Хеш-таблицы

- Хеш-функции
- Коллизии
- Примеры использования

Сравнение с известными структурами данных

Операция	Хеш-таблица	Массив	Список
Вставка	*O*(1)	*O*(N)	*O*(1)
Удаление	*O*(1)	*O*(N)	*O*(1)
Выборка	*O*(1)	*O*(1)	*O*(N)

- Структура данных, позволяющая хранить все элементы в виде пары ключ-значение
- Сложность *O*(1) на операции вставки/удаления/выборки

Основные операции

- insert(k , v)
- get(k)
- delete(k)
- contains(k) - для множества

Когда может понадобиться хеш-таблица

Если мы хотим **хранить большие ключи**, например k будут иметь диапазон от 0 до 10 в 9 степени, при этом не весь этот диапазон будет занят.
Нам бы в этом случае пригодилась функция, которая приводила бы наш большой диапазон к **маленькому диапазону**.
Например мы можем брать остаток от деления h = k%M где M это некоторая константа.
Тогда h(k) где k может быть сколь угодно большим приводит k принадлежащие диапазону 0 до L - 1 где L - большое число к диапазону 0… n-1 где n значительно меньше L.
Появляется вероятность возникновения индексов в таблице i1 и i2 имеющие равные значения при разных ключах.

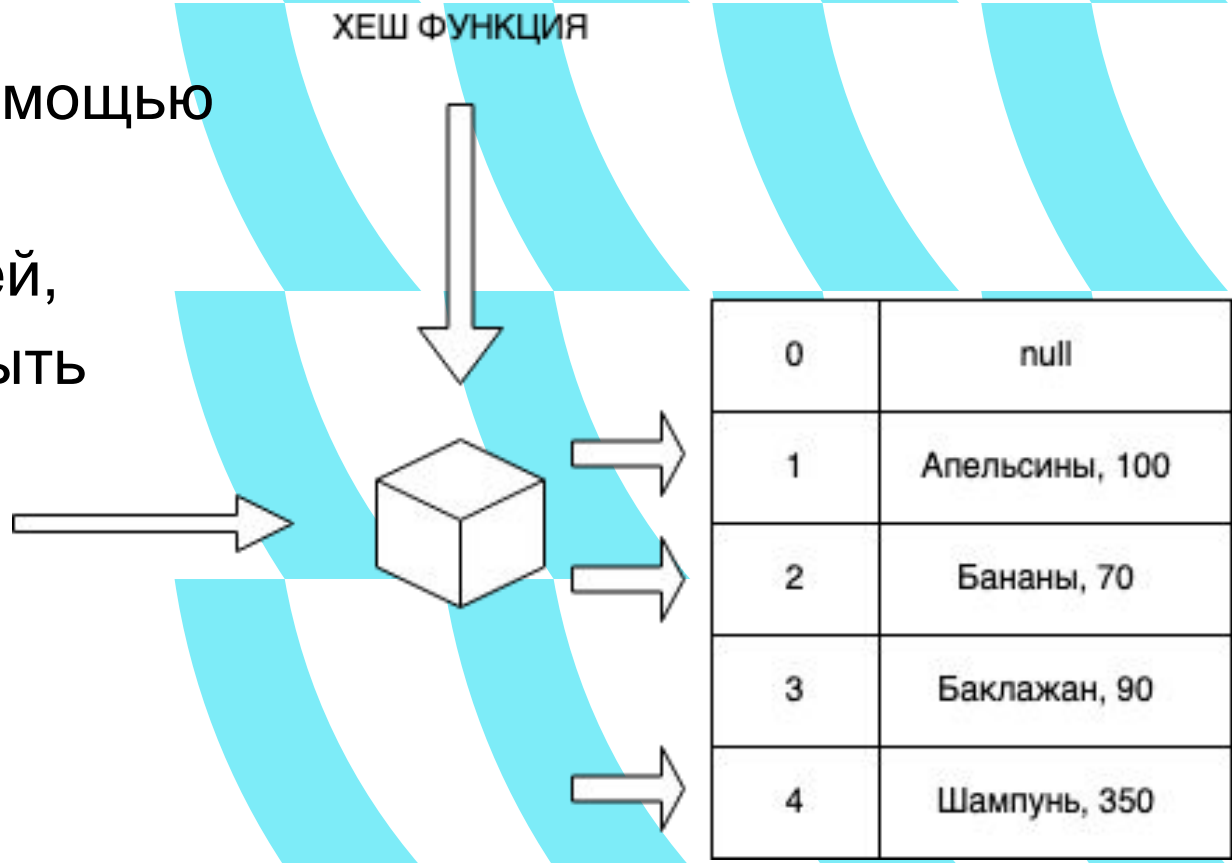
Когда может понадобиться хеш-таблица

- Можно хранить в списке, но нужен более быстрый доступ.
- Когда нет возможности получать по целочисленному индексу, например речь идет о строках или об объектах.

Товар	Цена
Апельсины	100
Бананы	70
Баклажан	90
...	...

Как работает

- Каждый ключ преобразовываем с помощью хеш функции в индекс таблицы
- В не зависимости от схожести ключей, результирующие индексы должны быть разными



Хеш-функция

Hash function

от англ. Hash - “превращать в фарш”, “мешанина”
или функция свёртки

Хеш-функция

- Определяет как и с каким индексом будет храниться значение
- Она должна быть **последовательна**, то есть для одного и того же ключа возвращать одно и тоже значение
- **Равномерно распределенной** (отсутствует связь между ключом и итоговым индексом) для конкретного размера. В нашем случае от 0 до n-1
- Должна быть **простой для вычисления**.
- **Лавинность**. При изменении одного бита во входной последовательности изменяется значительное число выходных битов.
- Для борьбы с соперником - **необратимость**, то есть невозможность восстановления ключа по значения его функции.

Пример хеширование

- Любой ключ можем перевести через таблицу ASCII
- tap - 116 97 112
- Далее можем провести любую операцию с этими числами, например сложение
- Берем остаток от деления результата h(325)
- Как быть со словом pat?

Построение хеш-функции

- Для диапазона от [0, m-1]

Метод остатка от деления:

- h(k) = k % m (остаток от деления). Индекс, который мы получим в результате такого преобразования не выйдет за пределы m-1.
- m простое число, отличное от степени 2. В таком случае у нас будет наилучшее распределение значений индексов. Но в то же время m - размер нашей хеш функции и подобного рода ограничения могут быть не очень удобны.

Метод умножения:

- h(k) = ⌊m * ((k * A) mod 1)⌋ ⌊⌋ - эти скобки - округление до наименьшего целого.
- A - вещественное число, константа. 0 < A < 1; A * k - тоже вещественное; mod 1 выделяет вещественную часть
- Необходимо подобрать A между 0 и 1 так, что бы было максимально равномерное распределение между 0 и m-1 и округляем в наименьшую сторону
- Кнут предложил A считать A=(√5-1)/2 = 0,618 Такое значение дает равномерное распределение значений хеш функции
- Этот метод дает возможность убрать ограничения на возможные значения m
- Всегда существует набор хеш функций, которые присваивают таблице при создании.

Переполнение таблицы

- Load factor n/m - соотношение количества элементов и размера таблицы. Если значение стремится к 1 (~0.75), то нам нужно новая аллокация памяти.
- Оптимизируем хеш функцию под новый объем данных.
- Перехешируем все ключи, заново переписывая все индексы

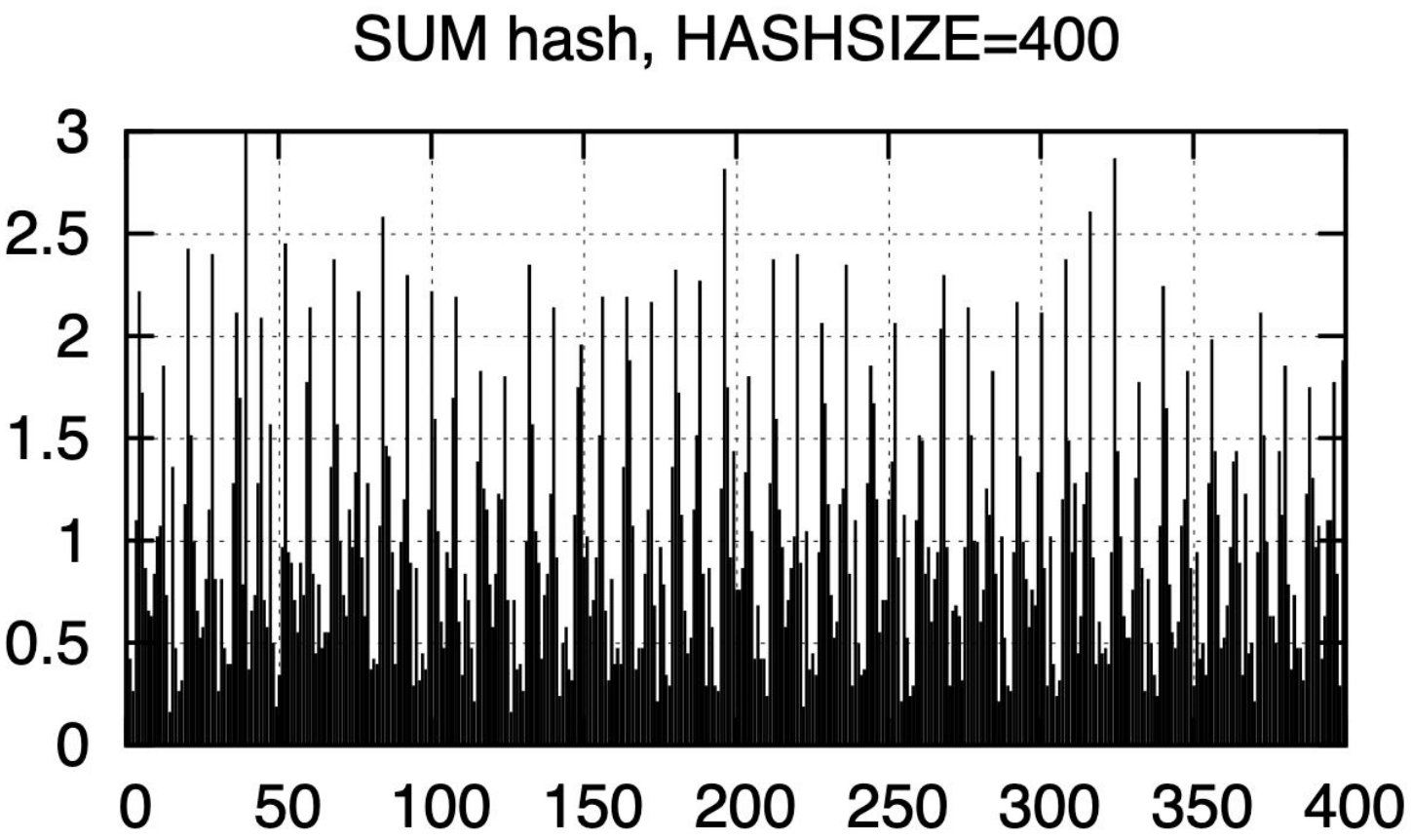
0	null
1	Апельсины, 100
2	Бананы, 70
3	null
4	Шампунь, 350

0	null
1	Апельсины, 100
2	Бананы, 70
3	null
4	null
5	Сахар
6	null
7	null
8	null
9	Шампунь, 350

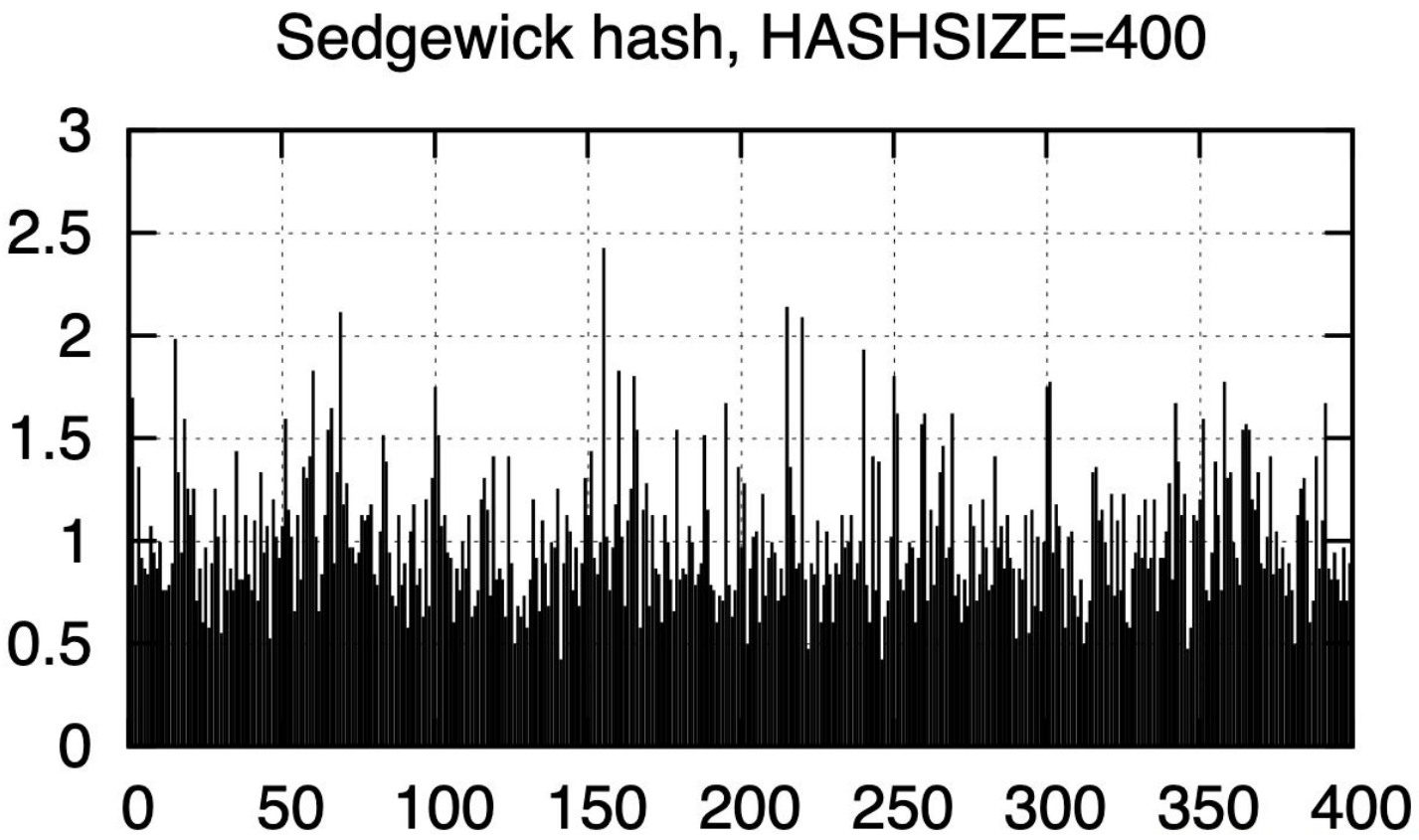
Проблемы хеширование

- Мы не можем всем ключам выдавать уникальные индексы. Мы можем лишь подобрать хорошую хеш-функцию, которая бы минимизировала такую вероятность
- Равномерность распределения, для избежания дублирования индексов

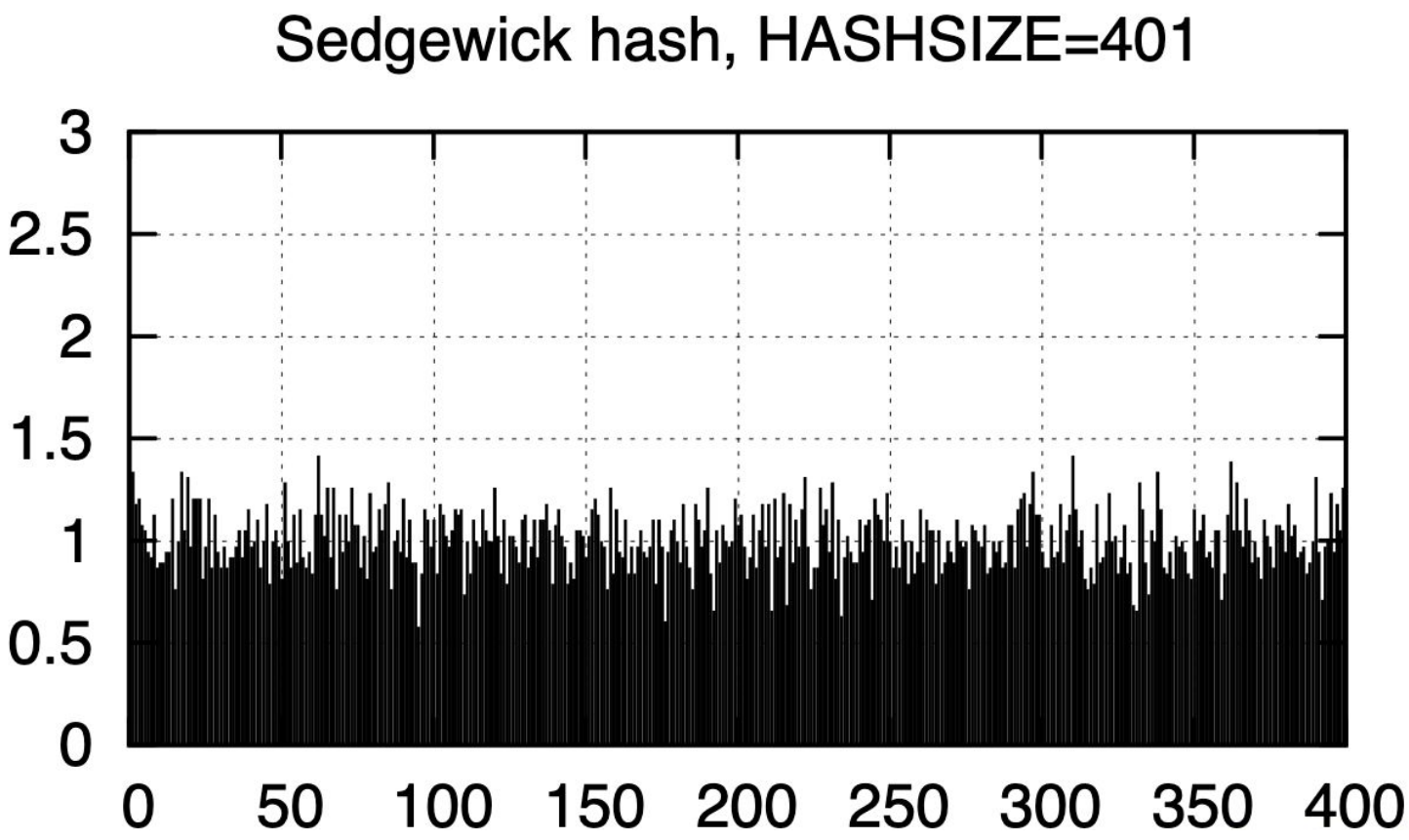
Плохая хеш функция



Хорошая хеш функция



Хорошая хеш функция



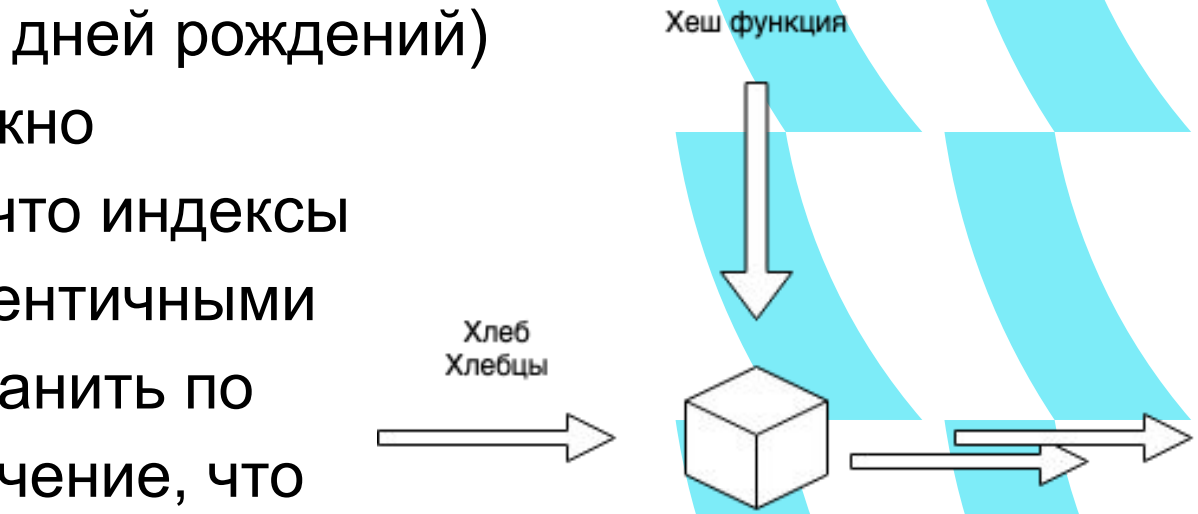
Коллизии

Hash collision or hash clash

от лат. collisio — столкновение

Коллизии

- Разным ключам наша хеш функция выдает одинаковые значения (парадокс дней рождений)
- Для каждой h(k) можно подобрать такие k, что индексы на выходе будут идентичными
- Теперь нам надо хранить по хешу пару ключ-значение, что бы каждый раз иметь возможность удостовериться, что мы выбрали нужное нам значение



0	null
1	Апельсины, 100
2	Бананы, 70
3	Баклажан, 90
4	null
5	null
6	null
7	null
8	null
9	Шампунь, 350

Парадокс дней рождений

Для группы в 23 человека вероятность совпадения дней рождений больше 50%

$$p(n) = 1 - \bar{p}(n)$$

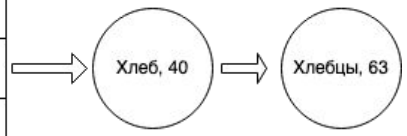
n	p (n)
10	12 %
20	41 %
30	70 %
50	97 %
100	99.99996 %
200	99.99999999999999999999999998 %
300	$(1 - 7\times10^{-73}) \times 100$ %
350	$(1 - 3\times10^{-131}) \times 100$ %
366	100 %

$$\bar{p}(n) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right) = \frac{365 \cdot 364 \cdots (365 - n + 1)}{365^n} = \frac{365!}{365^n (365 - n)!}$$

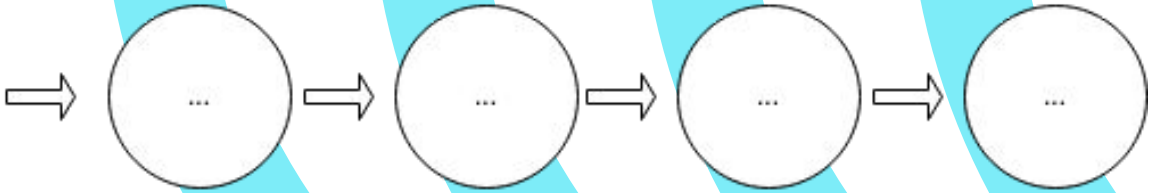
Метод цепочек

- На каждое одинаковое значение создается связанный список
- Если список уже есть, то значение дописываем в конец
- Простота реализации
- В худшем варианте выборка будет стремиться к $O(n)$
- Теперь каждая ячейка хеш таблицы должна хранить

0	null
1	Апельсины, 100
2	Бананы, 70
3	Баклажан, 90
4	null
5	null
6	ptr
7	null
8	null
9	Шампунь, 350



0	null
1	ptr
2	null
3	null
4	null



Метод цепочек
Вставка, удаление и поиск

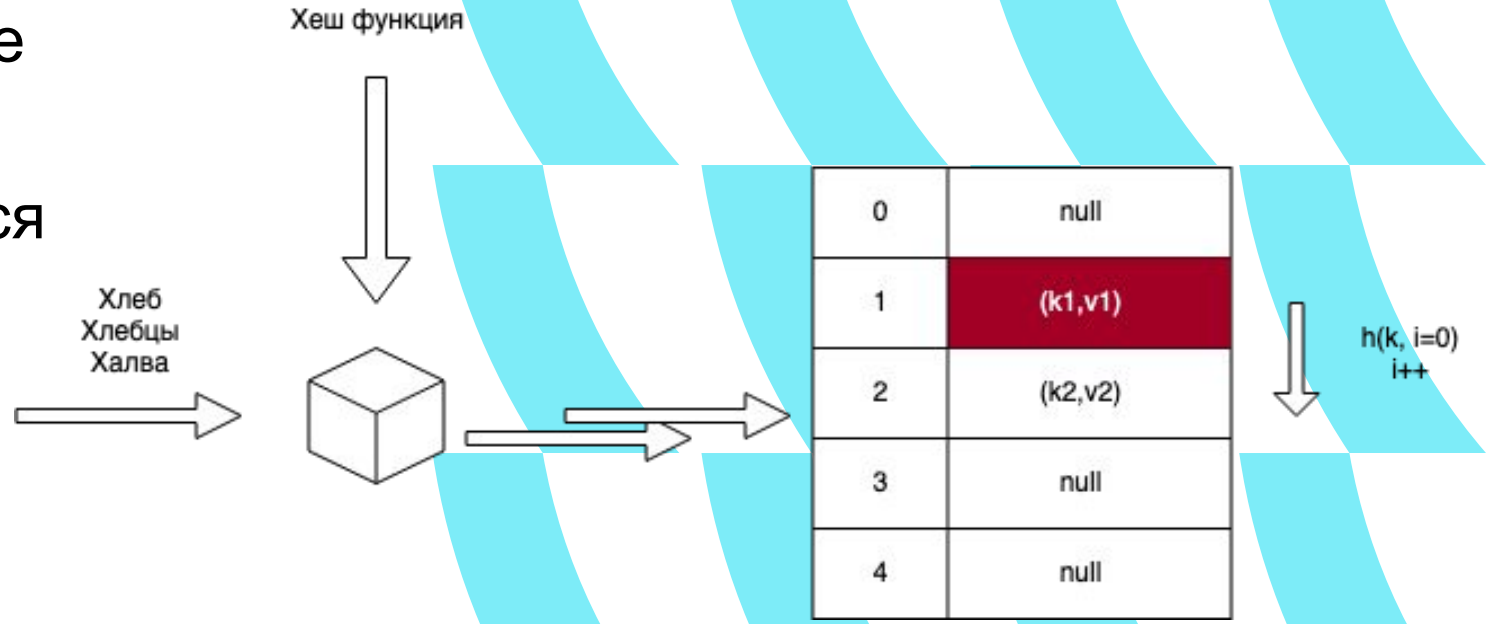
- Как будет работать вставка при методе цепочек У нас есть две пары ключ и значение K1 V1 и K2 V2 Пытаемся вставить первую пару K1 V1. Предположим что наши хэш-функция по ключу К1 присвоила индекс 3 следующим мы пытаемся вставить вторую пару K2 V2 ключу К2 также присваивают индекс 3 в данной ситуации мы просто записываем вторую пару в конец списка. Аналогичным образом будет работать поиск и удаление.
- Мы пропускаем через хф наш ключ, получаем индекс три, находим по этому индексу нужный список и уже по ключу будем искать в списке нашу пару.

Открытая адресация

- В случае коллизии ключ храним в соседней ячейки
- Значения всегда хранятся в ячейках таблицы
- При выборке в случае коллизии мы последовательно перебираем все значения с заданным ключом
- Наиболее эффективно когда заранее известны максимальные размеры входящих данных

Открытая адресация

- Допустим, у нас есть две пары ключ-значение (k1, v1) и (k2, v2)
- У первой пары по ключу вычислили индекс равный 1. Далее мы хотим вставить вторую пару и по k2 получаем тоже индекс равный 1
- В этой ситуации самым простым будет просто увеличивать индекс на 1
- Подобного рода разрешение коллизий, когда мы берем соседнюю ячейку называется линейным пробированием.



Открытая адресация - Вставка

- Добавим в нашу хеш функцию новый параметр i - количество попыток найти подходящую ячейку.
- Учитываем ее каждый раз в случае коллизии
- h(k,i) = (h(k) + i) mod m
- Важно отметить, что при вставке, если мы дошли до конца таблицы, а место так и не нашли, то мы переходим к ее началу. Таким образом мы равномерно распределяем все наши значения, но при этом и сложность операций так же увеличивается.
- Возможно формирование длинных последовательностей занятых ячеек, что негативно сказывается на скорости выборки
- При методе цепочек, сложность ухудшается когда у нас начинают увеличиваться списки, тут сложность ухудшается если нам приходится проходить много ячеек стоящих следом за значением, вызывающим коллизию.

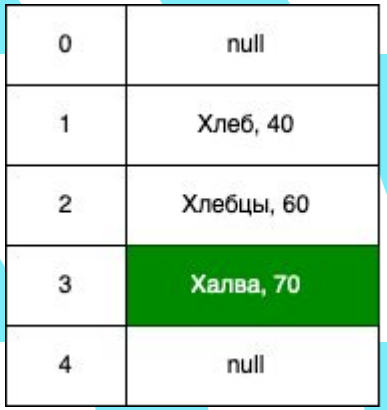
Открытая адресация

- h(v) для любых k находится в диапазоне [0, m-1]
- (h(v) + i) mod m не превышает m

```
function insert(key, value) {
    for i = 0 .. m-1 {
        hash = (h(v) + i) mod m
        if T[hash] == null ||
        T[hash] == REMOVED {
            t[hash] = value
            return
        }
    }
}
```

Открытая адресация - Выборка

- Допустим у нас есть три значения давшие коллизии. Нужный нам ключ - Халва
- Вычисляем индекс
- Проходимся от нашего индекса инкрементируя количество попыток.
- Если ключ не нашли и попали на пустую ячейку, значит нашего ключа нет в таблице
- Пустое значение может быть любым - это может быть как null, nil, None и т.д значение, так и любая константа, которую вы выберете



Открытая адресация

- h(v) для любых k находится в диапазоне [0, m-1]
- (h(v) + i) mod m не превышает m

```
function get(key) {
    for i = 0 .. m-1 {
        hash = (h(v) + i) mod m
        if T[hash] != null {
            if T[hash].key == key
                return T[hash]
        } else {
            return null
        }
    }

    return null
}
```

Открытая адресация - Удаление

- Так же как и в случае выборки ищем нужное значение
- При удалении записываем в освободившиеся ячейку предопределенную константу, например removed
- В случае если при чтении данных мы попадаем на removed, мы просто продолжаем итерироваться, а не прекращаем поиск, как это было бы окажись там null
- В случае большого количества REMOVED ячеек мы начнем терять время на их перебор
- Если появится очередная коллизия с ключом хлеб, то мы сможем вставить новую пару ключ-значение на место removed.

0	null
1	Хлеб, 40
2	REMOVED
3	Халва, 70
4	null

Открытая адресация

- h(v) для любых k находится в диапазоне [0, m-1]
- (h(v) + i) mod m не превышает m

```
function remove(key) {
    for i = 0 .. m-1 {
        hash = (h(v) + i) mod m
        if T[hash] != null
        and T[hash] != REMOVED {
            if T[hash].key == key
                T[hash] = REMOVED
        }
    }
}
```

Двойное хеширование

- Метод открытой адресации так же не идеален.
- Для диапазона от 0 до m-1
- Добавляем вспомогательную хеш функцию
- В случае если ячейка h(k) занята i (номер попытки) = 0, то рассматриваем ячейку
- (h(k) + h1(k)) mod m, затем (h(k) + 2 * h1(k)) mod m
- h(k, i) = (h(k) + i * h1(k)) mod m
- h1(k) в данном случае просто вспомогательная хеш функция. Как видно из примера результат ее работы никогда не должен быть равен 0.
- Чтобы обойти все ячейки таблицы h1(k) и m должны быть взаимно простыми, то есть у них не должно быть общих делителей кроме 1
- Чуть ранее мы обсуждали о циклическом проходе по таблице. Так вот значения, которые возвращает вспомогательная функция также должны гарантировать эту возможность.
- Как пример такой функции может быть: h1(k) = 1 + k mod (m - 1)

Итоги

- Вне зависимости от метода разрешения коллизий мы должны ограничить длину поиска - перебор минимального значения при выборке и при вставке.
- 3-4 сравнения для каждого ключа говорит о неэффективности хеш функции
- В идеальном случае все ячейки будут заняты

Примеры использования

от англ. Hash - “превращать в фарш”, “мешанина”
или функция свёртки

Примеры использования

- **Словарь или Справочник:**
  - Реализация словаря, где ключами могут быть, например, слова, а значениями их определения. Поиск определения по слову происходит очень быстро с использованием хеш-таблицы.
- **Кэширование:**
  - Кэширование результатов сложных вычислений. Если результат для определенного ввода уже был вычислен, он может быть кэширован в хеш-таблице для быстрого доступа в следующий раз.
- **Управление Ресурсами:**
  - Отслеживание занятых и свободных ресурсов. Каждый ресурс (например, номера комнат в отеле) может быть ассоциирован с хеш-таблицей для быстрого определения доступности.
- **Уникальные Значения:**
  - Проверка уникальности элементов в большом наборе данных. Хеш-таблицы обеспечивают быстрый доступ и проверку наличия элемента.
- **Контроль Доступа:**
  - Управление доступом, где ключами могут быть идентификаторы пользователей, а значениями – их права доступа.
- **Системы Кэширования Информации:**
  - Веб-браузеры используют хеш-таблицы для кэширования ранее загруженных страниц, чтобы ускорить доступ к ним при повторных запросах.
- **Контроль Частоты:**
  - Подсчет частоты встречаемости слов в тексте. Каждому слову можно присвоить хеш и использовать хеш-таблицу для подсчета частоты встречаемости.
- **Использование в Графиках:**
  - Графики маршрутов в сетях. Хеш-таблицы могут использоваться для быстрого поиска связей между различными узлами в графе.
- **Оптимизация Базы Данных:**
  - Индексация баз данных. Хеш-таблицы могут использоваться для быстрого поиска записей в базе данных.
- **Криптография:**
  - В хеш-таблицах можно хранить хеши паролей пользователей в системе без хранения самих паролей, обеспечивая безопасность.
- **Управление Потоками:**
  - Организация потоков данных в системах обработки потока. Хеш-таблицы могут быть использованы для быстрого поиска данных в потоке.

Примеры использования

- **Игровая Разработка:**
  - Управление объектами в играх. Хеш-таблицы могут помочь быстро находить объекты по их идентификаторам.
- **Работа с Кодами:**
  - Кэширование результатов выполнения сложных функций в программировании. Хеш-таблицы могут ускорить доступ к результатам функций для заданных параметров.
- **Кеширование Веб-страниц:**
  - Веб-прокси или браузеры могут использовать хеш-таблицы для кэширования веб-страниц и ускорения их загрузки при повторных запросах.
- **Обработка Команд и Событий:**
  - В системах обработки событий, где различные события (например, нажатие клавиши или клик мыши) обрабатываются с использованием хеш-таблицы для эффективного маршрутизации.
- **Системы Контроля версий:**
  - Системы контроля версий, такие как Git, могут использовать хеш-таблицы для индексации и быстрого поиска изменений в коде.
- **Работа с Распределенными Системами:**
  - Распределенные системы могут использовать хеш-таблицы для управления идентификаторами узлов и быстрого поиска данных в сети.
- **Управление Сетевыми Подключениями:**
  - Операционные системы могут использовать хеш-таблицы для отслеживания активных сетевых подключений и управления ими.
- **Оптимизация Графических Интерфейсов:**
  - В графических интерфейсах приложений хеш-таблицы могут использоваться для быстрого поиска и обновления элементов интерфейса.
- **Системы Маршрутизации в Сетях:**
  - В сетях хеш-таблицы могут использоваться для эффективной маршрутизации пакетов данных.
- **Системы Рекомендаций:**
  - В системах рекомендаций хеш-таблицы могут использоваться для быстрого поиска и анализа предпочтений пользователей.
- **Решение Коллизий:**
  - Многие базы данных используют хеш-таблицы для индексации данных. В случае коллизий (когда разным ключам соответствует один и тот же хеш), хеш-таблицы предоставляют эффективные методы их разрешения.

Спасибо!

Авторы:
Илья Почуев
Михаил Павлов