|  |  |
| --- | --- |
|  | МИНПРОСВЕЩЕНИЯ РОССИИ  Федеральное государственное бюджетное образовательное учреждение  высшего образования  «Тульский государственный педагогический университет им. Л.Н. Толстого»  (ТГПУ им. Л.Н. Толстого) |

Кафедра информатики и информационных технологий

**КУРСОВАЯ РАБОТА**

по дисциплине

Структуры и алгоритмы компьютерной обработки данных

на тему:

Программная реализация и анализ алгоритмов хеширования данных

Выполнил:

студент 2 курса группы 121691

факультета математики, физики и информатики

направления «Математическое обеспечение и администрирование информационных систем»

профиля «Информационные системы и базы данных»

Боронин Никита Александрович

Руководитель:

доцент, к.ф.-м.н., доцент

Ванькова В.С.

Тула 2021

СОДЕРЖАНИЕ

[ОСНОВНЫЕ ПОНЯТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc73925900)

[ВВЕДЕНИЕ 4](#_Toc73925901)

[ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ХЭШИРОВАНИЯ ДАННЫХ 5](#_Toc73925902)

[1.1. Требования к хеш-функциям 5](#_Toc73925903)

[1.2. Применение хеш-функций 8](#_Toc73925904)

[1.3. Алгоритмы в основе хеш-функций 12](#_Toc73925905)

[**1.3.1. Алгоритмы, определенные на множестве числовых ключей** 12](#_Toc73925906)

[**1.3.2. Алгоритмы, определенные на множестве строковых ключей** 16](#_Toc73925907)

[1.4. Методы разрешения коллизий 18](#_Toc73925908)

[**1.4.2. Метод цепочек** 19](#_Toc73925909)

[**1.4.1. Метод открытой адресации** 20](#_Toc73925910)

[ГЛАВА 2. РЕАЛИЗАЦИЯ ПРОГРАММЫ ДЛЯ ДЕМОНСТРАЦИИ РАБОТЫ АЛГОРИТМОВ ХЕШИРОВАНИЯ ДАННЫХ 21](#_Toc73925911)

[2.1. Реализация хеш-таблицы 21](#_Toc73925912)

[2.2. Реализация хеш-функций 28](#_Toc73925913)

[2.3. Описание главной функции 30](#_Toc73925914)

[2.4. Описание функции заполнения хеш-таблицы 32](#_Toc73925915)

[2.5. Описание функции поиска ключей в хеш-таблице 33](#_Toc73925916)

[2.6. Сравнение результатов работы алгоритмов хеширования данных 34](#_Toc73925917)

[ВЫВОД 38](#_Toc73925918)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 39](#_Toc73925919)

# **ВВЕДЕНИЕ**

В настоящей курсовой работе сформируем и будем использовать следующие термины с соответствующими определениями:

**хэш-функция**: функция, отображающая строки бит в строки бит фиксированной длины и удовлетворяющая следующим свойствам [1]:

1. по данному значению функции сложно вычислить исходные данные, отображаемые в это значение;
2. для заданных исходных данных сложно вычислить другие исходные данные, отображаемые в то же значение функции;
3. сложно вычислить какую-либо пару исходных данных, отображаемых в одно и то же значение.

**хеширование**: процесс, описанный в предыдущем пункте;

**хеш-адрес**, **хеш-сумма** или **хеш-код**: строка бит, являющаяся выходным результатом хеш-функции [1];

**ассоциативный массив**: абстрактный тип данных (АТД), с помощью которого хранятся пары ключ-значение [4];

**хеш-таблица**: это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу [4];

**коллизия**: ситуация, при которой для разных ключей хеш-функция возвращает одинаковый хеш-адрес [4];

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). Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Термин «хеширование» (hashing) в печатных работах по программированию появился сравнительно недавно (1967 г.), 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.

Настоящая курсовая работа имеет своей целью написание простой программы для хеширования данных и последующего поиска их в хеш-таблице.

Цель определяет следующие задачи: изучить теоретические основы хеширования данных; ознакомиться со структурой связного списка и реализовать его на практике; описать наиболее известные алгоритмы хеширования данных: алгоритм деления, алгоритм умножения, алгоритм середины квадрата; ознакомиться со структурой хеш-таблицы и реализовать её; сравнить работу описанных алгоритмов хеширования данных.

Данная работа состоит из двух глав. В первой главе описаны основные сведения из теории хеширования данных, а во второй главе даётся описание методики решения выполнения операций хеширования данных в среде программирования С++.

# **ГЛАВА 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ХЭШИРОВАНИЯ ДАННЫХ**

Сперва стоит подробно разобрать определение хеш-функции. Говоря простым языком, хеш-функция – это такая функция, которая преобразует входные данные (их часто называют ключом) каким-либо методом (методы мы разберем ниже) так, что в результате работы данной функции получается число либо строка определенной длины (обычно длина выходной строки задается степенью двойки).

Результатом работы хеш-функции, как было сказано выше, является число либо последовательность бит фиксированной длины. Их часто называют хеш-суммой или хеш-адресом.

## **1.1. Требования к хеш-функциям**

Хорошая хеш-функция должна соответствовать следующим требованиям [2][3][5]:

1. быстрая скорость работы;
2. хеш-функция обрабатывает ключи любой длины;
3. одинаковая хеш-сумма для одинаковых ключей;
4. хеш-функция возвращает хеш-сумму фиксированной длины (либо число);
5. хороший разброс хеш-суммы;
6. маленькая частота появления коллизий;
7. односторонность хеш-функии.

Далее рассмотрим каждое требование подробнее.

**Одинаковый результат для одинаковых ключей**

Правильная хеш-функция должна всегда и безусловно возвращать один и тот же результат для одних и тех же входных данных.

Чтобы было понятнее рассмотрим на примере: некоторая хеш-функция получает ключ «qwerty», результат работы хеш-функции – «27». Если мы снова отправим строку «qwerty» в нашу хеш-функцию, то она снова должна вернуть «27». Наша функция должна всегда возвращать «27» для ключа «qwerty».

**Фиксированная длина хеш-суммы**

Данное условие подразумевает, что хеш-функция независимо от ключа всегда будет возвращать хеш-суммой строку длины, которая заложена в алгоритме (обычно это двойка в n-ой степени), например, 256 или 512 символов.

**Примечание:** Данное условие соблюдается только для алгоритмов хеширования, результатом которых является строка, а не число!

**Обработка данных любой длины**

Как понятно из названия, хеш-функция должна вернуть хеш-сумму для входных данных любой длины. Иными словами, хорошая хеш-функция должна возвращать хеш-сумму для каждого ключа бесконечно большого множества ключей.

**Примечание:** Данное условие соблюдается только для алгоритмов хеширования, результатом которых является строка, а не число!

Из этого условия вытекает условие уменьшения появления коллизий для разных ключей.

**Уменьшение появления коллизий**

Для раскрытия условия уменьшения появления коллизий следует дать определение понятию коллизия. Коллизией называется ситуация, при которой некоторая хеш-функция для разных ключей возвращает одинаковые хеш-суммы [4].

**Важное замечание:** ни один алгоритм хеш-функции не даст полного отсутствия коллизий. Это сразу становится понятно – множество ключей бесконечно большое, а вот ресурсы компьютера ограничены, но даже в таких условиях разработаны алгоритмы с очень маленькой вероятность коллизии, например алгоритмы, которые в результате своей работы возвращают строку длиной 256 или 512 символов, а это в свою очередь и возможных вариантов хеш-суммы. Почему степень равна 62? Все просто, в таких алгоритмах хеш-сумма состоит из таких символов как: цифры (от 0 до 9, всего 10), латинские буквы верхнего (A-Z, всего 26) и нижнего (a-z, всего 26) регистров. Просуммировав количество символов получим 62. Из комбинаций данных символов получается строка хеш-суммы.

**Быстрая скорость работы**

Хорошая хеш-функция должна выполняться быстро. Это гарантирует малую нагрузку на систему. Чем быстрее работает хеш-функция (чем меньше действий необходимо выполнить процессору), тем лучше, но вместе с удобством использования быстрой хеш-функции появляется проблема безопасности – злоумышленник, зная хеш-сумму какого-либо ключа, может воспользоваться простым перебором ключей, чтобы попытаться узнать исходный ключ.

Рассмотрим на примере: злоумышленник узнал хеш-сумму пароля какого-либо пользователя. Он пробует пропустить часто используемые пароли через хеш-функцию, и, если пользователь установил простой пароль в роде «12345» или «qwerty», то рано или поздно злоумышленник получит хеш-сумму, совпадающую с исходной, и, как следствие, узнает пароль пользователя.

**Односторонность хеш-функции**

Односторонность хеш-функции подразумевает невозможность восстановить исходные данные по хеш-сумме.

Во-первых, алгоритм хеш-функции должен быть построен таким образом, чтобы по результату работы этого алгоритма(хеш-суммы) нельзя было восстановить входные данные(ключ) [2].

Во-вторых, мы подходим к условию хорошего разброса хеш-суммы.

**Хороший разброс хеш-суммы**

Данное условие заключается в том, что хеш-функция для очень похожих ключей должна возвращать максимально непохожие результаты.

Для наглядности разберем пример: некоторая хеш-функция получает на вход ключи «qwerty.» и «qwerty,». Как мы можем заметить, ключи очень похожи, их единственное отличие – последний символ. Плохая хеш-функция вернет для таких ключей, например, «125» и «126» соответственно. Хорошая же хеш-функция, в свою очередь, должна возвращать максимально непохожие хеш-суммы, на пример «999» и «111». В примере хорошей хеш-функции мы можем заметить, что по данным результатам не ясно сильно ли ключи отличаются друг от друга.

## **1.2. Применение хеш-функций**

Хеш-функции применяются в следующих случаях [2][3][4]:

* при построении ассоциативных массивов;
* при поиске дубликатов в сериях наборов данных;
* при построении уникальных идентификаторов для наборов данных;
* при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей в системах защиты в виде хеш-кода;
* и др.

Далее рассмотрим каждый случай отдельно.

**Построение ассоциативных массивов**

Ассоциативный массив — абстрактный тип данных (АТД), с помощью которого хранятся пары ключ-значение [4].

Ассоциативные массивы крайне популярны в прикладном программировании. С их помощью удобно представлять составные данные, содержащие множество различных параметров.

Ассоциативный массив, в отличие от обычного массива (называемого индексированным, так как значения в нем расположены по индексам), нельзя положить в память "как есть". У них нет индексов, которые бы могли определить порядок и простой способ добраться до значений. Для реализации ассоциативных массивов часто используют специальную структуру данных — хеш-таблицу. Она позволяет организовать данные ассоциативного массива удобным для хранения способом. Для этого хеш-таблица использует две вещи: индексированный массив и функцию для хеширования ключей.

**При поиске дубликатов в сериях наборов данных**

Смысл данного метода заключается в том, чтобы создать хеш-таблицу (ассоциативный массив про который мы говорили выше) для основного массива данных, а затем прогонять второй массив данных через хеш-функцию и обращаться к ячейке хеш-таблицы первого массива данных с индексом, равным хеш-сумме ключа из первого массива. Если в хеш-таблице по этому индексу лежит значение, и ключи обоих массивов совпадают, то можно быть уверенным в том, что мы нашли совпадение данных.

**Построение уникальных идентификаторов для наборов данных**

Принцип данного метода невероятно прост: хеш-сумма является идентификатором для каких-либо данных. Рассмотрим на примере:

Пусть есть некоторая хеш-функция h(key), которая для входящей строки возвращает шести-символьный хеш. Имеем ключ «Автор», после работы хеш-функции h(key), где key = «Автор», получим строку «aGs4p1». В данном случае строка «aGs4p1» и будет являться уникальным идентификатором для строки «Автор».

**Для сравнения файлов**

Принцип этого метода заключается в том, чтобы сравнивать хеш-суммы файлов, и, отталкиваясь от значений этих хеш-сумм, делать выводы.

К примеру, мы хотим передать какой-то очень важный файл, ошибка в котором может привести к плачевным последствиям. Вместе с файлом мы передаем хеш-сумму этого файла, а пользователь, который получит этот файл, в свою очередь, должен сравнить хеш-сумму полученного файла с исходной хеш-суммой. Если хеш-суммы обоих файлов совпадают, то можно сделать вывод, что данные успешно передались. В противном случае, разные хеш-суммы для одинаковых на первый взгляд файлов могут говорить о том, что файл был передан с непредвиденной или намеренной ошибкой, то есть файлы различны по содержанию.

**Хранение паролей**

Данный метод используется для безопасного хранения паролей, пользователей каких-либо сервисов. Смысл заключается в том, что при регистрации пароль пользователя записывается в базу данных в виде хеш-суммы некоторой хеш-функции, а при входе на сервер отправляется хеш-сумма введенного пароля для сравнения с исходной хеш-суммой. Таким образом, пароль пользователя в своем исходном значении никак не участвует в механике входа в аккаунт. Ну, а, поскольку, все пароли в базе данных хранятся в виде хеша, то, даже если злоумышленники получат доступ к этой базе данных, у них все равно не выйдет получить доступ к аккаунтам пользователей, ведь, как мы помним, одно из требований к хорошей хеш-функции – это невозможность восстановить исходные данные по хешу.

## **1.3. Алгоритмы в основе хеш-функций**

### **1.3.1. Алгоритмы, определенные на множестве числовых ключей**

**Алгоритм деления**

В своей книге «Структуры и алгоритмы обработки данных: учебное пособие» Дроздов Сергей Николаевич описывает алгоритм деления таким образом:

«Этот алгоритм является самым простым, хотя и далеко не лучшим. Он выражается следующей формулой:

**(1.1)**

Здесь N – размер хеш-таблицы, так что вычисленное значение индекса будет в пределах от 0 до N-1, как принято в программах на C. Если массив описан от 1 до N (как обычно бывает в языке Pascal), то нужно еще прибавить к индексу единицу.

Вычисление по формуле (1.1) требует выполнения всего одной команды целочисленного деления с остатком. Хотя для большинства процессоров это не самая быстрая команда, требование быстрого вычисления можно считать вполне удовлетворенным. С требованием хорошего рассеивания дело обстоит гораздо хуже. Хотя данная функция покрывает все значения индексов, отображая на каждый из них примерно одинаковое количество допустимых ключей, она плохо справляется с рассеиванием неравномерностей. Во-первых, легко видеть, что соседние (отличающиеся на 1) значения ключей будут почти всегда отображаться на соседние позиции хеш-таблицы. Это уже не хорошо с точки зрения разрешения возможных коллизий. Во-вторых, если значительная часть ключей отстоят друг от друга на расстояние, имеющее общий множитель с размером таблицы N, то число используемых позиций таблицы резко уменьшается, что приведет к частым коллизиям.

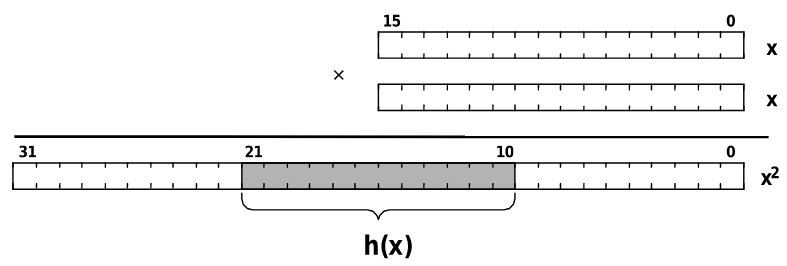
Поясним сказанное таким примером. Пусть N = 10000, а значения всех используемых ключей заканчиваются на 0. Очевидно, что тогда и все значения h(х) тоже будут заканчиваться на 0, т.е. будет использоваться лишь 1/10 часть всех позиций таблицы, что приведет к частым коллизиям. Чтобы избежать описанной неприятности, в случае использования алгоритма деления стараются в качестве размера хеш-таблицы выбирать простое число, не имеющее общих делителей ни с какими другими числами. Но даже с этим уточнением данный алгоритм используют лишь в задачах, не требующих особо качественного хеширования.» [2]

**Алгоритм середины квадрата**

Данный алгоритм Дроздов С.Н. описывает следующим образом:

«Алгоритм середины квадрата работает следующим образом. Значение ключа **x** возводится в квадрат (с двойной точностью, чтобы избежать переполнения разрядной сетки), а затем из двоичного представления результата вырезается такое количество средних разрядов, которое достаточно для представления индекса таблицы.

Суть алгоритма иллюстрируется на рис. 2, где принято, что значение ключа есть 16-разрядное число, его квадрат занимает 32 разряда, а для представления индекса используется 12 разрядов.



*Рис. 2. Выделение середины квадрата ключа* [2]

Для данного алгоритма, в отличие от предыдущего, в качестве размера хеш-таблицы удобно выбрать некоторую степень двойки, например, 1 024, 4096, 65 536 и т.п. При этом разряды, вырезанные из середины квадрата (соответственно, 10, 12, 16 разрядов), образуют готовое значение индекса.

Почему рекомендуется брать разряды из середины квадрата, а не младшие либо старшие разряды? Потому что на значение средних разрядов в равной степени влияют как младшие, так и старшие разряды ключа. Это позволяет рассеять некоторые виды неравномерностей. Например, если значительная часть ключей — небольшие числа в пределах, скажем, , то их квадраты будут в пределах , т.е. старшие 12 разрядов будут содержать нулевые значения. И наоборот, если ключами являются числа, кратные некоторой степени двойки, то их квадраты будут содержать много нулей в младших разрядах. Использование средних разрядов позволяет избежать этих неприятностей (если нулей в ключе не слишком много).

В вычислительном отношении алгоритм середины квадрата также достаточно экономен. Фактически он требует одной команды целочисленного умножения, а затем команды сдвига средних разрядов вправо и команды конъюнкции для отбрасывания ненужных старших разрядов.

Недостатком алгоритма является то, что в случае большого количества нулей в младших или в старших разрядах ключа результат может содержать много нулей даже в средних разрядах.» [2]

**Алгоритм умножения**

Алгоритм умножения Дроздов С.Н. описал так:

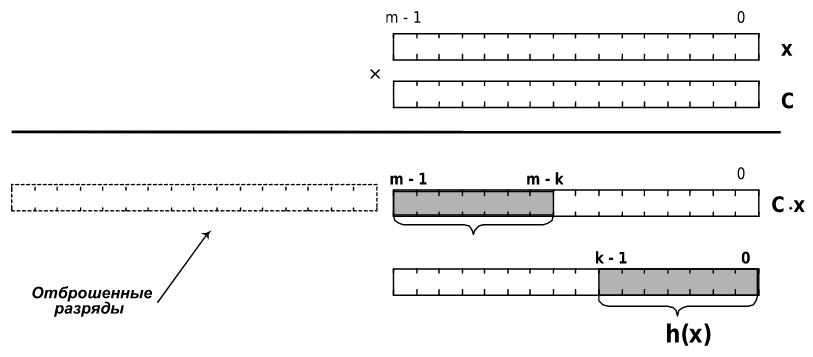
«По сути, алгоритм умножения отличается от предыдущего тем, что значение ключа умножается не на себя самого, а на некоторую числовую константу. Такой алгоритм может быть реализован очень эффективно, если учитывать особенности двоичного представления чисел в компьютере.

Первой операцией, которая выполняется в этом алгоритме, является умножение ключа **x** на константу **C** с отбрасыванием старших разрядов произведения, не уместившихся в разрядной сетке. Цель этой операции — рассеять ключи по всему множеству допустимых целых чисел. Вторая операция — переход от диапазона всех целых чисел к диапазону индексов хеш-таблицы. Для этого выполняют умножение числа на размер хеш-таблицы **N**, но из произведения выделяют только те разряды, которые выходят за разрядную сетку целых чисел. Очевидно, что число, образованное этими разрядами, будет находиться в пределах от 0 до **N**-1, причем, если результат первого умножения был равномерно рассеян по множеству целых чисел, то значение хеш-функции будет так же хорошо рассеяно по множеству индексов.

Описанные действия можно записать в виде формул с использованием операций **mod** и **div**, однако это намеренно не сделано, чтобы не создавать соблазна использовать **mod** и **div** на практике. На самом деле, все гораздо проще, особенно если размер хеш-таблицы **N =**. При выполнении умножения на константу **C** старшие разряды произведения просто отбрасываются, а вместо умножения на **N** с выделением переполняющих разрядов выполняют просто сдвиг числа вправо, так, чтобы оставить только **k** старших разрядов.

Работа алгоритма показана на рис. 3.

Как видно из рисунка, весь алгоритм сводится к одной операции умножения и одному сдвигу на **m - k** разрядов вправо.



*Рис. 3. Хеширование по алгоритму умножения* [2]

Важную роль в алгоритме умножения играет константа-множитель **C**. Приведем здесь лишь несколько простых советов, позволяющих избежать грубых ошибок при выборе (предполагается, что размер хеш-таблицы **N =**):

- значение **C** обязательно должно быть нечетным, а еще лучше - простым числом;

- не следует выбирать слишком маленькие значения; лучше, если **C**, по крайней мере, не меньше 1000;

- нежелательно также использовать числа, лишь на несколько единиц отличающиеся от одной из степеней двойки, например, такие, как 2047 (почти равно =2048) или 4099 (близко к =4096).» [2]

### **1.3.2. Алгоритмы, определенные на множестве строковых ключей**

Про хеширование строковых ключей Дроздов С.Н. говорит следующее:

«Следует подчеркнуть, что способ преобразования строки в число не менее важен для качественного хеширования, чем дальнейшее преобразование полученного числа. Пусть, например, в результате непродуманного преобразования все строковые ключи отображаются только в множество четных целых чисел. Это значит, что половина возможных значений числовых ключей не будет использоваться, а это затруднит хорошее рассеяние остальных значений. Если же преобразование строки настолько неудачно, что количество разных значений числового ключа получается меньше, чем размер хеш-таблицы, то уже никакая изощренная хеш-функция не сможет это компенсировать.

Хорошее рассеяние строковых ключей возможно только в том случае, если результат преобразования в число существенно зависит от каждого символа строки. Обычно это достигается следующим образом. Строка разбивается на отрезки длиной от 2 до 4 байт, каждый из которых рассматривается как двоичное число. Затем с помощью циклического повторения некоторой несложной операции полученные числа «сворачиваются» в одно число. Что это может быть за операция? Очень хорошо подходит логическая операция «Исключающее ИЛИ» (**XOR**), которая на любом процессоре выполняется как одна машинная команда и при этом существенно зависит от каждого бита каждого из своих операндов. Другой хороши кандидат — обыкновенная операция целочисленного сложения. В этом случае, однако, может возникнуть переполнение разрядной сетки; ничего страшного, но лучше все-таки не терять переполняющие разряды и прибавлять их к младшим разрядам результата.

Весьма желательно перед каждым сложением или **XOR** применять к ранее накопленному результату операцию циклического сдвига на один разряд. (Для не знающих Ассемблера: циклическим сдвигом называется сдвиг двоичных разрядов влево или вправо, при котором биты, выходящие за разрядную сетку, переносятся на противоположный конец числа.) Если не делать сдвиг, то некоторые строки, отличающиеся друг от друга только перестановкой букв, будут давать одинаковые числовые значения.

В принципе, полученные таким образом числовые ключи обычно бывают хорошо рассеяны на множестве всех целых чисел. Однако случается и иначе. Часто ключ бывает описан как строковая переменная достаточно большой длины, но при этом большая часть реально встречающихся ключей состоит из одного-двух-трех символов, дополненных справа пробелами до нужной длины. При этом все хитрые способы свертки ключа будут «месить» в основном одни и те же машинные слова, соответствующие пробелам.

Для большей уверенности в хорошем рассеивании ключей желательно применить к свернутому ключу описанные выше алгоритмы хеширования целочисленного ключа.

Хорошие алгоритмы хеширования строковых ключей могут быть также получены в классе ***полиномиальных хеш-функций***. Эти алгоритмы основаны на вычислении полинома от некоторого простого числа **p**, при этом в качестве коэффициентов полинома используются коды последовательных символов строки. Другими словами, на каждом шаге цикла текущее значение **h** умножается на константу **p** и к произведению прибавляется код следующего символа хешируемой строки. Поскольку значения полинома быстро становятся очень большими, все вычисления ведутся с отбрасыванием старших битов результата, выходящих за разрядную сетку. Как доказывается в теории чисел, если **p -** простое число, то при возведении **p** в степень по модулю **n** первые **n** степеней примут значения всех чисел от **0** до **n-1**. Это снижает вероятность случайного совпадения значений хеш-функции для разных ключей.

Для ключей, принадлежащих к иным типам данных (вещественные числа, даты и т.п.), можно аналогичным образом придумать способы преобразования в целые числа с перемешиванием.» [2]

## **1.4. Методы разрешения коллизий**

Рассмотрим следующий важный вопрос, связанный с хешированием. Поскольку никакая хеш-функция не даст возможности полностью избежать коллизий, следует решить, что делать в случае, если коллизия все-таки возникла.

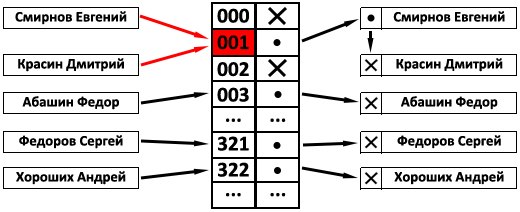
Дроздов С.Н. по поводу коллизий говорит следующее:

«В тех случаях, когда при попытке занести в хеш-таблицу запись со значением ключа **x2** выясняется, что соответствующий элемент массива уже занят (т.е. **h(x2) = h(x1)**, где **x1** – ключ, ранее уже занесенный в таблицу), должен быть использован какой-либо не слишком трудоемкий способ отыскания свободного места в массиве. В последствии тот же метод должен использоваться при поиске значения ключа **k2** в таблице.» [2]

Стоит отметить, что существует довольно внушительное множество способов обработки коллизий, мы же рассмотрим два наиболее известных из них: метод открытой адресации и метод цепочек.

### **1.4.2. 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 рис. 4 демонстрируется реализация метода цепочек при разрешении коллизий. На ключ 002 претендуют два значения, которые организуются в линейный список. [4]



*Рис. 4. Визуализация метода цепочек*

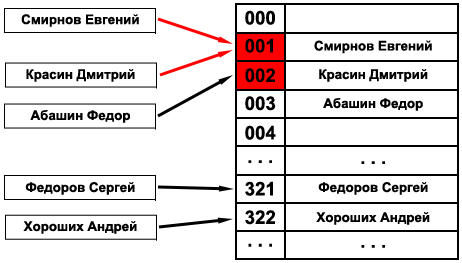
Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента. [2]

Операции поиска или удаления данных в худшем случае требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. [2]

### **1.4.1. 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 не содержит никакой информации.

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 (если мы хотим получить значение этого ключа) или пустая позиция (если мы хотим записать данные в таблицу) в таблице. Для вычисления шага можно также применить формулу, которая и определит способ изменения шага. На рис. 5 разрешение коллизий осуществляется методом открытой адресации. Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще незанятое) место в таблице. [2]



*Рис. 5. Визуализация метода открытой адресации*

# **ГЛАВА 2. РЕАЛИЗАЦИЯ ПРОГРАММЫ ДЛЯ ДЕМОНСТРАЦИИ РАБОТЫ АЛГОРИТМОВ ХЕШИРОВАНИЯ ДАННЫХ**

В качестве основной среды разработки программы для демонстрации работы алгоритмов хеширования целочисленных ключей целого типа была выбрана программа Microsoft Visual Studio Community 2019 года выпуска. Это бесплатная версия программы, которой может пользоваться любое физическое лицо, а доступного в версии Community функционала с избытком будет достаточно для поставленной нами задачи – продемонстрировать работу алгоритмов хеширования данных.

Основным языком для написания нашей будущей программы выберем язык C++ (C plus plus). Основой программы станет консоль. Именно в ней мы и разработаем удобный нам интерфейс и проведем несколько тестов, а после проделанной работы, проведем анализ результатов и подведем итоги.

## **2.1. Реализация хеш-таблицы**

Для разрешения коллизий будем использовать метод цепочек. Так как метод цепочек подразумевает использование списков данных, то сначала следует реализовать связный список.

Опишем класс List, его закрытые данные и открытые методы:

/\* КЛАСС ОДНОСВЯЗНЫЙ СПИСОК \*/

template <typename T>

class List {

public:

List(); // конструктор по умолчанию

~List(); // деструктор

void pop\_front(); // удаление первого в списке элемента

void push\_back(T data); // добавление нового элемента в конец списка

void push\_front(T data); // добавление нового элемента в начало списка

void insert(T data, int index); // вставка нового элемента в позицию по индексу

void remove(int index); // удаление элемента по индексу

void clear(); // полная очистка списка

int size() { return Size; } // получение количества элементов в списке

int find(T key); // поиск элемента в списке

T& operator[](const int index); // перегрузка оператора []. Возвращает элемент по заданному индексу

private:

template <typename T>

/\* Вспомогательный класс узел \*/

class Node {

public:

Node\* pNext; // указатель на следующий элемент списка

T data; // элемент списка

/\* конструктор узла списка \*/

Node(T data = T(), Node\* pNext = nullptr) {

this->data = data;

this->pNext = pNext;

}

};

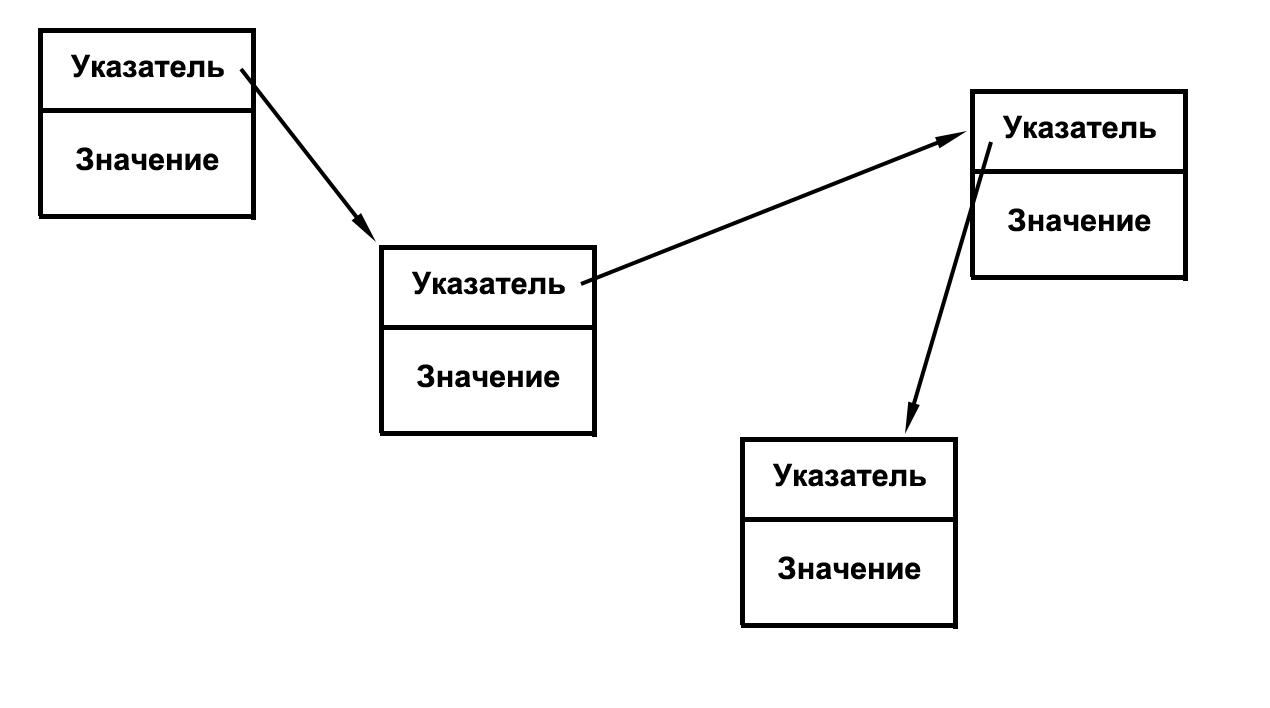
int Size; // Количество элементов в списке

Node<T> \*Head; // Указатель на начало списка

Node<T>\* End; // Указатель на конец списка

};

Хочется отметить закрытый вспомогательный класс Node (Узел). Объект данного класса хранит какие-либо данные и адрес на следующий объект такого же типа. Таким образом, мы и получаем необходимый нам односвязный список. На рис. 6 можно увидеть, как это выглядит.



*Рис. 6. Визуальное представление односвязного списка*

Теперь подробно рассмотрим методы и конструктор класса List.

Конструктор имеет следующий вид:

template <typename T>

List<T>::List() {

Size = 0;

Head = nullptr;

End = nullptr;

}

При создании нового (т.е. пустого) списка, задаем количество элементов в списке (за это отвечает переменная Size, как я уже говорил выше) равным нулю и указатели на начало и конец списка делаем пустыми (nullptr). Таким образом мы получаем пустой список без указателей на первый и последний элементы списка, так как их нет.

Следующими, немало важными методами нашего класса List являются методы вставки нового элемента в начало и в конец списка:

// Вставка нового элемента в начало списка

template<typename T>

void List<T>::push\_front(T data)

{

Head = new Node<T>(data, Head);

Size++;

}

// Вставка нового элемента в конец списка

template<typename T>

void List<T>::push\_back(T data)

{

if (Head == nullptr) {

Head = new Node<T>(data);

End = Head;

}

else {

End->pNext = new Node<T>(data);

End = End->pNext;

}

Size++;

}

При вставке нового элемента в начало списка, мы объявляем и инициализируем новый узел, который будет хранить указатель на текущий первый элемент в списке (Head) и новый элемент списка (data). Далее записываем в переменную-указатель Head адрес на только что созданный узел и увеличиваем количество элементов списка (Size) на единицу.

В случае вставки нового элемента в конец списка у нас может быть две ситуации: в списке еще нет элементов (т.е. список пуст) и в списке уже есть элементы.

В первой ситуации (условный оператор if) мы делаем тоже самое, что и при в ставке нового элемента в начало списка, но, помимо этого, мы записываем в переменную-указатель на конец списка (End) адрес на начало списка (т.к. мы вставляем новый элемент в конец пустого списка, т.е. теперь в списке единственный элемент, он одновременно будет являться и первым, и последним).

Также, следует рассказать о методах удаления элементов из списка:

// удаление первого в списке элемента

template<typename T>

void List<T>::pop\_front()

{

Node<T> \*temp = Head;

Head = Head->pNext;

delete temp;

Size--;

}

// удаление элемента списка по индексу

template<typename T>

void List<T>::remove(int index)

{

if (index == -1) return;

else if (index == 0) pop\_front();

else {

Node<T>\* previous = this->Head;

for (int i = 0; i < index - 1; i++) {

previous = previous->pNext;

}

Node<T>\* toDelete = previous->pNext;

previous->pNext = toDelete->pNext;

delete toDelete;

Size--;

}

}

// очистка списка

template<typename T>

void List<T>::clear()

{

while (Size) {

pop\_front();

}

}

Метод удаления первого в списке элемента делает следующее:

1. Создается временный указатель на текущий первый элемент, чтобы в последствии можно было освободить память, т.к. мы удаляем этот элемент;
2. В переменную-указатель на первый элемент записывается адрес на следующий элемент списка;
3. Высвобождается память, выделенная под элемент, который мы ходим удалить;
4. Уменьшается количество элементов в списке на единицу.

С методом очистки списка все еще проще – пока список не пустой запускается метод удаления первого элемента.

Наконец мы подошли в методу удаления элемента в списке по заданному индексу. Если индекс равен нулю, то мы запускаем метод удаления первого элемента в списке. В противном случае, мы идем до элемента, индекс которого на единицу меньше заданного, затем запоминаем элемент указатель на элемент по заданному индексу, а в элемент с индексом на единицу меньше заданного записываем адрес элемента с индексом на единицу больше заданного. Таким образом, мы убрали элемент по заданному индексу из нашего списка, остается только высвободить память выделенную под этот элемент, так как больше он нам не нужен, что мы и делаем далее. Разумеется, не забываем уменьшить количество элементов в списке на единицу.

Теперь рассмотрим методы вставки и обращения к элементу по заданному индексу:

// вставка в список на позицию-индекс

template<typename T>

void List<T>::insert(T data, int index)

{

if (index == 0) push\_front(data);

else if (index == Size - 1) push\_back(data);

else {

Node<T>\* previous = this->Head;

for (int i = 0; i < index - 1; i++) {

previous = previous->pNext;

}

Node<T>\* newNode = new Node<T>(data, previous->pNext);

previous->pNext = newNode;

Size++;

}

}

// возвращение элемента по индексу

template<typename T>

T& List<T>::operator[](const int index)

{

int counter = 0;

Node<T>\* current = this->Head;

while (current != nullptr) {

if (counter == index) {

return current->data;

}

current = current->pNext;

counter++;

}

}

Если мы вставляем новый элемент в начало или конец списка, то запускаем соответствующие методы, иначе доходим до элемента с индексом на единицу меньше заданного, в нем записываем указатель на новый элемент списка, а в новом элементе списка записываем указатель на следующий элемент списка и увеличиваем количество элементов списка на единицу.

Для обращения к элементу списка по индексу мы перегрузим (зададим свое действие) оператор []. В скобки помещается индекс узла, в котором хранится элемент списка, а возвращается значение переменной Data этого узла.

Остается рассмотреть метод поиска в списке по заданному элементу:

// поиск в списке по элементу

template<typename T>

int List<T>::find(T key)

{

int counter = 0;

Node<T>\* current = this->Head;

while (current != nullptr) {

if (key == current->data) {

return counter;

}

current = current->pNext;

counter++;

}

return -1;

}

Мы проходимся по всем элементам списка и сравниваем очередной элемент с заданным. Если такой элемент в списке есть, то возвращается индекс этого элемента в списке (Если таких элементов несколько, вернется индекс на самый первый такой элемент в списке). Если, пройдя по всем элементам списка мы не нашли заданный элемент, то возвращаем -1 (т.к. индексы идут от 0 до N).

Таким образом мы организовали класс List, который будет служить элементом хеш-таблицы. Хеш-таблица мы организуем как обычный индексированный массив списков, а в случае коллизии, просто будем добавлять очередной элемент в конец списка.

## **2.2. Реализация хеш-функций**

На практике реализуем следующие алгоритмы для хеширования целых чисел: алгоритм деления, алгоритм умножения, алгоритм середины квадрата.

Алгоритм деления опишем следующим образом:

// хеширование алгоритмом деления

int GetHash\_div(const long long& key, long long d = HASH\_TABLE\_SIZE) {

return key % d;

}

Здесь key – целочисленный ключ, d – размер хеш-таблицы. Функция возвращает остаток от деления на размер хеш-таблицы.

Теперь опишем алгоритм умножения, тут слегка сложнее:

// хеширование алгоритмом умножения

long long GetHash\_mult(const long long& key, long long N = HASH\_TABLE\_SIZE) {

double A = 0.618033;

long long h = N \* fmod(key \* A, 1);

return h;

}

Здесь N – размер хеш-таблицы, A – некоторая константа в диапазоне от 0 до 1 включительно, а функция fmod() отбрасывает целочисленную часть, оставляя только хвост числа. Таким образом имеем функцию, которая зависит от ключа, а возвращаемые хеш-индексы принимают значения от 0 до N – 1, что нам и нужно.

Последним алгоритмом хеширования, который мы опишем, будет алгоритм середины квадрата. Он выглядит следующим образом:

// хеширование алгоритмом середины квадрата

long long GetHash\_midSquar(const long long& key, long long N = HASH\_TABLE\_SIZE) {

long long hash = abs(key \* key);

hash = hash >> SizeOf(hash);

return hash % N;

}

Что происходит в этом коде:

1. Получение квадрата ключа (hash = key \* key);
2. Побитовый сдвиг полученного на предыдущем шаге значения в право на количество цифр этого значения;
3. Отбрасывание старших разрядов (деление по модулю, hash % d) промежуточного хеша для получения хеш-индексов в диапазоне от 0 до N – 1.

Пункт 2 стоит объяснить подробнее. Предположим, что очередной ключ равен 45. Его двоичное представление равно 101101. Побитовый сдвиг вправо на n бит подразумевает отбрасывание n нулей или единиц справа и дописывание n нулей слева в двоичном представлении числа. Проведя побитовый сдвиг вправо для числа 45 на два бита (количество цифр числа 45 равно двум), из 101101 получим 001011, что эквивалентно числу 11 в десятичной системе счисления.

Деление по модулю (пункт 3) мы выполняем для обработки случаев, когда ключ превышает размер хеш-таблицы.

## **2.3. Описание главной функции**

Подключим необходимые заголовочные файлы и объявим несколько глобальных констант и переменных для установки параметров хеширования и отслеживания ряда значений (все переменные и константы описаны в коде, поэтому не вижу необходимости дополнять что-либо):

#include <iostream> // стандартный ввод/вывод

#include <fstream> // ввод/вывод файл

using namespace std;

const long long HASH\_TABLE\_SIZE = 10000; // размер хеш-таблицы

long AVAILABLE\_KEYS = 0; // Количество случаев, когда хешированный ключ хешировался еще раз

long WORK\_TIME; // время выполнения программы

// Какую функцию хеширования использовать

// 1 - алгоритм деления

// 2 - алгоритм умножения

// 3 - алгоритм середины квадрата

const int MAIN\_HASH\_FUNCTION = 3;

const long long NUMBER\_OF\_KEYS = HASH\_TABLE\_SIZE; // количество ключей для заполнения хеш-таблицы

long long FILLED\_CELL = 0; // заполненные ячейки хеш-таблицы

List<long long> HashTable[HASH\_TABLE\_SIZE];// Хеш-таблица

ifstream fin("input.txt"); // файл с множеством целочисленных ключей

ifstream fkeys("input2.txt"); // файл с множеством целочисленных ключей, которые необходимо найти в хеш-таблице

long long FINDED\_KEYS = 0; // кол-во найденных ключей

long long SEARCHING\_KEYS = 0; // кол-во искомых ключей

Теперь напишем главную функцию программы:

int main() {

setlocale(LC\_ALL, "ru"); // подключение руской локализации для консоли

cout << "Начато заполнение хеш-таблицы..." << endl ;

WORK\_TIME = clock(); // запись времени старта заполнения хеш-таблицы

long long collisions = GenHash(HashTable, NUMBER\_OF\_KEYS); // кол-во коллизий и запуск функции заполнения хеш-таблицы

WORK\_TIME = clock() - WORK\_TIME; // время окончания заполнения хеш-таблицы

cout << "Х Е Ш И Р О В А Н И Е К Л Ю Ч Е Й З А В Е Р Ш Е Н О" << endl;

cout << "Время заполнения хеш-таблицы в мс = " << WORK\_TIME << endl << endl;

cout << "Функция хеширования: ";

switch (MAIN\_HASH\_FUNCTION) {

case 1: cout << "алгоритм деления" << endl; break;

case 2: cout << "алгоритм умножения" << endl; break;

case 3: cout << "алгоритм середины квадрата" << endl; break;

default: cout << "не выбрана" << endl;

}

cout << "Размер хеш-таблицы: " << HASH\_TABLE\_SIZE << endl;

cout << "Количество ключей: " << NUMBER\_OF\_KEYS << endl;

cout << "Кол-во коллизий = " << collisions << endl;

cout << "Процент коллизий = " << (float)collisions / (NUMBER\_OF\_KEYS - AVAILABLE\_KEYS) \* 100 << "%" << endl;

cout << "Процент заполненных ячеек = " << (float)FILLED\_CELL / HASH\_TABLE\_SIZE \* 100 << "%" << endl;

cout << "Кол-во совпадающих ключей = " << AVAILABLE\_KEYS << endl;

cout << endl << "Начат поиск ключей в хеш-таблице..." << endl;

WORK\_TIME = clock(); // Начало поиска ключей

FindKeys(HashTable, fkeys); // функция поиска ключей из заданного файла в хеш-таблице

WORK\_TIME = clock() - WORK\_TIME; // Завершение поиска ключей

cout << "П О И С К К Л Ю Ч Е Й З А В Е Р Ш Е Н" << endl;

cout << "Время поиска ключей в хеш-таблице в мс = " << WORK\_TIME << endl << endl;

cout << "Колличество искомых ключей = " << SEARCHING\_KEYS << endl;

cout << "Колличество найденных ключей = " << FINDED\_KEYS << endl;

fin.close();

return 0;

}

Прототип функции заполнения хеш-таблицы числами из подключаемого файла выглядит следующим образом GenHash(List<T>\* HashTable, long long iterations), где HashTable – хеш-таблица, а iterations – количество ключей, которые будут взяты из подключаемого файла, хешированы и записаны в хеш-таблицу.

Прототип функции поиска ключей из заданного файла выглядит следующим образом FindKeys(List<T> htable, ifstream& file), где htable – хеш-таблица, а file – заданный файл с ключами, которые необходимо найти в хеш-таблице.

Мы также можем заметить, что после выполнения функции заполнения хеш-таблицы в консоль выводятся те самые глобальные переменные и константы, которые мы объявляли в верхней части кода программы. Это нужно для того, чтобы иметь представление о работе всей программы.

## **2.4. Описание функции заполнения хеш-таблицы**

Мы уже говорили про функцию заполнения хеш-таблицы, когда описывали главную функцию. Теперь опишем её:

template <typename T>

long long GenHash(List<T> \*hTable, long long iterations) {

int collisions = 0; // кол-во коллизий

T key; // ключ произвольного типа

long long hash; // значение хеш-суммы

long long items; // кол-во элементов в очередном списке в хеш-таблице

long long key\_index = -1; // индекс искомого элемента в списке

for (long long i = 0; i < iterations; i++) {

GenKey(key); // получение ключа из файла

hash = GetHash(key); // получение хеш-суммы ключа

items = hTable[hash].size(); // получение кол-ва элементов списка

key\_index = hTable[hash].find(key); // поиск ключа в хеш-таблице

// если ключа нет в хеш-таблице

if (key\_index == -1) {

// если в ячейке хеш-таблицы уже есть данные, то увеличиваем кол-во коллизий на единицу, иначе увеличиваем кол-во заполненных ячеек на единицу

if (items > 0) { collisions++; }

else { FILLED\_CELL++; }

hTable[hash].push\_back(key); // добавляем ключ в ячейку хеш-таблицы

}

// если ключ уже есть в хеш-таблице, то увеличиваем кол-во случаев одинаковых ключей

else AVAILABLE\_KEYS++;

}

return collisions; // возвращаем кол-во коллизий

}

Данная функция получает в аргументах указатель на хеш-таблицу и количество ключей, которые будут записаны в эту хеш-таблицу. Сначала идет объявление вспомогательных переменных (key, collisions и т.д.), далее из файла с ключами берется очередной ключ и ищется его хеш-сумма. Следующим действием выполняется проверка на наличие (метод find(int index) мы уже рассматривали выше) очередного ключа в хеш-таблице: если ключ есть, то мы увеличиваем количество таких случаев (когда из файла был взят идентичный ключ) на единицу, иначе мы проверяем наличие данных в очередной ячейке хеш-таблицы – если данных нет, увеличиваем количество заполненных ячеек, в противном случае увеличиваем число коллизий. В случае, если очередного ключа еще нет в хеш-таблице, то не забываем записать его в таблицу.

## **2.5. Описание функции поиска ключей в хеш-таблице**

Функция поиска ключей из входного файла в хеш-таблице выглядит следующим образом:

template <typename T>

void FindKeys(List<T> \*htable, ifstream& file) {

T temp;

long long hash;

int i = 0;

while (!file.eof() && i < NUMBER\_OF\_KEYS) {

file >> temp;

SEARCHING\_KEYS++;

hash = GetHash(temp);

if (htable[hash].find(temp) != -1) {

FINDED\_KEYS++;

}

i++;

}

return;

}

В аргументах функция получает указатель на хеш-таблицу и указатель на файловый поток с множеством ключей, которые необходимо найти в хеш-таблице. Первым действием объявляем вспомогательные переменные (temp, hash) для хранения очередного ключа и его хеш-адреса. Затем, пока файл с ключами не закончился и пока количество взятых из файла ключей меньше размера хеш-таблицы, поочередно берем из файла ключ, получаем его хеш с помощью хеш-функции и, если ключ был найдет в списке по хеш-адресу, то увеличиваем количество найденных на единицу. Не забываем увеличить количество искомых ключей на единицу каждый раз, когда берем очередной ключ из файла.

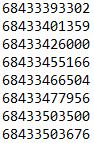
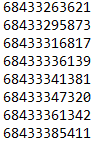
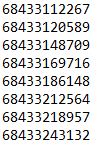
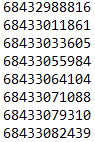
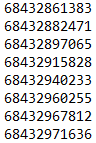
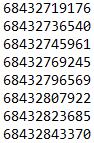
## **2.6. Сравнение результатов работы алгоритмов хеширования данных**

Мы уже описали три алгоритма хеширования целых неотрицательных чисел, главную функцию, функцию заполнения хеш-таблицы, функцию поиска ключей в хеш-таблице и представили хеш-таблицу в виде массива списков. Теперь проведем несколько тестовых запусков программы с заданными параметрами, по одному тесту на каждую хеш-функцию.

В качестве массива ключей выступит заведомо заполненный целыми неотрицательными числами файл. К сожалению, этот файл слишком велик (порядка 52Мб и общим количеством чисел равным 4 194 298), чтобы приложить его к данной курсовой работе, поэтому содержимое этого файла будет вынесено отдельно от неё. Вот первые 120 чисел из данного файла:



А вот последние 48 чисел данного файла:



Как мы можем заметить, данные числа были сгенерированы случайно таким образом, что очередное число не меньше предыдущего.

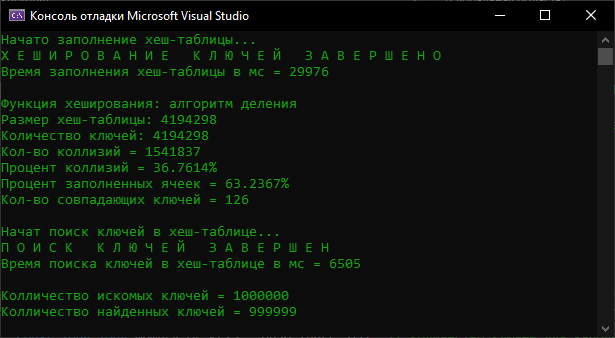
Файл с ключами, которые необходимо найти в хеш-таблице построен таким же образом, как и файл с множеством ключей, которые заполняют таблицу и содержит 1 000 000 ключей.

Все тесты будут проводиться со следующими параметрами для программы:

* Размер хеш-таблицы – 4 194 298;
* количество ключей для заполнения хеш-таблицы – 4 194 298;

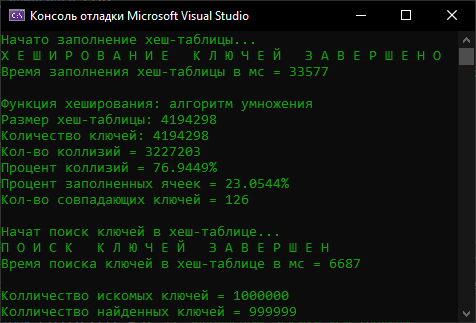
Во всех тестах количество потребляемой ОЗУ не превышало 400Мб.

Результаты алгоритма деления показаны на рис. 7.



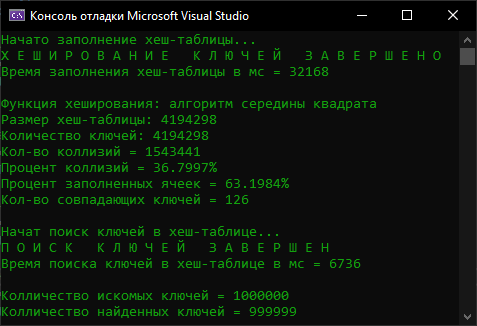
*Рис. 7. Алгоритм деления*

Результат работы хеширования с помощью алгоритма умножения представлен на рис. 8.



*Рис. 8. Алгоритм умножения*

Последний тест – алгоритм середины квадрата. Его результат отображен на рис. 9.



*Рис. 9. Алгоритм середины квадрата*

Как мы можем заметить алгоритм умножения в нашем случае является наихудшим с точки зрения возникновения коллизий (при хешировании почти 77% ключей возникает коллизия), это может говорить о том, что написанный нами алгоритм для этой функции не очень-то и хороший.

Посмотрев на результаты работы алгоритмов деления и середины квадрата, мы можем сделать вывод, что они практически идентичны с точки зрения предоставляемого функционала. Процент коллизий и время выполнения алгоритмов имеют очень близкие значения в обоих случаях.

# **ВЫВОД**

Хеширование – очень полезный и эффективный инструмент для быстрого поиска (хеш-таблица) и безопасности (хеширование паролей) данных всевозможного характера: от текста до видеоролика. Благодаря хешированию, мы можем быстро получать ответы на наши запросы в таких поисковых сервисах как Google или Yandex, не бояться, что кто-то может перехватить наши пароли от социальных сетей, когда мы подключены к незащищенной сети, убедиться, что, то или иное программное обеспечение действительно является оригинальным и в нем нет вредоносного кода. Вполне вероятно, что по мере дальнейшего развития информационных систем и информатики как науки в целом, хешированию данных найдут совершенно новое применение.

В результате написания настоящей курсовой работы нами успешно была реализована простая программа с консольным интерфейсом, которая хеширует натуральные числа, записывать их в хеш-таблицу, а после осуществляет поиск ключей в хеш-таблице. Безусловно, в нашей программе реализованы не все методы работы хеширования данных, но дальнейшее изучение этого направления дает возможность дополнить список операций хеширования и работы с хеш-таблицей этой программы в любое время.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. ГОСТ Р 34.11-2012;
2. Дроздов С.Н. Структуры и алгоритмы обработки данных: Учебное пособие – Таганрог: Издательство Южного федерального университета, 2016;
3. Кнут Д.: Искусство программирования, т.3. М.: Диалектика-Вильямс, 2000;
4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.: Алгоритмы: построение и анализ – Москва, Санкт-Петербург, Киев: Диалектика-Вильямс, 2-е издание, 2005;
5. Рябко Б. Я., Фионов А. Н.: Основы современной криптографии для специалистов в информационных технологиях – Москва: Научный мир, 2004.