

# Многопоточные хеш-таблицы

Никита Коваль, JetBrains  
Роман Елизаров, JetBrains

ITMO 2020

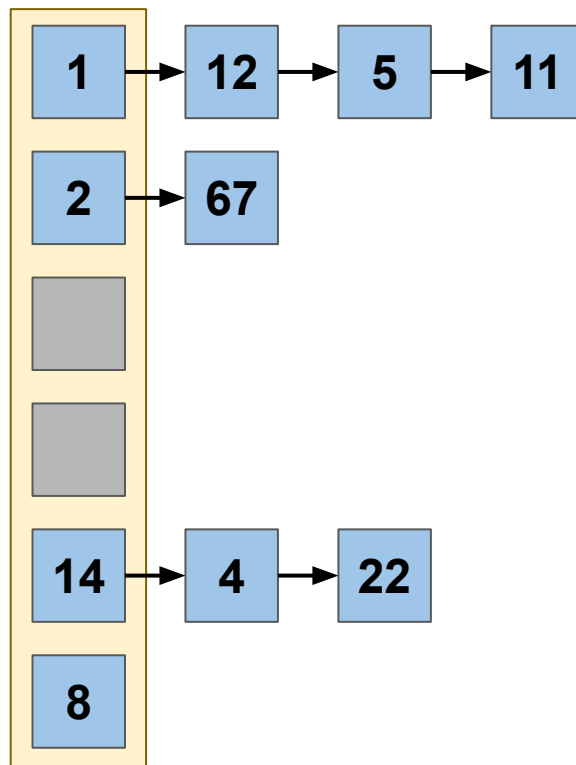
# Что все знают про хеш-таблицы?

- Самая часто используемая структура данных
- Нужны функции эквивалентности и хеша на ключах
  - `equals` и `hashCode` в Java
- Работает за  $O(1)$  в среднем
  - при “хорошей” хеш-функции
- Необходимо перестроение всей таблицы при расширении

# Краткий ликбез

Три базовые операции:

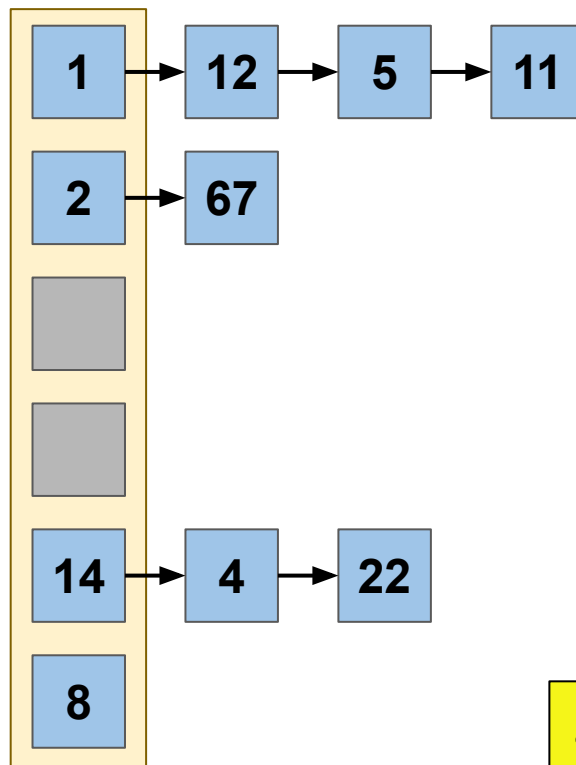
- `add(key, value)`
- `remove(key)`
- `get(key): value`



# Краткий ликбез

Три базовые операции:

- `add(key, value)`
- `remove(key)`
- `get(key): value`

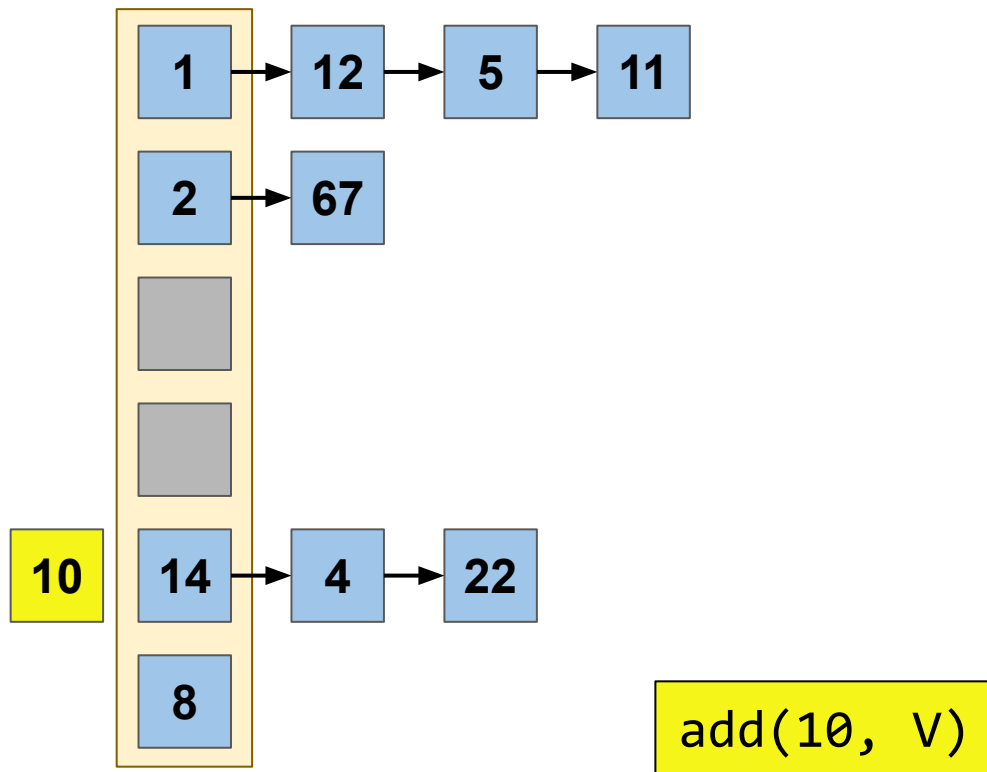


`add(10, V)`

# Краткий ликбез

Три базовые операции:

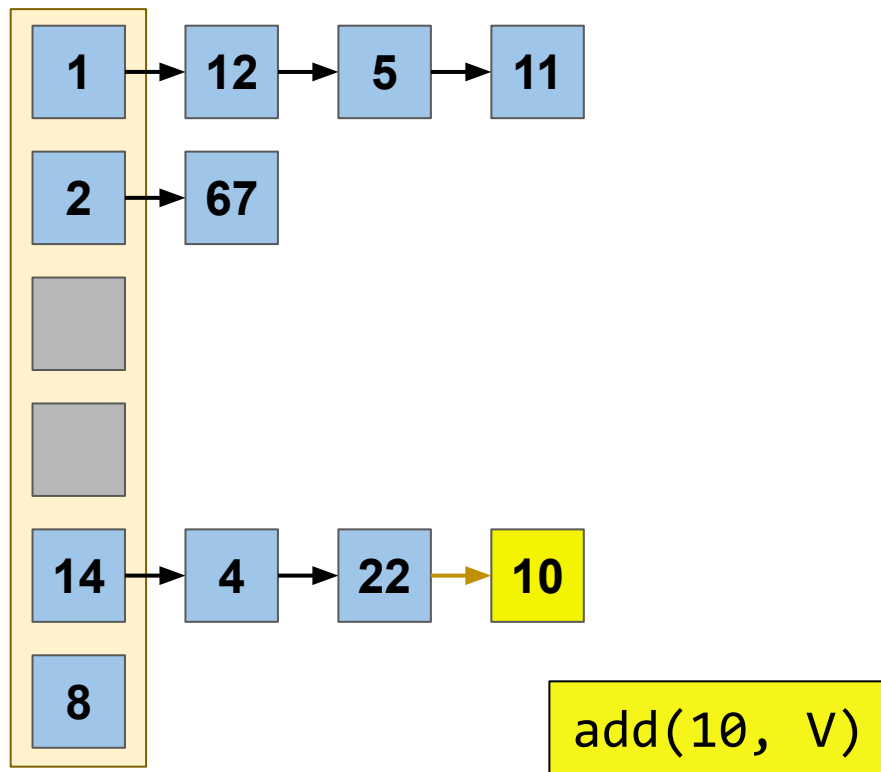
- `add(key, value)`
- `remove(key)`
- `get(key): value`



# Краткий ликбез

Три базовые операции:

- `add(key, value)`
- `remove(key)`
- `get(key): value`

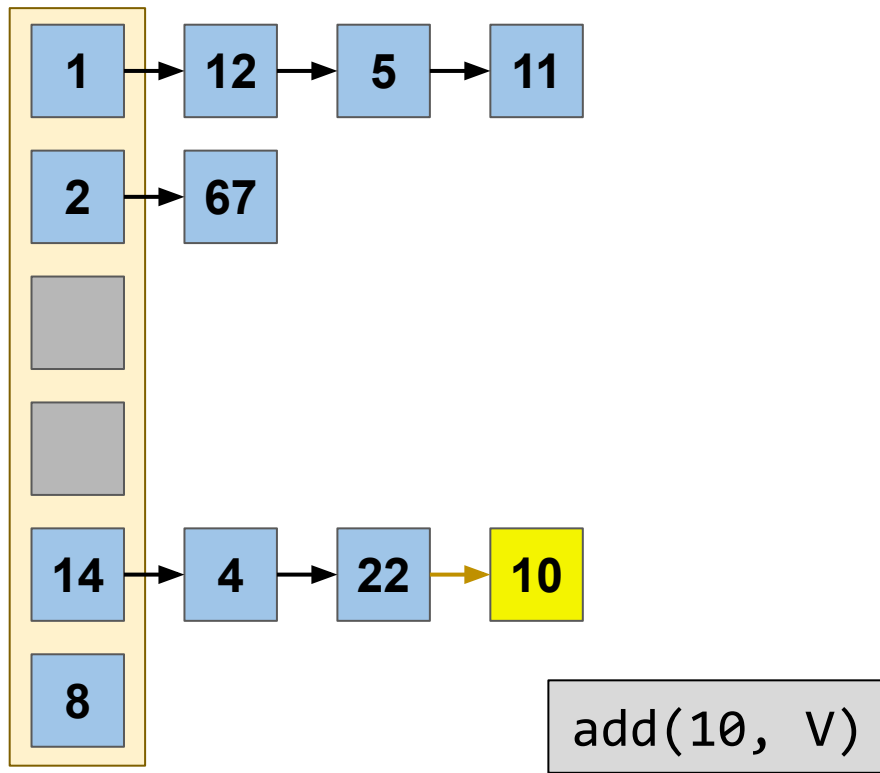


# Краткий ликбез

Три базовые операции:

- `add(key, value)`
- `remove(key)`
- `get(key): value`

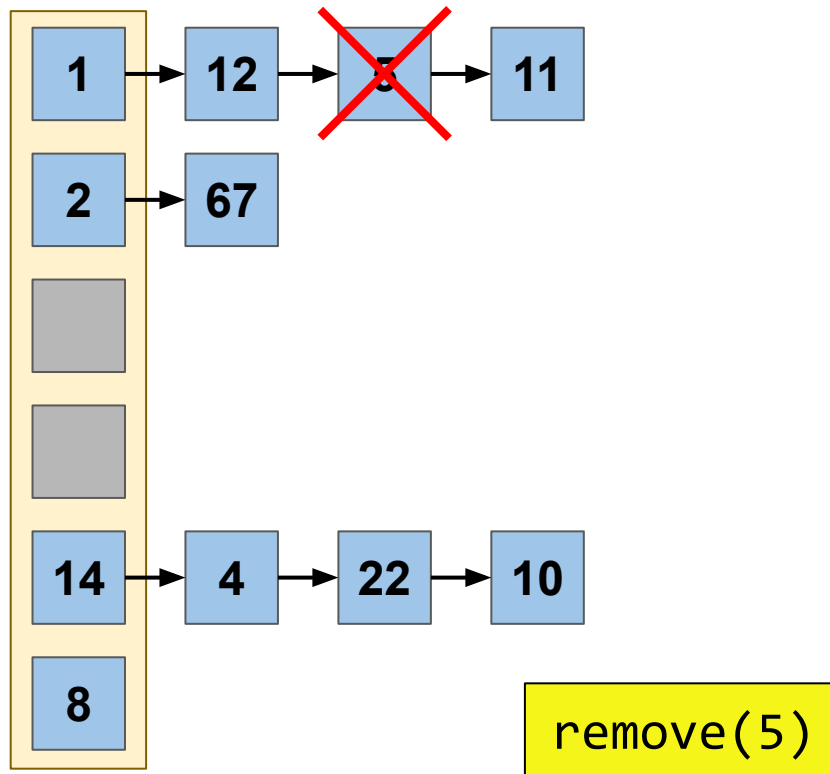
Стало много элементов =>  
создаём таблицу побольше



# Краткий ликбез

Три базовые операции:

- `add(key, value)`
- `remove(key)`
- `get(key): value`

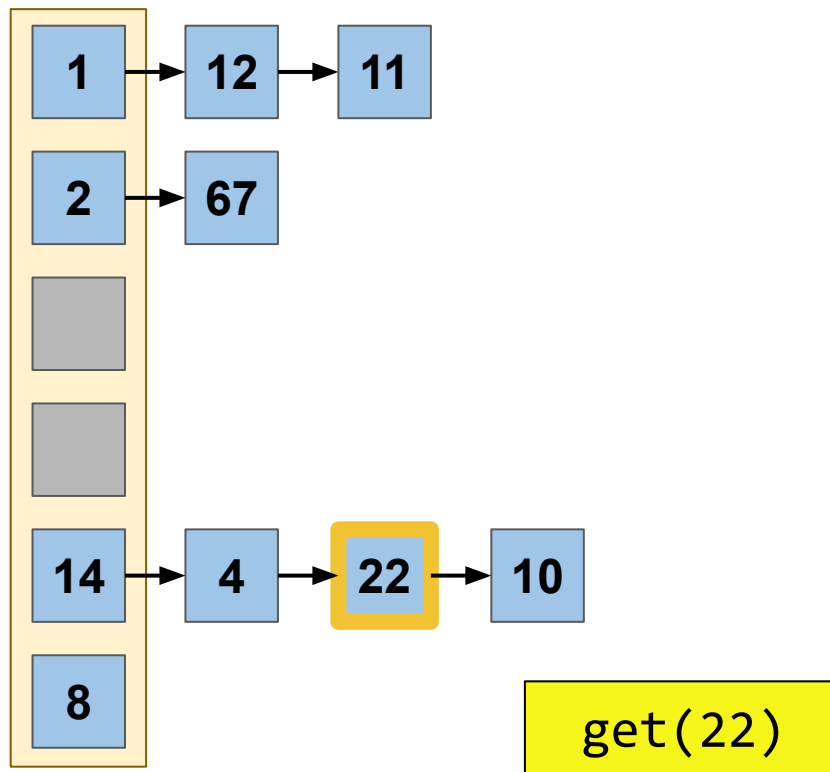




# Краткий ликбез

Три базовые операции:

- `add(key, value)`
- `remove(key)`
- `get(key): value`

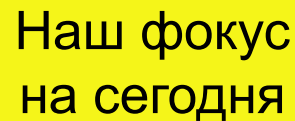


# Где нам нужны хеш-таблицы?

- Кеши
  - много чтений, мало обновлений, почти нет удалений
- Представление данных
  - часто immutable
- Обработка данных
  - количества чтений и обновлений сопоставимы, встречаются удаления

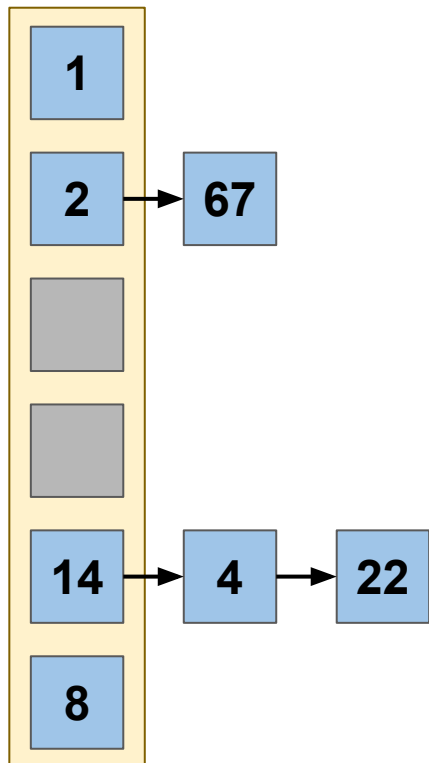
# Где нам нужны хеш-таблицы?

- Кеши
  - много чтений, мало обновлений, почти нет удалений
- Представление данных
  - часто immutable
- Обработка данных
  - количества чтений и обновлений сопоставимы, встречаются удаления



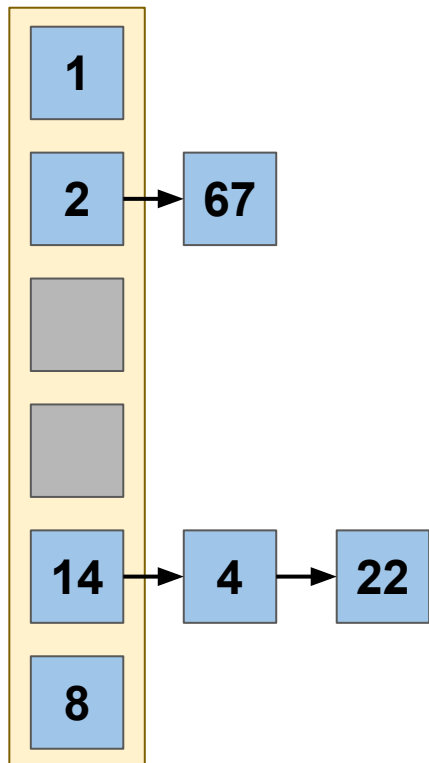
Наш фокус  
на сегодня

# Открытая адресация



Разрешение коллизий  
требуется доп. память

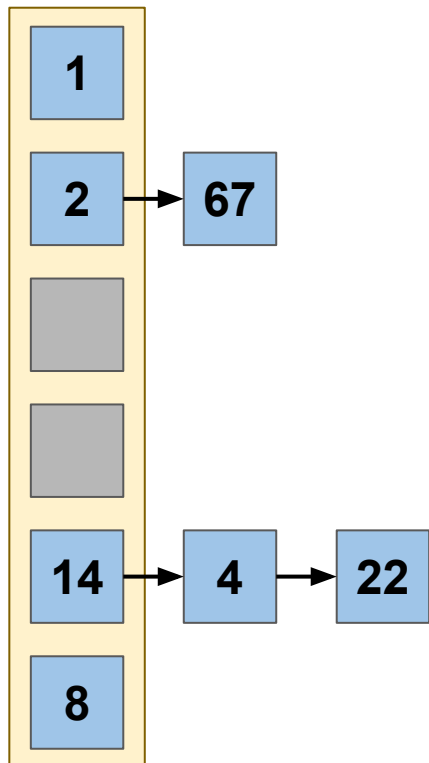
# Открытая адресация



Разрешение коллизий  
требуется доп. память

... а еще cache miss-ы

# Открытая адресация

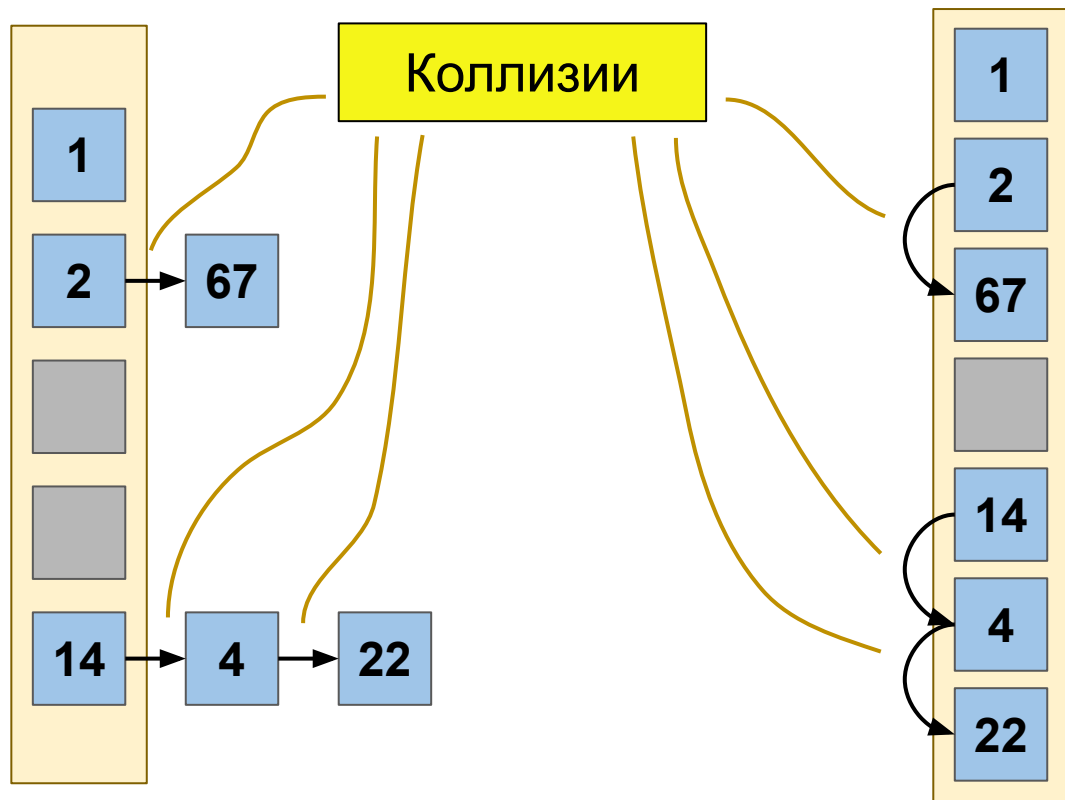


Разрешение коллизий  
требуется доп. память

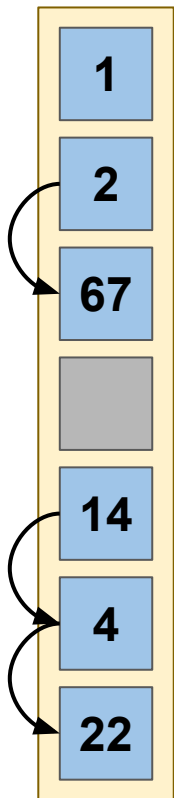
... а еще cache miss-ы

Даёшь открытую  
адресацию!

# Открытая адресация



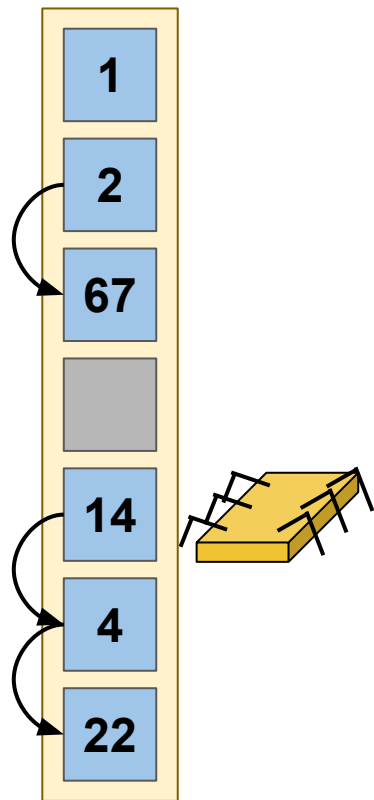
# Открытая адресация



get(22)



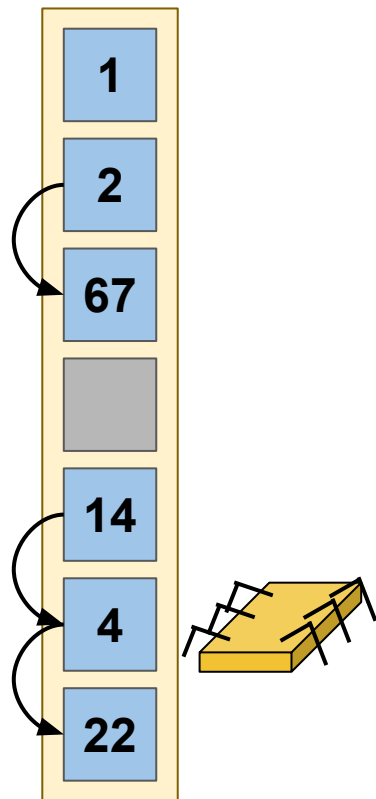
# Открытая адресация



`get(22)`

1. Идём в слот  $\text{hash}(22) \% N$

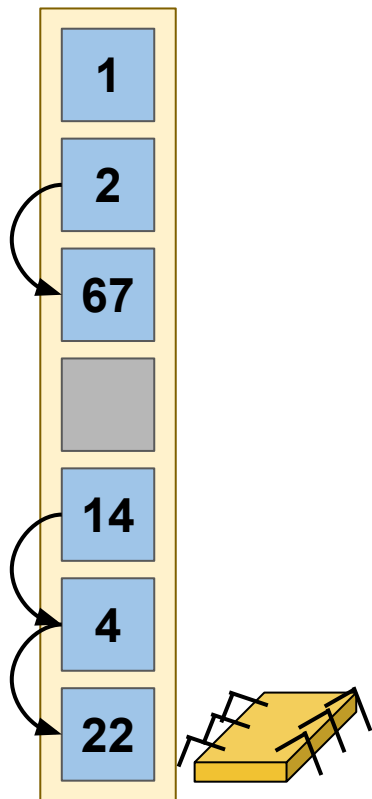
# Открытая адресация



`get(22)`

1. Идём в слот  $\text{hash}(22) \% N$
2. Коллизия  $\Rightarrow$  идём дальше

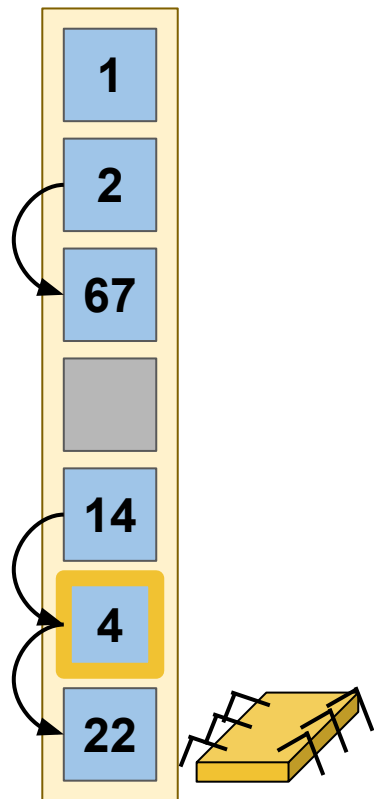
# Открытая адресация



`get(22)`

1. Идём в слот  $\text{hash}(22) \% N$
2. Коллизия  $\Rightarrow$  идём дальше
3. Ура, нашли!

# Открытая адресация

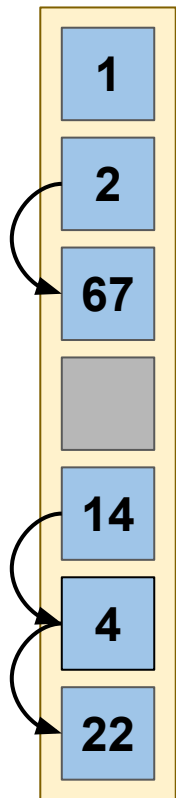


`get(22)`

1. Идём в слот  $\text{hash}(22) \% N$
2. Коллизия  $\Rightarrow$  идём дальше
3. Ура, нашли!

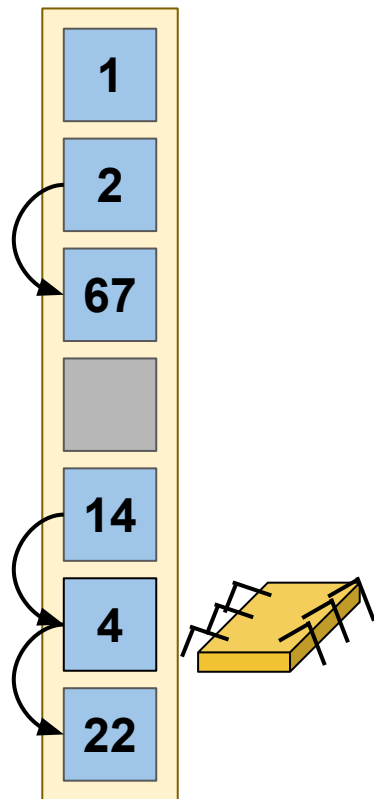
Что будет, если  
удалить `<4>`?

# Открытая адресация



remove(4)

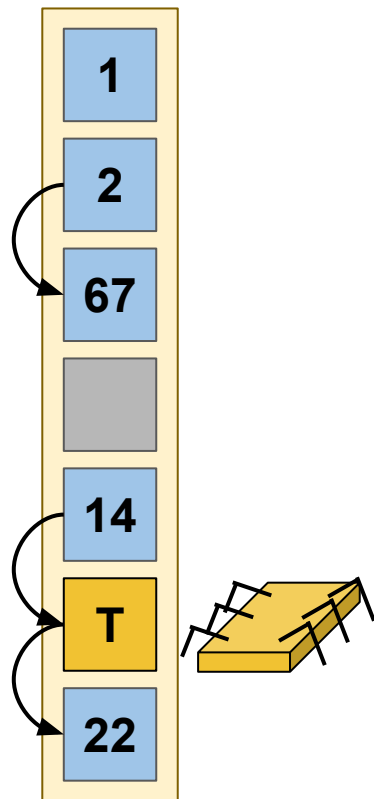
# Открытая адресация



remove(4)

1. Ищем слот для с <4>

# Открытая адресация



`remove(4)`

1. Ищем слот для с  $\langle 4 \rangle$
2. Заменяем на специальное значение “тут был элемент”

# Размер таблицы

Как выбрать “правильный” размер массива?



# Размер таблицы

Как выбрать “правильный” размер массива?

Раз мы делаем  $hash(key) \% N$ ,  
то лучше, если  $N$  - простое число

Теория

# Размер таблицы

Как выбрать “правильный” размер массива?

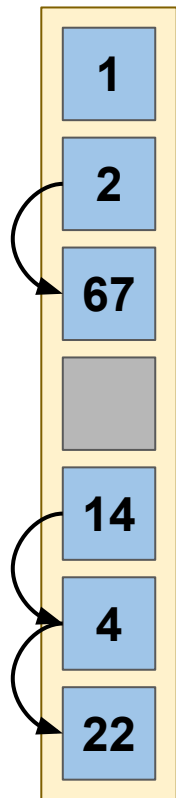
Раз мы делаем  $hash(key) \% N$ ,  
то лучше, если  $N$  - простое число

Практика

Но... 1) операция взятия по модулю дорогая

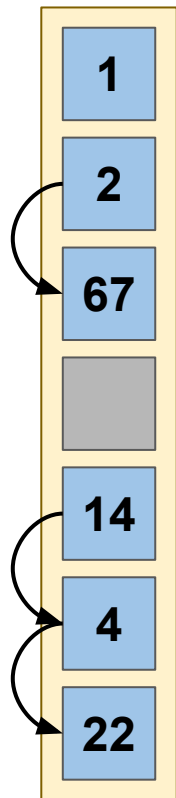
2)  $N$  - степень двойки  $\Rightarrow \%$  заменяется на сдвиг

# Поиск элемента



Куда смотреть в случае коллизий?

# Поиск элемента

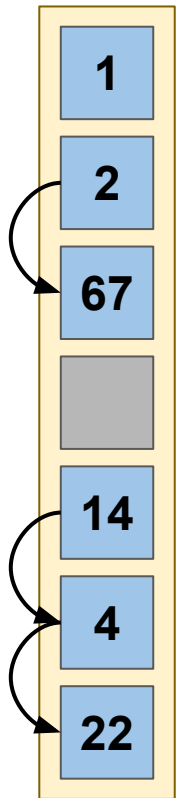


Куда смотреть в случае коллизий?

Лучше “шагать” по простым числам:  
+1, +2, +3, +5, +7, ...

Теория

# Поиск элемента



Куда смотреть в случае коллизий?

Лучше “шагать” по простым числам:  
+1, +2, +3, +5, +7, ...

Однако, соседние ячейки скорее всего  
закешированы, и их просмотр почти бесплатен

Практика

# Lock-free хеш-таблица



## A Lock-Free Wait-Free Hash Table

by Cliff Click

## Lock-free Dynamic Hash Tables with Open Addressing by Gao et al.

### Lock-free Dynamic Hash Tables with Open Addressing

Gao, H.<sup>1</sup>, Grooten, J.F.<sup>2</sup>, Hesselink, W.H.<sup>1</sup>

<sup>1</sup> Department of Mathematics and Computing Science, University of Groningen,  
P.O. Box 800, 9700 AV Groningen, The Netherlands (Email: [hui.win@cs.rug.nl](mailto:hui.win@cs.rug.nl))

<sup>2</sup> Department of Mathematics and Computing Science, Eindhoven University of  
Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands and CWI,  
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands (Email: [jfg@win.tue.nl](mailto:jfg@win.tue.nl))

#### Abstract

We present an efficient lock-free algorithm for parallel accessible hash tables with open addressing, which promises more robust performance and reliability than conventional lock-based implementations. “Lock-free” means that it is guaranteed that always at least one process completes its operation within a bounded number of steps. For a single processor architecture our solution is as efficient as sequential hash tables. On a multiprocessor architecture this is also the case when all processors have comparable speeds. The algorithm allows processors that have widely different speeds or come to a halt. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. The algorithm allows the hash tables to grow and shrink when needed.

Lock-free algorithms are hard to design correctly, even when apparently straightforward. Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. In view of the complexity of the algorithm, we turned to the interactive theorem prover PVS for mechanical support. We employ standard deductive verification techniques to prove around 200 invariance properties of our algorithm, and describe how this is achieved with the theorem prover PVS.

*CR Subject Classification (1991):* D.1 Programming techniques

*AMS Subject Classification (1991):* 68Q22 Distributed algorithms, 68P20 Information storage and retrieval

*Keywords & Phrases:* Hash tables, Distributed algorithms, Lock-free, Wait-free

# Lock-free хеш-таблица by Cliff Click

- Использует открытую адресацию
- Гарантирует lock-freedom
  - Даже можно читать и писать во время перестроения таблицы!
- Построена на простых и понятных идеях

# Single Writer ограничение

Только один поток может менять значения (add, remove)



# Single Writer ограничение

Только один поток может менять значения (add, remove)

Без перестроения таблицы чтения тривиальны

# Single Writer ограничение

Только один поток может менять значения (add, remove)

Без перестроения таблицы чтения тривиальны

Но как читать во время перестроения?

# Single Writer ограничение

Только один поток может менять значения (add, remove)

Без перестроения таблицы чтения тривиальны

Но как читать во время перестроения?

*Алгоритм перехеширования*

1. Создать новую таблицу
2. Скопировать туда элементы
3. Поменять ссылку на таблицу

# Single Writer - жизнь ячейки

*Initial state*

e	null
---	------

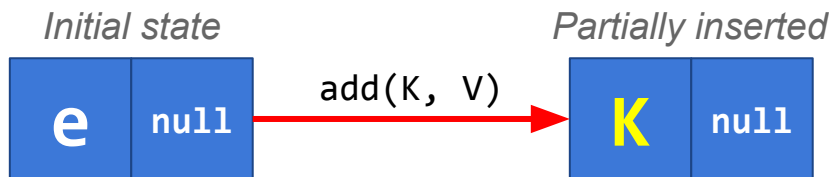
**e** = no key

**K** = some key

**V** = some value

**null** = empty

# Single Writer - жизнь ячейки



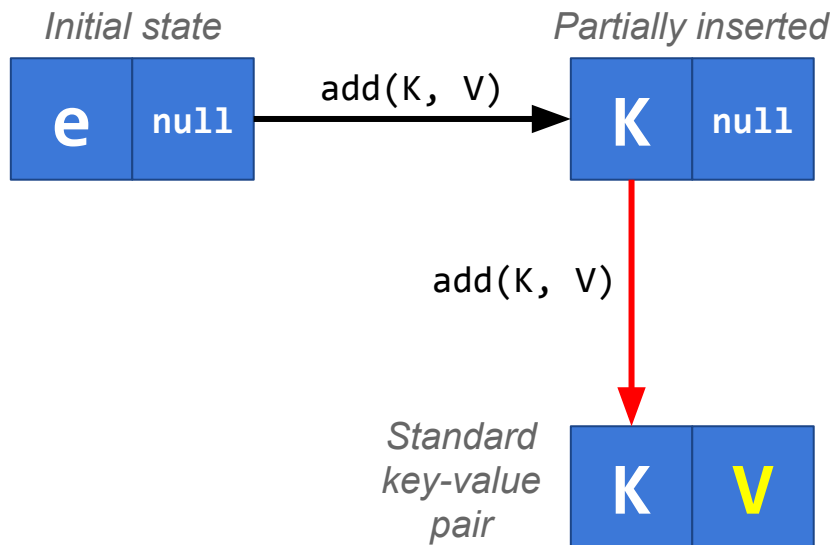
**e** = no key

**K** = some key

**V** = some value

**null** = empty

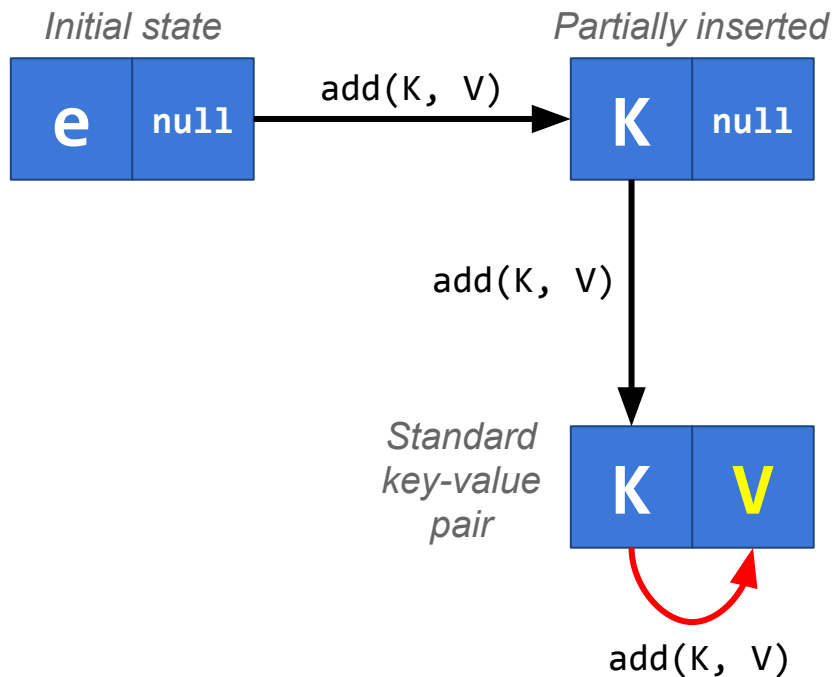
# Single Writer - жизнь ячейки



**e** = no key  
**K** = some key

**V** = some value  
**null** = empty

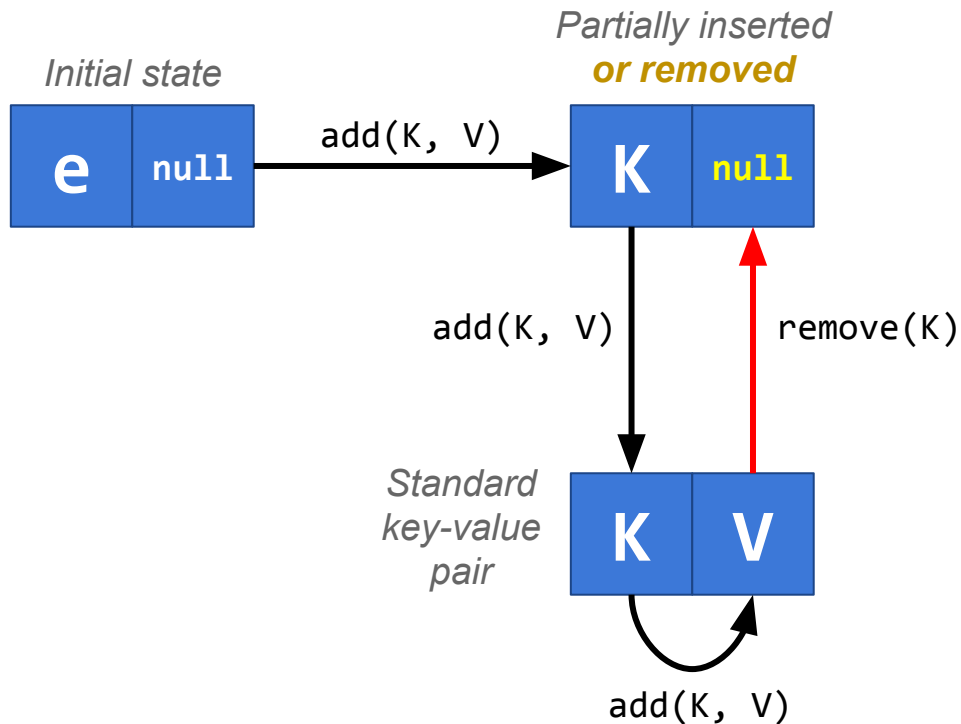
# Single Writer - жизнь ячейки



**e** = no key  
**K** = some key

**V** = some value  
**null** = empty

# Single Writer - жизнь ячейки



**e** = no key  
**K** = some key

**V** = some value  
**null** = empty

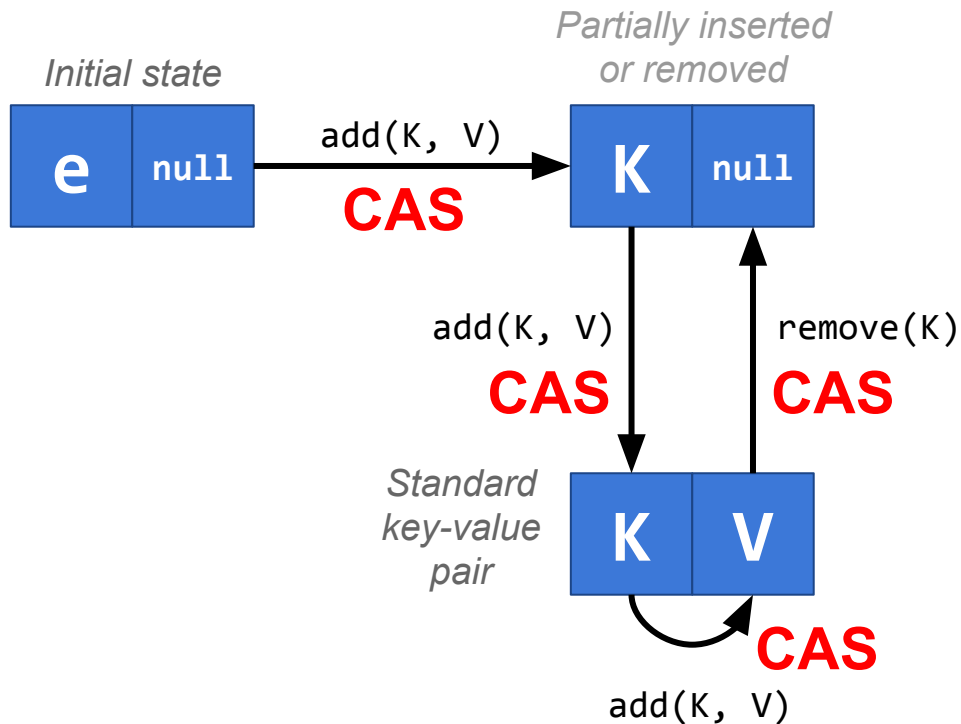


# Multiple Writers

Хотим сделать полноценную lock-free хеш-таблицу...

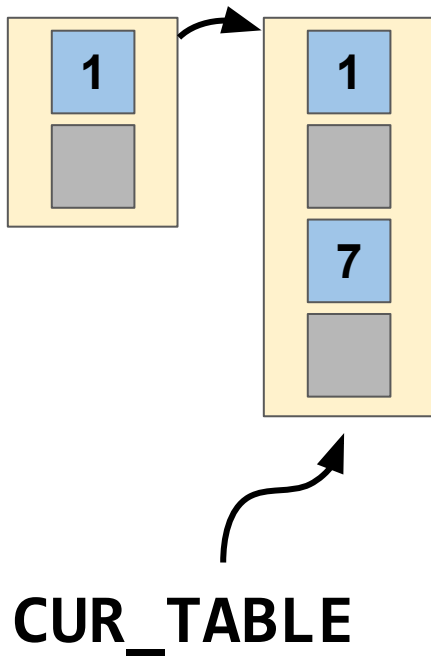
1. Как модифицировать ячейки из разных потоков?
2. Как организовать перехешерование?

# Жизнь ячейки без перехеширования



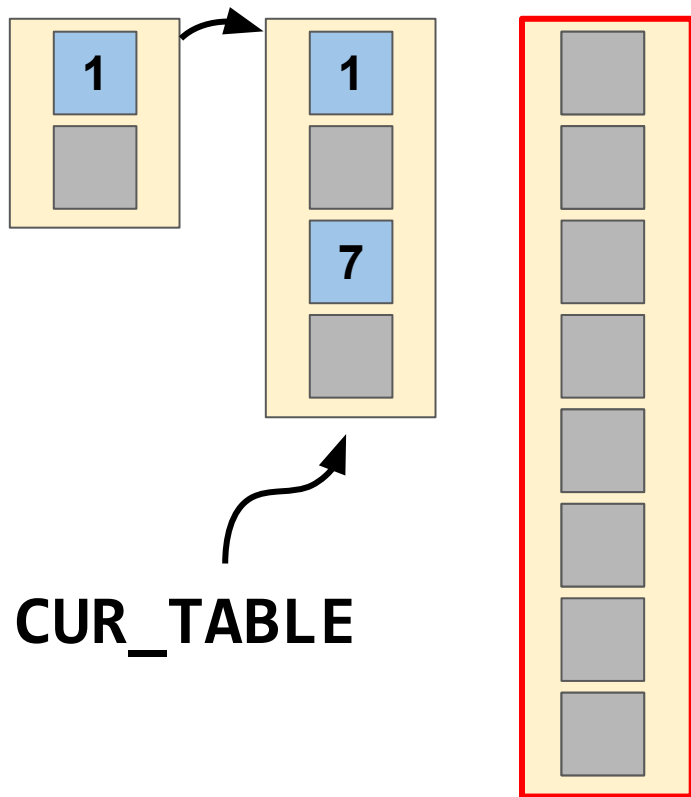
Все переходы  
выполняются с  
помощью **CAS**

# Перехеширование - как организовать?



Каждая таблица хранит ссылку на следующую

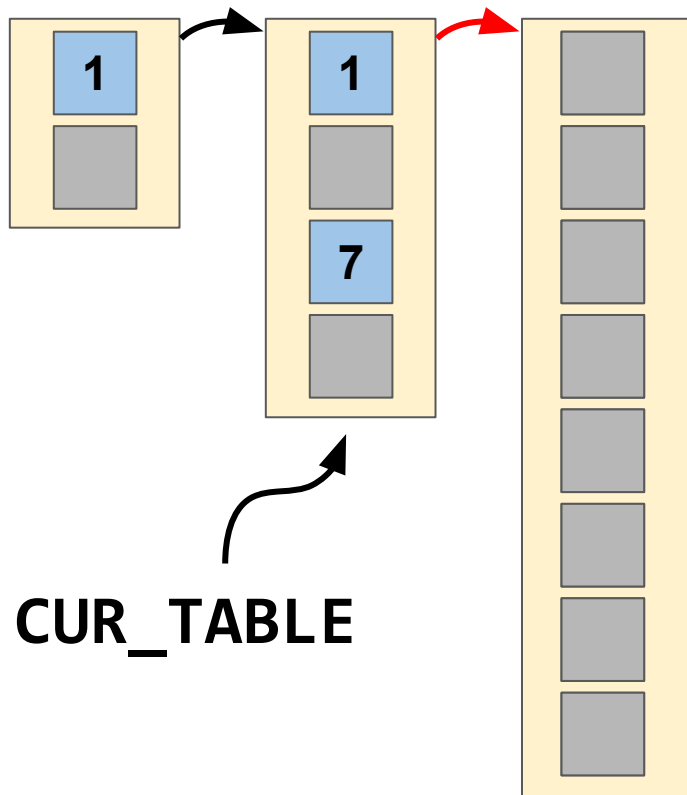
# Перехеширование - как организовать?



Каждая таблица хранит ссылку на следующую

1. Создаём новую таблицу

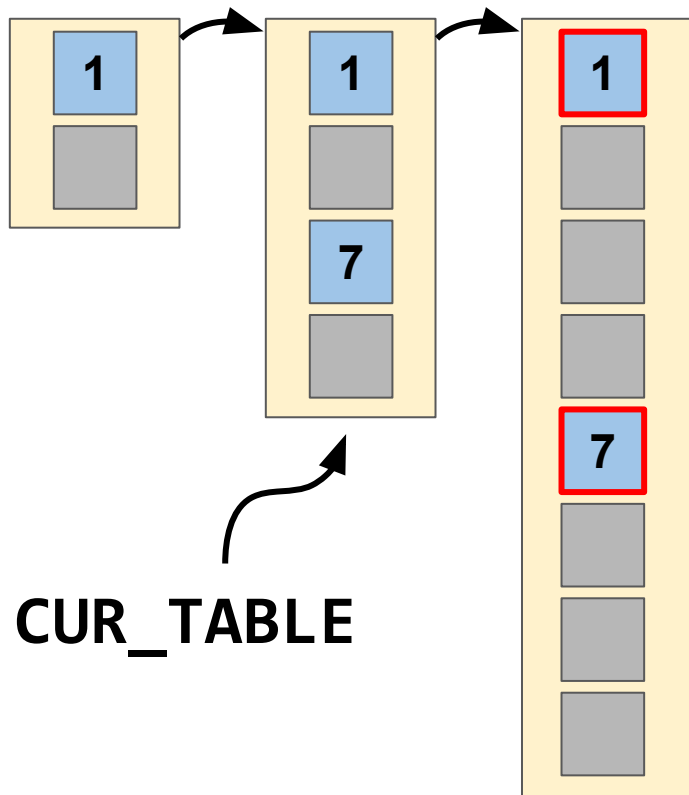
# Перехеширование - как организовать?



Каждая таблица хранит ссылку на следующую

1. Создаём новую таблицу
2. **CAS**-ом записываем ссылку на неё в NEXT текущей

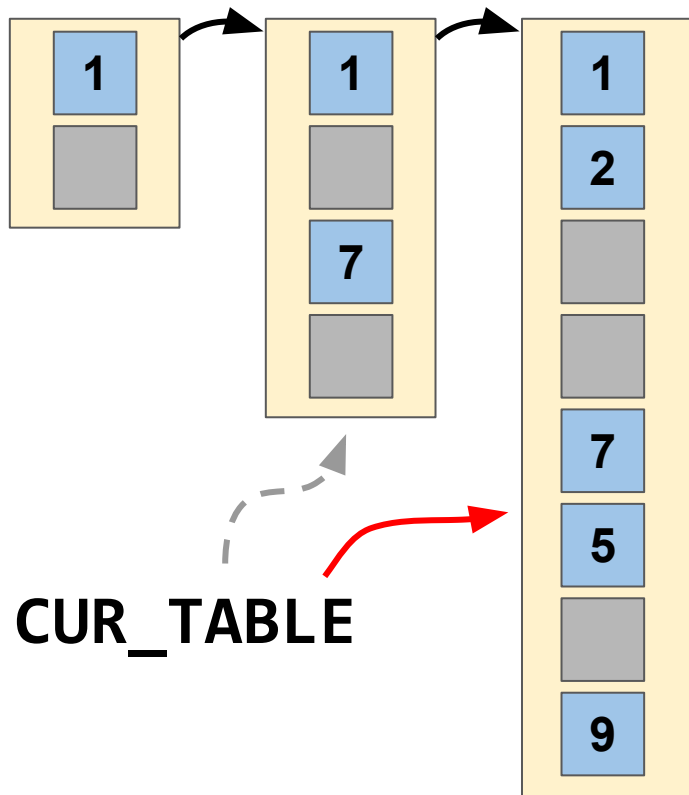
# Перехеширование - как организовать?



Каждая таблица хранит ссылку на следующую

1. Создаём новую таблицу
2. **CAS**-ом записываем ссылку на неё в NEXT текущей
3. Перемещаем все элементы

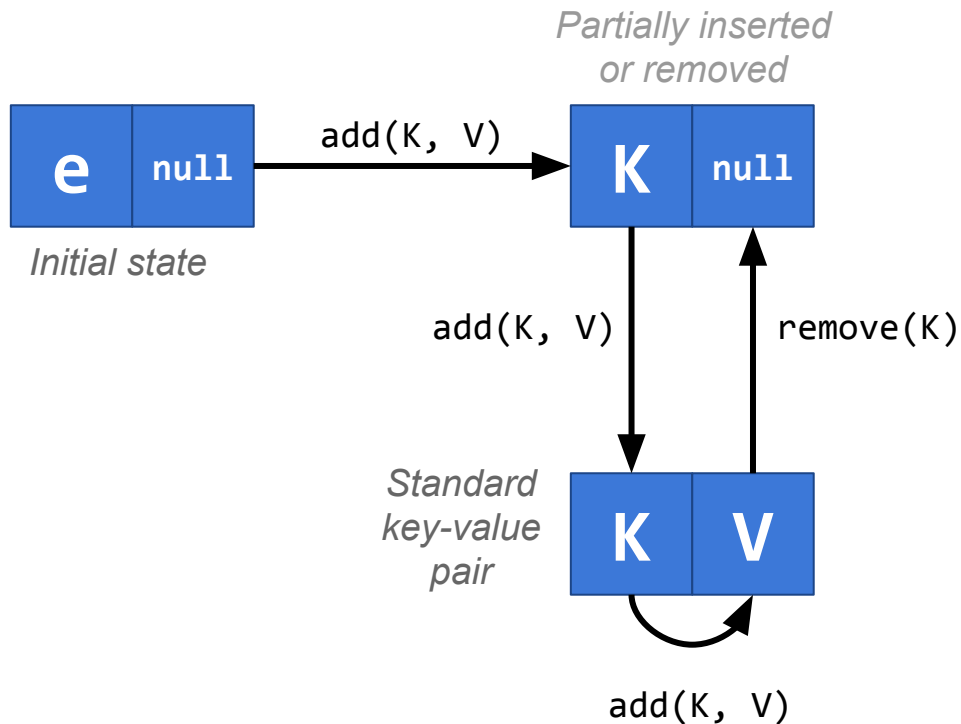
# Перехеширование - как организовать?



Каждая таблица хранит ссылку на следующую

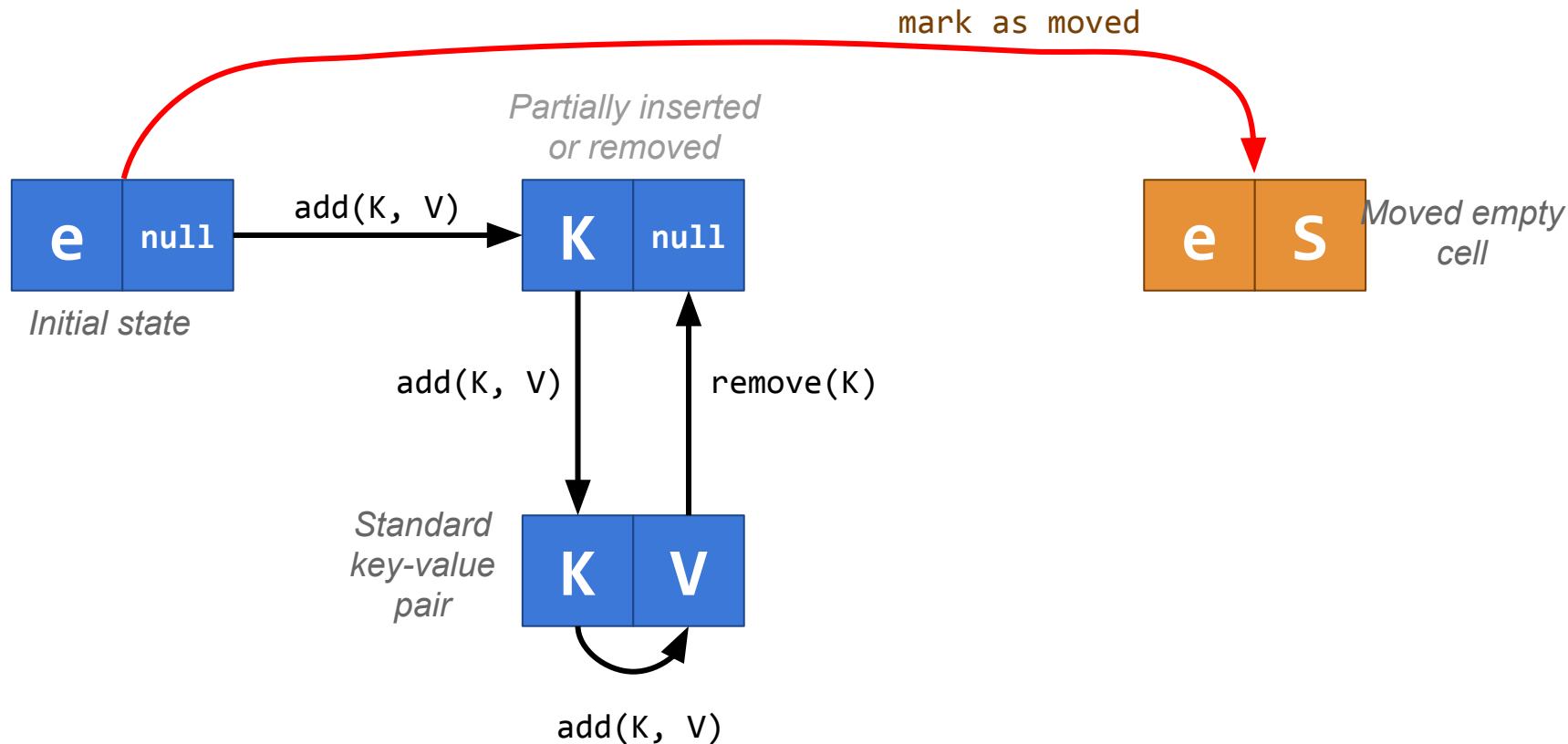
1. Создаём новую таблицу
2. **CAS**-ом записываем ссылку на неё в NEXT текущей
3. Перемещаем все элементы
4. **CAS**-ом обновляем ссылку на текущую таблицу

# Как переносить элементы?

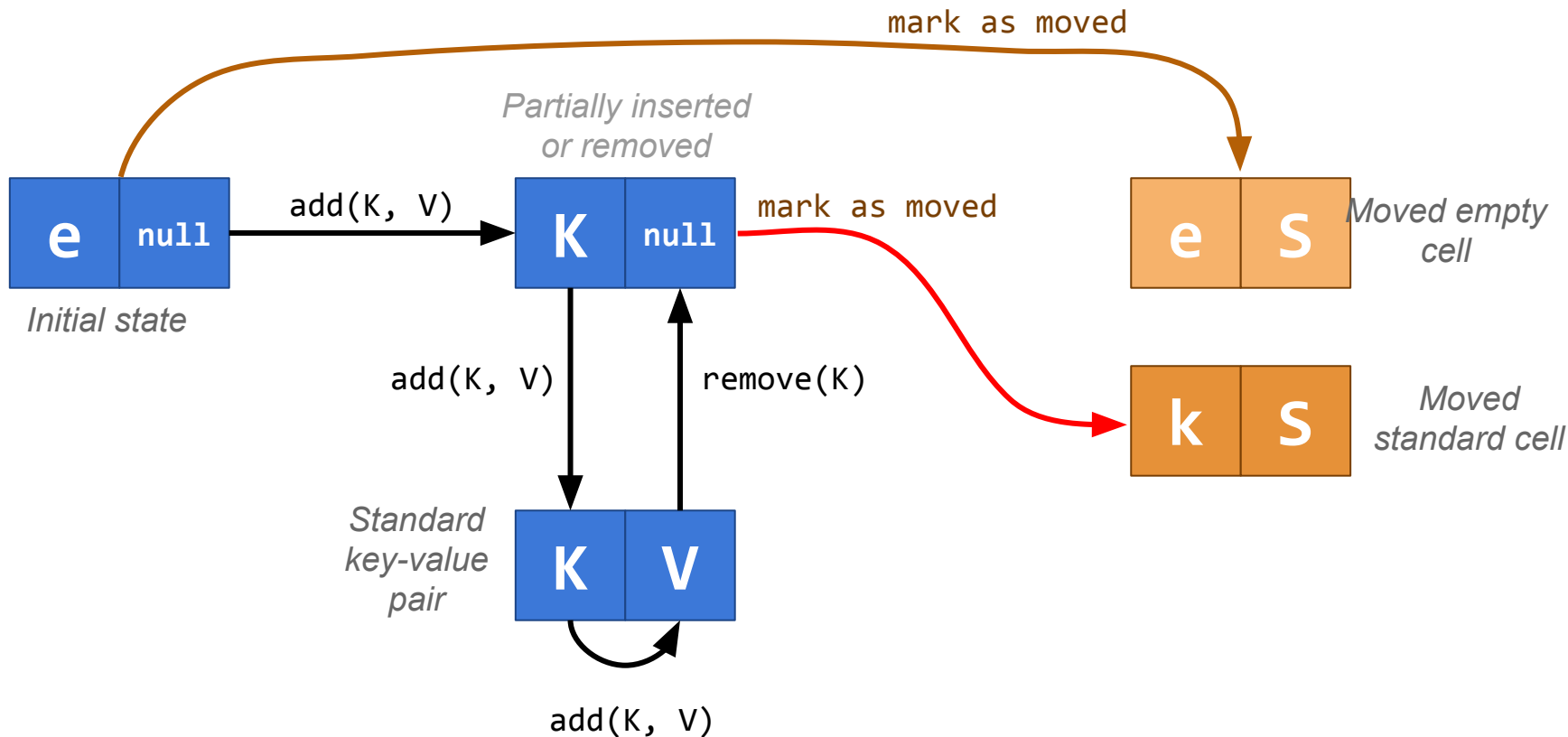




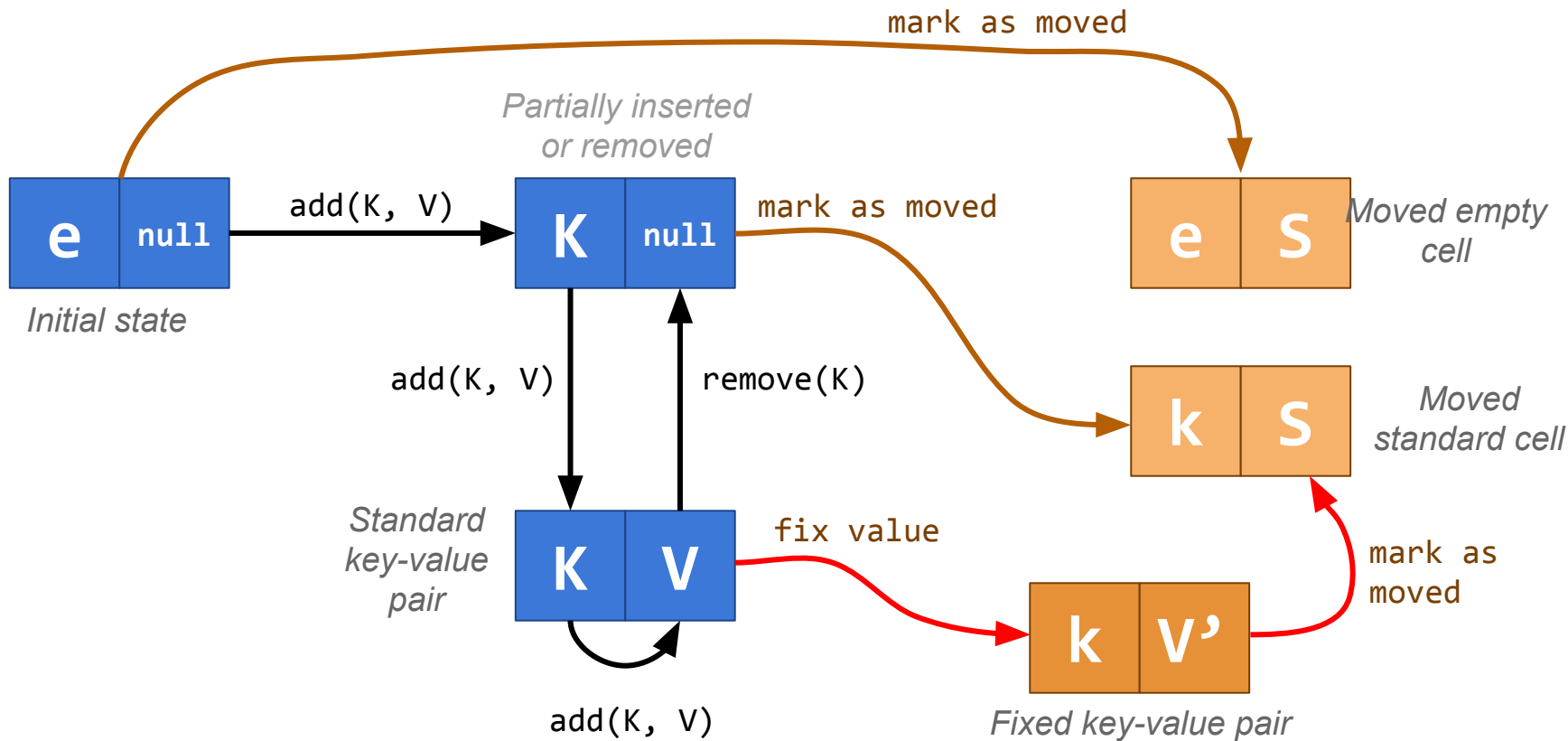
# Как переносить элементы?



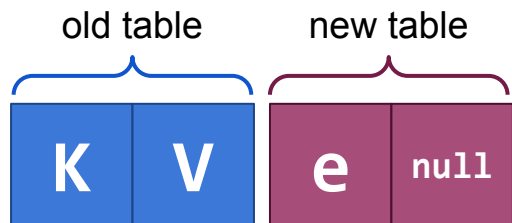
# Как переносить элементы?



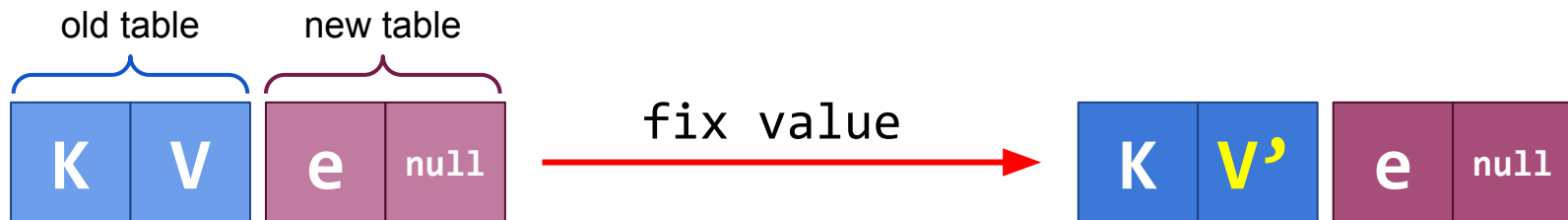
# Как переносить элементы?



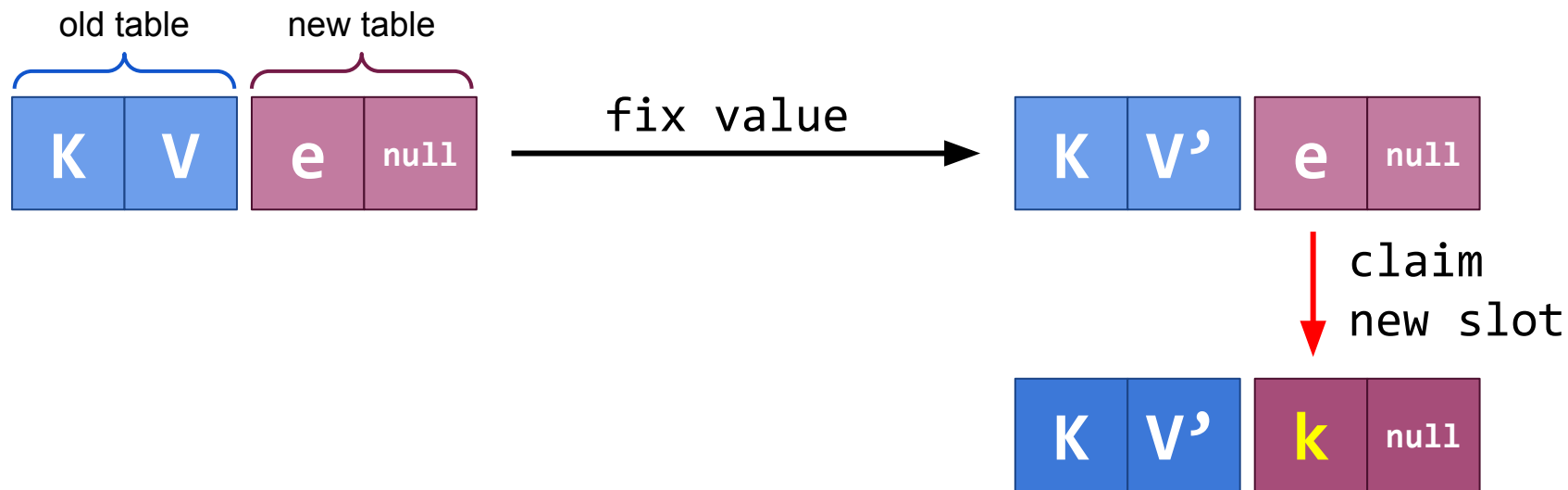
# Перенос элемента подробнее



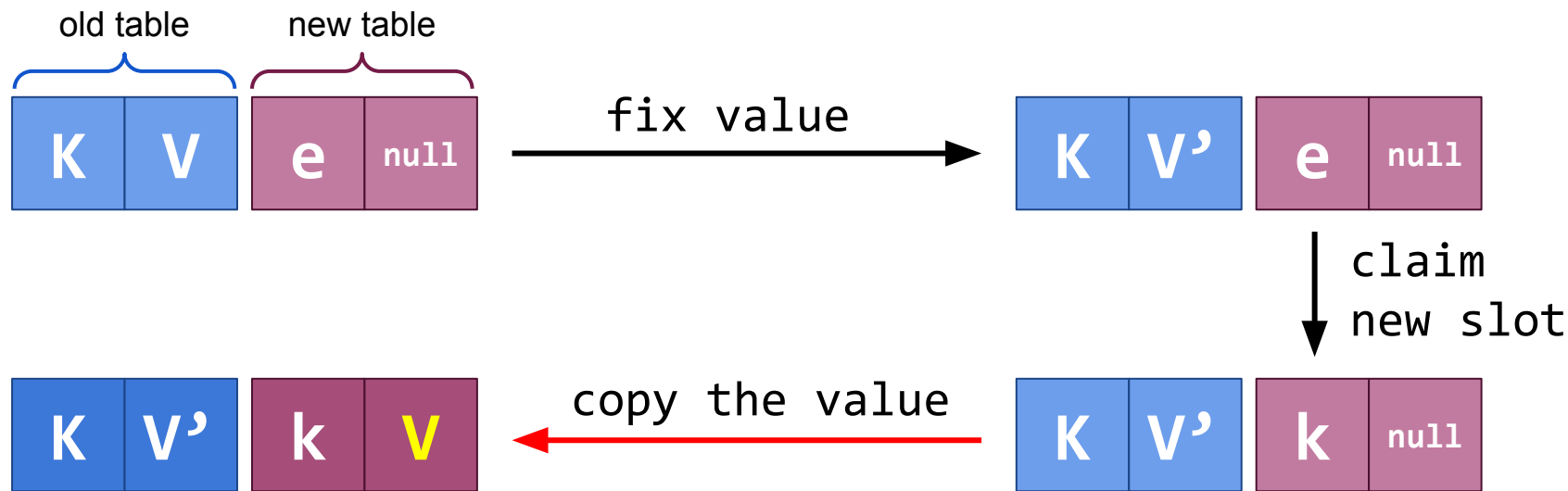
# Перенос элемента поподробнее



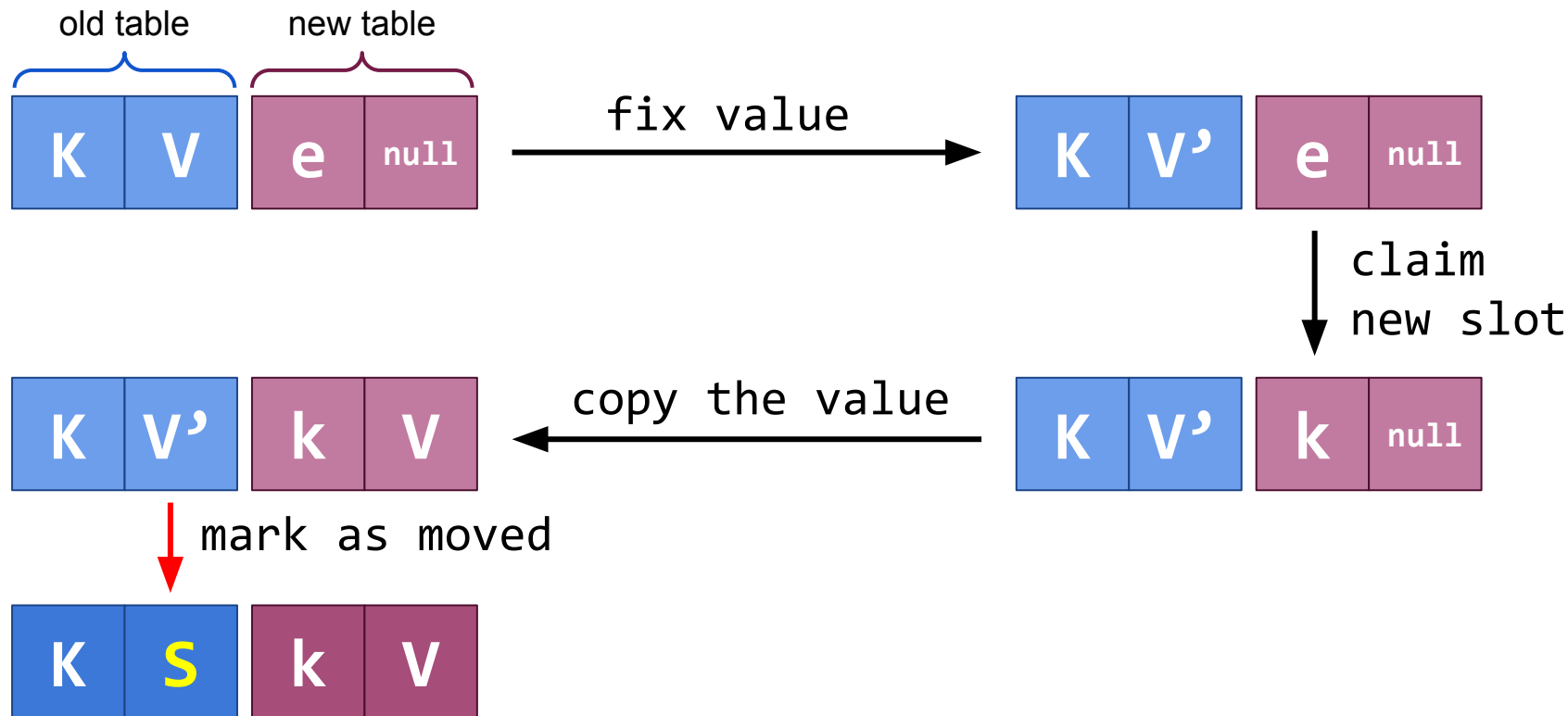
# Перенос элемента подробнее



# Перенос элемента подробнее



# Перенос элемента подробнее





И зачем мне это всё знать?

# Числа в качестве ключей

В кешах зачастую нужна Map-а вида Long -> Object

# Числа в качестве ключей

В кешах зачастую нужна Map-а вида Long -> Object

В [github.com/boundary/high-scale-lib](https://github.com/boundary/high-scale-lib) есть такая:

NonBlockingHashMapLong

# Числа в качестве ключей

В кешах зачастую нужна Map-а вида Long -> Object

В [github.com/boundary/high-scale-lib](https://github.com/boundary/high-scale-lib) есть такая:

NonBlockingHashMapLong

Есть “но”: `int idx = (int)(key & (len-1)); // The first slot`

Что будет, если все  
ключи имеют вид  
[ ???...?111..1 ] ?

# Считаем хеши правильно!

Нужно умножить на большое простое число:

```
int idx = (int)((key * MAGIC) & (len-1));
```

# Считаем хеши правильно!

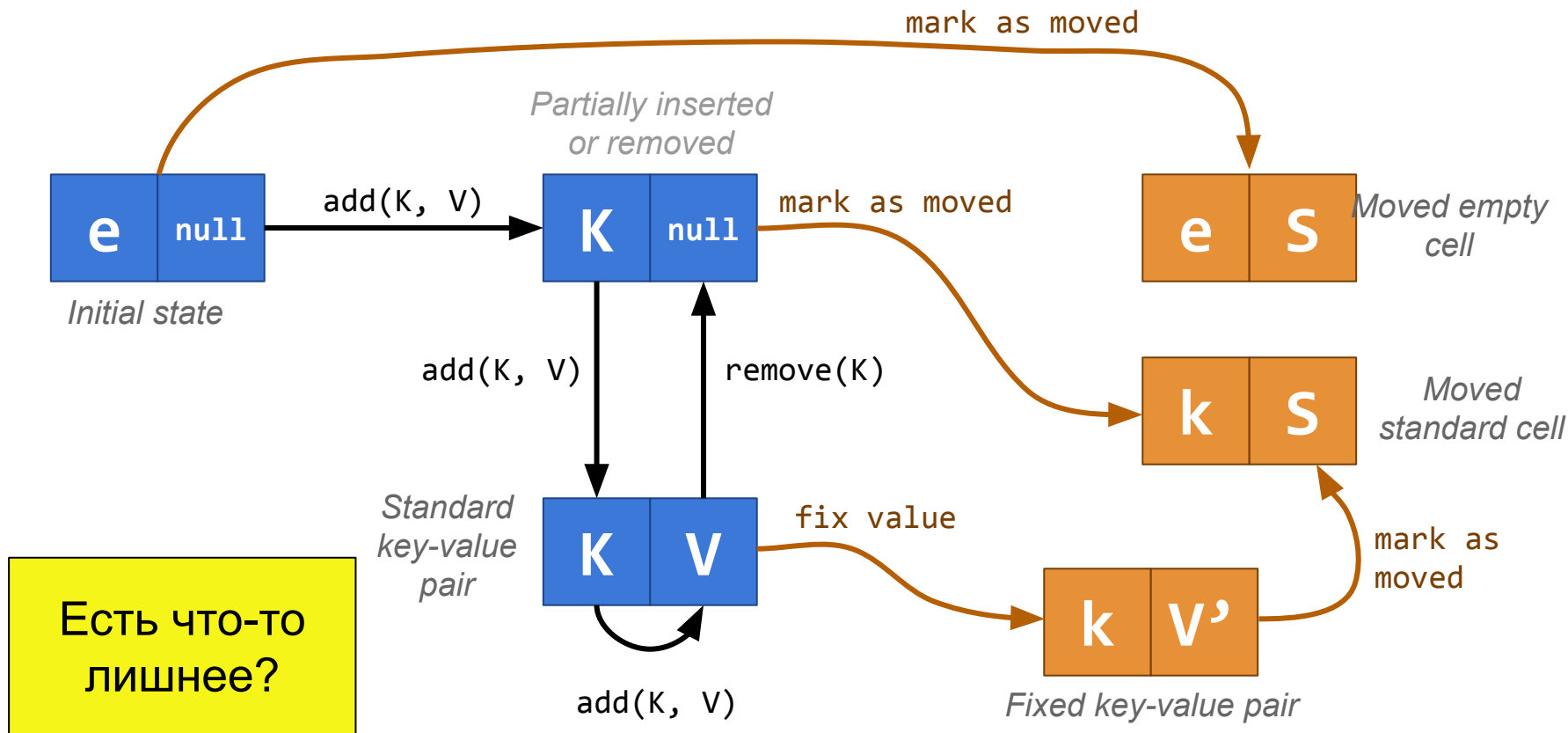
Нужно умножить на большое простое число:

```
int idx = (int)((key * MAGIC) & (len-1));
```

Еще можно использовать MurmurHash,  
много подробностей тут: [elizarov.livejournal.com/25221.html](http://elizarov.livejournal.com/25221.html)

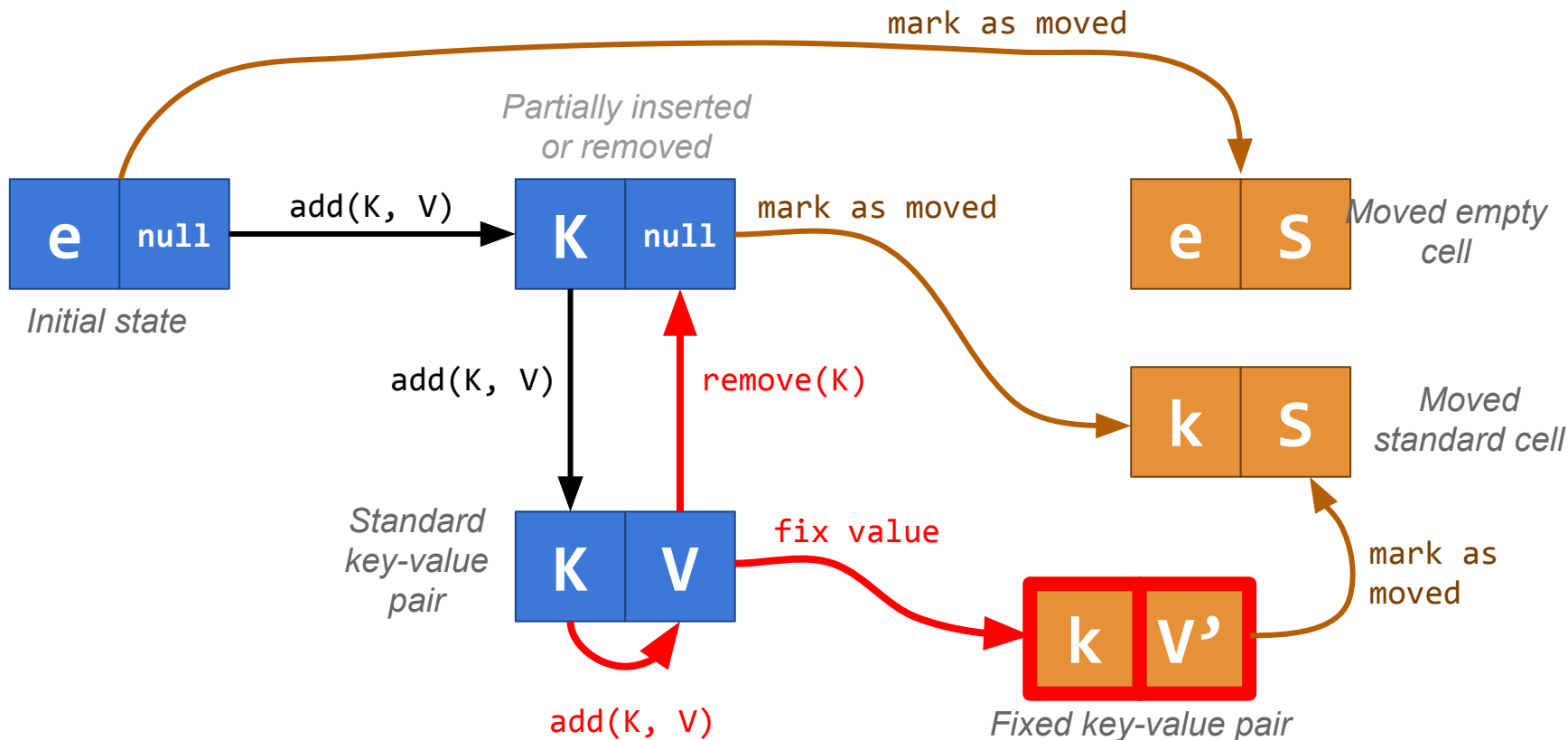
Давайте напишем кеш!

# Давайте напомним кеш!

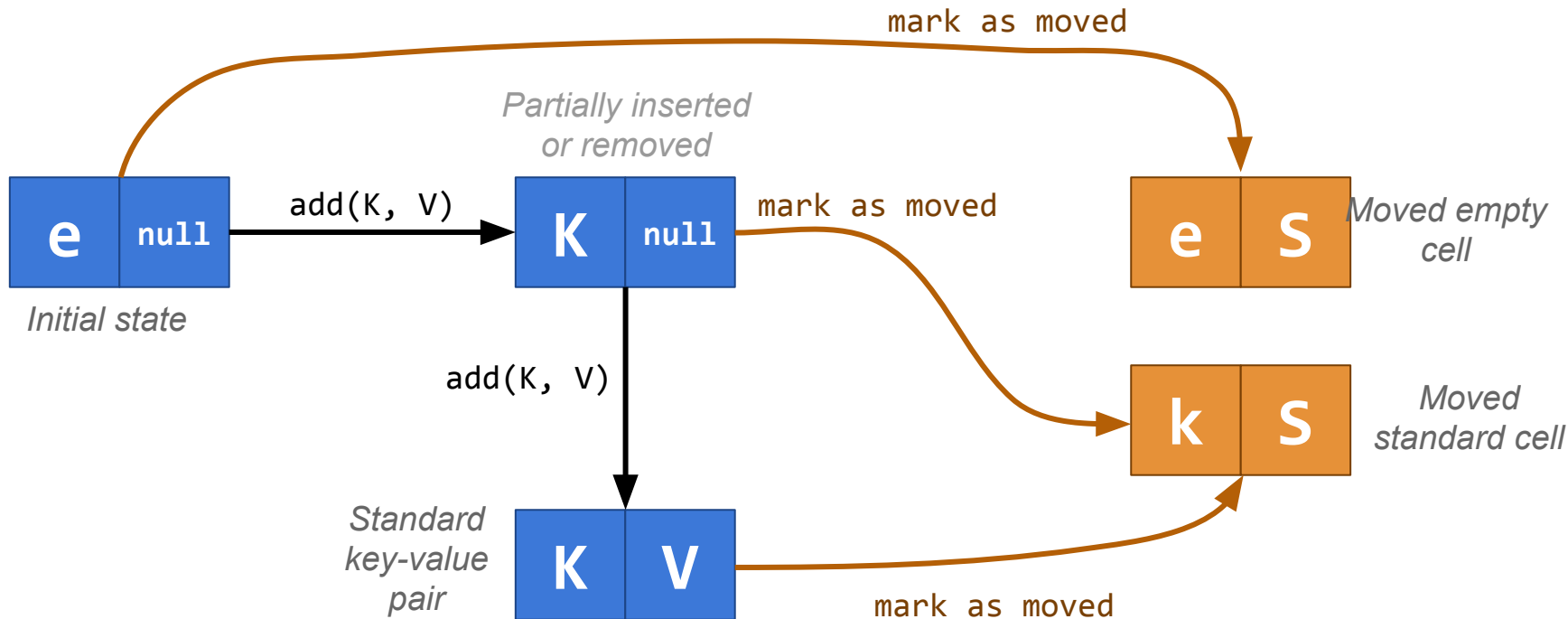




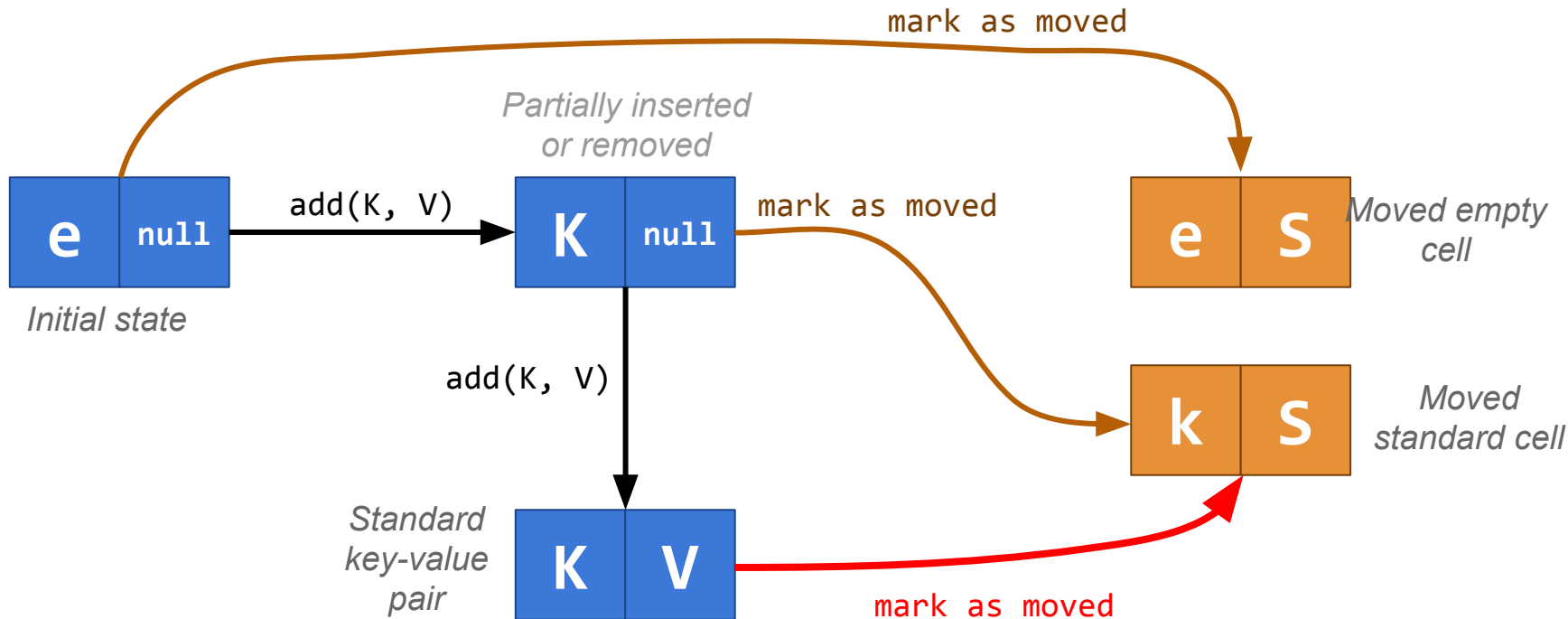
# Давайте напишем кеш!



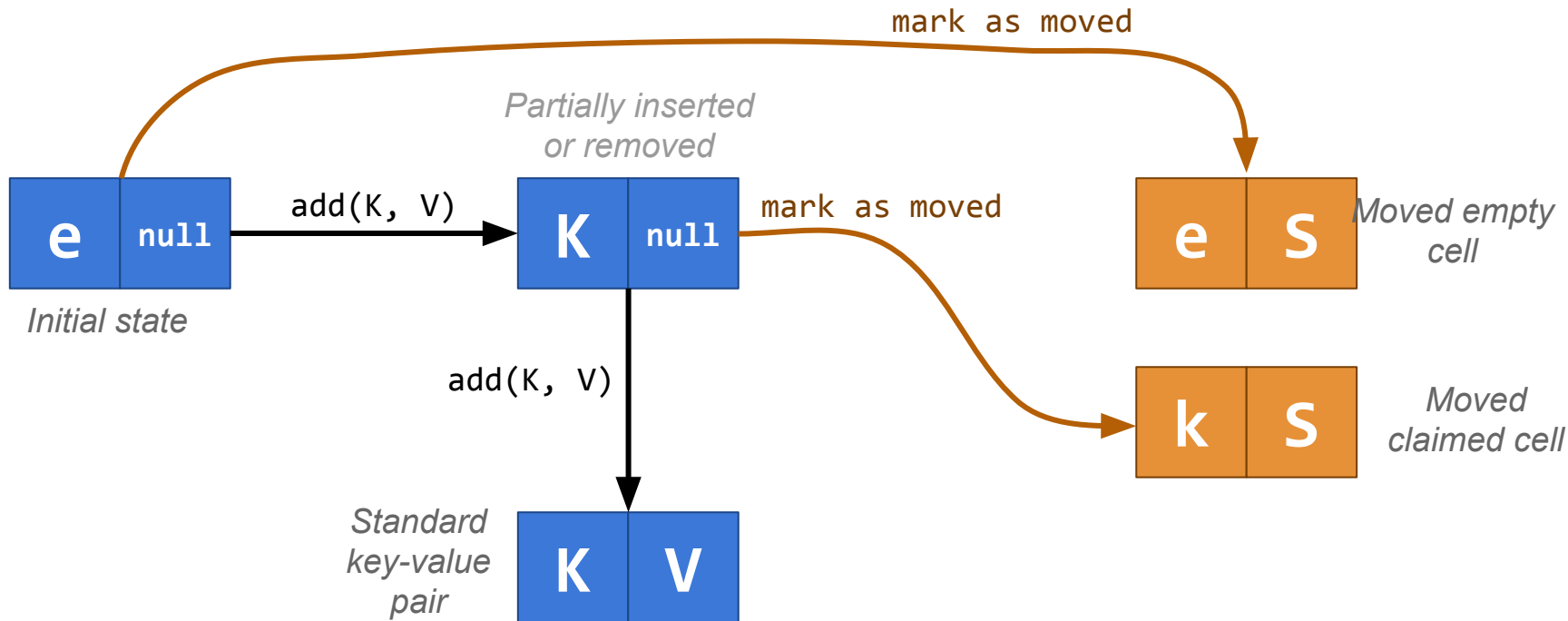
# Давайте напомним кеш!



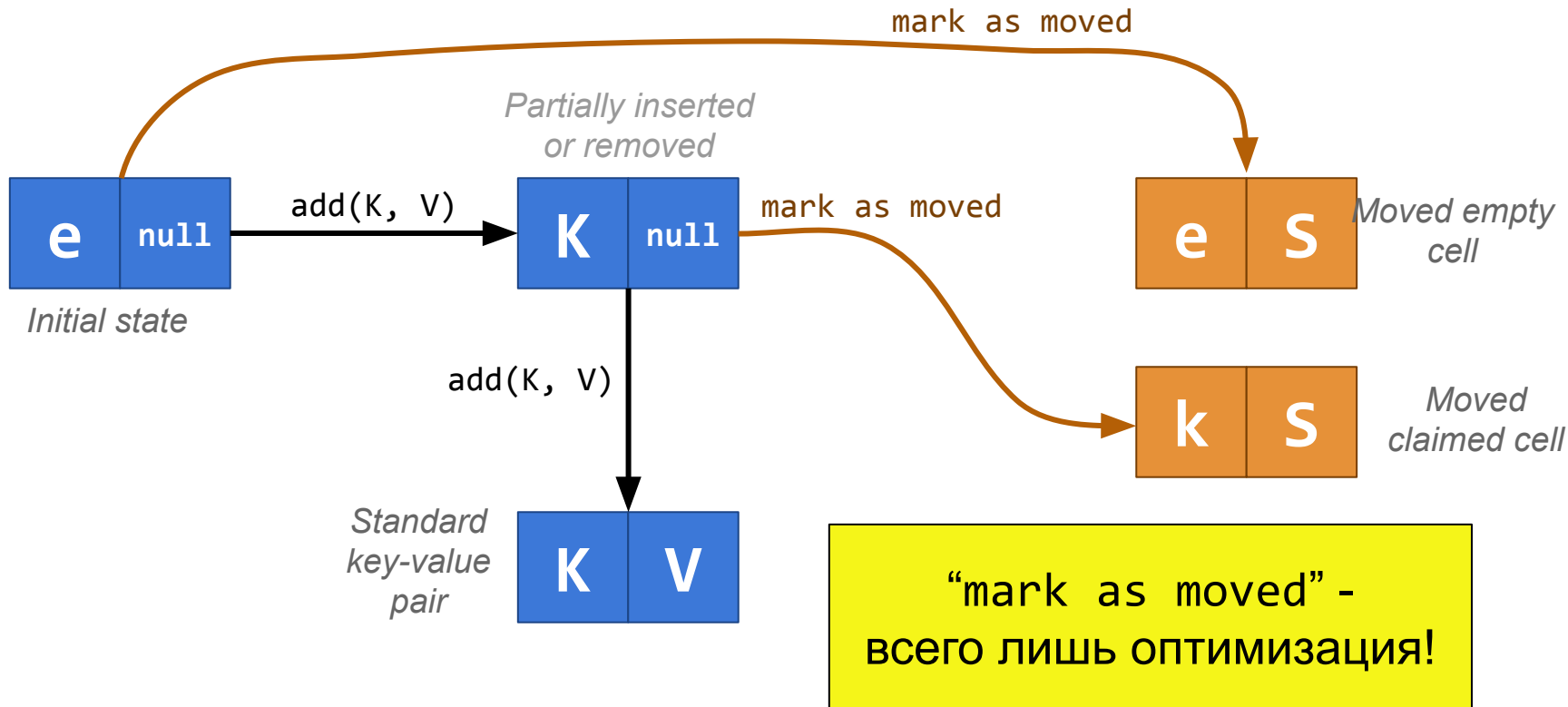
# Давайте напомним кеш!



# Давайте напомним кеш!



# Давайте напомним кеш!

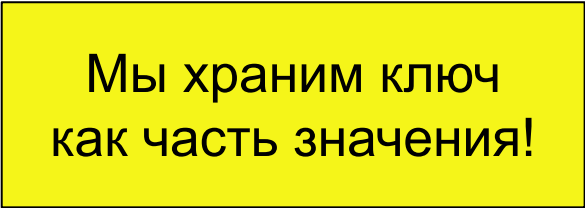


# Посмотрим на задачу внимательнее

```
class Account(val id: Int) {  
    val i1: Int = 0  
    val i2: Int = 0  
    val o1: Any? = null  
    val o2: Any? = null  
    val o3: Any? = null  
    val o4: Any? = null  
}
```

# Посмотрим на задачу внимательнее

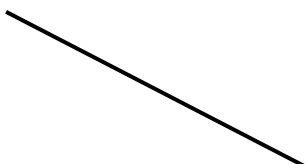
```
class Account(val id: Int) {  
    val i1: Int = 0  
    val i2: Int = 0  
    val o1: Any? = null  
    val o2: Any? = null  
    val o3: Any? = null  
    val o4: Any? = null  
}
```



Мы храним ключ  
как часть значения!

# Посмотрим на задачу внимательнее

```
class Account(val id: Int) {  
    val i1: Int = 0  
    val i2: Int = 0  
    val o1: Any? = null  
    val o2: Any? = null  
    val o3: Any? = null  
    val o4: Any? = null  
}
```

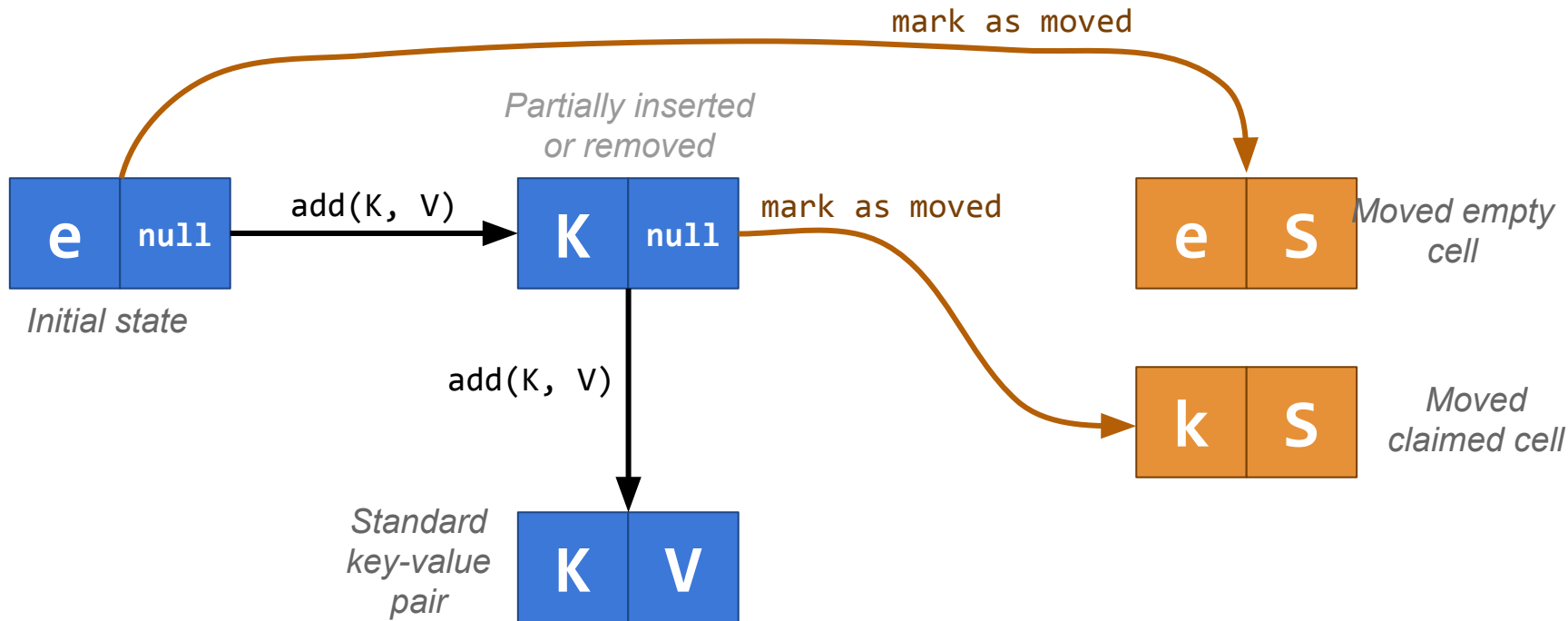


Мы храним ключ  
как часть значения!

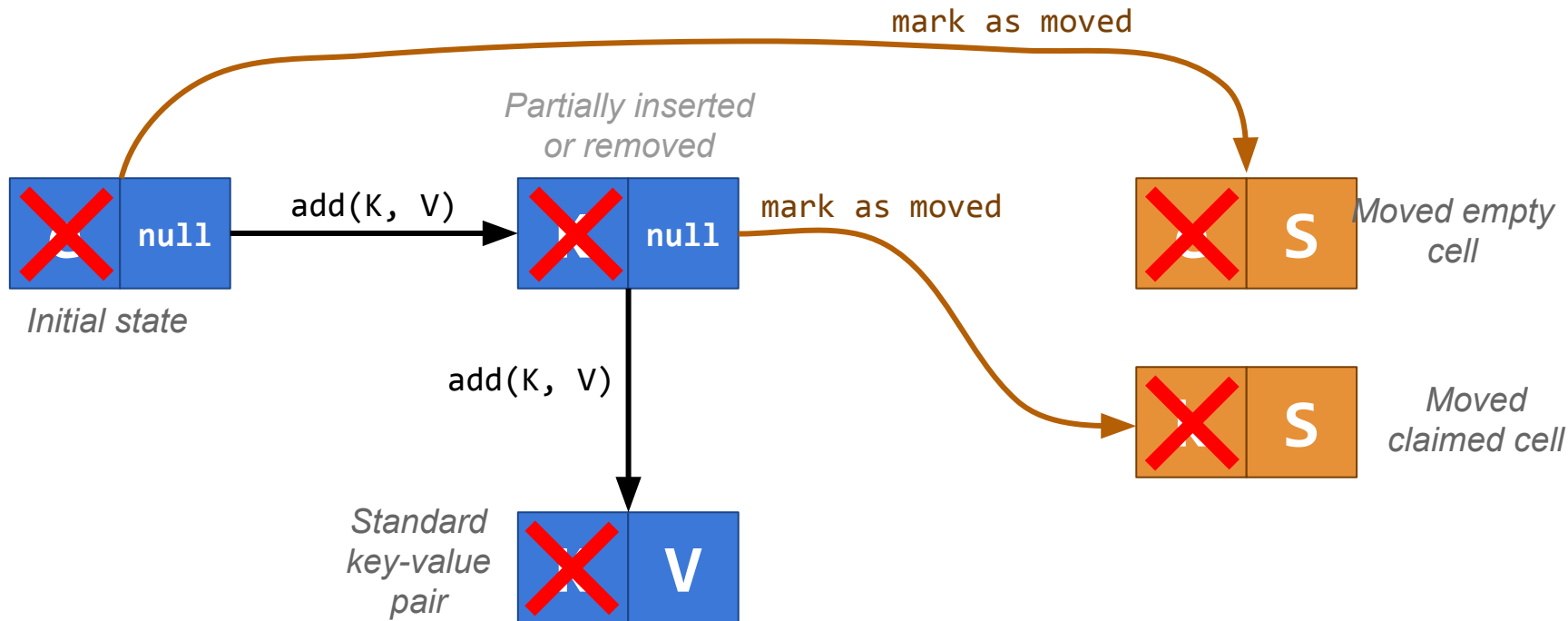
Может тогда не использовать  
отдельную ячейку для ключа?



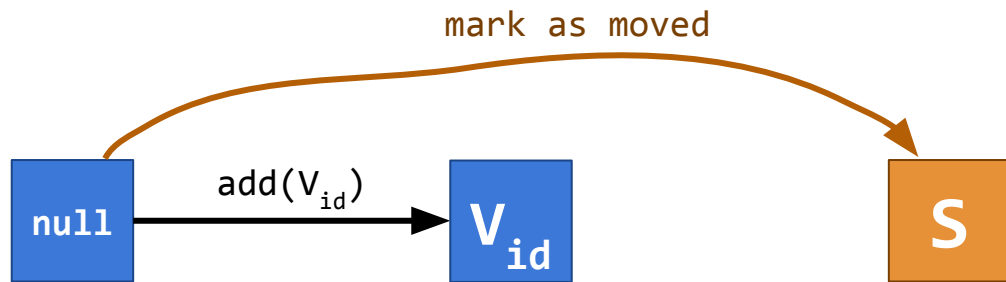
# Ключ как часть значения



# Ключ как часть значения



# Ключ как часть значения



# Где еще можно использовать эти знания?

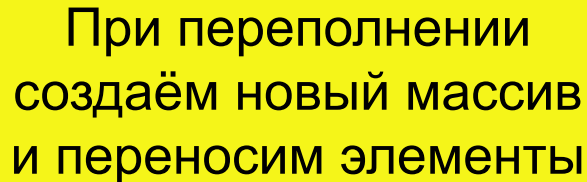
Напишем саморасширяющийся массив в качестве упражнения

- `get(i)`
- `set(i, x)`
- `push_back(x)`

# Где еще можно использовать эти знания?

Напишем саморасширяющийся массив в качестве упражнения

- `get(i)`
- `set(i, x)`
- `push_back(x)`



При переполнении  
создаём новый массив  
и переносим элементы

# Где еще можно использовать эти знания?

Напишем саморасширяющийся массив в качестве упражнения

- `get(i)`
- `set(i, x)`
- `push_back(x)`

**С деталями разберётесь  
сами в домашке :)**

При переполнении  
создаём новый массив  
и переносим элементы

# Вопросы?

# Графики

