**0Министерство науки и высшего образования Российской Федерации**

Федеральное государственное автономное образовательное учреждение

высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Инженерная школа информационных технологий и робототехники

Отделение - ОИТ

Направление – Информатика и вычислительная техника

**Отчёт по лабораторной работе № 2**

**«Хеширование»**

по дисциплине: Защита информации

Выполнил:

студент гр. 8В7Б Мальцев М.Ю.

Проверил:

Доцент ОИТ Ботыгин И.А.

Томск 2020 г.

Оглавление

Оглавление

[Оглавление 2](#_Toc55403693)

[Результат работы 3](#_Toc55403694)

[1. Введение 3](#_Toc55403695)

[2. Хеш-функции 5](#_Toc55403696)

[3. Метод деления 6](#_Toc55403697)

[4. Метод умножения (мультипликативный) 7](#_Toc55403698)

[5. Динамическое хеширование 8](#_Toc55403699)

[6. Расширяемое хеширование (extendible hashing) 9](#_Toc55403700)

[7. Функции, сохраняющие порядок ключей (order preserving hash functions) 11](#_Toc55403701)

[8. Минимальное идеальное хеширование 11](#_Toc55403702)

[9. Разрешение коллизий 12](#_Toc55403703)

[10. Метод цепочек 13](#_Toc55403704)

[11. Открытая адресация 13](#_Toc55403705)

[12. Линейная адресация 14](#_Toc55403706)

[13. Квадратичная и произвольная адресация 15](#_Toc55403707)

[14. Адресация с двойным хешированием 15](#_Toc55403708)

[15. Удаление элементов хеш-таблицы 16](#_Toc55403709)

[16. Применение хеширования 18](#_Toc55403710)

[17. Хеширование паролей 18](#_Toc55403711)

[18. Пример работы хеширования на собственной программе 19](#_Toc55403712)

[Вывод 23](#_Toc55403713)

[Приложение А. Исходный код приложения 24](#_Toc55403714)

Результат работы

1. Введение

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web-ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Хеширование есть разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде текстовых строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается функцией хеширования, или хеш-функцией, и называется хеш-адресом. Решение обратной задачи возложено на хеш-структуры (хеш-таблицы): по хеш-адресу они обеспечивают быстрый доступ к нужному элементу. В идеале для задач поиска хеш-адрес должен быть уникальным, чтобы за одно обращение получить доступ к элементу, характеризуемому заданным ключом (идеальная хеш-функция). Однако, на практике идеал приходится заменять компромиссом и исходить из того, что получающиеся наборы с одинаковым хеш-адресом содержат более одного элемента.

Термин «хеширование» (hashing) в печатных работах по программированию появился сравнительно недавно, хотя сам механизм был известен и ранее. Глагол «hash» в английском языке означает «рубить, крошить». Для русского языка академиком А.П. Ершовым был предложен достаточно удачный эквивалент — «расстановка», созвучный с родственными понятиями комбинаторики, такими как «подстановка» и «перестановка». Однако он не прижился.

Как отмечает Дональд Кнут, идея хеширования впервые была высказана Г.П. Ланом при создании внутреннего меморандума IBM в январе 1953 г. с предложением использовать для разрешения коллизий хеш-адресов метод цепочек. Примерно в это же время другой сотрудник IBM – Жини Амдал – высказала идею использования открытую линейную адресацию. В открытой печати хеширование впервые было описано Арнольдом Думи (1956), указавшим, что в качестве хеш-адреса удобно использовать остаток от деления на простое число. А. Думи описывал метод цепочек для разрешения коллизий, но не говорил об открытой адресации. Подход к хешированию, отличный от метода цепочек, был предложен А.П. Ершовым, который разработал и описал метод линейной открытой адресации. Среди других исследований можно отметить работу Петерсона. В ней реализовывался класс методов с открытой адресацией при работе с большими файлами. Петерсон определил открытую адресацию в общем случае, проанализировал характеристики равномерного хеширования, глубоко изучил статистику использования линейной адресации на различных задачах. В 1963 г. Вернер Букхольц опубликовал наиболее основательное исследование хеш-функций.

К концу шестидесятых годов прошлого века линейная адресация была единственным типом схемы открытой адресации, описанной в литературе, хотя несколькими исследователями независимо была разработана другая схема, основанная на неоднократном случайном применении независимых хеш-функции. В течение нескольких последующих лет хеширование стало широко использоваться, хотя не было опубликовано никаких новых работ. Затем Роберт Моррис обширный обзор по хешированию и ввел термин «рассеянная память» (scatter storage). Эта работа привела к созданию открытой адресации с двойным хешированием.

Далее будут рассмотрены основные виды хеш-функций и некоторые их модификации, методы разрешения коллизий, проблемы удаления элементов из хеш-таблицы, а также некоторые варианты применения хеширования.

1. Хеш-функции

Хеш-функция – это некоторая функция h(K), которая берет некий ключ K и возвращает адрес, по которому производится поиск в хеш-таблице, чтобы получить информацию, связанную с K. Например, K – это номер телефона абонента, а искомая информация – его имя. Функция в данном случае нам точно скажет, по какому адресу найти искомое. Пример с телефонным справочником иллюстрируется демонстрационной программой на прилагаемом компакт-диске.

Коллизия – это ситуация, когда h(K1) = h(K2), в то время как K1 ≠ K2. В этом случае, очевидно, необходимо найти новое место для хранения данных. Очевидно, что количество коллизий необходимо минимизировать. Методикам разрешения коллизий будет посвящен отдельный раздел ниже.

Хорошая хеш-функция должна удовлетворять двум требованиям:

- ее вычисление должно выполняться очень быстро;

- она должна минимизировать число коллизий.

Итак, первое свойство хорошей хеш-функции зависит от компьютера, а второе – от данных. Если бы все данные были случайными, то хеш-функции были бы очень простые (несколько битов ключа, например). Однако на практике случайные данные встречаются крайне редко, и приходится создавать функцию, которая зависела бы от всего ключа.

Теоретически невозможно определить хеш-функцию так, чтобы она создавала случайные данные из реальных неслучайных файлов. Однако на практике реально создать достаточно хорошую имитацию с помощью простых арифметических действий. Более того, зачастую можно использовать особенности данных для создания хеш-функций с минимальным числом коллизий (меньшим, чем при истинно случайных данных).

Возможно, одним из самых очевидных и простых способов хеширования является метод середины квадрата, когда ключ возводится в квадрат и берется несколько цифр в середине. Здесь и далее предполагается, что ключ сначала приводится к целому числу, для совершения с ним арифметических операций. Однако такой способ хорошо работает до момента, когда нет большого количества нолей слева или справа. Многочисленные тесты показали хорошую работу двух основных типов хеширования, один из которых основан на делении, а другой на умножении. Впрочем, это не единственные методы, которые существуют, более того, они не всегда являются оптимальными.

1. Метод деления

Метод деления весьма прост – используется остаток от деления на M:

h(K) = K mod M

Надо тщательно выбирать эту константу. Если взять ее равной 100, а ключом будет случить год рождения, то распределение будет очень неравномерным для ряда задач (идентификация игроков юношеской бейсбольной лиги, например). Более того, при четной константе значение функции будет четным при четном K и нечетным - при нечетном, что приведет к нежелательному результату. Еще хуже обстоят дела, если M – это степень счисления компьютера, поскольку при этом результат будет зависеть только от нескольких цифр ключа справа. Точно также можно показать, что M не должно быть кратно трем, поскольку при буквенных ключах два из них, отличающиеся только перестановкой букв, могут давать числовые значения с разностью, кратной трем. Приведенные рассуждения приводят к мысли, что лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен.

Другой пример – ключ, являющийся символьной строкой С++. Хеш-функция отображает эту строку в целое число посредством суммирования первого и последнего символов и последующего вычисления остатка от деления на 101 (размер таблицы). Эта хеш-функция приводит к коллизии при одинаковых первом и последнем символах строки. Например, строки «start» и «slant» будут отображаться в индекс 29. Так же ведет себя хеш-функция, суммирующая все символы строки. Строки «bad» и «dab» преобразуются в один и тот же индекс. Лучшие результаты дает хеш-функция, производящая перемешивание битов в символах. На практике, метод деления – самый распространенный.

1. Метод умножения (мультипликативный)

Для мультипликативного хеширования используется следующая формула:

h(K) = [M \* ((C \* K) mod 1)]

Здесь производится умножение ключа на некую константу С, лежащую в интервале [0..1]. После этого берется дробная часть этого выражения и умножается на некоторую константу M, выбранную таким образом, чтобы результат не вышел за границы хеш-таблицы. Оператор [ ] возвращает наибольшее целое, которое меньше аргумента.

Если константа С выбрана верно, то можно добиться очень хороших результатов, однако, этот выбор сложно сделать. Дональд Кнут отмечает, что умножение может иногда выполняться быстрее деления.

Мультипликативный метод хорошо использует то, что реальные файлы неслучайны. Например, часто множества ключей представляют собой арифметические прогрессии, когда в файле содержатся ключи {K, K + d, K + 2d, …, K + td}. Например, рассмотрим имена типа {PART1, PART2, …, PARTN}. Мультипликативный метод преобразует арифметическую прогрессию в приближенно арифметическую прогрессию h(K), h(K + d), h(K + 2d),… различных хеш-значений, уменьшая число коллизий по сравнению со случайной ситуацией. Впрочем, справедливости ради надо заметить, что метод деления обладает тем же свойством.

Частным случаем выбора константы является значение золотого сечения φ = (√5 - 1)/2 ≈ 0,6180339887. Если взять последовательность {φ}, {2φ}, {3φ},... где оператор { } возвращает дробную часть аргумента, то на отрезке [0..1] она будет распределена очень равномерно. Другими словами, каждое новое значение будет попадать в наибольший интервал. Это явление было впервые замечено Я. Одерфельдом (J. Oderfeld) и доказано С. Сверчковски (S. Świerczkowski). В доказательстве играют важную роль числа Фибоначчи. Применительно к хешированию это значит, что если в качестве константы С выбрать золотое сечение, то функция будет достаточно хорошо рассеивать ключи вида {PART1, PART2, …, PARTN}. Такое хеширование называется хешированием Фибоначчи. Впрочем, существует ряд ключей (когда изменение происходит не в последней позиции), когда хеширование Фибоначчи оказывается не самым оптимальным.

1. Динамическое хеширование

Описанные выше методы хеширования являются статическими, т.е. сначала выделяется некая хеш-таблица, под ее размер подбираются константы для хеш-функции. К сожалению, это не подходит для задач, в которых размер базы данных меняется часто и значительно. По мере роста базы данных можно

пользоваться изначальной хеш-функцией, теряя производительность из-за роста коллизий;

выбрать хеш-функцию «с запасом», что повлечет неоправданные потери дискового пространства;

периодически менять функцию, пересчитывать все адреса. Это отнимает очень много ресурсов и выводит из строя базу на некоторое время.

Существует техника, позволяющая динамически менять размер хеш-структуры. Это – динамическое хеширование. Хеш-функция генерирует так называемый псевдоключ (“pseudokey”), который используется лишь частично для доступа к элементу. Другими словами, генерируется достаточно длинная битовая последовательность, которая должна быть достаточна для адресации всех потенциально возможных элементов. В то время, как при статическом хешировании потребовалась бы очень большая таблица (которая обычно хранится в оперативной памяти для ускорения доступа), здесь размер занятой памяти прямо пропорционален количеству элементов в базе данных. Каждая запись в таблице хранится не отдельно, а в каком-то блоке (“bucket”). Эти блоки совпадают с физическими блоками на устройстве хранения данных. Если в блоке нет больше места, чтобы вместить запись, то блок делится на два, а на его место ставится указатель на два новых блока.

Задача состоит в том, чтобы построить бинарное дерево, на концах ветвей которого были бы указатели на блоки, а навигация осуществлялась бы на основе псевдоключа. Узлы дерева могут быть двух видов: узлы, которые показывают на другие узлы или узлы, которые показывают на блоки. Например, пусть узел имеет такой вид, если он показывает на блок:

Zero Null

Bucket Указатель

One Null

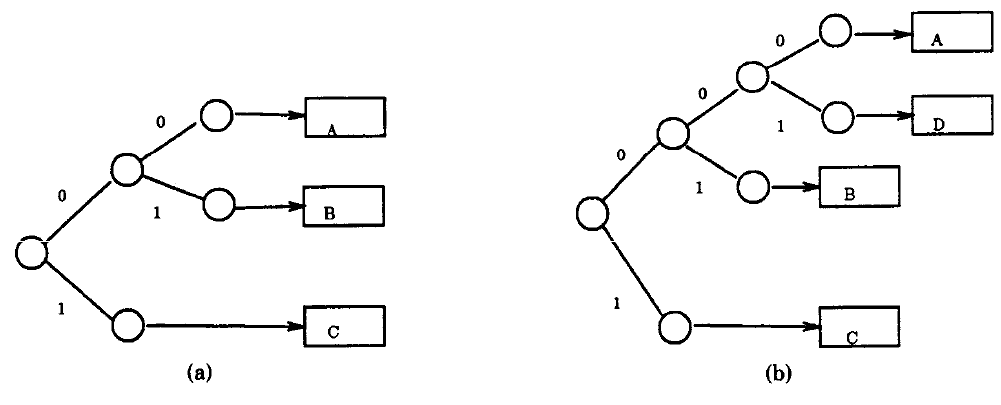
Если же он будет показывать на два других узла, то он будет иметь такой вид:

Zero Адрес a

Bucket Null

One Адрес b

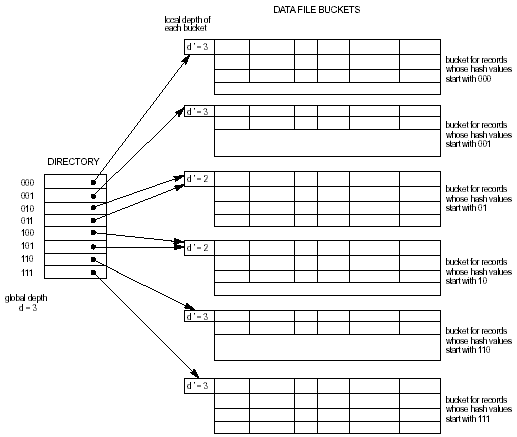
Вначале имеется только указатель на динамически выделенный пустой блок. При добавлении элемента вычисляется псевдоключ, и его биты поочередно используются для определения местоположения блока. Например (см. рисунок), элементы с псевдоключами 00… будут помещены в блок A, а 01… - в блок B. Когда А будет переполнен, он будет разбит таким образом, что элементы 000… и 001… будут размещены в разных блоках.



1. – Размещение элементов в блоках
2. Расширяемое хеширование (extendible hashing)

Расширяемое хеширование близко к динамическому. Этот метод также предусматривает изменение размеров блоков по мере роста базы данных, но это компенсируется оптимальным использованием места. Т.к. за один раз разбивается не более одного блока, накладные расходы достаточно малы.

Вместо бинарного дерева расширяемое хеширование предусматривает список, элементы которого ссылаются на блоки. Сами же элементы адресуются по некоторому количеству i битов псевдоключа (см. рис). При поиске берется i битов псевдоключа и через список (directory) находится адрес искомого блока. Добавление элементов производится сложнее. Сначала выполняется процедура, аналогичная поиску. Если блок неполон, добавляется запись в него и в базу данных. Если блок заполнен, он разбивается на два, записи перераспределяются по описанному выше алгоритму. В этом случае возможно увеличение числа бит, необходимых для адресации. В этом случае размер списка удваивается и каждому вновь созданному элементу присваивается указатель, который содержит его родитель. Таким образом, возможна ситуация, когда несколько элементов показывают на один и тот же блок. Следует заметить, что за одну операцию вставки пересчитываются значения не более, чем одного блока. Удаление производится по такому же алгоритму, только наоборот. Блоки, соответственно, могут быть склеены, а список – уменьшен в два раза.



1. – Сегменты файлов данных
2. Функции, сохраняющие порядок ключей (order preserving hash functions)

Существует класс хеш-функций, которые сохраняют порядок ключей. Другими словами, выполняется

K1 < K2  h(K1) < h(K2)

Эти функции полезны для сортировки, которая не потребует никакой дополнительной работы. Другими словами, мы избежим множества сравнений, т.к. для того, чтобы отсортировать объекты по возрастанию достаточно просто линейно просканировать хеш-таблицу.

В принципе, всегда можно создать такую функцию, при условии, что хеш-таблица больше, чем пространство ключей. Однако, задача поиска правильной хеш-функции нетривиальна. Разумеется, она очень сильно зависит от конкретной задачи. Кроме того, подобное ограничение на хеш-функцию может пагубно сказаться на ее производительности. Поэтому часто прибегают к индексированию вместо поиска подобной хеш-функции, если только заранее не известно, что операция последовательной выборки будет частой.

1. Минимальное идеальное хеширование

Как уже упоминалось выше, идеальная хеш-функция должна быстро работать и минимизировать число коллизий. Назовем такую функцию идеальной (perfect hash function). С такой функцией можно было бы не пользоваться механизмом разрешения коллизий, т.к. каждый запрос был бы удачным. Но можно наложить еще одно условие: хеш-функция должна заполнять хеш-таблицу без пробелов. Такая функция будет называться минимальной идеальной хеш-функцией. Это идеальный случай с точки зрения потребления памяти и скорости работы. Очевидно, что поиск таких функций – очень нетривиальная задача.

Один из алгоритмов для поиска идеальных хеш-функций был предложен Р. Чичелли. Рассмотрим набор некоторых слов, для которых надо составить минимальную идеальную хеш-функцию. Пусть это будут некоторые ключевые слова языка С++. Пусть это будет какая-то функция, которая зависит от некоего численного кода каждого символа, его позиции и длины. Тогда задача создания функции сведется к созданию таблицы кодов символов, таких, чтобы функция была минимальной и идеальной. Алгоритм очень прост, но занимает очень много времени для работы. Производится полный перебор всех значений в таблице с откатом назад в случае необходимости, с целью подобрать все значения так, чтобы не было коллизий. Если взять для простоты функцию, которая складывает коды первого и последнего символа с длиной слова (да, среди слов умышленно нет таких, которые дают коллизию), то алгоритм дает следующий результат:

1. – Результат работы алгоритма

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Буква | Код | Буква | Код | Буква | Код | Буква | Код | Буква | Код |
| a | -5 | e | 4 | i | 2 | n | -1 | t | 10 |
| b | -8 | f | 12 | k | 31 | o | 22 | u | 26 |
| c | -7 | g | -7 | l | 29 | r | 5 | v | 19 |
| d | -10 | h | 4 | m | -4 | s | 7 | w | 2 |

1. – Результат работы алгоритма

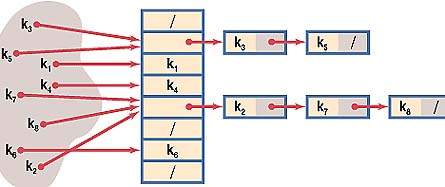
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Слово | Хеш | Слово | Хеш | Слово | Хеш | Слово | Хеш |
| Auto | 21 | Double | 0 | Int | 15 | Struct | 23 |
| Break | 28 | Else | 12 | Long | 26 | Switch | 17 |
| Case | 1 | Enum | 4 | Register | 18 | Typedef | 29 |
| Char | 2 | Extern | 9 | Return | 10 | Union | 30 |
| Const | 8 | Float | 27 | Short | 22 | Unsigned | 24 |
| Continue | 5 | For | 20 | Signed | 3 | Void | 13 |
| Default | 7 | Goto | 19 | Sizeof | 25 | Volatile | 31 |
| Do | 14 | If | 16 | Static | 6 | While | 11 |

1. Разрешение коллизий

Составление хеш-функции – это не вся работа, которую предстоит выполнить программисту, реализующему поиск на основе хеширования. Кроме этого, необходимо реализовать механизм разрешения коллизий. Как и с хеш-функциями существует несколько возможных вариантов, которые имеют свои достоинства и недостатки.

1. Метод цепочек

Метод цепочек – самый очевидный путь решения проблемы. В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, т.к. при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким. Но даже здесь возможна дополнительная оптимизация. Дональд Кнут отмечает, что возможна сортировка списков по времени обращения к элементам. С другой стороны, для повышения быстродействия желательны большие размеры таблицы, но это приведет к ненужной трате памяти на заведомо пустые элементы. На рисунке ниже показана структура хеш-таблицы и образование цепочек при возникновении коллизий.



1. – Структура хеш-таблица и образование цепочек при возникновении коллизий
2. Открытая адресация

Другой путь решения проблемы, связанной с коллизиями, состоит в том, чтобы полностью отказаться от ссылок, просто просматривая различные записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция. Идея заключается в формулировании правила, согласно которому по данному ключу определяется «пробная последовательность», т.е. последовательность позиций таблицы, которые должны быть просмотрены при вставке или поиске ключа K. Если при поиске встречается пустая ячейка, то можно сделать вывод, что K в таблице отсутствует, т.к. эта ячейка была бы занята при вставке, т.к. алгоритм проходил ту же самую цепочку. Этот общий класс методов назван открытой адресацией.

1. Линейная адресация

Простейшая схема открытой адресации, известная как линейная адресация (линейное исследование, linear probing) использует циклическую последовательность проверок

h(K), h(K - 1), …, 0, M – 1, M – 2, …, h(K) + 1

и описывается следующим алгоритмом. Он выполняет поиск ключа K в таблице из M элементов. Если таблица не полна, а ключ отсутствует, он добавляется.

Ячейки таблицы обозначаются как TABLE[i], где 0 ≤ i < M и могут быть или пустыми, или занятыми. Вспомогательная переменная N используется для отслеживания количества занятых узлов. Она увеличивается на 1 при каждой вставке.

Установить i = h(K)

Если TABLE[i] пуст, то перейти к шагу 4, иначе, если по этому адресу искомый, алгоритм завершается.

Установить i = i – 1, если i < 0, то i = i +M. Вернуться к шагу 2.

Вставка, т.к. поиск оказался неудачным. Если N = M – 1, то алгоритм завершается по переполнению. Иначе увеличить N, пометить ячейку TABLE[i] как занятую и установить в нее значение ключа K.

Опыты показывают, что алгоритм хорошо работает в начале заполнения таблицы, однако по мере заполнения процесс замедляется, а длинные серии проб становятся все более частыми.

1. Квадратичная и произвольная адресация

Вместо постоянного изменения на единицу, как в случае с линейной адресацией, можно воспользоваться следующей формулой:

h = h + a2,

где a – это номер попытки. Этот вид адресации достаточно быстр и предсказуем (он проходит всегда один и тот же путь по смещениям 1, 4, 9, 16, 25, 36 и т.д.). Чем больше коллизий в таблице, тем дольше этот путь. С одной стороны, этот метод дает хорошее распределение по таблице, а с другой занимает больше времени для просчета.

Произвольная адресация использует заранее сгенерированный список случайных чисел для получения последовательности. Это дает выигрыш в скорости, но несколько усложняет задачу программиста.

1. Адресация с двойным хешированием

Этот алгоритм выбора цепочки очень похож на алгоритм для линейной адресации, но он проверяет таблицу несколько иначе, используя две хеш-функции h1(K) и h2(K). Последняя должна порождать значения в интервале от 1 до M – 1, взаимно простые с М.

Установить i = h1(K)

Если TABLE[i] пуст, то перейти к шагу 6, иначе, если по этому адресу искомый, алгоритм завершается.

Установить c = h2(K)

Установить i = i – c, если i < 0, то i = i +M.

Если TABLE[i] пуст, то переход на шаг 6. Если искомое расположено по этому адресу, то алгоритм завершается, иначе возвращается на шаг 4.

Вставка. Если N = M – 1, то алгоритм завершается по переполнению. Иначе увеличить N, пометить ячейку TABLE[i] как занятую и установить в нее значение ключа K.

Очевидно, что этот вариант будет давать значительно более хорошее распределение и независимые друг от друга цепочки. Однако, он несколько медленнее из-за введения дополнительной функции.

Дональд Кнут предлагает несколько различных вариантов выбора дополнительной функции. Если M – простое число и h1(K) = K mod M, можно положить h2(K) = 1 + (K mod (M – 1)); однако, если M – 1 четно (другими словами, M нечетно, что всегда выполняется для простых чисел), было бы лучше положить h2(K) = 1 + (K mod (M – 2)).

Здесь обе функции достаточно независимы. Гари Кнотт (Gary Knott) в 1968 предложил при простом M использовать следующую функцию:

h2(K) = 1, если h1(K) = 0 и h2(K) = M – h1(K) в противном случае (т.е. h1(K) > 0).

Этот метод выполняется быстрее повторного деления, но приводит к увеличению числа проб из-за повышения вероятности того, что два или несколько ключей пойдут по одному и тому же пути.

1. Удаление элементов хеш-таблицы

Многие программисты настолько слепо верят в алгоритмы, что даже не пытаются задумываться над тем, как они работают. Для них неприятным сюрпризом становится то, что очевидный способ удаления записей из хеш-таблицы не работает. Например, если удалить ключ, который находится в цепочке, по которой идет алгоритм поиска, использующий открытую адресацию, то все последующие элементы будут потеряны, т.к. алгоритм производит поиск до первого незанятого элемента.

Вообще говоря, обрабатывать удаление можно, помечая элемент как удаленный, а не как пустой. Таким образом, каждая ячейка в таблице будет содержать уже одно из трех значений: пустая, занятая, удаленная. При поиске удаленные элементы будут трактоваться как занятые, а при вставке – как пустые, соответственно.

Однако, очевидно, что такой метод работает только при редких удалениях, поскольку однажды занятая позиция больше никогда не сможет стать свободной, а, значит, после длинной последовательности вставок и удалений все свободные позиции исчезнут, а при неудачном поиске будет требоваться М проверок (где М, напомним, размер хеш-таблицы). Это будет достаточно долгий процесс, так как на каждом шаге №4 алгоритма поиска (см. раздел Адресация с двойным хешированием) будет проверяться значение переменной i.

Рассмотрим алгоритм удаления элемента i при линейной адресации.

Пометить TABLE[i] как пустую ячейку и установить j = i

i = i – 1 или i = i + M, если i стало отрицательным

Если TABLE[i] пуст, алгоритм завершается, т.к. нет больше элементов, о доступе к которым следует заботиться. В противном случае мы устанавливаем r = h(KEY[i]), где KEY[i] – ключ, который хранится в TABLE[i]. Это нам даст его первоначальный хеш-адрес. Если i ≤ r < j или если r < j < i либо j < i ≤ r (другими словами, если r циклически лежит между этими двумя переменным, что говорит о том, что этот элемент находится в цепочке, звено которой мы удалили выше), вернуться на шаг 1.

Надо переместить запись TABLE[j] = TABLE[i] и вернуться на первый шаг.

Можно показать, что этот алгоритм не вызывает снижения производительности. Однако, корректность алгоритма сильно зависит от того факта, что используется линейное исследование хеш-таблицы, поэтому аналогичный алгоритм для двойного хеширования отсутствует.

Данный алгоритм позволяет перемещать некоторые элементы таблицы, что может оказаться нежелательно (например, если имеются ссылки извне на элементы хеш-таблицы). Другой подход к проблеме удаления основывается на адаптировании некоторых идей, использующихся при сборке мусора: можно хранить количество ссылок с каждым ключом, говорящим о том, как много других ключей сталкивается с ним. Тогда при обнулении счетчика можно преобразовывать такие ячейки в пустые. Некоторые другие методы удаления, позволяющие избежать перемещения в таблице и работающие с любой хеш-технологией.

1. Применение хеширования

Одно из побочных применений хеширования состоит в том, что оно создает своего рода слепок, «отпечаток пальца» для сообщения, текстовой строки, области памяти и т. п. Такой «отпечаток пальца» может стремиться как к «уникальности», так и к «похожести» (яркий пример слепка — контрольная сумма CRC). В этом качестве одной из важнейших областей применения является криптография. Здесь требования к хеш-функциям имеют свои особенности. Помимо скорости вычисления хеш-функции требуется значительно осложнить восстановление сообщения (ключа) по хеш-адресу. Соответственно необходимо затруднить нахождение другого сообщения с тем же хеш-адресом. При построении хеш-функции однонаправленного характера обычно используют функцию сжатия (выдает значение длины n при входных данных больше длины m и работает с несколькими входными блоками). При хешировании учитывается длина сообщения, чтобы исключить проблему появления одинаковых хеш-адресов для сообщений разной длины. Наибольшую известность имеют следующие хеш-функции: MD4, MD5, RIPEMD-128 (128 бит), RIPEMD-160, SHA (160 бит). В российском стандарте цифровой подписи используется разработанная отечественными криптографами хеш-функция (256 бит) стандарта ГОСТ Р 34.11—94.

1. Хеширование паролей

Ниже предполагается, что для шифрования используется 128-битный ключ. Разумеется, это не более, чем конкретный пример. Хеширование паролей – метод, позволяющей пользователям запоминать не 128 байт, то есть 256 шестнадцатиричных цифр ключа, а некоторое осмысленное выражение, слово или последовательность символов, называющуюся паролем. Действительно, при разработке любого криптоалгоритма следует учитывать, что в половине случаев конечным пользователем системы является человек, а не автоматическая система. Это ставит вопрос о том, удобно, и вообще реально ли человеку запомнить 128-битный ключ (32 шестнадцатиричные цифры). На самом деле предел запоминаемости лежит на границе 8-12 подобных символов, а, следовательно, если мы будем заставлять пользователя оперировать именно ключом, тем самым мы практически вынудим его к записи ключа на каком-либо листке бумаги или электронном носителе, например, в текстовом файле. Это, естественно, резко снижает защищенность системы.

Для решения этой проблемы были разработаны методы, преобразующие произносимую, осмысленную строку произвольной длины – пароль, в указанный ключ заранее заданной длины. В подавляющем большинстве случаев для этой операции используются так называемые хеш-функции. Хеш-функцией в данном случае называется такое математическое или алгоритмическое преобразование заданного блока данных, которое обладает следующими свойствами:

- хеш-функция имеет бесконечную область определения,

- хеш-функция имеет конечную область значений,

- она необратима,

- изменение входного потока информации на один бит меняет около половины всех бит выходного потока, то есть результата хеш-функции.

Эти свойства позволяют подавать на вход хеш-функции пароли, то есть текстовые строки произвольной длины на любом национальном языке и, ограничив область значений функции диапазоном 0..2N-1, где N – длина ключа в битах, получать на выходе достаточно равномерно распределенные по области значения блоки информации – ключи.

1. Пример работы хеширования на собственной программе

В данном разделе представлен пример работы хеширования на собственной программе, написанной при помощи языка Kotlin для операционной системы Android.

Для реализации функции хеширования был использован класс MessageDigest из библиотеки java.security. Для демонстрации работы функции был создан статический класс HashExample, имеющим метод хеширования с использованием библиотеки. На вход принимается изначальная строка и алгоритм хеширования. Класс MessageDigest возвращает нужный экземпляр класса в соответствие с указанным алгоритмом, после чего хеширует входную строку, преобразованную в массив байт. В заключении, метод возвращает хешированную шестнадцатеричную строку.

object HashExample {

**fun** **doHash**(input: String, algorithm: String = "SHA-256") =

MessageDigest.getInstance(algorithm).digest(

input.toByteArray(UTF\_8)

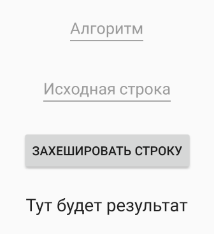
).toHex()

}

Так как метод digest возвращает представление хешированной строки в байтах, был реализован метод преобразования массива байтов в шестнадцатеричную строку.

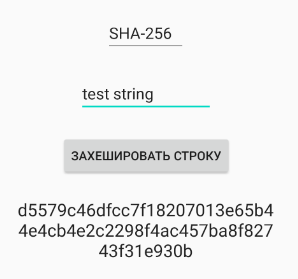
**private** **fun** ByteArray.toHex() = joinToString("") { "%02x".format(it) }

Для выполнения приведенного выше примера был создан простой интерфейс в приложении android, представленный на рисунках ниже.



1. – Изначальный интерфейс

На рисунке 4 представлены два поля ввода, принимающие алгоритм хеширования и исходную строку, а также кнопку и поле вывода, где при нажатии на «захешировать строку» в нижнем поле отобразится результат работы алгоритма. На рисунке 5 представлена работа программы с указанием алгоритма SHA-256.



1. – Хешированная строка (алгоритм SHA-256)

Далее, был протестирован другой алгоритм хеширования MD5 на той же строке. Как видно на рисунке 6, алгоритм генерирует абсолютно другую строку.



1. – Хеширование строки (алгоритм MD5)

Из приведенных выше рисунков можно сделать вывод, что программа корректно выполняет свою работу.

Вывод

В процессе выполнения лабораторной работы было проведено исследование хеш-функций - были изучены ее виды, особенности. Также был реализован пример работы хеширования на языке программирования Kotlin с графическим интерфейсом операционной системы Android.

Приложение А. Исходный код приложения

HashExample:

**class** **EncoderExample**(**private** **val** algorithm: String) {

object HashExample {

**fun** **doHash**(input: String, algorithm: String = "SHA-256") =

MessageDigest.getInstance(algorithm).digest(

input.toByteArray(UTF\_8)

).toHex()

**private** **fun** ByteArray.toHex() = joinToString("") { "%02x".format(it) }

}

Android View:

hashButton.setOnClickListener {

**try** {

**val** result = HashExample.doHash(

stringInputView.text.toString(),

algorithmInputView.text.toString()

)

resultView.text = result

}

**catch** (e: NoSuchAlgorithmException) {

e.printStackTrace()

Toast.makeText(requireContext(),

"Выбран неккоректный алгоритм",

Toast.LENGTH\_SHORT).show()

}

}