САНКТ – ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №1
По курсу «Алгоритмы и структуры данных»
Тема: Жадные алгоритмы. Динамическое программирование
Вариант 13

Выполнила:
Мкртчян А.А.
K3242

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Задача N3

Описание задания: Реализовать бинарное дерево поиска (Binary Search Tree) с возможностью добавления элементов и поиска минимального элемента, который больше заданного.

Исходный код:

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

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

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        elif key > node.val:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def find_min_greater_than(self, key):
        return self._find_min_greater_than(self.root, key, None)

    def _find_min_greater_than(self, node, key, successor):
        if node is None:
            return successor

        if node.val <= key:
            return self._find_min_greater_than(node.right, key,
successor)
        else:
            return self._find_min_greater_than(node.left, key, node.val)

# Чтение данных и выполнение запросов
bst = BinarySearchTree()
results = []

with open('input.txt', 'r') as f:
    for line in f:
        command = line.strip().split()
```

```
        if command[0] == '+':
            x = int(command[1])
            bst.insert(x)
        elif command[0] == '>':
            x = int(command[1])
            result = bst.find_min_greater_than(x)
            results.append(result if result is not None else 0)

# Запись результата в файл
with open('output.txt', 'w') as f:
    for result in results:
        f.write(str(result) + '\n')
```

Входные данные (input.txt):

+ 1

+ 3

+ 3

> 1

> 2

> 3

+ 2

> 1

Выходные данные (output.txt):

3

3

0

2

Описание проведенных тестов:

- Входные данные содержат команды для добавления значений в дерево и запросы для поиска минимального элемента, который больше указанного.
- Проверялись различные случаи, включая:
    - Вставку дубликатов (значение `3` было добавлено дважды).
    - Поиск элементов, которые больше заданного, и обработка случая, когда не найдено ни одного элемента, удовлетворяющего условию.

Выводы:

- Все операции выполняются корректно, включая добавление элементов и выполнение запросов.
- Входные и выходные данные соответствуют ожиданиям, что подтверждает правильность реализации алгоритма.

Задача N11

Описание задания: Реализовать AVL-дерево с возможностью выполнения операций вставки, удаления, проверки существования элемента, поиска следующего и предыдущего элементов.

Исходный код:

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.height = 1
        self.left = None
        self.right = None


class AVLTree:
    def get_height(self, node):
        if not node:
            return 0
        return node.height

    def get_balance(self, node):
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    def right_rotate(self, z):
```

```
        y = z.left
        T3 = y.right

        y.right = z
        z.left = T3

        z.height = max(self.get_height(z.left), self.get_height(z.right))
+ 1
        y.height = max(self.get_height(y.left), self.get_height(y.right))
+ 1

        return y

    def left_rotate(self, z):
        y = z.right
        T2 = y.left

        y.left = z
        z.right = T2

        z.height = max(self.get_height(z.left), self.get_height(z.right))
+ 1
        y.height = max(self.get_height(y.left), self.get_height(y.right))
+ 1

        return y

    def insert(self, node, key):
        if not node:
            return AVLNode(key)
        elif key < node.key:
            node.left = self.insert(node.left, key)
        elif key > node.key:
            node.right = self.insert(node.right, key)
        else:
            return node

        node.height = max(self.get_height(node.left),
self.get_height(node.right)) + 1
        balance = self.get_balance(node)

        if balance > 1 and key < node.left.key:
            return self.right_rotate(node)
        if balance < -1 and key > node.right.key:
            return self.left_rotate(node)
        if balance > 1 and key > node.left.key:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
        if balance < -1 and key < node.right.key:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def min_value_node(self, node):
        if node is None or node.left is None:
            return node
        return self.min_value_node(node.left)

    def delete(self, node, key):
        if not node:
```

```
            return node
        elif 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
            elif node.right is None:
                return node.left
            temp = self.min_value_node(node.right)
            node.key = temp.key
            node.right = self.delete(node.right, temp.key)

        node.height = max(self.get_height(node.left),
self.get_height(node.right)) + 1
        balance = self.get_balance(node)

        if balance > 1 and self.get_balance(node.left) >= 0:
            return self.right_rotate(node)
        if balance < -1 and self.get_balance(node.right) <= 0:
            return self.left_rotate(node)
        if balance > 1 and self.get_balance(node.left) < 0:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
        if balance < -1 and self.get_balance(node.right) > 0:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def exists(self, node, key):
        if not node:
            return False
        elif key < node.key:
            return self.exists(node.left, key)
        elif key > node.key:
            return self.exists(node.right, key)
        else:
            return True

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

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

```
def main():
    tree = AVLTree()
    root = None
    output = []

    with open("input.txt", "r") as f:
        for line in f:
            parts = line.strip().split()
            command = parts[0]
            if command == "insert":
                x = int(parts[1])
                root = tree.insert(root, x)
            elif command == "delete":
                x = int(parts[1])
                root = tree.delete(root, x)
            elif command == "exists":
                x = int(parts[1])
                output.append("true" if tree.exists(root, x) else
"false")
            elif command == "next":
                x = int(parts[1])
                successor = tree.next(root, x)
                output.append(str(successor.key) if successor else
"none")
            elif command == "prev":
                x = int(parts[1])
                predecessor = tree.prev(root, x)
                output.append(str(predecessor.key) if predecessor else
"none")

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


if __name__ == "__main__":
    main()
```

Входные данные (input.txt):

insert 2

insert 5

insert 3

exists 2

exists 4

next 4

prev 4

delete 5

next 4

prev 4

Выходные данные (output.txt):

true

false

5

3

none

3

Описание проведенных тестов:

- Входные данные содержат команды для работы с AVL-деревом, включая вставку, удаление, проверку существования элемента и нахождение следующего/предыдущего элемента.
- Проверялись разные сценарии, такие как:
  - Вставка нескольких элементов, включая проверку существования.
  - Поиск следующего и предыдущего элементов с учетом границ дерева.
  - Удаление элемента и проверка корректности последующих операций.

Выводы:

- Все операции выполняются корректно, что подтверждает правильность реализации алгоритма AVL-дерева.
- Входные и выходные данные соответствуют ожиданиям, что указывает на успешную обработку команд.

Задача N12

Описание задания: Реализовать программу, которая читает описание бинарного дерева и вычисляет баланс каждого узла, используя индексы для представления структуры дерева.

Исходный код:

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left_index = None
        self.right_index = None


def calculate_balance_and_height(node, nodes):
    if node is None:
        return 0, 0  # высота и баланс для пустого узла

    # Рекурсивно вычисляем высоты левого и правого поддерева
    left_height, _ = calculate_balance_and_height(nodes[node.left_index -
1], nodes) if node.left_index else (0, 0)
    right_height, _ = calculate_balance_and_height(nodes[node.right_index
- 1], nodes) if node.right_index else (0, 0)

    # Вычисляем баланс узла
    balance = right_height - left_height

    # Возвращаем высоту и баланс
    return max(left_height, right_height) + 1, balance


# Чтение данных
nodes = []

with open('input.txt', 'r') as f:
    N = int(f.readline().strip())

    for _ in range(N):
        K, L, R = map(int, f.readline().strip().split())
        node = TreeNode(K)
        node.left_index = L
        node.right_index = R
        nodes.append(node)

# Вычисляем баланс для каждого узла и сохраняем его
balances = []
for i in range(N):
    _, balance = calculate_balance_and_height(nodes[i], nodes)
    balances.append(balance)

# Запись результата в файл
with open('output.txt', 'w') as f:
    for balance in balances:
        f.write(str(balance) + '\n')
```

Входные данные (input.txt):

6

-2 0 2

8 4 3

9 0 0

3 6 5

6 0 0

0 0 0

Выходные данные (output.txt):

3

-1

0

0

0

0

Описание проведенных тестов:

- Программа читает данные о бинарном дереве, где каждый узел представлен с помощью ключа и индексов на левое и правое поддеревья.
- Для каждого узла рассчитывается баланс, который определяется как разница между высотами правого и левого поддеревьев.
- Проверялись различные структуры дерева, включая листья и узлы с одним поддеревом.

Выводы:

- Программа корректно вычисляет баланс для каждого узла в бинарном дереве.
- Входные и выходные данные соответствуют ожиданиям, что подтверждает правильность работы алгоритма.