[1. Хеш-таблицы](#_Toc515771271)

[1.1. Ключевые термины темы](#_Toc515771272)

[1.2. Простое представление хеш-таблиц](#_Toc515771273)

[1.3. Практическое применение хеш-таблиц](#_Toc515771274)

[1.4. Алгоритмы хеширования данных](#_Toc515771275)

[1.4.1. Таблица прямого доступа](#_Toc515771276)

[1.4.2. Метод остатков от деления](#_Toc515771277)

[1.4.3. Метод функции середины квадрата](#_Toc515771278)

[1.4.4. Метод свертки](#_Toc515771279)

[1.4.5. Открытое хеширование](#_Toc515771280)

[1.4.6. Закрытое хеширование (открытая индексация)](#_Toc515771281)

[1.5. Основные, наиболее часто используемые функции хэширования](#_Toc515771282)

[1.6. Коллизи (они же столкновения)](#_Toc515771283)

[1.6.1. Пример коллизий](#_Toc515771284)

[1.6.2. Методы разрешения коллизий](#_Toc515771285)

[1.6.3. Борьба с коллизиями](#_Toc515771286)

[1.6.3.1. Метод цепочек](#_Toc515771287)

[1.6.3.2. Открытая индексация (или закрытое хеширование)](#_Toc515771288)

[1.6.4. Переполнение таблицы и рехеширование](#_Toc515771289)

[1.6.5. Оценка качества хеш-функции](#_Toc515771290)

[1.7. Организация данных для ускорения поиска по вторичным ключам](#_Toc515771291)

[1.7.1. Инвертированные индексы](#_Toc515771292)

[1.7.2. Битовые карты](#_Toc515771293)

[1.8. Примеры](#_Toc515771294)

[1.8.1. Пример 1](#_Toc515771295)

[1.8.2. Пример 2.1](#_Toc515771296)

[1.8.3. Пример 2.2](#_Toc515771297)

[1.8.4. Пример 3.1](#_Toc515771298)

[1.8.5. Пример 3.2](#_Toc515771299)

[1.8.6. Лабораторная работа № 13. Хэш-таблицы c открытой адресацией](#_Toc515771300)

[1.8.7. Лабораторная работа № 14. Хэш-таблицы c цепочками](#_Toc515771301)

[1.9. Выводы](#_Toc515771302)

[1.10. Для самостоятельной работы. Лабораторная работа ХХ. Алгоритмы хеширования данных](#_Toc515771303)

1. Хеш-таблицы

Среди всех структур данных, имеющихся в распоряжении у замечательной науки информатики, есть одна, которой многие люди восхищаются больше, чем другими. Это – хеш-таблица (Hash Table), несомненное достижение в области компьютерных наук. Практически все современные языки программирования имеют реализации хеш-таблиц в своих библиотеках. Чаще всего мы работаем с ними в виде [**словарей**](http://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2) **(или ассоциативных массивов)**, представляющих собой **контейнеры множества пар ключ-значение**.

Так как же устроены хеш-таблицы, почему они удобны, эффективны, и зачем нам или пользоваться? Здесь приводится краткое введение в хеш-таблицы для тех, кто не знает и по каким-то причинам не хочет или не может изучить классические труды. Одна из причин, побудивших написать этот текст, так это то, как показал опыт, что тема хеш-таблиц достаточно часто встречается в вопросах на собеседованиях.

**Хеш-таблица (hash table)** — это специальная структура данных для хранения пар ключей и их значений. По сути это ассоциативный массив, в котором ключ представлен в виде хеш-функции.

01.wmf

Хеш-таблица - динамическая структура данных, которая реализует интерфейс ассоциативного массива, то есть позволяет хранить пару <Ключ, Значение> и выполнять 3 операции: **добавление** новой пары, операции **поиска**, операции **удаления** по ключу.

Чтобы понять, что такое хеш-таблица, вспомним массив. Мы можем получить доступ к элементу с помощью его ключа, причем может быть такой случай, что есть ячейки содержащие элементы и есть не содержащие.

Существуют **два основных варианта хеш-таблиц: с цепочками и открытой адресацией**. Хеш-таблица содержит некоторый [массив](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) **H**, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

**Выполнение операции в хеш-таблице начинается с вычисления** [**хеш-функции**](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) **от ключа. Получающееся хеш-значение I = hash(key) играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].**

**Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется** [**коллизией**](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8)**.** Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку). Поэтому **механизм разрешения коллизий — важная составляющая любой хеш-таблицы**.

В некоторых специальных случаях удаётся избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую [совершенную хеш-функцию](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#%D0%98%D0%B4%D0%B5%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%85%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), 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**.

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

Для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей.

При этом могут быть использованы методы поиска в упорядоченных структурах данных, например, метод половинного деления, что существенно сокращает время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание таблицы могут значительно превышать выигрыш от сокращения времени поиска. Поэтому для сокращения времени доступа к данным в таблицах используется так называемое случайное упорядочивание или хеширование. При этом данные организуются в виде таблицы при помощи хеш-функции h, используемой для вычисления адреса по значению ключа.

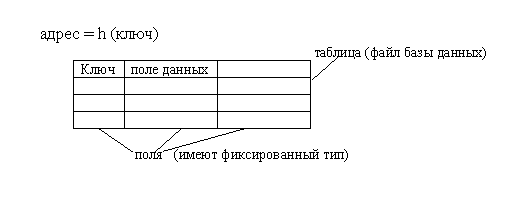


Рис.3.1. Хеш-таблица

Идеальной хеш-функцией является такая hash-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

pic3_01.gif (418 bytes)

Подобрать такую функцию можно в случае, если все возможные значения ключей заранее известны. Такая организация данных носит название совершенное хеширование. В случае заранее неопределенного множества значений ключей и ограниченной длины таблицы подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия.

Рассмотрим пример реализации алгоритма несовершенной хеш-функции. Предположим, что ключ состоит из четырех символов. При этом таблица имеет диапазон адресов от 0 до 10000.

long MyHashFunction(char key[4])

{

long f;

//ключ по коду ASCII для каждого символа

//вычисление функции по значению ключа

f = (int)key[1] - (int)key[2] + (int)key[3] - (int)key[4];

//совмещение начала области значений функции с начальным

//адресом хеш-таблицы (a=1)

f = f + 255 \* 2;

//совмещение конца области значений функции с конечным адресом

//хеш-таблицы (a=10 000)

f = (f \* 10000) / (255 \* 4); //целочисленное деление (div)

return(hash = f);

}

При заполнении таблицы возникают ситуации, когда для двух неодинаковых ключей функция вычисляет один и тот же адрес. Данный случай носит название коллизия, а такие ключи называются ключи-синонимы.

## Ключевые термины темы

**Хеширование** – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины.

**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.

**Коэффициент заполнения хеш-таблицы** – это количество хранимых элементов массива, деленное на число возможных значений хеш-функции.

**Первичные ключи** – это ключи, позволяющие однозначно идентифицировать запись.

**Вторичные ключи** – это ключи, не позволяющие однозначно идентифицировать запись в таблице.

**Пространство ключей** – это множество всех теоретически возможных значений ключей записи.

**Пространство записей** – это множество тех ячеек памяти, которые выделяются для хранения таблицы.

**Хеш-таблицы с прямой адресацией** – это хеш-таблицы, использующие инъективные хеш-функции и не нуждающиеся в механизме разрешения коллизий.

**Синонимы** – это совпадающие ключи в хеш-таблице.

**Коллизия** – это ситуация, когда разным ключам соответствует одно значение хеш-функции.

**Закрытое хеширование или Метод открытой адресации** – это технология разрешения коллизий, которая предполагает хранение записей в самой хеш-таблице.

**Открытое хеширование или Метод цепочек** – это технология разрешения коллизий, которая состоит в том, что элементы множества с равными хеш-значениями связываются в цепочку-список.

**Повторное хеширование** – это поиск местоположения для очередного элемента таблицы с учетом шага перемещения.

## Простое представление хеш-таблиц

Чтобы разобраться, что такое хеш-таблицы, представьте, что вас попросили создать библиотеку и заполнить ее книгами. Но вы не хотите заполнять шкафы в произвольном порядке.

Первое, что приходит в голову — разместить все книги в алфавитном порядке и записать все в некий справочник. В этом случае не придется искать нужную книгу по всей библиотеке, а только по справочнику.

А можно сделать еще удобнее. Если изначально отталкиваться от названия книги или имени автора, то лучше использовать некий алгоритм хеширования, который обрабатывает входящее значение и выдает номер шкафа и полки для нужной книги.

Зная этот алгоритм хэширования, вы быстро найдете нужную книгу по ее названию.

Учтите, что хеш-функция должна иметь следующие свойства:

* Всегда возвращать один и тот же адрес для одного и того же ключа;
* Не обязательно возвращает разные адреса для разных ключей;
* Использует все адресное пространство с одинаковой вероятностью;
* Быстро вычислять адрес.

**Пример:**

Каждой строке соответствует индекс равный (n=длина строки-1). Например: строка "Alex" индекс 3. Вот так можно записать строки.

* Хеш функция у нас будет: Hash\_Func(string)=Strlen(string) (то есть вычисление длины).
* Доступ к массиву мы получаем вот так: Mas[Hash\_Func(string)].

**length value**

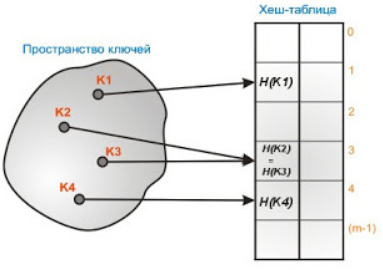
2 "joe"

3 "Alex"

4 "Pavel"

Но! вот в чем беда! Если мы хотим вставить еще 1 имя, например "Petr", то этому имени будет соответствовать ОДИН И ТОТ ЖЕ КЛЮЧ 3, который является индексом 3. Это называется **Коллизией**, когда один и тот же ключ указывает на несколько значений ключа.

Иллюстрация коллизии (2 ключа относятся к одной и той же ячейки массива, ключи k2 и k3):



Разрешение случаев с коллизией является важной задачей Computer Science. Решениями проблемы являются:

* Метод цепочек (открытое хеширование).
* Метод открытой адресации (закрытое хеширование).

## Практическое применение хеш-таблиц

Хеш-таблицы имеют очень большое практическое применение:

* **Для поиска информации** о водителе лишь по его номеру в водительском удостоверении.
* **Таблица символов компилятора**. Хеширование является методом ускорения поиска. Компилятор встречает некоторую лексему (***Лексема*** *— последовательность допустимых символов языка программирования, имеющая смысл для транслятора*) и пытается найти ее в своей базе данных или таблице символических имен. Таблица символических имен современного компилятора в MS Windows может содержать несколько тысяч или десятков тысяч лексем. Для ускорения поиска был придуман следующий прием. Компилятор, найдя лексему в тексте программы, определяет ее хеш-ключ. Наша лексема — это просто слово, состоящее из последовательности кодов символов. Для определения хеш-кода следует сложить все коды символов. Лексем с таким хеш-кодом в базе данных уже значительно меньше. Компилятор определяет номер списка с заданным кодом и перебирает этот список, а не всю базу данных.
* При **программировании шахмат** тоже используется хеширование. Основная особенность состоит в том, что хеш-индекс должен очень точно отражать позицию, и поиск должен производиться максимально быстро. У нас даже нет времени на перебор списков с одним индексом. Списков, как правило, нет вообще. Для каждого значения хеш-ключа существует только одна позиция. Если возникла коллизия и встретилась позиция с таким же ключом, то она записывается поверх прежней. Зачем при программировании шахмат и других подобных игр хеширование и запоминание позиций? Дело в том, что **при переборе** мы имеем **не дерево игры в прямом смысле, а граф**. Позиции повторяются через некоторое количество полуходов, и даже в одну и ту же позицию можно прийти различными путями. Можно воспользоваться некоторой информацией о позиции, которая уже встречалась. Самое простое и эффективное — это использовать позицию, чтобы улучшить порядок ходов. Это особенно выразительно при итеративных углублениях. **Улучшение упорядочивания ходов — основное назначение хеш-таблицы**. Перестановки или повторы позиций тоже играют свою роль. Особенно их много в конце игры.
* **Для баз данных телефонных номеров**.
* **Каталог книг**.
* **Для хранения паролей пользователей**.
* **Браузер хранит адреса посещенных страниц в хеш-таблице**.

## Алгоритмы хеширования данных

Существует ГОСУДАРСТВЕННЫЙ СТАНДАРТ РЕСПУБЛИКИ БЕЛАРУСЬ СТБ 34.101.77-2016 «Информационные технологии и безопасность. АЛГОРИТМЫ ХЭШИРОВАНИЯ»

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска значительного 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.

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 ).

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.

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.

Существует несколько типов функций хеширования, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных. Приведем обзор и анализ некоторых наиболее простых из применяемых на практике хеш-функций.

### Таблица прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице **ключ является адресом записи в таблице или может быть преобразован в адрес**, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех ячеек памяти, которые выделяются для хранения таблицы. **Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей**. В большинстве реальных задач размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то, даже ограничив длину ключа десятью символами кириллицы, получаем 3310 возможных значений ключей. Даже если ресурсы вычислительной системы и позволят выделить пространство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей.

**В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно**. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: a=h(k), где a – адрес, k – ключ.

Идеальной хеш-функцией является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

### Метод остатков от деления

**Простейшей** **хеш-функцией является деление по модулю числового значения ключа Key на размер пространства записи HashTableSize. Результат интерпретируется как адрес записи**. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к хеш-функции и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа Key на размерность массива HashTableSize, то есть:

Key % HashTableSize

Данная функция очень проста, хотя и не относится к хорошим. Вообще, можно использовать любую размерность массива, но она должна быть такой, чтобы минимизировать число коллизий. Для этого в качестве размерности лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен. **Для** **символьной строки ключом может являться остаток от деления, например, суммы** **кодов символов строки на HashTableSize**.

**На практике, метод деления – самый распространенный**.

//функция создания хеш-таблицы метод деления по модулю

int **Hash(**int **Key,** int **HashTableSize) {**

//HashTableSize

return **Key % HashTableSize;**

}

### Метод функции середины квадрата

Следующей хеш-функцией является функция середины квадрата. **Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи**.

### Метод свертки

Еще одной хеш-функцией можно назвать функцию свертки. **Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся определенные арифметические или поразрядные** **логические операции, результат которых интерпретируется как адрес**. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает **функция** **хеширования, в которой адрес записи получается в результате сложения** **кодов символов, составляющих строку-ключ**.

В качестве хеш-функции также применяют **функцию преобразования** **системы счисления. Ключ, записанный как число в некоторой** **системе счисления P, интерпретируется как число в** **системе счисления Q>P. Обычно выбирают Q=P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес**.

### Открытое хеширование

Основная идея базовой структуры при открытом (внешнем) хешировании заключается в том, что **потенциальное множество (возможно, бесконечное) разбивается на конечное** **число классов. Для В классов, пронумерованных от 0 до В-1, строится** **хеш-функция h(x) такая, что для любого элемента х исходного множества функция h(x) принимает целочисленное значение из интервала 0,1,...,В-1, соответствующее классу, которому принадлежит элемент х. Часто классы называют сегментами, поэтому будем говорить, что элемент х принадлежит сегменту h(x). Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0,1,...,В-1, содержит заголовки для B списков. Элемент х, относящийся к i -му списку – это элемент исходного множества, для которого h(x)=i**.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если можно оценить величину N и выбрать В как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N.

Пример 1. Программная реализация открытого хеширования.

#include <iostream>

#include <fstream>

using namespace std;

typedef int T; // тип элементов

typedef int hashTableIndex; // индекс в хеш-таблице

#define compEQ(a,b) (a == b)

typedef struct Node\_ {

T data;// данные, хранящиеся в вершине

struct Node\_ \*next; // следующая вершина

} Node;

Node \*\*hashTable;

int hashTableSize;

hashTableIndex myhash(T data);

Node \*insertNode(T data);

void deleteNode(T data);

Node \*findNode (T data);

int main()

{

int i, \*a, maxnum;

setlocale(LC\_ALL, "rus");

cout << "Введите количество элементов maxnum : ";

cin >> maxnum;

cout << "Введите размер хеш-таблицы HashTableSize : ";

cin >> hashTableSize;

a = new int[maxnum];

hashTable = new Node\*[hashTableSize];

for (i = 0; i < hashTableSize; i++)

hashTable[i] = NULL;

// генерация массива

for (i = 0; i < maxnum; i++)

a[i] = rand();

// заполнение хеш-таблицы элементами массива

for (i = 0; i < maxnum; i++) {

insertNode(a[i]);

}

// поиск элементов массива по хеш-таблице

for (i = maxnum-1; i >= 0; i--) {

findNode(a[i]);

}

// вывод элементов массива в файл List.txt

ofstream out("List.txt");

for (i = 0; i < maxnum; i++){

out << a[i];

if ( i < maxnum - 1 ) out << "\t";

}

out.close();

// сохранение хеш-таблицы в файл HashTable.txt

out.open("HashTable.txt");

for (i = 0; i < hashTableSize; i++){

out << i << " : ";

Node \*Temp = hashTable[i];

while ( Temp ){

out << Temp->data << " -> ";

Temp = Temp->next;

}

out << endl;

}

out.close();

// очистка хеш-таблицы

for (i = maxnum-1; i >= 0; i--) {

deleteNode(a[i]);

}

system("pause");

return 0;

}

// хеш-функция размещения вершины

hashTableIndex myhash(T data) {

return (data % hashTableSize);

}

// функция поиска местоположения и вставки вершины в таблицу

Node \*insertNode(T data) {

Node \*p, \*p0;

hashTableIndex bucket;

// вставка вершины в начало списка

bucket = myhash(data);

if ((p = new Node) == 0) {

fprintf (stderr, "Нехватка памяти (insertNode)\n");

exit(1);

}

p0 = hashTable[bucket];

hashTable[bucket] = p;

p->next = p0;

p->data = data;

return p;

}

//функция удаления вершины из таблицы

void deleteNode(T data) {

Node \*p0, \*p;

hashTableIndex bucket;

p0 = 0;

bucket = myhash(data);

p = hashTable[bucket];

while (p && !compEQ(p->data, data)) {

p0 = p;

p = p->next;

}

if (!p) return;

if (p0)

p0->next = p->next;

else

hashTable[bucket] = p->next;

free (p);

}

// функция поиска вершины со значением data

Node \*findNode (T data) {

Node \*p;

p = hashTable[myhash(data)];

while (p && !compEQ(p->data, data))

p = p->next;

return p;

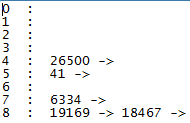
}



**List.txt (элементы массива)**



**HashTable. txt (хеш-таблица)**



### Закрытое хеширование (открытая индексация)

При закрытом (внутреннем) хешировании **в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом** **хешировании применяется методика повторного хеширования. Если осуществляется попытка поместить элемент х в сегмент с номером h(х), который уже занят другим элементом (коллизия), то в соответствии с методикой повторного** **хеширования выбирается последовательность других номеров сегментов h1(х), h2(х), .., куда можно поместить элемент х. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент х добавить нельзя**.

При поиске элемента х необходимо просмотреть все местоположения h(x),h1(х),h2(х),..., пока не будет найден х или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. Пусть h3(х) – первый пустой сегмент. В такой ситуации невозможно нахождение элемента х в сегментах h4(х), h5(х) и далее, так как при вставке элемент х вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(х). Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента х, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента х. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем, например, DEL. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы DEL и NULL – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой DEL, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Повторное хеширование

**Существует несколько методов повторного** **хеширования, то есть определения местоположений h(x), h1(х), h2(х),...:**

* линейное опробование;
* квадратичное опробование;
* двойное хеширование.

**Линейное опробование** сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом:

адрес=h(x)+ci,

где i – номер попытки разрешить коллизию;

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит последовательный перебор всех сегментов после текущего. **Квадратичное опробование** отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

адрес=h(x)+ci+di2,

где i – номер попытки разрешить коллизию,

c и d – константы.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется **двойным хешированием**, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:

адрес=h(x)+ih2(x).

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией. С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

Однако в случае многократного превышения адресного пространства и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные сегменты. Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает вероятность нахождения свободных сегментов.

В случае применения схемы закрытого хеширования скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования (опробования) для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты. Например, **методика линейного опробования для разрешения** **коллизий – не самый лучший выбор**.

Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операций.

Пример 2. Программная реализация закрытого хеширования.

#include <iostream>

#include <fstream>

using namespace std;

typedef int T; // тип элементов

typedef int hashTableIndex;// индекс в хеш-таблице

int hashTableSize;

T \*hashTable;

bool \*used;

hashTableIndex myhash(T data);

void insertData(T data);

void deleteData(T data);

bool findData (T data);

int dist (hashTableIndex a,hashTableIndex b);

int main(){

int i, \*a, maxnum;

setlocale(LC\_ALL, "rus");

cout << "Введите количество элементов maxnum : ";

cin >> maxnum;

cout << "Введите размер хеш-таблицы hashTableSize : ";

cin >> hashTableSize;

a = new int[maxnum];

hashTable = new T[hashTableSize];

used = new bool[hashTableSize];

for (i = 0; i < hashTableSize; i++){

hashTable[i] = 0;

used[i] = false;

}

// генерация массива

for (i = 0; i < maxnum; i++)

a[i] = rand();

// заполнение хеш-таблицы элементами массива

for (i = 0; i < maxnum; i++)

insertData(a[i]);

// поиск элементов массива по хеш-таблице

for (i = maxnum-1; i >= 0; i--)

findData(a[i]);

// вывод элементов массива в файл List.txt

ofstream out("List.txt");

for (i = 0; i < maxnum; i++){

out << a[i];

if ( i < maxnum - 1 ) out << "\t";

}

out.close();

// сохранение хеш-таблицы в файл HashTable.txt

out.open("HashTable.txt");

for (i = 0; i < hashTableSize; i++){

out << i << " : " << used[i] << " : " << hashTable[i] << endl;

}

out.close();

// очистка хеш-таблицы

for (i = maxnum-1; i >= 0; i--) {

deleteData(a[i]);

}

system("pause");

return 0;

}

// хеш-функция размещения величины

hashTableIndex myhash(T data) {

return (data % hashTableSize);

}

// функция поиска местоположения и вставки величины в таблицу

void insertData(T data) {

hashTableIndex bucket;

bucket = myhash(data);

while ( used[bucket] && hashTable[bucket] != data)

bucket = (bucket + 1) % hashTableSize;

if ( !used[bucket] ) {

used[bucket] = true;

hashTable[bucket] = data;

}

}

// функция поиска величины, равной data

bool findData (T data) {

hashTableIndex bucket;

bucket = myhash(data);

while ( used[bucket] && hashTable[bucket] != data )

bucket = (bucket + 1) % hashTableSize;

return used[bucket] && hashTable[bucket] == data;

}

//функция удаления величины из таблицы

void deleteData(T data){

int bucket, gap;

bucket = myhash(data);

while ( used[bucket] && hashTable[bucket] != data )

bucket = (bucket + 1) % hashTableSize;

if ( used[bucket] && hashTable[bucket] == data ){

used[bucket] = false;

gap = bucket;

bucket = (bucket + 1) % hashTableSize;

while ( used[bucket] ){

if ( bucket == myhash(hashTable[bucket]) )

bucket = (bucket + 1) % hashTableSize;

else if ( dist(myhash(hashTable[bucket]),bucket) < dist(gap,bucket) )

bucket = (bucket + 1) % hashTableSize;

else {

used[gap] = true;

hashTable[gap] = hashTable[bucket];

used[bucket] = false;

gap = bucket;

bucket++;

}

}

}

}

// функция вычисления расстояние от a до b (по часовой стрелке, слева направо)

int dist (hashTableIndex a,hashTableIndex b){

return (b - a + hashTableSize) % hashTableSize;

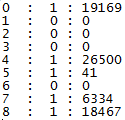
}



**List.txt**



**HashTable.txt (хеш-таблица)**



До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Такие ключи называются **первичными**. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются **вторичными** ключами. Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные.

Идея хеширования впервые была высказана Г.П. Ланом при создании внутреннего меморандума IBM в январе 1953 г. с предложением использовать для разрешения коллизий метод цепочек. Примерно в это же время другой сотрудник IBM, Жини Амдал, высказала идею использования открытой линейной адресации. В открытой печати хеширование впервые было описано Арнольдом Думи (1956 год), 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 (1957 год), который разработал и описал метод линейной открытой адресации.

### Примеры хеш-функций

Хеш-функция – функция, преобразовывающая входную последовательность данных произвольного размера в выходную последовательность фиксированного размера. Процесс преобразования данных называется хешированием. Результат хеширования – хеш-код (хеш-сумма, хеш).

**При решении класса практических задач выбирается такая хеш-функция, которая является наиболее оптимальной именно для данного класса. В общем случае следует использовать «хорошую» функцию.** Когда хеш-функцию называют «хорошей», то подразумевают под этим, что она:

* вычисляется достаточно быстро;
* сводит к минимуму число коллизий.

Предотвратить коллизии могут далеко не все хеш-функции, но «хорошие» способны минимизировать вероятность их появления. При определенных обстоятельствах (известна некоторая информация о ключах), можно найти идеальную хеш-функцию, т. е. такую, которая полностью исключает возможность появления коллизий.

Использовать хеш-функцию или нет, зависит от того насколько целесообразно применение ее свойств, а также свойств алгоритма, по которому она может быть реализована на ЯП. В одних ситуациях наиболее важна высокая скорость работы, в других равномерное распределение хеш-кодов и т. п. Далее будут рассмотрены два наиболее известных метода хеширования.

Метод деления

Пусть k – ключ (тот, что необходимо хешировать), а N – максимально возможное число хеш-кодов. Тогда метод хеширования посредством деления будет заключаться во взятии остатка от деления k на N:

h(k)=k mod N, где mod – операция взятия остатка от деления.

Например, на вход подаются следующие ключи **k**:

**3, 6, 7, 15, 32, 43, 99, 100, 133, 158**

Определим **N** равным 10, из чего последует, что возможные значения хешей лежат в диапазоне 0…9. Используя данную функцию, получим следующие значения хеш-кодов:

|  |  |
| --- | --- |
| Ключ | Хеш-код |
| 3 | h(3)=3 |
| 6 | h(6)=6 |
| 7 | h(7)=7 |
| 15 | h(15)=5 |
| 32 | h(32)=2 |
| 43 | h(42)=2 |
| 99 | h(99)=9 |
| 100 | h(100)=0 |
| 133 | h(133)=3 |
| 158 | h(158)=8 |

На C++ программу, выполняющую хеширование методом деления можно записать так:

#include <iostream>

using namespace std;

int HashFunction(int k)

{

return (k%10);

}

void main()

{

setlocale(LC\_ALL, "Rus");

int key;

cout<<"Ключ > "; cin>>key;

cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;

system("pause>>void");

}



Во избежание большого числа коллизий рекомендуется выбирать N простым числом, и не рекомендуется степенью с основанием 2 и показателем m (2m). Вообще, по возможности, следует выбирать N, опираясь на значения входящих ключей. Так, например если все или большинство k=10m (m – натуральное число), то неудачным выбором будет N=10\*m и N=10m.

Метод умножения.

Получить из исходной последовательности ключей последовательность хеш-кодов, используя метод умножения (мультипликативный метод), значит воспользоваться хеш-функцией:

h(k)=[N\*({k\*A})]

Здесь **A** – рациональное число (**Рациональное число** (лат. ratio — отношение, деление, дробь) — **число**, которое можно представить обыкновенной дробью , числитель — целое **число**, а знаменатель — натуральное **число**, к примеру 2/3), по модулю меньшее единицы (0<A<1), а **k** и **N** обозначают то же, что и в предыдущем методе: ключ и размер хеш-таблицы. Также правая часть функции содержит **три пары скобок**:

( ) – скобки приоритета;

[ ] – скобки взятия целой части;

{ } – скобки взятия дробной части.

Аргумент хеш-функции k (k≥0) в результате даст значение хеш-кода h(k)=x, лежащие в диапазоне 0…N-1. **Для работы с отрицательными числами можно число x взять по модулю**.

От выбора A и N зависит то, насколько оптимальным окажется хеширование умножением на определенной последовательности. Не имея сведений о входящих ключах, в качестве **N** следует выбрать одну из степеней двойки, т. к. **умножение на 2m равносильно сдвигу на m разрядов**, что компьютером производиться быстрее. Неплохим значением для **A** (в общем случае) будет **(√5-1)/2≈0,6180339887**. Оно основано на свойствах **золотого сечения**:

**Золотое сечение – такое деление величины на две части, при котором отношение большей части к меньшей равно отношению всей величины к ее большей части**.

Отношение большей части к меньшей, выраженное квадратичной иррациональностью:

**φ=(√5+1)/2≈1,6180339887**

Для мультипликативной хеш-функции было приведено обратное отношение:

**1/φ=(√5-1)/2≈0,6180339887**

При таком A, хеш-коды распределяться достаточно равномерно, но многое зависит от начальных значений ключей.

Для демонстрации работы мультипликативного метода, положим **N=13, A=0,618033**. В качестве ключей возьмем числа: **25, 44 и 97**. Подставим их в функцию:

h(k)=[13\*({25\*0,618033})]=[13\*{15,450825}]=[13\*0,450825]=[5,860725]=5

h(k)=[13\*({44\*0,618033})]=[13\*{27,193452}]=[13\*0,193452]=[2,514876]=2

h(k)=[13\*({97\*0,618033})]=[13\*{59,949201}]=[13\*0,949201]=[12,339613]=12

Реализация метода на C++ с использованием оговоренных N и A:

#include <iostream>

#include <math.h> /\* fmod \*/

using namespace std;

int HashFunction(int k)

{

int N=13; double A=0.618033;

int h=N\*fmod(k\*A, 1);

//fmod of 5.3 / 2 is 1.300000

//fmod of 18.5 / 4.2 is 1.700000

return h;

}

void main()

{

setlocale(LC\_ALL, "Rus");

int key;

cout<<"Ключ > "; cin>>key;

cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;

system("pause>>void");

}



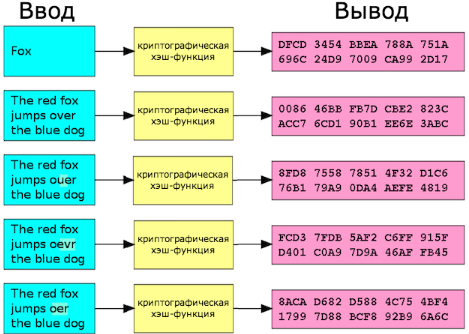




http://mf.grsu.by/UchProc/livak/b\_protect/zok\_7.htm

## Основные, наиболее часто используемые функции хэширования

Функции хэширования играют главную роль в современной криптографии.

.

|  |  |
| --- | --- |
| Красная лиса перепрыгивает через синюю собаку | |
| Fox | DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17 |
| The red fox jumps **over** the blue dog | 0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC |
| The red fox jumps **ouer** the blue dog | 8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819 |
| The red fox jumps **oevr** the blue dog | FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45 |
| The red fox jumps **oer** the blue dog | 8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C |

В настоящее время механизм хэш-функций используется на практике очень широко.

Кроме шифрования данных с помощью хэш-функций реализуют

* **Проверку целостности данных** (обнаружение изменений)  
  Идея заключается в сохранении хэш-кода и последующем сравнении с эталоном повторно вычисленного для тех же данных хэш-значения.  
  Очевидно, что неравенство сравниваемых величин означает нарушение целостности.
* **Системы аутентификации**  
  Используют хэширование паролей.
* **Создание и проверку ЭЦП**  
  Механизм хэш-функций применяется для уменьшения времени, необходимого для генерации и проверки подписи, а также для сокращения ее длины.

При построении функций хэширования требуется, чтобы вычислительно трудным являлось решение следующих криптоаналитических задач:

* по заданному y = h( x) определить x (односторонняя функция h);
* для заданного x найти другое x´ , такое, что h(x)= h(x´) (свободная от коллизий функция h);
* найти пару x, x´ (x ≠ x´) , такую, что h(x)= h(x´) (строго свободная от коллизий функция h).

**Замечание**

Значение хэш-функции также называют

* Хэш-код
* Функция (значение) свертки
* Профиль сообщения
* Дайджест сообщения
* Криптографическая контрольная сумма
* Цифровой отпечаток
* Код аутентичности сообщения
* Код обнаружения манипуляций

Характеристики, которыми должна обладать хеш-функция:

* должна уметь выполнять преобразования данных произвольной длины в фиксированную;
* должна иметь открытый алгоритм, чтобы можно было исследовать её криптостойкость;
* должна быть односторонней, то есть не должно быть математической возможности по результату определить исходные данные;
* должна «сопротивляться» коллизиям, то есть не должна выдавать одинаковых значений при разных входных данных;
* не должна требовать больших вычислительных ресурсов;
* при малейшем изменении входных данных результат должен существенно изменяться.

Перечислим и кратко охарактеризуем **наиболее часто используемые функции хэширования**, применяемые для ЭЦП.

* **SHA** - Secure Hash Algorithm (1992)  
  160-разрядный хэш-код (дайджест). НЕ устойчив к коллизиям.  
  512-битовые блоки.
* **SHA-1** - Secure Hash Algorithm 1 (1995)  
  Модификация SHA. Исправлены недостатки. Решает проблему коллизий.
* **MAC** - Message Authentication Code - код аутентификации (проверки подлинности) сообщения.  
  Это зависящая от ключа однонаправленная хэш-функция. Простейшим способом преобразования хэш-функции в MAC является шифрование хэш-значения симметричным алгоритмом.
* **HMAC**  
  Один из вариантов добавления секретного ключа в уже существующий алгоритм хэширования. Функция хэширования в этом алгоритме интерпретируется как «черный ящик», то есть функция хэширования реализована как отдельный модуль и ее можно менять.  
  Алгоритм HMAC [представлен в документе RFC 2104] принят как обязательный в протоколе IPSec и используется в ряде других протоколов Internet (TLS, SET и другие)  
  Широко используются на практике также функции, разработанные Роном Ривестом:
* **MD2** - Message Digest #2  
  Низкоскоростной, но очень надежный алгоритм, создающий 128-разрядные дайджесты данных любого объема.
* **MD4** - Message Digest #4 (1990)  
  Более скоростной, но менее надежный алгоритм, создающий 128-разрядные дайджесты данных любого объема. 512-битовые блоки. Есть дефекты.
* **MD5** - Message Digest #5 (1992)  
  Версия MD4 с повышенной надежностью, преимущества также и в скорости. 128-разрядные дайджесты данных любого объема.  
  Неустойчив к коллизиям! Не используется для долговременных ЭЦП.

Обратим внимание, что алгоритмы SHA надежнее алгоритмов MDx, так как вырабатывают более длинный хэш-код (160 бит против 128 бит), что снижает вероятность того, что разные входные последовательности будут преобразованы в одно значение хэш-кода.

Современные технологии распределенных вычислений и многопроцессорные компьютеры демонстрируют недостаточную защищенность 128-битовых хэш-кодов. «Кроме того, были разработаны сценарии целого ряда атак, демонстрирующих уязвимость MD5 в отношении современных методов криптоанализа» [Шнайер].

Однако, до сих пор не разработаны атаки, демонстрирующие уязвимость SHA в отношении современных методов криптоанализа; «сведения об успешных криптографических атаках на алгоритм SHA отсутствуют» [Шнайер].

Заметим также, что в российском стандарте ГОСТ Р 34.11-94 (в основе схемы Эль-Гамаля и Шнорра) длина хэш-кода равна 256 битам.

## Коллизи (они же столкновения)

### Пример коллизий

На самом деле ничего нового во всем этом нет. В действительности данный эффект достигается за счет методов быстрого поиска коллизий для хеш функции разработанных еще в 2004-2006 годах. Если кто не знает, **коллизия это два разных набора данных, имеющих одно и тоже хеш-значение**. Так вот, в 2004 году группа китайских исследователей разработала алгоритм, основанный на дифференциальном криптоанализе, позволяющий за относительно небольшое время находить два различных случайных блока данных, размером по 128 байт каждый, имеющих одну и ту же md5 сумму.

**Замечание**

Что такое **md5** сумма и зачем она нужна? Есть файл, который мы скачали из сети, например, файл образа **Ubuntu**. Как нам удостовериться, что закачанный файл (который находится теперь на нашем компьютере) совпадает с файлом, который лежит на сервере, откуда мы скачали файл? Другими словами как понять, что файл не был изменен при закачке. Для этого и используется механизм вычисления контрольной суммы. Автор или хозяин файла вычисляет при помощи специальной утилиты эту самую **md5** сумму, которая представляет собой, строку вида **2413b8c71bea3e0bccd2d51c630ff190**. Затем выкладывает ее рядом с файлом в текстовом виде или в виде файла с расширением **md5**. Скачав файл, мы можем также вычислить **md5** сумму и если числа совпадут, то файл является точной копией и не изменился во время закачки. Данную сумму каждого файла можно просчитать с помощью программы **MD5\_File Checker** ( не требует установки в системе) . Совпадение контрольных сумм скачиваемого и скачанного образа, дает гарантию, что при скачивании файл не был поврежден. Кроме данной суммы существуют еще CRC32 и SHA-1-суммы, подсчитывающие хэш другими способами.

И хотя алгоритм этот в свое время произвел эффект взорвавшейся бомбы быстродействие его оставляло желать лучшего. Но уже в 2006 году чешский криптограф Властимил Клима предложил для поиска коллизий новый метод, позволяющий найти разную пару случайных **128 байтных блоков с одной md5 суммой** на персональном компьютере меньше чем за минуту.

Вы скажите, но что нам даст обладание такой парой сообщений, мало того что они короткие(всего 128 байт), так еще, в добавок, и случайные, т.е. метод не позволяет для заданного сообщения подобрать другое, с идентичным хешем. Однако **это открывает огромный простор для различного рода атак на выполняемые файлы**. И **виной тому** служит следующая особенность работы любой хеш функции: Хеш функция по своей природе итеративна (повторяется). Это означает, что **при подсчете хеша сообщение разбивается на блоки, к каждому блоку применяется функция сжатия, зависящая от некоторой переменной, называемой вектор инициализации**. Результат этой функции будет являться вектором инициализации для следующего блока. Результат функции после работы с последним блоком и будет окончательным хеш значением нашего сообщения.

Схематично это можно представить следующим образом:

si+1 = f(si, Mi), где si вектор инициализации для i-го блока.

Метод Властимила Клима позволяет для любого заданного значения si подобрать два 128-байтных блока M,M` и N,N` таких, что f(f(s, M), M') = f(f(s, N), N').

Таким образом, с помощью данной методики можно сконструировать два файла с одинаковой md5 суммой, но имеющих различные 128 байт в середине.

M0, M1, ..., Mi-1, Mi, Mi+1, Mi+2, ..., Mn,

M0, M1, ..., Mi-1, Ni, Ni+1, Mi+2, ..., Mn.

Обратите внимание что **хеши обоих этих файлов совпадут, т.к. различающиеся блоки Mi, Mi+1 и Ni, Ni+1** вернут в качестве si+2 одно и тоже значение, т.к. f(f(s, Mi), Mi+1) = f(f(s, Ni), Ni+1), а поскольку все последующие данные идентичны то последующие значения функции сжатия для обоих файлов будут совпадать.

**Что это нам дает**

Теперь перейдем от вещей абстрактных и отдаленных к вопросу практическому. Предположим, что у нас есть исполняемый файл

M0, M1, X, X, …, Mn.

Но его основе мы можем создать два разных файла

**M0, M1, N1, N1, …, Mn** и

**M0, M1, N2, N1,…, Mn**

(просто меняем блоки X на N1 и N2). Если блоки N1 и N2 – это коллизии то хеш-сумма этих файлов будет совпадать. Теперь представим, что этот исполняемый файл имеет следующую структуру:

if (X == X) then

{ good\_program }

Else

{ evil\_program }

Вот собственно и весь секрет данного фокуса.

#include<iostream>

#include<string>

usingnamespace **std;**

//переменные str1 и str2 в данном примере являются теми самыми элементами X.

staticchar **\*str1=**"qwertyuioplkjhgfdaszxcvbnmkjhgfdsaqwertyuikjh"**\**

"gbvfdsazxdcvgbhnjikmjhbgfvcdsazxdcfrewqikolkjnhgfqwertyuioplkjh"**\**

"gfdaszxcvbnmkjhgfdsaqwertyuikjhgbvfdsazxdcvgbhnjikmjhbgfvcdsa"**\**

"zxdcfrewqikolkjnhgfq123";

staticchar **\*str2=**"qaswderftgyhujikolpmnbvcxzasxdcfvgbhnjmkijuy"**\**

"gtfdeswaqscfvgyjqaswderftgyhujikolpmnbvcxzasxdcfvgbhnjmkijuyg"**\**

"tfdeswaqscfvgyjqaswderftgyhujikolpmnbvcxzasxdcfvgbhnjmkijuygt"**\**

"fdeswaqscfvgyjqwertyuikja2";

int **good()**

{

int **a;**

**std::cout<<**"Good, nice programme!";

**std::cin>>a;**

return **0;**

}

int **bed()**

{

int **a;**

for(int **i=0; i<1000; i++)**

**{**

**std::cout<<**"Evil, evil code!"; // Зло, злой код

**}**

**std::cin>>a;**

return **0;**

}

int **main()**

{

//строки s1 и s2 содержат только блоки с коллизиями без лишних элементов

**string s1=str1;**

**string s2=str2;**

**s1.erase(0,56);**// Erases part of the [string](http://www.cplusplus.com/string), reducing its [length](http://www.cplusplus.com/string::length)

**s1.erase(128,8);**

**s2.erase(0,64);**

if **(s1==s2) {**

return **good();**

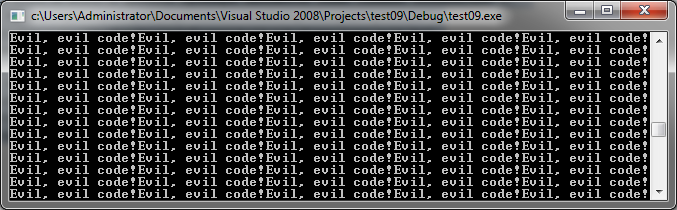
**}** else **{**

return **bed();**

**}**

return **0;**

}



**Для презентации проверка (коллизий нет)**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<conio.h>

#include<iostream>

#define **HASH\_MUL 31**

#define **HASH\_SIZE 128**

unsignedint **hash(**char **\*s,** int **inti[])**

{

unsignedint **h = 0;**

char **\*p;**

int **j=0;**

for **(p = s; \*p !=** '\0'; p++, j++){

//h = h \* HASH\_MUL + (unsigned int)\*p;

**h = h \* HASH\_MUL + inti[j];**

**printf(**"\*p=%c\t%d\th=%12d\thash=%3d\n", \*p, inti[j], h, h % HASH\_SIZE);

**}**

return **h % HASH\_SIZE;** //

}

int **main()**

{

**setlocale(LC\_ALL,** "rus");

//Волк 194 238 235 234

//Жираф 198 232 240 224 244

staticchar **\*str1=**"Волк";

staticchar **\*str2=**"Жираф";

staticint **itr1[4]={194, 238, 235, 234};**

staticint **itr2[5]={198, 232, 240, 224, 244};**

unsignedint **h1 = hash(str1, itr1);**

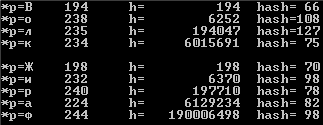
**printf(**"\n");

unsignedint **h2 = hash(str2, itr2);**

**\_getch();**

return **0;**

}



Но в большинстве случаев приходится бороться с коллизиями. Обычно применяются методы цепочек и открытой индексации.

Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными. Тем не менее, существуют способы преодоления возникающих сложностей:

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

### Методы разрешения коллизий

Для разрешения коллизий используются различные методы, которые в основном сводятся к методам цепочек и открытой адресации.

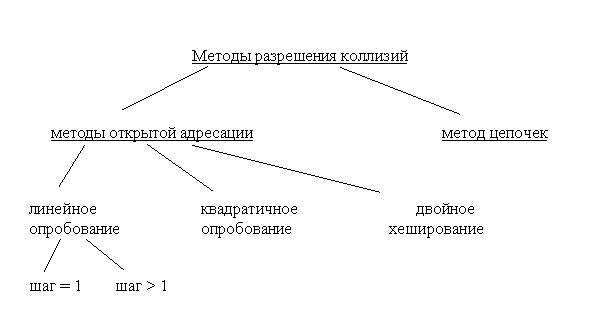


Рис.3.2. Разновидности методов разрешение коллизий

**Методом цепочек** называется метод, в котором для разрешения коллизий во все записи вводятся указатели, используемые для организации списков √ цепочек переполнения. В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент.

**Поиск** в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом.

Процедура **удаления** из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.

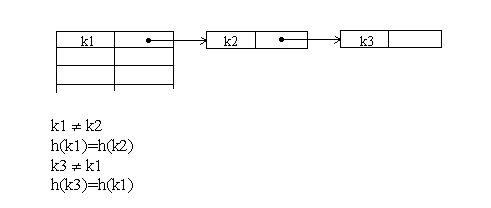


Рис.3.3. Разрешение коллизий при добавлении элементов методом цепочек

**Метод открытой адресации** состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.

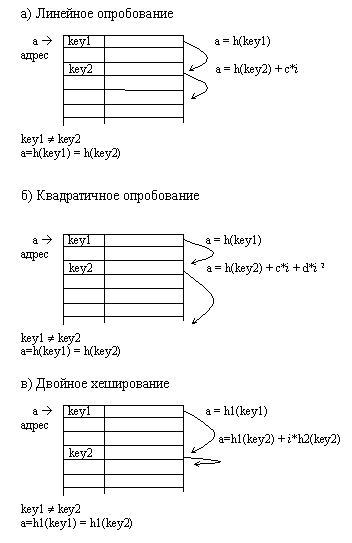


Рис.3.4. Разрешение коллизий при добавлении элементов методами открытой адресации.

**Линейное опробование** сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом

a=h(key) + c\*i ,

где i - номер попытки разрешить коллизию. При шаге равном единице происходит последовательный перебор всех элементов после текущего.

**Квадратичное опробование** отличается от линейного тем, что шаг перебора элементов не линейно зависит от номера попытки найти свободный элемент

a = h(key2) + c\*i + d\*i2

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов.

Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется **двойным хешированием**, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций

a=h1(key) + i\*h2(key).

Опишем **алгоритмы вставки и поиска** для метода линейное опробование.

**Вставка**

1. i = 0
2. a = h(key) + i\*c
3. Если t(a) = свободно, то t(a) = key, записать элемент, **стоп элемент добавлен**
4. i = i + 1, перейти к шагу 2

**Поиск**

1. i = 0
2. a = h(key) + i\*c
3. Если t(a) = key, то **стоп элемент найден**
4. Если t(a) = свободно, то **стоп элемент не найден**
5. i = i + 1, перейти к шагу 2

Аналогичным образом можно было бы сформулировать **алгоритмы добавления и поиска элементов для любой схемы открытой адресации**. Отличия будут только в выражении, используемом для вычисления адреса (шаг 2). С процедурой **удаления** дело обстоит не так просто, так как она в данном случае не будет являться обратной процедуре вставки.

Дело в том, что элементы таблицы находятся в двух состояниях: свободно и занято. Если удалить элемент, переведя его в состояние свободно, то после такого удаления алгоритм поиска будет работать некорректно. Предположим, что ключ удаляемого элемента имеет в таблице ключи синонимы. В том случае, если за удаляемым элементом в результате разрешения коллизий были размещены элементы с другими ключами, то поиск этих элементов после удаления всегда будет давать отрицательный результат, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно.

**Скорректировать** эту ситуацию можно различными способами. **Самый простой** из них заключается в том, чтобы **производить поиск элемента не до первого свободного места, а до конца таблицы**. Однако такая модификация алгоритма сведет на нет весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования.

**Другой способ** сводится к тому, чтобы **проследить адреса всех ключей-синонимов для ключа удаляемого элемента и при необходимости переразместить соответствующие записи в таблице**. Скорость поиска после такой операции не уменьшится, но затраты времени на само переразмещение элементов могут оказаться очень значительными.

Существует **подход, который свободен от перечисленных недостатков**. Его суть состоит в том, что для элементов хеш-таблицы **добавляется состояние удалено**. Данное состояние в процессе поиска интерпретируется, как занято, а в процессе записи как свободно.

Сформулируем алгоритмы вставки поиска и удаления для хеш-таблицы, имеющей три состояния элементов.

**Вставка**

1. i = 0
2. a = h(key) + i\*c
3. Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**
4. i = i + 1, перейти к шагу 2

**Удаление**

1. i = 0
2. a = h(key) + i\*c
3. Если t(a) = key, то t(a) = удалено, **стоп элемент удален**
4. Если t(a) = свободно, то **стоп элемент не найден**
5. i = i + 1, перейти к шагу 2

**Поиск**

1. i = 0
2. a = h(key) + i\*c
3. Если t(a) = key, то **стоп элемент найден**
4. Если t(a) = свободно, то **стоп элемент не найден**
5. i = i + 1, перейти к шагу 2

Алгоритм поиска для хеш-таблицы, имеющей три состояния, практически не отличается от алгоритма поиска без учета удалений. Разница заключается в том, что при организации самой таблицы необходимо отмечать свободные и удаленные элементы. Это можно сделать, зарезервировав два значения ключевого поля. Другой вариант реализации может предусматривать введение дополнительного поля, в котором фиксируется состояние элемента. Длина такого поля может составлять всего два бита, что вполне достаточно для фиксации одного из трех состояний. На языке TurboPascal данное поле удобно описать типом Byte или Char.

### Борьба с коллизиями

В идеальном случае, когда заранее известны все пары ключ-значение, достаточно легко реализовать идеальную хеш-таблицу, в которой время поиска будет постоянным (используется идеальная хеш-функция, которая определяет положения в таблице по целым значениям и без столкновений).

Метод цепочек

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. 38.1](https://www.intuit.ru/studies/courses/648/504/lecture/11467?page=1#image.38.1) демонстрируется реализация метода цепочек при разрешении коллизий. На ключ 002 претендуют два значения, которые организуются в линейный список.

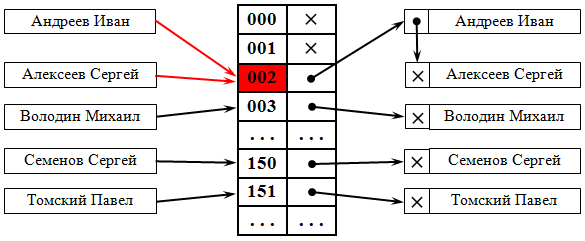


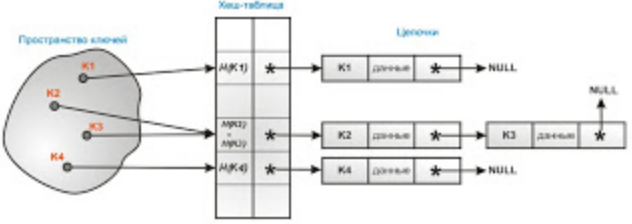
Рис. 38.1. Разрешение коллизий при помощи цепочек

**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**.

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.

Идея метода цепочек состоит в том, что все элементы множества, которые относятся к одному и тому же ключу входят в связный список.



Итак! В этом примере, в i позициях содержится указатель на голову списка. Т.е. в массиве с ячейками (1,3,4); в остальных других хранится null (0,2,5);

Возвращаясь к нашему примеру с именами, переделаем его в соответствии с новым методом:

**length value**

2 \*head1 ["Joe"->"Rex"->Null]

3 \*head2 ["Alex"->"Petr"->"Oleg"->Null]

4 \*head3 ["Pavel"->Null]

В этом примере 0,1,5 элементы указатели равны Null

Процедура вставки происходит так: Вычисляется хеш-функция от строки и вставляется в голову списка (push\_front) на определенный индекс равный значению функции.

Вставка происходит за O(1).

Поиск или удаление элемента зависит от длины списка, худший случай: O(n) - когда, все элементы хешируются в одну и ту же ячейку. Если функция распределяем n ключей по m ячейкам таблицы равномерно, то в каждом списке будет содержаться порядка k=n/m ключей. Это число называется коэффициентом заполнения хеш-таблицы. В этом случае время поиска: O(1+k), это показано в книжке Кормана, могу скинуть ссылку на нее, если интересно.

Добавлю, что в случае, **когда коэффициент заполнения будет слишком велик**, надо будет увеличить размер массива и, возможно, перестроить таблицу.

Этот метод часто называют **открытым хешированием**. Его суть проста — элементы с одинаковым хешем попадают в одну ячейку в виде [связного списка](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA).

02.wmf

То есть, если ячейка с хешем уже занята, но новый ключ отличается от уже имеющегося, то новый элемент вставляется в список в виде пары ключ-значение.

Если выбран метод цепочек, то вставка нового элемента происходит за O(1), а время поиска зависит от длины списка и в худшем случае равно O(n). Если количество ключей n, а распределяем по m-ячейкам, то соотношение n/m будет коэффициентом заполнения.

В C++ метод цепочек реализуется так:

//Проверка ячейки и создание списка

class LinkedHashEntry {

private:

int key;

int value;

LinkedHashEntry \*next;

public:

LinkedHashEntry(int key, int value) {

this->key = key;

this->value = value;

this->next = NULL;

}

int getKey() {

return key;

}

int getValue() {

return value;

}

void setValue(int value) {

this->value = value;

}

LinkedHashEntry \*getNext() {

return next;

}

void setNext(LinkedHashEntry \*next) {

this->next = next;

}

};

Открытая индексация (или закрытое хеширование)

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 [рис. 38.2](https://www.intuit.ru/studies/courses/648/504/lecture/11467?page=1#image.38.2) разрешение коллизий осуществляется методом открытой адресации. Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще незанятое) место в таблице.

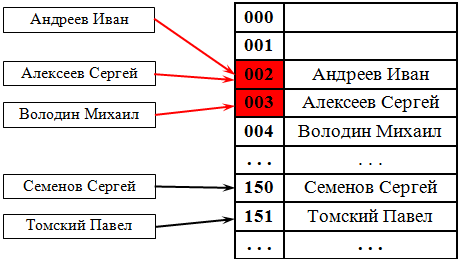


Рис. 38.2. Разрешение коллизий при помощи открытой адресации

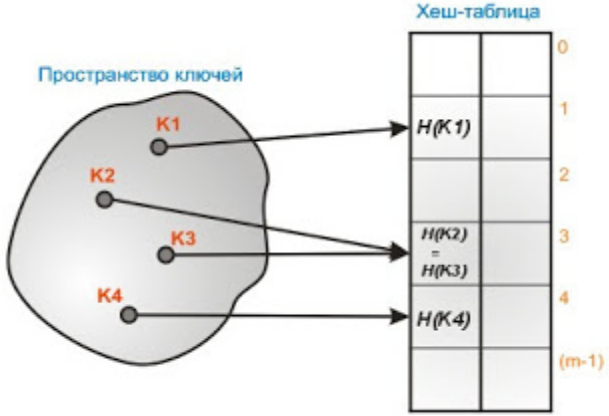
При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. **Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений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

Идея метода открытой адресации состоит в том, что нету никаких списков, а все элементы хранятся в хеш-таблице. Ячейка хранит либо элемент либо NULL (по желанию ваш граничный элемент, в C# если использовать тип object, то null).

Для того, чтобы вставить ячейку, можно двигаться по списку пока не найдем пустую ячейку и вставить туда элемент. Этот метод решения коллизий называется **линейное хеширование**.

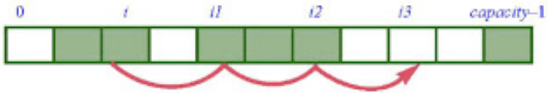


Линейное хеширование достаточно просто реализуется, однако с ним связана существенная проблема – кластеризация. Это явление создания длинных последовательностей занятых ячеек, которое увеличивает среднее время поиска в таблице. Для снижения эффекта кластеризации используется другая стратегия разрешения коллизий – двойное хеширование. Основная идея заключается в том, что для определения шага смещения исследований при коллизии в ячейке используется другая хеш-функция, вместо линейного смещения на одну позицию.

Общий прием состоит в следующем: если хеш-функция вырабатывает позицию для первого кандидата **i = Hash\_Func(key) Mod (capacity)**, то последующие позиции определяются как **i + increment, i +2 \* increment, i +3 \* increment** и так далее, все по модулю **capacity**. Величина increment вычисляется как:

Hash\_Func(key) Mod (capacity -1)

Пример:



i,i1,i2 ячейки заняты, i=2, i1=4, i2=6, i3=8, capacity=11

**Последующие index=i+increment (increment = 2 Mod 10 = 2)**

Второй распространенный метод — открытая индексация. Это значит, что пары ключ-значение хранятся непосредственно в хеш-таблице. А алгоритм вставки проверяет ячейки в некотором порядке, пока не будет найдена пустая ячейка. Порядок вычисляется на лету.

03.wmf

Самая простая в реализации последовательность проб — линейное пробирование (или линейное исследование). Здесь все просто — в случае коллизии, следующие ячейки проверяются линейно, пока не будет найдена пустая ячейка.

А алгоритм поиска ищет ячейки в том же порядке, что и при вставке, пока не найдет нужный элемент или пустую ячейку, которая говорит о том, что ключ отсутствует. В случае, если таблица будет заполнена, ее придется динамически расширять.

Метод линейного пробирования для открытой индексации на C++:

//Проверка ячеек и вставка значения

class HashEntry {

private:

int key;

int value;

public:

HashEntry(int key, int value) {

this->key = key;

this->value = value;

}

int getKey() {

return key;

}

int getValue() {

return value;

}

void setValue(int value) {

this->value = value;

}

};

Хеширование и хеш-таблицы применяются для более удобного хранения пар ключ-значение. Если нужна максимальная эффективность, то используйте хеш-таблицы со списками будет намного быстрее, чем обычная таблица.

### Переполнение таблицы и рехеширование

**Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы. Что бы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией.**

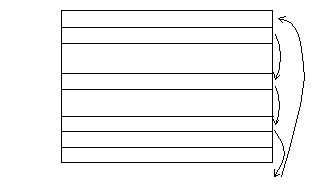


Рис.3.5. Циклический переход к началу таблицы.

С одной стороны это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой к нерациональному расходованию адресного пространства. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных элементов. Поэтому на практике используют **циклический переход к началу таблицы**.

Рассмотрим данный способ на примере метода линейного опробования. При вычислении адреса очередного элемента можно ограничить адрес, взяв в качестве такового остаток от целочисленного деления адреса на длину таблицы **n**.

**Вставка**

1. i = 0
2. a = (h(key) + c\*i) mod n
3. Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**
4. i = i + 1, перейти к шагу 2

В данном алгоритме мы **не учитываем возможность многократного превышения адресного пространства**. **Более корректным будет алгоритм, использующий сдвиг адреса на 1 элемент в случае каждого повторного превышения адресного пространства**. Это повышает вероятность найти свободные элементы в случае повторных циклических переходов к началу таблицы.

**Вставка**

1. i = 0
2. a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n
3. Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**
4. i = i + 1, перейти к шагу 2
5. a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n

Рассматривая возможность выхода за пределы адресного пространства таблицы, мы **не учитывали факторы заполненности таблицы и удачного выбора хеш-функции**. При **большой заполненности таблицы** возникают частые коллизии и циклические переходы в начало таблицы. При **неудачном выборе хеш-функции** происходят аналогичные явления. В наихудшем варианте при полном заполнении таблицы алгоритмы циклического поиска свободного места приведут к **зацикливанию**. Поэтому при использовании хеш-таблиц необходимо стараться избегать очень плотного заполнения таблиц. **Обычно длину таблицы выбирают из расчета двукратного превышения предполагаемого максимального числа записей.** Не всегда при организации хеширования можно правильно оценить требуемую длину таблицы, поэтому в случае большой заполненности таблицы может понадобиться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.

Производить отдельную оценку плотности заполнения таблицы после каждой операции вставки нецелесообразно, поэтому можно производить такую оценку косвенным образом по числу коллизий во время одной вставки. Достаточно определить некоторый порог числа коллизий, при превышении которого следует произвести рехеширование. Кроме того, такая проверка гарантирует невозможность зацикливания алгоритма в случае повторного просмотра элементов таблицы.

Рассмотрим алгоритм вставки, реализующий предлагаемый подход.

**Вставка**

1. i = 0
2. a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n
3. Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, **стоп элемент добавлен**
4. Если i > m , то **стоп требуется рехеширование**
5. i = i + 1, перейти к шагу 2

В данном алгоритме номер итерации сравнивается с пороговым числом m. Следует заметить, что алгоритмы вставки, поиска и удаления должны использовать идентичное образование адреса очередной записи.

**Удаление**

1. i = 0
2. a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n
3. Если t(a) = key, то t(a) =удалено, **стоп элемент удален**
4. Если t(a) = свободно или i>m, то **стоп элемент не найден**
5. i = i + 1, перейти к шагу 2

**Поиск**

1. i = 0
2. a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n
3. Если t(a) = key, то **стоп элемент найден**
4. Если t(a) = свободно или i>m, то **стоп элемент не найден**
5. i = i + 1, перейти к шагу 2

### Оценка качества хеш-функции

Как уже было отмечено, очень важен правильный выбор хеш-функции. **При удачном построении хеш-функции таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления.** Для того чтобы предварительно оценить качество хеш-функции можно провести **имитационное моделирование**. Моделирование проводится следующим образом. Формируется целочисленный массив, длина которого совпадает с длиной хеш-таблицы. Случайно генерируется достаточно большое число ключей, для каждого ключа вычисляется хеш-функция. В элементах массива просчитывается число генераций данного адреса. По результатам такого моделирования можно построить график распределения значений хеш-функции. Для получения корректных оценок число генерируемых ключей должно в несколько раз превышать длину таблицы.

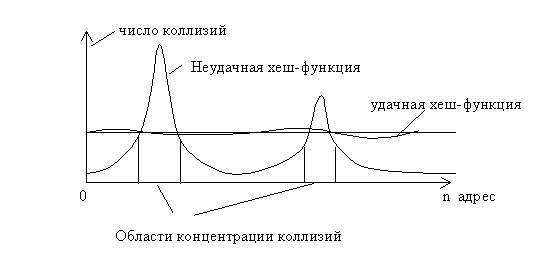


Рис. 3.6. Распределение коллизий в адресном пространстве таблицы

Если число элементов таблицы достаточно велико, то график строится **не для отдельных адресов, а для групп адресов**. Например, все адресное пространство разбивается на 100 фрагментов и подсчитывается число попаданий адреса для каждого фрагмента. Большие неравномерности свидетельствуют о высокой вероятности коллизий в отдельных местах таблицы. Разумеется, такая **оценка является приближенной**, но она позволяет предварительно оценить качество хеш-функции и избежать грубых ошибок при ее построении.

**Оценка будет более точной**, если генерируемые ключи будут более близки к реальным ключам, используемым при заполнении хеш-таблицы. Для символьных ключей очень важно добиться соответствия генерируемых кодов символов тем кодам символов, которые имеются в реальном ключе. Для этого стоит проанализировать, какие символы могут быть использованы в ключе.

Например, если ключ представляет собой фамилию на русском языке, то будут использованы русские буквы. Причем первый символ может быть большой буквой, а остальные малыми. Если ключ представляет собой номерной знак автомобиля, то также несложно определить допустимые коды символов в определенных позициях ключа.

Рассмотрим пример **генерации ключа из десяти латинских букв, первая из которых является большой, а остальные малыми**.

Пример

: ключ 10 символов, 1-й большая латинская буква

2-10 малые латинские буквы

var i:integer; s:string[10];

begin

s[1]:=chr(random(90-65)+65);

for i:=2 to 10 do s[i]:=chr(random(122-97)+97);

end

В данном фрагменте используется тот факт, что допустимые коды символов располагаются последовательными непрерывными участками в кодовой таблице. Рассмотрим более общий случай. Допустим, необходимо сгенерировать ключ из m символов с кодами в диапазоне от n1 до n2.

**Генерация ключа из m символов c кодами в диапазоне от n1 до n2**

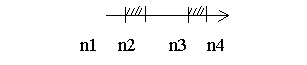
(диапазон непрерывный)

for i:=1 to m do str[i]:=chr(random(n2-n1)+n1);

На практике возможны варианты, когда символы в одних позициях ключа могут принадлежать к разным диапазонам кодов, причем между этими диапазонами может существовать разрыв.

**Генерация ключа из m символов c кодами**

**в диапазоне от n1 до n4 (диапазон имеет разрыв от n2 до n3)**



for i:=1 to m do

begin

x:=random((n4 - n3) + (n2 √ n1));

if x<=(n2 - n1) then str[i]:=chr(x + n1)

else str[i]:=chr(x + n1 + n3 √ n2)

end;

Рассмотрим еще один конкретный пример. Допустим известно, что ключ состоит из 7 символов. Из них три первые символа большие латинские буквы, далее идут две цифры, остальные малые латинские.

**Пример: длина ключа 7 символов**

* 3 большие латинские (коды 65-90)
* 2 цифры (коды 48-57)
* 2 малые латинские (коды 97-122)

var

key: string[7];

begin

for i:=1 to 3 do key[i]:=chr(random(90-65)+65);

for i:=4 to 5 do key[i]:=chr(random(57-48)+57);

for i:=6 to 7 do key[i]:=chr(random(122-97)+97);

end;

В рассматриваемых примерах мы исходили из предположения, что хеширование будет реализовано на языке Turbo Pascal, а коды символов соответствуют альтернативной кодировке.

## Организация данных для ускорения поиска по вторичным ключам

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим **однозначно идентифицировать запись**. Мы будем называть такие ключи **первичными ключами**. Возможен вариант организации таблицы, при котором **отдельный ключ не позволяет однозначно идентифицировать запись**. Такая ситуация часто встречается в базах данных. **Идентификация записи осуществляется по некоторой совокупности ключей**. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются **вторичными ключами**.

Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные. Например, поисковые системы internet часто организованы как наборы записей, соответствующих Web-страницам. В качестве вторичных ключей для поиска выступают ключевые слова, а сама задача поиска сводится к выборке из таблицы некоторого множества записей, содержащих требуемые вторичные ключи.

### Инвертированные индексы

Рассмотрим метод организации таблицы с инвертированными индексами. **Для таблицы строится отдельный набор данных, содержащий так называемые инвертированные индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ.**

Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов, является относительно небольшой по сравнению с другими методами организации таблиц.

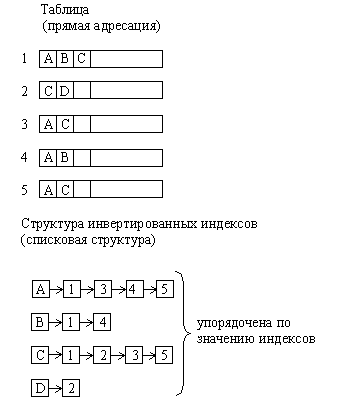


Рис.3.7. Метод организации таблицы с инвертированными индексами

**Недостатками** данной системы являются большие затраты времени на составление вспомогательной структуры данных и ее обновление. Причем эти затраты возрастают с увеличение объема базы данных.

Система инвертированных индексов является чрезвычайно удобной и эффективной при организации поиска в больших таблицах.

### Битовые карты

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт. **Для каждого значения вторичного ключа записей основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а ноль отсутствие.**

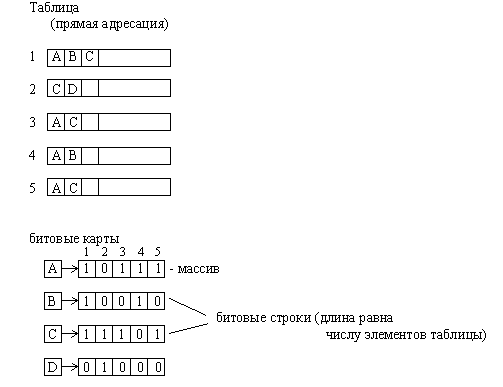


Рис.3.8. Организация вспомогательной структуры данных в виде битовых карт

Основным **преимуществом** такой организации является очень простая и эффективная организация обработки сложных запросов, которые могут объединять значения ключей различными логическими предикатами. В этом случае поиск сводится к выполнению логических операций запроса непосредственно над битовыми строками и интерпретации результирующей битовой строки. Другим преимуществом является простота обновления карты при добавлении записей.

К **недостаткам** битовых карт следует отнести увеличение длины строки пропорционально длине файла. При этом заполненность карты единицами уменьшается с увеличением длины файла. Для большой длине таблицы и редко встречающихся ключах битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.

## Примеры

### Пример 1

// Реализация хеш-таблицы через остаток от деления

// Добавление и поиск элементов осуществлятся через имя (name). В реальных условиях

// такая таблица будет иметь много коллизий, но, поскольку задание является тестовым,

// то избран самый простой, быстрый и наглядный метод реализации.

#include <iostream>

#include <string>

#define PRIME\_SIZE 29 // Использовано просто число

using namespace std;

class Person // Класс, который содержит немного информации о человеке.

{

public:

Person \*next; // При возникновении коллизии элементы будут помещены в односвязный список.

string name;

string surname;

int age;

Person()

{

this->next = NULL;

}

Person (string name, string surname, int age = 0)

{

this->name = name;

this->surname = surname;

this->age = age;

this->next = NULL;

}

~Person()

{

//cout << "Delete " << this->name << endl;

if ( this->next != NULL )

{

delete this->next;

}

}

};

class HashTable // Хеш-таблица, представленная в виде массива элементов (которые в свою очередь представляют список).

{

Person \*table[PRIME\_SIZE];

// Самая простоя хеш-функция. Считает сумму ASCII кодов, делит на константу и

// получает остаток от деления.

static int hash ( string str )

{

int asciisum = 0;

for ( int i = 0; i < str.length(); i++ )

{

asciisum += str[i];

}

return asciisum % PRIME\_SIZE;

}

public:

HashTable()

{

for ( int i = 0; i < PRIME\_SIZE; i++ )

{

table[i] = NULL;

}

}

~HashTable()

{

//cout << "Delete table\n";

for ( int i = 0; i < PRIME\_SIZE; i++ )

{

delete table[i];

}

}

// Вставляет элемент в таблицу

void push ( string name, string surname, int age )

{

int hashNumber = hash(name);

Person \*pers = new Person(name, surname, age);

Person \*place = table[hashNumber];

if ( place == NULL )

{

table[hashNumber] = pers;

return;

}

while ( place->next != NULL )

{

place = place->next;

}

place->next = pers;

}

// Получает элемент из таблицы по его имени.

Person\* find ( string name )

{

int hashNumber = hash ( name );

Person \*result = table[hashNumber];

if ( !result )

{

cout << "Element not found" << endl;

return NULL;

}

while ( result->name != name )

{

if ( !result->next )

{

cout << "Element not found" << endl;

break;

}

result = result->next;

}

return result;

}

};

int main()

{

HashTable newTable;

newTable.push("Artyom", "Devyatov", 20);

newTable.push("Vasya", "Petrov", 23);

newTable.push("Ilja", "Saveljev", 28);

newTable.push("Ilaj", "Savanna", 43); // Имеет коллизию с Ilja

newTable.push("Dmitry", "Kuzychev", 31);

Person \* search = newTable.find("Ilaj");

if ( search )

{

cout << search->surname << endl;

}

int a;

cin >> a;

return 0;

}

### Пример 2.1

#include <iostream>

#include <string>

#include <list>

#include <algorithm>

// Класс "Идентификатор"

// Поскольку это только пример, то в данном случае является просто обёрткой над

// строкой. В реалиности может содержать много дополнительной информации, такой,

// как тип переменной, признак используемости, уровень вложенности и так далее.

class Identifier

{

public:

// Задаётся имя переменной при создании

Identifier(const std::string &name):

m\_name(name)

{

}

public:

// Получается имя переменной

std::string name() const

{

return m\_name;

}

private:

std::string m\_name;

};

// Необходимо для поиска переменной по имени

bool operator==(const Identifier &left, const Identifier &right)

{

return left.name() == right.name();

}

// Функция, вычисляющая хэш

// Принимает идентификатор, хэш которого надо посчитать

// Возвращает вычисленный хэш

size\_t hash(const Identifier &id)

{

// Если имя переменной составляет один символ - возвращается его код,

// умноженный на два

if (id.name().length() == 1)

return 2 \* size\_t(id.name()[0]);

// Иначе возвращается сумма кодов первых двух символов

return size\_t(id.name()[0]) + size\_t(id.name()[1]);

}

// Класс исключение "Идентификатор не найден"

// Нужен для выдачи сообщения о том, что идентификатор не найден в таблице,

// наружу функции поиска идентификатора

class IDNotFoundException : std::exception

{

public:

IDNotFoundException(const std::string id\_name):

m\_what(std::string("Identifier \'") + id\_name + "\' not found!")

{

}

virtual ~IDNotFoundException() throw()

{

}

public:

const char \*what() const throw()

{

return m\_what.c\_str();

}

private:

std::string m\_what;

};

// Класс "Хэш-таблица", основанная на методе цепочек

// Метод цепочек заключается в следующем: таблица представляет собой массив

// связных списков фиксированного размера. Вычисленный хэш-функцией хэш является

// индексом в этом массиве списков. Известно, что список по этому индексу будет

// содержать все идентификаторы, для которых функция вернула одинаовый хэш.

// Осталось только найти идентификатор в данном списке и возвратить ссылку на

// него.

class HashTable

{

public:

// Минимально возможное значение хэша

static const size\_t min\_hash\_value = int('A') + int('0');

// Максимально возможное значение хэша

static const size\_t max\_hash\_value = int('z') + int('z');

// Размер массива списков - хэш-таблицы

static const size\_t hash\_table\_size = max\_hash\_value - min\_hash\_value;

public:

// Добавление идентификатора в хэш-таблицу

void add(const Identifier &id)

{

// Добавление идентификатора в список, расположенный в таблице по

// индексу, вычисленному хэш-функцией (с учётом смещения)

m\_hash\_table[hash(id) - min\_hash\_value].push\_back(id);

}

// Поиск идентификатора в таблице по имени

Identifier &get(const std::string &id\_name)

{

// Сохраняется ссылка на список, в котором потенциально будет

// расположен идентификатор (для простоты)

std::list<Identifier>& line = m\_hash\_table[hash(id\_name) - min\_hash\_value];

// Поиск идентификаторы в списке по имени

std::list<Identifier>::iterator it =

std::find(line.begin(), line.end(), id\_name);

// Если при поиске были просмотренны все элементы списка,и ни один не

// подошёл - сообщаем о том, что идентификатор не найден, посредством

// исключения

if (it == line.end())

throw IDNotFoundException(id\_name);

// Если идентификатор найден - возвращаем ссылку на него

return \*it;

}

private:

// Хэш-таблица - массив связных списков идентификаторов

std::list<Identifier> m\_hash\_table[hash\_table\_size];

};

int main()

{

// Создаём хэш-таблицу

HashTable ht;

// Добавляем в неё различные идентификаторы

ht.add(Identifier("a"));

ht.add(Identifier("aa"));

ht.add(Identifier("if"));

ht.add(Identifier("fi"));

// На случай, если идентификатор не будет найден, заворачиваем код поиска

// идентификаторов в блок try/catch

try

{

// Выводим на экран информацию о различных идентификаторах

std::cout << ht.get("a").name() << std::endl;

std::cout << ht.get("aa").name() << std::endl;

std::cout << ht.get("if").name() << std::endl;

std::cout << ht.get("fi").name() << std::endl;

// Проверяем случай, когда идентификатор не должен быть найден

std::cout << ht.get("hello").name() << std::endl;

}

catch (const IDNotFoundException &ex)

{

// Если идентификатор не найден - сообщаем об этом

std::cerr << ex.what() << std::endl;

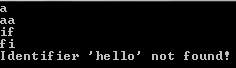
}

int x;

std::cin >> x;

return 0;

}



### Пример 2.2

#include <iostream>

#include <string>

#include <list>

#include <algorithm>

class Identifier

{

public:

Identifier(const std::string& name):

m\_name(name)

{

}

public:

std::string name() const

{

return m\_name;

}

private:

std::string m\_name;

};

bool operator==(const Identifier& left, const Identifier& right)

{

return left.name() == right.name();

}

size\_t hash(const Identifier& id)

{

if (id.name().length() == 1)

return 2 \* size\_t(id.name()[0]);

return size\_t(id.name()[0]) + size\_t(id.name()[1]);

}

class IDNotFoundException : std::exception

{

public:

IDNotFoundException(const std::string id\_name):

m\_what(std::string("Identifier \'") + id\_name + "\' not found!")

{

}

virtual ~IDNotFoundException() throw()

{

}

public:

const char \*what() const throw()

{

return m\_what.c\_str();

}

private:

std::string m\_what;

};

class HashTable

{

public:

static const size\_t min\_hash\_value = int('A') + int('0');

static const size\_t max\_hash\_value = int('z') + int('z');

static const size\_t hash\_table\_size = max\_hash\_value - min\_hash\_value;

public:

void add(const Identifier& id)

{

m\_hash\_table[hash(id) - min\_hash\_value].push\_back(id);

}

Identifier& get(const std::string& id\_name)

{

std::list<Identifier>& line = m\_hash\_table[hash(id\_name) - min\_hash\_value];

std::list<Identifier>::iterator it =

std::find(line.begin(), line.end(), id\_name);

if (it == line.end())

throw IDNotFoundException(id\_name);

return \*it;

}

private:

std::list<Identifier> m\_hash\_table[hash\_table\_size];

};

int main()

{

HashTable ht;

ht.add(Identifier("a"));

ht.add(Identifier("aa"));

ht.add(Identifier("if"));

ht.add(Identifier("fi"));

try

{

std::cout << ht.get("a").name() << std::endl;

std::cout << ht.get("aa").name() << std::endl;

std::cout << ht.get("if").name() << std::endl;

std::cout << ht.get("fi").name() << std::endl;

std::cout << ht.get("hello").name() << std::endl;

}

catch (const IDNotFoundException& ex)

{

std::cerr << ex.what() << std::endl;

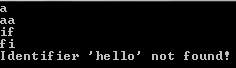
}

int x;

std::cin >> x;

return 0;

}



### Пример 3.1

**Смотри презентацию dsa-spring2015-lec6.pdf**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<conio.h>

#define **HASH\_MUL 31**

#define **HASH\_SIZE 128**

// Хеш-функция для строк [KP, "Practice of Programming"]

unsignedint **hash(**char **\*s)**

{

unsignedint **h = 0;**

char **\*p;**

for **(p = s; \*p !=** '\0'; p++){

**h = h \* HASH\_MUL + (**unsignedint)\*p;

**printf(**"\*p=%c\th=%12d\thash=%3d\n", \*p, h, h % HASH\_SIZE);

**}**

return **h % HASH\_SIZE;** // hash("ivanov") = 1

}

int **main()**

{

unsignedint **h = hash(**"ivanov");

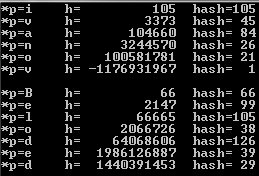
**printf(**"\n");

unsignedint **h2 = hash(**"Beloded");

**\_getch();**

return **0;**

}



### Пример 3.2

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

#define HASHTAB\_SIZE 71

#define HASHTAB\_MUL 31

struct listnode {

char \*key;

int value;

struct listnode \*next;

};

struct listnode \*hashtab[HASHTAB\_SIZE];

unsigned int hashtab\_hash(char \*key)

{

unsigned int h = 0;

char \*p;

for (p = key; \*p != '\0'; p++) {

h = h \* HASHTAB\_MUL + (unsigned int)\*p;

}

return h % HASHTAB\_SIZE;

}

void hashtab\_init(struct listnode \*\*hashtab)

{

int i;

for (i = 0; i < HASHTAB\_SIZE; i++) {

hashtab[i] = NULL;

}

}

void hashtab\_add(struct listnode \*\*hashtab,

char \*key, int value)

{

struct listnode \*node;

int index = hashtab\_hash(key);

// Вставка в начало списка

//node = malloc(sizeof(\*node));//вызывает ошибку

node = (listnode\*)malloc(sizeof(listnode));//это работает

//node = new listnode;//это работает

if (node != NULL) {

node->key = key;

node->value = value;

node->next = hashtab[index];

hashtab[index] = node;

}

}

struct listnode \*hashtab\_lookup(

struct listnode \*\*hashtab,

char \*key)

{

int index;

struct listnode \*node;

index = hashtab\_hash(key);

for (node = hashtab[index];

node != NULL; node = node->next){

if (strcmp(node->key, key) == 0)

return node;

}

return NULL;

}

void hashtab\_delete(struct listnode \*\*hashtab, char \*key)

{

int index;

struct listnode \*p, \*prev = NULL;

index = hashtab\_hash(key);

for (p = hashtab[index]; p != NULL; p = p->next) {

if (strcmp(p->key, key) == 0) {

if (prev == NULL)

hashtab[index] = p->next;

else

prev->next = p->next;

free(p);

return;

}

prev = p;

}

}

int main()

{

struct listnode \*node;

hashtab\_init(hashtab);

hashtab\_add(hashtab, "Tigr", 190);

hashtab\_add(hashtab, "Slon", 2300);

hashtab\_add(hashtab, "Volk", 60);

node = hashtab\_lookup(hashtab, "Slon");

printf("Node: %s, %d\n", node->key, node->value);

hashtab\_delete(hashtab, "Slon");

node = hashtab\_lookup(hashtab, "Slon");

if (node != NULL) {

printf("Node: %s, %d\n", node->key, node->value);

}

else {

printf("Key 'Slon' not found\n");

}

\_getch();

return 0;

}

//int main()

//{

// struct listnode \*node;

// hashtab\_init(hashtab);

// hashtab\_add(hashtab, "Tigr", 190);

// hashtab\_add(hashtab, "Slon", 2300);

// hashtab\_add(hashtab, "Volk", 60);

// node = hashtab\_lookup(hashtab, "Slon");

// printf("Node: %s, %d\n", node->key, node->value);

// hashtab\_delete(hashtab, "Slon");

// node = hashtab\_lookup(hashtab, "Slon");

// if (node != NULL) {

// printf("Node: %s, %d\n", node->key, node->value);

// }

// else {

// printf("Key 'Slon' not found\n");

// }

// \_getch();

// return 0;

//}



### Лабораторная работа № 13. Хэш-таблицы c открытой адресацией

**Hash.h**

#pragma once

#define HASHDEL (void\*) -1

struct Object

{

void\*\* data;

Object(int, int(\*)(void\*));

int size; int N;

int(\*getKey)(void\*);

bool insert(void\*);

int searchInd(int key);

void\* search(int key);

void\* deleteByKey(int key);

bool deleteByValue(void\*);

void scan(void(\*f)(void\*));

};

static void\* DEL = (void\*)HASHDEL;

Object create(int size, int(\*getkey)(void\*));

#undef HASHDEL

**Hash.cpp**

#include "Hash.h"

#include <iostream>

int HashFunction(int key, int size, int p) //Хэш-функция

{

double key2=5 \* ((0.6180339887499 \* key) - int((0.6180339887499 \* key)));

return (p + key) % size;

//или return ((int)(p+fmod(((key\*(sqrt(5.0)-1))/2), 1)))%size;

}

//-------------------------------

int Next\_hash(int hash, int size, int p)

{

return (hash + 5 \* p + 3 \* p \* p) % size;

}

//-------------------------------

Object create(int size, int(\*getkey)(void\*))

{

return \*(new Object(size, getkey));

}

//-------------------------------

Object::Object(int size, int(\*getkey)(void\*))

{

N = 0;

this->size = size;

this->getKey = getkey;

this->data = new void\*[size];

for (int i = 0; i < size; ++i)

data[i] = NULL;

}

//-------------------------------

bool Object::insert(void\* d)

{

bool b = false;

if (N != size)

for (int i = 0, t = getKey(d), j = HashFunction(t, size, 0);

i != size && !b; j = Next\_hash(j, size, ++i))

if (data[j] == NULL || data[j] == DEL)

{

data[j] = d;

N++;

b = true;

}

return b;

}

//-------------------------------

int Object::searchInd(int key)

{

int t = -1;

bool b = false;

if (N != 0)

for (int i = 0, j = HashFunction(key, size, 0);

data[j] != NULL && i != size && !b; j = HashFunction(key, size, ++i))

if (data[j] != DEL)

if (getKey(data[j]) == key)

{

t = j;

b = true;

}

return t;

}

//-------------------------------

void\* Object::search(int key)

{

int t = searchInd(key);

return(t >= 0) ? (data[t]) : (NULL);

}

//-------------------------------

void\* Object::deleteByKey(int key)

{

int i = searchInd(key);

void\* t = data[i];

if (t != NULL)

{

data[i] = DEL;

N--;

}

return t;

}

//-------------------------------

bool Object::deleteByValue(void\* d)

{

return(deleteByKey(getKey(d)) != NULL);

}

//-------------------------------

void Object::scan(void(\*f)(void\*))

{

for (int i = 0; i < this->size; i++)

{

std::cout << " Элемент" << i;

if ((this->data)[i] == NULL)

std::cout << " пусто" << std::endl;

else

if ((this->data)[i] == DEL)

std::cout << " удален" << std::endl;

else

f((this->data)[i]);

}

}

**MyF.cpp**

#include "Hash.h"

#include <iostream>

using namespace std;

struct AAA

{

int key;

char \*mas;

AAA(int k, char\*z)

{

key = k; mas = z;

} AAA() {}

};

//-------------------------------

int key(void\* d)

{

AAA\* f = (AAA\*)d; return f->key;

}

//-------------------------------

void AAA\_print(void\* d)

{

cout << " ключ " << ((AAA\*)d)->key << " - " << ((AAA\*)d)->mas << endl;

}

//-------------------------------

int main()

{

setlocale(LC\_ALL, "rus");

AAA a1(1, "one"), a2(2, "two"), a3(4, "three"), a4(2, "fo");

int siz = 10;

cout << "Введите размер хэш-таблицы" << endl;

cin >> siz;

Object H = create(siz, key); //создать

int choice; int k;

for (;;)

{

cout << "1 - вывод хэш-таблицы" << endl;

cout << "2 - добавление элемента" << endl;

cout << "3 - удаление элемента" << endl;

cout << "4 - поиск элемента" << endl;

cout << "0 - выход" << endl;

cout << "сделайте выбор" << endl; cin >> choice;

switch (choice)

{

case 0:

exit(0);

case 1:

H.scan(AAA\_print);

break;

case 2:

{AAA \*a = new AAA;

char \*str = new char[20];

cout << "введите ключ" << endl;

cin >> k;

a->key = k;

cout << "введите строку" << endl;

cin >> str; a->mas = str;

if (H.N == H.size)

cout << "Таблица заполнена" << endl;

else

H.insert(a);

} break;

case 3:

{ cout << "введите ключ для удаления" << endl;

cin >> k;

H.deleteByKey(k);

} break;

case 4:

{cout << "введите ключ для поиска" << endl;

cin >> k;

if (H.search(k) == NULL)

cout << "Элемент не найден" << endl;

else

AAA\_print(H.search(k));

}

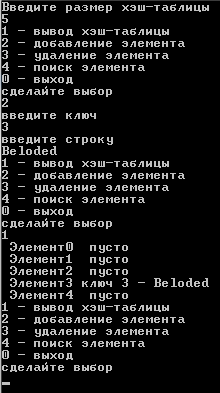
break;

}

}

return 0;

}



### Лабораторная работа № 14. Хэш-таблицы c цепочками

**Hash\_Twin\_Chain.h**

//Заголовочный файл Hash\_ Twin\_Chain.h

#pragma once

#include "Lists.h"

namespace hashTC

{

struct Object

{

int size;

int(\*FunKey)(void\*);

listx::Object\* Hash;

Object(int size, int(\*f)(void\*))

{

size = size;

FunKey = f;

Hash = new listx::Object[size];

};

int hashFunction(void\* data);

bool insert(void\* data);

listx::Element\* search(void\* data);

bool deleteByData(void\* data);

void Scan();

};

Object create(int size, int(\*f)(void\*));

}

**Lists.h**

//Заголовочный файл Lists.h

#include <stddef.h>

#pragma once

#define LISTNIL (Element\*)-1

namespace listx

{

struct Element

{

Element\* prev;

Element\* next;

void\* data;

Element(Element\* prev, void\* data, Element\* next)

{

prev = prev;

data = data;

next = next;

}

Element\* getNext()

{

return next;

};

Element\* getPrev()

{

return prev;

};

};

static Element\* NIL = NULL;

struct Object

{

Element\* head;

Object()

{

head = NIL;

};

Element\* getFirst()

{

return head;

};

Element\* getLast();

Element\* search(void\* data);

bool insert(void\* data);

bool deleteByElement(Element\* e);

bool deleteByData(void\* data);

void scan();

};

Object create();

}

#undef LISTNIL

**Hash\_Table.cpp**

#include "Hash\_Twin\_Chain.h"

#include "Lists.h"

#include <iostream>

struct AAA

{

int key;

char \*mas;

AAA(int k, char\*z)

{

key = k;

mas = z;

}

};

namespace hashTC

{

Object create(int size, int(\*f)(void\*))

{

return \*(new Object(size, f));

}

int Object::hashFunction(void\* data)

{

return (FunKey(data) % size);

};

bool Object::insert(void\* data)

{

return (Hash[hashFunction(data)].insert(data));

};

bool Object::deleteByData(void\* data)

{

return (Hash[hashFunction(data)].deleteByData(data));

};

listx::Element\* Object::search(void\* data)

{

return Hash[hashFunction(data)].search(data);

};

void Object::Scan()

{

for (int i = 0; i < size; i++)

{

Hash[i].scan();

std::cout << '\n';

}

};

}

**Lists.cpp**

#include "Lists.h"

#include <iostream>

struct AAA //элемент таблицы

{

int key;

char \*mas;

};

namespace listx

{

bool Object::insert(void\* data)

{

bool rc = NULL;

if (head == NULL)

head = new Element(NULL, data, head);

else

head = (head->prev = new Element(NULL, data, head));

return rc;

}

//-------------------------------

Element\* Object::search(void\* data)

{

Element\* rc = head;

while ((rc != NULL) && ((((AAA\*)rc->data)->key) != ((AAA\*)data)->key))

rc = rc->next;

return rc;

}

//-------------------------------

bool Object::deleteByElement(Element\* e)

{

bool rc = NULL;

if (rc = (e != NULL))

{

if (e->next != NULL)

e->next->prev = e->prev;

if (e->prev != NULL)

e->prev->next = e->next;

else

head = e->next;

delete e;

}

std::cout << "Элемент удален" << std::endl;

return rc;

}

//-------------------------------

bool Object::deleteByData(void\* data)

{

return deleteByElement(search(data));

}

//-------------------------------

Element\* Object::getLast()

{

listx::Element\* e = this->getFirst(),

\*rc = this->getFirst();

while (e != NULL)

{

rc = e;

e = e->getNext();

};

return rc;

}

Object create()

{

return \*(new Object());

};

//-------------------------------

void Object::scan()

{

listx::Element\* e = this->getFirst();

while (e != NULL)

{

std::cout << ((AAA\*)e->data)->key << '-' << ((AAA\*)e->data)->mas << " / ";

e = e->getNext();

};

}

}

**MyF.cpp**

#include "Hash\_Twin\_Chain.h"

#include <iostream>

#include <stddef.h>

using namespace std;

struct AAA

{

int key;

char \*mas;

AAA(int k, char\*z)

{

key = k;

mas = z;

}

AAA()

{

key = 0;

mas = "";

}

};

//-------------------------------

int hf(void\* d)

{

AAA\* f = (AAA\*)d;

return f->key;

}

//-------------------------------

void AAA\_print(listx::Element\* e)

{

std::cout << ((AAA\*)e->data)->key << '-' << ((AAA\*)e->data)->mas << " / ";

}

//-------------------------------

int main()

{

setlocale(LC\_ALL, "rus");

int current\_size = 7;

cout << "Ведите размер хэш-таблицы" << endl;

cin >> current\_size;

hashTC::Object H = hashTC::create(current\_size, hf);

int choice;

int k;

for (;;)

{

cout << "1 - вывод хэш-таблицы" << endl;

cout << "2 - добавление элемента" << endl;

cout << "3 - удаление элемента" << endl;

cout << "4 - поиск элемента" << endl;

cout << "0 - выход" << endl;

cout << "сделайте выбор" << endl;

cin >> choice;

switch (choice)

{

case 0: //выход

exit(0);

case 2: //добавление элемента

{

AAA \*a = new AAA;

char \*str = new char[20];

cout << "введите ключ" << endl;

cin >> k;

a->key = k;

cout << "введите строку" << endl;

cin >> str;

a->mas = str;

H.insert(a);

}

break;

case 1:

H.Scan();

break;

case 3: //удаление элемента

{

AAA \*b = new AAA;

cout << "введите ключ" << endl;

cin >> k;

b->key = k;

H.deleteByData(b);

}

break;

case 4: //поиск элемента

AAA \*c = new AAA;

cout << "введите ключ" << endl;

cin >> k;

c->key = k;

if (H.search(c) == NULL)

cout << "Элемент не найден" << endl;

else {

cout << "Первый элемент с данным клю-чом" << endl;

AAA\_print(H.search(c));

cout << endl;

}

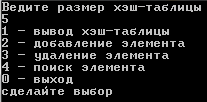
break;

}

}

return 0;

}



## Выводы

* В настоящее время используется широко распространенный метод обеспечения быстрого доступа к большим объемам информации – хеширование.
* Для установления соответствия ключей и данных строится хеш-таблица.
* Хеш-таблица строится при помощи хеш-функций. Практическое применение получили функции прямого доступа, остатков от деления, середины квадрата, свертки.
* При построении хеш-таблиц могут возникать коллизии, то есть ситуации неоднозначного соответствия данных ключу.
* Разрешение коллизий проводится методом цепочек (открытое или внешнее хеширование) или методом открытой адресации (закрытое хеширование).
* Поиск свободных ключей в методе открытой адресации может проводиться методом повторного хеширования с помощью линейного опробования, квадратичного опробования или двойного хеширования.
* Идентификация данных в таблицах может осуществляться как по первичному, так и по вторичному ключу.
* Хеширование имеет широкое практическое применение в теории баз данных, кодировании, банковском деле, криптографии и других областях.

## Для самостоятельной работы. Лабораторная работа ХХ. Алгоритмы хеширования данных

**Цель работы:** изучить построение функции хеширования и алгоритмов хеширования данных и научиться разрабатывать алгоритмы открытого и закрытого хеширования при решении задач на языке C++.

При выполнении лабораторной работы для каждого задания требуется написать программу на языке С++, которая получает на данные с клавиатуры или из входного файла, выполняет их обработку в соответствии с требованиями задания и выводит результат в выходной файл. Для обработки данных необходимо реализовать функции алгоритмов хеширования данных. Ограничениями на входные данные являются максимальный размер строковых данных, допустимый диапазон значений используемых числовых типов в языке С++.

**Задания к лабораторной работе.**

Выполните приведенные ниже задания.

* Составьте хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Осуществить поиск введенной буквы в хеш-таблице.
* Постройте хеш-таблицу из слов произвольного текстового файла, задав ее размерность с экрана. Выведите построенную таблицу слов на экран. Осуществите поиск введенного слова. Выполните программу для различных размерностей таблицы и сравните количество сравнений. Удалите все слова, начинающиеся на указанную букву, выведите таблицу.
* Постройте хеш-таблицу для зарезервированных слов, используемого языка программирования (не менее 20 слов), содержащую HELP для каждого слова. Выдайте на экран подсказку по введенному слову. Добавьте подсказку по вновь введенному слову, используя при необходимости реструктуризацию таблицы. Сравните эффективность добавления ключа в таблицу или ее реструктуризацию для различной степени заполненности таблицы.
* В текстовом файле содержатся целые числа. Постройте хеш-таблицу из чисел файла. Осуществите поиск введенного целого числа в хеш-таблице. Сравните результаты количества сравнений при различном наборе данных в файле.

**Указания к выполнению работы.**

Каждое задание необходимо решить в соответствии с изученным алгоритмами хеширования данных, реализовав программный код на языке С++. Рекомендуется воспользоваться материалами лекции 38, где подробно рассматриваются описание используемых в работе алгоритмов, примеры их реализации на языке С++. Программу для решения каждого задания необходимо разработать методом процедурной абстракции, используя функции, коды которых требуется сопроводить комментариями. Результаты обработки данных следует выводить в выходной файл и дублировать вывод на экране. В отчете следует отразить разработку и обоснование математической модели решения задачи и результаты тестирования программ.

Следует реализовать каждое задание в соответствии с приведенными этапами:

* изучить словесную постановку задачи, выделив при этом все виды данных;
* сформулировать математическую постановку задачи;
* выбрать метод решения задачи, если это необходимо;
* разработать графическую схему алгоритма;
* записать разработанный алгоритм на языке С++;
* разработать контрольный тест к программе;
* отладить программу;
* представить отчет по работе.

**Требования к отчету.**

Отчет по лабораторной работе должен соответствовать следующей структуре.

* Титульный лист.
* Словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов.
* Математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке.
* Алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, задача разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.003-80 и ГОСТ 19.002-80).
* Листинг программы. Подраздел должен содержать текст программы на языке программирования С++, реализованный в среде MS Visual Studio 2010.
* Контрольный тест. Подраздел содержит наборы исходных данных и полученные в ходе выполнения программы результаты.
* Выводы по лабораторной работе.
* Ответы на контрольные вопросы.

**Контрольные вопросы**

* Каков принцип построения хеш-таблиц?
* Существуют ли универсальные методы построения хеш-таблиц? Ответ обоснуйте.
* Почему возможно возникновение коллизий?
* Каковы методы устранения коллизий? Охарактеризуйте их эффективность в различных ситуациях.
* Назовите преимущества открытого и закрытого хеширования.
* В каком случае поиск в хеш-таблицах становится неэффективен?
* Как выбирается метод изменения адреса при повторном хешировании?