Контейнеры объектов

Библиотека утилит *Java* (***java.util.****) содержит достаточно полный набор классов контейнеров.

Контейнеры обладают весьма изощренными возможностями для хранения объектов и работы с ними, и с их помощью удается решить огромное количество задач.

```
import java.util.*;
class MyData{ int i; int j;}
class MyData2{ double d;boolean b;}
public class Test1 {
  public static void main(String[] args){
  List<MyData> l = new ArrayList<MyData>();
  List l1 = new ArrayList();
  MyData md = new MyData();
  MyData2 md2 = new MyData2();
  l.add(md);
 // l.add(md2);
  l1.add(md);
  l1.add(md2);}}
```

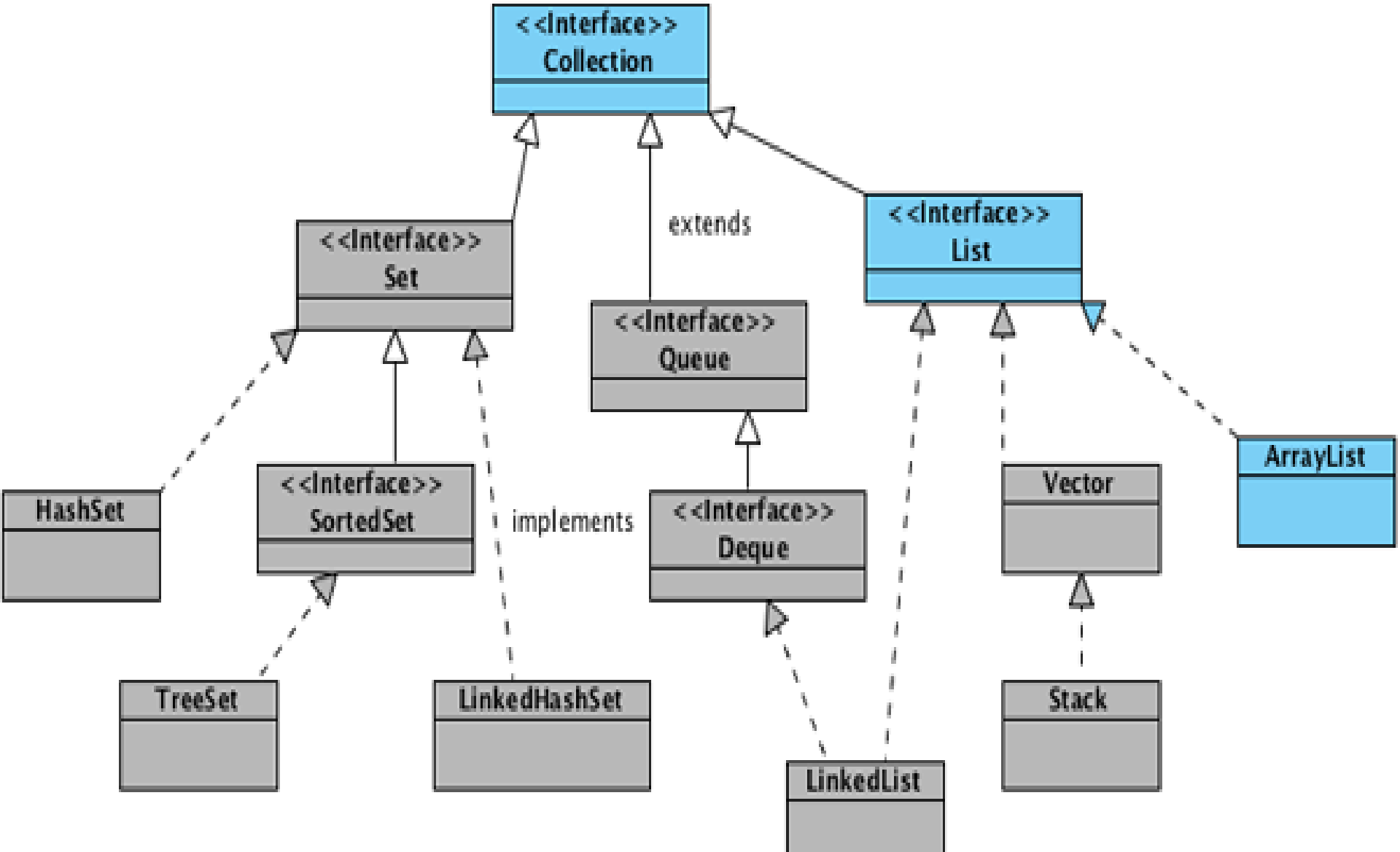
Контейнеры объектов

В библиотеке контейнеров *Java* проблема хранения объектов делится на две концепции, выраженные в виде базовых интерфейсов библиотеки:

Коллекция: группа отдельных элементов, сформированная по некоторым правилам. Класс ***List*** (список) хранит элементы в порядке вставки, в классе ***Set*** (множество) нельзя хранить повторяющиеся элементы, а класс ***Queue*** (очередь) выдает элементы в порядке, определяемом спецификой очереди (обычно это порядок вставки элементов в очередь).

Карта: набор пар объектов «ключ-значение», с возможностью выборки по ключу. ***ArrayList*** позволяет искать объекты по порядковым номерам, поэтому в каком-то смысле он связывает числа с объектами. Класс ***Map*** (карта — также встречаются термины ассоциативный массив и словарь) позволяет искать объекты по другим объектам — например, получить объект значения по объекту ключа, по аналогии с поиском определения по слову.

Коллекции


<<Interface>>
Collection
<<Interface>>
Set
extends
<<Interface>>
List
<<Interface>>
Queue
HashSet
<<Interface>>
SortedSet
implements
<<Interface>>
Deque
Vector
ArrayList
TreeSet
LinkedHashSet
LinkedList
Stack

interface Collection

Method Summary

boolean	**add**(E e)// Добавление элемента в коллекцию. true если добавление прошло успешно
boolean	**addAll**(Collection<? extends E> c)//добавление коллекции в коллекцию
void	**clear**()//очистка коллекции
boolean	**contains**(Object o)// true если коллекция содержит указанные объект
boolean	**containsAll**(Collection<?> c)// true если коллекция содержит всю указанную коллекцию целиком.
boolean	**equals**(Object o)// сравнение коллекций друг с другом.
int	**hashCode**()//возвращает хеш-код
boolean	**isEmpty**()//проверка на пустоту
Iterator<E>	**iterator**()//возвращает итератор для навигации по коллекции

interface Collection

boolean	**remove**(Object o)//уделение объекта из коллекции. True если объект был найден и удален
boolean	**removeAll**(Collection<?> c)//удаление всех объектов коллекции C из текущей коллекции. True если коллекция была изменена. Т.е. если хоть дин элемент был удален.
boolean	**retainAll**(Collection<?> c)//Удаляет из текущей коллекции те элементы, которые не входят в коллекцию C
int	**size**()//возвращает размер колекции
Object[]	**toArray**()//возвращает массив состоящий из элементов коллекции
<T> T[]	**toArray**(T[] a) массив из элементов коллекции но с типом, как у массива a

interface List

Method Summary

void	**add**(int index, E element) Добавление элемента в коллекцию на определенную позицию. Существующий на этой позиции элемент и все следующие сдвигаются на одну позицию. index должен быть меньше либо равен size
boolean	**addAll**(int index, Collection<? extends E> c) //Тоже самое, но добавляет целую коллекцию.
E	**get**(int index) //Возвращает элемент на указанной позиции
int	**indexOf**(Object o)// Возвращает индекс объекта в коллекции. -1 если нет
int	**lastIndexOf**(Object o) // Индекс последнего вхождения объекта в коллекцию. -1 если нет.
ListIterator<E>	**listIterator**() // Возвращает итератор по элементам

interface List

ListIterator<E>	**listIterator**(int index) //Возвращает итератор по элементам начиная с позиции index
E	**remove**(int index) //Удаляет элемент с указанной позиции, при этом возвращая его
E	**set**(int index, E element) //Заменяет элемент стоящий на позиции index новым элементом, при этом возвращает старый элемент
List<E>	**subList**(int fromIndex, int toIndex) возвращает List состоящий из элементов исходного списка стоящих на позиция от и до.

ArrayList

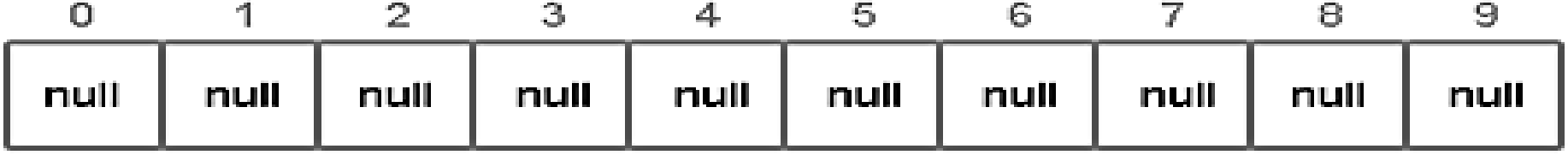
ArrayList — реализует интерфейс List. Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

Только что созданный объект list, содержит свойства **elementData** и **size**.

Хранилище значений **elementData** есть ни что иное как массив определенного типа (указанного в generic).

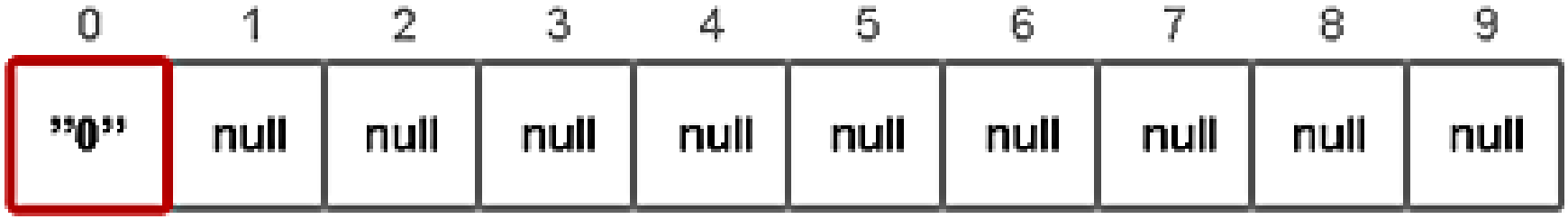
Если вызывается конструктор без параметров, то по умолчанию будет создан массив из 10-ти элементов типа Object (с приведением к типу, разумеется).

Вы можете использовать конструктор **ArrayList(capacity)** и указать свою начальную емкость списка.



ArrayList. Добавление элемента

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



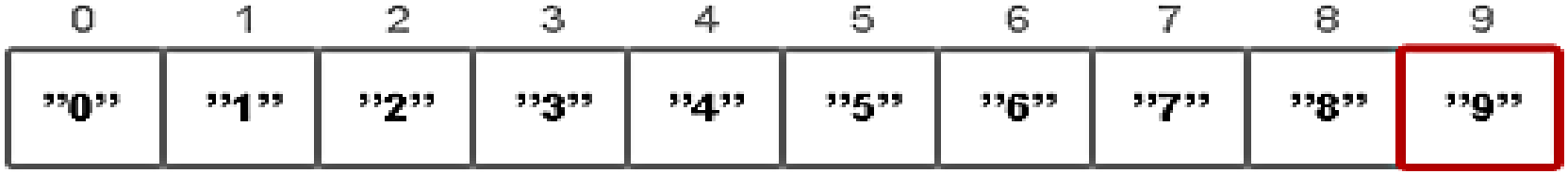
Внутри метода **add(value)** происходят следующие вещи:
1) проверяется, достаточно ли места в массиве для вставки нового элемента; **ensureCapacity(size + 1);**

2) добавляется элемент в конец (согласно значению **size**) массива.
elementData[size++] = element;

Если места в массиве не достаточно, новая емкость рассчитывается по формуле **(oldCapacity * 3) / 2 + 1**. Второй момент это копирование элементов. Оно осуществляется с помощью **native** метода **System.arraycopy()**, который написан не на Java.

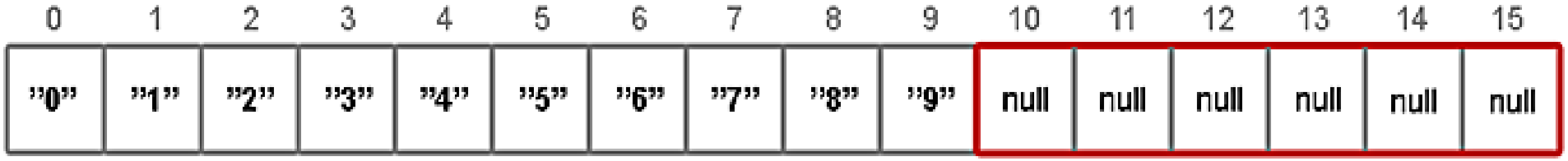
```
// newCapacity - новое значение емкости
elementData = (E[])new Object[newCapacity];
// oldData - временное хранилище текущего массива с данными
System.arraycopy(oldData, 0, elementData, 0, size);
```

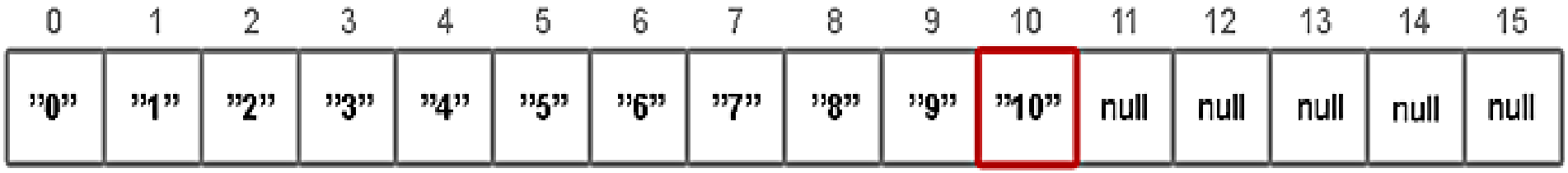
ArrayList. Добавление элемента



list.add("10");

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





Добавление элемента в середину

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

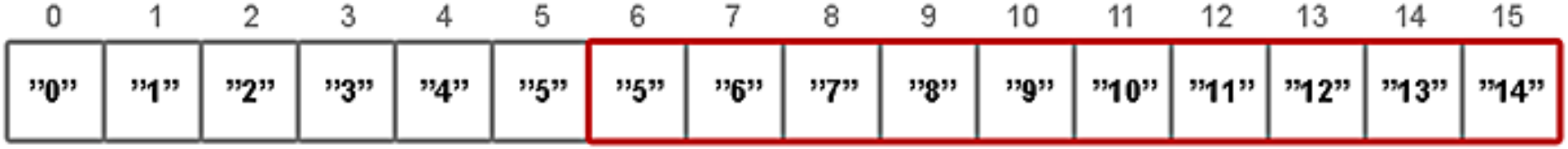
Добавление элемента на позицию с определенным индексом происходит в три этапа:

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

```
ensureCapacity(size+1);
```

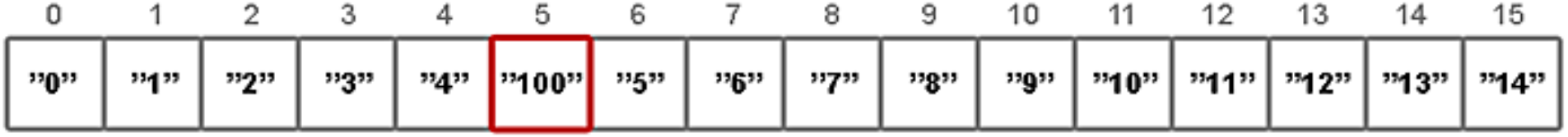
2) подготавливается место для нового элемента с помощью **System.arraycopy()**;

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



3) перезаписывается значение у элемента с указанным индексом.

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



Добавление элемента в середину

Как можно догадаться, в случаях, когда происходит вставка элемента по индексу и при этом в вашем массиве нет свободных мест, то вызов **System.arraycopy()** случится дважды: первый в **ensureCapacity()**, второй в самом методе **add(index, value)**, что явно скажется на скорости всей операции добавления.

В случаях, когда в исходный список необходимо добавить другую коллекцию, да еще и в «середину», стоит использовать метод **addAll(index, Collection)**. И хотя, данный метод скорее всего вызовет **System.arraycopy()** три раза, в итоге это будет гораздо быстрее поэлементного добавления.

Удаление элемента

Удалять элементы можно двумя способами:

— по индексу **remove(index)**

— по значению **remove(value)**

При удалении по индексу: `list.remove(5);`

1. Определяется количество элементов которое надо скопировать

```
int numMoved = size - index - 1;
```

2. Затем копируем элементы используя **System.arraycopy()**

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

3. Уменьшаем размер массива и забываем про последний элемент

```
elementData[--size] = null;
```

При удалении по значению, в цикле просматриваются все элементы списка, до тех пор пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.

При удалении элементов текущая величина capacity не уменьшается, что может привести к своеобразным утечкам памяти. Поэтому не стоит пренебрегать методом **trimToSize()**.

Резюме

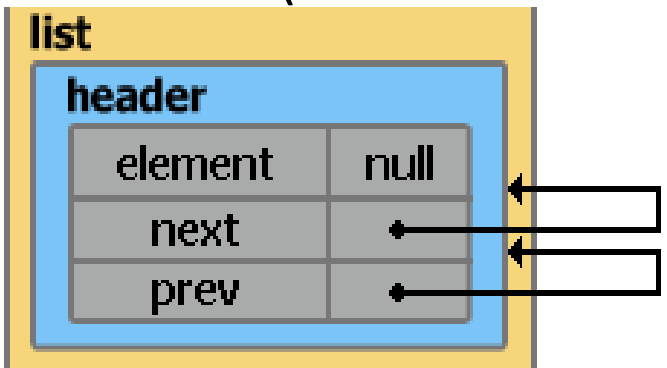
— Быстрый доступ к элементам по индексу за время O(1);
— Доступ к элементам по значению за линейное время O(n);
— Медленный, когда вставляются и удаляются элементы из «середины» списка;
— Позволяет хранить любые значения в том числе и null;

LinkedList

LinkedList — реализует интерфейс List. Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Итератор поддерживает обход в обе стороны. Реализует методы получения, удаления и вставки в начало, середину и конец списка. Позволяет добавлять любые элементы в том числе и null.

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

Только что созданный объект list, содержит свойства **header** и **size**. **header** — псевдо-элемент списка. Его значение всегда равно **null**, а свойства **next** и **prev** всегда указывают на первый и последний элемент списка соответственно. Так как на данный момент список еще пуст, свойства **next** и **prev** указывают сами на себя (т.е. на элемент **header**). Размер списка **size** равен 0



LinkedList. Добавление элемента

Добавление элемента в конец списка с помощью методом **add(value)**, **addLast(value)** и добавление в начало списка с помощью **addFirst(value)** выполняется за время O(1).

Внутри класса **LinkedList** существует static inner класс **Entry**, с помощью которого создаются новые элементы.

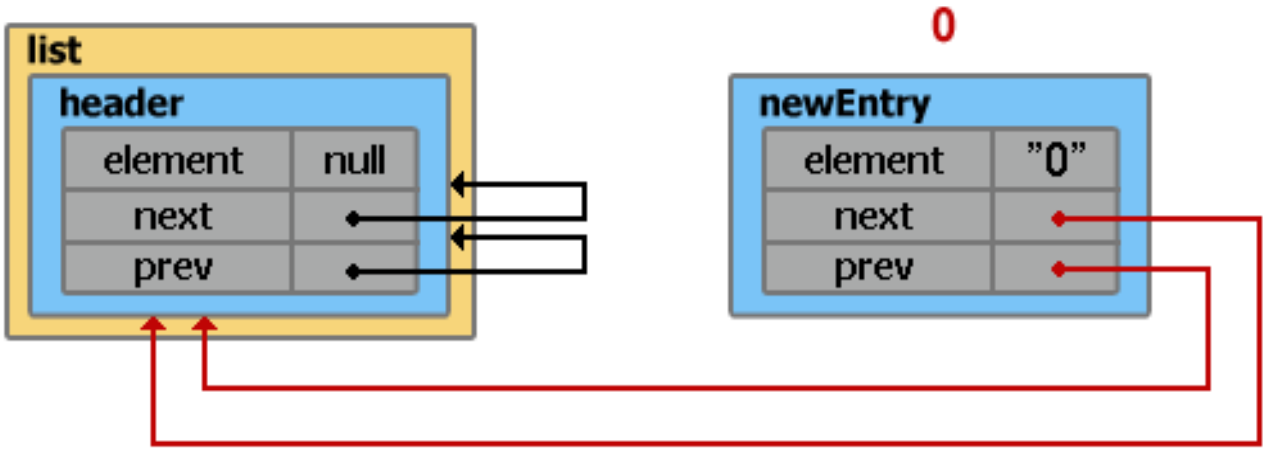
```
private static class Entry<E>{
    E element;
    Entry<E> next;
    Entry<E> prev;
    Entry(E element, Entry<E> next, Entry<E> prev){
            this.element = element;
            this.next = next;
            this.prev = prev;
        }
}
```

LinkedList. Добавление элемента

Каждый раз при добавлении нового элемента, по сути выполняется два шага:

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

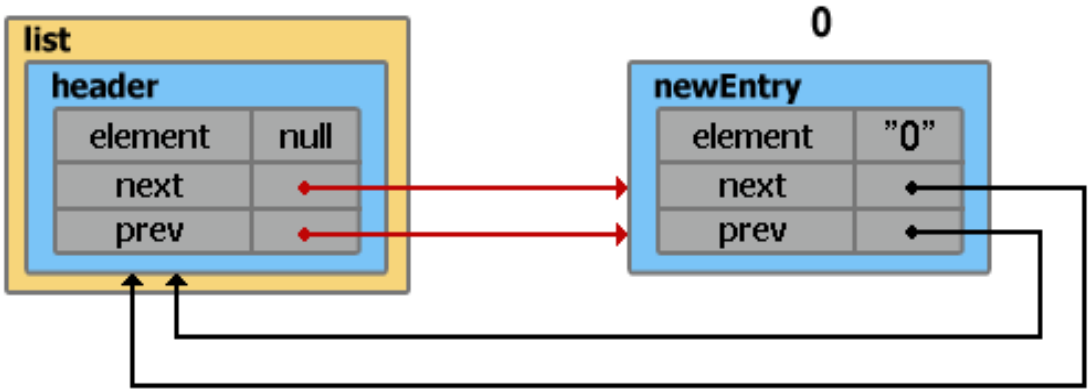
```
Entry newEntry = new Entry("0", header, header.prev);
```



2) переопределяются указатели на предыдущий и следующий элемент

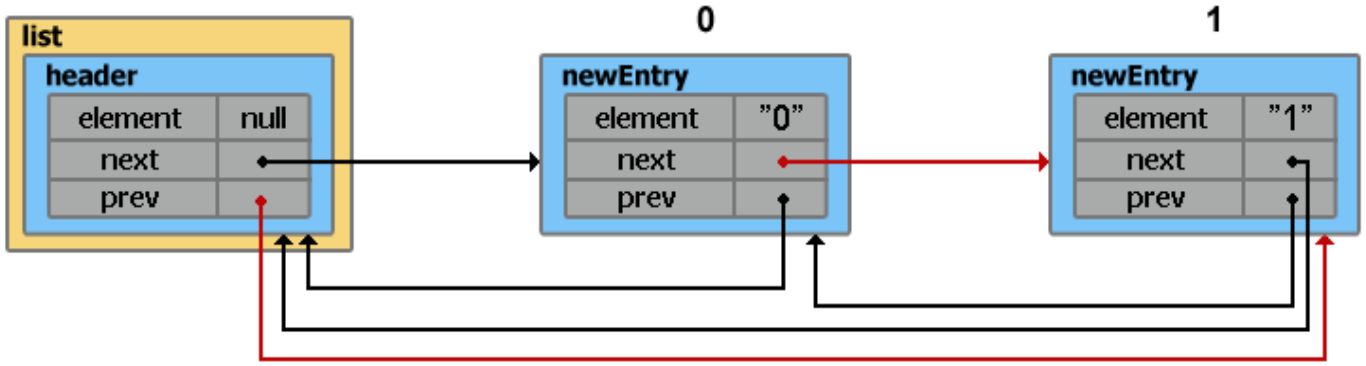
```
newEntry.prev.next = newEntry;
newEntry.next.prev = newEntry;
size++;
```

LinkedList. Добавление элемента



Добавим еще один элемент list.add("1");

```
// header.prev указывает на элемент с индексом 0
newEntry = new Entry("1", header, header.prev);
newEntry.prev.next = newEntry;
newEntry.next.prev = newEntry;
size++;
```



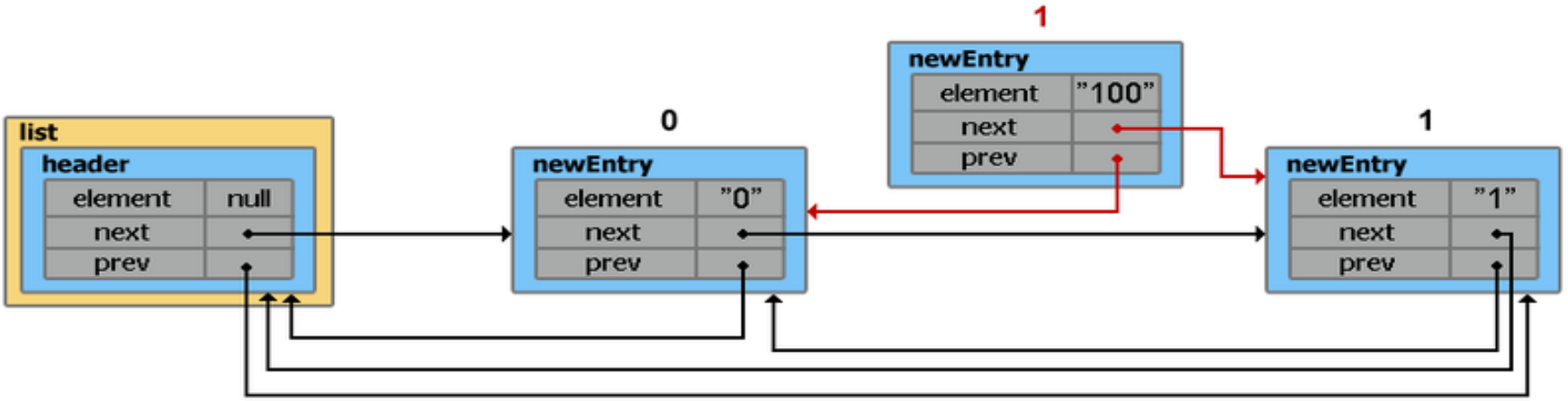
LinkedList. Добавление элемента в середину

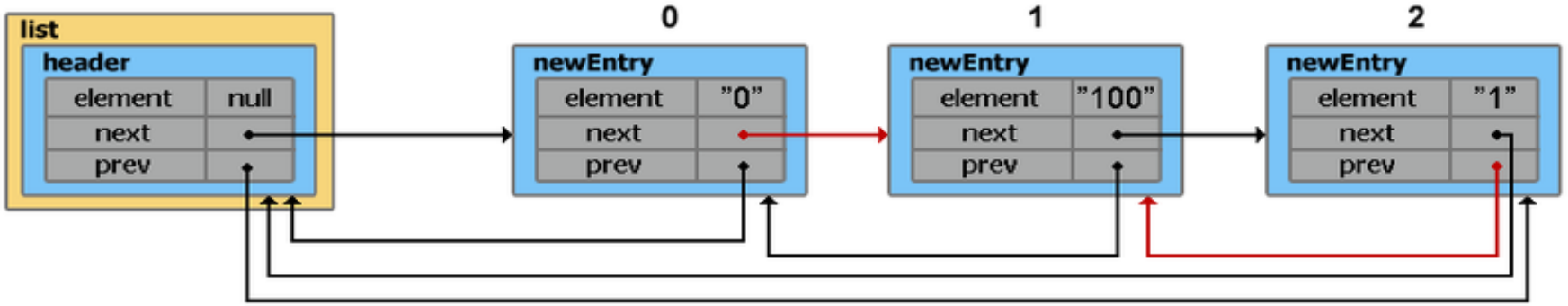
Для того чтобы добавить элемент на определенную позицию в списке, необходимо вызвать метод **add(index, value)**. Отличие от **add(value)** состоит в определении элемента перед которым будет производиться вставка

```
(index == size ? header : entry(index))
private Entry<E> entry(int index){
 if (index < 0 || index >= size)
     throw new
IndexOutOfBoundsException("Index:"+index+", Size: "+size);
 Entry<E> e = header;
 if (index < (size >> 1)){
        for (int i = 0; i <= index; i++) e = e.next;}
    else{
        for (int i = size; i > index; i--)e = e.prev;}
 return e;
}
```

LinkedList. Добавление элемента в середину

```
Entry newEntry = new Entry("100", entry, entry.prev);
```





LinkedList. Удаление элемента

Удалять элементы из списка можно несколькими способами:

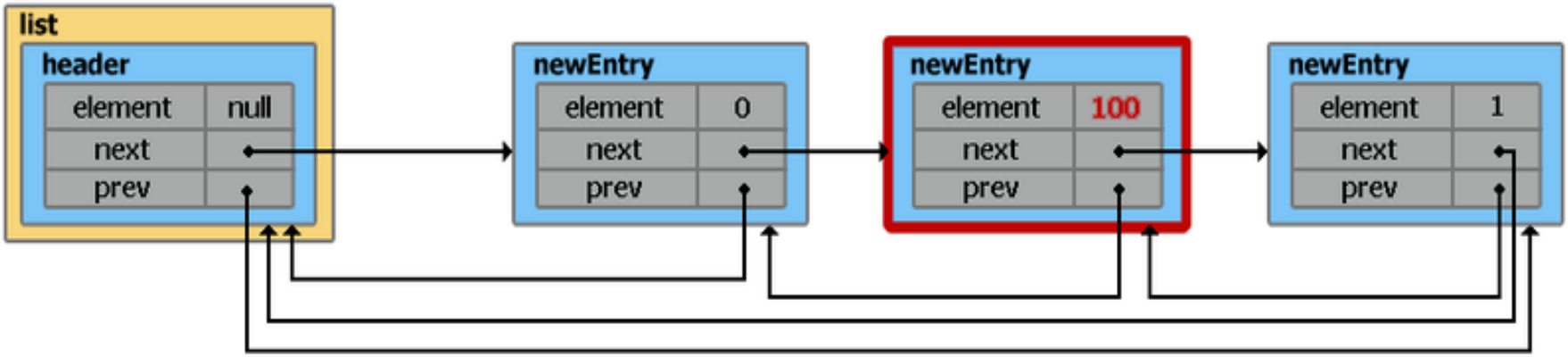
— из начала или конца списка с помощью removeFirst(), removeLast() за время O(1);

— по индексу remove(index) и по значению remove(value) за время O(n).

Внутри метода remove(value) просматриваются все элементы списка в поисках нужного. Удален будет лишь первый найденный элемент.

удаление из списка можно условно разбить на 3 шага:

1) поиск первого элемента с соответствующим значением



LinkedList. Удаление элемента

2) переопределяются указатели на предыдущий и следующий элемент

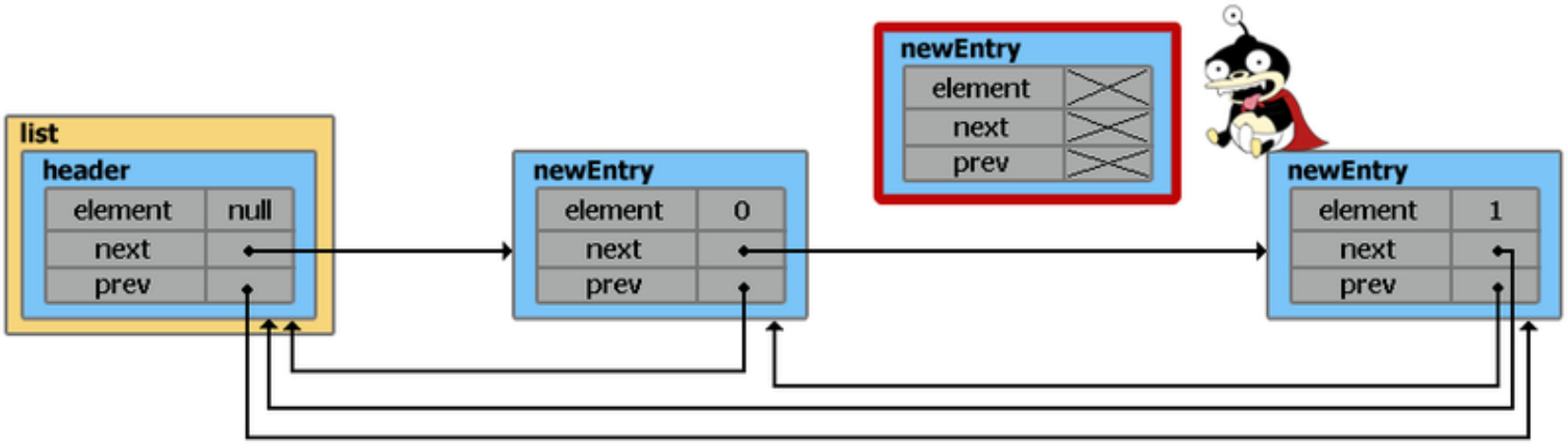
E result = e.element;

e.prev.next = e.next;

e.next.prev = e.prev;

3) удаление указателей на другие элементы и предание забвению самого элемента

e.next = e.prev = null;e.element = null;size--;



LinkedList. Резюме

— Из LinkedList можно организовать стэк, очередь, или двойную очередь, со временем доступа O(1);

— На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);

— Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-оберки;

Поскольку **LinkedList** реализует также интерфейс Queue, то он имеет дополнительные методы

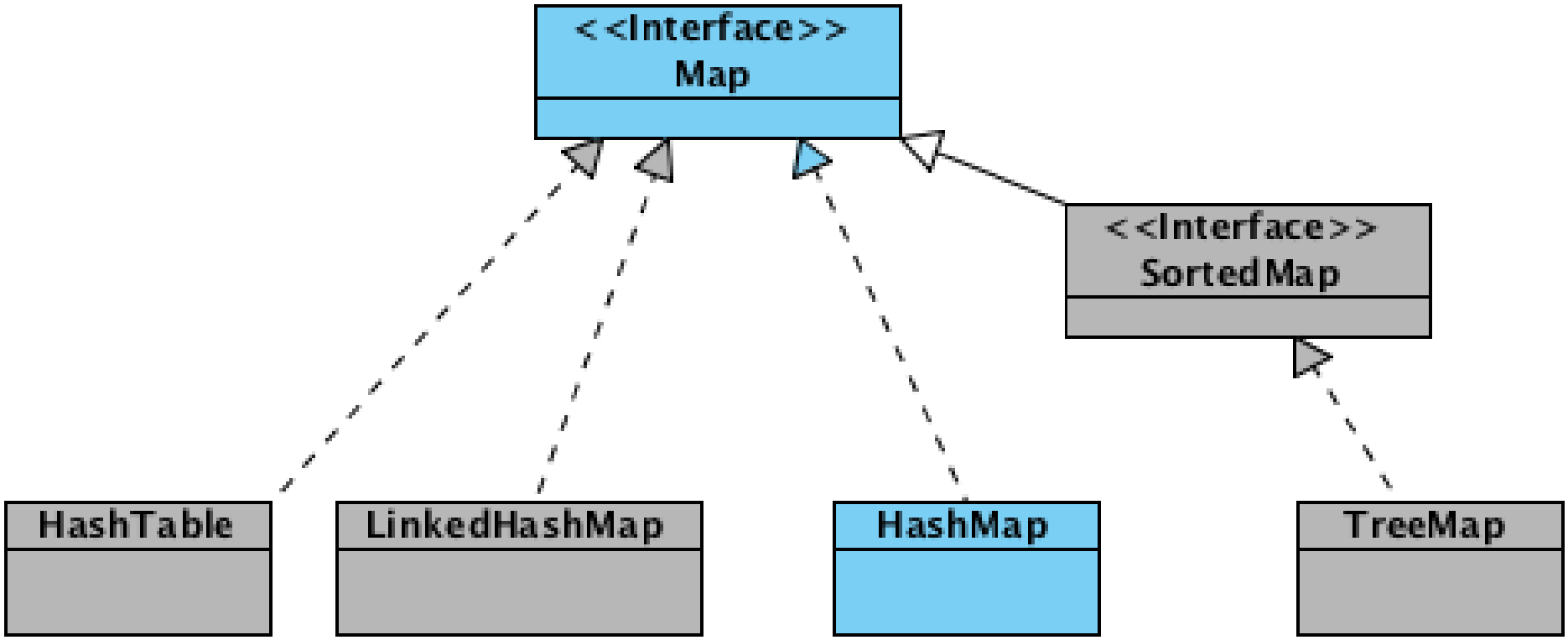
- peek() – получение первого элемента коллекции без его удаления.
- poll() – получение и удаление первого элемента в коллекции
- offer() – добавить элемент в конец коллекции

— Не синхронизирован.

LinkedList и стэк

```
import java.util.LinkedList;
public class Stack<T>  {
  private LinkedList<T> storage = new LinkedList<T>();
  public void push(T v) { storage.addFirst(v); }
  public T peek() { return storage.getFirst(); }
  public T pop() { return storage.removeFirst(); }
  public boolean empty() { return storage.isEmpty(); }
  public String toString() { return storage.toString(); }
}
```

Карты


<<Interface>>
Map
<<Interface>>
SortedMap
HashTable
LinkedHashMap
HashMap
TreeMap

Interface Map

```
public interface Map<K,V> {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

HashMap

```
Map<String, String> hashmap=new HashMap<String,String>();
```

Новоявленный объект hashmap, содержит ряд свойств:

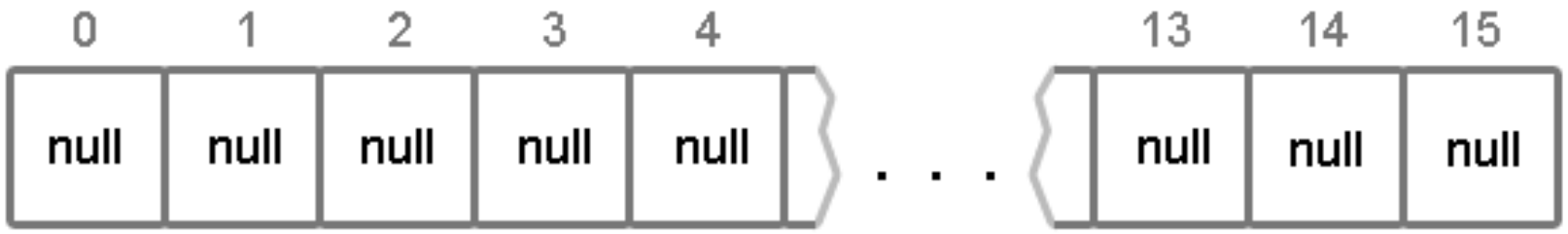
table — Массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;

loadFactor — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;

threshold — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле **(capacity * loadFactor)**;

size — Количество элементов HashMap-а;

В конструкторе, выполняется проверка валидности переданных параметров и установка значений в соответствующие свойства класса.



HashMap. Исходники

```
public class HashMap<K,V> extends AbstractMap<K,V>
       implements Map<K,V>, Cloneable, Serializable{
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final int MAXIMUM_CAPACITY = 1 << 30;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
transient volatile int modCount;
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("…..");
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("….");
    int capacity = 1;
    while (capacity < initialCapacity) capacity <<= 1;
    this.loadFactor = loadFactor;
    threshold = (int)(capacity * loadFactor);
    table = new Entry[capacity];
    init();}
```

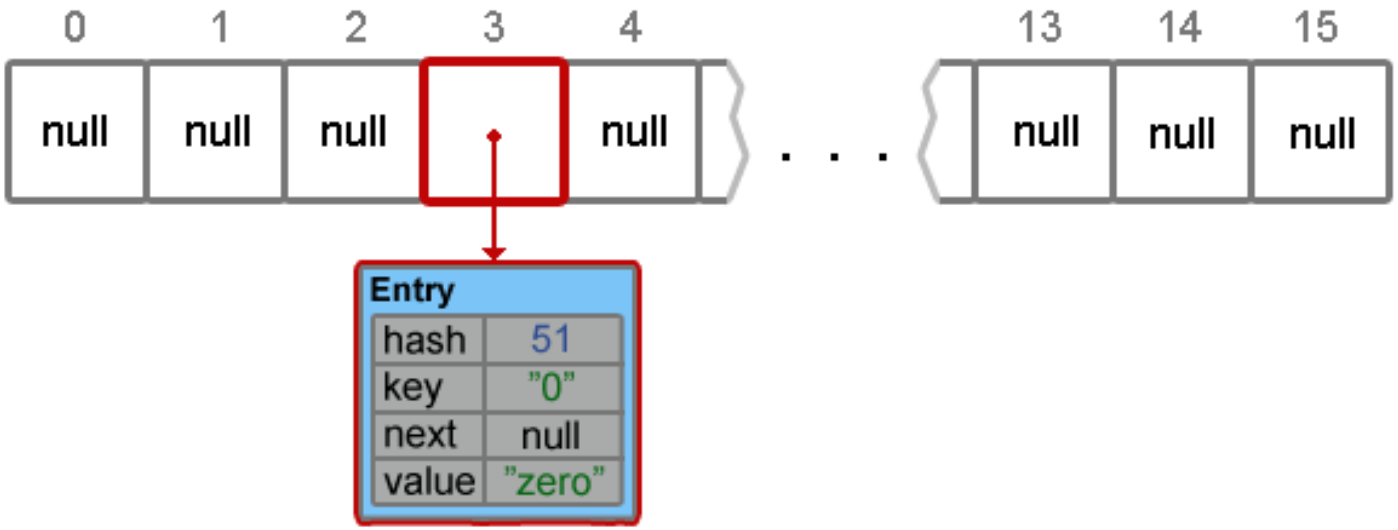
HashMap. Исходники

```
public V put(K key, V value) {
  if (key == null)return putForNullKey(value);
  int hash = hash(key.hashCode());
  int i = indexFor(hash, table.length);
  for (Entry<K,V> e = table[i]; e != null; e = e.next) {
    Object k;
    if (e.hash == hash && ((k = e.key) == key ||
                           key.equals(k))) {
        V oldValue = e.value;
        e.value = value;
        e.recordAccess(this);
        return oldValue;}}
    modCount++;
    addEntry(hash, key, value, i);
    return null;}
void addEntry(int hash, K key, V value, int bucketIndex) {
 Entry<K,V> e = table[bucketIndex];
 table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
 if (size++ >= threshold) resize(2 * table.length);}
```

Entry. Исходники

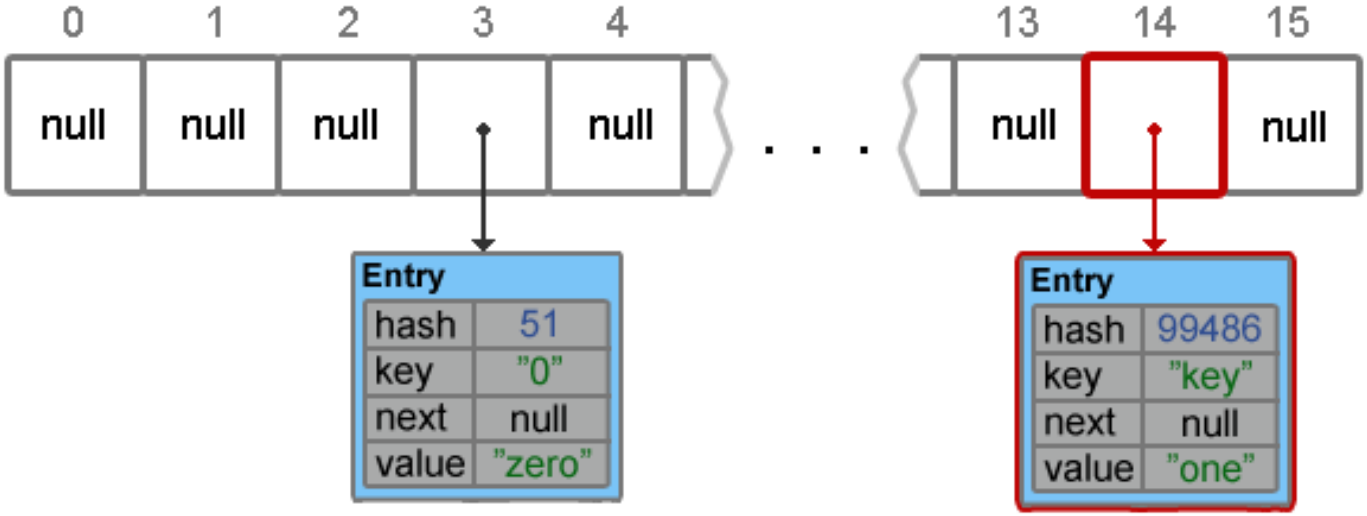
```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;next = n;key = k;hash = h;}
……
void recordAccess(HashMap<K,V> m) {}
void recordRemoval(HashMap<K,V> m) {}}
```

```
hashmap.put("0", "zero");
```

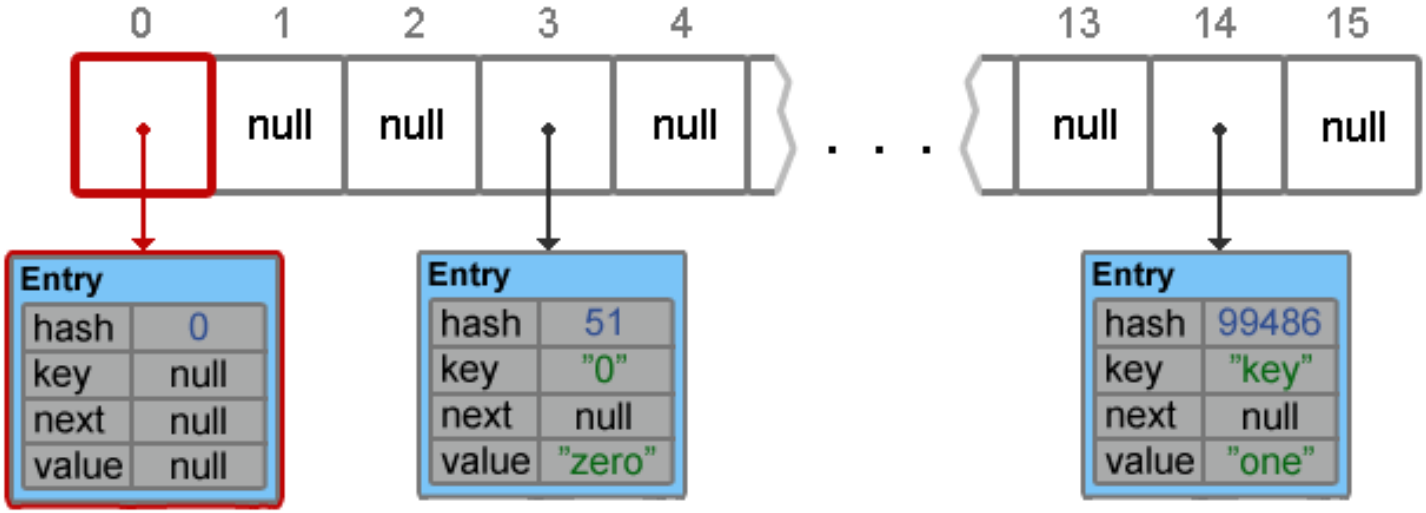


HashMap. Добавление

```
hashmap.put("key", "one");
```

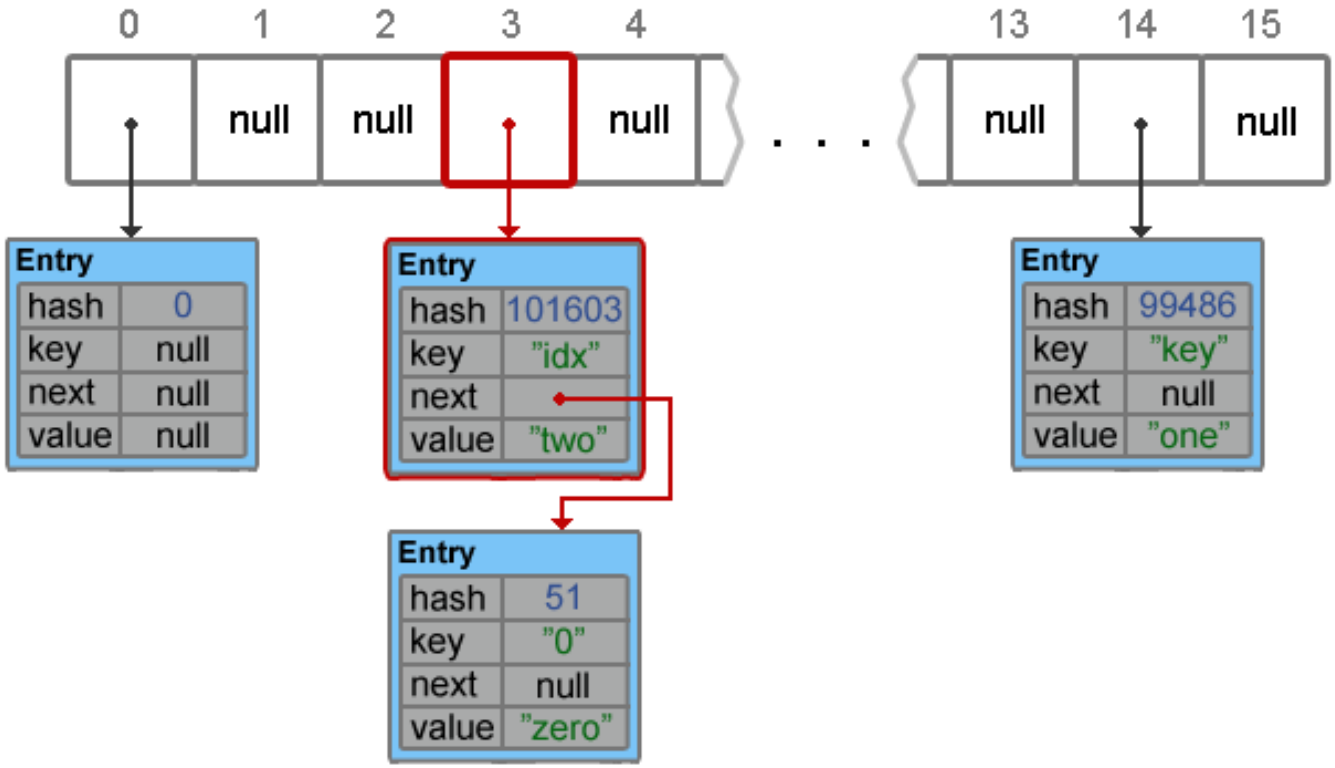



```
hashmap.put(null, null);
```



HashMap. Добавление

```
hashmap.put("idx", "two");
```



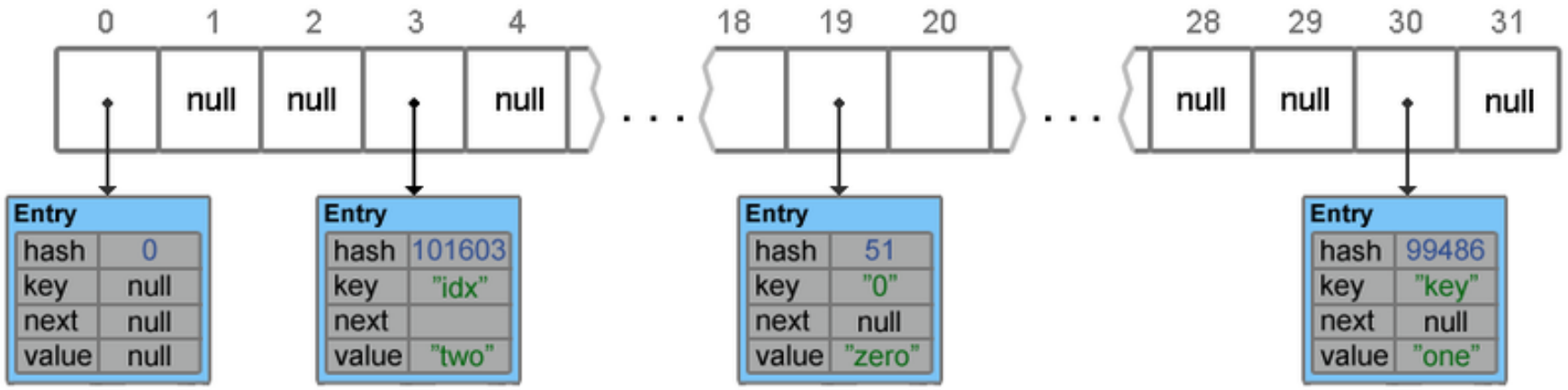
HashMap. Поиск

```
public V get(Object key) {
  if (key == null)
        return getForNullKey();
  int hash = hash(key.hashCode());
  for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e != null;
        e = e.next) {
       Object k;
       if (e.hash == hash && ((k = e.key) == key ||
                              key.equals(k)))
           return e.value;
    }
  return null;
}
private V getForNullKey() {
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
       if (e.key == null)
           return e.value;
    }
    return null;}
```

HashMap. Resize и Transfer

Когда массив **table[]** заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов.

```
void resize(int newCapacity){
 if (table.length == MAXIMUM_CAPACITY){
        threshold = Integer.MAX_VALUE;
        return;}
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);}
```



HashMap. Как теряется элемент

```
class DataOnly{
int i;int p;
DataOnly(int i,int p){this.i=i;this.p=p;}
 @Override
 public boolean equals(Object S){
  if (S instanceof DataOnly) {
    DataOnly temp = (DataOnly) S;
      return ((this.i==temp.i)&&(this.p==temp.p));}
  else return false;}
 @Override
 public int hashCode(){return i%p;}}
public class TestSet {
  public static void main(String[] args){
HashMap<DataOnly,String>hm=new HashMap<DataOnly,String>();
  DataOnly d1 = new DataOnly(1, 1);
  DataOnly d2 = new DataOnly(2, 2);
  DataOnly d1_1 = new DataOnly(1, 1);
  hm.put(d1, "one"); hm.put(d2, "two");
   System.out.println(hm.get(d1_1)); d1.i=2;
  System.out.println(hm.get(d1_1));}}
```

HashMap. Удаление элементов. Итоги

У HashMap есть такая же «проблема» как и у ArrayList — при удалении элементов размер массива **table[]** не уменьшается. И если в ArrayList предусмотрен метод **trimToSize()**, то в HashMap таких методов нет

```
hashmap = new HashMap<String, String>(hashmap);
```

Итоги

— Добавление элемента выполняется за время O(1), потому как новые элементы вставляются в начало цепочки;

— Операции получения и удаления элемента могут выполняться за время O(1), если хэш-функция равномерно распределяет элементы и отсутствуют коллизии. Среднее же время работы будет Θ(1 + α), где α — коэффициент загрузки. В самом худшем случае, время выполнения может составить Θ(n) (все элементы в одной цепочке);

— Ключи и значения могут быть любых типов, в том числе и null. Для хранения примитивных типов используются соответствующие классы-оберки;

— Не синхронизирован.

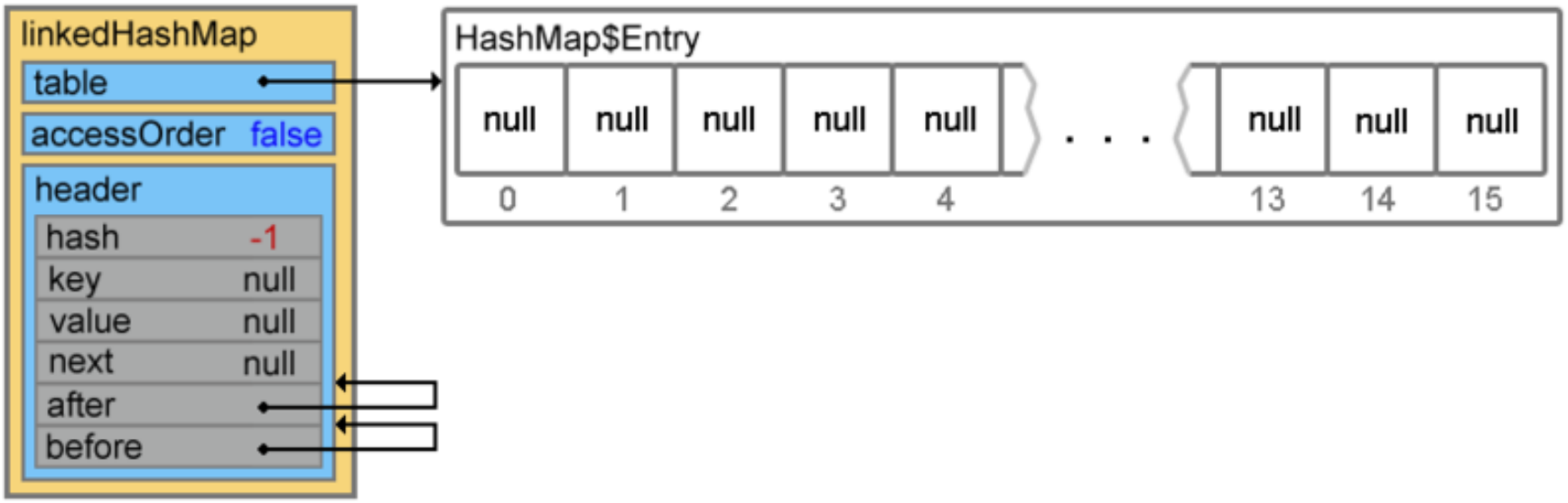
LinkedHashMap.

```
public class LinkedHashMap<K,V> extends HashMap<K,V>
         implements Map<K,V>{
private transient Entry<K,V> header;
private final boolean accessOrder;
private static class Entry<K,V> extends
HashMap.Entry<K,V>{
 Entry<K,V> before, after;
 Entry(int hash, K key, V value, HashMap.Entry<K,V> next){
        super(hash, key, value, next);}
 private void remove() {
       before.after = after;after.before = before;}
 private void addBefore(Entry<K,V> existingEntry) {
   after  = existingEntry;before = existingEntry.before;
   before.after = this;after.before = this;}
 void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    if (lm.accessOrder) {
      lm.modCount++;remove();addBefore(lm.header);}
  }
 void recordRemoval(HashMap<K,V> m) {remove();}}
```

LinkedHashMap.

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

void init() {
header = new Entry<K,V>(-1, null, null, null);
 header.before = header.after = header; }
```

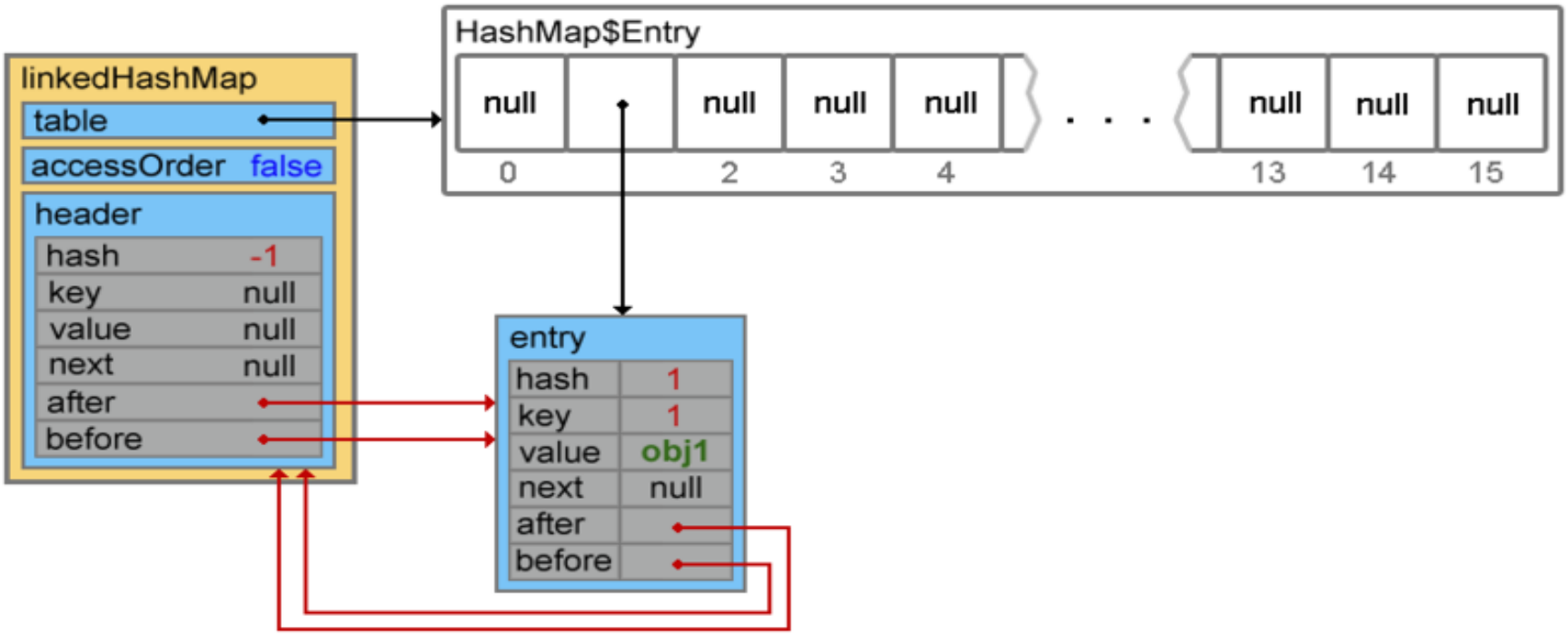


LinkedHashMap.

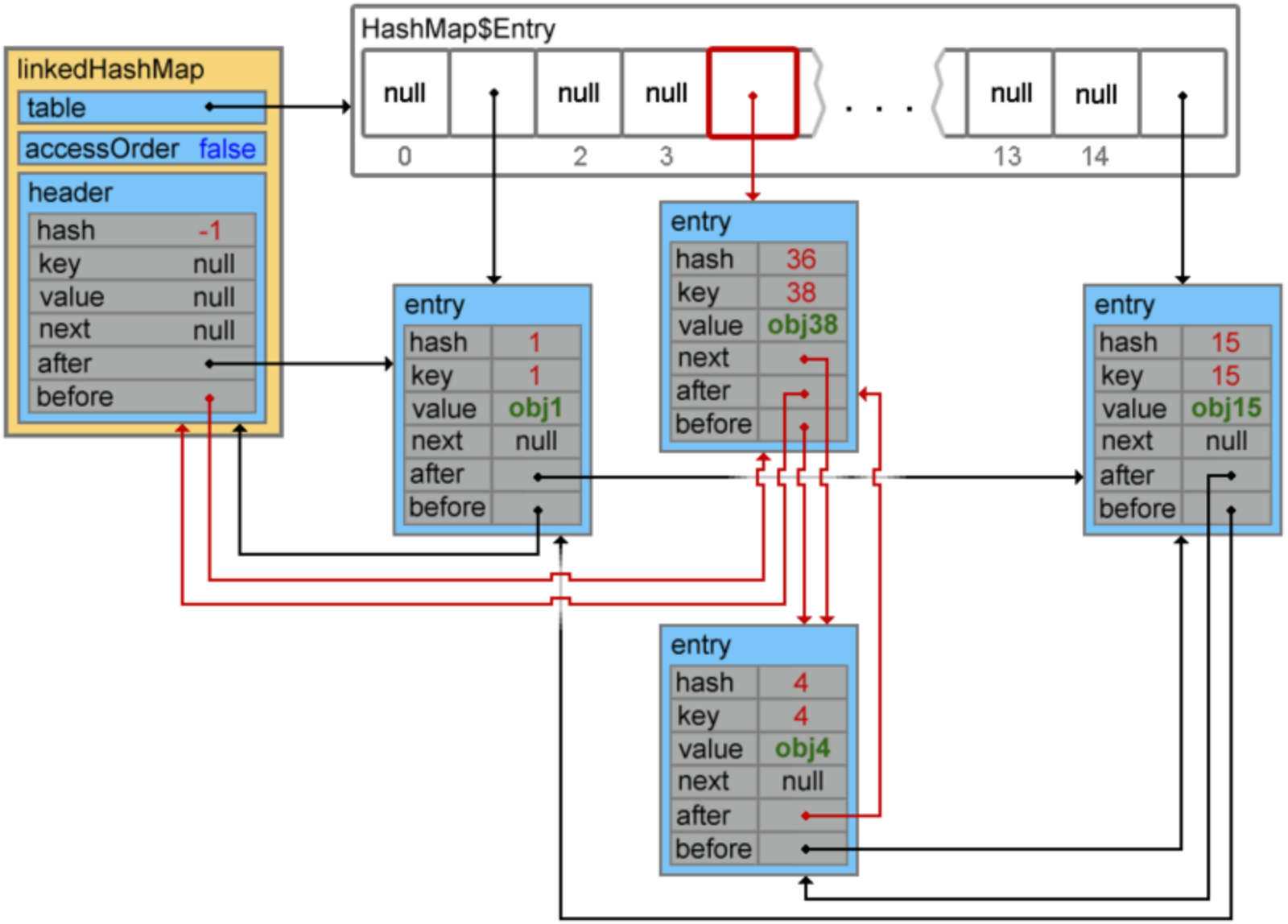
```
linkedHashMap.put(1, "obj1");


void addEntry(int hash, K key, V value, int bucketIndex) {
    createEntry(hash, key, value, bucketIndex);
    Entry<K,V> eldest = header.after;
    if (removeEldestEntry(eldest)) {
        removeEntryForKey(eldest.key);
    } else {
        if (size >= threshold)
            resize(2 * table.length);
    }
}
void createEntry(int hash,K key, V value, int bucketIndex)
{
    HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
    table[bucketIndex] = e;
    e.addBefore(header);
    size++;}
```

LinkedHashMap.



LinkedHashMap.



в случае повторной вставки элемента (элемент с таким ключом уже существует) порядок доступа к элементам не изменится.

LinkedHashMap.

В ситуации когда **accessOrder == true** поведение **LinkedHashMap** меняется и при вызовах методов **get()** и **put()** порядок элементов будет изменен — элемент к которому обращаемся будет помещен в конец.

```
Map<Integer, String> linkedHashMap =
    new LinkedHashMap<Integer, String>(15, 0.75f, true) {{
    put(1, "obj1");
    put(15, "obj15");
    put(4, "obj4");
    put(38, "obj38");
}};
// {1=obj1, 15=obj15, 4=obj4, 38=obj38}

linkedHashMap.get(1); //or linkedHashMap.put(1,"Object1");
// {15=obj15, 4=obj4, 38=obj38, 1=obj1}
```

Set

В множествах (***Set***) каждое значение может храниться только в одном экземпляре. Попытки добавить новый экземпляр эквивалентного объекта блокируются. Множества часто используются для проверки принадлежности, чтобы вы могли легко проверить, принадлежит ли объект заданному множеству. Следовательно, важнейшей операцией ***Set*** является операция поиска, поэтому на практике обычно выбирается реализация ***HashSet***, оптимизированная для быстрого поиска.

Set имеет такой же интерфейс, что и ***Collection***. В сущности, ***Set*** и является ***Collection***, но обладает несколько иным поведением.

Существует три реализации этого интерфейса

1. HashSet
2. TreeSet
3. LinkedHashSet

Для правильной работы множества должен быть как минимум реализован метод equals() у хранимых объектов.

Set. Как не надо делать

```
import java.util.HashSet;
import java.util.Set;
class DataOnly1{
  int i;
  int p;
}
public class TestSet {
  public static void main(String[] args){
    Set<DataOnly1> s = new HashSet<DataOnly1>();
    DataOnly1 d1 =new DataOnly1();
    d1.i=1; d1.p=1;
    DataOnly1 d2 = new DataOnly1();
    d2.i=1; d2.p=1;
    s.add(d1);
    s.add(d2);
    System.out.println(s.size());
}}
```

HashSet. Как надо делать

HashSet –неотсортированная и неупорядоченная коллекция, для вставки элемента используются методы hashCode() и equals(**…).Чем эффективней реализован метод hashCode(), тем эфеективней работает коллекция.**

```
class DataOnly1{
int i; int p;
@Override
public boolean equals(Object S){
   if (S instanceof DataOnly1) {
      DataOnly1 temp = (DataOnly1) S;
      return ((this.i==temp.i)&&(this.p==temp.p));}
 else return false;}
@Override
public int hashCode(){ return i%p;}}
public class TestSet {
public static void main(String[] args){
Set<DataOnly1> s = new HashSet<DataOnly1>();
DataOnly1 d1 =new DataOnly1();d1.i=1; d1.p=1;
DataOnly1 d2 = new DataOnly1();d2.i=1; d2.p=1;
s.add(d1);s.add(d2);}}
```

HashSet. Интересная реализация

```
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>,Cloneable, java.io.Serializable{
static final long serialVersionUID= -5024744406713321676L;
  private transient HashMap<E, Object> map;
 // Dummy value to associate with an Object
  private static final Object PRESENT = new Object();
public HashSet() {
        map = new HashMap<E,Object>();    }
public Iterator<E> iterator() {
        return map.keySet().iterator();  }
public int size() {
        return map.size(); }
public boolean add(E e) {
        return map.put(e, PRESENT)==null;}
…….
```

TreeMap и TreeSet.

Позволяют хранить данные упорядоченно. По умолчанию в порядке возрастания. Для хранения используется красно-черное дерево. Для работы с этими контейнерами необходимо, чтобы был реализован интерфейс Comparable либо был передан экземпляр класса реализующего Comparator. В противном случае в контейнер нельзя будет добавить больше одного значения.

По этой же причине нельзя использовать в качестве ключа null.

```
import java.util.Map;
import java.util.TreeMap;
public class TestTree {
  private static class DataOnly{
    int i; int p;
    DataOnly(int i,int p){this.i=i; this.p=p;}}
  public static void main(String[] arg){
 Map<DataOnly,String> tree=new TreeMap<DataOnly,String>();
  tree.put(new DataOnly(1, 1), "one");
  tree.put(new DataOnly(2,2), "two");
}}
```

TreeMap и TreeSet.

```
public class TestTree {
private static class DataOnly implements Comparable{
  int i; int p;
  DataOnly(int i,int p){this.i=i; this.p=p;}
  @Override
  public int compareTo(Object arg0) {
  if (!(arg0 instanceof DataOnly)){
        throw new ClassCastException();}
  else{
    DataOnly c = (DataOnly)arg0;
    if (i*p>c.i*c.p) return 1;
    else
      if (i*p<c.i*c.p) return -1;
      else
        return 0;}}}
public static void main(String[] arg){
Map<DataOnly,String>tree =new TreeMap<DataOnly, String>();
tree.put(new DataOnly(1, 1), "one");
tree.put(new DataOnly(2,2), "two");
}}
```

TreeMap и TreeSet.

```
import java.util.Comparator;
import java.util.Map;
import java.util.TreeMap;
public class TestTree {
private static class DataOnly{
  int i; int p;
  DataOnly(int i,int p){this.i=i; this.p=p;}}
private static class DataOnlyComporator
                        implements Comparator<DataOnly>{
  @Override
  public int compare(DataOnly o1, DataOnly o2) {
    if (o1.i*o1.p>o1.i*o1.p) return 1;
    else if (o1.i*o1.p>o1.i*o1.p) return -1;
      else return 0;}}
public static void main(String[] arg){
  Map<DataOnly,String> tree =
  new TreeMap<DataOnly, String>(new DataOnlyComporator());
  tree.put(new DataOnly(1, 1), "one");
  tree.put(new DataOnly(2,2), "two");
}}
```

TreeMap и TreeSet.

```
import java.util.Comparator;
import java.util.Map;
import java.util.TreeMap;
public class TestTree {
  private static class DataOnly{
    int i; int p;
    DataOnly(int i,int p){this.i=i; this.p=p;}}
public static void main(String[] arg){
 Map<DataOnly,String> tree =
 new TreeMap<DataOnly, String>(new Comparator<DataOnly>(){
    @Override
    public int compare(DataOnly o1, DataOnly o2) {
      if (o1.i*o1.p>o1.i*o1.p) return 1;
       else if (o1.i*o1.p>o1.i*o1.p) return -1;
        else return 0;}});
 tree.put(new DataOnly(1, 1), "one");
 tree.put(new DataOnly(2,2), "two");
}}
```

Итераторы

Итератор – это объект, который позволяет программисту пробежать по элементам **коллекции**. С его помощью можно сделать следующее:

- Запросить у контейнера итератор вызовом метода ***iterator()***. Полученный итератор готов вернуть начальный элемент последовательности при первом вызове своего метода ***next()***.
- Получить следующий элемент последовательности вызовом метода ***next()***.
- Проверить, остались ли еще объекты в последовательности (метод ***hasNext()***).
- Удалить из последовательности последний элемент, возвращенный итератором, методом ***remove()***.

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

Итераторы

```
Set<String> set = new HashSet<String>();
//Set<String> set = new LinkedHashSet<String>();
set.add("One");
set.add("Two");
set.add("Three");
Iterator<String> iterator = set.iterator();
while (iterator.hasNext()) {
System.out.println(iterator.next());}
// Three One Two
//One Two Three
Map<String,String> pets = new HashMap<String,String>();
pets.put("A", "Cat");
pets.put("B", "dog");
pets.put("C", "parrot");
Set<Entry<String, String>> set = pets.entrySet();
Iterator<Entry<String, String>> iter = set.iterator();
while (iter.hasNext()) {
 Entry<String, String> pet = iter.next();
 System.out.println(pet.getKey()+" is a "+pet.getValue());
}
```

ListIterator

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e);
}
```

Литература

1. Брюс Эккель Философия Java. 4-е издание
2. Хорстманн К. С., Корнелл Г. -- Java 2. Том 1. Основы
3. Habrahabr.ru
4. Sql.ru
5. http://grepcode.com/project/repository.grepcode.com/java/root/jdk/openjdk/
6. !!! Google.com
7. http://habrahabr.ru/post/127864/