

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 12

Выполнил:
Кузнецов А.Г.
К3140

Проверила:
Артамонова В.Е.

Санкт-Петербург
2023 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №6 Оpozнание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]	3
Задача №11 Сбалансированное двоичное дерево поиска [2 s, 512 Mb, 2 балла]	5
Задача №12 Проверка сбалансированности [2 s, 256 Mb, 2 балла]	10
Дополнительные задачи	13
Задача №1 Обход двоичного дерева [5 s, 512 Mb, 1 балл]	13
Задача №3 Простейшее BST [2 s, 256 Mb, 1 балл]	16
Задача №4 Простейший неявный ключ [2 s, 256 Mb, 1 балл]	18
Задача №5 Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл]	21
Задача №8 Высота дерева возвращается [2 s, 256 Mb, 2 балла]	25
Задача №9 Удаление поддеревьев [2 s, 256 Mb, 2 балла]	28
Задача №10 Проверка корректности [2 s, 256 Mb, 2 балла]	32
Задача №16 К-й максимум [2 s, 512 Mb, 3 балла]	34
Вывод	37

Задачи по варианту

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

В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска. Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет. Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Для каждой вершины дерева V выполняется следующее условие: • все ключи вершин из левого поддеревья меньше ключа вершины V ; • все ключи вершин из правого поддеревья больше ключа вершины V . Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию. Вам гарантируется, что входные данные содержат допустимое двоичное дерево. То есть это дерево, и каждый узел имеет не более двух ребенков.

- 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. $0 \leq n \leq 105$, $-2^{31} \leq K_i \leq 2^{31} - 1$, $-1 \leq L_i, R_i \leq n - 1$. 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. 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.

- Ограничение по памяти. 512 мб.

```
from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.key = key # Значение узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок

# Функция для проверки, является ли дерево деревом поиска
def is_binary_search_tree(root):
    def is_bst_helper(node, low=float('-inf'), high=float('inf')):
        # Если узел пустой, то дерево - бинарное дерево поиска
        if node is None:
            return True
        # Если значение узла находится вне границ, то дерево не является
        # бинарным деревом поиска
        if not low <= node.key <= high:
            return False
        # Рекурсивно проверяем левое и правое поддеревья
        return is_bst_helper(node.left, low, node.key) and
        is_bst_helper(node.right, node.key, high)

    return is_bst_helper(root)

if __name__ == '__main__':
    with open("output.txt", "w+") as f_out, open('input.txt') as f:
        n = int(f.readline())
        if n == 0: # Если узлов нет, то дерево корректно
            f_out.write('CORRECT')
        else:
            nodes = []
            # Считываем данные об узлах и создаем объекты Node
            for i in range(n):
                key, left, right = map(int, f.readline().split())
                nodes.append(Node(key, left, right))
            root = nodes[0] # Первый узел - корневой
            # Обходим узлы дерева и устанавливаем связи между узлами
            for node in nodes:
                if node.left != -1:
                    node.left = nodes[node.left]
                else:
                    node.left = None
                if node.right != -1:
                    node.right = nodes[node.right]
                else:
```

```

        node.right = None
    # Проверяем, является ли дерево деревом поиска
    if is_binary_search_tree(root):
        f_out.write('CORRECT')
    else:
        f_out.write('INCORRECT')
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Данный код представляет реализацию бинарного дерева поиска и содержит методы для вставки, удаления, поиска значения в дереве, а также методы next и prev, которые ищут следующий и предыдущий элемент в дереве соответственно. На вход в файле input.txt ожидаются следующие данные: первая строка содержит одно целое число n ($1 \leq n \leq 105$) - количество запросов; следующие n строк содержат запросы в следующем формате: "+ x" - добавление числа x в дерево (гарантируется, что x отсутствует в дереве); "- x" - удаление числа x из дерева (гарантируется, что x присутствует в дереве); "? x" - проверка наличия числа x в дереве; "next x" - поиск наименьшего числа, большего x в дереве; "prev x" - поиск наибольшего числа, меньшего x в дереве. На выход в файл output.txt будут записаны результаты выполнения запросов.



Вывод по 6 задаче: В ходе работы над шестой задачей был реализован алгоритм, который проверяет правильно ли реализована структура данных бинарного дерева поиска.

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

Реализуйте сбалансированное двоичное дерево поиска.

- Формат ввода / входного файла (input.txt). Входной файл содержит описание операций с деревом, их количество N не превышает 105 . 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 109.

- Ограничения на входные данные. $0 \leq N \leq 105$, $|x_i| \leq 109$.
- 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.

```
from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.key = key # Значение узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок

# Класс бинарного дерева поиска
class BST:
    def __init__(self):
        self.root = None

    # Вставка значения в дерево
    def insert(self, key):
        # Если дерево пустое, создаем корень
        if self.root is None:
            self.root = Node(key)
        # Иначе находим место для вставки нового узла
        else:
            curr = self.root
            while True:
                # Если ключ уже есть в дереве, выходим
                if key == curr.key:
```

```

        return
    elif key < curr.key:
        # Если левый потомок отсутствует, создаем новый узел
        if curr.left is None:
            curr.left = Node(key)
            return
        else:
            # Иначе переходим к левому потомку
            curr = curr.left
    else:
        # Если правый потомок отсутствует, создаем новый узел
        if curr.right is None:
            curr.right = Node(key)
            return
        # Иначе переходим к правому потомку
        curr = curr.right

# Удаления значения из дерева
def delete(self, key):
    # Нахождение минимального узла в дереве
    def find_min_node(node):
        while node.left is not None:
            node = node.left
        return node

    def delete_helper(node, key):
        if node is None:
            return node
        if key < node.key:
            node.left = delete_helper(node.left, key)
        elif key > node.key:
            node.right = delete_helper(node.right, key)
        else:
            # Если удаляемый узел является листовым узлом
            if node.left is None and node.right is None:
                node = None
            # Если у удаляемого узла нет левого поддерева
            elif node.left is None:
                node = node.right
            # Если у удаляемого узла нет правого поддерева
            elif node.right is None:
                node = node.left
            # Если у удаляемого узла есть оба поддерева
            else:
                # Находим минимальный узел в правом поддереве
                min_node = find_min_node(node.right)
                # Заменяем ключ удаляемого узла на ключ минимального
                узла
                node.key = min_node.key
                # Рекурсивно удаляем минимальный узел из правого
                поддерева
                node.right = delete_helper(node.right, min_node.key)
        return node

    # Удаление значения с корневого узла
    self.root = delete_helper(self.root, key)

```

```

# Поиск значения в дереве
def exists(self, key):
    curr = self.root
    while curr is not None:
        # Если ключ найден, возвращаем True
        if key == curr.key:
            return True

        # Если ключ меньше текущего, идем налево
        elif key < curr.key:
            curr = curr.left

        # Иначе идем направо
        else:
            curr = curr.right

    # Если ключ не найден, возвращаем False
    return False

def next(self, key):
    curr = self.root
    next_key = None
    while curr is not None:
        # Если ключ текущей ноды больше искомого
        if curr.key > key:
            # Если следующий ключ еще не найден, или найденный ключ
            # больше текущего
            if next_key is None or curr.key < next_key:
                # Запоминаем текущий ключ как следующий
                next_key = curr.key
            # Идем налево
            curr = curr.left
        else:
            # Иначе идем направо
            curr = curr.right

    # Возвращаем следующий ключ или "none", если его нет
    return str(next_key) if next_key is not None else "none"

def prev(self, key):
    curr = self.root
    prev_key = None
    while curr is not None:
        # Если ключ текущей ноды меньше искомого
        if curr.key < key:
            # Если предыдущий ключ еще не найден, или найденный ключ
            # меньше текущего
            if prev_key is None or curr.key > prev_key:
                # Запоминаем текущий ключ как предыдущий
                prev_key = curr.key
            # Идем направо
            curr = curr.right
        else:
            # Иначе идем налево
            curr = curr.left

```



```

        # Возвращаем предыдущий ключ или "none", если его нет
        return str(prev_key) if prev_key is not None else "none"

bst = BST()
with open("input.txt", "r") as f:
    n = 0
    for line in f:
        n += 1
f.close()
with open("output.txt", "w+") as f_out, open('input.txt') as f:
    for i in range(n):
        line = f.readline().strip().split()
        # Обработка команды insert
        if line[0] == "insert":
            bst.insert(int(line[1]))
        # Обработка команды delete
        elif line[0] == "delete":
            bst.delete(int(line[1]))
        # Обработка команды exists
        elif line[0] == "exists":
            # Запись результата поиска в файл вывода
            f_out.write(str(bst.exists(int(line[1]))) + '\n')
        # Обработка команды next
        elif line[0] == "next":
            # Запись результата поиска следующего элемента в файл вывода
            f_out.write(str(bst.next(int(line[1]))) + '\n')
        # Обработка команды prev
        elif line[0] == "prev":
            # Запись результата поиска предыдущего элемента в файл вывода
            f_out.write(str(bst.prev(int(line[1]))) + '\n')
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Данный код реализует бинарное дерево поиска. Входные данные считываются из файла `input.txt`, который содержит последовательность операций с деревом. Каждая операция находится на новой строке и задается в виде двух чисел и символа, разделенных пробелами: первое число - ключ, над которым выполняется операция, второе число - необходимый аргумент (для операций 'insert', 'delete', 'exists' он не требуется, для 'next' и 'prev' - ключ, относительно которого нужно найти следующий или предыдущий). Символ определяет выполняемую операцию: 'insert', 'delete', 'exists', 'next', 'prev'. Результаты выполнения операций записываются в файл `output.txt`. Каждый результат находится на новой строке. Если результатом операции является ключ, то он записывается на соответствующей строке файла `output.txt`. Если результатом является строка "next none" или "prev none", то соответствующая строка в файле `output.txt` содержит только слово "none".

```
Задача 11 (2.6) [упр.]
1 insert 2
2 insert 5
3 insert 3
4 exists 2
5 exists 4
6 next 4
7 prev 4
8 delete 5
9 next 4
10 prev 4

output.txt
1 True
2 False
3 5
4 3
5 none
6 3
```

Вывод по 11 задаче: В ходе работы над одиннадцатой задачей было реализовано сбалансированное двоичное дерево поиска.

Задача №12 Проверка сбалансированности [2 s, 256 Mb, 2 балла]

АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева отличается от высоты ее правого поддерева не больше, чем на единицу.

Введем понятие баланса вершины: для вершины дерева V ее баланс $B(V)$ равен разности высоты правого поддерева и высоты левого поддерева. Таким образом, свойство АВЛ-дерева, приведенное выше, можно сформулировать следующим образом: для любой ее вершины V выполняется следующее неравенство:

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

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой недели «зеркально отражено» по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит Вам неудобств. В литературе по алгоритмам – как российской, так и мировой – ситуация, как правило, примерно та же.

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

- 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

```
from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.key = key # Значение узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок
        self.height = 1 # Высота узла

    def balance(self):
        # Вычисляем баланс узла (разницу между высотами правого и левого
        поддеревьев)
        right_height = self.right.height if self.right else 0
        left_height = self.left.height if self.left else 0
        return right_height - left_height

def read_tree_from_input_file():
    with open('input.txt', 'r') as f:
        n = int(f.readline())
        # Создаем словарь узлов дерева, где ключ - номер узла, значение -
        объект Node
        nodes = {i + 1: Node(None) for i in range(n)}
        # Заполняем поля узлов
        for i in range(n):
            # Считываем значения ключа и потомков
            key, left, right = map(int, f.readline().split())
            # Получаем узел по его номеру
            node = nodes[i + 1]
            node.key = key
            if left:
                node.left = nodes[left]
            if right:
                node.right = nodes[right]
        # Возвращаем корневой узел дерева
        return nodes[1]
```

```

# Функция для обновления высоты узла
def update_height(node):
    # Обновляем высоты всех узлов в дереве (построчный обход дерева)
    left_height = node.left.height if node.left else 0
    right_height = node.right.height if node.right else 0
    node.height = max(left_height, right_height) + 1

# Функция для обновления высот всех узлов в дереве
def update_heights(node):
    # Записываем балансы узлов дерева в файл
    if not node:
        return
    update_heights(node.left) # Обновляем высоты у левого поддерева
    update_heights(node.right) # Обновляем высоты у правого поддерева
    update_height(node) # Обновляем высоту узла

# Функция для записи балансов всех узлов в выходной файл
def write_balances_to_output_file(root):
    with open('output.txt', 'w') as f:
        nodes = [root]
        while nodes:
            node = nodes.pop(0)
            # Записываем баланс текущего узла в файл
            f.write(str(node.balance()) + '\n')
            if node.left:
                # Добавляем левого потомка в список узлов для обхода
                nodes.append(node.left)
            if node.right:
                # Добавляем правого потомка в список узлов для обхода
                nodes.append(node.right)

root = read_tree_from_input_file()
update_heights(root)
write_balances_to_output_file(root)
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Данный код считывает из файла input.txt дерево, обновляет высоты узлов и записывает балансы узлов в файл output.txt. Входные данные в файле input.txt должны быть представлены в следующем формате: первая строка содержит количество узлов дерева, далее идут n строк, каждая из которых содержит информацию об узле в формате "key left right", где key - значение узла, left и right - номера левого и правого потомков соответственно (если потомка нет, то указывается 0). На выходе записывается в файл output.txt баланс каждого узла дерева.

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

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

Вывод по 12 задаче: В ходе работы над двенадцатой задачей был реализован алгоритм, определяющий баланс каждой вершины двоичного дерева поиска.

Дополнительные задачи

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

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

Вам дано корневое двоичное дерево. Выведите центрированный (inorder), прямой (pre-order) и обратный (postorder) 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. $1 \leq n \leq 105$, $0 \leq K_i \leq 109$, $-1 \leq L_i$, $R_i \leq n-1$. 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.

- 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.

```
# класс Node для представления узлов дерева
class Node:
    def __init__(self, key, left_child, right_child):
        self.key = key # Ключ узла
        self.left_child = left_child # Левый потомок
        self.right_child = right_child # Правый потомок

# класс Tree для представления дерева
class Tree:
    def __init__(self, nodes):
        self.nodes = nodes # Массив узлов дерева

    # Центрированный обход в глубину
    # node_index - индекс текущего узла, traversal_list - список для
    # сохранения значений ключей
    def inorder_traversal(self, node_index, traversal_list):
        if node_index == -1:
            return
        node = self.nodes[node_index]
        # Обходим левое поддерево
        self.inorder_traversal(node.left_child, traversal_list)
        # Добавляем ключ текущего узла
        traversal_list.append(node.key)
        # Обходим правое поддерево
        self.inorder_traversal(node.right_child, traversal_list)

    # Прямой обход в глубину
    # node_index - индекс текущего узла, traversal_list - список для
    # сохранения значений ключей
    def preorder_traversal(self, node_index, traversal_list):
        if node_index == -1:
            return
        node = self.nodes[node_index]
        # добавляем ключ текущего узла
        traversal_list.append(node.key)
        # обходим левое поддерево
        self.preorder_traversal(node.left_child, traversal_list)
        # обходим правое поддерево
        self.preorder_traversal(node.right_child, traversal_list)

    # Обратный обход в глубину
    # node_index - индекс текущего узла, traversal_list - список для
    # сохранения значений ключей
    def postorder_traversal(self, node_index, traversal_list):
        if node_index == -1:
            return
        node = self.nodes[node_index]
        # обходим левое поддерево
        self.postorder_traversal(node.left_child, traversal_list)
```

```

        # обходим правое поддереву
        self.postorder_traversal(node.right_child, traversal_list)
        # добавляем ключ текущего узла
        traversal_list.append(node.key)

with open("input.txt") as f:
    n = int(f.readline()) # Количество узлов
    nodes = [] # Список узлов дерева
    for i in range(n):
        key, left_child, right_child = map(int, f.readline().split())
        nodes.append(Node(key, left_child, right_child)) # создаем узел с
заданным ключом и потомками
tree = Tree(nodes) # создаем объект дерева
inorder_list = []

# обходим дерево в порядке in-order, сохраняем значения ключей в список
tree.inorder_traversal(0, inorder_list)
with open("output.txt", 'w+') as f:
    # записываем значения ключей в порядке in-order в файл
    f.write(' '.join(map(str, inorder_list)) + '\n')
    preorder_list = []

    # обходим дерево в порядке pre-order, сохраняем значения ключей в
список
    tree.preorder_traversal(0, preorder_list)
    # записываем значения ключей в порядке pre-order в файл
    f.write(' '.join(map(str, preorder_list)) + '\n')
    postorder_list = []

    # обходим дерево в порядке post-order, сохраняем значения ключей в
список
    tree.postorder_traversal(0, postorder_list)
    # записываем значения ключей в порядке post-order в файл
    f.write(' '.join(map(str, postorder_list)) + '\n')

```

Данные на вход подаются из файла input.txt и содержат количество узлов дерева, а также ключи, левые и правые потомки каждого узла. Код строит бинарное дерево из узлов, обходит его в порядке in-order, pre-order и post-order, сохраняя значения ключей в соответствующие списки. Затем значения ключей каждого обхода записываются в файл output.txt в порядке in-order, pre-order и post-order соответственно.

The screenshot shows a code editor with two files: input.txt and output.txt. The input.txt file contains the following data:

Node Index	Key	Left Child	Right Child
1	5		
2	4	1	2
3	2	3	4
4	5	-1	-1
5	1	-1	-1
6	3	-1	-1

The output.txt file shows the following traversals:

Traversal Type	Sequence of Keys
In-order	1 2 3 4 5
Pre-order	4 2 1 3 5
Post-order	1 3 2 5 4

Вывод по 1 задаче: В ходе работы над первой задачей был реализован алгоритм, который выводит центрированный, прямой и обратный обходы в глубину.

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

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

«+ x» – добавить в дерево x (если x уже есть, ничего не делать).

«> x» – вернуть минимальный элемент больше x или 0, если таких нет.

- 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. $1 \leq x \leq 10^9$, $1 \leq N \leq 300000$

- 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.

```
from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок
        self.key = key # Значение узла
        self.size = 1 # Размер поддерева с корнем в текущем узле

# Класс бинарного дерева поиска
class BST:
    def __init__(self):
        # Инициализируем пустым корнем
        self.root = None

    def insert(self, node, key):
        # Если достигли конца дерева, создаем новый узел и возвращаем его
        if node is None:
```



```

        return Node(key)

    # Если значение меньше текущего узла, рекурсивно добавляем новый
    # узел в левое поддерево
    if key < node.key:
        node.left = self._insert(node.left, key)

    # Если значение больше текущего узла, рекурсивно добавляем новый
    # узел в правое поддерево
    elif key > node.key:
        node.right = self._insert(node.right, key)

    # Обновляем размер поддерева с корнем в текущем узле
    node.size = 1 + self.size(node.left) + self.size(node.right)
    return node

# Добавление узлов в дерево
def insert(self, key):
    self.root = self._insert(self.root, key)

# Определение размера дерева
def size(self, node):
    return node.size if node else 0

def _find_min_greater_than(self, node, key):
    # Если достигли конца дерева, возвращаем 0
    if node is None:
        return 0

    # Если текущий узел имеет значение меньше или равно заданному,
    # рекурсивно идем вправо, а иначе влево
    if node.key <= key:
        return self._find_min_greater_than(node.right, key)
    else:
        # Если в левом поддерева есть хотя бы (key - 1) элементов,
        # рекурсивно идем влево
        if node.left and node.left.size >= key - 1:
            return self._find_min_greater_than(node.left, key)
        return node.key

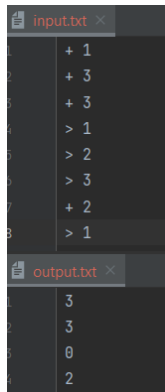
# Поиск узла с наименьшим значением, большим заданного ключа
def find_min_greater_than(self, key):
    return self._find_min_greater_than(self.root, key)

bst = BST()
with open('input.txt') as f_in, open('output.txt', 'w') as f_out:
    for line in f_in:
        cmd, arg = line.split() # Запрос
        if cmd == '+':
            bst.insert(int(arg))
        elif cmd == '>':
            f_out.write(str(bst.find_min_greater_than(int(arg))) + '\n')
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]

```

```
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
```

Данные на вход принимаются из файла 'input.txt', каждая строка содержит команду и ее аргумент, которые передаются в методы класса BST для добавления узла или поиска узла с наименьшим значением, большим заданного ключа. Результаты поиска записываются в файл 'output.txt'. Код создает классы Node и BST для представления бинарного дерева поиска, который может хранить числовые значения.



Вывод по 3 задаче: В ходе работы над третьей задачей было реализовано BST по явному ключу, которое отвечает на определенные запросы.

Задача №4 Простейший неявный ключ [2 s, 256 Mb, 1 балл]

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

«+ x» – добавить в дерево x (если x уже есть, ничего не делать).

«? k» – вернуть k-й по возрастанию элемент.

- 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. $1 \leq x \leq 109$, $1 \leq N \leq 300000$, 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

```
from time import perf_counter
import tracemalloc
```

```

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.key = key # Значение узла
        self.size = 1 # Размер поддерева
        self.left = None # Левый потомок
        self.right = None # Правый потомок

# Класс неявного двоичного дерева поиска
class ImplicitBST:
    def __init__(self):
        # Инициализируем пустым корнем
        self.root = None

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

    # Обновление размера поддерева
    def update_size(self, node):
        node.size = self.size(node.left) + self.size(node.right) + 1

    # Вставка нового узла в дерево
    def insert(self, key):
        def _insert(node, key):
            # Если дерево пустое, то создать узел с заданным ключом
            if node is None:
                return Node(key)

            # Если узел с заданным ключом уже существует, то ничего не
            # делать
            if key == node.key:
                return node

            # Если ключ меньше, то добавить в левое поддерево
            if key < node.key:
                node.left = _insert(node.left, key)

            # Если ключ больше, то добавить в правое поддерево
            else:
                node.right = _insert(node.right, key)

            # Обновление размера поддерева
            self.update_size(node)
            return node

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

```

```

# Получение k-го элемента по возрастанию в дереве
def kth_smallest(self, k):
    def _kth_smallest(node, k):
        # Если дерево пустое, то вернуть None
        if node is None:
            return None
        left_size = self.size(node.left)

        # Если размер левого поддерева равен k-1, то вернуть ключ
        # текущего узла
        if left_size == k - 1:
            return node.key

        # Если размер левого поддерева больше или равен k, то искать k-
        # й элемент в левом поддерева
        if left_size >= k:
            return _kth_smallest(node.left, k)

        # Иначе искать (k-left_size-1)-й элемент в правом поддерева
        else:
            return _kth_smallest(node.right, k - left_size - 1)

    return _kth_smallest(self.root, k)

# Создание экземпляра неявного двоичного дерева поиска
bst = ImplicitBST()
with open('input.txt') as f_in, open('output.txt', 'w') as f_out:
    for line in f_in:
        op, val = line.split() # Запрос
        if op == '+':
            bst.insert(int(val))
        elif op == '?':
            k = int(val)
            f_out.write(str(bst.kth_smallest(k)) + '\n')
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Этот код создает неявное двоичное дерево поиска (Implicit Binary Search Tree) и выполняет операции вставки и поиска k-го элемента по возрастанию в этом дереве. Данные на вход подаются из файла 'input.txt', в котором каждая строка содержит запрос в формате операции и значения, разделенные пробелом. Операции могут быть двух типов: '+' - вставка значения в дерево и '?' - поиск k-го элемента в дереве. Результаты операций записываются в файл 'output.txt'.

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

Вывод по 4 задаче: В ходе работы над четвертой задачей было реализовано BST по неявному ключу, которое отвечает на определенные запросы.

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

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 109 .

- Ограничения на входные данные. $0 \leq N \leq 100$, $|x_i| \leq 109$.
- 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.

```
# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
```

```

        self.key = key # Значение узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок

# Класс бинарного дерева поиска
class BST:
    def __init__(self):
        # Инициализируем пустым корнем
        self.root = None

    # Вставка узла в дерево
    def insert(self, key):
        # Проверяем, если дерево пустое, то создаем корневой узел
        if self.root is None:
            self.root = Node(key)
        else:
            curr = self.root
            # Идем по дереву вниз, пока не найдем подходящее место для
            # вставки нового узла
            while True:
                # Если ключ уже есть в дереве, то вставка не производится
                if key == curr.key:
                    return

                # Если новый ключ меньше текущего, то идем налево
                elif key < curr.key:
                    # Если левая ветвь пуста, создаем новый узел там
                    if curr.left is None:
                        curr.left = Node(key)
                        return
                    # Иначе продолжаем идти по левой ветви
                    else:
                        curr = curr.left

                # Если новый ключ больше текущего, то идем направо
                else:
                    # Если правая ветвь пуста, создаем новый узел там
                    if curr.right is None:
                        curr.right = Node(key)
                        return
                    # Иначе продолжаем идти по правой ветви
                    else:
                        curr = curr.right

    # Удаление узла из дерева
    def delete(self, key):
        def find_min_node(node):
            while node.left is not None:
                node = node.left
            return node

        def delete_helper(node, key):
            # Если узел не найден, возвращаем его
            if node is None:
                return node

```

```

# Если ключ меньше значения узла, рекурсивно ищем его в левом
поддереве
if key < node.key:
    node.left = delete_helper(node.left, key)

# Если ключ больше значения узла, рекурсивно ищем его в правом
поддереве
elif key > node.key:
    node.right = delete_helper(node.right, key)

else:
    # Если у узла нет потомков, просто удаляем его
    if node.left is None and node.right is None:
        node = None

    # Если у узла есть только один потомок, просто заменяем его
    на этого потомка
    elif node.left is None:
        node = node.right
    elif node.right is None:
        node = node.left

    # Если у узла есть два потомка
    else:
        # Находим узел с минимальным значением в правом
        поддереве
        min_node = find_min_node(node.right)
        # Заменяем удаляемый узел на узел с минимальным
        значением
        node.key = min_node.key
        # Рекурсивно удаляем узел с минимальным значением из
        правого поддерева
        node.right = delete_helper(node.right, min_node.key)
    return node

# Вызываем вспомогательную функцию для удаления узла
self.root = delete_helper(self.root, key)

# Проверка наличия узла в дереве
def exists(self, key):
    curr = self.root
    # Продолжаем обход дерева, пока не дойдем до конца или не найдем
    узел с искомым ключом
    while curr is not None:
        # Если ключ текущего узла равен искомому, возвращаем True
        if key == curr.key:
            return True
        # Если искомый ключ меньше ключа текущего узла, идем влево
        elif key < curr.key:
            curr = curr.left
        # Если искомый ключ больше ключа текущего узла, идем вправо
        else:
            curr = curr.right
    # Если не нашли искомый узел, возвращаем False
    return False

# Поиск ключа следующего по величине

```

```

def next(self, key):
    curr = self.root
    next_key = None
    # Продолжаем обход дерева, пока не дойдем до конца или не найдем
    # узел с искомым ключом
    while curr is not None:
        # Если ключ текущего узла больше искомого, запоминаем его, идем
        # влево
        if curr.key > key:
            if next_key is None or curr.key < next_key:
                next_key = curr.key
            curr = curr.left
        # Если ключ текущего узла меньше или равен искомому, идем
        # вправо
        else:
            curr = curr.right
    # Если нашли следующий ключ, возвращаем его в виде строки, иначе
    # возвращаем "none"
    return str(next_key) if next_key is not None else "none"

# Поиск ключа предыдущего по величине
def prev(self, key):
    curr = self.root
    prev_key = None
    # Продолжаем обход дерева, пока не дойдем до конца или не найдем
    # узел с искомым ключом
    while curr is not None:
        # Если ключ текущего узла меньше искомого, запоминаем его, идем
        # вправо
        if curr.key < key:
            if prev_key is None or curr.key > prev_key:
                prev_key = curr.key
            curr = curr.right
        # Если ключ текущего узла больше или равен искомому, идем влево
        else:
            curr = curr.left
    # Если нашли предыдущий ключ, возвращаем его в виде строки, иначе
    # возвращаем "none"
    return str(prev_key) if prev_key is not None else "none"

bst = BST()
with open("input.txt", "r") as f:
    n = 0
    for line in f:
        n += 1
f.close()
with open("output.txt", "w+") as f_out, open('input.txt') as f:
    for i in range(n):
        line = f.readline().strip().split()
        if line[0] == "insert": # если первое слово в строке - "insert"
            bst.insert(int(line[1])) # вызываем метод insert
        elif line[0] == "delete": # если первое слово в строке - "delete"
            bst.delete(int(line[1])) # вызываем метод delete
        elif line[0] == "exists": # если первое слово в строке - "exists"
            f_out.write(str(bst.exists(int(line[1]))) + '\n') # вызываем
            # метод exists

```



```

elif line[0] == "next": # если первое слово в строке - "next"
    f_out.write(str(bst.next(int(line[1]))) + '\n') # вызываем
метод next
elif line[0] == "prev": # если первое слово в строке - "prev"
    f_out.write(str(bst.prev(int(line[1]))) + '\n') # вызываем
метод prev

```

Данный код реализует бинарное дерево поиска (BST). Он принимает на вход команды для работы с BST из файла "input.txt", выполняет их и записывает результаты в файл "output.txt". Команды могут быть "insert" (добавление элемента), "delete" (удаление элемента), "exists" (проверка наличия элемента), "next" (поиск следующего элемента) и "prev" (поиск предыдущего элемента). Каждая команда в файле "input.txt" должна быть представлена на отдельной строке.

```

input.txt
1 insert 2
2 insert 5
3 insert 3
4 exists 2
5 exists 4
6 next 4
7 prev 4
8 delete 5
9 next 4
10 prev 4

output.txt
1 True
2 False
3 5
4 3
5 none
6 3

```

Вывод по 5 задаче: В ходе работы над 5 задачей было реализовано простое двоичное дерево поиска

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

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакой вершину дважды.

Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырём.

Дано двоичное дерево поиска. В вершинах этого дерева записаны ключи — целые числа, по модулю не превышающие 109. Для каждой вершины дерева V выполняется следующее условие:

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

Найдите высоту данного дерева.

- 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. $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$. Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

- Формат вывода / выходного файла (output.txt). Выведите одно целое число – высоту дерева.

- Ограничение по времени. 2 сек.

- Ограничение по памяти. 256 мб.

```
from time import perf_counter
import tracemalloc
```

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

```
# Класс узла дерева
```

```
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок
        self.key = key # Значение узла
```

```
# Вставка узла в дерево
```

```
def insert(root, key):
    # Если дерево пустое, создаем новый узел
    if root is None:
        return Node(key)
    else:
        # Если значение ключа больше значения текущего узла, переходим в
        # правое поддерево
        if root.key < key:
            root.right = insert(root.right, key)
        # Если значение ключа меньше или равно значению текущего узла,
        # переходим в левое поддерево
        else:
            root.left = insert(root.left, key)
    return root
```

```

# Нахождение высоты дерева
def height(root):
    # Если дерево пустое, высота равна 0
    if root is None:
        return 0
    else:
        # Рекурсивно находим высоту левого и правого поддеревьев
        left_height = height(root.left)
        right_height = height(root.right)
        # Возвращаем максимальное значение высоты и прибавляем 1, чтобы
        # учесть текущий узел
        return max(left_height, right_height) + 1

root = None
with open("input.txt", "r") as f_in:
    n = int(f_in.readline())
    for i in range(n):
        key, left, right = map(int, f_in.readline().split())
        if i == 0:
            root = Node(key)
        else:
            insert(root, key)
with open("output.txt", "w+") as f_out:
    f_out.write(str(height(root)))
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Данный код считывает из файла "input.txt" число узлов дерева и их значения. Затем, используя функцию "insert", строит дерево из узлов. Далее, находит высоту дерева с помощью функции "height" и записывает результат в файл "output.txt". В конце код выводит время работы и используемую память.

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

Вывод по 8 задаче: В ходе работы над восьмой задачей был реализован алгоритм, который выводит высоту дерева.

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

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

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

В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие 109. 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 ($1 \leq i \leq N$) 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 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 109 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. $1 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$, $1 \leq M \leq 2 \cdot 10^5$

- Формат вывода / выходного файла (output.txt). Выведите М строк. На i-ой строке требуется вывести число вершин, оставшихся в дереве после выполнения i-го запроса на удаление.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

```

from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Класс узла дерева
class Node:
    def __init__(self, key):
        # Инициализируем поля узла
        self.key = key # Значение узла
        self.left = None # Левый потомок
        self.right = None # Правый потомок

# Класс бинарного дерева
class BinaryTree:
    def __init__(self):
        self.root = None

# Добавление узла в дерево
def addNode(self, key):
    x = self.root
    y = None
    cmp = 0
    # Пока не достигнем конца дерева
    while x is not None:
        # Сравниваем ключ текущего узла с ключом нового узла
        cmp = x.key - key
        if cmp == 0: # Если ключи равны, новый узел не добавляем
            return
        else:
            y = x # Сохраняем текущий узел в y
            # Если ключ нового узла больше текущего, идем направо
            if cmp < 0:
                x = x.right
            # Иначе идем налево
            else:
                x = x.left
    # Создаем новый узел
    newNode = Node(key)
    # Если дерево пустое, новый узел становится корневым
    if y is None:
        self.root = newNode
        print(key, "корень")
    else:

```

```

        # Если ключ нового узла больше у, новый узел становится правым
        # потомком у
        if cmp > 0:
            y.left = newNode
            print(key, "левый для", y.key)
        # Иначе новый узел становится левым потомком у
        else:
            y.right = newNode
            print(key, "правый для", y.key)

# Удаление узла из дерева
def removeSubtree(self, key):
    x = self.root
    y = None
    cmp = 0
    # Пока не достигнем конца дерева
    while x is not None:
        # Вычисляем разницу ключей
        cmp = x.key - key
        # Если ключи совпадают, то удаляем узел
        if cmp == 0:
            break
        else:
            y = x
            # Если ключ искомого узла больше текущего ключа, идем
            # направо
            if cmp < 0:
                x = x.right
            # Иначе идем налево
            else:
                x = x.left
    # Если узел не найден
    if x is None:
        print("Ничего не удалено")
        return 0
    count = self.nodesCount(x)
    print("удаляется", x.key)
    print("его поддереву", count, "узлов")
    # Удаляем ссылку на узел из родительского узла
    if x.key > y.key:
        y.right = None
    else:
        y.left = None
    # Удаляем ссылку на узел
    x = None
    # Возвращаем количество узлов в поддереве
    return count

# Подсчет количества узлов в поддереве
def nodesCount(self, node):
    # Если у узла нет потомков, возвращаем 1
    if node.left is None and node.right is None:
        return 1
    left = right = 0

    # Если есть левый потомок, считаем количество узлов в его поддереве
    if node.left is not None:

```

```

        left = self.nodesCount(node.left)

        # Если есть правый потомок, считаем количество узлов в его
        поддереве
        if node.right is not None:
            right = self.nodesCount(node.right)

        # Возвращаем количество узлов в поддереве
        return left + right + 1

if __name__ == "__main__":
    with open("input.txt") as f:
        nodesCount = int(f.readline()) # количество узлов в дереве
        arrayNodes = [] # массив для хранения узлов дерева
        for i in range(nodesCount):
            node = [int(x) for x in f.readline().split()]
            arrayNodes.append(node)

        removesCount = int(f.readline())
        arrayRemove = [int(x) for x in f.readline().split()]
        tree = BinaryTree()
        for node in arrayNodes:
            tree.addNode(node[0])
        with open("output.txt", "w") as f:
            for i in range(removesCount):
                nodesCount -= tree.removeSubtree(arrayRemove[i])
                print(nodesCount)
                f.write(str(nodesCount) + "\n")
    print("Время работы: %s секунд" % (perf_counter() - t_start))
    snapshot = tracemalloc.take_snapshot()
    top_stats = snapshot.statistics('traceback')
    stat = top_stats[0]
    print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Этот код представляет собой реализацию бинарного дерева поиска с возможностью добавления и удаления узлов, а также подсчетом количества узлов в поддереве. Он принимает на вход данные из файла input.txt, в котором первой строкой записано количество узлов в дереве, а далее на каждой строке через пробел записано значение ключа искомого узла и 1, если узел должен быть удален, или 0, если он должен быть добавлен. После выполнения работы программы результат записывается в файл output.txt.

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

Вывод по 9 задаче: В ходе работы над девятой задачей был реализован алгоритм, который выводит число вершин, оставшихся в дереве после выполнения определенного запроса на удаление

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

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

- все ключи вершин из левого 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


```

from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

# Проверка, является ли данное дерево двоичным деревом поиска
def is_bst(node, min_val=float('-inf'), max_val=float('inf')):
    # Если узел не существует, значит дерево пустое и является двоичным
    деревом поиска
    if node is None:
        return True
    # Если значение узла находится вне заданных пределов, значит дерево не
    является двоичным деревом поиска
    if node.value < min_val or node.value > max_val:
        return False
    # Рекурсивно проверяем левое и правое поддеревья на то, являются ли они
    двоичными деревьями поиска
    # Для левого поддерева значение верхней границы изменяется на
    (node.value - 1),
    # а для правого поддерева - на (node.value + 1)
    return is_bst(node.left, min_val, node.value - 1) and
is_bst(node.right, node.value + 1, max_val)

# Класс узла двоичного дерева
class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value # Значение узла
        self.left = left # Левый потомок
        self.right = right # Правый потомок

with open("input.txt", "r") as f_in:
    n = int(f_in.readline())
    # Список узлов
    nodes = [Node() for i in range(n)]
    for i in range(n):
        value, left, right = map(int, f_in.readline().split())
        nodes[i].value = value
        if left != 0:
            nodes[i].left = nodes[left - 1]
        if right != 0:
            nodes[i].right = nodes[right - 1]
with open("output.txt", "w") as f_out:
    # Если дерево пустое, то оно является двоичным деревом поиска
    if n == 0:
        f_out.write('YES')
    # Если дерево не пустое, проверяем, является ли оно двоичным деревом
    поиска
    else:
        if is_bst(nodes[0]):
            f_out.write('YES')
        else:
            f_out.write('NO')
print("Время работы: %s секунд" % (perf_counter() - t_start))

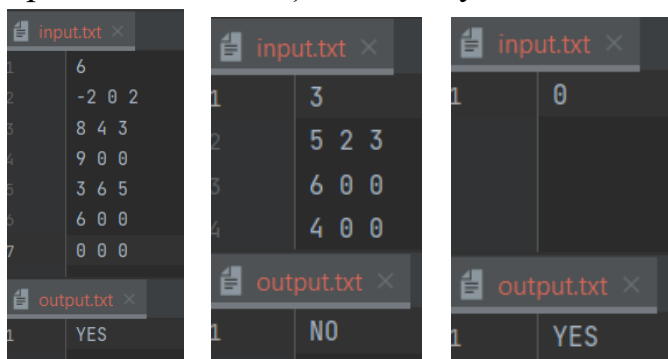
```

```

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Этот код считывает информацию о двоичном дереве из файла input.txt, проверяет, является ли оно двоичным деревом поиска, и записывает результат в файл output.txt (YES, если дерево является двоичным деревом поиска, и NO в противном случае). Код также выводит время работы и количество используемой памяти. Файл input.txt содержит количество узлов в дереве и информацию об узлах (значение узла и номера левого и правого потомков), каждый узел на отдельной строке.



Вывод по 10 задаче: В ходе работы над десятой задачей был реализован алгоритм, который проверяет, является ли дерево двоичным деревом поиска.

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

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

- 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. $n \leq 100000$, $|k_i| \leq 109$.
- 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

```

from bisect import bisect_left, insort_left # двоичный поиск
from time import perf_counter
import tracemalloc

t_start = perf_counter()
tracemalloc.start()

class OrderStatisticTree:
    def __init__(self):
        self.tree = [] # Списка для хранения элементов
        self.count = 0 # Счетчик числа элементов в списке

    # Добавляем элемент в список
    def insert(self, x: int):
        insort_left(self.tree, x) # Вставляем элемент в отсортированный
        self.count += 1 # Увеличиваем счетчик числа элементов в списке

    # Поиска k-го максимума
    def find_by_order(self, k: int) -> int:
        return self.tree[k] # Возвращаем k-й элемент отсортированного
        список

    # Удаления элемента из списка
    def erase(self, x: int):
        index = bisect_left(self.tree, x) # Находим индекс элемента в
        отсортированном списке
        if index < len(self.tree) and self.tree[index] == x: # Если
        элемент найден
            self.tree.pop(index) # Удаляем элемент из списка
            self.count -= 1 # Уменьшаем счетчик числа элементов в списке

if __name__ == '__main__':
    with open("input.txt", "r") as f:
        n = int(f.readline())
        f.close()
        # Создаем объект класса OrderStatisticTree для хранения дерева и
        счетчика
        oSet = OrderStatisticTree()
        with open("output.txt", "w+") as f_out, open('input.txt') as f:
            for i in range(n + 1):
                command, *args = map(int, f.readline().split())
                # Если команда равна 1, то добавляем аргумент как элемент в
                дерево

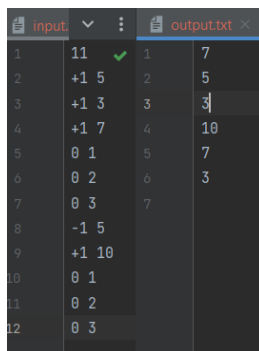
```

```

        if command == 1:
            oSet.insert(*args)
            # Если команда равна 0, то находим k-й максимум, где k =
oSet.count - args[0]
        elif command == 0:
            k = oSet.count - args[0]
            f_out.write(str(oSet.find_by_order(k)) + '\n')
            # Если команда равна -1, то удаляем элемент из дерева по
аргументу
        elif command == -1:
            oSet.erase(*args)
print("Время работы: %s секунд" % (perf_counter() - t_start))
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))

```

Данный код реализует структуру данных "дерево порядковой статистики", которая хранит отсортированный список элементов и позволяет выполнять операции вставки, удаления и поиска k-го максимума. На вход программе подается файл input.txt, содержащий число n, а затем n+1 команду и аргументы для них. В файл output.txt записываются результаты выполнения команды 0 - поиска k-го максимума.



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	

Вывод по 16 задаче: В ходе работы над шестнадцатой задачей был реализован алгоритм, который позволяет добавлять и удалять элементы, а также находить k-й максимум.

Вывод

В ходе лабораторной работы были реализованы и применены в задачах: двоичные деревья поиска и сбалансированные деревья поиска (AVL и Splay).