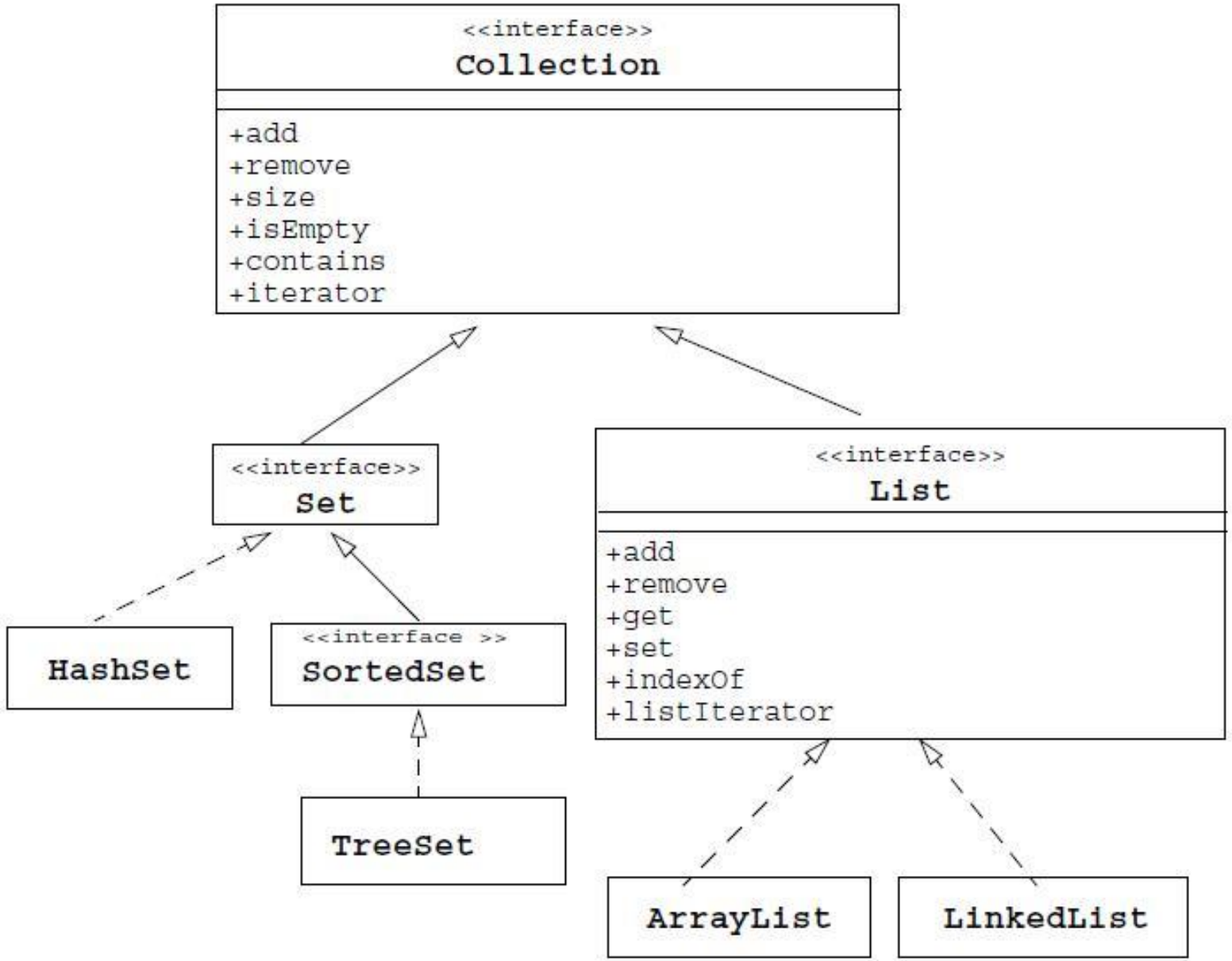
Коллекции

Владимир Сонькин

The Collections API



`Collection` interface

- Добавление в коллекцию:
  - `add(o)`
  - `addAll(Collection)`
- Удаление из коллекции
  - `clear`
  - `remove(o)`
  - `removeAll(Collection)`
  - `retainAll(Collection)`

`Iterator` interface – common methods

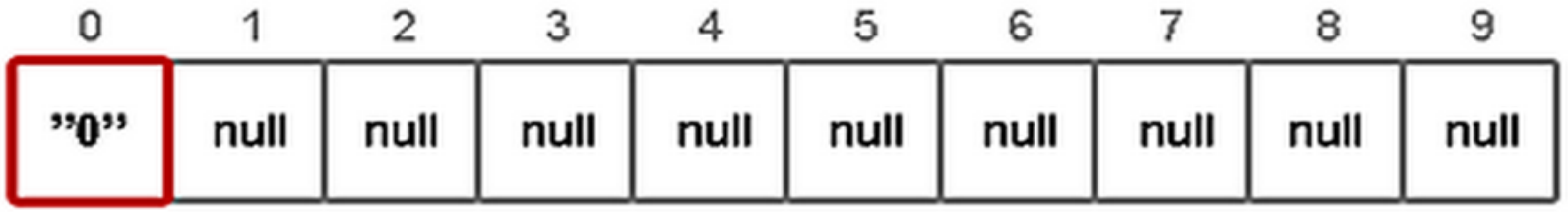
- `hasNext()`
- `next` – return next element or throws `NoSuchElementException`
- `remove` – safe way to remove elements during iteration

`List` interface

- `List` interface extends `Collection`
- They add few new methods:
  - `add(int, o)` – adds object on given possition.
  - `get(int)` – get object at given position.
  - `indexOf(o)` – get position of object.
  - `remove(int)` – remove object from given possition.

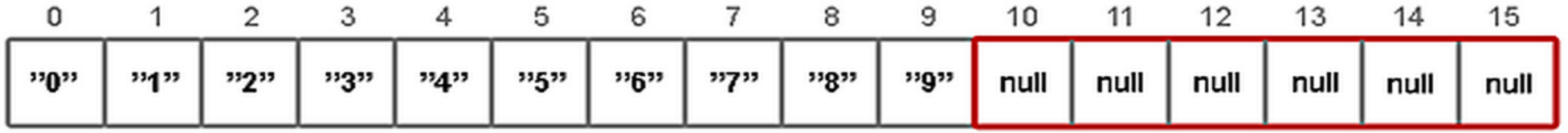
ArrayList

```
list.add("0");
```



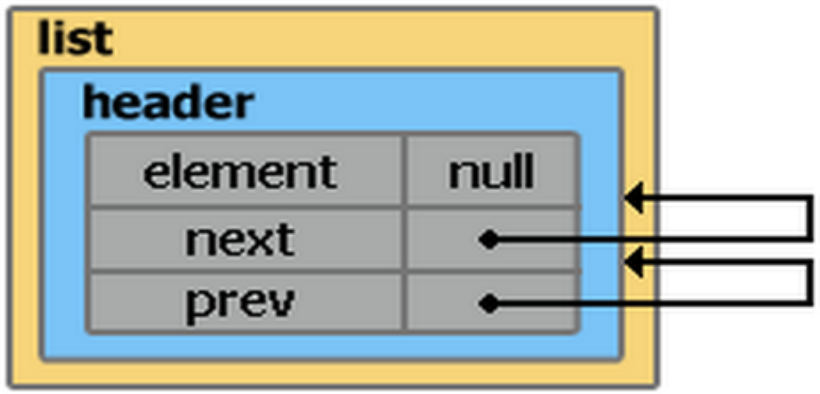

```
list.add("10");
```

При добавлении 11-го элемента, проверка показывает что места в массиве нет. Соответственно создается новый массив и вызывается **System.arraycopy()**.



LinkedList

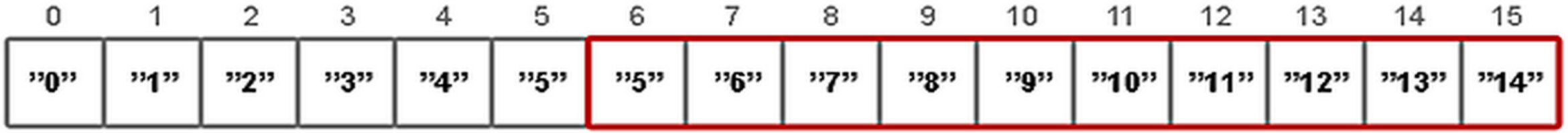
```
private static class Entry<E>
{
    E element;
    Entry<E> next;
    Entry<E> prev;

    Entry(E element, Entry<E> next, Entry<E> prev)
    {
        this.element = element;
        this.next = next;
        this.prev = prev;
    }
}
```



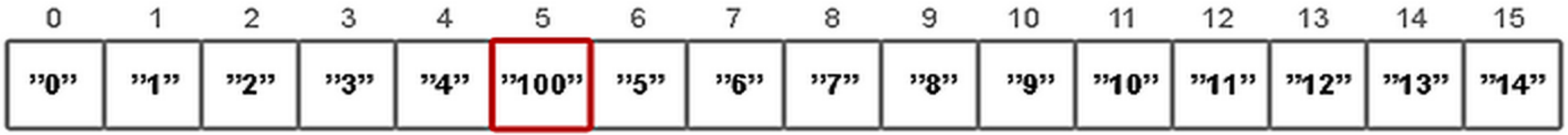
`ArrayList` – inserting

```
list.add(5, "100");
```

```
System.arraycopy(elementData, index, elementData, index + 1, size - index);
```



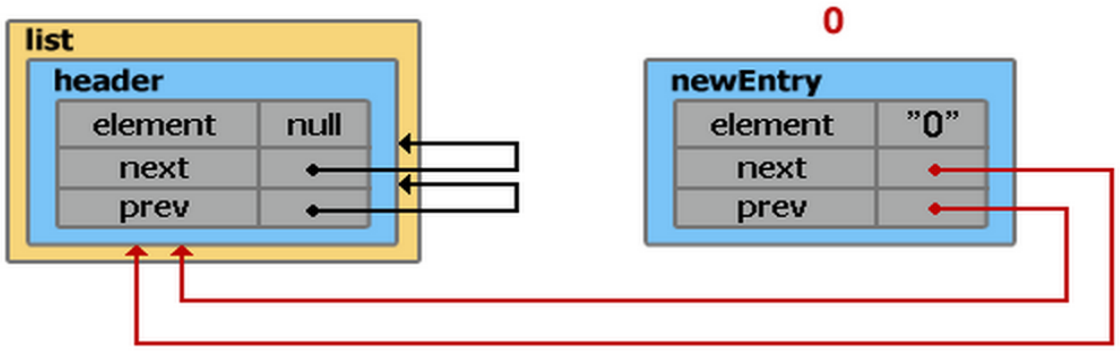

```
elementData[index] = element;
size++;
```



`LinkedList` – adding an element

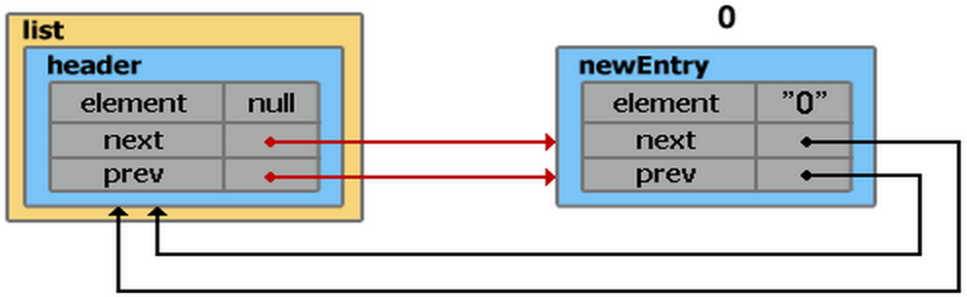
1) создается новый новый экземпляр класса **Entry**

```
Entry newEntry = new Entry("0", header, header.prev);
```



2) переопределяются указатели на предыдущий и следующий элемент

```
newEntry.prev.next = newEntry;
newEntry.next.prev = newEntry;
size++;
```



`Set` interface

- `Set` interface extends `Collection`
- `Set` – нет концепции порядка
- `Set` может содержать дубликаты

`Set` interface

- When you try to add element that already exists in set, method add() will return false and set will not be changed
- Sets use th equlas method to check duplicates.

`Set` - adding the duplicate

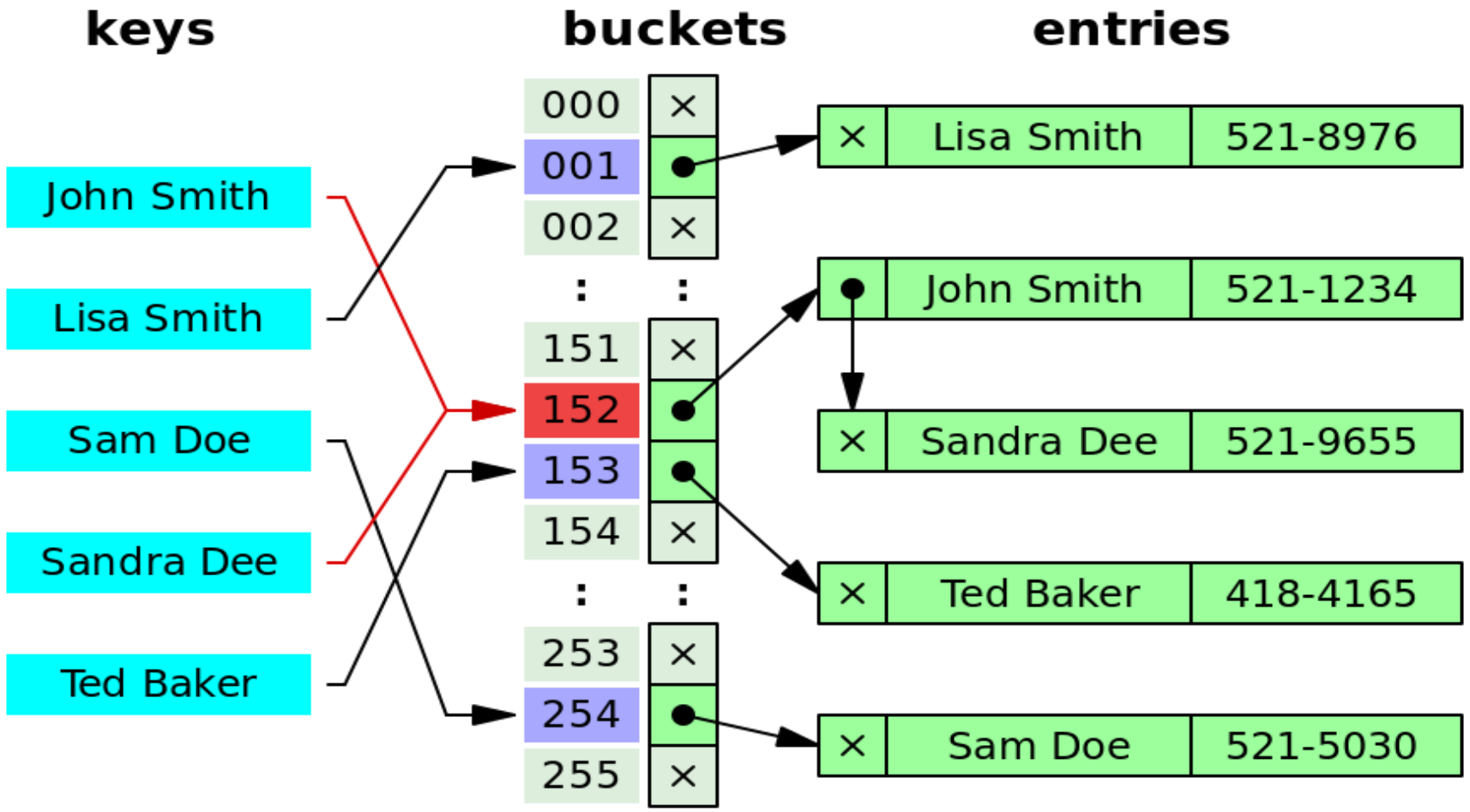
```
public class DuplicateInSet {

    public static void main(String[] args) {
        Set<Point> points = new HashSet<>();
        Point p1 = new Point(1, 1);
        Point p2 = new Point(1, 1);
        points.add(p1);
        points.add(p2);
        System.out.println(points.size());
    }
}
```

Hash Set

HashSet

- Basic operations will always execute in constant time – O(n).
- HashSet base on concept of „buckets”.
- Elements are put to bucked base on hashCode value.

HashSet


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Sorted Set

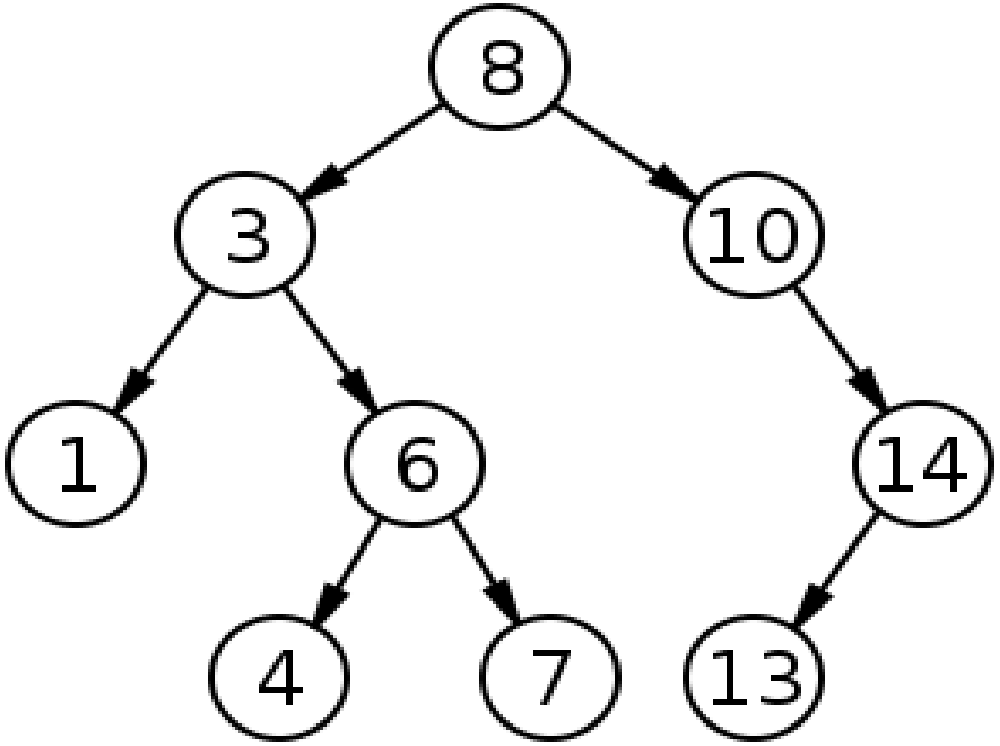
SortedSet

- SortedSet interface extends Set
- TreeSet implements SortedSet
- TreeSet stores elements in sorted order
- Basic operations has logarithmic complexity – O(logN)

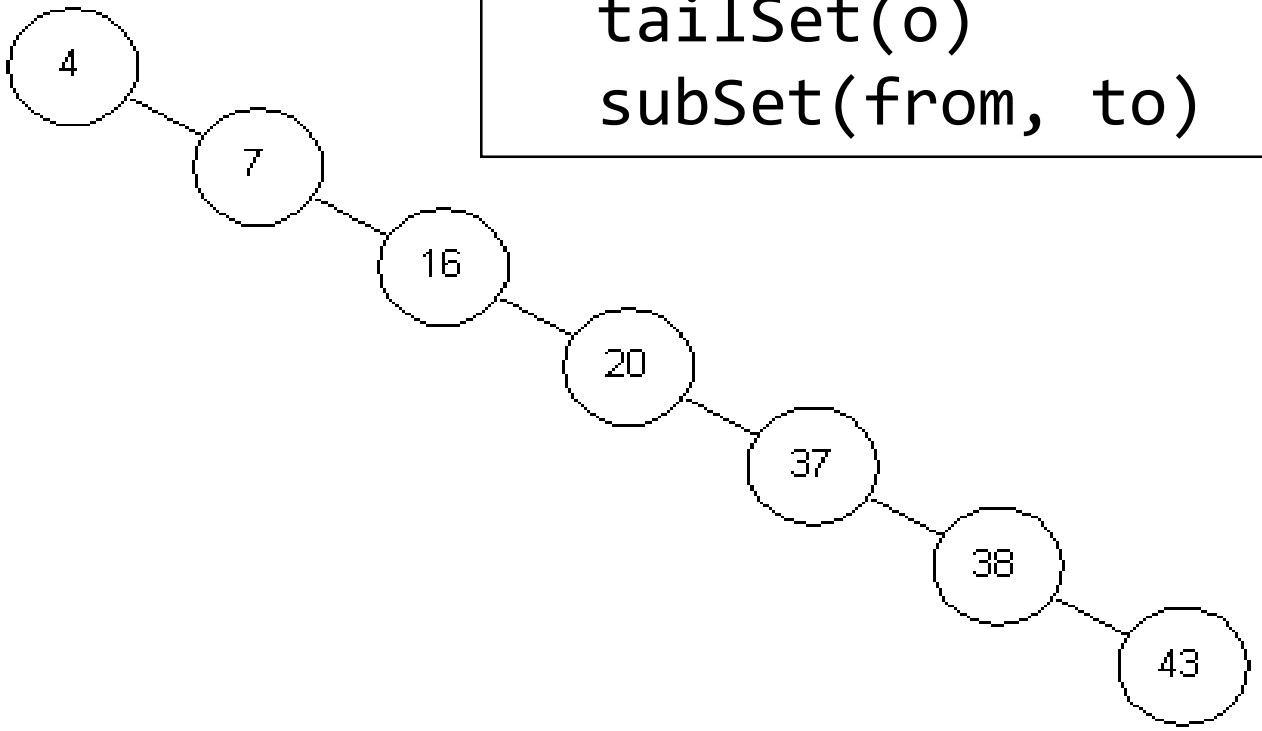
`TreeSet` – binary tree

`SortedSet` methods :

- `first` and `last`
- `headSet(o)`
- `tailSet(o)`
- `subSet(from, to)`

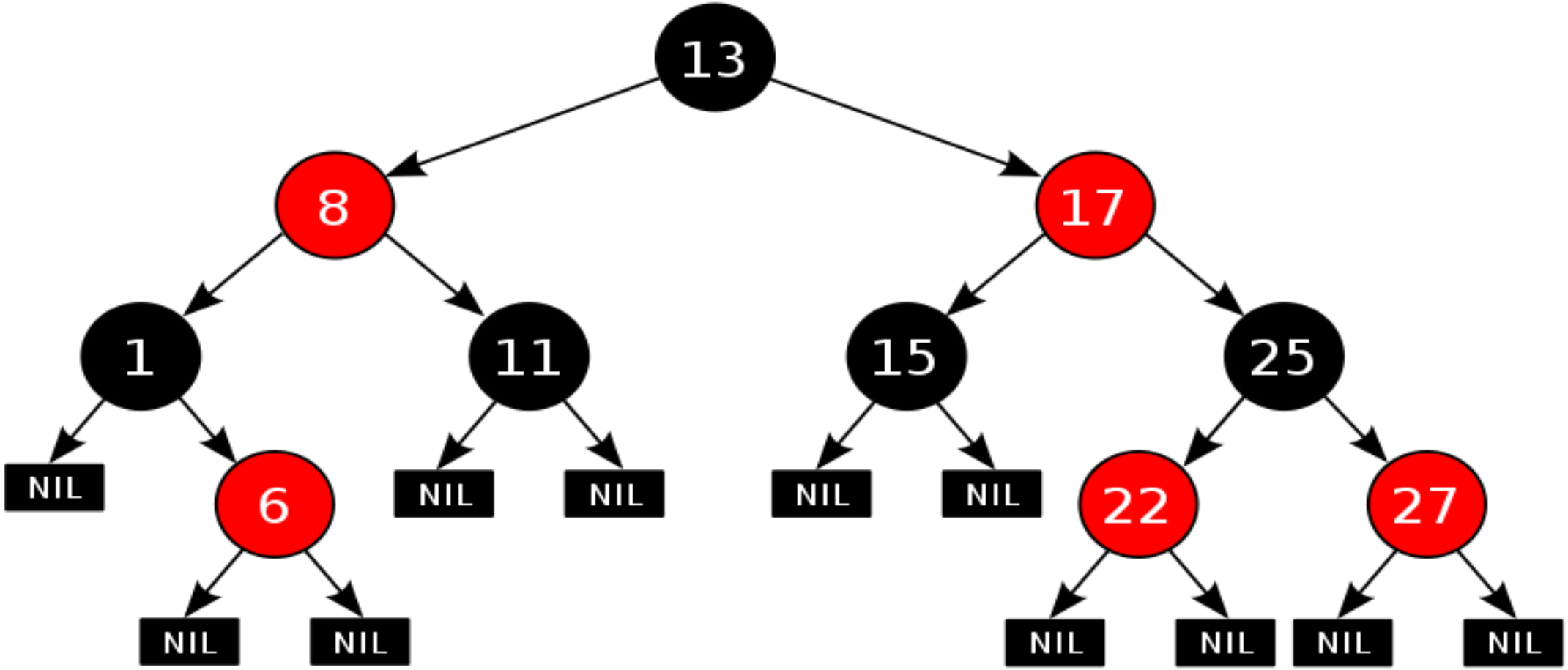


balanced tree



unbalanced tree

`TreeSet` – red-black tree



Self-balancing binary tree

`Comparable` interface

- `Comparable` interface defines `compareTo(that)` method.
  - return positive number if this > that
  - return zero if this is equal to that
  - return negative number if this < that
- `Comparable` implementation represents natural order.
- Many Core Java classes are `Comparable`

`Comparator` interface

- `Comparable` could be not enough.
- We would like to sort data in other than natural order.
- In that case we use `Comparator` interface.
- `Comparator` has method `compare(o1, o2)`. That method returns integer value in the same way as `Comparable.compareTo`.

Map

`Map` interface

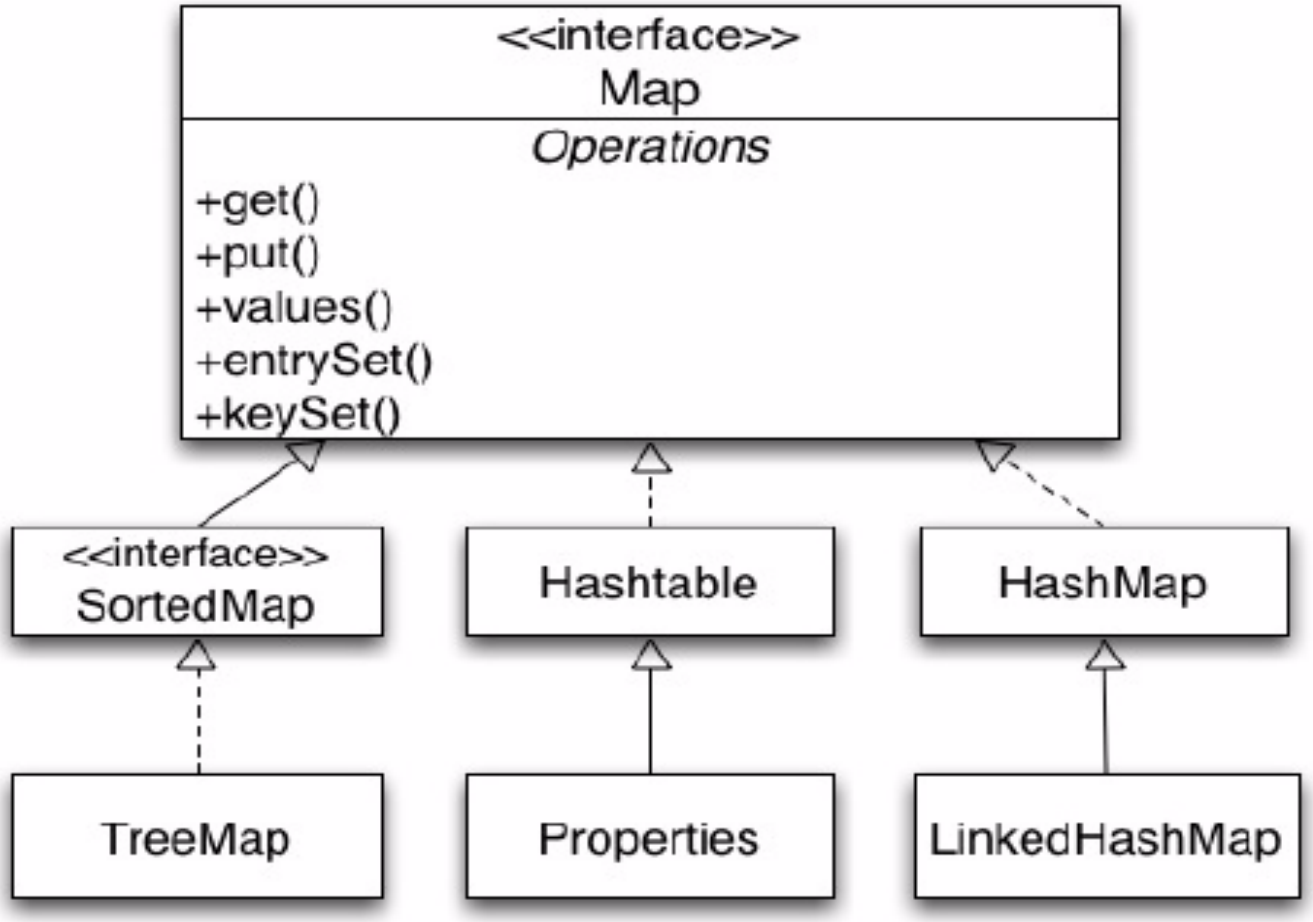
- `Map` combines two collections, called **keys** and **values**
- `Map` associates exactly one value (it can be `null`) with each key
- Each key is used in `Map` only once
- A good example is a dictionary

Key	Value
Иванов	9590000
Петров	4230043
Сидоров	2340055

`Map` interface

- `Map` gives us different set of operations:
  - `put(k,v)` – inserting key-value
  - `get(k)` – returns value assosioted with key or **null**
  - `containsKey(k)` – check if contains key
  - `clear` – remove everything
  - `keySet` – set of keys
  - `values` – collection of values
  - `entrySet` – set of `Entry` objects, pairs of key-value.

Map hierarchy



HashMap

- HashMap store keys like a HashSet
- Order is unpredictable
- Use hashCode and equals, so you need to implement the methods correctly.

SortedMap

- SortedMap interface guarantees ordering keys.
- TreeMap is implementation of SortedMap.
- TreeMap is similar to TreeSet and works with Comparable and Comparator interfaces.

Хэш-структуры

Как они устроены и зачем

Частый вопрос на собеседовании

- Что такое хэш-код и для чего он?

Задача: хранить Иванова и искать Иванова

Как ищем?

=> Бежим по массиву и сравниваем Иванова с ячейкой.

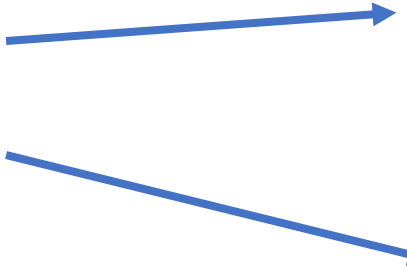
А может там другой Иванов? Ивановых много!

=> Давайте сравнивать не только по имени!

А тот ли это Иванов?

Перестанет ли он быть тем самым Ивановым при смене должности?
А даты рождения?

Фамилия: Иванов
Имя: Иван
Отчество: Иванович
Дата рождения: 01.01.1990
Должность: эксперт



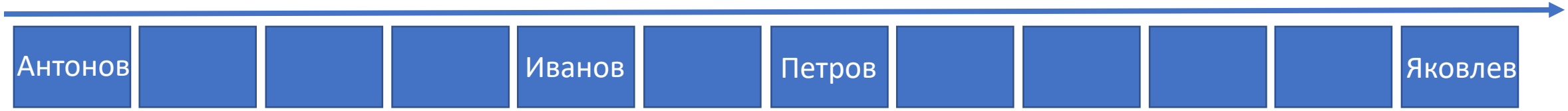
идентичность

состояние

⇒ На основе идентичности создается метод **equals()**
⇒ сравнивает один объект с другим
⇒ сравнивает идентичность, но **не состояние**
⇒ возвращает **true** если объекты одинаковые или **false** если разные

⇒ **А приведите еще примеры состояния?**

Ищем перебором… А побыстрей никак?



1) *Зададим порядок объектов (кто больше Иванова, а кто меньше)*
   - *Например, **по имени** – по алфавиту*
   - *Или скажем **по дате рождения***
   - *Для этого в Java объекты должны имплементировать интерфейс **Comparable***
   - ***compare()** возвращает -1, 0, 1*

     Иванов compare Петров => -1

     Иванов compare Антонов => 1

     Иванов compare Иванов => 0
2) *А теперь можно **отсортировать массив** и искать половинным делением: прыгаем сразу на середину*
3) *Скорость поиска – **$\log_2 N$***

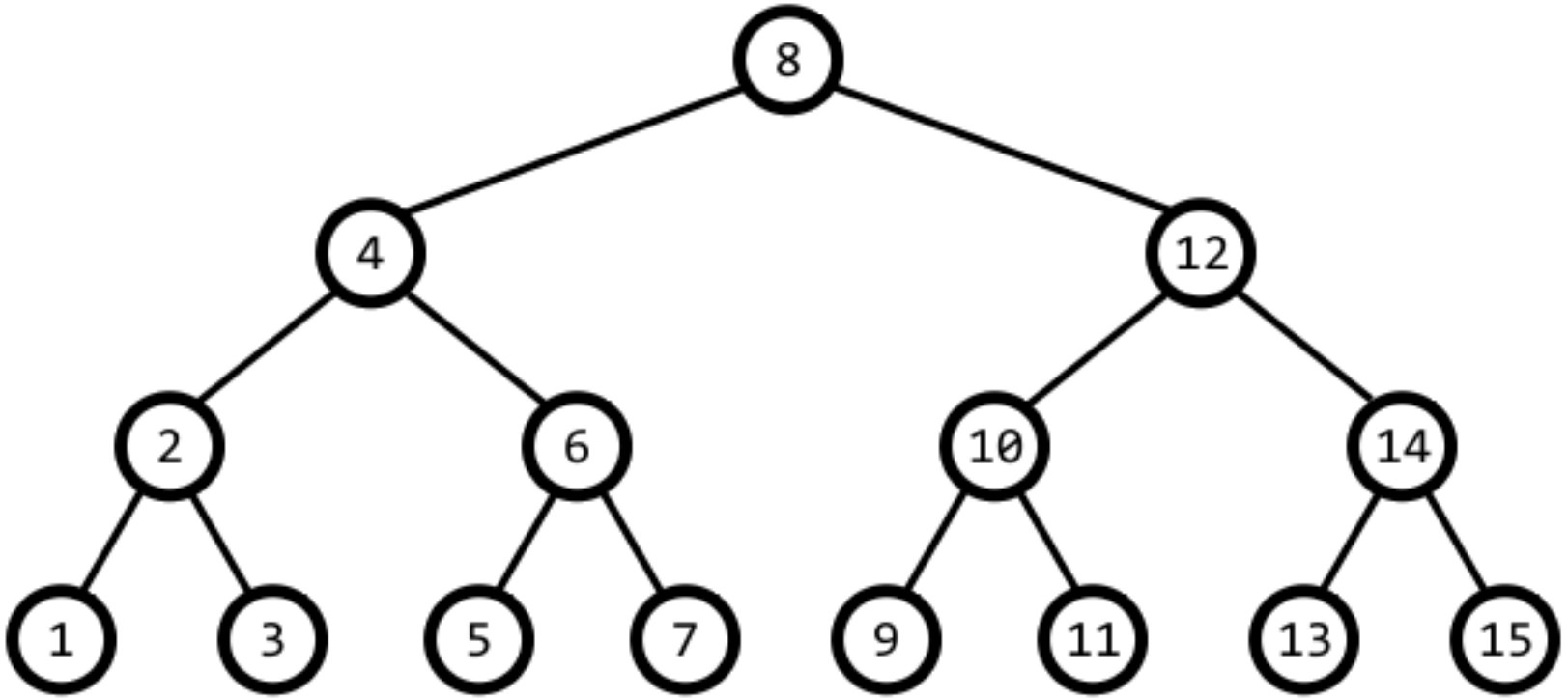
   это очень быстро, например для 10 тыс. элементов это 13. А для миллиона – 20.

4) *Но до поиска его надо сортировать!*

Или… поддерживать всегда отсортированным!

А как поддерживать отсортированным?

=> Используем бинарное дерево!

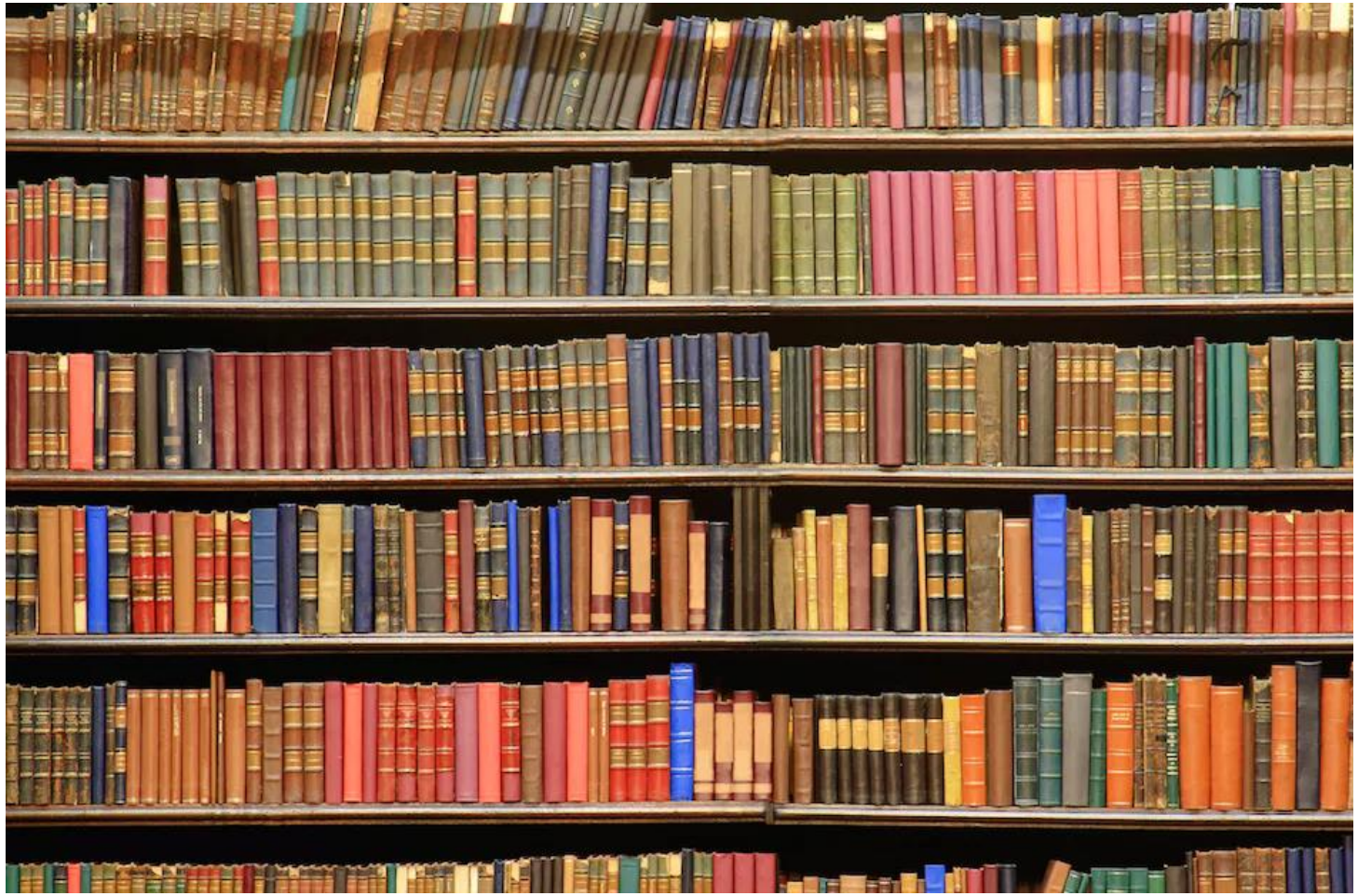


Скорость:
$\log_2 N$

В Java это структура данных TreeSet.
Ну вот, теперь неплохо.

А еще быстрее можно?

За первой космической идем…



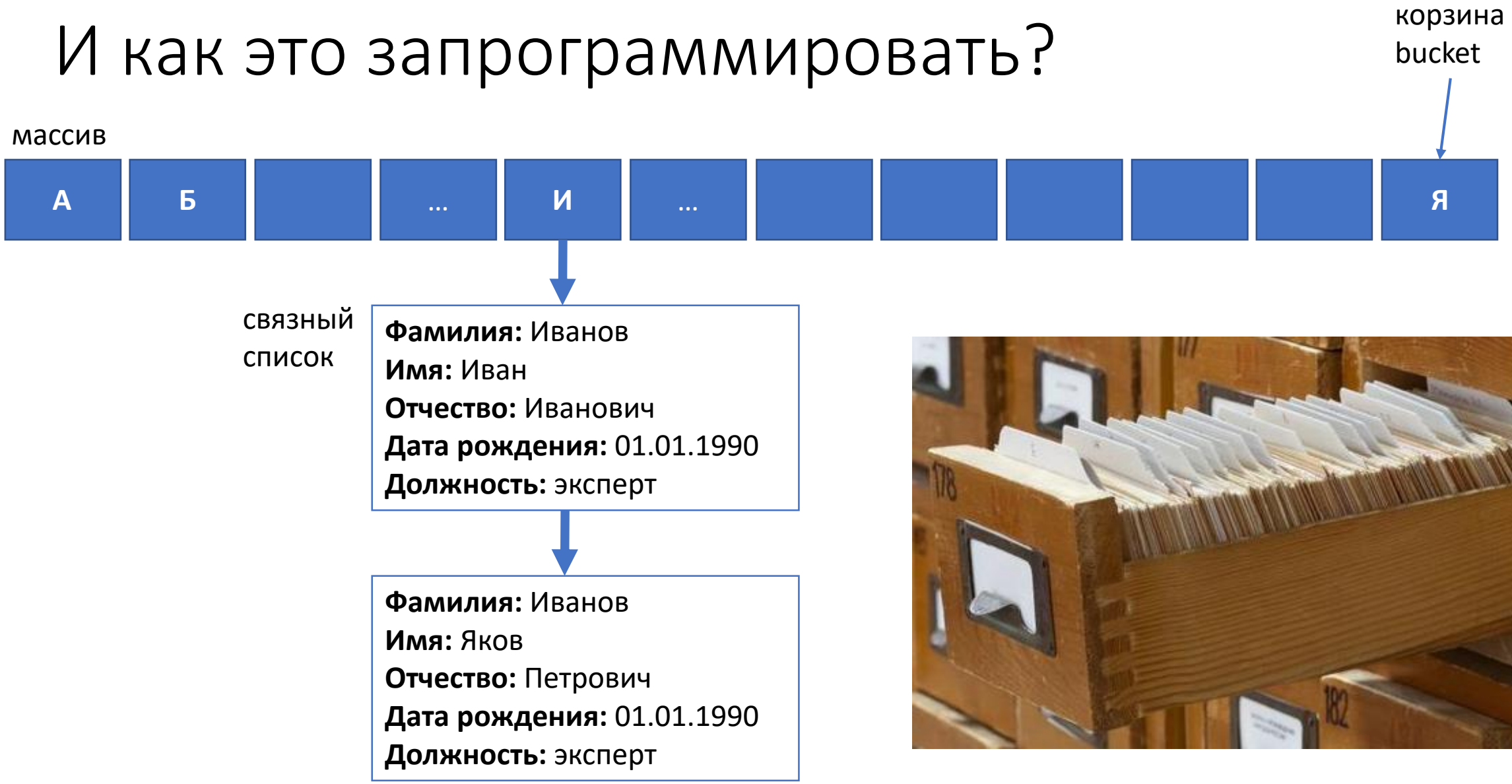
в библиотеку!

В библиотеке давным-давно придумали…

Как работает картотека?

Фамилия: Иванов
Имя: Иван
Отчество: Иванович
Дата рождения: 01.01.1990
Должность: эксперт

И как это запрограммировать?



А что если в ящике битком?

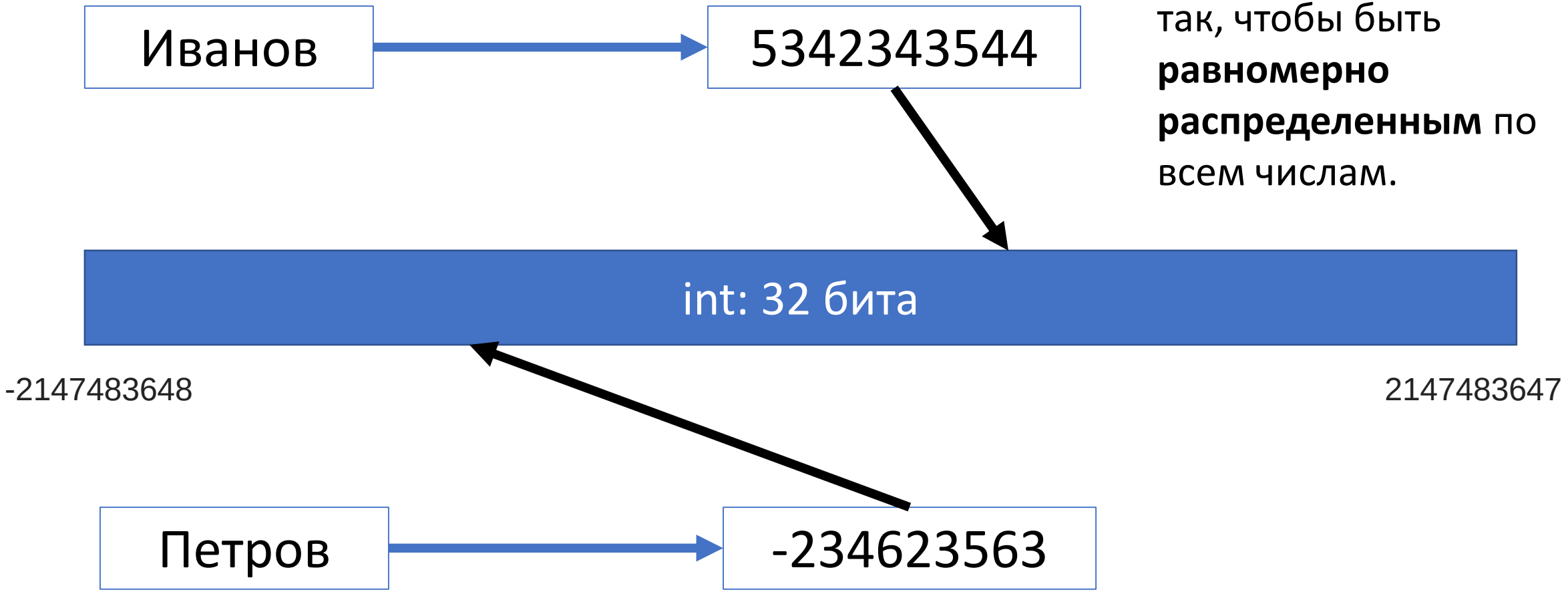
И вообще поиск в ящике занимает время, надо же найти нужную карточку перебором...

Делаем больше ящичков: возьмем 2 буквы

Делаем больше ящичков: давайте возьмем диапазоны

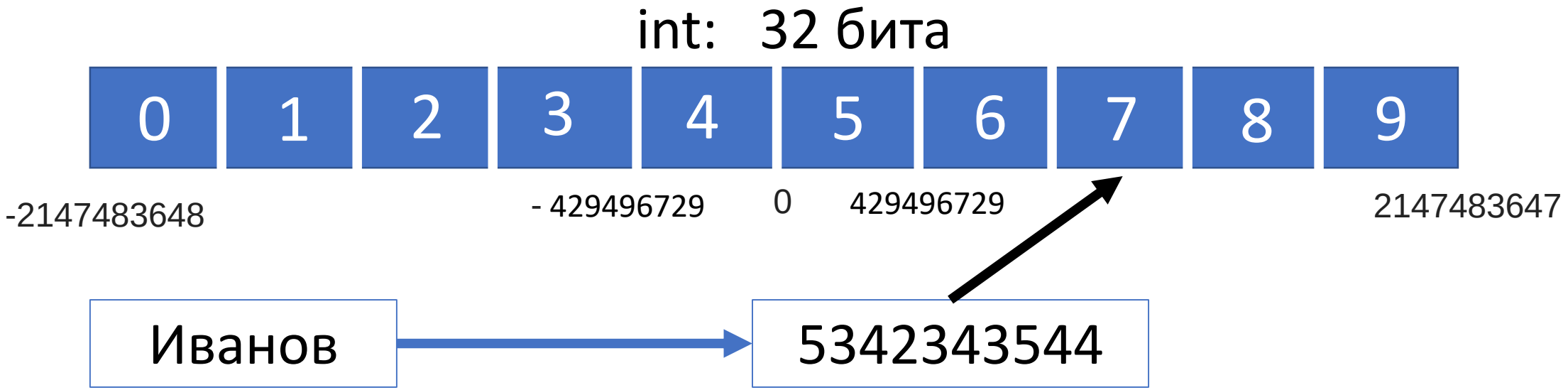
=> Только вот как выбрать такие диапазоны?
Почему именно Аа-Аг?

Для этого используют хэш-код



Идея такая:
пусть хэш-код генерится так, чтобы быть **равномерно распределенным** по всем числам.

Тогда его можно использовать для равномерного распределения по корзинам



⇒Кладем Иванова в корзину 7

⇒Как теперь найти Иванова?

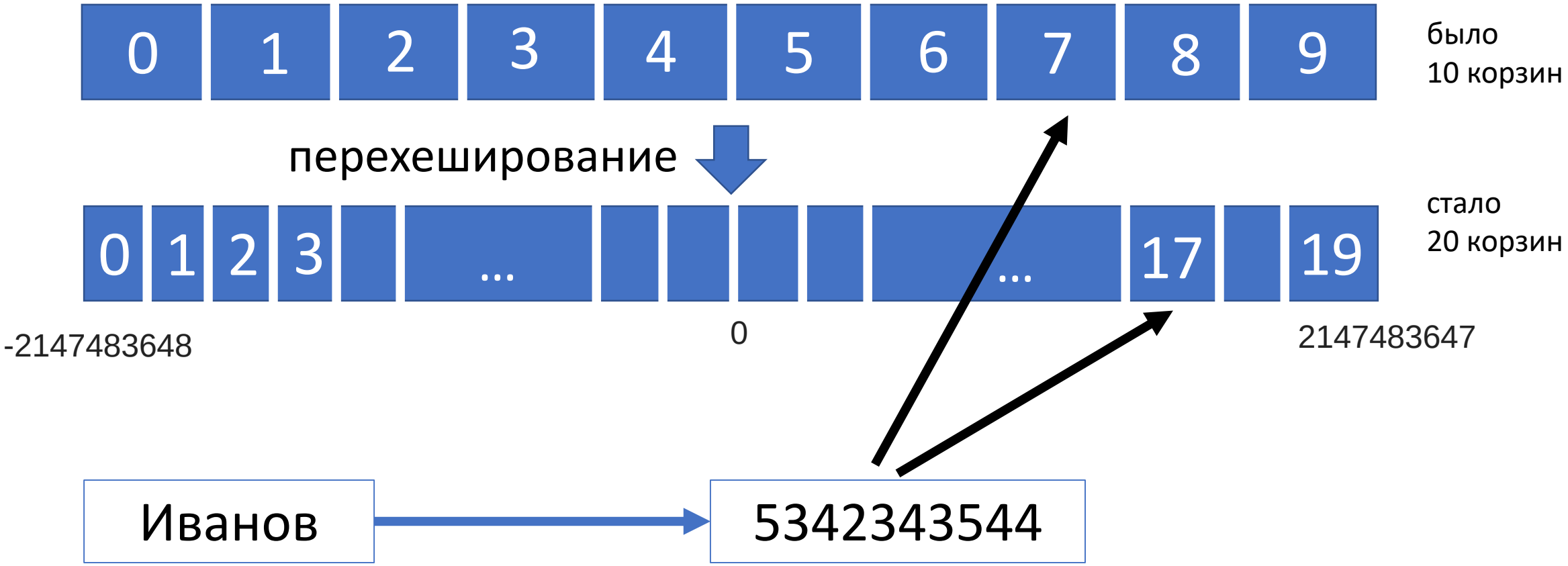
⇒Вычисляем хэш-код

⇒Вычисляем номер корзины

⇒Обращаемся сразу к нужной корзине по ее номеру

Когда в ящике много карточек… перекладываем карточки в 2 ящика

Когда в корзине много данных…
делаем больше корзин и перекладываем объекты



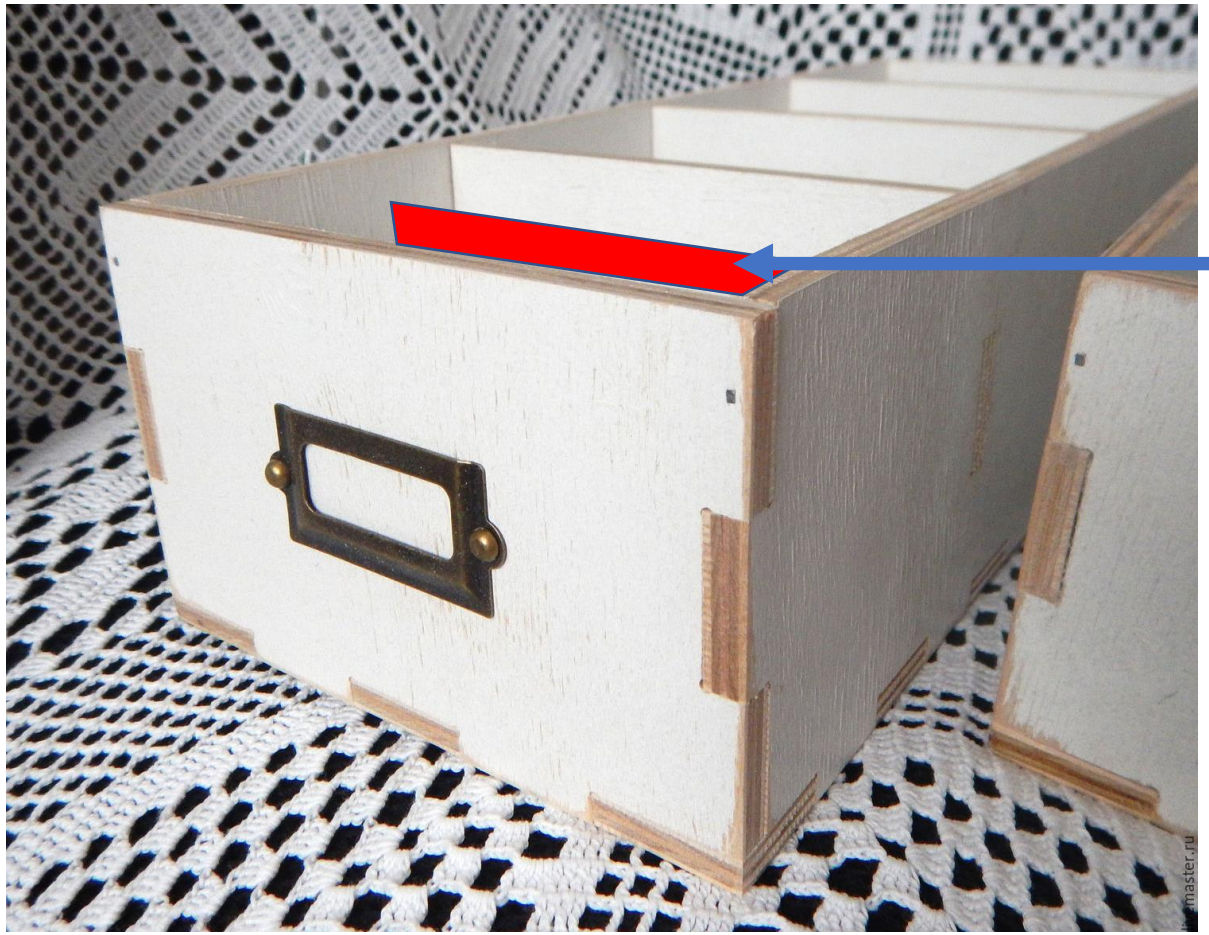
А сколько карточек должно быть в ящике для быстрого поиска?

⇒ Может быть 5?

⇒ А может быть 2?

⇒ Или даже 1?

А Java для быстрого поиска – **0.75**! (в среднем)



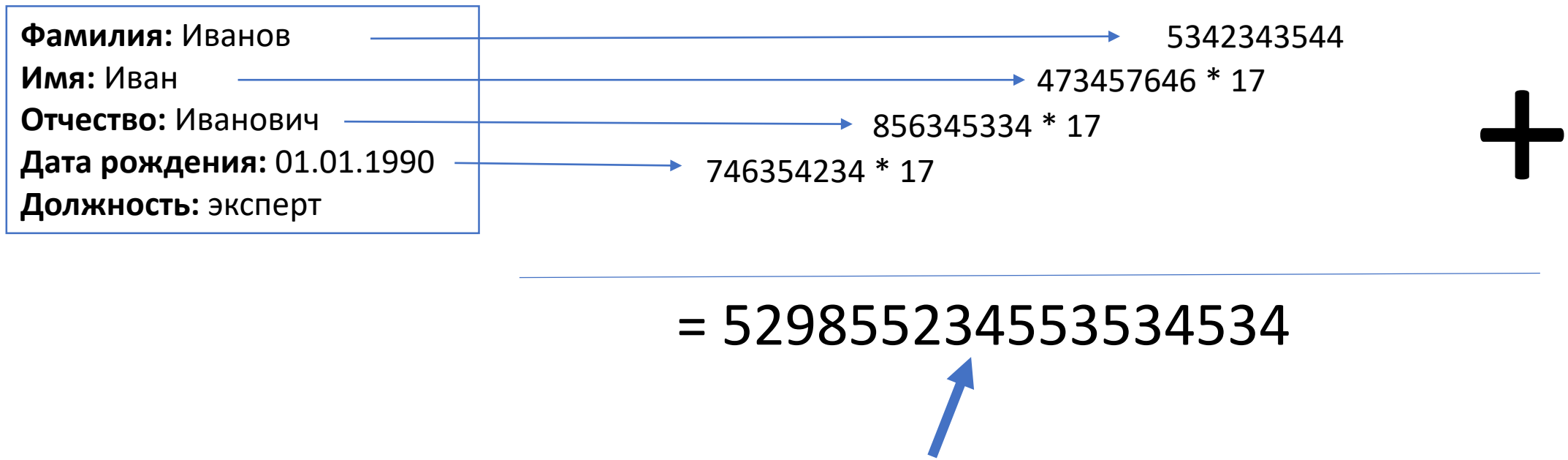
75% карточки
(в среднем)

Это значит, что в среднем 25% ящиков должны быть пусты.

На самом деле в некоторых может быть 10 карточек,
а во многих – 0.

Это называется loadFactor и этим можно управлять при создании хэш-структуры

Но Ивановых-то много!
А если надо искать не только по фамилии?



Хэш-код по фамилии, имени, отчеству, дате рождения

⇒ А как же должность? Почему ее не учли?

Но такое большое число не влезет в int!

= 52985523455353453 4

Хэш-код по фамилии, имени, отчеству, дате рождения

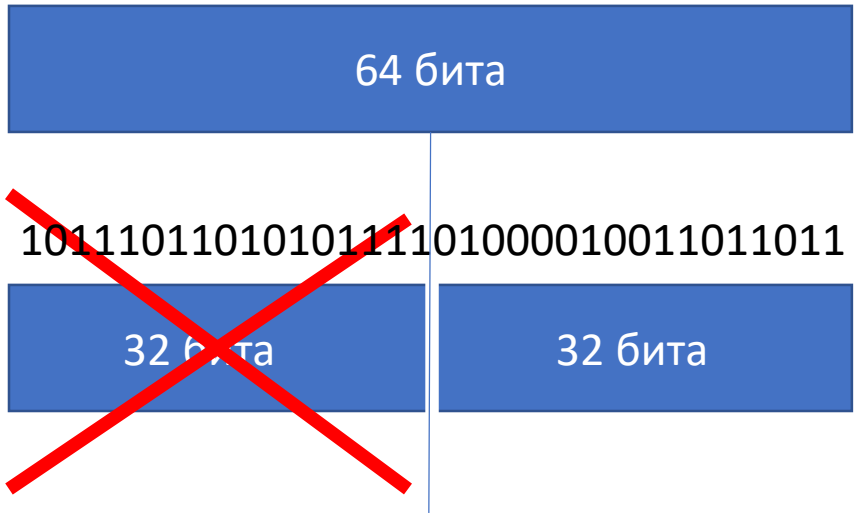
$\Rightarrow$ Чо делать-то??

А ничего. Не париться. Обрежется само ☺

52985523455353453 4

кастинг

7563456523



Итого: ищем Иванова!

Фамилия: Иванов
Имя: Иван
Отчество: Иванович
Дата рождения: 01.01.1990
Должность: эксперт

Вычисляем хэш-коды ФИО и даты рождения, складываем со сдвигом

52985523455353453 4

кастинг к int

7563456523

Находим нужную корзину

Он в корзине один:
ведь loadFactor
0.75!

Фамилия: Иванов
Имя: Иван
Отчество: Иванович
Дата рождения: 01.01.1990
Должность: эксперт

Вуаля!

Спасибо!
Всем быстрого поиска и эффективного кода!