Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2
ПО КУРСУ «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»
ТЕМА: ДВОИЧНЫЕ ДЕРЕВЬЯ ПОИСКА
ВАРИАНТ 10

Выполнил:
Соболев А. А
K3240

Проверила:
Ромакина О. М.

Санкт-Петербург
2025

СОДЕРЖАНИЕ

ЗАДАЧИ ПО ВАРИАНТУ

5 задача. Простое двоичное дерево поиска (1 балл)

Постановка задачи

Реализуйте простое двоичное дерево поиска.

Листинг кода

```
import time, tracemalloc

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


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

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

    def _insert(self, node, key):
        if node is None:
            return BSTNode(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
            node.right = self._insert(node.right, key)
        return node

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

    def _delete(self, node, key):
        if node is None:
            return None
        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            if node.left is None:
                return node.right
            if node.right is None:
                return node.left
            min_larger_node = self._min_value_node(node.right)
            node.key = min_larger_node.key
            node.right = self._delete(node.right, min_larger_node.key)
        return node

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

    def _exists(self, node, key):
        if node is None:
            return False
        if key == node.key:
```

```
            return True
        if key < node.key:
            return self._exists(node.left, key)
        return self._exists(node.right, key)

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

    def prev(self, key):
        node = self.root
        predecessor = None
        while node:
            if node.key < key:
                predecessor = node
                node = node.right
            else:
                node = node.left
        return predecessor.key if predecessor else "none"

    def _min_value_node(self, node):
        while node.left:
            node = node.left
        return node

with open("input.txt", "r") as file:
    commands = file.readlines()


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

bst = BST()
results = []

for command in commands:
    operation, x = command.split()
    if operation == "insert":
        bst.insert(x)
    elif operation == "delete":
        bst.delete(x)
    elif operation == "exists":
        results.append("true" if bst.exists(x) else "false")
    elif operation == "next":
        results.append(str(bst.next(x)))
    elif operation == "prev":
        results.append(str(bst.prev(x)))

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("\n".join(results) + "\n")
```

Объяснение кода

Я реализовал 2 класса: BSTNode, BST. Первый класс содержит значение узла, а также ссылки на левого и правого ребенка. Основную логику двоичного дерева поиска я поместил в класс BST. Изначально оно создается пустым. При вставке элемента, если дерево пустое, то значение помещается в вершину. Если дерево не пустое, то происходит сравнение с элементом, если он больше, то запускается рекурсивный спуск вправо, иначе влево. Так происходит, пока не найдется нужная вершина, которая не занята. Удаление происходит по похожему принципу. Метод рекурсивно обходит дерево и ищет нужное значение. Если у найденного значения один ребенок, то на место вершины помещается его ребенок. Если два ребенка, то ищется минимальный элемент в правом ребенке. При таком подходе гарантируется, что будет найден минимальный элемент, который будет больше левого ребенка и меньше правого. Для поиска элемента происходит рекурсивный обход. Поиск следующего элемента происходит также. Значение сравнивается с вершиной и ищется минимальное, которое больше требуемого. Происходит предыдущего происходит похожим образом

Время и память

- Время выполнения: 0.001386 секунд (2 секунды ограничение)
- Использовано памяти: 0.001714 Мб (512 мб ограничение)

Примеры

Вводный файл	Выходной файл
insert 2 insert 5 insert 3 exists 2 exists 4 next 4 prev 4 delete 5 next 4 prev 4	true false 5 3 none 3

Вывод

Я реализовал простейшее двоичное дерево поиска с операциями insert/delete/exists/next/prev.

8 задача. Высота дерева возвращается (2 балла)

Постановка задачи

Дано двоичное дерево поиска. Найдите высоту данного дерева.

Листинг кода

```
import time, tracemalloc

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

def bst_height(node):
    if node is None:
        return 0
    left_height = bst_height(node.left)
    right_height = bst_height(node.right)
    return max(left_height, right_height) + 1

with open("input.txt", "r") as file:
    lines = file.readlines()

N = int(lines[0].strip())
if N == 0:
    with open("output.txt", "w") as file:
        file.write("0\n")
    exit()


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

nodes = {i: BSTNode(int(lines[i].split()[0])) for i in range(1, N + 1)}

for i in range(1, N + 1):
    _, left, right = map(int, lines[i].split())
    if left:
        nodes[i].left = nodes[left]
    if right:
        nodes[i].right = nodes[right]

height = bst_height(nodes[1])

end_time = time.perf_counter()

current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write(f"{height}\n")
```

Объяснение

Я создал словарь, в который добавил все вершины по порядку. После я к каждой вершине добавил их детей, если они есть. В функции я рекурсивно обходил дерево по каждому из детей, находя максимальную высоту.

Время и память

- Время выполнения: 0.000089 секунд (2 секунды ограничение)
- Использовано памяти: 0.002918 Мб (256 мб ограничение)

Примеры

Вводный файл	Выходной файл
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	4

Вывод

Я вычислил высоту BST через рекурсивный обход.

11 задача. Сбалансированное двоичное дерево поиска (2 балла)

Постановка задачи

Реализуйте сбалансированное двоичное дерево поиска.

Листинг кода

```
import time, tracemalloc

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

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

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

    def _insert(self, node, key):
        if node is None:
            return BSTNode(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
            node.right = self._insert(node.right, key)
        return node

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

    def _delete(self, node, key):
        if node is None:
            return None
        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            if node.left is None:
                return node.right
            if node.right is None:
                return node.left
            min_larger_node = self._min_value_node(node.right)
            node.key = min_larger_node.key
            node.right = self._delete(node.right, min_larger_node.key)
        return node

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

    def _exists(self, node, key):
        if node is None:
            return False
        if key == node.key:
            return True
        return self._exists(node.left, key) if key < node.key else
self._exists(node.right, key)
```

```
    def _min_value_node(self, node):
        while node.left:
            node = node.left
        return node

    def find_bound(self, key, find_next):
        node = self.root
        bound = None
        while node:
            if (find_next and node.key > key) or (not find_next and node.key
< key):
                bound = node
                node = node.left if find_next else node.right
            else:
                node = node.right if find_next else node.left
        return bound.key if bound else "none"

    def next(self, key):
        return self.find_bound(key, True)

    def prev(self, key):
        return self.find_bound(key, False)

with open("input.txt", "r") as file:
    commands = file.readlines()

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

bst = BST()
results = []

for command in commands:
    if not command.strip():
        continue
    operation, x = command.split()
    x = int(x)  # Преобразуем в число

    if operation == "insert":
        bst.insert(x)
    elif operation == "delete":
        bst.delete(x)
    elif operation == "exists":
        results.append("true" if bst.exists(x) else "false")
    elif operation == "next":
        results.append(str(bst.next(x)))
    elif operation == "prev":
        results.append(str(bst.prev(x)))

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("\n".join(results) + "\n"
```

Объяснение

Я создал классы BSTNode и BST: узел хранит ключ и ссылки на левого и правого ребёнка, а дерево рекурсивно вставляет и удаляет ключи, проверяет существование, находит следующий и предыдущий ключи, затем считывает команды из input.txt, для операций exists/next/prev.

Время и память

- Время выполнения: 0.000082 секунд (2 секунды ограничение)
- Использовано памяти: 0.001834 Мб (512 Мб ограничение)

Примеры

Вводный файл	Выходной файл
insert 2 insert 5 insert 3 exists 2 exists 4 next 4 prev 4 delete 5 next 4 prev 4	true false 5 3 none 3

Вывод

Я реализовал рекурсивный BST с операциями insert/delete/exists/next/prev.

ДОПОЛНИТЕЛЬНЫЕ ЗАДАЧИ

3 задача. Простейшее BST (1 балл)

Постановка задачи

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.

Листинг кода

```
import time, tracemalloc

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

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

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
            return
        current = self.root
        while True:
            if key < current.key:
                if current.left is None:
                    current.left = Node(key)
                    return
                current = current.left
            elif key > current.key:
                if current.right is None:
                    current.right = Node(key)
                    return
                current = current.right
            else:
                return

    def next(self, x):
        answer = None
        current = self.root
        while current:
            if current.key > x:
                answer = current.key
                current = current.left
            else:
                current = current.right
        return answer if answer is not None else 0

start_time = time.perf_counter()
```

```
tracemalloc.start()

bst = BST()
output_lines = []

with open("input.txt", "r") as infile:
    for line in infile:
        line = line.strip()
        if not line:
            continue
        op, x_str = line.split()
        x = int(x_str)
        if op == '+':
            bst.insert(x)
        elif op == '>':
            result = bst.next(x)
            output_lines.append(str(result))


end_time = time.perf_counter()

current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as outfile:
    outfile.write("\n".join(output_lines) + "\n")
```

Объяснение

Я создал BST с помощью класса, у каждого экземпляра вершины есть ее значение и ссылка на левого и правого ребенка. При вставке если есть вершина, то значение сравнивается с вершиной. Если оно больше, то идет сравнение с правым ребенком вершины, иначе с левым. Так происходит, пока не найдется нужное незанятое место

Время и память

- Время выполнения: 0.000335 секунд (2 секунды ограничение)
- Использовано памяти: 0.019208 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
+ 1 + 3 + 3 > 1 > 2 > 3 + 2 > 1	3 3 0 2

Вывод

Я реализовал простой BST с вставкой и поиском, научился обходить дерево без рекурсии.

4 задача. Простейший неявный ключ (1 балл)

Постановка задачи

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

Листинг кода

```
import time, tracemalloc

class BST:
    def __init__(self):
        self.data = []

    def insert(self, x):
        if x not in self.data:
            self.data.append(x)
            self.data.sort()

    def kth_element(self, k):
        return self.data[k - 1] if 1 <= k <= len(self.data) else "0"

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

bst = BST()
results = []

with open("input.txt", "r") as file:
    for line in file:
        parts = line.strip().split()
        command, x = parts[0], int(parts[1])
        if command == '+':
            bst.insert(x)
        elif command == '?':
            results.append(str(bst.kth_element(x)))

end_time = time.perf_counter()

current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("\n".join(results) + "\n")
```

Объяснение

Я читаю команды: при + x добавляю x в список, если его там нет, и сортирую список, при ? k возвращаю k-й по возрастанию элемент или 0, если k вне диапазона.

Время и память

- Время выполнения: 0.018053 секунд (2 секунды ограничение)
- Использовано памяти: 0.017599 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
+ 1 + 4 + 3 + 3 ? 1 ? 2 ? 3 + 2 ? 3	1 3 4 3

Вывод

Я реализовал BST по неявному ключу через поддержание отсортированного списка для быстрого получения k-го элемента.

6 задача. Опознание двоичного дерева поиска (1.5 балла)

Постановка задачи

Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска.

Листинг кода

```
import time
import tracemalloc

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

def is_valid_bst(node, min_val=float('-inf'), max_val=float('inf')):
    if node is None:
        return True
    if not (min_val < node.key < max_val):
        return False
    return (is_valid_bst(node.left, min_val, node.key) and
            is_valid_bst(node.right, node.key, max_val))

with open("input.txt", "r") as file:
    lines = file.readlines()

N = int(lines[0].strip())
if N == 0:
    with open("output.txt", "w") as file:
        file.write("CORRECT\n")
    exit()

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

nodes = {i: BSTNode(int(lines[i+1].split()[0])) for i in range(N)}

for i in range(N):
    _, left, right = map(int, lines[i+1].split())
    if left != -1:
        nodes[i].left = nodes[left]
    if right != -1:
        nodes[i].right = nodes[right]

is_bst = is_valid_bst(nodes[0])

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("CORRECT\n" if is_bst else "INCORRECT\n")
```

Объяснение

Я создал словарь, в который добавил все вершины по порядку. После я к каждой вершине добавил их детей, если они есть. Проверка на корректное двоичное дерево происходит в функции, в которую передается вершина. Она рекурсивно обходит дерево, сравнивая вершину с минимальным значением и максимальным. Если на всех этапах проверка проходит, то двоичное дерево является правильным, иначе неправильным.

Время и память

- Время выполнения: 0.000053 секунд (10 секунд ограничение)
- Использовано памяти: 0.002085 Мб (512 Мб ограничение)

Примеры

Вводный файл	Выходной файл
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT
0	CORRECT

5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

Вывод

Я построил дерево из списка узлов и проверил его корректность через рекурсивный обход.

7 задача. Опознание двоичного дерева поиска (усложненная версия) (2.5 балла)

Постановка задачи

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 .

Листинг кода

```
import time
import tracemalloc

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

def is_valid_bst(node, min_val=float('-inf'), max_val=float('inf')):
    if node is None:
        return True
    if not (min_val <= node.key < max_val):
        return False
    return (is_valid_bst(node.left, min_val, node.key) and
            is_valid_bst(node.right, node.key, max_val))

with open("input.txt", "r") as file:
    lines = file.readlines()

N = int(lines[0].strip())
if N == 0:
    with open("output.txt", "w") as file:
        file.write("CORRECT\n")
    exit()

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

nodes = {i: BSTNode(int(lines[i+1].split()[0])) for i in range(N)}

for i in range(N):
    _, left, right = map(int, lines[i+1].split())
    if left != -1:
        nodes[i].left = nodes[left]
    if right != -1:
        nodes[i].right = nodes[right]
```

```
is_bst = is_valid_bst(nodes[0])

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("CORRECT\n" if is_bst else "INCORRECT\n")
```

Объяснение

Я использовал ту же функцию, что и в прошлой задаче, но в прошлой задаче было строгое сравнение, а в этой нестрогое.

Время и память

- Время выполнения: 0.000049 секунд (10 секунд ограничение)
- Использовано памяти: 0.001335 Мб (512 Мб ограничение)

Примеры

Вводный файл	Выходной файл
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

3 2 1 2 1 -1 -1 2 -1 -1	CORRECT
3 2 1 2 2 -1 -1 3 -1 -1	INCORRECT
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT
1 2147483647 -1 -1	CORRECT

Вывод

То же, что в задаче 6, но с учётом нестрогого сравнения для равных ключей.

9 задача. Удаление поддеревьев (2 балла)

Постановка задачи

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.

Листинг кода

```
import time, tracemalloc
from collections import deque

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

with open('input.txt', 'r') as f:
    lines = f.readlines()

n = int(lines[0])
raw_tree = {}
key_to_index = {}

for i in range(1, n + 1):
    k, l, r = map(int, lines[i].split())
    raw_tree[i] = (k, l, r)
    key_to_index[k] = i

m = int(lines[n + 1])
delete_keys = list(map(int, lines[n + 2].split()))

def build_children(tree):
    children = {}
    for i, (_, l, r) in tree.items():
        if i not in children:
            children[i] = []
        if l != 0:
            children[i].append(l)
        if r != 0:
            children[i].append(r)
    return children

def collect_subtree_nodes(start, children):
    result = set()
    q = deque([start])
    while q:
        u = q.popleft()
        result.add(u)
        for v in children.get(u, []):
            q.append(v)
    return result

with open('output.txt', 'w') as f:
    current_tree = raw_tree.copy()
    for k in delete_keys:
        if k in key_to_index and key_to_index[k] in current_tree:
            root = key_to_index[k]
            children = build_children(current_tree)
```

```
            to_delete = collect_subtree_nodes(root, children)
            for d in to_delete:
                current_tree.pop(d, None)
        f.write(f"{len(current_tree)}\n")

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")
```

Объяснение кода

Я создаю словарь, где для каждой вершины храню ключ и номера детей. Также создаю словарь, чтобы по ключу можно было быстро найти номер вершины. Затем обрабатываю запросы. Для каждого запроса я копирую текущее состояние дерева. Если в дереве ещё есть вершина с указанным ключом, я запускаю обход от неё вниз и собираю все вершины поддерева. Потом удаляю эти вершины из дерева. После удаления считаю, сколько вершин осталось, и записываю это число в файл. Так как дерево меняется после каждого запроса, я каждый раз пересчитываю структуру, чтобы учесть только оставшиеся вершины.

Время и память

- Время выполнения: 0.000478 секунд (2 секунды ограничение)
- Использовано памяти: 0.017502 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
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

Вывод

Я построил словарь детей, использовал обход в ширину (BFS) для сбора и удаления всех узлов поддерева по ключу, после каждого удаления обновлял структуру дерева и считал оставшиеся вершины.

10 задача. Проверка корректности (2 балла)

Постановка задачи

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

Листинг кода

```
import time
import tracemalloc

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

def is_valid_bst(node, min_val=float('-inf'), max_val=float('inf')):
    if node is None:
        return True
    if not (min_val < node.key < max_val):
        return False
    return (is_valid_bst(node.left, min_val, node.key) and
            is_valid_bst(node.right, node.key, max_val))

with open("input.txt", "r") as file:
    lines = file.readlines()

N = int(lines[0].strip())
if N == 0:
    with open("output.txt", "w") as file:
        file.write("YES\n")
    exit()

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

nodes = [BSTNode(int(lines[i + 1].split()[0])) for i in range(N)]

for i in range(N):
    parts = list(map(int, lines[i + 1].split()))
    key, left, right = parts[0], parts[1], parts[2]
    if left != 0:
        nodes[i].left = nodes[left - 1]
    if right != 0:
        nodes[i].right = nodes[right - 1]

is_bst = is_valid_bst(nodes[0])

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write("YES\n" if is_bst else "NO\n")
```

Объяснение

Я создаю список узлов и потом добавляю значения в узлы. После начинаю проверку на BST с вершины. На каждом этапе я проверяю лежит ли ключ в диапазоне. Если это не выполняется, то дерево не является BST, если все проверки прошли, то BST.

Время и память

- Время выполнения: 0.000492 секунд (2 секунды ограничение)
- Использовано памяти: 0.003086 Мб (256 секунд ограничение)

Примеры

Входной файл	Выходной файл
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	YES
3 5 2 3 6 0 0 4 0 0	NO
0	YES

Вывод

Я проверил корректность дерева через рекурсивную проверку ключей.

12 задача. Проверка сбалансированности (2 балла)

Постановка задачи

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

Листинг кода

```
import time, tracemalloc

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

with open('input.txt', 'r') as f:
    lines = f.readlines()

n = int(lines[0])
tree = [None] * (n + 1)
for i in range(1, n + 1):
    k, l, r = map(int, lines[i].split())
    tree[i] = (k, l, r)

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

def dfs(v):
    if v == 0:
        return 0
    _, l, r = tree[v]
    hl = dfs(l)
    hr = dfs(r)
    height[v] = 1 + max(hl, hr)
    balance[v] = hr - hl
    return height[v]

if n > 0:
    dfs(1)

with open('output.txt', 'w') as f:
    for i in range(1, n + 1):
        f.write(f"{balance[i]}\n")

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")
```

Объяснение кода

Я считываю дерево в список кортежей (ключ, левый, правый), затем рекурсивно обхожу его функцией, которая возвращает высоту узла, сохраняет в массив высот и вычисляет баланс = высота правого - высота левого поддерева.

Время и память

- Время выполнения: 0.000571 секунд (2 секунды ограничение)
- Использовано памяти: 0.017384 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	3 -1 0 0 0 0

Вывод

Для каждой вершины получен её баланс.

ВЫВОД

Я изучил двоичные деревья поиска и операции над ними: вставка и удаление, проверка корректности, вычисления высоты. Написал рекурсивные и итеративные обходы.