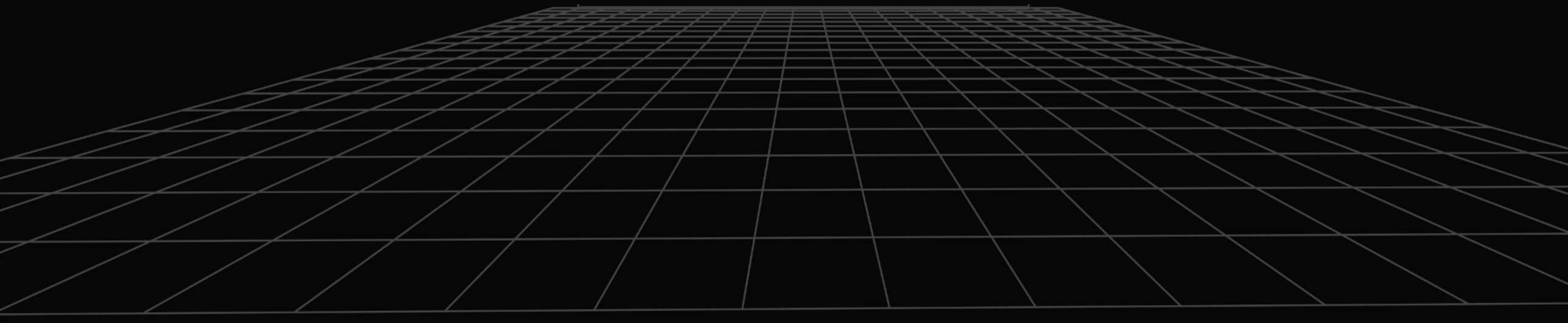
СЛОВАРИ

Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

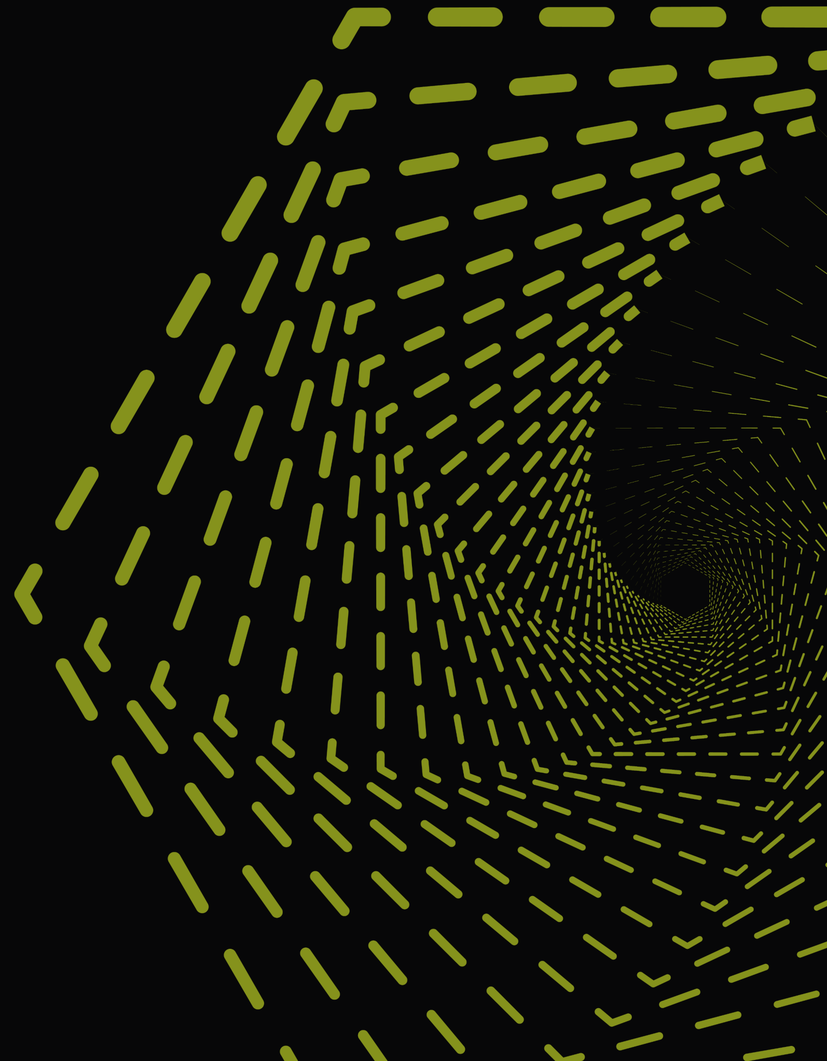


ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать
   в любое время

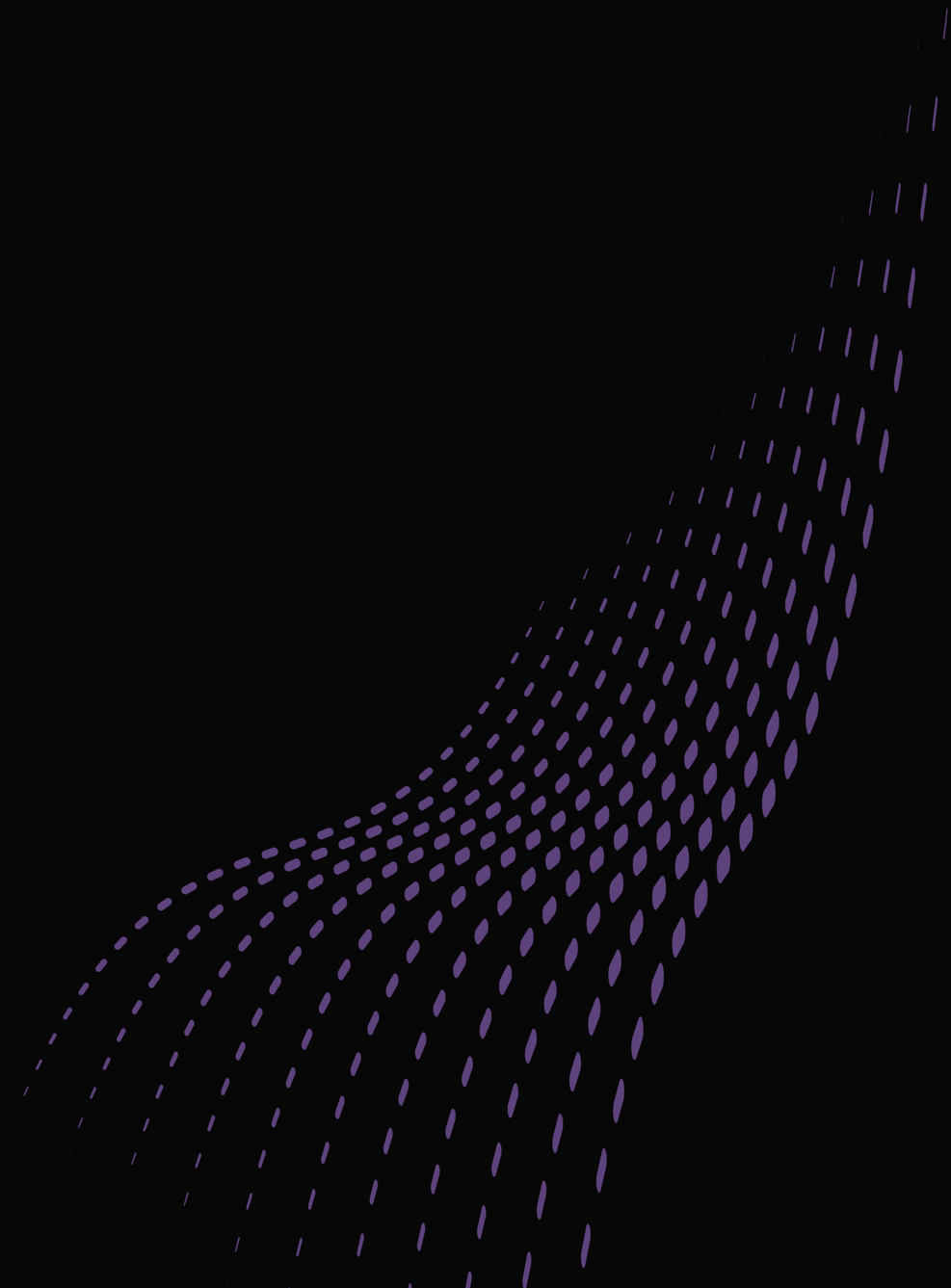
2. вопросы голосом задаем
   по поднятой руке в Zoom

3. ответы на вопросы будут
   в запланированных местах



ПЛАН ЛЕКЦИИ

1. Устройство хэш-таблиц
2. Разрешение коллизий методом цепочек
3. Разрешение коллизий методом открытой адресации
4. Внутреннее устройство словарей в Go
5. Тонкости словарей в Go



УСТРОЙСТВО ХЭШ-ТАБЛИЦЫ

Идентификатор	Баланс
1	200
3	500

Представим, что нам нужно хранить пару идентификатора пользователя с его балансом и уметь быстро искать баланс пользователя по идентификатору

МАССИВ ПАР

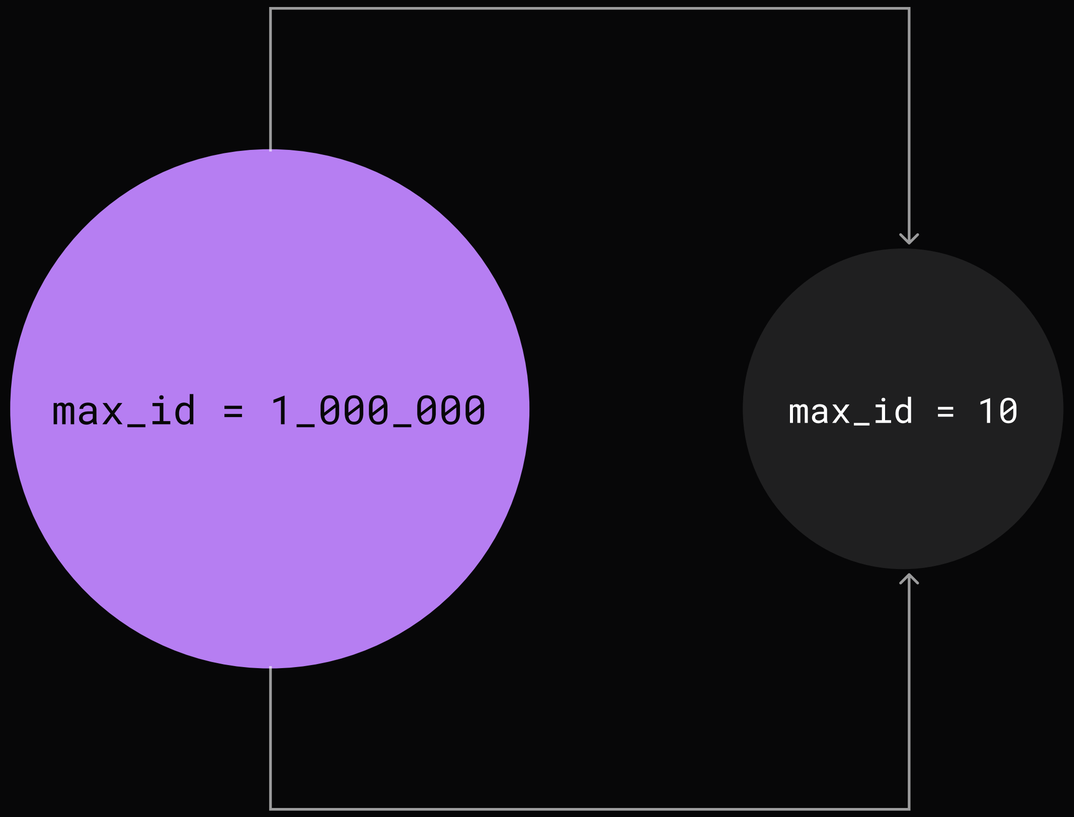
Будет работать медленно, так как в худшем случае придется пробегаться по всему массиву, чтобы найти в нем значение *(можно улучшить, путем поддержания отсортированного порядка элементов)*

idx	key	value
0	3	500
1	1	200
...		

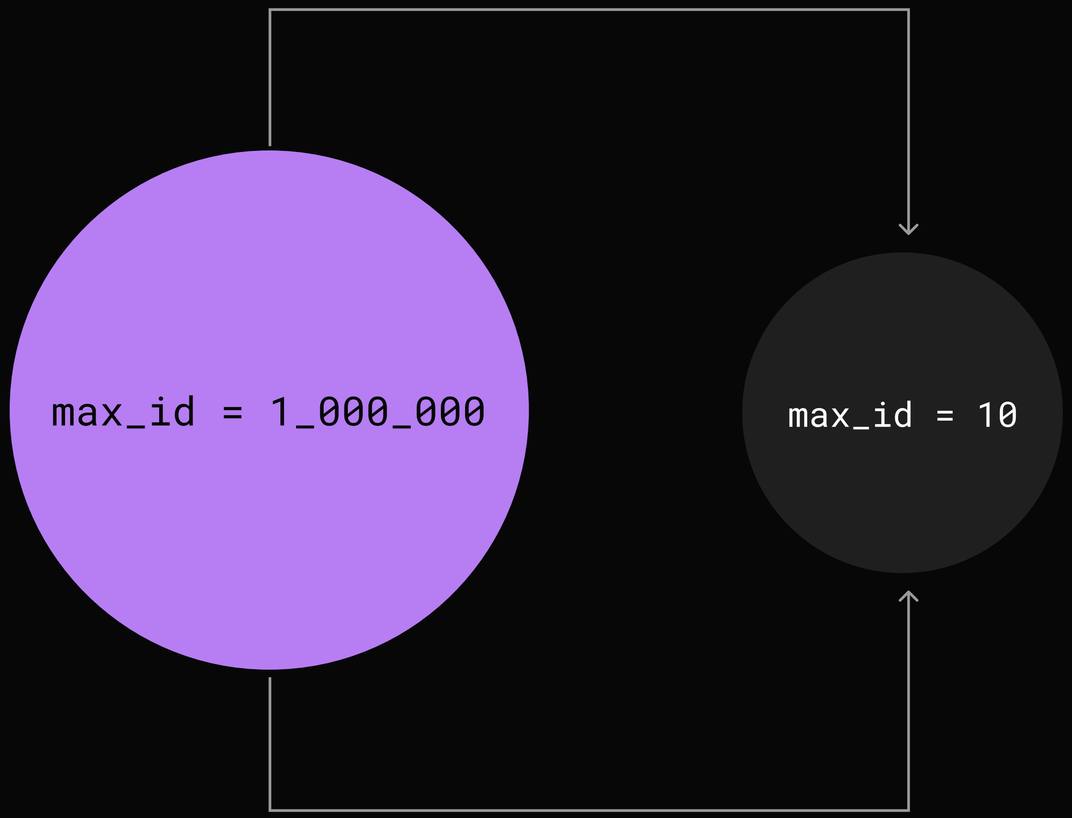
DIRECT ACCESSING

Будет работать быстро, но **массив получится разряженным** – если нужно хранить не инкрементальные идентификаторы пользователей, то у нас будут пустые места в массиве *(например, когда будет 5 пользователей, а максимальный идентификатор пользователя будет равен 1_000_000)*

Хотелось бы не терять в скорости,
но в тоже самое время избавиться от
пустых мест в массиве, преобразовывая
идентификаторы пользователей из
большого множества в маленькое
множество *(представление)*...



Нужен преобразователь — **хэш функция**



ХОРОШАЯ ХЭШ-ФУНКЦИЯ

1. **Эффективность** – вычисление преобразования не должно занимать много времени

2. **Необратимость** – не должно быть функции для обратного преобразования *(например, f1(547) => 3, f2(3) !=> 547)*

3. **Детерминированность** – для одного и того же ключа должна возвращать один и тот же результат

4. **Распределенность** – должна равномерно распределять данные по множеству *(например, не должна всегда возвращать 0)*

ПРИМИТИВНАЯ ХЭШ-ФУНКЦИЯ

F (x,capacity) = x % capacity

Insert(3, 300)

F (3, 5) = 3 % 5 = 3

Insert(317, 500)

F (317, 5) = 317 % 5 = 2

Insert(1_000_000, 700)

F (1_000_000, 5) = 1_000_000 % 5 = 0

idx	value	
0	700	// key = 1_000_000
1		
2	500	// key = 317
3	300	// key = 3
4		

КОЛЛИЗИИ

Ситуация, когда для разных ключей мы получаем один тот же результат хэш-функции.

Конечно, коллизии будут, так как мы преобразовываем большое множество в маленькое, поэтому определенному набору чисел будет соответствовать только одно число.

```
Insert(8, 300)
F (8, 5) = 8 % 5 = 3

Insert(1_000_003,700)
F (1_000_003, 5) = 1_000_003 % 5 = 3
```

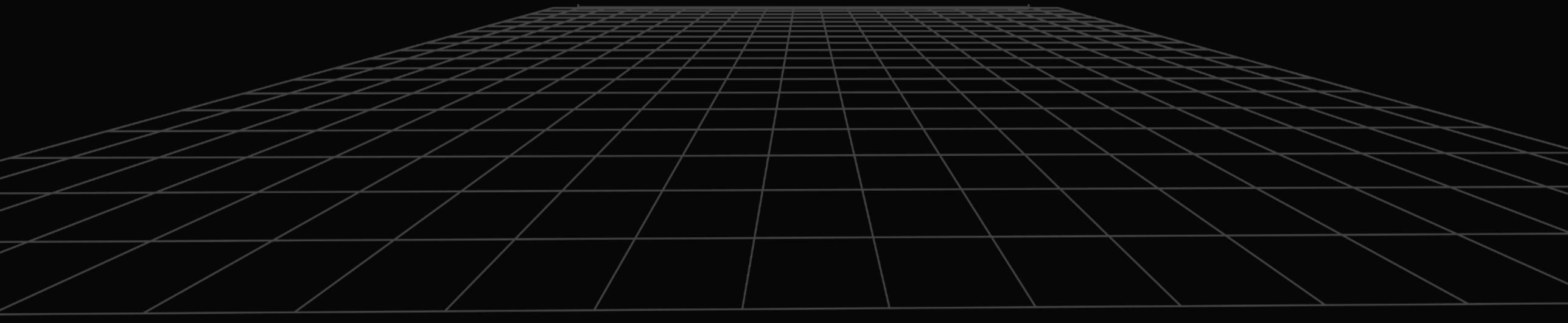
Коллизий не будет если только размер входного множества будет меньше выходного, но в большинстве случаев коллизий не избежать, но наша задача – **научиться работать с хэш коллизиями!**

РАЗРЕШЕНИЕ КОЛЛИЗИЙ

1. Метод цепочек
2. Метод открытой адресации

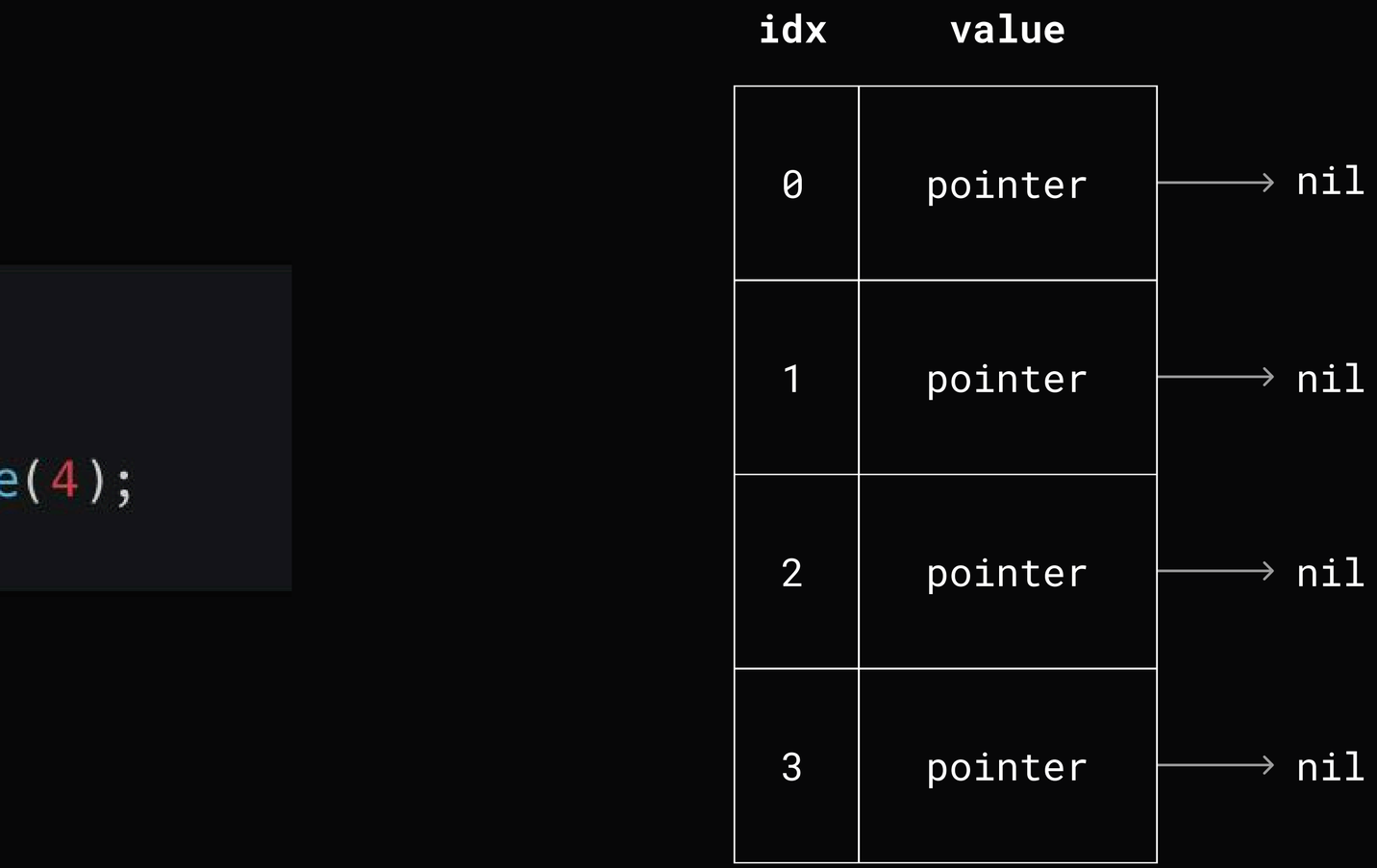
FAQ

Устройство хэш-таблицы

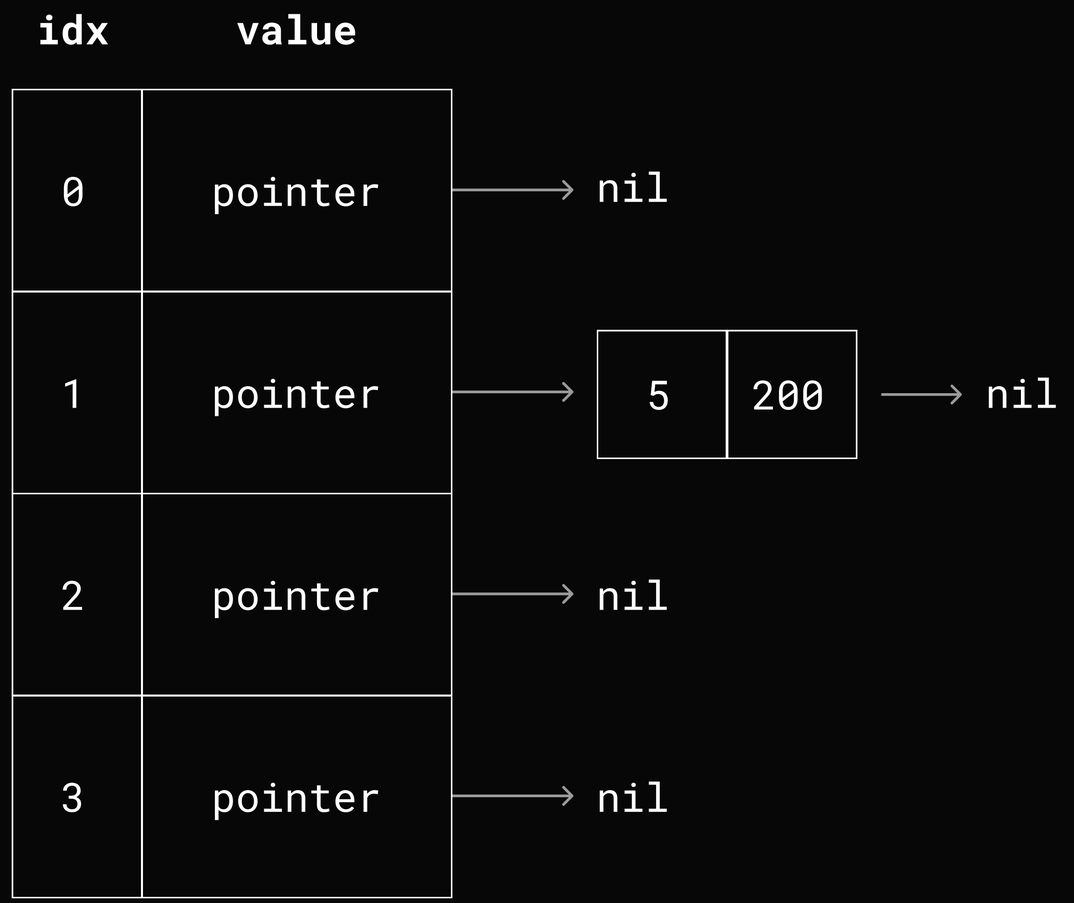


МЕТОД ЦЕПОЧЕК

```
1 HashTable table(4);
```

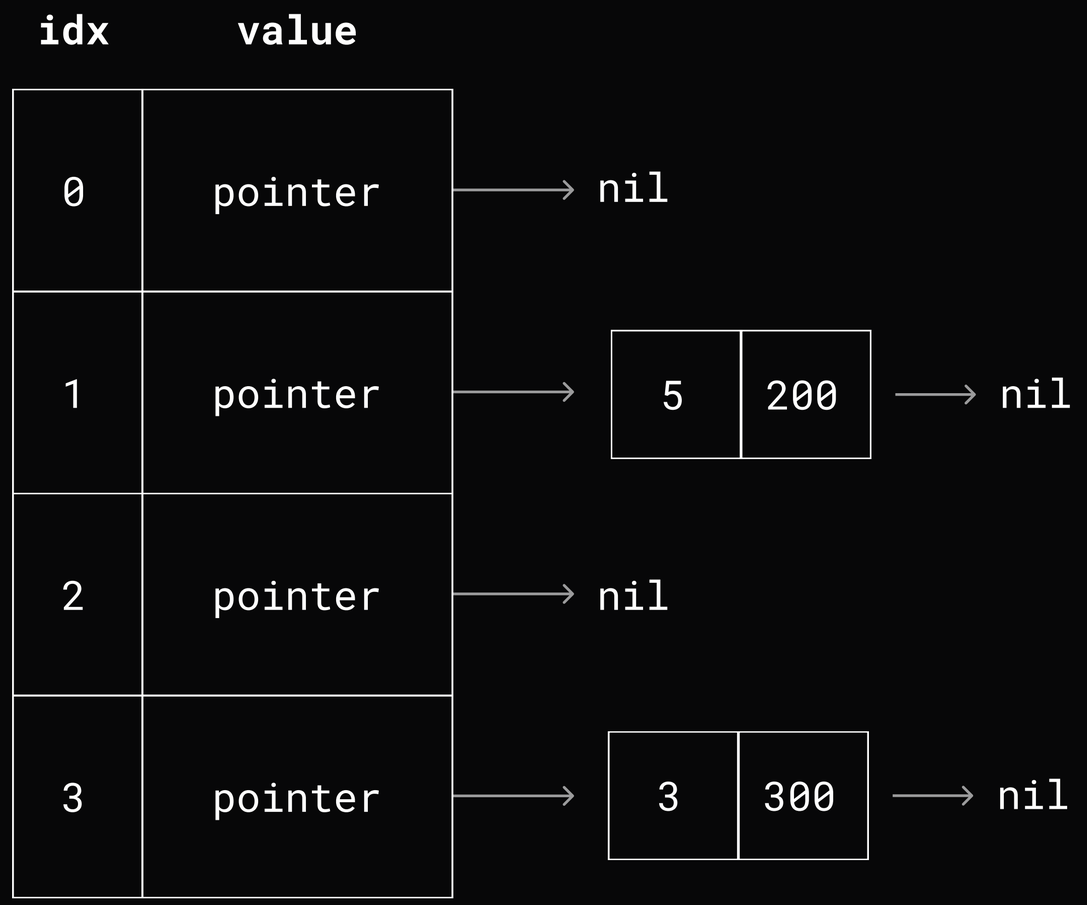


```
1 HashTable table(4);
2 table.insert(5, 200); // 5%4=1
```

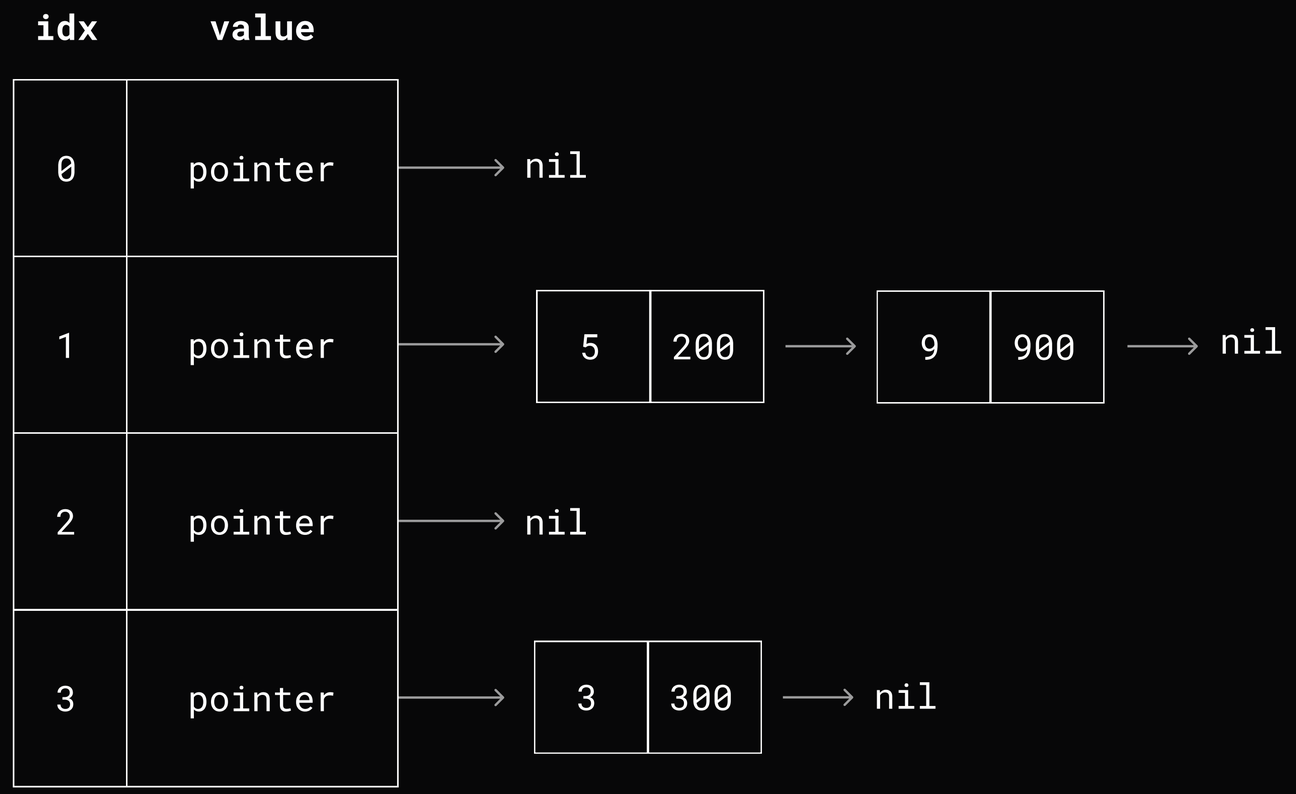


В качестве цепочек можно использовать не только связные списки, но и массивы, деревья и так далее...

```
1 HashTable table(4);
2 table.insert(5, 200); // 5%4=1
3 table.insert(3, 300); // 3%4=3
```



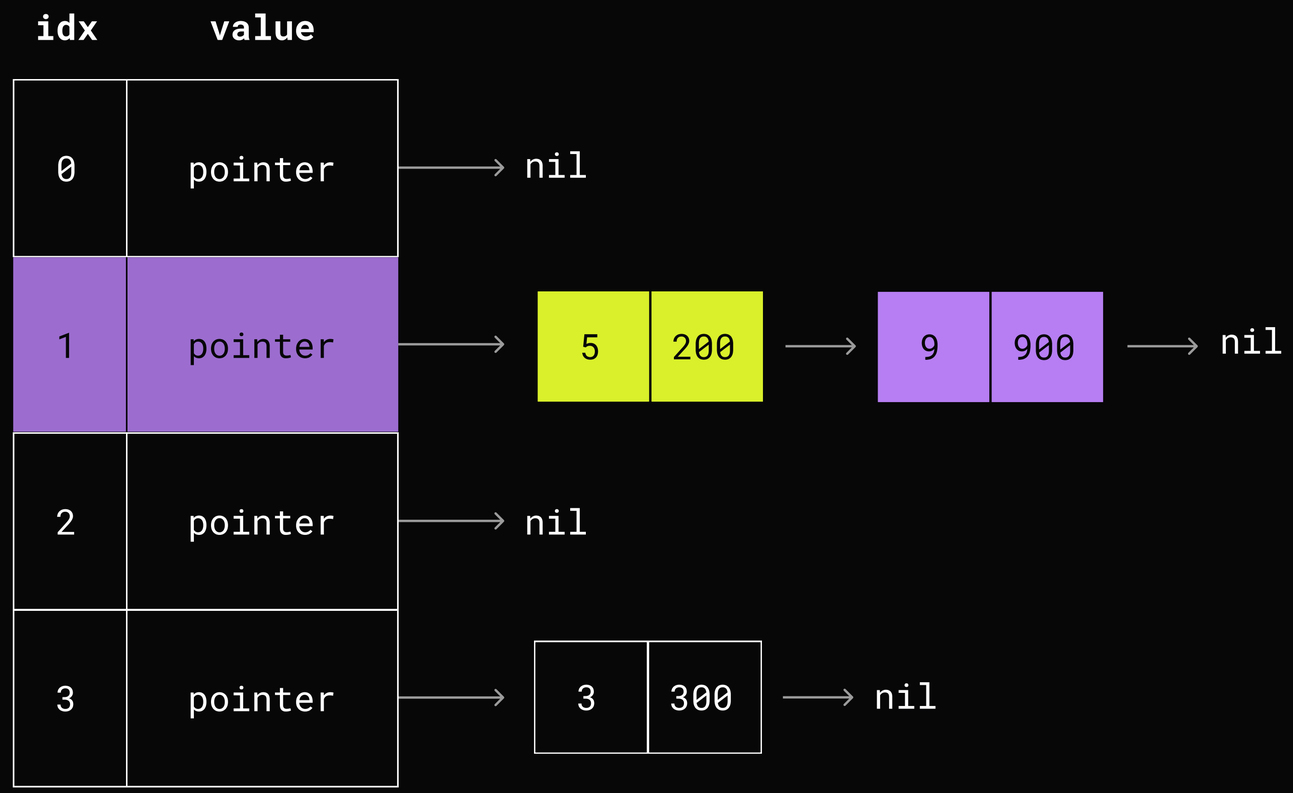
```
1 HashTable table(4);
2 table.insert(5, 200); // 5%4=1
3 table.insert(3, 300); // 3%4=3
4 table.insert(9, 900); // 9%4=1
```



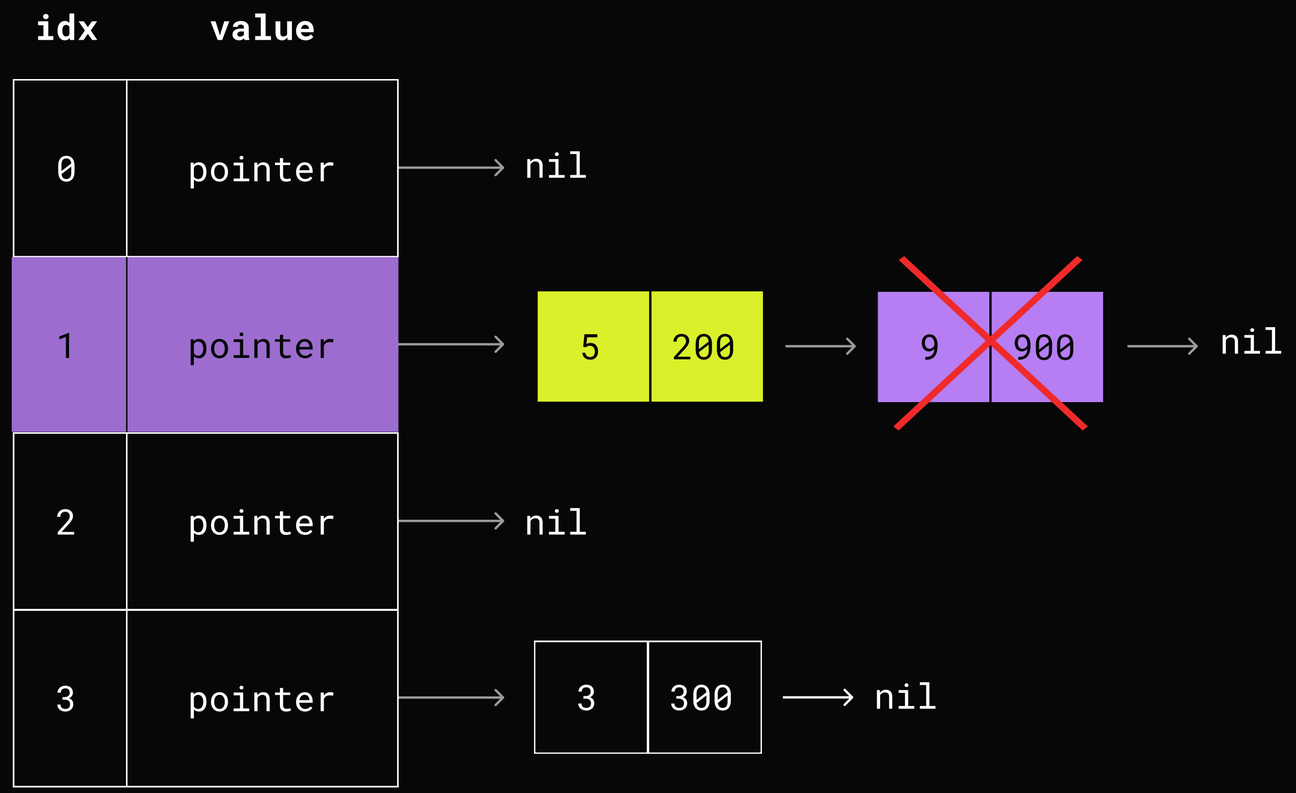
Terminal: question + ⚙ –

ЗАЧЕМ ХРАНИТЬ КЛЮЧ В БАКЕТЕ?

```
1 HashTable table(4);
2 table.insert(5, 200); // 5%4=1
3 table.insert(3, 300); // 3%4=3
4 table.insert(9, 900); // 9%4=1
5 table.find(9);        // 9%4=1
```



```
1 HashTable table(4);
2 table.insert(5, 200); // 5%4=1
3 table.insert(3, 300); // 3%4=3
4 table.insert(9, 900); // 9%4=1
5 table.find(9);        // 9%4=1
6 table.remove(9);      // 9%4=1
```

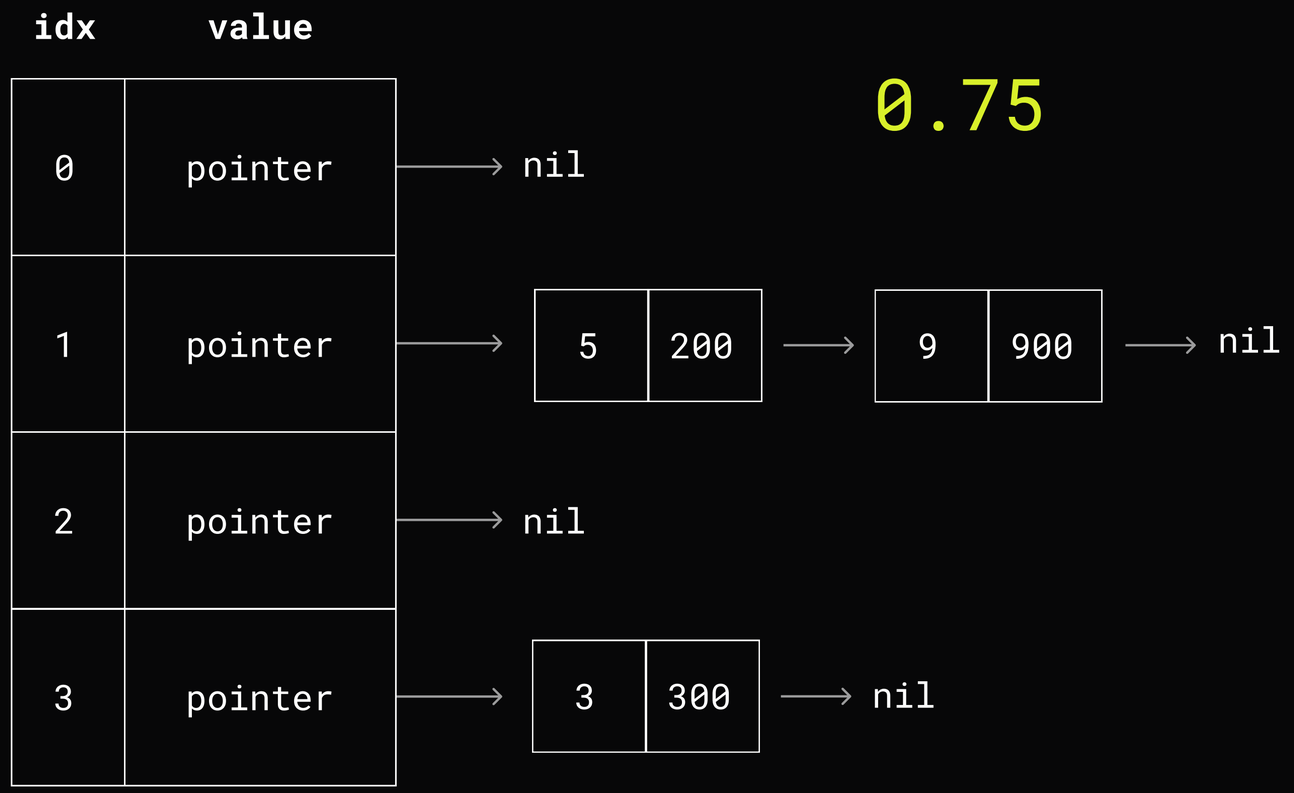


Если хэш-таблица не будет расширяться, то коллизий будет все больше и больше (так как хэш-функции нужно преобразовывать ключи в очень маленькое множество)

LOAD FACTOR

Коэффициент загруженности хэш-таблицы, считается по формуле **f = size / buckets_size**

В большинстве реализаций, когда коэффициент больше **0.6/0.7**, то происходит **переиндексация**



Переиндексация хэш-таблицы практически тоже самое, что и реаллокация памяти для динамического массива, но разница в том, что мы **не можем просто бездумно копировать элементы!**

При реиндексации мы выделяем в два раза больше памяти и из-за этого меняется **capacity**, поэтому:

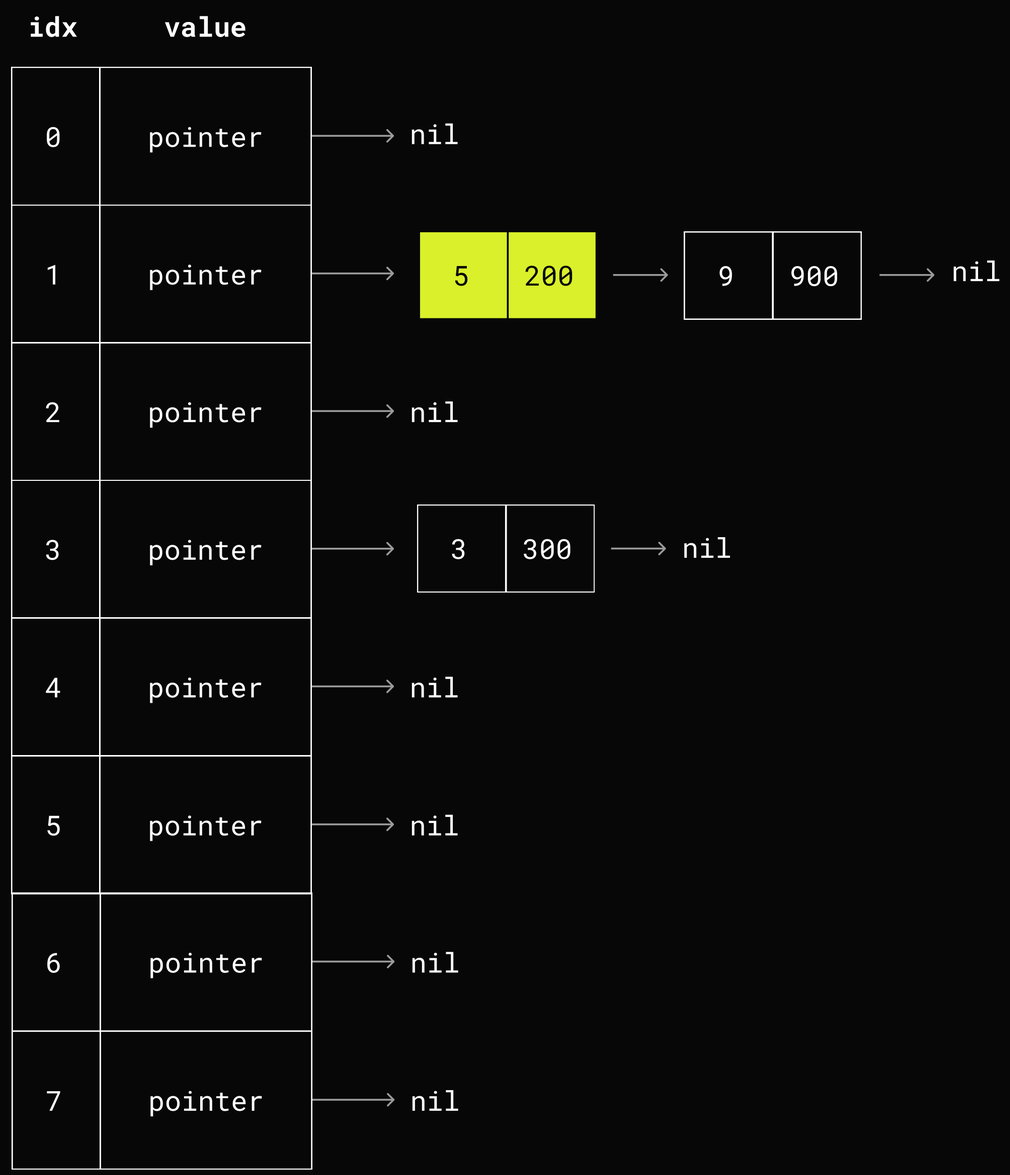
F(5,4) = 5 % 4 (capacity) = **1**

F(5,8) = 5 % 8 (capacity) = **5**

Итого: 1 != 5

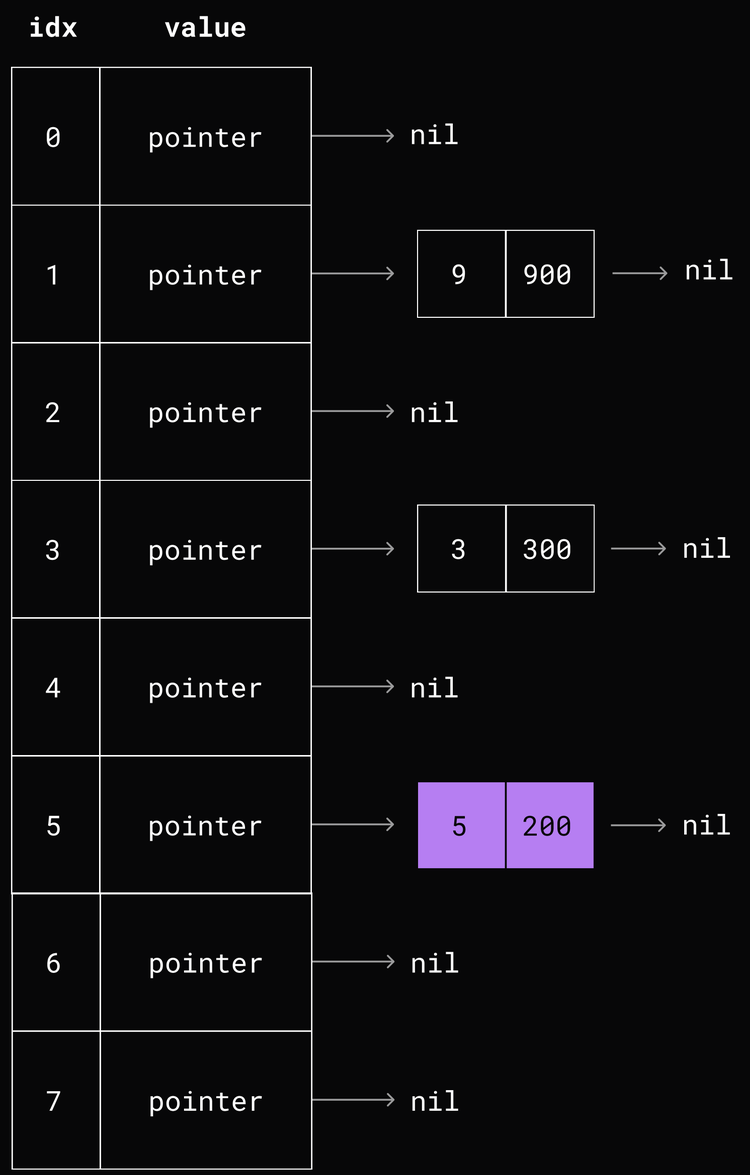
КОПИРОВАНИЕ

```
1 HashTable table(4);
2 table.find(5); // 5%8=5  != 1
3 table.find(3); // 3%8=3  == 3
4 table.find(9); // 9%8=1  == 1
```

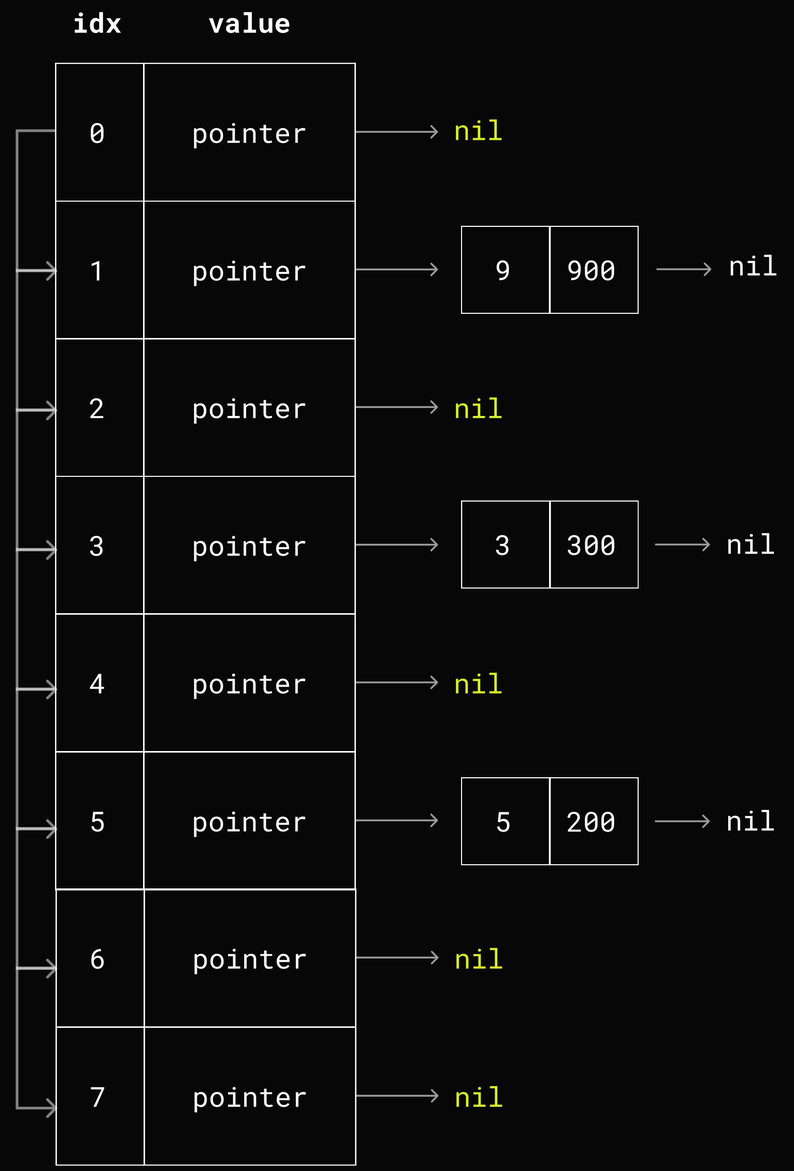


ПЕРЕИНДЕКСАЦИЯ

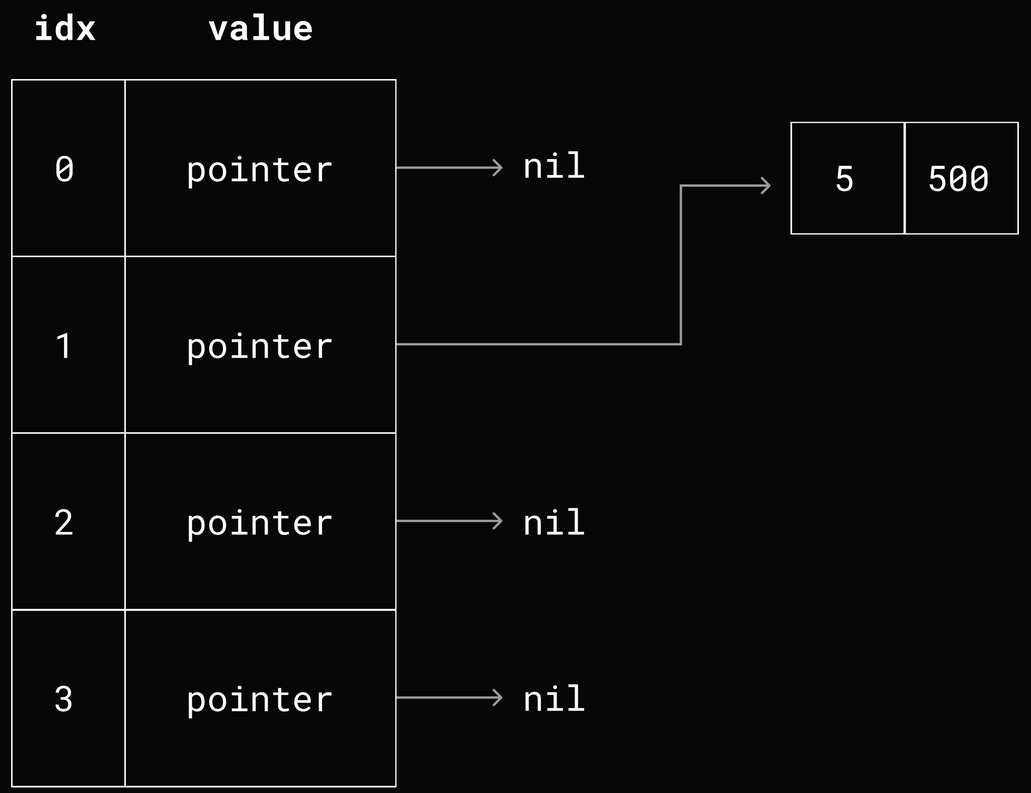
```
1 HashTable table(4);
2 table.find(5); // 5%8=5   == 5
3 table.find(3); // 3%8=3   == 3
4 table.find(9); // 9%8=1   == 1
```



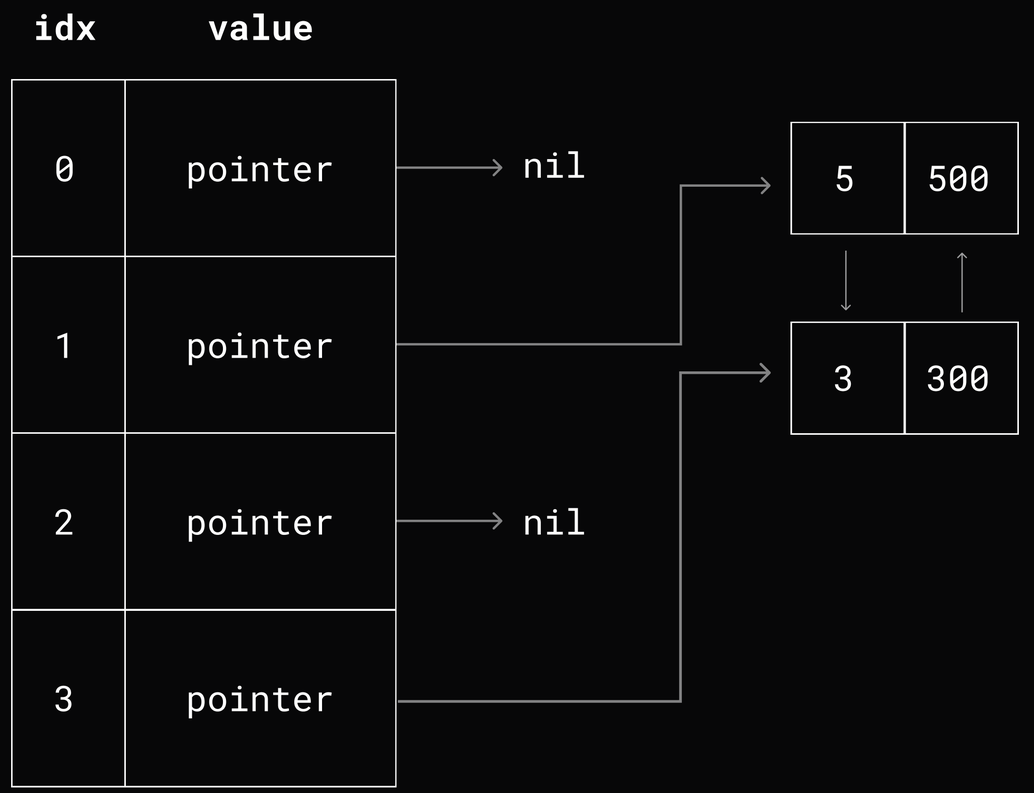
Как при проходе по хэш-таблице пропускать пустые бакеты?



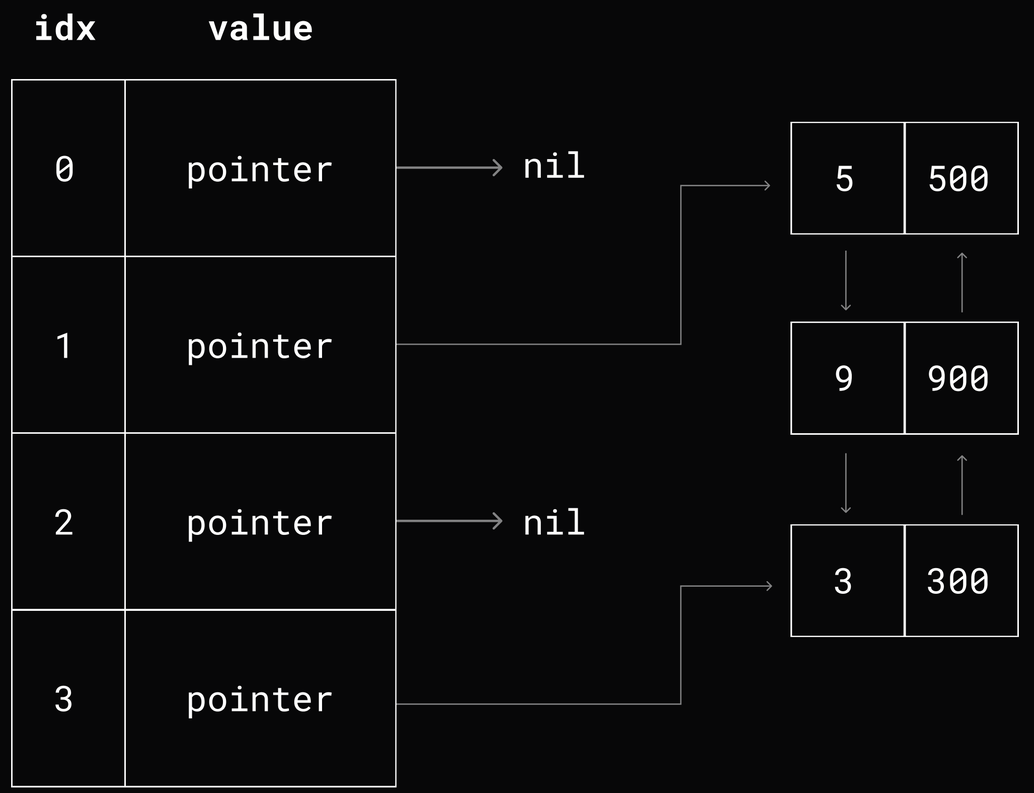
```
1 HashTable table(4);
2 table.insert(5, 500);
```



```
1 HashTable table(4);
2 table.insert(5, 500);
3 table.insert(3, 300);
```

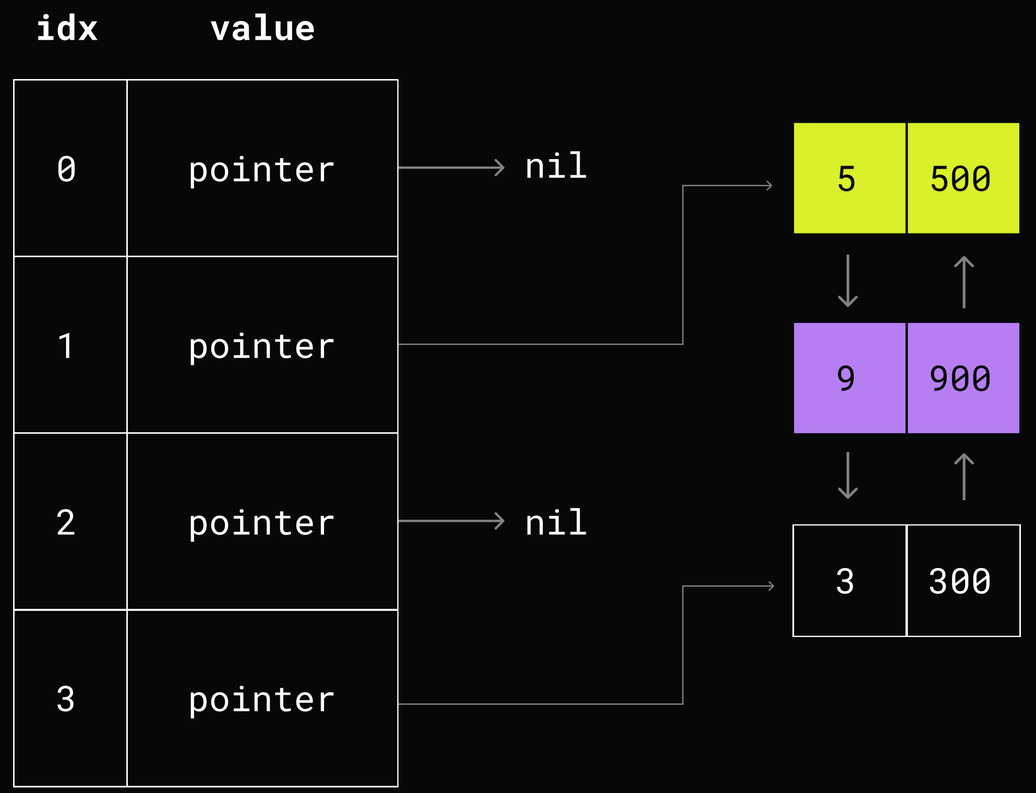


```
1 HashTable table(4);
2 table.insert(5, 500);
3 table.insert(3, 300);
4 table.insert(9, 900);
```

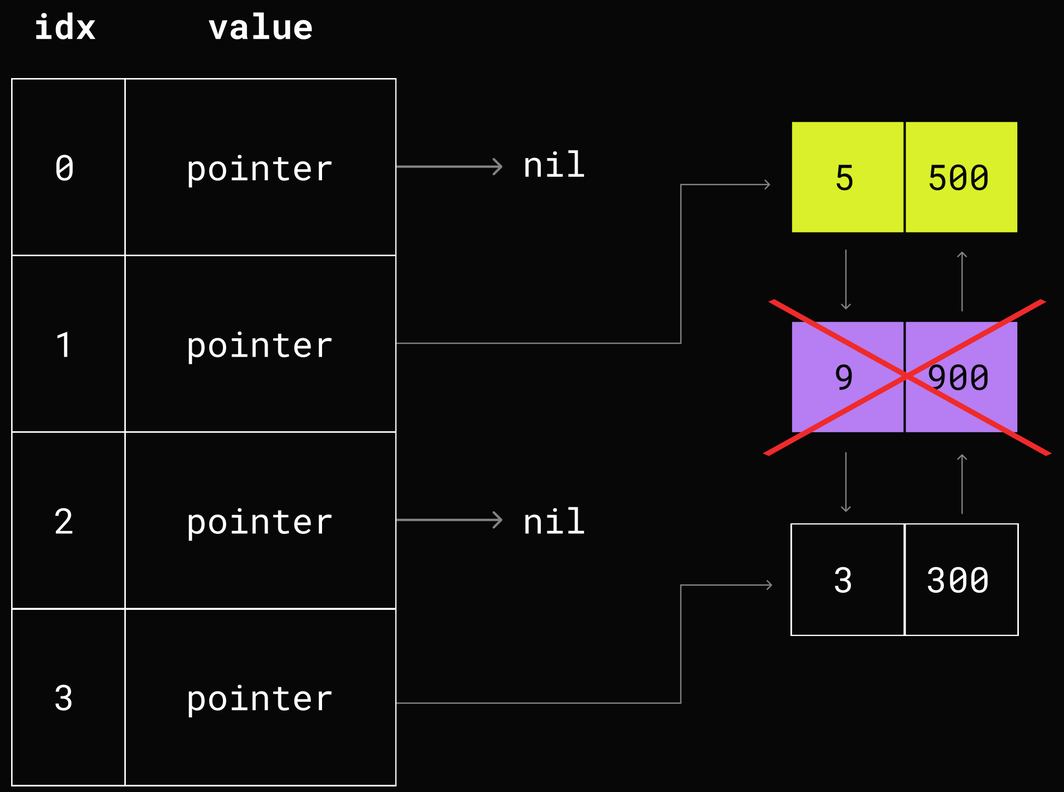


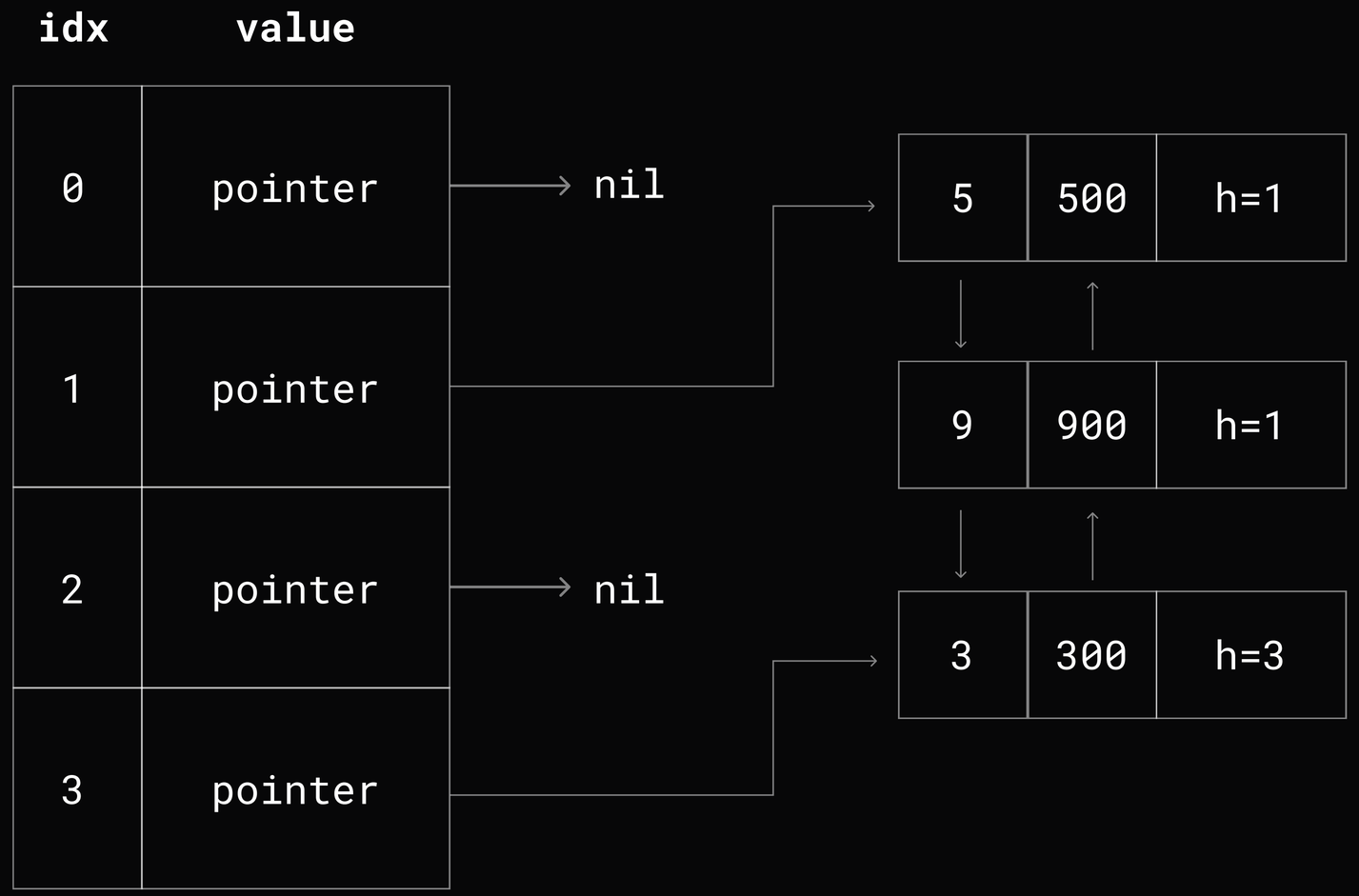
Теперь можно идти по связному списку, минуя пустые бакеты

```
1 HashTable table(4);
2 table.insert(5, 500);
3 table.insert(3, 300);
4 table.insert(9, 900);
5 table.find(9);
```



```
1 HashTable table(4);
2 table.insert(5, 500);
3 table.insert(3, 300);
4 table.insert(9, 900);
5 table.find(9);
6 table.erase(9);
```

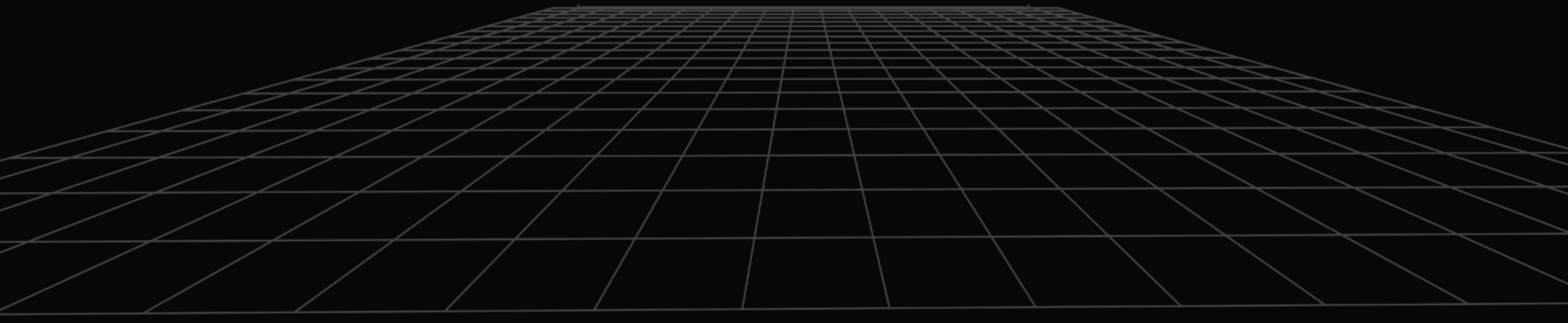




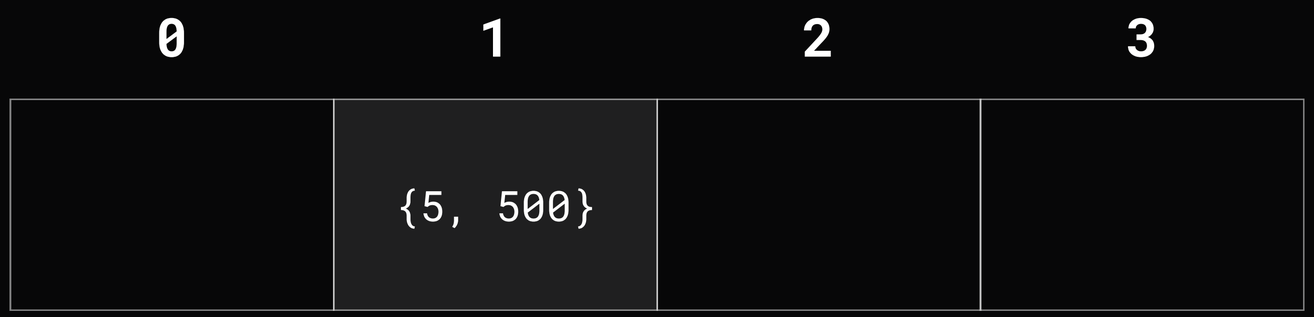
Еще можно, в случае необходимости, кэшировать хэш, чтобы не приходилось его пересчитывать при поиске и удалении

FAQ

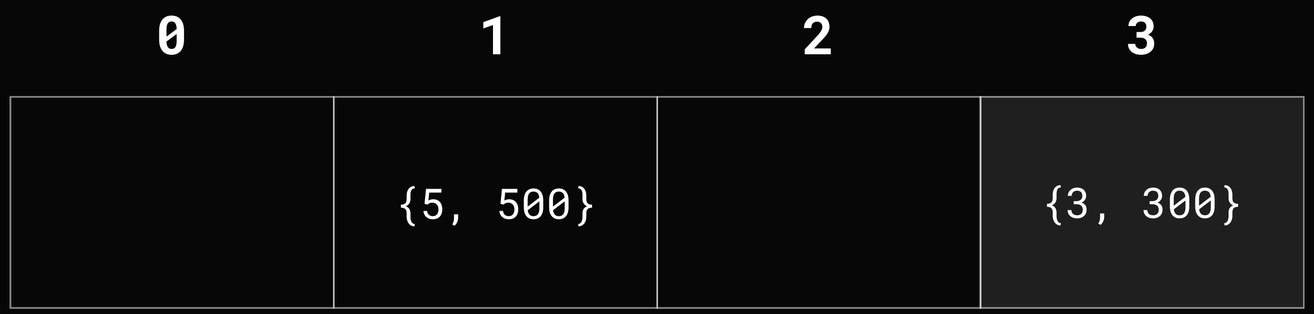
Метод цепочек



МЕТОД ОТКРЫТОЙ АДРЕСАЦИИ

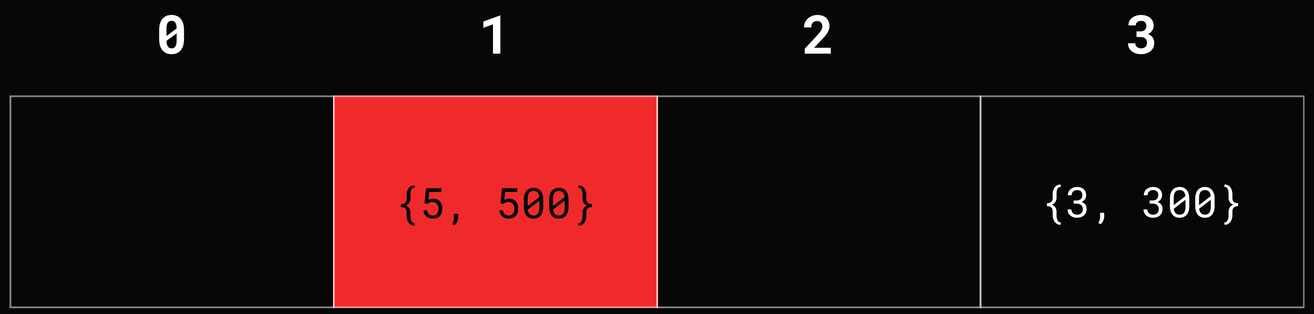



```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
```



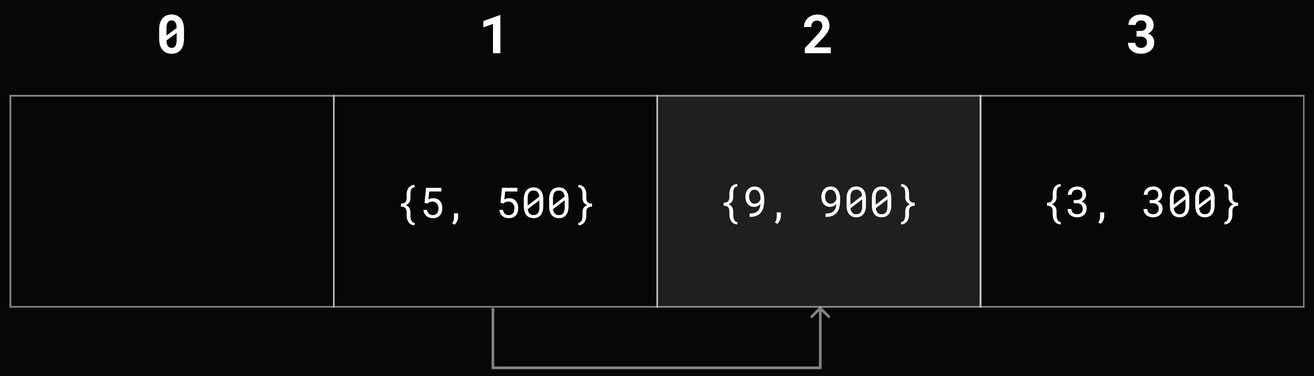

```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(3, 300); // 3%4=3
```

Ячейка занята - нужно пробировать




```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(3, 300); // 3%4=3
4 table.insert(9, 900); // 9%4=1
```

Идем по массиву, пока не найдем свободное место

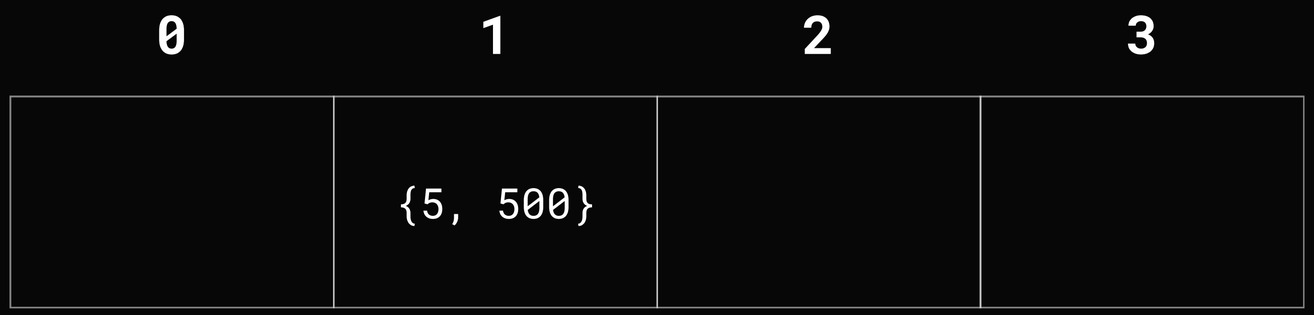



```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(3, 300); // 3%4=3
4 table.insert(9, 900); // 9%4=1
```

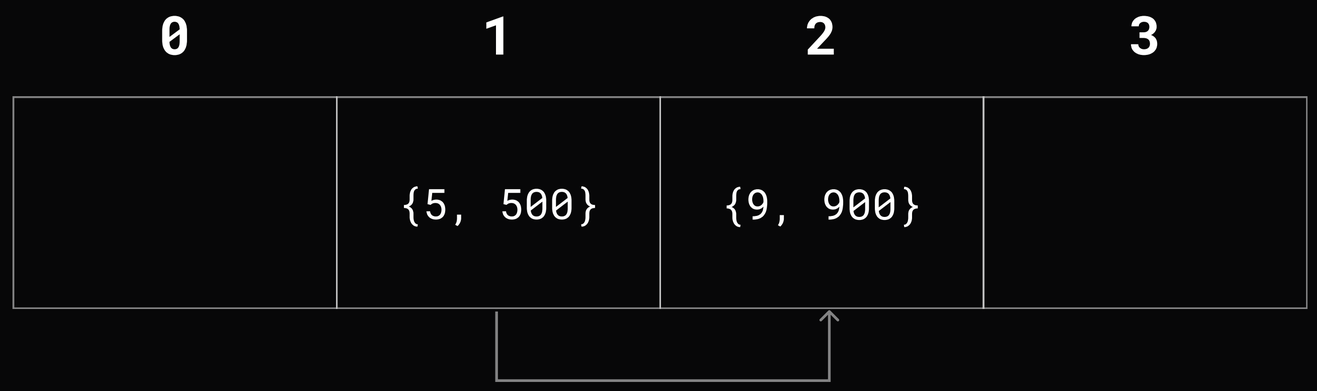
По хорошему, в этой ситуации пора уже расширять и переиндексировать таблицу, так как **коэффициент загруженности больше 60-70%**

Имея такую структуру хранения данных – получаем более дружелюбную структуру данных для кэша, но работает поиск хуже, потому что придется сравниваться до первого пустого элемента

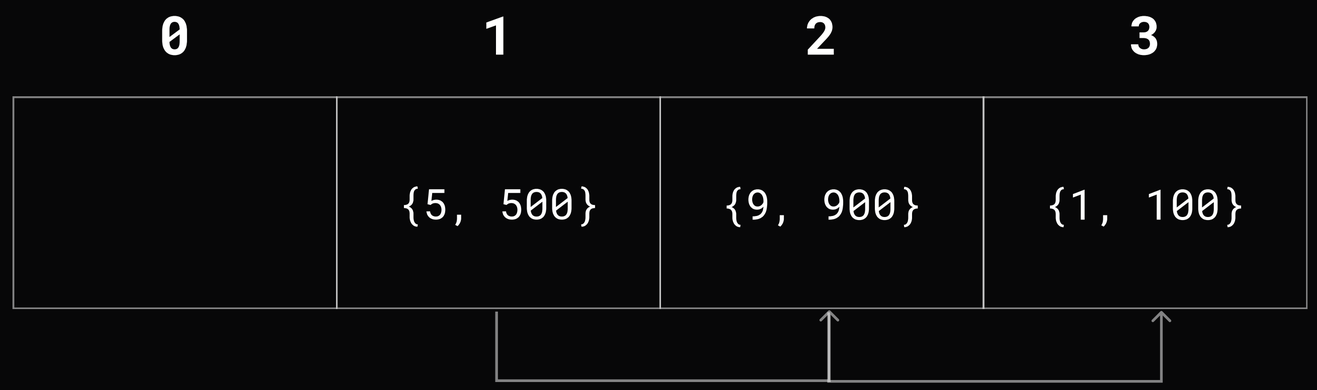
РАССМОТРИМ НЕ САМЫЙ ЛУЧШИЙ СЛУЧАЙ




```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
```



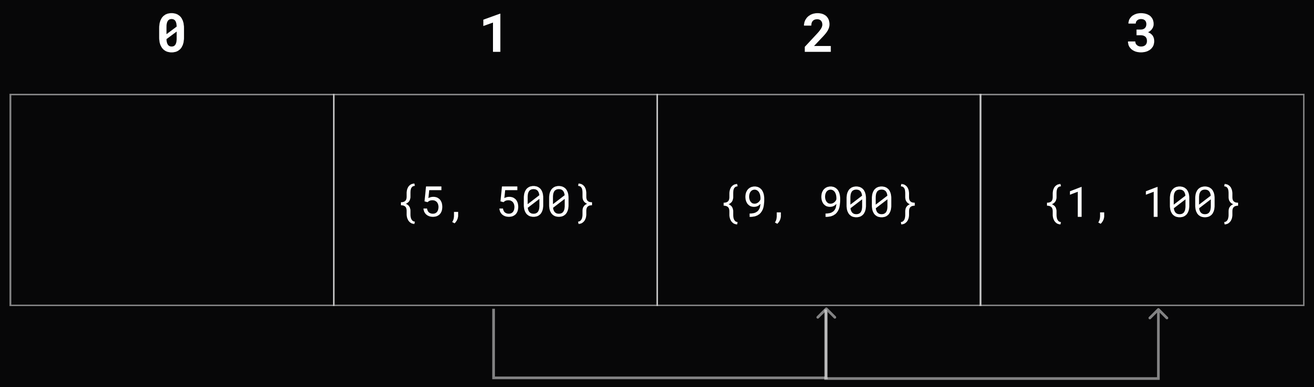

```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(9, 900); // 9%4=1
```




```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(9, 900); // 9%4=1
4 table.insert(1, 100); // 1%4=1
```

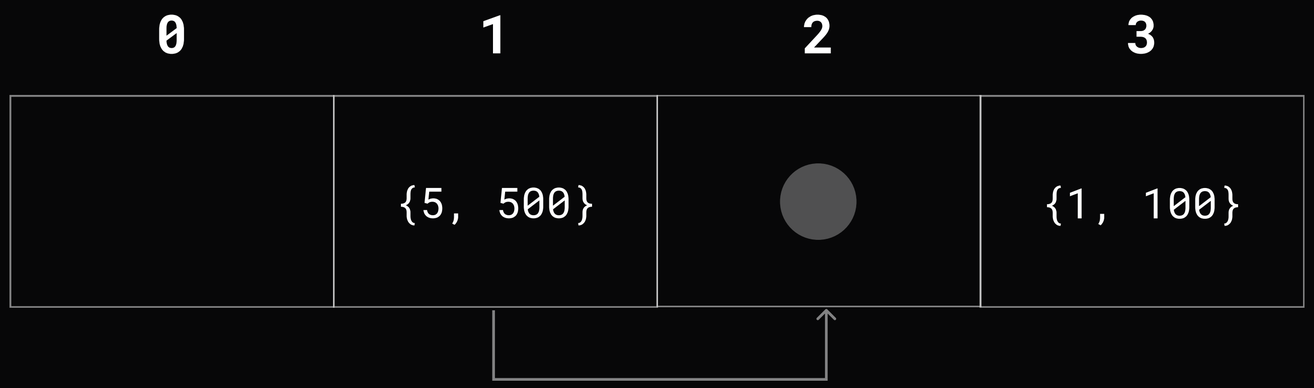
Можно ограничивать количество пробирований, например, если вы прыгнули 2–3 шага, но так и не нашли свободную ячейку — то можно расширять и переиндексировать таблицу

Поиск работает до тех пор, пока не не найдет пустую ячейку

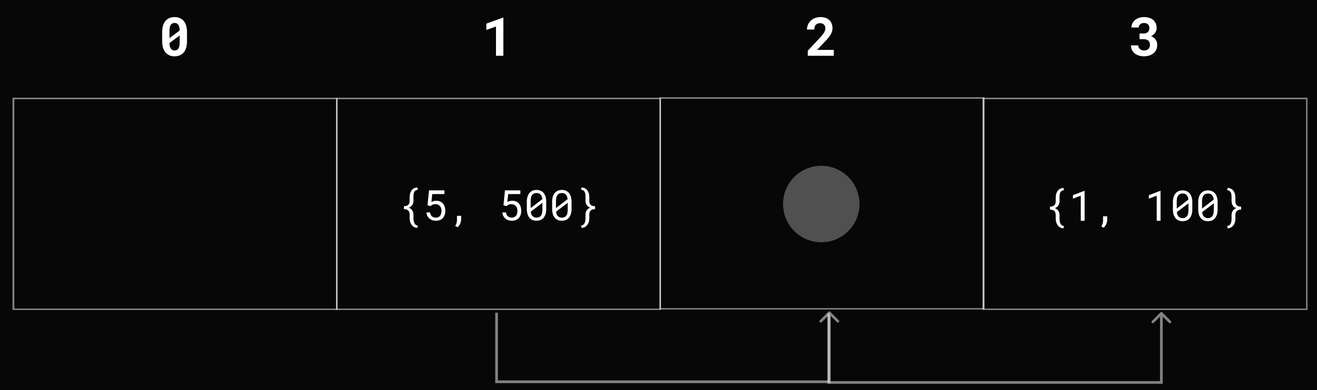



```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(9, 900); // 9%4=1
4 table.insert(1, 100); // 1%4=1
5 table.find(1);        // 1%4=1
```

При удалении важно помечать, что элемент удален,
либо сдвигать элементы с коллизиями влево




```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(9, 900); // 9%4=1
4 table.insert(1, 100); // 1%4=1
5 table.find(1);        // 1%4=1
6 table.remove(9);      // 9%4=1
```



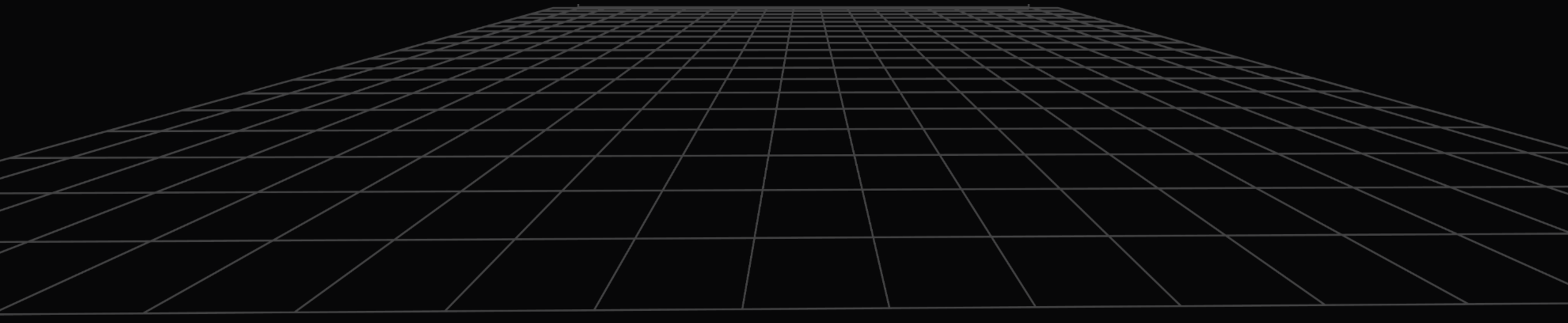

```
1 HashTable table(4);
2 table.insert(5, 500); // 5%4=1
3 table.insert(9, 900); // 9%4=1
4 table.insert(1, 100); // 1%4=1
5 table.find(1);        // 1%4=1
6 table.remove(9);      // 9%4=1
7 table.find(1);        // 1%4=1
```

ВИДЫ ПРОБИРОВАНИЯ

1. **Линейное** = `i + 1, i + 1, i + 1`
2. **Квадратичное** = `(i + i * i) / 2, (i + i * i) / 2, (i + i * i) / 2`
3. **Двойное хэширование** = `hash_fn2(i), hash_fn2(i), hash_fn2(i)`

FAQ

Метод открытой адресации



ВНУТРЕННЕЕ УСТРОЙСТВО СЛОВАРЯ В GO

```go
// A header for a Go map.
type hmap struct {
    // Note: the format of the hmap is also encoded in cmd/compile/internal/reflectdata/reflect.go.
    // Make sure this stays in sync with the compiler's definition.
    count     int // # live cells == size of map.  Must be first (used by len() builtin)
    flags     uint8
    B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
    noverflow uint16 // approximate number of overflow buckets; see incrnoverflow for details
    hash0     uint32 // hash seed

    buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==0.
    oldbuckets unsafe.Pointer // previous bucket array of half the size, non-nil only when growing
    nevacuate  uintptr        // progress counter for evacuation (buckets less than this have been evacuated)

    extra *mapextra // optional fields
}
```

Словарь - это **указатель на структуру hmap**

EXAMPLE

Map with function 1

EXAMPLE

Map with function 2

Terminal: question + ⌄ ⚙ –

ТИП ДАННЫХ ДЛЯ КЛЮЧА СЛОВАРЯ ДОЛЖЕН БЫТЬ СРАВНИВАЕМЫМ, НО ПОЧЕМУ?

EXAMPLE

Comparable keys

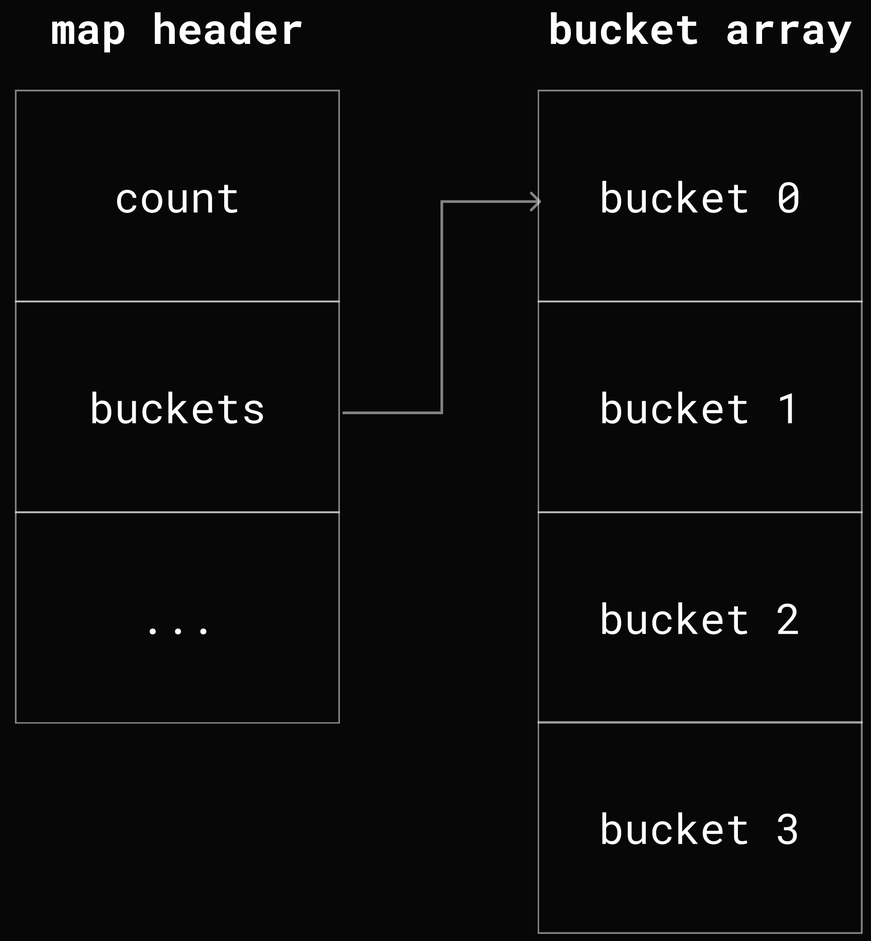
У каждого словаря разное **hash seed** значение,
поэтому у каждого объекта словаря будут
абсолютно разные хэш-значения для ключей

EXAMPLE

Hash functions

Словарь начинается с одного бакета,
когда не указывается количество элементов,
но указание размера n не означает создание
словаря с максимальным количеством элементов,
равным n *(выделяется место для, как минимум n элементов)*

```go
data := make(map[int]int, 20)
```

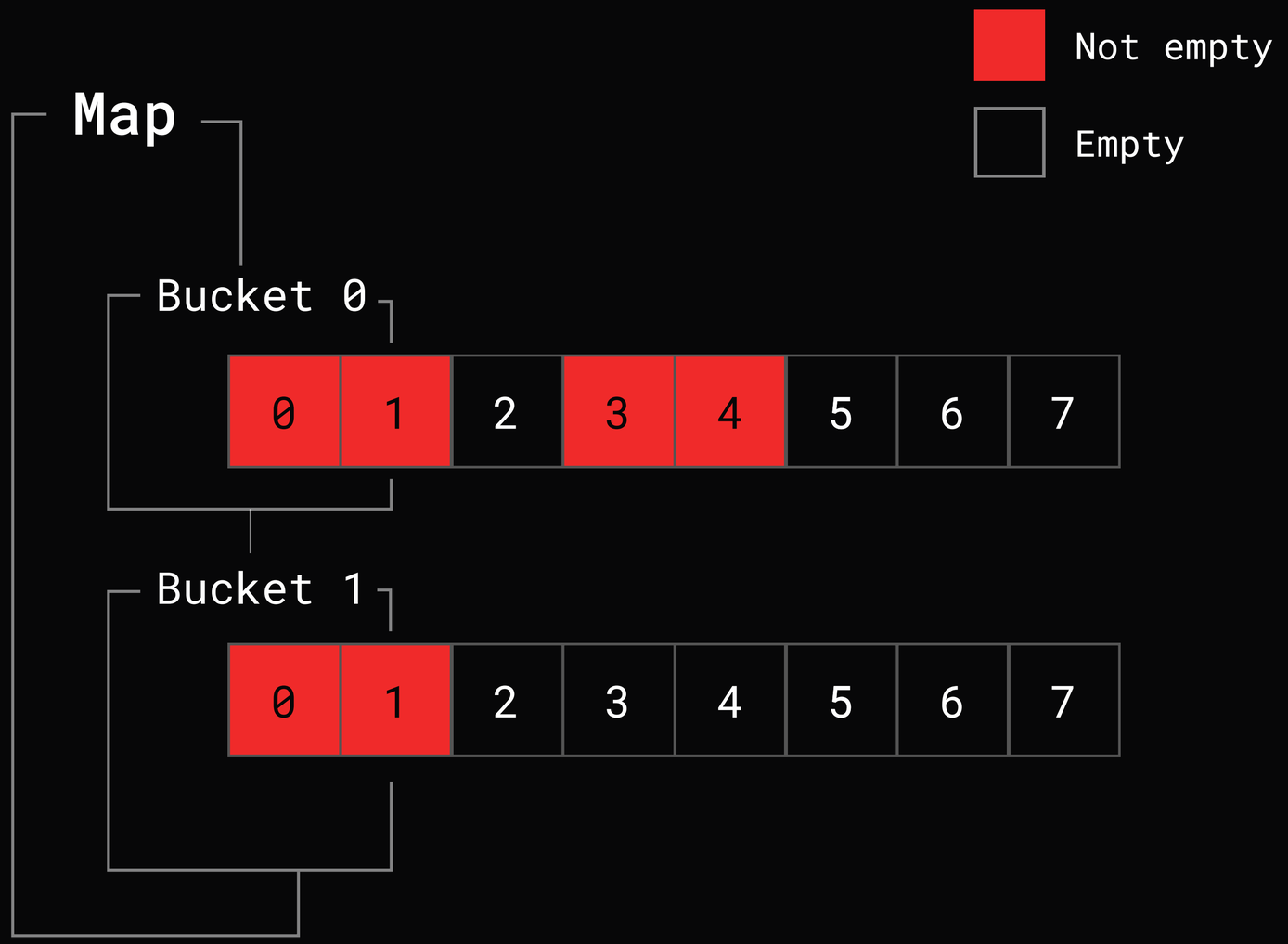


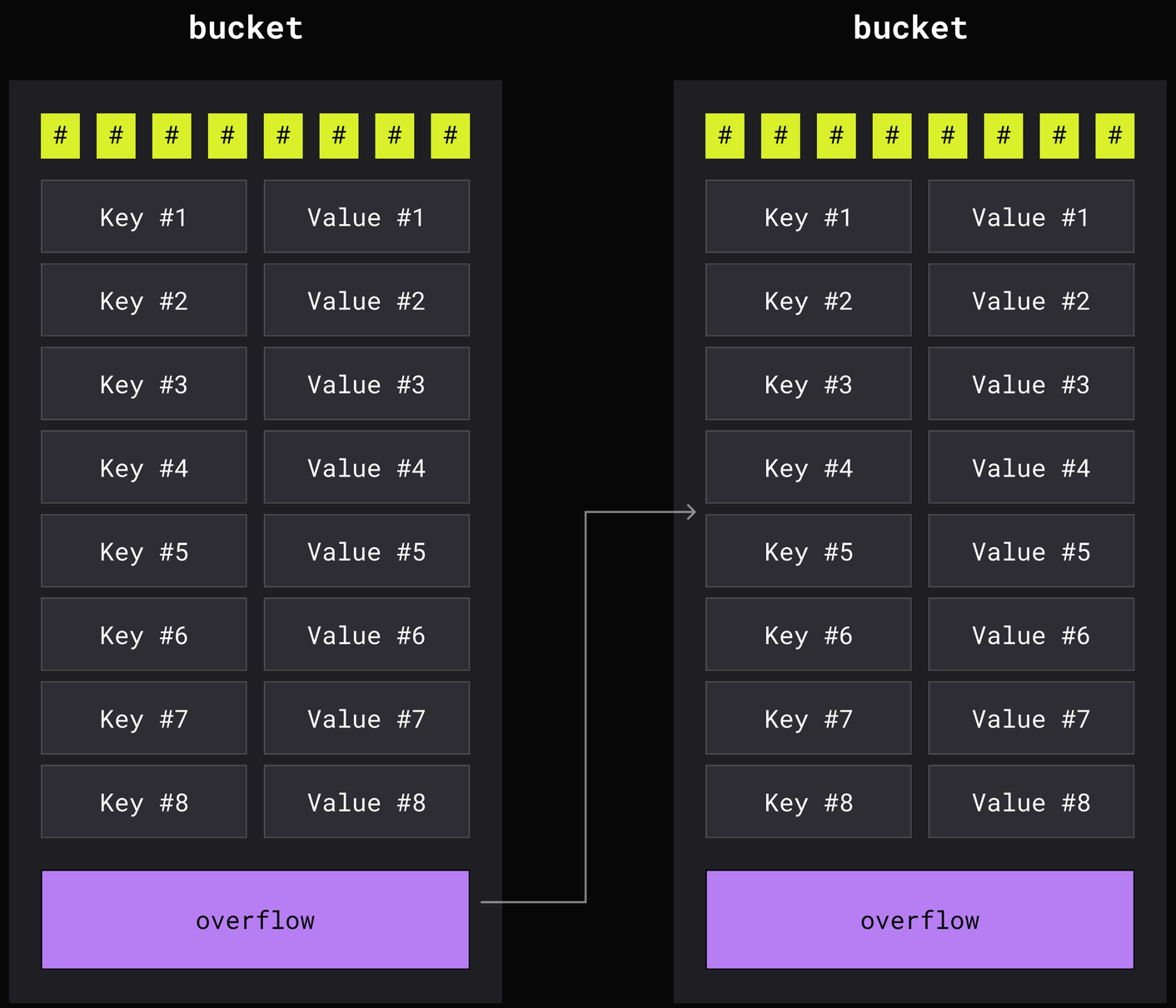
Screenshot

Hint Range	Bucket Count	Capacity
0 - 8	1	8
9 - 13	2	16
14 - 26	4	32
27 - 52	8	64
53 - 104	16	128
105 - 208	32	256
209 - 416	64	512
417 - 832	128	1024
833 - 1664	256	2048

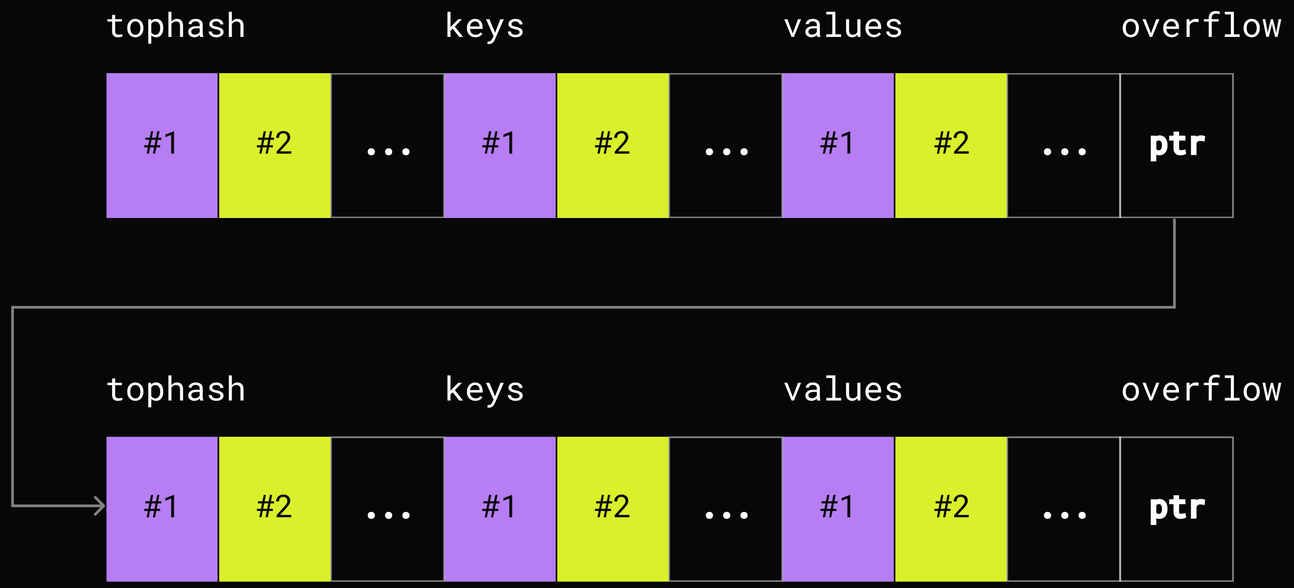
Terminal: question + ⚙ –

КАК УСТРОЕН БАКЕТ?


Not empty
Empty
Map
Bucket 0
0
1
2
3
4
5
6
7
Bucket 1
0
1
2
3
4
5
6
7


bucket
bucket
# # # # # # # #
Key #1
Value #1
Key #2
Value #2
Key #3
Value #3
Key #4
Value #4
Key #5
Value #5
Key #6
Value #6
Key #7
Value #7
Key #8
Value #8
overflow
# # # # # # # #
Key #1
Value #1
Key #2
Value #2
Key #3
Value #3
Key #4
Value #4
Key #5
Value #5
Key #6
Value #6
Key #7
Value #7
Key #8
Value #8
overflow

```go
// My implementation
type bmap struct {
    tophash  [8]uint8
    keys     [8]int
    values   [8]int
    overflow *bmap
}
```



Terminal: question

ЗАЧЕМ НУЖНО УПАКОВЫВАТЬ КЛЮЧИ

и значения в бакете таким образом?

EXAMPLE

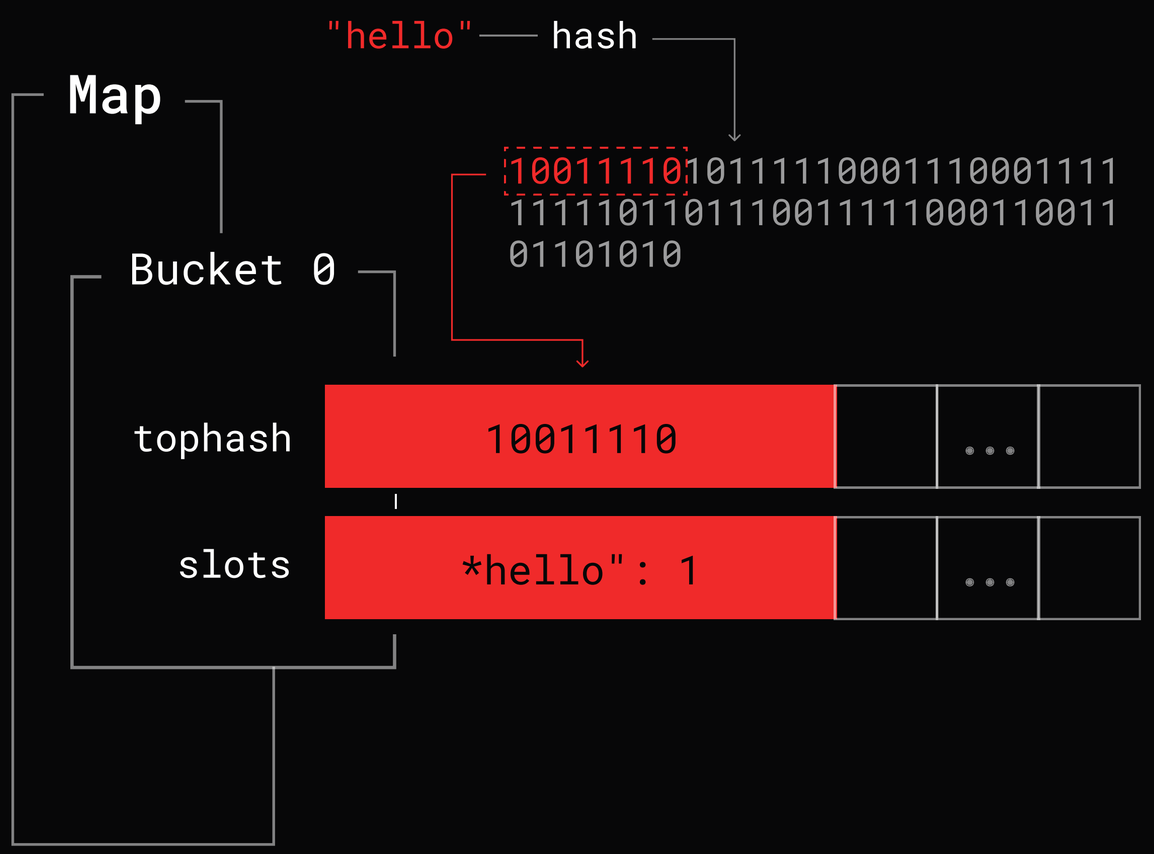
Bucket structure

Terminal: question + ⚙ –

ЗАЧЕМ НУЖЕН TOPHASH?

Когда вы хотите добавить, удалить или найти элемент, речь идет не только о проверке наличия места в бакете, но и о сравнении ключа с каждым существующим ключом в этом бакете (становится еще хуже, когда есть бакеты переполнения)

Если после сравнения tophash они совпадают, это означает, что ключи «могут» быть одинаковыми - затем **Go** переходит к более медленному процессу проверки идентичности ключей

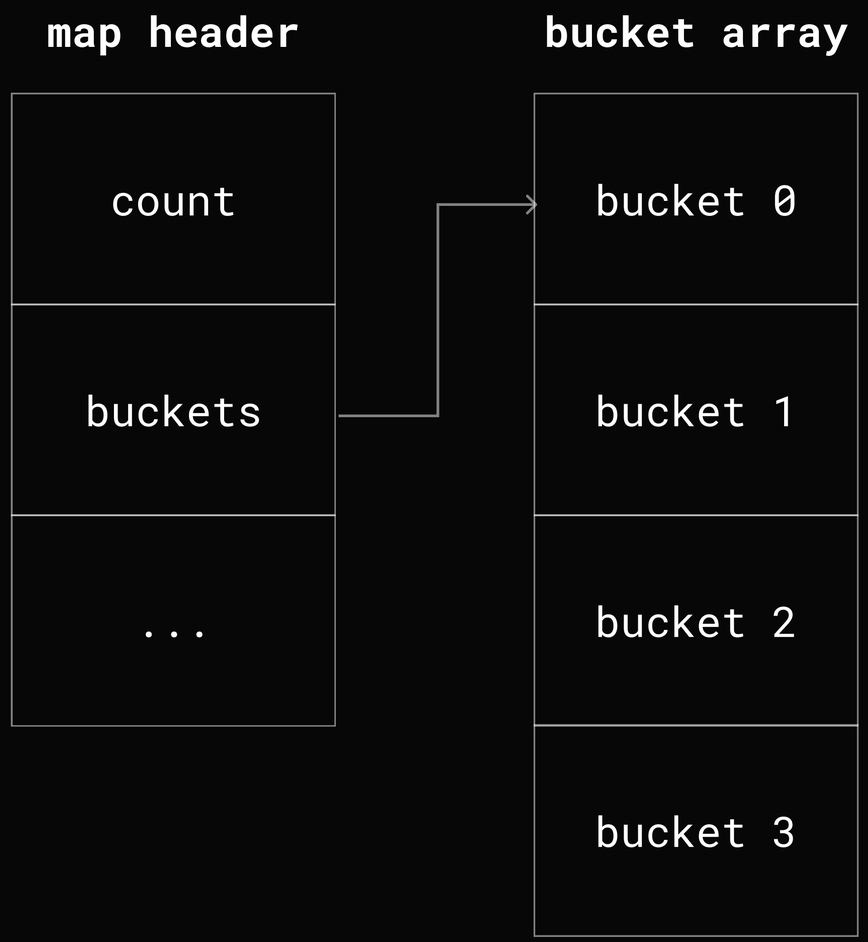


```
1     // Possible tophash values. We reserve a few possibilities for special marks.
2     // Each bucket (including its overflow buckets, if any) will have either all or none of its
3     // entries in the evacuated* states (except during the evacuate() method, which only happens
4     // during map writes and thus no one else can observe the map during that time).
5     emptyRest      = 0 // this cell is empty, and there are no more non-empty cells at higher indexes or overflows.
6     emptyOne       = 1 // this cell is empty
7     evacuatedX     = 2 // key/elem is valid.  Entry has been evacuated to first half of larger table.
8     evacuatedY     = 3 // same as above, but evacuated to second half of larger table.
9     evacuatedEmpty = 4 // cell is empty, bucket is evacuated.
10    minTopHash     = 5 // minimum tophash for a normal filled cell.
```

КАК ПРОИСХОДИТ РЕИНДЕКСАЦИЯ

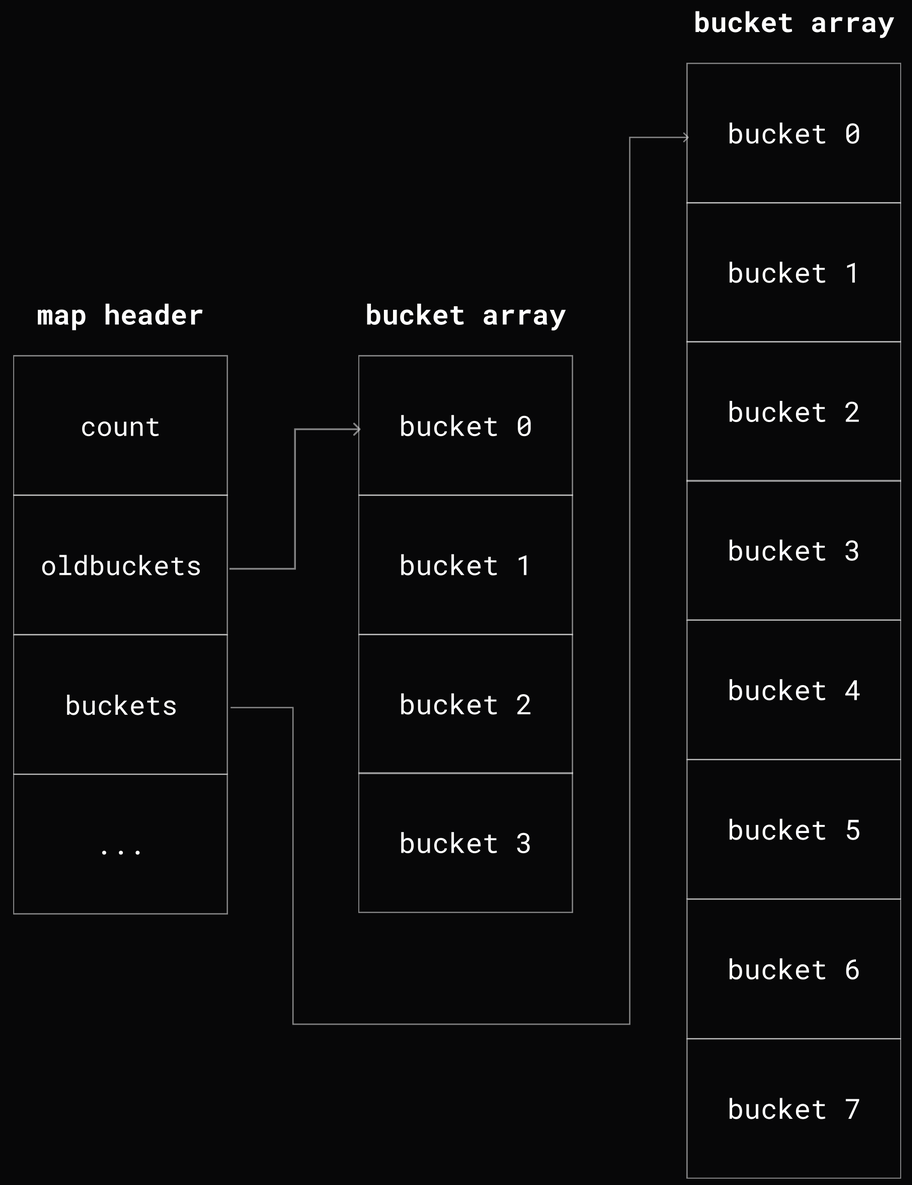
при увеличении массива бакетов?

Увеличение размеров массива бакетов происходит, когда коэффициент заполненности хэш-таблицы равен **80%**, что примерно равно **6.5 элементам** в каждом бакете *(массив увеличивается в два раза)*

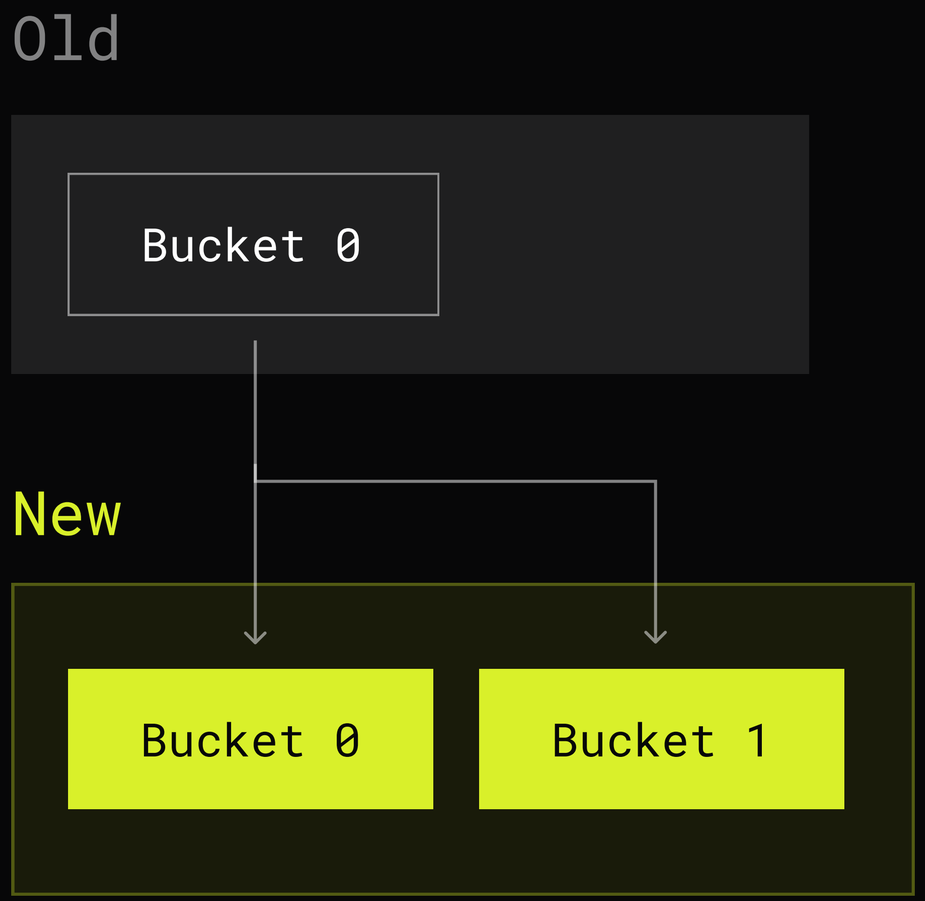


Эвакуация происходит инкрементально

и выполняется при записи и удалении ключей (за один вызов может перераспределиться несколько бакетов)



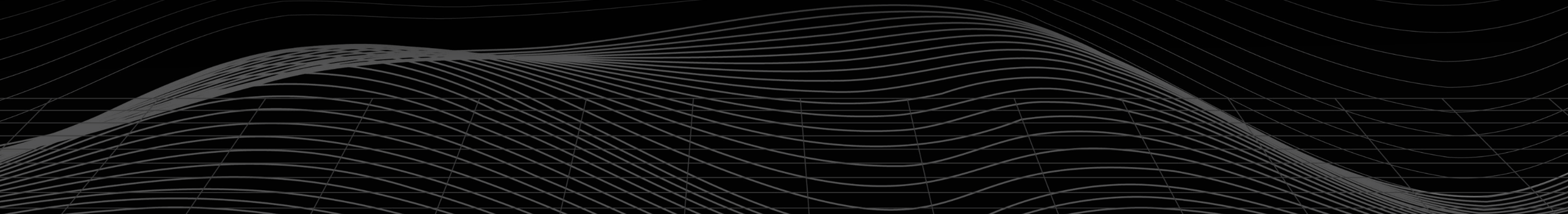
Map



Terminal: question + ⚙ –

КАКИЕ ПРЕИМУЩЕСТВА У ТАКОГО ПОДХОДА?

Не будет больших пауз во время переиндексации, но тем не менее — операции записи и удаления будут чуть медленнее в процессе эвакуации данных, а также при поиске тоже придется искать в двух местах



ПЕРЕРЫВ 5 МИНУТ

Terminal: question + ⚙ –

КАК ПРОИСХОДИТ РАБОТА СО СЛОВАРЕМ?

```go
// func makemap64(t *maptype, hint int64, h *hmap) *hmap
data := make(map[string]int, 5)

// func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
value := data["key"]

// func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool)
value, found := data["key"]

// func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
data["key"] = 100

// func mapdelete(t *maptype, h *hmap, key unsafe.Pointer)
delete(data, "key")
```

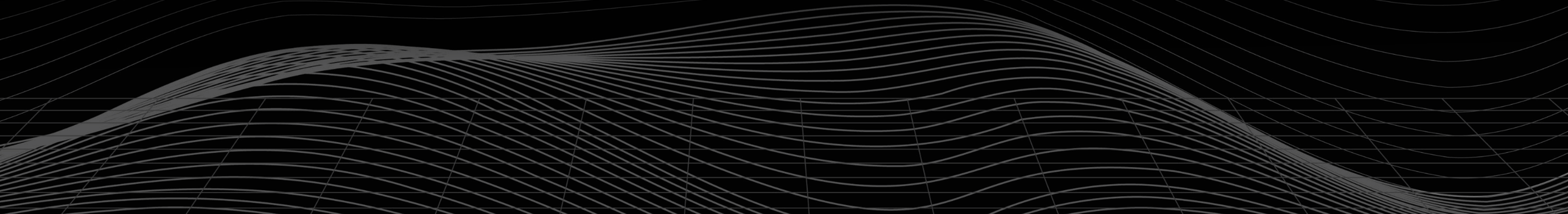
EXAMPLE

Mapaccess

Объекты **maptype** переиспользуются, например если вы создали три словаря **map[string]int**, то у вас только один объект **maptype** для этого типа

```
type maptype struct {
    typ    _type
    key    *_type
    elem   *_type
    bucket *_type // internal type representing a hash bucket
    // function for hashing keys (ptr to key, seed) -> hash
    hasher     func(unsafe.Pointer, uintptr) uintptr
    keysize    uint8  // size of key slot
    elemsize   uint8  // size of elem slot
    bucketsize uint16 // size of bucket
    flags      uint32
}
```

Hasher — это хэш-функция, которая принимает аргументы key, hash0 (seed) и возвращает hash.

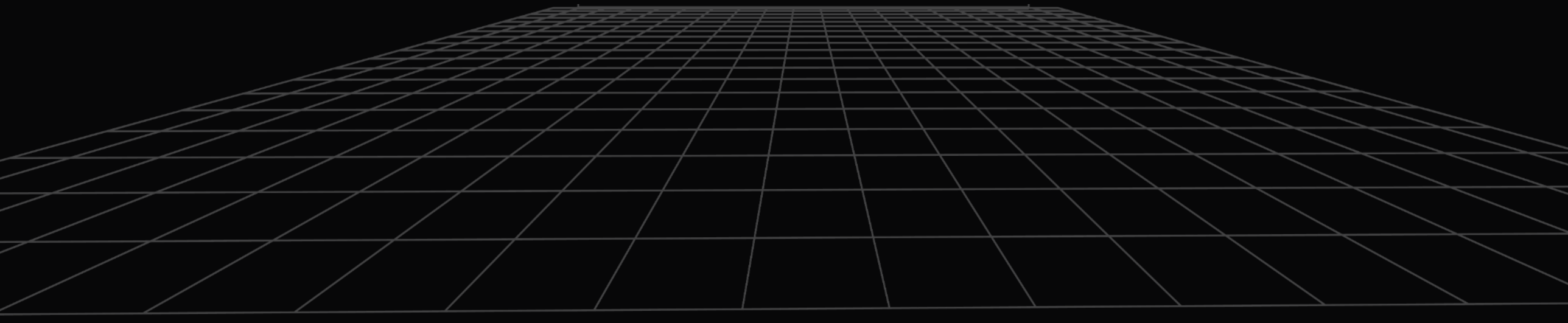


`_type` – дескриптор типа

```
type _type struct {
    size_       uintptr
    ptrdata     uintptr // size of memory prefix holding all pointers
    hash        uint32  // hash of type; avoids computation in hash tables
    tflag       tflag   // extra type information flags
    align       uint8   // alignment of variable with this type
    fieldalign  uint8   // alignment of struct field with this type
    kind        uint8   // enumeration for C
    // function for comparing objects of this type
    // (ptr to object A, ptr to object B) -> ==?
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    // GCData stores the GC type data for the garbage collector.
    // If the KindGCProg bit is set in kind, GCData is a GC program.
    // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
    gcata    *byte
    str       nameOff // string form
    ptrToThis typeOff // type for pointer to this type, may be zero
}
```

FAQ

Внутреннее устройство словаря в Go



ТОНКОСТИ СЛОВАРЕЙ В GO

EXAMPLE

Map operations

EXAMPLE

Map with float

СЛОВАРИ В GO

1. Не хранят данные, отсортированные по ключу
2. Не сохраняют порядок, в котором были добавлены данные
3. Тип ключа словаря должен быть сравниваемым
4. При итерации порядок не определен

EXAMPLE

Iteration order

fmt.Println() сортирует список ключей, поэтому порядок вывода элементов словаря на печать в этой ситуации определен

Terminal: question

ПОЧЕМУ ПРИ ИТЕРАЦИИ НЕ ОПРЕДЕЛЕН ПОРЯДОК?

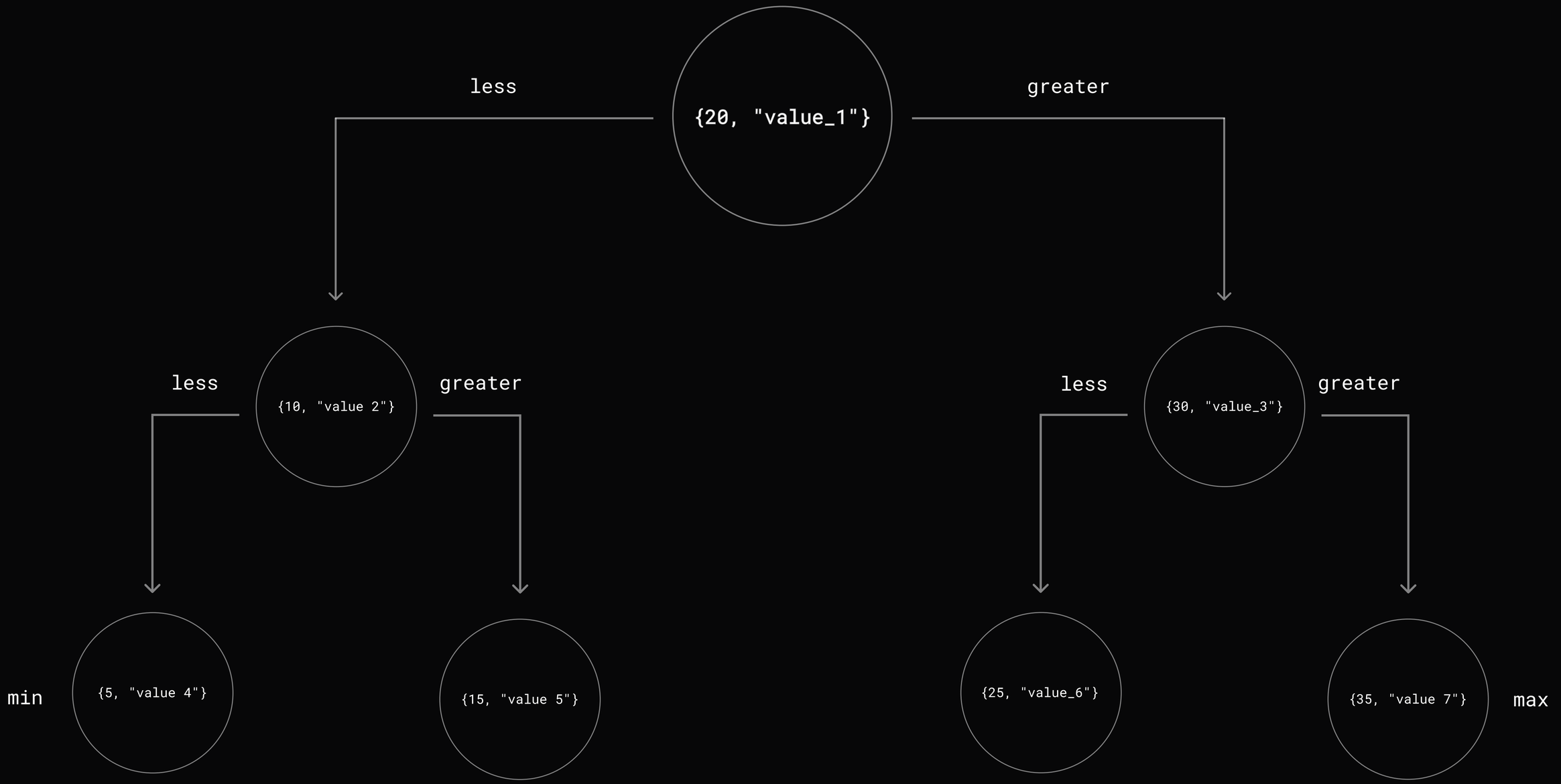
Разработчики **Go** решили добавить элемент случайности, чтобы программисты никогда не основывались на каких-либо предположениях об упорядочивании

EXAMPLE

Range by map

Terminal: question + ⚙ –

КАК МОЖНО РЕАЛИЗОВАТЬ УПОРЯДОЧЕННОЕ ХРАНЕНИЕ ДАННЫХ В СЛОВАРЕ?


less
{20, "value_1"}
greater
less
{10, "value 2"}
greater
less
{30, "value_3"}
greater
min
{5, "value 4"}
{15, "value 5"}
{25, "value_6"}
{35, "value 7"}
max

Screenshot

TreeMap v2

`TreeMap` is a generic key-sorted map using a red-black tree under the hood. It requires and relies on Go 1.18 generics feature. Iterators are designed after C++.

Usage

```go
package main

import (
	"fmt"

	"github.com/igrmk/treemap/v2"
)

func main() {
	tr := treemap.New[int, string]()
	tr.Set(1, "World")
	tr.Set(0, "Hello")
	for it := tr.Iterator(); it.Valid(); it.Next() {
		fmt.Println(it.Key(), it.Value())
	}
}
```

В **Go** разрешено обновление словаря
(вставка или удаление) во время итерации

EXAMPLE

Update during iteration

Если запись словаря создается во время итерации, она может быть произведена во время итерации или пропущена (выбор может варьироваться для каждой созданной записи и от одной итерации к другой)

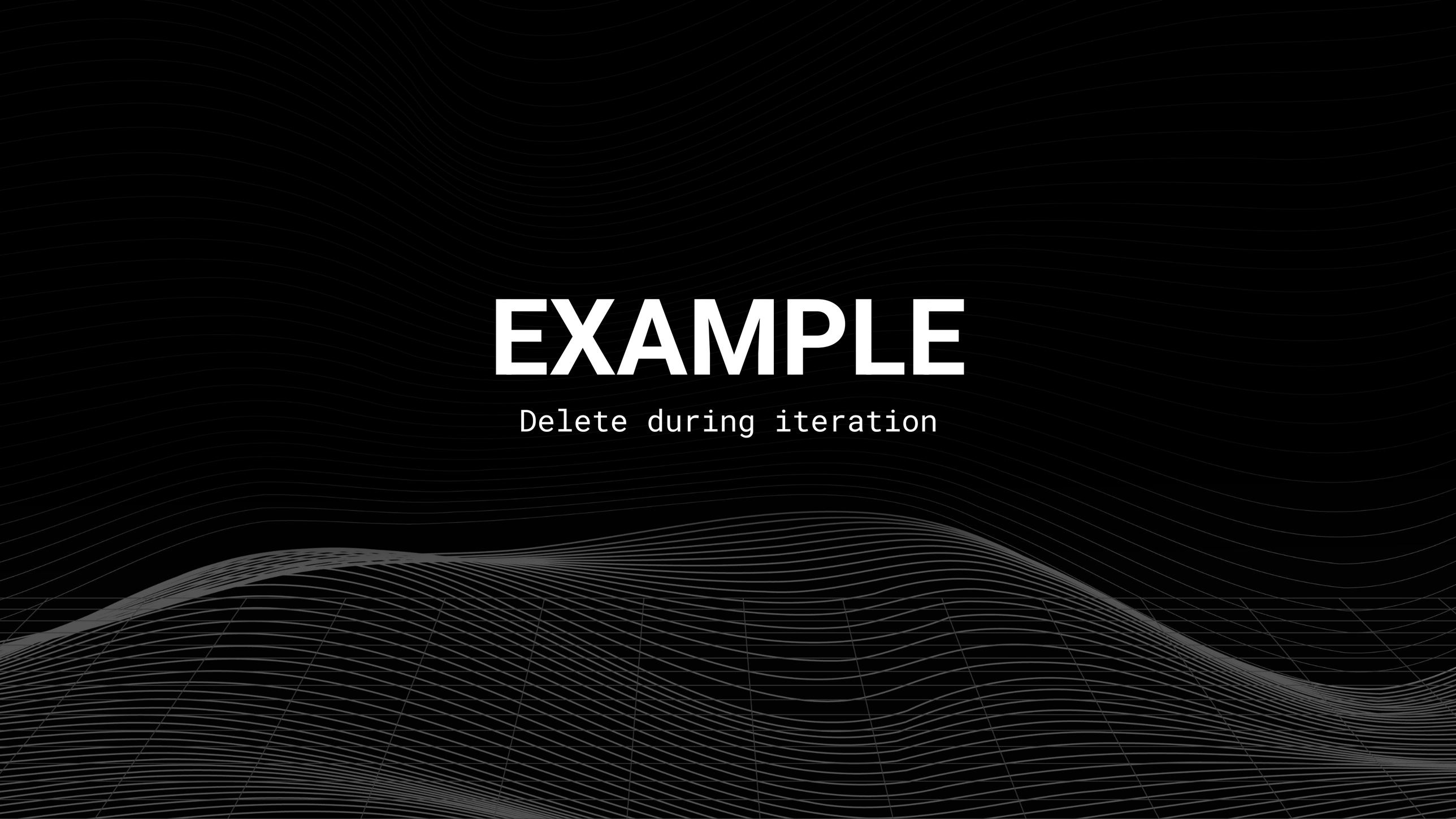
“Go maps are hash maps. Simplified, it means they are arrays under the hood. Keys are converted to indexes via a hashing (hence hash maps). Crucially, every map has a unique salt that is used by the hash function to make sure different maps will store elements in different order in the underlying array.

Iterating over a map is simply walking over the array. When you write the map during iteration, you insert new elements into the array. **This may happen either to the left of the current element, in which case it won't be visited by the iteration, or to the right, in which case it will be visited.**

Because of the unique salt, sometimes the inserted element will end up to the left, sometimes to the right, hence the non-deterministic result”

EXAMPLE

Delete during iteration



НЕЛЬЗЯ ПОЛУЧИТЬ АДРЕС ЗНАЧЕНИЯ ЭЛЕМЕНТА СЛОВАРЯ

EXAMPLE

Element value address

ПОЧЕМУ НЕЛЬЗЯ ПОЛУЧИТЬ АДРЕС ЗНАЧЕНИЯ ЭЛЕМЕНТА СЛОВАРЯ?

НЕЛЬЗЯ СРАВНИВАТЬ МЕЖДУ СОБОЙ С ИСПОЛЬЗОВАНИЕМ ОПЕРАТОРОВ != И ==

EXAMPLE

Comparison

КАК ОЧИСТИТЬ СЛОВАРЬ?

EXAMPLE

Map clearing

Емкость массива никогда не уменьшается!

EXAMPLE

Big map

Удаление из мапы не влияет на количество сегментов (даже количество сегментов из-за переполнений остается прежним). Решение - либо периодически перезатирать словать, либо использовать указатели в качестве значений


Пустой бакет
Пустой бакет
Пустой бакет
Пустой бакет

Если ключ или значение превышает 128 байт, Go не будет хранить их непосредственно в сегменте словаря (вместо этого хранится указатель — для ссылки на ключ или значение)

EXAMPLE

Map internals

Значение элемента словаря не может быть изменено, но может быть изменено (перезаписано) только целиком. Для большинства типов элементов это не является проблемой, но если тип элемента типа словаря является массивом или структурой - все становится немного нелогичным

EXAMPLE

Update value

EXAMPLE

Reservation

Terminal: deep_go

СОВЕТ

Резервируйте заранее емкость
словаря, если вам известно
его будущее количество элементов

EXAMPLE

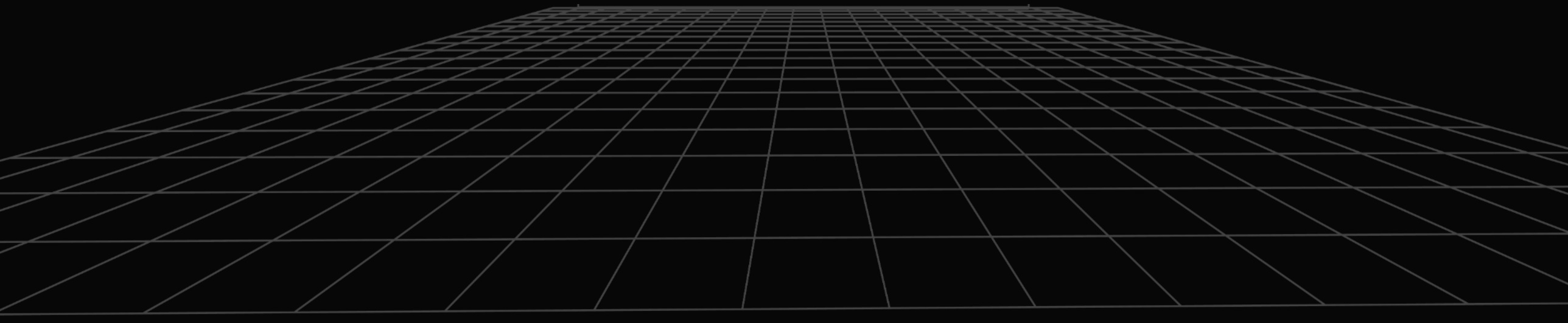
Map performance

EXAMPLE

Map empty assign

FAQ

Тонкости словарей в Go



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников