: 2.6865234375 КБ



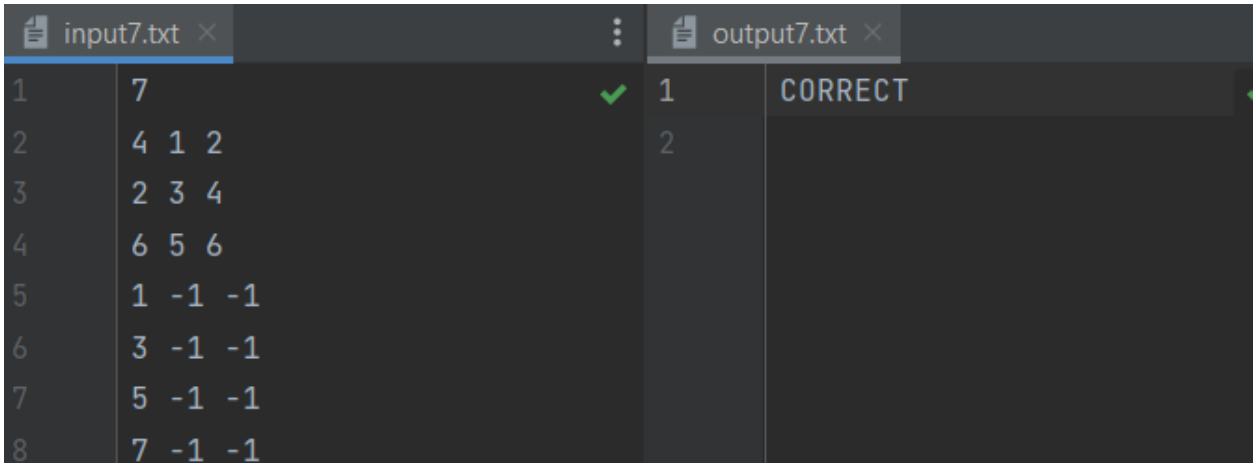
Время выполнения: 0.005648199992720038 секунд

Затраты памяти: 2.6865234375 КБ



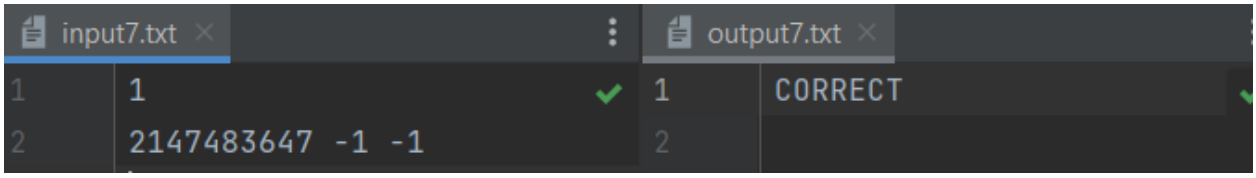
Время выполнения: 0.006782499986002222 секунд

Затраты памяти: 3.6865234375 КБ



Время выполнения: 0.006498500006273389 секунд

Затраты памяти: 2.7646484375 КБ



Время выполнения: 0.006130299996584654 секунд

Затраты памяти: 2.6552734375 КБ

Вывод: данная задача решается с помощью рекурсивной функции (центрированного обхода).

Задача №9. Удаление поддеревьев

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

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

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

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

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

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

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

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

  В следующей строке находится число M – число запросов на удаление. В следующей строке находятся M чисел, разделенных пробелами – ключи, вершины с которыми (вместе с их поддеревьями) необходимо удалить. Все эти числа не превосходят 10^9 по абсолютному значению. Вершина с таким ключом не обязана существовать в дереве – в этом случае дерево изменять не требуется. Гарантируется, что корень дерева никогда не будет удален.

- **Ограничения на входные данные.** $1 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$, $1 \le M \le 2 \cdot 10^5$
- **Формат вывода / выходного файла (output.txt).** Выведите M строк. На i-ой строке требуется вывести число вершин, оставшихся в дереве после выполнения i-го запроса на удаление.

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

Листинг кода:

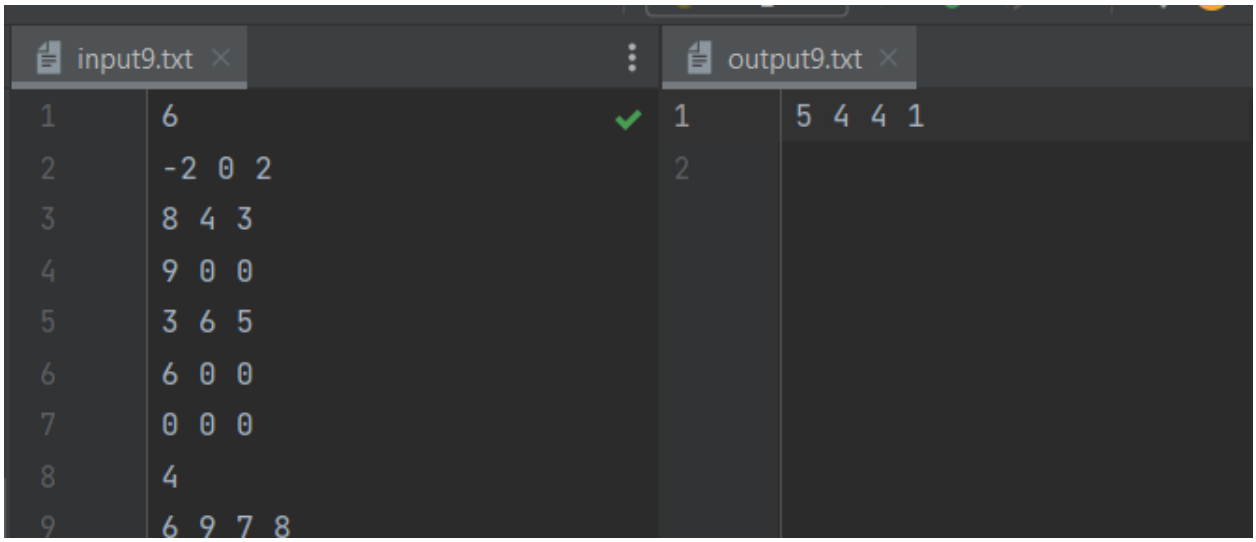
```
import time
import tracemalloc

t_start = time.perf_counter()
tracemalloc.start()


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

```
def find(k, root):
    if root is None or root.key == k:
        return root
    if k < root.key:
        return find(k, root.left)
    if k > root.key:
        return find(k, root.right)


def delete(node):
    if node is not None:
        parent = node.parent
        if parent.key < node.key:
            parent.right = None
        else:
            parent.left = None
        node.parent = None


def count_nodes(node):
    if node is None:
        return 0
    return 1 + count_nodes(node.left) + count_nodes(node.right)


def solve(n, array, m, requests):
    answers = []
    nodes = []
    for i in range(n):
        nodes.append(Node(array[i][0]))
    for i in range(n):
        l_node_index = array[i][1] - 1
        r_node_index = array[i][2] - 1
        if l_node_index != -1:
            nodes[i].left = nodes[l_node_index]
            nodes[l_node_index].parent = nodes[i]
        if r_node_index != -1:
            nodes[i].right = nodes[r_node_index]
            nodes[r_node_index].parent = nodes[i]
    root = nodes[0]
    for k in requests:
        node = find(k, root)
        n -= count_nodes(node)
        delete(node)
        answers.append(n)
    return answers


def write_to_file():
    f = open("input9.txt")
    n = int(f.readline())
    array = [[int(i) for i in f.readline().strip().split()] for _ in
range(n)]
    m = int(f.readline())
    requests = [int(i) for i in f.readline().strip().split()]
```

```
    f.close()

    answers = [str(el) for el in solve(n, array, m, requests)]
    file2 = open("output9.txt", "w+")
    file2.write(" ".join(answers) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.
Функция find находит вершину с заданным ключом. Если такой вершины нет, то возвращается None. Функция delete (node) удаляет у вершины node запись о родителе, а у вершины node.parent – запись о ребенке. Функция count_nodes(node) рекурсивно считает кол-во вершин в поддереве с корнем node. В функции solve сначала создается дерево, а затем для каждого запроса последовательно вызываются функции find, count_nodes, delete. На каждой итерации обновляется переменная n, равная количеству оставшихся в дереве вершин.

Результат работы кода на примерах из текста задачи:



Время выполнения: 0.008966900000814348 секунд

Затраты памяти: 7.7021484375 КБ

Вывод: реализовано удаление поддеревьев.

Задача №16. K-й максимум

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 ki-го мак- симума, 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.

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

t_start = time.perf_counter()
tracemalloc.start()


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


Root = None
```

```
def get_size(root):
    if root is None:
        return 0
    return root.size


def insert(from_root, key):
    global Root

    if Root is None:
        Root = Node(key)
        return Root

    if key < from_root.key:
        if from_root.left is None:
            from_root.left = Node(key)
        else:
            from_root.left = insert(from_root.left, key)

    elif key > from_root.key:
        if from_root.right is None:
            from_root.right = Node(key)
        else:
            from_root.right = insert(from_root.right, key)

    from_root.size = 1 + get_size(from_root.left) + get_size(from_root.right)
    return from_root


def tree_min(x):
    while x.left is not None:
        x = x.left
    return x


def delete_node(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = delete_node(root.left, key)
    elif key > root.key:
        root.right = delete_node(root.right, key)
    else:
        if root.left is None:
            return root.right
        if root.right is None:
            return root.left
        temp = tree_min(root.right)
        root.key = temp.key
        root.right = delete_node(root.right, temp.key)
    root.size = 1 + get_size(root.left) + get_size(root.right)
    return root
```

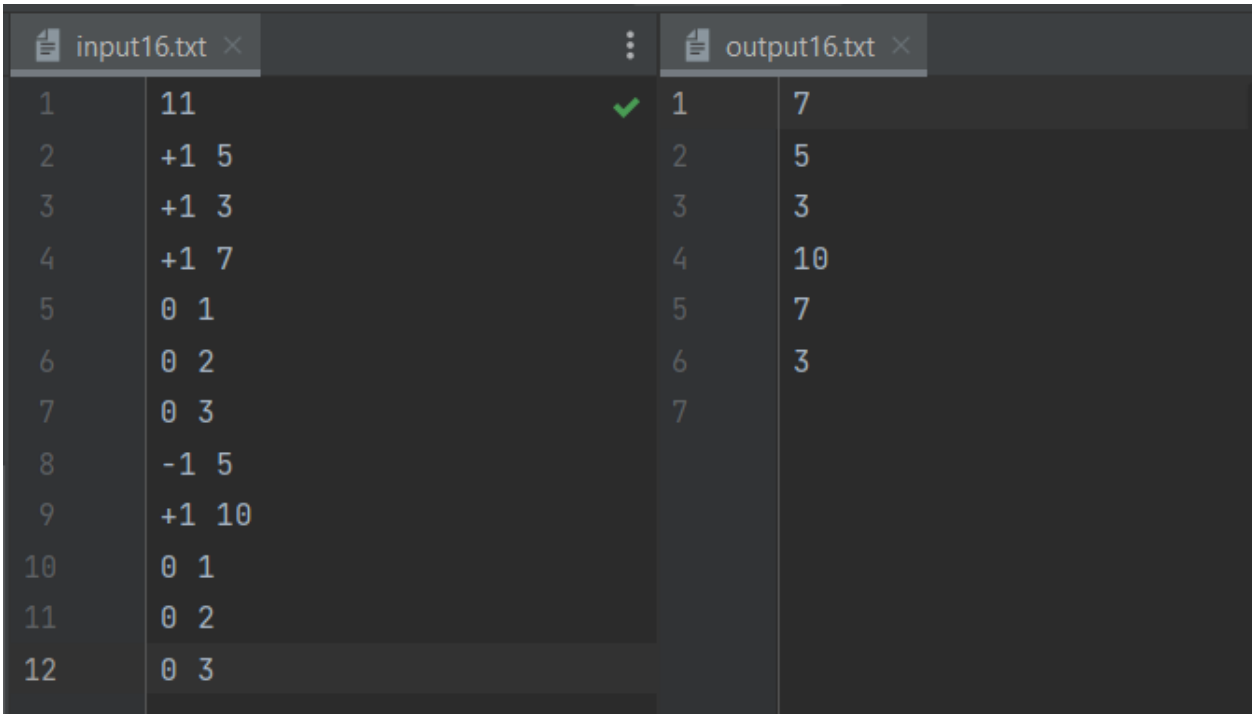
```
def find_k_max(root, k):
    if root is None:
        return None
    right_size = get_size(root.right)
    if right_size + 1 == k:
        return root.key
    elif k <= right_size:
        return find_k_max(root.right, k)
    else:
        return find_k_max(root.left, k - right_size - 1)


def solve(commands):
    global Root
    answers = []
    commands = [line.split() for line in commands]
    for command in commands:
        if command[0] == "+1":
            insert(Root, int(command[1]))
        elif command[0] == "0":
            result = find_k_max(Root, int(command[1]))
            if result:
                answers.append(result)
        elif command[0] == "-1":
            Root = delete_node(Root, int(command[1]))
    return answers


def write_to_file():
    f = open("input16.txt")
    n = int(f.readline())
    commands = [f.readline().strip() for _ in range(n)]
    f.close()

    answers = solve(commands)
    file2 = open("output16.txt", "w+")
    for i in range(len(answers)):
        file2.write(str(answers[i]) + "\n")
    file2.close()


if __name__ == "__main__":
    write_to_file()

    print("Время выполнения: %s секунд " % (time.perf_counter() - t_start))
    print("Затраты памяти: %s КБ " % (tracemalloc.get_traced_memory()[0] / 2
** 10))
```

Текстовое объяснение решения.

В переменной Root хранится ячейка-корень дерева. У каждого экземпляра класса Node появляется новый атрибут size – колво ячеек в поддереве, корнем которого является данный экземпляр. Функция get_size возвращает размер

поддерева, проверяя не является ли корень поддерева None. Функция insert рекурсивно вставляет элемент по ключу. Если Root равен None, то функция прекращает свое действие на первой итерации, иначе запускается рекурсия по поиску места для вставки в поддереве с корнем from_root. Как только такое место найдено, пересчитываются значения size для каждой ячейки из найденного пути. Функция delete_node рекурсивно находит ячейку, которую нужно удалить и перезаписывает size. Функция find_k_max находит к-ый максимум, сравнивая значение k и размер правого и левого поддеревьев.

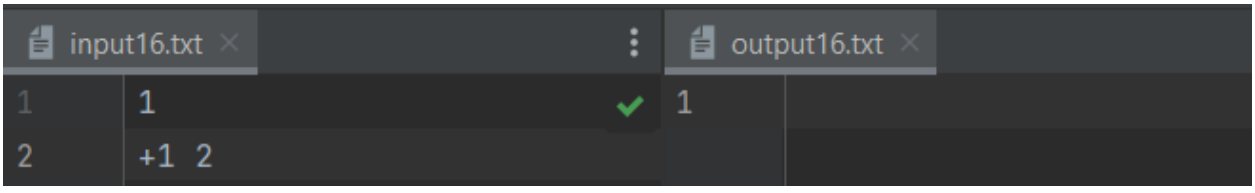
Результат работы кода на примерах из текста задачи:



Время выполнения: 0.011277099998551421 секунд

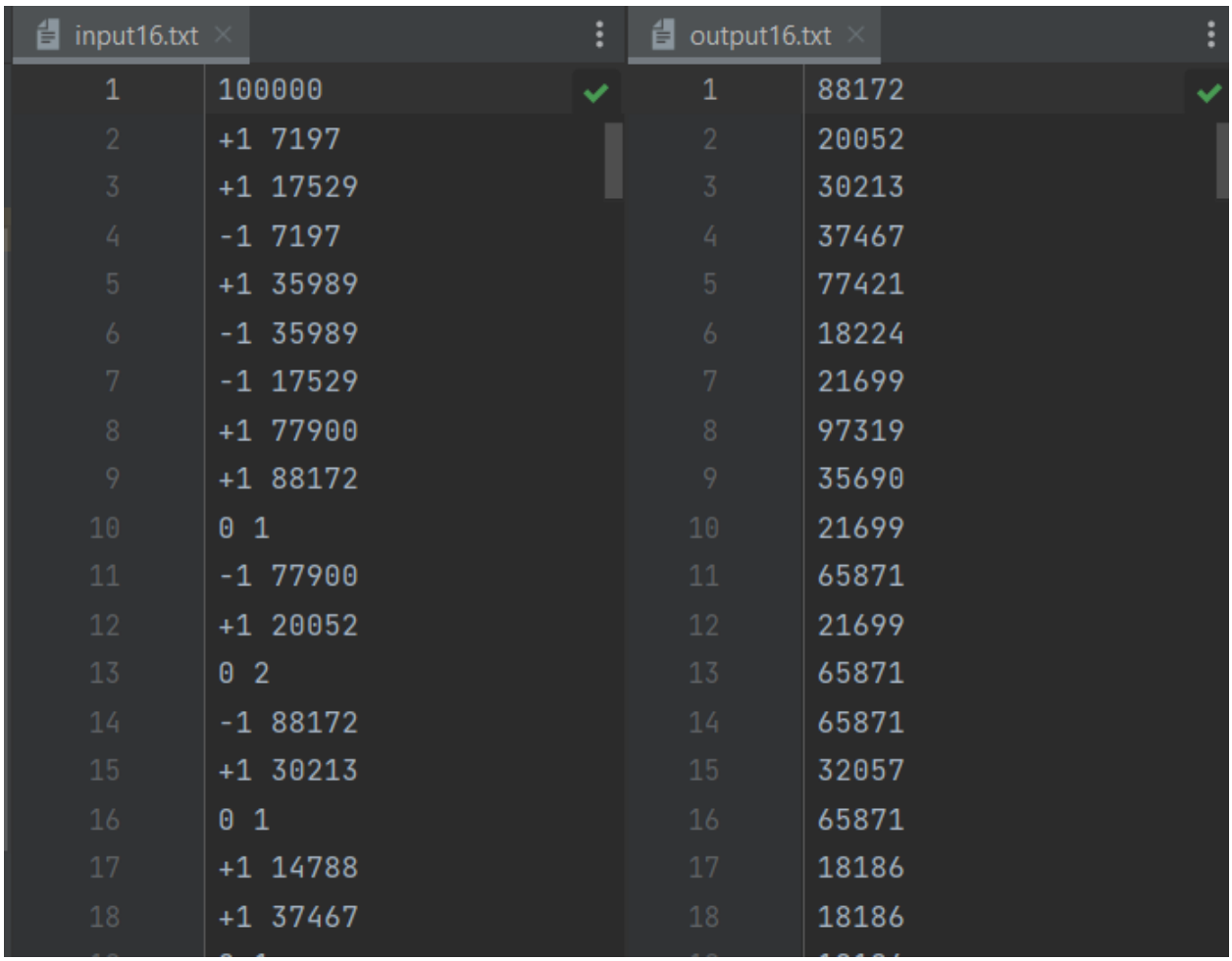
Затраты памяти: 8.2919921875 КБ

Результат работы кода на максимальных и минимальных значениях:



Время выполнения: 0.006697600008919835 секунд

Затраты памяти: 5.5654296875 КБ



input16.txt

```
100000
+1 7197
+1 17529
-1 7197
+1 35989
-1 35989
-1 17529
+1 77900
+1 88172
0 1
-1 77900
+1 20052
0 2
-1 88172
+1 30213
0 1
+1 14788
+1 37467
```

output16.txt

```
88172
20052
30213
37467
77421
18224
21699
97319
35690
21699
65871
21699
65871
65871
32057
65871
18186
18186
```

Время выполнения: 1.4134424999938346 секунд
Затраты памяти: 49.4111328125 КБ

Вывод: реализована структура бинарного дерева с функциями удаления и вставки элемента, а также поиска к-го максимума.

Вывод

В данной лабораторной работе я ознакомилась со структурой бинарного дерева. Написала алгоритмы основных способов обхода бинарного дерева. Построила структуру BST, позволяющую вставлять, удалять элементы, находить k-ый минимальный и максимальный элементы. В бинарном дереве поиск элемента выполняется быстрее, чем в массиве, а вставка и удаление элементов быстрее, чем в отсортированном массиве.