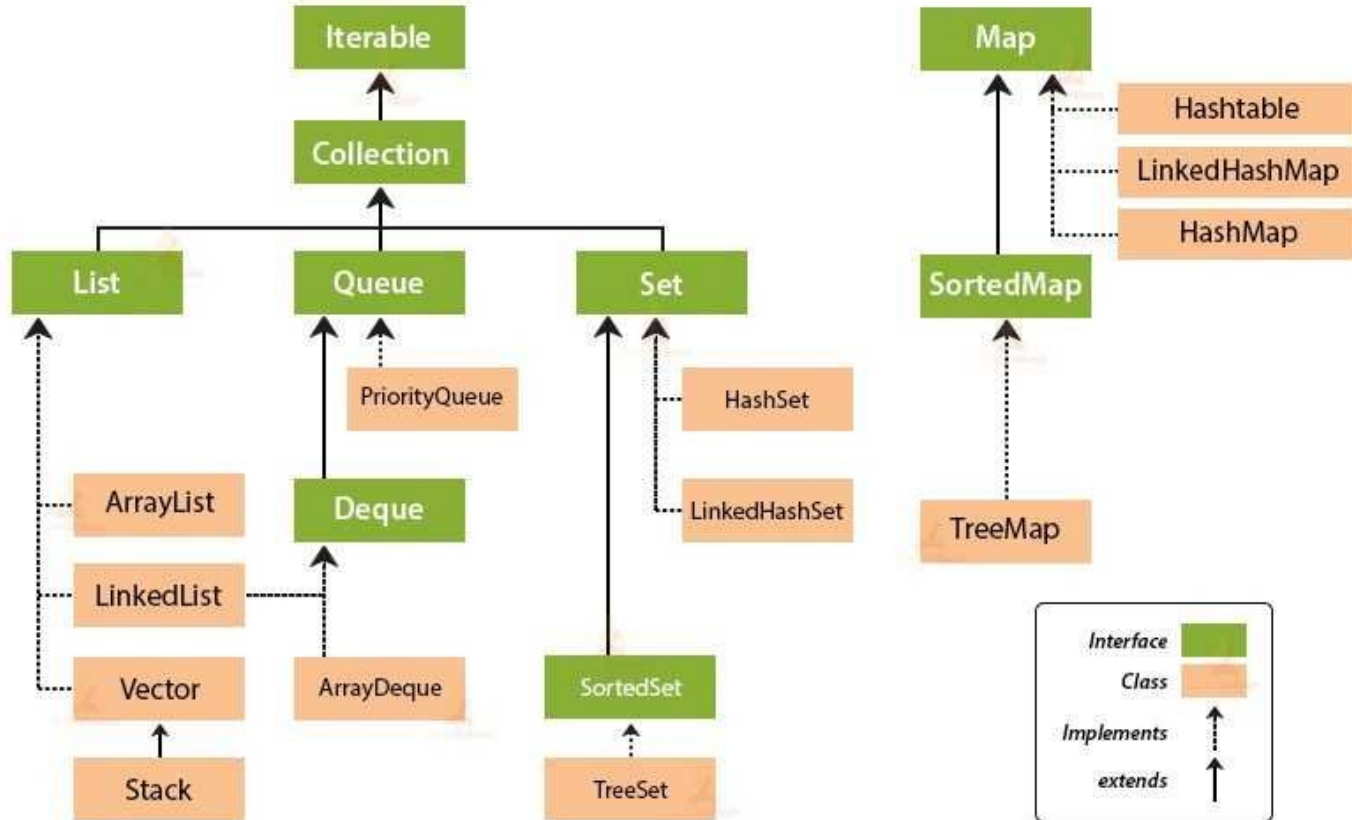
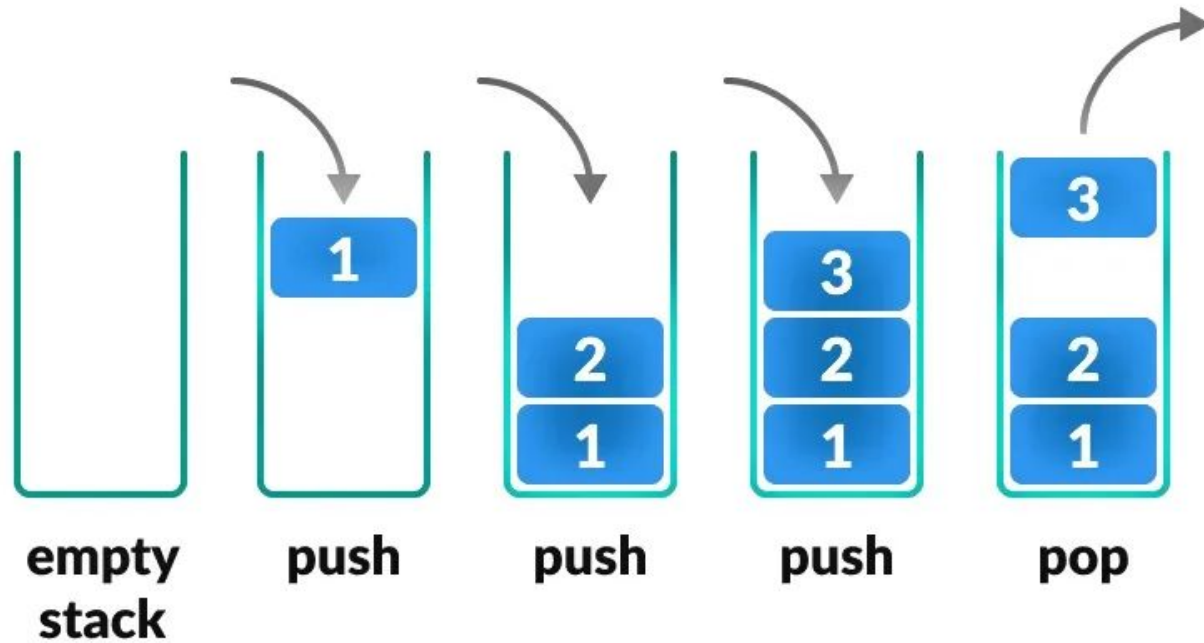


Core Java 7

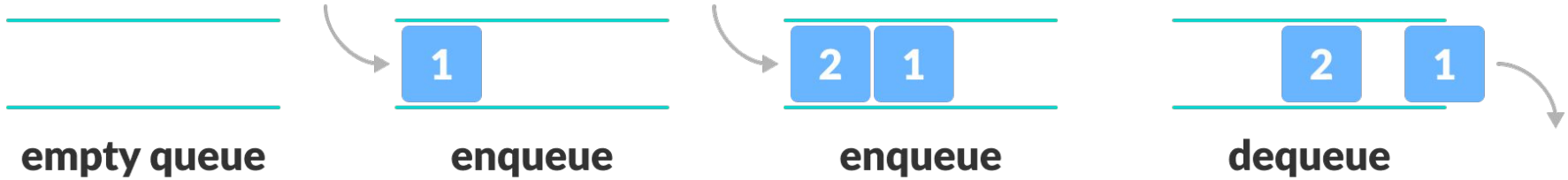
Collection Framework Hierarchy in Java



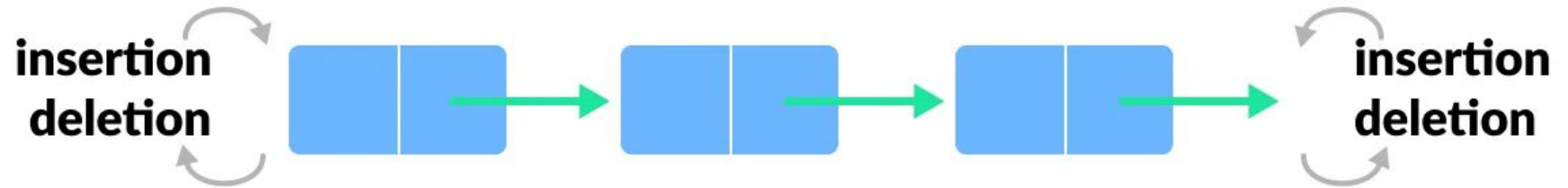
Stack - LIFO



Queue - FIFO



Deque



List

```
public interface List<E> {  
  
    void add(int index, E el);  
  
    E get(int index);  
  
    void remove(int index);  
}
```

Iterable<E> и Iterator<E>

- ✓ **Iterable** (обычно – кроме `DirectoryStream!`) можно обходить много раз, вызывая `iterator()` повторно
- ✓ **Iterator** обходится ровно один раз, вперёд:
 - ✓ **hasNext()**: true, если ещё есть элемент, false если нет, идемпотентна
 - ✓ **next()**: следующий элемент или `NoSuchElementException()`
 - ✓ **remove()**: удалить последний полученный через `next()` элемент или `UnsupportedOperationException()`
- ✓ **Iterable** может быть неизменяемым, **Iterator** – нет (разве что если пустой)

Collection<E> extends Iterable<E>

- ✓ `int size();`
- ✓ `boolean isEmpty();`
- ✓ `boolean contains(Object o);`
- ✓ `boolean containsAll(Collection<?> c);`
- ✓ `Object[] toArray();`
- ✓ `<T> T[] toArray(T[] a);`
- ✓ `boolean add(E e);`
- ✓ `boolean remove(Object o);`
- ✓ `boolean addAll(Collection<? extends E> c);`
- ✓ `boolean removeAll(Collection<?> c);`
- ✓ `boolean retainAll(Collection<?> c);`
- ✓ `void clear();`

Set<E> extends Collection<E>

- ✓ Не содержит новых методов
- ✓ Содержит уточнённый контракт
- ✓ Не может содержать сам себя
- ✓ Если вы меняете изменяемый объект, который лежит в множестве, то сами виноваты
- ✓ Сравним с любым другим Set по содержимому

List<E> extends Collection<E>

- ✓ `boolean addAll(int index, Collection<? extends E> c);`
- ✓ `void sort(Comparator<? super E> c);`
- ✓ `E get(int index);`
- ✓ `E set(int index, E element);`
- ✓ `void add(int index, E element);`
- ✓ `E remove(int index); // boolean remove(Object o);`
- ✓ `int indexOf(Object o);`
- ✓ `int lastIndexOf(Object o);`
- ✓ `ListIterator<E> listIterator();`
- ✓ `ListIterator<E> listIterator(int index);`
- ✓ `List<E> subList(int fromIndex, int toIndex);`

Comparable<T> и Comparator<T>

```
public interface Comparable<T>{  
    /**  
     * @param o the object to be compared.  
     * @return a negative integer, zero, or a positive  
     *         is less than, equal to, or greater than  
     */  
    int compareTo(T o);  
}
```

*/*Применяется в случае, если сравниваемые объекты не реализуют Comparable*/*

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Comparable<T>

(естественный порядок)

✓ **public int** *compareTo*(**T** o);

✓ $< 0 \rightarrow this < o$

✓ $== 0 \rightarrow this == o$

✓ $> 0 \rightarrow this > o$

Естественный порядок

```
class User implements Comparable<User> {  
    final String name;  
  
    User(String name) { this.name = name; }  
  
    @Override  
    public int compareTo(@NotNull User o) {  
        return name.compareTo(o.name);  
    }  
}
```

Контракт compareTo

- ✓ $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$
- ✓ $x.\text{compareTo}(y)$ throws $\Leftrightarrow y.\text{compareTo}(x)$ throws
- ✓ $x.\text{compareTo}(\text{null})$ throws `NullPointerException`
- ✓ $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0) \Rightarrow x.\text{compareTo}(z) > 0$
- ✓ $x.\text{compareTo}(y) == 0 \Rightarrow \forall z: \text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$
- ✓ (рекомендуется) $x.\text{compareTo}(y) == 0 \Leftrightarrow x.\text{equals}(y)$
- ✓ (рекомендуется) $x.\text{compareTo}(y) \neq \text{Integer.MIN_VALUE}$

```
static class User implements Comparable<User> {  
    private boolean valid;  
  
    User(boolean valid) { this.valid = valid; }  
  
    boolean isValid() { return valid; }  
  
    public int compareTo(User o) {  
        ✖ return this.valid && !o.valid ? 1 : -1;  
    }  
  
    public String toString() { return String.valueOf(valid); }  
}
```

```
Set<User> set = new TreeSet<>();  
User u = new User(true);  
set.add(u);  
set.add(u);  
System.out.println(set);
```

[true, true]


```
static class User implements Comparable<User> {  
    private boolean valid;  
  
    User(boolean valid) { this.valid = valid; }  
  
    boolean isValid() { return valid; }  
  
    public int compareTo(@NotNull User o) {  
        return Boolean.compare(this.valid, o.valid);  
    }  
  
    public String toString() { return String.valueOf(valid); }  
}
```



```
static class User implements Comparable<User> {  
    private int age;
```

```
    User(int age) { this.age = age; }
```

```
    public int getAge() { return age; }
```

```
    public int compareTo(@NotNull User o) {  
 return this.age - o.age;  
}
```

```
    public String toString() { return String.valueOf(age); }  
}
```

```
Set<User> set = new TreeSet<>();  
set.add(new User(1_000_000_000));  
set.add(new User(2_000_000_000));  
set.add(new User(0));  
set.add(new User(-1_000_000_000));  
set.add(new User(-2_000_000_000));  
System.out.println(set);
```

```
[-1000000000, 0, 1000000000, 2000000000, -2000000000]
```

```
static class User implements Comparable<User> {  
    private int age;  
  
    User(int age) { this.age = age; }  
  
    public int getAge() { return age; }  
  
    public int compareTo(User o) {  
        return this.age < o.age ? -1 : this.age == o.age ? 0 : 1;  
    }  
  
    public String toString() { return String.valueOf(age); }  
}
```

```
static class User implements Comparable<User> {  
    private int age;  
  
    User(int age) { this.age = age; }  
  
    public int getAge() { return age; }  
  
    public int compareTo(User o) {  
        return Integer.compare(this.age, o.age);  
    }  
  
    public String toString() { return String.valueOf(age); }  
}
```

```
static class User implements Comparable<User> {  
    private final double income;  
  
    User(double income) { this.income = income; }  
  
    public String toString() { return String.valueOf(income); }  
  
    @Override  
    public int compareTo(User o) {  
        return income < o.income ? -1 : income == o.income ? 0 : 1;  
    }  
}
```

```
List<User> list = new ArrayList<>();  
list.add(new User(Double.NaN));  
list.add(new User(20));  
list.add(new User(Double.NaN));  
list.add(new User(10));  
list.add(new User(Double.NaN));  
list.add(new User(0));  
list.add(new User(Double.NaN));  
list.add(new User(30));  
list.sort(null);  
System.out.println(list);
```

```
[NaN, 20.0, NaN, 10.0, NaN, 0.0, NaN, 30.0]
```

Comparator<T>

✓ `int compare(T o1, T o2);`

✓ `<0` → `o1 < o2`

✓ `==0` → `o1 == o2`

✓ `>0` → `o1 > o2`

Контракт compare

Такой же, как и в compareTo за исключением сравнения null (Тут можно, к примеру null меньше любого объекта)


```
static class User {  
    private final String name;  
    private final int age;  
  
    User(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
    public int getAge() { return age; }  
  
    public String toString() { return name+": "+age; }  
}
```

```
// lambda!  
static final Comparator<User> USER_COMPARATOR = (u1, u2) -> {  
    int res = u1.getName().compareTo(u2.getName());  
    return res == 0 ? Integer.compare(u1.getAge(), u2.getAge()) : res;  
};
```

Combinator

```
static final Comparator<User> USER_COMPARATOR =  
    Comparator.comparing((User u) -> u.getName())  
                .thenComparingInt(u -> u.getAge());  
  
    Comparator.comparing((User u) -> u.getName(), Comparator.reverseOrder())  
                .thenComparingInt(u -> u.getAge());  
  
    Comparator.comparing((User u) -> u.getName(), String.CASE_INSENSITIVE_ORDER)  
                .thenComparingInt(u -> u.getAge());  
  
    Comparator.nullsFirst(Comparator.comparing((User u) -> u.getName())  
                            .thenComparingInt(u -> u.getAge()));  
  
    Comparator.comparing((User u) -> u.getName(),  
                          Comparator.nullsFirst(Comparator.naturalOrder()))  
                .thenComparingInt(u -> u.getAge());
```

Combinator + static import + method reference

```
static final Comparator<User> USER_COMPARATOR =  
    comparing(User::getName).thenComparingInt(User::getAge);  
  
    comparing(User::getName, reverseOrder()).thenComparingInt(User::getAge);  
  
    comparing(User::getName, String.CASE_INSENSITIVE_ORDER)  
        .thenComparingInt(User::getAge);  
  
    nullsFirst(comparing(User::getName).thenComparingInt(User::getAge);  
  
    comparing(User::getName, nullsFirst(naturalOrder()))  
        .thenComparingInt(User::getAge);
```

Промежуточные итоги

- Используйте интерфейсы как типы переменных или аргументы в методах
- Помимо возможности подмены реализаций существуют суррогатные коллекции

//ИММУТАБЕЛЬНЫЕ

//Пустые

```
Collections.emptyList();  
Collections.emptySet();  
Collections.emptyMap();
```

//Из одного элемента

```
Collections.singletonList(o);  
Collections.singleton(o);  
Collections.singletonMap(k,v);
```

Стандартные списки

- ✓ **ArrayList** – изменяемый список общего назначения
- ✓ **Arrays.asList** – изменяемая обёртка над массивом
- ✓ **Collections.emptyList()** – неизменяемый пустой список
- ✓ **Collections.singletonList(x)** – неизменяемый список из одного элемента
- ✓ **Collections.nCopies(n, x)** – неизменяемый список из n одинаковых элементов
- ✓ **List.of(...)** – (Java 9) неизменяемый список из указанных элементов (или массива), null не приемлет
- ✓ **List.copyOf(...)** – (Java 10) неизменяемая копия указанного списка
- ✓ **Collections.unmodifiableList(list)** – неизменяемая обёртка над списком
- ✓ **Collections.synchronizedList(list)** – синхронизированная обёртка над списком
- ✓ **Collections.checkedList(list, type)** – проверяемая обёртка

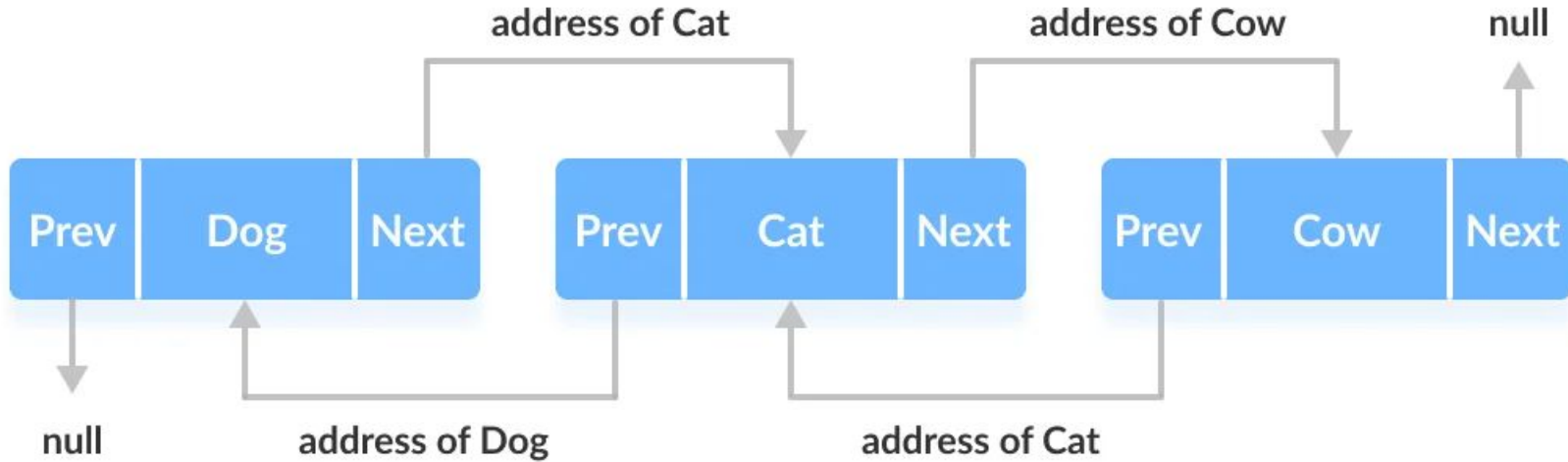
Стандартные множества

- ✓ **HashSet** – изменяемое неупорядоченное множество общего назначения
- ✓ **LinkedHashSet** – изменяемое упорядоченное (ordered) множество общего назначения
- ✓ **TreeSet** – изменяемое сортированное (sorted) множество
- ✓ **EnumSet** – изменяемое множество элементов enum
- ✓ **Collections.emptySet()** – неизменяемое пустое множество
- ✓ **Collections.singleton(x)** – неизменяемое множество из одного элемента
- ✓ **Set.of(...)** – (Java 9) неупорядоченное неизменяемое множество заданных элементов (без null и повторов)
- ✓ **Set.copyOf(...)** – (Java 10) неизменяемая копия
- ✓ **Collections.unmodifiableSet(set)** – неизменяемая обёртка над множеством
- ✓ **Collections.synchronizedSet(set)** – синхронизированная обёртка над множеством
- ✓ **Collections.checkedSet(set, type)** – проверяемая обёртка

Свойства ArrayList<E>

- `get(int index)` is **$O(1)$** ← **main benefit of ArrayList<E>**
- `add(E element)` is **$O(1)$** amortized, but **$O(n)$** worst-case since the array must be resized and copied
- `add(int index, E element)` is **$O(n)$** (with **$n/2$** steps on average)
- `remove(int index)` is **$O(n)$** (with **$n/2$** steps on average)
- `Iterator.remove()` is **$O(n)$** (with **$n/2$** steps on average)
- `ListIterator.add(E element)` is **$O(n)$** (with **$n/2$** steps on average)

Свойства LinkedList<E>



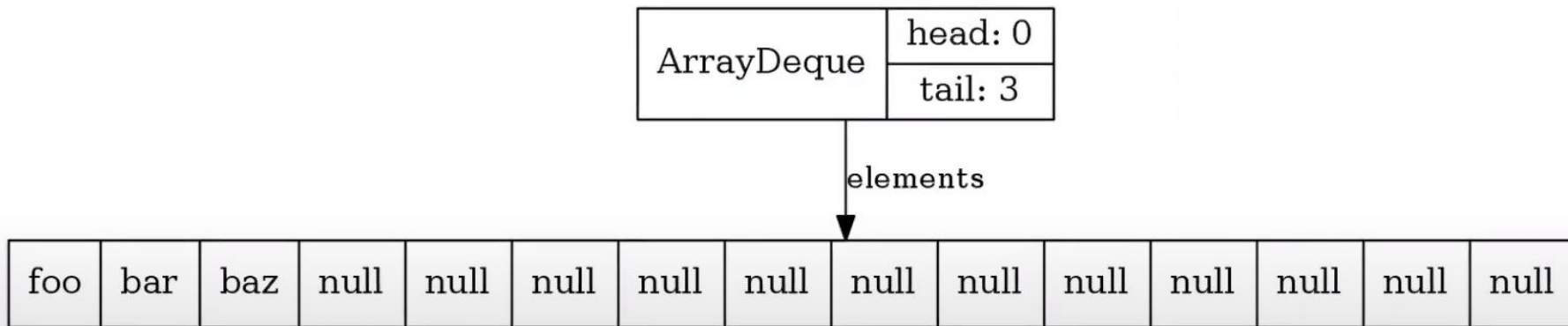
LinkedList Implementation in Java

Свойства LinkedList<E>

- Implements List and Deque
- `get(int index)` is **$O(n)$** (with $n/4$ steps on average)
- `add(E element)` is **$O(1)$**
- `add(int index, E element)` is **$O(n)$** (with $n/4$ steps on average), but **$O(1)$** if `index = 0` ← **main benefit of LinkedList<E>**
- `remove(int index)` is **$O(n)$** (with $n/4$ steps on average)
- `Iterator.remove()` is **$O(1)$** ← **main benefit of LinkedList<E>**
- `ListIterator.add(E element)` is **$O(1)$** This is one of the main benefits of LinkedList<E>

Если нужен Deque

- ArrayDeque
- Circular array
- Более быстрый, чем LinkedList.



PriorityQueue

- Постановка в очередь с сортировкой по приоритету за счёт Comparable Comparator.
- **Balanced binary heap:** "the two children of `queue[n]` are `queue[2*n+1]` and `queue[2*n+2]` "

```
PriorityQueue<String> q = new PriorityQueue<>();  
q.add("foo"); q.add("bar"); q.add("baz");
```

