

Язык С. Введение в алгоритмы и структуры данных.

Лекция 13 Хеширование

Сергей Леонидович Бабичев

План лекции

- 1 Обобщённый быстрый поиск.
- 2 Хеш-функции.
- 3 Хеш-таблицы

Абстракция отображение как хранилище

Структура данных операция	Худшее время	Среднее время
Balanced BST: C reate	$O(\log N)$	$O(\log N)$
Balanced BST: R ead	$O(\log N)$	$O(\log N)$
Balanced BST: U psert	$O(\log N)$	$O(\log N)$
Balanced BST: D elete	$O(\log N)$	$O(\log N)$

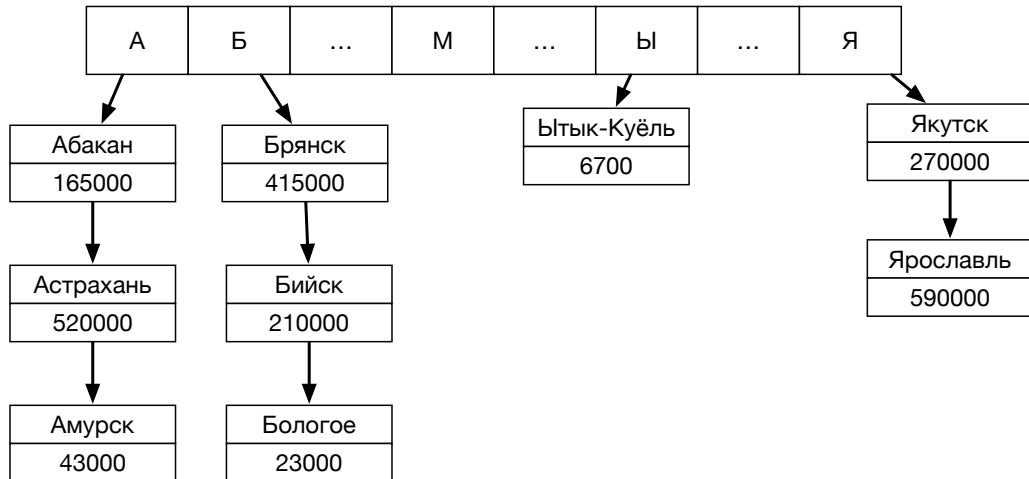
Обобщённый быстрый поиск

Обобщённый быстрый поиск

- Требуется:
 - ▶ Уменьшить амортизационную стоимость поиска.
 - ▶ Уменьшить сложность функции, например, $O(\log N) \rightarrow O(\log \log N)$.

Обобщённый быстрый поиск

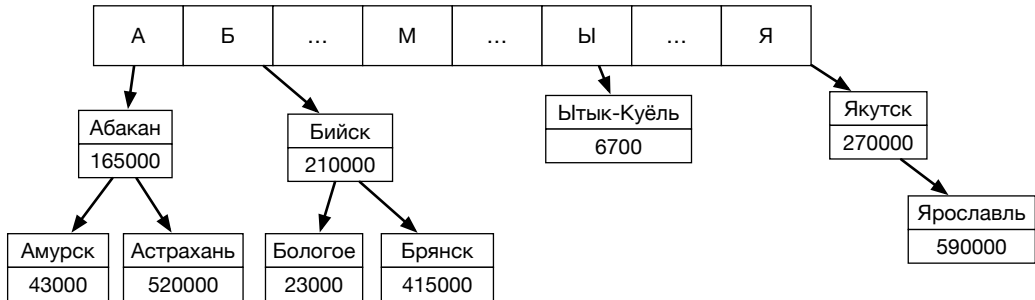
- База данных названий городов и их численности.



33 связанных списка.

Обобщённый быстрый поиск

- База данных названий городов и численности их населения.



33 сбалансированных дерева.

Обобщённый быстрый поиск

- Основная идея — разбиение пространства ключей на независимые подпространства (*partitioning*).
- При независимом разбиении на M подпространств сложность уменьшается.

При разбиении множества N ключей на примерно равные M подмножеств каждое подмножество имеет меньший размер (N/M) и, соответственно, меньшую сложность вычислений.

Обобщённый быстрый поиск

- При увеличении M

$$\lim_{K \rightarrow \infty} T(N, M) = O(1)$$

$$\lim_{K \rightarrow \infty} Mem(N, M) = \infty$$

- Имеется зона оптимальности при $M \approx N$

Обобщённый быстрый поиск

- Требуется иметь детерминированный способ разбиения пространства ключей на M независимых подпространств.
- Условия разбиения:

$$|K_1| \approx |K_2| \approx \dots \approx |K_M|$$

$$\sum_{i=1}^M |K_i| = |K|$$

- Эврика! Создаём функцию $H(K)$, удовлетворяющую некоторым условиям.

Хеш-функции

Хеш-функции

- Функция преобразования:

$$H(K) \rightarrow V$$

$$|D(V)| = M$$

- Отображение пространства ключей K на пространство значений V .
- M — мощность множества пространства значений.

Хеш-функции

- Введём понятие *соперника*, то есть того, кто предоставляет нам ключи.
- Цель *соперника* — предоставлять ключи таким образом, чтобы значения функции оказались не равновероятными.
- *Соперник* знает хеш-функцию и может выбирать ключи.

Хеш-функции

Хотелось бы обеспечить свойства:

- **Эффективность.**

$$T(H(K)) \leq O(L(K)),$$

где $L(K)$ — мера длины ключа K .

- **Равномерность.** Каждое выходное значение равновероятно.

$$p_{H(K_1)} = p_{H(K_2)} = \dots = p_{H(K_M)}$$

- **Лавинность.** При изменении одного бита во входной последовательности изменяется значительное число выходных битов.
- Для борьбы с *соперником* — **необратимость**, то есть невозможность восстановления ключа по значению его функции.

Хеш-функции

Следствия их требуемых свойств.

- Функция не должна быть непрерывной. Для близких значений аргумента должны получаться сильно различающиеся результаты.
- В значениях функции не должно образовываться *кластеров*, множеств близко стоящих точек.

Определение непрерывности для дискретных функций может быть дано неформально.

Примеры плохих функций:

- $H = K^2$

Функция монотонно возрастает. Пространство значений ключа слишком велико и часть значений недостижима.

- $H = \sum_{i=0}^{s.size()-1} s[i]$ для строки s .

Функция даёт одинаковые значения для строк $abcd$ и $abdc$ и отличающиеся на единицу для строк $abcd$ и $abde$. Сопернику легко найти ключи, которые дают равные значения функции.

Универсальная хеш-функция

- Совпадение значений функции для разных значений ключа называется **коллизией**.
- Введём H^* — множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений.
- Это множество **универсально**, если для каждой пары ключей $K_i, K_j, i \neq j$ количество хеш-функций, для которых $H^*(K_i) = H^*(K_j)$ не более $\frac{|H^*|}{m}$.

Универсальная хеш-функция

- Если случайным образом выбирается функция из множества H^* , то для случайной пары ключей $K_i, K_j, i \neq j$ вероятность коллизии не должна превышать $\frac{1}{m}$

Теорема об универсальном множестве хеш-функций

Теорема.

- Пусть множество $Z_p = \{0, 1, \dots, p - 1\}$, множество $Z_p^* = \{1, 2, \dots, p - 1\}$, p — простое число, $a \in Z_p^*$, $b \in Z_p$.
- Тогда множество

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \bmod p) \bmod m\}$$

есть универсальное множество хеш-функций.

Хеш-функции

- Не универсальная, не не столь уж и отвратительная функция

$$h = \sum_{i=0}^n s_i \times 8^i \pmod{HASHSIZE}$$

Обратная схема Горнера:

```
unsigned hash_sum(const char *s, unsigned HASHSIZE)
{
    unsigned sum = 0;
    while (*s != 0) {
        sum <<= 3;
        sum += *s++;
    }
    return sum % HASHSIZE;
}
```

Хеш-функции

- Хеш-функция получше

```
unsigned hash_sedgwick(const char *s, unsigned HASHSIZE) {
    unsigned h, i, a = 31415, b = 27183;
    for (h = 0, i = 0; s[i] != 0;
        i++, a = a * b % (HASHSIZE-1)) {
        h = (a * h + s[i]) % HASHSIZE;
    }
    return h;
}
```

Хеш-функции

- Лучшие по статистическим показателям функции — криптографические.
- Недостатки:
 - ▶ длинный код
 - ▶ медленные

Весьма хорошая хеш-функция

Пользуется свойствами полей Галуа $GF(2^{32})$:

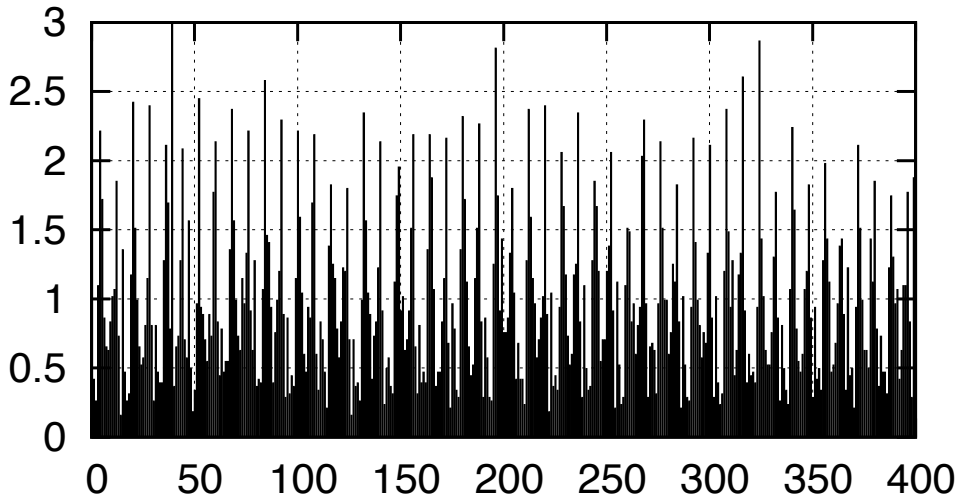
```
uint32 hash(const char *ss) {
    uint32 ret = 0xFFFFFFFF;
    while (*s != 0) {
        ret ^= *s++ & 0xFF;
        ret = (ret >> 8) ^ table[ret & 0xFF];
    }
    return ret ^ 0xFFFFFFFF;
}
```

table вычисляется заранее по какому-нибудь неприводимому полиному в поле $GF(2^{32})$.

Хеш-функции: исследование свойств

Распределение значений для случайных идентификаторов. Плохая функция.

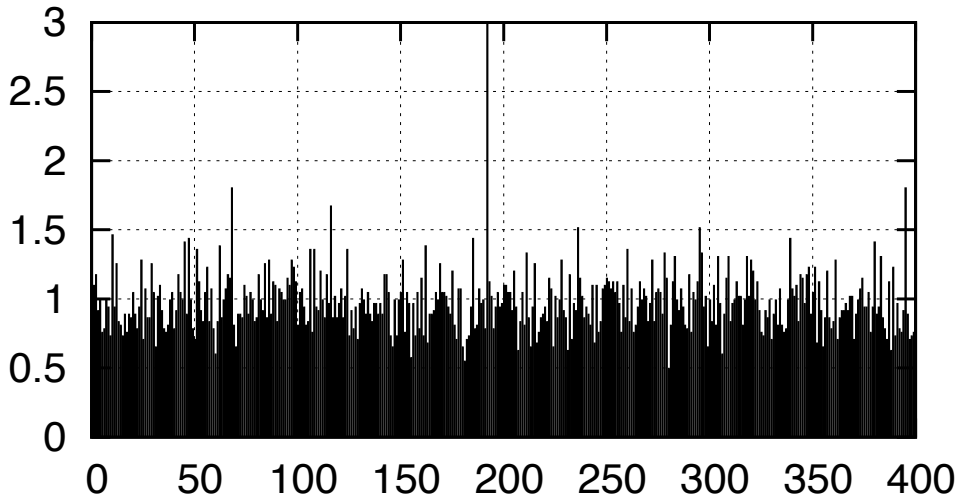
SUM hash, HASHSIZE=400



Хеш-функции

Распределение значений для случайных идентификаторов. Плохая функция.

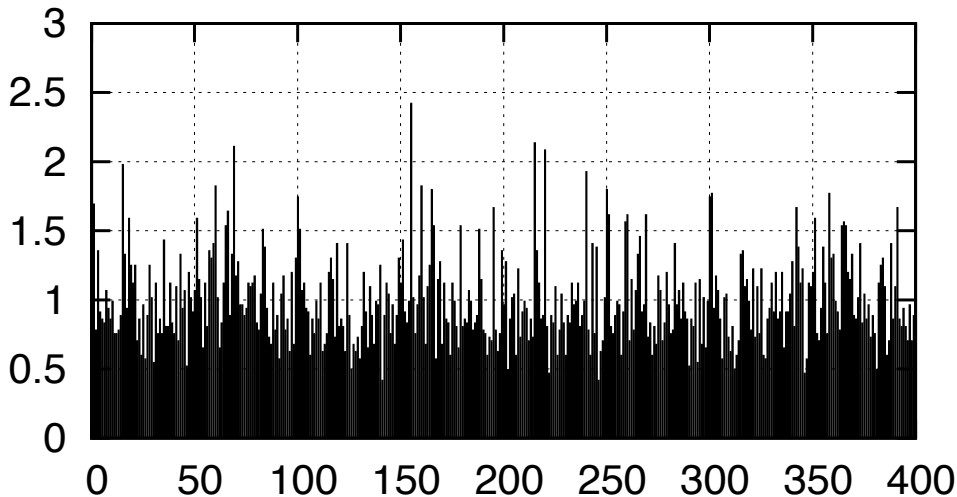
SUM hash, HASHSIZE=401



Хеш-функции

Хорошая функция.

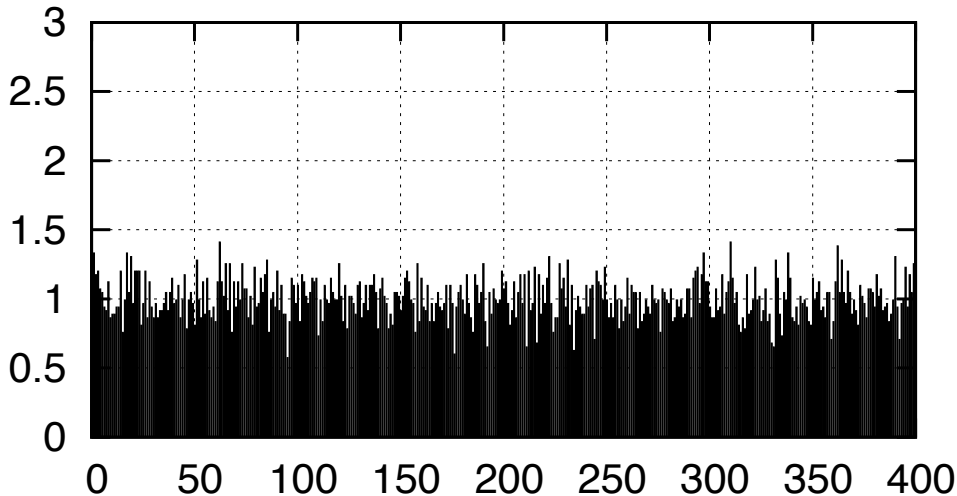
Sedgewick hash, HASHSIZE=400



Хеш-функции

Хорошая функция.

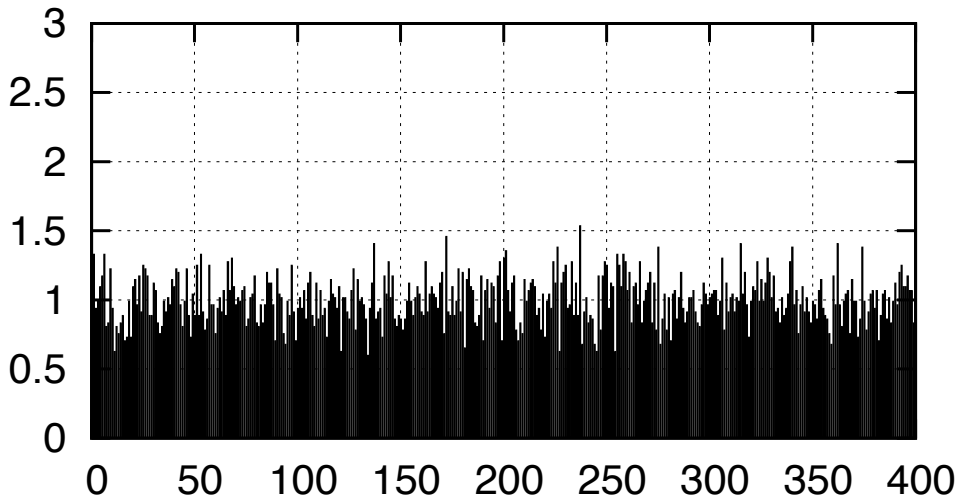
Sedgewick hash, HASHSIZE=401



Хеш-функции

Отличная функция.

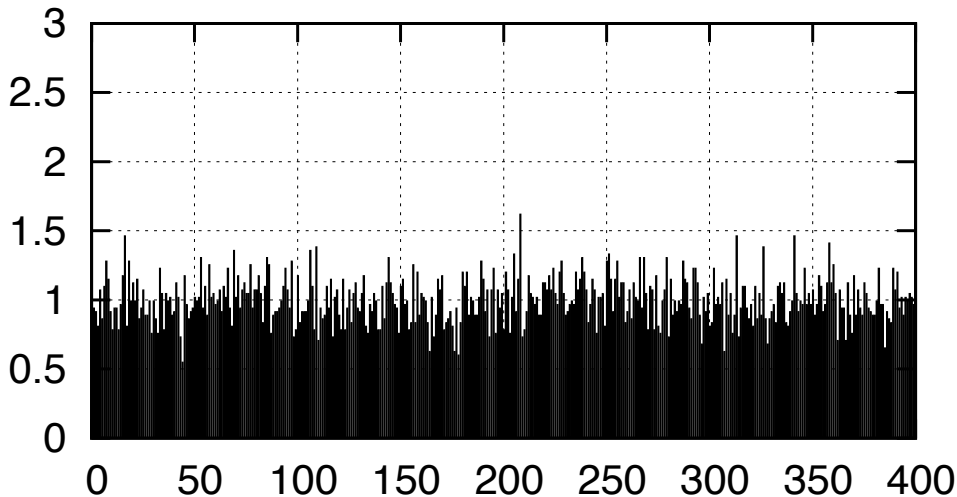
CRC32 hash, HASHSIZE=400



Хеш-функции

Отличная функция.

CRC32 hash, HASHSIZE=401

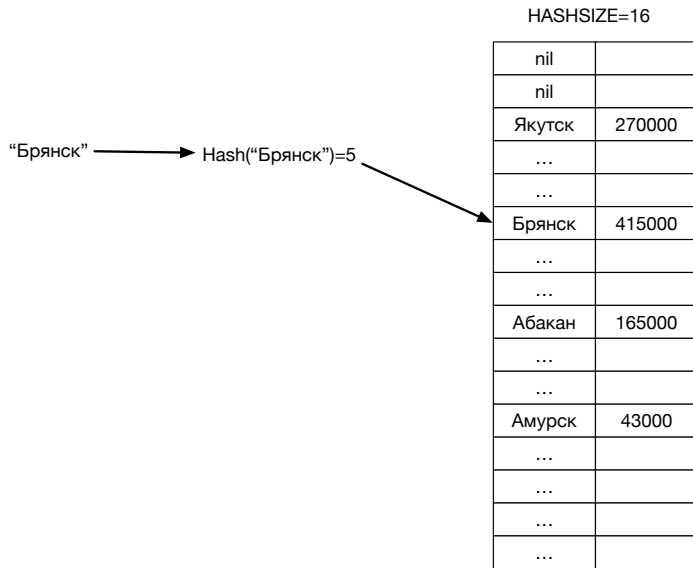


Затраты времени на исполнение хеш-функций

Алгоритм/набор	include.txt	source.txt
hash_sum	890	786
hash_sedgewick	2873	2312
hash_crc	912	801

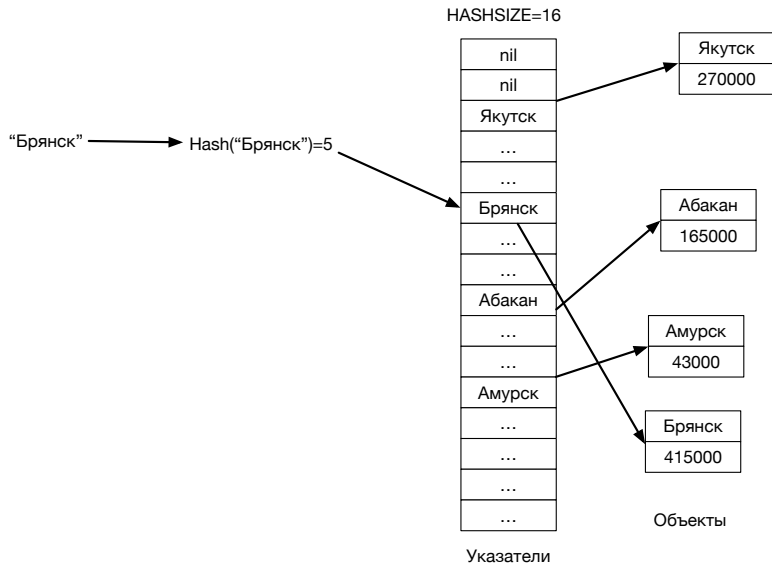
Хеш-таблицы

- Простая хеш-таблица



Хеш-таблицы

- Простая хеш-таблица, обычная реализация в виде массива указателей

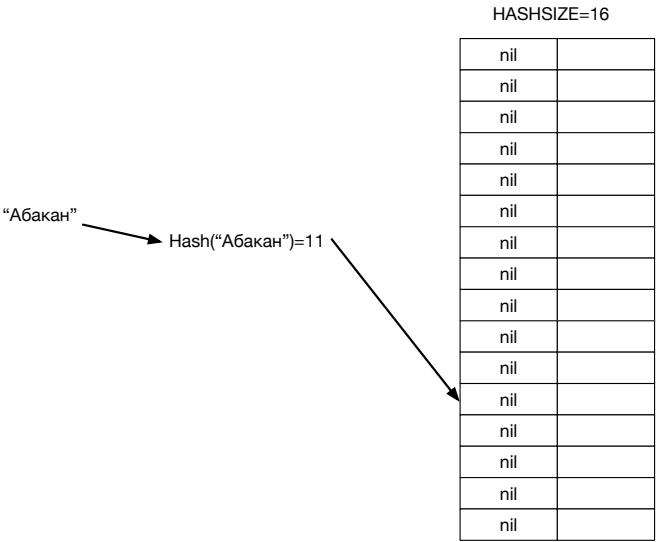


Хеш-таблицы

- Известно количество элементов в контейнере C
- Известен размер массива M
- $\alpha = \frac{C}{M}$ — коэффициент заполнения, *fill-factor*, *load-factor*.
- α — главный показатель хеш-таблицы.

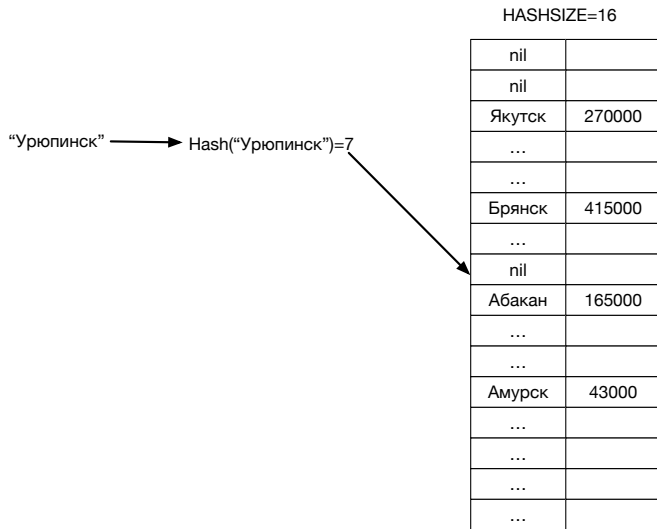
Хеш-таблицы

- Операция создания хеш-таблицы



Хеш-таблицы

- Операция создания хеш-таблицы требует операцию поиска.

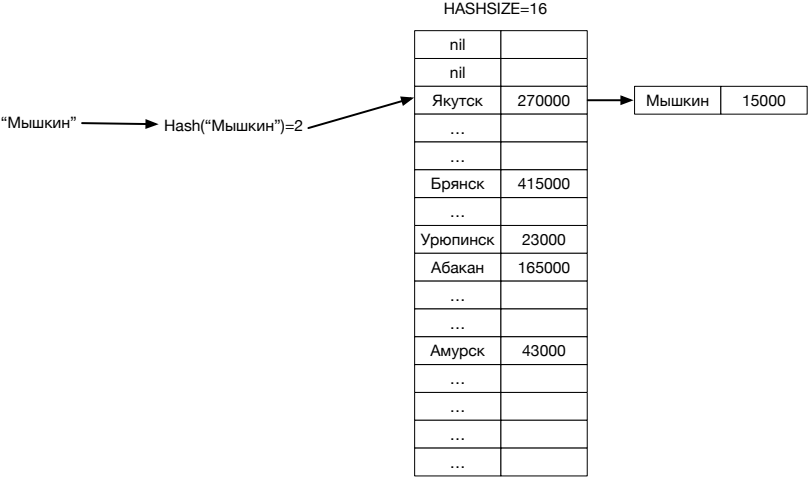


Хеш-таблицы

- $\text{Hash}(\text{"Якутск"}) = 2$
- $\text{Hash}(\text{"Мышкин"}) = 2$
- Это — *коллизия*
- Коллизии — нежелательны.
- Без коллизий сложность операций поиска и вставки равна $O(1)$
- Способы борьбы с коллизиями:
 - ▶ Прямая или закрытая адресация
 - ▶ Открытая адресация
 - ▶ Рехеширование

Хеш-таблицы с прямой адресацией

- При коллизии во время создания элемента создаётся связный список конфликтующих.
- Можно использовать любую поисковую структуру данных.



Хеш-таблицы с прямой адресацией

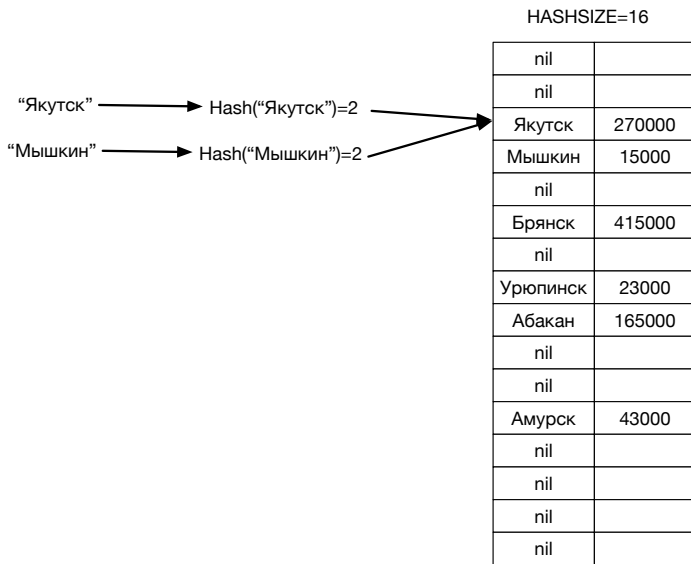
- 1 При поиске вычисляется хеш-функция.
- 2 Определяется место поиска — вторичная поисковая структура данных.
- 3 Если вторичной структуры нет, то нет и элемента.
- 4 Иначе элемент ищется во вторичной структуре.

Хеш-таблицы с прямой адресацией

- 1 При удалении вычисляется хеш-функция.
- 2 Определяется место поиска — вторичная поисковая структуре данных.
- 3 Если вторичной структуры нет, то нет и элемента.
- 4 Иначе элемент удаляется из вторичной структуре.
- 5 Если вторичная структура пуста, удаляет точку входа.

Хеш-таблицы с открытой адресацией

- Другой способ поиска — искать в той же таблице повторно.



Хеш-таблицы с открытой адресацией

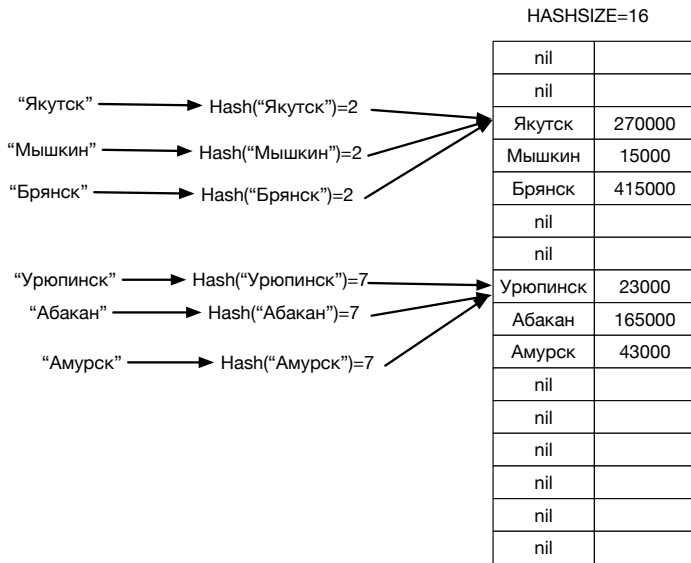
- 1 При поиске существующего вычисляется хеш-функция.
- 2 Определяется место поиска — индекс в хеш-таблице.
- 3 Если по индексу ничего нет, то нет и элемента.
- 4 Иначе по индексу — элемент с нашим ключом — элемент найден.
- 5 Если по индексу — элемент с другим ключом или элемент помечен удалённым, индекс увеличиваем на единицу и переходим к пункту 3.
- 6 Следующий индекс вычисляется по формуле $(index + 1) \bmod M$.

Хеш-таблицы с открытой адресацией

- 1 При вставке вычисляется хеш-функция.
- 2 Определяется место поиска — индекс в хеш-таблице.
- 3 Если по индексу ничего нет или элемент помечен удалённым, то вставляем по индексу и выходим.
- 4 Если по индексу элемент с нашим ключом — меняем данные и выходим.
- 5 Если по индексу элемент с другим ключом то индекс увеличиваем на единицу и переходим к пункту 3.
- 6 Следующий индекс вычисляется по формуле $(index + 1) \bmod M$.

Хеш-таблицы с открытой адресацией

1 Почему мы требуем свойства равномерности от хеш-функции.



Хеш-таблицы с открытой адресацией

- 1 При удалении вычисляется хеш-функция.
- 2 Определяется место поиска — индекс в хеш-таблице.
- 3 Если по индексу ничего нет, то нет и элемента.
- 4 Иначе по индексу — элемент с нашим ключом — элемент найден.
- 5 Если по индексу — элемент с другим ключом, индекс увеличивается на единицу и переходим к пункту 3.
- 6 Следующий индекс вычисляется по формуле $(index + 1) \bmod M$.

Расширение хеш-таблиц

Когда *fill-factor* начинает превосходить 0.5-0.6 таблицу расширяют.

- Создаётся другой массив указателей с нужным размером
- Из оригинального массива в порядке увеличения индексов извлекаются элементы и вставляются в новый массив (таблицу).
- Старый массив удаляется.

Варианты хеш-таблиц: рехеширование

- В ряде случаев можно уменьшить кластеризацию ключей, используя *рехеширование*.
- При конфликте хеша вычисляется вторая хеш-функция $S = H_2(key)$ от ключа или его части, которая тоже должна давать число в диапазоне $[0 \dots M)$.
- Все методы — вставки, поиска, удаления — немного изменяются.
- Следующие попытки поиска ключа производится по адресам $(index + k \cdot S) \bmod M$.
- Для эффективной работы рехеширования требуется, чтобы $\gcd(S, M) = 1$.
- Проще всего добиться этого выбором M как простого числа.

Несколько примеров

Тестовый пример: хеш-таблица, в которую вставляются 100000 случайных строк, содержащих от 10 до 30 букв от А до Z, которые после всех вставок удалялись. Фиксировалось количество сравнений строк (коллизий)

```
size_t h1(const char *s) {
    size_t ret = 0;
    while (*s != 0) {
        ret += *s++ & 0xFF;
    }
    return ret;
}
```

```
size_t h2(const char *s) {
    size_t ret = 0;
    while (*s != 0) {
        ret += ret + *s++ & 0xFF;
    }
    return ret;
}
```

```
size_t h3(const char *s) {
    size_t ret = 0;
    while (*s != 0) {
        ret *= 257;
        ret += *s++ & 0xFF;
    }
    return ret;
}
```

```
size_t h4(const char *s) {
    size_t ret = 0;
    while (*s != 0) {
        ret *= 259;
        ret += *s++ & 0xFF;
    }
    return ret;
}
```

Результаты испытаний

Количество сравнений

Первичная/Вторичная	h1	h2	h3	h4	-
h1	-	11782017	889315	893720	12330837079
h2	17404350	-	798321	785311	12118515706
h3	354821	354596	-	354986	505338
h4	356858	355647	355057		503271