Лабораторна робота 2

Головата Карина (МІ-41)

Тема: Реалізація фільтра Блума

Мета роботи

Реалізувати структуру даних **фільтр Блума** для роботи з рядками довжиною до 15 символів латинських літер. Забезпечити підтримку базових операцій додавання елементів та перевірки наявності елемента в множині, з можливістю реалізувати вилучення з підрахунком.

Теоретична частина

Фільтр Блума

Фільтр Блума — це **ймовірнісна структура даних**, яка використовується для перевірки належності елементів множині. Він забезпечує швидкі перевірки та ефективне використання пам’яті. Однак, фільтр Блума допускає **хибнопозитивні спрацювання** — коли елемент вважається наявним, хоча насправді його немає в множині.
Особливість фільтра полягає в тому, що він **ніколи не дає хибнонегативних спрацювань**, тобто якщо елемент позначено як відсутній, це завжди достовірно.

Принцип роботи фільтра Блума

1. **Хеш-функції** обчислюють кілька індексів для кожного елемента, що додається у фільтр.
2. **Додавання елемента**: на всіх відповідних позиціях у бітовому масиві встановлюються значення (або збільшуються лічильники у лічильниковому фільтрі).
3. **Перевірка наявності**: якщо всі відповідні позиції містять 1 (або лічильники більше 0), елемент може бути в множині. Інакше — він точно відсутній.
4. **Вилучення елемента**: у **лічильниковому фільтрі** зменшуються відповідні значення лічильників.

Розрахунок параметрів фільтра Блума

Розмір масиву m:

$$m = -\frac{n \cdot \ln p}{(\ln 2)^2}$$

де:

- n — максимальна кількість елементів у множині.
- p — ймовірність хибнопозитивного спрацювання (1%).

Кількість хеш-функцій k:

$$k = \frac{m}{n} \cdot \ln 2$$

Опис програми

Вхідні дані

Файл із рядками формату:
`<символ операції> <рядок довжиною до 15 латинських символів>`

- **+** `<рядок>` — додати елемент до множини.
- **?** `<рядок>` — перевірити, чи належить елемент множині.
- **-** `<рядок>` — вилучити елемент із множини (за бажанням).
  Список операцій завершується символом `#`.

Вихідні дані

- Для кожної операції перевірки `?` програма виводить:
  - **"Y"** — якщо рядок присутній у множині.
  - **"N"** — якщо рядок відсутній.

Код програми

Модуль `bloom_filter.py`

```
import math
import random

class BloomFilterUniversal:
    def __init__(self, n, p):
        self.n = n  # Максимальна кількість елементів
        self.p = p  # Ймовірність хибнопозитивних спрацювань
        self.m = self.calculate_size()  # Розмір масиву лічильників
        self.l = self.calculate_hash_count()  # Кількість хеш-функцій
        self.counter_array = [0] * self.m  # Масив лічильників
        self.big_prime = 1000000007  # Велике просте число
```

```
        # Ініціалізація випадкових коефіцієнтів a та b для кожної хеш-функції
        self.a = [random.randint(1, self.big_prime - 1) for _ in
range(self.l)]
        self.b = [random.randint(0, self.big_prime - 1) for _ in
range(self.l)]

    def calculate_size(self):
        """Розраховує розмір масиву лічильників"""
        m = -(self.n * math.log(self.p)) / (math.log(2) ** 2)
        return int(m)

    def calculate_hash_count(self):
        """Розраховує кількість хеш-функцій для оптимальної роботи фільтра"""
        l = (self.m / self.n) * math.log(2)
        return int(l)

    def hash_item(self, item, i):
        """Обчислює універсальне хеш-значення для item з використанням i-ї
хеш-функції"""
        x = self.string_to_int(item)  # Перетворюємо рядок у числове значення
        return ((self.a[i] * x + self.b[i]) % self.big_prime) % self.m

    def string_to_int(self, s):
        """Перетворює рядок у числове значення"""
        # Проста функція, яка конвертує рядок у ціле число
        result = 0
        for char in s:
            result = result * 31 + ord(char)  # Використовуємо базу 31 для
уникнення колізій
        return result

    def get_hash_values(self, item):
        """Генерує кілька хеш-значень для рядка"""
        return [self.hash_item(item, i) for i in range(self.l)]

    def add(self, item):
        """Додає елемент до фільтра Блума"""
        hash_values = self.get_hash_values(item)
        for index in hash_values:
            self.counter_array[index] += 1  # Збільшуємо лічильник

    def remove(self, item):
        """Видаляє елемент з фільтра Блума, якщо він існує"""
        if not self.check(item):
            return False  # Елемент не знайдено

        hash_values = self.get_hash_values(item)
        for index in hash_values:
            self.counter_array[index] -= 1  # Зменшуємо лічильник
        return True

    def check(self, item):
        """Перевіряє наявність елемента у фільтрі"""
        hash_values = self.get_hash_values(item)
        return all(self.counter_array[index] > 0 for index in hash_values)
)
```

Реалізація BloomFilterUniversal:

1. **Ініціалізація:**

- `self.n` та `self.p` — задаються користувачем, де `n` — максимальна кількість елементів, а `p` — ймовірність хибнопозитивних спрацювань.
- `self.m` — розраховується методом `calculate_size()` для визначення розміру масиву лічильників.
- `self.l` — розраховується методом `calculate_hash_count()` для визначення оптимальної кількості хеш-функцій.
- `self.counter_array` — масив лічильників розміру `self.m`, ініціалізується нулями.
- `self.big_prime` — велике просте число, використовується для зменшення колізій при хешуванні.
- `self.a` та `self.b` — масиви випадкових коефіцієнтів, що генеруються для кожної хеш-функції. Вони використовуються у формулі хешування для забезпечення унікальності та випадковості розподілу.

2. **Метод `calculate_size()`:**
   - Обчислює необхідний розмір масиву лічильників за формулою `m = -(self.n * math.log(self.p)) / (math.log(2) ** 2)`.
   - Формула забезпечує досягнення заданої ймовірності хибнопозитивного результату для вказаної кількості елементів.
3. **Метод `calculate_hash_count()`:**
   - Обчислює оптимальну кількість хеш-функцій за формулою `l = (self.m / self.n) * math.log(2)`.
   - Це дозволяє зменшити ймовірність хибнопозитивних спрацювань, забезпечуючи ефективне хешування.
4. **Метод `hash_item(item, i)`:**
   - Обчислює універсальне хеш-значення для елемента `item` з використанням і-ї хеш-функції.
   - Хешування відбувається за формулою: `((self.a[i] * x + self.b[i]) % self.big_prime) % self.m`, де `x` — числове значення рядка.
   - Використання великих простих чисел та випадкових коефіцієнтів `a` та `b` забезпечує рівномірний розподіл значень хеш-функцій.
5. **Метод `string_to_int(s)`:**
   - Перетворює рядок `s` у числове значення.
   - Виконує це шляхом обчислення хешу на основі кожного символу з використанням бази `31` для уникнення колізій: `result = result * 31 + ord(char)`.
   - Такий підхід дозволяє отримати унікальне числове представлення рядка для подальшого хешування.
6. **Метод `get_hash_values(item)`:**
   - Генерує `l` хеш-значень для рядка `item` за допомогою `hash_item`.
   - Повертає список індексів для лічильників, які відповідають цьому рядку.
7. **Метод `add(item)`:**
   - Додає елемент до фільтра, генеруючи `l` хешів для нього та збільшуючи лічильники на відповідних позиціях у `self.counter_array`.
8. **Метод `remove(item)`:**
   - Перевіряє наявність елемента у фільтрі за допомогою `check()`.
   - Якщо елемент присутній, зменшує значення лічильників на відповідних позиціях.
   - Використовується для видалення елемента з фільтра.
9. **Метод `check(item)`:**
   - Перевіряє наявність елемента в фільтрі за допомогою зчитування значень лічильників.

- Якщо всі відповідні лічильники більші за нуль, елемент може бути присутнім (хибнопозитивний результат).
- Якщо хоча б один лічильник дорівнює нулю, елемент точно відсутній.

Головний модуль `main.py`

```
from bloom_filter import BloomFilter  # Імпортуємо реалізацію фільтра Блума


def process_file(filename, bloom_filter):
    """Обробляє вхідний файл з операціями додавання, перевірки та
видалення"""
    operations = {
        "add": 0,
        "check": 0,
        "found": 0,  # Кількість успішних перевірок
        "removed": 0,  # Кількість успішних видалень
        "checked_items": [],
    }

    with open(filename, 'r') as file:
        for line in file:
            if line.startswith('#'):
                break
            operation, item = line[0], line[2:].strip()
            if operation == '+':
                bloom_filter.add(item)
                operations["add"] += 1
            elif operation == '?':
                operations["check"] += 1
                operations["checked_items"].append(item)
                result = "Y" if bloom_filter.check(item) else "N"
                if result == "Y":
                    operations["found"] += 1  # Підраховуємо успішні
перевірки
                print(result)
            elif operation == '-':
                removed = bloom_filter.remove(item)
                if removed:
                    operations["removed"] += 1
                    print(f"Елемент '{item}' успішно видалений.")
                else:
                    print(f"Елемент '{item}' не знайдено у фільтрі.")

    return operations


def print_statistics(operations):
    """Виводить статистику після обробки файлу"""
    print("\n=== Статистика ===")
    print(f"Додано елементів: {operations['add']}")
    print(f"Перевірено елементів: {operations['check']}")
    print(f"Знайдено елементів: {operations['found']}")
    print(f"Видалено елементів: {operations['removed']}")
    if operations['check'] > 0:
        found_percentage = (operations['found'] / operations['check']) * 100
        print(f"Відсоток знайдених елементів: {found_percentage:.2f}%")
    else:
        print("Перевірки не виконувались.")
```

```
def main():
    n = 10 ** 6  # максимальна кількість елементів
    p = 0.01  # ймовірність хибнопозитивних спрацювань
    bloom_filter = BloomFilter(n, p)

    # Обробка файлу з операціями
    operations = process_file('input.txt', bloom_filter)

    # Виведення статистики
    print_statistics(operations)


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

Пояснення коду:

1. **Функція `process_file(filename, bloom_filter)`:**
   - Читає вхідний файл построчно.
   - Виконує операції залежно від символу на початку рядка:
     - `'+'` — додавання елемента.
     - `'?'` — перевірка наявності елемента.
     - `'-'` — видалення елемента.
   - Веде підрахунок статистики операцій.
   - Виводить результати операцій у консоль.
2. **Функція `print_statistics(operations)`:**
   - Виводить зведену статистику виконаних операцій.
3. **Функція `main()`:**
   - Ініціалізує фільтр Блума з заданими параметрами.
   - Викликає функцію `process_file()` для обробки вхідного файлу.
   - Після завершення операцій виводить статистику.

Тестовий модуль `test_bloom_filter.py`

```
import random
import string
import time
from bloom_filter import BloomFilter


def generate_random_string(min_length=1, max_length=15):
    """ Генерує випадковий рядок із малих латинських літер довжиною від
min_length до max_length """
    length = random.randint(min_length, max_length)
    return ''.join(random.choice(string.ascii_lowercase) for _ in
range(length))


def test_false_positive_rate():
    N = 1_000_000  # Половина ключів буде додана до фільтра
    p = 0.01  # Теоретична ймовірність хибнопозитивних спрацювань
    bloom_filter = BloomFilter(N, p)

    print(f"Тестування фільтра Блума з {N} елементами.")
    start_time = time.time()
```

```
    # Генерація 2 мільйонів унікальних випадкових рядків
    print("Генерація унікальних ключів...")
    keys = set()
    while len(keys) < 2 * N:
        keys.add(generate_random_string())
    print(f"Згенеровано {len(keys)} унікальних ключів.")

    keys = list(keys)
    inserted_keys = keys[:N]
    test_keys = keys[N:]

    # Додаємо першу половину рядків до фільтра Блума
    print("Додавання першої половини ключів до фільтра Блума...")
    for i, key in enumerate(inserted_keys, 1):
        bloom_filter.add(key)
        if i % 50000 == 0:
            print(f"Додано {i} ключів у фільтр Блума.")
    print("Додавання завершено.")

    # Перевіряємо другу половину рядків і підраховуємо хибнопозитивні
спрацювання
    print("Початок перевірки на хибнопозитивні спрацювання...")
    false_positives = 0
    for i, key in enumerate(test_keys, 1):
        if bloom_filter.check(key):
            false_positives += 1
        if i % 50000 == 0:
            print(f"Перевірено {i} ключів, хибнопозитивні:
{false_positives}")
    print("Перевірка завершена.")

    # Розраховуємо фактичну ймовірність хибнопозитивних спрацювань
    false_positive_rate = (false_positives / N) * 100
    print(f"Теоретична ймовірність хибнопозитивних спрацювань: {p * 100}%")
    print(f"Кількість хибнопозитивних спрацювань: {false_positives}")
    print(f"Фактична ймовірність хибнопозитивних спрацювань:
{false_positive_rate:.2f}%")

    # Підсумковий час виконання тесту
    end_time = time.time()
    print(f"Тест завершено за {end_time - start_time:.2f} секунд.")


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

Пояснення коду:

1. **Функція `generate_random_string()`:**
   - Генерує випадкові рядки довжиною від 1 до 15 символів з малих латинських літер.
2. **Функція `test_false_positive_rate()`:**
   - Генерує 2×N унікальних ключів.
   - Додає перші N ключів до фільтра Блума.
   - Перевіряє решту N ключів, які не були додані, щоб визначити кількість хибнопозитивних спрацювань.
   - Розраховує фактичну ймовірність хибнопозитивних спрацювань і порівнює з теоретичною.
   - Виводить результати та час виконання тесту.

Модуль `generate_file.py`

```
import random
import string


def generate_random_string(length=15):
    # Генерує випадковий рядок з малих латинських літер довжиною до 15
символів
    return ''.join(random.choice(string.ascii_lowercase) for _ in
range(random.randint(1, length)))


def generate_input_file(filename, num_lines):
    operations = ['+', '?', '-']  # Операції: додавання, перевірка, видалення
    with open(filename, 'w') as file:
        for _ in range(num_lines):
            operation = random.choice(operations)  # Випадкова операція
            random_string = generate_random_string()  # Випадковий рядок
            file.write(f"{operation} {random_string}\n")
        file.write('#')  # Додаємо символ завершення операцій


if __name__ == "__main__":
    num_lines = int(input("Введіть бажану кількість рядків для генерації: "))
    generate_input_file('input.txt', num_lines)
    print(f"Згенеровано файл 'input.txt' з {num_lines} рядками.")
```

Пояснення коду:

1. **Функція `generate_random_string()`:**
   - Генерує випадковий рядок довжиною від 1 до 15 символів.
2. **Функція `generate_input_file()`:**
   - Генерує файл `input.txt` з заданою кількістю рядків.
   - Кожен рядок містить випадкову операцію (`'+'`, `'?'`, `'-'`) та випадковий рядок.
   - В кінці файлу додається символ `'#'` для позначення завершення операцій.

Модуль `evaluate_time.py`

```
import time
import random
import string
from bloom_filter import BloomFilter


def generate_random_string(length=15):
    """Генерує випадковий рядок із малих латинських літер довжиною до 15
символів."""
    return ''.join(random.choice(string.ascii_lowercase) for _ in
range(length))


def evaluate_performance(bloom_filter, num_operations=1000000):
    """Оцінює середній час виконання операцій додавання, перевірки та
```

```
видалення."""
    add_times = []
    check_times = []
    remove_times = []

    # Генеруємо випадкові рядки для тестування
    test_strings = [generate_random_string() for _ in range(num_operations)]

    # Вимірюємо час додавання
    for item in test_strings:
        start_time = time.time()
        bloom_filter.add(item)
        end_time = time.time()
        add_times.append(end_time - start_time)

    # Вимірюємо час перевірки
    for item in test_strings:
        start_time = time.time()
        bloom_filter.check(item)
        end_time = time.time()
        check_times.append(end_time - start_time)

    # Вимірюємо час видалення
    for item in test_strings:
        start_time = time.time()
        bloom_filter.remove(item)
        end_time = time.time()
        remove_times.append(end_time - start_time)

    # Розрахунок середніх значень
    avg_add_time = sum(add_times) / num_operations
    avg_check_time = sum(check_times) / num_operations
    avg_remove_time = sum(remove_times) / num_operations

    print(f"Середній час додавання: {avg_add_time * 1000:.6f} мс")
    print(f"Середній час перевірки: {avg_check_time * 1000:.6f} мс")
    print(f"Середній час видалення: {avg_remove_time * 1000:.6f} мс")


if __name__ == "__main__":
    n = 10 ** 6  # Максимальна кількість елементів
    p = 0.01  # Ймовірність хибнопозитивних спрацювань
    bloom_filter = BloomFilter(n, p)

    # Оцінка продуктивності на великій кількості операцій
    evaluate_performance(bloom_filter)
```

Код виконує оцінку середнього часу виконання основних операцій фільтра Блума: додавання, перевірки та видалення елементів. Це дозволяє зрозуміти, наскільки ефективно фільтр працює з великим обсягом даних, у даному випадку — з 1,000,000 випадково згенерованих рядків.

Основні кроки:

1. **Генерація випадкових рядків:**
   - Використовується функція generate_random_string() для створення набору з 1,000,000 випадкових рядків довжиною до 15 символів.
   - Це необхідно для тестування роботи фільтра з різними вхідними даними.
2. **Вимірювання часу операцій:**

- Для кожного згенерованого рядка виконуються операції додавання, перевірки та видалення.
- Час виконання кожної операції вимірюється за допомогою time.time() і зберігається в списках.
- Після завершення операцій розраховується середній час для кожного типу операцій.

3. **Розрахунок та виведення результатів:**
   - Середній час для кожної операції обчислюється шляхом підсумовування часу всіх операцій і ділення на кількість операцій.
   - Результати виводяться у мілісекундах, що дозволяє легко оцінити ефективність реалізованого фільтра Блума.

```
Середній час додавання: 0.002401 мс
Середній час перевірки: 0.001942 мс
Середній час видалення: 0.003910 мс

Process finished with exit code 0
```

Приклад згенерованого вхідного файлу input.txt

```
+ lwtehylohqxc
- ullciqpxbozhh
+ apxtus
? xdzaig
- d
+ zctpojwjaowai
? xszwsjpjxtm
+ akslsxwdkz
- bmbcksfdhp
#
```

Приклад роботи програми

Результат запуску main.py:

```
Елемент 'zyfmxb' не знайдено у фільтрі.
N
N
Елемент 'ullciqpxbozhh' відсутній у фільтрі.
Елемент 'ullciqpxbozhh' не знайдено у фільтрі.
N
Елемент 'd' успішно видалений.
N
Елемент 'bmbcksfdhp' відсутній у фільтрі.
Елемент 'bmbcksfdhp' не знайдено у фільтрі.

=== Статистика ===
Додано елементів: 1685
Перевірено елементів: 1610
Знайдено елементів: 77
Видалено елементів: 72
Відсоток знайдених елементів: 4.78%

Process finished with exit code 0
```

Результат запуску test_bloom_filter.py:

1. Генерація ключів для тесту і додавання в фільтр;

```
Тестування фільтра Блума з 1000000 елементами.
Генерація унікальних ключів...
Згенеровано 2000000 унікальних ключів.
Додавання першої половини ключів до фільтра Блума...
Додано 50000 ключів у фільтр Блума.
Додано 100000 ключів у фільтр Блума.
Додано 150000 ключів у фільтр Блума.
Додано 200000 ключів у фільтр Блума.
```

2. Закінчення додавання ключів і перевірка на хибнопозитивні спрацювання;

```
Додано 1000000 ключів у фільтр Блума.
Додавання завершено.
Початок перевірки на хибнопозитивні спрацювання...
Перевірено 50000 ключів, хибнопозитивні: 496
Перевірено 100000 ключів, хибнопозитивні: 1004
Перевірено 150000 ключів, хибнопозитивні: 1489
```

3. Закінчення перевірки та підрахунки результатів.

```
Перевірено 800000 ключів, хибнопозитивні: 8113
Перевірено 850000 ключів, хибнопозитивні: 8623
Перевірено 900000 ключів, хибнопозитивні: 9117
Перевірено 950000 ключів, хибнопозитивні: 9645
Перевірено 1000000 ключів, хибнопозитивні: 10174
Перевірка завершена.
Теоретична ймовірність хибнопозитивних спрацювань: 1.0%
Кількість хибнопозитивних спрацювань: 10174
Фактична ймовірність хибнопозитивних спрацювань: 1.02%
Тест завершено за 15.23 секунд.
```

Аналіз асимптотичної складності фільтра Блума:

1. Ініціалізація:

- `__init__()`:
  - Розрахунок розміру масиву `m` та кількості хеш-функцій `k` займає O(1).
  - Ініціалізація масиву лічильників `self.counter_array` розміром `m` займає O(m).
  - Генерація коефіцієнтів `a` та `b` займає O(k), оскільки для кожної хеш-функції створюються випадкові коефіцієнти.
  - Загальна складність ініціалізації: O(m+k).

2. Обчислення розміру масиву `calculate_size()`:

- Виконується за допомогою формули: $m = -\frac{n \cdot \ln(p)}{(\ln(2))^2}$
  - Формула розрахунку має постійну складність O(1), оскільки виконує арифметичні операції.

3. Обчислення кількості хеш-функцій `calculate_hash_count()`:

- Виконується за формулою: $k = \left(\frac{m}{n}\right) \ln(2)$

- Формула також має постійну складність O(1).

4. Додавання елемента `add(item)`:

- Виконує такі дії:
  - Викликає `get_hash_values(item)` для обчислення `k` хеш-значень.
    - `get_hash_values(item)` викликає `hash_item()` `k` разів.
    - `hash_item()` виконує операції з числом, що мають складність O(1).
    - Перетворення рядка в ціле число `string_to_int()` має складність O(L), де `L` — довжина рядка, оскільки кожен символ обробляється один раз.
    - Загальна складність `get_hash_values()` дорівнює O(k·L).
  - Після обчислення хешів виконує збільшення лічильників для кожного індексу, що займає O(k).
  - Загальна складність `add(item)` дорівнює O(k·L)+O(k)=O(k·L).

5. Перевірка наявності `check(item)`:

- Аналогічно до `add()` викликає `get_hash_values()` з тією ж складністю O(k·L).
- Перевірка значень лічильників займає O(k).
- Загальна складність `check(item)` дорівнює O(k·L).

6. Видалення елемента `remove(item)`:

- Спочатку виконує перевірку наявності елемента за допомогою `check(item)`, що має складність O(k·L).
- Якщо елемент знайдено, зменшує значення лічильників, що займає O(k)
- Загальна складність `remove(item)` дорівнює O(k·L).

Загальна асимптотична складність:

- Ініціалізація: O(m+k)
- Додавання елемента: O(k·L)
- Перевірка наявності: O(k·L)
- Видалення елемента: O(k·L)