Обобщенный быстрый поиск

Наумов Д.А., доц. каф. КТ

Алгоритмы и структуры данных, 2021

Содержание лекции

Содержание лекции

Двоичный поиск – это хорошо и достаточно быстро, но можно ли искать быстрее?

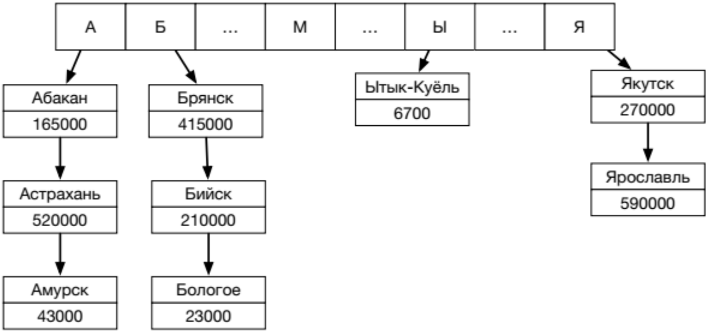


Рис.: Быстрый поиск: разбиение множества ключей на подмножества в виде связных списков

Собственно говоря, почему мы обязаны использовать именно связные списки, почему, например, не уже изученные нами деревья?

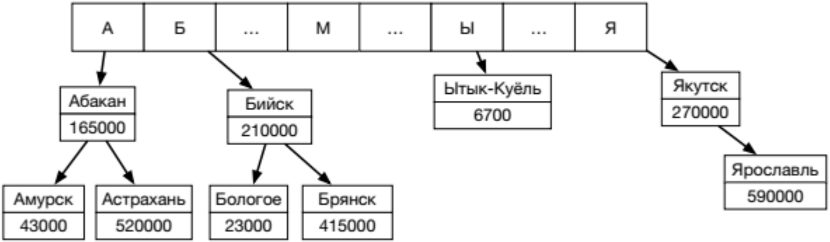


Рис.: Быстрый поиск: разбиение множества ключей на подмножества в виде деревьев поиска

Но такое прямолинейное разбиение не вполне хорошо – на мягкий и твёрдый знак названий нет, для некоторых букв названий совсем мало, некоторые буквы очень популярны.

Основная идея действительно быстрого поиска – разбиение пространства ключей на независимые подпространства (*partitioning*). При независимом разбиении на М подпространств сложность поиска уменьшается.

$$C \cdot O(N) \to \frac{C}{M} O(N)$$
$$C \cdot O(N \cdot log(N)) \to \frac{C}{M} O(N \cdot log(N))$$

При увеличении М время поиска уменьшается:

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

а требуемая память увеличивается:

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

При $M \approx N$ имеется зона оптимальности – поиск уже будет проводиться за $O(1)$, а вот потребная память ещё не столь велика – $O(N)$.

Примитивное разбиение пространства ключей по первым буквам – не очень хороший вариант. Хотелось бы иметь детерминированный способ разбиения пространства ключей на M независимых подпространств.
Условие разбиения – мощность множеств ключей, принадлежащих каждому подпространству, должна быть примерно равна.

$$|K_1| \approx |K_2| \approx \ldots \approx |K_M|$$

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

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

Содержание лекции

Хеш-функция

есть функция преобразования множество ключей K на множество V мощностью M.

$$H(K) \to V$$

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

Введём понятие **соперника**, 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.

Чтобы быть независимыми от соперника – и для удобства практического применения, хотелось бы для функции H(K) обеспечить следующие свойства:

- **Эффективность** – время вычисления хеш-функции не должно быть велико.

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

  где L(K) – мера длины ключа K.
- **Равномерность** – Каждое выходное значение равновероятно

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

- **Лавинность** – при незначительном изменении входной последовательности выходное значение должно меняться значительно, иначе соперник может просто подобрать ключ.
- Для борьбы с соперником – **необратимость**, то есть невозможность восстановления ключа по значению его функции.

Следствия из требуемых свойств:

- Функция не должна быть близка к непрерывной. Неплохо было бы, если бы для близких значений аргумента получались сильно различающиеся результаты.
- В значениях функции не должно образовываться кластеров, множеств близко стоящих точек.

Примеры плохих функций:

- $H = K^2 \ mod \ 10000 для K < 100$ 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.

Совпадение значений функции для разных значений ключа называется **коллизией**.

Большое количество коллизий для данного множества ключей – плохая хеш-функция. Конечно, без коллизий обойтись не удастся, но, если оно больше $\frac{1}{|D(M)|}$, с хеш-функцией какие-то проблемы.

Введём H – множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений:

Множество хеш-функций универсально

если для каждой пары ключей K_i, K_j, ij количество хеш-функций, для которых $H(K_i) = H({}_K j)$ не более $\frac{|H^*|}{m}$.

При наличии такого универсального множества борьба с соперником закончится нашей победой, если мы каждый раз будем выбирать случайным образом функцию из этого множества.

Пусть множество $Z_p = \{0, 1, \ldots p1\}$, множество $Z_p = \{1, 2, \ldots p - 1\}$, p – простое число, а $a \in Z_p, b \in Z_p$.

Тогда множество $H(p, m) = \{H(a, b, K) = ((aK + b) \ mod \ p) \ mod \ m\}$ есть универсальное множество хеш-функций.

Обратим внимание на то, что для хорошей универсальной функции множество Z_p есть множество неотрицательных целых чисел, что не всегда совпадает с нашими запросами

- часто требуется получить значение хешфункции (часто говорят: получить хеш) от строк, объектов и прочих сущностей.
- Вполне достаточно рассматривать значение ключа как последовательность битов, только трактовать эту последовательность придётся как целое число соответствующей разрядности, то есть перейти к операциям над (N)-числами.
- Для ключей с большой длиной это будет весьма неэффективно. Поэтому на практике часто применяют специальные, не универсальные хеш-функции.

Для строки можно использовать вариант полиномиальной хешфункции:

$$h = \sum_{i=0}^{n} s_i \times q^i (mod\ HASHSIZE)$$

```
unsigned
hash_sum(string s, unsigned q, unsigned HASHSIZE)
{
    unsigned sum = 0;
    for (size_t i = 0; i < s.size(); i++) {
        sum = sum * q + s[i];
    }
    return sum % HASHSIZE;
}
```

Предлагается хеш-функция, основанная на идеях из теории генерации псевдослучайных чисел:

```
unsigned
hash_sedgewick(string s, unsigned HASHSIZE)
{
    unsigned h, i, a = 31415, b = 27183;
    for (h = 0, i = 0; i < s.size();
        i++, a = a * b % (HASHSIZE-1)) {
        h = (a * h + s[i]) % HASHSIZE;
    }
    return h;
}
```

Лучшие по статистическим показателям функции – хеш-функции, применяемые в криптографии. К сожалению, у них есть и свои недостатки: у них длинный код и они достаточно медленные.

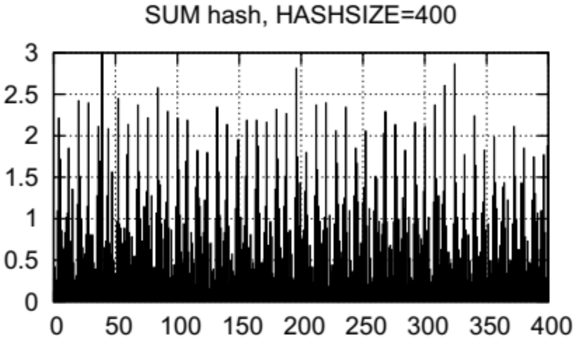
Одна из хороших и быстрых функций основана на полях Галуа. Это функция CRC, очень популярная в архиваторах для контроля целостности данных (есть варианты с различным количеством бит). Она использует специальную таблицу _table[256] и её код очень прост:

```
uint32
crc32(uchar *ptr, unsigned length)
{
    uint32 c = 0xFFFFFFFF;
    while (length) {
        c ^= (uint32) (ptr[0]);
        c = (c >> 8) ^ _table[c & 0xFF];
        ptr++;
        length--;
    }
    return c ^ 0xFFFFFFFF;
}
```

Саму таблицу _table можно создать функцией генерации таблицы – или иметь уже готовую.

Исследование хэш-функций

Каждая из картинок даёт распределение относительной встречаемости значения хеш-функции от значения аргумента. В качестве аргументов были взяты названия идентификаторов, встречающихся в одном комплексе программ примерно в 2 миллиона строк на C++ и C. Идеальная картинка – закрашенный прямоугольник высотой 1.



Попробуем поменять HASHSIZE на 401.

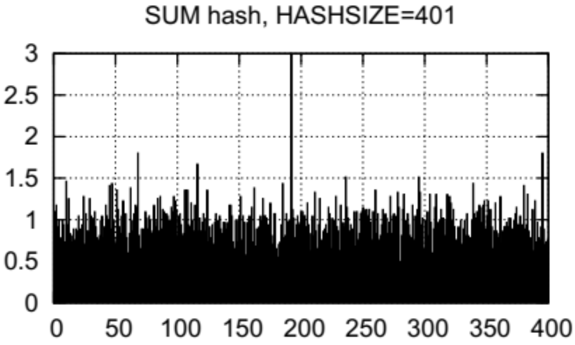


Рис.: hash_sum для q=8, HASHSIZE принято за 401

А что там за пик в районе 190? Не приведёт ли он к проблемам в дельнейшем? Может и привести.

Этот же набор ключей для функции hash_sedgewick и HASHSIZE=400.

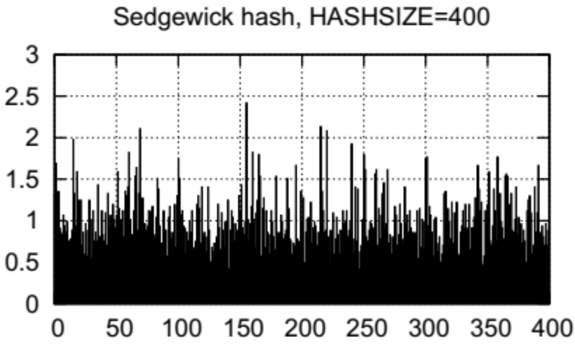


Рис.: hash_sedgewick для q=8, HASHSIZE принято за 400

Этот же набор ключей для функции hash_sedgewick и HASHSIZE=401.

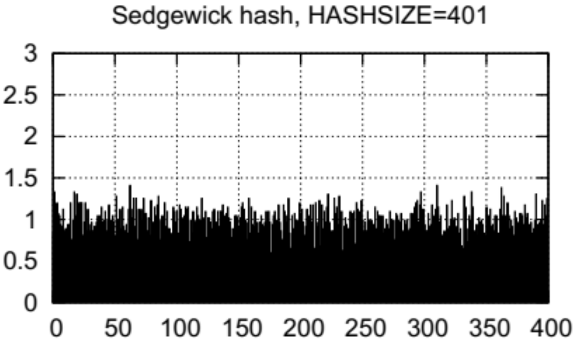


Рис.: hash_sedgewick для q=8, HASHSIZE принято за 401

Попробуем hash_crc для HASHSIZE, равном 400.

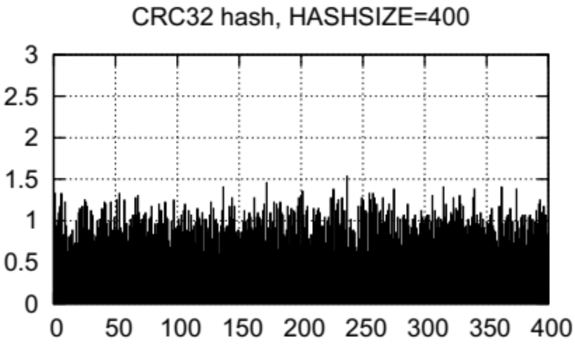


Рис.: hash_crc для HASHSIZE=400

Попробуем hash_crc для HASHSIZE, равном 401.

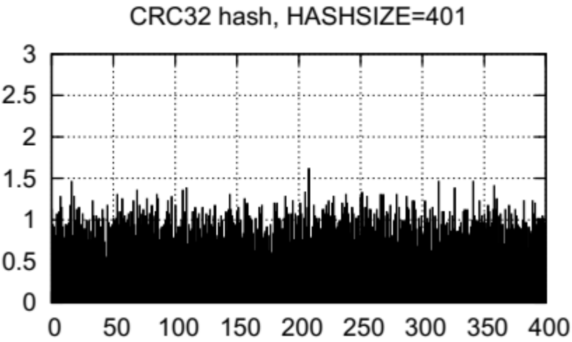


Рис.: hash_crc для HASHSIZE=401

Мы видим, что на результат влияет размер множества значений хешфункции (HASHSIZE). Для простого числа 401 результат почти всегда лучше, чем для составного 400. Сама хеш-функция тоже очень важна.

Алгоритм/набор	include.txt	source.txt
hash_sum	890	786
hash_sedgewick	2873	2312
hash_crc32	912	801

В таблице приведены затраты времени на исполнение программы исполнения хеш-функций для упомянутого набора идентификаторов. Функция hash_sedgewick оказалась самой медленной, так как для каждого символа входной строки потребовалось исполнять операцию нахождения остатка по модулю. А эта операция – одна из самых медленных на современных компьютерах.

Содержание лекции

Вероятностное множество

Вероятностное множество

структура данных, реализующая функциональность абстракции «множество», имеющая операции **insert** и **find** с отсутствием гарантии точности результата поиска в этом множестве.

- Результаты поиска могут быть ложноположительными, если элемент отсутствует, но операция find вернула истину.
- Отсутствие элемента всегда определяется точно, то есть ложноотрицательных результатов быть не может.

Фильтр Блума

один из вариантов реализации вероятностных множеств. В основе его представления лежит битовый массив из m бит, и для его функционирования требуется n различных хеш-функций $h_1, \ldots, h_n$, равномерно отображающих входные ключи на номера битов (от 0 до $m1$).

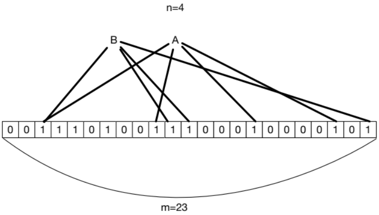


Рис.: Фильтр Блума: внесены ключи A и B

- Операция **insert(key)**: вычисляются все n хеш-функций от key — и устанавливаются соответствующие биты в массиве.

- При операции find вычисляются все n хеш-функций. Если хотя бы один бит в массиве не присутствует, то мы точно знаем, что такого элемента НЕТ — если бы он был, все биты, соответствующие хеш-функциям, были бы установлены. А если совпали все биты, то ответ: МОЖЕТ БЫТЬ — вполне могло оказаться, что биты установлены другими ключами.

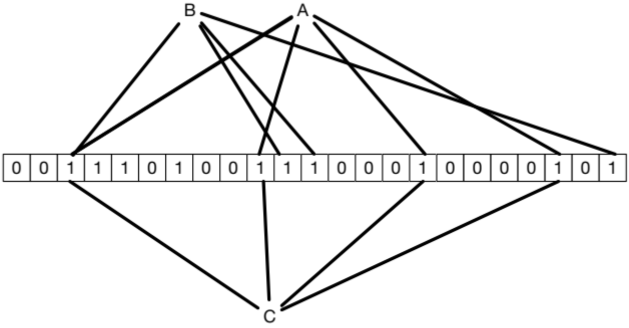


Рис.: Фильтр Блума: ключ C дает ответ МОЖЕТ БЫТЬ

Содержание лекции

Хэш-таблицы

- Простая хеш-таблица есть массив пар *{ключ,значение}*.
- Пусть размер этого массива будет HASHSIZE, и хеш-функция от ключа будет давать числа в диапазоне [0..HASHSIZE).
- Тогда применение хеш-функции к ключу даст нам индекс в этом массиве, который поможет нам найти пару *{ключ,значение}*.

Каким образом мы будем искать эту пару далее, зависит от организации **хеш-таблицы**.

Коэффициент заполнения

Если известны количество элементов в контейнере C и размер массива M, то $\alpha = \frac{C}{M}$ – коэффициент заполнения, *fill-factor* или *load-factor*

α – главный показатель хеш-таблицы.

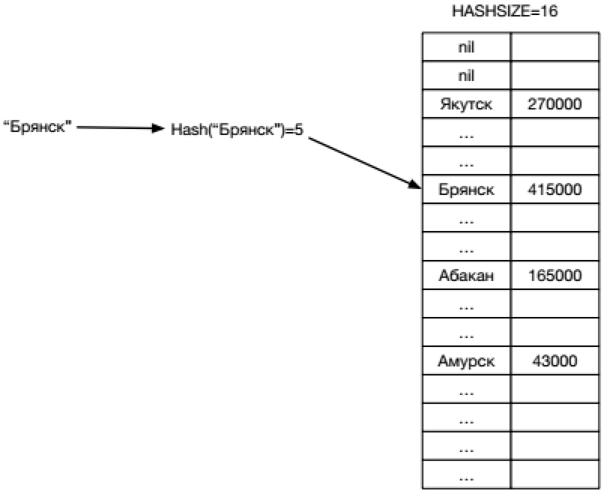


Рис.: Пример хэш-таблицы

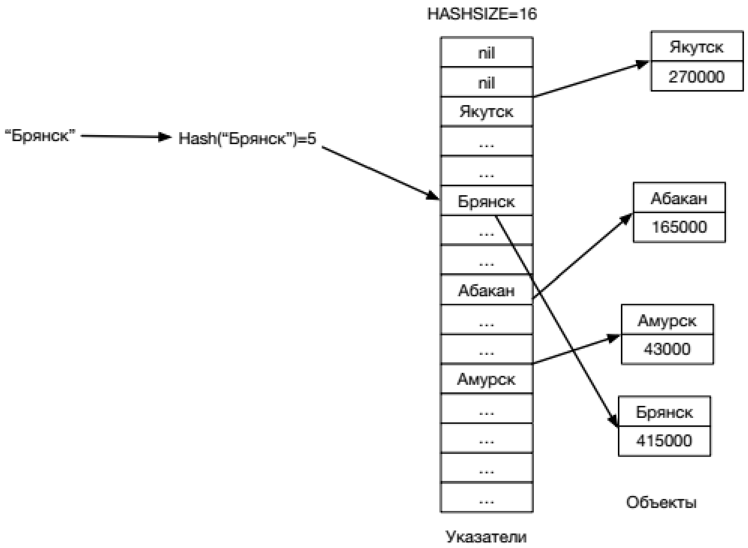


Рис.: Хеш-таблица с хранением указателей на пары ключ/значение

Хэш-таблицы

Операция **create**

Операция create для хеш-таблицы часто имеет аргумент, равный начальному количеству элементов массива.
Массив заполняется либо nullptr, либо парами с невозможным значением ключей — нам всегда необходимо знать, свободен ли слот для размещения пары.

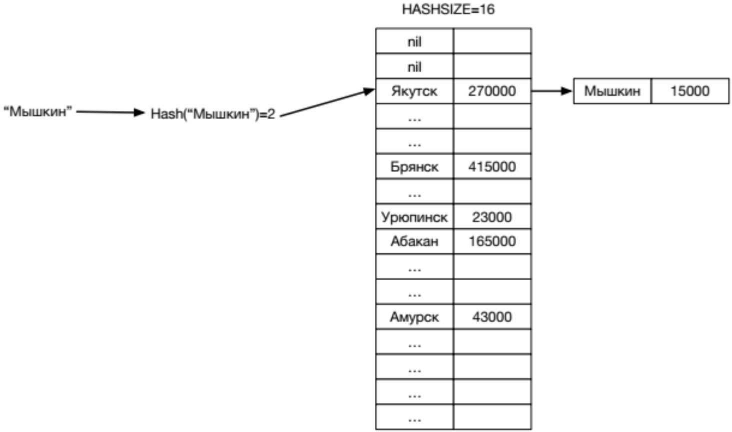
Добавление элементов (операция **insert**)

Добавление элементов (операция **insert**) требует поиска (операции **find**).

Если после нахождения значения хеш-функции от вновь прибывшего ключа оказывается, что запись с таким значением хеш-функции уже есть (например, Hash("Якутск") = 2 и Hash("Мышкин") = 2), то говорят, что произошла коллизия.

Хэш-таблицы с прямой адресацией

При коллизии во время создания элемента создаётся связный список конфликтующих.



Хэш-таблицы с прямой адресацией

Операция поиска и вставки

1. При поиске вычисляется значение хеш-функции от ключа.
2. По этому значению определяется место поиска — вторичная поисковая структура данных.
3. Если вторичной структуры нет, то нет и элемента, который мы ищем.Теперь, если это требуется, то создаётся вторичная структура данных и элемент вставляется в неё.
4. Иначе элемент ищется во вторичной структуре и вставляется туда при необходимости.

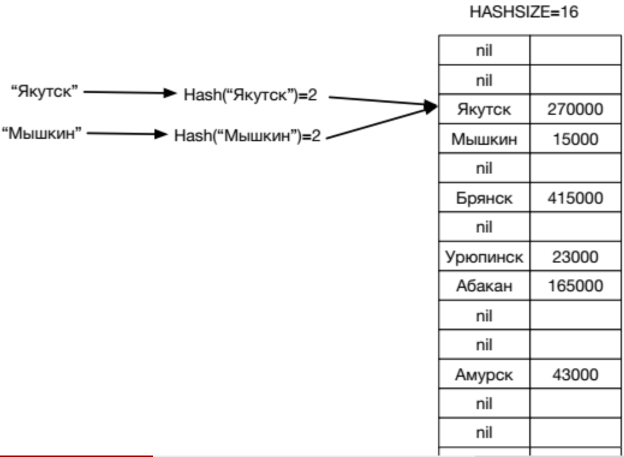
Хэш-таблицы с прямой адресацией

Операция удаления

1. При удалении вычисляется хеш-функция от ключа.
2. Определяется место нахождения ключа — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент удаляется из вторичной структуры.
5. Если вторичная структура пуста, удаляется сама структура и точка входа в неё.

Хэш-таблицы с открытой адресацией

При другой организации хеш-таблиц вторичные структуры данных не используются — и все пары ключ/значение хранятся в самой таблице. Это означает, что требуется удобный способ разрешения коллизий.



Хэш-таблицы с открытой адресацией

Операция поиска по ключу

1. При поиске существующего элемента вычисляется хеш-функция от его ключа.
2. По значению хеш-функции определяется место поиска — индекс в хештаблице.
3. Если по индексу ничего нет, то нет и элемента, алгоритм завершён.
4. Иначе по индексу находится элемент с нашим ключом (требуется операция сравнения ключей)— элемент найден.
5. Если по индексу находится элемент с другим ключом или элемент помечен удалённым, увеличиваем индекс на единицу (возвращаясь в начало таблицы при необходимости) и переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(\mathrm{index} + 1) \bmod M$.

Хэш-таблицы с открытой адресацией

Операция вставки по ключу

1. При вставке нового элемента вычисляется хеш-функция.
2. По значению хеш-функции определяется место поиска — индекс в хештаблице.
3. Если по индексу находится пустой элемент или имеется элемент, помеченный как удалённый, то мы нашли подходящее место — вставляем по индексу элемент.
4. Если по индексу уже присутствует элемент с искомым ключом — не трогая ключа, 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 $(\mathrm{index} + 1) \bmod M$.

Хэш-таблицы с открытой адресацией

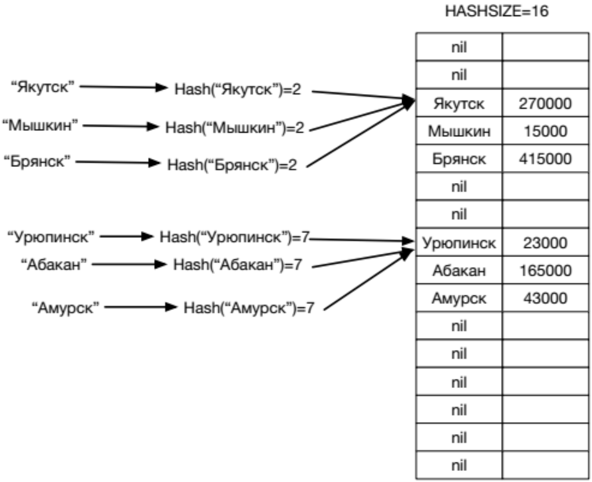


Рис.: Кластеризация коллизий

Хэш-таблицы с открытой адресацией

Операция удаления

1. При удалении вычисляется хеш-функция от ключа.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента.
4. Иначе, если по индексу расположен элемент с требуемым ключом, то элемент найден. Помечаем его удалённым и заканчиваем алгоритм.
5. Если по индексу расположен элемент с другим ключом, индекс увеличивается на единицу — и мы переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(\text{index} + 1) \bmod M$.

Хэш-таблицы с открытой адресацией

- Мы рассмотрели простой способ нахождения пустого элемента, увеличивая номер позиции-кандидата каждый раз на единицу, K = 1 (вспомните формулу (index+ 1) mod M).
- При не очень удачной хеш-функции в таблице быстро образуются кластера из элементов, ключи которых оказались в коллизии.
- Если мы обобщим эту формулу до (index+K) mod M, то, вроде бы, ничего не изменится. Но K можно сделать функцией от ключа, то есть K = H2(key).
- Оказывается, в этом случае кластеризация уменьшается, и коэффициент амортизации уменьшается вместе с ней. Это – **рехеширование**.