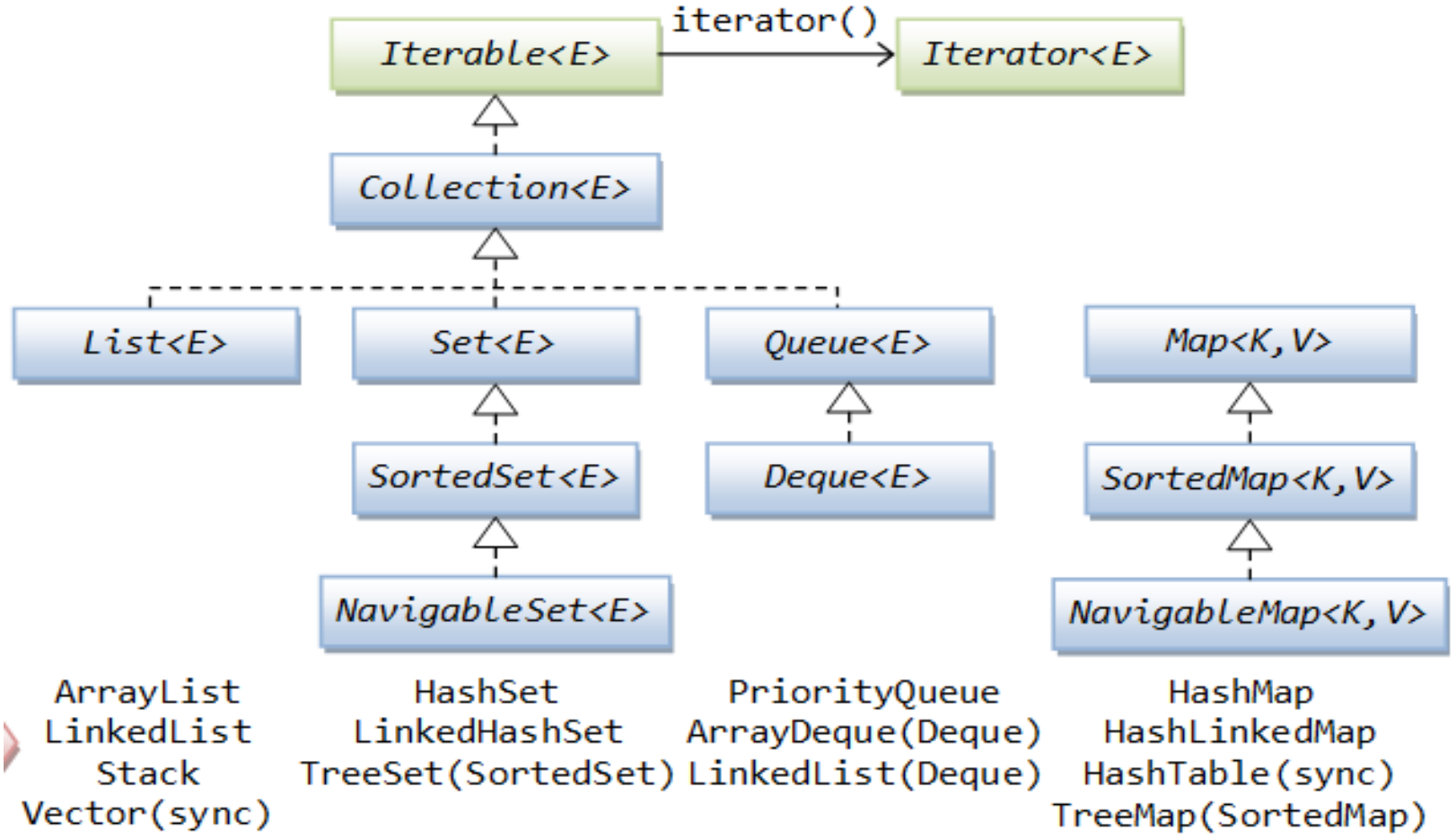
Фреймфорк коллекций Java

Фреймворк коллекций Java

- ***Фреймворк коллекций*** представляет собой набор интерфейсов, которые принуждают пользователя принимать конкретные построение и поведение наборов данных.
- ***Фреймворк коллекции*** обеспечивает унифицированный интерфейс для хранения, извлечения и управления элементами наборов данных вне зависимости от фактической их реализации.
- ***Фреймворк коллекций*** находится в пакете **java.util** и содержит:
  - набор интерфейсов;
  - реализацию классов;
  - алгоритмы (такие как сортировка и поиск).

Набор интерфейсов фреймворка коллекций:



Фреймворк коллекций Java

- **Collection** – корень иерархии коллекций, который содержит методы, определяющие общее поведения для любой коллекции;

- **List** – это список, который может содержать повторяющиеся элементы и пользователь имеет полный контроль над расположением элемента в списке, а также может получить доступ к элементам по их целочисленной позиции;

- **Queue** – это коллекция, которая используется для хранения элементов до их обработки и предоставляет дополнительные операции вставки, извлечения и управления. Типично для **Queue** располагать элементы по правилу FIFO (первый вошел, первый вышел);

Каждая реализация очереди должна указывать свои свойства упорядочивания.

Фреймворк коллекций Java

- **Deque** – это коллекция, которая представляет собой двунаправленную очередь, т.е. может использовать как *FIFO* (первый вошел, первый вышел), так и *LIFO* (последний вошел, первый вышел);

- **Set** – это коллекция, которая не может содержать повторяющиеся элементы и представляет собой модель математической абстракции – множество (такое как карты, расписание студента или процессы, запущенные на компьютере);

- **Map** – это коллекция, которая отображает ключи на значения:
  - не может содержать дубликаты ключей;
  - каждый ключ может отображать не более чем одно значение.

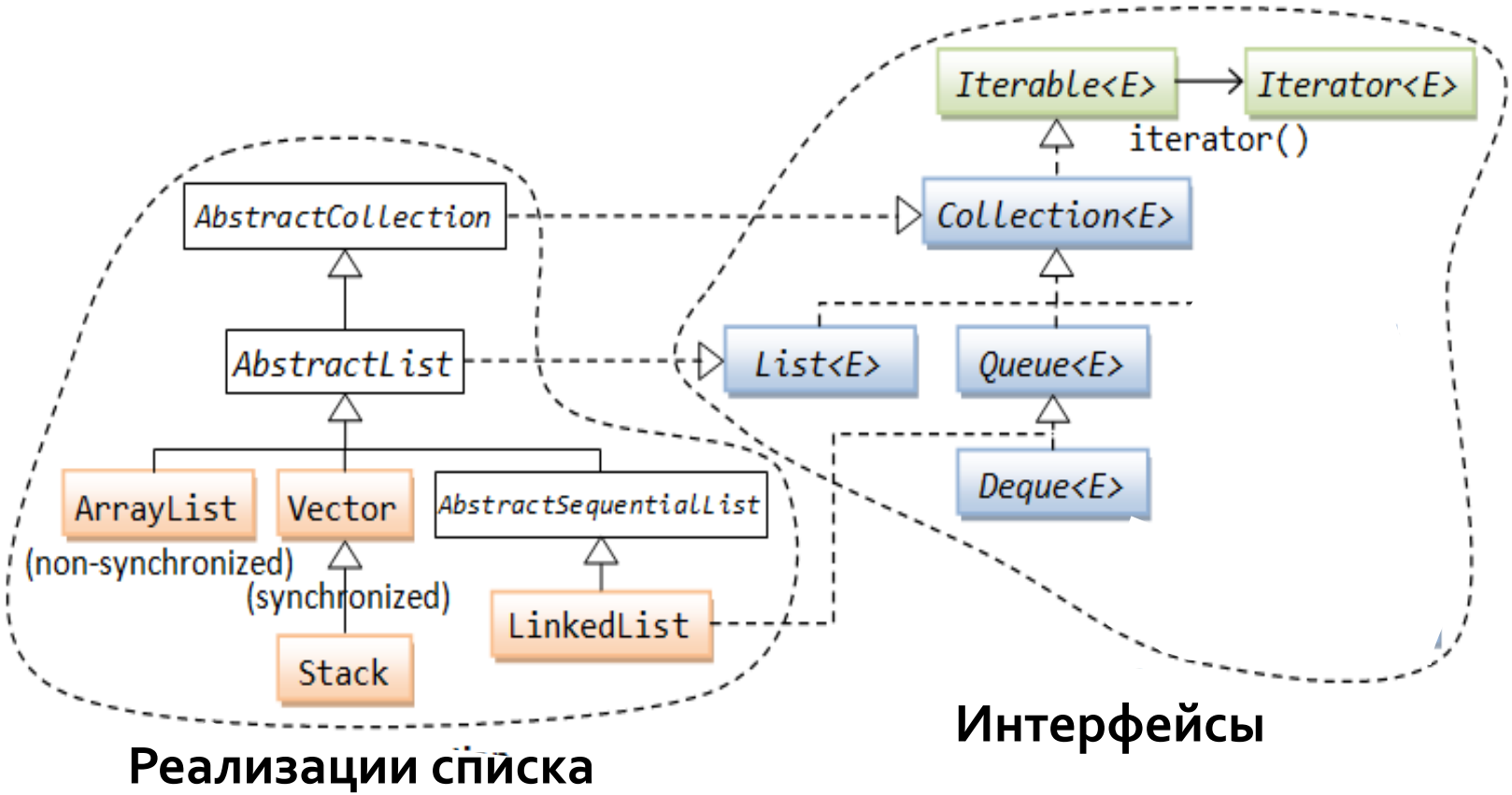
Фреймворк коллекций Java

- **SortedSet** – это *множество*, которое сохраняет свои элементы в порядке возрастания; используется для естественно упорядоченных множеств, таких как списки слов;

- **SortedMap** – это *карта*, которая сохраняет свои отображения в порядке возрастания ключа (аналог **SortedSet**); используется для естественно упорядоченных наборов пар ключ/значение, *например*, словарей и телефонных справочников.

Фреймворк коллекций Java

Пример реализации интерфейсов коллекции:



ИНТЕРФЕЙС Collection

- Является наименьшим общим знаменателем, который все коллекции реализуют.

Методы объявленные в Collection

1. boolean ***add***(Object obj); - добавить элемент к коллекции
2. void ***clear***(); - удалить все элементы
3. boolean ***contains***(Object obj); - определить существует ли объект в коллекции
4. boolean ***remove***(Object obj); - удалить элемент из коллекции
5. int ***size***(); - определить размер коллекции
6. Iterator ***iterator***(Object obj); - получить итератор для коллекции
7. Object [] ***toArray***(); - получить массив из элементов коллекции
8. boolean ***isEmpty***(); - определить пуста ли коллекция

Пример 1:

```
Collection<String> myColl = new ArrayList<>();
myColl.add("data");
myColl.add("text");
myColl.add("hello");
myColl.add("java");
System.out.println(myColl);
System.out.println(myColl.size());
myColl.clear();
System.out.println(myColl);
```

Вывод в консоли:

```
[data, text, hello, java]
4
[]
```

Пример 2:

```
Collection<String> myColl =
        new ArrayList<>();
myColl.add("data");
myColl.add("text");
myColl.add("hello");
myColl.add("java");
System.out.println(myColl);
System.out.println(myColl.remove("text"));
System.out.println(myColl.remove("abcd"));
System.out.println(myColl);
System.out.println(myColl.contains("data"));
System.out.println(myColl.contains("text"));
```

Вывод в консоли:

```
[data, text, hello, java]
true
false
[data, hello, java]
true
false
```

Пример 3, *Массовые операции*:

```
Collection<String> myColl1 = new ArrayList<>();
Collection<String> myColl2 = new ArrayList<>();
myColl1.add("data");
myColl1.add("number");
myColl1.add("lock");
myColl2.add("data");
myColl2.add("number");
System.out.println(myColl1.containsAll(myColl2));
System.out.println(myColl1.removeAll(myColl2));
System.out.println(myColl1);
```

Вывод в консоли:

```
true
true
[lock]
```

Пример 4, *Методы-мосты между коллекциями и старым API*:

```
Collection<String> myColl1 = new ArrayList<>();
myColl1.add("data");
myColl1.add("text");
myColl1.add("java");
myColl1.add("lock");
Object[] myArrObj = myColl1.toArray();
String[] myArrStr = myColl1.toArray(new String[3]);
System.out.println(Arrays.toString(myArrObj));
System.out.println(Arrays.toString(myArrStr));
```

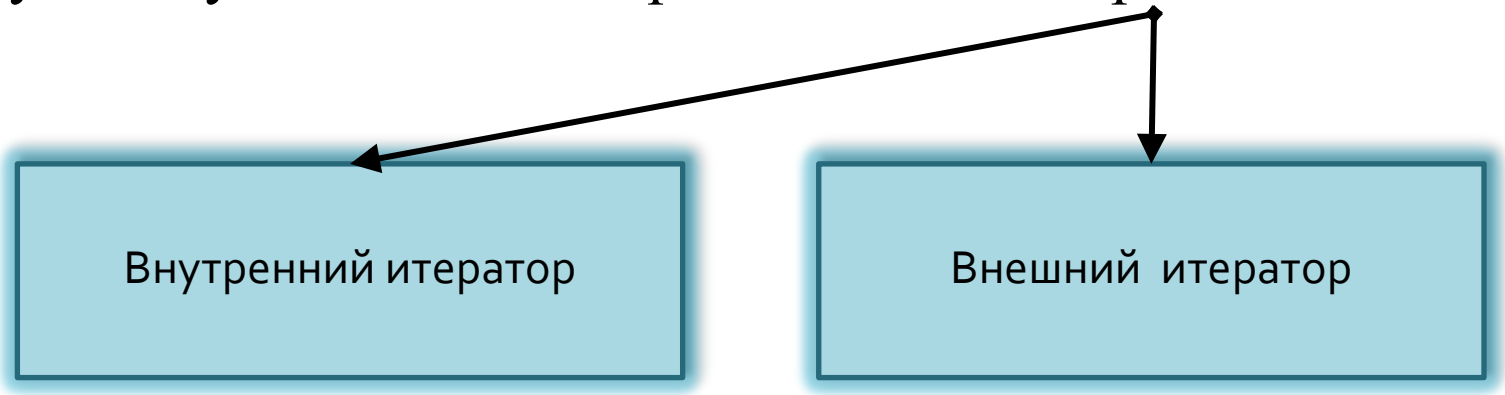
Вывод в консоли:

```
[data, text, java, lock]
[data, text, java, lock]
```

ИНТЕРФЕЙС Iterator

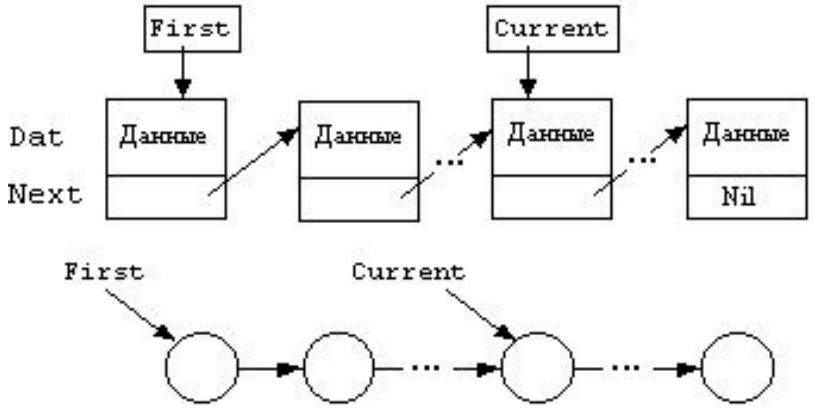
- **Итератор** – это объект, который позволяет последовательно перебирать (обходить) все элементы списка для выполнения какой-либо однотипной операции над каждым элементом.

Существует два способа реализации «Итерации»:



Внутренний итератор

- Под ним понимается собственная операция набора данных, которая в цикле проходит по всем элементам и выполняет обработку каждого элемента.



Ограничения: требуется определить обработку каждого элемента в виде отдельной структуры данных (класса) и итератору передавать указание, какую работу требуется выполнить.

```
Iterator(Handling)
        current  ← first
        while current ≠  NULL  do
                call   Handling.working(current.data)
                current ←  next(current)
```

НЕДОСТАТКИ

- Для каждой задачи обработки набора данных необходимо создавать отдельный объект, реализующий обработку, и передавать его итератору набора данных.

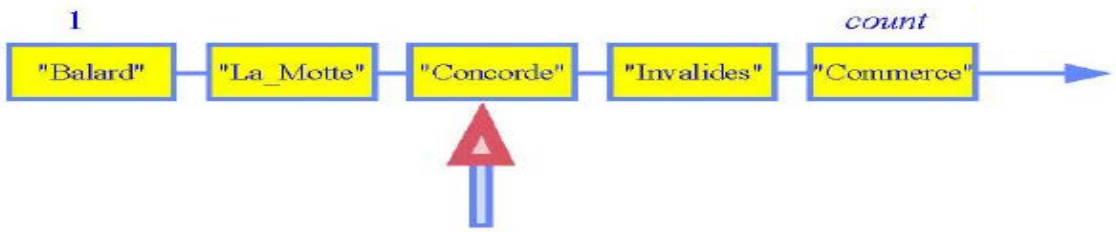
- Невозможна одновременная работа двух итераторов (например, для сравнения двух элементов набора данных), поскольку итератор не может прервать свою работу пока не будет закончена обработка всех элементов набора данных.

Внешний итератор

- Определяет интерфейс для доступа и перебора элементов, при котором следит за текущей позицией при обходе:

Характеристика:

- Предоставляет способ последовательного доступа ко всем элементам набора данных, не раскрывая его внутреннего строения;
- Поддерживает различные виды перебора (обхода);
- Предоставляет возможность нескольких одновременных переборов элементов набора данных.



Особенностью такого итератора является наличие *курсора* – это позиция текущего элемента, над которым можно производить различные действия.

Фреймворк коллекций Java

- Удобным способом организации «Итерации» является использование обычного цикла:

```
for (<начать итерацию>; <еще есть элементы>;
      <перейти к следующему элементу>) {
  <взять очередной элемент>;
  <обработать очередной элемент>;
}
```

- Для каждого элементарного действия необходимо описать метод:

✓ установить начало итерации (*iterator()*);

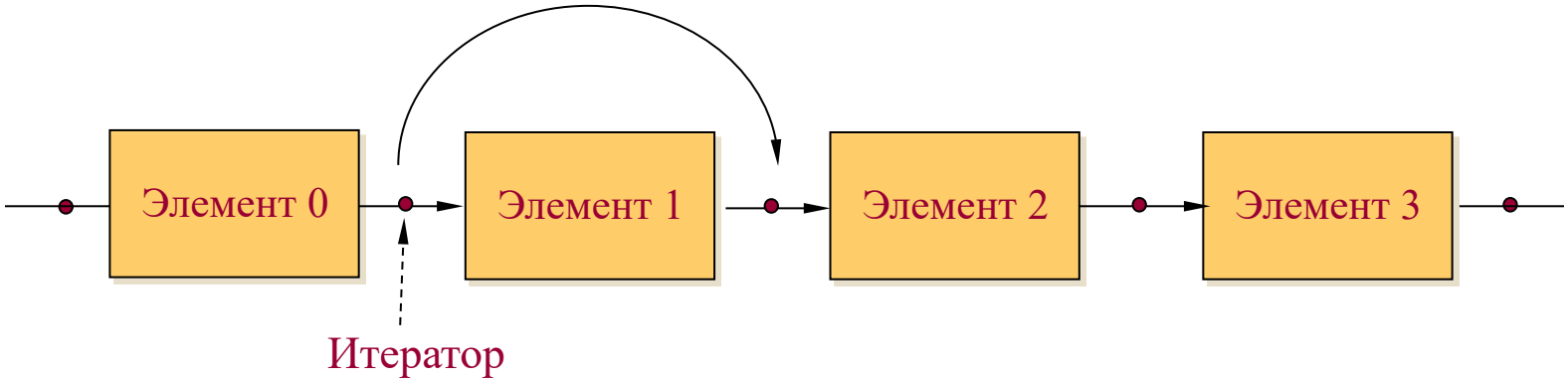
✓ проверить, есть ли еще элемент для получения (*hasNext()*);

✓ получить элемент и перейти к следующему (*next()*).

Фреймворк коллекций Java

- Интерфейс ***Iterator*** и представляет собой «внешний итератор»

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();          // удалить текущий элемент
}
```



Пример 5:

```
Collection<String> myColl = new ArrayList<>();
myColl.add("fortran");
myColl.add("c#");
myColl.add("java");
System.out.println(myColl);
Iterator<String> itr = myColl.iterator();
while(itr.hasNext()) {
        itr.next();
        itr.remove();
}
System.out.println(myColl);
```

Вывод в консоли:

[fortran, c#, java]

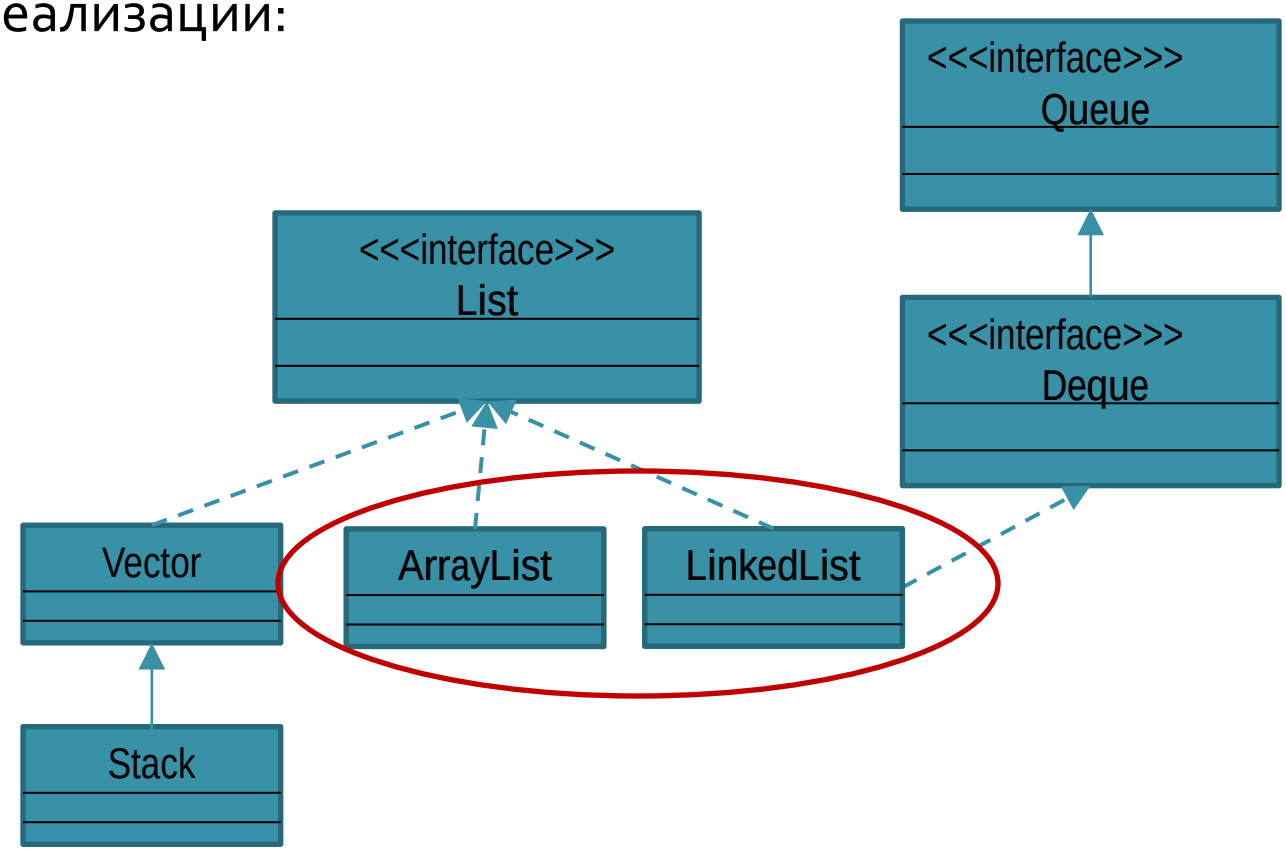
[]

СПИСКИ

- **интерфейс** List
- **интерфейс** Queue
- **интерфейс** Deque

ИНТЕРФЕЙС List

- **List** является линейной коллекцией, которая может содержать повторяющиеся элементы и имеет две реализации:

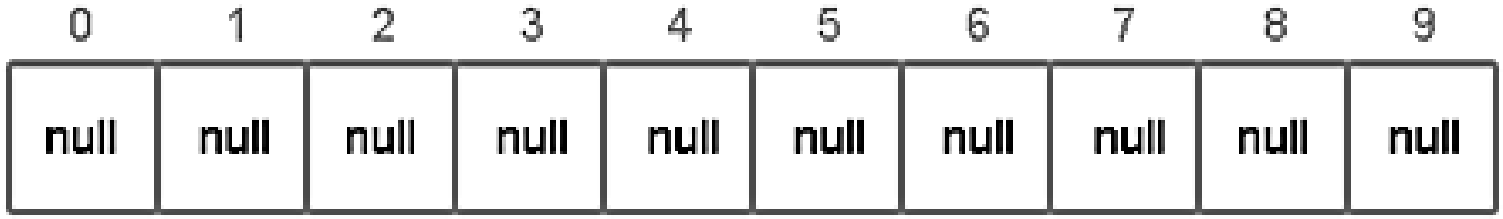


МЕТОДЫ List

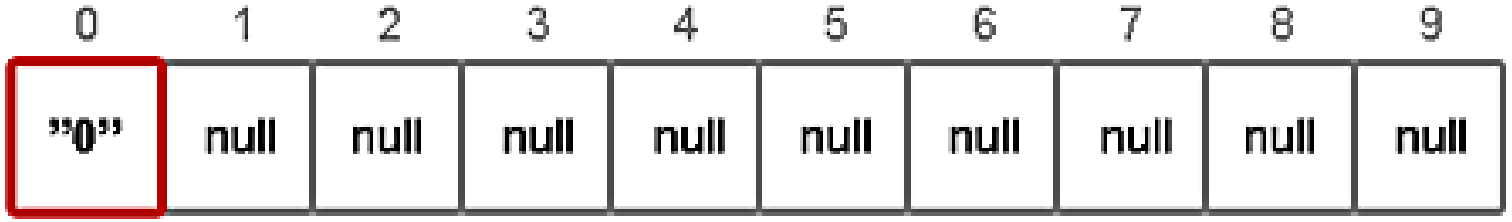
1. ListIterator ***listIterator***(); - получить итератор для коллекции
2. ListIterator ***listIterator***(int index); - получить итератор, для которого первый вызов метода *next()* возвращает элемент из позиции ***index***
3. void ***add***(int index, Object obj); - добавить элемент в позицию ***index***
4. void ***addAll***(int index, Collection c); - добавить коллекцию **c** в позицию ***index***
5. Object ***remove***(int index); - удаляет и возвращает и удаляет элемент из позиции ***index***
6. Object ***get***(int index); - получить элемент из позиции ***index***
7. Object ***set***(int index, Object obj); - заменить элемент в позиции ***index*** и возвратить старый
8. int ***indexOf***(Object obj); - получить позицию первого вхождения элемента
9. int ***lastIndexOf***(Object obj); - получить позицию последнего вхождения элемента

- **ArrayList** – представляет собой динамический массив.

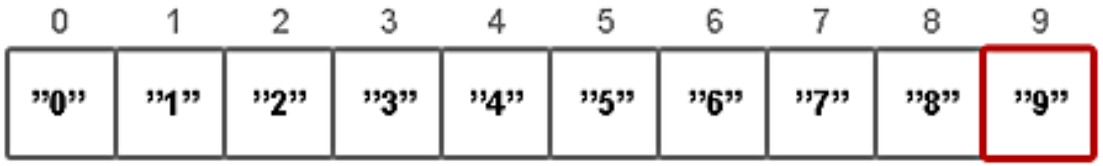
```
List<String> myList = new ArrayList<>();
```



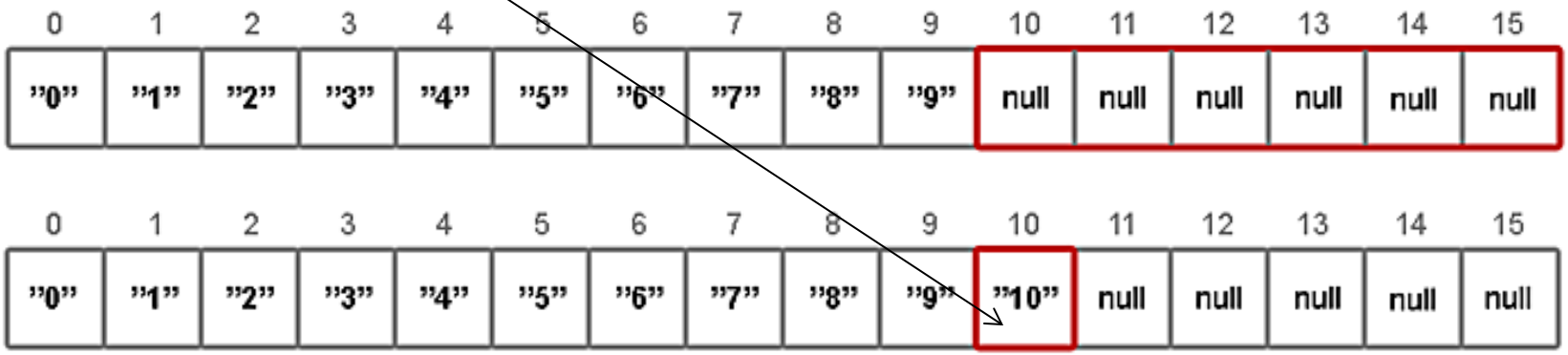

```
myList.add("0");
```



```
myList.add("1");
    //...
myList.add("9");
```

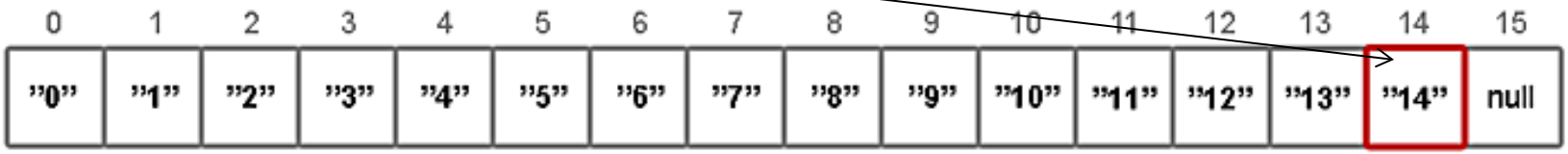



```
myList.add("10");
```



Новый массив создается с размером:
(myList.length * 3) / 2 + 1

```
//....
myList.add("14");
```



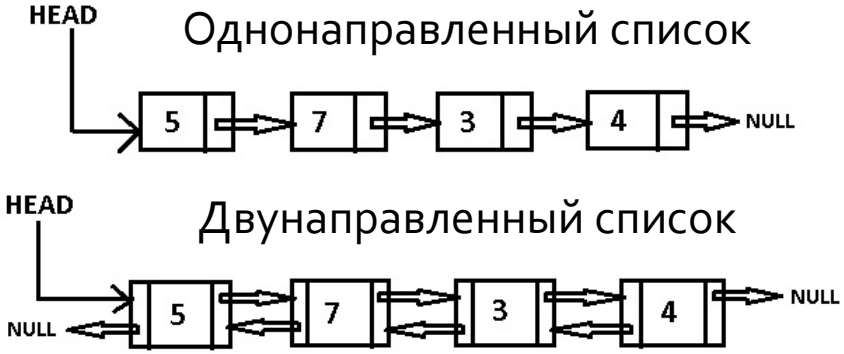

```
myList.add(5, "100");
```

Примечание: Операции вставки и удаления в середину списка могут быть очень длительными из-за операций смещения и копирования массивов.

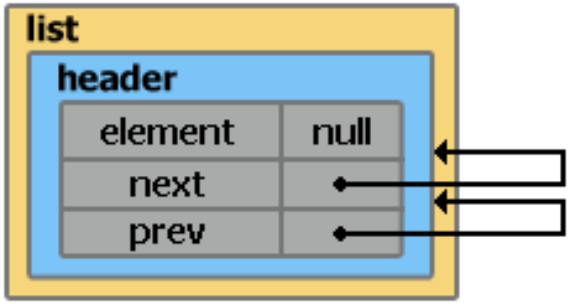
- **LinkedList -** это линейная структура данных, в которой данные хранятся в узлах и каждый узел содержит ссылку на следующий узел в списке.

Есть два типа **LinkedList**:

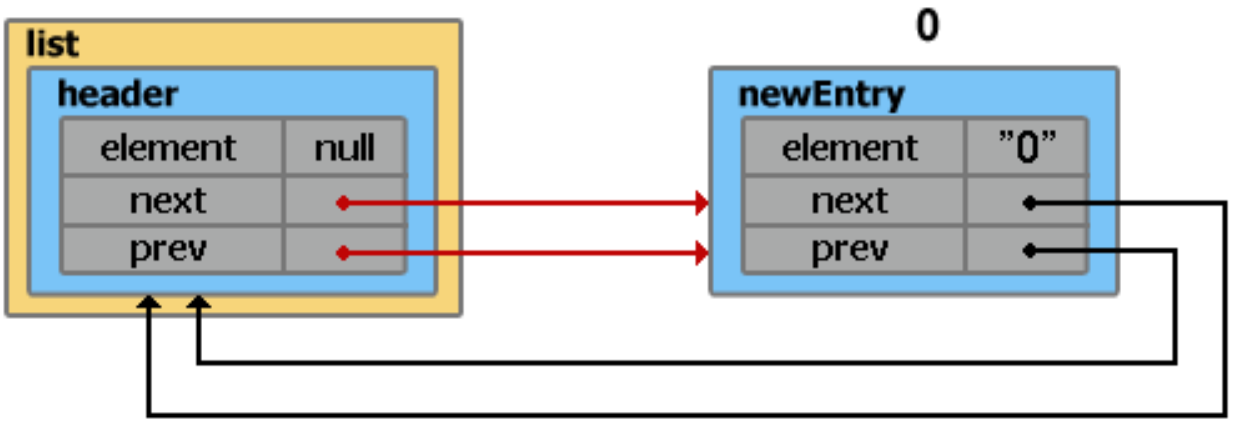
- *Однонаправленный список* состоит из узлов, где каждый узел содержит данные и указатель на следующий элемент в списке;
- *двунаправленный список* состоит из узлов, где каждый узел содержит данные и два указателя: на предыдущий и следующий элемент в списке.



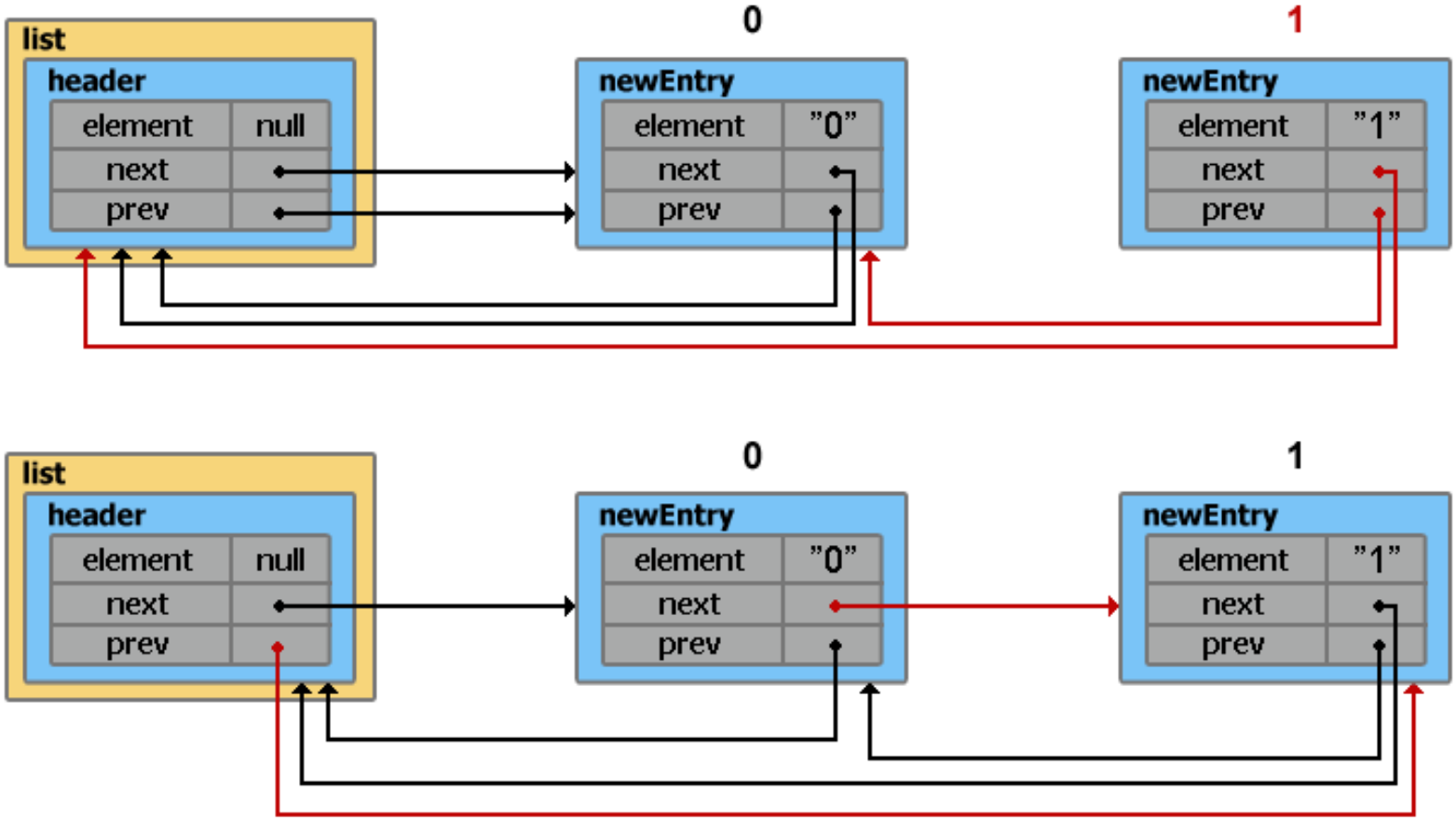
```
List<String> list = new LinkedList<>();
```



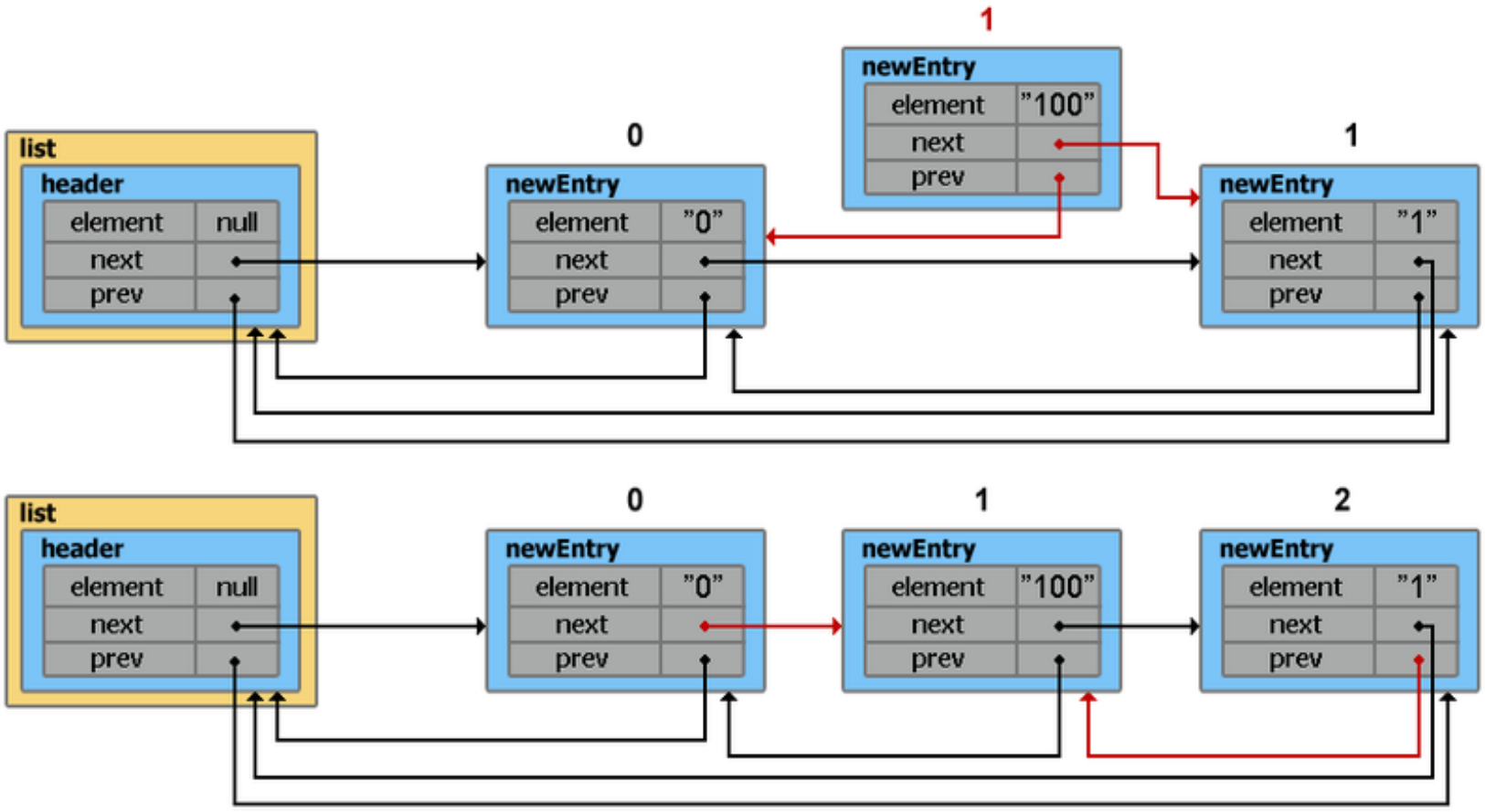

```
list.add("0");
```



list.add("1");

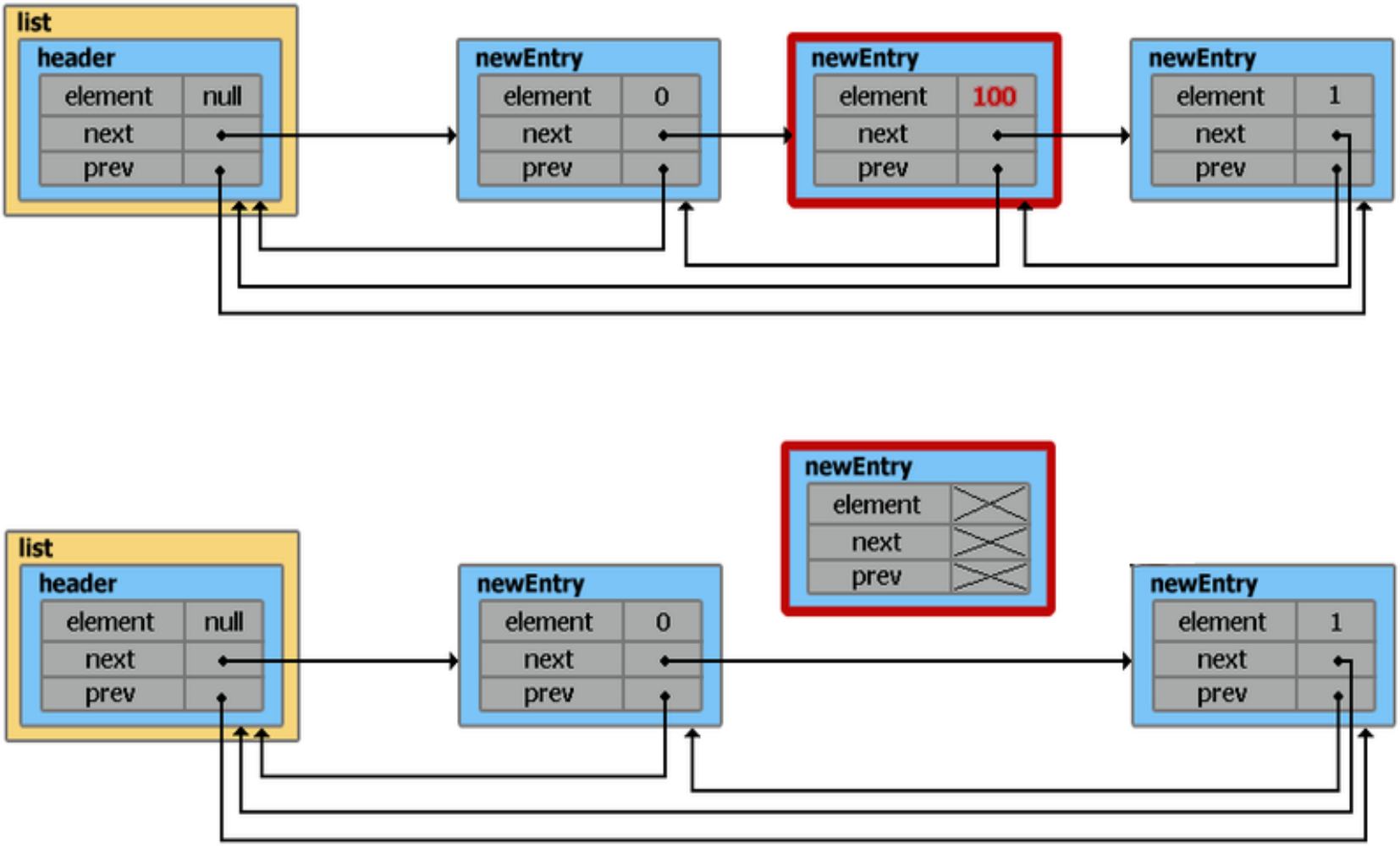

list
header
element
null
next
prev
0
newEntry
element
"0"
next
prev
1
newEntry
element
"1"
next
prev
list
header
element
null
next
prev
0
newEntry
element
"0"
next
prev
1
newEntry
element
"1"
next
prev

```
list.add(1, "100");
```



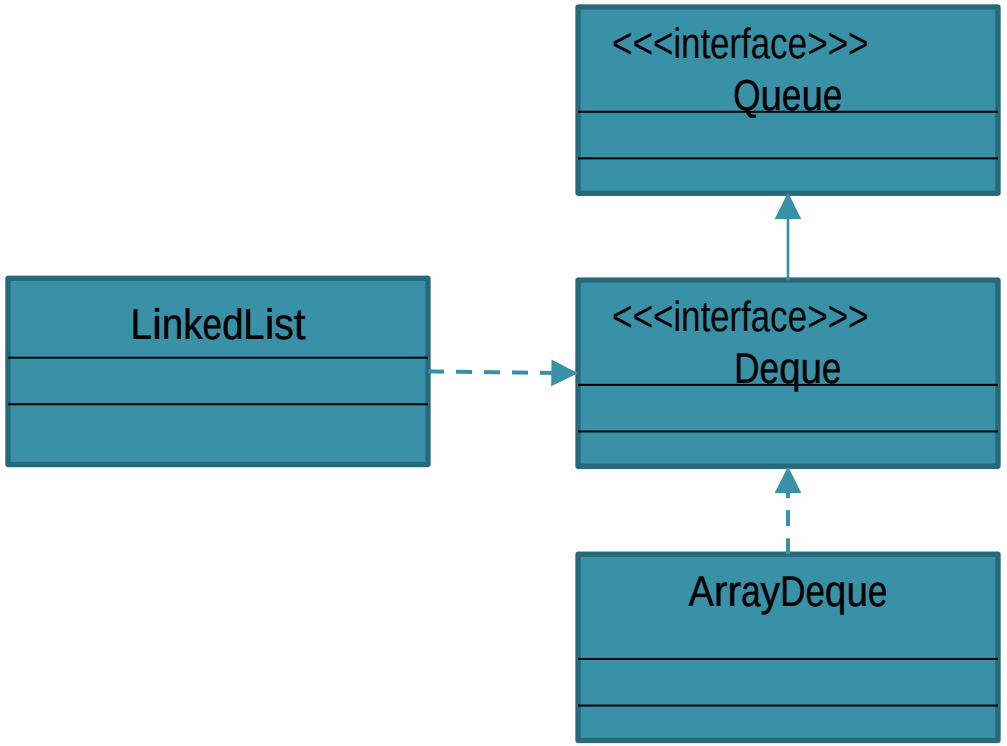
Примечание: Вставить можно только в существующую позицию!

list.remove("100");



ИНТЕРФЕЙС Queue

- *Интерфейс* ***Queue*** содержит методы по манипуляции первым и последним элементами списка, т.е. придает списку свойства очереди (элементы хранятся в порядке вставки и удаляются в том же порядке - FIFO).



Методы интерфейса Queue<E>

E ***element***(); – получить первый элемент очереди

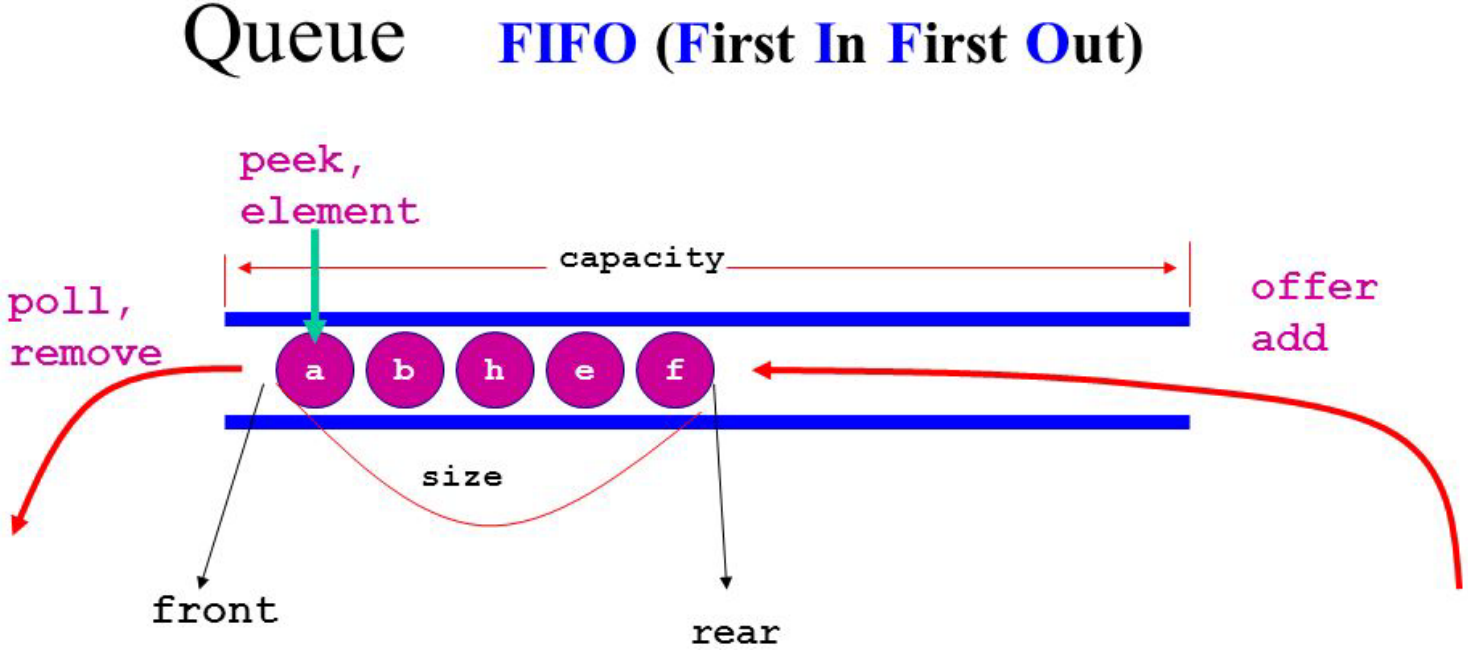
E ***peek***(); – получить первый элемент очереди (возвращает **null**, если очередь пуста)

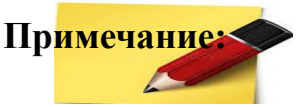
boolean ***offer***(E o); – добавить элемент в очередь, если возможно

boolean ***add***(E o); – добавить элемент в очередь, если возможно

E ***remove***(); – получить и удалить первый элемент очереди

E ***poll***(); – получить и удалить первый элемент очереди (возвращает **null**, если очередь пуста)





Методы ***element()***, ***remove()*** и ***add()*** отличаются от методов ***peek()***, ***poll()*** и ***offer()*** тем, что генерируют исключение, если не могут выполнить свои действия.

Пример 6:

```
import  java.util.*;
public class  DemoLinkedList {
public static void main(String[] args) {
        List<Number> list = new LinkedList<>();
        for (int i = 10; i <= 15; i++)
            list.add(i);
        for (int i = 16; i <= 20; i++)
           list.add(new Float(i));
        System.out.println(list.size() + " элементов");

        Iterator<Number> it = list.iterator();
        while (it.hasNext())
              System.out.print(it.next() + " ");

        ListIterator<Number> listL = list.listIterator(10);
        System.out.println("\n"+ listL.nextIndex() + "-й индекс");
        listL.next();
        System.out.println(listL.nextIndex() + "-й индекс");
```

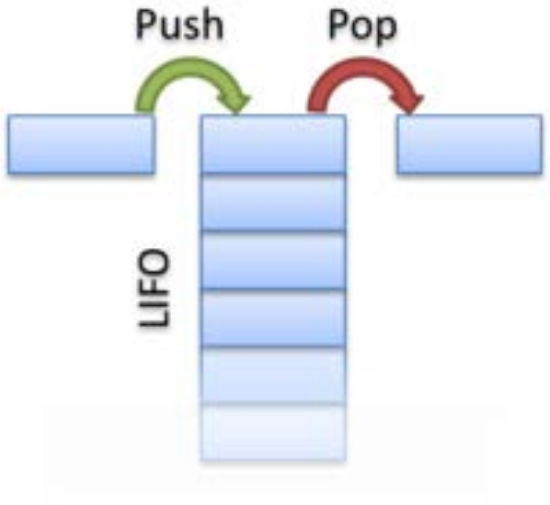
```
    while (listL.hasPrevious())
            System.out.print(listL.previous() + "  ");

    Queue <Number> que = (LinkedList)list;
    que.remove();
    que.offer(71);
    que.poll();
    que.remove();
    que.remove(1);
    System.out.println("\n" + que);

    for (Number i : que)
            System.out.print(i + " ");

    System.out.println(" : size= " + que.size());
    for (int i = 0; i < 5; i++)
            que.poll();
    System.out.println(" :size= " + que.size());
  }
}
```

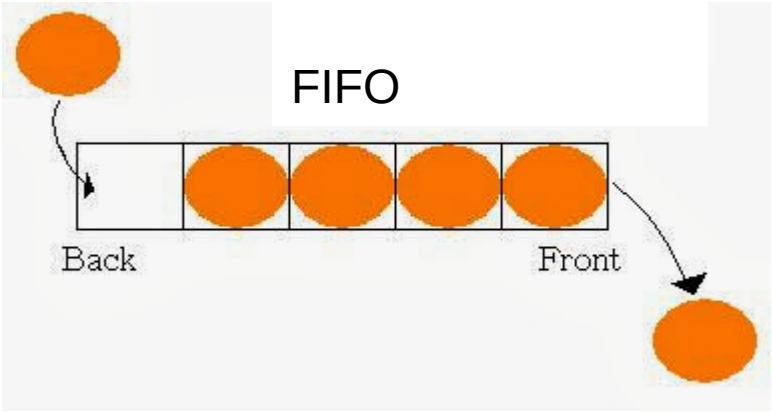
Вывод в консоли:

11 элементов
10 11 12 13 14 15 16.0 17.0 18.0 19.0 20.0
10-й индекс
11-й индекс
20.0 19.0 18.0 17.0 16.0 15 14 13 12 11 10
[13, 14, 15, 16.0, 17.0, 18.0, 19.0, 20.0, 71]
13 14 15 16.0 17.0 18.0 19.0 20.0 71 : size= 9
: size= 4

ИНТЕРФЕЙС Deque

- *Интерфейс* ***Deque*** поддерживает вставку и удаление элементов с обеих конечных точек (т.е. может быть использован как стек (last-in-first-out), так и как очередь (first-in-first-out)).





Методы интерфейса Deque

Вставить	`addLast(e)` `offerLast(e)`	`addFirst(e)` `offerFirst(e)`
Удалить	`removeLast()` `pollLast()`	`removeFirst()` `pollFirst()`
Получить	`getLast()` `peekLast()`	`getFirst()` `peekFirst()`

- Методы *addFirst()* и *addLast()* при ограниченной емкости **Deque** выбросят исключение, если она полная.
- Методы *removeFirst()* и *removeLast()* выбросят исключение, если очередь пуста.
- Методы *getFirst()* и *getLast()* выбросят исключение, если очередь пуста.

ВЫВОДЫ

- **ArrayList**:
  - Произвольный доступ к элементу по индексу за постоянное время;
  - Минимум накладных расходов при хранении такого списка;
  - Вставка в конец списка *в среднем* производится так же за постоянное время;
  - Удаление последнего элемента происходит за постоянное время;
  - Вставка/удаление элементов *в середине списка* взывает перезапись всех элементов размещенных «правее» в списке;
  - Удаление элементов не уменьшает размер массива.

ВЫВОДЫ

- **LinkedList**:
  - Вставка/удаление элементов в списке происходит за постоянное время (поиск позиции сюда не входит);
  - Доступ к произвольному элементу осуществляется за линейное время (не поддерживается произвольный доступ);
  - Доступ к первому и последнему элементу списка всегда осуществляется за постоянное;
  - Потребляемая память и скорость выполнения операций больше, чем у **ArrayList**.

- **LinkedList** предпочтительно применять, когда происходит активная с серединой списка или когда необходимо гарантированное время добавления элемента в список.

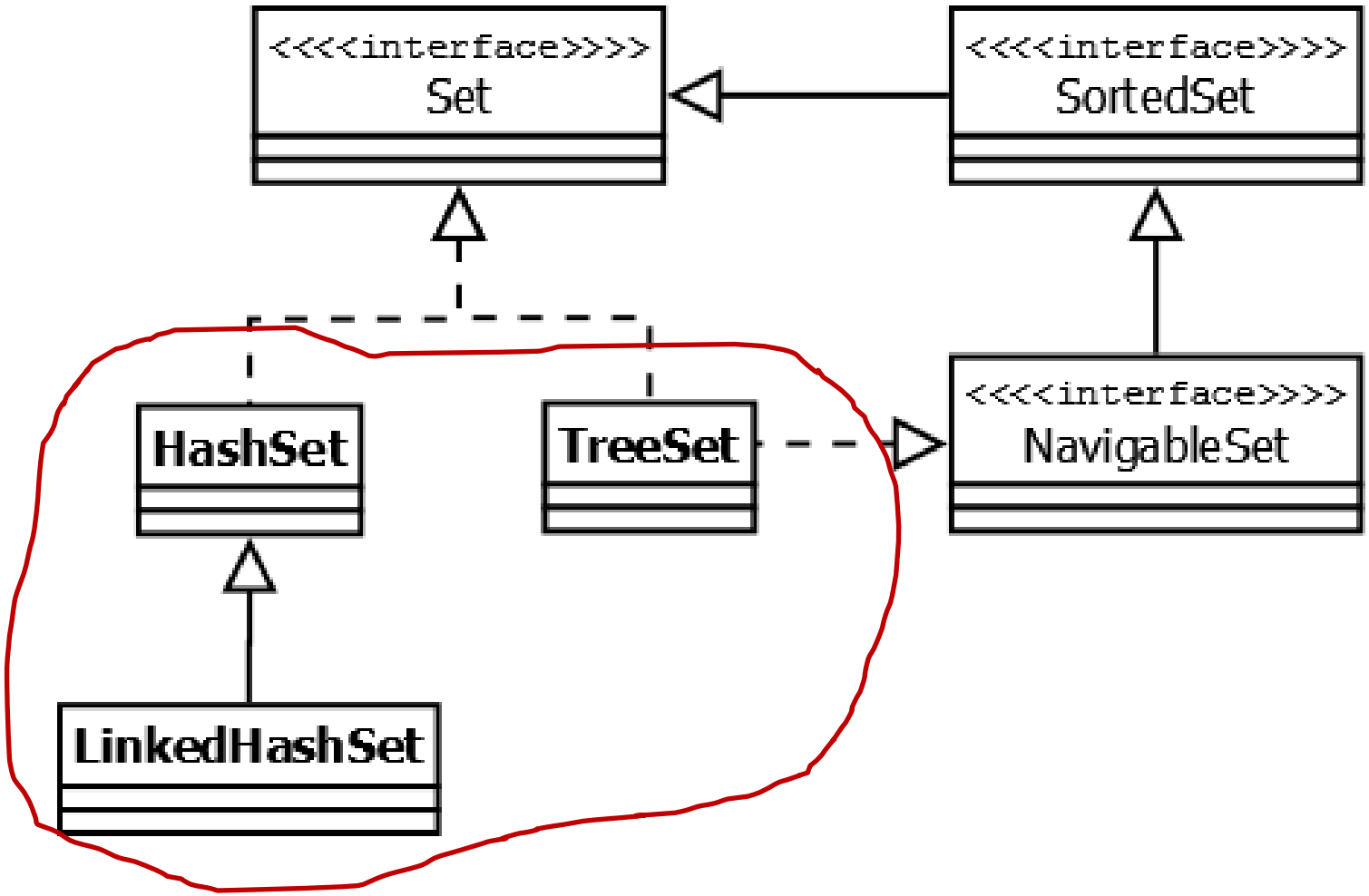
МНОЖЕСТВА

- **интерфейс** Set
- **интерфейс** SortedSet

ИНТЕРФЕЙС Set

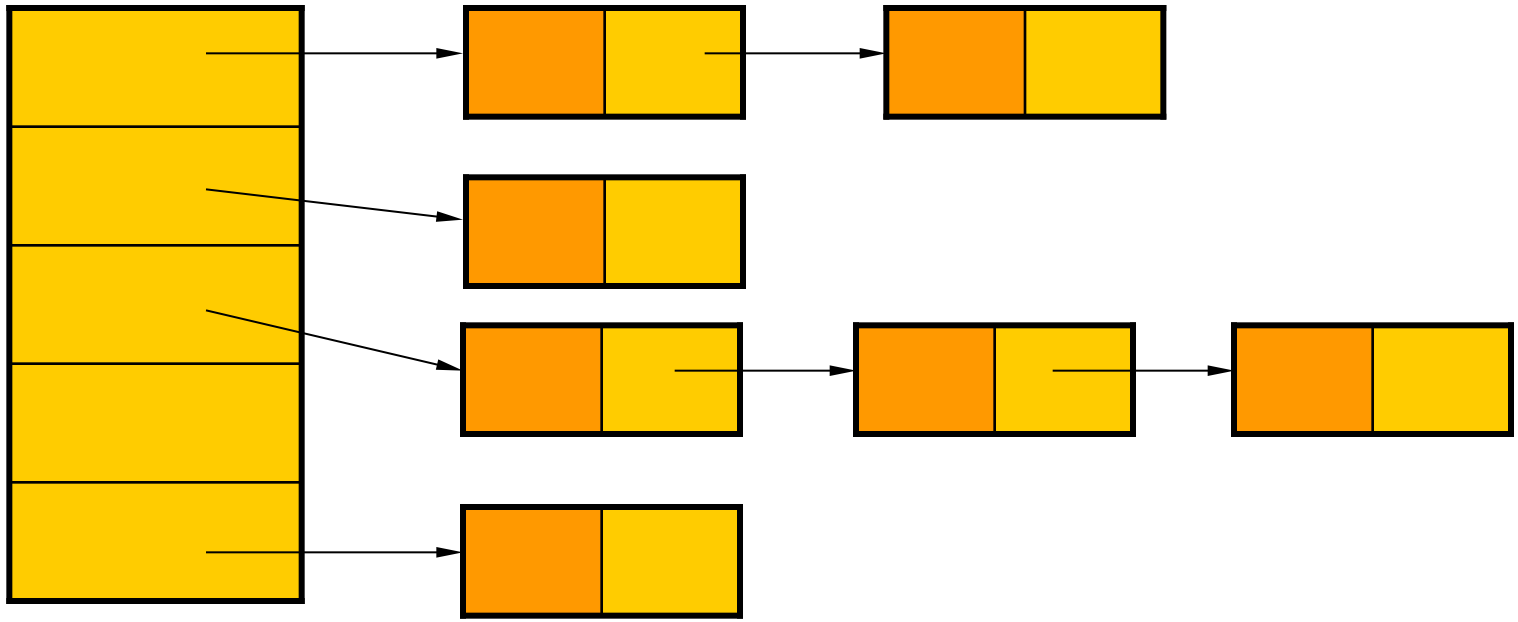
- *Интерфейс* ***Set*** – это коллекция, которая не может содержать повторяющихся элементов, так как поддерживает математическую абстракцию множества;
- *Интерфейс* ***Set*** содержит только методы, унаследованные от интерфейса **Collection** и добавляет ограничение запрещающее дублировать элементы;
- *Интерфейс* ***Set*** добавляет строгое поведение на операции *equals()* и *hashCode()*, что позволяет объектам **Set** быть сравнимыми по значениям, даже если их типы реализации отличаются.

Интерфейс **Set** имеет три основных реализации:



Фреймворк коллекций Java

- **HashSet** – осуществляет хранение и поиск элементов множества на основе хэш-таблицы, в которой индекс представляет собой ключ (хэш-код), а сама таблица массив связнных списков.



КОНСТРУКТОРЫ HashSet

HashSet()	- создается пустое хэш-множество (емкость 16 и коэффициент загруженности 0.75);
HashSet(Collection e)	- создается хэш-множество из элементов переданной коллекции;
HashSet(int size)	- создается пустое хэш-множество заданной емкости;
HashSet(int size, float Factor)	- создается пустое хэш-множество заданной емкости и коэффициентом загруженности.

Коэффициент загруженности является одной из важных характеристик хэш-таблиц;

- ***Коэффициент загруженности*** – это отношение количества объектов, которые хранятся в хеш-таблице ***(N),*** к емкости хеш-таблицы ***(m):***

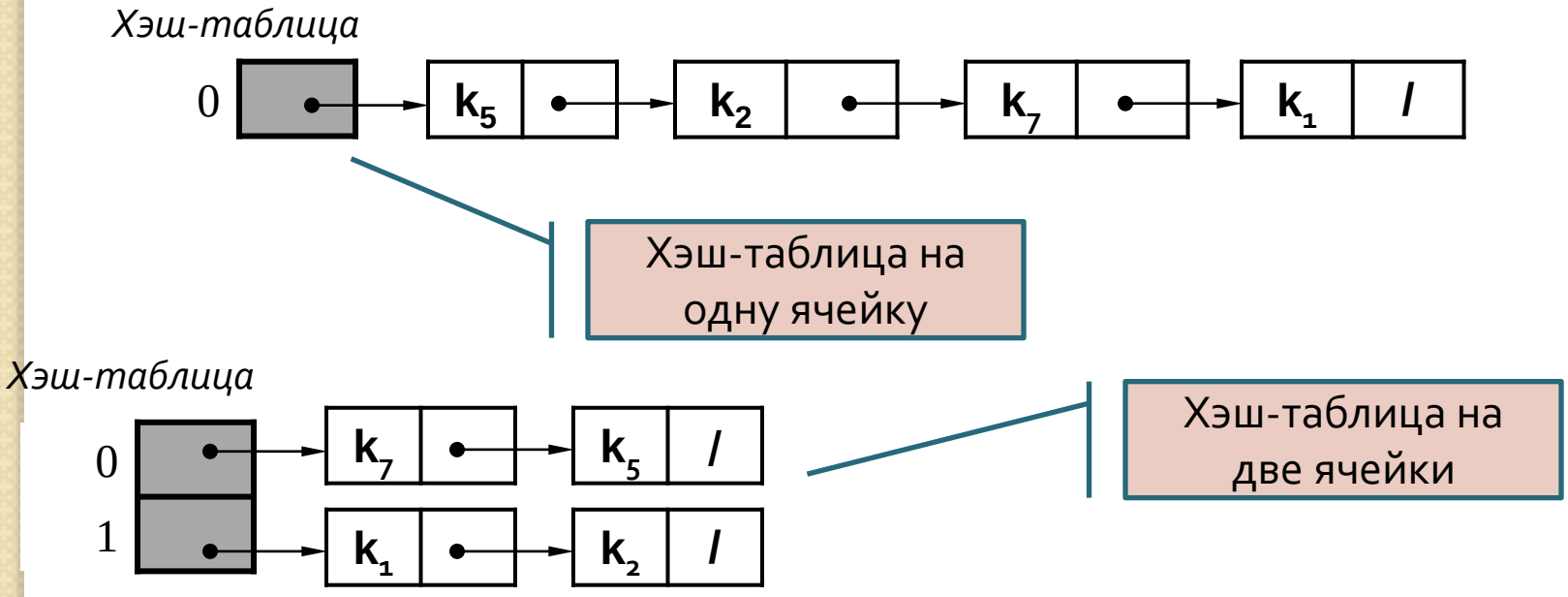
$$\alpha = \frac{N}{m}$$

- *Коэффициент загруженности* может быть как меньше, равен, так и больше единицы;

- От значения *коэффициента загруженности* зависит среднее время выполнения операций добавления, поиска и удаления элементов.

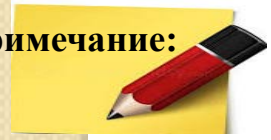
Время выполнения операций (1/2)

- Чем меньше таблица, тем больше среднее время поиска ключа в ней (при фиксированном количестве элементов);
  - *Например*, если хэш-таблицу рассматривать как совокупность связных списков, то, по мере роста таблицы, увеличивается и количество списков (соответственно, среднее число элементов в каждом списке уменьшается);

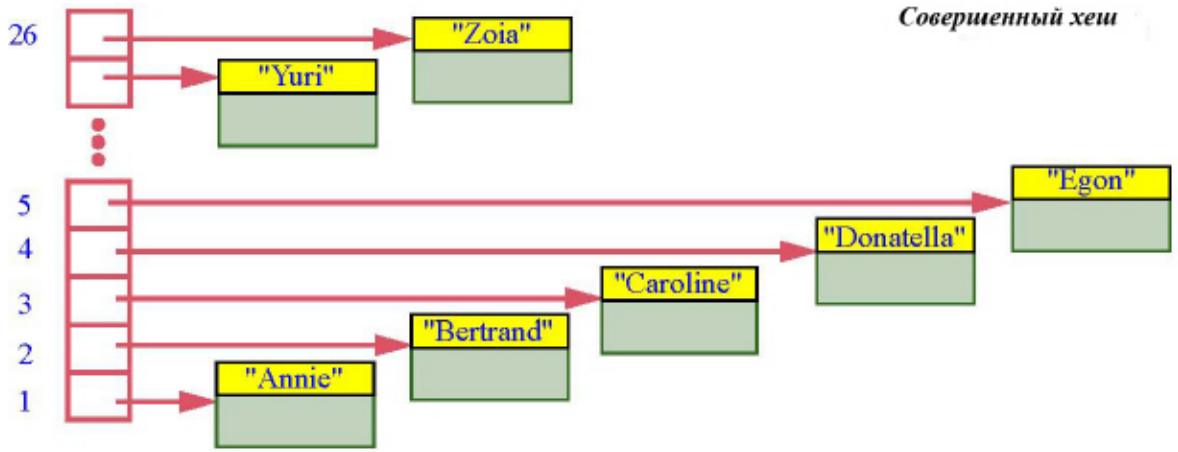


Время выполнения операций (2/2)

Примечание:

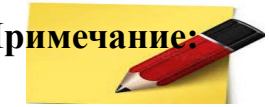


- ✓ Среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка **α**, т.к. поиск сводится к просмотру одного из списков (среднее время просмотра которого равно **α**).
- ✓ Поскольку среднее время вычисления хеш-функции равно **Θ(1)**, то среднее время выполнения каждой операции с учетом вычисления хеш-функции равно **Θ(1 + α)**.



Как добавляется элемент

- Для добавления элемента в **HashSet** вычисляется его хэш-код;
- Затем вычисляется остаток от деления хэш-кода на емкость **HashSet** (хэш-таблицы);
- Полученное значение является индексом списка в **HashSet**, в котором будет находится элемент множества.



HashSet *не дает никаких гарантий относительно порядка итерации, т.е. порядок вставки элементов и порядок итерирования – РАЗЛИЧАЮТСЯ!*

Пример 7:

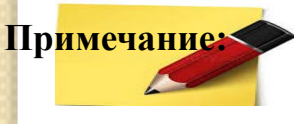
```
Set<String> mySet = new HashSet<>();
mySet.add("data");
mySet.add("java");
mySet.add("java");
mySet.add("data");
mySet.add("sorting");
mySet.add("hi");
mySet.add("hello");
System.out.println(mySet);
```

Вывод в консоли:
[hi, java, data, sorting, hello]

Порядок итерирования не совпадает с порядком вставки

Фреймворк коллекций Java

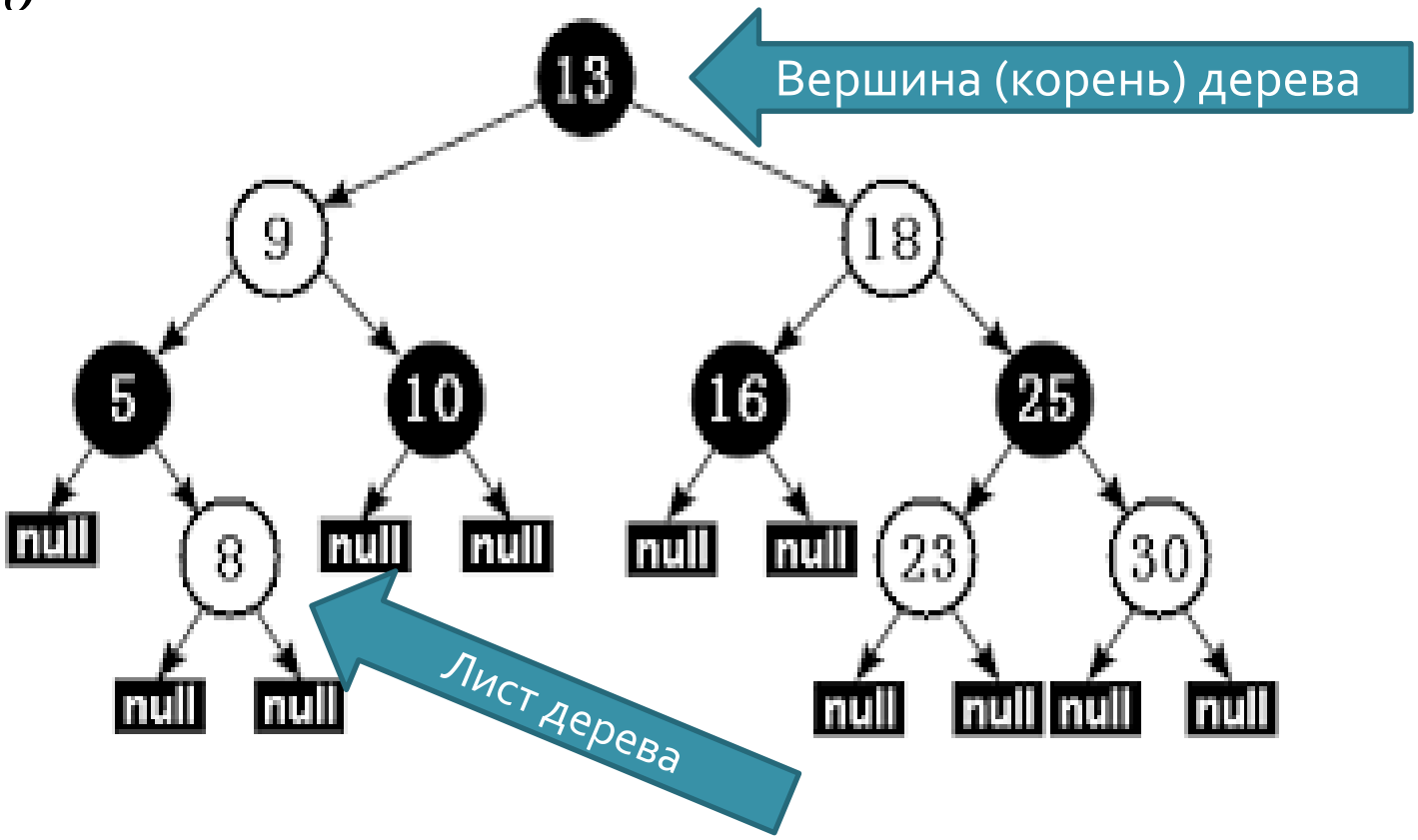
- ***TreeSet*** *- реализует интерфейс* **SortedSet** и для хранения объектов использует бинарное дерево, которое при добавлении объекта размещает его в необходимой позиции с учетом сортировки (т.е. все добавляемые элементы должны реализовывать интерфейс **Comparable или Comparator**);



1) По умолчанию используется естественный порядок сортировки;

2) Можно использовать интерфейс **Comparator**, чтобы изменить правило сортировки.

- Естественный порядок сортировки – *каждый узел имеет левого потомка со значением меньше, чем у него, и правого потомка со значением, больше, чем у него*



КОНСТРУКТОРЫ TreeSet

TreeSet()	- создается пустое множество (естественный порядок сортировки);
TreeSet(Collection e)	- создается из элементов переданной коллекции;
TreeSet(Comparator)	- создается пустое множество с заданным порядком сортировки;

МЕТОДЫ ИНТЕРФЕЙСА SortedSet<E>

Comparator<? super E> ***comparator***() – получить компаратор, используемый для сортировки

E ***first***() - получить наименьший элемент

E ***last***() - получить наибольший элемент

SortedSet<E> ***subSet***(E from, E to) - извлечь множество между элементами *from* и *to*

SortedSet<E> ***tailSet***(E from) - извлечь множество-хвост от элемента *from*

SortedSet<E> ***headSet***(E to) - извлечь множество-голову от элемента *to*

Пример 8:

```
Collection<String> col = new ArrayList<> ();
for (int i = 0; i < 6; i++)
      col.add((int) (Math.random() * 71) + "Y ");
System.out.println(col + " - список");
TreeSet <String> set = new TreeSet <> (col);
System.out.println(set + " - множество");
set.add("5 Element");
System.out.println(set + " - add");
Iterator <String> it = set.iterator();
while (it.hasNext()) {
     if (it.next() == "5 Element")
           it.remove();
}
System.out.println(set + " - delete");
System.out.println(set.last() + " " + set.first());
```

Вывод в консоли:

```
[18Y , 11Y , 19Y , 9Y , 4Y , 11Y ] - список
[11Y , 18Y , 19Y , 4Y , 9Y ] - множество
[11Y , 18Y , 19Y , 4Y , 5 Element, 9Y ] - add
[11Y , 18Y , 19Y , 4Y , 9Y ] - delete
9Y  11Y
```

Пример 9:

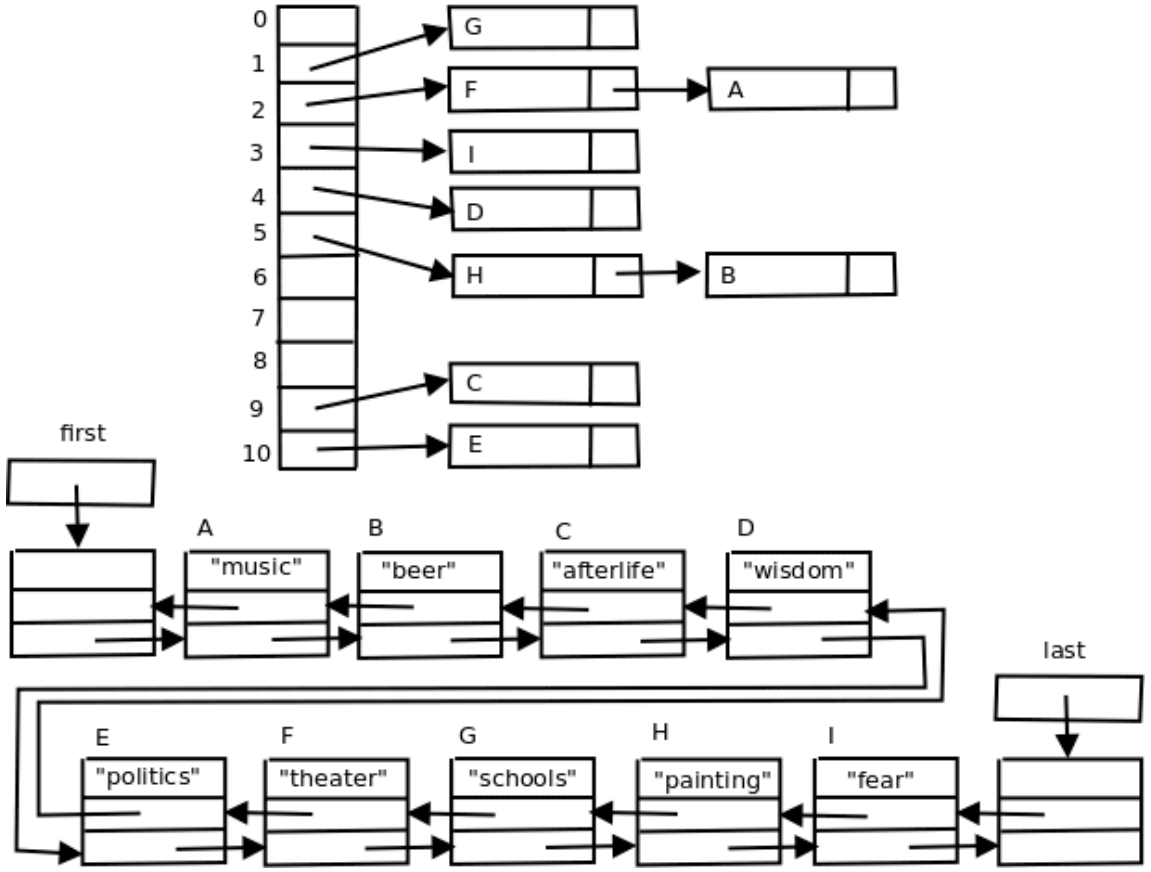
```
SortedSet<Integer> tree = new TreeSet<>();
int[] array = {100, 50, 70, 150, 200, 120,
                    30, 10, 60, 80, 110, 180};
for (int element : array) {
    tree.add(element);
}
SortedSet<Integer> subTree1 = tree.tailSet(100);
System.out.println("Tail\n" + subTree1);

System.out.println("------------------------------");
SortedSet<Integer> subTree2 = tree.headSet(100);
System.out.println("Head\n" + subTree2);

System.out.println("------------------------------");
SortedSet<Integer> subTree3 = tree.subSet(60, 120);
System.out.println("SubTree\n" + subTree3);
```

Вывод в консоли:

```
Tail
[100, 110, 120, 150, 180, 200]
------------------------------
Head
[10, 30, 50, 60, 70, 80]
------------------------------
SubTree
[60, 70, 80, 100, 110]
```

Фреймворк коллекций Java

- ***LinkedHashSet*** – это хэш-таблица со связным двунаправленным списком, проходящим через ее, для хранения порядка, в котором элементы были вставлены в набор.



Пример 10:

```
Set<String> mySet = new LinkedHashSet<>();
mySet.add("data");
mySet.add("java");
mySet.add("java");
mySet.add("data");
mySet.add("sorting");
mySet.add("hi");
mySet.add("hello");
System.out.println(mySet);
```

Вывод в консоли:
[data, java, sorting, hi, hello]

Порядок итерирования совпадает с порядком вставки

ВЫВОДЫ

- **HashSet**:
  - Добавление, поиск и удаление элементов выполняется за постоянное время;
  - Порядок перебора элементов не соответствует порядку вставки.
- **TreeSet**:
  - Время выполнения основных операций соответствует ***logN*** (N – текущее количество узлов в дереве);
  - Обеспечивает упорядоченное хранение элементов в виде красно-черного дерева (чтобы гарантировать не вырождение дерева в список при вставке элементов в порядке возрастания значений).

КАРТЫ ОТОБРАЖЕНИЙ

- **интерфейс** Map
- **класс** HashMap
- **класс** TreeMap

ИНТЕРФЕЙС Map

- **Map** – предоставляет методы для работы с данными вида «ключ/значение» (ключ — это объект, который используется для последующего извлечения данных – значения);
  - **Map** – не может содержать дубликаты ключей;
  - В **Map** – каждый ключ может отображать не более чем одно значение;

Методы интерфейса Map<K,V>

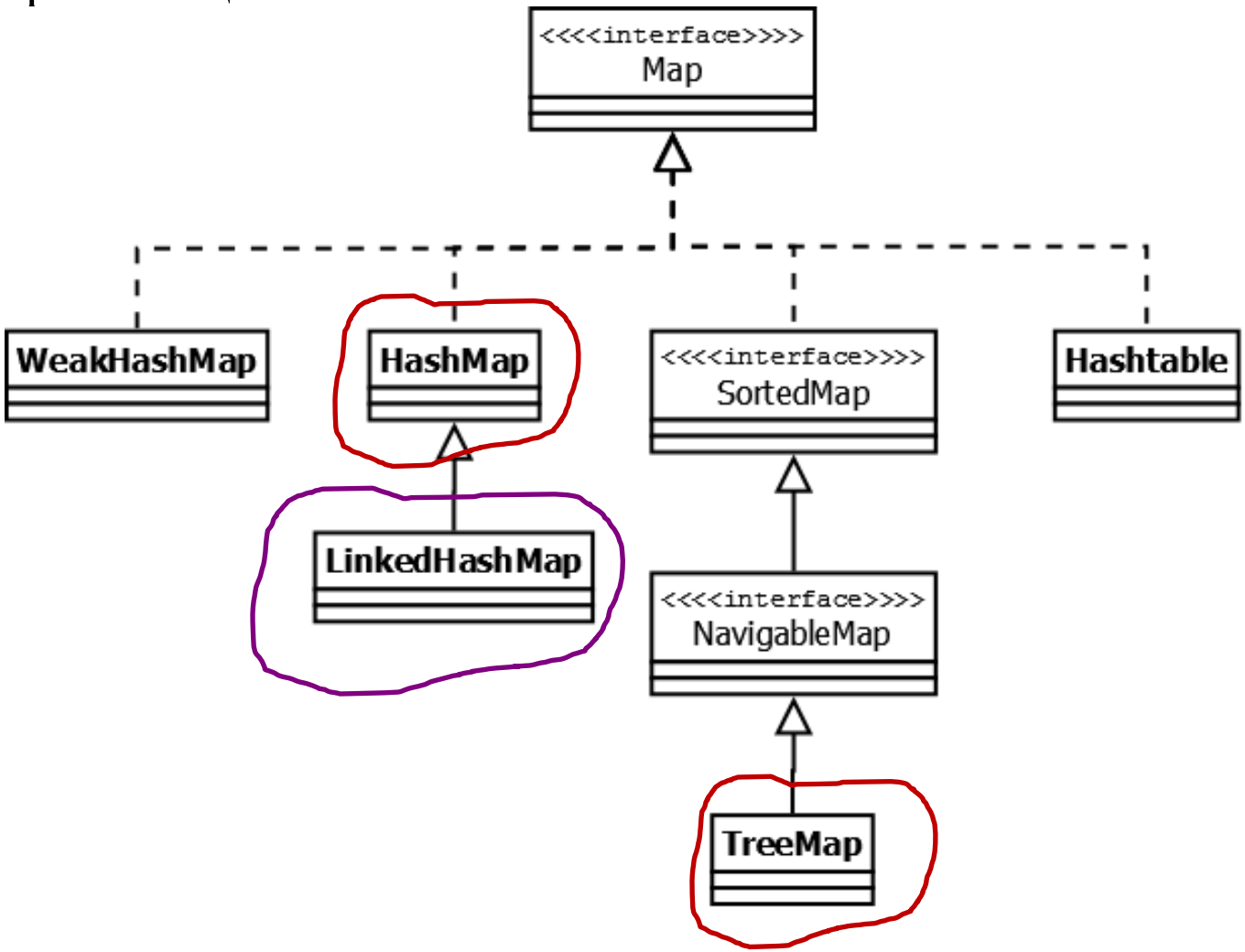
V ***get***(K key)	- получить объект, по указанному ключу
V ***put***(K key, V value)	- добавить объект в карту
boolean ***containsKey***(Object key)	- проверить есть ли в карте заданный ключ
boolean ***containsValue***(Object value)	- проверить есть ли в карте заданный объект
Set <K> ***keySet***()	- получить множество ключей
Collection<V> ***values***()	- получить набор объектов
Set<Map.Entry<K, V>> ***entrySet***()	- получить множество пар «ключ-значение»

Методы интерфейса Map.Entry<K,V>

K ***getKey***()	- получить ключ текущего элемента
V ***getValue***()	- получить значение текущего элемента
V ***setValue***(V newValue)	- заменить текущий объект

Фреймворк коллекций Java

Интерфейс **Map** имеет две наиболее часто используемые реализации:

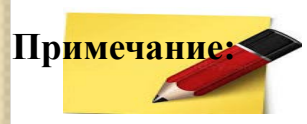


КЛАСС HashMap

- **HashMap** не является упорядоченной коллекцией: порядок хранения элементов зависит от хэш-функции.
  - добавление элемента выполняется за константное время;
  - время удаления/получения зависит от распределения хэш-функции;
  - позволяет использовать литерал **null** как в качестве ключа, так и значения.

КОНСТРУКТОРЫ HashMap

- *HashMap()* – пустая карта с начальной емкостью 16 и коэффициентом загруженности 0,75);
- *HashMap(int initialCapacity)* – пустая карта с заданной начальной емкостью и коэффициентом загруженности 0,75;
- *HashMap(int initialCapacity, float loadFactor)* – пустая карта с заданной начальной емкостью и коэффициентом загруженности.



Емкость – это число ячеек в хэш-таблице.

Коэффициент загруженности – это мера того, насколько заполненную хэш-таблицу позволено получить, прежде чем ее емкость автоматически увеличится.

Пример 11:

```
Map<Integer, String> hm = new HashMap<>(5);
for (int i = 1; i < 5; i++)
        hm.put(i, i + " el");
System.out.println(hm);
hm.put(2, "NEW");
System.out.println(hm + "с заменой элемента");
String a = hm.get(2);
System.out.println(a + " - найден по ключу '2'");
Set<Map.Entry<Integer, String>> set = hm.entrySet();
Iterator<Map.Entry<Integer, String>> i = set.iterator();
while (i.hasNext()) {
        Map.Entry<Integer, String> me = i.next();
        System.out.print(me.getKey() + " : ");
        System.out.println(me.getValue());
}
```

Вывод в консоли:

{1=1 el, 2=2 el, 3=3 el, 4=4 el}

{1=1 el, 2=NEW, 3=3 el, 4=4 el}с заменой элемента

NEW - найден по ключу '2'

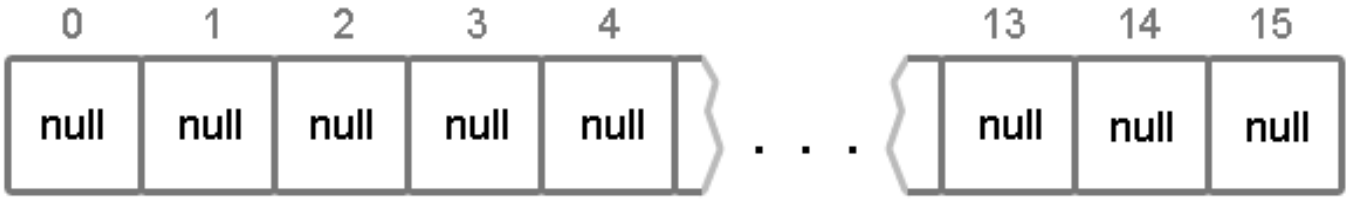
1 : 1 el

2 : NEW

3 : 3 el

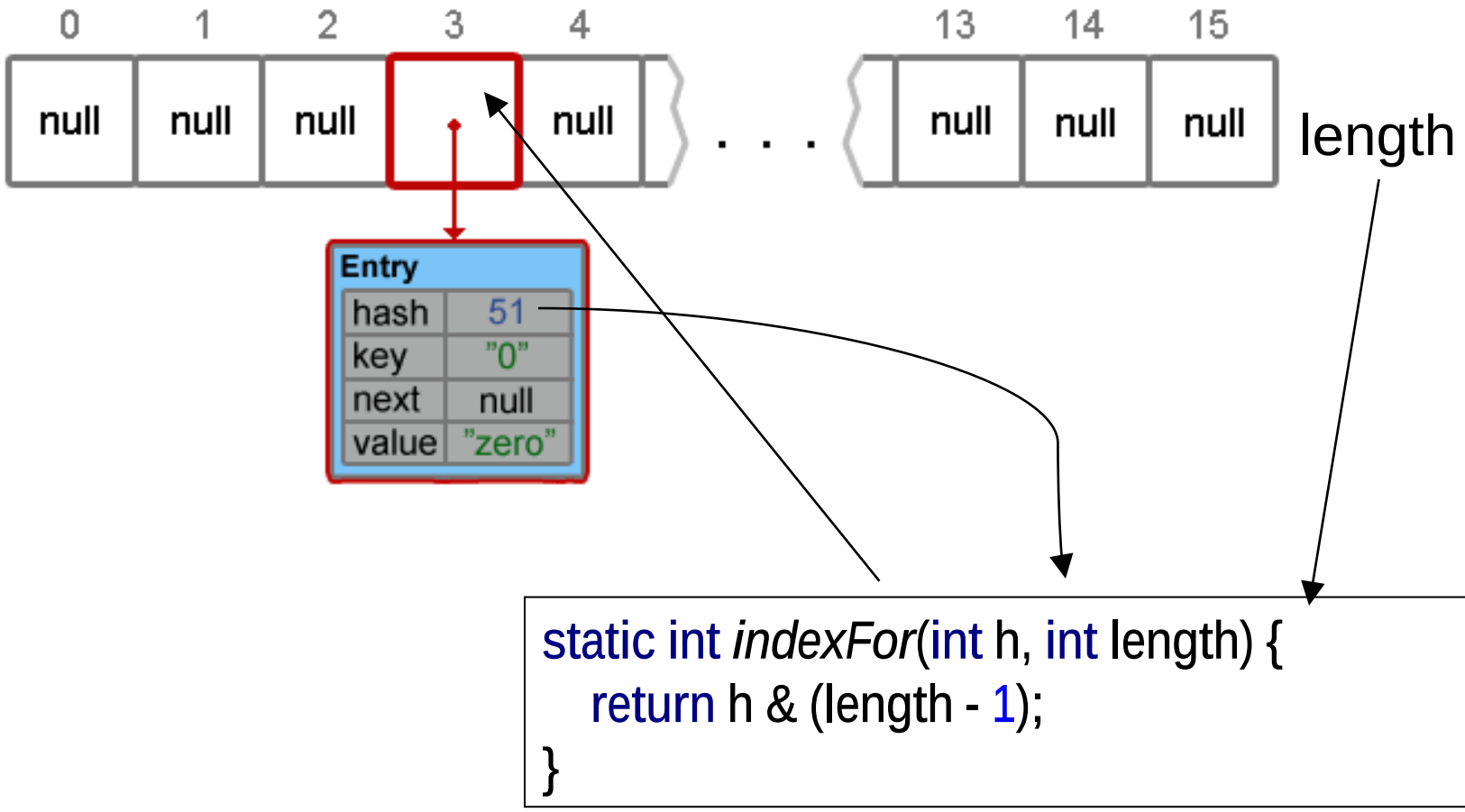
4 : 4 el

1) Map<String, String> hashMap = **new** HashMap<>();

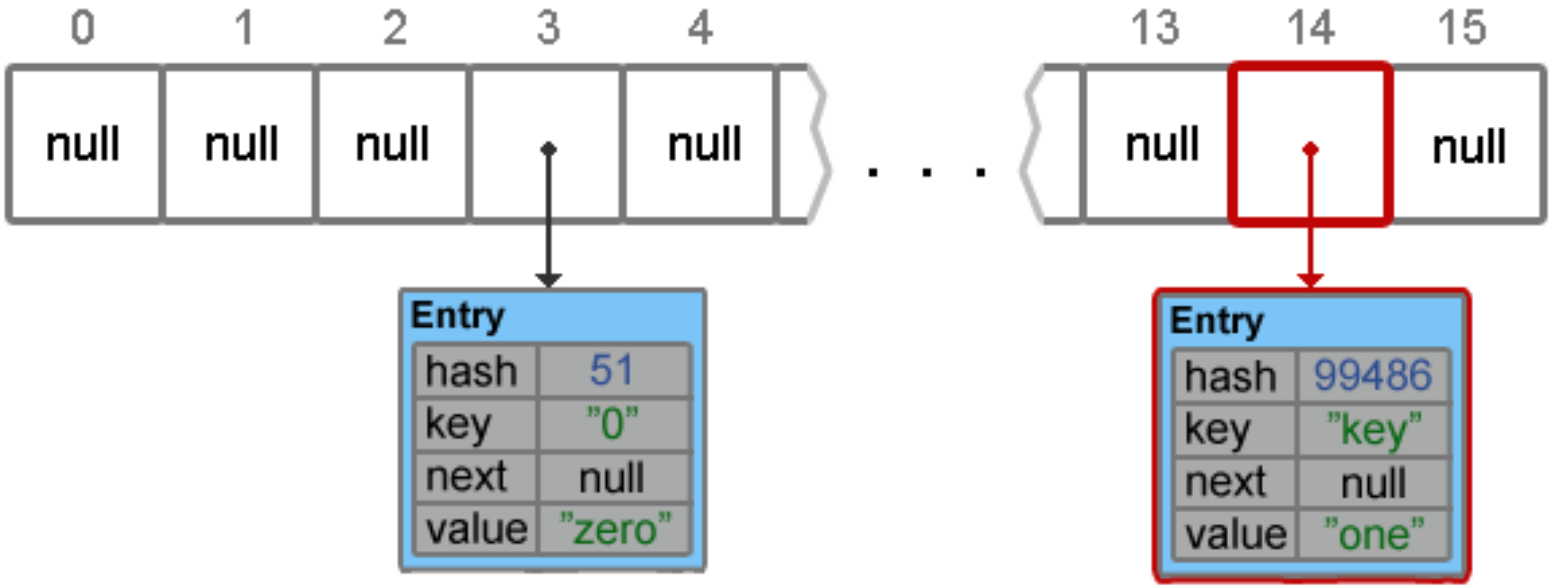



```
public class HashMap<K,V> extends AbstractMap<K,V>
        implements Map<K,V>, Cloneable, Serializable {

    transient Entry<K, V>[] table;
      //..
    static class Entry<K, V> implements Map.Entry<K, V> {
            final K key;
            V value;
            Entry<K, V> next;
            int hash;
            //...
```

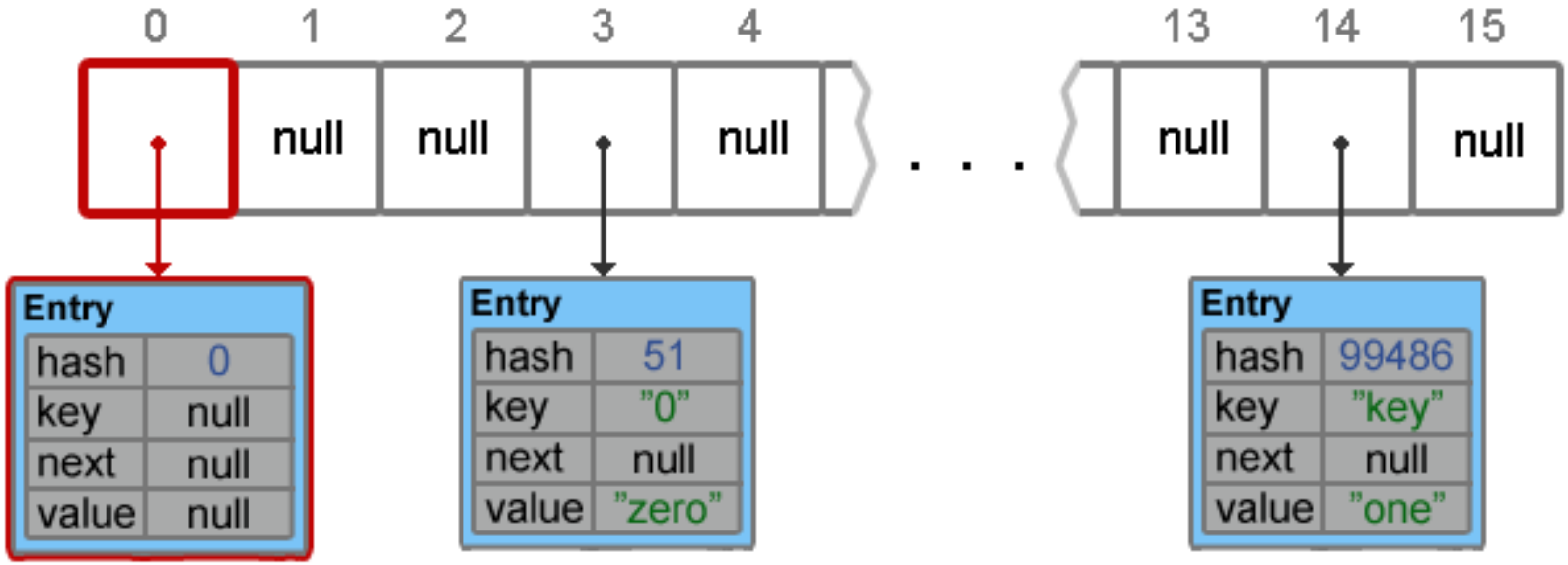
2) hashMap.put("0", "zero");



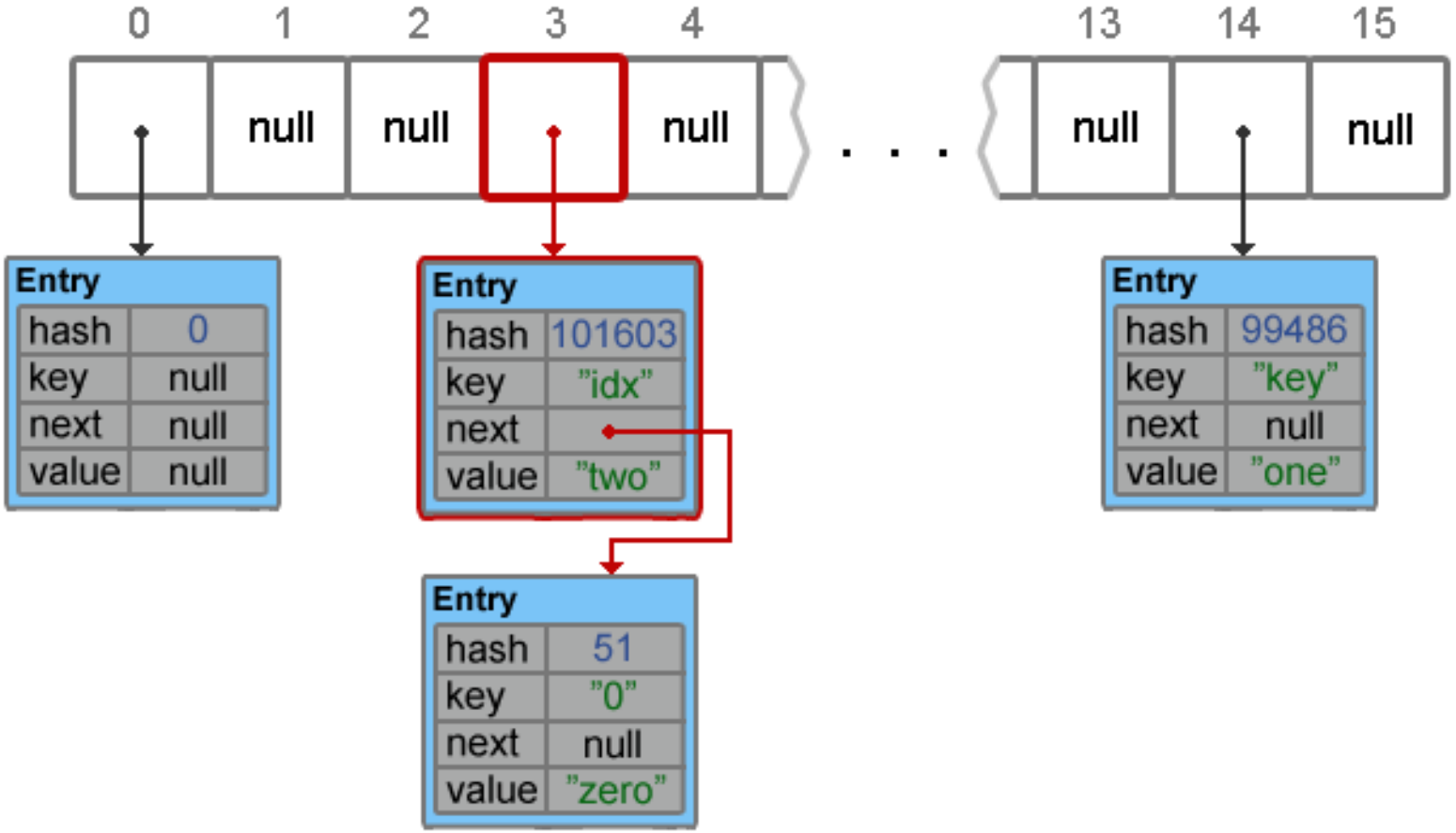
3) hashMap.put("key", "one");



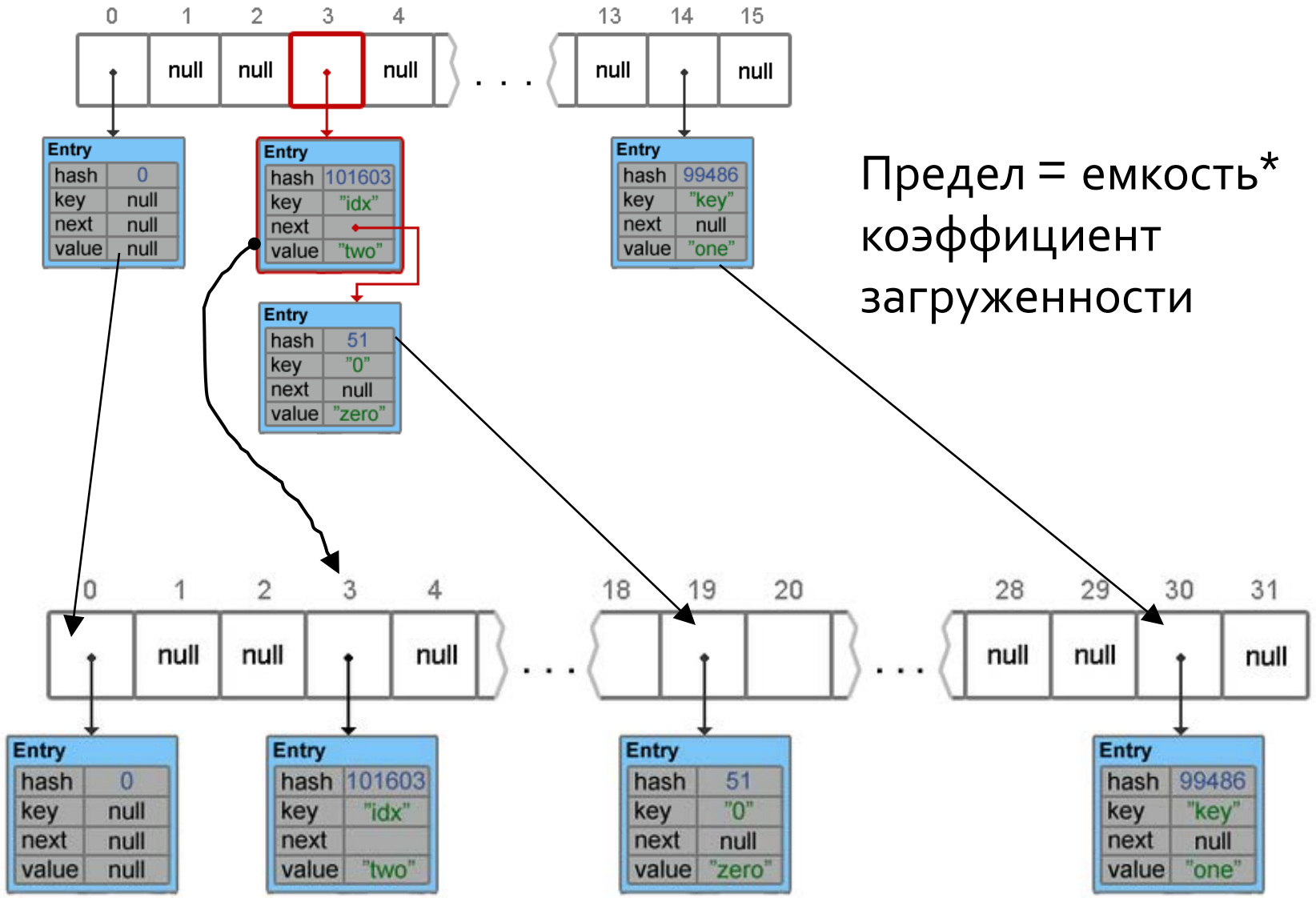
4) hashMap.put(**null**, **null**);



5) hashMap.put("idx", "two");



Фреймворк коллекций Java

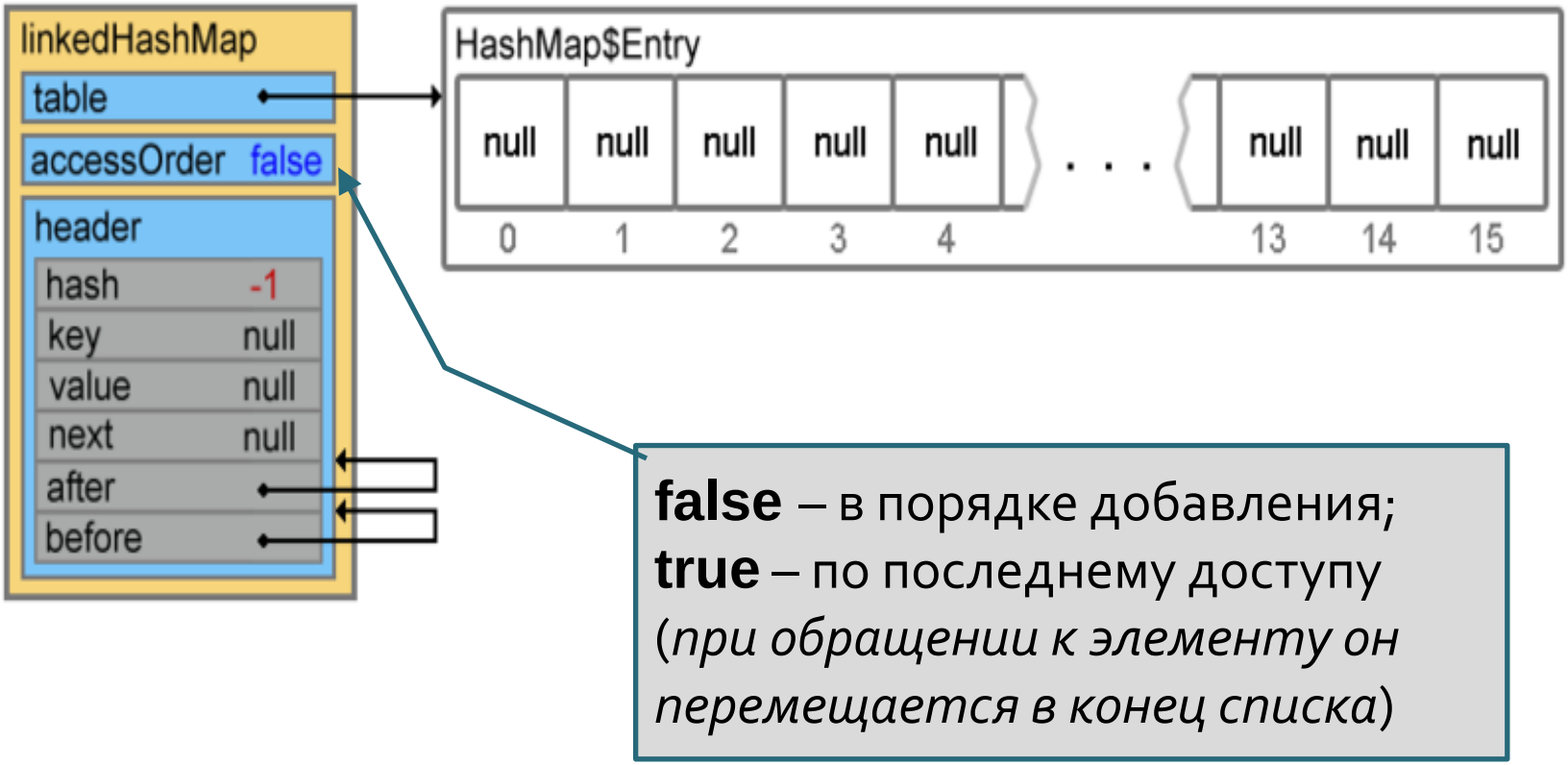


Предел = емкость* коэффициент загруженности

КЛАСС LinkedHashMap

- **LinkedHashMap** – это коллекция, которая поддерживает двунаправленный связный список, проходящей через все его записи:
  - **LinkedHashMap** определяет порядок итерации, который соответствует порядку, в котором ключи были вставлены в карту;
  - повторная вставка ключа, не влияет на порядок добавления;
  - метод *put(key,value)*, всегда вызывает метод *containsKey(key)* для проверки существования ключа (если ключ есть, то происходит замена значения).

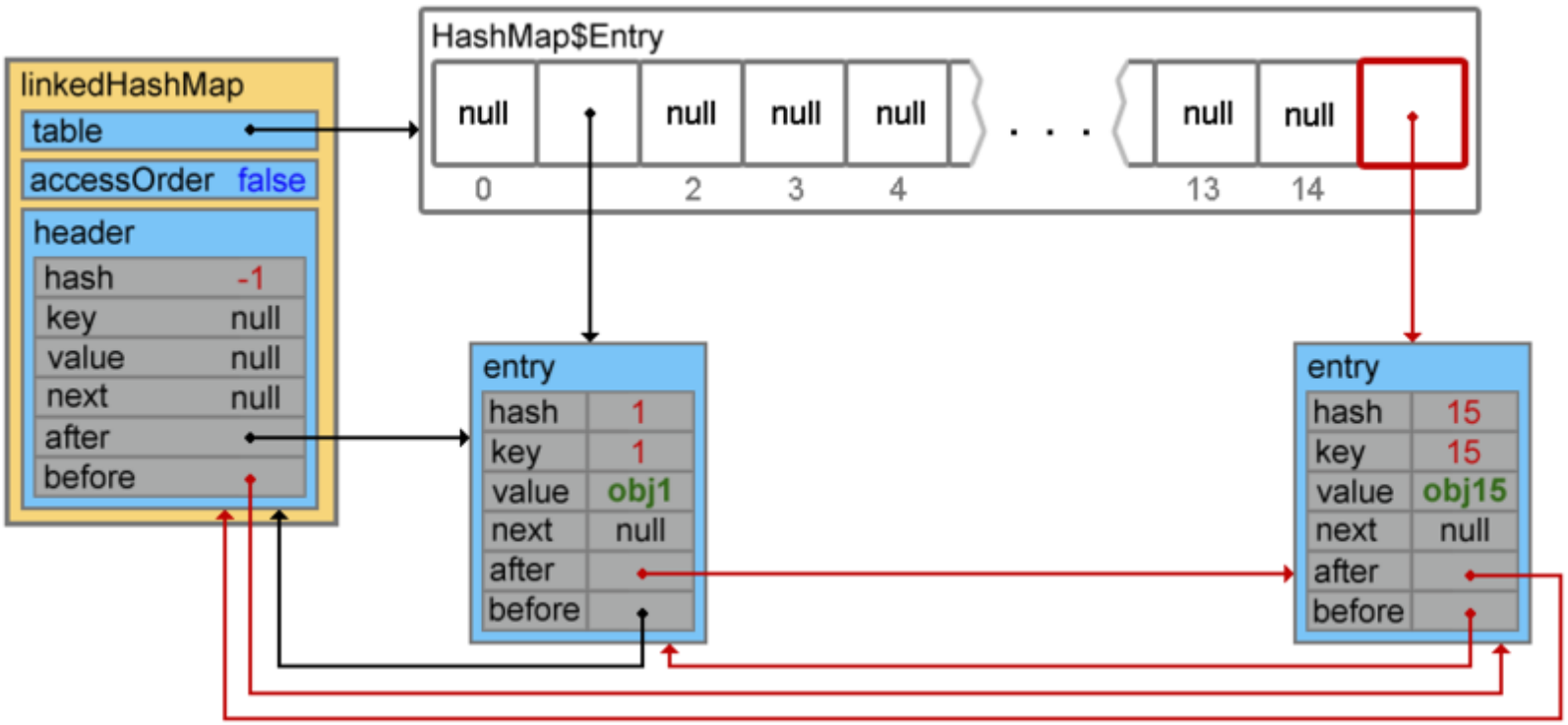
1) Map<Integer, String> hm = **new** LinkedHashMap<>();



false – в порядке добавления;
true – по последнему доступу (*при обращении к элементу он перемещается в конец списка*)

2) linkedHashMap.put(1, "obj1");

3) linkedHashMap.put(15, "obj15");



Пример 12:

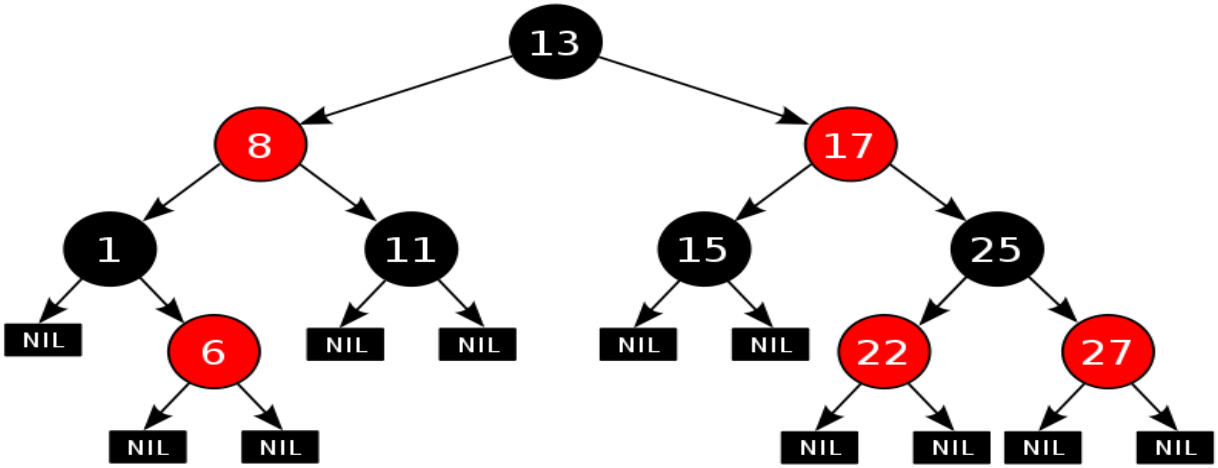
```
LinkedHashMap<String, Integer> hm = new
      LinkedHashMap<>();
hm.put("one", 1);
hm.put("two", 2);
hm.put("tree", 3);
hm.put("four", 4);
hm.put("five", 5);
Iterator<Map.Entry<String, Integer>> itr1 =
      hm.entrySet().iterator();
while (itr1.hasNext()) {
    Map.Entry<String,Integer> entry = itr1.next();
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
```

Вывод в консоли

```
one = 1
two = 2
tree = 3
four = 4
five = 5
```

КЛАСС TreeMap

- **TreeMap** является упорядоченной коллекцией.
  - по умолчанию коллекция сортируется по ключам с использованием принципа «натурального порядка»;
  - это поведение может быть изменено при помощи объекта **Comparator**, который указывается в качестве параметра при создании объекта **TreeMap**;
  - реализация основана на красно-чёрных деревьях.



Свойства красно-черных деревьев

- каждый узел является красным или черным;
- корневой узел всегда черный;
- все конечные узлы (пустые узлы - **null**) – черные;
- два дочерних узла каждого красного узла - черные (т.е. на соседних уровнях находятся узлы различного цвета);
- содержит одинаковое количество черных узлов на пути следования от корневого узла к любому листу.

Пример 13:

```
Map<String, Integer> hm = new TreeMap<>();
hm.put("ee", 5);
hm.put("cc", 3);
hm.put("aa", 1);
hm.put("bb", 2);
hm.put("dd", 4);
hm.put("ff", 6);
System.out.println(hm);
int x = hm.get("dd");
System.out.println(x);
```

Вывод в консоли

```
{aa=1, bb=2, cc=3, dd=4, ee=5, ff=6}
4
```

Пример 14:

В некую систему одновременно разрешен доступ двум пользователям. Контроль и управление.

```
public class DemoSecurity {
    public static void main(String[] args) {
        CheckRight.startUsing(2041, "Bill G.");
        CheckRight.startUsing(2420, "George B.");
        CheckRight.startUsing(2437, "Phillip K.");
        CheckRight.startUsing(2041, "Bill G.");
    }
}
```

```
class CheckRight {
    private static HashMap<Integer, String> map = new
                    HashMap<> ();
    public static void startUsing(int id, String name) {
            if (canUse(id)) {
                    map.put(id, name);
                    System.out.println("доступ разрешен");
            }else
                    System.out.println("в доступе отказано");
    }
    public static boolean canUse(int id) {
            final int MAX_NUM = 2;
            int currNum = 0;
            if ( !map.containsKey(id))
                     currNum = map.size();
            return currNum < MAX_NUM;
    }
}
```

Вывод в консоли

доступ разрешен
доступ разрешен
в доступе отказано
доступ разрешен

КЛАСС Collections

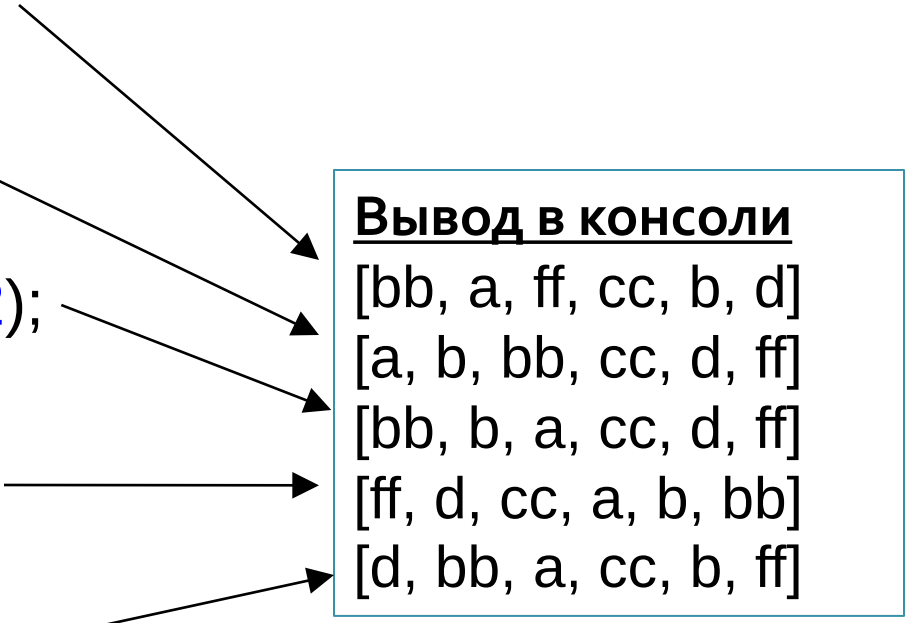
- Класс **java.util.Collections** состоит только из статических методов, которые работают на или возвращают коллекции;
- Он содержит алгоритмы, которые работают на коллекциях как "обертки" и возвращают новую коллекцию на базе указанной коллекции;
- Все методы этого класса бросают исключение типа **NullPointerException**, если коллекции или объекты класса, предоставляемые им, имеют значение **null**.

Методы класса Collection

- ***addAll***(Collection<? super T> c, T... elements) – добавляет все указанные элементы к указанной коллекции;
- ***sort***(List<T> list) – упорядочивает список по возрастанию;
- ***sort***(List<T> list, Comparator<? super T> c) – упорядочивает список в указанном порядке;
- ***swap***(List<?> list, int i, int j) – меняет местами элементы в указанных позициях указанного списка;
- ***reverse***(List<?> list) – располагает элементы списка в обратном порядке;
- ***shuffle***(List<?> list) – перемешивает элементы списка с использованием генератора случайных чисел по умолчанию.

Пример 15:

```
List<String> list = new ArrayList<String>();
Collections.addAll(list,"bb","a","ff","cc","b","d");
System.out.println(list);
Collections.sort(list);
System.out.println(list);
Collections.swap(list,0,2);
System.out.println(list);
Collections.reverse(list);
System.out.println(list);
Collections.shuffle(list);
System.out.println(list);
```



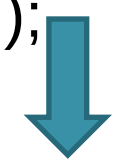
Фреймворк коллекций Java

- Методы, которые возвращают неподдающееся изменению представление указанной коллекции:

- ***unmodifiableCollection***(Collection<? extends T> c)
- ***unmodifiableList***(List<? extends T> list)
- ***unmodifiableMap***(Map<? extends K,? extends V> m)
- ***unmodifiableSet***(Set<? extends T> s)
- ***unmodifiableSortedMap***(SortedMap<K,? extends V> m)
- ***unmodifiableSortedSet***(SortedSet<T> s)

Пример 16:

```
List<String> myList = new ArrayList<String>();
myList.add("aaa");
myList.add("bbbb");
myList.add("ccccc");
List<String> readOnlyList =
        Collections.unmodifiableList(myList);
readOnlyList.add("fff");
```



```
Exception in thread "main"
java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add
```