

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ В.Н. КАРАЗІНА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК

КУРСОВА РОБОТА
з дисципліни «Теорія алгоритмів»
Тема «Bloom фільтр»

Оцінка _____ / _____

Члени комісії:

Виконав студент 2 курсу
групи КБ-22
Юрченко Максим Петрович
Перевірили:
доц. Олешко О.І

ЗМІСТ

ВСТУП	3
ТЕОРЕТИЧНІ ВІДОМОСТІ.....	4
1.1 Опис алгоритму	4
1.2 Де застовують алгоритм?	8
1.3 Переваги та недоліки фільтра Блума	8
ПРАКТИЧНА РЕАЛІЗАЦІЯ.....	10
2.1 Інструменти ДЛІЯ реалізування	10
2.2 Загальні характеристики системи та файлів.....	10
2.3 Складність алгоритму, її дослідження.	11
ВИСНОВКИ	13
ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАННЯ	14
ДОДАТОК А.....	16
1.1 Вихідний код програми:.....	16

ВСТУП

Моє дослідження полягає в порівнянні теоретичної і практичної складності алгоритма Блума.

Структура даних — це спосіб організації даних в комп'ютерах. Часто разом зі структурою даних пов'язується і специфічний перелік операцій, що можуть бути виконаними над даними, організованими в таку структуру.

Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх обробки. Добре побудовані структури даних дозволяють оптимізувати використання часу та пам'яті комп'ютера для виконання найкритичніших операцій.

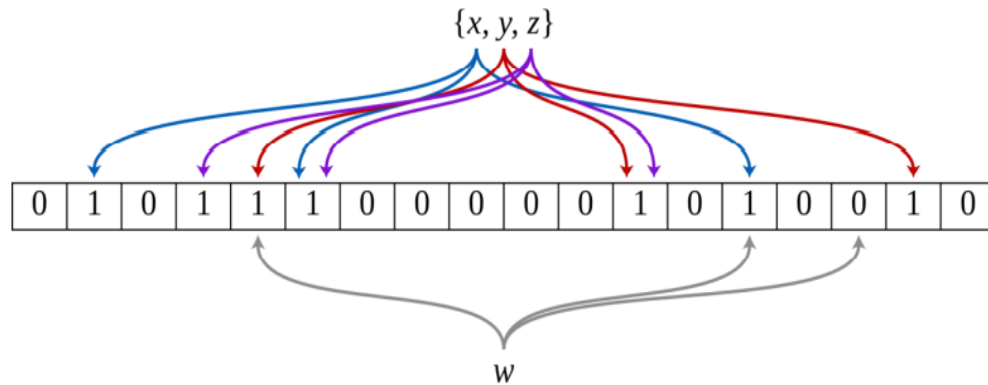
Алгоритм — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час.

Фільтр Блума — заощадлива до пам'яті ймовірнісна структура даних, призначена для перевірки приналежності елементів до множини. Запропонована Бартоном Говардом Блумом в 1970 році.

ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Опис алгоритму

Приклад фільтра Блума з $m = 18$ і $k = 3$ зберігає множину $\{x, y, z\}$



Малюнок 1.1 – “Приклад роботи Фільтра Блума”

Порожній фільтр Bloom - це бітовий масив з m бітів, всі встановлені в 0. Також має бути визначено k різних хеш-функцій, кожна з яких відображає або хешує якийсь елемент набору в одне з m -позицій масиву, створюючи рівномірний випадковий розподіл. Як правило, k - це мала константа, яка залежить від бажаної частоти помилкових помилок ϵ , тоді як m пропорційна k та кількості елементів, які потрібно додати. Щоб додати елемент, подайте його до кожної з k хеш-функцій, щоб отримати k позицій масиву. Встановіть біти у всіх цих положеннях на 1. Для запиту елемента (перевірити, чи є він у наборі), подайте його до кожної з k хеш-функцій, щоб отримати k позицій масиву. Якщо будь-який з бітів у цих положеннях дорівнює 0, елемента точно немає в наборі; якби це було так, тоді всі біти були б встановлені в 1, коли він був вставлений. Якщо всі 1, то або елемент знаходиться в наборі, або біти випадково встановлені в 1 під час вставки інших елементів, що призводить до помилково позитивного. У простому фільтрі Блум неможливо розрізнити два випадки, але більш досконалі методи можуть вирішити цю проблему. Вимога щодо проектування k різних незалежних хеш-функцій може бути надмірною

для великих k . Для хорошої хеш-функції з широким висновком має бути мало, якщо взагалі існує взаємозв'язок між різними бітовими полями такого хешу, тому цей тип хешу можна використовувати для генерації декількох "різних" хеш-функцій, нарізаючи її вихід на кілька бітів поля. Як варіант, б можна передати k різних початкових значень (наприклад, $0, 1, \dots, k - 1$) у хешфункцію, яка приймає початкове значення; або додайте (або додайте) ці значення до ключа. Для більших m та / або k незалежність між хешфункціями може бути послаблена незначним збільшенням коефіцієнта помилково позитивних. (Зокрема, Dillinger & Manolios (2004b) показують ефективність виведення k -індексів за допомогою посиленого подвійного хешування або потрійного хешування, варіантів подвійного хешування, які є фактично простими генераторами випадкових чисел, засіяними двома або трьома хеш-значеннями.) Видалити елемент із цього простого фільтра Bloom неможливо, оскільки немає можливості визначити, який з k бітів, для якого він відображається, слід очистити. Хоча встановлення будь-якого з цих k бітів на нуль достатньо для видалення елемента, воно також видалить будь-які інші елементи, які випадково зіставляються з цим бітом. Оскільки простий алгоритм не дає можливості визначити, чи були додані будь-які інші елементи, які впливають на біти для елемента, що підлягає видаленню, очищення будь-якого з бітів створить можливість помилкових негативних негативних наслідків. Одноразове вилучення елемента з фільтра Bloom можна змодельовати, маючи другий фільтр Bloom, який містить елементи, які були видалені. Однак помилкові спрацьовування у другому фільтрі стають помилковими негативними у складеному фільтрі, що може бути небажаним. При такому підході повторне додавання раніше вилученого елемента неможливе, оскільки його потрібно було б вилучити з фільтра "вилученого". Часто буває, що всі ключі доступні, але їх перерахування дороге (наприклад, вимагає багато читань на диску). Коли коефіцієнт помилково позитивних позицій стає занадто високим, фільтр можна відновити; це має бути порівняно рідкісною подією.

Нехай розмір бітового масиву дорівнює m біт і задано k хеш-функцій. Припустимо, що безліч хеш-функцій вибирається випадково, і для будь-якого елемента x кожна хеш-функція h_i призначає йому одне з місць у бітовому масиві з однаковою ймовірністю: $\Pr(h_i(x) = p) = \frac{1}{m}, \quad p = 1 \dots m,$

і крім того, значення є незалежними в сукупності випадковими величинами (для спрощення подальшого аналізу). Тоді ймовірність того, що в певний p -й біт НЕ буде записана одиниця під час операції вставки чергового елемента дорівнює:

$$\Pr(h_1(x) \neq p \cap \dots \cap h_k(x) \neq p) = \Pr(h_i(x) \neq p)^k = \left(1 - \frac{1}{m}\right)^k.$$

А ймовірність того, що p -й біт залишиться рівним нулю після вставки n різних елементів $x_1, \dots, x_n$ в спочатку порожній фільтр Блума дорівнює

$$\Pr(\forall i \in \{1 \dots k\}, \forall j \in \{1 \dots n\}, h_i(x_j) \neq p) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

для досить великого m зважаючи другого чудового краю.

Хибно позитивні спрацьовування полягає в тому, що для деякого елемента y , що не рівного жодному з вставлених, все k біт з номерами $h_i(y)$ виявляться ненульовими, і фільтр Блума помилково відповість, що y входить у безліч вставлених елементів. Ймовірність такої події приблизно дорівнює:

$$(1 - e^{-kn/m})^k.$$

Очевидно, що ця ймовірність зменшується з ростом m (розміру бітового масиву) і збільшується з ростом n (числа вставлених елементів). Для фіксованих m і n оптимальне число k (число хеш-функцій), які мінімізують її, дорівнюють:

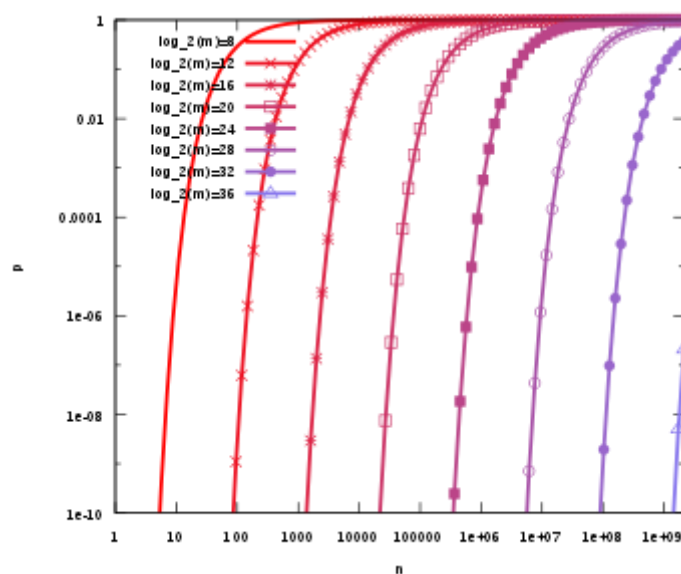
$$k = \frac{m}{n} \ln 2 \approx 0,6931 \frac{m}{n}.$$

При цьому сама ймовірність помилкового спрацювання дорівнює:

$$k = \frac{m}{n} \ln 2 \approx 0,6931 \frac{m}{n}.$$

Як наслідок, зауважимо, що для того, щоб фільтр Блума підтримував задану обмежену вірогідність помилкового спрацювання, розмір бітового масиву повинен бути лінійно пропорційний числу вставлених елементів.

Порівняно з хеш-таблицями фільтр Блума може обходитися у кілька порядків меншими обсягами пам'яті, жертвуючи детермінізмом. Зазвичай вона використовується для зменшення кількості запитів до неіснуючих даних у структурі даних з більш дорогим доступом (наприклад, розташованої на жорсткому диску або в базі даних), тобто для «фільтрації» запитів до неї.



Малюнок 1.2 “Оцінка вірогідності помилкового спрацювання”

1.2 Де застосовують алгоритм?

Приклади практичних застосувань:

Проксі-сервер Squid використовує фільтри Блума для налаштування cache digests.

Google BigTable використовує фільтри Блума для зменшення кількості звернень до жорсткого диска при перевірці існування заданого рядка чи стовпця у таблиці бази даних.

Комп'ютерні програми для перевірки орфографії.

Біткойн використовував фільтри Блума для прискорення синхронізації гаманця, доки не було виявлено вразливість конфіденційності під час реалізації фільтрів Блума.

1.3 Переваги та недоліки фільтра Блума

Фільтр Блума має важливу перевагу: він економить пам'ять і збільшує швидкість перевірки інформації про наявність конкретного об'єкта. Але через це проявляється значний недолік.

Фільтр працює з непрямыми даними, що обчислюються на основі хеш-функцій. Вони піддаються колізіям — коли за різних вхідних документів із певною частотою видаються однакові відповіді.

Через це недолік у фільтра Блума з'являється обмеження - ймовірність помилки. Фільтр відповідає, що об'єкт, можливо, є, хоча його немає у початковій послідовності.

Можливість помилки можна зменшити, якщо підібрати оптимальну кількість хеш-функцій, які обробляють вхідну послідовність. Ще можливість помилки можна знизити, якщо збільшити розмірність масиву бітів.

ПРАКТИЧНА РЕАЛІЗАЦІЯ

2.1 Інструменти для реалізування

Було вирішено взяти одну з найпопулярніших і найшвидших мов для алгоритмів – С, яку нам запропонували для виконання цієї роботи. IDE – CLion (версія 2022.2.3), бо я вважаю його одним з найзручніших для роботи. Також, було застосовано перелік джерел літератури, що наведено нижче

2.2 Загальні характеристики системи та файлів

Процесор: Intel Core i5-11320H (2.5 - 4.5 ГГц)

Оперативна пам'ять: RAM 8 ГБ

Жорсткий диск: SSD 256 ГБ

Відеокарта: nVidia GeForce GTX 1650, 4 ГБ

ОС: Windows 10

Файли: main.c, bloom.c, hashes.h, bloom.h, data.txt, dictionary.txt.

Розміри файлів відповідно: 2.25Кб, 2.87Кб, 1.69Кб, 779б, 12Кб, 518Кб

Файл main.c запускає інші файли, також в ньому обчислюється час виконання програми

Файли bloom.c та bloom.h мають реалізацію фільтра Блума

Файл hashes.h має в собі хеш-функції

Файл dictionary.txt при запуску видає повний словник офіційних слів англійської мови, що взагалі існують.

У файлі data.txt містяться дані користувача.

2.3 Складність алгоритму, її дослідження.

Теорична складність алгоритму фільтр Блума є $O(1)$, тобто швидкість виконання алгоритму не залежить від вхідних даних. Тобто фільтр Блума не шукає об'єкт, а перевіряє, що його не існує. На цьому і була базована моя програма, що дозволяє знайти слово, якого не існує і вивести його.

Для дослідження практичної складності була реалізована програма, яка дозволяє користувачу вводити свої данні в файл, а потім виконувати фільтрацію. Для того, щоб пришвидшити дослідження та зробити його більш зручним було вирішено проводити 1000 фільтрацій, записувати час кожної і вираховувати середнє значення. Дослідження проводилось без зайвих операцій вводу та виводу, що значно сповільнюють програму. Результати тестування наведено на графіку та таблиці:

Bloom filter			
Size	Duration, s		
1000	0,0000126	0,0000093	0,0000104
10000	0,0000126	0,0000169	0,0000151
50000	0,0000179	0,0000176	0,0000126
100000	0,0000116	0,0000097	0,0000118

Таблиця 3.1 – вимірювання середнього часу за 1000 фільтрацій

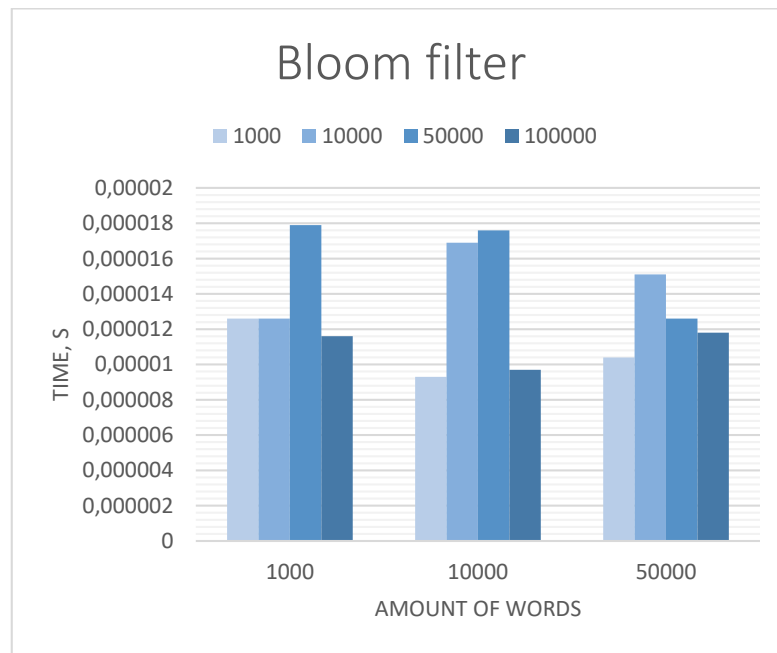


Рисунок 3.2 – графік вимірювань середнього часу за 1000 фільтрацій

ВИСНОВКИ

Виконавши дослідницьку роботу з теми фільтр Блума можна сказати що:

Це - реалізація імовірнісної множини, придумана Бертоном Блумом в 1970 році, що дозволяє компактно зберігати елементи і перевіряти належність заданого елемента до множини. При цьому існує можливість отримати хибнопозитивне спрацьовування (елемента в безлічі немає, але структура даних повідомляє, що він є), але не хибнонегативне.

Що фільтр Блума має такі переваги, як невеликі затрати по пам'яті, нескладну реалізацію та високу швидкість. Ціною за такі переваги стало хибнопозитивне спрацьовування.

Теоретична складність алгоритма є $O(1)$, що практично було підтверджено.

Фільтр Блума – актуальний алгоритм, що широко використовується в сучасному програмуванні

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАННЯ

- [1] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. Accepted to Internet Mathematics, 2005.
- [2] Francis Chang, Wu-Chang Feng, and Kang Li. Approximate caches for packet classification. In Proceeding of IEEE Infocom 2004, Hongkong, 2004.
- [3] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. SIGCOMM, (Karlsruhe, Germany), Aug, 2003.
- [4] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. HOTI'03: Hot Interconnects 11:2003, Stanford, 8/03.
- [5] P.C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. FMCAD, Formal Methods in Computer-Aided Design, 2004.
- [6] C. Estan and G. Varghese. New directions in traffic measurement and accounting. Proceedings of ACM SIGCOMM, 2002.
- [7] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. Proceedings of the USENIX/ACM Internet Measurement Conference, October 2003.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache

sharing protocol. *IEEE/ACM Transaction on Networking*, 8(3):281–293, 2000.

[9] A. Kirsch and M. Mitzenmacher. Building a better bloom filter. Technical Report of Computer Science

Group, Harvard University, (TR-02-05), 2005.

[10] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with multiple choices. 43rd Annual

Allerton Conference on Communication, Control and Computing, Sep, 2005.

[11] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *Proceedings of the joint international conference on Measurement and*

modeling of computer systems, pages 177–188, 2004.

[12] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow

traffic measurement. *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*,

pages 167–172, 2003.

[13] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang. A simple algorithm for tracking elephant flows,

and taking advantage of power laws. *Proceedings of Allerton Conference*, 2005.

ДОДАТОК А

1.1 Вихідний код програми:

Файл main.c

```
#include <stdio.h>

#include <time.h>

#include <string.h>

#include "bloom.h"

#define NANOSECONDS_IN_SECOND 1000000000

struct timespec begin, end;

/**
 * Strip '\r' or '\n' chars from a string
 */

static int strip(char *string)
{
    char *c;

    if ((c = strchr(string, '\r'))) {
        *c = 0;
    }

    if ((c = strchr(string, '\n'))) {
        *c = 0;
    }
}
```



```

}

/**
 * Ad-hoc command-line spell checker
 */

int main(int argc, char *argv[])
{
    // Open the dictionary file

    FILE *fp;

    if (!(fp = fopen("dictionary.txt", "r"))) {
        fprintf(stderr, "E: Couldn't open words file\n");
        fflush (stderr);

        return 1;
    }

    // Read data from another file

    FILE *file;

    if (!(file = fopen("data.txt", "r"))) {
        fprintf(stderr, "E: Couldn't open words file\n");
        fflush (stderr);

        return 1;
    }

```

```

// Create a bloom filter

bloom_t *filter = bloom_filter_new(2500000);


// Add all dictionary words to the filter

double totalTime = 0;

for (int i = 0; i <= 1000; i++) {

    char *p;

    char line[1024];

    while (fgets(line, 1024, fp)) {

        strip(line);

        bloom_filter_add(filter, line);

    }

    fclose(fp);

    //printf("bloom filter count : %u\n", bloom_filter_count(filter));

    //printf("bloom filter size : %u\n", bloom_filter_size(filter));

    // Read words from stdin and print those words not in the bloom filter

    clock_gettime(CLOCK_REALTIME, &begin);

    while (fgets(line, 1024, file)) {

        strip(line);

```

```

p = strtok(line, " \t,.;\r\n?!-/()");

while (p) {

    if (!bloom_filter_contains(filter, p)) {

        // printf("%s\n", p);

    }

    p = strtok(NULL, " \t,.;\r\n?!-/()");

}

// Cleanup

clock_gettime(CLOCK_REALTIME, &end);

long long int time_spent = NANoseconds_IN_SECOND *
(end.tv_sec - begin.tv_sec) + (end.tv_nsec - begin.tv_nsec);

//printf("Time spent: %.7lf sec\n", (double) time_spent /
NANoseconds_IN_SECOND);

totalTime += (double) time_spent / NANoseconds_IN_SECOND;

//printf("Time spent: %.7lf sec\n", totalTime / 1000);

}

printf("Time spent: %.7lf sec\n", totalTime / 1000);

bloom_filter_free(filter);

return 0;

```

```
}}
```

Файл bloom.c

```
#include <limits.h>
```

```
#include <stdarg.h>
```

```
#include "bloom.h"
```

```
#include "hashes.h"
```

```
#define SETBIT(bitset, i) (bitset[i / CHAR_BIT] |= (1 << (i % CHAR_BIT)))
```

```
#define GETBIT(bitset, i) (bitset[i / CHAR_BIT] & (1 << (i % CHAR_BIT)))
```

```
/*
```

Bloom filter type implementation

```
*/
```

```
struct _bloom_t
```

```
{
```

```
    unsigned char *bitset;    // The bitset data structure
```

```
    size_t size;              // The size of the bitset
```

```
    size_t count;             // The number of keys in the bloom filter
```

```
    bloom_hashfunc *functions; // The array of hash function pointers
```

```
    size_t num_functions;     // The number of hash function pointers
```

```
};
```

```

/*

* Internal bloom filter builder function

*/

bloom_t *bloom_filter_builder(size_t size, size_t num_functions, ...)

{

    bloom_t *filter;

    va_list valist;

    if (!(filter = malloc(sizeof(bloom_t)))) {

        return NULL;

    }

    if (!(filter->bitset = calloc((size + CHAR_BIT-1) / CHAR_BIT,
sizeof(char)))) {

        free(filter);

        return NULL;

    }

    if (!(filter->functions = malloc(num_functions *
sizeof(bloom_hashfunc)))) {

        free(filter->bitset);

        free(filter);

        return NULL;

    }

```

```

    va_start(valist, num_functions);

    int i; for (i = 0; i < num_functions; ++i) {

        filter->functions[i] = va_arg(valist, bloom_hashfunc);

    }

    va_end(valist);

    filter->num_functions = num_functions;

    filter->size = size;

    filter->count = 0;

    return filter;

}

/*

* Allocate a new bloom filter

*/

bloom_t *bloom_filter_new(size_t size)

{

    return bloom_filter_builder(

        size, 4, jenkins_hash, murmur_hash, sax_hash, sdbm_hash);

}

```

```
/*  
  
 * Free and allocated bloom filter  
  
 */  
  
int bloom_filter_free(bloom_t *filter)  
{  
  
    if (!filter) {  
  
        return 0;  
  
    }  
  
    if (filter->bitset) {  
  
        free(filter->bitset);  
  
    }  
  
    if (filter->functions) {  
  
        free(filter->functions);  
  
    }  
  
    free(filter);  
  
    return 1;  
  
}  
  
/*  
  
 * Add a key to a bloom filter  
  
 */
```

```

int bloom_filter_add(bloom_t *filter, const char *key)
{
    if (!filter || !key) {
        return 0;
    }

    int i; for (i = 0; i < filter->num_functions; ++i) {
        SETBIT(filter->bitset, filter->functions[i](key) % filter->size);
    }

    ++(filter->count);

    return 1;
}

/*
 * Check bloom filter membership
 */

int bloom_filter_contains(bloom_t *filter, const char *key)
{
    if (!filter || !key) {
        return 0;
    }

    int i; for (i = 0; i < filter->num_functions; ++i) {

```



```

        if (!(GETBIT(filter->bitset, filter->functions[i](key) % filter->size))) {

            return 0;

        }

    }

    return 1;

}

/*

* The number of keys in the bloom filter

*/

size_t bloom_filter_count(bloom_t *filter)

{

    return filter->count;

}

/*

* The size of the bloom filter

*/

size_t bloom_filter_size(bloom_t *filter)

{

    return filter->size;

}

```

```
}
```

Файл bloom.h

```
#ifndef BLOOM_H_INCLUDED
```

```
#define BLOOM_H_INCLUDED
```

```
#include <stdlib.h>
```

```
/*
```

```
 * Hash function pointer
```

```
*/
```

```
typedef unsigned int(*bloom_hashfunc)(const char*);
```

```
/*
```

```
 * Opaque type definition
```

```
*/
```

```
typedef struct _bloom_t bloom_t;
```

```
/*
```

```
 * Allocate a new bloom filter
```

```
*/
```

```
bloom_t *bloom_filter_new(size_t);
```

```
/*
```

```
 * Free and allocated bloom filter
```

```
*/
```

```
int bloom_filter_free(bloom_t*);
```

```
/*
```

```
 * Add a key to a bloom filter
```

```
*/
```

```
int bloom_filter_add(bloom_t*, const char*);
```

```
/*
```

```
 * Check bloom filter membership
```

```
*/
```

```
int bloom_filter_contains(bloom_t*, const char*);
```

```
/*
```

```
 * The number of keys in the bloom filter
```

```
*/
```

```
size_t bloom_filter_count(bloom_t*);
```

```

/*

* The size of the bloom filter

*/

size_t bloom_filter_size(bloom_t*);


#endif // BLOOM_H_INCLUDED

Файл hashes.h

#ifndef HASHES_H_INCLUDED
#define HASHES_H_INCLUDED

#include <string.h>

/*

* SAX hash function

*/

static unsigned int sax_hash(const char *key)
{
    unsigned int h = 0;

    while (*key) {
        h ^= (h << 5) + (h >> 2) + (unsigned char) *key;
    }
}

```

```

        ++key;

    }

    return h;
}

/*

* SDBM hash function

*/

static unsigned int sdbm_hash(const char *key)
{
    unsigned int h = 0;

    while (*key) {
        h = (unsigned char) *key + (h << 6) + (h << 16) - h;

        ++key;
    }

    return h;
}

/*

* Murmur2 hash function

*/

```

```

static unsigned murmur_hash(const char *key)
{
    if (!key) {
        return 0;
    }

    unsigned m = 0x5bd1e995;

    unsigned r = 24;

    unsigned seed = 0xdeadbeef;

    size_t len = strlen(key);

    unsigned h = seed ^ len;

    while (len >= 4) {

        unsigned k = *(unsigned*)key;

        k *= m;

        k ^= k >> r;

        k *= m;

        h *= m;

        h ^= k;

        key += 4;

        len -= 4;

    }

    switch(len) {

```

```

        case 3:

            h ^= key[2] << 16;

        case 2:

            h ^= key[1] << 8;

        case 1:

            h ^= key[0];

            h *= m;

    };

    h ^= h >> 13;

    h *= m;

    h ^= h >> 15;

    return h;

}

/*

* Jenkins hash function

*/

static unsigned jenkins_hash(const char *key)

{

    if (!key) {

        return 0;

```

```
    }

    unsigned hash, i;

    size_t len = strlen(key);

    for (hash = i = 0; i < len; ++i) {

        hash += key[i];

        hash += (hash << 10);

        hash ^= (hash >> 6);

    }

    hash += (hash << 3);

    hash ^= (hash >> 11);

    hash += (hash << 15);

    return hash;

}
```

```
#endif // HASHES_H_INCLUDED
```