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 23

Выполнил:
Тарасов А.Н.
К3244

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Лабораторная работа №2.

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: ž «+ x» – добавить в дерево x (если x уже есть, ничего не делать). ž «> x» – вернуть минимальный элемент больше x или 0, если таких нет.

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

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

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self.inserting(self.root, key)

    def inserting(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = Node(key)
            else:
                self.inserting(root.left, key)
        elif key > root.val:
            if root.right is None:
                root.right = Node(key)
            else:
                self.inserting(root.right, key)

    def finding(self, root, key, min_greater):
        if root is None:
            return 0 if min_greater == float('inf') else min_greater
        if root.val > key:
            min_greater = min(min_greater, root.val)
            return self.finding(root.left, key, min_greater)
        else:
            return self.finding(root.right, key, min_greater)

    def find_min(self, key):
        return self.finding(self.root, key, float('inf'))

def process_commands(commands):
    bst = BST()
    result = []
    for command in commands:
        if command.startswith('+'):
```

```
            _, x = command.split()
            bst.insert(int(x))
        elif command.startswith('>'):
            _, x = command.split()
            result.append(bst.find_min(int(x)))
    return result

def main():
    with open('input.txt', 'r', encoding='utf-8') as input_file,
open('output.txt', 'w', encoding='utf-8') as output_file:
        commands = input_file.readlines()
        result = process_commands(commands)
        for elem in result:
            output_file.write(f'{elem}\n')

if __name__ == '__main__':
    main()

kon = time.time()
c = kon - nach
print('Время :', c)

import os, psutil; print(psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2)
```

Если введенная строка начинается с '+' добавляем x в дерево с помощью метода insert. Рекурсивно находя правильное место для нового узла. Если запрос начинается с '>' ищем минимальное значение больше x (def find_min) def finging: Если введенное значение меньше текущего узла, рекурсивно вставляется в левое поддерево. Если введенное значение больше текущего узла, рекурсивно вставляем в правое поддерево. В finding ищем минимальный элемент в дереве, который больше введенного x. -Если значение текущего узла больше x, обновляет минимальное значение и продолжаем поиск в левом поддереве, чтобы возможно найти ещё меньшее значение. -Если текущее значение меньше или равно x, поиск продолжается в правом поддереве

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: $-1 \leq B(V)$

≤ 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

```
import time
nach = time.time()

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


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

    def add_node(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._add_node(self.root, key)

    def _add_node(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._add_node(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._add_node(node.right, key)

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

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


def build_tree_from_list(node_list):
    tree = BST()
    for key in node_list:
        tree.add_node(key)
    return tree
```

```
def get_balances(tree):
    balances = []
    queue = [tree.root]
    while queue:
        node = queue.pop(0)
        balances.append(tree.balance(node))
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return balances


def main():
    with open("input.txt", "r") as in_file, open("output.txt", "w") as
out_file:
        n = int(in_file.readline())

        if n == 0:
            out_file.write(str(0))
        else:
            nodes = [int(in_file.readline().split()[0]) for _ in range(n)]

            tree = build_tree_from_list(nodes)

            balances = get_balances(tree)
            for bl in balances:
                out_file.write(f"{bl}\n")


if __name__ == '__main__':
    main()

kon = time.time()
c = kon - nach
print('Время :', c)

import os, psutil; print(psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2)
```

Создадим дерево добавлением узлов, а конкретно для этой задачи напишем метод balance, который определяет баланс узла как разности высот левого и правого поддеревьев. С помощью очередь совершим обход дерева по уровням, для каждого узла будем сразу печатать его баланс в файл.

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

Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом: • путем спуска от корня и проверки ключей находится V – удаляемая вершина; • если вершина V – лист (то есть,

у нее нет детей): – удаляем вершину; – поднимаемся к корню, начиная с бывшего родителя вершины V , при этом если встречается несбалансированная вершина, то производим поворот. • если у вершины V не существует левого ребенка: – следовательно, баланс вершины равен единице и ее правый ребенок – лист; – заменяем вершину V ее правым ребенком; – поднимаемся к корню, производя, где необходимо, балансировку. • иначе: – находим R – самую правую вершину в левом поддереве; – переносим ключ вершины R в вершину V ; – удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являющегося листом); – поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку. Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины - корня. Результатом удаления в этом случае будет пустое дерево. Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как тестирующая система проверяет точное равенство получающихся деревьев.

```
import time
nach = time.time()


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



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


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

    def height(self, node):
        if node is None:
```

```
            return 0
        return node.height


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


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


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

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


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

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

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

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

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

```
                root.key = left_tree.key

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

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

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

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


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


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


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

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

if __name__ == '__main__':
```

```
    main()

kon = time.time()
c = kon - nach
print('Время :', c)

import os, psutil; print(psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2)
```

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

Дополнительные задачи

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

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды. Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырем. Дано двоичное дерево поиска. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 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.

```
import time
nach = time.time()

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

```
        self.left = None
        self.right = None


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

    def _add_node(self, node, key):
        if key < node.key:
            if node.left:
                self._add_node(node.left, key)
            else:
                node.left = Node(key)
        else:
            if node.right:
                self._add_node(node.right, key)
            else:
                node.right = Node(key)

    def add_node(self, key):
        if not self.root:
            self.root = Node(key)
        else:
            self._add_node(self.root, key)

    def height(self, node):
        if not node:
            return 0
        return 1 + max(self.height(node.left), self.height(node.right))


def build_tree_from_input(input_data):
    tree = BST()
    for key in input_data:
        tree.add_node(key)
    return tree

def get_height(nodes):
    tree = build_tree_from_input(nodes)
    return tree.height(tree.root)
def main():
    with open('input.txt', 'r', encoding='utf-8') as input_file,
open('output.txt', 'w',

encoding='utf-8') as output_file:
        n = int(input_file.readline().strip())

        nodes = [list(map(int, input_file.readline().strip().split())) for
_ in range(n)]
        result = get_height(nodes)
        output_file.write(str(result))

if __name__ == '__main__':
    main()

kon = time.time()
c = kon - nach
print('Время :', c)
```

```
import os, psutil; print(psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2)
```

Про класс BST:__init__ инициализирует пустое дерево с корнем root None.Метод add_node обертка для _add_node который рекурсивно добавляет узел в дерево: если ключ меньше текущего узла идем в левое поддерево, иначе — в правое. Если дерево пустое, новый узел - корень. Метод height рекурсивно вычисляет высоту дерева. Если узел пустой возвр. 0. Иначе вычисляется высота левого и правого поддерева и возвращается максимальная плюс 1. Считываем кол-во узлов. Считываем массив списков узлов, где каждый из списков содержит ключ узла и индексы левого и правого ребенка. В цикле добавляем все узлы в дерево по ключам. Вычисляем и выводим высоту дерева передавая корень.

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

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

```
import time
nach = time.time()

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

        self.size = 1


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

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

    def _inserting(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = Node(key)
            else:
                root.left = self._inserting(root.left, key)
        elif key > root.val:
            if root.right is None:
                root.right = Node(key)
            else:
```

```
                root.right = self._inserting(root.right, key)
        root.size = 1 + self.get_size(root.left) +
self.get_size(root.right)
        return root

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._inserting(self.root, key)

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

    def _delete_node(self, root, key):
        if root is None:
            return root
        if key < root.val:
            root.left = self._delete_node(root.left, key)
        elif key > root.val:
            root.right = self._delete_node(root.right, key)
        else:
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left
            temp = self._min_val_node(root.right)
            root.val = temp.val
            root.right = self._delete_node(root.right, temp.val)
        root.size = 1 + self.get_size(root.left) +
self.get_size(root.right)
        return root

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

    def _find_k_max(self, root, k):
        if root is None:
            return None
        right_size = self.get_size(root.right)
        if right_size + 1 == k:
            return root.val
        elif k <= right_size:
            return self._find_k_max(root.right, k)
        else:
            return self._find_k_max(root.left, k - right_size - 1)

    def find_k_max(self, k):
        return self._find_k_max(self.root, k)


def process_commands(operations):
    bst = BST()
    ans = []
    for line in operations:
        if line.startswith('+1'):
            _, x = line.split()
            bst.insert(int(x))
```

```
        elif line.startswith('-1'):
            _, x = line.split()
            bst.delete(int(x))
        elif line.startswith('0'):
            _, x = line.split()
            result = bst.find_k_max(int(x))
            if result is not None:
                ans.append(result)

    return ans


def main():
    with open('input.txt', 'r', encoding='utf-8') as input_file,
open('output.txt', 'w',

encoding='utf-8') as output_file:
        n = int(input_file.readline().strip())
        commands = input_file.readlines()
        ans = process_commands(commands)
        for elem in ans:
            output_file.write(f'{elem}\n')


if __name__ == '__main__':
    main()

kon = time.time()
c = kon - nach
print('Время :', c)

import os, psutil; print(psutil.Process(os.getpid()).memory_info().rss /
1024 ** 2)
```

В функции поиска к-го максимума: если дерево пустое возвращаем none, в остальных случаях вычисляем размер правого поддерева, и проверяем, если размер правого поддерева + 1 <= к, то текущий узел это к-й максимум, возвращаем его значение. Если размер правого поддерева + 1 > к, рекурсивно вызываем поиск в левом поддереве.

Вывод

Во время выполнения лабораторной работы я разобралался с основными понятиями, такими как, корень, узел, предок и ребенок, высота, листья. А также научился решать задачи про двоичные деревья поиска BST и сбалансированные АВЛ деревья. Отработал на практике обход бинарных деревьев в глубину, добавление и удаление узлов, поиск элемента в диапазоне.