**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:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Кількість хеш-функцій k:

Изображение выглядит как Шрифт, текст, часы, число

Автоматически созданное описание

**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 i-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.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

**Приклад згенерованого вхідного файлу input.txt**

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

**Приклад роботи програми**

Результат запуску main.py:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Результат запуску test\_bloom\_filter.py:

* 1. Генерація ключів для тесту і додавання в фільтр;

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

* 1. Закінчення додавання ключів і перевірка на хибнопозитивні спрацювання;

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

* 1. Закінчення перевірки та підрахунки результатів.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

### 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: Изображение выглядит как Шрифт, текст, число, Графика

  Автоматически созданное описание
  + Формула розрахунку має постійну складність O(1), оскільки виконує арифметичні операції.

**3. Обчислення кількості хеш-функцій calculate\_hash\_count():**

* Виконується за формулою: Изображение выглядит как Шрифт, символ, рукописный текст, Графика

  Автоматически созданное описание
  + 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)