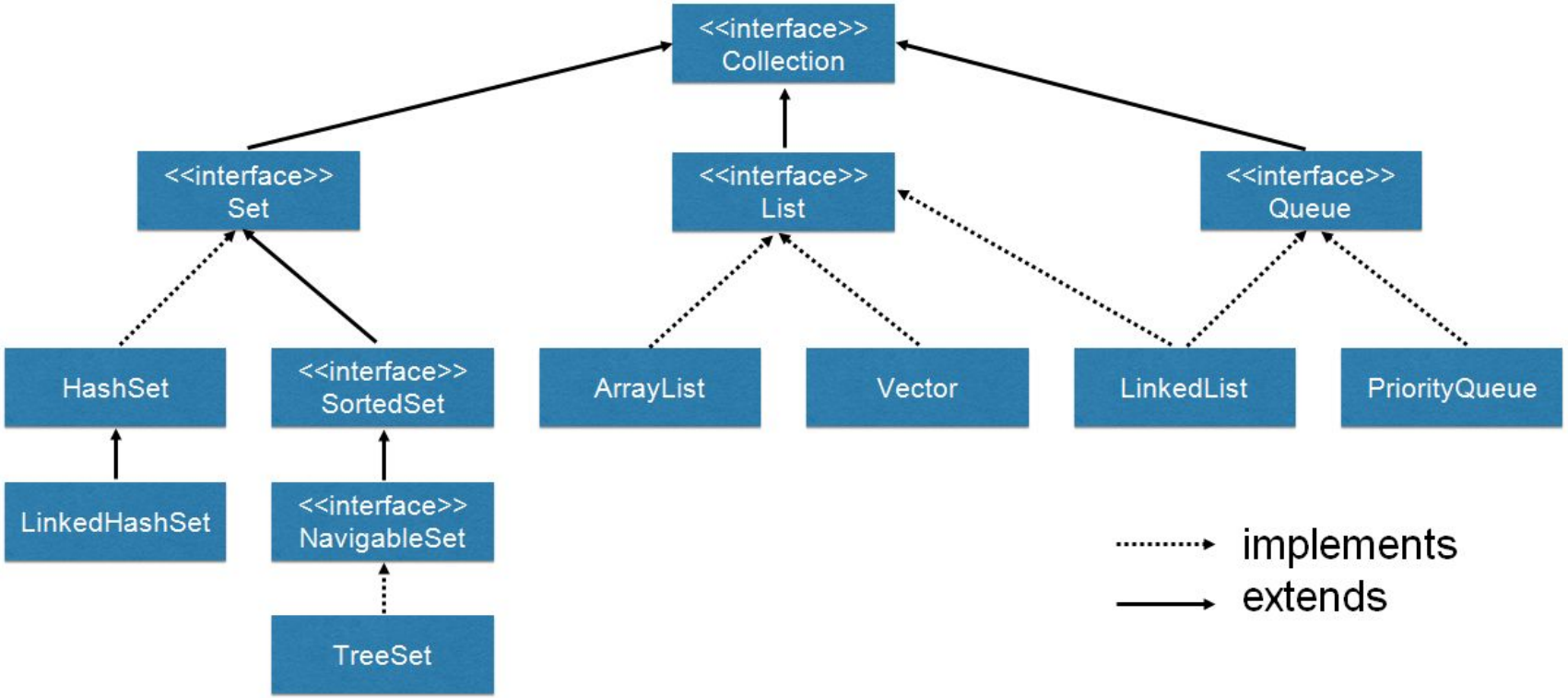
Collections

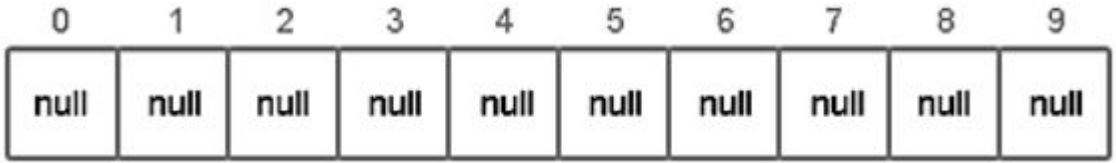
Collection Interface



Структура ArrayList

```
int size;

Object[] elementData;

ArrayList(){...}; //Конструктор с capacity по умолчанию

ArrayList(int capacity){...}; //Конструктор с capacity

ArrayList(Collection<? extends E> c){...} //Конструктор, который принимает другую коллекцию
```

- Default **capacity** = 10 (Какую вместимость имеет массив)
- Можно задать в конструкторе значение, если знаете что будет сразу много элементов
- По умолчанию все элементы null

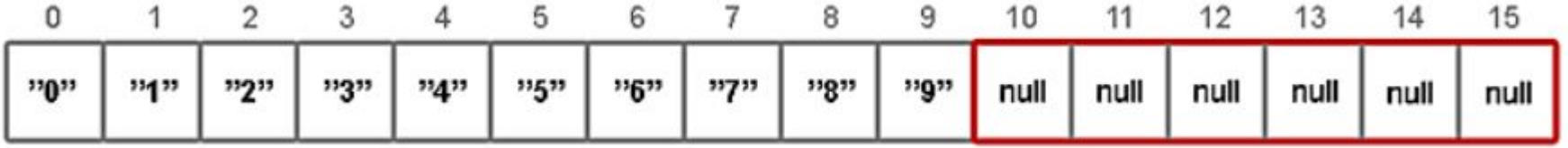


Добавление элементов

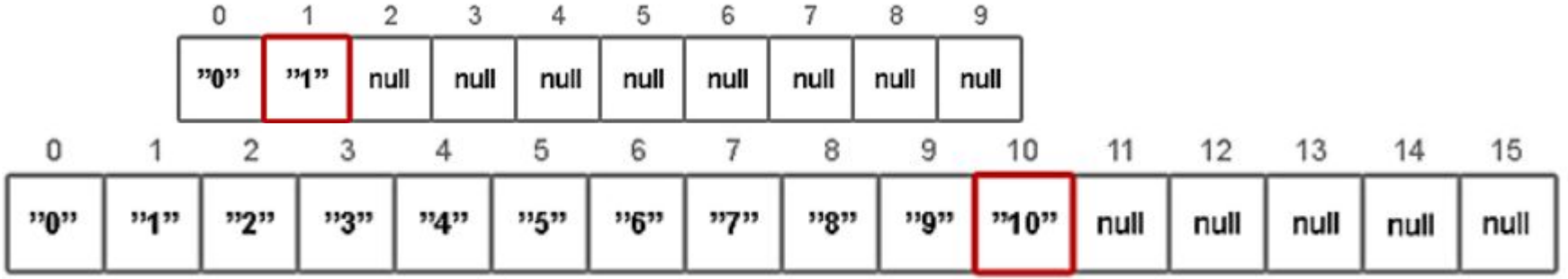
Делается в два этапа:

1. Проверяется количество элементов (size) во внутреннем массиве (elementData)

Если требуется - elementData увеличивается в 1.5 раз (`int newCapacity = oldCapacity + (oldCapacity >> 1);`)



2. “Элемент вставляется сразу за последним элементом

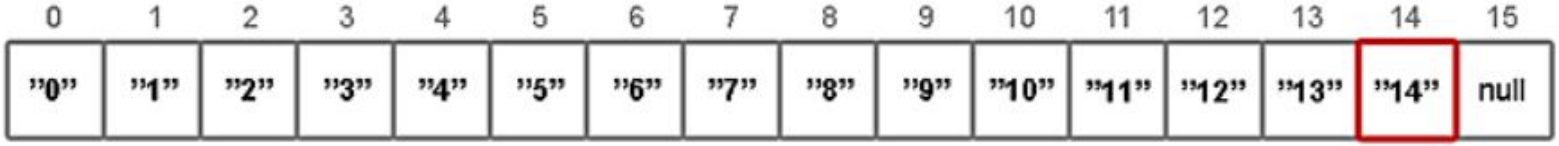


Вставка в середину

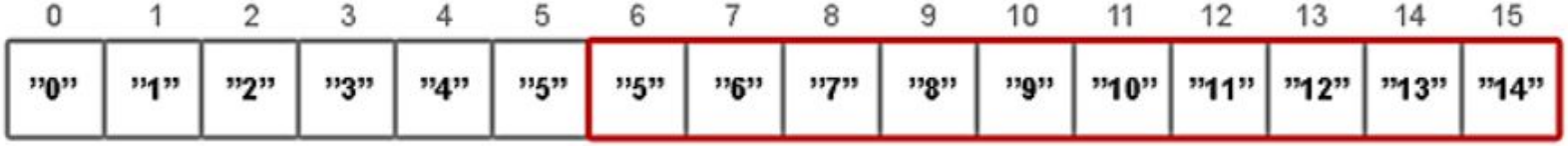
```
list.add(5, "100");
```

Добавление элемента на позицию с определенным индексом происходит в три этапа:

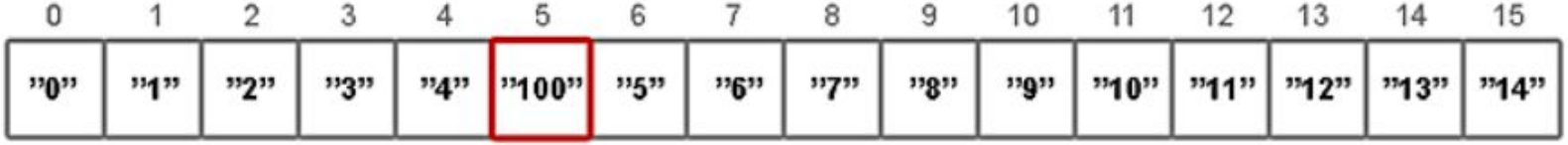
1) проверяется, достаточно ли места в массиве для вставки нового элемента;



2) подготавливается место для нового элемента с помощью System.arraycopy();



3) перезаписывается значение у элемента с указанным индексом.



Удаление

1. Удаление элемента
2. Копирование массива на один элемент влево чтобы не было пустых мест

- Количество элементов в массиве не меняется, чтобы сократить до текущего количества элементов надо вызвать trimToSize()
- При удалении по значению идет перебор всех элементов в цикле
- При удалении элемента по значению удаляется только первый встречный

Структура LinkedList

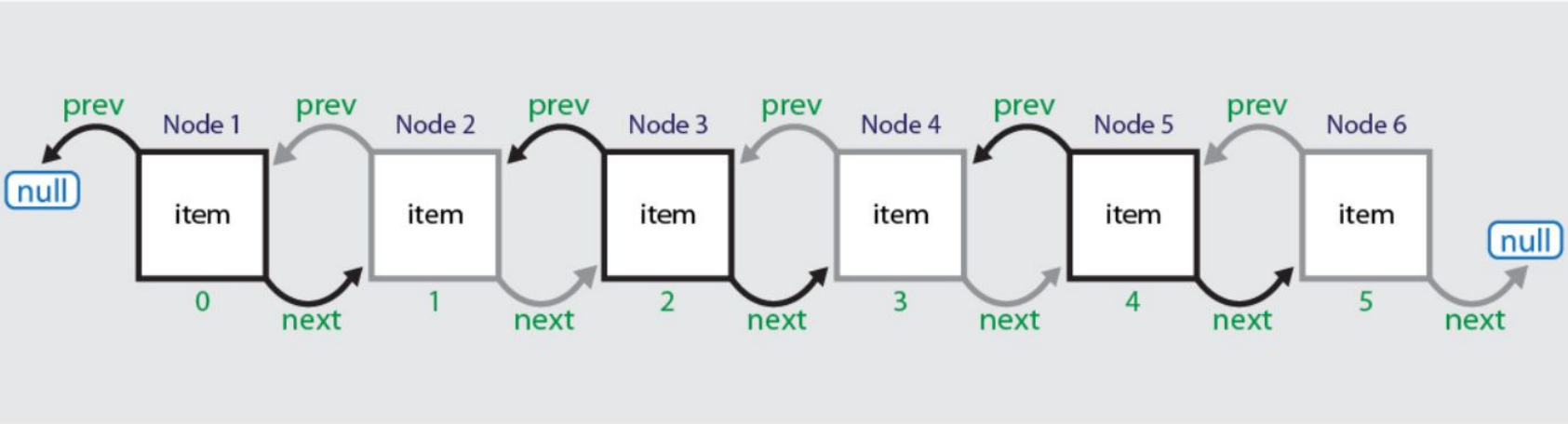
```
int size = 0;

Node<E> first;

Node<E> last;

public LinkedList(){}  // Пустой конструктор

LinkedList(Collection<? extends E> c) // Конструктор с другой коллекцией
```

- LinkedList хранит в себе объект Node, который хранит в себе бизнес-сущность
- Двусвязный список

Объект Node

```
private static class Node<E> {
    E item; // Бизнес-сущность
    Node<E> next; // Ссылка на следующий элемент
    Node<E> prev; // Ссылка на предыдущий элемент

    // Конструктор для ноды
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Структура LinkedList



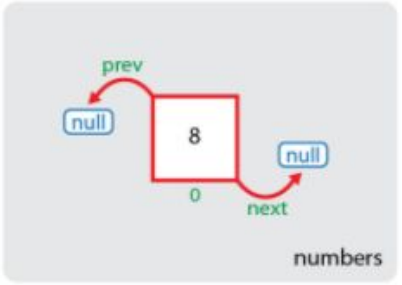
Добавление элемента в LinkedList

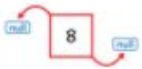
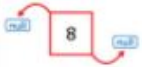
Пустой LinkedList
после созданий

size	0
first	null
last	null

---

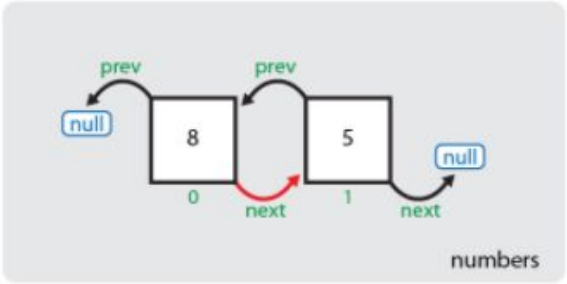
numbers.add(8);

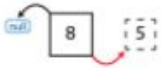


size	1
first	
last	

---

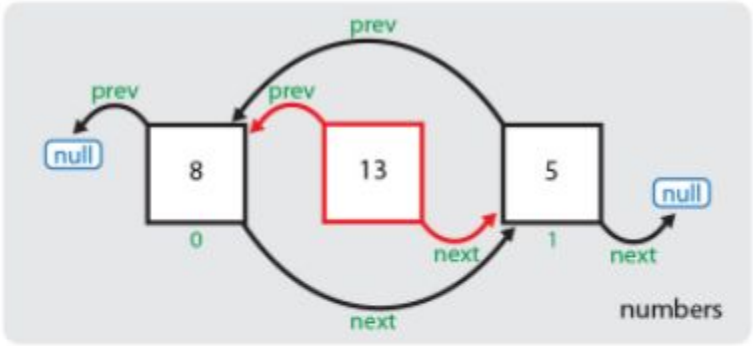
numbers.add(5);



size	2
first	
last	

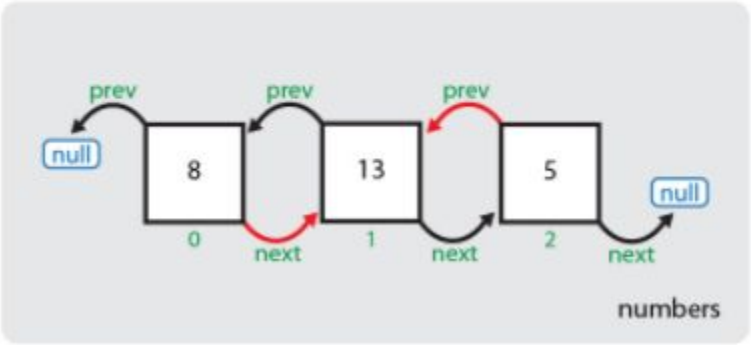
Добавление элемента в середину

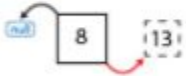
Создается объект



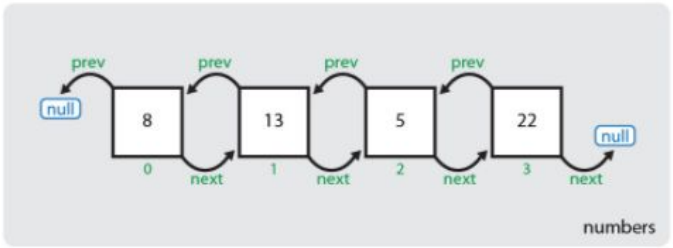
size	2
first	
last	

Соседние ноды меняют ссылки на новый объект

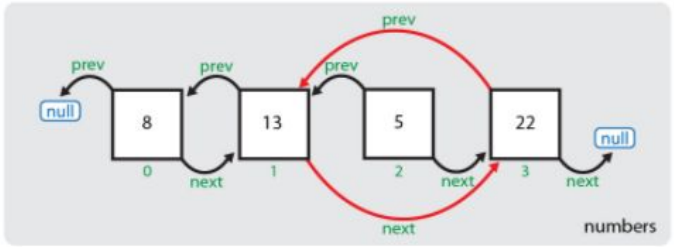


size	3
first	
last	

Удаление элемента из LinkedList

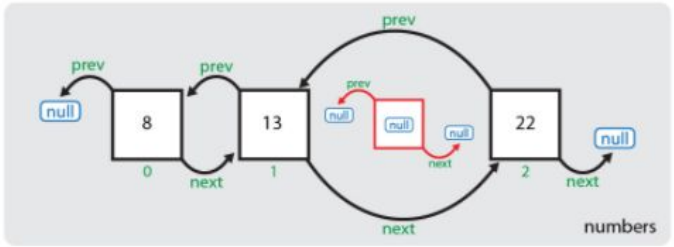


size	4
first	null 8 13
last	5 22 null



E element = 5

size	4
first	null 8 13
last	5 22 null



E element = 5

size	3
first	null 8 13
last	13 22 null

Дополнительные методы интерфейса LinkedList

```
public E getFirst() // Получить первый элемент
public E getLast() // Получить последний элемент

public E removeFirst() // Удалить первый элемент (и вернуть его)
public E removeLast() // Удалить последний элемент (и вернуть его)
```

Сложности операций

	Структура данных	Временная сложность								Сложность по памяти
		В среднем				В худшем				В худшем
		Индексация	Поиск	Вставка	Удаление	Индексация	Поиск	Вставка	Удаление	
	Обычный массив	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
ArrayList -->	Динамический массив	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Односвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
LinkedList -->	Двусвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Список с пропусками	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
HashMap -->	Хеш таблица	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
	Бинарное дерево поиска	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Декартово дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
	Б-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
TreeMap -->	Красно-черное дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Расширяющееся дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
	АВЛ-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Вывод по использованию

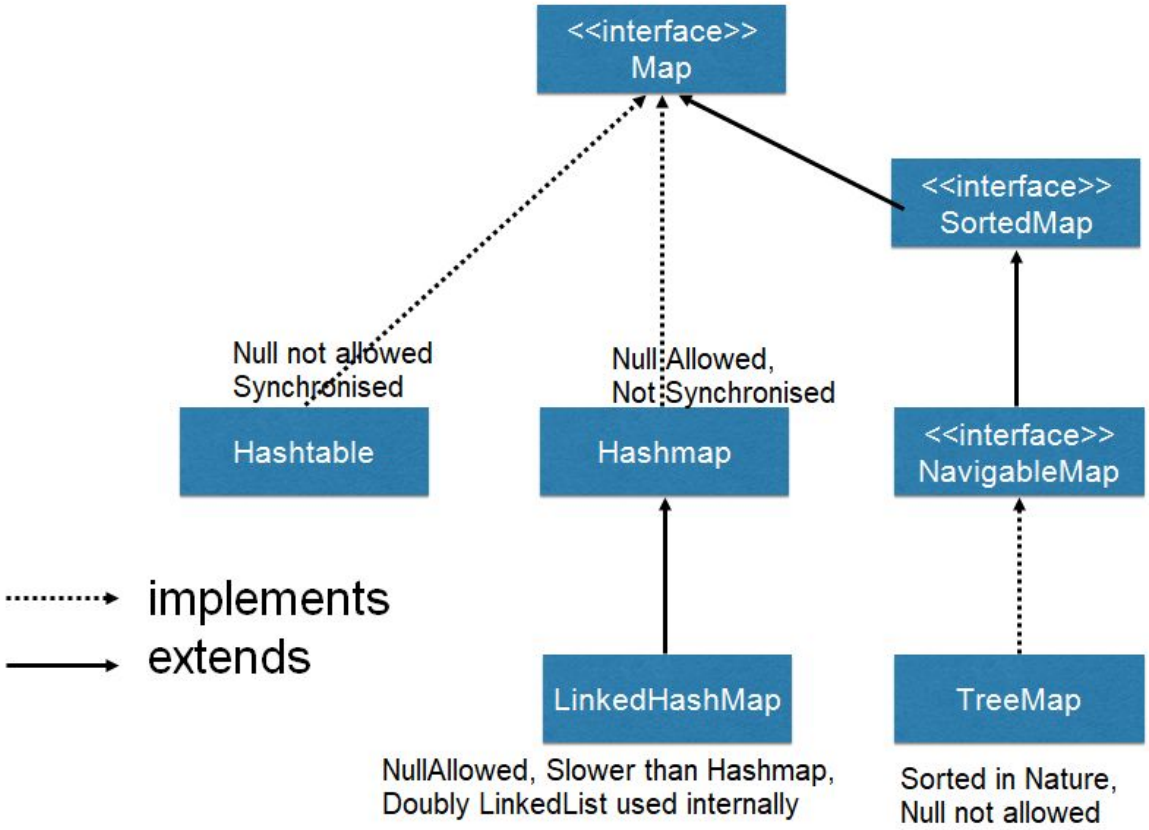
ArrayList

- Быстрее при поиске объекта (по индексу)
- Быстрее при вставке в середину за счет доступа по индексу

LinkedList

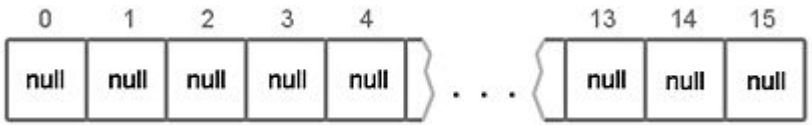
- Быстрее при вставке в начало за счет переписывания ссылок
- Быстр и удобен при работе с началом и концом списка с использованием специальных методов

Map Interface



Структура HashMap

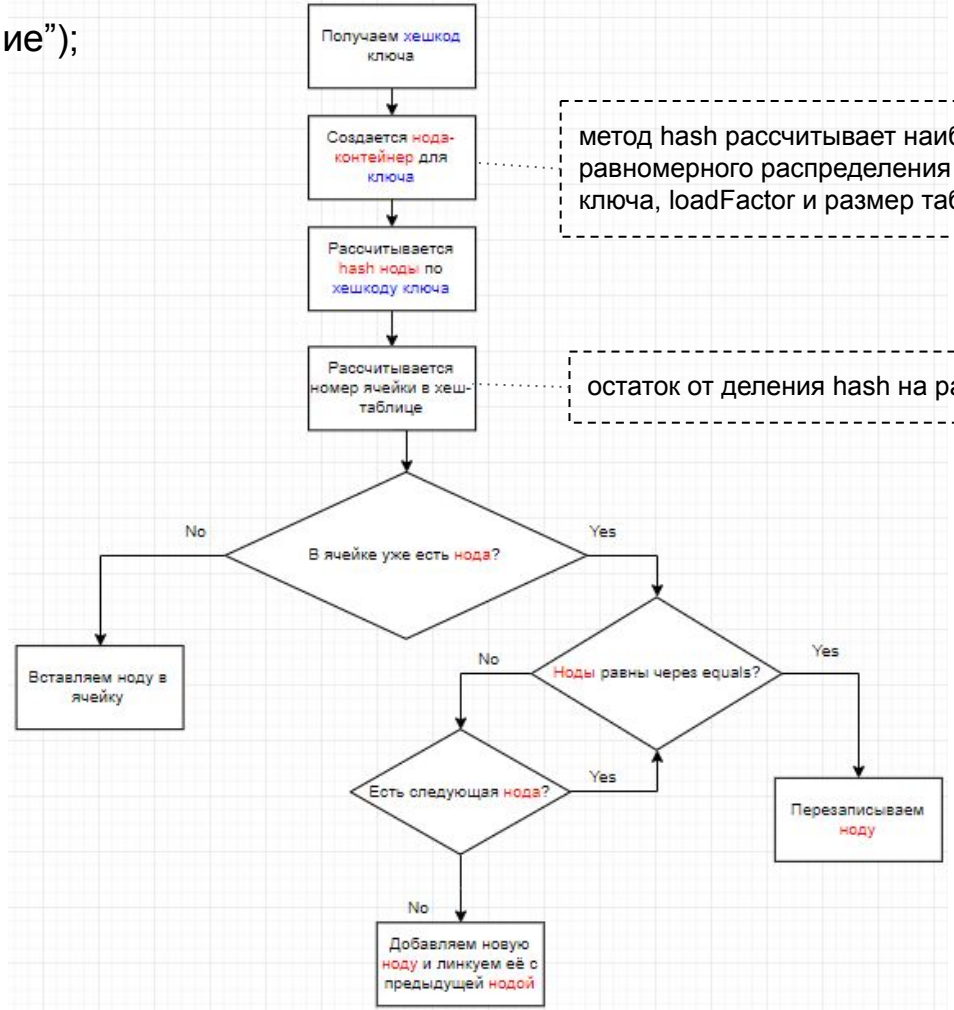
```
Node<K,V>[] table; // хеш-таблица
int size; // Размер хеш-таблицы
int threshold; // Кол-во эл-тов когда пора увеличивать таблицу
float loadFactor; // Коэффициент нагрузки таблицы
Set<Map.Entry<K,V>> entrySet; // готовый к использованию набор key-value


float DEFAULT_LOAD_FACTOR = 0.75f;
int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

HashMap()
HashMap(int initialCapacity, float loadFactor)
HashMap(int initialCapacity)
HashMap(Map<? extends K, ? extends V> m)
```



Структура Node

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash; // хеш для лушчего распределения в таблице
        final K key; // ключ
        V value; // значение
        Node<K,V> next; // ссылка на следующую ноду

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
        public final int hashCode() // хешкод ноды
        public final boolean equals(Object o)
}
```

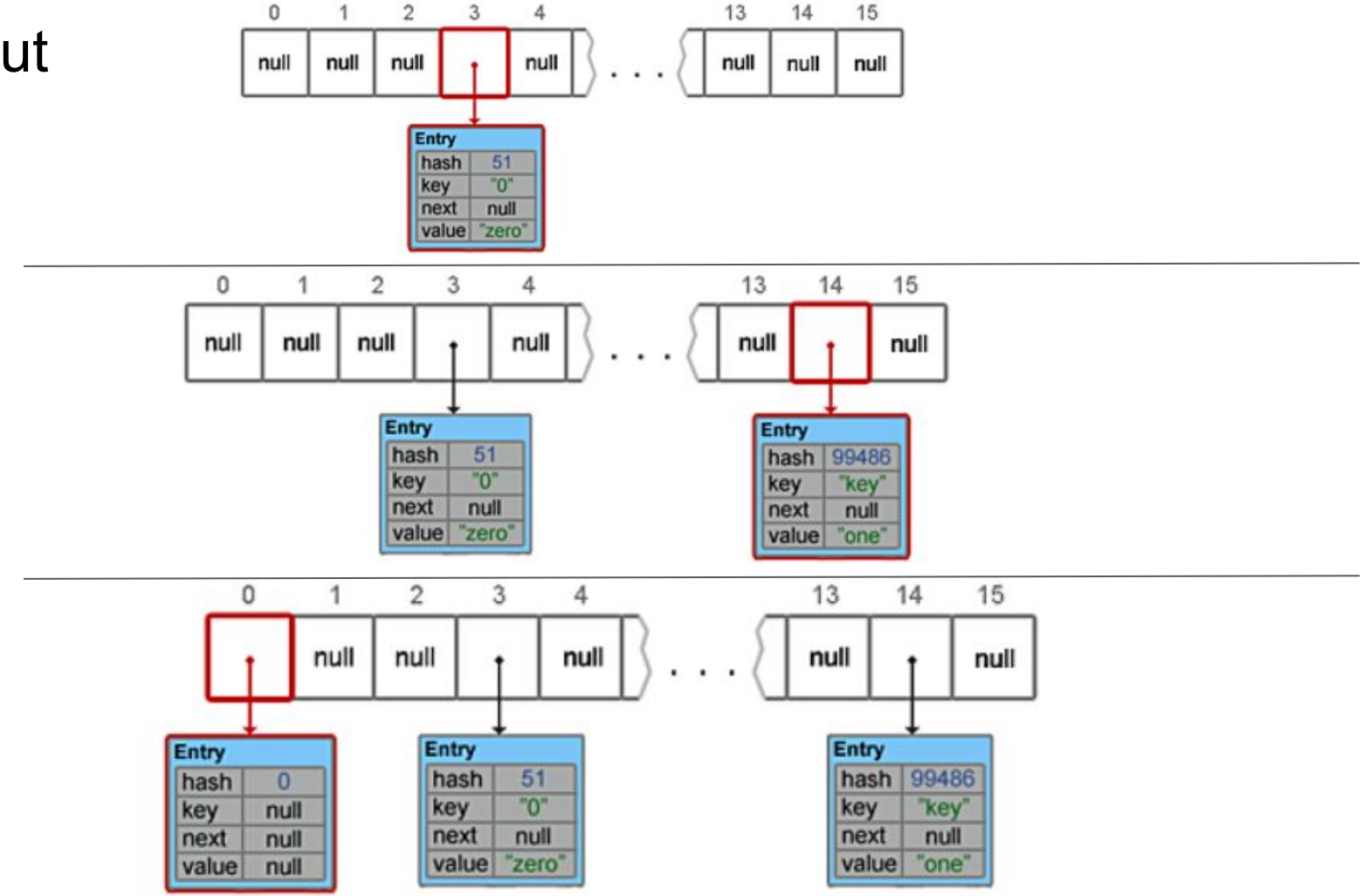
put(“ключ”, “значение”);



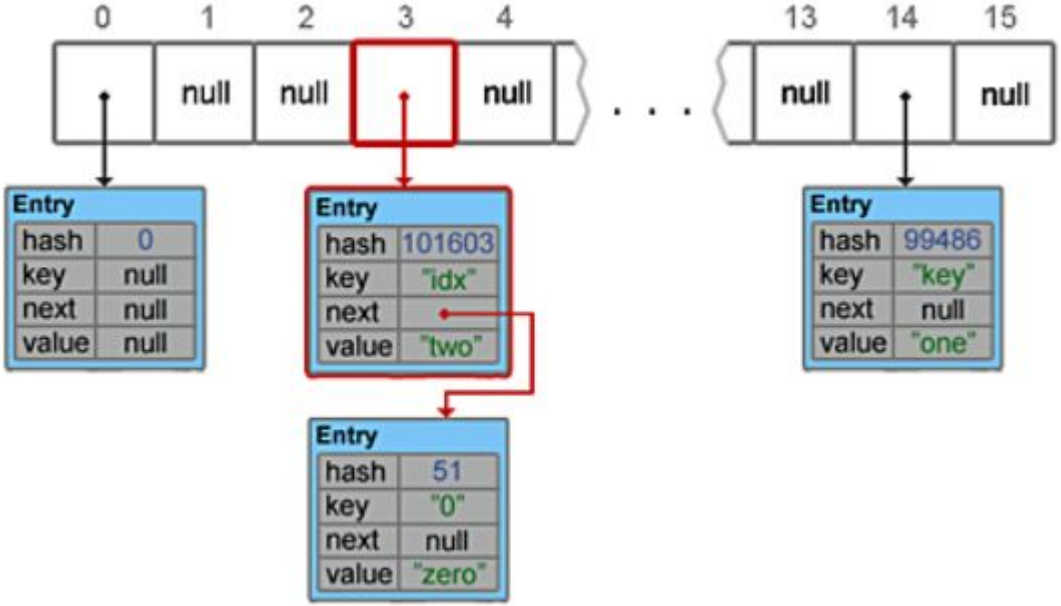
метод hash рассчитывает наиболее оптимальное значение для равномерного распределения в таблице (учитывает хешкод ключа, loadFactor и размер таблицы)

остаток от деления hash на размер таблицы

put


0 1 2 3 4 13 14 15
null null null null null null null null
Entry
hash 51
key "0"
next null
value "zero"
0 1 2 3 4 13 14 15
null null null null null null null
Entry
hash 51
key "0"
next null
value "zero"
Entry
hash 99486
key "key"
next null
value "one"
0 1 2 3 4 13 14 15
null null null null null null
Entry
hash 0
key null
next null
value null
Entry
hash 51
key "0"
next null
value "zero"
Entry
hash 99486
key "key"
next null
value "one"

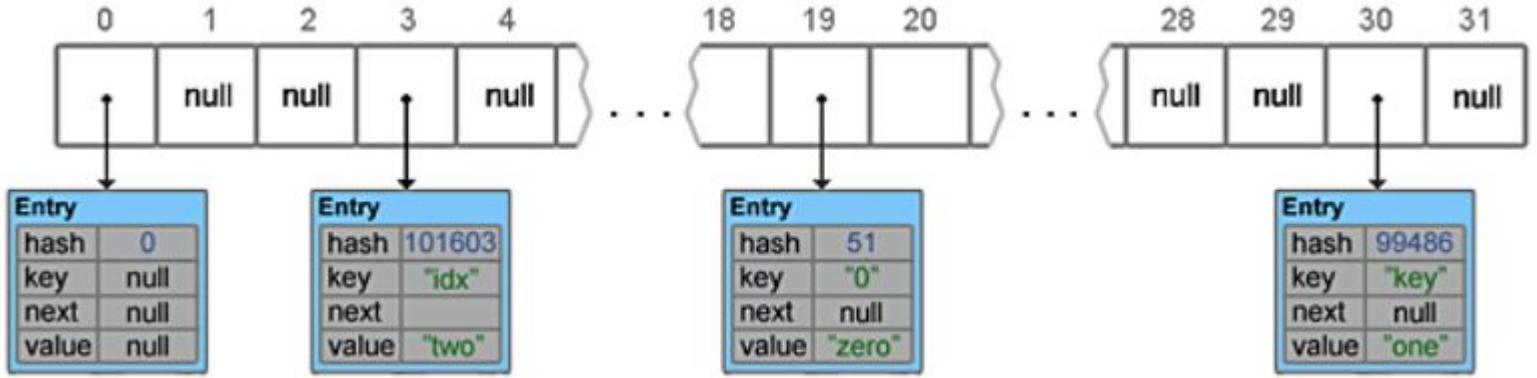
put когда hash ноды одинаковый



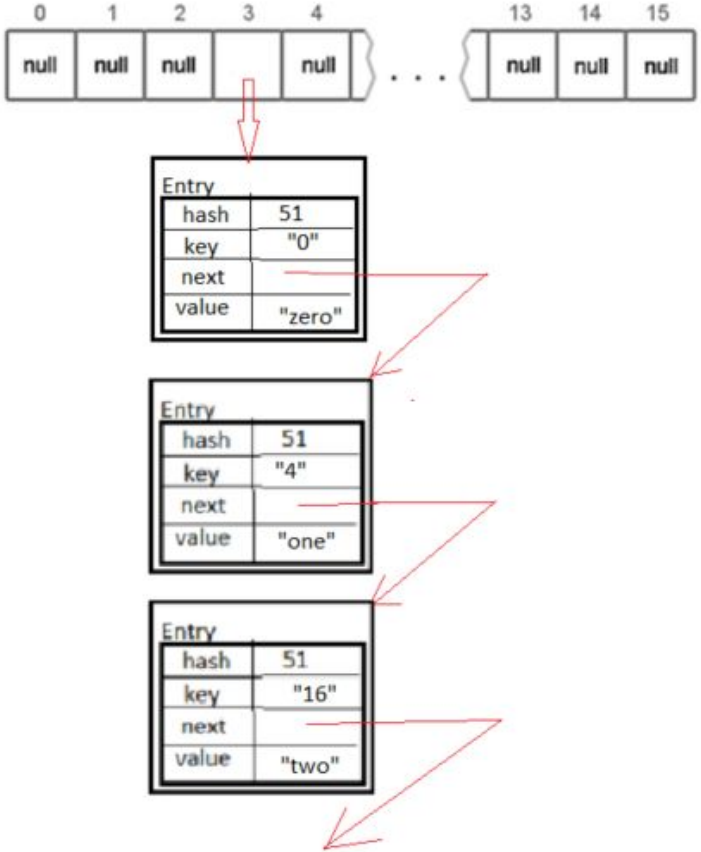
После заполнения loadFactor*size

Когда количество элементов в хеш-таблице равно порогу - таблица увеличивается вдвое и происходит перераспределение всех элементов

16*0.75 = 12



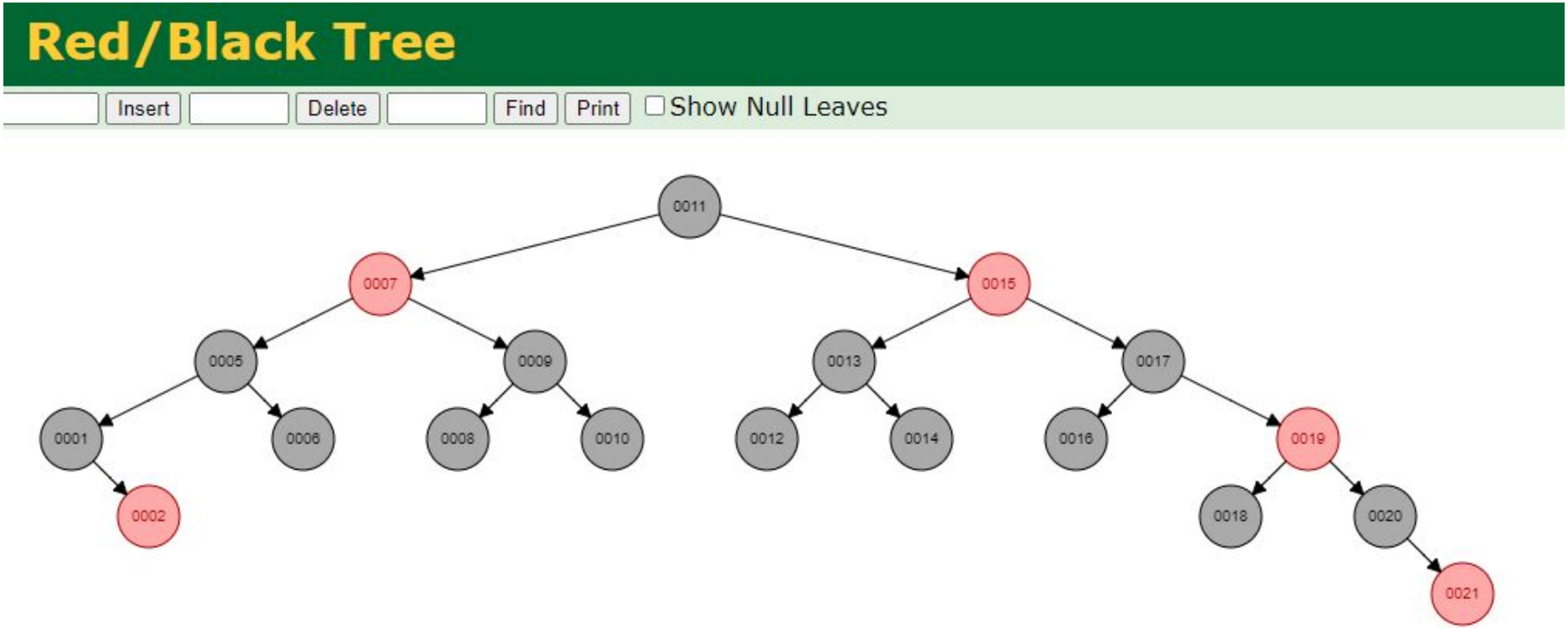
Что будет если hashcode неправильно переопределить?



HashSet

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}

private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();

/**
 * Constructs a new, empty set; the backing <tt>HashMap</tt> instance has
 * default initial capacity (16) and load factor (0.75).
 */
public HashSet() { map = new HashMap<>(); }
```

TreeMap



Сложность поиска log(n)

Hash vs Tree

Hash

- Работает быстрее (при правильном переопределении hashcode и equals) за счет хеширования

Tree

- Есть сортировка (Можно описать свою через Comparator)