Звіт до лабораторної роботи №1
“Застосування поліноміальних хеш-функцій”

Підготував студент групи МІ-31
Гришечкін Тихон

Завдання

Умови лабораторної роботи:

I. Спільна частина.
Реалізувати структуру даних типу «множина рядків». Рядки – непусті послідовності довжиною до 15 символів з рядкових латинських літер. Структура даних повинна підтримувати операції додавання рядку до множини, вилучення та перевірки належності даного рядку множині. Максимальна кількість рядків у множині, що зберігається, не більше 10^6
.
Вхідні дані. Кожен рядок вхідних даних задає одну операцію над мно-жиною. Запис операції складається з типу операції та наступного заним через пробіл ряду, над яким проводять операцію. Типи операції вказують символи: «+» – додавання рядку, «–» – вилучення, «?» –перевірка на належність. Загальні кількість операцій у вхідному файлі не більше 10^6. Список операцій завершується рядком із символом #
(ознака кінця вхідних даних).
При додаванні елементу до множини не гарантується, що він відсутній у множині, а при вилученні елементу з множини не гарантується, що він є у множині.
Вихідні дані. Виводяться для кожної операції типу «?» рядок yes або no, в залежності від того, чи зустрічається дане слово у множині.

1 варіант

II. Частина за варіантами.
Варіант 1. Знайти усі повторювані рядки та поділити їх на групи, щоб у кожній групі виводився повторюваний рядок та кількість його повторювань. Оцінити час виконання.

Виконання

Лабораторну виконано мовою Python, в середовищі розробки програмного забезпечення PyCharm.

Поліноміальні хеші

Реалізовано клас Hash, для обчислення поліномаіальних хешів для рядків.

```python
class Hash:
    PRIME_INTERVAL = (31, 100)  # Interval for prime p
    MOD_INTERVAL = (int(1e9), int(1e9) + 100)  # Interval for m near 1e9

    Codeium: Refactor Explain Docstring
    def __init__(self, p=None, m=None):
        # Default m value and random p from the prime interval if not provided
        self.m = m if m is not None else random.randint(*self.MOD_INTERVAL)
        self.p = p if p is not None and isprime(p) else self._get_random_prime(*self.PRIME_INTERVAL)

    Codeium: Refactor Explain Docstring
    1 usage
    def _get_random_prime(self, start, end):
        primes = [x for x in range(start, end + 1) if isprime(x)]
        return random.choice(primes)
```

Коефіцієнти p та m, обираються при ініціалізації, або випадковим чином на встановлених інтервалах.

```python
def compute_hash(self, s: str):
    """Computes the hash of the given string s."""
    hash_value = 0
    p_pow = 1
    for c in s:
        hash_value = (hash_value + (ord(c) - ord('a') + 1) * p_pow) % self.m
        p_pow = (p_pow * self.p) % self.m
    return hash_value
```

Функція обчислення хешу.

Хеш-таблиця

```python
class HashTable:
    DEFAULT_LEN = int(1e5)  # Default length if not provided

    Codeium: Refactor Explain Docstring
    def __init__(self, Len: int = None, hasher: Hash = None):
        self.Len = Len if Len is not None else self.DEFAULT_LEN
        self.table = [None] * self.Len
        self.hasher = hasher if hasher is not None else Hash.Hash()
```

реалізовано клас хеш-таблиці, що використовує вказаний hash, для обчислення позицій в які будуть заноситись елементи таблиці. довжина масиву хеш-таблиці задається при ініціалізації або береться як DEFAULT_LEN.

```python
class Node:
    Codeium: Refactor  Explain  Docstring
    def __init__(self, value):
        self.value = value
        self.count = 1  # Лічильник появ елемента
        self.next = None
```

```python
    def add(self, s: str):
        index = self._get_index(s)
        current = self.table[index]

        # Шукаємо, чи є вже такий елемент
        while current:
            if current.value == s:
                current.count += 1  # Якщо елемент знайдено, збільшуємо лічильник
                return
            current = current.next

        # Якщо елемент не знайдено, додаємо новий
        new_node = Node(s)
        new_node.next = self.table[index]
        self.table[index] = new_node
```

в комірці хеш-таблиці, зберігаються однозв’язний список елементів що туди попали наведено реалізацію додавання рядка до таблиці, та структури однозв’язного списку

```python
    def get_groups(self):
        """Returns pairs of strings and the number of times they appear in the hash table."""
        groups = []

        # Проходимо по кожній комірці хеш-таблиці
        for i in range(self.Len):
            current = self.table[i]

            # Переходимо по зв'язаному списку комірки
            while current:
                groups.append([current.value, current.count])  # Додаємо елемент та його лічильник
                current = current.next

        return groups
```

Реалізація знаходження усіх повторюваних рядків хеш таблиці. (просто обходяться всі елементи хеш-таблиці)

Тестування

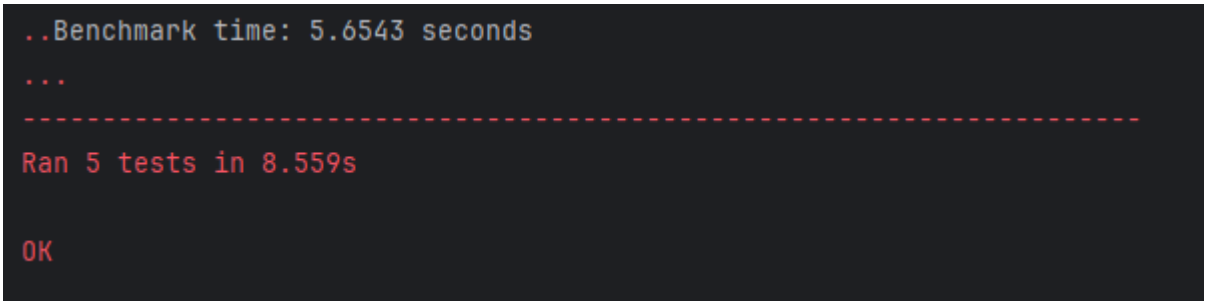
```python
def test_basic_functionality(self):
    """Тестує базові функції хеш-таблиці на простому прикладі."""
    hasher = Hash.Hash()
    ht = HashTable.HashTable(hasher=hasher)

    commands = ["+ abc", "+ xyz", "? abc", "? xyz", "- abc", "? abc", "#"]
    test_results = run_test(commands, ht)
    validation_results = validate_test_results(commands)

    result, message = compare_results(test_results, validation_results)
    self.assertTrue(result, message)
```

```python
def test_with_generated_test(self):
    """Тестує хеш-таблицю на згенерованому тесті."""
    filename = "test_generated.txt"
    hasher = Hash.Hash()
    ht = HashTable.HashTable(hasher=hasher)

    generate_and_save_test(filename, num_commands=100)

    result, message = read_and_validate_test(filename, ht)
    self.assertTrue(result, message)
```

написано декілька юніт-тестів та також рандомно згенеровані тести, заданої кількості операцій.
також відповіді провалідовані за допомогою іншого рішення задачі вбудованим словним в стандартну бібліотеку python, і так перевіряється валідність нашого рішення.

```python
def benchmark():
    filename = "test_benchmark.txt"
    # generate_and_save_test(filename, num_commands=1000000)
    read_and_run_test(filename, ht)

# Вимірюємо час виконання
time_taken = timeit.timeit(benchmark, number=1)
print(f"Benchmark time: {time_taken:.4f} seconds")
```

написано бенчмарк для тесту з мільйоном операцій.



Результати виконання тестів та бенчмарку. Затримки можуть бути пов’язані з I/O операціями

Оцінка складності

Будемо вважати що операція порівняння рядків виконується за O(1), так як за умовою їх довжина обмежена до 15, отже можемо вважати що на їх порівняння надається константна кількість операцій.

Операція додавання та видалення/пошуку виконуються за O(len), де len - довжина однозв’язного списку, що зберігається в комірці куди додається або звідки вилучається рядок.

За властивістю поліноміальних хеш функцій O(len)=O(a), де a=n/m.

При заданих умовах (n=1000000, m=100000) a=O(1).

Також для отримання усіх повторень рядків, просто обходиться вся хеш-таблиця, що очевидно виконується за O(n) операцій.