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 | Григорьев Юрий Вадимович |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС-142 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ст. преп. Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

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

Новосибирск – 2022

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#__RefHeading___Toc4327_3153689509)

[1 Структура фильтра Блума 3](#__RefHeading___Toc4329_3153689509)

[1.1 Составные компоненты 3](#__RefHeading___Toc4331_3153689509)

[1.2 Алгоритм работы 5](#__RefHeading___Toc4333_3153689509)

[1.3 Оптимальные значения параметров фильтра Блума 5](#__RefHeading___Toc4335_3153689509)

1.4 Выводы……………………………………………………………………………5

[2 Экспериментальное исследование эффективности структуры данных 6](#__RefHeading___Toc4337_3153689509)

[2.1 Организация моделирования 6](#__RefHeading___Toc4339_3153689509)

[2.2 Результаты моделирования 6](#__RefHeading___Toc4341_3153689509)

[ЗАКЛЮЧЕНИЕ 7](#__RefHeading___Toc4343_3153689509)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 8](#__RefHeading___Toc4345_3153689509)

[ПРИЛОЖЕНИЕ 9](#__RefHeading___Toc4347_3153689509)

[1 Исходный код программы 9](#__RefHeading___Toc4349_3153689509)

1.1 Файл "main.c"……………………………………………………………………11

1.2 Файл "bits.c"……………………………………………………………………..11

1.3 Файл "bits.h"……………………………………………………………………..11

1.4 Файл "bloom.c"…………………………………………………………………..11

1.5 Файл "bloom.h"………………………………………………………………….11

ВВЕДЕНИЕ

Работа с большим массивом данных требует больших вычислительных мощностей. Для того, чтобы отыскать какой-то объект в массиве/словаре/списке (например, по строковому ключу) сервер может затратить много драгоценного времени и электроэнергии. В случае, когда требуется проверить лишь отсутствие элемента в структуре данных (таблице/массиве/дереве), поиск целого элемента (со всеми его полями и свойствами) становится и вовсе бессмысленным. Чтобы сократить время обращений к «пустым» записям и освободить много драгоценного места в памяти, хоть и ценой вероятности ложноположительного результата «поиска», Бертон Говард Блум придумал собственную структуру данных, назвав его фильтром Блума, по своей же фамилии, описав его в статье *Space/time trade-offs in hash coding with allowable errors* журнала *Communications of the ACM* в 1970 году.

1 Структура фильтра Блума

1.1 Составные компоненты

Фильтр задействует два главных (и единственных) компонента — битовый массив (рис. 1.1) и некоторое фиксированное количество хеш-функций.

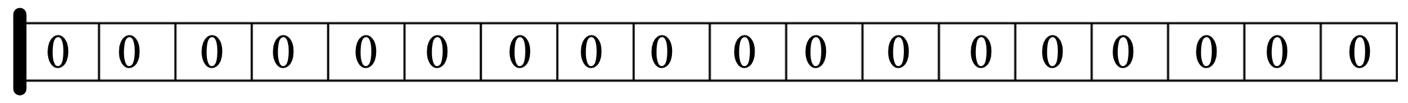


Рисунок 1.1 — Массив битов

Как и для хеш-таблицы, взятое множество хеш-функций должно давать равномерное распределение результатов хеширования и минимальное число коллизий. Если разработчик не хочет сильно утруждаться «придумыванием» множества хеш-функций, можно добавлять/отнимать от составных частей хеш-функции () некоторый итератор (1, 2, 3… для соответствующих вызовов одного и того же ключа), для чего нужна хеш-функция с подобающим лавинным эффектом.

1.2  Алгоритм работы

Создание фильтра: выделяется память под битовый массив некоторого размера, исходя из количества элементов и количества хеш-функций. Все биты в массиве битов приравниваются к 0 (демонстрация созданного фильтра Блума — рис.1.1).

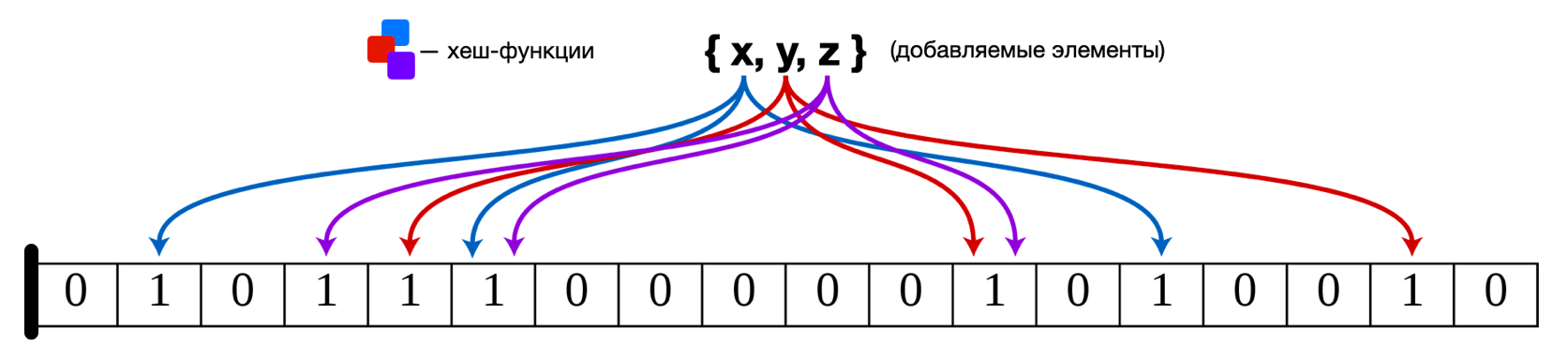
Добавление элемента: строковый ключ подаётся в хеш-функции, на выход программа получает результат в виде индексов в массиве битов. Биты на данных местах устанавливаются в 1. Модификация: перед добавлением проверить, нет ли данного элемента уже в множестве.

Рисунок 1.2 — добавление элементов в фильтр Блума

Проверка на вхождение в множество (поиск): строковый элемент подаётся в хеш-функции, на выход программа получает результат в виде индексов в массиве битов. Проверяется, все ли биты на соответствующих позициях равны 1. Если да, элемент возможно есть в фильтре. Если нет (хотя бы один из битов равен 0) — элемента гарантировано нет в фильтре (рис.1.3), а на местах, где биты были равны 1, мы столкнулись с результатом хеша для других элементов (коллизия) (рис.1.4).

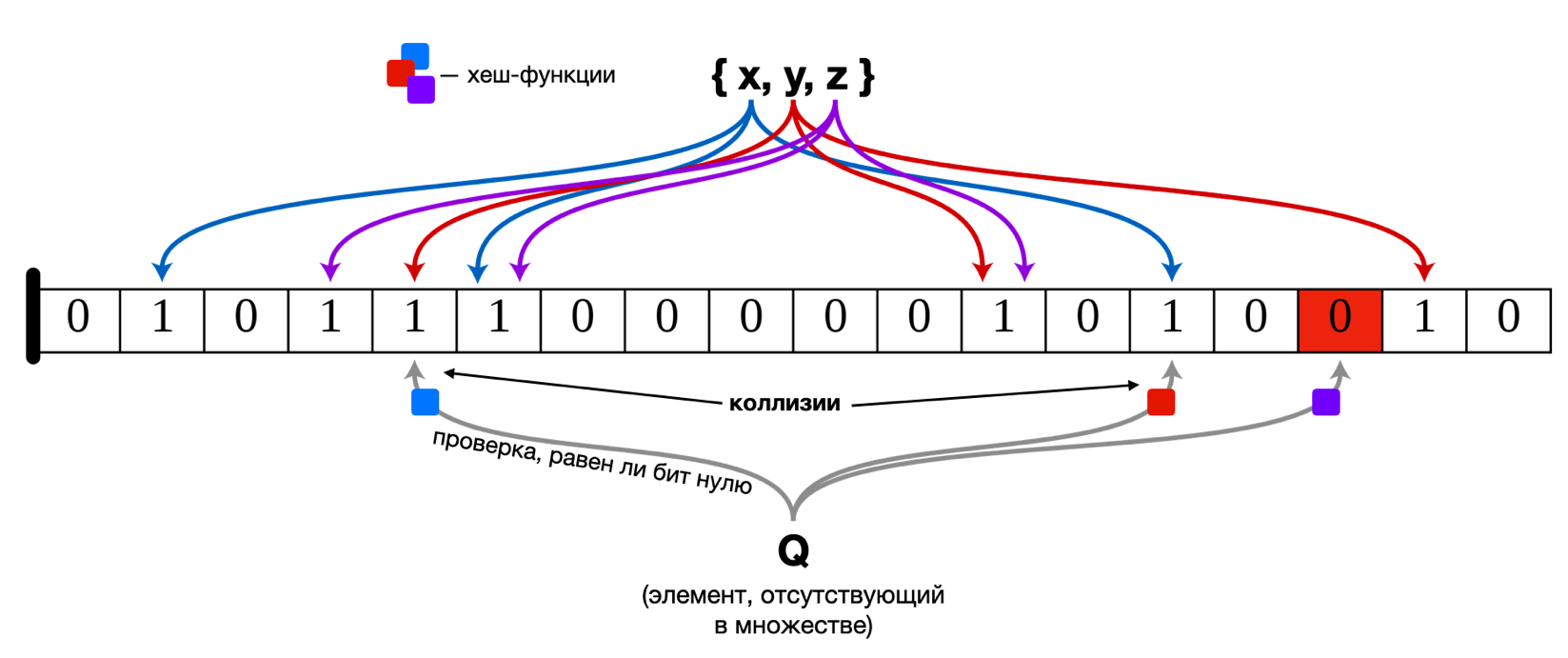


Рисунок 1.3 — проверка строкового ключа на вхождение в множество (случай 1)

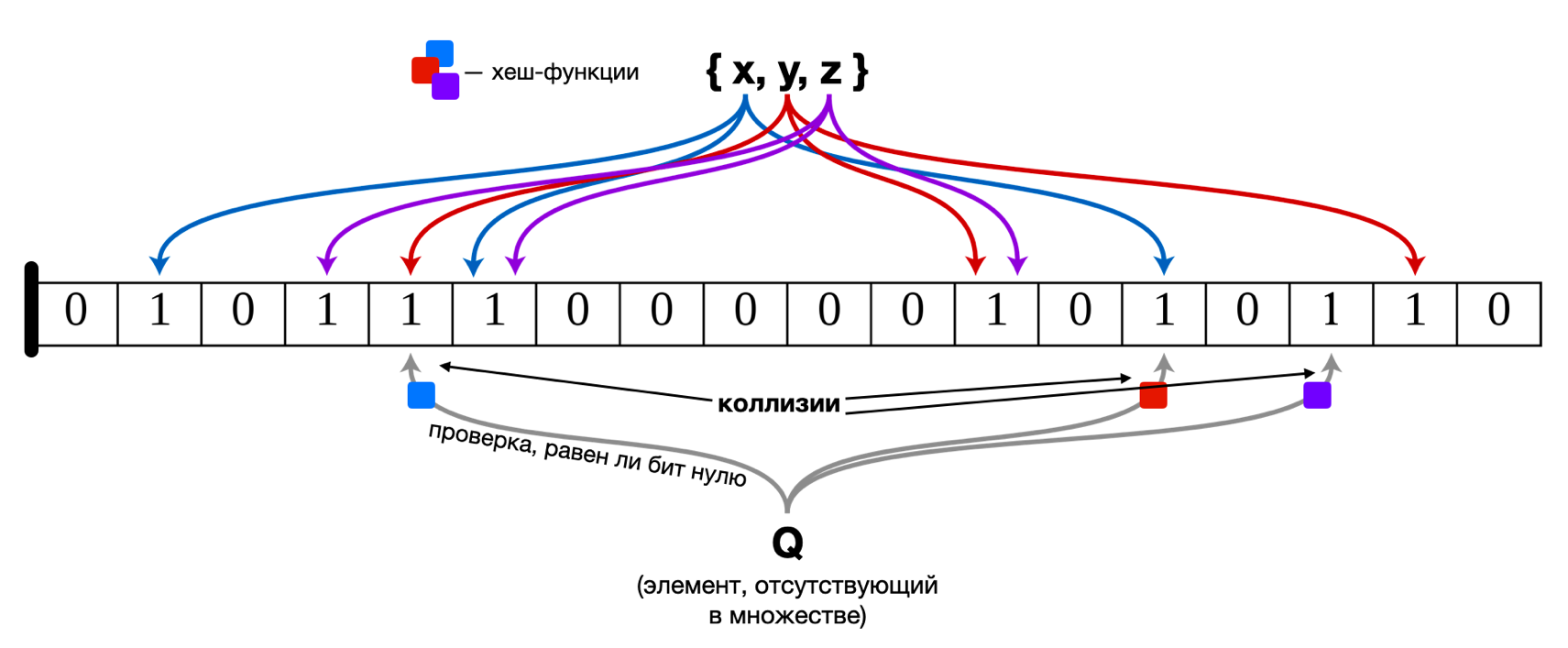


Рисунок 1.4 — проверка строкового ключа на вхождение в множество (случай 2)

1.3 Оптимальные значения параметров фильтра Блума

Главный недостаток фильтра Блума — вероятность получения ложноположительного результата, из-за чего разработчику необходимо подобрать оптимальные параметры для размера массива битов и количества хеш-функций: даже если взять слишком много того и другого (что непременно даст положительный результат в уменьшении вероятности ложноположительного результата), программа задействует неоптимальный объём памяти и будет тратить больше времени на заполнение массива битов.

Прибегнем к математическому анализу данной проблемы:

*m* — размер битового массива (фильтра Блума)

*n* — количество элементов

*k* — количество хеш-функций

— вероятность хеш-функции не установить определённый бит в 1

*—* вероятность для *k* хеш-функций не установить определённый бит в 1

—вероятность того, что после вставки *n* элементов *k* хеш-функциями определённый бит равен 0

— вероятность того, что после вставки *n* элементов *k* хеш-функциями определённый бит равен 1

Таким образом, — вероятность ложноположительного срабатывания «поиска» в фильтре Блума (вероятность того, что для отсутствующего элемента каждая позиция (бит), вычисленная *k* хеш-функциями, будет равна 1).

*=> k —* оптимальноеколичество хеш-функций, минимизирующее ложноположительные срабатывания ( – количество битов на один элемент) => подстановка: =>

Выражаем (количество битов на элемент): , а далее выражаем *m* (размер фильтра): .

Мы нашли оптимальные значения параметров для фильтра Блума с любым количеством элементов и вероятностью ложноположительного срабатывания.

1.4 Выводы

Операции добавления и «поиска» (на самом же деле — проверки на отсутствие в фильтре) некоторого ключа в фильтре Блума сопоставимы с теми же операциями в хеш-таблицах: они задействуют только хеш-функции, чья вычислительная сложность равна **O(1)** или, если более точно, **O(*m*)**, где *m* – длина строкового ключа.

2 Экспериментальное исследование эффективности алгоритма

2.1 Организация моделирования

Хеш-функция, вариации которой используются для получения индексов в массив битов — FNV-1a. Она не является криптографической (что в нашем случае и не требуется: вряд ли кто-то будет взламывать фильтр Блума, ведь его единственное назначение — проверка строкового ключа на отсутствие в фильтре) и показывает хорошие результаты в равномерном распределении значений, а соответственно и в меньшем количестве коллизий. Также FNV-1a более адаптивна к изменению её базового значения (*hash\_base* или *hash\_value*), что позволяет нам, например, отнимать от этого значения наш итератор без ухудшения лавинного эффекта функции и распределения её хеша (главную роль в FNV-1a всё же играет простое число *FNV\_32\_PRIME*).

Для работы с битами был создан отдельный модуль bits.c, в котором обозначены функции задания конкретного бита в 1 (*set\_bit()*), проверки его значения (*get\_bit()*) и очистки (установки в 0) (*clear\_bit()*). Базой для работы с битами установлен тип *uint\_8t*, который содержит в записи натурального числа ровно 8 бит. Соответственно, битовый массив на от 1 до 8 элементов будет содержать лишь одно такое число, от 9 до 16 — 2 числа типа *uint8\_t* и т.д.

Модуль для работы с фильтром Блума — *bloom.c*, в котором описана структура фильтра (массив битов — *uint8\_t \*bits*, размер фильтра *m*, количество хеш-функций *k*) и функции для работы с ним: *bloom\_init()* для инициализации фильтра Блума, принимающая на вход количество элементов и желаемую вероятность получения ложноположительного результата — в ней также используются функции *bloom\_optimalFilterSize()* и *bloom\_optimalHashNumber()* для нахождения оптимальных значений размера фильтра и количества хеш-функций (максимального итератора) соответственно, *bloom\_lookup()* — для «поиска» строкового ключа в фильтре Блума (проверки на его отсутствие), *bloom\_insert()* для вставки в фильтр и *bloom\_FNVHash()* для получения хеша от заданного ключа и итератора.

В файле тестовой программы (*main.c*) происходит следующее: инициализация нового фильтра Блума для 20 элементов и вероятностью ложноположительного результата в 1% (0,01), далее цикл for на 2 итерации, в котором добавляются строковые ключи. Вторая итерация смоделирована для того, чтобы проверить, добавит ли фильтр один и тот же ключ 2 раза подряд. Как мы можем видеть из результатов, фильтр Блума сообщает нам, что данные элементы уже возможно находятся в фильтре, так как при добавлении также используется функция *bloom\_lookup()*. После проверки всех уже находящихся в фильтре элементов, программа проверяет сторонние ключ на вхождение в фильтр. Результат ожидаем — программа сообщает, что данных строковых ключей точно нет в фильтре. Для всех операций над фильтром Блума в терминал выводится лог его работы со специальным префиксом.

2.2 Результаты моделирования

Скриншоты результатов работы программы:

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработан и исследован фильтр Блума.

Осуществлено моделирование структуры данных. Показано, что фильтр Блума временно- и пространственно-эффективен в задаче на гарантированное отсутствие строкового ключа в множестве. Также стоит заметить, что это вероятностная структура данных с возможным получением от неё ложноположительного результата, которая применяется в основном как предварительный фильтр запросов, и для последующего поиска возможно существующего элемента применяются «медленные» и точные алгоритмы поиска. Фильтр Блума создаётся исходя из количества элементов множества и желаемой вероятности ошибки (ложноположительного результата). Он не поддерживает операцию удаления строкового ключа из фильтра, т.к. это может затронуть результаты хеширования других элементов, а значит при их поиске может быть выдан результат «гарантированного отсутствия в фильтре».

Фильтр Блума используется во многих продуктах, связанных с хранением данных (Google BigTable, Apache HBase, Apache Cassandra, PostgreSQL), системами архивного хранения (Venti, WebArchive), блокчейн-системах (Ethereum).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Burton H. Bloom* Space/time trade-offs in hash coding with allowable errors // Communications of the ACM. – 1970. – T. 13. – №7. – С. 422-426.
2. *Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber* Bigtable: A Distributed Storage System for Structured Data // 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). – Berkeley: USENIX, 2006. – С. 205-218.
3. *Adam Kirsch, Michael Mitzenmacher* Less Hashing, Same Performance: Building a Better Bloom Filter, edited by Yossi Azar, Thomas Erlebach, Algorithms // ESA 2006, 14th Annual European Symposium, Lecture Notes in Computer Science, Springer-Verlag, Lecture Notes in Computer Science. – 2006. – T. 4168. – С. 456–467.
4. *Alex Yakunin* Nice Bloom filter application // Alex Yakunin's blog URL: blog.alexyakunin.com (дата обращения: 20.11.2022).
5. *Jamie Talbot* What are Bloom filters? // Medium URL: https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff#.xlkqtn1vy (дата обращения: 22.11.2022).
6. *Michael Mitzenmacher, Eli Upfal* Probability and computing: Randomized algorithms and probabilistic analysis. – Cambridge: Cambridge University Press, 2005. – С. 107–112.

ПРИЛОЖЕНИЕ

1 Исходный код программы

1.1 Файл “main.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | #include "bloom.h"  #include <stdio.h>  **int** **main**() {  printf("**\n**Initialization of filter:**\n**");  BloomFilter \*f = bloom\_init(**20**, **0.01**); // create a filter of 10 elements with falsepositive probability 1%  printf("**\n**Adding fruits:**\n**");  **for**(**int** i = **0**; i < **2**; i++) {  **if**(i == **1**) {  printf("**\n**Checking fruits:**\n**");  }  bloom\_insert(f, "apple");  bloom\_insert(f, "coconut");  bloom\_insert(f, "watermelon");  bloom\_insert(f, "pomegranate");  bloom\_insert(f, "pineapple");  bloom\_insert(f, "grapes");  bloom\_insert(f, "orange");  bloom\_insert(f, "banana");  bloom\_insert(f, "melon");  bloom\_insert(f, "dragonfruit");  bloom\_insert(f, "strawberry");  bloom\_insert(f, "blueberry");  bloom\_insert(f, "blackberry");  bloom\_insert(f, "cherry");  bloom\_insert(f, "apricot");  bloom\_insert(f, "kiwi");  bloom\_insert(f, "lime");  bloom\_insert(f, "peach");  bloom\_insert(f, "lemon");  bloom\_insert(f, "pear");  }  printf("**\n**Checking vegetables:**\n**");  bloom\_lookup(f, "tomato");  bloom\_lookup(f, "aubergine");  bloom\_lookup(f, "potato");  bloom\_lookup(f, "cucumber");  bloom\_lookup(f, "cabbage");  bloom\_lookup(f, "pumpkin");  bloom\_lookup(f, "corn");  bloom\_lookup(f, "onion");  **return** **0**;  } |

1.2 Файл “bits.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include "bits.h"  #include <inttypes.h>  void set\_bit(uint8\_t A[], int k)  {  int i = k / 8; *// index in the array A*  int pos = k % 8; *// bit position in A[i]*  unsigned int flag = 1; *// flag 00001*  flag = flag << pos; *// flag shifted k positions (e.g. 00100)*  A[i] = A[i] | flag; *// set the bit at k position in A[i] to 1*  }  void clear\_bit(uint8\_t A[], int k)  {  int i = k / 8;  int pos = k % 8;  unsigned int flag = 1;  flag = flag << pos;  flag = ~flag; *// negative (e.g. 11011)*  A[i] = A[i] & flag;  }  uint8\_t get\_bit(uint8\_t A[], int k)  {  int i = k / 8;  int pos = k % 8;  unsigned int flag = 1;  flag = flag << pos;  **return** ((A[i] & flag) != 0); *// compare xxXxx & 00100 - getting needed bit*  } |

1.3 Файл “bits.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #pragma once  #include <inttypes.h>  void set\_bit(uint8\_t A[], int k);  void clear\_bit(uint8\_t A[], int k);  uint8\_t get\_bit(uint8\_t A[], int k); |

1.4 Файл “bloom.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | #include "bloom.h"  #include "bits.h"  #include <math.h>  #include <stdio.h>  #include <stdlib.h>  #define BITS\_FOR\_TYPE 8.0 // for uint8\_t  #define FNV\_32\_PRIME 16777619  **char** \*bloom\_prefix = "[bloom filter]";  **unsigned** **int** **bloom\_FNVHash**(**const** **char** \*key, **int** i) // fnv-1a  {  **unsigned** **int** hash\_value = **2166136261** - i; // for different hash functions  **while**(\*key) {  hash\_value ^= (**unsigned** **int**)\*key++;  hash\_value \*= FNV\_32\_PRIME;  }  **return** hash\_value;  }  **int** **bloom\_optimalFilterSize**(**unsigned** **int** n, **double** p) // n - number of elements  { // p - probability of false positive  **return** (**unsigned** **int**) ceil(-((**double**) n \* log(p)) / (log(**2**) \* log(**2**)));  }  **int** **bloom\_optimalHashNumber**(**unsigned** **int** n, **unsigned** **int** m) // n - number of elements  { // m - filter size  **return** (**unsigned** **int**) ceil(((**double**) m / (**double**) n) \* log(**2**));  }  BloomFilter \***bloom\_init**(**unsigned** **int** number\_of\_elements, **double** falsepositive\_probability)  {  printf("%s n (elements) = %d**\n**", bloom\_prefix, number\_of\_elements);  printf("%s p (falsepositive probability) = %.1f%%**\n**", bloom\_prefix, falsepositive\_probability\***100**);  BloomFilter \*f = malloc(**sizeof**(BloomFilter));  **if**(f != NULL) {  printf("%s malloc \*f - success**\n**", bloom\_prefix);  }  f->m = bloom\_optimalFilterSize(number\_of\_elements, falsepositive\_probability);  **int** arraySize = (**int**) ceil(f->m / BITS\_FOR\_TYPE);  printf("%s m (optimal filter size) = %d (%d bytes)**\n**", bloom\_prefix, f->m, arraySize);  f->bits = malloc(**sizeof**(**uint8\_t**) \* arraySize);  **if**(f->bits != NULL) {  printf("%s malloc f->bits - success**\n**", bloom\_prefix);  }  **for**(**int** i = **0**; i < arraySize; i++) {  f->bits[i] = **0**;  }  f->k = bloom\_optimalHashNumber(number\_of\_elements, f->m);  printf("%s k (optimal hash functions number) = %d**\n**", bloom\_prefix, f->k);  **return** f;  }  **int** **bloom\_lookup**(BloomFilter \*f, **const** **char** \*s)  {  **for**(**int** i = **0**; i < f->k; i++) {  **if**(get\_bit(f->bits, bloom\_FNVHash(s, i) % f->m) == **0**) {  printf("%s '%s' is not in filter**\n**", bloom\_prefix, s);  **return** **0**;  }  }  printf("%s '%s' is probably in filter**\n**", bloom\_prefix, s);  **return** **1**;  }  **void** **bloom\_insert**(BloomFilter \*f, **const** **char** \*s)  {  **if**(!bloom\_lookup(f, s)){  **for**(**int** i = **0**; i < f->k; i++) {  set\_bit(f->bits, bloom\_FNVHash(s, i) % f->m);  }  printf("%s successfully inserted '%s'**\n**", bloom\_prefix, s);  }  } |

1.5 Файл “bloom.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #pragma once  #include <inttypes.h>  **typedef** **struct** {  **uint8\_t** \*bits;  **int** m; // filter size  **int** k; // number of hash functions  } BloomFilter;  **unsigned** **int** **bloom\_FNVHash**(**const** **char** \*key, **int** i);  **int** **bloom\_optimalFilterSize**(**unsigned** **int** n, **double** p);  **int** **bloom\_optimalHashNumber**(**unsigned** **int** n, **unsigned** **int** m);  BloomFilter \***bloom\_init**(**unsigned** **int** number\_of\_elements, **double** falsepositive\_probability);  **int** **bloom\_lookup**(BloomFilter \*f, **const** **char** \*s);  **void** **bloom\_insert**(BloomFilter \*f, **const** **char** \*s); |