ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

**Фильтр Блума**

|  |  |
| --- | --- |
| Выполнил студент | Игнатенко Георгий Дмитриевич |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС-242 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ассистент Кафедры ВС Насонова А. О. |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2023

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#__RefHeading___Toc4327_3153689509)

[Описание структуры данных 5](#__RefHeading___Toc4327_3153689509)

[Описание основных операций 7](#__RefHeading___Toc4327_3153689509)

1. [Вставка элемента 6](#__RefHeading___Toc4327_3153689509)
2. [Поиск элемента 7](#__RefHeading___Toc4327_3153689509)

[Вероятность ложноположительного срабатывания 10](#__RefHeading___Toc4327_3153689509)

[АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ 11](#__RefHeading___Toc4327_3153689509)

1. [Вставка элемента 12](#__RefHeading___Toc4339_3153689509)

[2.  Поиск элемента 13](#__RefHeading___Toc4341_3153689509)

[ЗАКЛЮЧЕНИЕ 14](#__RefHeading___Toc4343_3153689509)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 15](#__RefHeading___Toc4345_3153689509)

[ПРИЛОЖЕНИЕ 16](#__RefHeading___Toc4347_3153689509)

**ВВЕДЕНИЕ**

Фильтр Блума (англ. Bloom filter) — это вероятностная структура данных, придуманная Бёртоном Блумом в 1970 году, позволяющая проверять принадлежность элемента к множеству. При этом существует возможность получить ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное.

Фильтр Блума может использовать любой объём памяти, заранее заданный пользователем, причём чем он больше, тем меньше вероятность ложного срабатывания. Поддерживается операция добавления новых элементов в множество, но не удаления существующих (если только не используется модификация со счётчиками).

Примеры практических применений:

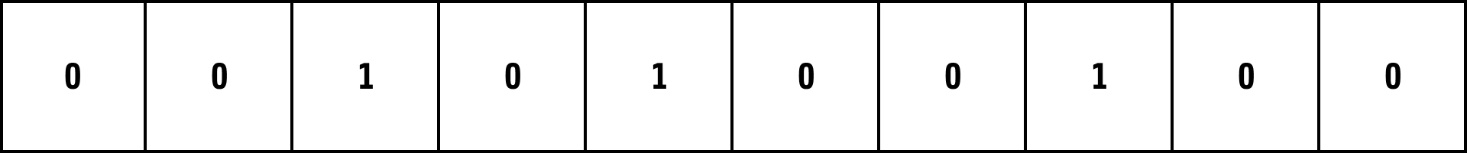
* Прокси-сервер Squid использует фильтры Блума для опции cache digests.
* Google BigTable использует фильтры Блума для уменьшения числа обращений к жесткому диску при проверке на существование заданной строки или столбца в таблице базы данных.
* Компьютерные программы для проверки орфографии.
* Medium использует фильтры bloom для рекомендации публикации пользователям путем фильтрации сообщений, которые были просмотрены пользователем.
* Quora реализовала общий фильтр bloom в обратной связи и для фильтрации историй, которые люди видели раньше.
* Веб-браузер Google Chrome использовал фильтр Bloom для выявления вредоносных URL-адресов
* Google BigTable, Apache HBase и Apache Cassandra, а также Postgresql используют фильтры Bloom, чтобы сократить время поиска на диске несуществующих строк или столбцов

Описание структуры данных

Фильтр Блума представляет собой битовый массив из m бит. Изначально, когда структура данных хранит пустое множество, все m бит обнулены. Пользователь должен определить k независимых хеш-функций h1, …, hk, каждая из которых отображает множество элементов в множество мощностью m. (Иными словами, каждому элементу хеш-функция сопоставляет число от 1 до m.) Для каждого элемента e биты массива с номерами h1(e), …, hk(e) равными значениям хеш-функций устанавливаются в 1.

Для проверки принадлежности элемента e к множеству хранимых элементов необходимо проверить состояние битов h1(e), …, hk(e). Если хотя бы один из них равен нулю, элемент не может принадлежать множеству (иначе бы при его добавлении все эти биты были установлены). Если все они равны единице, то структура данных сообщает, что е может принадлежать множеству. При этом может возникнуть две ситуации: либо элемент действительно принадлежит множеству, либо все эти биты оказались установлены по случайности при добавлении других элементов, что и является источником ложных срабатываний в этой структуре данных.

Независимость хеш-функций обеспечивает минимальную вероятность повторения индексов hk(e), минимизируя число бит установленных в 1 несколько раз. (А это главный источник ложноположительных срабатываний.)



Пример фильтра Блума с m = 18 �=18 mи �=3k = 3, хранящего множество {*x*, *y*, *z*}. Цветные стрелки указывают на места в битовом массиве, соответствующие каждому элементу множества. Этот фильтр Блума может определить, что элемент *w* не входит в множество, так как один из соответствующих ему битов равен нулю.

**ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ**

**1. Вставка элемента**

|  |  |
| --- | --- |
| Алгоритм 1 bloom\_insert | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **void** **bloom\_insert**(**bool** \*bitarray, **int** arrSize, **char** \*key) {  **int** a = hash1(key, arrSize);  **int** b = hash2(key, arrSize);  **int** c = hash3(key, arrSize);  **int** d = hash4(key, arrSize);  **if** (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d]){  **return**;  }**else** {  bitarray[a] = true;  bitarray[b] = true;  bitarray[c] = true;  bitarray[d] = true;  }  }; |

Для того, что бы вставить в дерево новый ключ, требуется вычислить ключ через хеш-функции, проверить есть ли элемент во множестве, затем поставить 1 в ячейках битового массива.

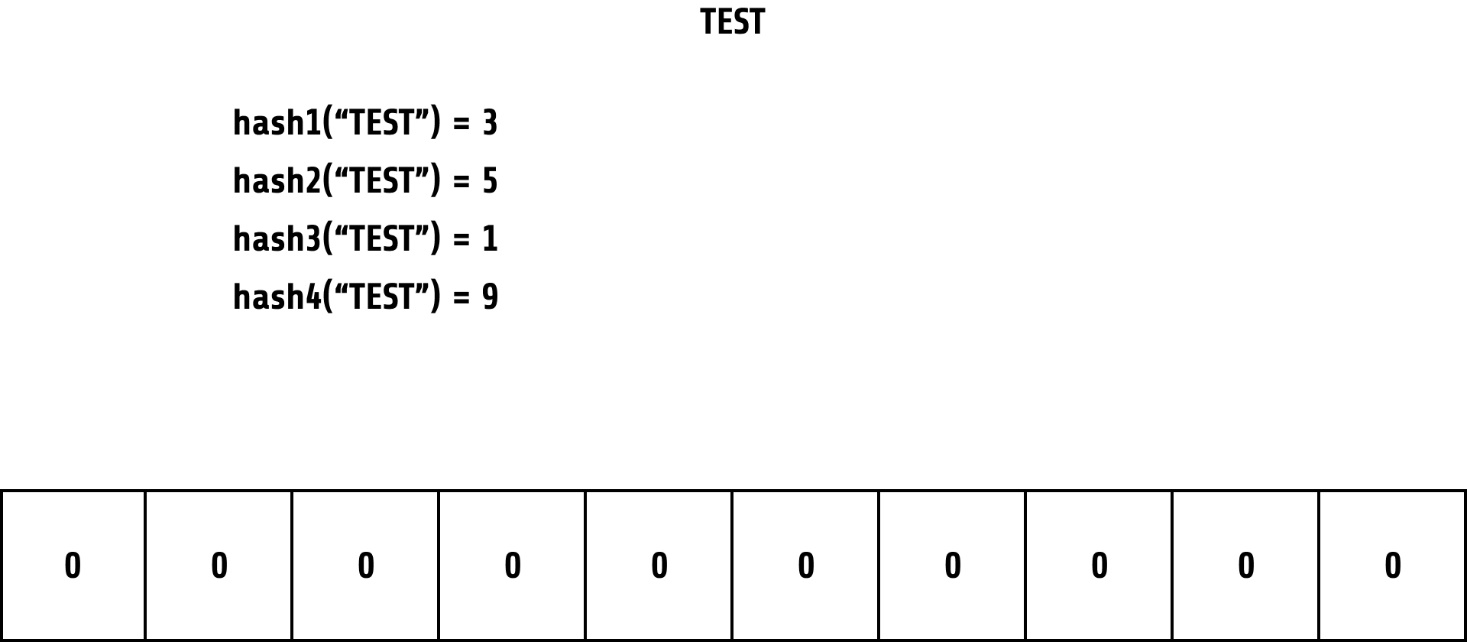


Рисунок 1. Вычисление хэш функции.

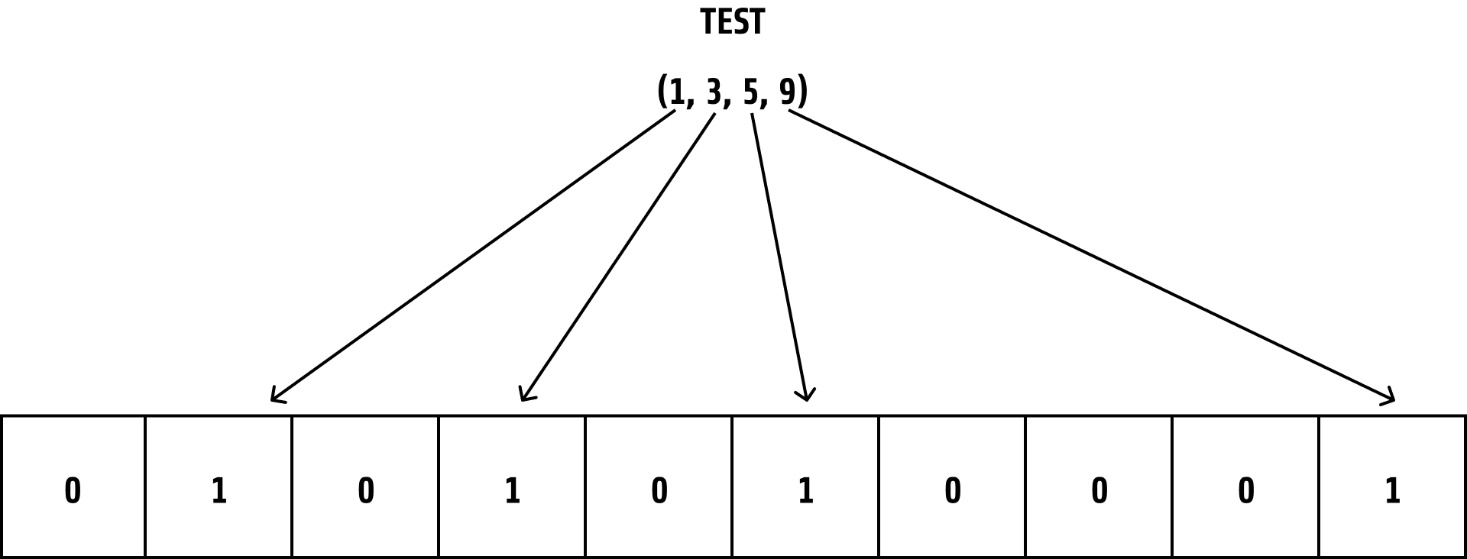


Рисунок 2. Вставка 1 в ячейки.

**2.Поиск элемента**

Для того, чтобы понять, есть ли элемент во множестве, нужно вычислить хеш и проверить, если во всех ячейках битового массива стоят 1, то элемент возможно во множестве, если нет, то элемент точно не во множестве.

|  |  |
| --- | --- |
| Алгоритм 2 bloom\_lookup | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **bool** **bloom\_lookup**(**bool** \*bitarray, **int** arrSize, **char** \*key) {  **int** a = hash1(key, arrSize);  **int** b = hash2(key, arrSize);  **int** c = hash3(key, arrSize);  **int** d = hash4(key, arrSize);  **if** (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d]){  **return** true;  }**else** {  **return** false;  }  }; |

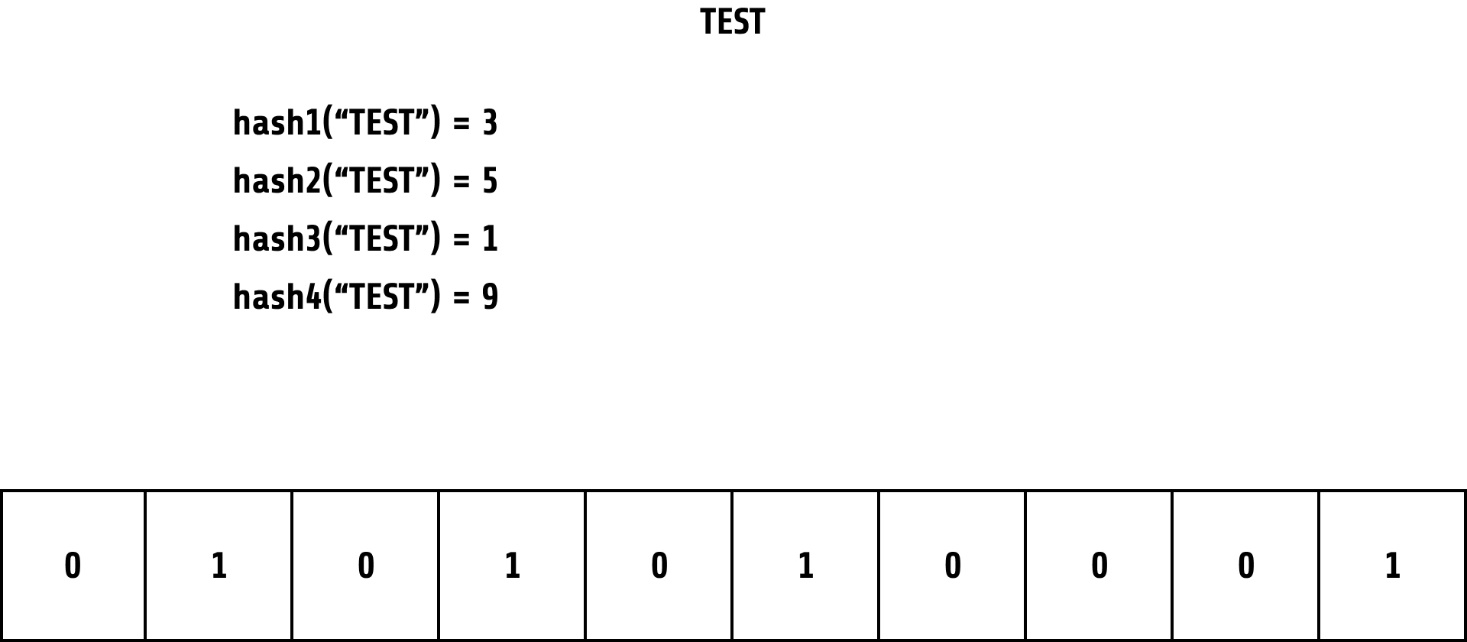


Рисунок 3. Поиск элемента в массиве

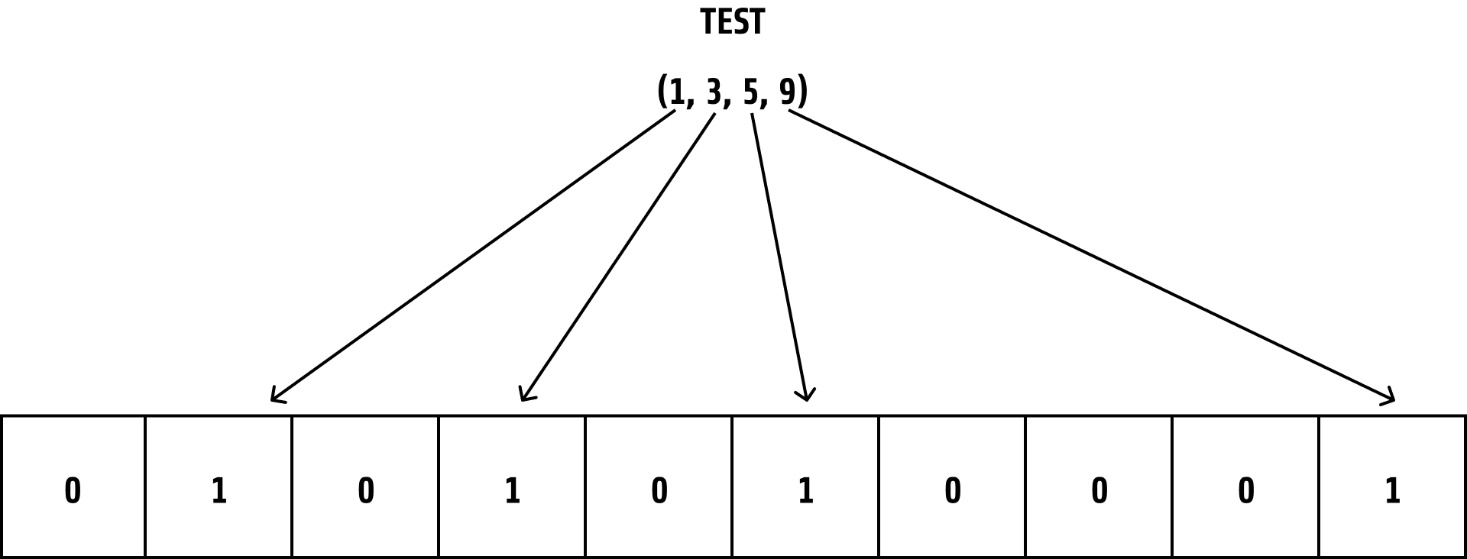


Рисунок 4. Поиск элемента

Вероятность ложноположительного срабатывания

Пусть размер битового массива равен *m* бит и задано *k* хеш-функций. Предположим, что множество хеш-функций выбирается случайно, и для любого элемента *x* каждая хеш-функция *hi* назначает ему одно из мест в битовом массиве с равной вероятностью.

и, кроме того, значения ℎ�(�) являются независимыми в совокупности случайными величинами (для упрощения последующего анализа).

Тогда вероятность того, что в некоторый *p*-й бит не будет записана единица во время операции вставки очередного элемента равна

А вероятность того, что *p*-й бит останется равным нулю после вставки *n* различных элементов *x*1, …, *xn* в изначально пустой фильтр Блума равна

для достаточно большого *m* ввиду второго замечательного предела.

Ложноположительное срабатывание состоит в том, что для некоторого элемента *y*, не равного ни одному из вставленных, все *k* бит с номерами *hi*(*y*) окажутся ненулевыми, и фильтр Блума ошибочно ответит, что *y* входит во множество вставленных элементов. Вероятность такого события примерно равна

Очевидно, что эта вероятность уменьшается с ростом *m* (размера битового массива) и увеличивается с ростом *n* (числа вставленных элементов). Для фиксированных *m* и *n* оптимальное число *k* (число хеш-функций), минимизирующих её, равно (1−�−��/�)�.

При этом сама вероятность ложного срабатывания равна

Как следствие, заметим, что для того, чтобы фильтр Блума поддерживал заданную ограниченную вероятность ложного срабатывания, размер битового массива должен быть линейно пропорционален числу вставленных элементов.

Сложность алгоритма

Время

Временная сложность вставки и поиске элемента в фильтре Блума с n-количеством элементов и k-количеством хеш-функций составляет O(k). В обоих случаях нам нужно только провести ввод через все хеш-функций.

|  |  |
| --- | --- |
| Операция | Сложность |
| insertion | O(k) |
| search | O(k) |

Пространственная сложность

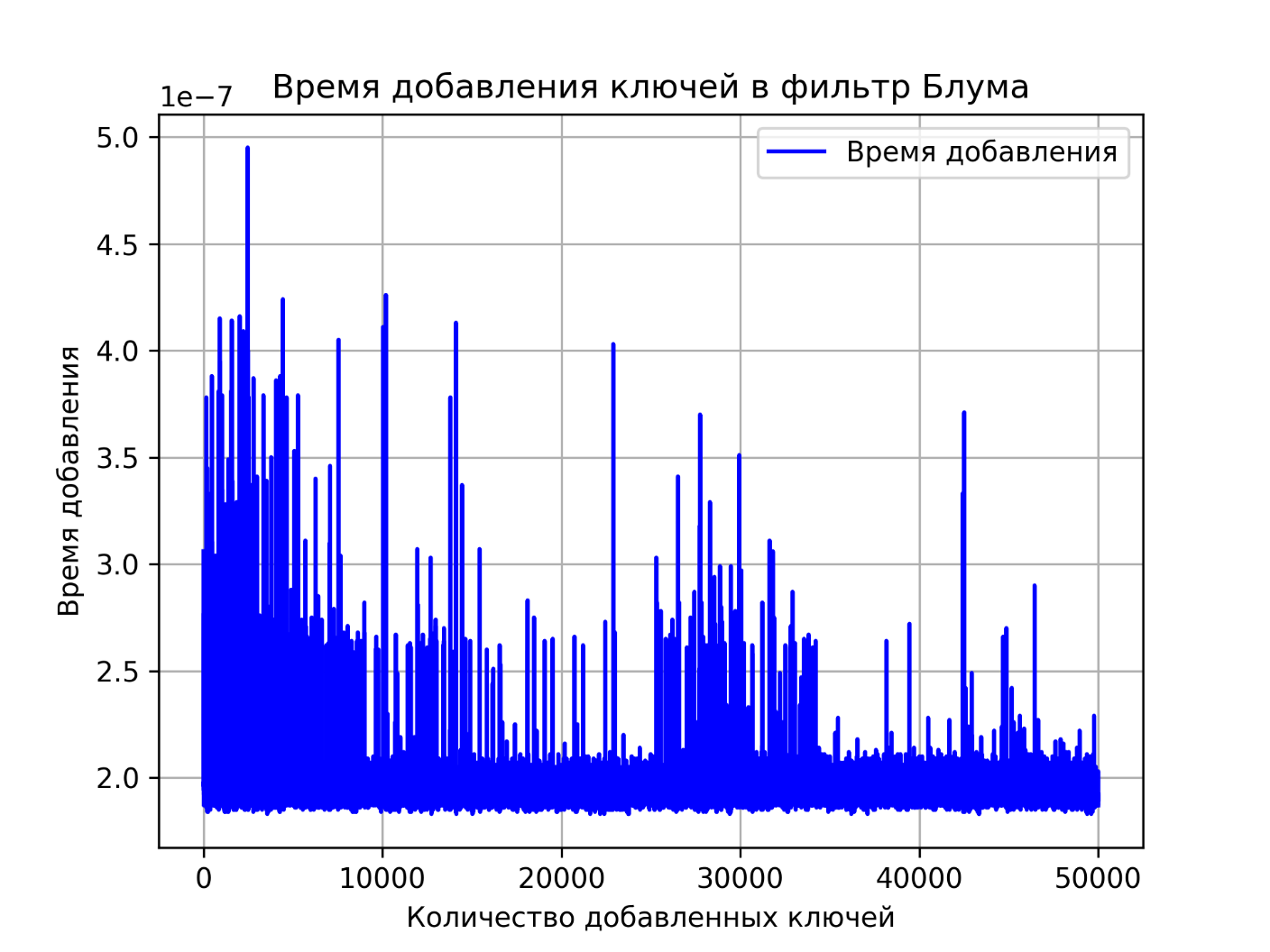
Память занимаемая фильтром Блума будет зависеть от максимального размера фильтра n.

|  |  |
| --- | --- |
|  | Сложность |
| Память | O(n) |

2 Экспериментальное исследование эффективности алгоритма

2.1 Организация моделирования

Было замерено время добавления слов, поиска в фильтр блума.





2.2 Результаты моделирования

В результате получилось, что среднее время добавления элементов примерно равно, не возрастает от количества добавленных элементов.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения данной курсовой работы была реализована и проанализирована такая структура данных, как фильтр Блума, которая способна предположить есть ли элемент в коллекции за константное время, эффективно используя память.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. *Wikipedia.* [*https://ru.wikipedia.org/wiki/Фильтр\_Блума*](https://ru.wikipedia.org/wiki/Фильтр_Блума)
2. *Курносов М.Г., Берлизов Д.М.* К93 Алгоритмы и структуры обработки информации. – Новосибирск:Параллель, 2019. -C. 161-177.

**ПРИЛОЖЕНИЕ**

Исходный код программы

main.c

#include "bloom.h"

#define size 50

**int** **main**() {

**bool** bitarray[size] = { false };

**char**\* words[] = {"test", "test2", "math", "dsa",

"bloom", "bonus"};

**int** c = **0**;

**for** (**int** i = **0**; i < **sizeof**(words)/**sizeof**(words[**0**]); i++){

bloom\_insert(bitarray, size, words[i]);

printf("Added element: %s**\n**", words[i]);

c++;

}

printf("**\n**");

printf("False positive possibility is %f%%**\n**", bloom\_probability(size, c));

**char**\* checkupWords[] = {"test", "test2", "test3", "sda", "boom", "bones"};

**for** (**int** i = **0**; i < **sizeof**(checkupWords)/**sizeof**(checkupWords[**0**]); i++){

**if** (bloom\_lookup(bitarray, size, checkupWords[i]))

printf("Word: %s probably in set**\n**", checkupWords[i]);

**else**

printf("Word: %s definetly not in set**\n**", checkupWords[i]);

}

**return** **0**;

}

**tests.c**

#include "bloom.h"

#include "time.h"

#define size 500000

**int** **main**() {

**bool** bitarray[size] = { false };

**char** words[size/**10**][**7**];

**FILE** \*f = fopen("./data/out", "r");

**int** i = **0**;

**while**(i < size/**10**){

fscanf(f, "%s**\n**", words[i]);

words[i][**6**] = '\0';

i++;

}

fclose(f);

printf("**\n**");

**FILE** \*out = fopen("./data/insertion\_time.csv", "w");

fprintf(out, "n,t,w**\n**");

**struct** timespec start, end;

**for** (**int** i = **0**; i < **sizeof**(words)/**sizeof**(words[**0**]); i++){

clock\_gettime(CLOCK\_MONOTONIC, &start);

bloom\_insert(bitarray, size, words[i]);

clock\_gettime(CLOCK\_MONOTONIC, &end);

**long** **long** duration = (end.tv\_sec - start.tv\_sec) \* **1000000000** + (end.tv\_nsec - start.tv\_nsec);

**double** durationInSec = (**double**)duration/**1000000000**;

fprintf(out, "%d, %.9f, %s**\n**", i, durationInSec, words[i]);

}

fclose(out);

printf("**\n**");

out = fopen("./data/lookup\_time.csv", "w");

fprintf(out, "n,t,w**\n**");

**for** (**int** i = **0**; i < **sizeof**(words)/**sizeof**(words[**0**]); i++){

clock\_gettime(CLOCK\_MONOTONIC, &start);

bloom\_lookup(bitarray, size, words[i]);

clock\_gettime(CLOCK\_MONOTONIC, &end);

**long** **long** duration = (end.tv\_sec - start.tv\_sec) \* **1000000000** + (end.tv\_nsec - start.tv\_nsec);

**double** durationInSec = (**double**)duration/**1000000000**;

fprintf(out, "%d, %.9f, %s**\n**", i, durationInSec, words[i]);

}

fclose(out);

printf("**\n**");

printf("False positive possibility is %f%%**\n**", bloom\_probability(size, i));

**return** **0**;

}

**hash.h**

#pragma once

#include <math.h>

**int** **hash1**(**char** \*key, **int** arrSize);

**int** **hash2**(**char** \*key, **int** arrSize);

**int** **hash3**(**char** \*key, **int** arrSize);

**int** **hash4**(**char** \*key, **int** arrSize);

**hash.c**

#include "hash.h"

**int** **hash1**(**char** \*key, **int** arrSize) {

**long** **long** **int** h = **0**;

**for** (**int** i = **0**; key[i] != '\0'; i++){

h = (h \* **127** + key[i]) % arrSize;

}

**return** h;

};

**int** **hash2**(**char** \*key, **int** arrSize) {

**long** **long** **int** hash = **0**;

**for** (**int** i = **0**; key[i] != '\0'; i++)

{

hash = (hash + ((**int**)key[i]));

hash = hash % arrSize;

}

**return** hash;

};

**int** **hash3**(**char** \*key, **int** arrSize) {

**long** **long** **int** hash = **1**;

**for** (**int** i = **0**; key[i] != '\0'; i++){

hash = hash + pow(**19**, i) \* key[i];

hash = hash % arrSize;

}

**return** hash % arrSize;

};

**int** **hash4**(**char** \*key, **int** arrSize) {

**long** **long** **int** hash = **3**;

**int** p = **7**;

**for** (**int** i = **0**; key[i] != '\0'; i++){

hash += hash \* **7** + key[**0**] \* pow(p, i);

hash = hash % arrSize;

}

**return** hash;

};

**bloom.h**

#pragma once

#include <stdbool.h>

#include <stdio.h>

#include <math.h>

#include "hash.h"

**void** **bloom\_insert**(**bool** \*bitarray, **int** arrSize, **char** \*key);

**bool** **bloom\_lookup**(**bool** \*bitarray, **int** arrSize, **char** \*key);

**double** **bloom\_probability**(**int** arrSize, **int** elementsCount);

**bloom.c**

#include "bloom.h"

#define hash\_count 4

**void** **bloom\_insert**(**bool** \*bitarray, **int** arrSize, **char** \*key) {

**int** a = hash1(key, arrSize);

**int** b = hash2(key, arrSize);

**int** c = hash3(key, arrSize);

**int** d = hash4(key, arrSize);

**if** (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d]){

**return**;

}**else** {

bitarray[a] = true;

bitarray[b] = true;

bitarray[c] = true;

bitarray[d] = true;

}

};

**bool** **bloom\_lookup**(**bool** \*bitarray, **int** arrSize, **char** \*key) {

**int** a = hash1(key, arrSize);

**int** b = hash2(key, arrSize);

**int** c = hash3(key, arrSize);

**int** d = hash4(key, arrSize);

**if** (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d]){

**return** true;

}**else** {

**return** false;

}

};

**double** **bloom\_probability**(**int** arrSize, **int** elementsCount){

**long** **double** falseProbability = **0.0**;

falseProbability = pow((**1** - pow((**1** - **1.0**/arrSize),hash\_count\*elementsCount)),hash\_count);

**return** falseProbability;

}