

Алгоритмы и структуры данных-2

Хеширование. Часть 2

Хеш-таблицы

Нестеров Р.А., PhD, старший преподаватель
Департамент программной инженерии

02

ЯНВАРЬ 2024

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

План

01

Закрытая адресация
в хеш-таблицах и
связные списки

02

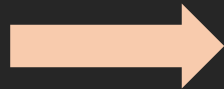
Открытая адресация
в хеш-таблицах и
пробирование

03

Идеальное (perfect)
хеширование без
коллизий

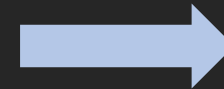
Объект

```
class Object {  
private:  
    std::string field;  
    int key;  
    bool flag;  
    ...  
public:  
    ...  
};
```



32-битное целое число





```
11010000  
10100001  
00000000  
00011100
```



Индекс

567

`std::hash<...>`

-  имеет специализации для фундаментальных типов `bool, char, int, long, float, double, ...`
-  возвращает значение типа `std::size_t` и не порождает исключений
-  обладает свойством **детерминированности**
-  вероятность коллизии должна стремиться к $1.0 / \text{std::numeric_limits}<\text{std::size_t}>::\text{max}()$



C++ HashingWithSTL.cpp

```
struct Person {
    std::string firstName;
    std::string lastName;
};

bool operator==(const Person& lhs, const Person& rhs) {
    return lhs.firstName == rhs.firstName &&
           lhs.lastName == rhs.lastName;
}

template<>
struct std::hash<Person> {
    std::size_t operator()(const Person& s) const noexcept {
        std::size_t h1 = std::hash<std::string>{}(s.firstName);
        std::size_t h2 = std::hash<std::string>{}(s.lastName);
        return h1 ^ (h2 << 1);
    }
};
```



C++ DictionaryADT.cpp

```
template<class T>
class HashTable {
public:
    void INSERT(T object);
    T SEARCH(T object);
    void DELETE(T object);
    void UPDATE(T object);

private:
    // внутренний контейнер
}
```

Во многих практических приложениях требуется поддержка операций
ADT Словарь

	INSERT	SEARCH	DELETE	UPDATE
Массив	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Связный список	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Сбалансированное дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Хеш-таблица	$O(1)^*$	$O(1)^*$	$O(1)^*$	$O(1)^*$

ЗАКРЫТАЯ АДРЕСАЦИЯ

Формируется список
объектов с одним и
тем же хешем

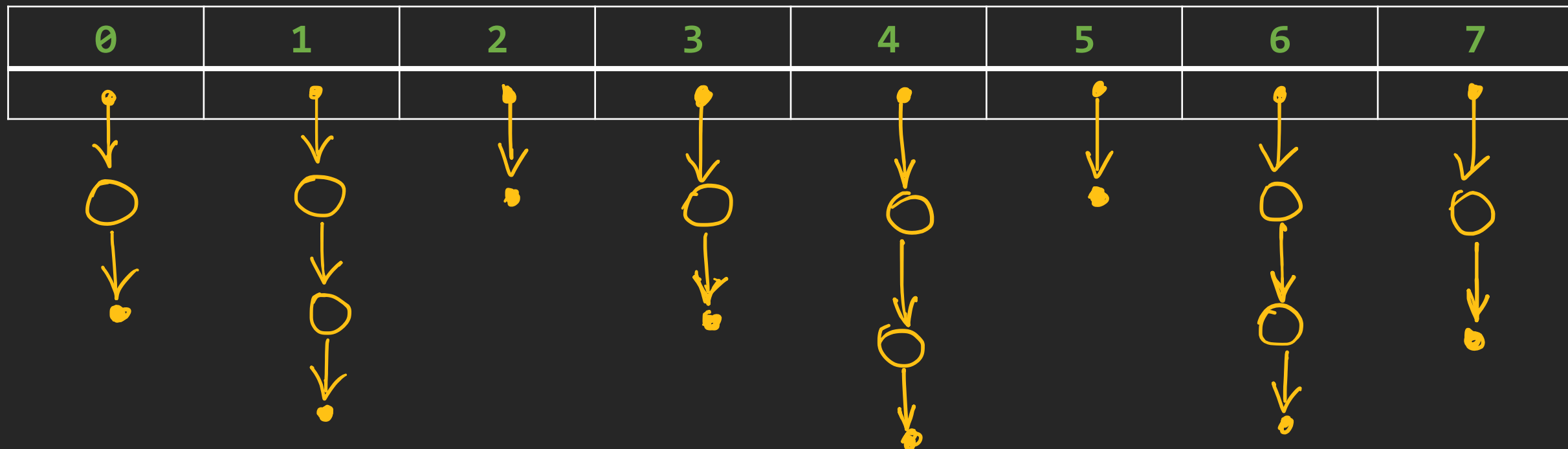
ОБРАБОТКА КОЛЛИЗИЙ

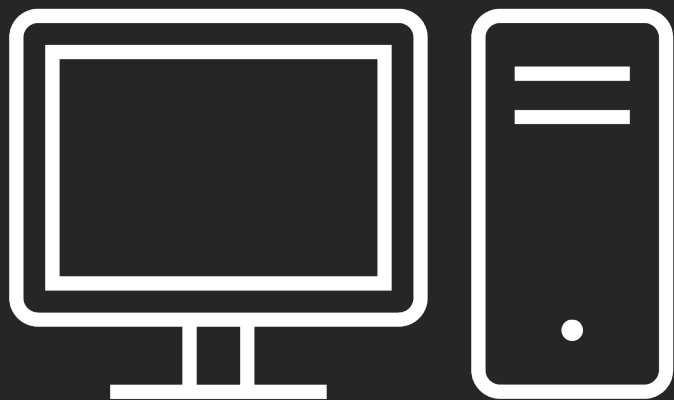
ОТКРЫТАЯ АДРЕСАЦИЯ

Смещение для
поиска свободных
ячеек в хеш-таблице

Метод цепочек – Separate Chaining

При таком способе обработки коллизий хеш-таблица представляет собой **массив связанных списков**





<http://129.97.10.179/>

<http://batman.hse.ru/>

Хешируем информацию о компьютерах в локальной сети **по их доменному имени**

В качестве хеш-кода берем **последние три бита первого символа** доменного имени



C++ HashingDomainNames.cpp

```
size_t hash(const string &str) {  
    if (str.length() == 0) {  
        return 0;  
    }  
  
    return str[0] & 7;  
}
```

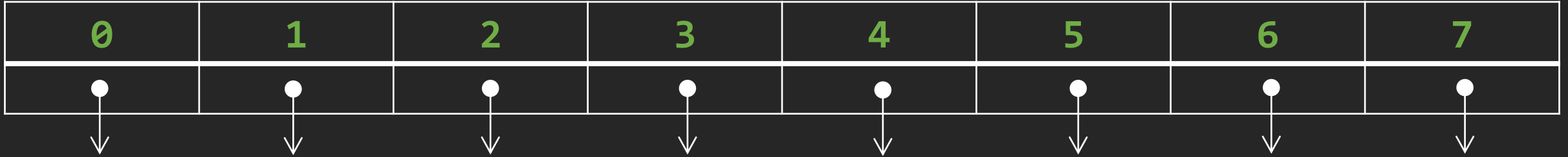
a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

parrot
129.97.94.45

hash_M →



$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

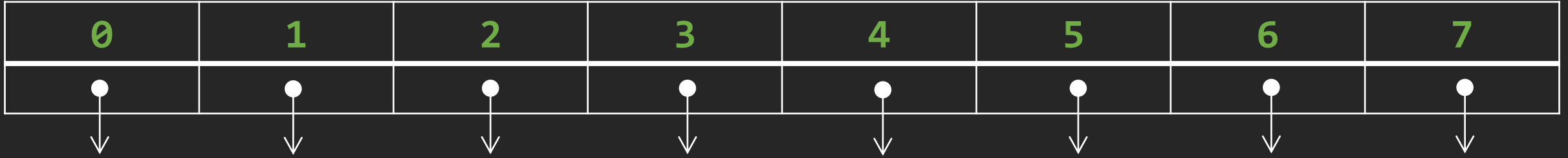


parrot
129.97.94.45

hash_M →

0

$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

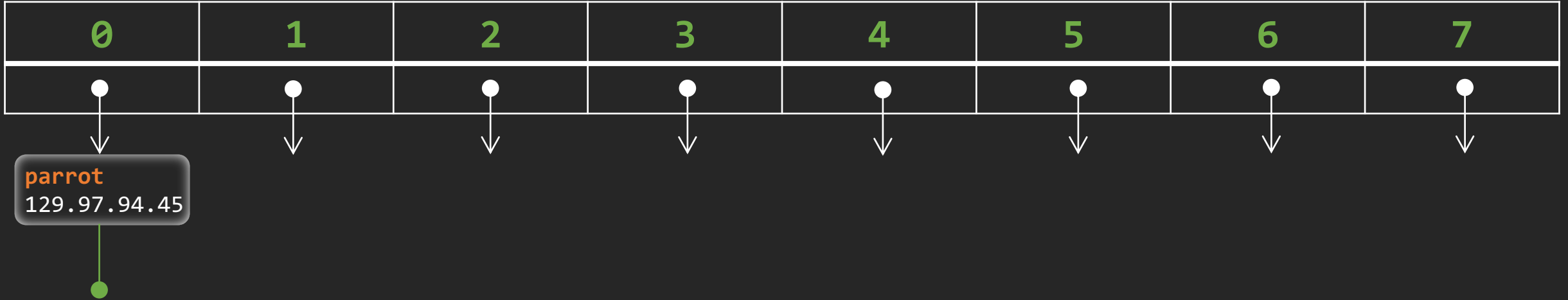


hamster
129.97.94.53

hash_M →



$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

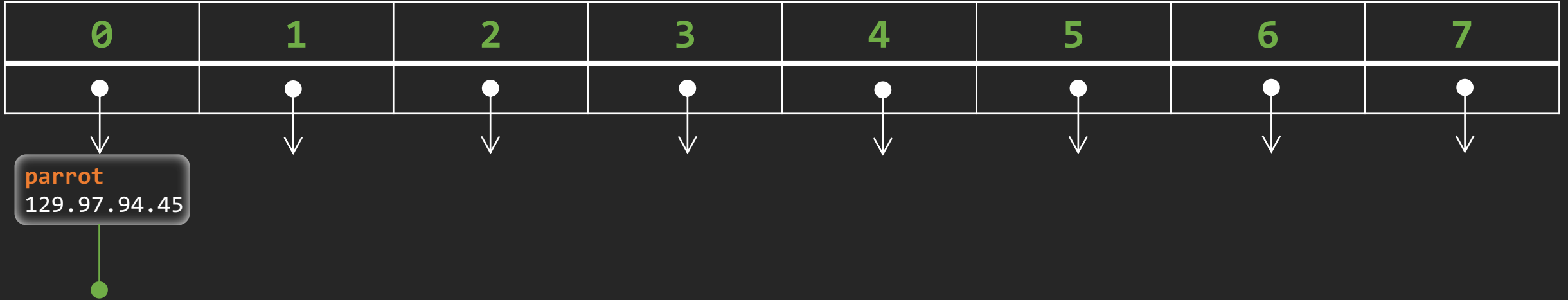


hamster
129.97.94.53

hash_M →

0

$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$



antelope
129.97.93.50

hash_M →



$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

0	1	2	3	4	5	6	7
●	●	●	●	●	●	●	●

hamster
129.97.94.53

parrot
129.97.94.45

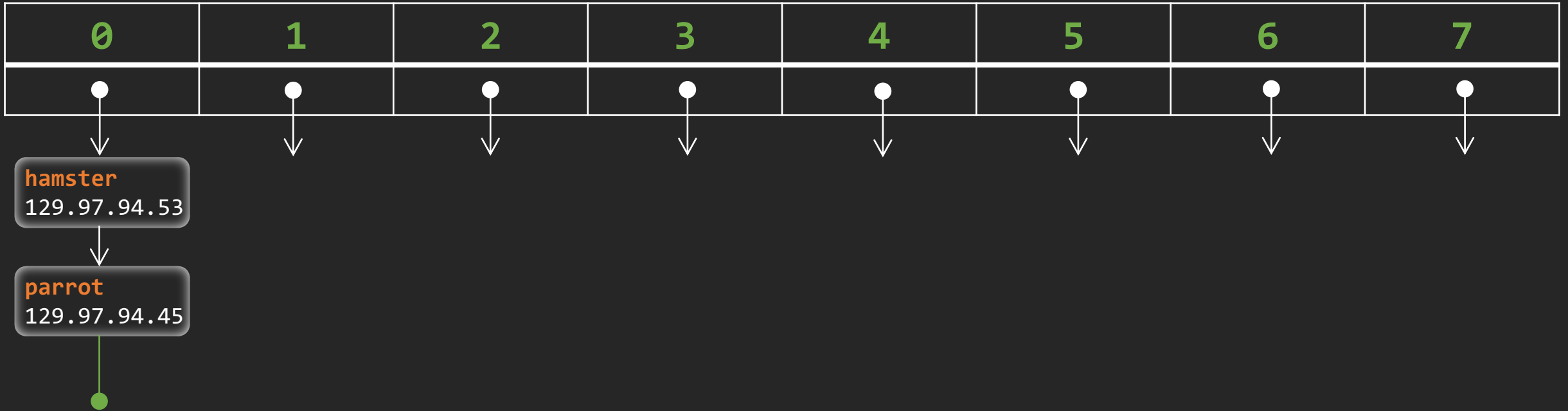


antelope
129.97.93.50

hash_M →

1

$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

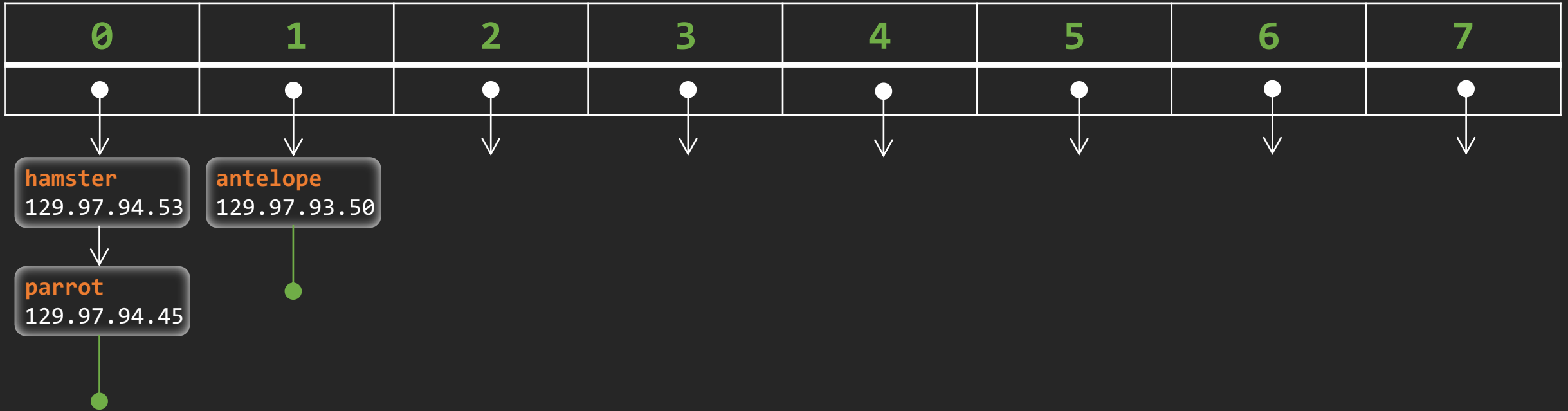


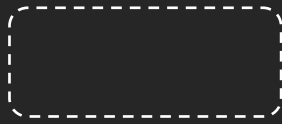
frog
129.97.93.35

hash_M →



$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$

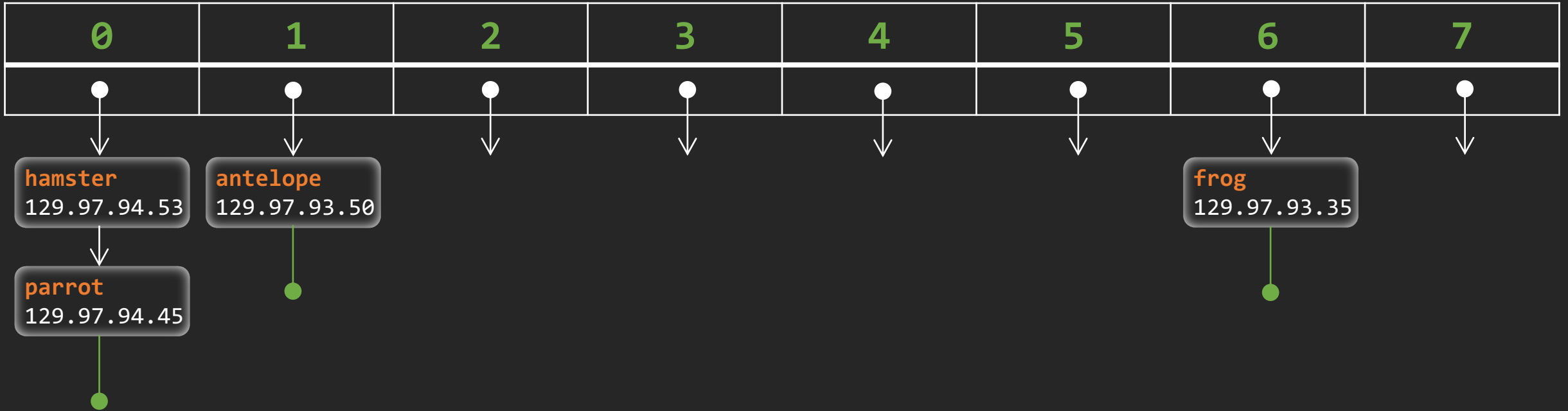


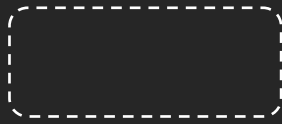


hash_M →



$$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \ \& \ 7$$

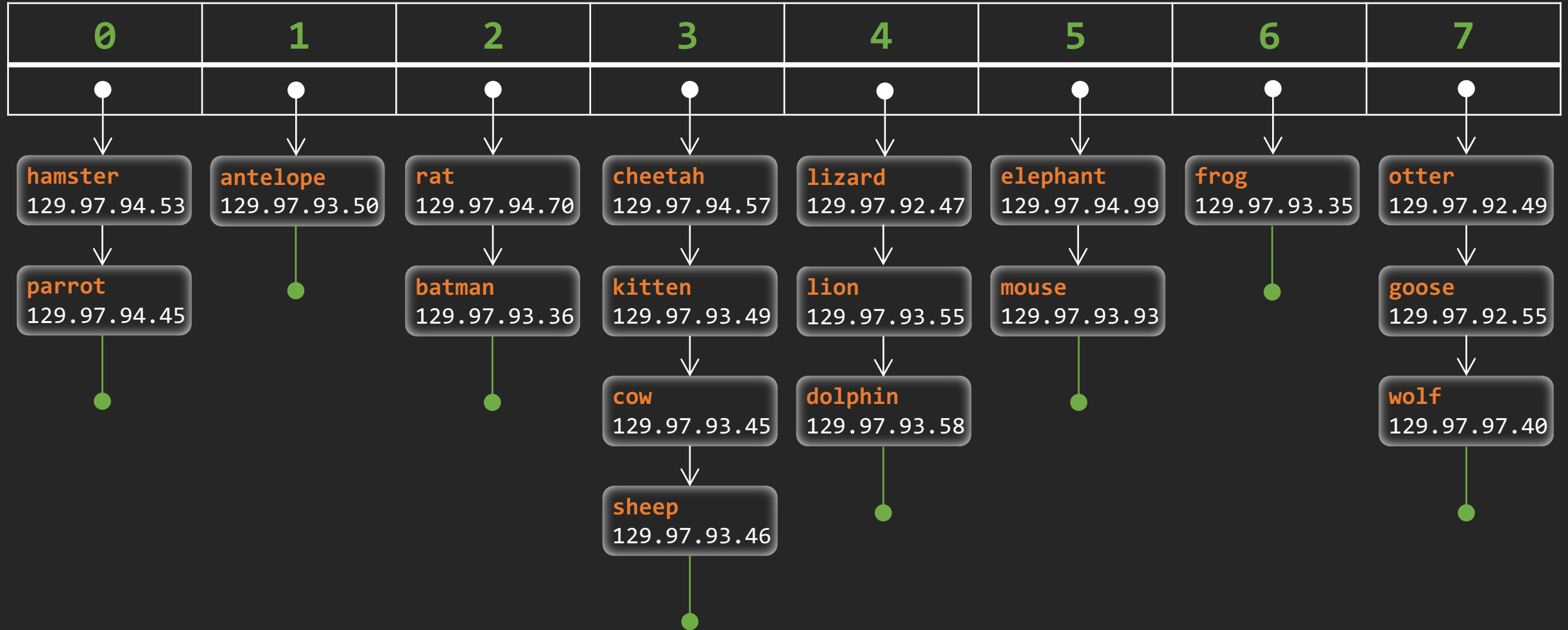




hash_M →



$$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \& 7$$



Load Factor

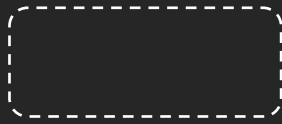
Для описания текущего состояния хеш-таблицы вычисляется **коэффициент заполненности**

$$\lambda = \frac{n}{M}$$

В нашем примере с хешированием доменов:

$$\lambda = \frac{18}{8} = 2.25$$

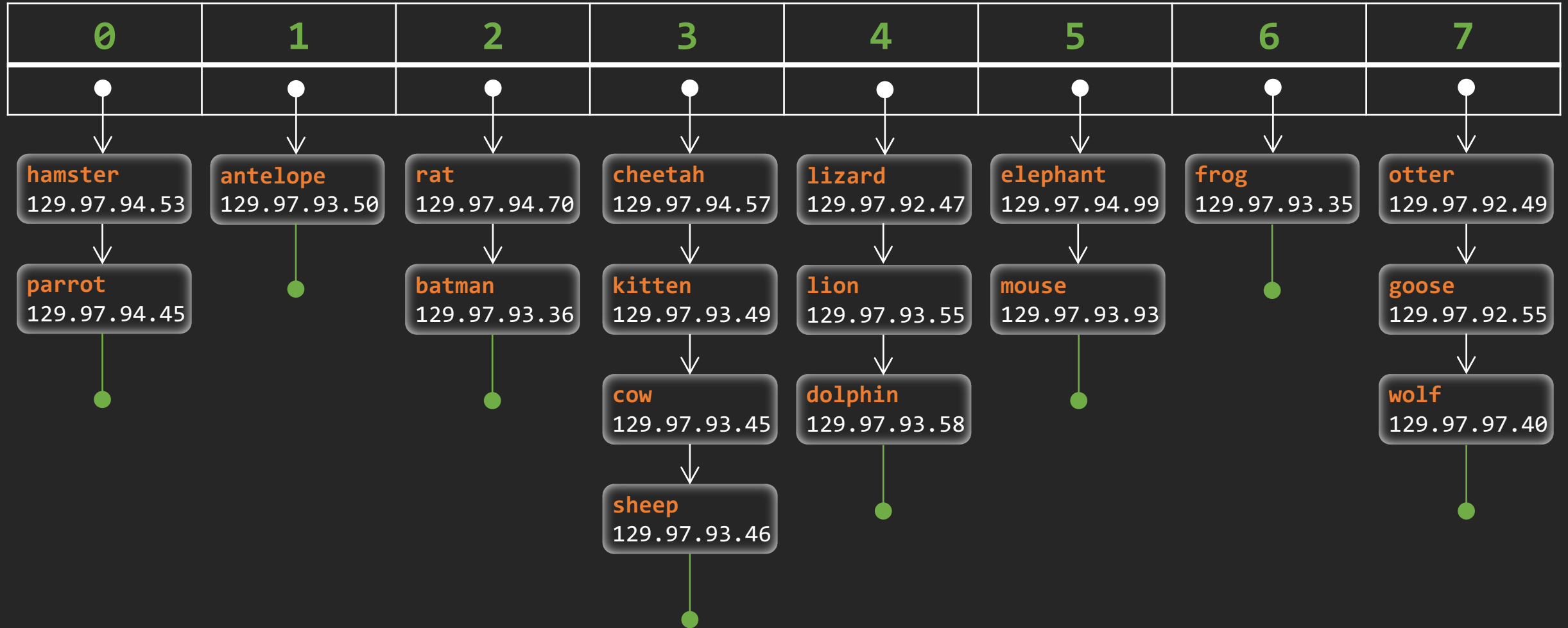
Большое значение коэффициента заполненности существенно влияет на сложность основных операций – $O(\lambda)$

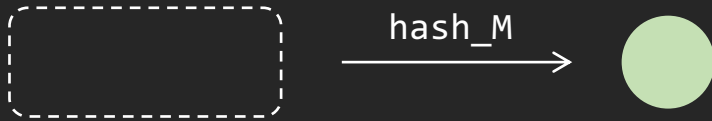


hash_M →

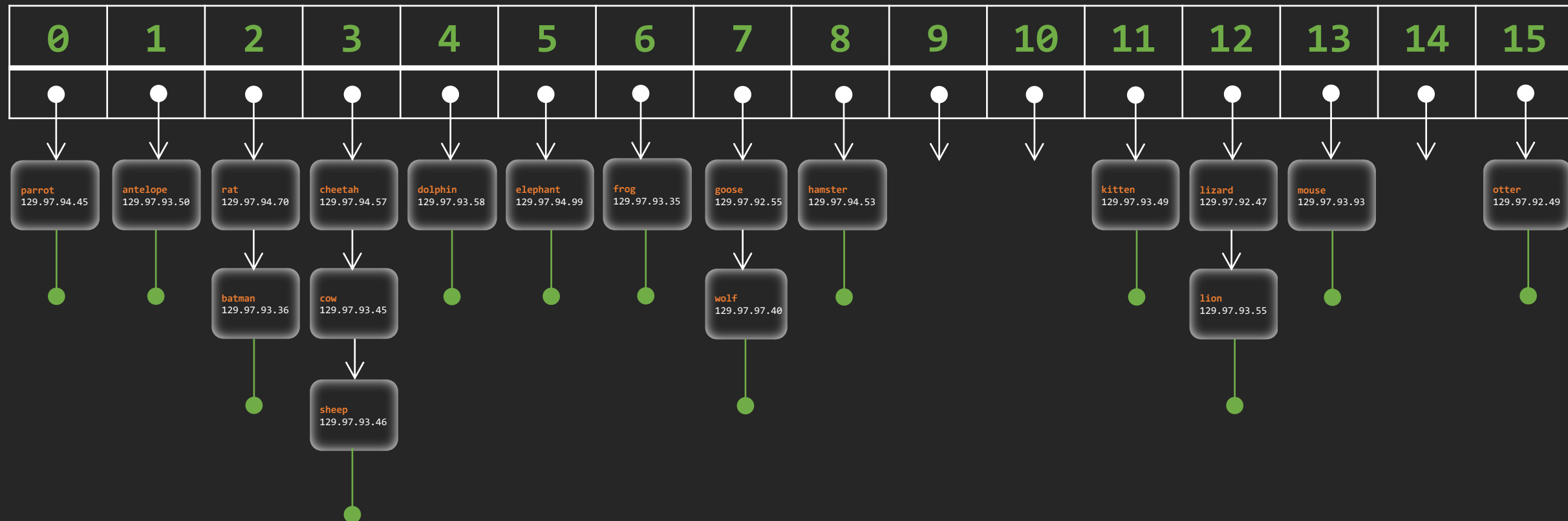


$$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \& 7$$





$$\text{hash_M}(\text{domainName}) = \text{domainName}[0] \& 15$$



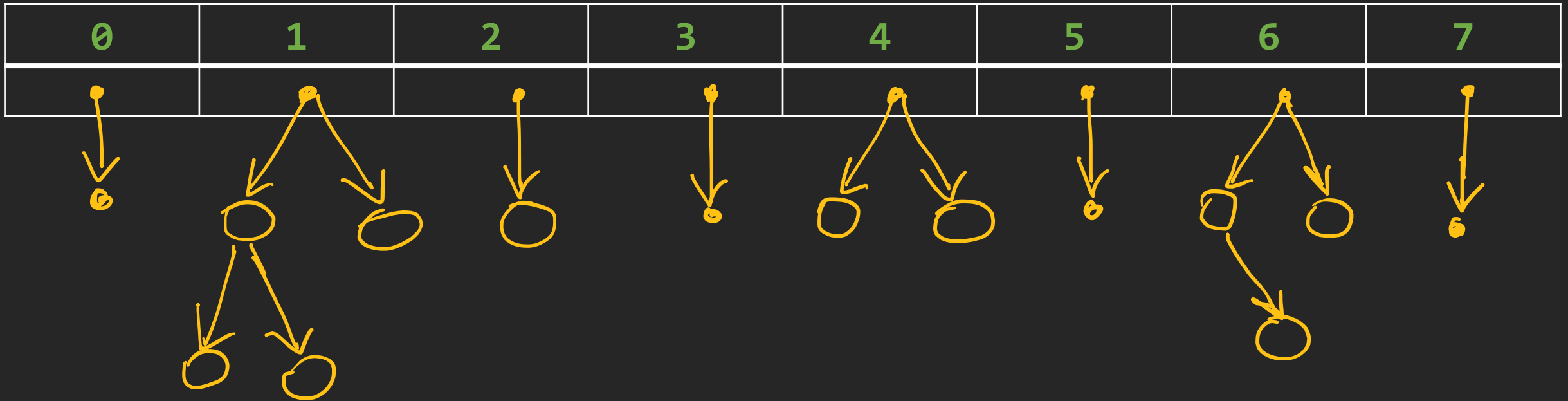
Перехеширование

Если коэффициент заполненности увеличился до некоторого порога (например, $\lambda = 0.5$), то следует выполнить перехеширование

Нам необходимо увеличить размер M хеш-таблицы (обычно, $2 \cdot M$), а также соответствующим образом изменить хеш-функцию

Метод цепочек – Separate Chaining

Почему бы вместо списков не использовать
сбалансированные деревья поиска?



Слабости метода цепочек



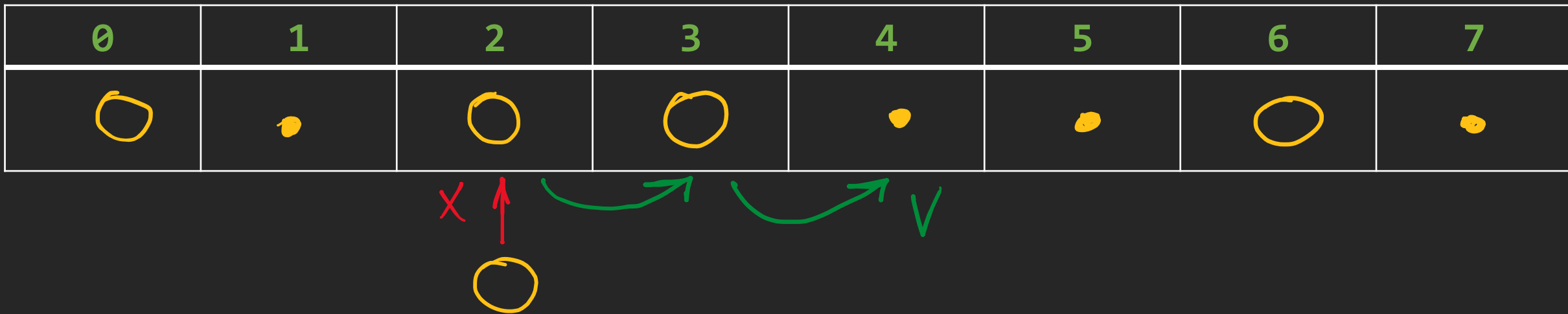
Использование дополнительной памяти для организации линейных одно(дву)связных списков



Основные операции ADT Словарь деградируют до операций на связных списках

Открытая адресация

При таком методе обработки коллизий объекты сохраняются **напрямую в хеш-таблицу**



19A

207

3AD

488

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Занято ячеек	0
Коэффициент заполненности	0
Общее число проб	0
Среднее число проб	0



C++ LinearProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

5BA

680

74C

826

0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

Занято ячеек	4
Коэффициент заполненности	0.25
Общее число проб	4
Среднее число проб	1



C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

680

826

5BA

74C

0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

Занято ячеек	4
Коэффициент заполненности	0.25
Общее число проб	4
Среднее число проб	1

C++ LinearProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

680

826

5BA

74C

0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

Занято ячеек	4
Коэффициент заполненности	0.25
Общее число проб	4
Среднее число проб	1

C++ LinearProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

946

ACD

B32

1	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488		19A	5BA	74C	3AD		

Занято ячеек	8
Коэффициент заполненности	0.5
Общее число проб	9
Среднее число проб	1.125



C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```


1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	

Занято ячеек	11
Коэффициент заполненности	0.6875
Общее число проб	16
Среднее число проб	1.36



C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

C8B

1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	

Занято ячеек	11
Коэффициент заполненности	0.6875
Общее число проб	16
Среднее число проб	1.36



C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

D59

1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

Занято ячеек	12
Коэффициент заполненности	0.75
Общее число проб	21
Среднее число проб	1.75



C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

Занято ячеек	13
Коэффициент заполненности	0.8125
Общее число проб	30
Среднее число проб	2.31

C++ LinearProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);


for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```


1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B


 Для выполнения операции **SEARCH(object)** нам также потребуется просматривать следующие ячейки

 Возможные результаты: объект найден, найдена пустая ячейка или просмотрен весь массив при $\lambda = 1$

SEARCH(C8B)


1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B


 Для выполнения операции **SEARCH(object)** нам также потребуется просматривать следующие ячейки


 Возможные результаты: объект найден, найдена пустая ячейка или просмотрен весь массив при $\lambda = 1$

SEARCH(C8B)

1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

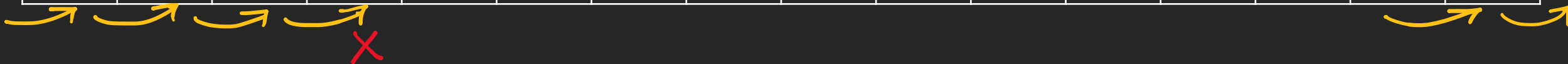



 Для выполнения операции **SEARCH(object)** нам также потребуется просматривать следующие ячейки


 Возможные результаты: объект найден, найдена пустая ячейка или просмотрен весь массив при $\lambda = 1$

SEARCH(23E)

1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B



 Для выполнения операции **SEARCH(object)** нам также потребуется просматривать следующие ячейки

 Возможные результаты: объект найден, найдена пустая ячейка или просмотрен весь массив при $\lambda = 1$

DELETE(3AD)

1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C		ACD	C8B

? Можем ли мы физически удалить требуемый элемент из ячейки хеш-таблицы с линейным пробированием?

⊘ Нет, так как мы потеряем возможность найти некоторые элементы в «связанных» ячейках хеш-таблицы

DELETE(3AD)

1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	ACD	C8B	D59

? Можем ли мы физически удалить требуемый элемент из ячейки хеш-таблицы с линейным пробированием?

⊘ Нет, так как мы потеряем возможность найти некоторые элементы в «связанных» ячейках хеш-таблицы

Линейное пробирование – Анализ

Использование линейного сдвига для поиска свободных ячеек хеш-таблицы приводит к образованию **первичных кластеров**



Размеры получаемых кластеров **вливают на оценки сложности** всех ключевых операций ADT Словарь

Линейное пробирование – Анализ

Для известного коэффициента заполненности хеш-таблицы λ можно рассчитать **среднее количество проб** в случае **успешного** поиска значения:

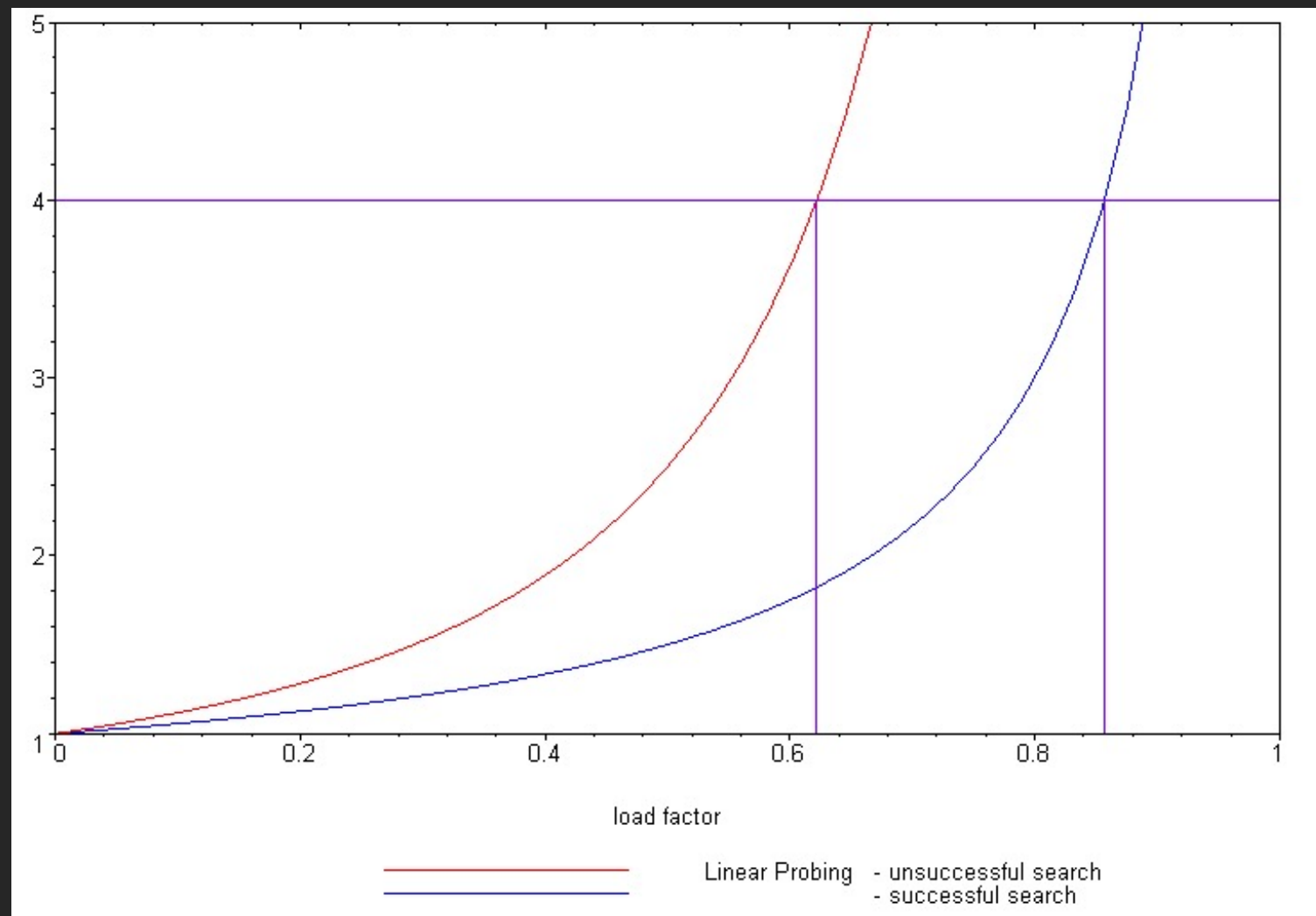
$$\frac{1}{2} \left(1 + \frac{1}{1 - \lambda} \right)$$

Линейное пробирование – Анализ

Для известного коэффициента заполненности хеш-таблицы λ можно рассчитать **среднее количество проб** в случае **неуспешного** поиска или вставки значения:

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2} \right)$$

Линейное пробирование – Анализ



Линейное пробирование – Анализ



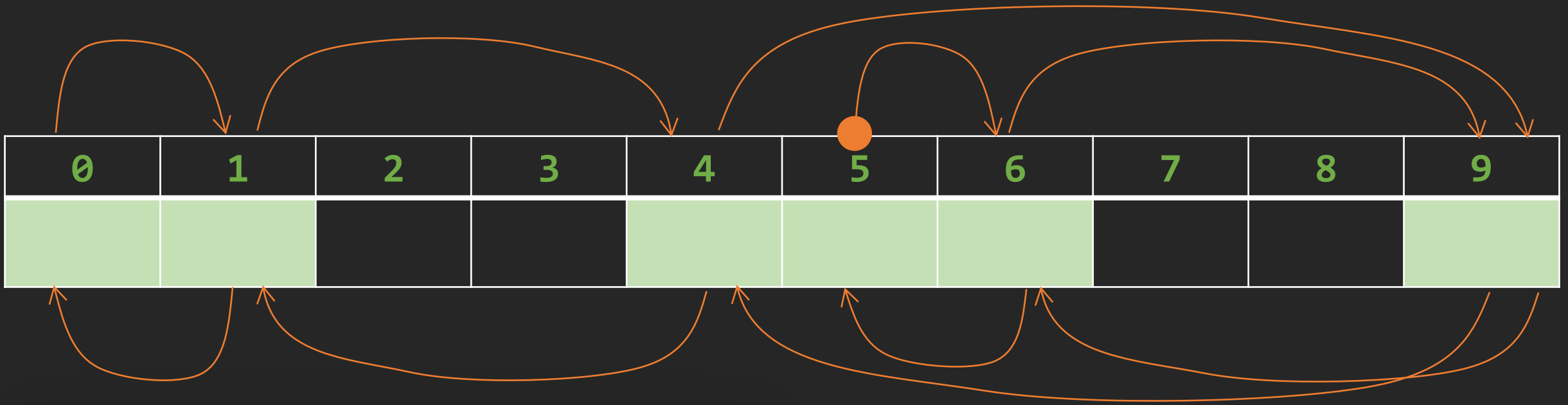
Использовать хеш-таблицу **очень большого размера**, чтобы никогда не достичь предела заполненности



Удерживать заполненность в некоторых пределах с помощью **перехеширования**



Использовать **другие стратегии вычисления сдвига** – квадратичное и двойное пробирование



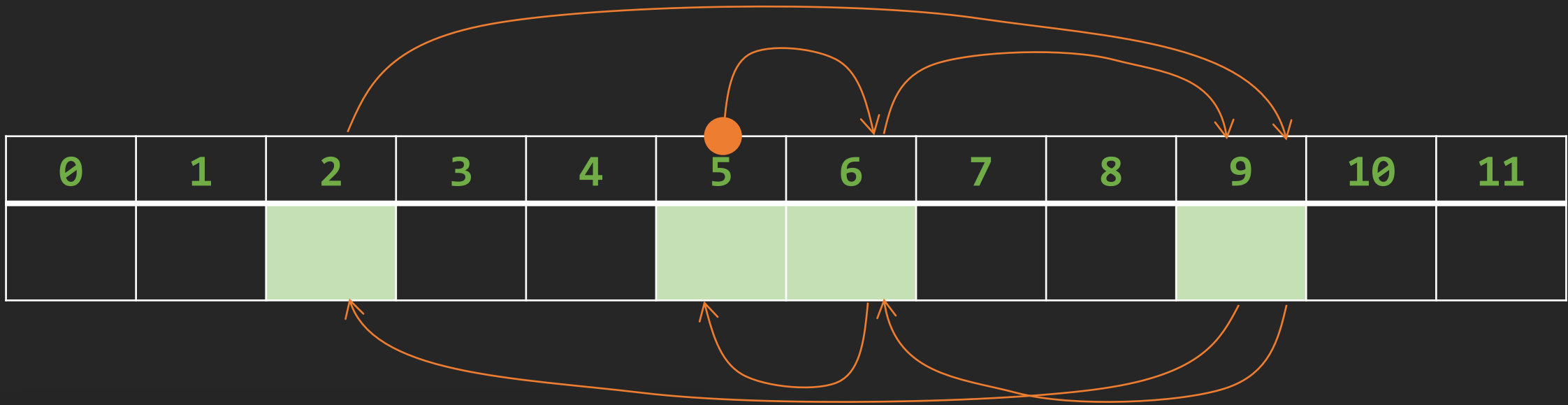
```
C++ QuadraticProbing.cpp

size_t initial = hash_M(obj.hash(), M);

for (int k = 0; k < M; ++k) {
    size_t bin = (initial + k * k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

В таком подходе к вычислению
индекса следующей свободной
ячейки есть **существенная
проблема!**



C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for (int k = 0; k < M; ++k) {
    size_t bin = (initial + k * k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

В таком подходе к вычислению
индекса следующей свободной
ячейки есть **существенная
проблема!**

Пусть M будет **простым числом** $p...$

- ✓ Гарантируется, что простое квадратичное пробирование «посетит» $\lceil p/2 \rceil$ **ячеек хеш-таблицы**
- ✗ Нельзя использовать побитовые операции – все придется делать с помощью **%...**
- ✗ Как вычислить **следующее простое число** после $2 \cdot M$ для перехеширования?..

Квадратичное пробирование обобщенно



C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + c1 * k + c2 * k * k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

Вернемся к $M = 2^m$...

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + c1 * k + c2 * k * k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

Положим

$$c_1 = c_2 = \frac{1}{2}$$

Вернемся к $M = 2^m$...



C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + (k + k * k) / 2) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

Положим

$$c_1 = c_2 = \frac{1}{2}$$

Вспомним, что

$$\sum_{i=0}^k i = \frac{k(k+1)}{2}$$

Вернемся к $M = 2^m$...



C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

Положим

$$c_1 = c_2 = \frac{1}{2}$$

Вспомним, что

$$\sum_{i=0}^k i = \frac{k(k+1)}{2}$$



C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (initial + (k + k * k) / 2) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

Все ячейки хеш-таблицы были
посещены при таком подходе
к **квадратичному пробирванию**
в таблице размера $M = 16!!!$

9A

07

AD

88

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Занято ячеек	0
Коэффициент заполненности	0
Общее число проб	0
Среднее число проб	0

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```


BA

80

4C

26

0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A			AD		

Занято ячеек	4
Коэффициент заполненности	0.25
Общее число проб	4
Среднее число проб	1

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

1	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

Занято ячеек	8
Коэффициент заполненности	0.5
Общее число проб	9
Среднее число проб	1.125

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

46

1	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

Занято ячеек	8
Коэффициент заполненности	0.5
Общее число проб	9
Среднее число проб	1.125

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

46

1	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

Занято ячеек	8
Коэффициент заполненности	0.5
Общее число проб	9
Среднее число проб	1.125



C++ QuadraticProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

46

1	0	0	0	0	0	1	1	1	0	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

Занято ячеек	8
Коэффициент заполненности	0.5
Общее число проб	9
Среднее число проб	1.125

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		

Занято ячеек	9
Коэффициент заполненности	0.5
Общее число проб	11
Среднее число проб	1.22

C++ QuadraticProbing.cpp

```
size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;
```

1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		C9

Занято ячеек	10
Коэффициент заполненности	0.625
Общее число проб	15
Среднее число проб	1.5

C++ QuadraticProbing.cpp

```

size_t initial = hash_M(obj.hash(), M);

for ( int k = 0; k < M; ++k ) {
    bin = (bin + k) % M;
    if (!occupied[bin]) break;
}

table[bin] = obj;

```

DELETE(4C)

1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		C9

? Можем ли мы воспользоваться таким же способом удаления, как и в случае с линейным пробированием?

⊘ Нет, так как через этот элемент может проходить несколько «связанных» последовательностей ключей

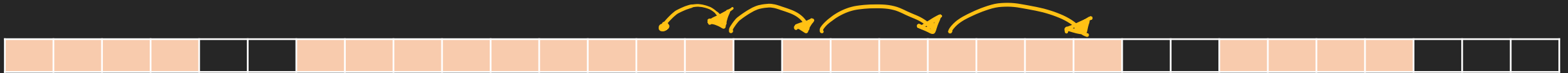
DELETE(4C)

1	0	0	0	0	0	1	1	1	1	1	1	2	1	0	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		C9

✓ Воспользуемся механизмом **ленивого удаления**
и пометим данную ячейку новым статусом (ERASED)

Квадратичное пробирование – Анализ

Использование квадратичного сдвига для поиска свободных ячеек хеш-таблицы приводит к образованию **вторичных кластеров**



Объекты, которые помещаются в одну и ту же ячейку, будут следовать по одной и той же последовательности проб

Квадратичное пробирование – Анализ

Для известного коэффициента заполненности хеш-таблицы λ можно рассчитать **среднее количество проб** в случае **успешного** поиска значения:

$$\frac{\ln\left(\frac{1}{1-\lambda}\right)}{\lambda}$$

Квадратичное пробирование – Анализ


Для известного коэффициента заполненности хеш-таблицы λ можно рассчитать **среднее количество проб** в случае **неуспешного** поиска значения и вставки:

$$\frac{1}{1 - \lambda}$$


Квадратичное vs линейное пробирование

Пусть коэффициент заполненности $\lambda = 2/3$.

	Среднее количество проб	
	Успешный поиск	Вставка и неуспешный поиск
Линейное пробирование	3	5
Квадратичное пробирование	1.65	3



При **двойном хешировании** сдвиг
вычисляется с помощью
второй хеш-функции



Идеальное хеширование – Пример

Шаг 1 Случайным образом выбрать **универсальную** хеш-функцию
 $hash_M(key) = ((64 \cdot key + 5) \bmod 87) \bmod 8.$

0	• →
1	• →
2	• →
3	• →
4	• →
5	• →
6	• →
7	• →

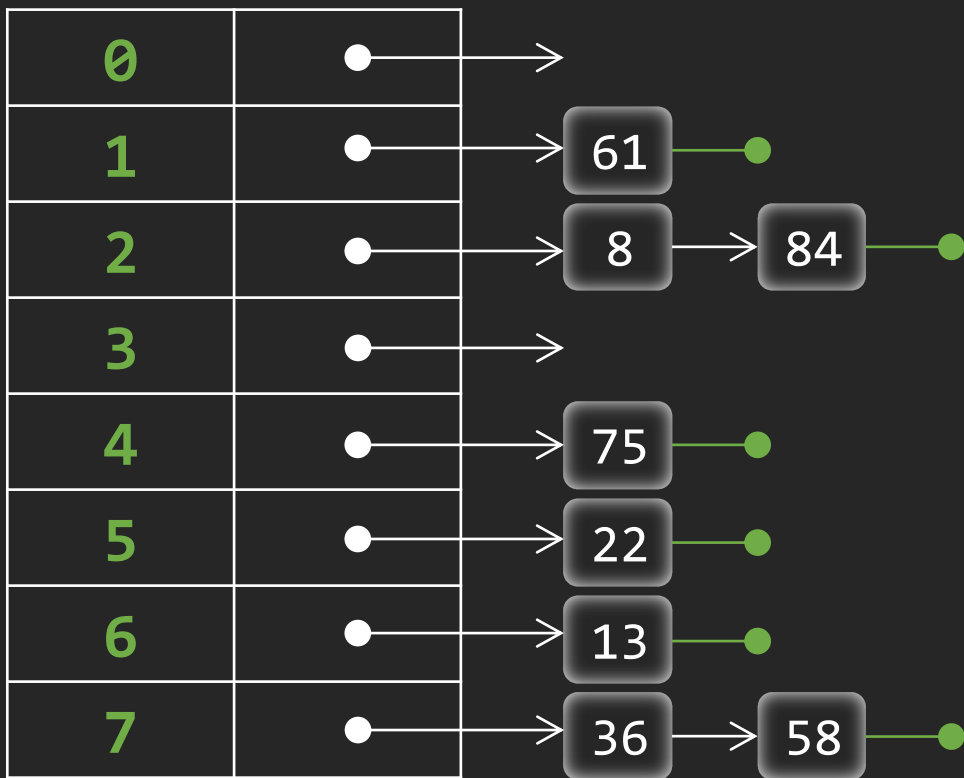
Идеальное хеширование – Пример

Шаг 2 Выполнить заполнение хеш-таблицы с применением метода цепочек $U = \{8, 22, 36, 75, 61, 13, 84, 58\}$.

0	• →
1	• →
2	• →
3	• →
4	• →
5	• →
6	• →
7	• →

Идеальное хеширование – Пример

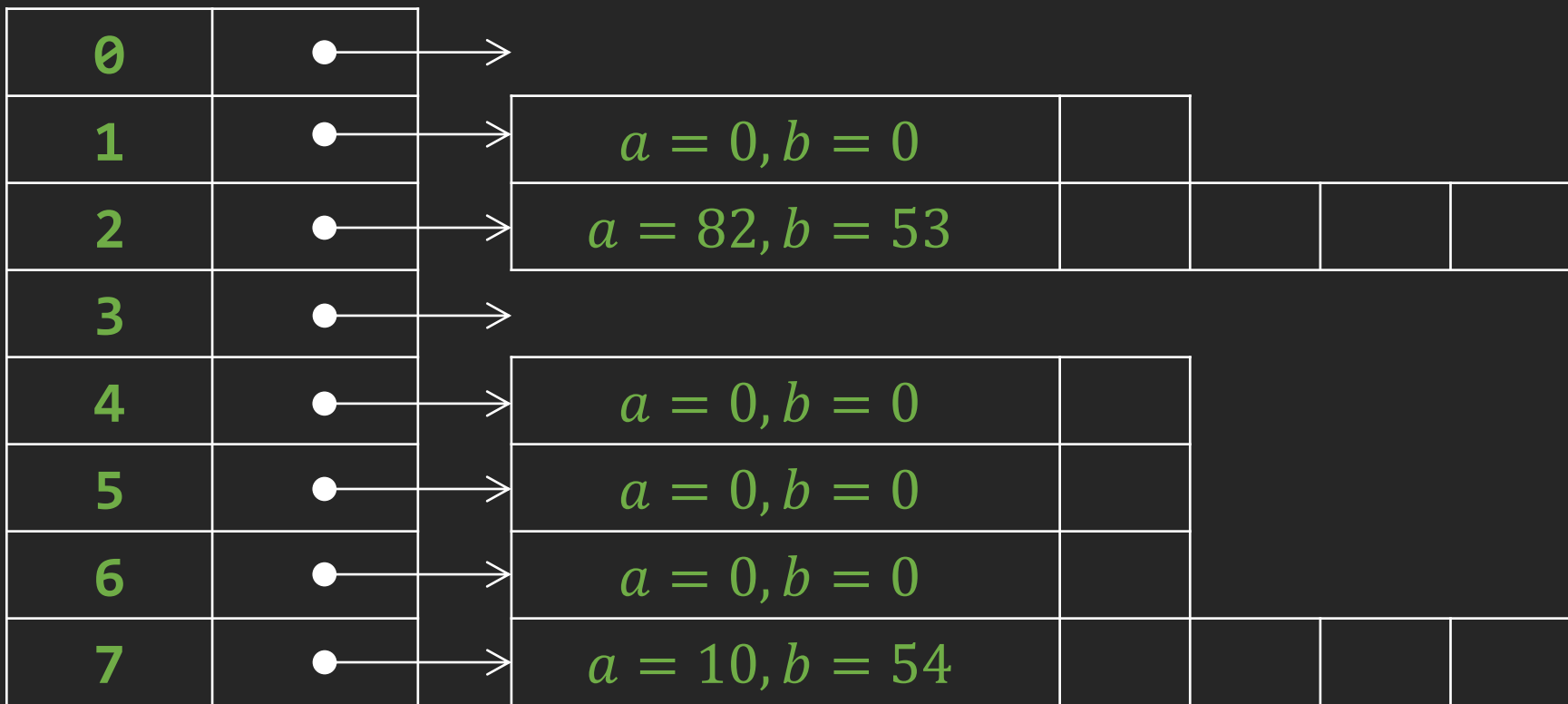
Шаг 2 Выполнить заполнение хеш-таблицы с применением метода цепочек $U = \{8, 22, 36, 75, 61, 13, 84, 58\}$.



Использование универсальной хеш-функции обеспечивает вероятность коллизии $\frac{1}{M}$.

Идеальное хеширование – Пример

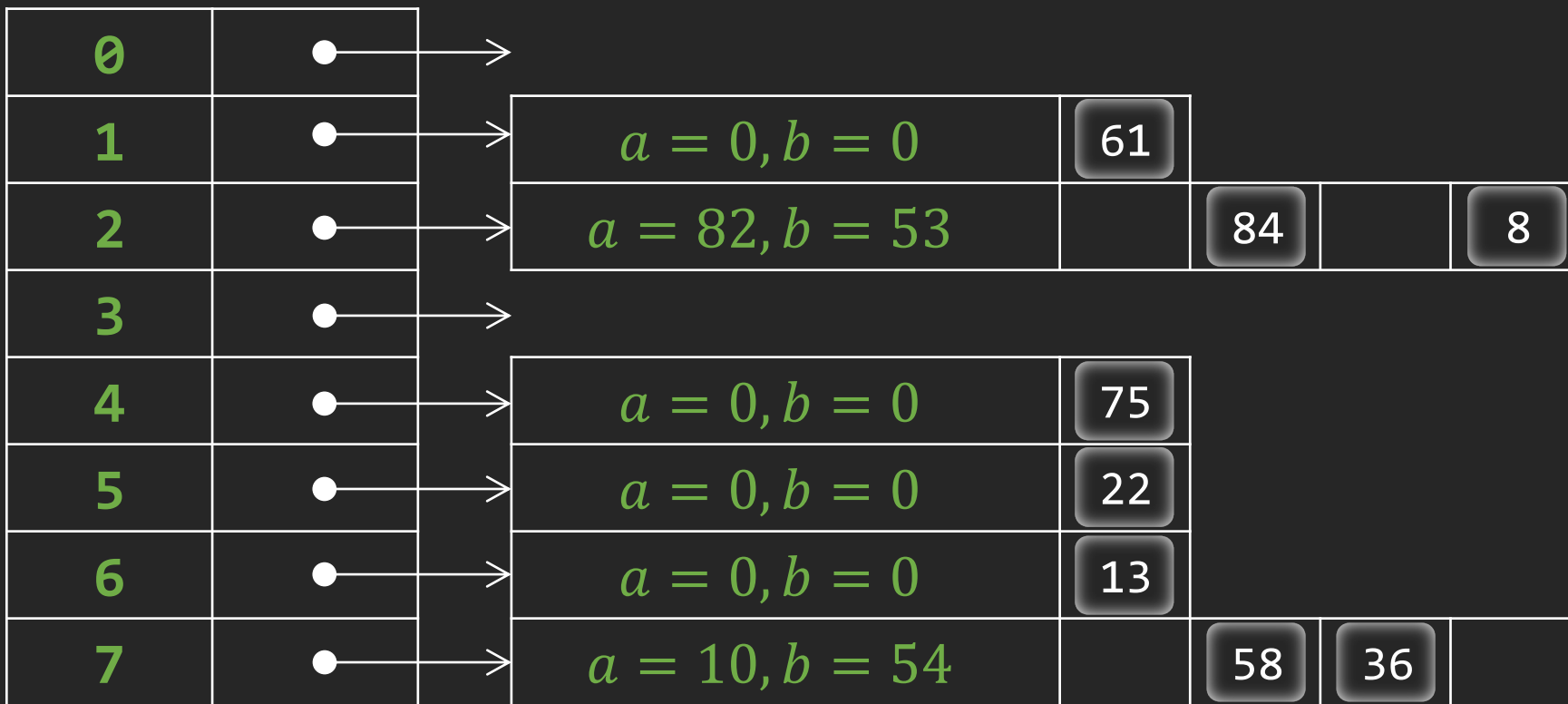
Шаг 3 Внутри каждой цепочки выполняем **«перехеширование»** с отдельной хеш-функцией в **новую хеш-таблицу** размера N_i^2 .



Длина цепочки в i -ой ячейке хеш-таблицы

Идеальное хеширование – Пример

Шаг 3 Внутри каждой цепочки выполняем **«перехеширование»** с отдельной хеш-функцией в **новую хеш-таблицу** размера N_i^2 .



Длина цепочки в i -ой ячейке хеш-таблицы

Идеальное хеширование – Анализ

Пусть выполняется хеширование M ключей в хеш-таблицу размера $N = M^2$ с использованием **случайно выбранной универсальной** хеш-функции.

Вероятность того, что коллизии будут возникать в принципе, составляет **менее** $1/2$.

Идеальное хеширование – Анализ

Пусть выполняется хеширование N ключей
в хеш-таблицу размера $M = N$ с использованием
случайно выбранной универсальной хеш-функции.

Тогда справедливо:

Суммарная
длина всех
таблиц 2 уровня

$$E \left[\sum_{i=0}^{M-1} N_i^2 \right] < 2N.$$

Идеальное хеширование разумно
применять для статических наборов данных