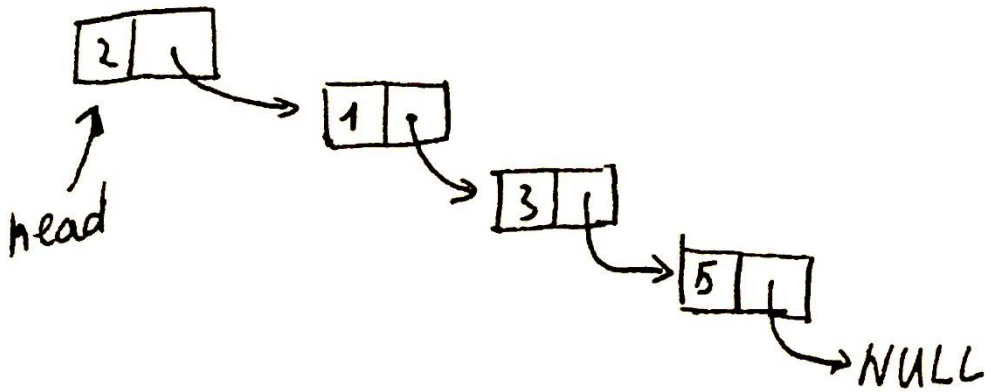
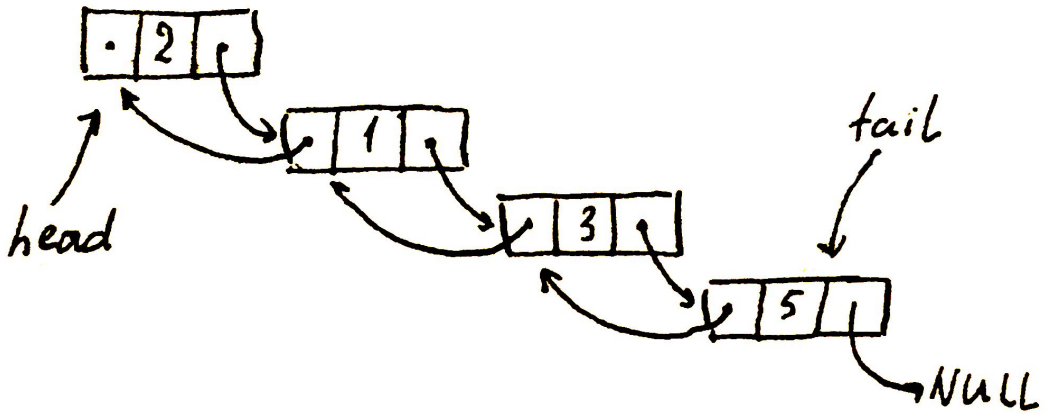
Linked lists

Односвязный список



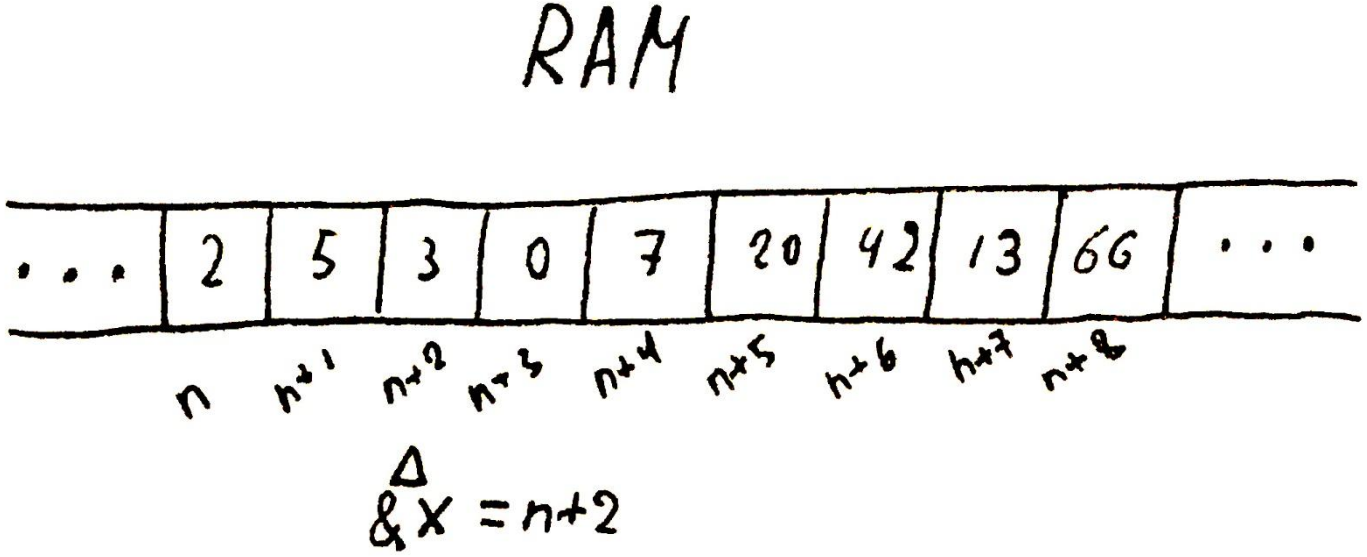
Двусвязный список



Операции над списками

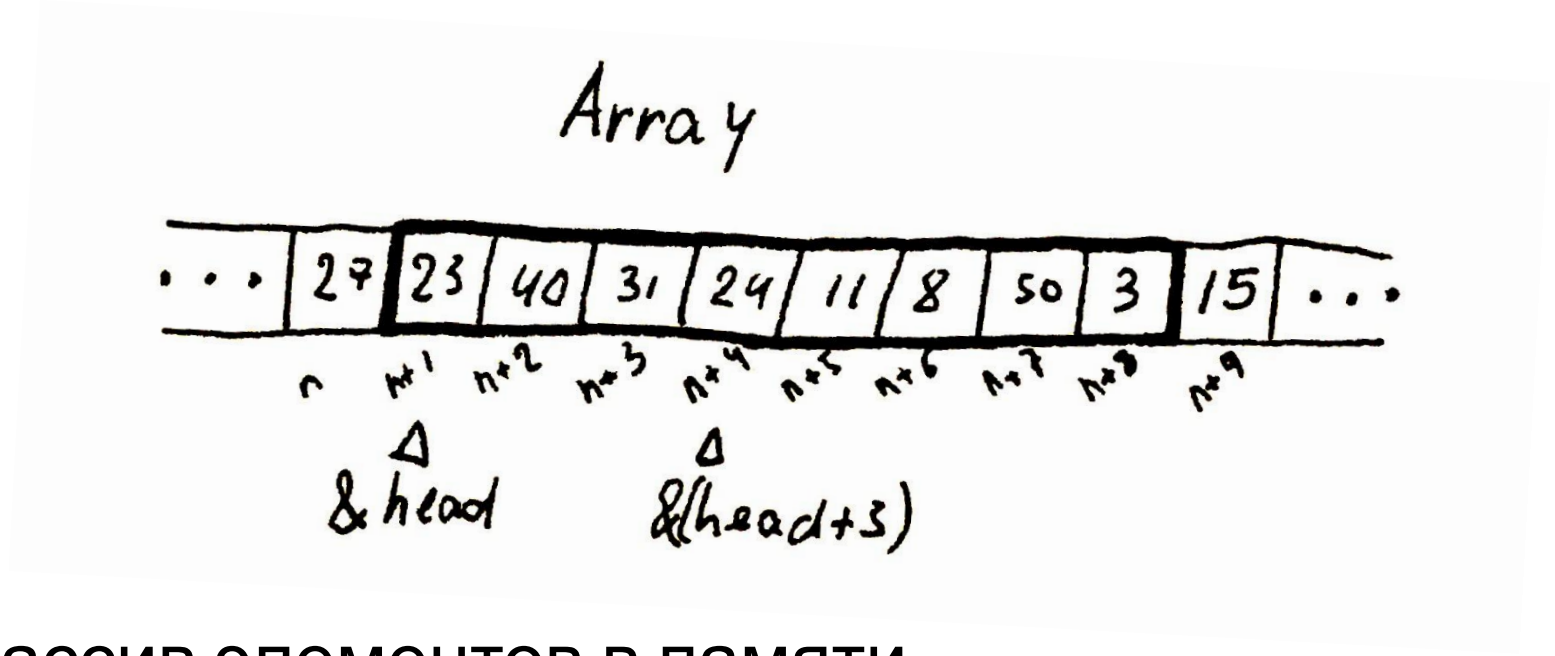
Вставка/удаление элемента в начале списка	add/remove(list, element)	Θ(1)
Вставка элемента в середине списка	insert/delete(list, index, element)	Θ(n)
Вставка/удаление элемента в конце списка	push/pop(list, element)	Θ(1) при наличии ссылки на последний элемент
		Θ(n) при последовательном доступе
Доступ к элементу по индексу	get(list, index)	Θ(n)
Поиск элемента	search(list, element)	Θ(n)

Random Access Memory



- У каждой ячейки свой адрес
- Быстрый доступ к ячейке по адресу
- Поддержка адресной арифметики

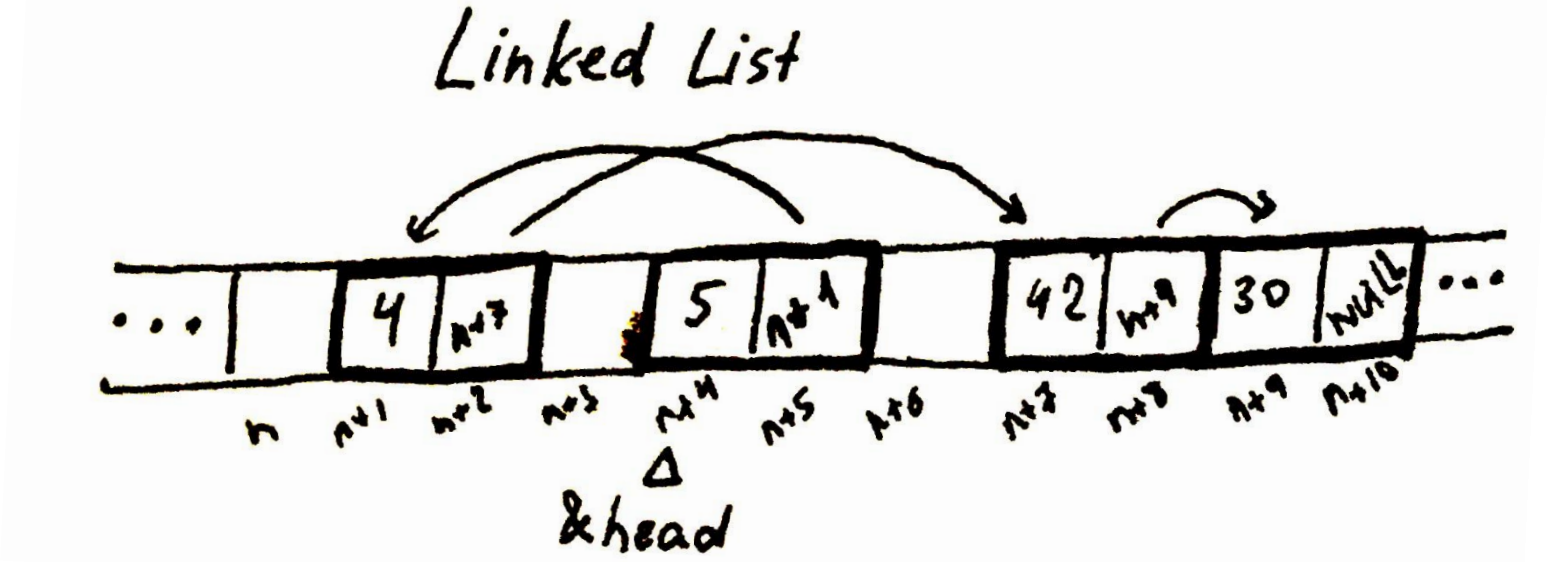
Array



- Сплошной массив элементов в памяти
- Задан адрес начального элемента
- Быстрое вычисление адреса элемента по индексу
- Невозможность расширения
- Сложность операций:

Доступ к элементу по индексу	get(array, index)	Θ(1)
Поиск элемента	search(array, element)	Θ(n)

Linked lists once again….



- Произвольное расположение элементов
- Возможность расширения
- Медленный доступ по индексу:

Добавление/удаление элемента	add/delete(list, element, index)	Θ(1)
Доступ к элементу по индексу	get(list, index)	Θ(n)
Поиск элемента	search(list, element)	Θ(n)

Формулировка проблемы

- Необходимость быстрого поиска данных в огромных массивах
- Где применить:
  - Справочники
  - Базы данных пользователей
  - Реализация множеств
  - Ассоциативные массивы

- Желаемая сложность выполнения операций:

Добавление элемента	Θ(1)
Удаление элемента	Θ(1)
Поиск элемента по значению	Θ(1)

Создание баз данных логинов-паролей

- Огромное количество пользователей системы
- Время доступа – критический параметр
- Огромное количество возможных комбинаций
- Доступ по индексу не требуется
- Безопасность хранения данных

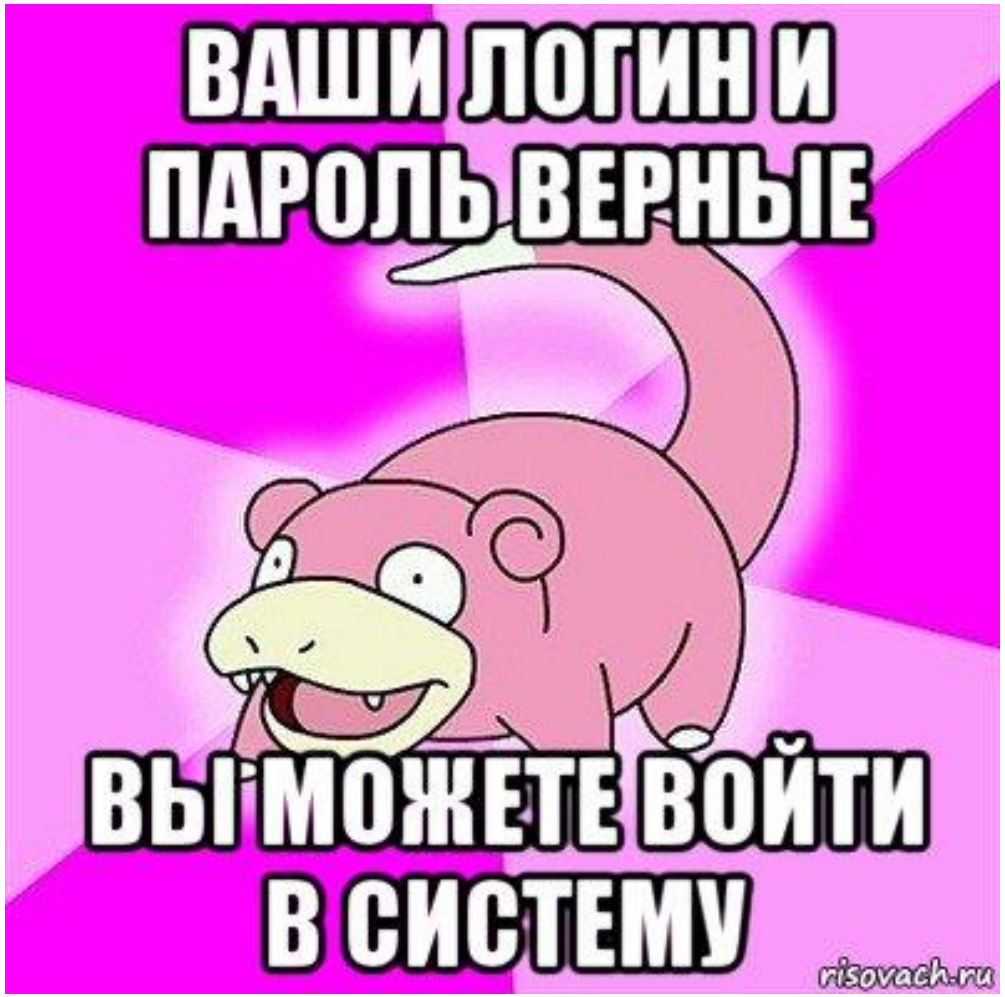
База данных телефонных номеров

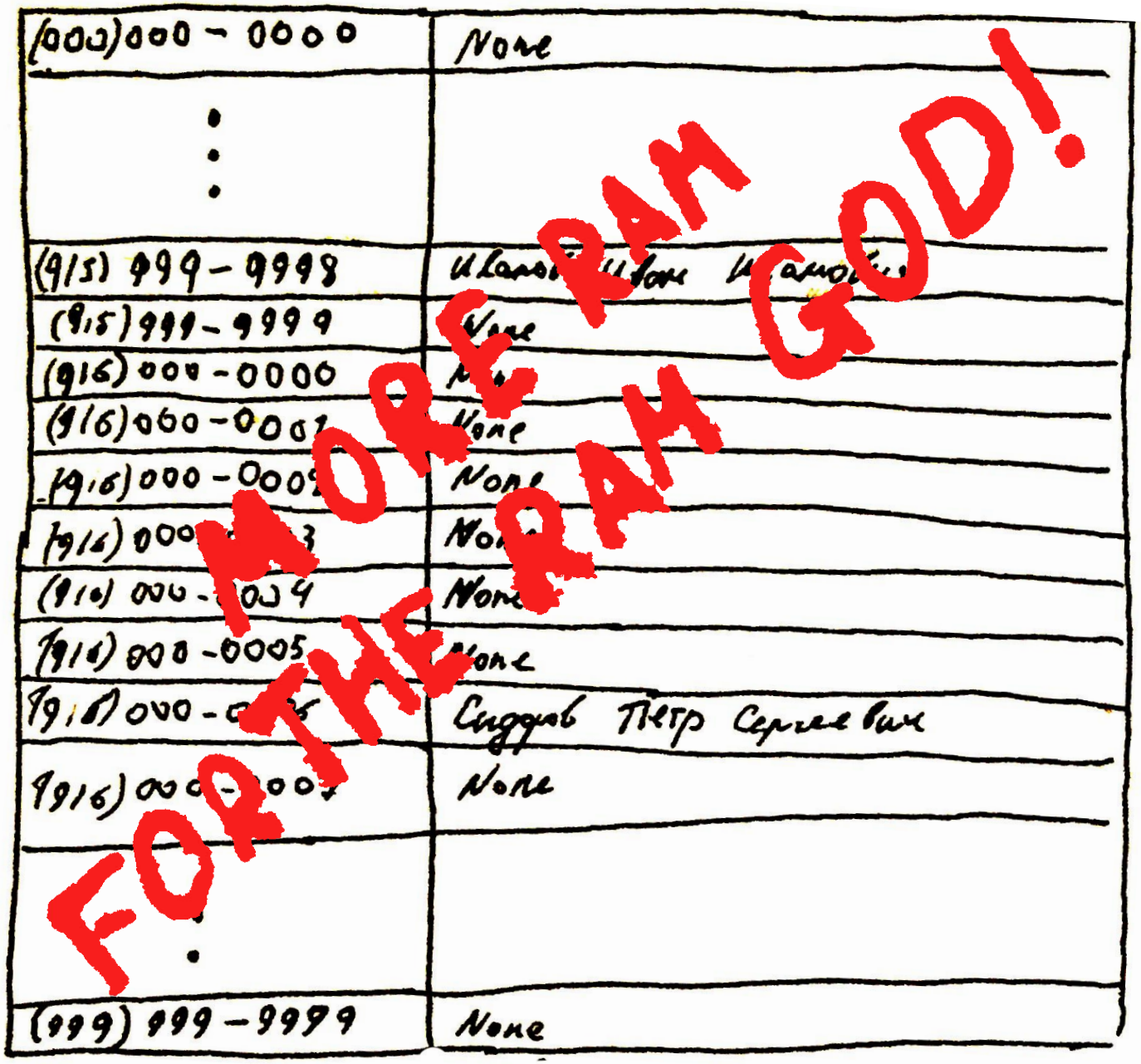
- Ограниченное количество всевозможных номеров
- Ключевой параметр – скорость поиска
- Каждый номер уникален
- Относительно небольшое количество номеров реально задействовано
- База данных постоянно пополняется

База данных телефонных номеров

- Решение – список
- Проблемы:
  - Очень долгий поиск

- Решение – огромный массив




(000)000 - 0000
None
(915) 999 - 9998
(915)999 - 9999
None
(916) 000 - 0000
(916)000 - 0001
None
(916)000 - 0002
None
(916) 000 - 0003
(916) 000 - 0004
None
(916) 000 - 0005
None
(916) 000 -
Сидоров Петр Сергеевич
(916) 000 - 0007
None
(999) 999 - 9999
None
MORE RAM
FOR THE RAM GOD!

База данных телефонных номеров

- Решение – список
- Проблемы:
  - Очень долгий поиск

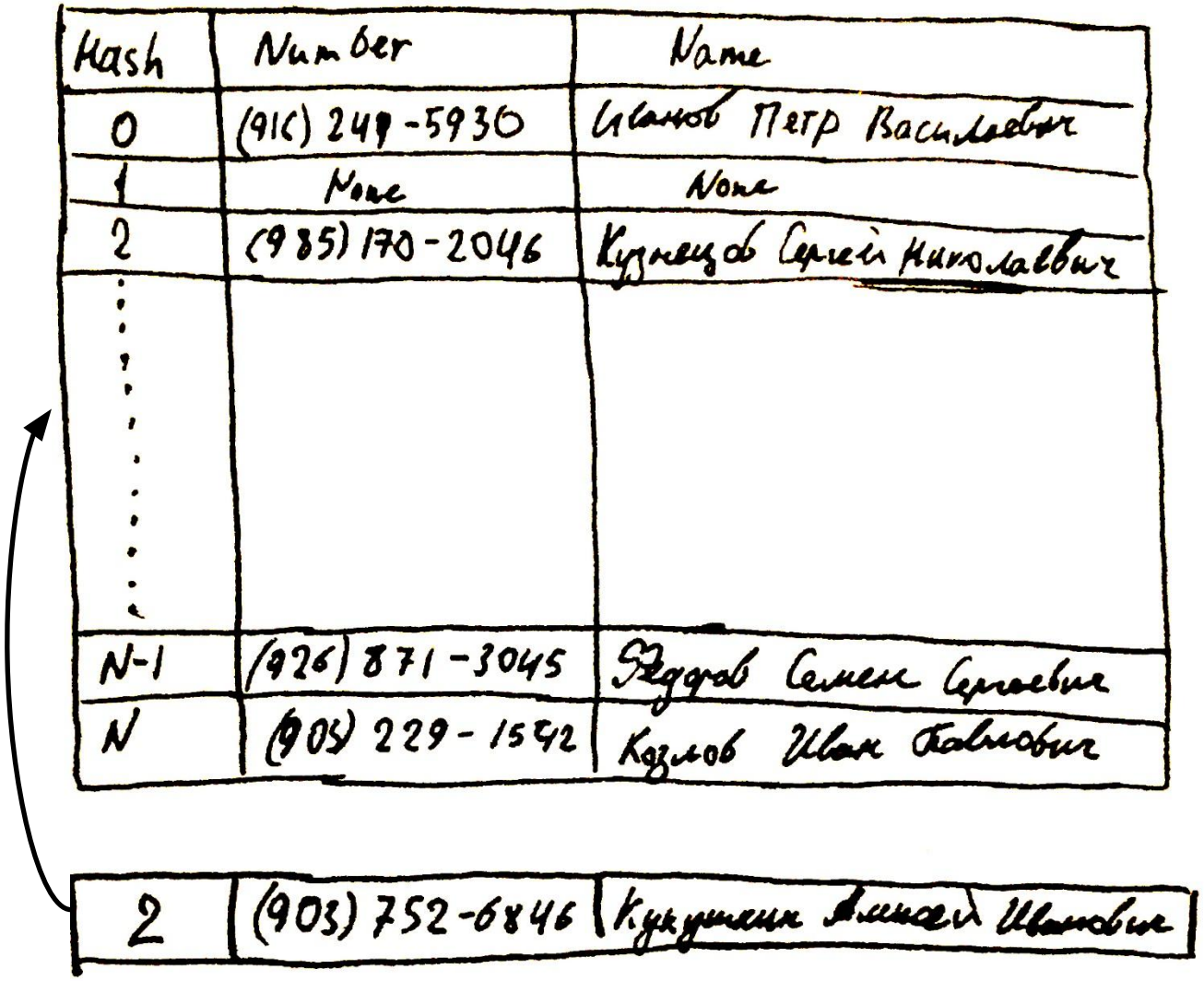
- Решение – огромный массив
- Проблемы:
  - Массив занимает очень много места
  - Большое количество полей массива не заполнено

Концепция Хеш-таблицы

- Массив фиксированной длинны
- Положение элемента определяется хеш-функцией
- Отображение элементов на множество индексов не взаимно-однозначное

- Достоинства:
  - Занимает относительно мало места
  - Быстрый поиск элемента
- Недостатки:
  - Не сохраняется порядок
  - Не эффективны при малом количестве элементов
  - В одну ячейку могут попасть много элементов

Организация хеш-таблицы и проблема коллизий



Решение проблемы коллизий

Метод цепочек (закрытая адресация)

- Элементы оформлены в список
- Поиск элемента в списке
- Произвольный размер таблицы

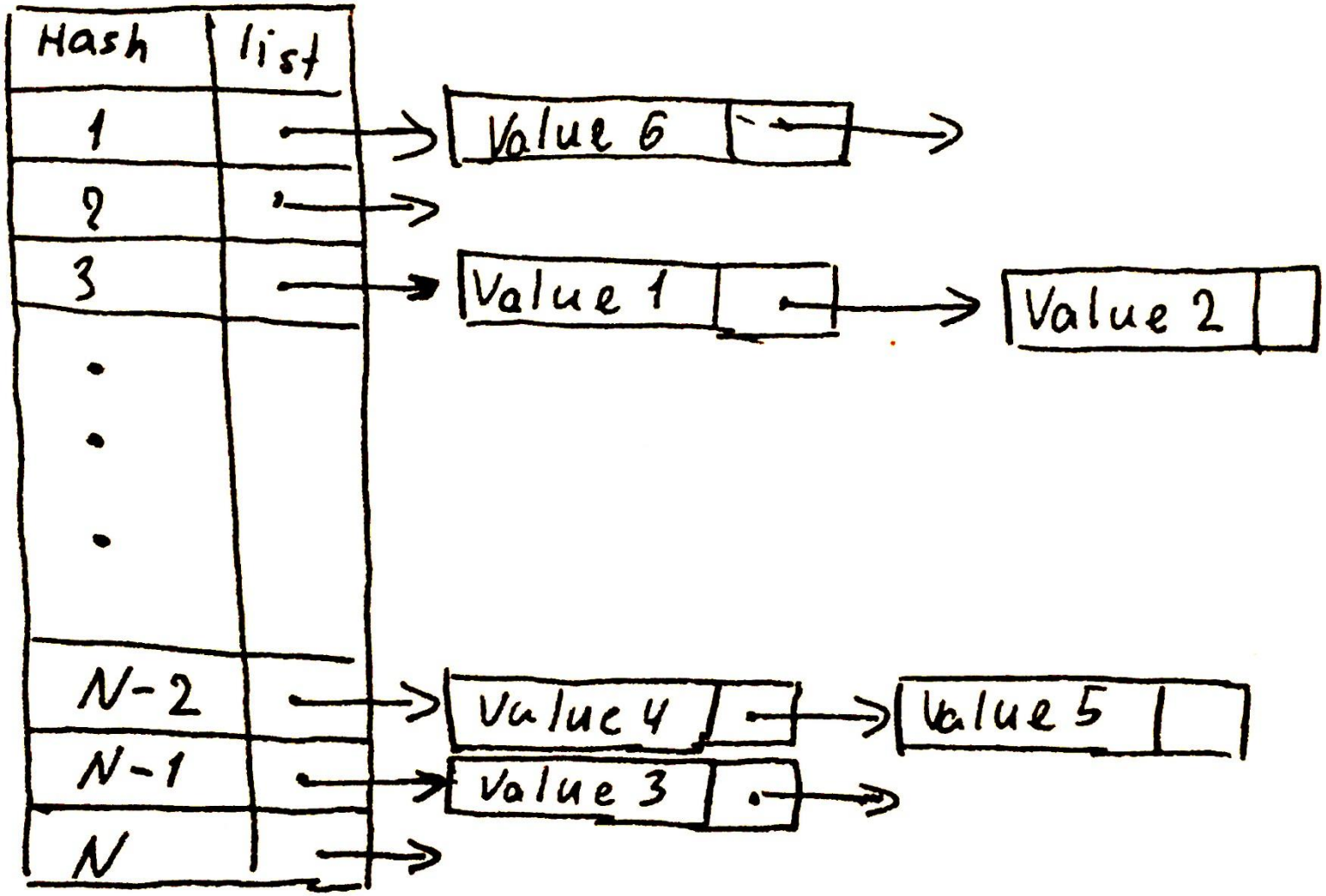
Открытая адресация

- Фиксированный размер таблицы
- Вставка элемента в ближайшую свободную ячейку
- Сложное удаление элемента
- Порядок просмотра элементов – последовательность проб

Закрытая адресация

- В каждой ячейке хранится список значений
- При добавлении элемент с заданным хешом добавляется в конец списка
- Простое удаление элемента из списка
- При хорошей хеш-функции сложность поиска $\Theta(1+\alpha)$, где α – коэффициент заполненности
- При большом коэффициенте заполненности необходима перестройка таблицы
- Тратится место на хранение ссылок в списках

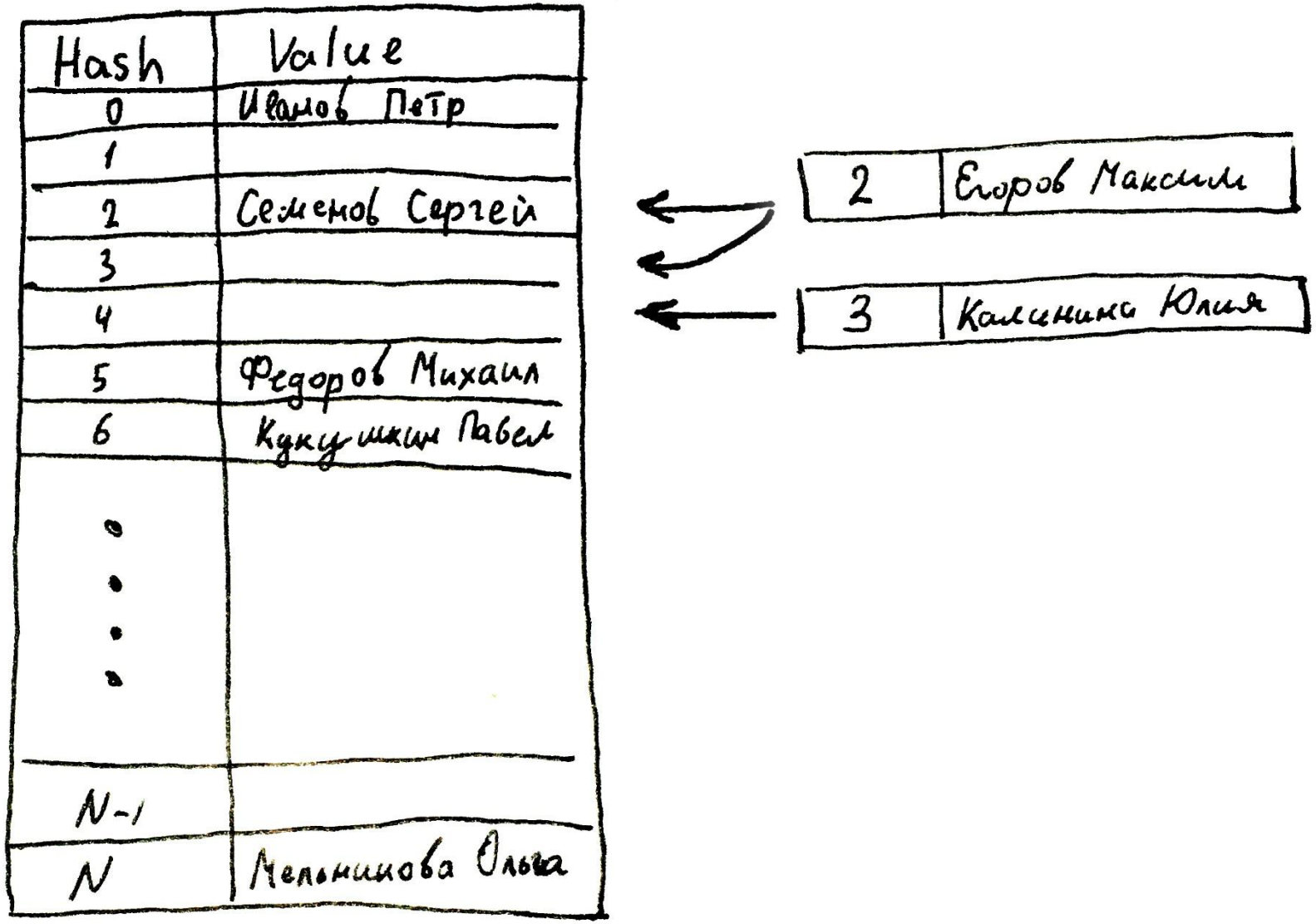
Закрытая адресация



Открытая адресация

- Порядок вставки определяется последовательностью проб
- Последовательность проб должна содержать каждую ячейку таблицы не более одного раза
- При коэффициенте заполненности, близком к 1 поиск занимает $\Theta(N)$, где N – размер таблицы
- Сложная операция удаления

Открытая адресация



Последовательность проб: пробивание

Линейное пробирование

- Последовательность просмотра: $[H(x)\ mod\ N, (H(x) + k)\ \ mod\ N, (H(x) + 2k)\ \ mod\ N, \dots]$
- Коэффициент k выбирается взаимно простым с N

Квадратичное пробирование

- Увеличение интервала со временем
- Последовательность просмотра: $[H(x)\ mod\ N, (H(x) + 1^2)\ \ mod\ N, (H(x) + 2^2)\ \ mod\ N, \dots]$
- При размере таблицы $N = 2^p$ все элементы будут просмотрены по одному разу

Последовательность проб: double hashing

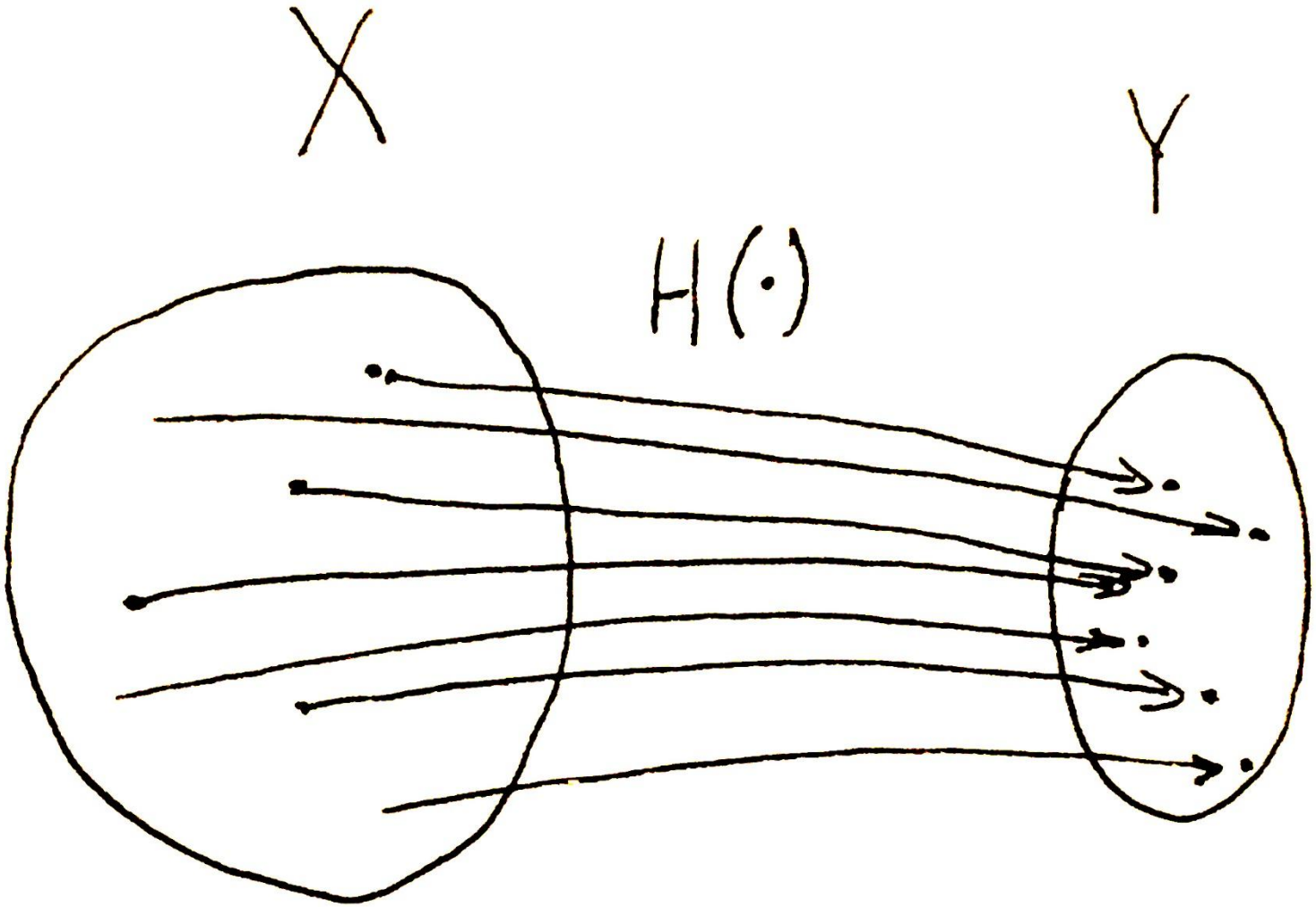
- Используются две хеш-функции H_1 и H_2
- Последовательность просмотра: $\left[H_1(x)\ mod\ N, \left(H_1(x) + H_2(x)\right)\ mod\ N, \left(H_1(x) + 2H_2(x)\right)\ mod\ N, \ldots\right]$
- Просмотры в общем случае аналогичны линейному пробированию
- При правильном выборе хеш-функций работает быстрее


Допустим, функция не ограничена на отрезке, тогда для любого разбиения данного отрезка, функция будет неограничена хотябы на одном из отрезков, полученных после разбиения. Выберем промежуточную точку .....
Че за бред ? Я просто хотел научиться писать игры!

Хеширование

- **Хеширование (англ. hashing)** — преобразование массива входных данных произвольной длины в битовую строку фиксированной длины, выполняемое определённым алгоритмом.
- **Хеш-функция –** функция, реализующая алгоритм хеширования
- **Хеш (хеш-сумма, хеш-код)** – результат выполнения хеширования

Хеш-функции



Хеш-функции

- Результат применения хеш-функции имеет фиксированную длину
- При небольшом изменении входных данных существенно меняется результат
- Нет взаимно-однозначного соответствия между входными данными и ответом
- Ключевые свойства хеш-функций:
  - Вычислительная сложность
  - Разрядность
  - Криптостойкость

Хорошие хеш-функции

- Свойства «хорошоих» хеш-функций:
  - Минимальное количество коллизий
  - Равномерное распределение ответов
  - Быстрое вычисление

- **Идеальная хеш-функция** – хеш-функция которая отображает каждый ключ из набора S во множество целых чисел без коллизий

Применение хеш-функций

- Хранение и поиск данных
- Компьютерная графика
- Контрольные суммы
- Информационная безопасность

Примеры хеш-функций

- Остаток от деления

$$H(x) = x \; mod \; N$$

- Не криптостойкий
- Отсутствует лавинный эффект
- При некоторых N возникает много коллизий

Примеры хеш-функций

- Полиномиальный хеш

$$H(x) = \left(\sum x_i * k^i\right) mod\ N$$

- Не криптостойкий
- При некоторых N возникает много коллизий

Примеры хеш-функций

- XOR – хеширование

```
h = 0
for c in X:
    h = h ^ c
```

- Не криптостойкий
- Отсутствует лавинный эффект

Примеры хеш-функций

- Cyclic redundancy check (CRC)

$$R(x) = P(x)x^n \; mod \; G(x)$$

- Не криптостойкий

Примеры хеш-функций

- Семейство MD (Message Digest)
  - MD4 (взломан)
  - MD5 (взломан)
  - MD6
- Семейство SHA (Secure Hash Algorithm)
  - SHA-1 (взломан)
  - SHA-2 (взломан)
    - SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/256, SHA-512/224
  - SHA-3 (*Keccak)*


НЕЛЬЗЯ ПРОСТО ТАК ВЗЯТЬ
И НАПИСАТЬ ХЕШ-ТАБЛИЦУ
imgflip.com

```
def calc_hash(data):
    k = 3571
    s = 0
    i = 1
    data += 84832941
    while data>0:
        s += data % 2 * k**i
        i += 1
        data //= 2
    return s % 2**32
```

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def add(self, element):
        if not self.search(element):
            node = [element, None]
            if self.head is None:
                self.head = node
            else:
                self.tail[1] = node
            self.tail = node

    def search(self, element):
        curr = self.head
        while curr is not None:
            if curr[0] == element:
                return True
            curr = curr[1]
        return False
```

```
class HashTable:
    def __init__(self):
        self.table = [LinkedList() for _ in range(256)]

    def add(self, element):
        hsh = calc_hash(element)
        self.table[hsh].add(element)

    def search(self, element):
        hsh = calc_hash(element)
        return self.table[hsh].search(element)
```

Сравнение производительности
Добавление 10000 элементов

```
%%time
for i in randlist:
    lst.add(i)
```

```
Wall time: 4.77 s
```

```
%%time
for i in randlist:
    tab.add(i)
```

```
Wall time: 88.2 ms
```

Сравнение производительности
Поиск несуществующего элемента

```
%%time
lst.search(42)
```

```
Wall time: 1.03 ms

False
```

```
%%time
tab.search(42)
```

```
Wall time: 0 ns

False
```

Сравнение производительности
Поиск 1000 случайных элементов

```
%%time
for i in randseq:
    lst.search(i)
```

```
Wall time: 980 ms
```

```
%%time
for i in randseq:
    tab.search(i)
```

```
Wall time: 17 ms
```

Сравнение производительности
Поиск 1000 существующих элементов

```
%%time
for i in randseq:
    lst.search(i)
```

```
Wall time: 563 ms
```

```
%%time
for i in randseq:
    tab.search(i)
```

```
Wall time: 9.02 ms
```

Программируем, как умеем!

Встроенные хеш-таблицы в Python

- Словари (dict)
  - Ассоциативный массив
  - Хранит пары ключ-значение
  - Быстрый поиск по ключам
- Множества (set)
  - Операции над множествами
  - Быстрый поиск элемента

Слайд
№42