

➤ **ES6 коллекции на примере V8:  
у ней внутри неонка**

**Андрей Печкуров**

# О докладчике

- Пишу на Java (очень долго), Node.js (долго)
- Node.js core collaborator
- Интересы: веб, архитектура, распределенные системы, производительность
- Можно найти тут:
  - <https://twitter.com/AndreyPechkurov>
  - <https://github.com/puzpuzpuz>
  - <https://medium.com/@apechkurov>

ECMAScript 2015 (ES6) привнес в JS стандартные коллекции:

- Map
- Set
- WeakMap
- WeakSet

```
const map = new Map();
map.set('foo', { bar: 'baz' });
for (let [key, value] of map) {
  console.log(`${key}:`, value);
}
```

```
const set = new Set();
set.add('foo');
set.add('bar');
set.forEach((item) => {
  console.log(item);
});
```

Спецификация не настаивает ни на чем конкретном:

Map object must be implemented using **either hash tables or other mechanisms** that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Map objects specification is only intended to describe the required observable semantics of Map objects. It is not intended to be a viable implementation model.

java.util

## **Interface Map<K,V>**

### **Type Parameters:**

K - the type of keys maintained by this map

V - the type of mapped values

### **All Known Subinterfaces:**

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

### **All Known Implementing Classes:**

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

# Disclaimer

- Изложение основано на V8 8.4, Node.js [commit 238104c](#)
- Полагаться можно (и нужно) только на спецификацию ECMAScript
- Автор не работает в команде V8

# План на сегодня

- Map/Set
  - Алгоритм
  - Особенности реализации
  - Сложность
  - Память
- WeakMap/WeakSet
  - Алгоритм
  - Особенности реализации



## ➤ Map/Set: алгоритм

# Хеш-функция

(any) => number

-----

42      ➡      n1

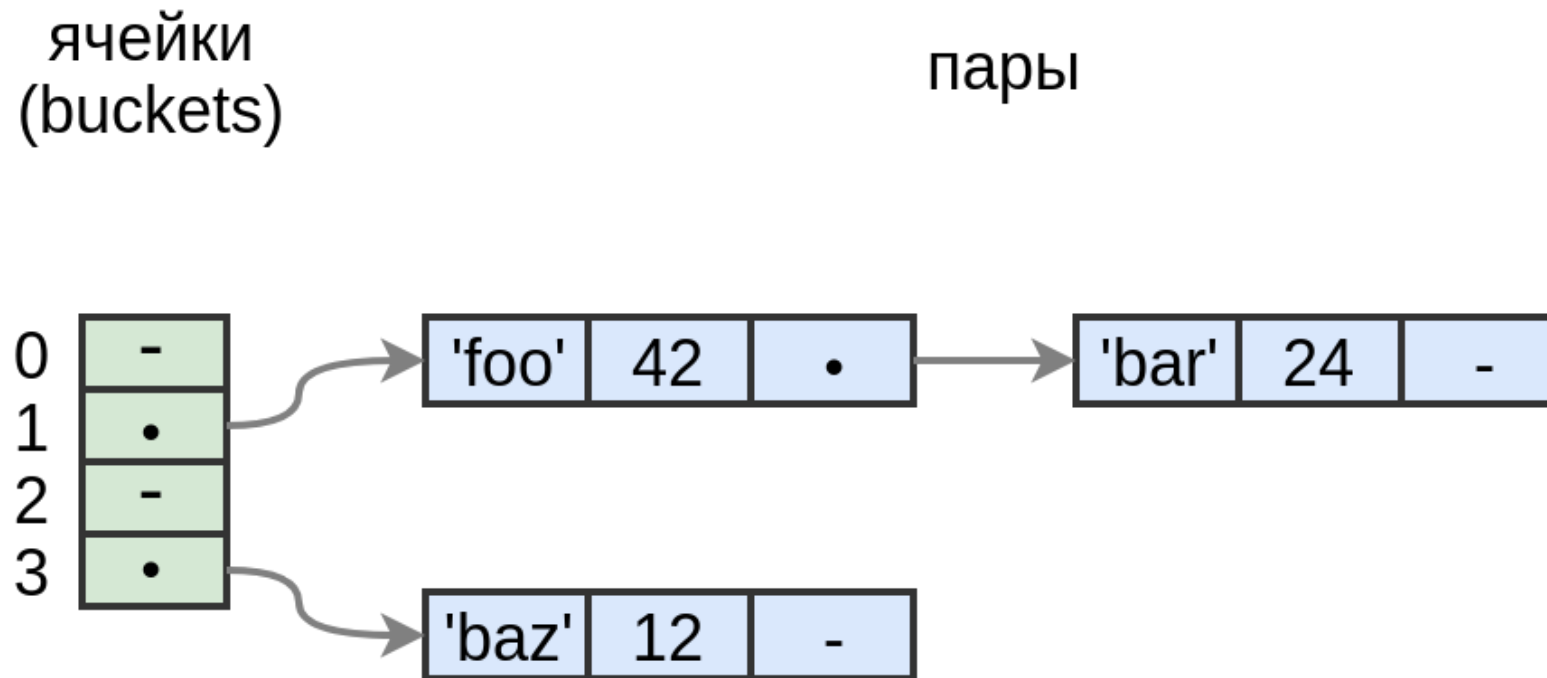
'foobar'      ➡      n2

{ foo: 'bar' }      ➡      n3

# Классическая хеш-таблица

- Хеш-таблица с цепочками
- Хеш-таблица с открытой адресацией

# Хеш-таблица с цепочками



# Перехеширование

- При достижении определенного размера таблицу нужно перехешировать
  - Емкость равна  $\text{коэффициент\_заполнения} * \text{кол\_во\_ячеек}$
- Новый размер таблицы при этом:  $\text{множитель} * \text{кол\_во\_ячеек}$

## Что внутри у Map/Set?

В силу спецификации в Map/Set не может быть "классической" хеш-таблицы

## Фрагмент спецификации

When the `forEach` method is called with one or two arguments, the following steps are taken:

...

7. Repeat for each Record `{[[key]], [[value]]}` `e` that is an element of `entries`, in original key **insertion order**

V8 реализует [deterministic hash table](#) (Tyler Close, 2012)



# Deterministic hash table

```
interface CloseTable {  
    hashTable: number[];  
    dataTable: Entry[];  
    nextSlot: number;  
    size: number;  
}
```

```
interface Entry {  
    key: any;  
    value: any;  
    chain: number;  
}
```

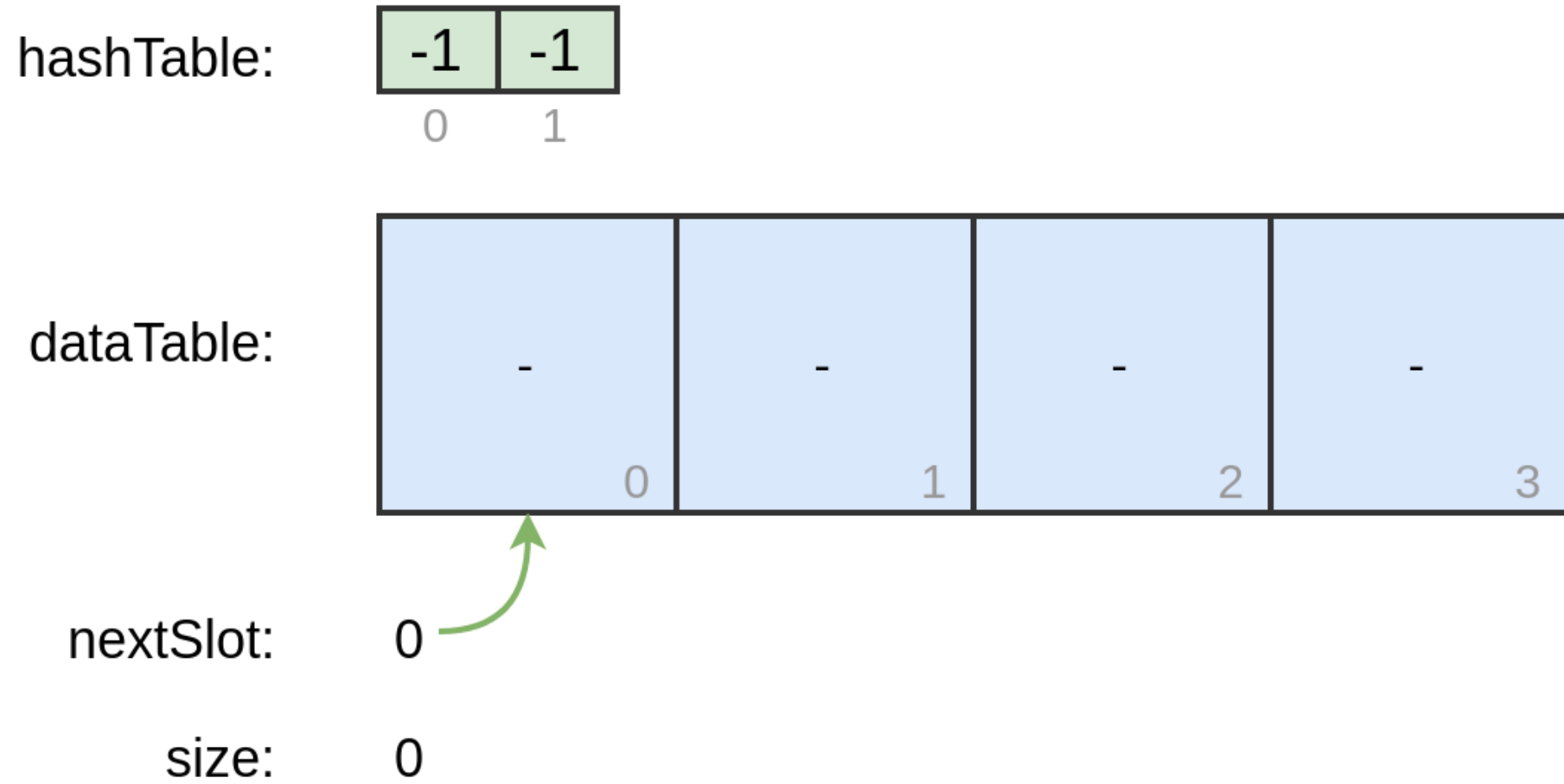


table.set(1, 'a')



hashTable:

-1	-1
0	1

dataTable:

-	-	-	-
0	1	2	3

nextSlot:

0



size:

0

hashTable:

-1	-1
0	1

table.set(1, 'a')

hash(1) % 2 === 1

dataTable:

-	-	-	-
0	1	2	3

nextSlot:

0

size:

0

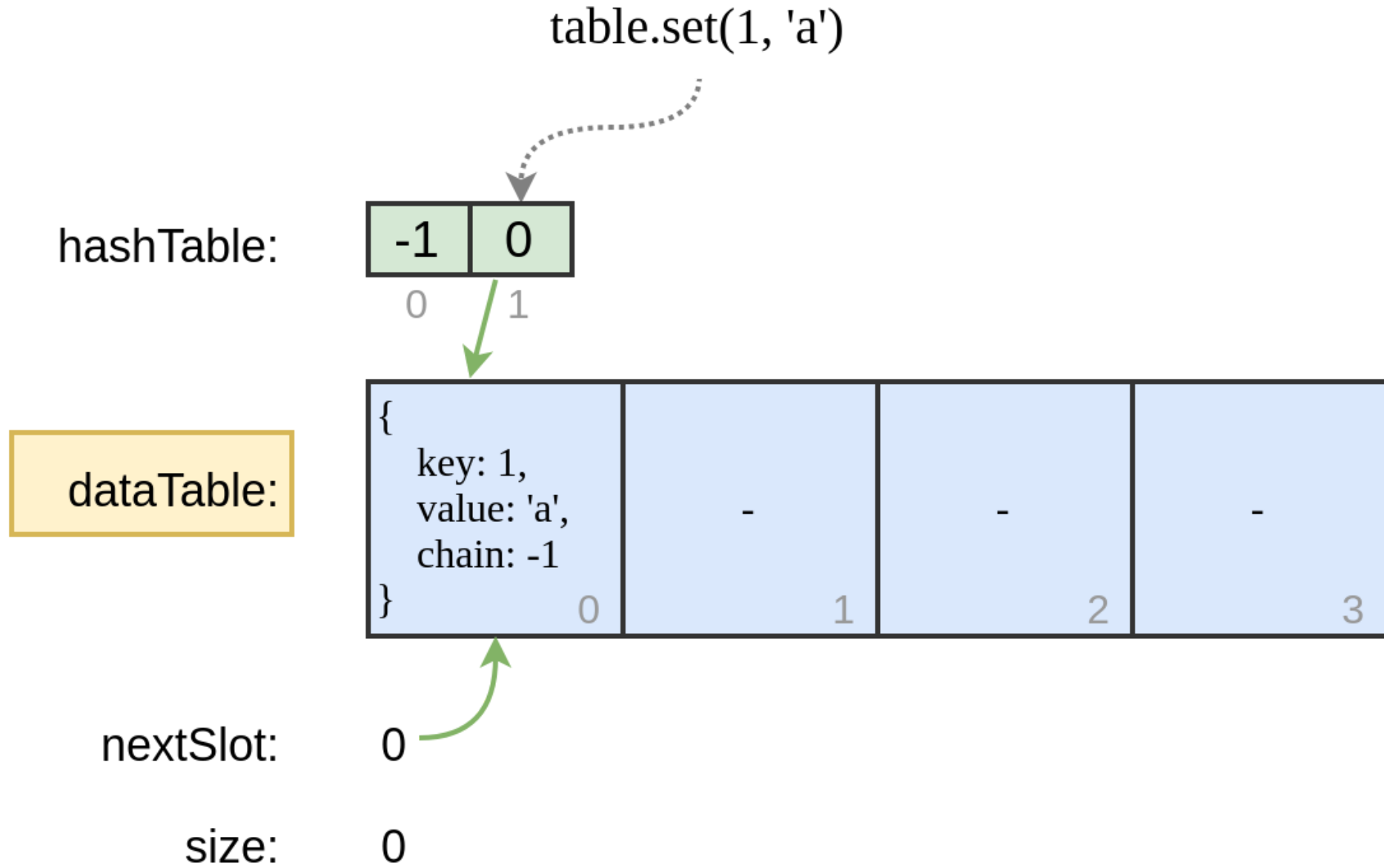


table.set(1, 'a')

hashTable:

-1	0
0	1

dataTable:

{ key: 1, value: 'a', chain: -1 }	-	-	-
0	1	2	3

nextSlot:

1

size:

1

table.set(3, 'b')



hashTable:

-1	0
0	1

dataTable:

{ key: 1, value: 'a', chain: -1 }	-	-	-
0	1	2	3

nextSlot:

1

size:

1

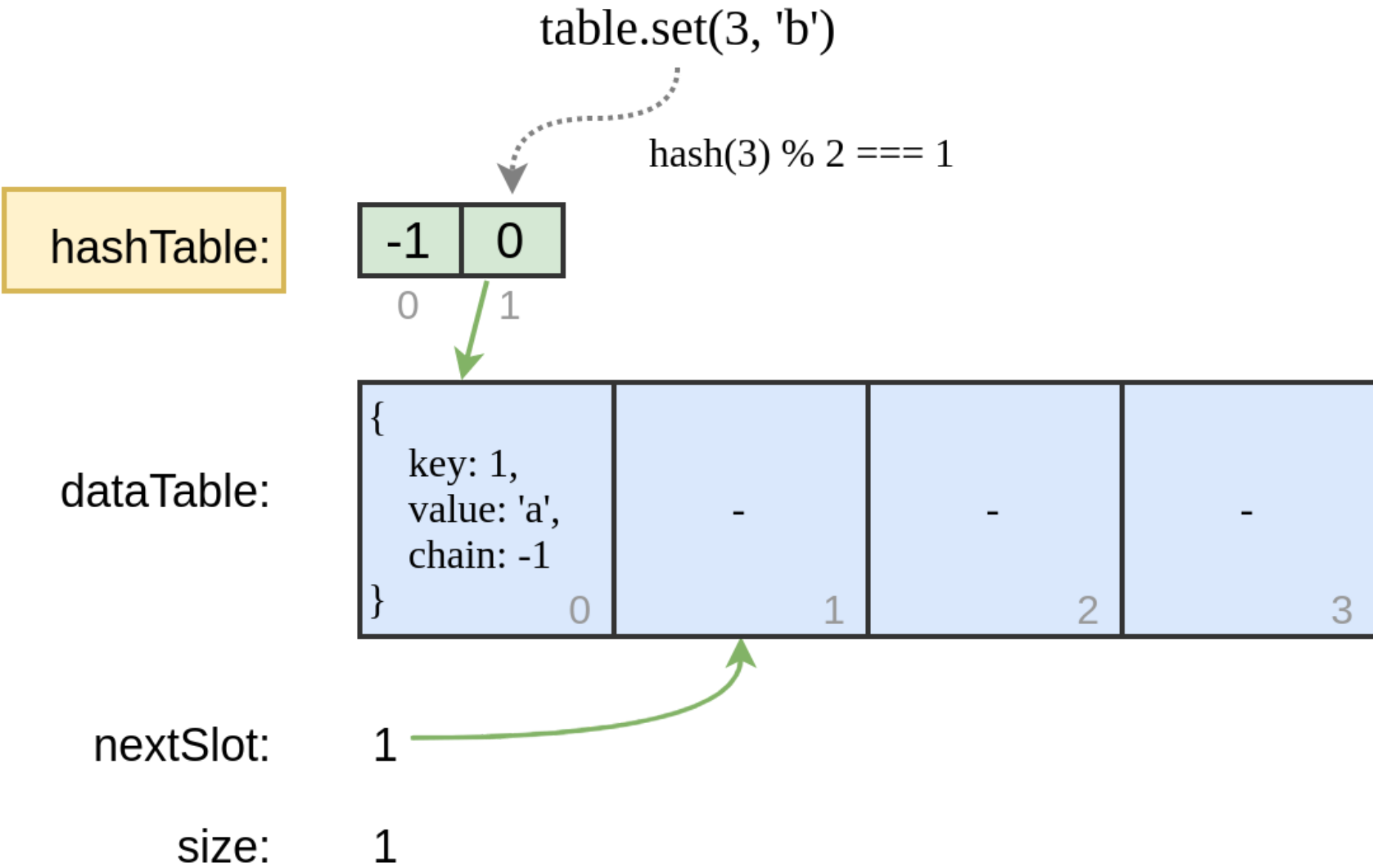




table.set(3, 'b')

hashTable:

-1	0
----	---

0 1

dataTable:

{ key: 1, value: 'a', chain: 1 }	{ key: 3, value: 'b', chain: -1 }	-	-
0	1	2	3

nextSlot:

2

size:

2

table.delete(1)



hashTable:

-1	0
----	---

0 1



dataTable:

{ key: 1, value: 'a', chain: 1 }	{ key: 3, value: 'b', chain: -1 }	-	-
0	1	2	3



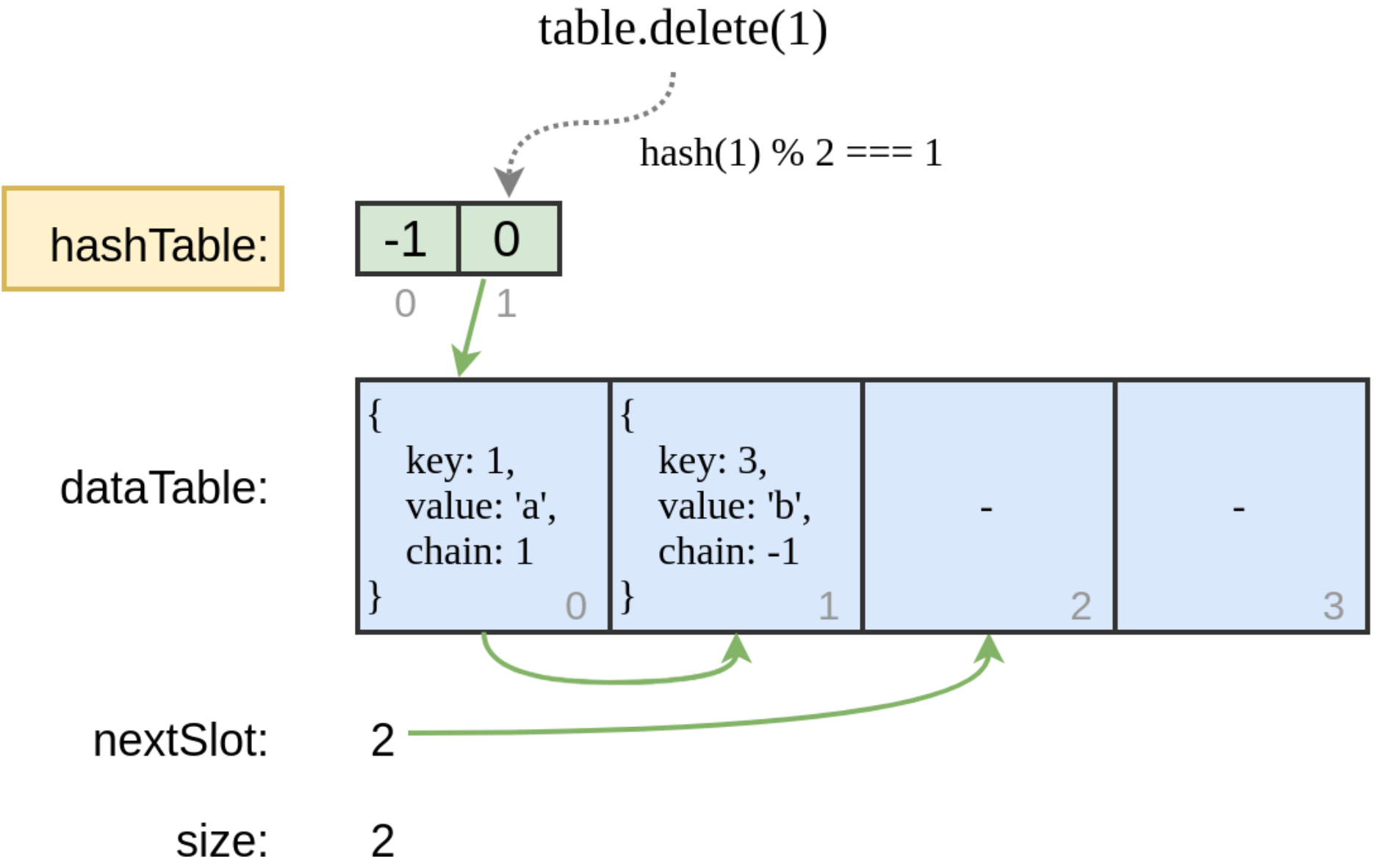
nextSlot:

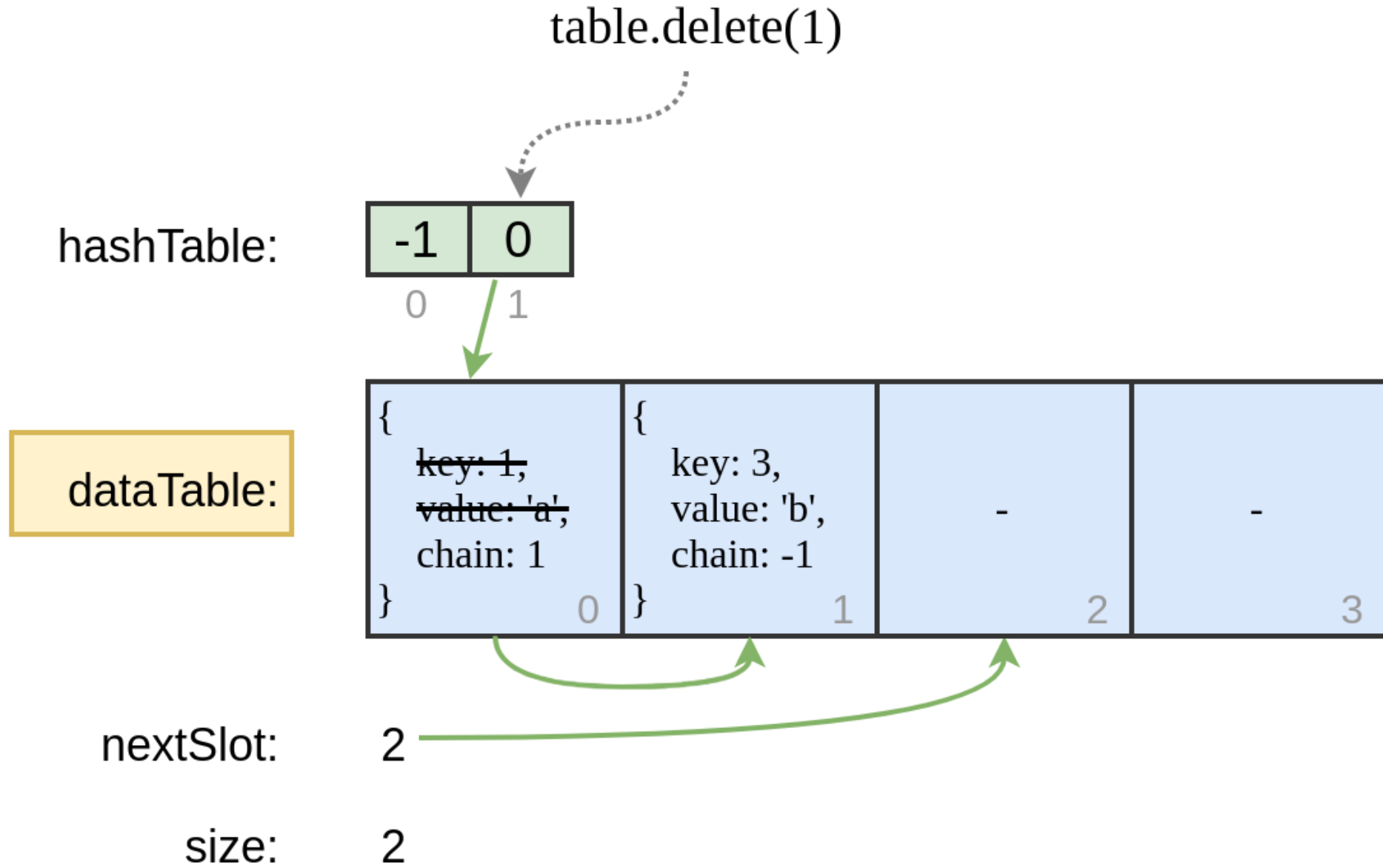
2

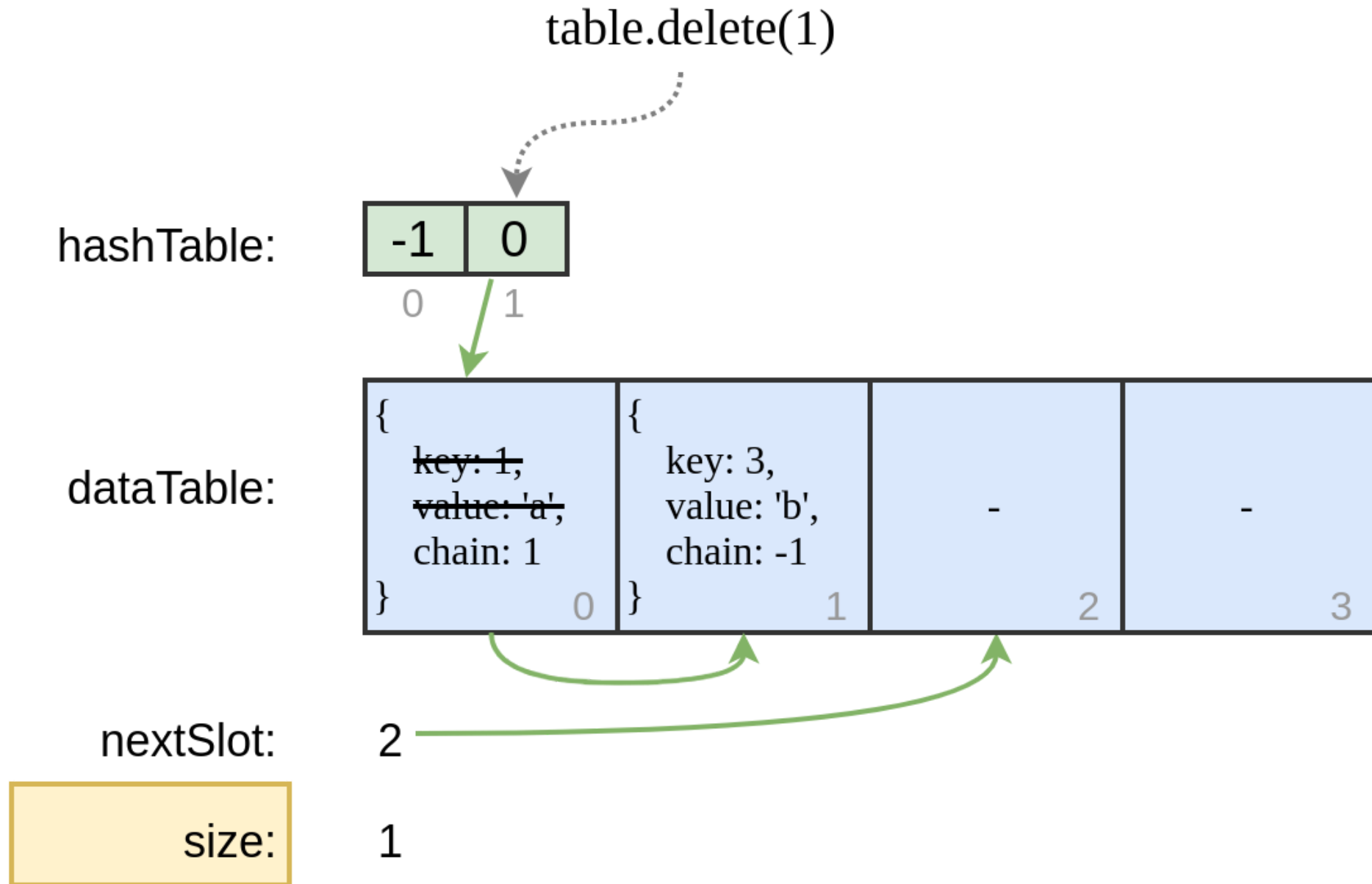


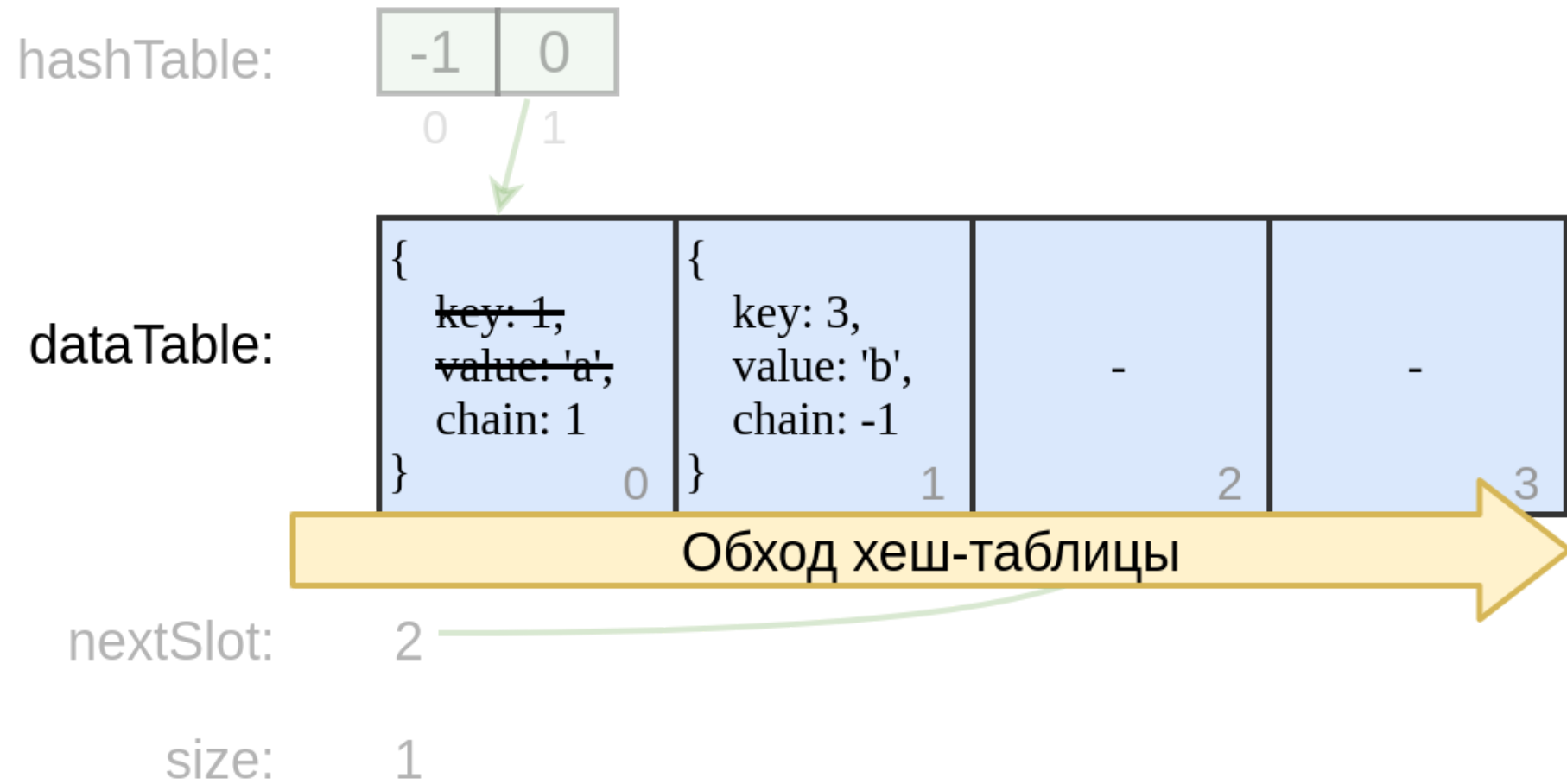
size:

2



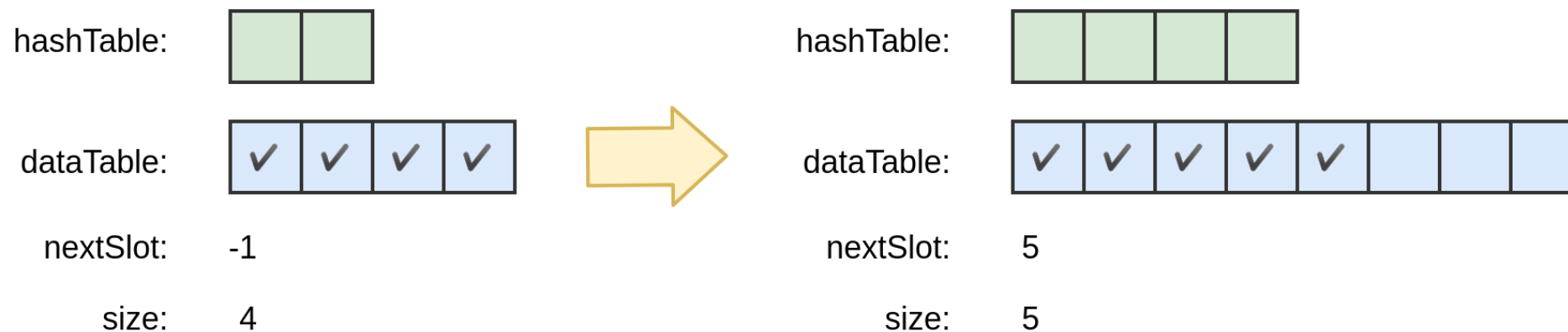






# Перехеширование при увеличении таблицы

`table.set(42, 'baz')`



## ➤ Map/Set: особенности реализации



Основа реализации Map/Set - классы `OrderedHashTable` и `OrderedHashMap` :

- [ordered-hash-table.h](#)
- [ordered-hash-table.cc](#)
- [builtins-collections-gen.cc](#)

# ЕМКОСТЬ

- Емкость это всегда степень двойки
- Коэффициент заполнения равен 2
  - Емкость равна  $2 * \text{кол\_во\_ячеек}$

# Границы

- Начальная емкость: `new Map()` содержит 2 ячейки (емкость равна 4)
- Максимальная емкость: на 64-битной системе емкость Map ограничена  $2^{27}$  (~16.7 млн. пар)

# Перехеширование

- Множитель при перехешировании тоже 2
  - Таблица увеличивается/уменьшается в 2 раза

## Экспериментальная ветка Node.js

```
puzpuzpuz@apechkurov-laptop: ~/projects/node
puzpuzpuz@apechkurov-laptop:~/projects/node$ ./node
Welcome to Node.js v15.0.0-pre.
Type ".help" for more information.
> const map = new Map()
undefined
> map.buckets
2
> for (let i = 0; i < 5; i++) map.set(i, i)
Map(5) { 0 => 0, 1 => 1, 2 => 2, 3 => 3, 4 => 4 }
> map.buckets
4
> for (let i = 5; i < 10; i++) map.set(i, i)
Map(10) {
  0 => 0,
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4,
  5 => 5,
  6 => 6,
  7 => 7,
  8 => 8,
  9 => 9
}
> map.buckets
8
> 
```

## ➤ Map/Set: сложность

# Big O

	В среднем случае	В худшем случае
Поиск	$O(1)$	$O(n)$
Вставка*	$O(1)$	$O(n)$
Удаление**	$O(1)$	$O(n)$

\* Может привести к увеличению таблицы

\*\* Может привести к уменьшению таблицы

## ➤ Map/Set: память

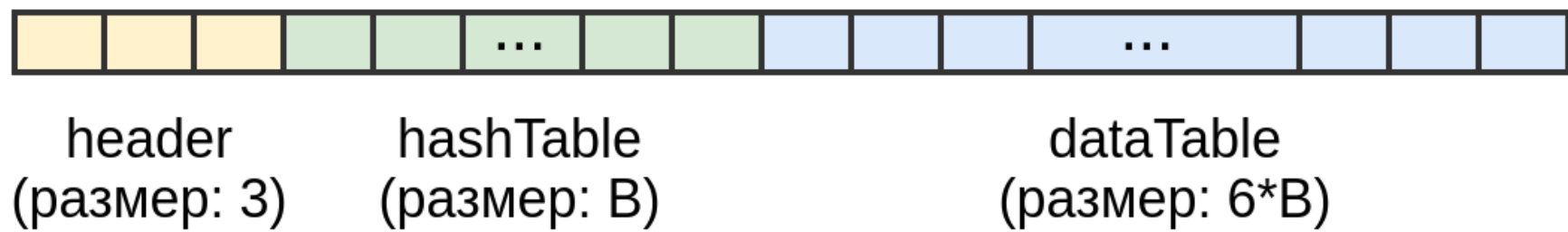


## Как Map/Set хранятся в памяти?

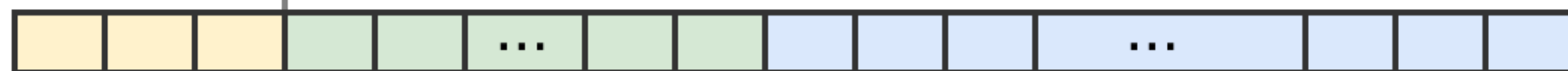
```
interface CloseTable {  
    hashTable: number[];  
    dataTable: Entry[];  
    nextSlot: number;  
    size: number;  
}
```

С точки зрения хранения данных в куче V8 Map/Set это JS массивы

P.S. Отсюда следует ограничение на размеры



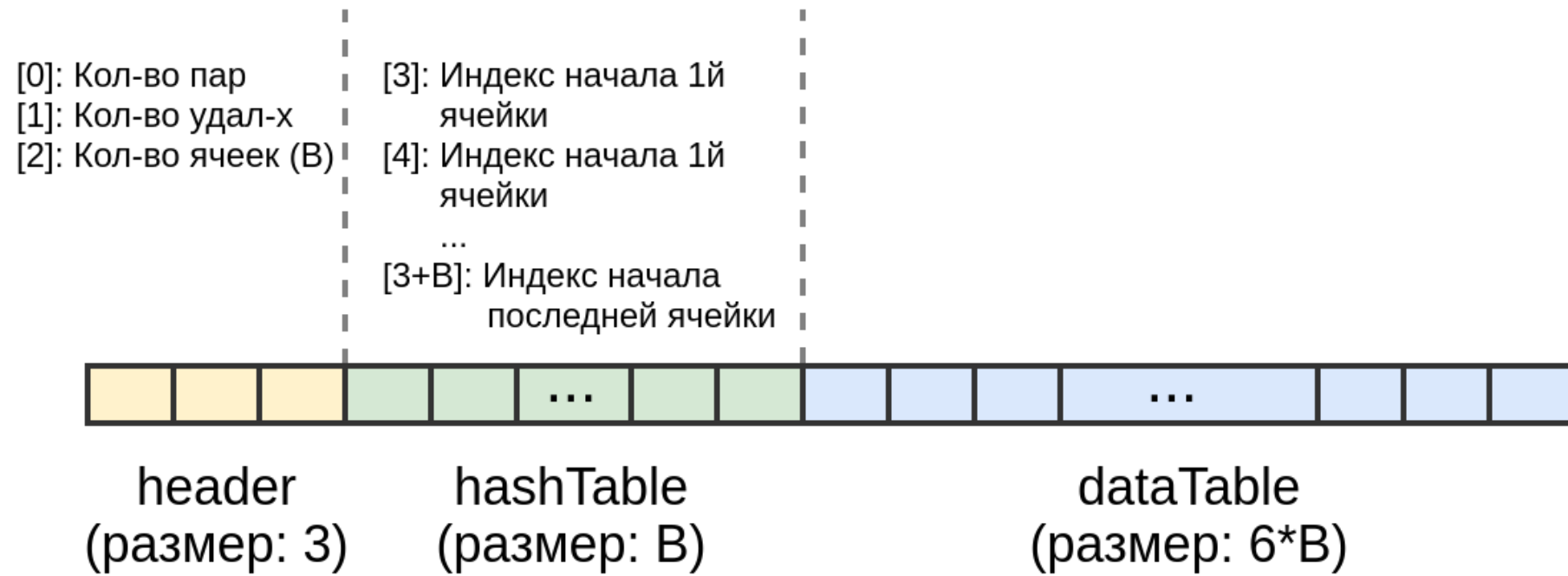
[0]: Кол-во пар  
[1]: Кол-во удал-х  
[2]: Кол-во ячеек (B)

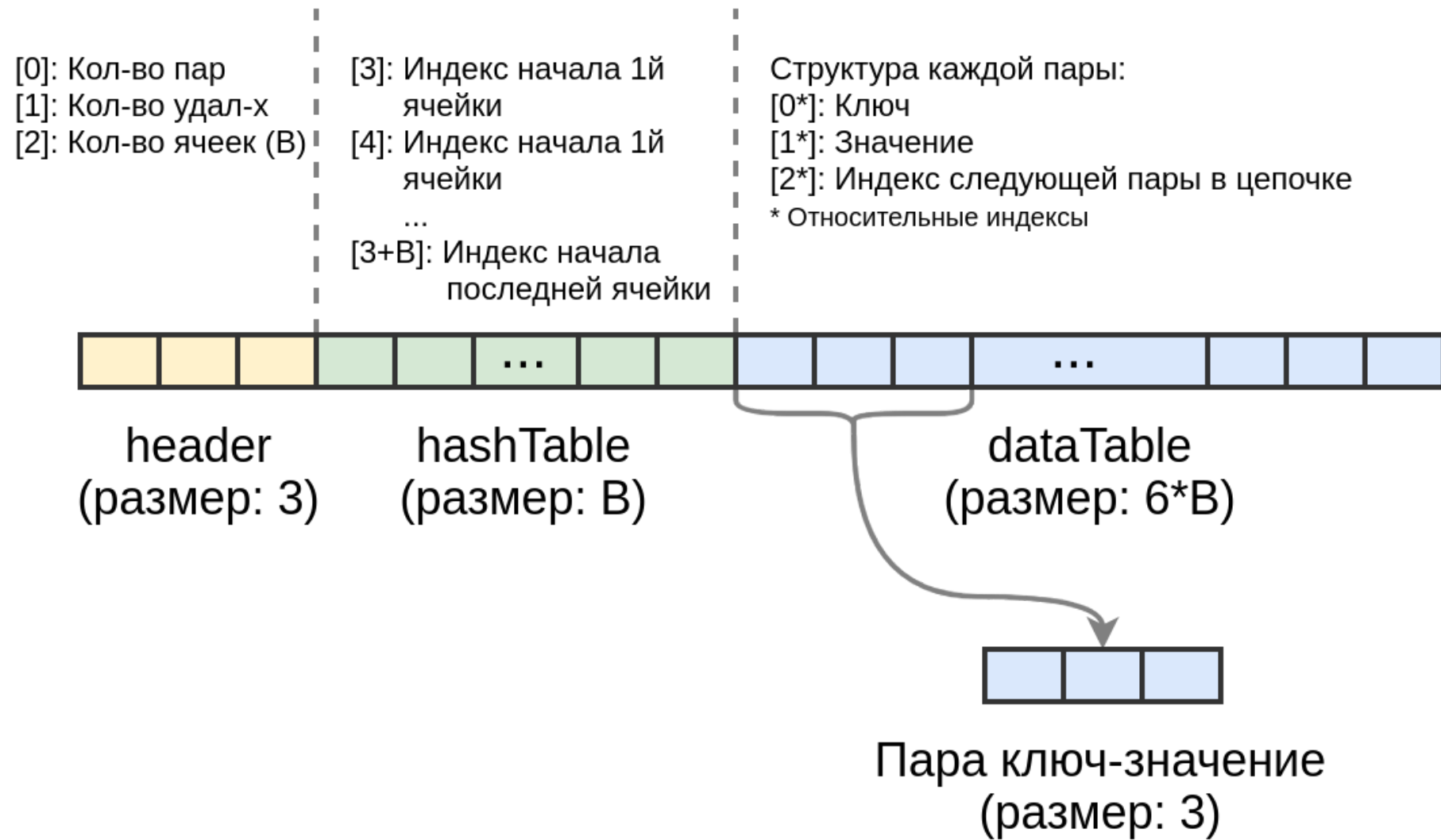


header  
(размер: 3)

hashTable  
(размер: B)

dataTable  
(размер:  $6*B$ )





## Немного арифметики

Размер массива можно оценить примерно как:

- $3.5 * N$  , где  $N$  - емкость Map
- $2.5 * N$  , где  $N$  - емкость Set

## Входные условия

Для 64-битной системы (без учета [pointer compression](#)) каждый элемент массива занимает 8 байтов



## Итого

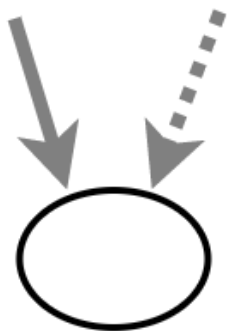
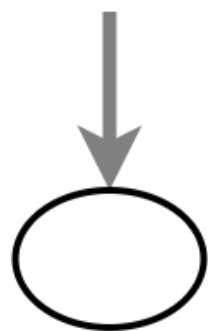
- Map с  $2^{20}$  (~1 млн.) пар займет ~29MB памяти
- Set с  $2^{20}$  (~1 млн.) элементов займет ~21MB памяти

## Map/Set в сухом остатке

- В основе - deterministic hash table
- Алгоритмическая сложность такая же, как у "классической" хеш-таблицы
- Размер - степень двойки
- Коэффициент заполненности и множители тоже равны 2
- Потребление памяти - умеренное

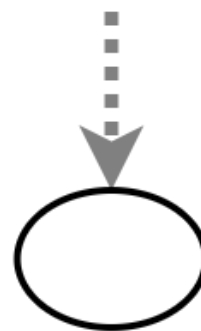
## ➤ WeakMap/WeakSet: алгоритм

Не подлежит GC



— strong reference

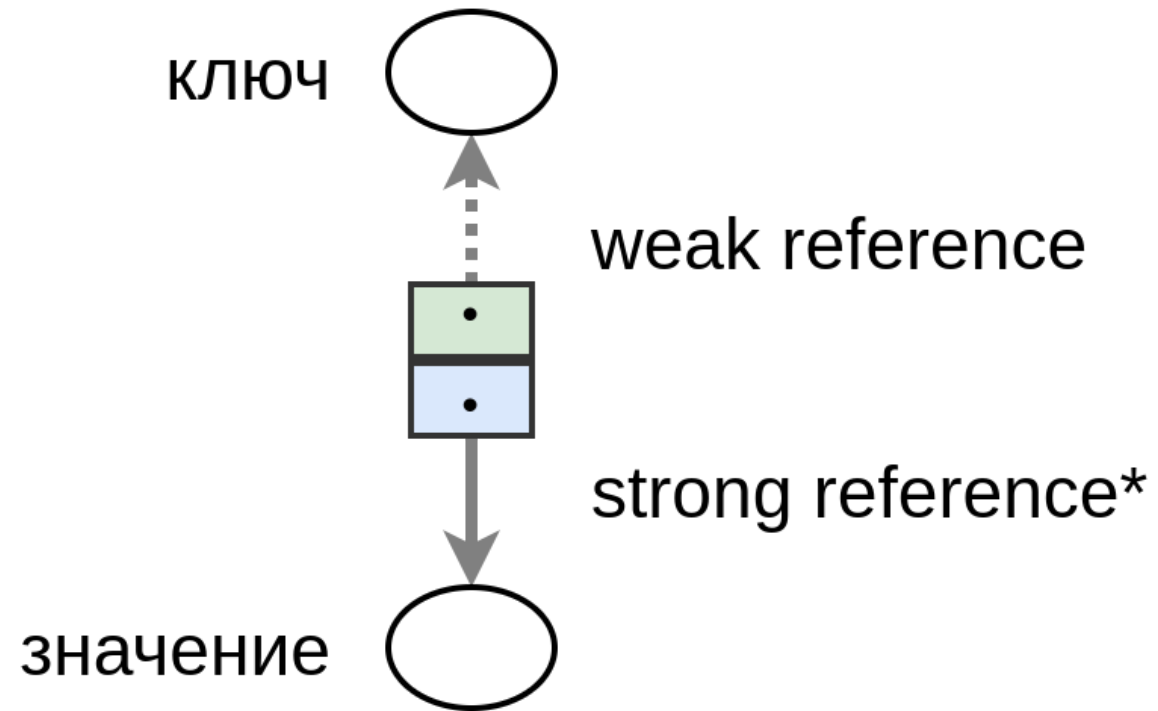
Подлежит GC 



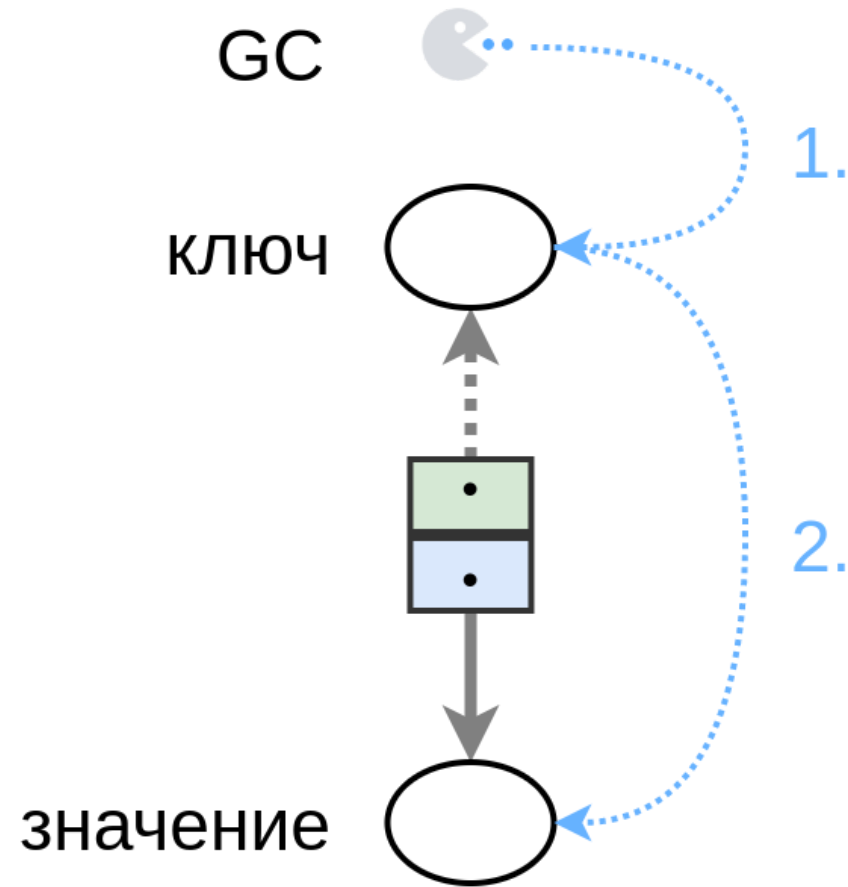
..... weak reference

Так почему бы не взять Map + WeakRef в основу WeakMap?

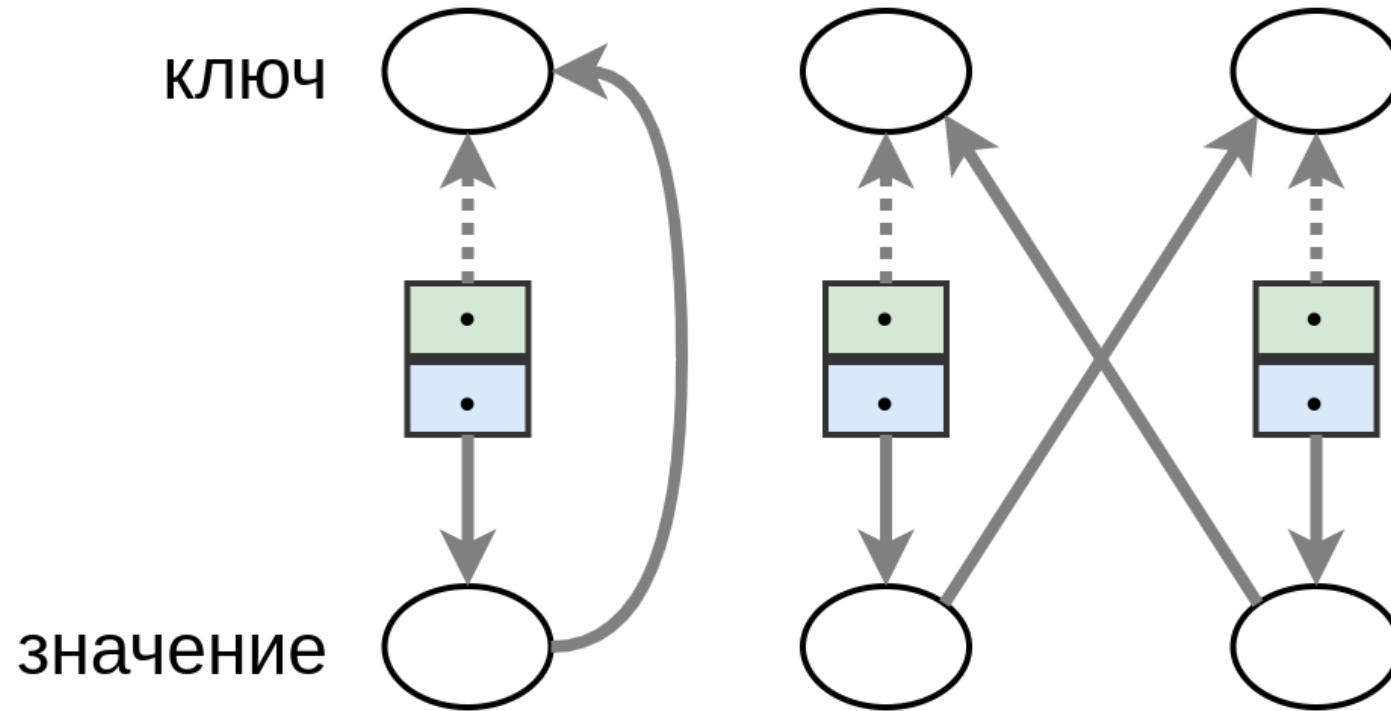
# Map + WeakRef



# Map + WeakRef: поведение при GC



# Map + WeakRef: проблема



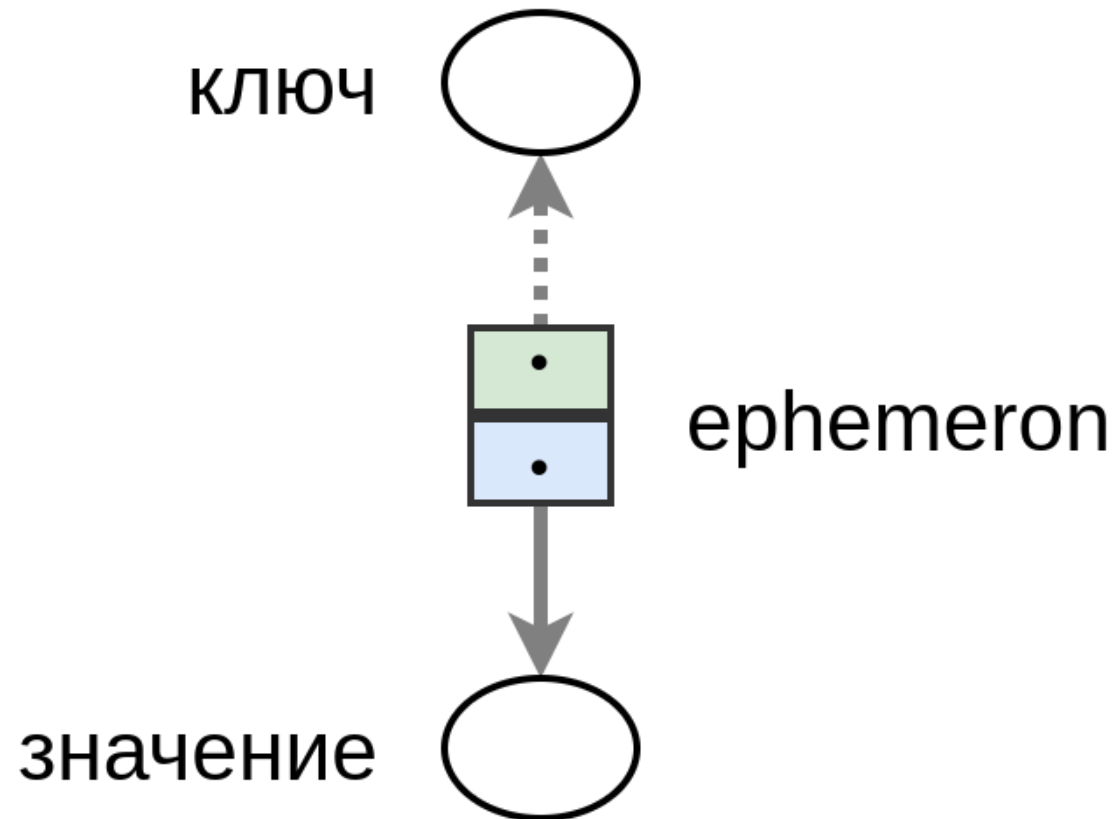


## Внимание, ответ

На помощь спешит механизм [ephemeron](#)\*

\* Справка: можно встретить в Lua и .NET

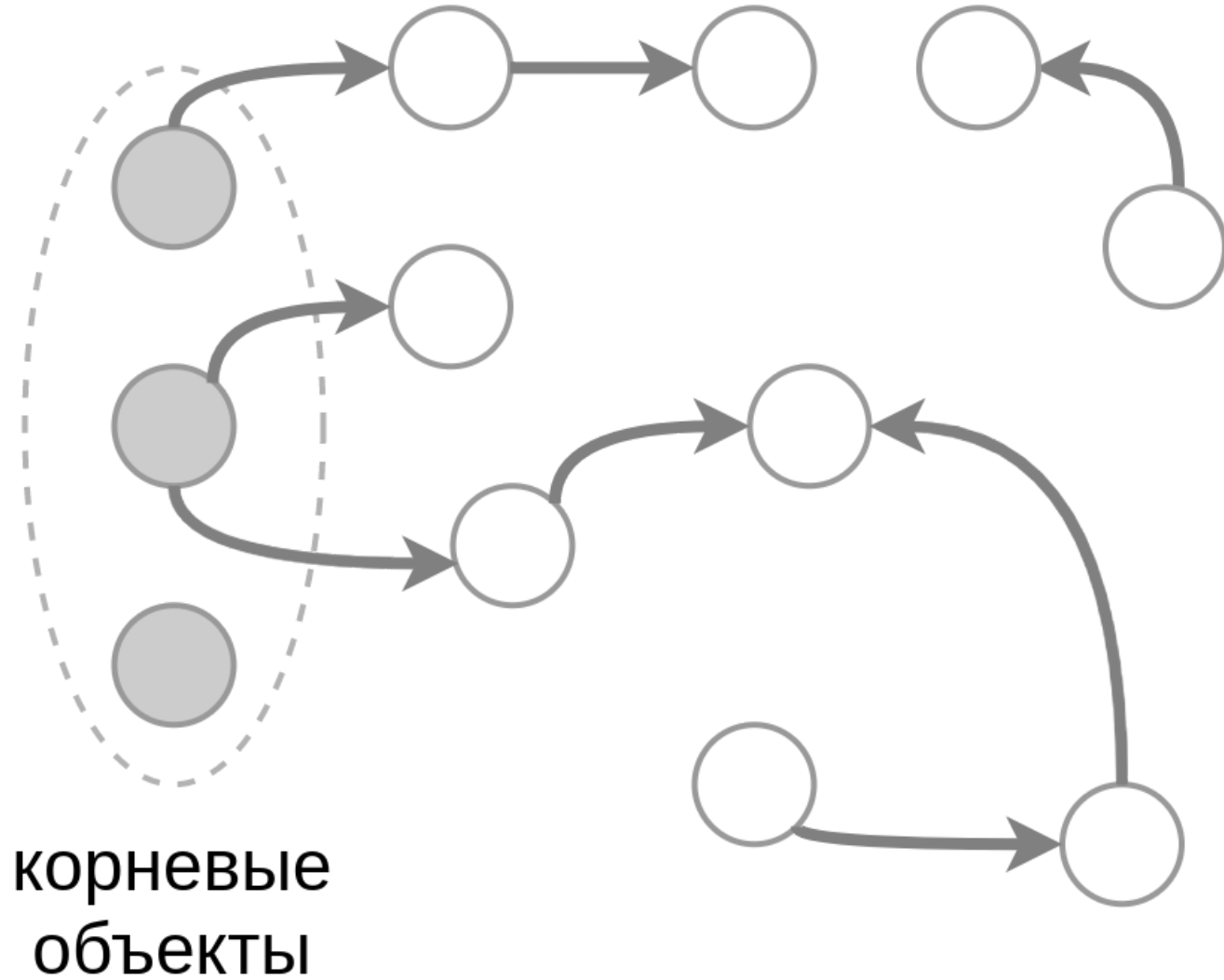
Еphemeron это пара



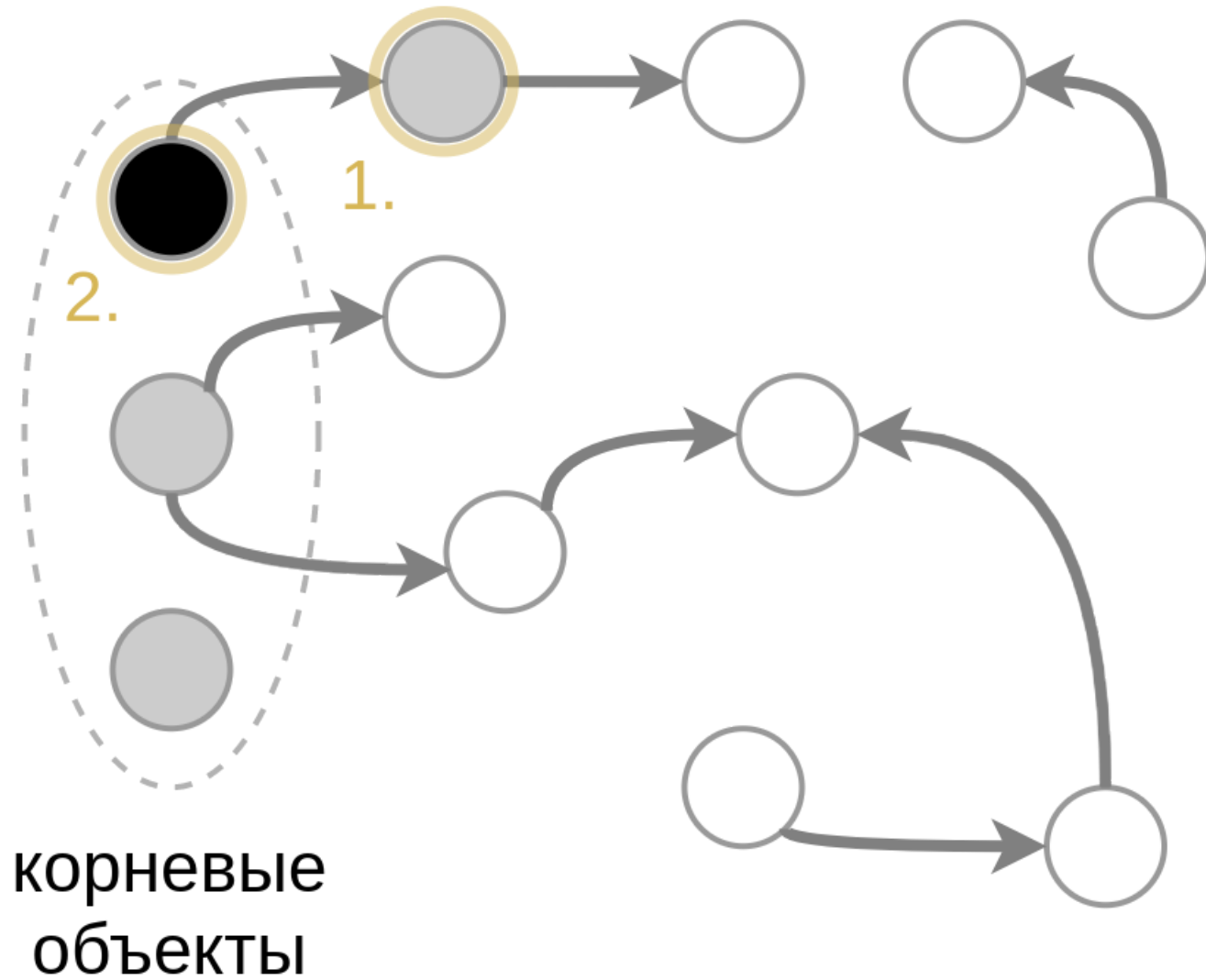
# Начнем издалека

В основе GC в V8 - трёхцветный (tri-color) **алгоритм** отметки  
(Э.В. Дейкстра, 1978)

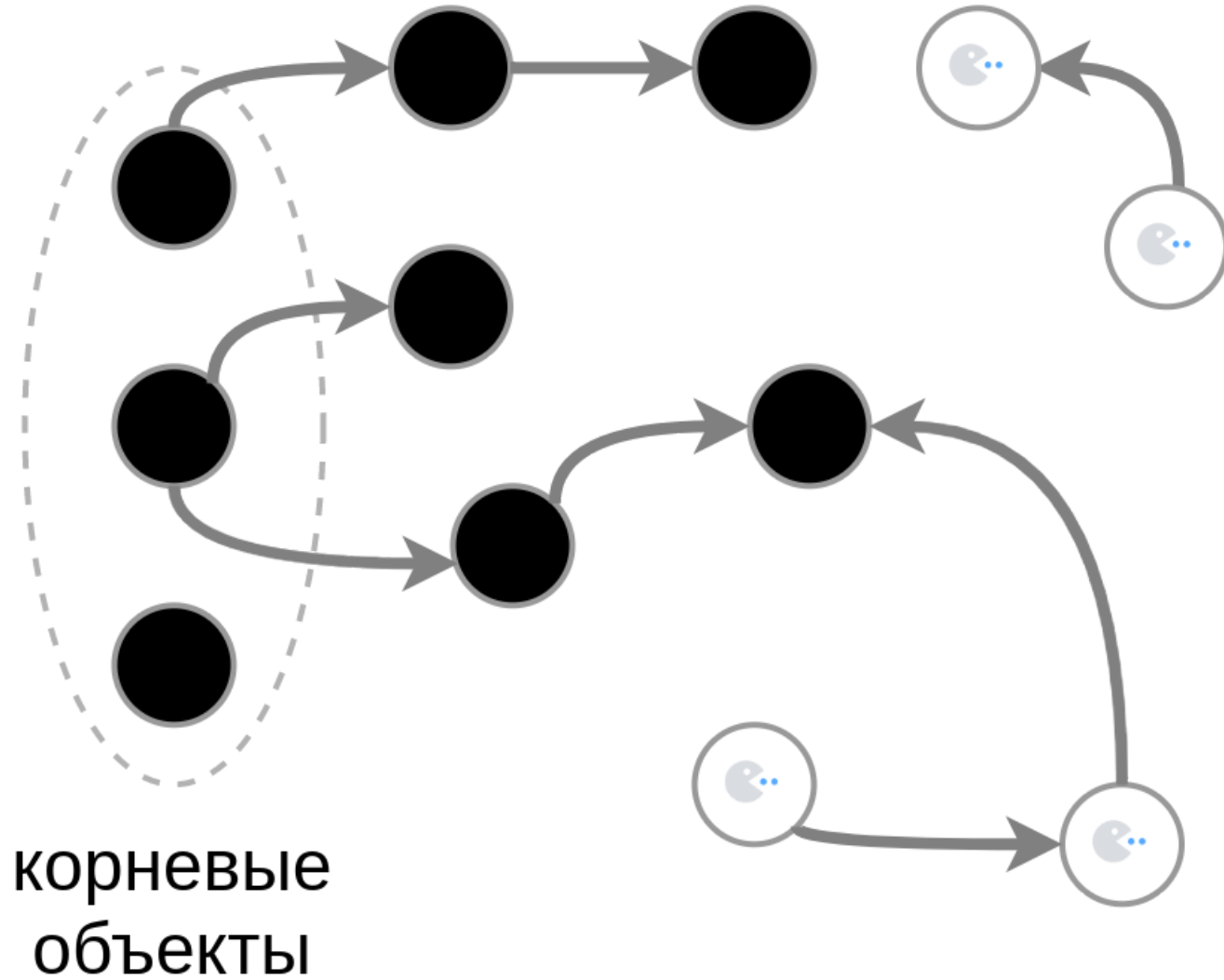
# Tricolor marking



# Tricolor marking

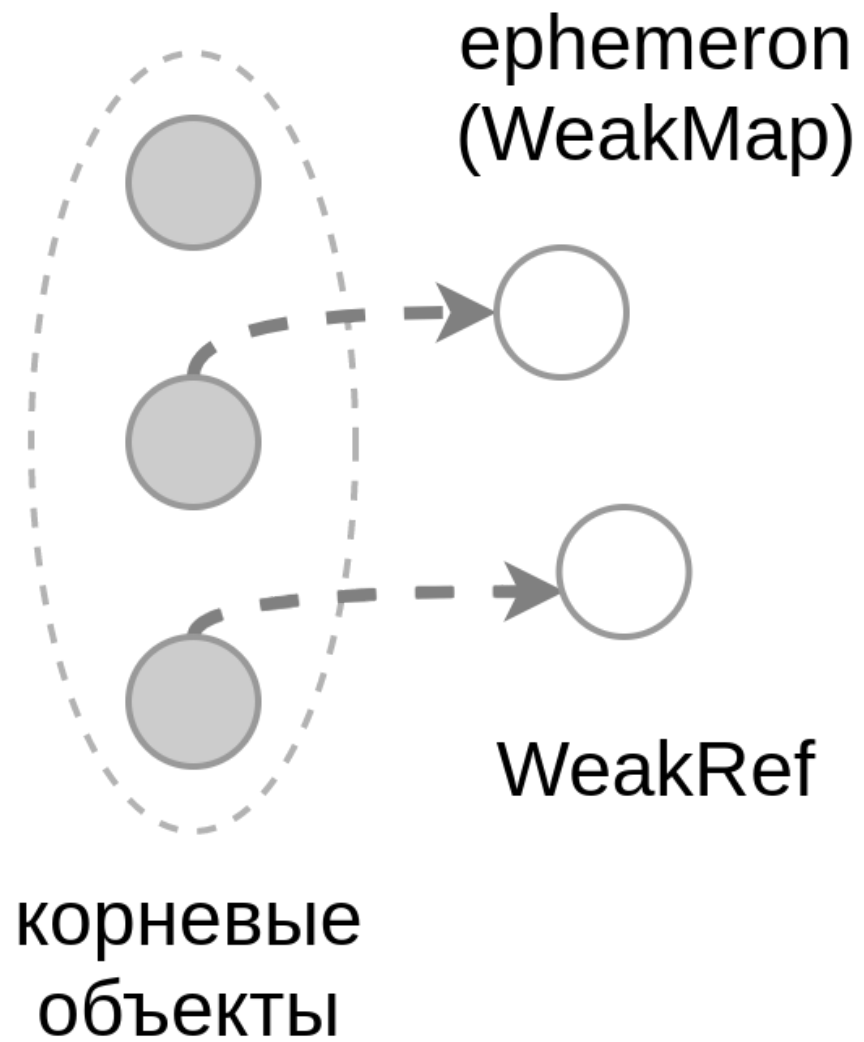


# Tricolor marking



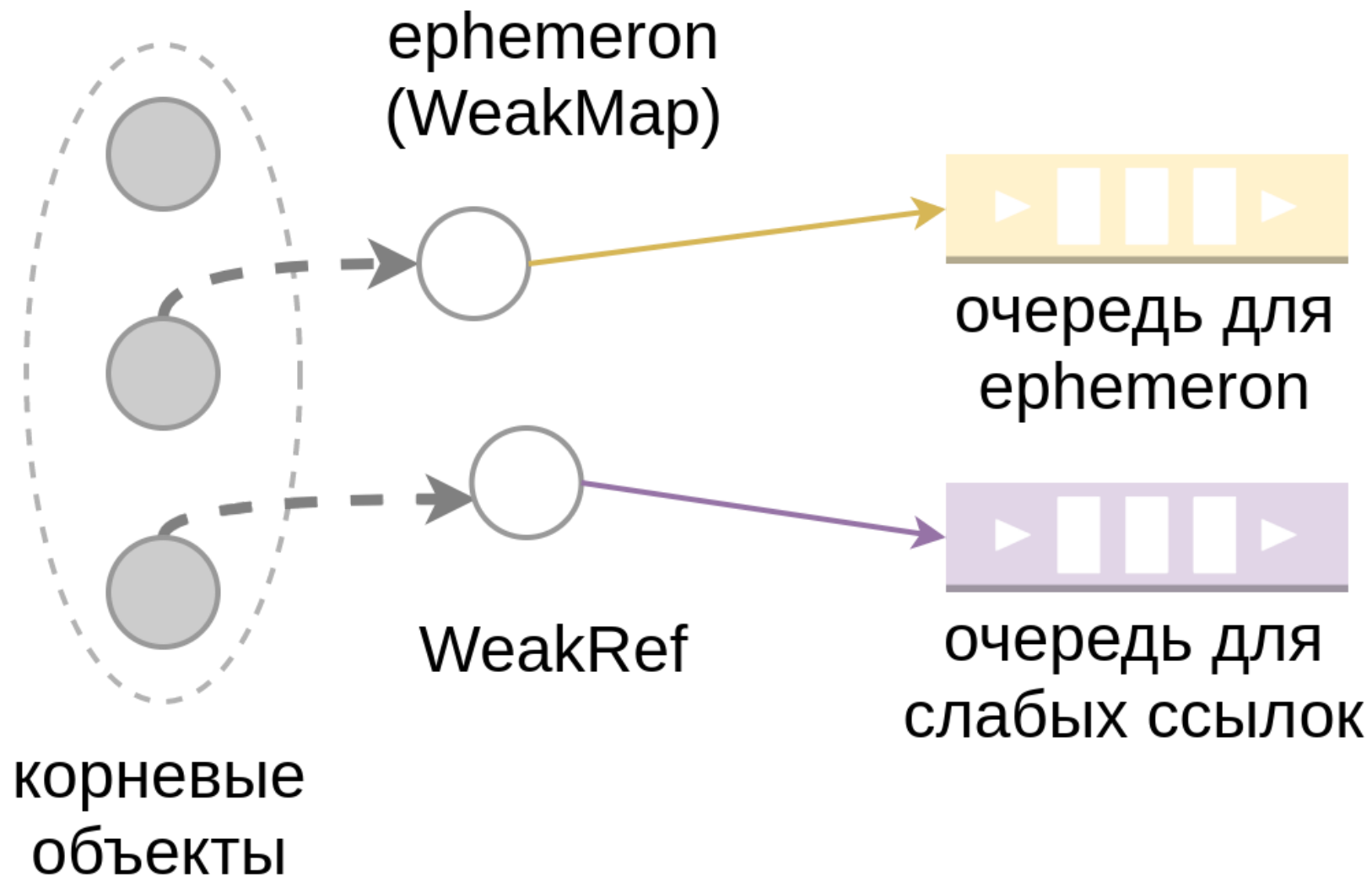
Как сборщик обрабатывает ephemeron'ы?

# Обработка ephemeron





# Обработка ephemeron



```
function convergeEphemeron(queue) {  
  while (true) {  
    let changed = false;  
  
    for (const weakMap of queue) {  
      if (traverseEphemeron(weakMap)) {  
        changed = true;  
        // tri-color marking для помеченных значений  
        propagateAll(/*...*/);  
      }  
    }  
  
    if (!changed) break;  
  }  
}
```

```
function traverseEphemeron(weakMap) {  
  let marked = false;  
  
  // прим.: такого публичного API не существует  
  for (let [key, value] of weakMap) {  
    if (isMarked(key)) {  
      mark(value);  
      marked = true;  
    }  
  }  
  
  return marked;  
}
```

## ➤ WeakMap/WeakSet: особенности реализации

- WeakMap/WeakSet:
  - [EphemeronHashTable](#)
  - [MarkingVisitorBase::VisitEphemeronHashTable](#)
  - [MarkCompactCollector::ProcessEphemeronsUntilFixpoint, ProcessEphemeronsLinear, ProcessEphemeron](#)
- WeakRef:
  - [MarkingVisitorBase::VisitJSWeakRef](#)
  - [MarkCompactCollector::ClearJSWeakRefs](#)

## Что же такое WeakMap/WeakSet?

Формально - "классическая" хеш-таблица с открытой адресацией и квадратичным пробированием

## "Цена" WeakMap/WeakSet

- Когда нет циклов, WeakMap/WeakSet равнозначны Map + WeakRef
- Когда циклы есть, стоимость GC возрастает (вплоть до квадратичной)
  - С другой стороны, Map + WeakRef не может обработать циклы

## WeakMap/WeakSet в сухом остатке

- В основе - хеш-таблица с открытой адресацией + ephemeron
- Алгоритмическая сложность операций: как у "классической" хеш-таблицы
- Алгоритмическая сложность GC:  $O(n)$  ( $O(n^2)$  в худшем случае)



**Спасибо за внимание!**



## Полезные ссылки

- <https://itnext.io/v8-deep-dives-understanding-map-internals-45eb94a183df>
- [http://www.jucs.org/jucs\\_14\\_21/eliminating\\_cycles\\_in\\_weak/jucs\\_14\\_21\\_3481\\_3497\\_barros.pdf](http://www.jucs.org/jucs_14_21/eliminating_cycles_in_weak/jucs_14_21_3481_3497_barros.pdf)
- <https://v8.dev/blog/concurrent-marking>

➤ Bonus unlocked 🏆🏆🏆

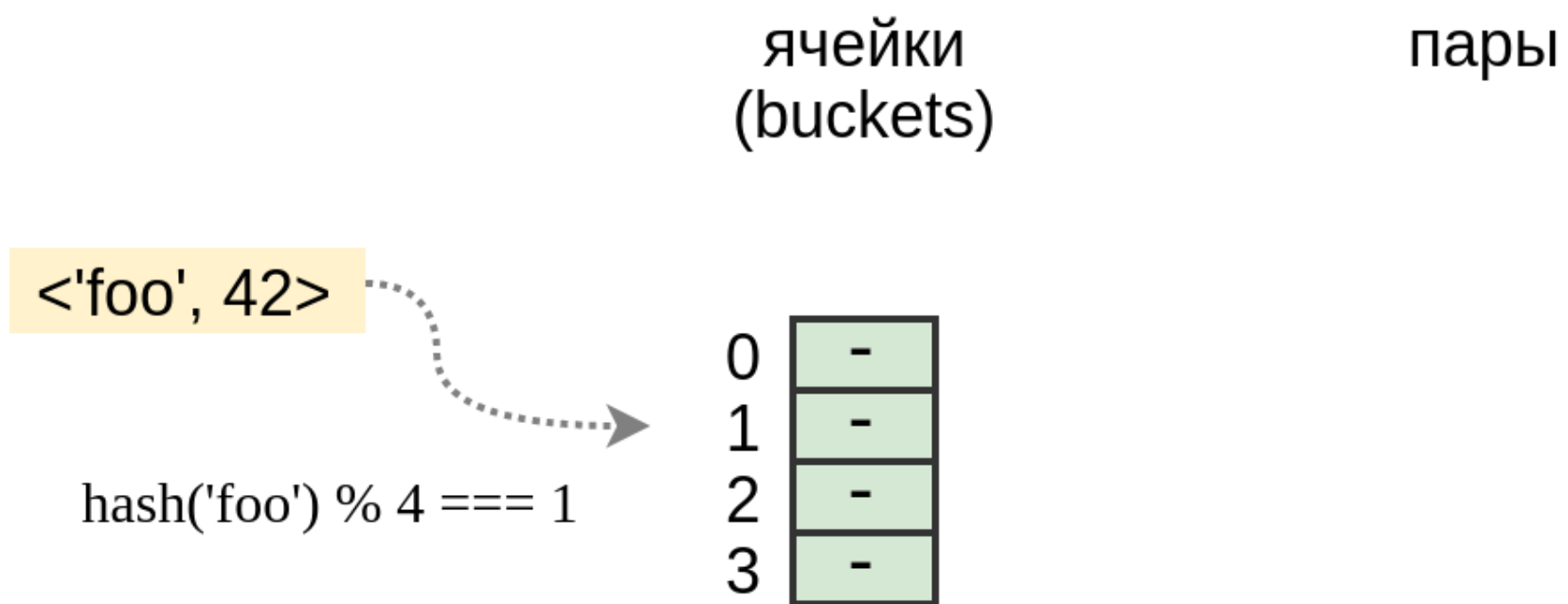
# Пустая хеш-таблица с цепочками

ячейки  
(buckets)

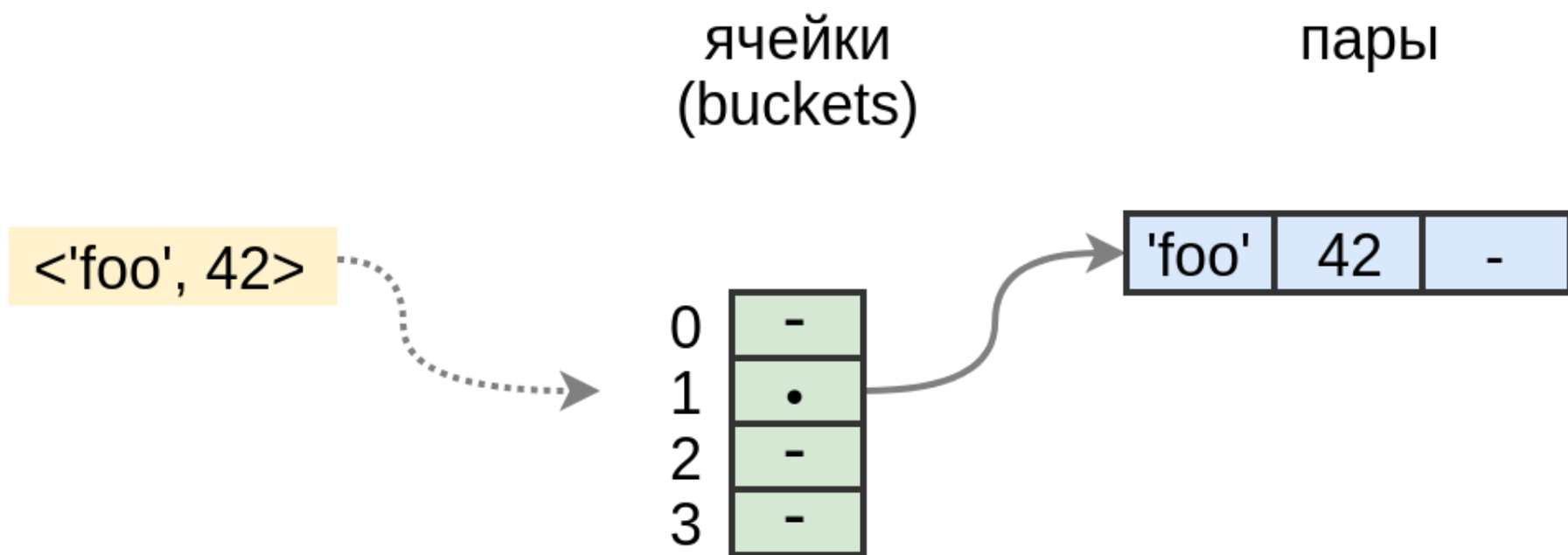
пары

0	-
1	-
2	-
3	-

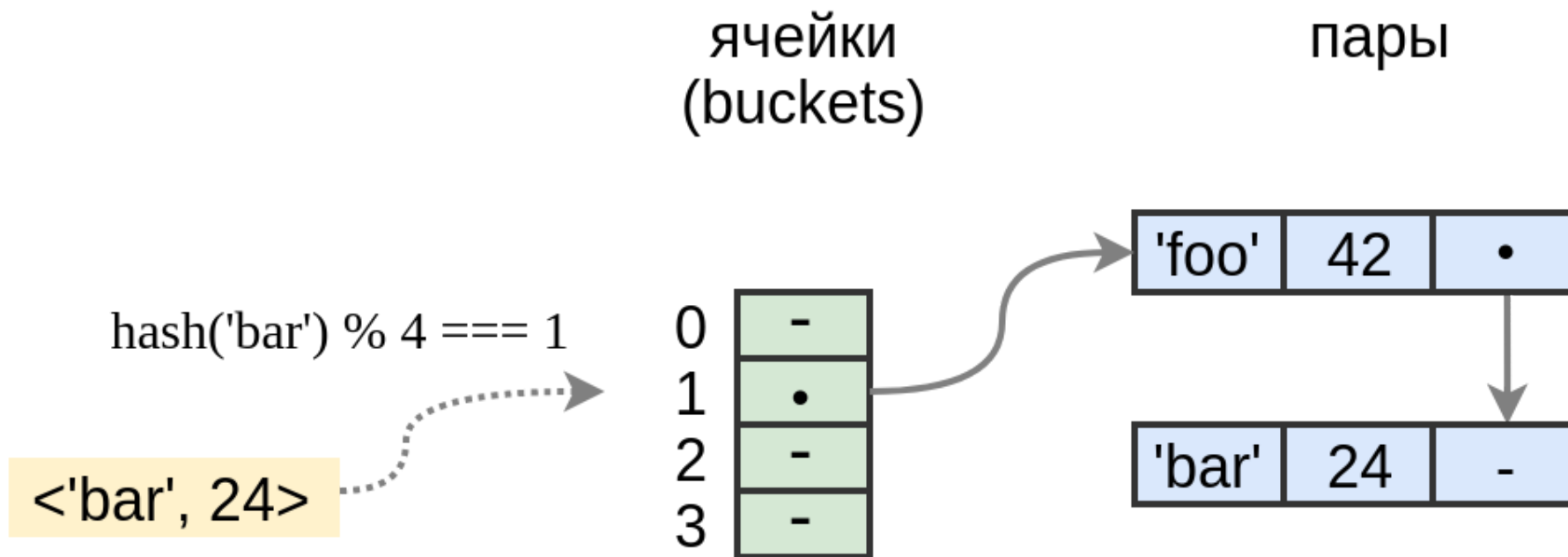
# Вставка



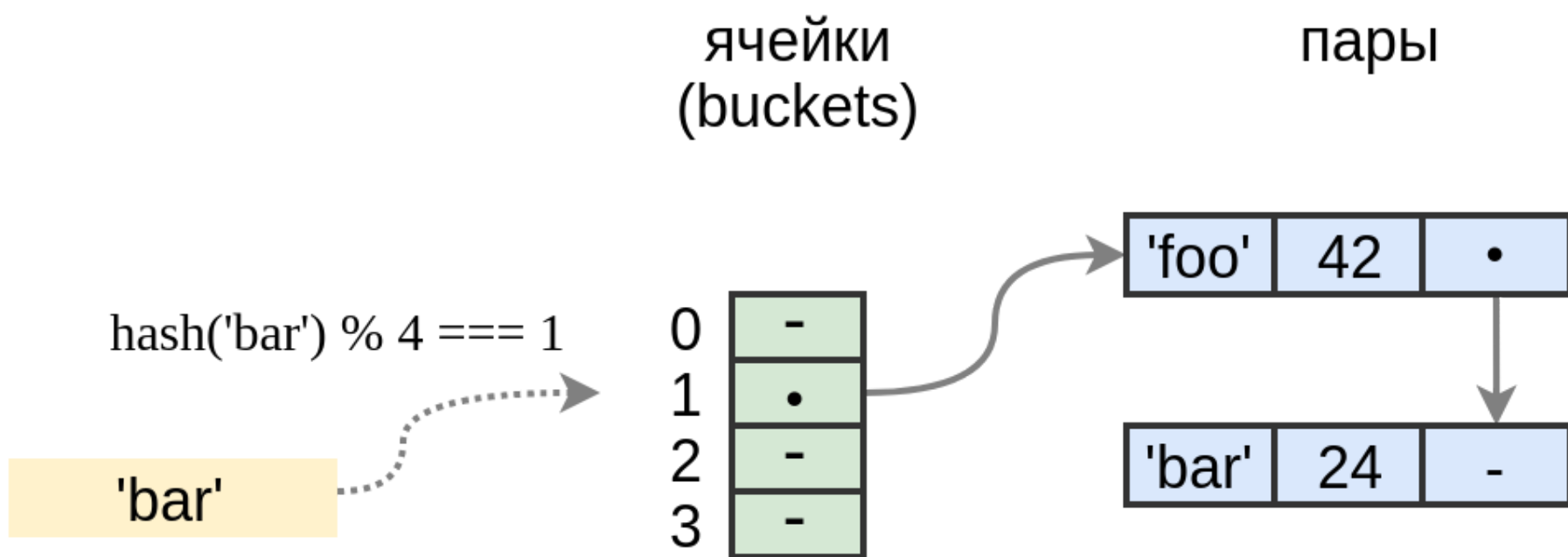
## Результат вставки



# Обработка коллизий



# Поиск





# Map vs Object

	Object	Map
Заранее известная структура	✓	
Гарантированный порядок обхода		✓*
Ключи произвольного типа		✓
Предсказуемая производительность при частых вставках/удалениях		✓

\* С недавних пор (ES 2015, 2020) у объектов порядок обхода тоже гарантирован, но правила обхода сложнее