# Хеш-таблицы

Филипп Воронов

# Филипп Воронов

Teamlead, Поиск Mail.ru

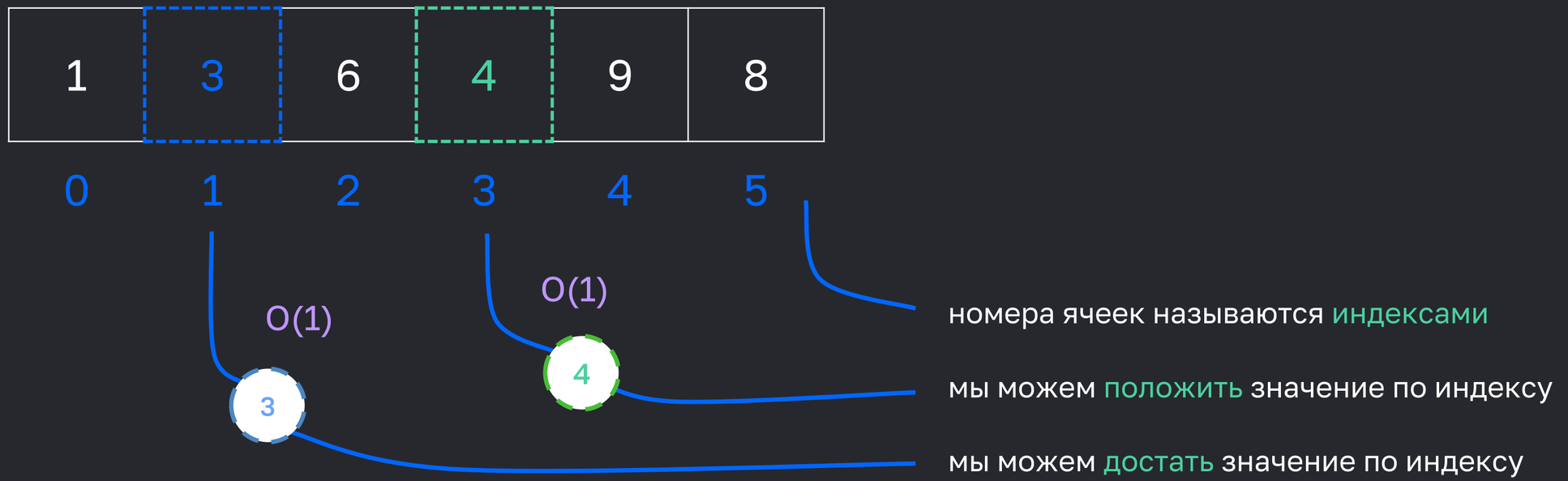
**Аккаунты в соц.сетях**

@Филипп Воронов

# Ассоциативный массив

# Ассоциативный массив

Что такое массив? Это набор пронумерованных ячеек фиксированного размера



# Ассоциативный массив

Положить и достать значение по индексу занимает время,
не зависящее от размера массива, т. е. за O(1)

> arr = [n пустых ссылок]
> arr[0] = $info_0$
> arr[1] = $info_1$
> arr[11] = $info_{11}$

# Ассоциативный массив

Что такое ассоциативный массив? Это структура данных, похожая на массив, только где роль индексов играют другие объекты (например, строки).

Такие индексы называются ключами

> arr = новый ас. массив
> arr[“Наташа”] = $info_{Наташа}$
> arr[“Вася”] = $info_{Вася}$
> arr[“Яна”] = $info_{Яна}$

# Ассоциативный массив

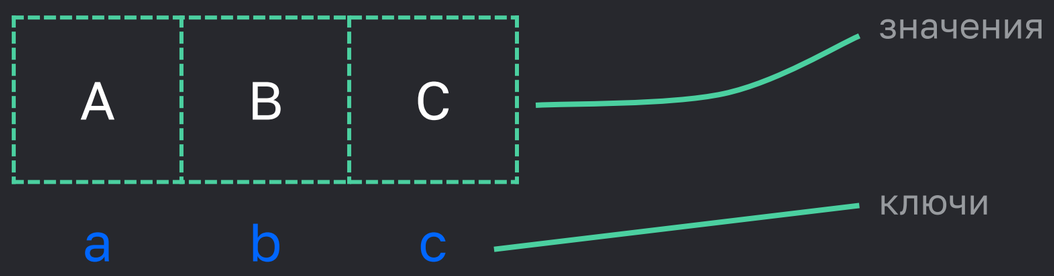
Аналогия. Ассоциативный массив можно представить себе как менеджера большого дома, в который заселяются люди (значения), предоставляя менеджеру своё уникальное Ф. И. О. (ключ). Если под этим Ф. И. О. уже заселён человек, то предыдущий тёска выселяется. Также у менеджера могут попросить позвать проживающего человека указав Ф. И. О. и менеджер должен быстро его найти в доме

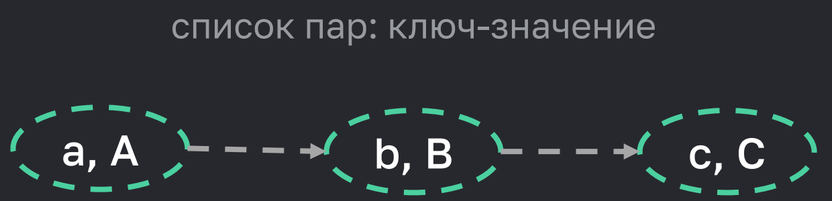
> arr = новый ас. массив
> arr[“Наташа”] = $info_{Наташа}$
> arr[“Вася”] = $info_{Вася}$
> arr[“Яна”] = $info_{Яна}$

# Наивная реализация ассоциативного массива

# Наивная реализация ассоциативного массива

Ассоциативный массив на динамическом массиве.
Давайте заведём массив пар ключ-значение:



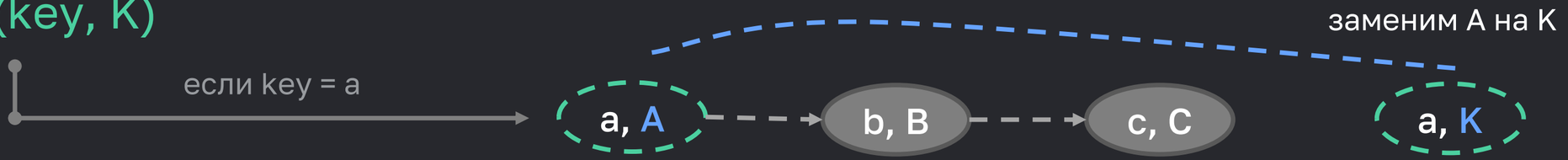




Доступ к значению по ключу будем реализовывать через перебор этих пар и сверкой с ключом

# Наивная реализация ассоциативного массива

Реализация как списка пар:

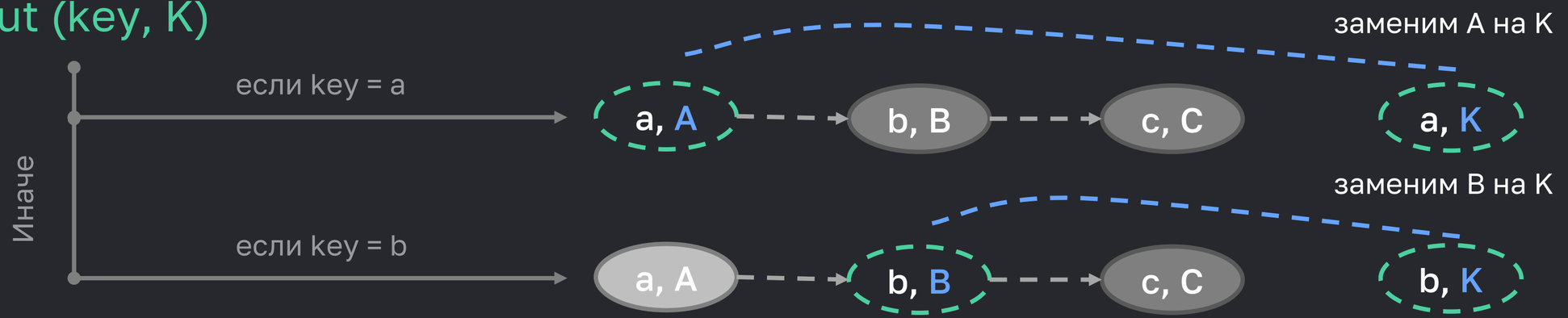

put (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

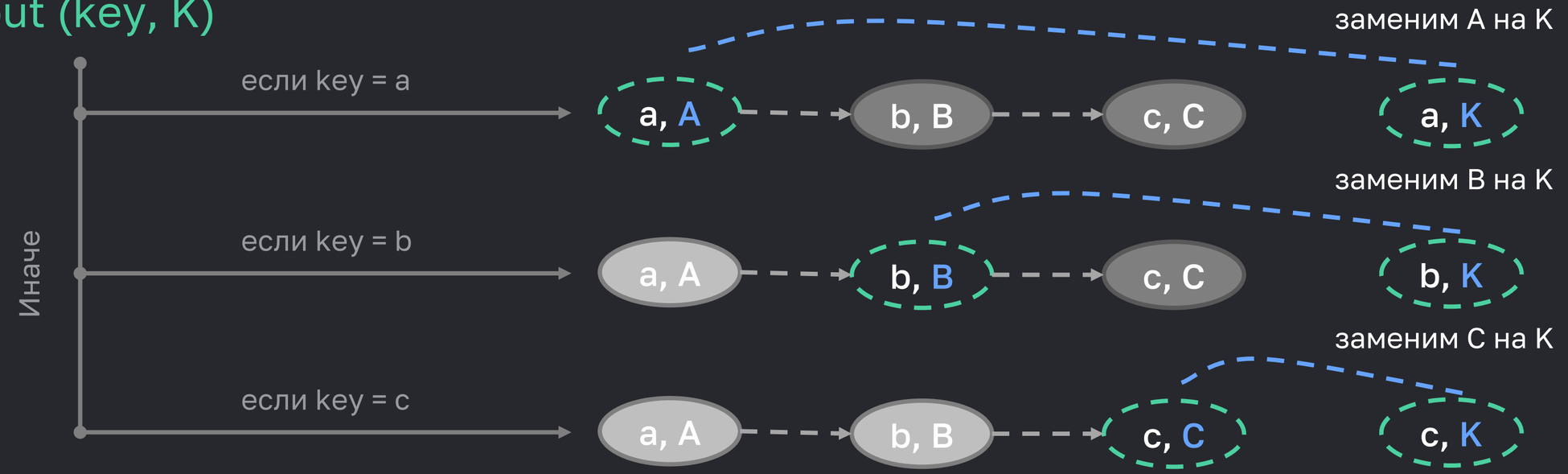

put (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

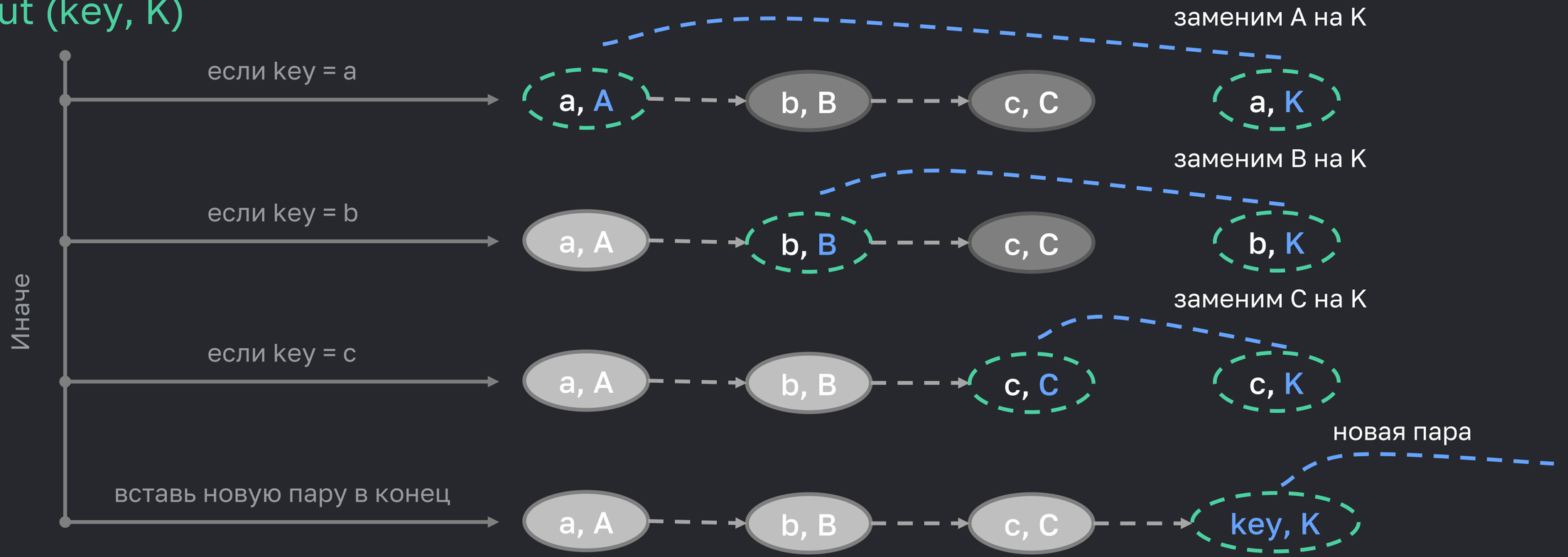

put (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

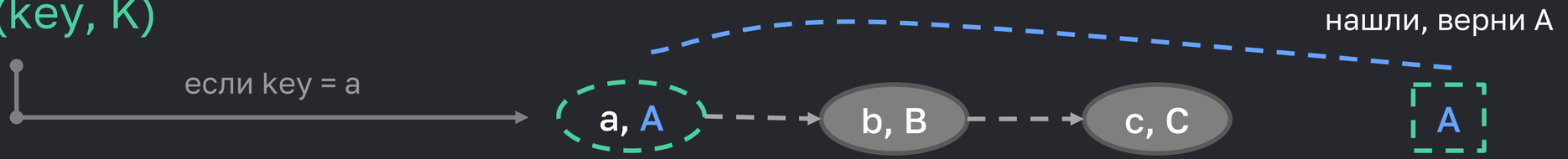

put (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

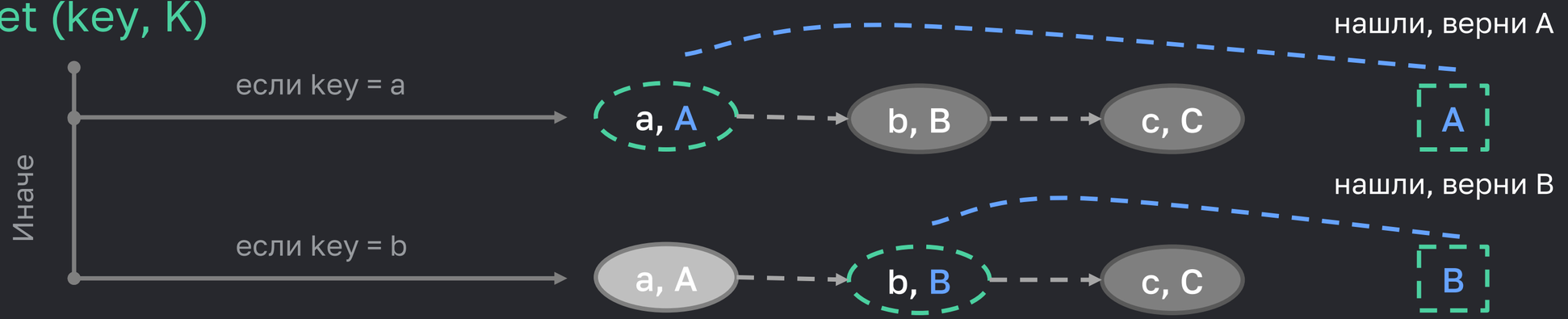

get (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

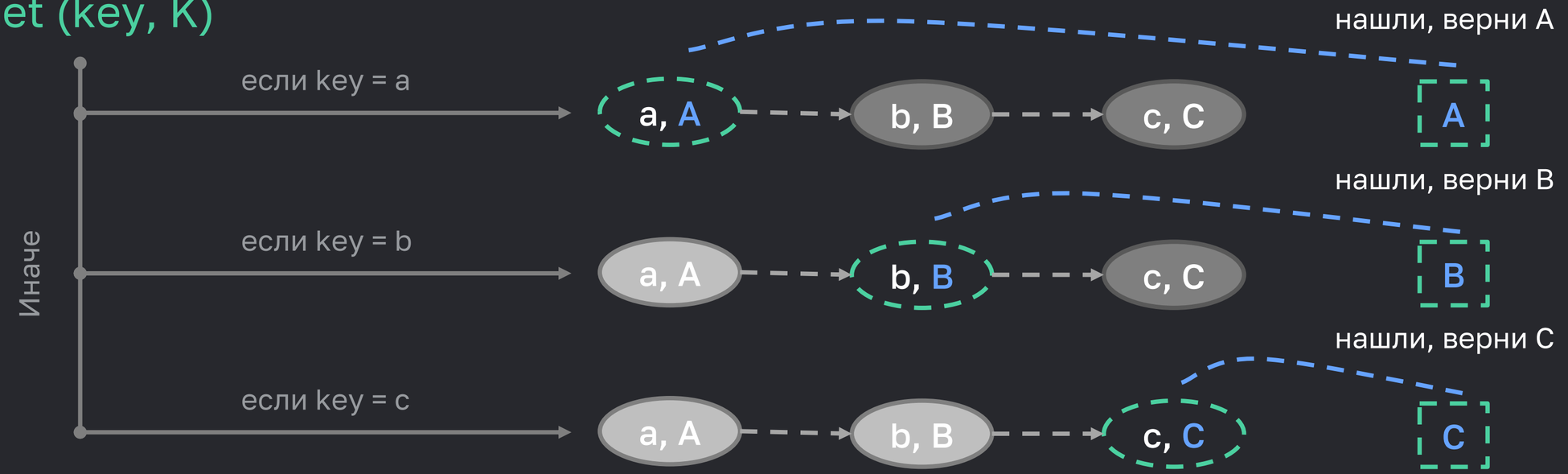

get (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:

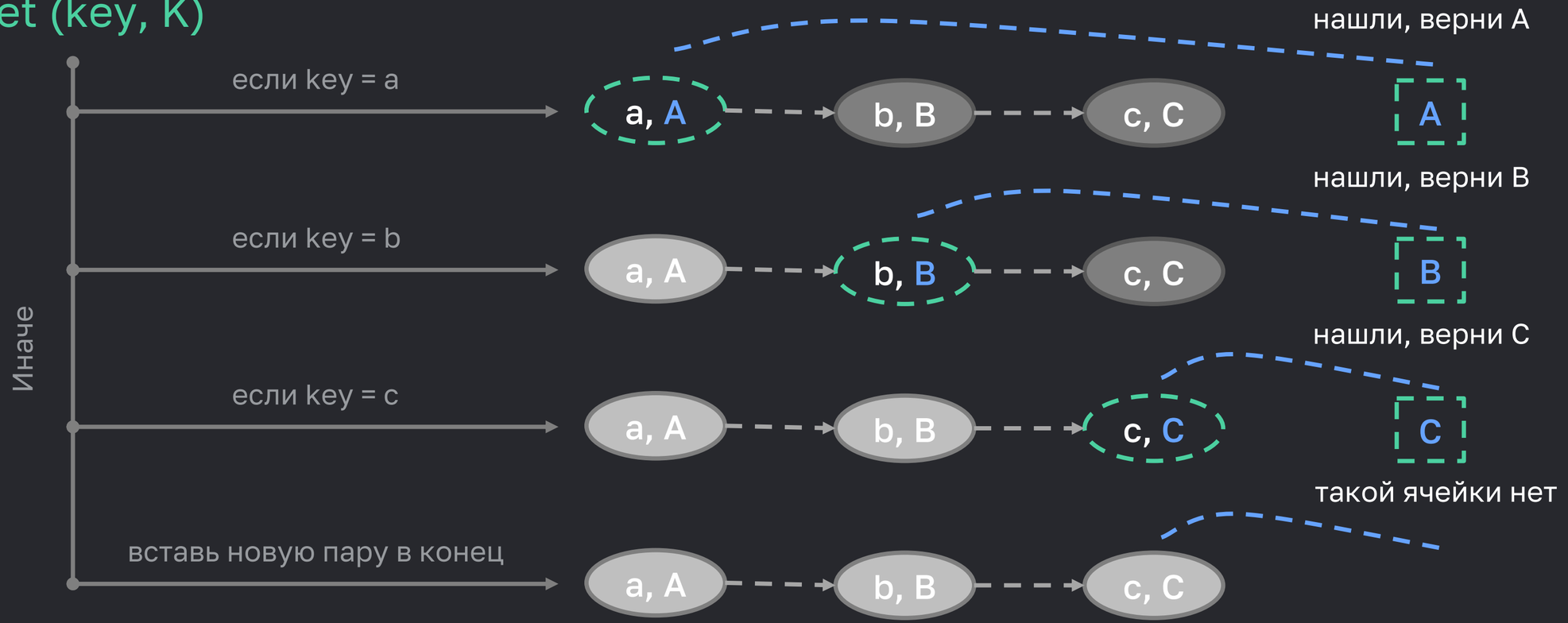

get (key, K)



# Наивная реализация ассоциативного массива

Реализация как списка пар:


get (key, K)



# Наивная реализация ассоциативного массива

Память.
В нашем внутреннем массиве лежат пары ключ-значение, ключей n штук, значит и памяти будем занимать $O(n)$.
Асимптотика не может быть лучше

Добавление.
Для добавления мы вынуждены пробежаться по внутреннему массиву для проверки, что такого ключа ещё нет.
Это займет $O(n)$

Доступ.
Мы также вынуждены пробежаться по массиву, что превращает временную асимптотику в $O(n)$.
Или $O(\log_2 n)$, если будем держать отсортированным по ключу. Медленно!
Хотим асимптотику как в массиве!

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
 data: []

 put(key, value):
  for i от 0 до длина(data)
   k, v = data[i]
   if k = key
    data[i] = (key, value)
    return
  положить в конец data пару (key, value)

 get(key):
  for i от 0 до длина(data)
   k, v = data[i]
   if k = key
    return v
  return “Нет такого ключа!”
}
```

Комментарий
То, где храним пары

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Операция присвоения по ключу

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий
Пробежимся по списку

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Если в рассматриваемой ячейке наш ключ, то меняем значение

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Если не нашли такой ключ, то вставляем новую пару в список

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Операция получения по ключу

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий
Пробежимся по списку

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Если в рассматриваемой ячейке наш ключ, то возвращаем значение из этой ячейки

# Наивная реализация ассоциативного массива

Псевдокод:

```
AsArray {
  data: []

  put(key, value):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        data[i] = (key, value)
        return
    положить в конец data пару (key, value)

  get(key):
    for i от 0 до длина(data)
      k, v = data[i]
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Если не нашли такой ключ, то говорим, что нет значения для такого ключа

# Хеш-функция

# Хеш-функция

Хеш-функцией объекта называется алгоритм генерации какого-либо числа (хеша), который гарантирует, что при применении к равным объектам даст одно и то же значение

# Хеш-функция

Хорошей хеш-функцией называется функция, генерирующая достаточно разные значения, которая на похожих объектах может возвращать настолько же далёкие друг от друга числа, как на абсолютно непохожих объектах


Хорошая хеш-функция


Плохая хеш-функция

# Хеш-функция

Примеры хеш-функций. В качестве простой хеш-функции для строки можно взять сумму числовых кодов, входящих в неё символов:

# Хеш-функция


Но это не очень хорошая хеш-функция, т. к. от перемены мест символов хеш-строки даже не меняются

# Хеш-функция

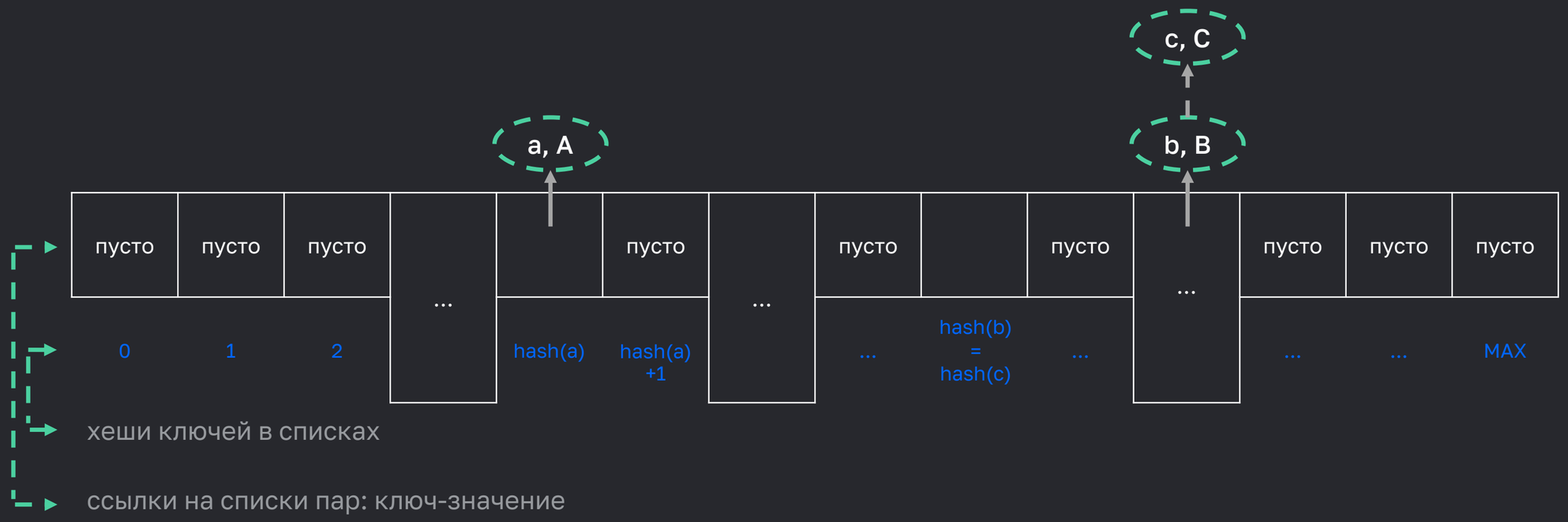
В качестве примера хеша составного объекта можно взять сумму хешей всех данных, входящих в него

```
MyClass {
  a: какое-то поле
  b: какое-то поле

  hash():
    return hash(a) + hash(b)
}
```

# Наивный ассоциативный массив на хешах

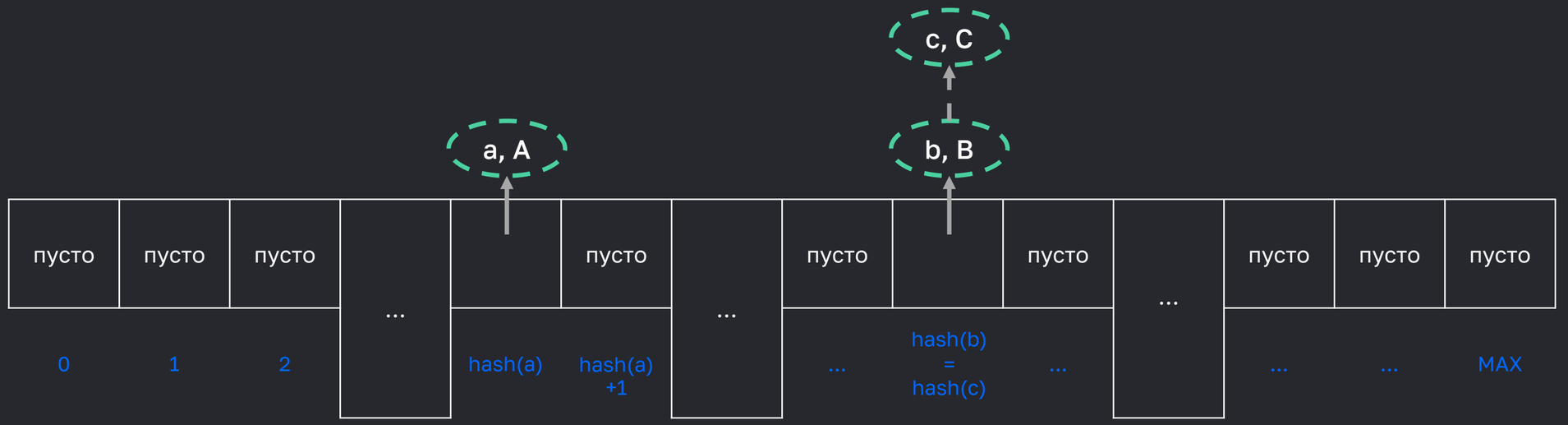
# Наивный ассоциативный массив на хешах

Наивная реализация на хешах. Давайте заведем массив длиною в количество всевозможных значений хешей. В каждой ячейке будем хранить список пар ключ-значение, у которых значение хеша у ключа совпадает с индексом ячейки



# Наивный ассоциативный массив на хешах

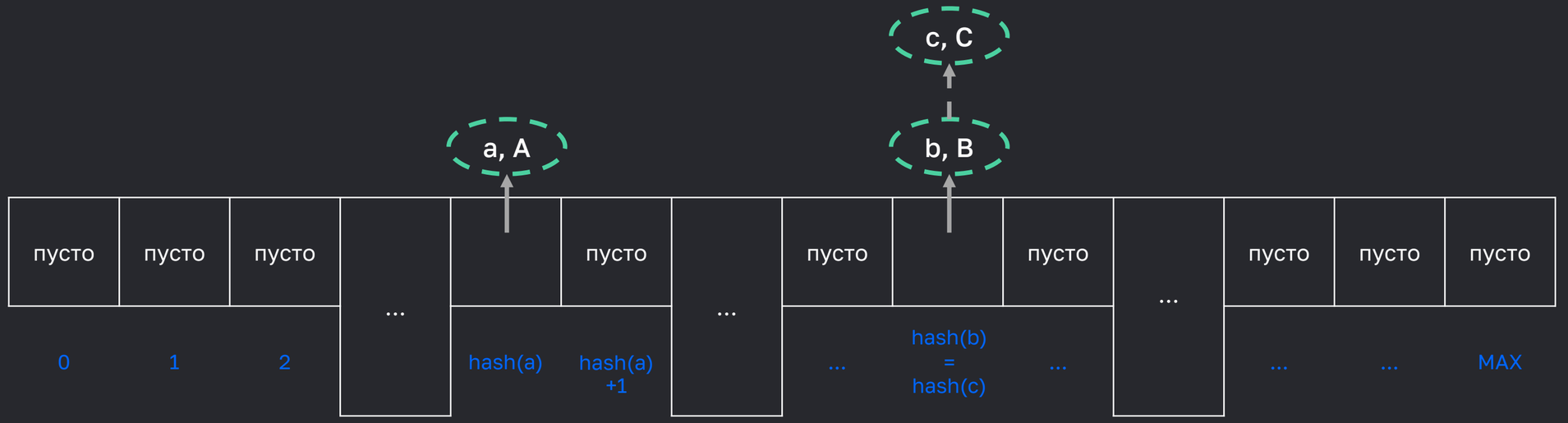
Добавление. Посчитаем хеш от ключа и возьмём список из ячейки под таким индексом.




Проверим, что там нет уже пары с таким ключом (в другом списке он быть не может). Если есть — заменим значение, если нет — добавим новое. Время работы — O(длина списка)

# Наивный ассоциативный массив на хешах

Доступ. Делается аналогично добавлению. Время работы — O(длина списка)



# Наивный ассоциативный массив на хешах

Псевдокод:

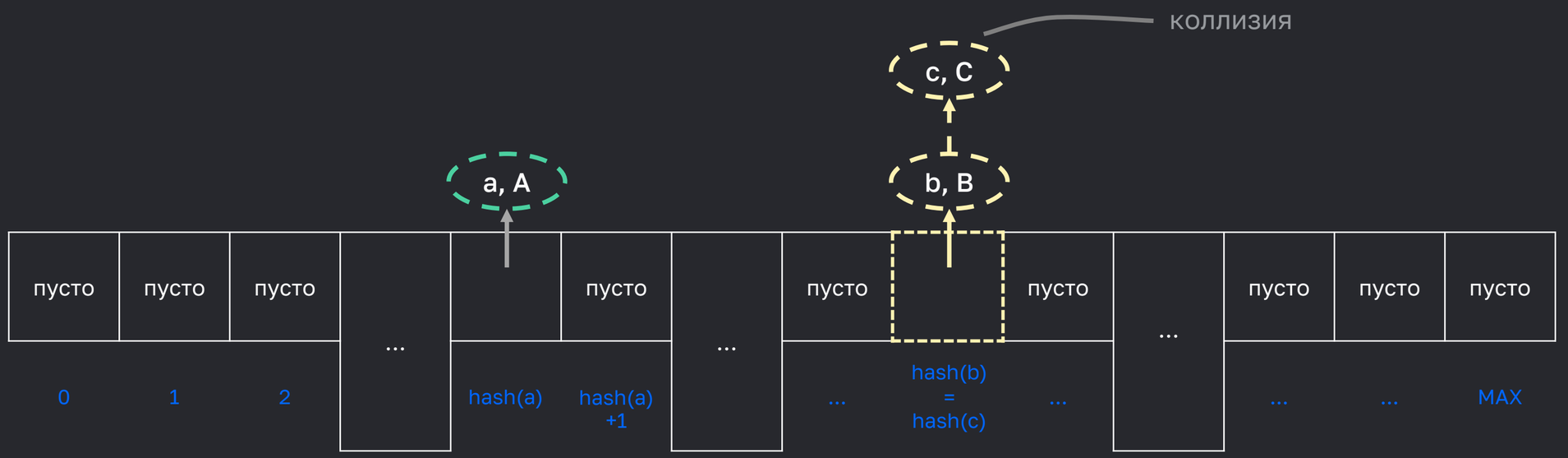
```
put (key, K):
    достать список из arr[hash(key)]
    добавить в него, как в прошлом алгоритме

get (key):
    достать список из arr[hash(key)]
    добавить в него, как в прошлом алгоритме
}
```

Комментарий

- константное время, если массив большой, а хеш-функция хорошая
- огромная память — массив размером с диапазон всех чисел (всех хешей)

# Наивный ассоциативный массив на хешах

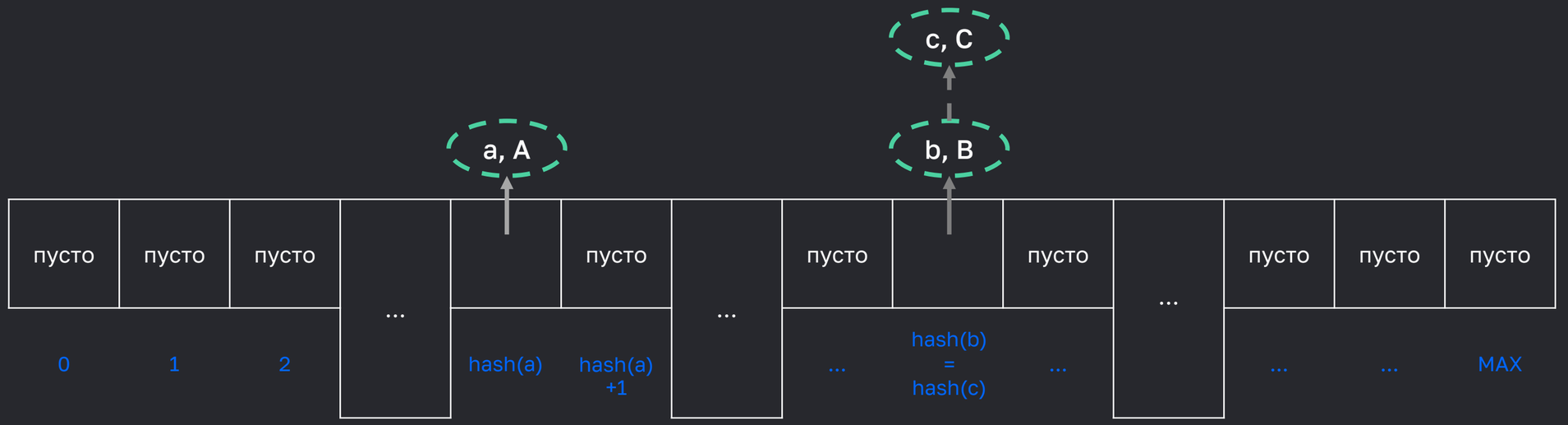
Длина списка. Попадание разных ключей в одну ячейку называется коллизией




Если ячеек больше, чем элементов, то для хорошей хеш-функции количество коллизий в среднем не будет зависеть от n, стало быть и добавление, и доступ за O(1)

# Наивный ассоциативный массив на хешах

Память. Такой алгоритм сильно проигрывает по памяти




Если хеш, например, 32-битное число, то это 4+ млрд ячеек для таблицы (гигабайты) в 100 элементов. Это очень затратно!

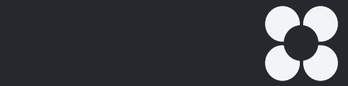
# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
 data: [(max_hash - min_hash) пустых списков]

 put(key, value):
  list = data[hash(key)]
  for (k, v) из list
   if k = key
    v = value
    return
  положить в конец list пару (key, value)

 get(key):
  list = data[hash(key)]
  for (k, v) из list
   if k = key
    return v
  return “Нет такого ключа!”
}
```

Комментарий

Массив списков размером в количество всевозможных хешей



# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий
Операция добавления

# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Достаем список из ячейки с номером равным хешу ключа

# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Работаем с этим списком так же,
как и в предыдущем наивном подходе

# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий
Операция получения элемента по ключу

# Наивный ассоциативный массив на хешах

Псевдокод:

```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий

Достаем список из ячейки с номером равным хешу ключа

# Наивный ассоциативный массив на хешах

Псевдокод:

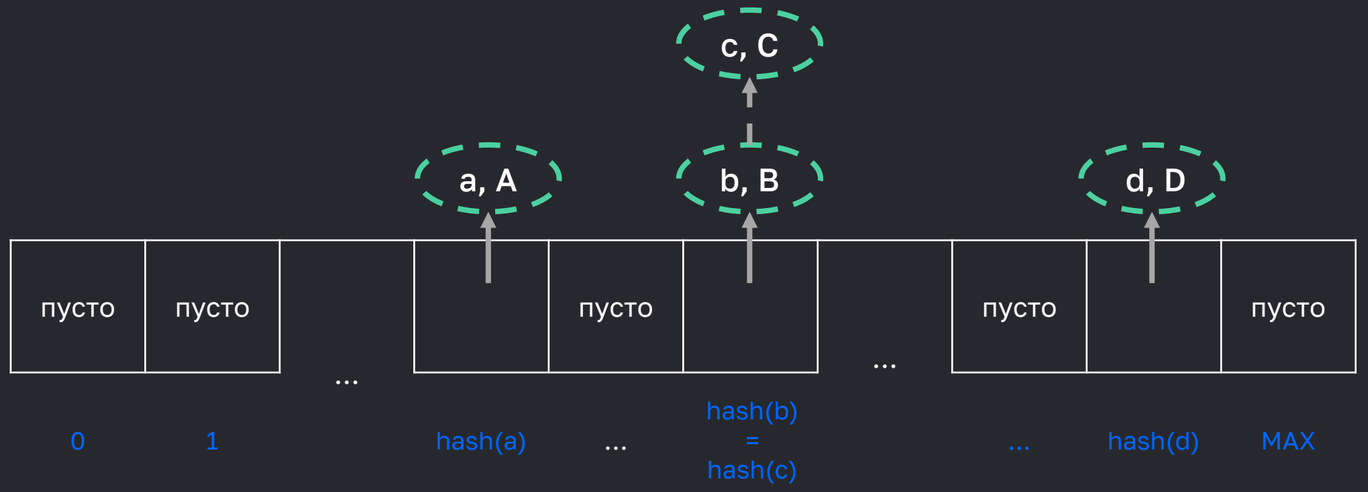
```
AsArray {
  data: [(max_hash - min_hash) пустых списков]

  put(key, value):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        v = value
        return
    положить в конец list пару (key, value)

  get(key):
    list = data[hash(key)]
    for (k, v) из list
      if k = key
        return v
    return “Нет такого ключа!”
}
```

Комментарий
Работаем с этим списком так же,
как и в предыдущем наивном подходе

# Хеш-таблица

# Хеш-таблица

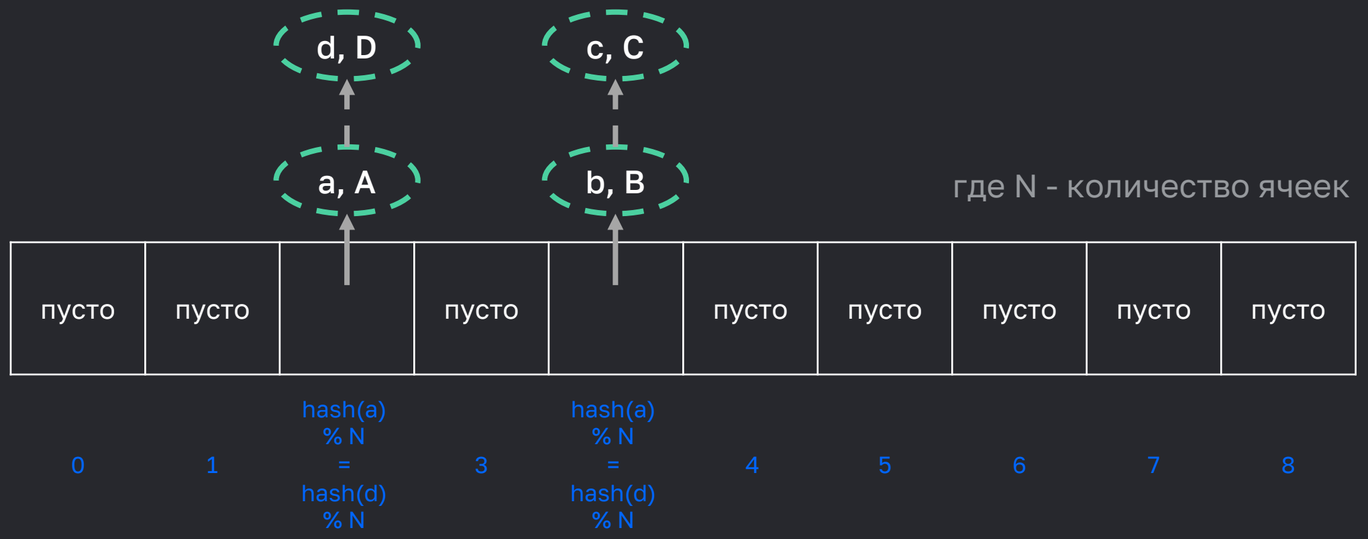
Решение проблемы с памятью. Одно маленькое улучшение позволит нам избавиться от проблемы с памятью. Сократим диапазон значений хеш-функции через, взятие её значения по модулю.




Раньше был ключ, мы брали хеш и получали номер ячейки

# Хеш-таблица

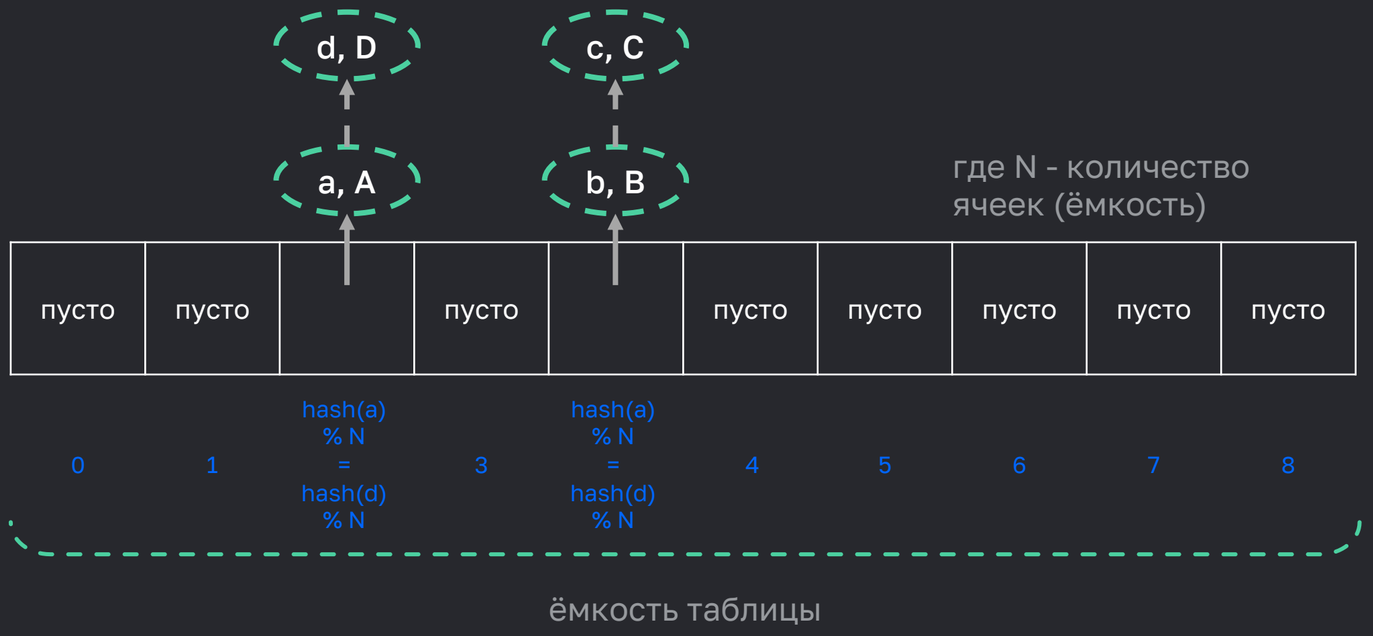
Решение проблемы с памятью. Одно маленькое улучшение позволит нам избавиться от проблемы с памятью. Сократим диапазон значений хеш-функции через, взятие её значения по модулю.




Теперь есть ключ, есть хеш. Мы берём его по модулю какого-то числа и получаем номер ячейки

# Хеш-таблица

Теперь будем называть длину внутреннего массива ёмкостью таблицы. Если она больше n, то количество коллизий, т. е. длина списков, O(1)



# Хеш-таблица

Добавление и доступ. Все остаётся по-старому, но надо учесть, что при добавлении элементов мы можем приближаться количеством ключей к ёмкости и нам надо будет её увеличить.

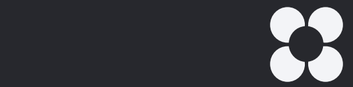
Это дорогая операция O(C + n), т. к. мы создаём новый внутренний массив большей длины и перехешируем все вставленные пары ключ-значения в него. Если использовать подход из динамического массива, то амортизационно операция добавления нового элемента останется O(1)

# Хеш-таблица

Псевдокод:

```
HashTable {
 C: число - ёмкость таблицы,
 data: [C пустых списков],
 n: число ключей

 put(key, value):
  list = data[hash(key) % C]
  ...
  if вставили новый ключ: n += 1
  if n близко к C:
   увеличим C
   new_data = [C пустых списков]
   вставим все элементы в new_data
   data = new_data

 get(key):
  list = data[hash(key) % C]
  … }
```

Комментарий
Размер внутреннего массива



# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий
Массив списков

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий
Число ключей в таблице

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий
Операция добавления

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий

Достаём список из ячейки с номером хеш ключа по модулю C

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий

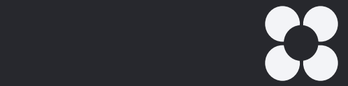
Достаём список из ячейки с номером хеш ключа по модулю C

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий
Меняем счётчик ключей



# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий
Если массив сильно заполнен — пересоздаём его размером побольше

# Хеш-таблица

Псевдокод:

```
HashTable {
  C: число - ёмкость таблицы,
  data: [C пустых списков],
  n: число ключей

  put(key, value):
    list = data[hash(key) % C]
    ...
    if вставили новый ключ: n += 1
    if n близко к C:
      увеличим C
      new_data = [C пустых списков]
      вставим все элементы в new_data
      data = new_data

  get(key):
    list = data[hash(key) % C]
    … }
```

Комментарий

Если массив сильно заполнен — пересоздаём его размером побольше

# Хеш-функция для строк

# Наивная хеш-функция

Сумма числовых кодов строк. Просто берём и суммируем все символы, а точнее их числовые коды

# Наивная хеш-функция

Асимптотика: вычисление хеш-функции линейно относительно длины строки

# Наивная хеш-функция

Плюсы и минусы:

\+ простота вычисления

# Наивная хеш-функция

Плюсы и минусы:

\+ простота вычисления

\- не очень хорошая хеш-функция, т. к. от перемены мест символов значение хеша даже не меняется

# Наивная хеш-функция

Псевдокод:

```
hash(s):
  ans = 0
  for i от 0 до длина(s)
    c = s[i]
    ans += код(c)
  return ans
```

Комментарий
Функция вычисления хеша строки

# Наивная хеш-функция

Псевдокод:

```
hash(s):
  ans = 0
  for i от 0 до длина(s)
    c = s[i]
    ans += код(c)
  return ans
```

Комментарий
Пробежимся по строке и посчитаем сумму кодов символов

# Наивная хеш-функция

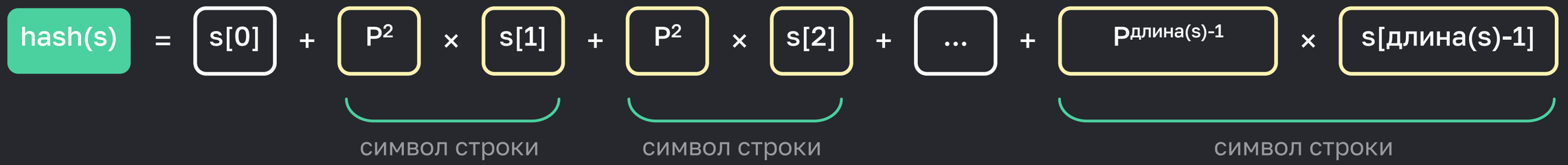
Псевдокод:

```
hash(s):
  ans = 0
  for i от 0 до длина(s)
    c = s[i]
    ans += код(c)
  return ans
```

Комментарий
Эта сумма и есть значение нашей хеш-функции для строки

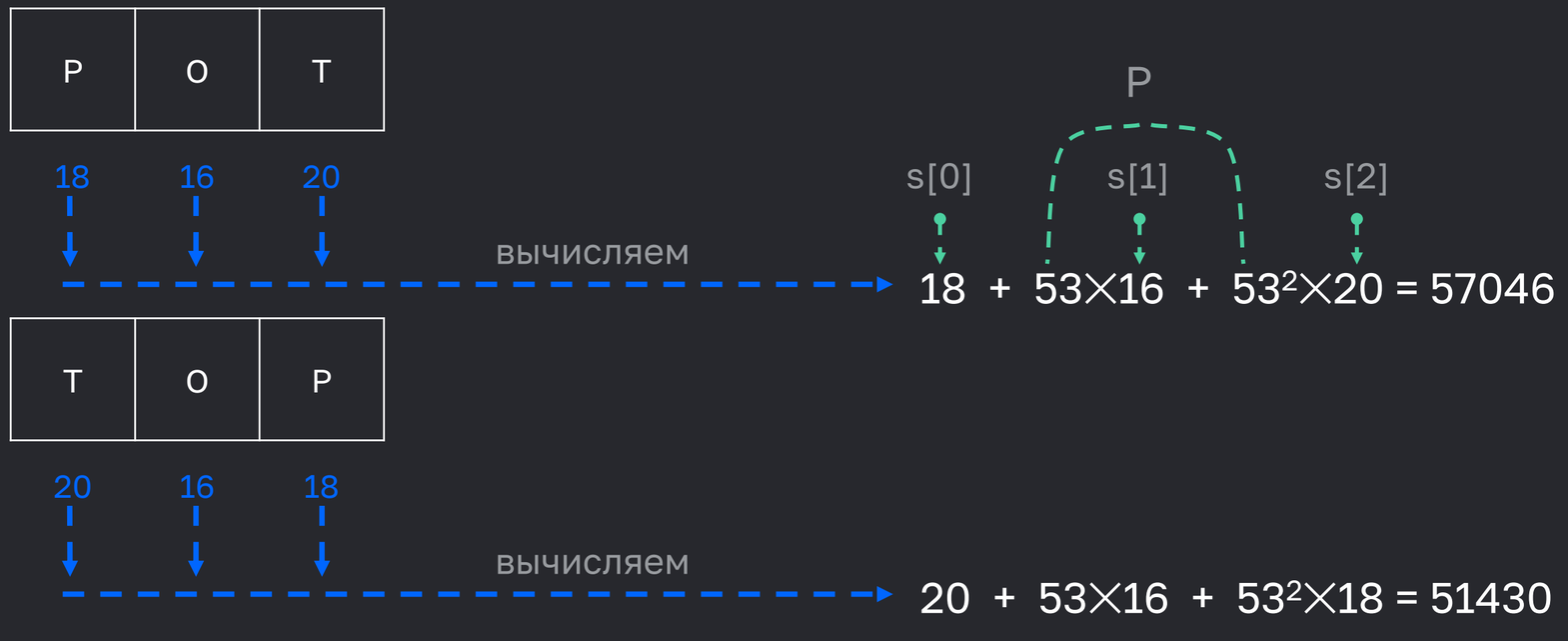
# Классическая хеш-функция для строки

Возьмем простое число P, например, 53. Посчитаем следующую сумму для строки s:



# Классическая хеш-функция для строки

$$\text{hash}(s) = s[0] + P^2 \times s[1] + P^2 \times s[2] + \ldots + P^{\text{длина}(s)-1} \times s[\text{длина}(s)-1]$$

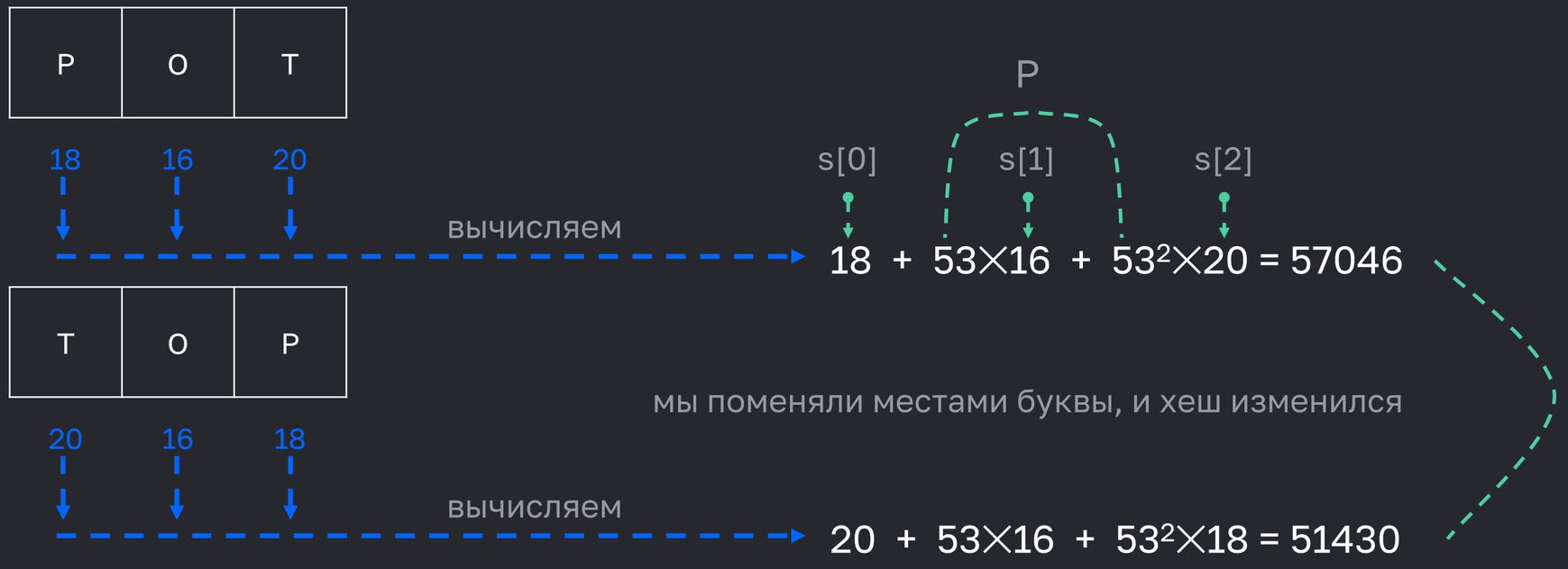


# Классическая хеш-функция для строки

Асимптотика. Вычисление хеш-функции линейно относительно длины строки.

Плюсы

+ Такая хеш-функция является хорошей. Также она подойдёт для алгоритма Рабина — Карпа



# Классическая хеш-функция для строки

Псевдокод:

```
hash(s):
  P = 53
  ans = 0
  mult = 1
  for i от 0 до длина(s)
    c = s[i]
    ans += mult ⨯ код(c)
    mult = mult ⨯ P
  return ans
```

Комментарий

Вычисление такое же, как в прошлом примере, только теперь мы коды символов умножаем на множитель из степени P

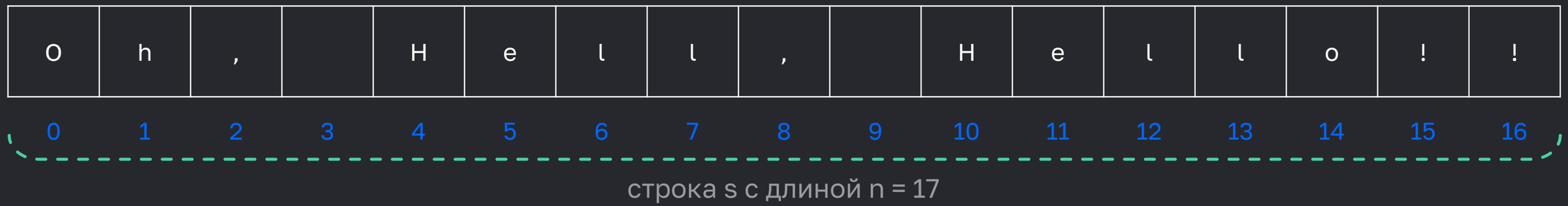
# Алгоритм Рабина — Карпа

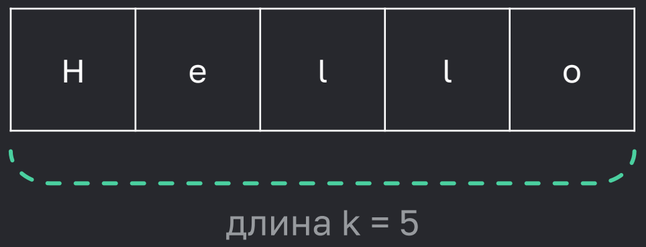
# Поиск подстроки в строке

Задача. У нас есть две строки:

- s с длиной n
- p с длиной k

Пример:



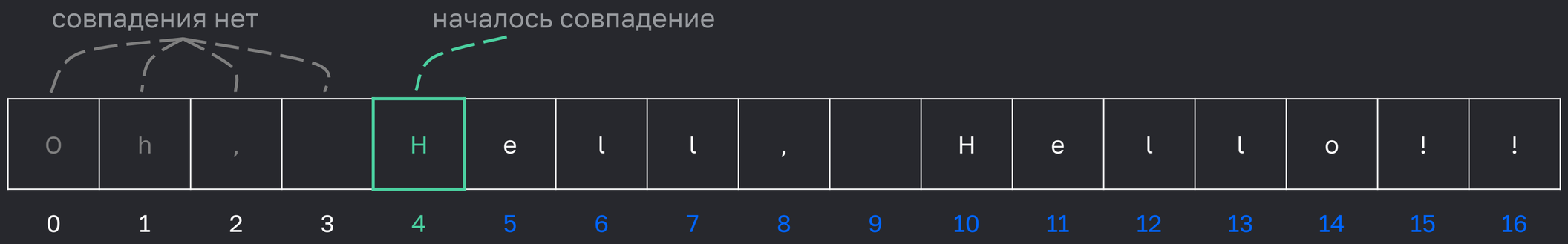




Требуется найти индекс в s, начиная с которого в ней идёт строка p, или сказать, что такого не существует

# Поиск подстроки в строке

Решение:

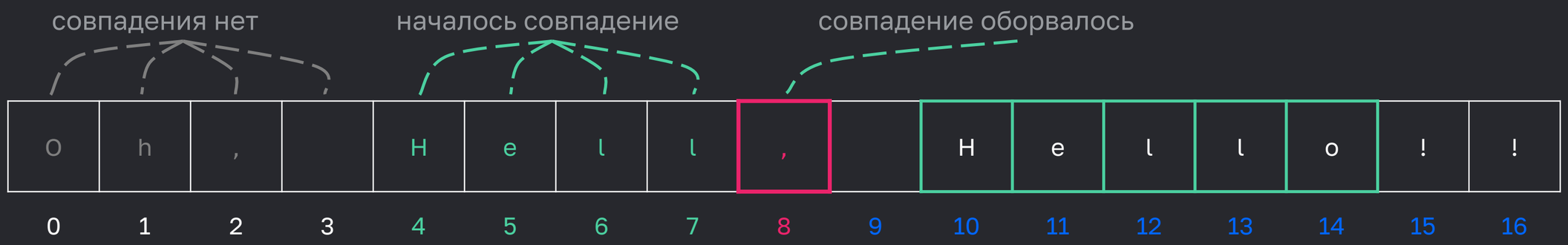
- найдём место, с которого начинается строка p




# Поиск подстроки в строке

Решение:

- проверим оставшиеся совпадения




# Поиск подстроки в строке

Решение:

- повторим операции



Нашли! Задача решена. Ответ: 10

# Поиск подстроки в строке

Асимптотика такой ложной проверки — O(k)
максимальное количество таких проверок — O(n)


# Поиск подстроки в строке

Задача. У нас есть две строки:

- s с длиной n
- p с длиной k

Рассмотрим псевдокод:

```
find(s, p):
  for i от 0 до длина(s)
    for j от 0 до длина(p)
      if s[i+j] != p[j]
        i не подходит
    if i подошёл
      return i
```

Комментарий

Просто пробежимся по индексам строки s и каждый проверим на то, не начинается ли с него строка p

# Поиск подстроки в строке

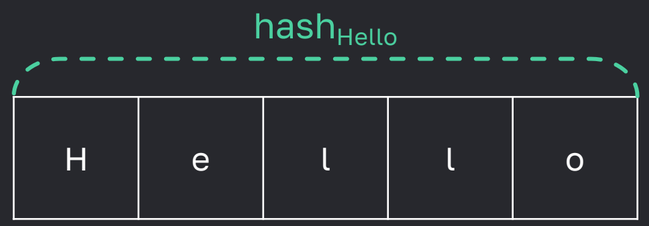
Задача. У нас есть две строки:

- s с длиной n
- p с длиной k

Асимптотика: дополнительной памяти не требуется — O(1).
Время же O(nk).

# Предпроверка через хеш

Улучшенный подход. Перед проверкой индекса циклом проверим хеши hash(s[i...i + k]) и p. Если они разные, то проверять нечего, т. к. строки тоже будут разными. Если одинаковые, то проверить, точно ли строки совпадают всё равно придётся, т. к. совпадение хешей не гарантирует равенство

# Предпроверка через хеш

Улучшенный подход. Перед проверкой индекса циклом проверим хеши и p. Если они разные, то проверять нечего, т. к. строки тоже будут разными. Если одинаковые, то проверить, точно ли строки совпадают всё равно придётся, т. к. совпадение хешей не гарантирует равенство

```
find(s, p):
  for i от 0 до длина(s)
    if hash(s[i…i+k-1]) != hash(p)
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

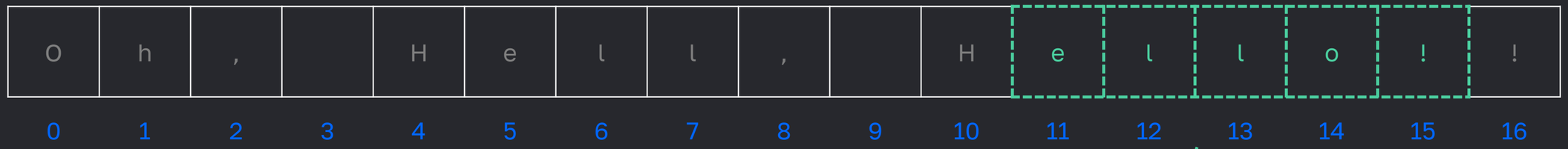
Комментарий

Наивная реализация алгоритма ничем не поможет, т. к. вычисление хеша: hash(s[i...i + k]) будет за O(k). Стало быть, асимптотика продолжает быть O(1) памяти и O(nk) времени

# Алгоритм Рабина — Карпа (упрощённый)

Рассмотрим упрощённый алгоритм Рабина — Карпа, используя хеш-функцию в виде суммы кодов символов.

На каждой итерации цикла строка, которую мы сравниваем с шаблоном, меняется совсем немного: из её начала удаляется первый элемент, а в конец вставляется новый



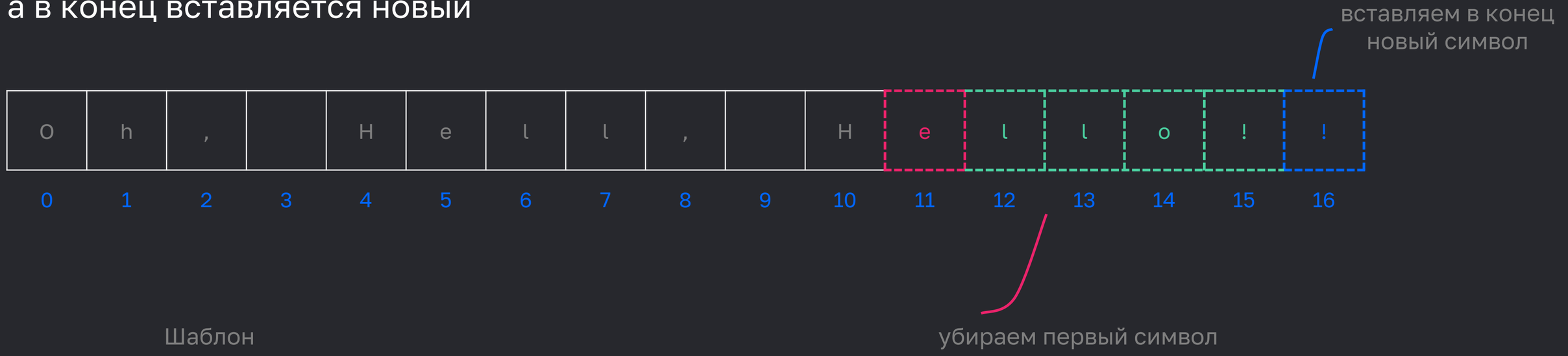

Шаблон


строка, которую мы сравниваем с шаблоном

# Алгоритм Рабина — Карпа (упрощённый)

Рассмотрим упрощённый алгоритм Рабина — Карпа, используя хеш-функцию в виде суммы кодов символов.

На каждой итерации цикла строка, которую мы сравниваем с шаблоном, меняется совсем немного: из её начала удаляется первый элемент, а в конец вставляется новый

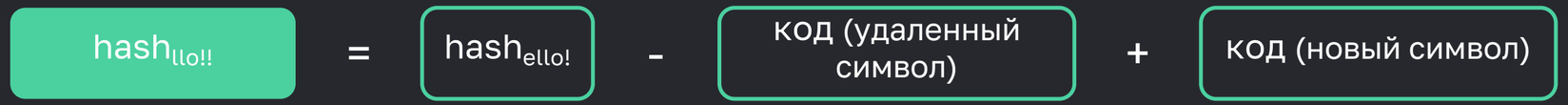



Шаблон

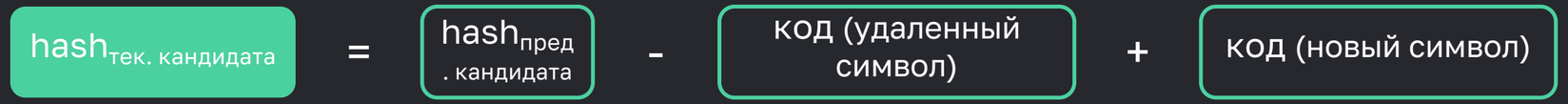
| H | e | l | l | o |
|---|---|---|---|---|

# Алгоритм Рабина — Карпа (упрощённый)

Хеш-функция такой строки меняется: $hash_{llo!!} = hash_{ello!} - code_{e} + code_{!}$




Иначе говоря, мы имеем формулу:




Поэтому нам не надо каждый раз считать хеш с нуля! Занимает O(1)

# Алгоритм Рабина — Карпа (упрощённый)

Асимптотика памяти остается O(1). При использовании хорошей хеш-функции асимптотика времени становится O(n), т. к. число коллизий (т.е. когда нам приходится циклом за O(k) проверять совпадение) встречается O(1) раз за всю работу программы

Настоящий алгоритм Рабина-Карпа использует хорошую хеш-функцию с простым числом. С ней тоже можно провернуть трюк с быстрым вычислением хеша для следующей итерации, но оно математически сложнее

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Функция поиска p в s

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий
Вычисляем хеш от p

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Пробегаем циклом по индексам s, которые будем проверять на начало потенциального совпадения с p

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Если это первая итерация, честно посчитаем хеш от первых k символов

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Иначе быстро сгенерируем хеш для следующих k символов, используя предыдущее значение такого хеша

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Если хеш следующих k символов не совпадает с хешом для p, то по правилами хеш-функции совпадения точно нет

# Алгоритм Рабина — Карпа (упрощённый)

Псевдокод:

```
find(s, p):
  p_hash = hash(p)
  for i от 0 до длина(s)
    if i = 0
      h = hash(s[0…k-1])
    else
      h -= код(s[i-1])
      h += код(s[i+k-1])
    if h != p_hash
      i неподходит
    else
      for j от 0 до длина(p)
        if s[i+j] != p[j]
          i не подходит
      if i подошёл
        return i
```

Комментарий

Если хеши совпали, то совпадение следующих k символов с p возможно. Проверим оно ли это