Департамент образования администрации г. Перми
МБОУ «Лицей №1» г. Перми

Учебно-исследовательская работа
«Разрешение коллизий в хеш-функциях»

Выполнила:
Мухина Алиса Васильевна,
102 класс, группа бета.
Научный руководитель:
Котельникова Наталья Васильевна
Преподаватель информатики в МБОУ «Лицей №1»

Пермь – 2017

Хеширование

В настоящее время количество хранимой информации стремительно растет. Это влечет за собой появление множества новых задач, связанных с хранением, сортировкой, поиском и другими видами обработки информации. Также важнейшим моментом является обеспечение безопасности и надежной передачи информации.

Безопасность всегда была неоднозначной темой, провоцирующей многочисленные горячие споры. И всё благодаря обилию самых разных точек зрения и «идеальных решений», которые устраивают одних и совершенно не подходят другим. Есть мнение, что взлом системы безопасности приложения – всего лишь вопрос времени. Из-за быстрого роста вычислительных мощностей и увеличения сложности безопасные сегодня приложения перестанут завтра быть таковыми.

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска, т.е. определяющего значения, большого числа элементов. Сокращение поиска возможно осуществить путем *локализации* области просмотра. Например, отсортировать данные по ключу поиска, разбить на непересекающиеся блоки по некоторому групповому признаку или поставить в соответствие реальным данным некий код, который упростит процедуру поиска.

В настоящее время используется широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти – *хеширование*.

Хеширование (или *хэширование*, англ. *Hashing*) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются *хеш-функциями* или функциями *свертки*, а их результаты называют *хешем, хеш-кодом, хеш-таблицей* или *дайджестом* сообщения (англ. *message digest*).

Хеш-таблица – это *структура данных*, реализующая *интерфейс* ассоциативного массива, то есть она позволяет хранить пары вида "*ключ-значение*" и выполнять три *операции*: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хеш-таблица является массивом, формируемым в определенном порядке *хеш-функцией*.

Хеширование полезно, когда широкий *диапазон* возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа. *Хэш-таблицы* часто применяются в базах данных, и, особенно, в *языковых процессорах* типа компиляторов и *ассемблеров*, где они повышают скорость обработки таблицы

идентификаторов. В качестве использования *хеширования* в повседневной жизни можно привести примеры распределение книг в библиотеке по тематическим каталогам, упорядочивание в словарях по первым буквам слов, *шифрование* специальностей в вузах и т.д.

Принято считать, что хорошей, с точки зрения практического применения, является такая *хеш-функция*, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно;
- функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- функция должна минимизировать число *коллизий* – то есть ситуаций, когда разным ключам соответствует одно значение *хеш-функции*(ключи в этом случае называются *синонимами*).

Коллизией хеш-функции *hash* называются два параметра *a* и *b*, при *hash(a) = hash(b).* На практике это означает, что двум значениям(символам или последовательностям символов) соответствует один ключ-значение.

Коллизии существуют для большинства хеш-функкций, но для «хороших» хеш-функций частота их возникновения близка к теоретическому минимуму. В некоторых частных случаях, когда множество различных входных данных конечно, можно задать инъективную хеш-функцию, по определению не имеющую коллизий. Однако для хеш-функций, принимающих вход переменной длины и возвращающих хеш постоянной длины, коллизии обязаны существовать, поскольку хотя бы для одного значения хеш-функции соответствующее ему множество входных данных будет бесконечно — и любые два набора данных из этого множества образуют коллизию.

При этом первое свойство хорошей *хеш-функции* зависит от характеристик компьютера, а второе – от значений данных.

Если бы все данные были случайными, то *хеш-функции* были бы очень простые (например, несколько битов ключа). Однако на практике случайные данные встречаются достаточно редко, и приходится создавать функцию, которая зависела бы от всего ключа. Если *хеш-функция* распределяет совокупность *возможных ключей* равномерно по множеству индексов, то *хеширование* эффективно разбивает множество ключей. Наихудший случай – когда все ключи хешируются в один *индекс*.

При возникновении *коллизий* необходимо найти новое *место* для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы. Причем, если *коллизии* допускаются, то их количество необходимо минимизировать. В некоторых специальных случаях удается избежать *коллизий* вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без *коллизий*. Хеш-таблицы, использующие подобные *хеш-функции*, не нуждаются в механизме разрешения *коллизий*, и называются хеш-таблицами с *прямой адресацией*.

Хеш-таблицы должны соответствовать следующим *свойствам:*

- Выполнение операции в хеш-таблице начинается с вычисления *хеш-функции* от ключа. Получающееся хеш-значение является индексом в исходном массиве.

- Количество хранимых элементов массива, деленное на число возможных значений *хеш-функции*, называется *коэффициентом заполнения хеш-таблицы* (*load factor*) и является важным параметром, от которого зависит среднее время выполнения операций.

- Операции поиска, вставки и удаления должны выполняться в среднем за время O(1). Однако при такой оценке не учитываются возможные аппаратные затраты на перестройку индекса хеш-таблицы, связанную с увеличением значения размера массива и добавлением в хеш-таблицу новой пары.

- Механизм разрешения *коллизий* является важной составляющей любой хеш-таблицы.

Таким образом, особо актуальным является исследование разрешения коллизий при хранении паролей, имён пользователей и других конфиденциальных информаций, хранящихся в базах данных серверов компаний, т.к. является не только российской, но и мировая проблема, учитывая быстрое развитие как информационной безопасности, так и способов взлома информации.

Цель работы:

Найти наилучший способ разрешения коллизий, который будет работать как можно быстрее и требует как можно меньше памяти для хранения защищённых паролей в базах данных.

Решение:

Существует несколько подходов к разрешению коллизий, они основаны одном важном требовании: при незначительном изменении аргумента должно происходить значительное изменение самой функции. Таким образом, значение хеша не должно давать информации даже об отдельных битах аргумента. Рассмотрим наиболее популярные методы:

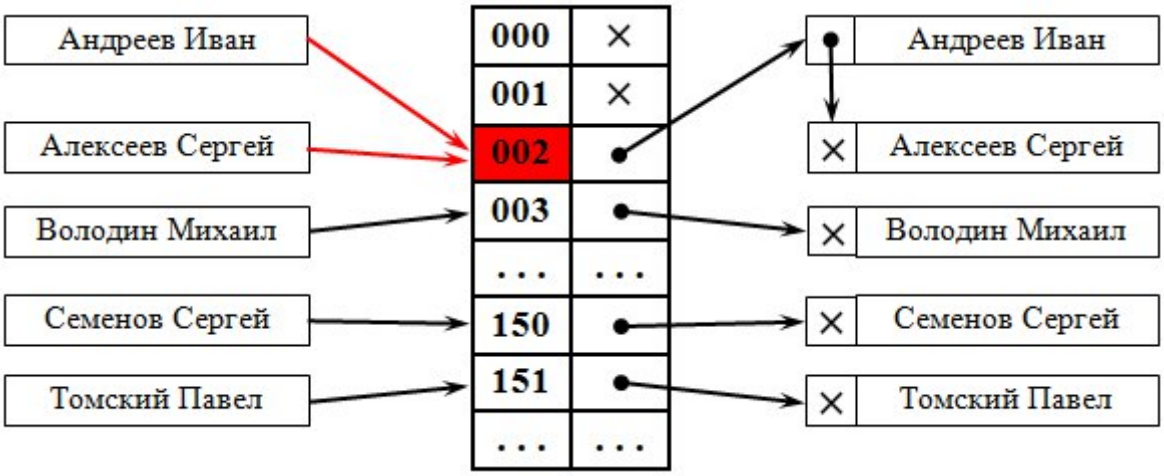
Методы разрешения коллизий:

- метод цепочек (внешнее или открытое *хеширование*);
- метод открытой адресации (закрытое *хеширование*).

Метод цепочек.

Технология сцепления элементов состоит в том, что *элементы множества*, которым соответствует одно и то же хеш-*значение*, связываются в цепочку-*список*. В позиции номер i хранится *указатель* на *голову списка* тех элементов, у которых хеш-*значение* ключа равно i; если таких элементов в множестве нет, в позиции i записан NULL.

На рисунке демонстрируется реализация метода цепочек при разрешении *коллизий*. На *ключ* 002 претендуют два значения, которые организуются в линейный *список*.



Каждая *ячейка* массива является указателем на связный *список* (цепочку) пар *ключ-значение*, соответствующих одному и тому же хеш-значению ключа. *Коллизии* просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу.

При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, *среднее время* работы *операции* поиска элемента составляет O(1+k), где k – коэффициент заполнения таблицы.

Код:

```
#include <bits/stdc++.h>
using namespace std;
vector <int> hash_[1001];
const int MD = 100, p = 26;
string s[1001];
```

```
int main()
{   int n;
    cin >> n;
    for (size_t i = 0; i < n; ++i){
        cin >> s[i];
        for (size_t j = 0 ; j < n; ++j){
            hash_[i].push_back(-1);
        }
    }
    for (size_t sp_i = 0; sp_i < n; ++sp_i){
        long long now_h = 0;
        for (size_t i = 1; i <= s[sp_i].length(); ++i){
            now_h = (now_h * p + (s[sp_i][i - 1] - 'a' + 1)) % MD;
        }
        cout << now_h << ' ' << sp_i << '\n';
        hash_[now_h].push_back(sp_i);
    }
    return 0;
}
```

Представление кода в памяти компьютера на примере:

Входные данные	*Представления данных в памяти*			
4 film индекс = 0; хеш-функция = 65 show индекс = 1; хеш-функция = 65 ten индекс = 2; хеш-функция = 64 box индекс = 3; хеш-функция = 66	...	-1	-1	...
	63	-1	-1	...
	64	2 ten	-1	...
	65	0 film	1 show	...
	66	3 box	-1	...
	...	-1	-1	...

Анализ метода цепочек:

Плюсы:

- Метод цепочек эффективен и имеет чёткую структуру.
- Его удобно использовать, когда неизвестно количество коллизий на одно хеш-значение.
- Поиск нужного значения будет происходит за минимально возможное время.

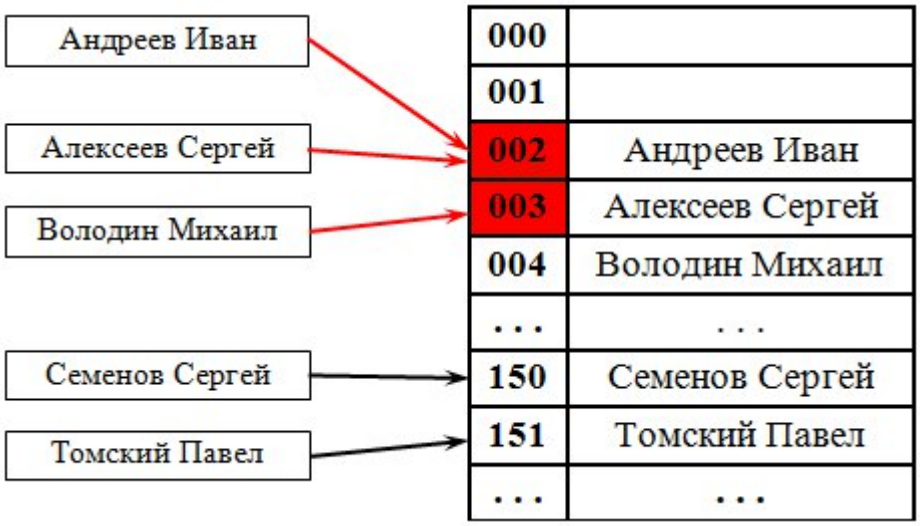
Минусы:

- Он использует много памяти: для хранения n хеш-значений выделяется $\sim n^2$ ячеек памяти.
- Время работы метода $O(n^2)$.

Метод открытой адресации.

В отличие от *хеширования* с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице. Каждая *ячейка* таблицы содержит либо элемент динамического *множества*, либо NULL.

В этом случае, если *ячейка* с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найден *ключ* K или пустая позиция в таблице. Для вычисления шага можно также применить формулу, которая и определит способ изменения шага. На рисунке разрешение *коллизий* осуществляется методом открытой адресации. Два значения претендуют на *ключ* 002, для одного из них находится первое свободное (еще незанятое) *место* в таблице.



При любом методе разрешения *коллизий* необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента

Для успешной работы алгоритмов поиска, последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу.

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят *логический* флаг для каждой ячейки, помечающий, удален ли элемент в ней или нет. Тогда *удаление элемента* состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удаленные ячейки занятыми, а процедуру добавления – чтобы она их считала свободными и сбрасывала *значение* флага при добавлении.

Код:

```
#include <bits/stdc++.h>
```

```
using namespace std;

vector <int> hash_;
const int MD = 100, p = 26;
string s[1001];

int main()
{   int n;
    cin >> n;
    p_pow[0] = 1;
    for (size_t i = 1; i < n; ++i){
        cin >> s[i];
    }
    for (size_t sp_i = 0; sp_i < n; ++sp_i){
        long long now_h;
        for (size_t i = 1; i <= s[sp_i].length(); ++i){
            now_h = (now_h * p + (s[sp_i][i - 1] - 'a' + 1)) % MD;
        }
        while (hash_[now_h] != -1) {
            now_h++;
        }
        hash_[now_h] = sp_i;
    }
    return 0;
}
```

Представление кода в памяти компьютера на примере:

Входные данные	*Представления данных в памяти*
4 film индекс = 0; хеш-функция = 65 show индекс = 1; хеш-функция = 65 ten индекс = 2; хеш-функция = 64 box индекс = 3; хеш-функция = 66	… -1 63 -1 64 2 ten 65 0 film 66 1 show 67 3 box ... -1

Анализ метода открытой адресации:

Плюсы:

- Использует мало памяти: для хранения *n* значений резервируется только *n* ячеек в памяти
- Удобно использовать при малом количестве коллизий на одно хеш-значение(не более 3-х)

Минусы:

- Поиск определённого значения в хеш-таблице неоптимально
- Время работы *O(n^2)*

- Нет чёткой структуры, хеш-значения могут храниться не в отсортированном виде

Заключение

Целью данной исследовательской работы был анализ популярных методов разрешения коллизий в хеш-функциях и нахождение способа, который будет более практичным, универсальным и одновременно оптимальным.

Рассматривалось два наиболее популярных метода, основанных на абсолютной разнице между значением данных и значением хеш-функции данных: метод цепочек и метод открытой адресации. Получено, что лучшим способом по указанным критериям оказался метод цепочек.

В дальнейшем планируется не останавливаться на хеш-функциях, оставляя изучение криптографии, и двигаться дальше в этом, на данный момент очень обширном и разнообразном направлении в программировании.

Список литературы

1.

2.

3.