**Отчёт.**

1.Структура данных: Фильтр Блума

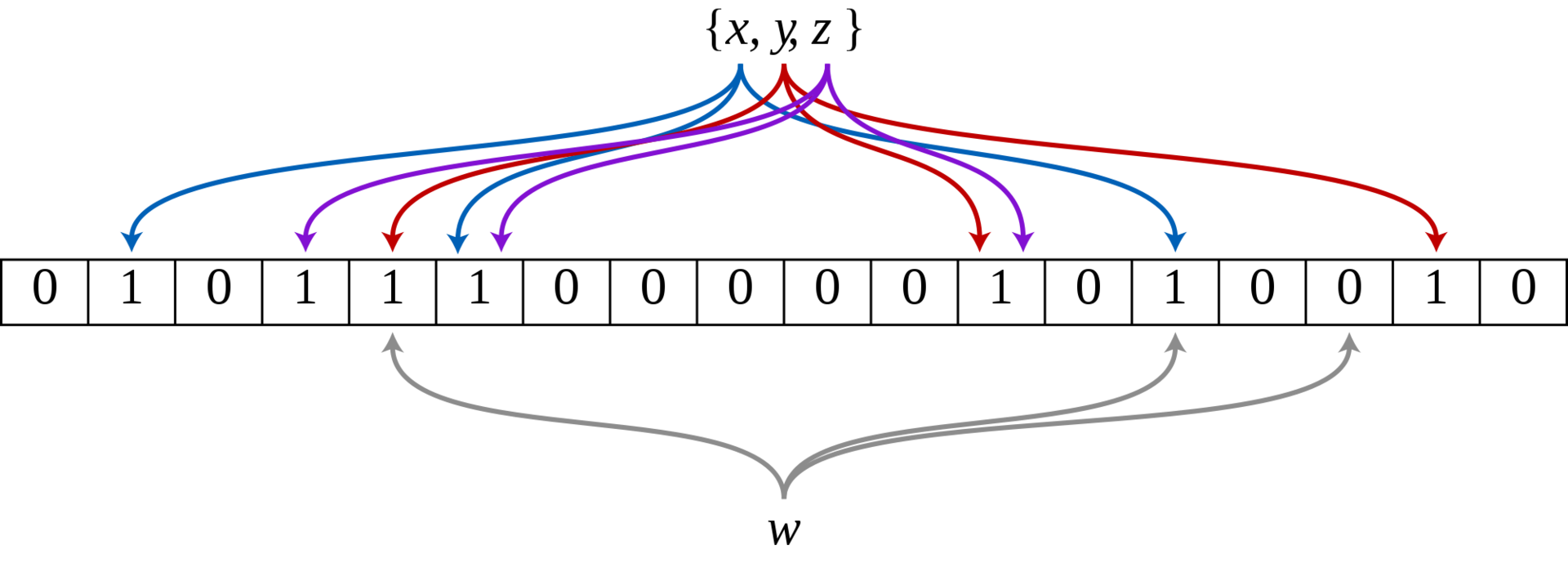
Историческая справка: изобретена Бёртоном Блумом в 1970 году. Фильтр Блума - это вероятностная структура данных, которая может эффективно определить является ли элемент возможным элементом набора или точно не относится к нему.

2.Пустой фильтр Блума - это [битовый массив](https://en.wikipedia.org/wiki/Bit_array) из m битов, все они имеют значение 0. Также должно быть определено k различных [хэш-функций](https://en.wikipedia.org/wiki/Hash_function), каждая из которых [отображает](https://en.wikipedia.org/wiki/Map_(mathematics)) или хэширует некоторый элемент набора в одну из позиций массива m, генерируя равномерное случайное распределение. Как правило, k-это небольшая константа , которая зависит от желаемой частоты ложных ошибок ε.

Чтобыдобавить элемент, скормите его каждой хэш-функции, чтобы получить k позиций массива. Установите биты во всех этих положениях равными 1.

Чтобы запросить элемент (проверить, находится ли он в наборе), нужно передать его каждой хэш-функции, чтобы получить позиции массива. Если какой-либо из битов в этих позициях равен 0, элемент определенно не входит в набор. Если всё равно 1, то либо элемент находится в наборе, либо биты случайно были установлены на 1 во время вставки других элементов, что привело к [ложному срабатыванию](https://en.wikipedia.org/wiki/False_positive). В простом фильтре Блума нет способа различить эти два случая.

Удаление элемента из этого простого фильтра Блума невозможно, потому что нет способа определить, какой из *k* битов, к которым он сопоставляется, должен быть очищен. Хотя установка любого из этих *k* битов на ноль достаточна для удаления элемента, она также удалит любые другие элементы, которые случайно сопоставляются с этим битом. Поскольку простой алгоритм не дает возможности определить, были ли добавлены какие-либо другие элементы, влияющие на биты для удаляемого элемента, очистка любого из битов приведет к возможности ложных срабатываний.



Разберемся на примере картинки:

1) Изначально у нас есть массив, заполненный нулями, и 3 элемента x,y,z (их мы хотим добавить в фильтр)

2) Мы прогнали элементы x,y,z через 3 хэш-функции (от каждого элемента идёт по 3 стрелочки), получили какие - то позиции массива, установили там 1

3) Теперь мы хотим проверить, а был ли добавлен элемент? Мы также прогоняем его через хэш-функции, получаем позиции, и смотрим, что стоит в этих позициях. В одной из ячеек стоит 0, а значит, что w во множестве нет.

Особенности:

* Возможность ложноположительных срабатываний
* Он никогда не даст *ложноотрицательного* ответа и никогда не скажет "НЕТ", если ответ будет "ДА" (*определенно не в наборе*).
* Нельзя *удалить* элемент из фильтра.
* Он также требует *очень мало места* по сравнению с количеством предметов, которые вам нужно хранить и проверять.

Размер битового массива и количество хэш-функций играют важную роль в частоте ложных срабатываний фильтра Блума. Если размер битового массива слишком мал, все биты будут установлены на 1 быстрее. После этого он начнет давать ложноположительный результат для каждого входа. Чем больше битовый массив, тем меньше ложноположительный результат.

Если у нас слишком мало хэш-функций, есть больше шансов на ложноположительный результат. Если у нас будет слишком много хэш-функций, фильтр Блума станет медленным, потому что ему придется хэшировать каждый вход так много раз.

(1-e^(nk/m))^k - вероятность ложных срабатываний  
  
(n/m)\*ln(2) - оптимальное количество хэш-функций

Где m - размер входных данных, n - на основе размера фильтра, k - количество хэш-функций.

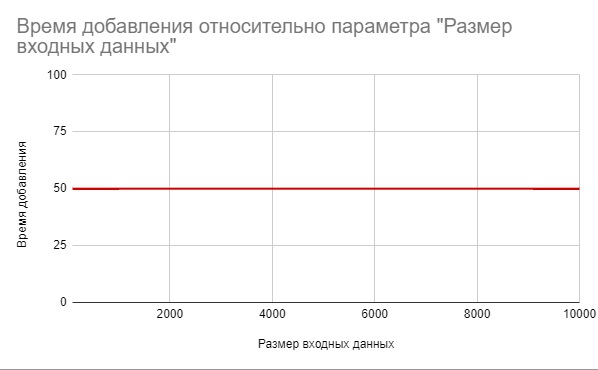
3. Фильтр Блума имеет *постоянную временную* сложность, то есть O(1), как для добавления элементов, так и для запроса наличия ключа или его отсутствия.

Чтобы добавить элемент нужно прогнать элемент через хэш-функции и установить биты, равными 1, на получившихся позициях массива. Тестирование проходит за О(k), где k - количество хэш-функций (какое-то конкретное число). Для проверки нужно так же получить позиции массива, и обратиться к значениям в этих позициях. Обращение к элементу массива - константное время.

5) Получившийся график:



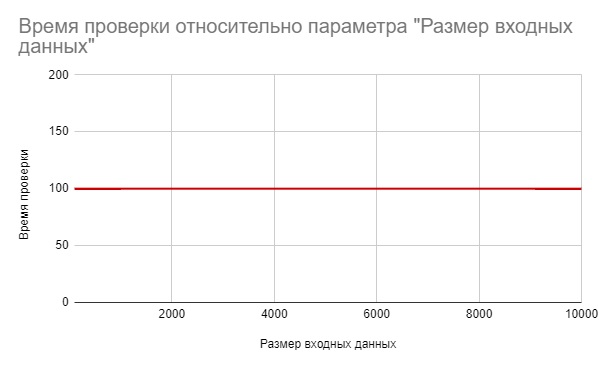
Какой график должен быть (примерно):



Получившийся график:



Каким график должен быть (примерно):



Время добавления и проверки должно быть константным, то есть графиком является горизонтальная прямая у = С. На получившихся графиках выше мы не наблюдаем ровную прямую, но он примерно похож, время проверки скачет у одной прямой. На это повлияли хэш-функции, время работы которых зависит от поступающего на обработку числа.

Почему график примерный? Потому что в этих графиках значения времени проверки и добавления, равные 50 и 100, могут отличаться. Суть в том, чтобы показать, что это время одно и то же при всех входных данных.

6.*Плюсы:* Фильтр Блума мало затратен по памяти и по времени. Он может быть очень полезен для предотвращения излишнего выполнения задач, требующих интенсивных вычислений, просто проверяя, что элемент *совершенно точно не* входит в множество.

*Минусы:* Ложноположительные срабатывания

* Google Bigtable, Apache HBase и Apache Cassandra, а также Postgresql используют фильтры bloom для уменьшения поиска несуществующих строк или столбцов на диске. Избегание дорогостоящих поисков на диске значительно повышает производительность операции запроса базы данных.
* Bloom filter используется провайдером сети доставки контента Akamai, чтобы избежать кэширования веб-объектов, которые запрашиваются пользователями только один раз ( также известный как “one-hit-wonders”). Использование блум-фильтра для обнаружения второго запроса веб-объекта и кэширование этого объекта только по его второму запросу предотвращает попадание в дисковый кэш однократных чудес, значительно уменьшая нагрузку на диск и увеличивая частоту попаданий в дисковый кэш.
* Веб-[браузер Google Chrome](https://en.wikipedia.org/wiki/Google_Chrome) использовал фильтр Bloom для идентификации вредоносных URL-адресов. Любой URL-адрес сначала проверялся по локальному фильтру Блума, и только если фильтр Блума возвращал положительный результат, выполнялась полная проверка URL-адреса (и пользователь предупреждался, если это тоже возвращало положительный результат)

7) Список литературы:

<https://en.wikipedia.org/wiki/Bloom_filter>

<https://findingprotopia.org/posts/how-to-write-a-bloom-filter-cpp/>

<https://medium.com/system-design-blog/bloom-filter-a-probabilistic-data-structure-12e4e5cf0638>

<https://ru.wikipedia.org/wiki/Фильтр_Блума>

<https://habr.com/ru/company/otus/blog/541378/>

<https://otus.ru/nest/post/972/>

<http://espressocode.top/bloom-filters-introduction-and-python-implementation/>

8)

#include <iostream>  
#include <string>  
#include <vector>  
#include <iterator>  
using namespace std;  
/\*Задача: У нас есть множество чисел,  
 \* вводятся какие- то другие числа,  
 \* нужно проверить, находтсчя ли они в исходном множестве\*/  
  
struct BloomFilter {  
 static void add(vector<int> data, bool bf [],int s);  
 static bool possiblyContains(int element, bool bf[], int s);  
};  
  
  
int HashFunction1(int element, int filtersize) {  
 element = element \* 100 / 345;  
 while (element >= filtersize) {  
 element %= filtersize;  
 }  
 return element;  
}  
  
int HashFunction2(int element, int filtersize) {  
 element = element \* 13 / 56 \* 43;  
 while (element >= filtersize) {  
 element /= filtersize;  
 }  
 return element;  
}  
  
int HashFunction3(int element, int filtersize) {  
 element = element + 473 / 39;  
 while (element >= filtersize) {  
 element = (element \* 3 + 456) / filtersize;  
 }  
 return element;  
}  
  
void BloomFilter::add(vector<int> data, bool bf [], int s) {  
 for(int i : data) {  
 bf[HashFunction1(i, s)] = true;  
 bf[HashFunction2(i, s)] = true;  
 bf[HashFunction3(i, s)] = true;  
 }  
}  
  
bool BloomFilter::possiblyContains(int element, bool bf [], int s) {  
 if(!bf[HashFunction1(element,s)] || !bf[HashFunction2(element, s)] || !bf[HashFunction3(element, s)]) {  
 return false;  
 }  
 return true;  
}  
  
int main() {  
 int N;  
 cin >> N;  
 bool bf [N];  
 vector<int> data = {13,56,34,8,9,3,1};  
 BloomFilter bf1 = \*new BloomFilter;  
 bf1.add(data, bf, N);  
 vector<int> check = {13, 5, 7, 9, 34, 70};  
  
 for(int i : check) {  
 if (bf1.possiblyContains(i, bf, N)) {  
 cout << i << " " << ": This number is contained in set\n";  
 } else {  
 cout << i << " " << ": This number is NOT contained in set\n";  
 }  
 }  
}

9) Входные данные: <https://github.com/Alsu11/ALSUAiSD/blob/main/BloomFilter/Входные%20данные.docx>

4)

|  |  |  |
| --- | --- | --- |
| Время добавления | Время проверки | Размер входных данных |
| 150 | 0 | 100 |
| 100 | 0 | 200 |
| 50 | 0 | 300 |
| 500 | 0 | 400 |
| 0 | 0 | 500 |
| 300 | 0 | 600 |
| 10 | 0 | 700 |
| 250 | 50 | 800 |
| 400 | 0 | 900 |
| 300 | 120 | 1000 |
| 450 | 20 | 1300 |
| 20 | 30 | 1600 |
| 0 | 0 | 1900 |
| 312 | 230 | 2200 |
| 220 | 124 | 2500 |
| 64 | 0 | 2800 |
| 392 | 0 | 3100 |
| 228 | 130 | 3400 |
| 473 | 140 | 3700 |
| 149 | 24 | 4000 |
| 201 | 25 | 4300 |
| 640 | 132 | 4600 |
| 202 | 4 | 4900 |
| 161 | 160 | 5200 |
| 145 | 0 | 5500 |
| 133 | 0 | 5800 |
| 242 | 25 | 6100 |
| 59 | 13 | 6400 |
| 69 | 45 | 6700 |
| 180 | 260 | 7000 |
| 40 | 237 | 7300 |
| 351 | 47 | 7500 |
| 745 | 145 | 7700 |
| 161 | 41 | 7900 |
| 1489 | 134 | 8100 |
| 162 | 234 | 8300 |
| 532 | 312 | 8500 |
| 846 | 178 | 8700 |
| 765 | 189 | 8800 |
| 319 | 123 | 8900 |
| 234 | 94 | 9000 |
| 345 | 98 | 9100 |
| 615 | 274 | 9200 |
| 478 | 123 | 9300 |
| 543 | 149 | 9400 |
| 324 | 201 | 9500 |
| 234 | 156 | 9600 |
| 367 | 151 | 9700 |
| 263 | 143 | 9800 |
| 196 | 75 | 9900 |
| 200 | 132 | 10000 |