Понятие коллекции

Коллекция — это группа однотипных элементов, которые представляют собой единое целое с базовым набором функций (последовательность, множество, очередь).

Все коллекции находятся в пакете java.util

Коллекция включает три составляющие:

✓ набор базовых интерфейсов (обеспечивают абстрактный тип данных для представления коллекции);

✓ набор классов для реализации базовых интерфейсов;

✓ набор алгоритмов для работы с коллекциями (полезны методы, которые решают тривиальные задачи, например: поиск, сортировка и перетасовка элементов коллекции).

В коллекциях используется обобщённое программирование — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания, названный в Java **generics** (дженерики), аналог "Шаблонов"(template) в C++ .

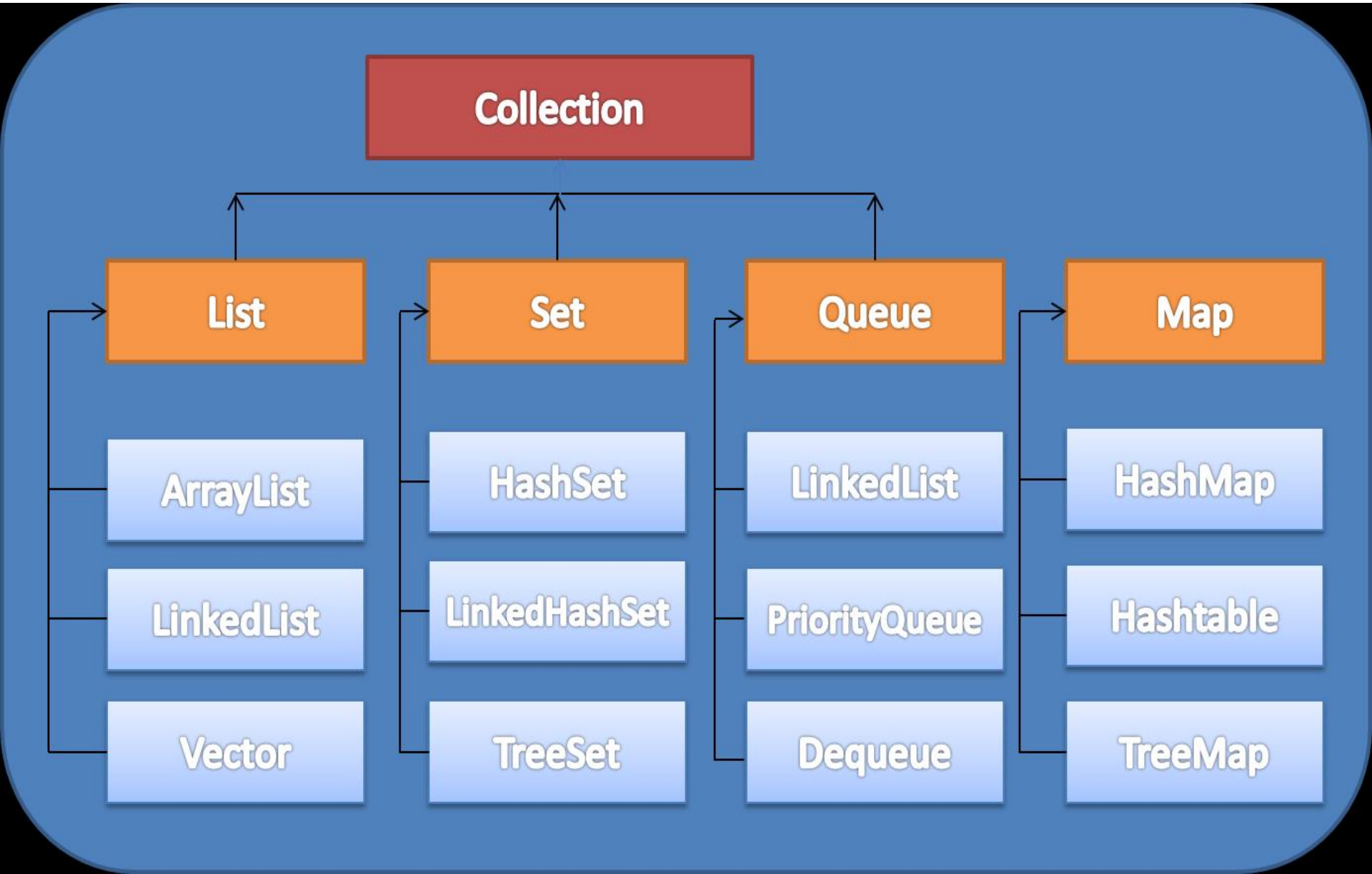
Свойства Generics

- Строгая типизация.
- Единая реализация.
- Отсутствие информации о типе.

Объявить generic можно: для классов, интерфейсов, методов

class Имя <T> { T используется в теле**}** T- имя параметра типа

Список коллекций в Java



Интерфейс Collection и его производные

Итератор - метод, позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации.

базовый интерфейс для всех коллекций

представляет функциональность простых списков

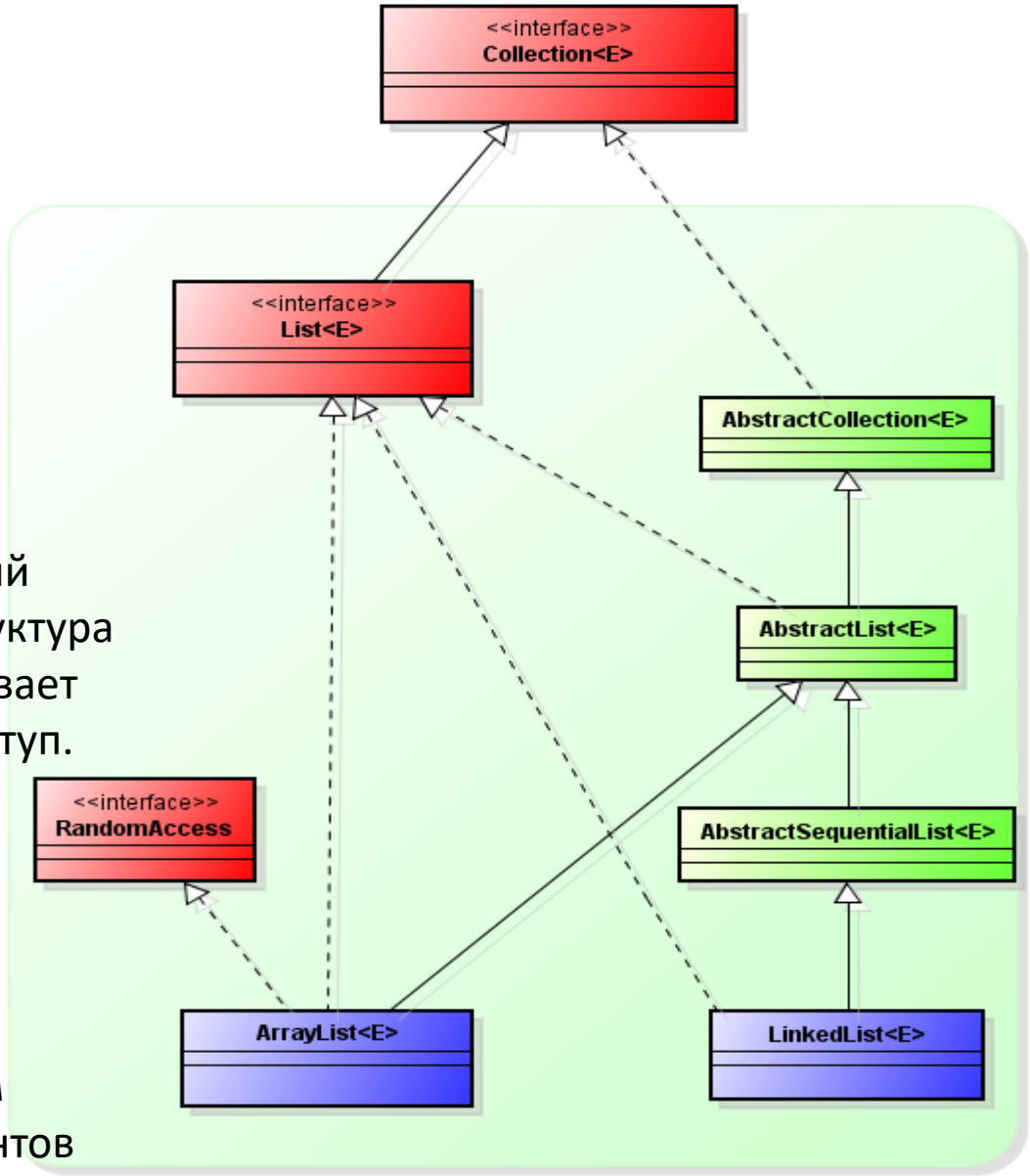
представляет функционал для структур данных в виде очереди

используется для хранения множеств

Методы интерфейса Collection

- **boolean add (E item)**: добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
- boolean addAll (Collection<? extends E> col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
- void clear (): удаляет все элементы из коллекции
- boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
- boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false
- **Iterator<E> iterator (**): возвращает объект Iterator для обхода элементов коллекции
- **boolean remove (Object item)**: возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
- boolean removeAll (Collection<?> col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
- boolean retainAll (Collection<?> col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
- **int size ()**: возвращает число элементов в коллекции
- **Object[] toArray ()**: возвращает массив, содержащий все элементы коллекции

Реализации интерфейса List



интерфейс, который указывает, что структура данных поддерживает произвольный доступ.

Массив с произвольным числом элементов

Двухсвязанный список

Конструкторы классов ArrayList и LinkedList

ArrayList () создает пустой список из 10 элементов.

ArrayList (Collection c) создает список массива, который инициализируется элементами коллекции с.

ArrayList (INT мощность) создает список массива, который имеет заданную начальную емкость.

LinkedList() создает пустой список;

LinkedList(Collection<? extends E> collection) создает список, в который добавляет все элементы коллекции collection (**<? extends E > – с любым типом и подтипом E)**

Вместо объекта ArrayList и LinkedList можно использовать интерфейс List

ArrayList <String> list = new ArrayList <String>();

List<String> list = new ArrayList<String>();

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

Методы классов ArrayList и LinkedList

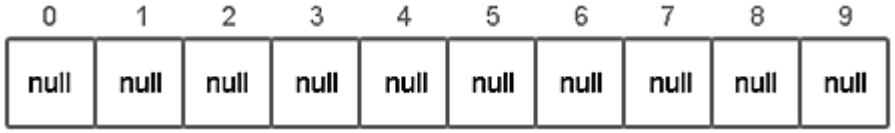
void add(int index, E obj) добавляет в список по индексу, **void add(E obj)** в конец списка
E get(int index): возвращает объект из списка по индексу
int indexOf(Object obj) возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1
int lastIndexOf(Object obj) возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1
E remove(int index) удаляет объект из списка по индексу, возвращая при этом удаленный объект
boolean remove(Object obj) удаляет объект из списка по значению
E set(int index, E obj) присваивает значение объекта obj элементу, который находится по индексу
void sort(Comparator<? super E> comp) сортирует список с помощью компаратора comp, для E или любого супертипа E, вплоть до Object
List<E> subList(int start, int end) получает набор элементов, которые находятся в списке между индексами start и end

LinkedList

addFirst() / offerFirst(): добавляет элемент в начало списка
addLast() / offerLast(): добавляет элемент в конец списка
removeFirst() / pollFirst(): удаляет первый элемент из начала списка
removeLast() / pollLast(): удаляет последний элемент из конца списка
getFirst() / peekFirst(): получает первый элемент
getLast() / peekLast(): получает последний элемент

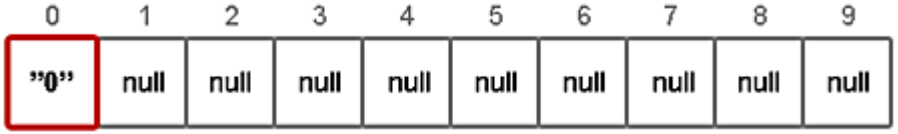
Представление ArrayList

ArrayList<String> list = new ArrayList<String>();

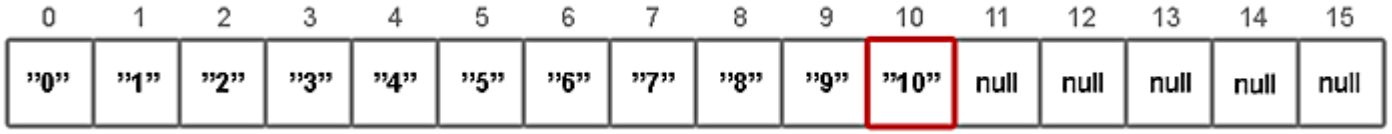


N=10

list.add("0"); Добавляем строку в конец массива

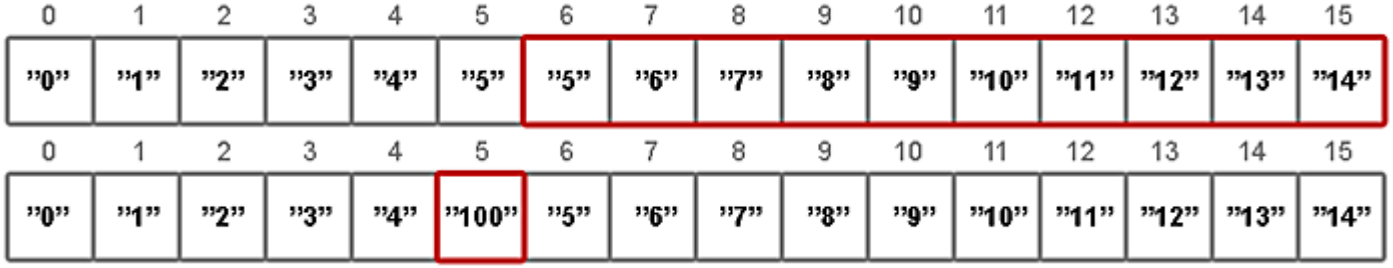


list.add("1"); … list.add("9"); list.add("10");



Создается новая копия
N=(N * 3) / 2 + 1

list.add(5, "100");



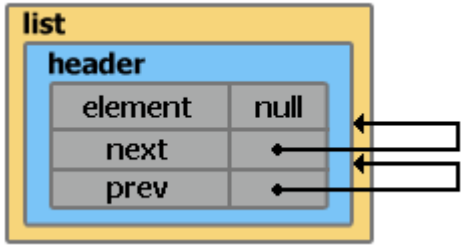
Удаление элемента (создается новая копия, N=N-1):

— по индексу;

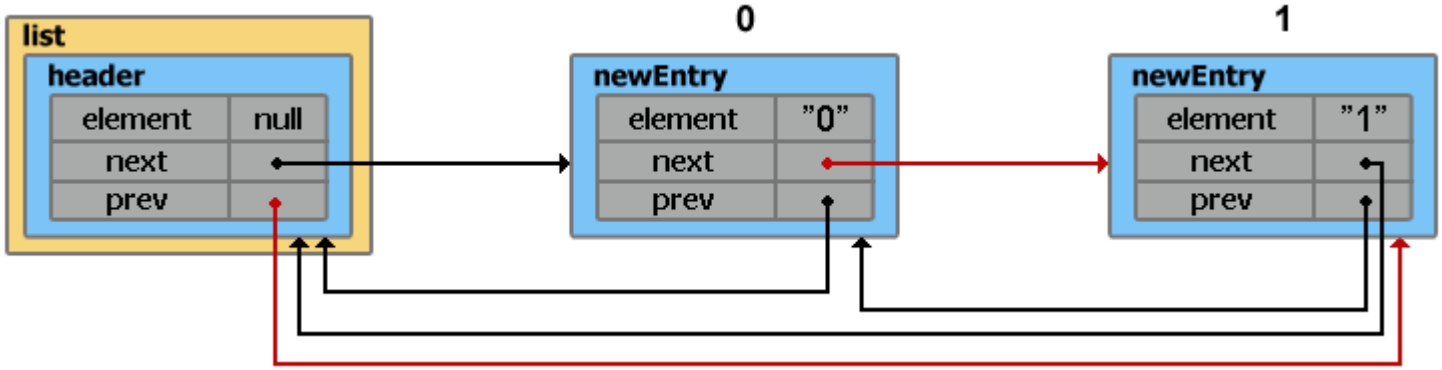
— по значению (первый найденный элемент)

Представление LinkedList

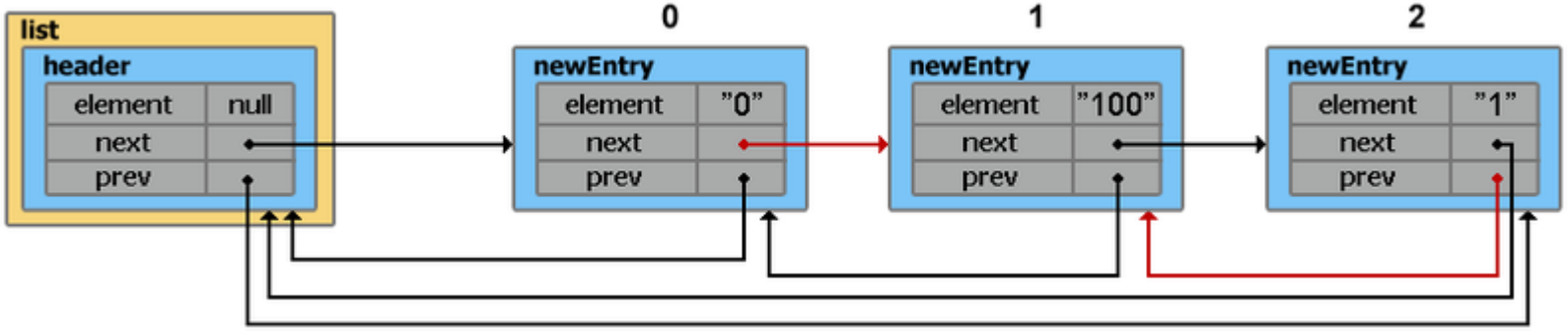
List<String> list = new LinkedList<String>();



list.add("0"); list. addLast("1"); Добавление в конец списка



list.add(1, "100"); Добавление по индексу (перед которым будет производиться вставка)



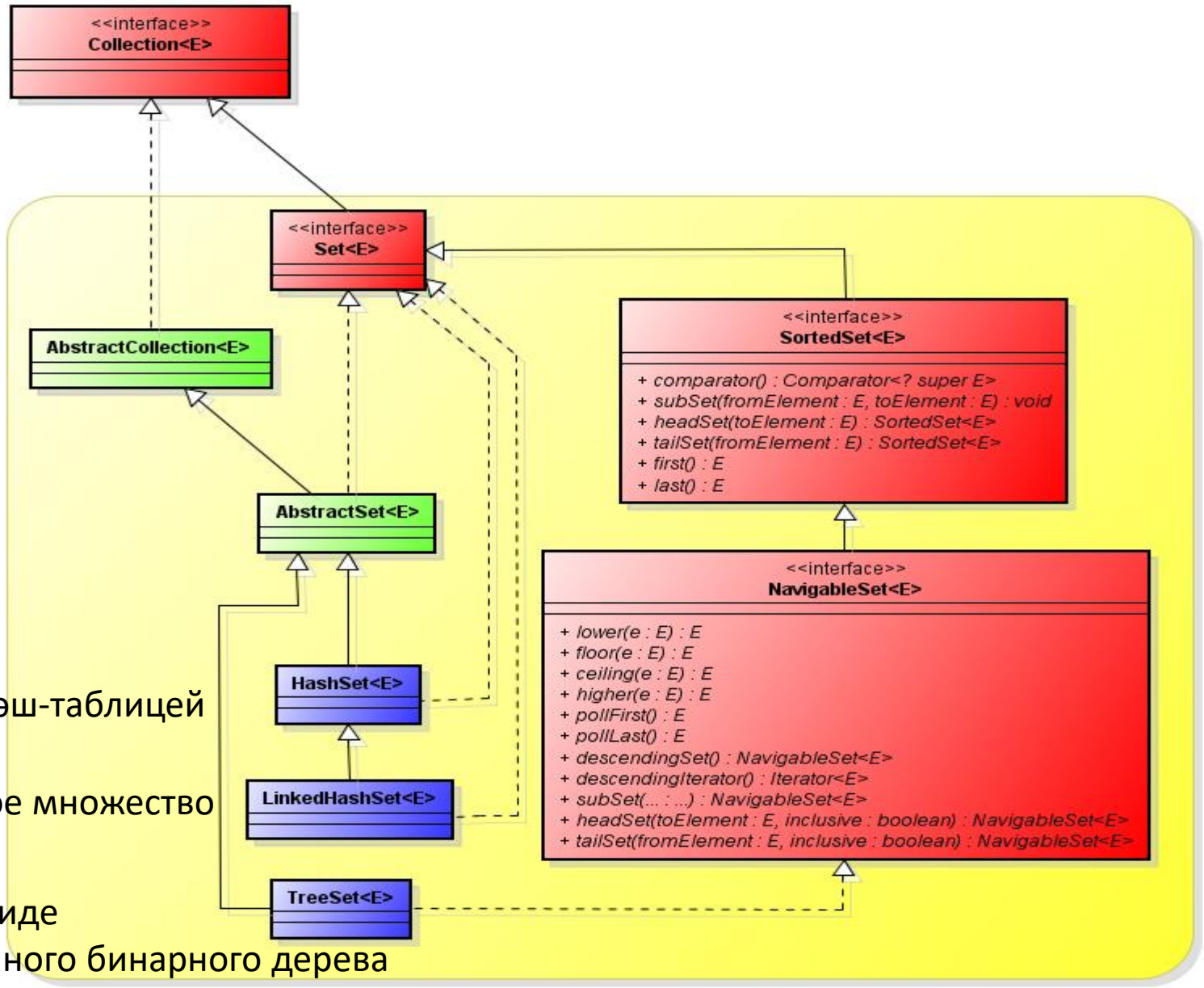
Сравнение ArrayList и LinkedList

Описание	Операция	ArrayList	LinkedList
Взятие элемента	get	Быстро	Медленно
Присваивание элемента	set	Быстро	Медленно
Добавление элемента	add	Быстро	Быстро
Вставка элемента	add(i, value)	Медленно	Быстро
Удаление элемента	remove	Медленно	Быстро

Если необходимо вставлять (или удалять) в середину коллекции много элементов, то лучше использовать LinkedList. Во всех остальных случаях – ArrayList.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку.

Реализации интерфейса Set

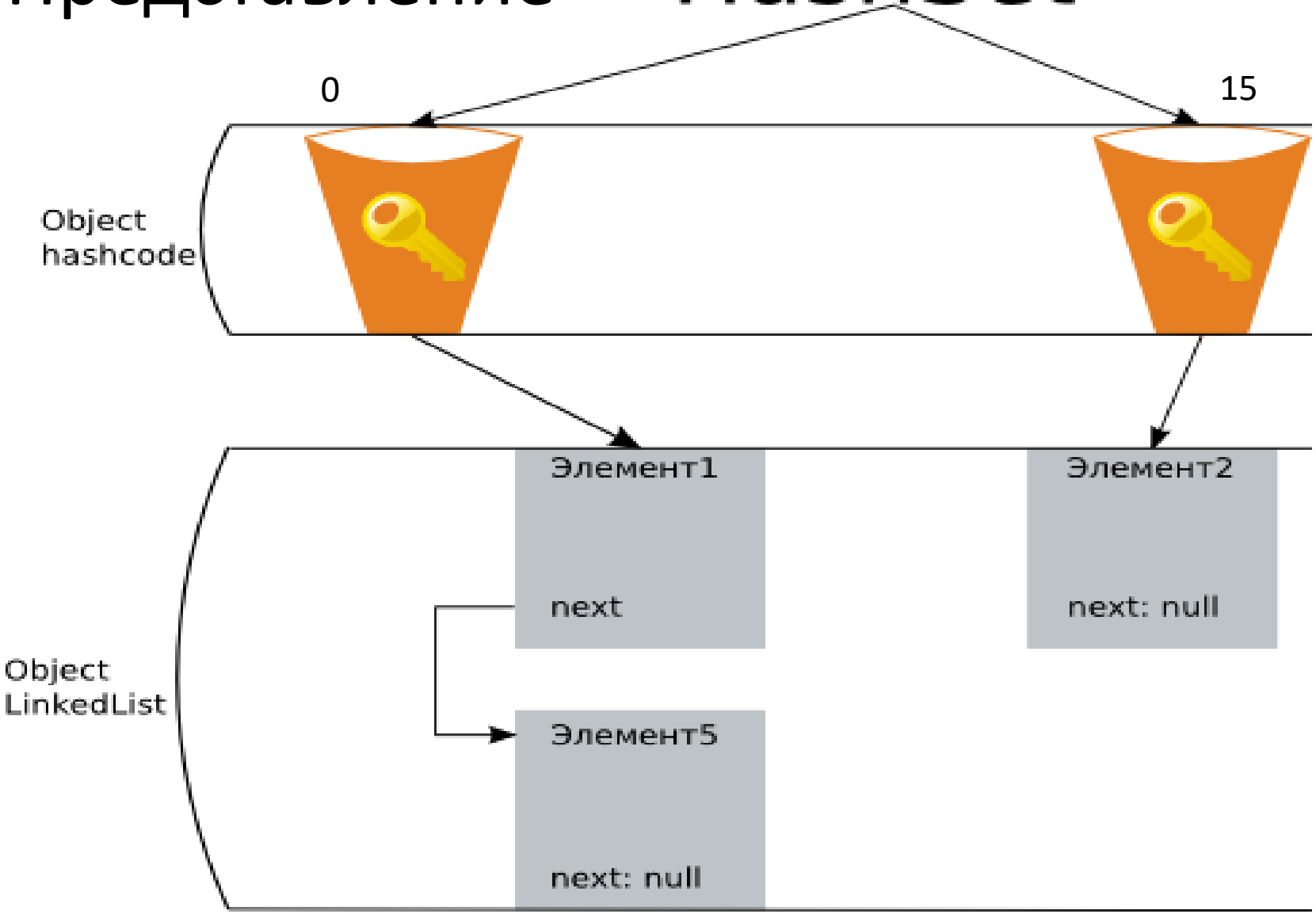


Множество с хэш-таблицей

Упорядоченное множество

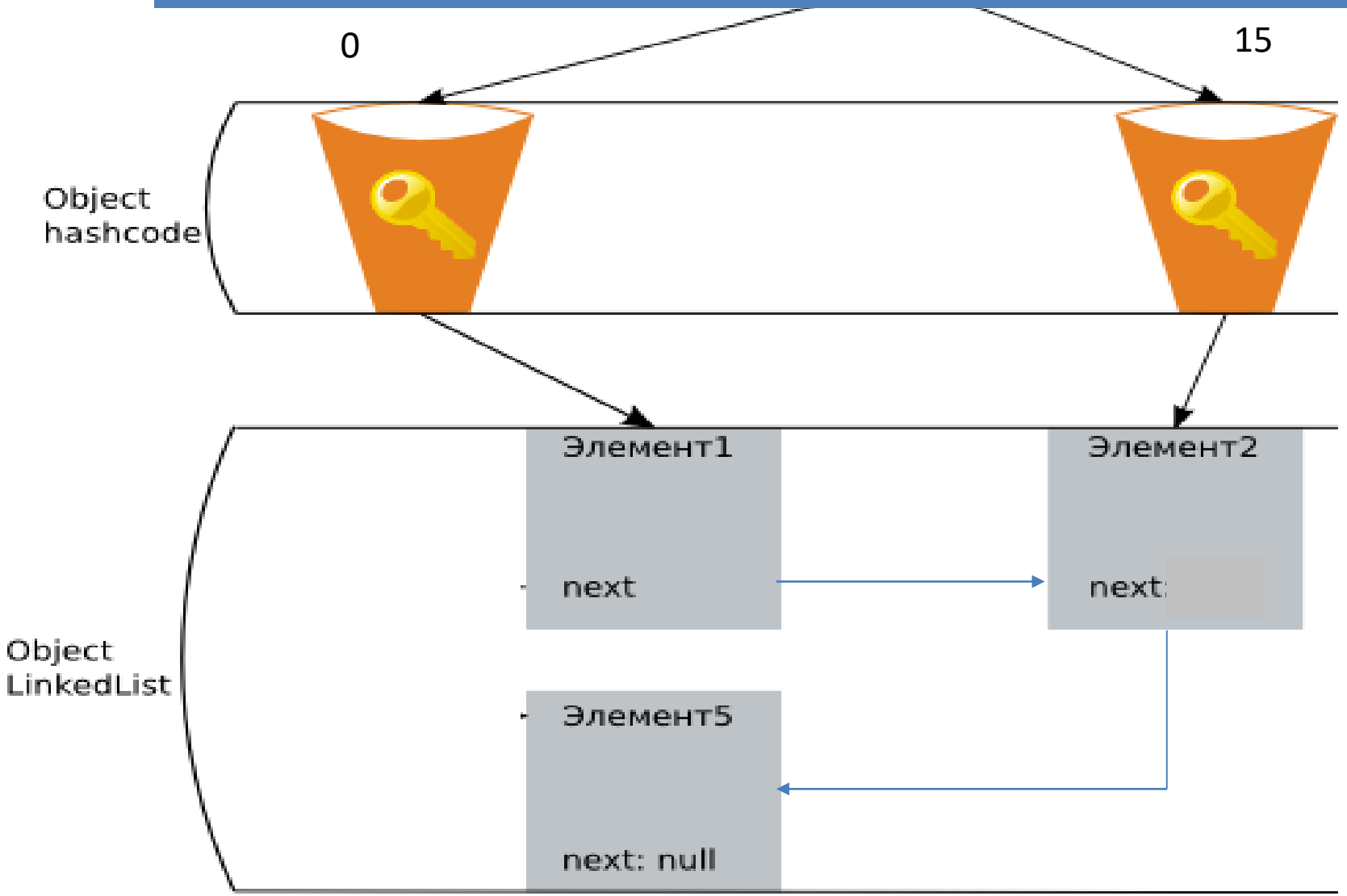
Множество в виде сбалансированного бинарного дерева

Представление HashSet



Множество на основе хэш-кода

Представление LinkedHashSet



Множество на основе хэш-кода с сохранением порядка обхода

Методы классов HashSet и LinkedHashSet

HashSet(): создает пустое множество

HashSet(Collection<? extends E> col): создает хеш-таблицу, в которую добавляет все элементы коллекции col

HashSet(int capacity): параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна **16**

add(E e) — добавляем элемент в множество, если такого там ещё нет. Возвращает true, если элемент добавлен

addAll(Collection c) — добавляет все элементы коллекции c (если их ещё нет)

clear() — удаляет все элементы множества

contains(Object o) — возвращает true, если элемент есть в множестве

containsAll(Collection c) — возвращает true, если все элементы содержатся в множестве

equals(Object o) — проверяет, одинаковы ли множества

hashCode() — возвращает hashCode

isEmpty() — возвращает true если в коллекции нет ни одного элемента

iterator() — возвращает интерфейс **Iterator** для перебора коллекции

remove(Object o) — удаляет элемент

removeAll(Collection c) — удаляет элементы, принадлежащие переданной коллекции

retainAll(Collection c) — удаляет элементы, не принадлежащие переданной коллекции

size() — количество элементов множества

toArray() — возвращает массив, содержащий элементы множества

toArray(T[] a) — также возвращает массив, но (в отличии от предыдущего метода, который возвращает массив объектов типа Object) возвращает массив объектов типа, переданного в параметре.

Класс **LinkedHashSet** расширяет класс **HashSet**, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.

Пример использования HashSet

```
import java.util.*;
public class HashSetDemo {
public static void main(String args[]) {
HashSet hs = new HashSet();
hs.add("B"); hs.add("A"); hs.add(“F"); hs.add("E");
    hs.add("C"); hs.add("D");
System.out.println(hs); } }
```

HashSet не поддерживает порядок своих элементов, а это значит, что элементы будут возвращены в любом порядке

[D, E, F, A, B, C]

Пример LinkedHashSet

```
import java.util.*;
public class LinkedHashSet Demo {
public static void main(String args[]) {
LinkedHashSet hs = new LinkedHashSet();
hs.add("B"); hs.add("A"); hs.add("D"); hs.add("E");
    hs.add("C"); hs.add("F");
System.out.println(hs);
Iterator iterator = hs.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");}}
[B, A, D, E, C, F]
```

Свойства TreeSet

1.TreeSet реализует интерфейс **SortedSet,** поэтому повторяющиеся значения не допускаются.
2.Объекты в TreeSet хранятся в отсортированном и возрастающем порядке.
3.TreeSet не сохраняет порядок вставки элементов, но элементы сортируются по ключам.
4.TreeSet не позволяет вставлять гетерогенные объекты. Он выдаст исключение classCastException во время выполнения, если попытается добавить гетерогенные объекты.
5.TreeSet служит отличным выбором для хранения больших объемов отсортированной информации, к которой предполагается быстрый доступ из-за более быстрого доступа и времени поиска.
6.TreeSet — это, по сути, реализация самобалансирующегося бинарного дерева поиска, такого как **Red-Black Tree** . Поэтому такие операции, как добавление, удаление и поиск, занимают O (Log n) время A такие операции, как печать n элементов в отсортированном порядке, занимают O (n) времени.

МетодыTreeSet

reeSet(): создает пустое сбалансированное дерево

TreeSet(Collection<? extends E> col): создает дерево, в которое добавляет все элементы коллекции col со значениями типа E или любого из подтипов E

TreeSet(SortedSet <E> set): создает дерево, в которое добавляет все элементы типа E сортированного набора set

TreeSet(Comparator<? super E> comparator): создает пустое дерево, где все добавляемые элементы типа E или любого супертипа E впоследствии будут отсортированы компаратором.

В дополнение к методам, которые есть в HashSet, TreeSet имеет ряд методов, связанных с упорядоченностью множества

public SortedSet subSet(Object fromElement, Object toElement) : возвращает подмножество в заданном интервале объектов (toElement не входит в это подмножество)

public SortedSet headSet(Object toElement) : аналогично предыдущему, но с начала исходного множества — до элемента toElement, исключая его.

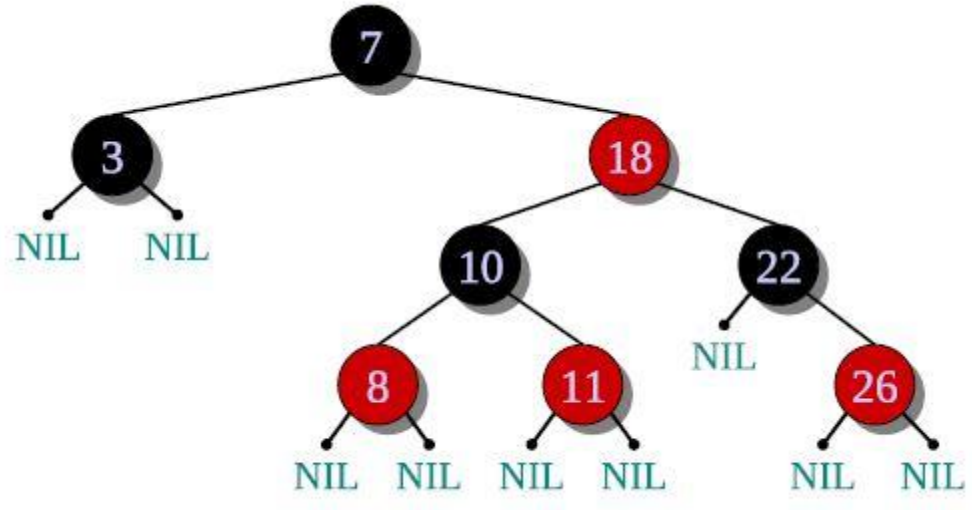
public SortedSet tailSet(Object fromElement): аналогично subset, но начиная с элемента fromElement до конца множества.

public Object first() :первый элемент множества

public Object last() :последний элемент множества

Пример TreeSet

Объекты сохраняются в отсортированном порядке по возрастанию.




```
import java.util.*;
public class TreeSetDemo {
public static void main(String args[]) {
TreeSet ts = new TreeSet();
ts.add(7); ts.add(18); ts.add(3); ts.add(10); ts.add(22); ts.add(26); ts.add(11); ts.add(8);
System.out.println(ts);
} }
[3, 7, 8, 10, 11, 18, 22, 26]
```

Классы *HashSet, TreeSet, LinkedHashSet*

```
Set<String> hs = new HashSet<String>();
hs.add("Иванов");
hs.add("Сидоров");
hs.add("Петров");
hs.add("Иванов");
hs.add("Алексеев");
for (String s : hs) System.out.println(s);
```

Иванов
Петров
Алексеев
Сидоров

```
Set<String> hs = new LinkedHashSet<String>();
hs.add("Иванов");
hs.add("Сидоров");
hs.add("Петров");
hs.add("Иванов");
hs.add("Алексеев");
for (String s : hs) System.out.println(s);
```

Иванов
Сидоров
Петров
Алексеев

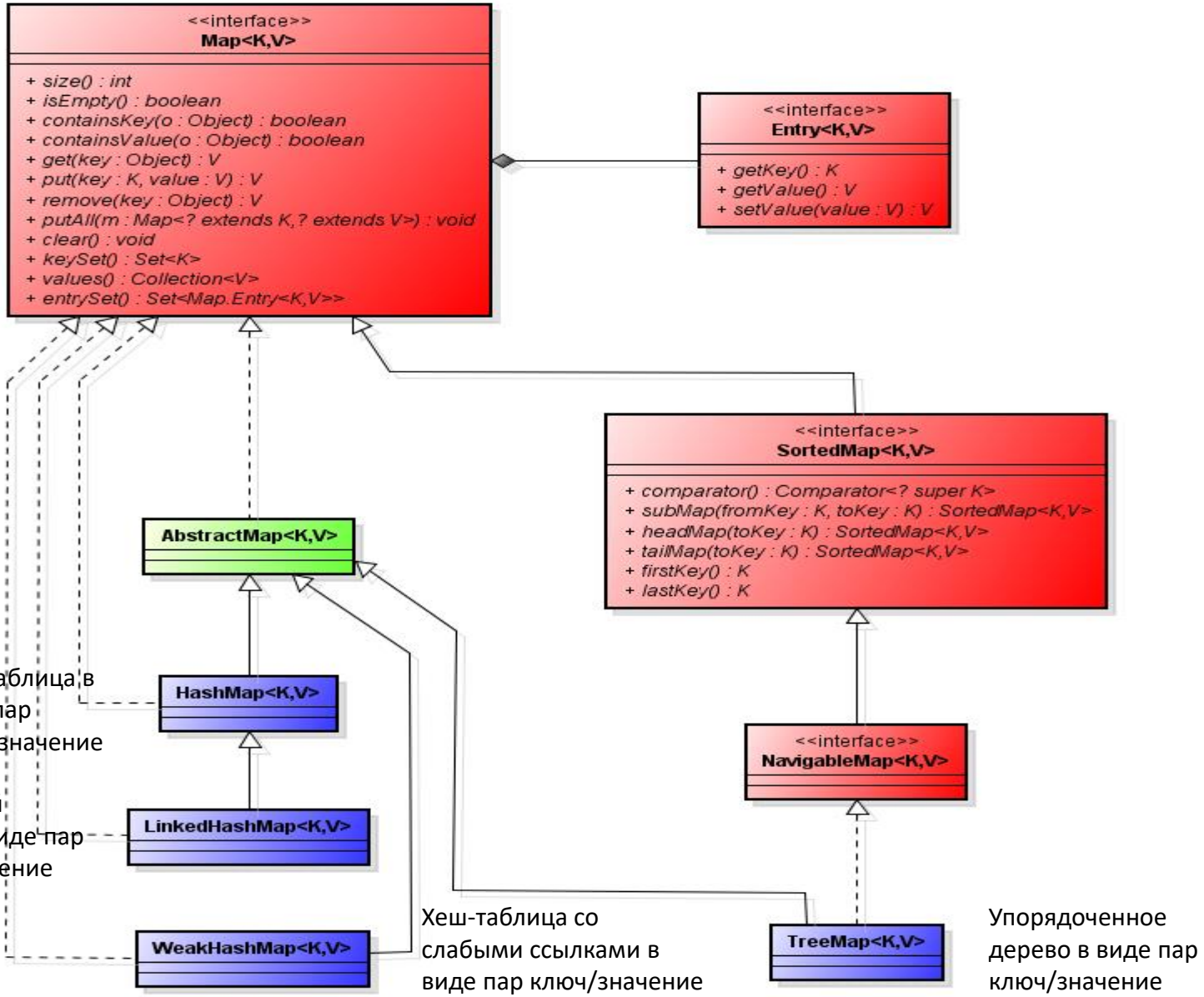
```
Set<String> hs = new TreeSet<String>();
hs.add("Иванов");
hs.add("Сидоров");
hs.add("Петров");
hs.add("Иванов");
hs.add("Алексеев");
for (String s : hs) System.out.println(s);
```

Алексеев
Иванов
Петров
Сидоров

Отличия HachSet, LinkedHachSet и TreeSet

HachSet	LinkedHachSet	TreeSet
использует хэш-память для хранения элементов	использует хэш-память и списки для хранения элементов	использует бинарное дерево для хранения элементов
не поддерживает порядок элементов	поддерживает порядок вставки элементов	по умолчанию поддерживает естественный порядок сортировки элементов
дает лучшую производительность чем, LinkedHachSet и TreeSet	производительность находится между HachSet и TreeSet	дает меньшую производительность чем, HachSet и LinkedHachSet
разрешает один нулевой элемент	разрешает один нулевой элемент	не разрешает даже одного нулевого элемента

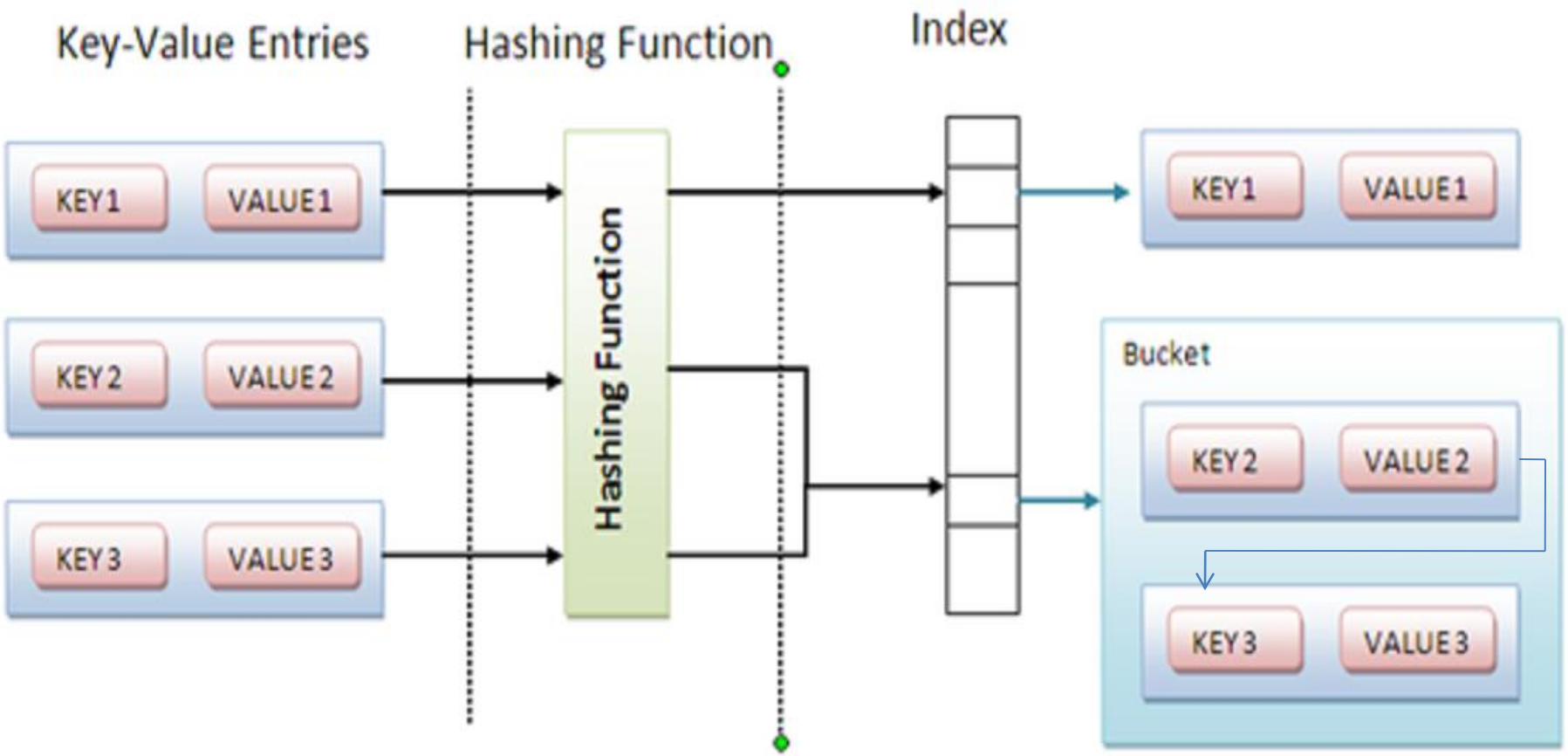
Интерфейс Map



Методы интерфейса Map

void clear(): очищает коллекцию
boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k
boolean containsValue(Object v): возвращает true, если коллекция содержит значение v
Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции.
Все элементы представляют объект Map.Entry
boolean isEmpty: возвращает true, если коллекция пуста
V get(Object k): возвращает значение объекта, ключ которого равен k.
Если такого элемента не окажется, то возвращается значение null
V getOrDefault(Object k, V defaultValue): возвращает значение объекта,
ключ которого равен k.
Если такого элемента не окажется, то возвращается значение defaultVlue
V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v.
Если в коллекции уже есть объект с подобным ключом, то он перезаписывается.
После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null
V putIfAbsent(K k, V v): помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом.
Set<K> keySet(): возвращает набор всех ключей отображения
Collection<V> values(): возвращает набор всех значений отображения
int size(): возвращает количество элементов коллекции

Хеширование



Вычисление хеш-кода ключа

```
static int hash(int h) {
h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
```

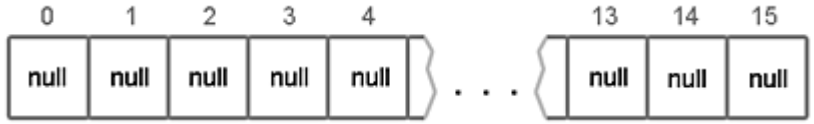
По хэш-коду ключа находится номер элемента массива записей длиной length

```
static int indexFor(int h, int length) {
    return h & (length-1);
```

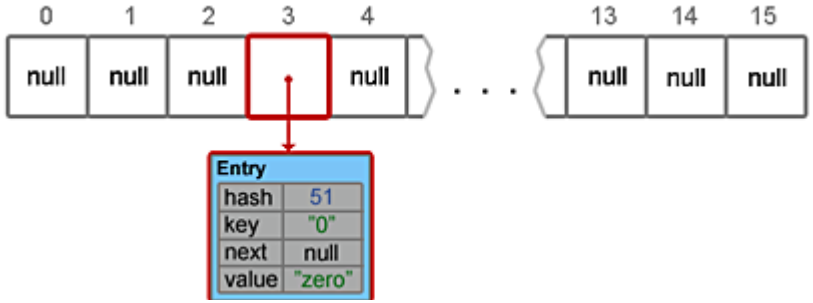
Представление HashSet (HashMap)

hs = new HashMap<String, String>();

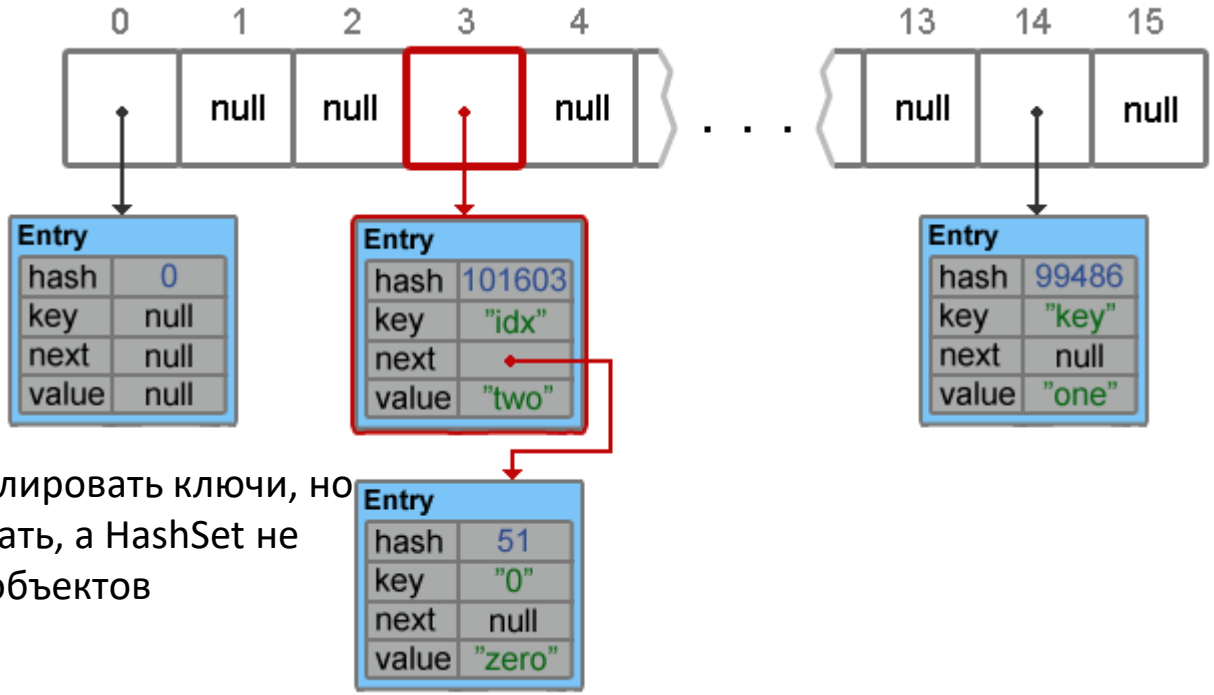
Значение рассматривается,
как ключ, по умолчанию 16 элементов



hs.put("0", "zero");



hs . put("key", "one");
hs . put(null, null);
hs . put("idx", "two");



HashMap не позволяет дублировать ключи, но значения можно дублировать, а HashSet не разрешает дублирование объектов

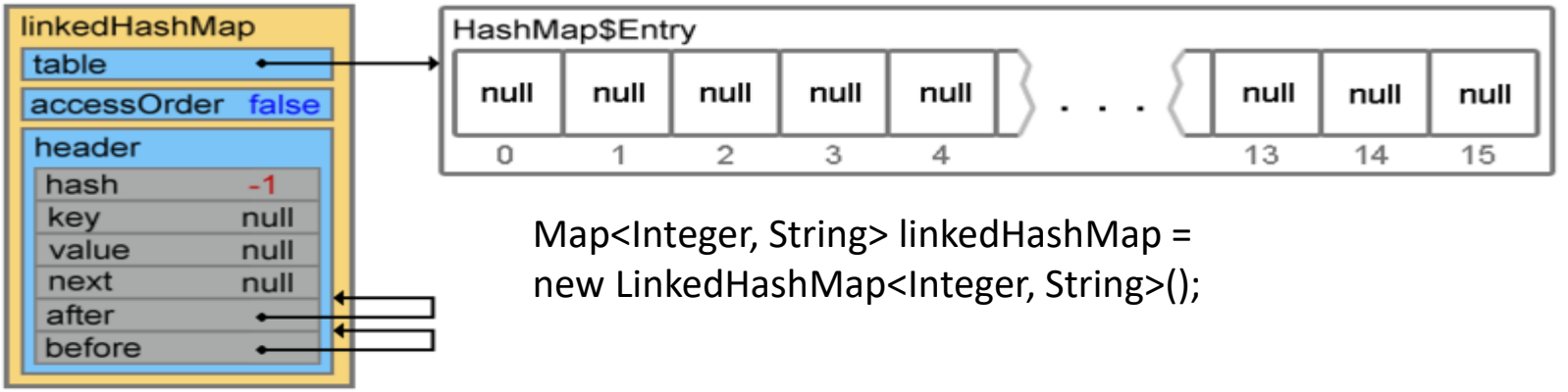
Пример использования методов HashMap

```
public class TestHashMap {
public static void main(String[] a) {
Map<String, String> map = new HashMap<String, String>();
map.put("key1", "value1"); map.put("key2", "value2"); map.put("key3", "value3");
System.out.println(map);
Map<String, String> map2 = new HashMap<String, String>();
map2.put("key4", "value4"); map2.put("key5", "value5"); map2.put("key6", "value6");
// Добавление набора данных
map.putAll(map2);
// Удаление объекта по ключу
map.remove("key5");
// Размер набора
System.out.println("Размер набора данных : " + map.size());
// Поиск по ключу
String exists = (map.containsKey("key2")) ? "найден" : "не найден";
System.out.println("Объект с ключом 'key2' " + exists);
// Поиск по значению
exists = (map.containsValue("value2")) ? "найден" : "не найден";
System.out.println("Объект со значением 'value2' " + exists);
// Перебор значений
Set<Map.Entry<String, String>> set = map.entrySet(); // возвращает набор ключ-значений
 for (Map.Entry<String, String> me : set) { System.out.print("ключ : " + me.getKey() + ", значение = " +
      me.getValue()); } // цикл foreach
map.clear(); } // Очистка объекта
```

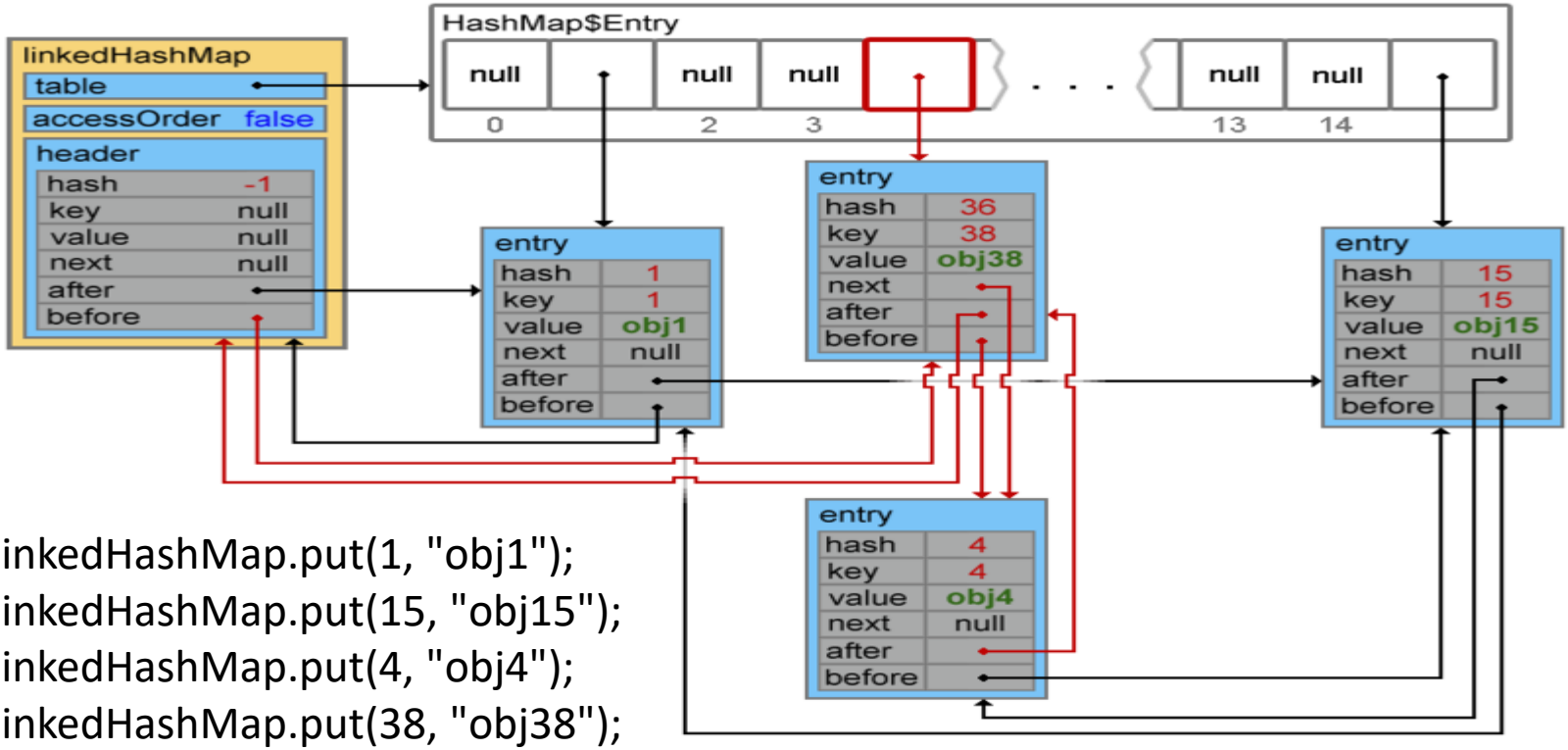
Результат работы программы :

```
key3=value3, key2=value2, key1=value1
Размер набора данных : 5
Объект с ключом 'key2' найден
Объект со значением 'value2' найден
Список записей набора :
ключ : key4, значение = value4
ключ : key3, значение = value3
ключ : key6, значение = value6
ключ : key2, значение = value2
ключ : key1, значение = value1
```

Представление LinkedHashSet (LinkedHashMap)

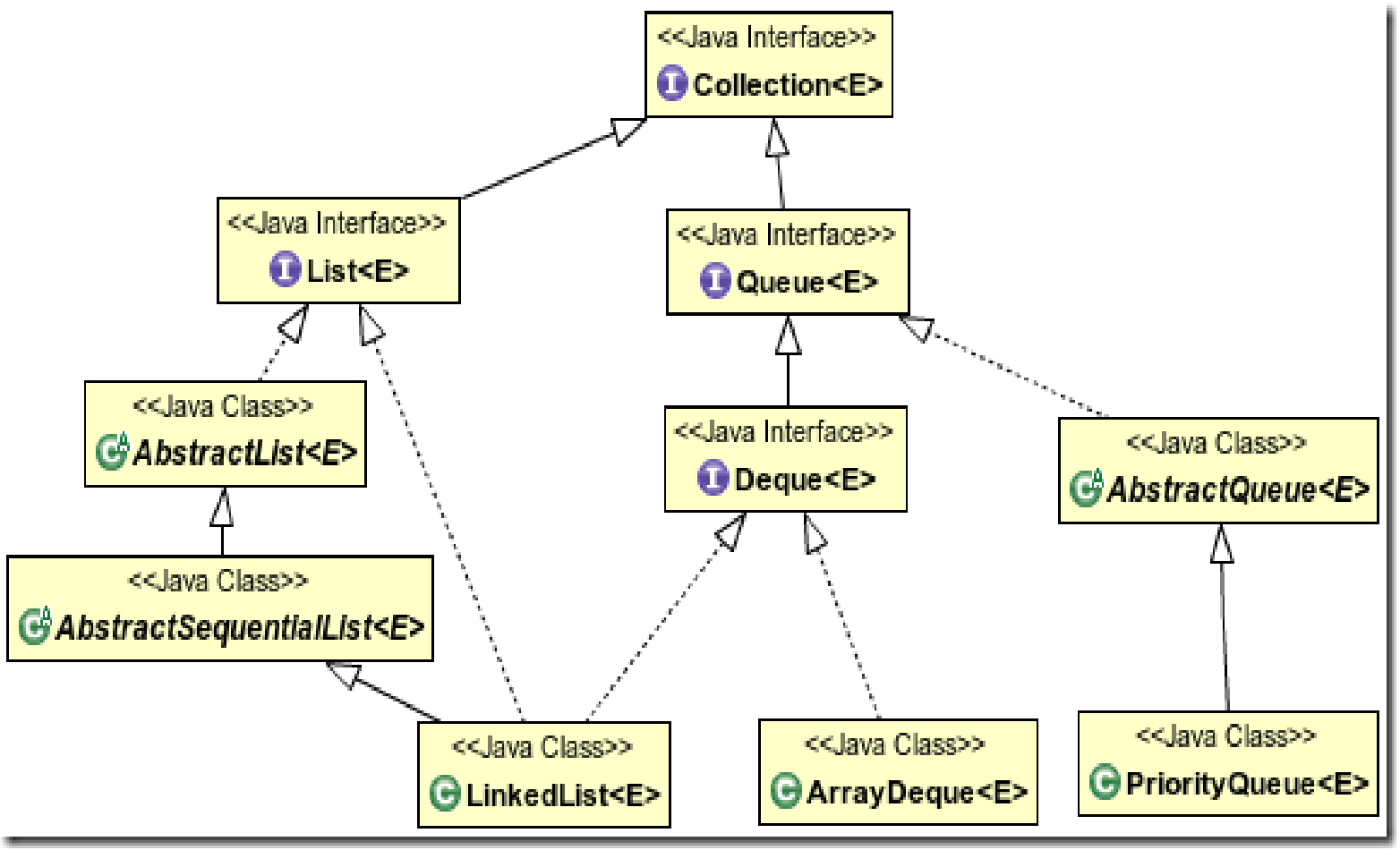


Map<Integer, String> linkedHashMap =
new LinkedHashMap<Integer, String>();




```
linkedHashMap.put(1, "obj1");
linkedHashMap.put(15, "obj15");
linkedHashMap.put(4, "obj4");
linkedHashMap.put(38, "obj38");
```

Реализации интерфейса Queue



Двухсвязный список

Упорядоченная последовательность в виде циклического массива

Приоритетная очередь с сортировкой

Конструкторы PriorityQueue

Элементы упорядочены по умолчанию в естественном порядке или же отсортированы с помощью компаратора. Голова этой очереди является наименьшим элементом по отношению к указанному порядку.

PriorityQueue() создает PriorityQueue с начальной емкостью по умолчанию (11) , который сортирует свои элементы в соответствии с их естественным порядком.

PriorityQueue(Collection<? extends E> c) создает PriorityQueue, содержащий элементы в указанной коллекции.

PriorityQueue(int initialCapacity) создает PriorityQueue с указанной начальной емкостью, которая упорядочивает ее элементы в соответствии с их естественным порядком.

PriorityQueue(int initialCapacity, Comparator<? super E> comparator)

создает PriorityQueue с указанной начальной емкостью, которая упорядочивает ее элементы в соответствии с указанным компаратором.

PriorityQueue(PriorityQueue<? extends E> c) создает PriorityQueue, содержащий элементы в указанной очереди приоритета.

PriorityQueue(SortedSet<? extends E> c) создает PriorityQueue, содержащий элементы в указанном наборе отсортированные.

Методы PriorityQueue

boolean add(E e) вставляет заданный элемент с учетом сортировки (генерирует исключение, если операция не удалась).

void clear() удаляет все элементы из очереди.

Comparator<? super E> comparator() возвращает компаратор, используемый для упорядочения элементов в очереди, или нулевое значение, если эта очередь сортируется в соответствии с естественным порядком ее элементов.

boolean contains(Object o) возвращает истину, если очередь содержит заданный элемент.

Iterator<E> iterator() возвращает начальный элемент очереди.

boolean offer(E e) вставляет заданный элемент в очередь с учетом сортировки (генерирует false, если операция не удалась). .

E peek() извлекает, но не удаляет, первый элемент очереди, или возвращает нулевое значение, если эта очередь пуста.

E poll() извлекает и удаляет первый элемент очереди, или возвращает нулевое значение, если эта очередь пуста.

boolean remove(Object o) удаляет один экземпляр указанного элемента из очереди, если он присутствует.

int size() возвращает количество элементов в этой коллекции.

Object[] toArray() возвращает массив, содержащий все элементы в этой очереди.

<T> T[] toArray(T[] a) возвращает массив, содержащий все элементы в этой очереди; тип выполнения возвращаемого массива является то, что из указанного массива.

Пример PriorityQueue

```
import java.util.*;
        public class PriorityQueueDemo {
        public static void main(String[] args) {
PriorityQueue<Integer> priorityQueue = new PriorityQueue<Integer>();
Random rand = new Random(30); //генератор случайных чисел с начальным числом
for(int i = 0; i < 10; i++)
priorityQueue.add(rand.nextInt(i + 10)); //вставляет с учетом сортировки случайное числа в диапазоне 0 – (i + 10)
while( ! priorityQueue.isEmpty() ) {
System.out.print( priorityQueue.remove()+" " ); // печатает и удаляет первые элементы очереди
}
System.out.println();
// создаем список из элементов массива
List<Integer> ints = Arrays.asList(25, 22, 20, 18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
 System.out.println(ints);
priorityQueue = new PriorityQueue<Integer>(ints); //помещает массив в очередь с учетом сортировки
while( ! priorityQueue.isEmpty() ){ System.out.print( priorityQueue.remove()+" " );
}
System.out.println();
// создает очередь с обратным упорядочиванием, по умолчанию Comparator.naturalOrder()
priorityQueue = new PriorityQueue<Integer>(ints.size(), Collections.reverseOrder());
priorityQueue.addAll(ints);
while( ! priorityQueue.isEmpty() ){ System.out.print( priorityQueue.remove()+" " );
}}}
```

```
3 3 5 5 6 6 7 8 8 13
[25, 22, 20, 18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25]
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
```

Использование интерфейса компаратор

В интерфейсе **Comparator** объявлен метод **compare** (Object obj1, Object obj2), который позволяет сравнивать между собой два объекта. На выходе метод возвращает значение 0, если объекты равны, положительное значение или отрицательное значение, если объекты не тождественны.

Метод может вызвать исключение **ClassCastException**, если типы объектов не совместимы при сравнении.

```
//Анонимный класс компаратора
public static Comparator<Integer> idComparator = new Comparator<Integer>(){
public int compare(Integer o1, Integer o2) {
if( o1 > o2 ){ return -1; }
if( o1 < o2 ){ return 1; }
return 0;
}
};
Queue<Integer> intQueue = new PriorityQueue<>(10, idComparator);
intQueue.add(3);
intQueue.add(1);
intQueue.add(4);
while( !intQueue.isEmpty() ){ System.out.println( intQueue.remove() ); }
```

Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

4 3 1

Сравнение коллекций

Коллекция	Упорядочивание	Произвольный доступ Random Access	Ключ-значение	Дубликат Элементы	Нулевой элемент	Потоко безопасность
ArrayList	Да	Да	Нет	Да	Да	Нет
LinkedList	Да	Нет	Нет	Да	Да	Нет
HashSet	Нет	Нет	Нет	Нет	Да	Нет
TreeSet	Да	Нет	Нет	Нет	Нет	Нет
HashMap	Нет	Да	Да	Нет	Да	Нет
TreeMap	Да	Да	Да	Нет	Нет	Нет
Vector	Да	Да	Нет	Да	Да	Да
Hashtable	Нет	Да	Да	Нет	Нет	Да
Properties	Нет	Да	Да	Нет	Нет	Да
Stack	Да	Нет	Нет	Да	Да	Да
CopyOnWriteArrayList	Да	Да	Нет	Да	Да	Да
ConcurrentHashMap	Нет	Да	Да	Нет	Нет	Да
CopyOnWriteArraySet	Нет	Нет	Нет	Нет	Да	Да

http://javastudy.ru/interview/collections/