Лекция 6

Хеш-таблицы

Хеш-функция

Пусть K – множество всех значений данных

Пусть M – множество всех слотов (строк фиксированного размера)

Тогда h: K -> M – хеш-функция

Хеширование — это процесс, который преобразует любые данные в уникальную строку фиксированной длины.

Коллизия хеш-функции

Пусть заданы пространства K, M и хеш-функция h: K -> M.

Тогда коллизией элементов $A, B \in K$ является следующее:

$$A \neq B, \text{но } h(A) = h(B)$$

Идеальная хеш-функция (не содержит коллизий):

$$A \neq B \Rightarrow hash(A) \neq hash(B)$$

Примеры применения

- Проверка целостности сообщений и файлов
- Поиск дубликатов в последовательностях наборов данных
- Построение уникальных идентификаторов для наборов данных
- Криптография:
  - Верификация пароля
  - ЭЦП

Свойства

- Фиксированная длина хеша — независимо от входных данных;
- Вычислительная сложность – зависимость объёма работы алгоритма от размера входных данных (в идеале константа);
- Необратимость: для полученного значения хеш-функции h(X)=A невозможно найти исходное сообщение X;
- Детерминированность: если подать хеш-функции одинаковые данные, то и хеш у них будет одинаковым;
- Частота коллизий;
- Криптостойкость (не всегда нужна)

Криптостойкость

Чтобы хеш-функция h считалась криптографически стойкой, она должна иметь:

- Необратимость (криптографическая) – для заданного значения хеш-функции A должно быть вычислительно неосуществимо (тяжело) найти какой-то блок данных X, что h(X)=A;
- Стойкость к коллизиям первого рода: для заданного сообщения A должно быть вычислительно неосуществимо подобрать другое сообщение B, что h(A)=h(B);
- Стойкость к коллизиям второго рода: должно быть вычислительно неосуществимо подобрать пару сообщений A и B, имеющих одинаковый хеш h(A)=h(B);
- Чувствительность к изменениям — если изменить хоть один символ исходного сообщения, то хеш полностью поменяется.

Применение в СУБД

- Хеш-функция в СУБД может использоваться при поиске записей для вставки, доступа и удаления (не обязательно индекс, скорее даже редко индекс).
- **Bucket (бакет, корзина, сегмент)** – единица хранения, содержащая одну или более записей.
  - Мы получаем значение сегмента для записи из значения ключа поиска, используя **хеш-функцию**
- Могут быть записи с разными ключами поиска, но одним bucket; поэтому весь сегмент необходимо последовательно просматривать для поиска записи
- В хеш-индексе, сегмент хранит записи с указателями на записи файлов
- В хеш-файлах сегменты хранят записи
- В общем случае структуры данных, организованные с помощью хеш-функций, эффективно работают для точечных запросов, но не работают для запроса по диапазону

Хешированные файлы

- Хешированный файл – файл, в котором для вычисления адреса блока, в котором должна находиться та или иная запись, используется хеш-функция, аргументами которой являются значения одного или нескольких полей этой записи.
- Записи в хешированном файле произвольным образом распределены по всему доступному для файла пространству. По этой причине данные файлы иногда называют файлами с произвольной структурой или файлами прямого доступа.

Пример хеш-файлов

Организация файла в виде хешей для *instructor*, используя *dept_name* как ключ

- Всего 10 сегментов,
- Бинарное представление *i-го* символа предполагается целым числом *i.*
- Хеш функция:
  - h(«Music») = 1
  - h(«History») = 2
  - h(«Physics») = 3
  - h(«Elec. Eng.») = 3

Пример хеш-файлов

Хэш файл *instructor*,
dept_name - ключ.

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

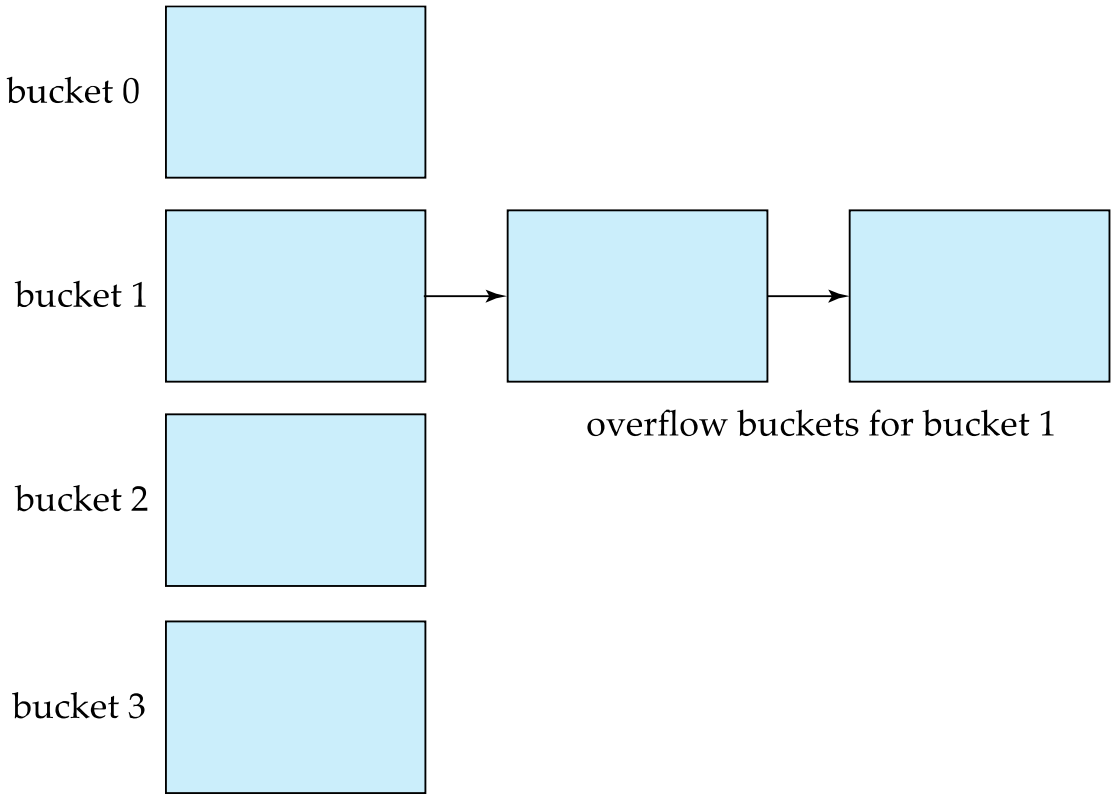
bucket 7

Управление переполнением сегментов

- Переполнение сегментов может возникнуть из-за
  - Недостаточного числа сегментов
  - Смещении в распределении записей
    - Много записей имеют одно и то же значение ключа поиска
    - Выбранная хэш функция создает неравномерное распределение значений ключа
- Хотя вероятность переполнения корзины может быть уменьшена, ее нельзя устранить; это обрабатывается с помощью **сегментов *переполнения*** (overflow buckets).

Управление переполнением сегментов

- Цепочка переполнений (overflow chaining) – сегменты переполнения определенного сегмента объединены в связанный список.



Хеш-таблица

Хэш-таблица - это структура данных для хранения пар «ключ» – «значение». Доступ к элементам осуществляется по ключу.

Для вычисления смещения в массиве заданного ключа используется хеш-функция.

- Хеш-таблицы позволяют в среднем за время O(1) выполнять добавление, поиск и удаление элементов
- Сложность в памяти O(n)

Хеш-таблица

- Хеш-функция
  - Как построить отображение с большого пространства ключей на небольшое пространство
  - Необходимо найти консенсус между быстротой и количеством коллизий
- Механизм обработки коллизий
  - Баланс между большим объектом в памяти и дополнительными инструкциями для нахождения/добавления ключа

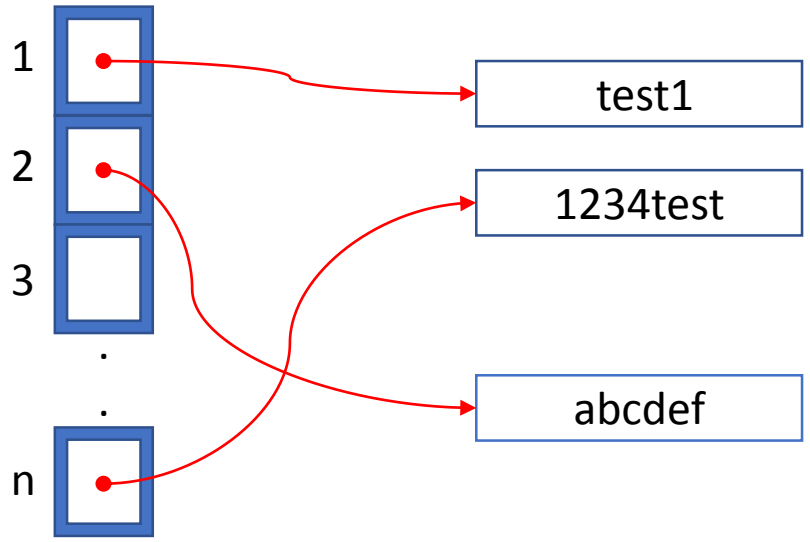
Хеш-таблица

- Статическая хеш-схема
- Динамическая хеш-схема

Статическая хеш-таблица

Происходит аллоцирование большого массива, в котором есть один слот для каждого элемента, который вам необходимо хранить.

Для нахождения записи применяется хеш-функция, определяющая смещение, необходимое для получения указателя, на ключ.



Хеш-функция

- Для каждого ключа (множества атрибутов), возвращается некое целочисленное число.
- Примеры хеш-функций: MD5, SHA-3, MurmurHash2
- Для внутренних хеш-таблиц СУБД не требуется криптографическая хеш-функция, а необходимо довольно быстрое решение с минимизацией числа коллизий.

Хеш-функции

- CRC-64 (1975) – Cyclic Redundancy Check
- MurmurHash - семейство хеш-функций общего назначения
- City Hash (разработан в Google 2011 году)
- xxHash (разработан в Facebook в 2012 году)

MurmurHash2 (для примера)

Инициализация

(Java)

Входные параметры: key, len, seed

```
m = 0x5bd1e995;
r = 24;
h = seed ^ len;
data = key;
k = 0;
```

Возвращает 32-разрядное беззнаковое число.

Шаг 1

```
while (len >= 4)
{
    k  = data[0];
    k |= data[1] << 8;
    k |= data[2] << 16;
    k |= data[3] << 24;

    k *= m;
    k ^= k >> r;
    k *= m;

    h *= m;
    h ^= k;

    data += 4;
    len -= 4;
}
```

Шаг 2

```
switch (len)
 {
  case 3:
    h ^= data[2] << 16;
  case 2:
    h ^= data[1] << 8;
  case 1:
    h ^= data[0];
    h *= m;
 };
```

Шаг 3

```
h ^= h >> 13;
 h *= m;
 h ^= h >> 15;

 return h;
```

Типы разрешения коллизий в хеш-таблице

- Открытое хеширование
  - Для разрешения коллизий для каждого слота используется отдельная структура данных (связный список, сбалансированное дерево), по которому приходится пройти, чтобы получить значение.
  - Минус открытого хеширования – довольно затратное удаление
- Закрытое хеширование или открытая адресация
  - Для разрешения коллизий, если адрес слота уже содержит значение, используется алгоритм поиска свободного места внутри хеш-таблицы

Схемы статического хеширования

- Linear Probe Hashing
- Robin Hood hashing
- Кукушкино хеширование

Линейное зондирование

Общая таблица слотов

Разрешение коллизий осуществляется линейным поиском для следующего свободного слота в таблице

- Для определения элемента, применяется хеш-функция для поиска индекса, затем происходит поиск.
- Необходимо понимать, когда остановиться для поиска
- Вставки и удаление – обобщение поисков

Линейное зондирование (историческая справка)

Было разработано в 1954 году Джином Амдалом, Элейном Макгроу и Артуром Сэмуэлем, в 1963 году Кнут Дональд провел анализ сложности.

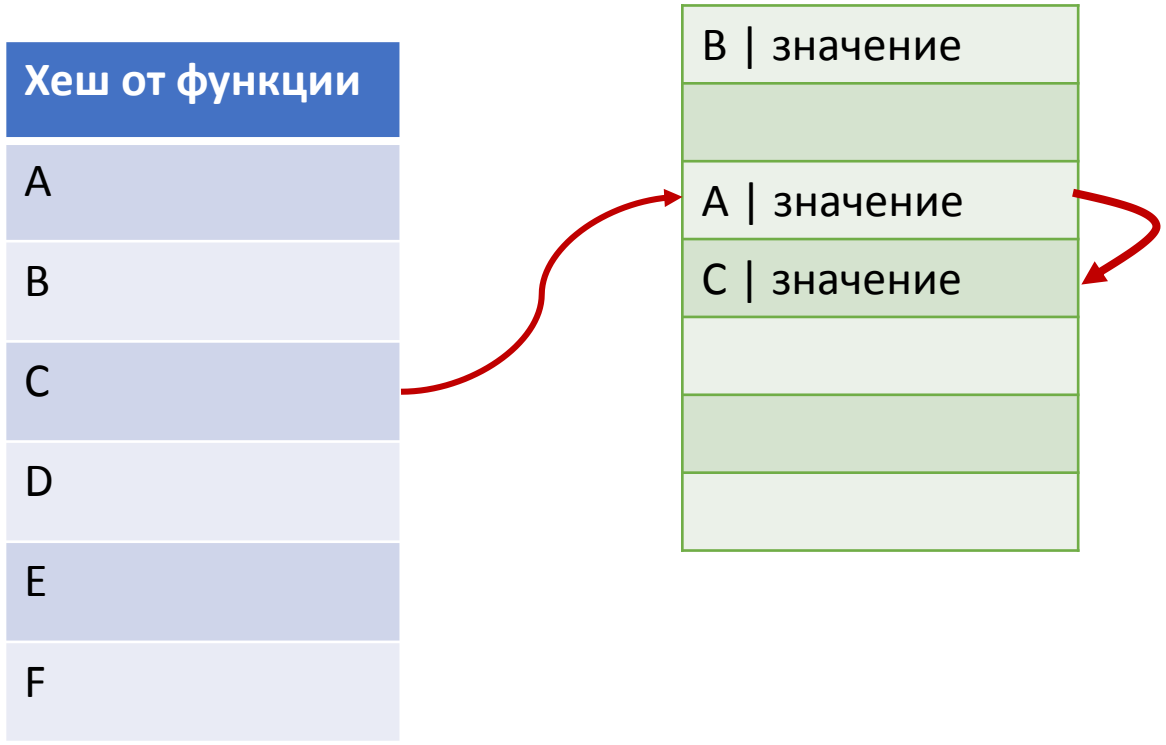
Является формой открытого хеширования. В данной схеме каждая ячейка хеш-таблицы хранит пару ключ-значение.

Очень много имплементаций хеш-таблиц используют именно линейное зондирование.

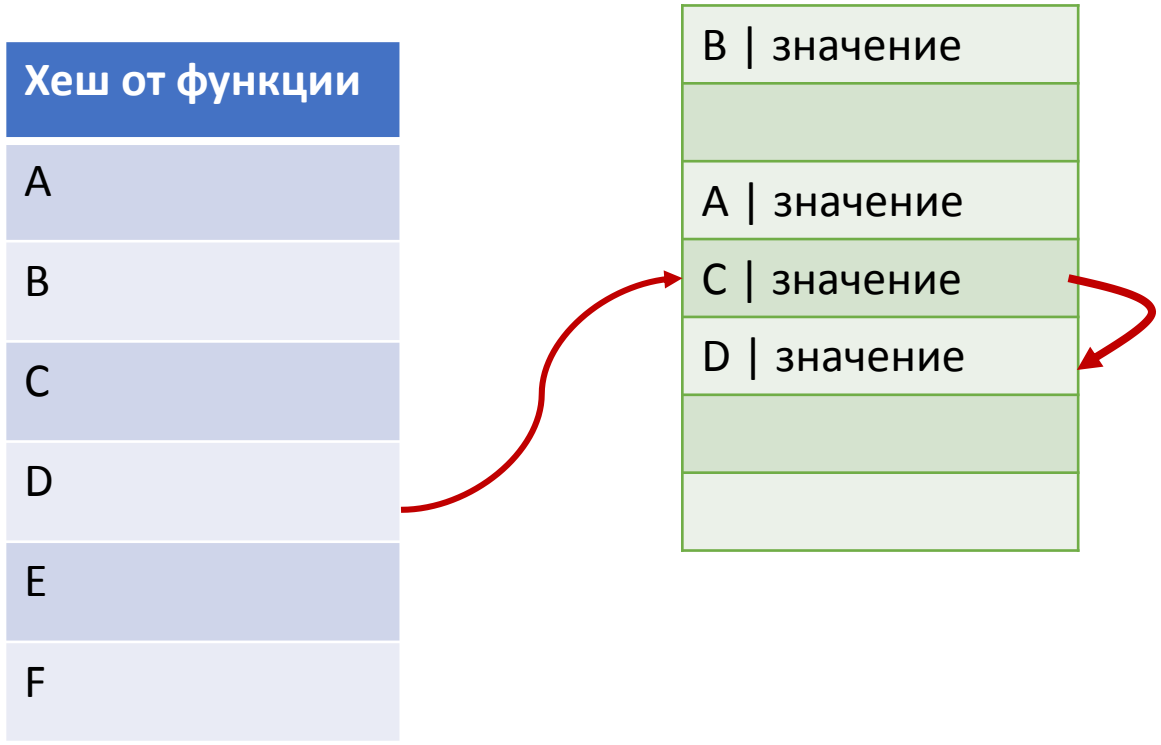
Линейное зондирование

Линейное зондирование

Линейное зондирование

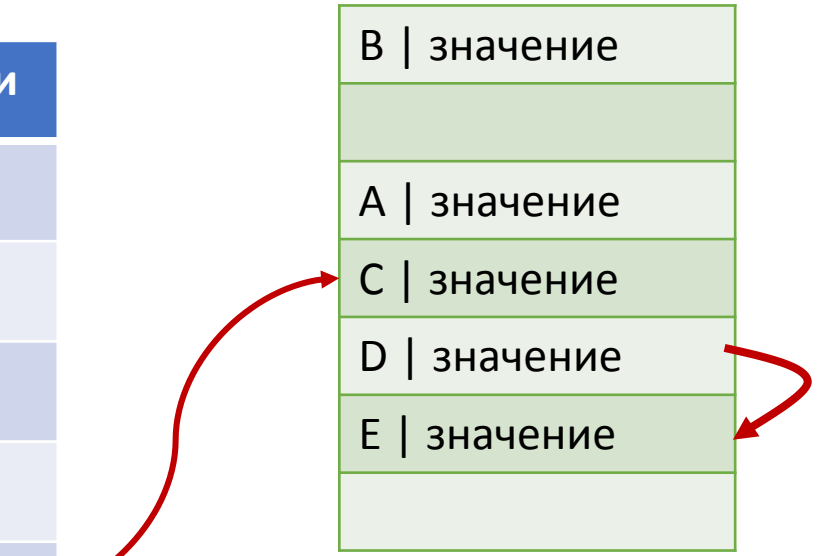


Линейное зондирование



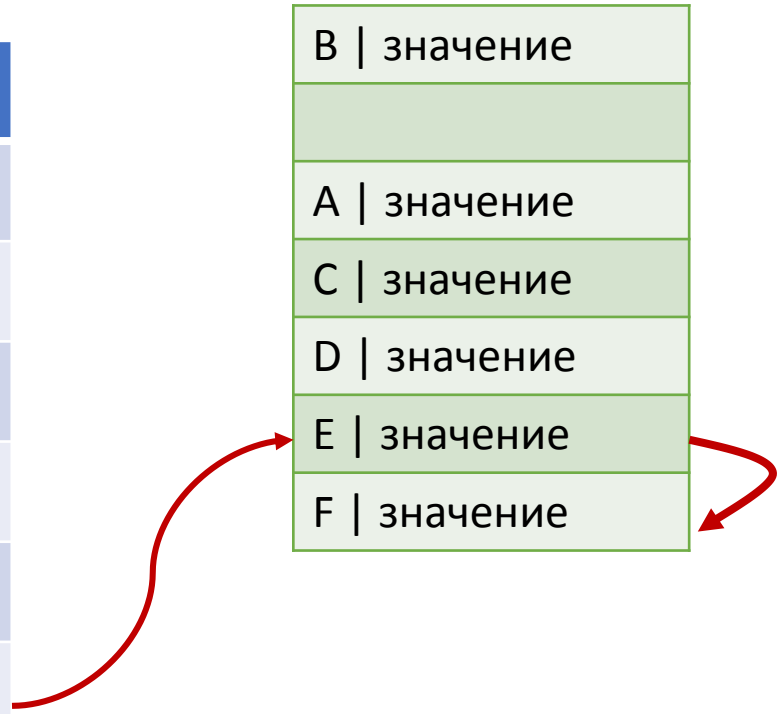
Линейное зондирование

Хеш от функции
A
B
C
D
E
F

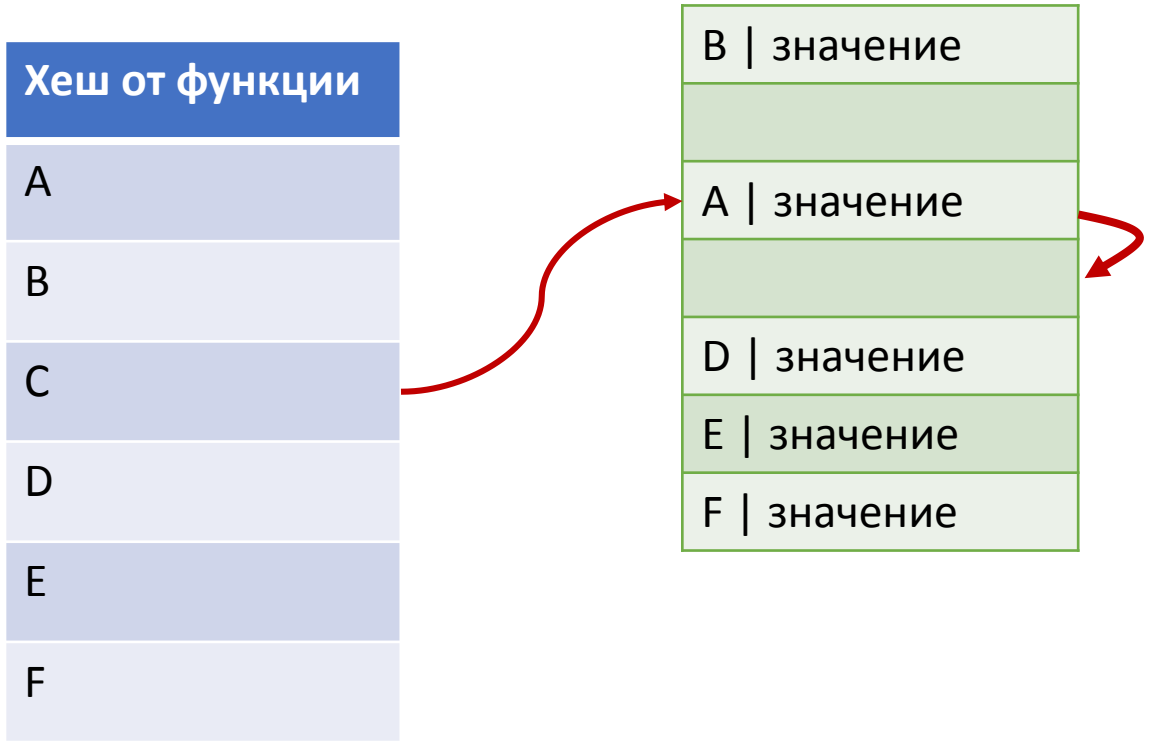


Линейное зондирование

Хеш от функции
A
B
C
D
E
F

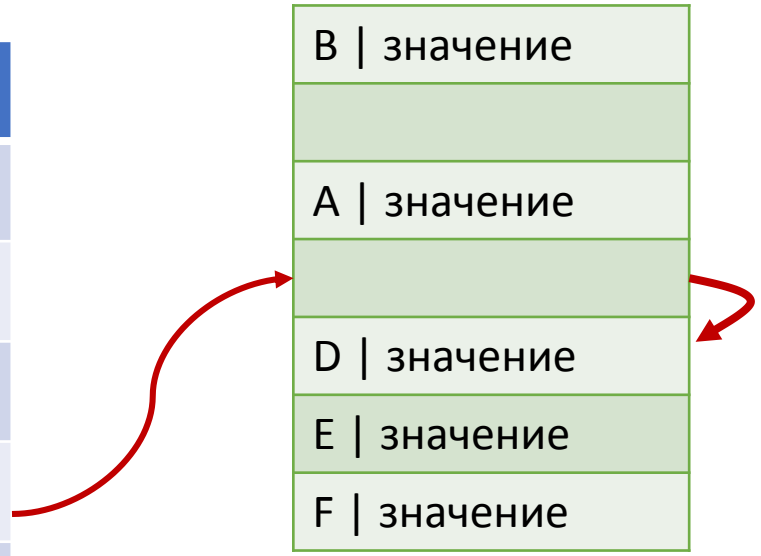


Линейное зондирование (Удаление)



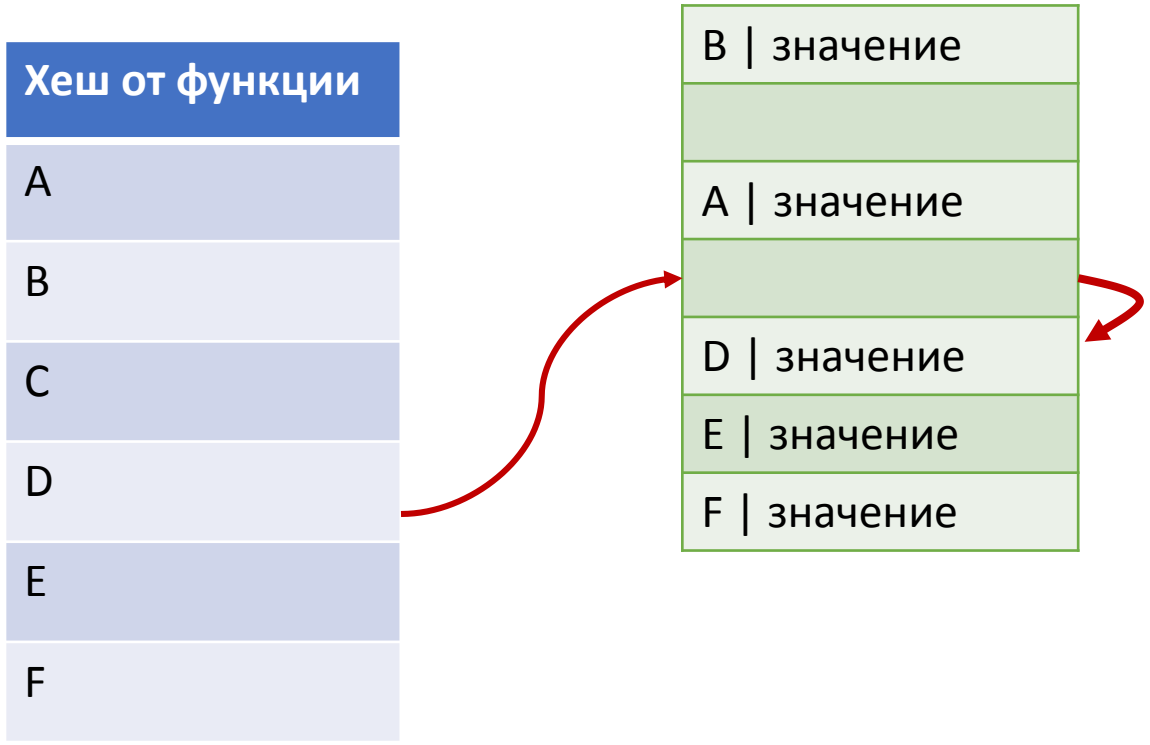
Линейное зондирование (Удаление)

Хеш от функции
A
B
C
D
E
F



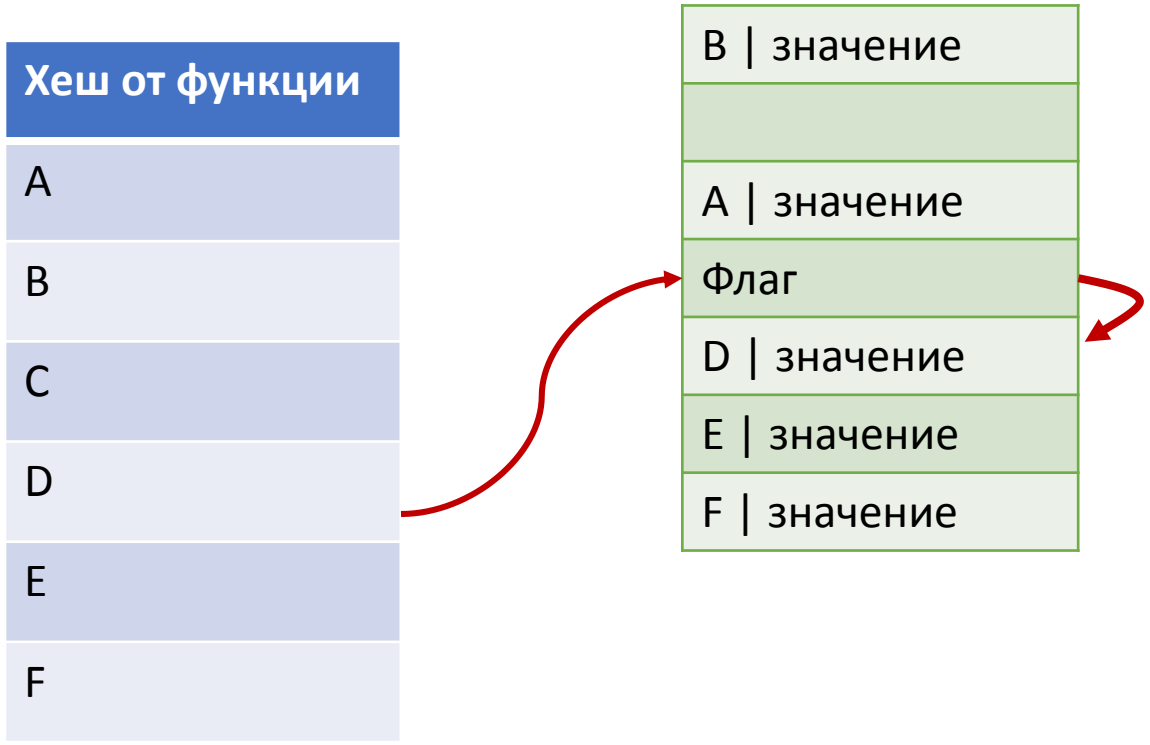
Как понять это отсутствие значений, или же удаленное значение

Линейное зондирование (Удаление)



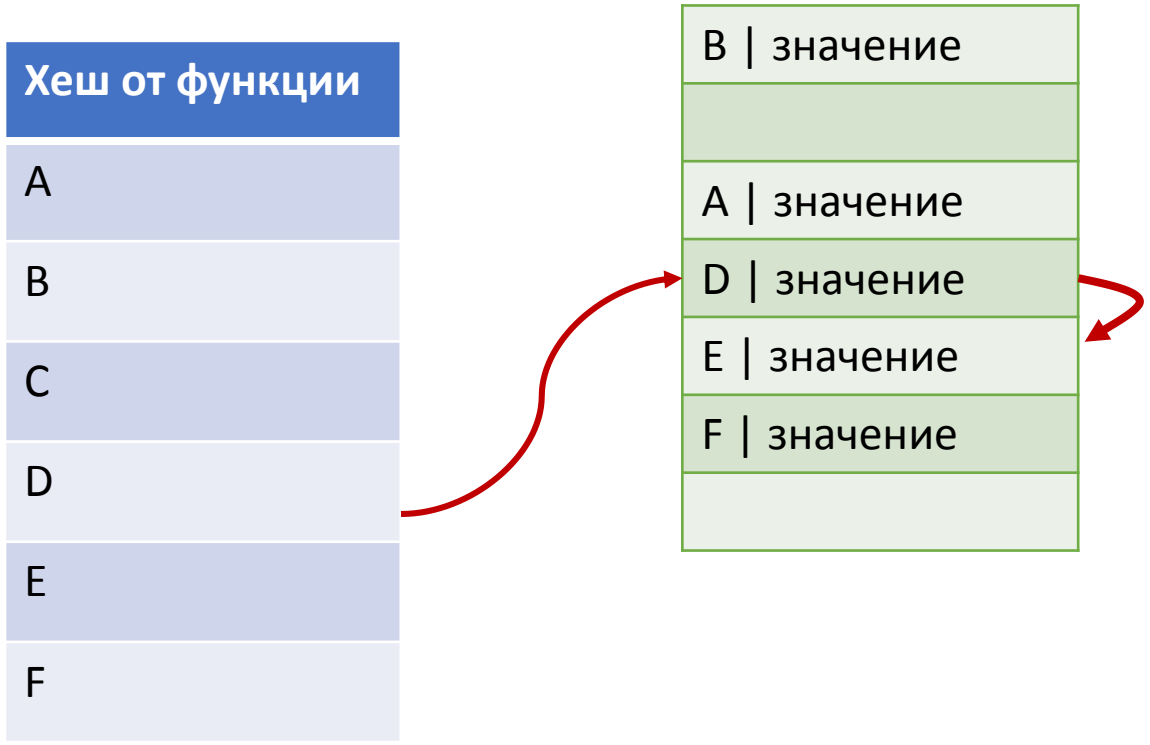
Вариант 1: специальный флаг
Вариант 2: перемещение

Линейное зондирование (Удаление)



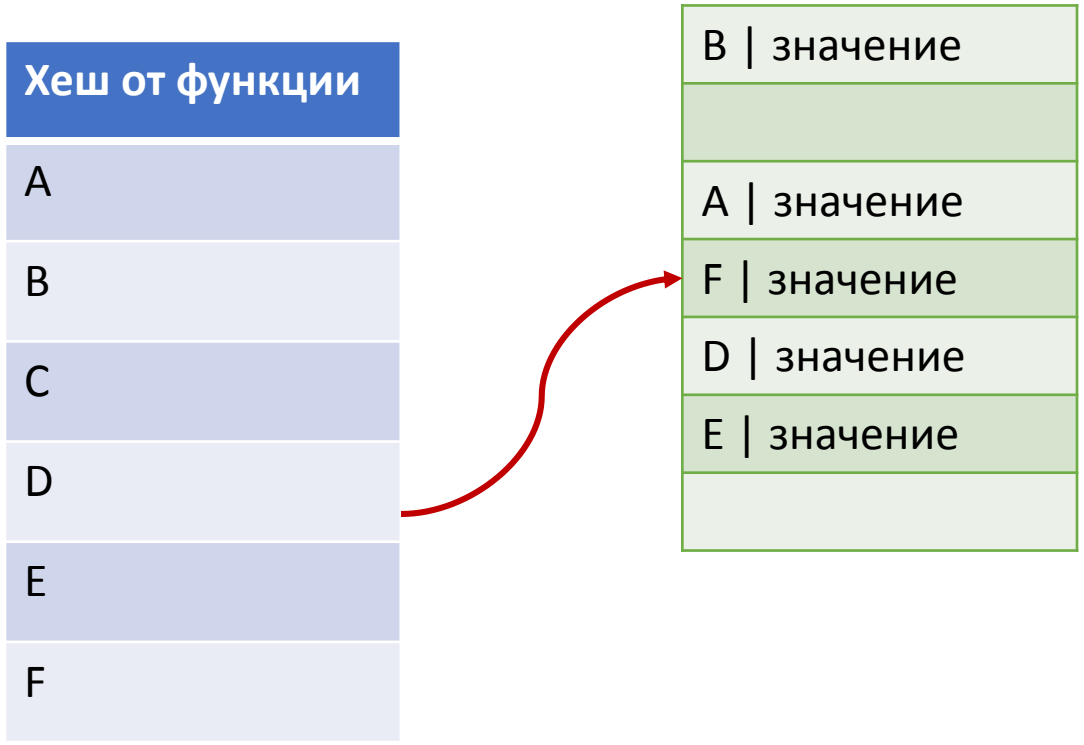
Вариант 1: специальный флаг
Вариант 2: перемещение

Линейное зондирование (Удаление)



Вариант 1: специальный флаг
Вариант 2: перемещение

Линейное зондирование (Удаление)



Вариант 1: специальный флаг
Вариант 2: перемещение

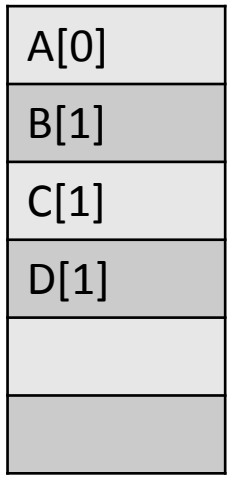
Robin Hood Hashing

- Первая реализация была в 1986 году
- P. Celis. Robin Hood Hashing. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1986.
- Переработанная версия была в Rust collections
- Алгоритм крадет у «богатых», и передает «бедным». За богатые принимаются удачные величины, которые нашли слот близко к реальному значению хеш-функции

Robin Hood Hashing

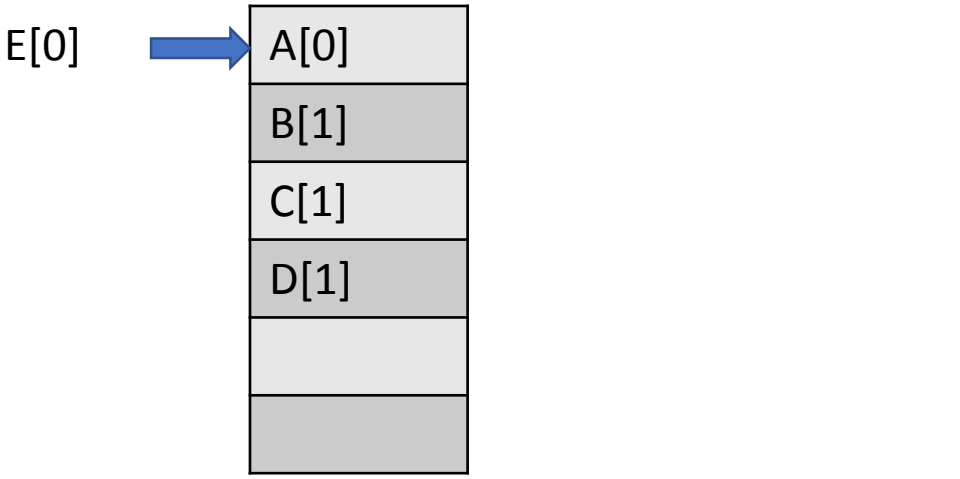
- Введем понятие DFB (Distance From expected Bucket) – расстояние от ожидаемого слота.
- Алгоритм Robin Hood решает проблему неравенства в момент вставки, осуществляя процедуру сдвига уже существующих вставок.
- Когда элемент алгоритма имеет больший DFB, чем элемент, занявший слот, то происходит замена одного элемента на другой, и продолжается поиск свободного слота.

Robin Hood Hashing. Вставка



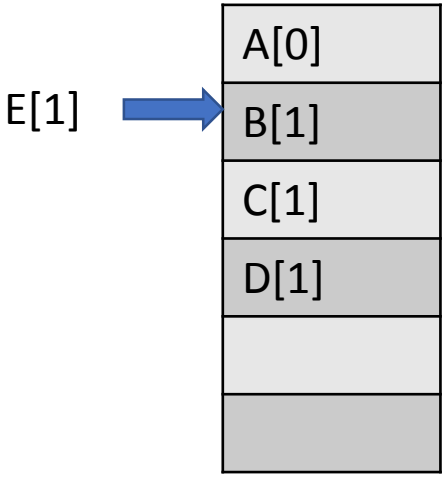
В скобках указано DFB

Robin Hood Hashing. Вставка



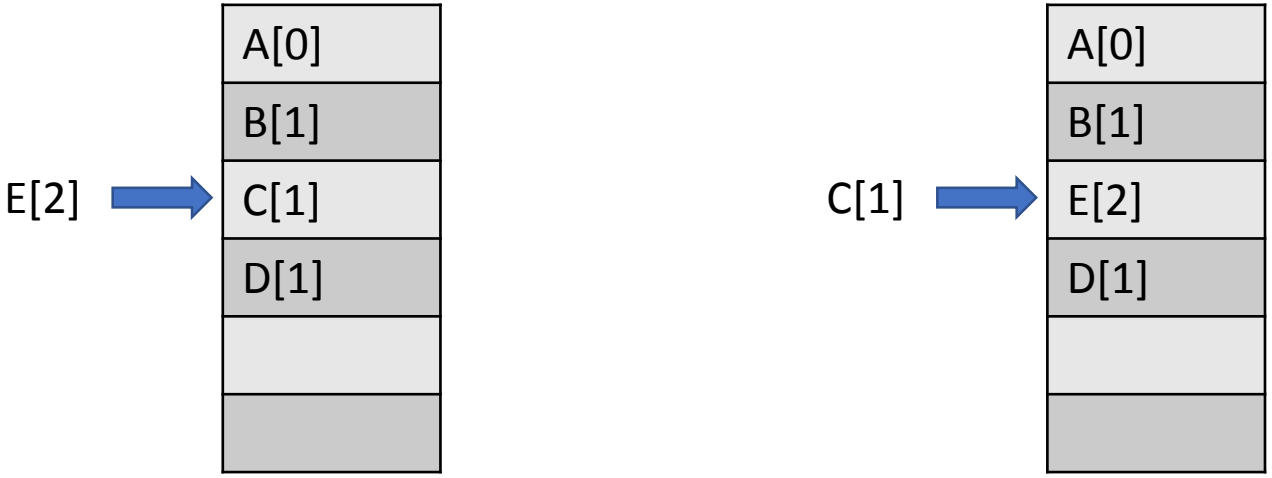
В скобках указано DFB

Robin Hood Hashing. Вставка



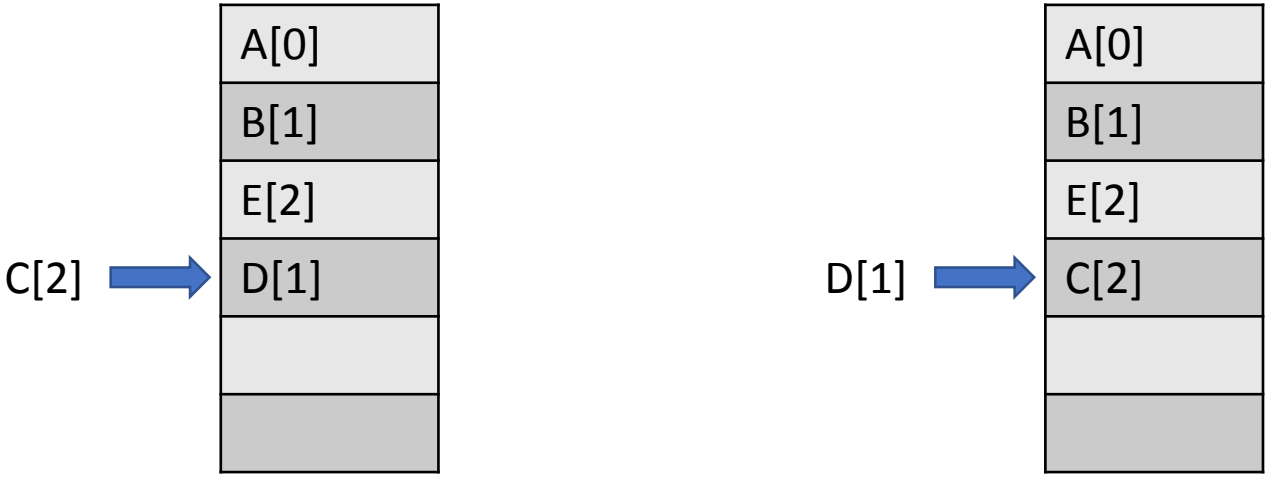
В скобках указано DFB

Robin Hood Hashing. Вставка



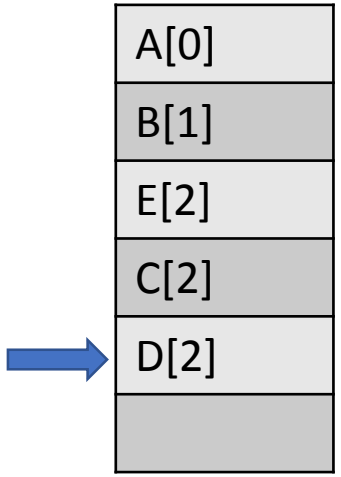
В скобках указано DFB

Robin Hood Hashing. Вставка



В скобках указано DFB

Robin Hood Hashing. Вставка



В скобках указано DFB

Robin Hood Hashing. Вставка

Разница с линейным зондированием

Robin Hood

A[0]
B[1]
E[2]
C[2]
D[2]

Линейное зондирование

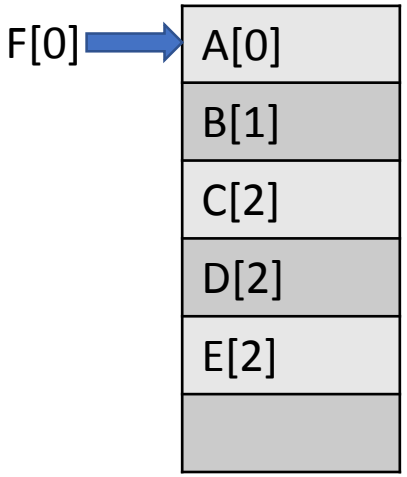
A[0]
B[1]
C[1]
D[1]
E[4]

В скобках указано DFB

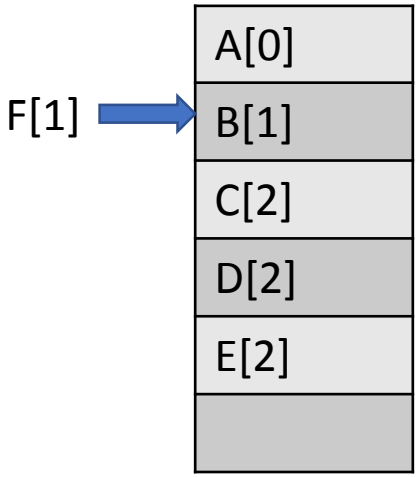
Robin Hood Hashing

- Результатом применения алгоритма Robin Hood Hashing является уменьшенная дисперсия, что позволяет заканчивать поиск горазд быстрее, чем нахождение пустого слота (как например, при линейном зондировании)
- Если элемент содержит меньший DFB, чем текущий DFB в слоте, то можно сделать вывод, что данный элемент отсутствует в хеш-таблице.

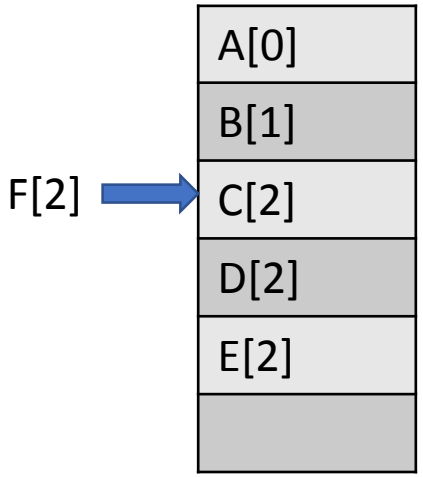
Robin Hood Hashing. Поиск элемента



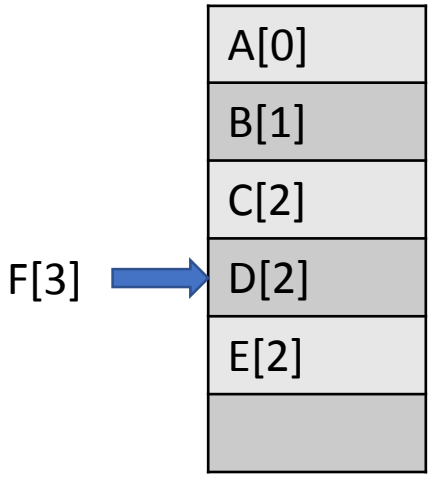
Robin Hood Hashing. Поиск элемента



Robin Hood Hashing. Поиск элемента



Robin Hood Hashing. Поиск элемента



3 > 2 => Поиск окончен

- Такой поисковый механизм называется инвариантом алгоритма Robin Hood.
- Было подсчитано, что для успешного поиска в среднем нужно всего 2.6 слота, и O(ln n) для неуспешного

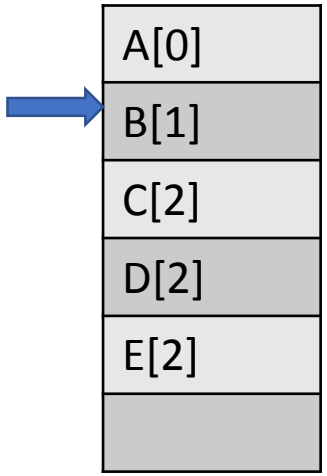
Robin Hood Hashing. Удаление

Удаление в алгоритме RH сложнее, чем при линейном зондировании. Так как в RH важно DFB элемента слота, то нельзя просто логически пометить, что элемент в слоте был удален.

Остаётся смещать остальные слоты вверх до момента нахождения пустого значения или элемента со значением DFB = 0.

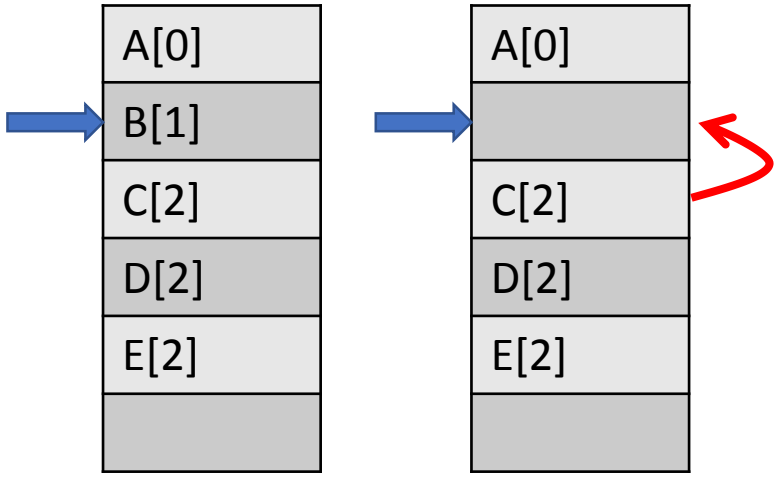
Robin Hood Hashing. Удаление

Необходимо удалить B



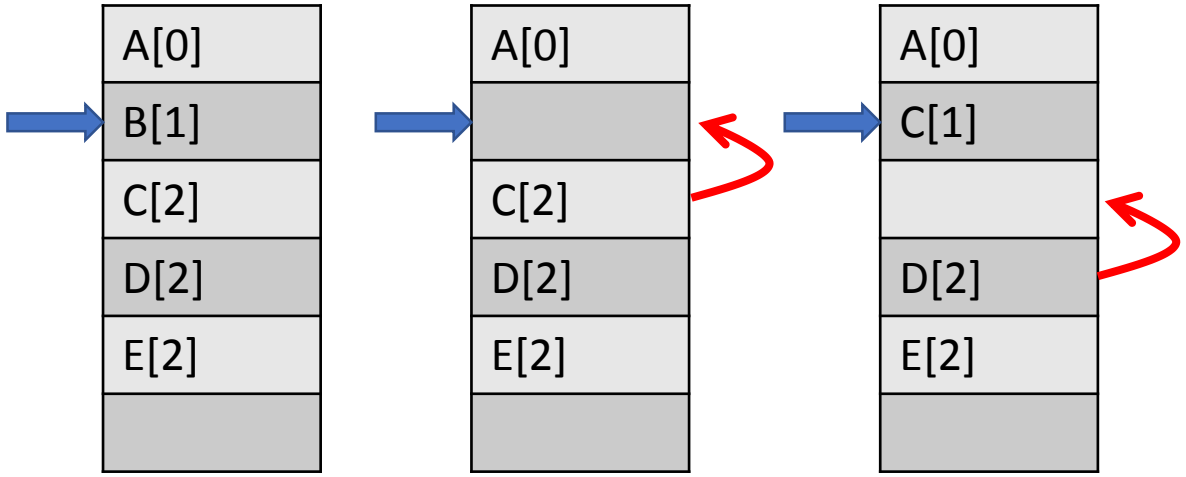
Robin Hood Hashing. Удаление

Необходимо удалить B



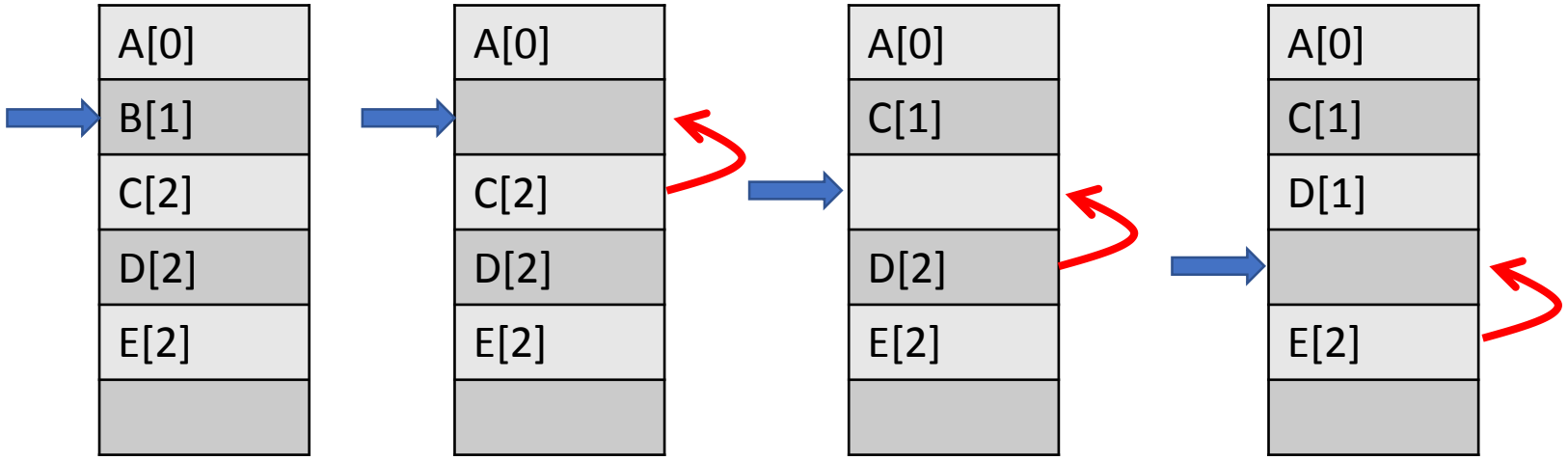
Robin Hood Hashing. Удаление

Необходимо удалить B



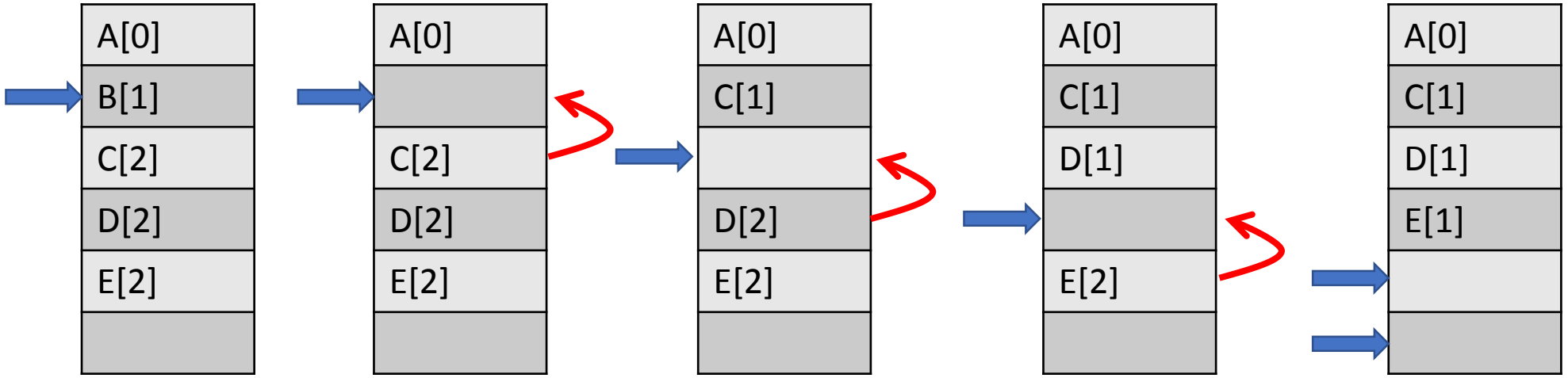
Robin Hood Hashing. Удаление

Необходимо удалить B



Robin Hood Hashing. Удаление

Необходимо удалить B



Кукушкино хеширование

- Первая версия появилась в 2001 году. *Rasmus Pagh, Flemming Friche Rodler.* Cuckoo Hashing
- Базовый вариант содержит в себе 2 таблицы T_1 и T_2, каждая из которых содержит m элементов.
- Выбираются 2 разные хеш-функции $h_1(x)$ и $h_2(x)$ для каждой из таблиц
- Каждый элемент x находится на позиции $h_1(x)$ или $h_2(x)$

Кукушкино хеширование (поиск)

- Каждый ключ x хранится либо в T_1, либо в T_2 , но не может хранится и там, и там одновременно
- Время поиска O(1)

Кукушкино хеширование (удаление)

- Каждый ключ x хранится либо в T_1, либо в T_2 , но не может хранится и там, и там одновременно. Соответственно, необходимо удалить ключ.
- Время удаления O(1)

Кукушкино хеширование (вставка)

- Для вставки элемента x, осуществляется попытка вставки в T_1
- Если $h_1(x)$ пусто, то осуществляется вставка туда
- Если $h_1(x)$, не пусто, то элемент y извлекается из $h_1(x)$. И для него осуществляется попытка вставки в T_2.
- Данный процесс повторяется, переходя от таблице к таблице, до стабилизации всех элементов

Кукушкино хеширование (вставка)

- Существует вероятность получения цикла
- В таком случае происходит рехеширование с новым выбором функций h_1 и h_2, осуществляется вставка всех элементов
- До успеха могут потребоваться несколько рехешированый

Статические хеш-таблицы. Ограничение

Если место в хеш-таблице заканчивается, обычно увеличиваются размер в 2 раза, и полностью пересчитывается хеш-таблица иногда уже с новой хеш-функцией.

Недостатки статического хеширования

- При статическом хешировании функция h отображает значения ключа поиска в фиксированный набор B адресов сегментов. Базы данных растут или уменьшаются со временем.
- Если начальное количество сегментов слишком мало, а размер файла увеличивается, производительность будет снижаться из-за слишком большого количества переполнений.

Недостатки статического хеширования

- Одно из решений: периодическая реорганизация файла с новой хэш-функцией
  - Дорого, нарушает нормальную работу
- Более правильное решение: разрешить динамическое изменение количества сегментов.

Динамическое хеширование

- Периодическое рехеширование
  - Если количество записей в хеш-таблице становится (скажем) в 1,5 раза больше хеш-таблицы,
    - создать новую хеш-таблицу размером (скажем) в 2 раза больше предыдущей хеш-таблицы
    - Рехешировать все записи в новую таблицу
- Линейное хеширование
  - Рехешировать инкрементально
- Расширяемое хеширование
  - Предназначен для хэширования на основе диска, с сегментами, совместно используемыми несколькими хэш-значениями
  - Удвоение количества записей в хэш-таблице без удвоения количества блоков

Динамические хеш-таблицы

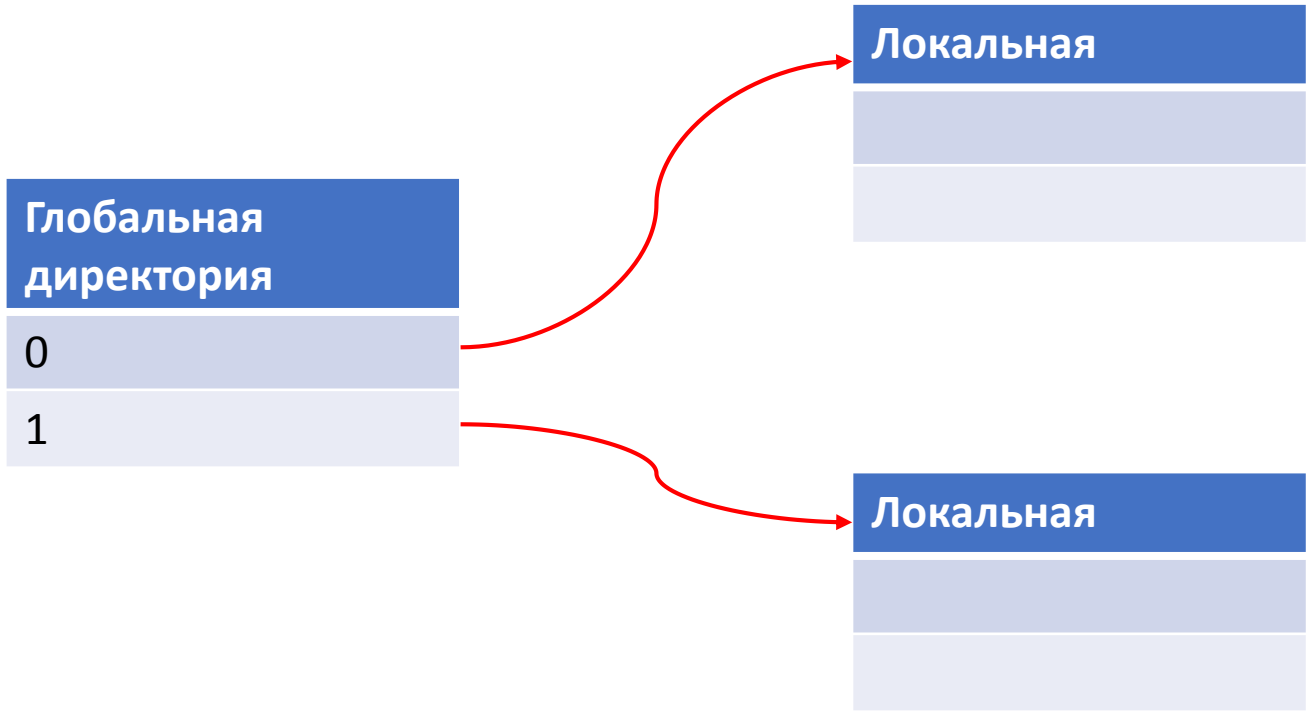
- Под динамической хеш-таблицей подразумевается возможность изменения количества слотов при операциях вставки или удаления
- Динамические хеш-таблицы самостоятельно осуществляют дополнительное выделение памяти (или наоборот, обратный забор)

Динамические хеш-таблицы

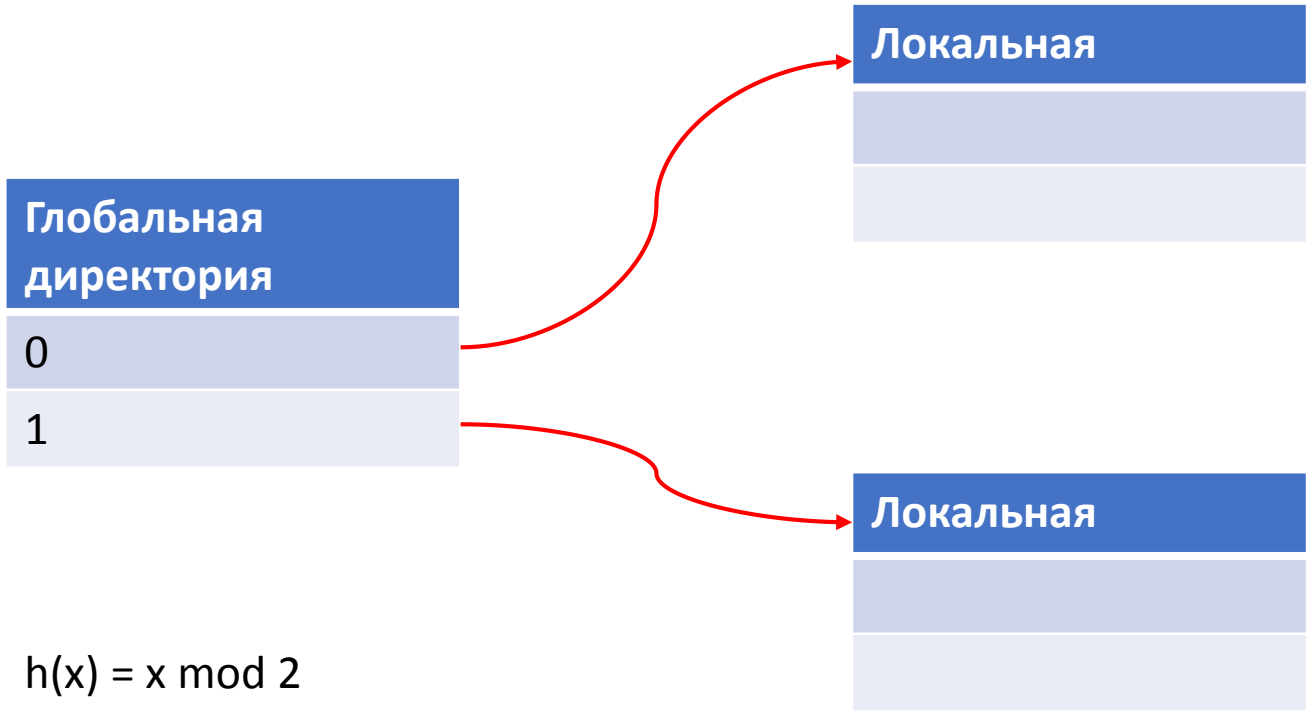
Динамические хеш-таблицы подразделяются на 2 семейства:

- С использованием структуры вида Директория
  - Расширяемое хеширование
- Без использования структуры вида Директория
  - Линейное хеширование

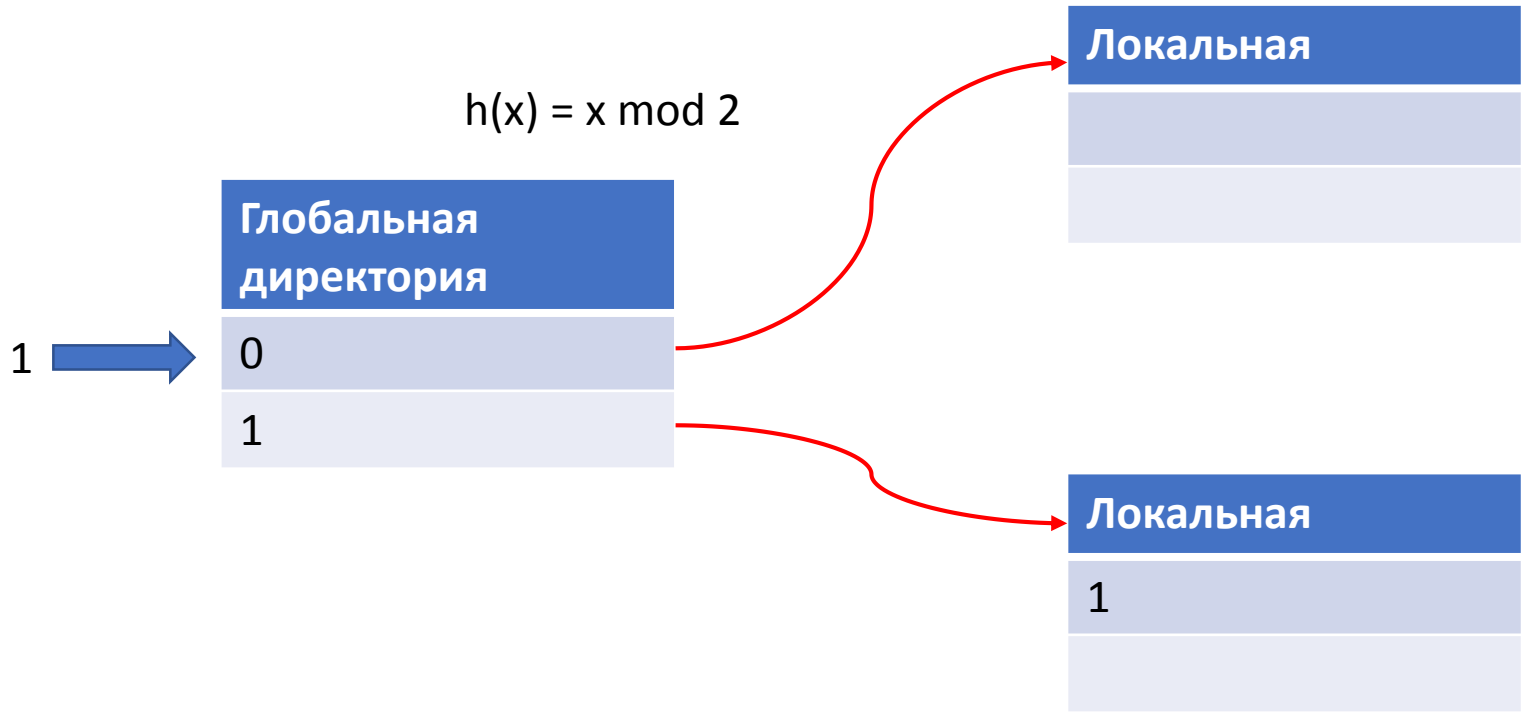
Расширяемое хеширование



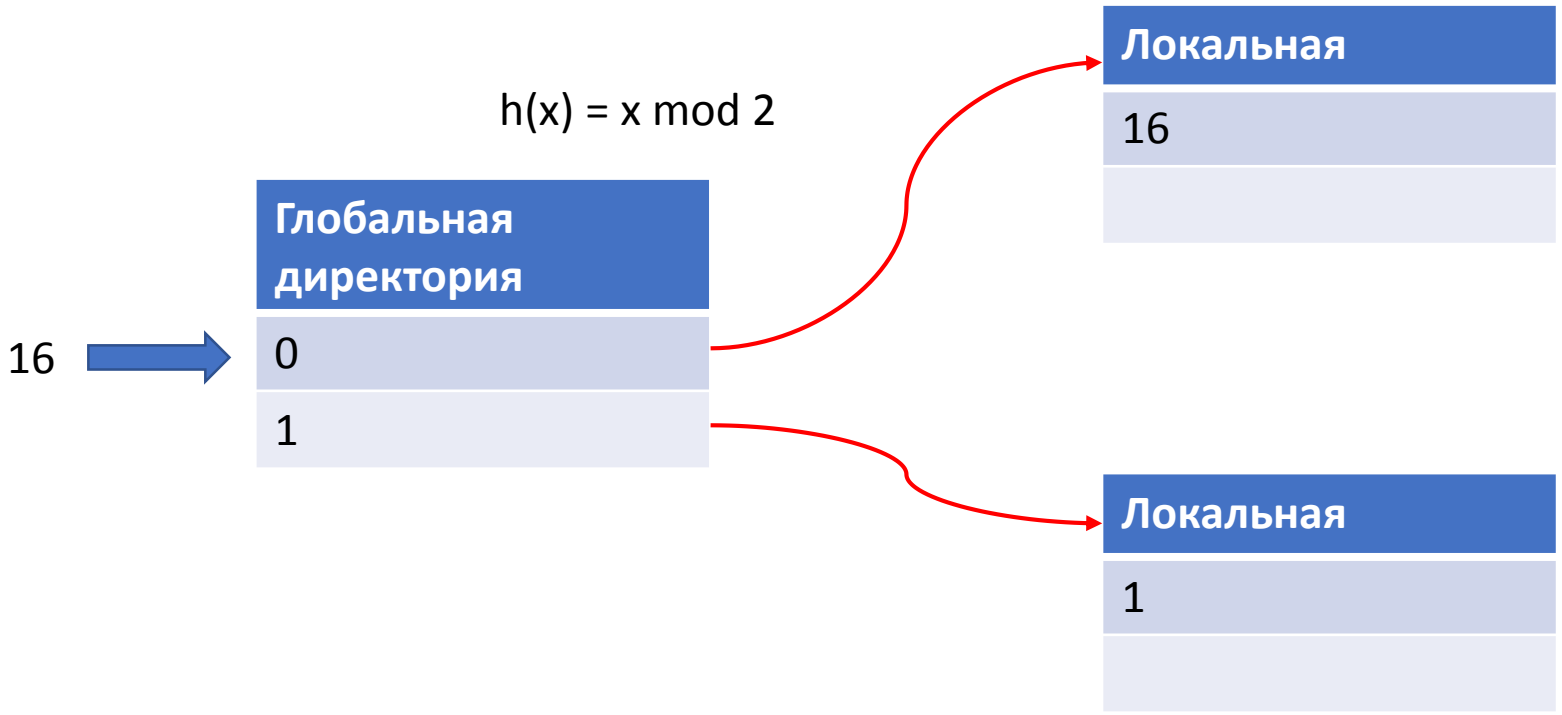
Расширяемое хеширование



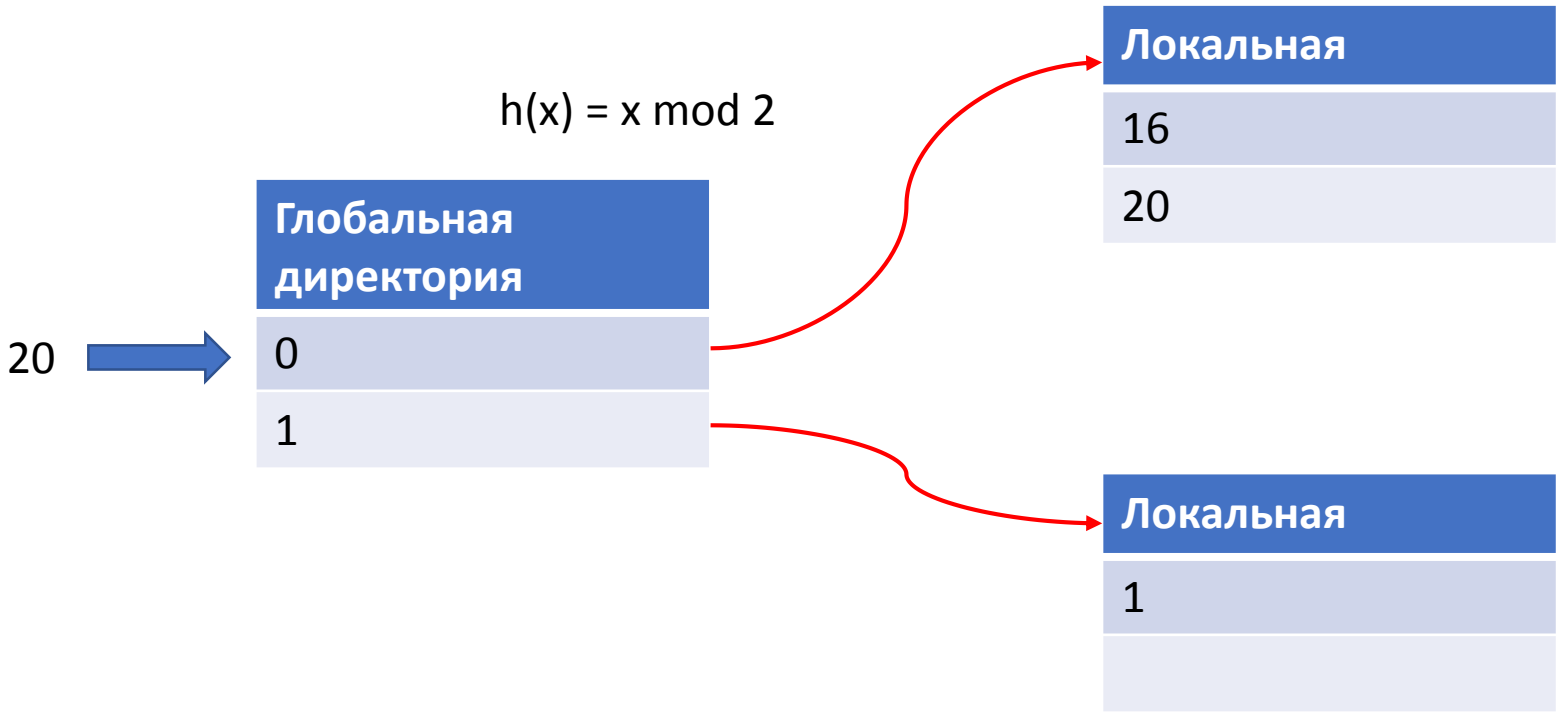
Расширяемое хеширование



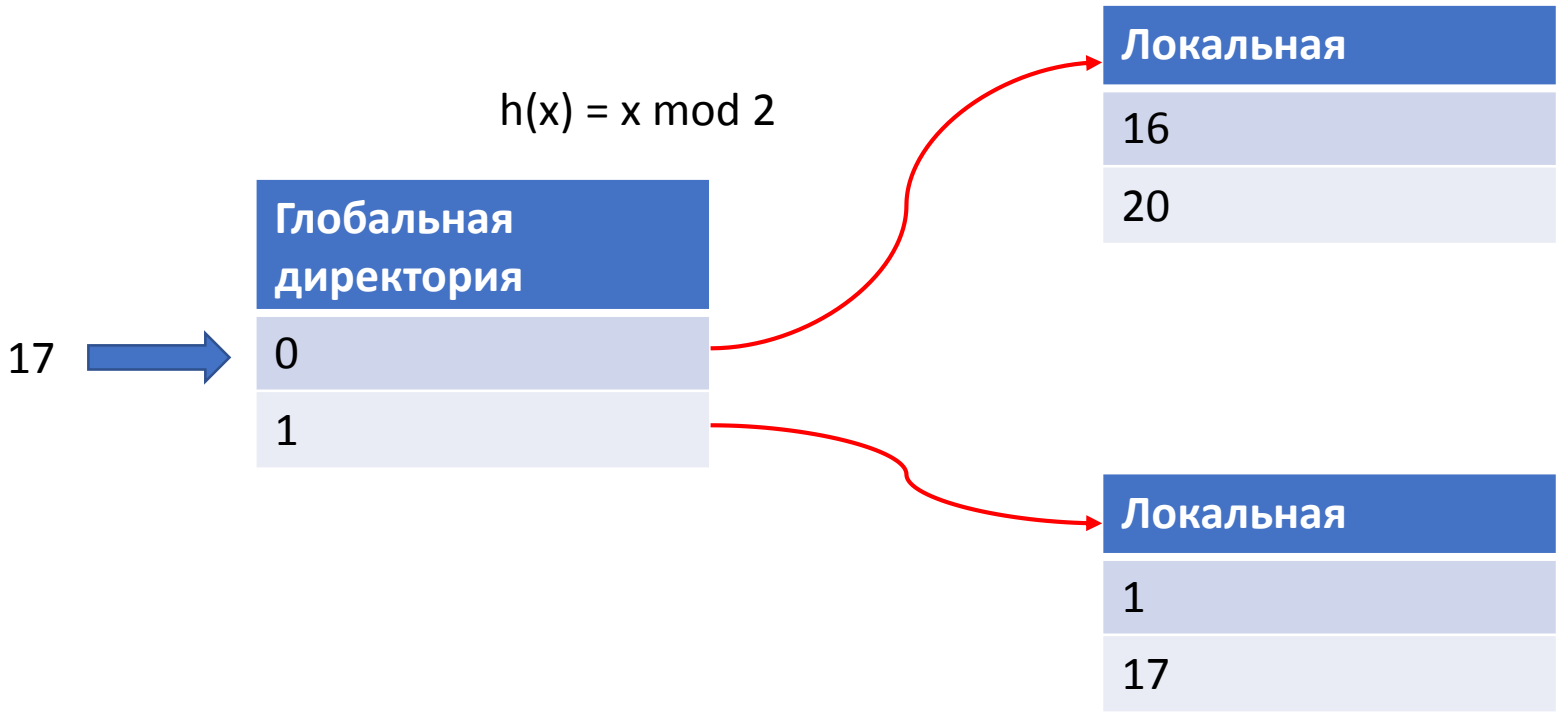
Расширяемое хеширование



Расширяемое хеширование



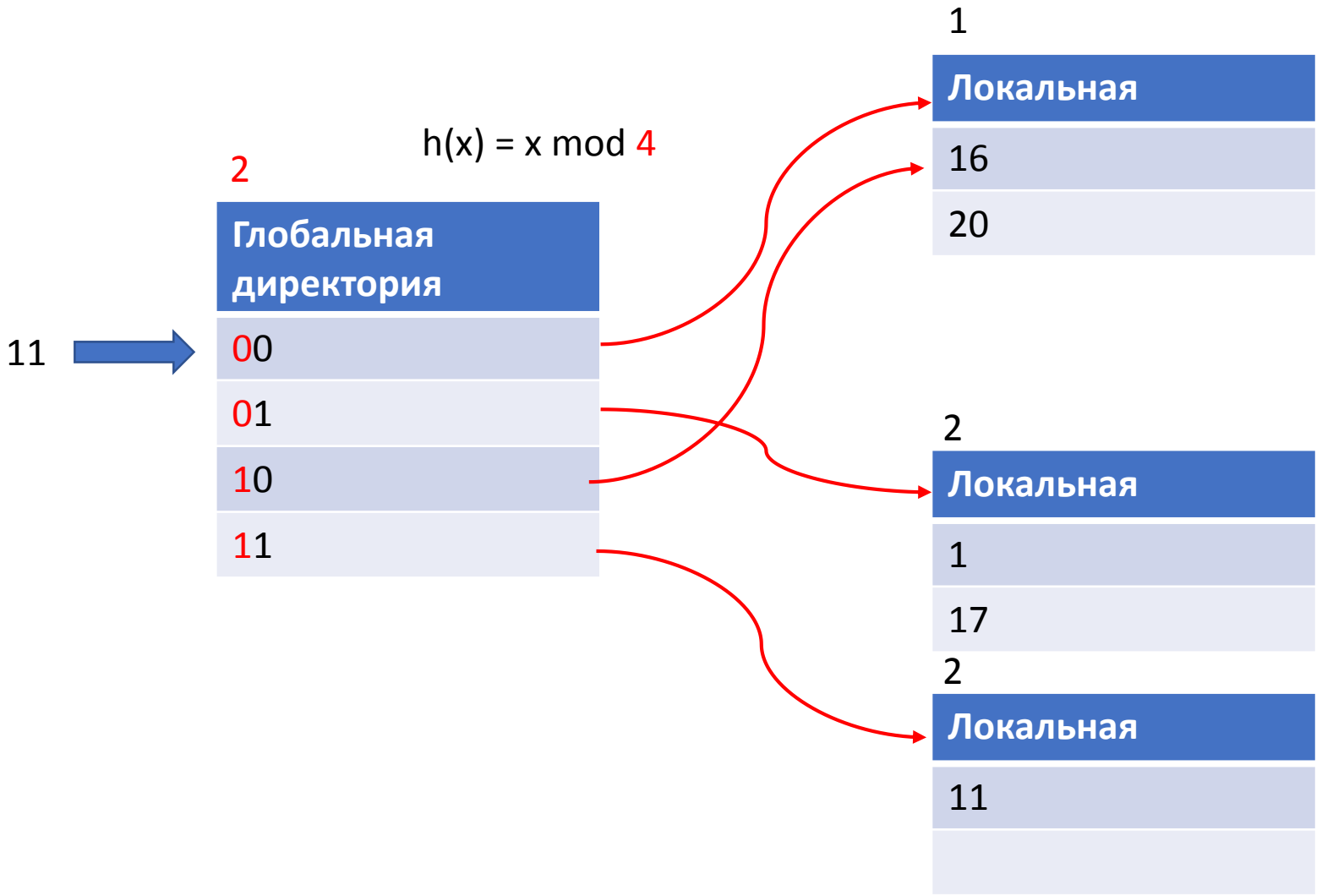
Расширяемое хеширование



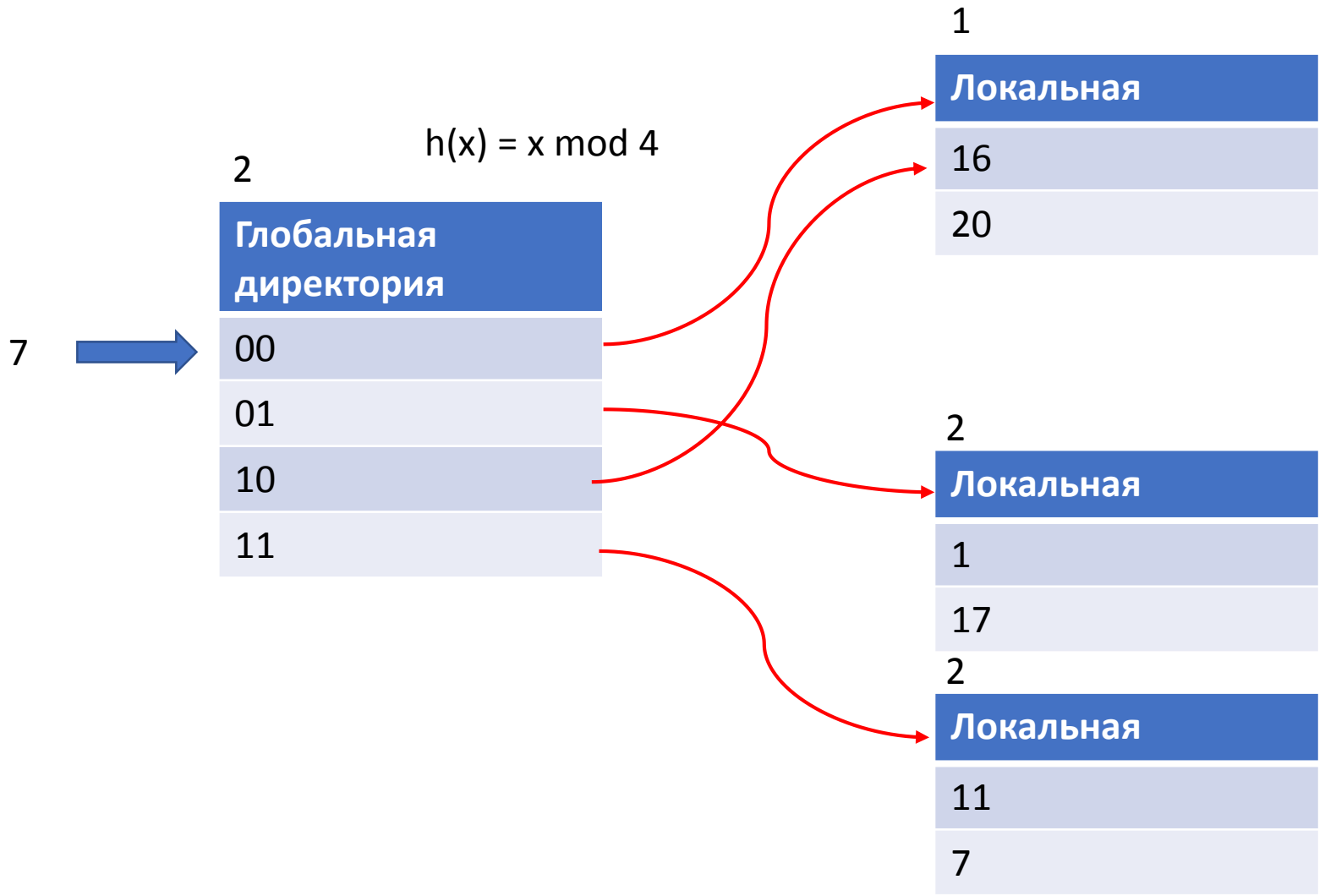
Расширяемое хеширование

Расширяемое хеширование

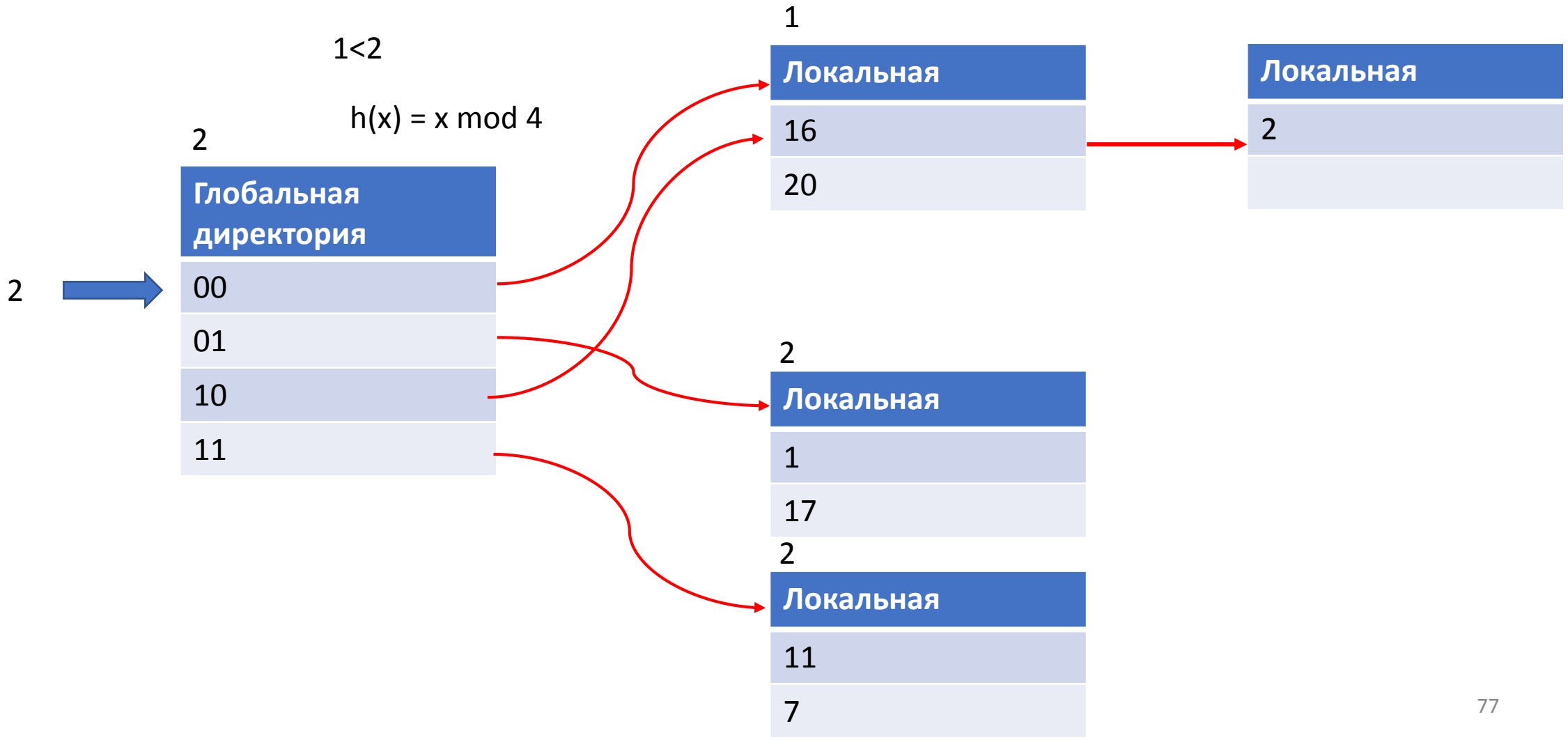
Расширяемое хеширование



Расширяемое хеширование



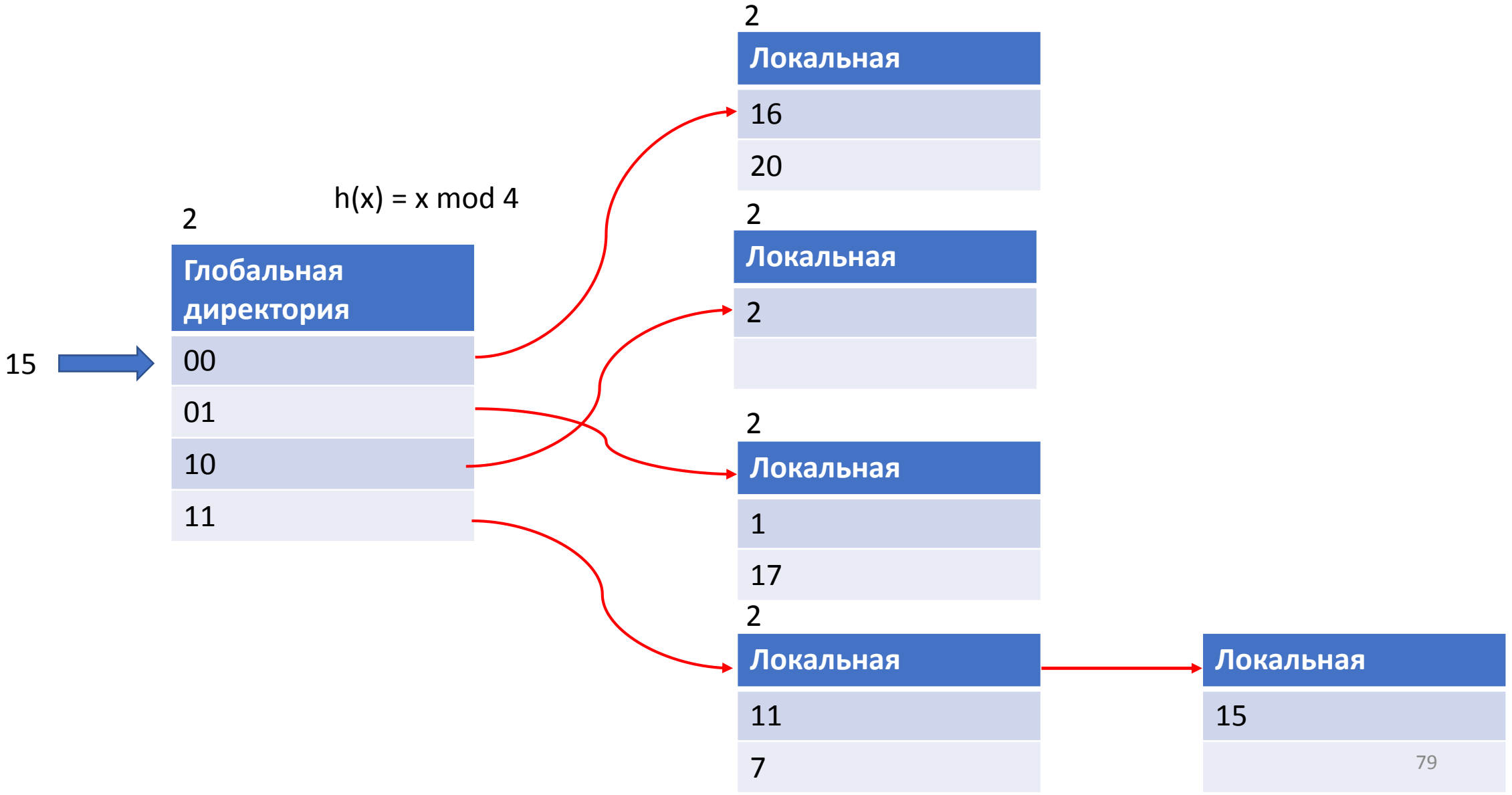
Расширяемое хеширование



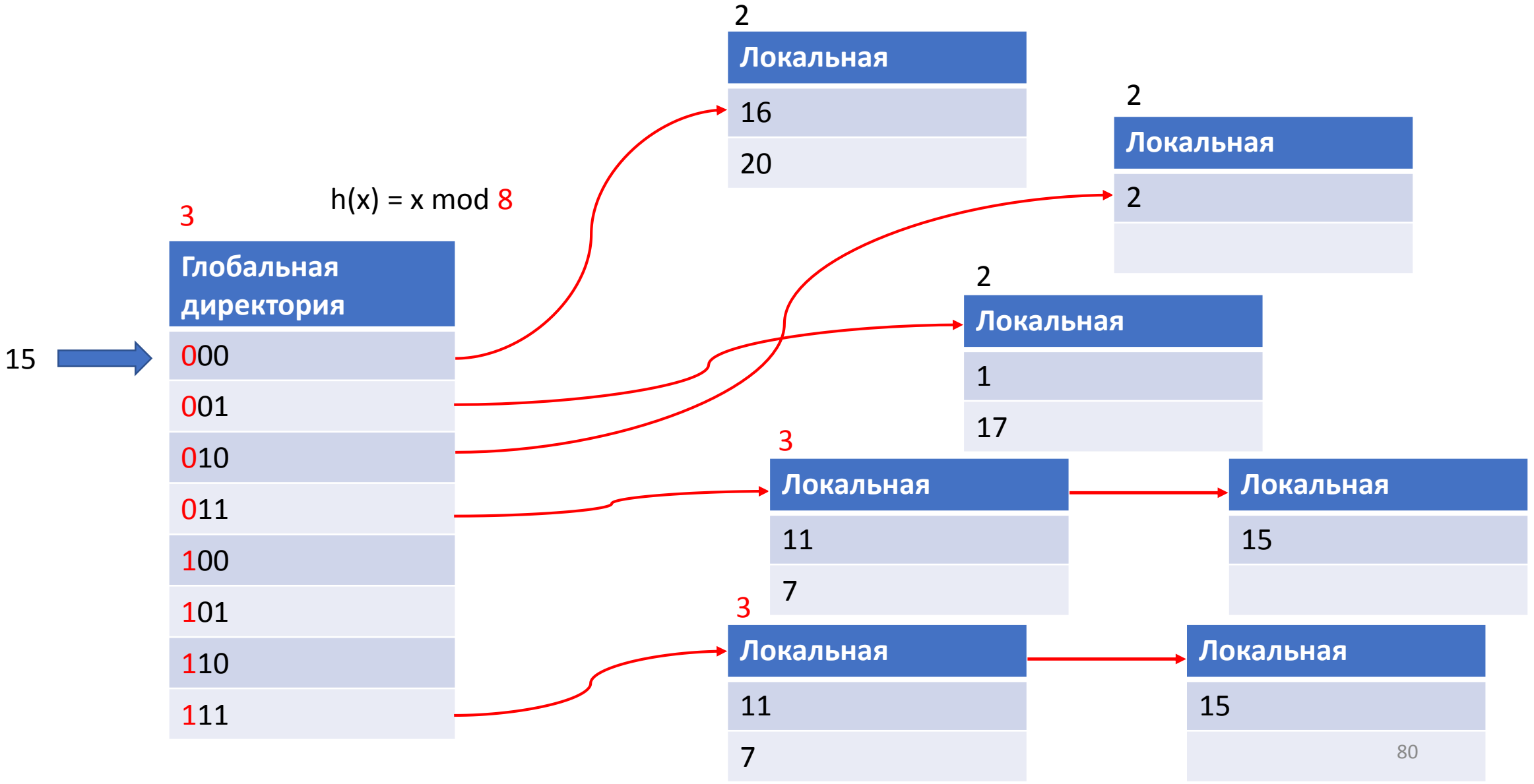
Расширяемое хеширование

1<2 Локальная степень < Глобальной

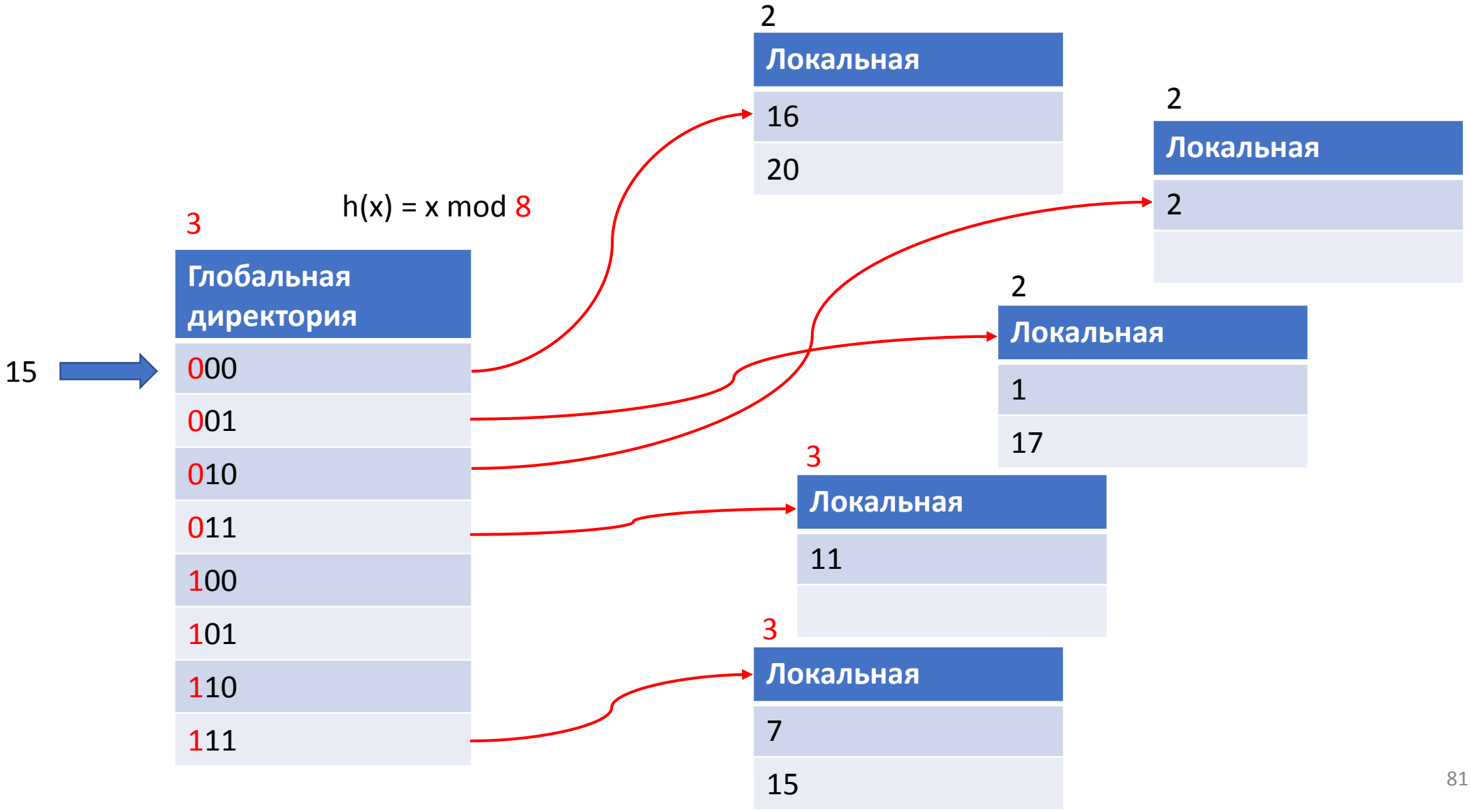
Расширяемое хеширование



Расширяемое хеширование



Расширяемое хеширование



Линейное хеширование

$H_0(key) = key \bmod 2$

$B_s = 2$ – количество слотов

$N_0 = 2$ – размер слота

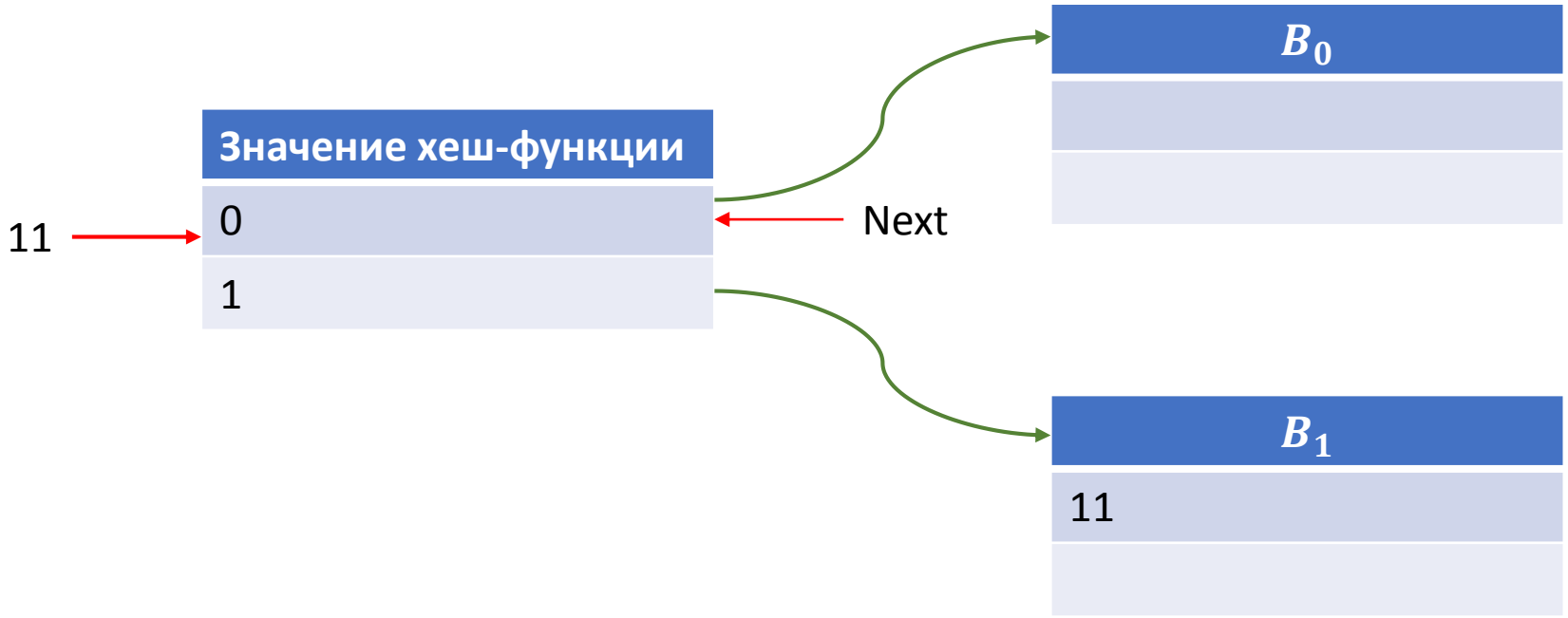
Значение хеш-функции	
0	Next
1	

Линейное хеширование

$H_0(key) = key \ mod \ 2$

$B_s = 2$ – количество слотов

$N_0 = 2$ – размер слота

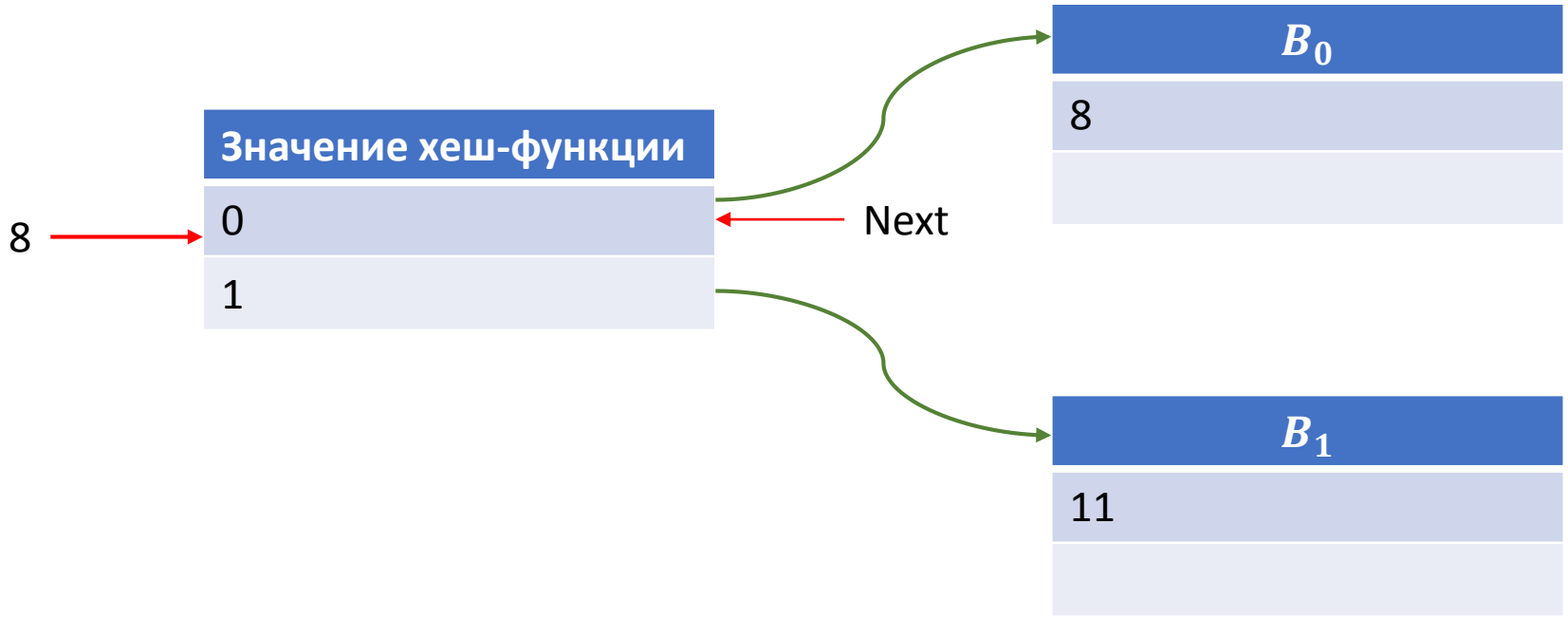


Линейное хеширование

$H_0(key) = key \ mod \ 2$

$B_s = 2$ – количество слотов

$N_0 = 2$ – размер слота

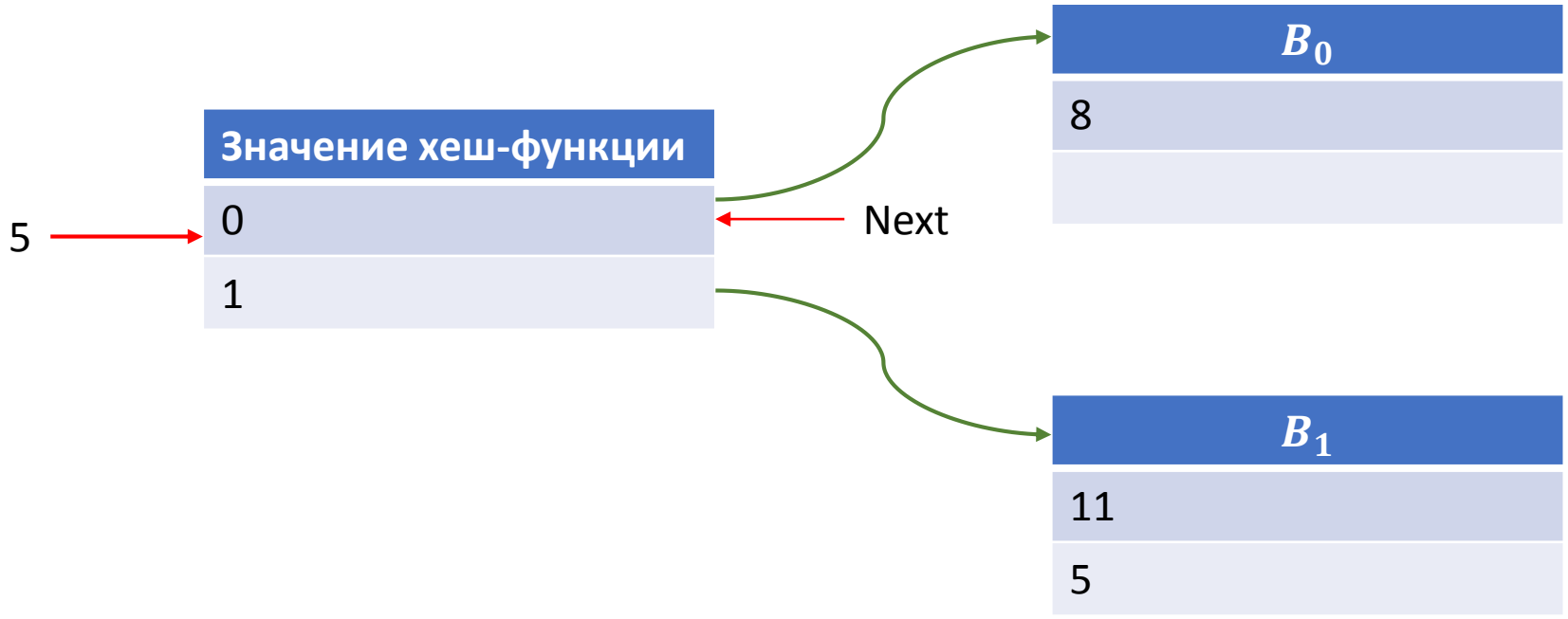


Линейное хеширование

$H_0(key) = key \bmod 2$

$B_s = 2$ – количество слотов

$N_0 = 2$ – размер слота

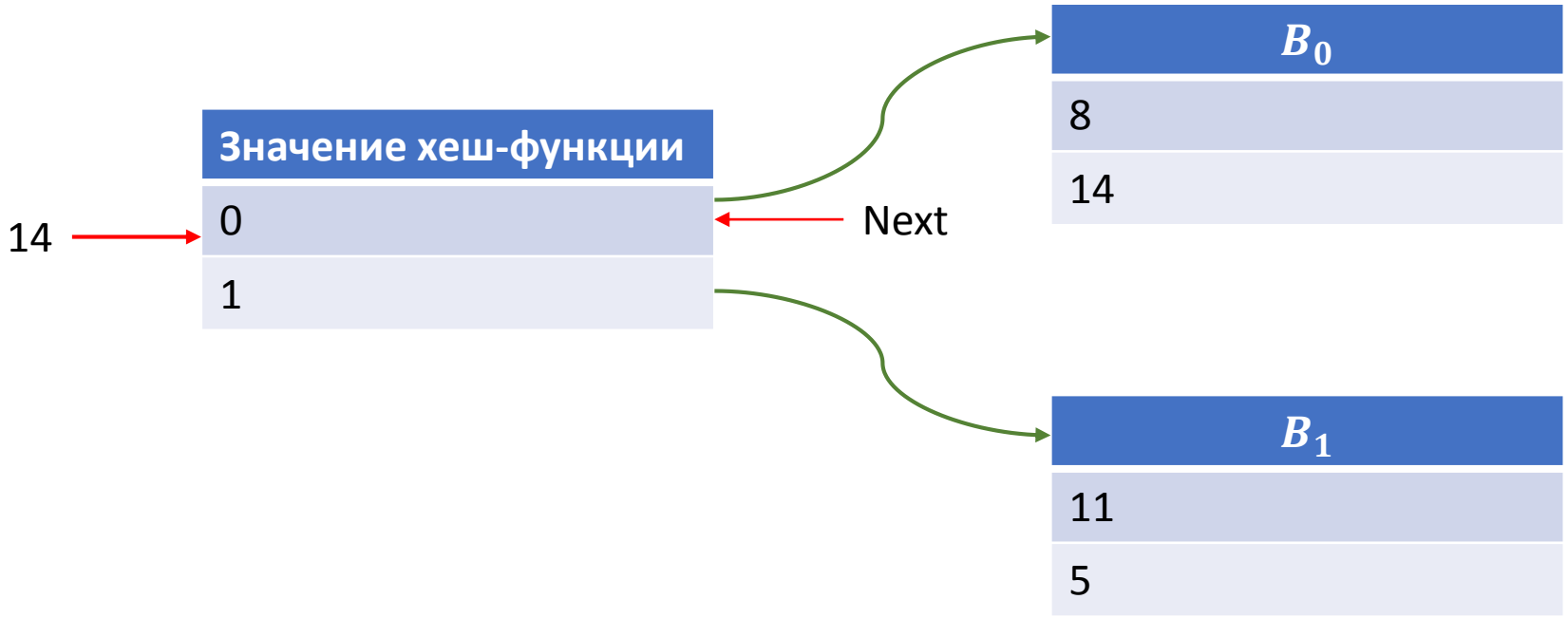


Линейное хеширование

$H_0(key) = key \ mod \ 2$

$B_s = 2$ – количество слотов

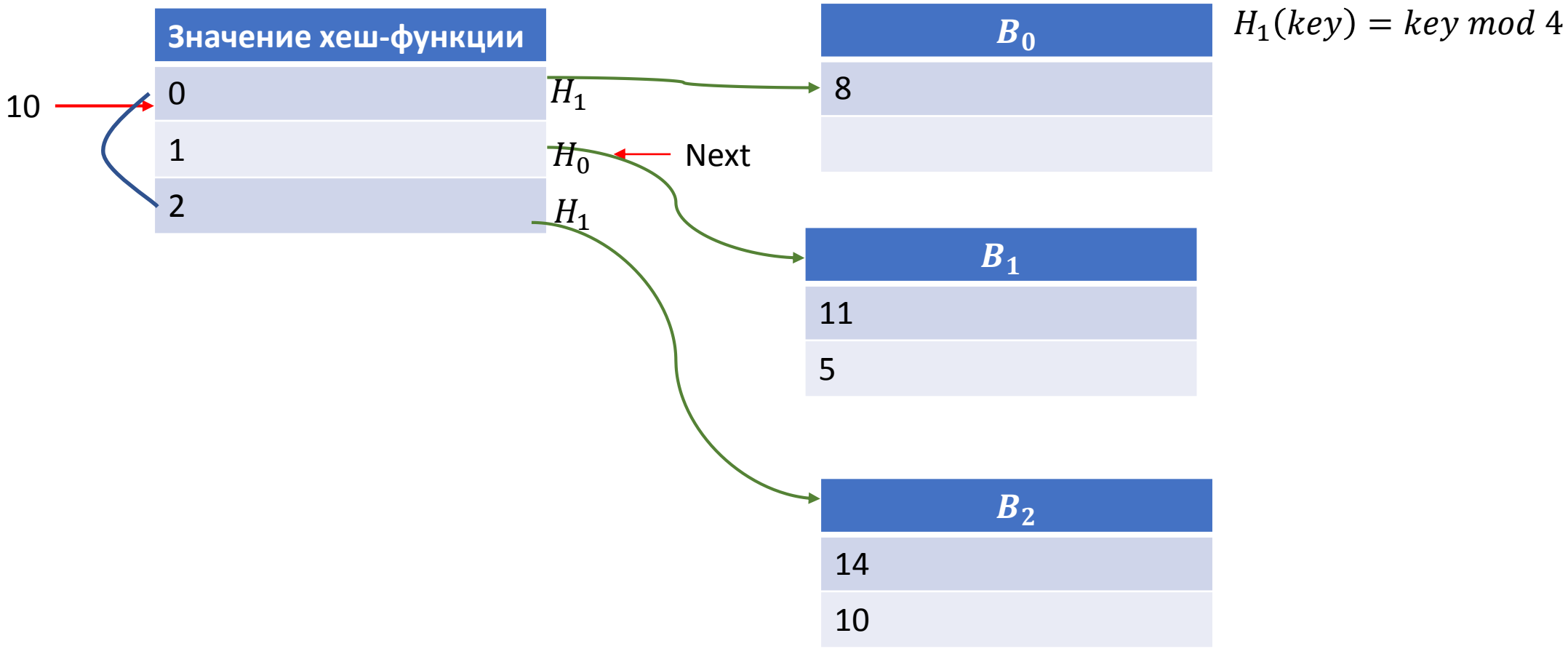
$N_0 = 2$ – размер слота



Линейное хеширование

R_0
- $H_0(key) = key \bmod 2$
- $B_s = 2$ – количество слотов
- $N_0 = 2$ – размер слота

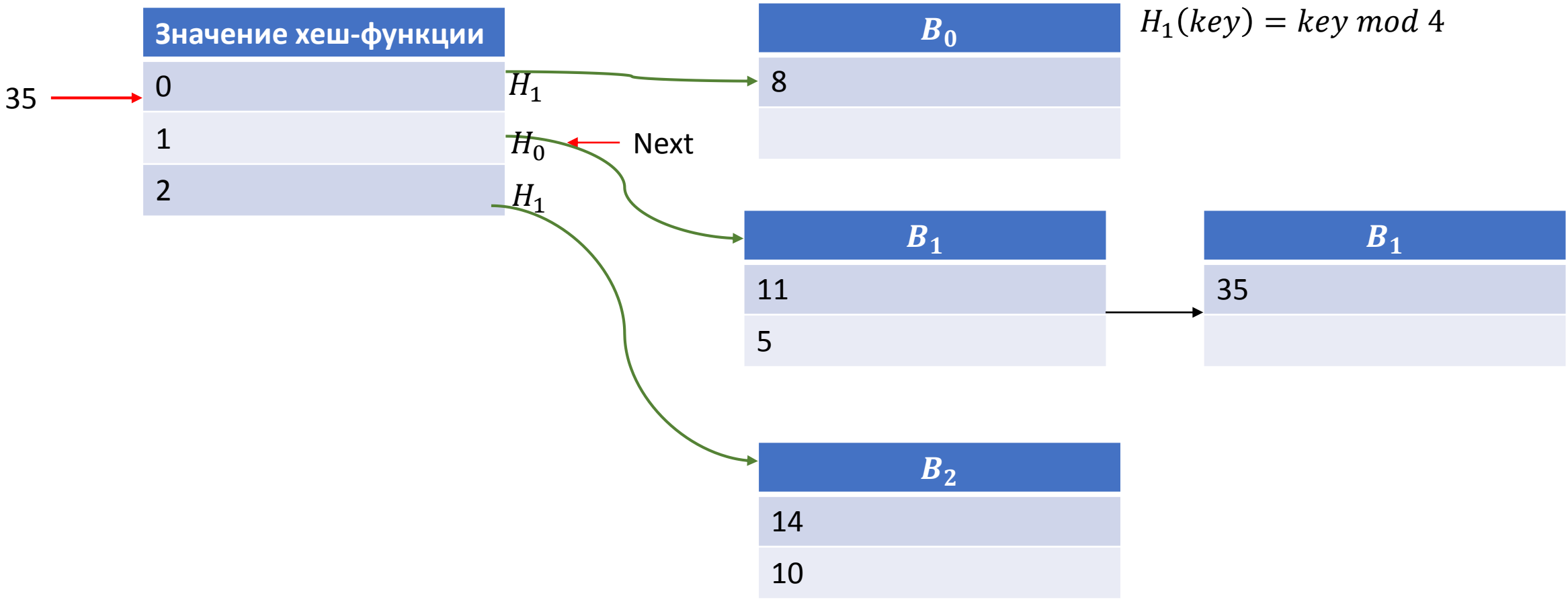
$H_1(key) = key \bmod 4$



Линейное хеширование

R_0
- $H_0(key) = key \bmod 2$
- $B_s = 2$ – количество слотов
- $N_0 = 2$ – размер слота

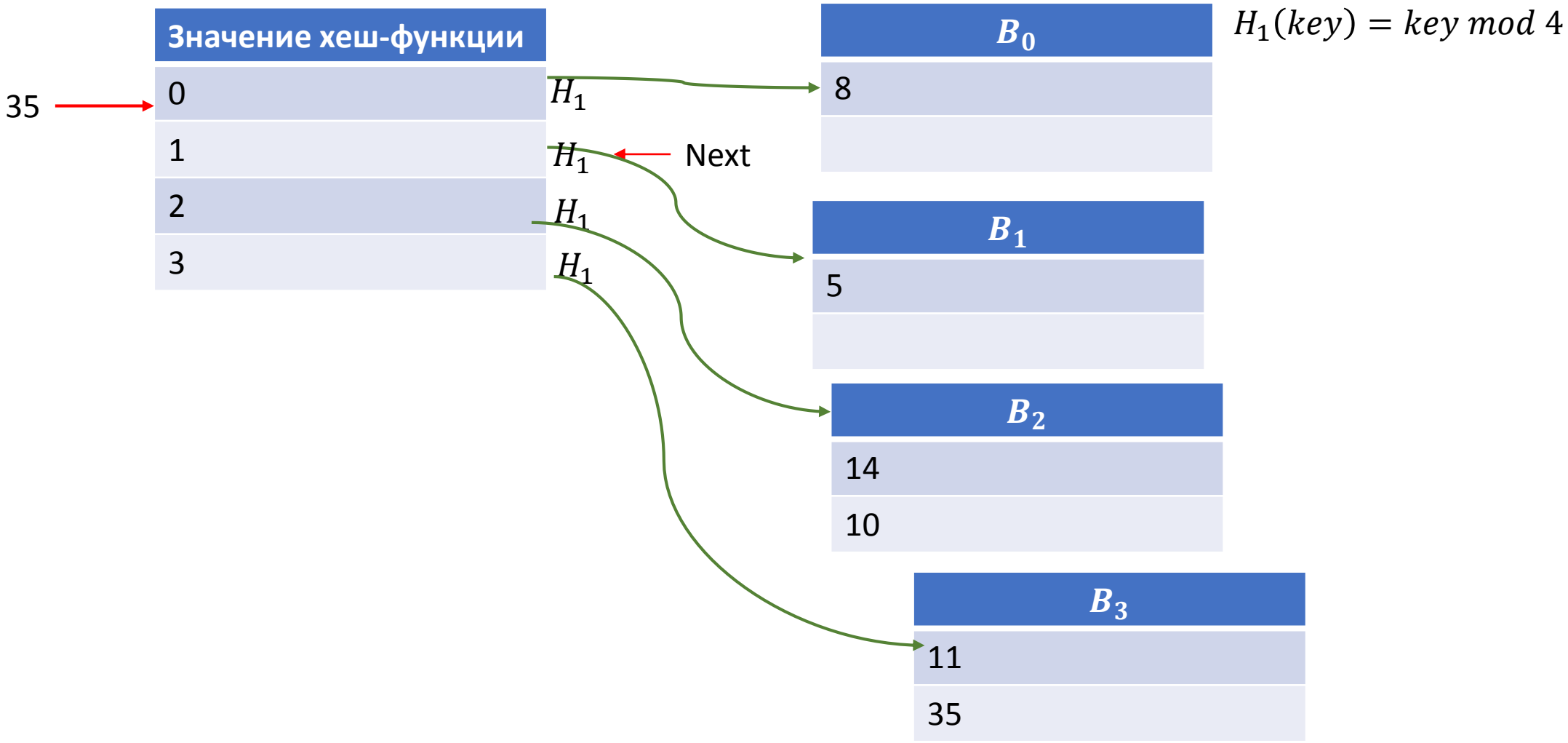
$H_1(key) = key \bmod 4$



Линейное хеширование

R_0
- $H_0(key) = key \bmod 2$
- $B_s = 2$ – количество слотов
- $N_0 = 2$ – размер слота

$H_1(key) = key \bmod 4$

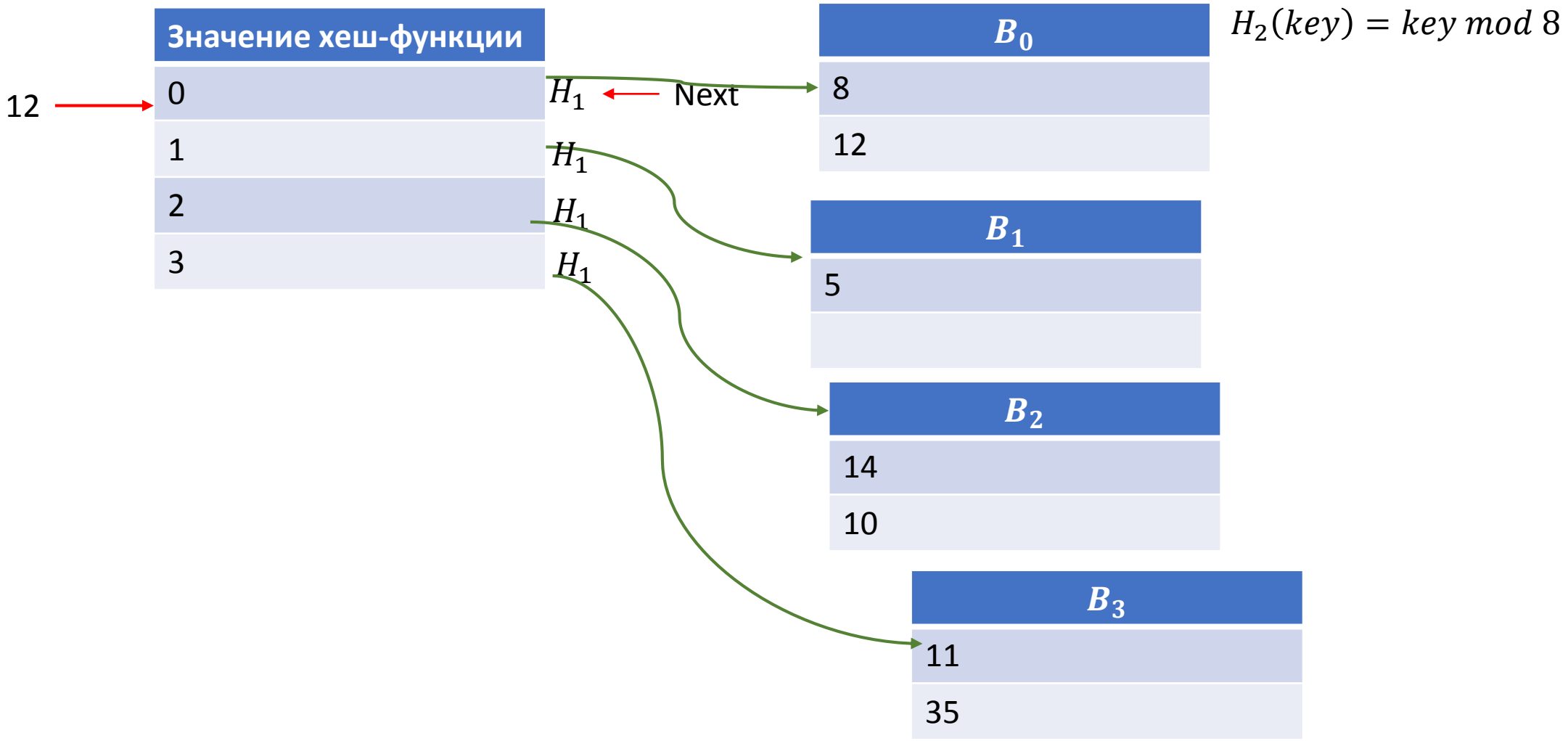


Линейное хеширование

R_1
- $H_1(key) = key \ mod \ 4$
- $B_s = 4$ – количество слотов

$N_0 = 2$ – размер слота

$H_2(key) = key \ mod \ 8$



Сравнение упорядоченного индексирования и хеширования

- Стоимость периодической реорганизации
- Относительная частота вставок и удалений
- Желательно ли оптимизировать среднее время доступа за счет наихудшего времени доступа?
- Ожидаемый тип запросов:
  - Хеширование обычно лучше при извлечении записей, имеющих указанное значение ключа.
  - Если запросы распространены с диапазонами, упорядоченные индексы должны быть предпочтительными

Сравнение упорядоченного индексирования и хеширования

- На практике:
  - PostgreSQL поддерживает хеш-индексы, но не рекомендует использовать его из-за низкой производительности
  - Oracle поддерживает статическую хеш-организацию, но не хеш-индексы
  - SQLServer поддерживает только B+ -дерево