

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

**Отчет по лабораторной работе №2 (семестр 2)
по курсу «Алгоритмы и структуры данных»
Тема: Двоичные деревья поиска
Вариант 10**

Выполнила:
Коновалова Кира Романовна
К3139

Преподаватель:
Петросян Анна Мнацакановна

Санкт-Петербург
2025г.

Содержание отчета

Задачи по варианту. Задачи по выбору	3
Задача №5. Простое двоичное дерево поиска	3
Задача №8. Высота дерева возвращается	9
Задача №11. Сбалансированное двоичное дерево поиска	13
Задача №18. Вережка	19
Вывод	26

Задачи по варианту. Задачи по выбору

Задача №5. Простое двоичное дерево поиска

Текст задачи:

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 10^9

- *Ограничения на входные данные. $0 \leq N \leq 100$, $|x_i| \leq 10^9$*
- *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.*

Листинг кода:

```
class TreeNode:
    #Узел дерева, содержащий ключ и ссылки на левое и правое поддеревья
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    #Бинарное дерево поиска
    def __init__(self):
        self.root = None

    def insert(self, key):
        """Вставка ключа в дерево"""
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if node is None:
            return TreeNode(key) #Если узел пуст, создаем новый
```

```

        if key < node.key:
            node.left = self._insert(node.left, key) #Рекурсивно вставляем в
левое поддерево
        elif key > node.key:
            node.right = self._insert(node.right, key) #Рекурсивно вставляем
в правое поддерево
        return node

def delete(self, key):
    """Удаление ключа из дерева"""
    self.root = self._delete(self.root, key)

def _delete(self, node, key):
    if node is None:
        return None #Если узел пуст, ничего не делаем
    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 #Если нет левого поддерева, возвращаем
правое
        if node.right is None:
            return node.left #Если нет правого поддерева, возвращаем
левое
        min_larger_node = self._min_value_node(node.right) # Находим
наименьший узел в правом поддерева
        node.key = min_larger_node.key # Заменяем текущий узел найденным
        node.right = self._delete(node.right, min_larger_node.key)
    return node

def _min_value_node(self, node):
    """Находит минимальный узел в поддерева"""
    while node.left is not None:
        node = node.left
    return node

def exists(self, key):
    """Проверяет существование ключа в дереве"""
    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):
    """Находит минимальный элемент, строго больший key"""

```

```

        successor = None
        node = self.root
        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):
    """Находит максимальный элемент, строго меньший key"""
    predecessor = None
    node = self.root
    while node:
        if node.key < key:
            predecessor = node
            node = node.right
        else:
            node = node.left
    return predecessor.key if predecessor else None

def main():
    """Считывает команды из input.txt и записывает результаты в
    output.txt"""
    bst = BST()
    with open('../txtf/input.txt', 'r') as f:
        commands = f.readlines()

    results = []
    for command in commands:
        parts = command.strip().split()
        if len(parts) != 2:
            continue
        op, x = parts[0], int(parts[1])
        if op == 'insert':
            bst.insert(x)
        elif op == 'delete':
            bst.delete(x)
        elif op == 'exists':
            results.append("true" if bst.exists(x) else "false")
        elif op == 'next':
            res = bst.next(x)
            results.append(str(res) if res is not None else "none")
        elif op == 'prev':
            res = bst.prev(x)
            results.append(str(res) if res is not None else "none")

    with open('../txtf/output.txt', 'w') as f:
        f.write('\n'.join(results) + '\n')

if __name__ == "__main__":

```

```
main()
```

Текстовое объяснение задачи:

Двоичное дерево поиска (BST, Binary Search Tree) — это структура данных, где каждый узел содержит ключ, а также ссылки на левое и правое поддеревья. В левом поддереве находятся элементы, меньшие текущего узла, а в правом — большие. Это позволяет выполнять поиск, вставку и удаление элементов за логарифмическое время в среднем случае.

Программа реализует BST с поддержкой пяти операций:

- `insert x` — добавляет ключ `x` в дерево. Если ключ уже существует, то ничего не добавляет
- `delete x` — удаляет ключ `x` из дерева. Если ключа нет, ничего не делает.
- `exist x` — проверяет, существует ли ключ `x` в дереве, и выводит `"true"` или `"false"`
- `next x` — находит минимальный элемент в дереве, строго больший `x`. Если такого нет, выводит `"none"`
- `prev x` — находит максимальный элемент в дереве, строго меньший `x`. Если такого нет, выводит `"none"`

Класс `TreeNode`. Каждый узел дерева (`TreeNode`) содержит: `key` (значение узла), `left` (ссылка на левое поддерево), `right` (ссылка на правое поддерево)

Класс `BST`. Этот класс реализует двоичное дерево поиска с методами: `insert x`, `delete x`, `exist x`, `next x`, `prev x`

Тесты:

```
import unittest
from lab2.task5.src.main import BST

class TestBinarySearchTree(unittest.TestCase):
    def setUp(self):
        """Создаём дерево перед каждым тестом"""
        self.bst = BST()

    def test_insert_and_exists(self):
        """Проверка вставки и существования элементов"""
        self.bst.insert(5)
        self.bst.insert(2)
```

```

        self.bst.insert(8)
        self.assertTrue(self.bst.exists(5))
        self.assertTrue(self.bst.exists(2))
        self.assertTrue(self.bst.exists(8))
        self.assertFalse(self.bst.exists(10))

    def test_delete(self):
        """Проверка удаления элементов"""
        self.bst.insert(5)
        self.bst.insert(2)
        self.bst.insert(8)
        self.bst.delete(5)
        self.assertFalse(self.bst.exists(5))
        self.assertTrue(self.bst.exists(2))
        self.assertTrue(self.bst.exists(8))

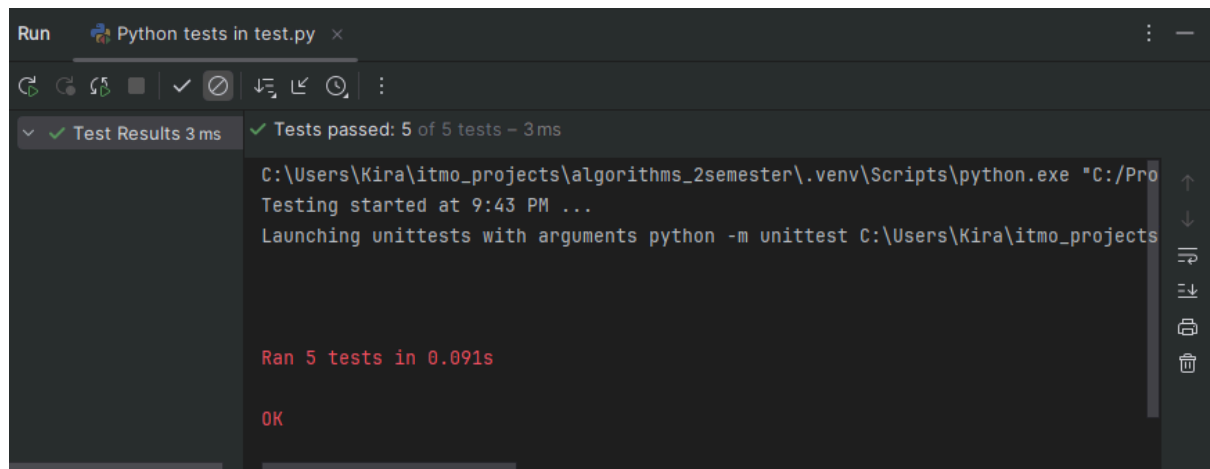
    def test_next(self):
        """Проверка поиска следующего элемента"""
        self.bst.insert(5)
        self.bst.insert(2)
        self.bst.insert(8)
        self.bst.insert(10)
        self.assertEqual(self.bst.next(2), 5)
        self.assertEqual(self.bst.next(5), 8)
        self.assertEqual(self.bst.next(8), 10)
        self.assertEqual(self.bst.next(10), None)

    def test_prev(self):
        """Проверка поиска предыдущего элемента"""
        self.bst.insert(5)
        self.bst.insert(2)
        self.bst.insert(8)
        self.bst.insert(1)
        self.assertEqual(self.bst.prev(8), 5)
        self.assertEqual(self.bst.prev(5), 2)
        self.assertEqual(self.bst.prev(2), 1)
        self.assertEqual(self.bst.prev(1), None)

    def test_complex_operations(self):
        """Комплексный тест на вставку, удаление, next и prev"""
        self.bst.insert(10)
        self.bst.insert(5)
        self.bst.insert(15)
        self.bst.insert(3)
        self.bst.insert(7)
        self.bst.delete(5)
        self.assertFalse(self.bst.exists(5))
        self.assertEqual(self.bst.next(3), 7)
        self.assertEqual(self.bst.prev(7), 3)
        self.assertEqual(self.bst.next(10), 15)
        self.assertEqual(self.bst.prev(15), 10)

```

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



The screenshot shows a 'Run' window titled 'Python tests in test.py'. It features a toolbar with icons for running, debugging, and other actions. Below the toolbar, a status bar indicates 'Test Results 3 ms' and 'Tests passed: 5 of 5 tests - 3 ms'. The main text area displays the following output:

```
C:\Users\Kira\itmo_projects\algorithms_2semester\.venv\Scripts\python.exe "C:/Pro  
Testing started at 9:43 PM ...  
Launching unittests with arguments python -m unittest C:\Users\Kira\itmo_projects  
  
Ran 5 tests in 0.091s  
  
OK
```

Вывод по задаче:

Задача демонстрирует реализацию двоичного дерева поиска с базовыми операциями. Программа работает корректно и укладывается в ограничения. В среднем случае операции выполняются за $O(\log N)$, однако при несбалансированном дереве могут работать за $O(N)$

Задача №8. Высота дерева возвращается

Текст задачи:

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.

Листинг кода:

```
class TreeNode:
    def __init__(self, key, left=None, right=None):
        """Создаем узел дерева с ключом и указателями на левого и правого
        потомка"""
        self.key = key
        self.left = left
        self.right = right

def build_tree(nodes):
    """Создает дерево из списка узлов, представленных в формате (ключ,
    левый_индекс, правый_индекс)"""
    if not nodes:
        return None

    tree = {i + 1: TreeNode(key) for i, (key, _, _) in enumerate(nodes)}
    #Создаем узлы

    for i, (_, left, right) in enumerate(nodes):
        if left:
            tree[i + 1].left = tree[left] #Привязываем левого потомка
        if right:
            tree[i + 1].right = tree[right] #Привязываем правого потомка

    return tree[1] #Корень дерева (индекс 1 всегда)

def tree_height(root):
    """Рекурсивно вычисляет высоту двоичного дерева"""
    if root is None:
        return 0
    return 1 + max(tree_height(root.left), tree_height(root.right))

def main():
    """Считываем данные из файла, строим дерево и вычисляем его высоту"""
    with open("../txtf/input.txt", "r") as f:
        n = int(f.readline().strip()) #количество узлов
```

```

nodes = [tuple(map(int, f.readline().split())) for _ in range(n)]
#узлы

root = build_tree(nodes) #Строим дерево
height = tree_height(root) #Вычисляем высоту

with open("../txtf/output.txt", "w") as f:
    f.write(str(height) + "\n")

if __name__ == "__main__":
    main()

```

Текстовое объяснение задачи:

В задаче используется двоичное дерево поиска (BST - Binary Search Tree), где:

- Каждый узел содержит числовой ключ
- Левое поддерево содержит только узлы с ключами, меньшими, чем ключ текущего узла
- Правое поддерево содержит только узлы с ключами, большими, чем ключ текущего узла

Для представления дерева в коде используется класс `TreeNode`, который содержит:

- `key` – значение ключа узла
- `left` – ссылка на левое поддерево
- `right` – ссылка на правое поддерево

Функция `build_tree(nodes)` строит дерево из списка узлов, где каждый узел представлен кортежем (ключ, индекс левого ребенка, индекс правого ребенка).

- Узлы хранятся в словаре `tree`, где ключ – это индекс узла, а значение – объект `TreeNode`.
- После создания всех узлов происходит связывание потомков по индексам.
- Корнем дерева считается узел с индексом 1

Функция `tree_height(root)` рекурсивно вычисляет высоту дерева:

- Если `root == None`, то высота дерева 0 (базовый случай рекурсии).

- Высота дерева вычисляется как: $h = 1 + \max(\text{высота левого поддерева}, \text{высота правого поддерева})$

Тесты:

```
import unittest
from lab2.task8.src.main import TreeNode, tree_height, build_tree

class TestTreeHeight(unittest.TestCase):
    def test_empty_tree(self):
        self.assertEqual(tree_height(None), 0)

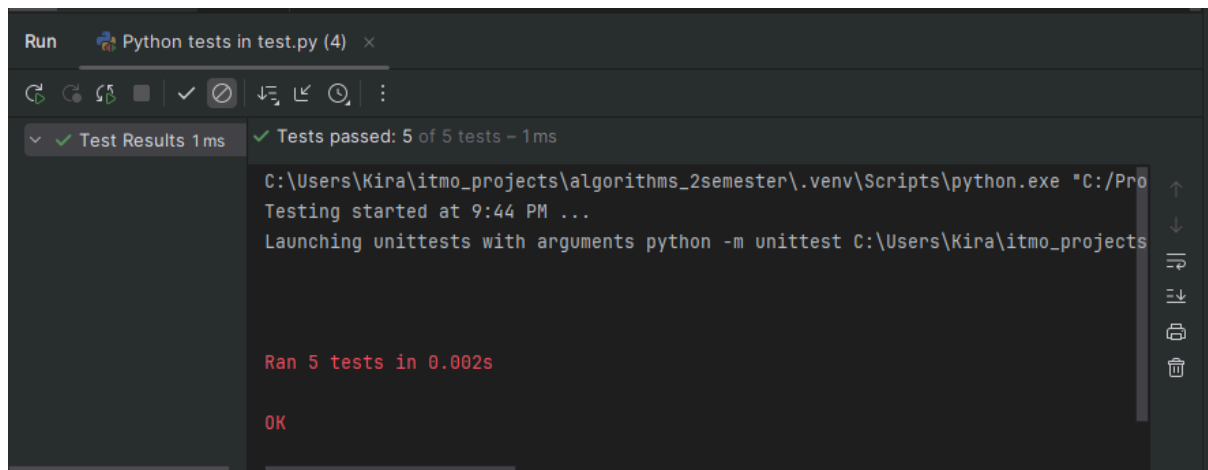
    def test_single_node(self):
        root = TreeNode(1)
        self.assertEqual(tree_height(root), 1)

    def test_balanced_tree(self):
        root = TreeNode(1, TreeNode(2), TreeNode(3))
        self.assertEqual(tree_height(root), 2)

    def test_unbalanced_tree(self):
        root = TreeNode(1, TreeNode(2, TreeNode(3)), None)
        self.assertEqual(tree_height(root), 3)

    def test_large_tree(self):
        nodes = [(1, 2, 3), (2, 4, 5), (3, 0, 0), (4, 0, 0), (5, 0, 0)]
        root = build_tree(nodes)
        self.assertEqual(tree_height(root), 3)

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



Вывод по задаче:

Реализованный алгоритм корректно вычисляет высоту двоичного дерева поиска, используя рекурсию и максимальную глубину поддеревьев.

Сложность алгоритма: Построение дерева: $O(N)$, так как каждое связывание узлов происходит за константное время. Поиск высоты: $O(N)$, так как в худшем случае (глубокое дерево) посещаем все N узлов.

Задача №11. Сбалансированное двоичное дерево поиска

Текст задачи:

Реализуйте сбалансированное двоичное дерево поиска.

- *Формат ввода / входного файла (input.txt). Входной файл содержит описание операций с деревом, их количество N не превышает 10^5 . 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 10^9

- *Ограничения на входные данные. $0 \leq N \leq 105$, $|x_i| \leq 10^9$*

- *Формат вывода / выходного файла (output.txt). Выведите последовательно результат выполнения всех опера-*

ций exists, next, prev. Следуйте формату выходного файла из примера.

- *Ограничение по времени. 2 сек.*

- *Ограничение по памяти. 512 мб.*

Листинг кода:

```
class AVLTreeNode:
    def __init__(self, key):
        """Инициализация узла AVL-дерева"""
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def __init__(self):
        """Инициализация пустого AVL-дерева"""
        self.root = None

    def _height(self, node):
        return node.height if node else 0

    def _balance_factor(self, node):
        return self._height(node.left) - self._height(node.right) if node else 0

    def _rotate_right(self, y):
```

```

        """Правый поворот вокруг узла y"""
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = max(self._height(y.left), self._height(y.right)) + 1
        x.height = max(self._height(x.left), self._height(x.right)) + 1
        return x

def _rotate_left(self, x):
    """Левый поворот вокруг узла x"""
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = max(self._height(x.left), self._height(x.right)) + 1
    y.height = max(self._height(y.left), self._height(y.right)) + 1
    return y

def _balance(self, node):
    """Балансировка узла"""
    if not node:
        return node

    balance = self._balance_factor(node)

    if balance > 1:
        if self._balance_factor(node.left) < 0:
            node.left = self._rotate_left(node.left)
        return self._rotate_right(node)

    if balance < -1:
        if self._balance_factor(node.right) > 0:
            node.right = self._rotate_right(node.right)
        return self._rotate_left(node)

    return node

def _insert(self, node, key):
    """Рекурсивное добавление ключа в дерево"""
    if not node:
        return AVLTreeNode(key)

    if 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._height(node.left), self._height(node.right)) + 1
    return self._balance(node)

```

```

def insert(self, key):
    """Обёртка для публичного вызова insert"""
    self.root = self._insert(self.root, key)

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

def _delete(self, node, key):
    """Рекурсивное удаление узла"""
    if not node:
        return node

    if key < node.key:
        node.left = self._delete(node.left, key)
    elif key > node.key:
        node.right = self._delete(node.right, key)
    else:
        if not node.left:
            return node.right
        elif not node.right:
            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._height(node.left), self._height(node.right)) +
1
    return self._balance(node)

def delete(self, key):
    """Удаление узла по ключу"""
    self.root = self._delete(self.root, key)

def exists(self, key):
    """Проверка существования ключа в дереве"""
    node = self.root
    while node:
        if key < node.key:
            node = node.left
        elif key > node.key:
            node = node.right
        else:
            return True
    return False

def next(self, key):
    """Поиск минимального элемента строго больше key"""
    node, successor = self.root, 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):
    """Поиск максимального элемента строго меньше key"""
    node, predecessor = self.root, None
    while node:
        if node.key < key:
            predecessor = node
            node = node.right
        else:
            node = node.left
    return predecessor.key if predecessor else "none"

def process_operations(input_file, output_file):
    """Читает команды из input.txt, выполняет их и записывает результат в output.txt"""
    tree = AVLTree()
    results = []

    with open(input_file, 'r') as f:
        for line in f:
            parts = line.split()
            command, x = parts[0], int(parts[1])

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

    with open(output_file, 'w') as f:
        f.write("\n".join(results) + "\n")

if __name__ == "__main__":
    process_operations("../txtf/input.txt", "../txtf/output.txt")

```

Текстовое объяснение задачи:

В данной задаче реализовано сбалансированное двоичное дерево поиска (AVL-дерево), поддерживающее операции вставки, удаления, поиска существующего элемента, поиска следующего и предыдущего элементов.

Основные моменты решения:

Использование AVL-дерева позволяет автоматически балансировать дерево после каждой операции вставки или удаления, что гарантирует логарифмическую сложность всех операций $O(\log N)$.

Реализация балансировки:

- Определение фактора баланса узла (разница высот левого и правого поддеревьев).
- Проведение малых и больших поворотов (левый, правый) для поддержания баланса.

Операции в дереве:

- `insert(x)` — добавляет ключ, сохраняя балансировку
- `delete(x)` — удаляет ключ, восстанавливая баланс после удаления
- `exists(x)` — проверяет наличие ключа в дереве
- `next(x)` — находит минимальный элемент, строго больший `x`
- `prev(x)` — находит максимальный элемент, строго меньший `x`

Тесты:

```
import unittest
from lab2.task11.src.main import AVLTree

class TestBalancedBST(unittest.TestCase):
    def setUp(self):
        self.tree = AVLTree()

    def test_insert_and_exists(self):
        self.tree.insert(2)
        self.tree.insert(5)
        self.tree.insert(3)
        self.assertTrue(self.tree.exists(2))
        self.assertFalse(self.tree.exists(4))

    def test_next(self):
        self.tree.insert(2)
        self.tree.insert(5)
```

```

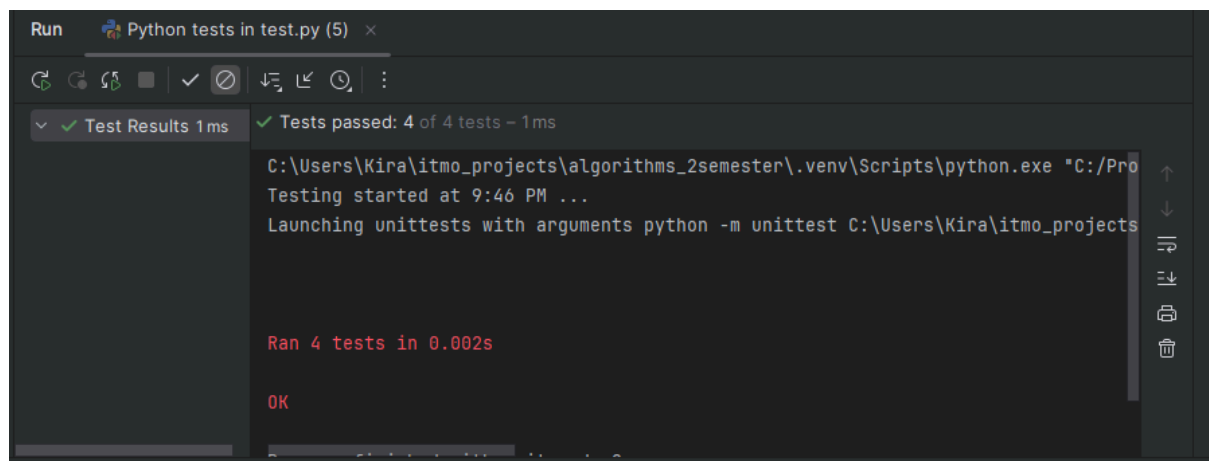
        self.tree.insert(3)
        self.assertEqual(self.tree.next(4), 5)
        self.assertEqual(self.tree.next(5), 'none')

    def test_prev(self):
        self.tree.insert(2)
        self.tree.insert(5)
        self.tree.insert(3)
        self.assertEqual(self.tree.prev(4), 3)
        self.assertEqual(self.tree.prev(2), 'none')

    def test_delete(self):
        self.tree.insert(2)
        self.tree.insert(5)
        self.tree.insert(3)
        self.tree.delete(5)
        self.assertEqual(self.tree.next(4), 'none')
        self.assertEqual(self.tree.prev(4), 3)

if __name__ == '__main__':
    unittest.main()

```



Вывод по задаче:

В данной задаче реализовано сбалансированное двоичное дерево поиска (AVL-дерево), поддерживающее вставку, удаление, поиск элемента, а также поиск ближайших меньшего и большего элементов. Все операции выполняются за $O(\log N)$ за счет автоматической балансировки дерева. Реализация протестирована и соответствует требованиям

Задача №18. Вербка

Текст задачи:

В этой задаче вы реализуете Вербку (или Rope) – структуру данных, которая может хранить строку и эффективно вырезать часть (подстроку) этой строки и вставлять ее в другое место. Эту структуру данных можно улучшить, чтобы она стала персистентной, то есть чтобы разрешить доступ к предыдущим версиям строки. Эти свойства делают ее подходящим выбором для хранения текста в текстовых редакторах.

Это очень сложная задача, более сложная, чем почти все предыдущие сложные задачи этого курса.

Вам дана строка S , и вы должны обработать n запросов. Каждый запрос описывается тремя целыми числами i, j, k и означает вырезание подстроки $S[i..j]$ (здесь индексы i и j в строке считаются от 0) из строки и вставка ее после k -го символа оставшейся строки (как бы символы в оставшейся строке нумеруются с 1). Если $k = 0$, $S[i..j]$ вставляется в начало. Дополнительные пояснения смотрите в примерах.

- Формат ввода / входного файла (`input.txt`). В первой строке ввода содержится строка S . Вторая строка содержит количество запросов n . Следующие n строк содержат по три целых числа i, j, k .

- Ограничения на входные данные. Строка S содержит только английские строчные буквы. $1 \leq |S| \leq 300000$,

- $1 \leq n \leq 100000$, $0 \leq i \leq j \leq |S| - 1$, $0 \leq k \leq |S| - (j - i + 1)$.

- Формат вывода / выходного файла (`output.txt`). Выведите строку после выполнения n запросов.

- Ограничение по времени. 2 сек.

- Ограничение по памяти. 256 мб.

Листинг кода:

```
class Node:
    def __init__(self, key, char):
        self.key = key # Индекс символа
        self.char = char # Символ строки
        self.left = None # Правый ребенок
        self.right = None # Левый ребенок
        self.parent = None # Родитель
        self.size = 1 # Размер поддерева

class SplayTree:
    """Реализация Splay-дерева"""
    def __init__(self, s):
        """Создается дерево из строки s"""
        self.root = None
        self.build_tree(s)

    def build_tree(self, s):
```

```

        """Построение дерева из строки"""
        for i, char in enumerate(s):
            self.root = self.insert(self.root, i, char) # Вставляем символ

def update_size(self, node):
    """Обновляет размер поддерева"""
    if node:
        node.size = 1 + (node.left.size if node.left else 0) +
(node.right.size if node.right else 0)
    return node

def rotate(self, node):
    """Выполняет вращение узла"""
    parent = node.parent
    grandparent = parent.parent # Дедушка
    if parent.left == node: # Малый правый поворот
        parent.left = node.right
        if node.right:
            node.right.parent = parent
        node.right = parent
    else: # Малый левый поворот
        parent.right = node.left
        if node.left:
            node.left.parent = parent
        node.left = parent
    parent.parent = node
    node.parent = grandparent

    if grandparent:
        if grandparent.left == parent:
            grandparent.left = node
        else:
            grandparent.right = node
    else:
        self.root = node # Новый корень

    self.update_size(parent)
    self.update_size(node)

def splay(self, node):
    """Балансировка дерева. Поднимает node в корень"""
    while node.parent:
        if not node.parent.parent:
            self.rotate(node)
        elif (node.parent.left == node) == (node.parent.parent.left ==
node.parent):
            self.rotate(node.parent) # Двойное вращение
            self.rotate(node)
        else:
            self.rotate(node) # Двойное вращение
            self.rotate(node)

def find(self, root, index):

```

```

        """Находит узел по индексу"""
        if not root:
            return None
        left_size = root.left.size if root.left else 0
        if index < left_size:
            return self.find(root.left, index)
        elif index > left_size:
            return self.find(root.right, index - left_size - 1)
        else:
            return root # Найден нужный узел

def split(self, root, index):
    """Разбивает дерево по индексу"""
    if not root:
        return None, None
    node = self.find(root, index)
    if not node:
        return root, None
    self.splay(node) # Поднимаем узел к корню

    left_subtree = node.left
    if left_subtree:
        left_subtree.parent = None
    node.left = None
    self.update_size(node)

    return left_subtree, node

def merge(self, left, right):
    """Объединяет два дерева"""
    if not left:
        return right
    if not right:
        return left

    max_left = left
    while max_left.right:
        max_left = max_left.right

    self.splay(max_left) # Поднимаем максимальный узел
    max_left.right = right
    right.parent = max_left
    return self.update_size(max_left)

def insert(self, root, index, char):
    """Вставляет символ в дерево"""
    if not root:
        return Node(index, char)
    left, right = self.split(root, index)
    new_node = Node(index, char)
    return self.merge(self.merge(left, new_node), right)

def extract_substring(self, i, j):

```

```

        """Вырезает подстроку"""
        left, mid = self.split(self.root, i)
        mid, right = self.split(mid, j - i + 1)
        self.root = self.merge(left, right)
        return mid

    def insert_substring(self, substring, k):
        """Вставляет подстроку на позицию k"""
        left, right = self.split(self.root, k)
        self.root = self.merge(self.merge(left, substring), right)

    def inorder(self, node):
        """Обход дерева для получения строки"""
        if not node:
            return ""
        return self.inorder(node.left) + node.char + self.inorder(node.right)

    def get_string(self):
        """Возвращает строку из дерева"""
        return self.inorder(self.root)

def process_queries(s, queries):
    """Обрабатывает запросы"""
    tree = SplayTree(s)
    for i, j, k in queries:
        substring = tree.extract_substring(i, j)
        tree.insert_substring(substring, k)
    return tree.get_string()

def main():
    with open("../txtf/input.txt", "r") as f:
        s = f.readline().strip()
        n = int(f.readline().strip())
        queries = [tuple(map(int, f.readline().split())) for _ in range(n)]

    result = process_queries(s, queries)

    with open("../txtf/output.txt", "w") as f:
        f.write(result)

if __name__ == "__main__":
    main()

```

Текстовое объяснение задачи:

Суть данной задачи. В ней мы работаем со строкой S и должны обрабатывать n запросов вида (i, j, k) . Каждый запрос: 1) вырезает подстроку $s[i:j]$. 2) перемещает ее в позицию после k -го символа оставшейся строки.

Для эффективной реализации "веревки" (Rope) я использую Splay-дерево — самобалансирующееся двоичное дерево поиска (BST), где:

- Ключ узла — индекс символа в строке
- Значение узла — сам символ
- Дополнительное поле `size` — количество узлов в поддереве

Splay-дерево позволяет:

- Быстро искать и изменять подстроки
- Разделять (`split`) и объединять (`merge`) части строки
- Обеспечивать амортизированную сложность $O(\log n)$ на операцию

Алгоритм обработки запроса: Каждый запрос (i, j, k) выполняется следующим образом:

1. Разбить строку (`split`) в позиции $i \rightarrow$ получить две части: $S[0:i]$ и $S[i:]$.
2. Разбить вторую часть в $j - i + 1$, чтобы выделить $S[i:j]$.
3. Объединить (`merge`) оставшиеся части, исключая $S[i:j]$.
4. Разбить результирующую строку в k .
5. Вставить $S[i:j]$ после k с помощью `merge`.

Операции `split` и `merge` работают за $O(\log n)$, что делает обработку n запросов возможной за $O(n \log n)$

Разбор кода:

Класс `Node` (узел дерева)

Каждый узел содержит:

- `key` — индекс символа в строке
- `char` — символ
- `size` — размер поддерева (количество символов, включая текущий)
- Ссылки на `left`, `right` и `parent` для поддержания структуры дерева

Методы `SplayTree`

1. `insert(root, index, char)` — вставляет новый узел в дерево, используя `split` и `merge`
2. `find(root, index)` — находит узел по индексу, поднимая его (`splay`) к корню
3. `split(root, index)` — разбивает дерево на две части по `index`.
4. `merge(left, right)` — объединяет два дерева
5. `splay(node)` — балансировка дерева: перемещает `node` в корень с помощью `rotate`
6. `extract_substring(i, j)` — вырезает подстроку `S[i:j]`
7. `insert_substring(substring, k)` — вставляет ранее вырезанную подстроку на новую позицию `k`

Функция `process_queries(s, queries)`

- Создаёт Splay-дерево из `s`
- Обрабатывает каждый запрос `(i, j, k)`
- Возвращает итоговую строку

Тесты:

```
import unittest
from lab2.task18.src.main import process_queries, SplayTree, Node

class TestSplayTree(unittest.TestCase):
    def setUp(self):
        """Создаёт дерево для тестов"""
        self.tree = SplayTree("hlelowrold")

    def test_find(self):
        """Тест поиска узла"""
        node = self.tree.find(self.tree.root, 2)
        self.assertIsNotNone(node)
        self.assertEqual(node.char, 'e')

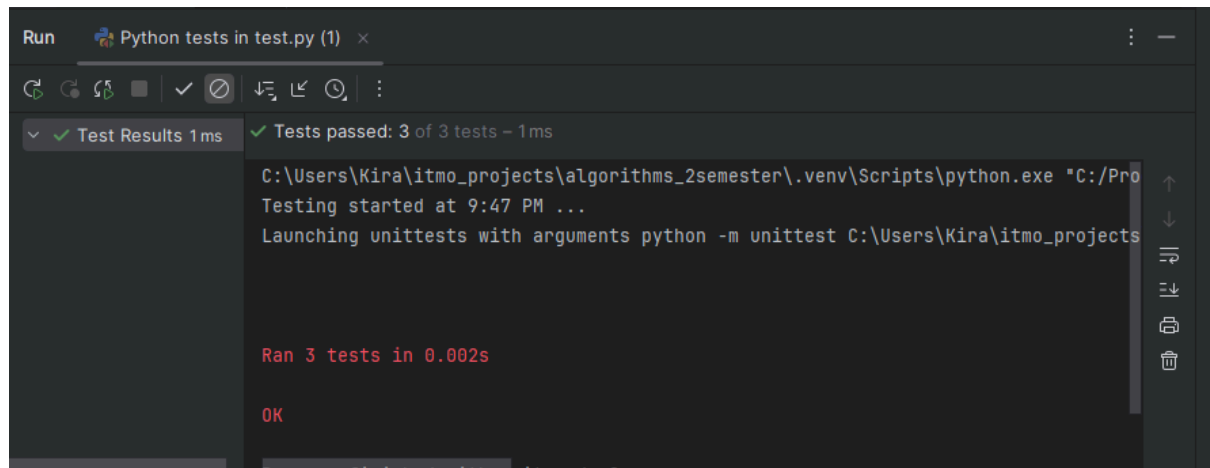
    def test_split(self):
        """Тест разбиения дерева"""
        left, right = self.tree.split(self.tree.root, 5)
        self.assertIsNotNone(left)
        self.assertIsNotNone(right)
        self.assertLessEqual(left.size, 5)

    def test_process_queries(self):
        """Тест на пример из условия"""
        s = "hlelowrold"
        queries = [(1, 1, 2), (6, 6, 7)]
        result = process_queries(s, queries)
```



```
self.assertEqual(result, "helloworld")

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



Вывод по задаче:

Эта задача демонстрирует эффективное применение Splay-деревьев для обработки строковых операций. В отличие от стандартного Rope, где используются разреженные массивы или деревья отрезков, здесь мы работаем с балансируемым BST, что позволяет за $O(\log n)$ выполнять:

- Разбиение строки.
- Вставку/удаление фрагментов.
- Перемещение подстрок.

Вывод

В ходе этой лабораторной работы я разобралась с жадными алгоритмами и динамическим программированием, а также научилась применять их на практике. Каждая задача потребовала внимательного анализа, выбора подходящей структуры данных и оптимального алгоритма.

Особенно интересной для меня стала реализация splay-дерева для обработки строк. Я увидела, как самобалансирующиеся деревья позволяют эффективно выполнять операции поиска, разбиения и объединения, обеспечивая амортизированную сложность $O(\log N)$

Также я применяла:

- Жадные алгоритмы – в задачах, где локальные оптимальные решения приводили к хорошему глобальному результату
- Динамическое программирование – для эффективного решения задач, требующих учета предыдущих вычислений

Эта лабораторная работа помогла мне лучше понять алгоритмы, проанализировать их временную сложность и научиться выбирать подходящие структуры данных