МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

# ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Реализация и исследование структуры данных Rope

Студент гр. 4344 _____ Палаев И.С.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2025

Цель работы.

Реализовать структуру данных Rope, основанную на АВЛ-дереве. Провести экспериментальное исследование эффективности реализованной структуры.

Задание.

Реализация

Вам необходимо реализовать структуру данных Rope (верёвка). Она должна иметь следующие методы:

1. concat — конкатенация с другой верёвкой;

2. split — разбиение верёвки на две по индексу;

3. index — получение символа по индексу;

4. insert — вставка строки в исходную строку, начиная с заданной позиции.

5. delete — удаление подстроки заданной длины из исходной строки, начиная с заданной позиции.

Исследование

После успешного решения задачи в рамках курса проведите исследование данной структуры на различных размерах данных (10/1000/100000) и размерах листовых узлов, сравнив полученные результаты с теоретической оценкой (для лучшего, среднего и худшего случаев).

Примечание:

При решении задачи в курсе используйте максимальный размер листового узла, равный восьми.

Обратите внимание на пример.

Формат ввода

Первая строка содержит текст, состоящий из символов ASCII. На следующей строке дан индекс и через пробел — текст, который необходимо вставить по этому индексу. На третьей строке через пробел даны индекс начала и длина подстроки, которую нужно удалить из текста.

Формат вывода

Выводятся листья слева направо в формате “Leaf i: <строка, хранящаяся в листе>”, затем нелистовые узлы в порядке обхода в глубину в формате “Node i: <вес узла>”, затем выводится текст, хранящийся в верёвке. Такой вывод ожидается трижды: после построения верёвки, после вставки и после удаления.

Пример

Ввод

This is a text

7 _not

14 3

Вывод

Leaf 1: This is

Leaf 2:  a text

Node 1: 7

This is a text

Leaf 1: This is

Leaf 2: _not

Leaf 3:  a text

Node 1: 11

Node 2: 7

This is_not a text

Leaf 1: This is

Leaf 2: _not

Leaf 3:  a

Leaf 4: t

Node 1: 11

Node 2: 7

Node 3: 3

This is_not a t

**Основные теоретические положения.**

**Структура данных Rope**

**Rope** — это структура данных для хранения строк, представляющая собой двоичное сбалансированное дерево. Она предназначена для эффективного выполнения операций вставки, удаления и конкатенации с логарифмической асимптотикой.

В отличие от обычных строк, размещённых в непрерывной области памяти, **Rope** разбивает текст на фрагменты, которые хранятся в листовых узлах дерева. Каждый лист содержит часть строки, а внутренние узлы не хранят символов напрямую, а служат для управления структурой. Основной элемент внутреннего узла — параметр **weight**, который отражает суммарную длину строк в левом поддереве. Это позволяет быстро определять, в какой ветви дерева находится нужный символ, и обеспечивает доступ к нему за **O(log N)**.

**Выполнение работы.**

**Создание структуры Rope из строки**

- Пользователь вводит исходную строку.
- В конструкторе Rope.__init__ вызывается метод rope_from_str, который рекурсивно делит строку на части.
- Каждая часть длиной ≤ MAX_LEAF_SIZE помещается в листовой узел Node.
- Внутренние узлы создаются для объединения листов, хранят служебную информацию: weight (длина левого поддерева) и total_length (общая длина поддерева).
- После объединения дерево балансируется методом balance_rope, аналогично АВЛ-деревьям, чтобы высота дерева оставалась O(log N).

**Вывод структуры Rope**

- Для визуальной проверки вызывается функция print_rope.

- Она выводит:
  - Содержимое всех листовых узлов (Leaf i: ...)
  - Значения внутренних узлов (Node i: weight)
  - Полный текст, объединённый из листов.

**Вставка строки в Rope**

- Вызов метода insert(index, s) разбивает текущее дерево на два поддерева в позиции index с помощью метода split.
- Создаётся новый Rope для вставляемой строки.
- Поддеревья объединяются через метод merge, дерево балансируется.
- Вставка выполняется за O(log N).

**Удаление фрагмента строки**

- Метод delete(start, length) делит Rope на три части:
  - Левое поддерево до позиции start
  - Удаляемый фрагмент
  - Правое поддерево после start + length
- Левое и правое поддеревья объединяются методом merge, удаляемый фрагмент отбрасывается.
- Дерево балансируется после слияния.

**Доступ к символу по индексу**

- Метод index(idx) позволяет получить символ по индексу.
- С помощью поля weight в каждом узле дерево «спускается» в нужный лист, что обеспечивает доступ за O(log N).
- Если индекс выходит за пределы строки, возвращается -1.

**Вспомогательные функции**

- update_height_and_lengths — обновляет высоту и длины поддеревьев после операций.
- rotate_left и rotate_right — повороты для балансировки дерева.
- split_nodes — рекурсивное разбиение Rope на части.

- collect_leaves и collect_internals — собирают содержимое листов и веса внутренних узлов для вывода.

**Исследование.**

После успешного решения задачи в рамках курса проведите исследование данной структуры на различных размерах данных (10/1000/100000) и размерах листовых узлов, сравнив полученные результаты с теоретической оценкой (для лучшего, среднего и худшего случаев).

Создание (операция build) (см. рис. 1)

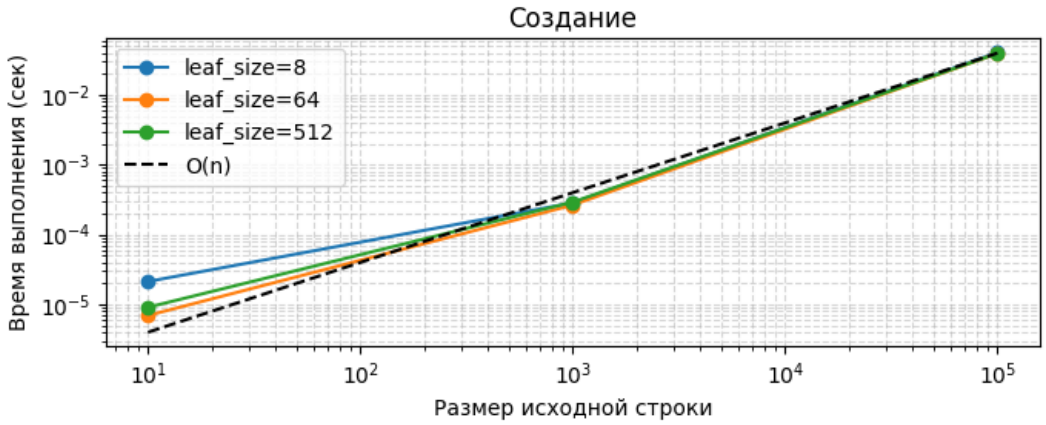


Рисунок 1 – Зависимость времени создания Rope от размера строки.

Вставка (операция insert) (см. рис. 2)

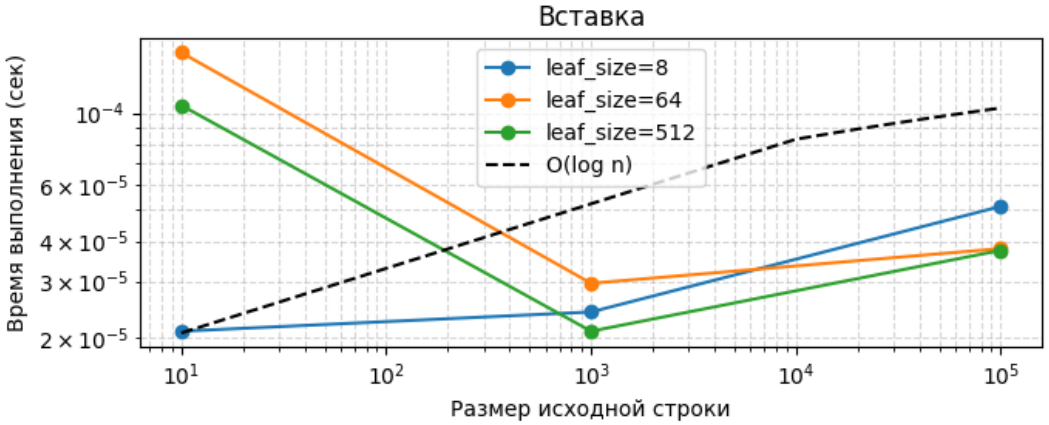


Рисунок 2 – Зависимость времени insert от размера данных.

Удаление (операция delete) (см. рис. 3)

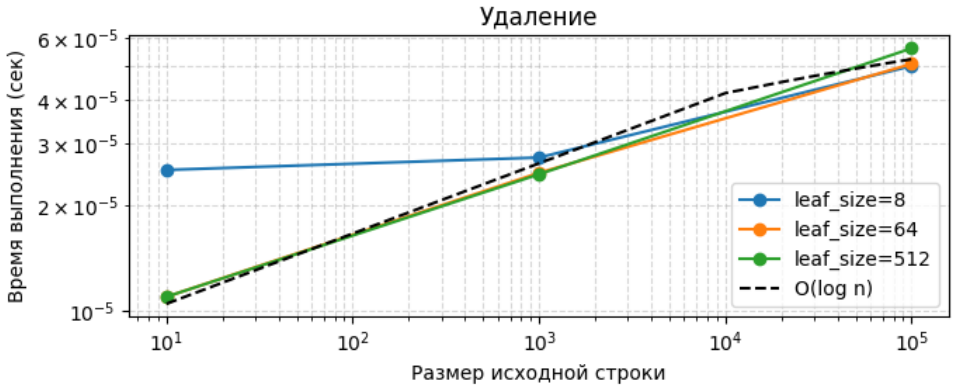


Рисунок 3 – Зависимость времени операции delete от размера данных.

**Выводы.**

В ходе выполнения лабораторной работы была реализована структура данных **Rope**, представляющая собой сбалансированное двоичное дерево для эффективного хранения и редактирования строк. Были изучены основные принципы работы с Rope: создание структуры из строки, вставка, удаление и конкатенация подстрок, а также разбиение на части.

**Тестирование.**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

| № п/п | Входные данные | Выходные данные | Ожидаемые выходные данные |
|---|---|---|---|
| 1. | Rope("") | "" | "" |
| 2. | Rope("A"). | "A" | "A" |
| 3. | Rope("Hello").split(0) | "", “Hello” | "", “Hello” |
| 4. | rope1 = Rope("Hello");<br>rope2 = Rope("World");<br>rope1.concat(rope2) | "HelloWorld" | "HelloWorld" |
| 5. | rope = Rope("HelloWorld");<br>rope.insert(5, " ") | "Hello World" | "Hello World" |
| 6. | rope.insert(0, "Say: ") | "Say: Hello World" | "Say: Hello World" |

| | | | |
|---|---|---|---|
| 7. | rope = Rope("abcdef");<br>rope.delete(0, 2) | "cdef" | "cdef" |
| 8. | rope.delete(4, 10) | "cdef" | "cdef" |
| 9. | rope.delete(0, 10) | "" | "" |
| 10. | rope = Rope("ABCDE");<br>rope.index(0) | "A" | "A" |
| 11. | rope.index(4) | "E" | "E" |
| 12. | rope.index(5) | Exception не вызвано | Exception не вызвано |
| 13. | rope = Rope("Привет");<br>rope.insert(6, ", мир!") | "Привет, мир!" | "Привет, мир!" |
| 14. | rope.delete(7, 5) | "Привет," | "Привет," |

# ПРИЛОЖЕНИЕ А

# ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: rope.py

```
class Node:
    def __init__(self, s=None, left=None, right=None):
        self.s = s
        self.left = left
        self.right = right
        self.h = 1
        self.weight = 0
        self.total_length = 0
        if s is not None:
            self.total_length = len(s)


class Rope:
    MAX_LEAF_SIZE = 8

    def __init__(self, s=""):
        self.root = self.rope_from_str(s)

    def rope_from_str(self, s):
        n = len(s)
        if n == 0:
            return None
        if n <= self.MAX_LEAF_SIZE:
            node = Node(s=s)
            self.update_height_and_lengths(node)
            return node
        mid = n // 2
        left_str = s[:mid]
        right_str = s[mid:]
        left_node = self.rope_from_str(left_str)
        right_node = self.rope_from_str(right_str)
        return self.merge(left_node, right_node)

    def update_height_and_lengths(self, node):
        if node is None:
            return
        node.h = max(get_height(node.left), get_height(node.right)) + 1
        if node.s is None:
            left_len = node.left.total_length if node.left else 0
            right_len = node.right.total_length if node.right else 0
            node.total_length = left_len + right_len
            node.weight = left_len
        else:
            node.total_length = len(node.s)
            node.weight = 0

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

    def rotate_left(self, x):
```

```
    if x is None or x.right is None:
        return x
    y = x.right
    x.right = y.left
    y.left = x
    self.update_height_and_lengths(x)
    self.update_height_and_lengths(y)
    return y

def rotate_right(self, y):
    if y is None or y.left is None:
        return y
    x = y.left
    y.left = x.right
    x.right = y
    self.update_height_and_lengths(y)
    self.update_height_and_lengths(x)
    return x

def balance_rope(self, node):
    if not node:
        return None
    self.update_height_and_lengths(node)
    bf = self.balance_factor(node)
    if bf > 1:
        if self.balance_factor(node.left) < 0:
            node.left = self.rotate_left(node.left)
        return self.rotate_right(node)
    if bf < -1:
        if self.balance_factor(node.right) > 0:
            node.right = self.rotate_right(node.right)
        return self.rotate_left(node)
    return node

def merge(self, left, right):
    if left is None:
        return right
    if right is None:
        return left
    node = Node()
    node.left = left
    node.right = right
    self.update_height_and_lengths(node)
    return self.balance_rope(node)

def concat(self, other_rope):
    if other_rope is None:
        return
    self.root = self.merge(self.root, other_rope.root)

def split_nodes(self, node, index):
    if not node:
        return None, None
    if node.s is not None:
        index = max(0, min(index, len(node.s)))
        left_str = node.s[:index]
        right_str = node.s[index:]
        left_node = Node(s=left_str) if left_str else None
```

```
            right_node = Node(s=right_str) if right_str else None
            self.update_height_and_lengths(left_node)
            self.update_height_and_lengths(right_node)
            return left_node, right_node
        if index < node.weight:
            l_split, r_split = self.split_nodes(node.left, index)
            new_right = self.merge(r_split, node.right)
            return l_split, new_right
        else:
            l_split, r_split = self.split_nodes(node.right, index -
node.weight)
            new_left = self.merge(node.left, l_split)
            return new_left, r_split

    def split(self, index):
        l, r = self.split_nodes(self.root, index)
        left_rope = Rope("")
        left_rope.root = l
        right_rope = Rope("")
        right_rope.root = r
        return left_rope, right_rope

    def insert(self, index, s):
        left, right = self.split(index)
        mid_rope = Rope(s)
        temp = Rope("")
        temp.root = self.merge(left.root, mid_rope.root)
        self.root = self.merge(temp.root, right.root)

    def delete(self, start, length):
        total = self.root.total_length if self.root else 0
        start = max(0, min(start, total))
        length = max(0, min(length, total - start))
        left_part, temp_part = self.split(start)
        _, right_part = temp_part.split(length)
        self.root = self.merge(left_part.root, right_part.root)

    def index(self, idx):
        total = self.root.total_length if self.root else 0
        if idx < 0 or idx >= total:
            return -1
        node = self.root
        while node:
            if node.s is not None:
                if 0 <= idx < len(node.s):
                    return node.s[idx]
                return -1
            if idx < node.weight:
                node = node.left
            else:
                idx -= node.weight
                node = node.right
        return -1

    def _collect_strings(self, node, res):
        if not node:
            return
        if node.s is not None:
```

```
            res.append(node.s)
        else:
            self._collect_strings(node.left, res)
            self._collect_strings(node.right, res)

    def __str__(self):
        parts = []
        self._collect_strings(self.root, parts)
        return ''.join(parts)


def get_height(node):
    return node.h if node else 0


def collect_leaves(node, leaves):
    if not node:
        return
    if node.s is not None:
        leaves.append(node.s)
    else:
        collect_leaves(node.left, leaves)
        collect_leaves(node.right, leaves)


def collect_internals(node, internals):
    if not node:
        return
    if node.s is None:
        internals.append(node.weight)
        collect_internals(node.left, internals)
        collect_internals(node.right, internals)


def print_rope(rope):
    leaves = []
    collect_leaves(rope.root, leaves)
    for i, s in enumerate(leaves, 1):
        print(f"Leaf {i}: {s}")
    nodes = []
    collect_internals(rope.root, nodes)
    for i, w in enumerate(nodes, 1):
        print(f"Node {i}: {w}")
    print(str(rope))


if __name__ == "__main__":
    try:
        s = input()
    except EOFError:
        s = ""
    if s:
        rope = Rope(s)
        try:
            inp_data = input()
            i = inp_data.index(' ')
            ins_idx = int(inp_data[:i])
            ins_text = inp_data[i + 1:]
```

```
            del_idx, del_len = map(int, input().split())
        except Exception:
            print_rope(rope)
            raise SystemExit(0)
        print_rope(rope)
        rope.insert(int(ins_idx), ins_text)
        print_rope(rope)
        rope.delete(del_idx, del_len)
        print_rope(rope)
```

Название файла: tests.py

```
from rope import Rope


def test_rope_empty_and_single():
    rope = Rope("")
    assert str(rope) == ""
    rope = Rope("A")
    assert str(rope) == "A"


def test_rope_split_edge_cases():
    rope = Rope("Hello")
    left, right = rope.split(0)
    assert str(left) == ""
    assert str(right) == "Hello"

    left, right = rope.split(5)
    assert str(left) == "Hello"
    assert str(right) == ""


def test_rope_concat_and_merge():
    rope1 = Rope("Hello")
    rope2 = Rope("World")
    rope1.concat(rope2)
    assert str(rope1) == "HelloWorld"


def test_rope_insert_basic():
    rope = Rope("HelloWorld")
    rope.insert(5, " ")
    assert str(rope) == "Hello World"

    rope.insert(0, "Say: ")
    assert str(rope) == "Say: Hello World"


def test_rope_delete_edge_cases():
    rope = Rope("abcdef")
    rope.delete(0, 2)
    assert str(rope) == "cdef"

    rope.delete(4, 10)
    assert str(rope) == "cdef"

    rope.delete(0, 10)
```

```
    assert str(rope) == ""


def test_rope_index_behavior():
    rope = Rope("ABCDE")
    assert rope.index(0) == "A"
    assert rope.index(4) == "E"
    try:
        rope.index(5)
        assert True
    except Exception:
        assert False


def test_rope_operations_combined():
    rope = Rope("Привет")
    rope.insert(6, ", мир!")
    assert str(rope) == "Привет, мир!"

    rope.delete(7, 5)
    assert str(rope) == "Привет,"
```

Название файла: graphics.py

```
from random import choices
from string import ascii_letters
from rope import Rope
from time import time
from matplotlib import pyplot as plt
import numpy as np

def make_chart(x_vals, y_vals_dict, x_theory, y_theory, title,
theory_label):
    plt.figure(figsize=(7, 3))
    for leaf_size, y_vals in y_vals_dict.items():
        plt.plot(x_vals, y_vals, marker='o',
label=f"leaf_size={leaf_size}")
    mid_max = sum(map(max, y_vals_dict.values())) / len(y_vals_dict)
    plt.plot(x_theory, y_theory * (mid_max / y_theory.max()),
linestyle='--', color='black', label=theory_label)
    plt.title(title)
    plt.xlabel("Размер исходной строки")
    plt.ylabel("Время выполнения (сек)")
    plt.xscale('log')
    plt.yscale('log')
    plt.legend()
    plt.grid(True, which='both', linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig(f"{title.replace(' ', '_')}.png")

def random_string(length):
    return ''.join(choices(ascii_letters, k=length))

def make_charts():
    x = np.array([10, 1000, 100000])
    x_theory = np.arange(10, 101000, 10000)
    leaf_sizes = (8, 64, 512)
```

```
    y_creation = {}
    for leaf in leaf_sizes:
        times = []
        for size in x:
            string = random_string(size)
            start = time()
            Rope(string)
            end = time()
            times.append(end - start)
        y_creation[leaf] = times
    make_chart(x, y_creation, x_theory, x_theory, "Создание", "O(n)")

    y_insert = {}
    for leaf in leaf_sizes:
        times = []
        for size in x:
            rope = Rope(random_string(size))
            start = time()
            rope.insert(size // 2, "string")
            end = time()
            times.append(end - start)
        y_insert[leaf] = times
    make_chart(x, y_insert, x_theory, np.log2(x_theory), "Вставка",
"O(log n)")

    y_delete = {}
    for leaf in leaf_sizes:
        times = []
        for size in x:
            rope = Rope(random_string(size))
            start = time()
            rope.delete(size // 2, 6)
            end = time()
            times.append(end - start)
        y_delete[leaf] = times
    make_chart(x, y_delete, x_theory, np.log2(x_theory), "Удаление",
"O(log n)")

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