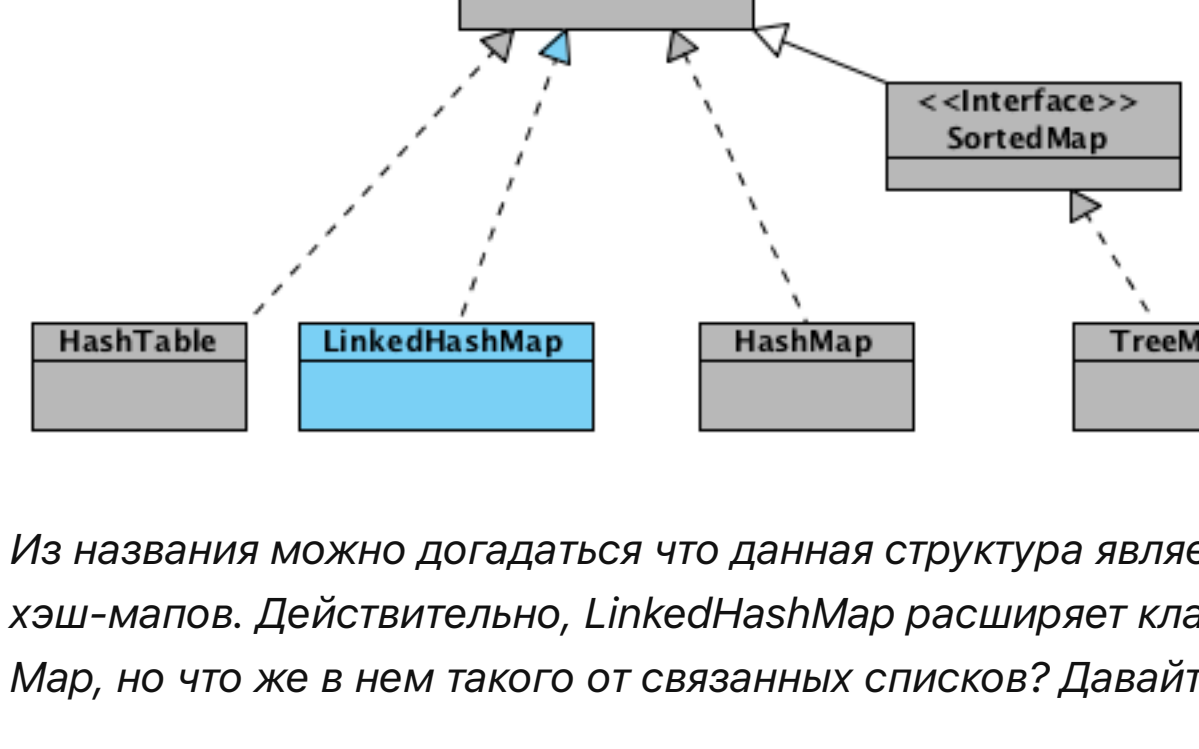


Структуры данных в картинках. LinkedHashMapMap

Java?

Привет Хабрачеловеки!

После затяжной паузы, я попробую продолжить визуализировать структуры данных в Java. В предыдущих статьях были замечены: [ArrayList](#), [LinkedList](#), [HashMap](#). Сегодня заглянем внутрь к [LinkedHashMap](#).



Из названия можно догадаться что данная структура является симбиозом связанных списков и хэш-мапов. Действительно, [LinkedHashMap](#) расширяет класс [HashMap](#) и реализует интерфейс [Map](#), но что же в нем такого от связанных списков? Давайте будем разбираться.

Создание объекта

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

Footprint{Objects=3, References=26, Primitives=[int x 4, float, boolean]}
size: 160 bytes

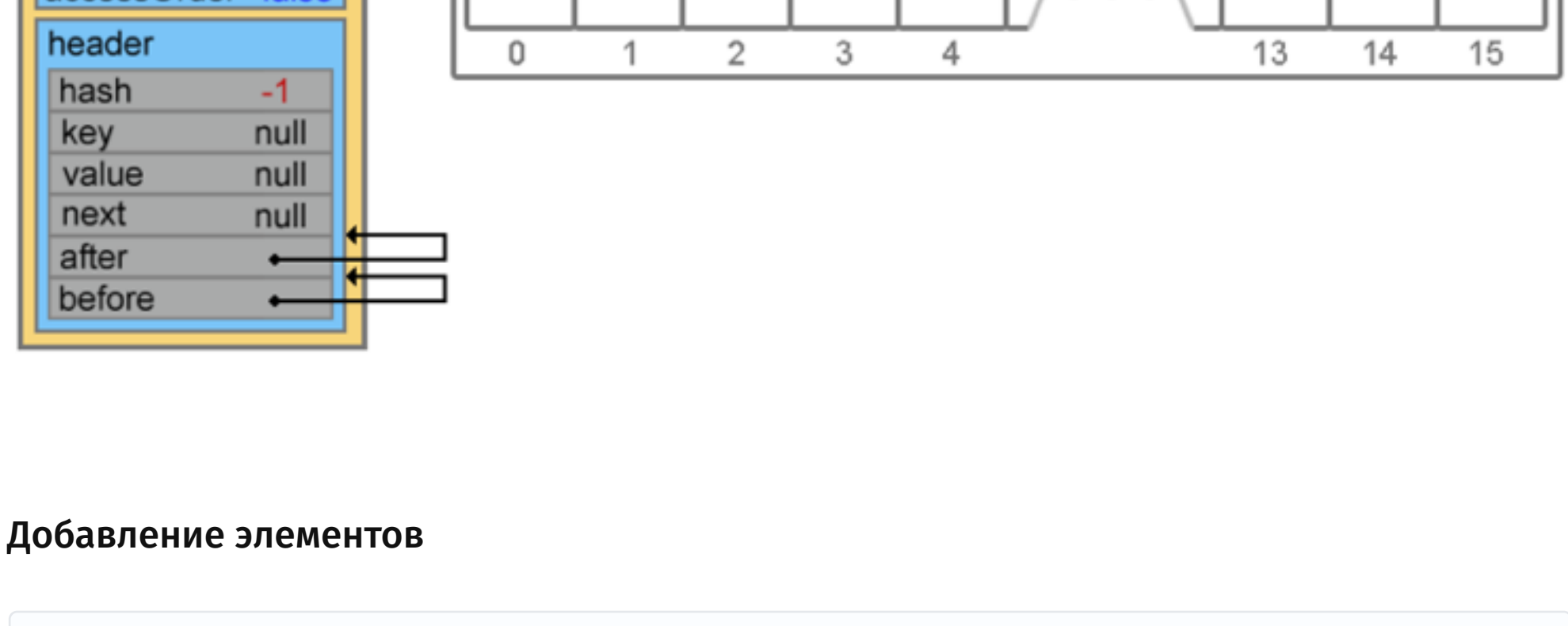
Только что созданный объект **linkedHashMap**, помимо свойств унаследованных от **HashMap** (такие как `table`, `loadFactor`, `threshold`, `size`, `entrySet` и т.п.), так же содержит два доп. свойства:

- header** — «голова» двусвязного списка. При инициализации указывает сам на себя;
- accessOrder** — указывает каким образом будет осуществляться доступ к элементам при использовании итератора. При значении **true** — по порядку последнего доступа (об этом в [конце](#) статьи). При значении **false** доступ осуществляется в том порядке, в каком элементы были вставлены.

Конструкторы класса **LinkedHashMap** достаточно скучные, вся их работа сводится к вызову конструктора родительского класса и установке значения свойству **accessOrder**. А вот инициализация свойства **header** происходит в переопределенном методе **init()** (теперь становится понятно для чего в конструкторах класса **HashMap** присутствует вызов этой, ничегонеделющей функции).

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

Новый объект создан, свойства проинициализированы, можно переходить к добавлению элементов.



Добавление элементов

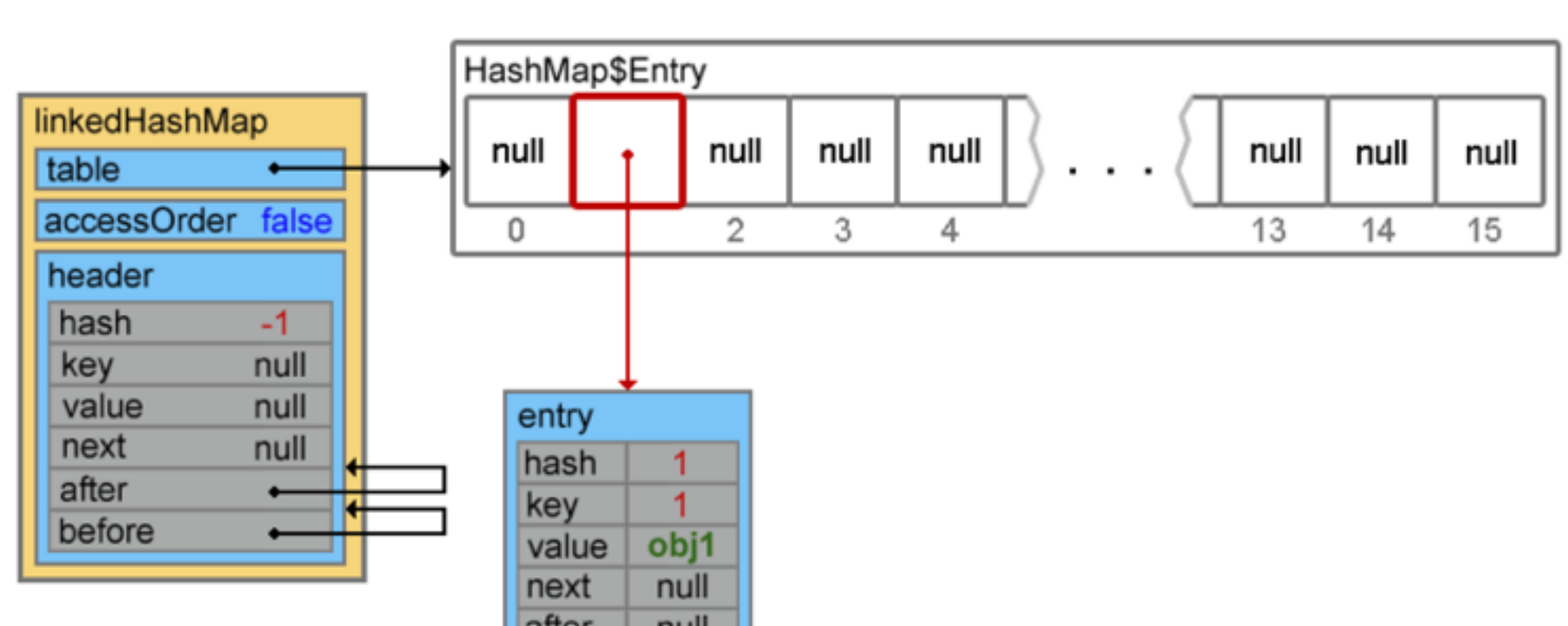
```
linkedHashMap.put(1, "obj1");
```

Footprint{Objects=7, References=32, Primitives=[char x 4, int x 9, float, boolean]}
size: 256 bytes

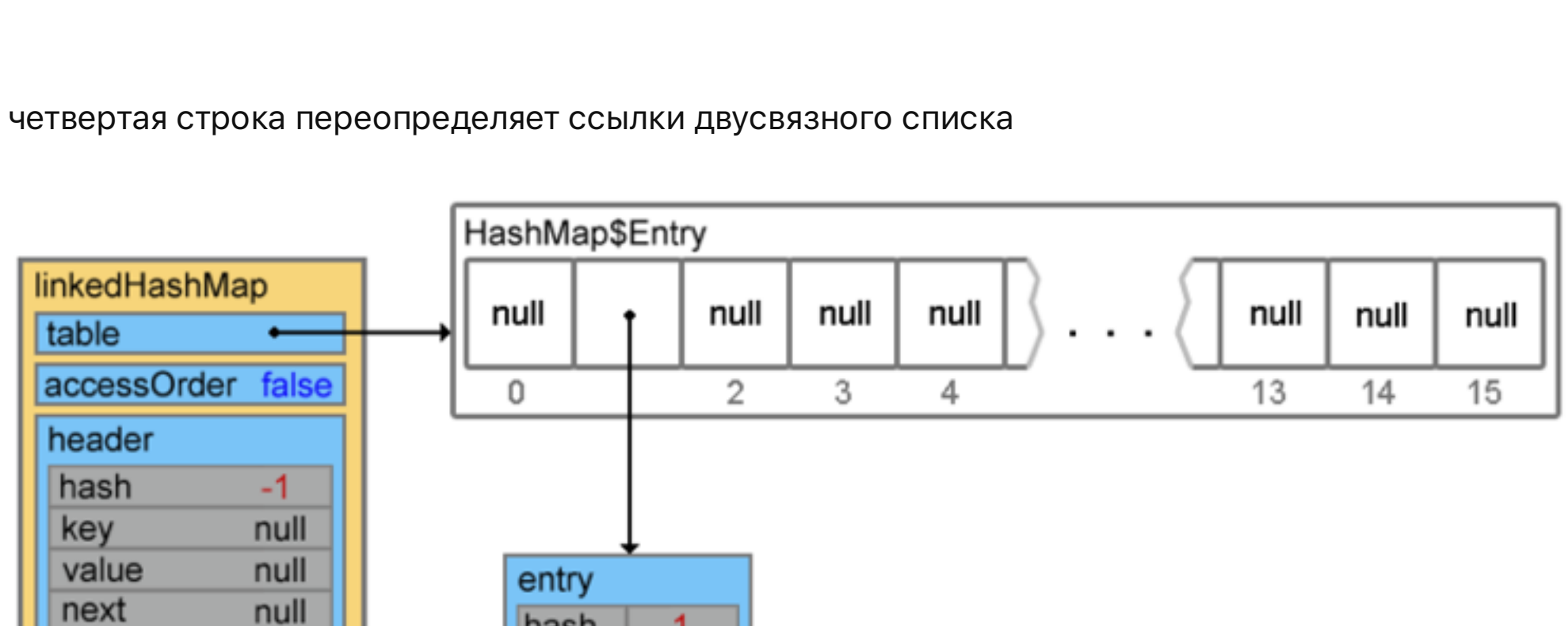
При добавлении элемента, первым вызывается метод **createEntry(hash, key, value, bucketIndex)** (по цепочке **put()** -> **addEntry()** -> **createEntry()**)

```
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++;
}
```

первые три строки добавляют элемент (при коллизиях добавление произойдет в начало цепочки, далее мы это увидим)



четвертая строка переопределяет ссылки двусвязного списка

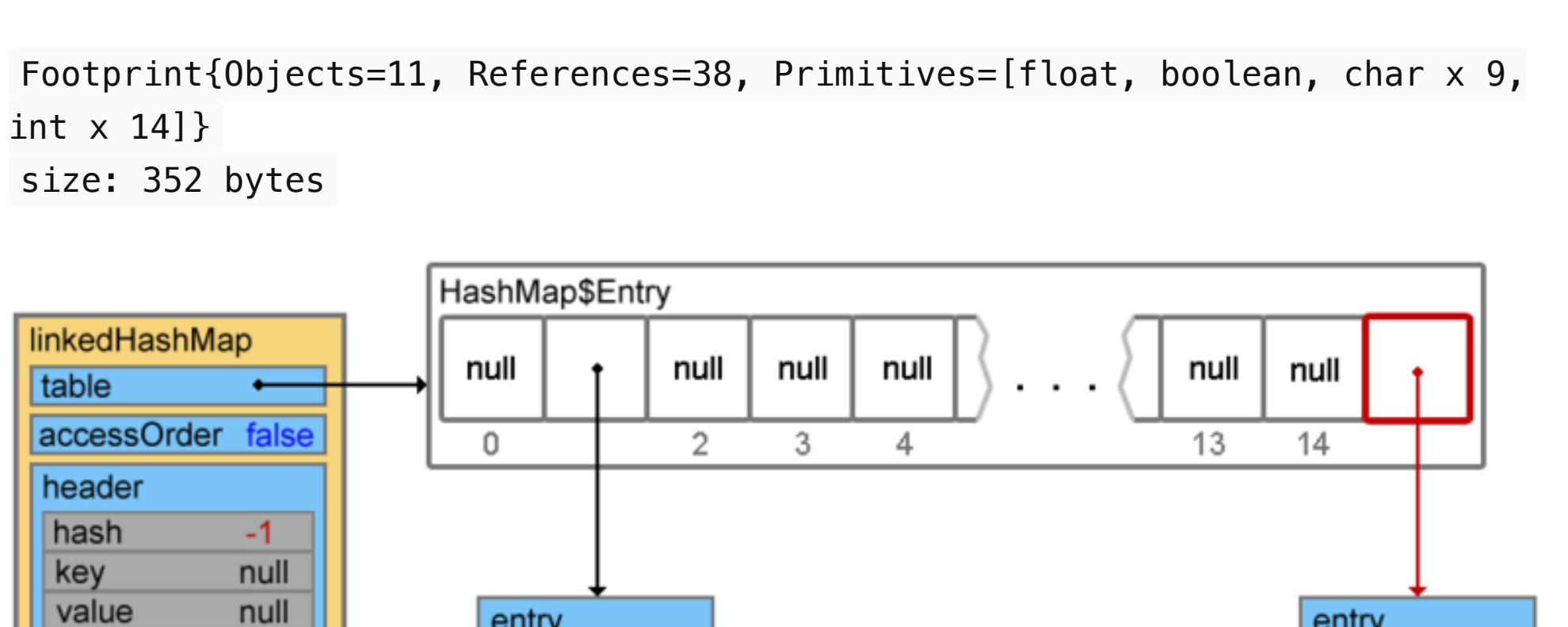


Всё что дальше происходит в методе **addEntry()** либо не представляет «функционального интереса»¹ либо повторяет функционал родительского класса.

Добавим еще парочку элементов

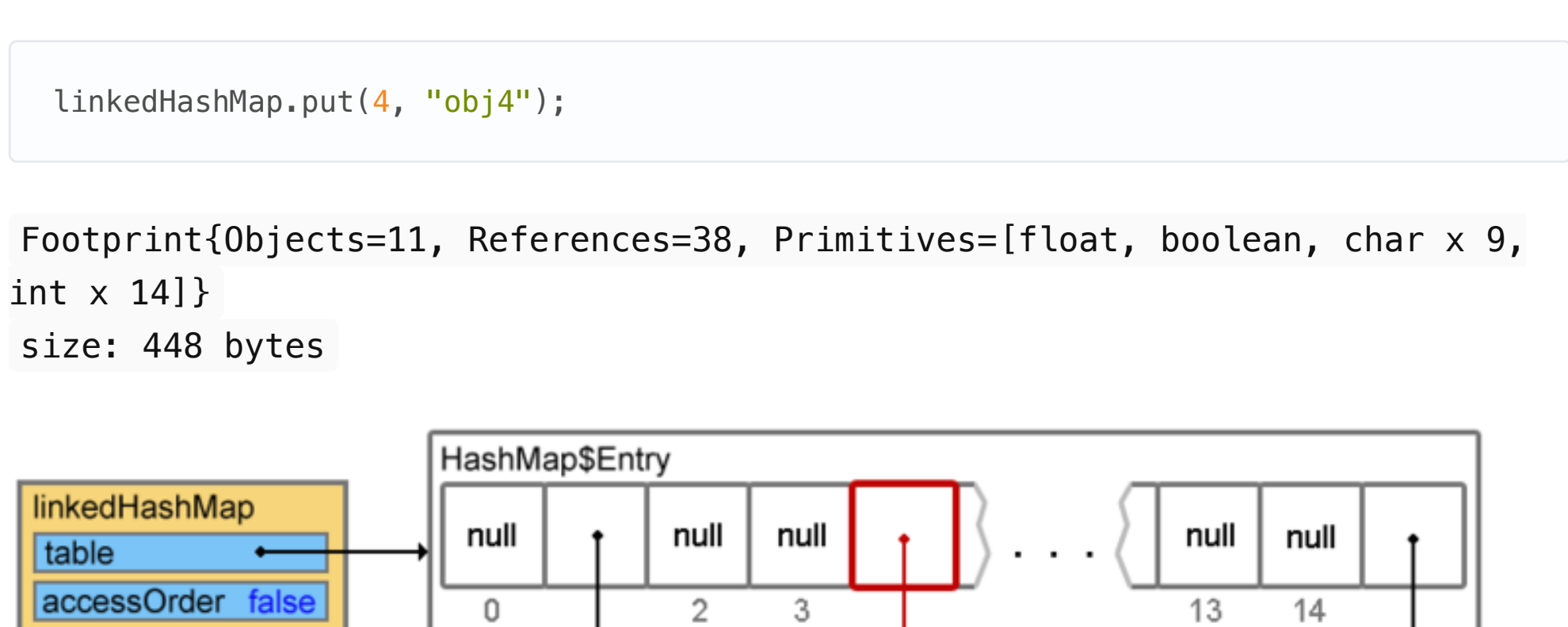
```
linkedHashMap.put(15, "obj15");
```

Footprint{Objects=11, References=38, Primitives=[float, boolean, char x 9, int x 14]}
size: 352 bytes



```
linkedHashMap.put(4, "obj4");
```

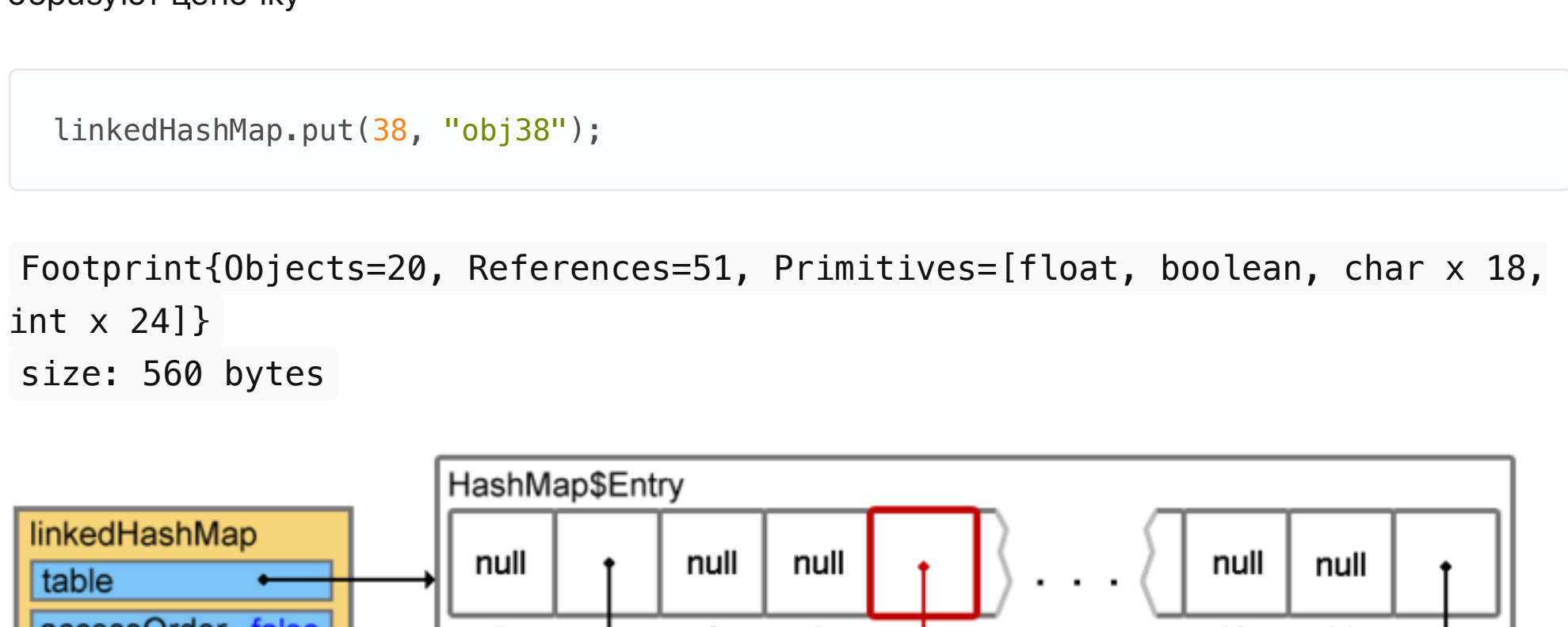
Footprint{Objects=11, References=38, Primitives=[float, boolean, char x 9, int x 14]}
size: 448 bytes



При добавлении следующего элемента происходит коллизия, и элементы с ключами 4 и 38 образуют цепочку

```
linkedHashMap.put(38, "obj38");
```

Footprint{Objects=20, References=51, Primitives=[float, boolean, char x 18, int x 24]}
size: 560 bytes



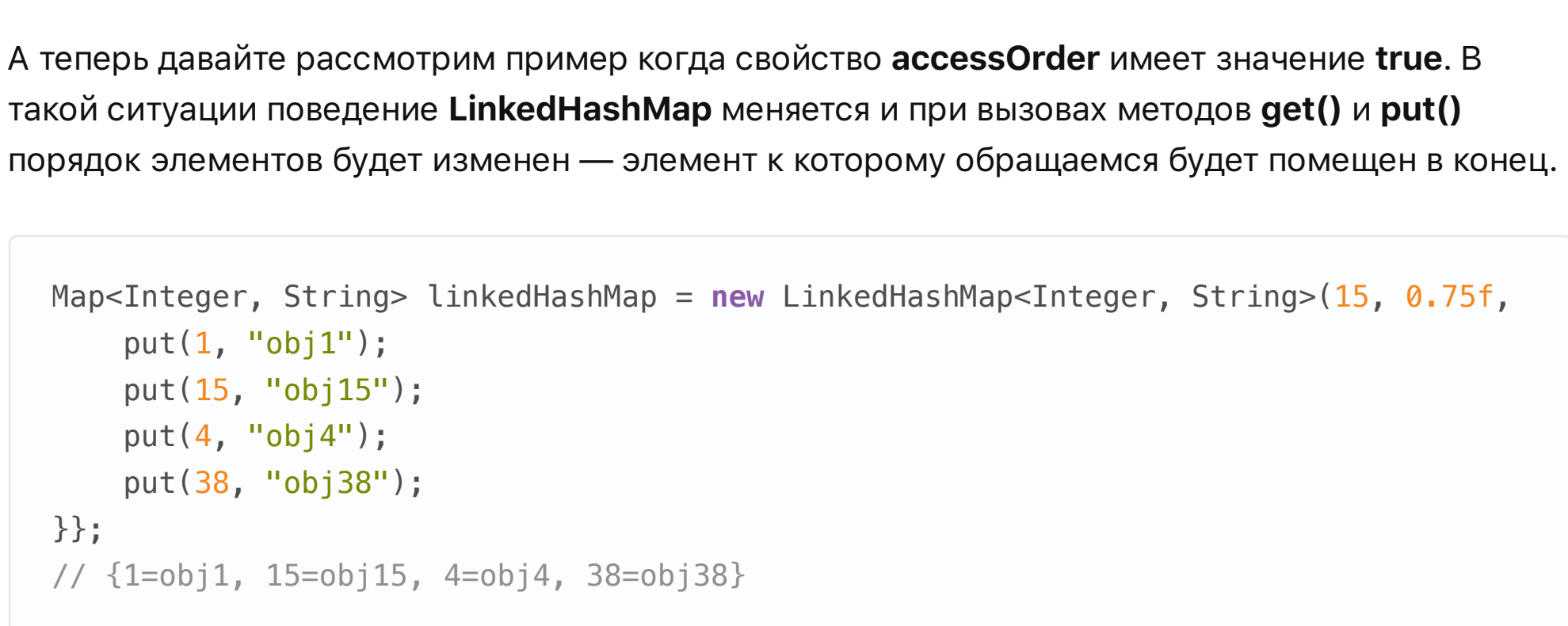
Обращаю ваше внимание, что в случае повторной вставки элемента (элемент с таким ключом уже существует) порядок доступа к элементам не изменится.

accessOrder == true

А теперь давайте рассмотрим пример когда свойство **accessOrder** имеет значение **true**. В такой ситуации 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,
    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}
```



Итераторы

Всё достаточно банально:

```
// 1.
Iterator<Entry<Integer, String>> itr1 = linkedHashMap.entrySet().iterator();
while (itr1.hasNext()) {
    Entry<Integer, String> entry = itr1.next();
    System.out.println(entry.getKey() + " = " + entry.getValue());
}

// 2.
Iterator<Integer> itr2 = linkedHashMap.keySet().iterator();
while (itr2.hasNext())
    System.out.println(itr2.next());

// 3.
Iterator<String> itr3 = linkedHashMap.values().iterator();
while (itr3.hasNext())
    System.out.println(itr3.next());
```

Ну и не забывайте про fail-fast. Коли уж начали перебор элементов — не изменяйте содержимое или заранее позаботьтесь о синхронизации.

Вместо итогов

Данная структура может слегка уступать по производительности родительскому **HashMap**, при этом время выполнения операций **add()**, **contains()**, **remove()** остается константой — $O(1)$. Понадобится чуть больше места в памяти для хранения элементов и их связей, но это совсем небольшая плата за дополнительные фишечки.

Вообще, из-за того что всю основную работу на себя берет родительский класс, серьезных отличий в реализации **HashMap** и **LinkedHashMap** не много. Можно упомянуть о парочке мелких:

- Методы **transfer()** и **containsValue()** устроены чуть проще из-за наличия двунаправленной связи между элементами;
- В классе **LinkedHashMap.Entry** реализованы методы **recordRemoval()** и **recordAccess()** (тот самый, который помещает элемент в конец при **accessOrder = true**). В **HashMap** оба этих метода пустые.

Ссылки

Исходник [LinkedHashMap](#)
Исходники JDK [OpenJDK & trade 6 Source Release — Build b23](#)
Инструменты для замеров — [memory-measurer](#) и [Guava](#) (Google Core Libraries).

¹ — Вызов метода **removeEldestEntry(Map.Entry eldest)** всегда возвращает **false**. Предполагается, что данный метод может быть переопределен для каких-либо нужд, например, для реализации кэширующих структур на основе **Map** (см. [ExpiringCache](#)). После того как **removeEldestEntry()** станет возвращать **true**, самый старый элемент будет удален при превышении макс. количества элементов.