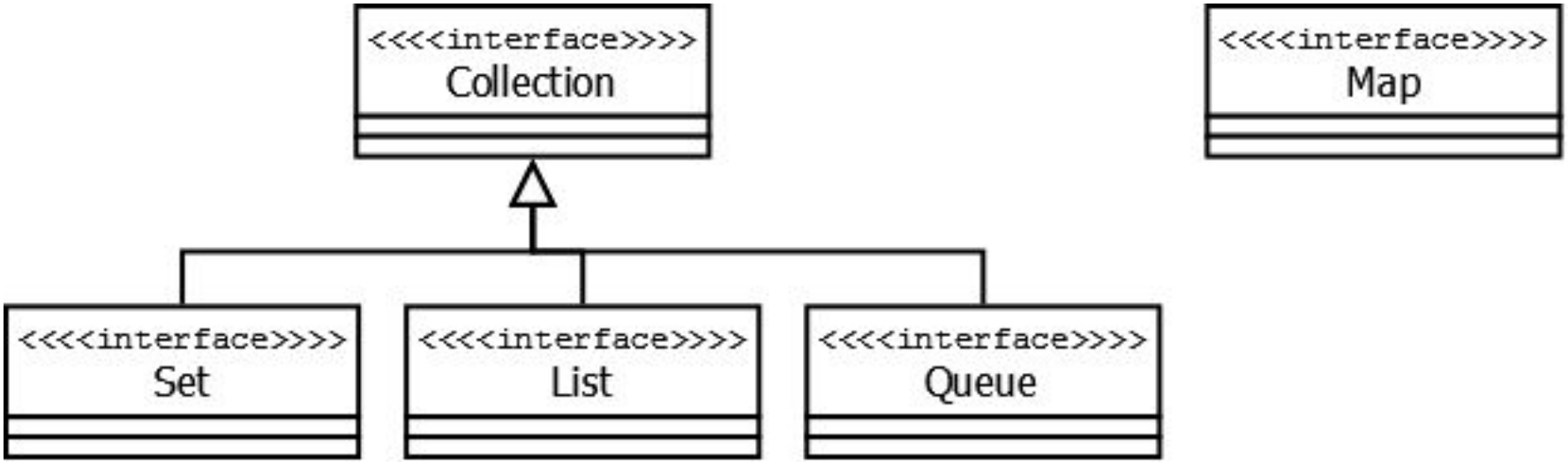
Java

HashMap

Базовые понятия

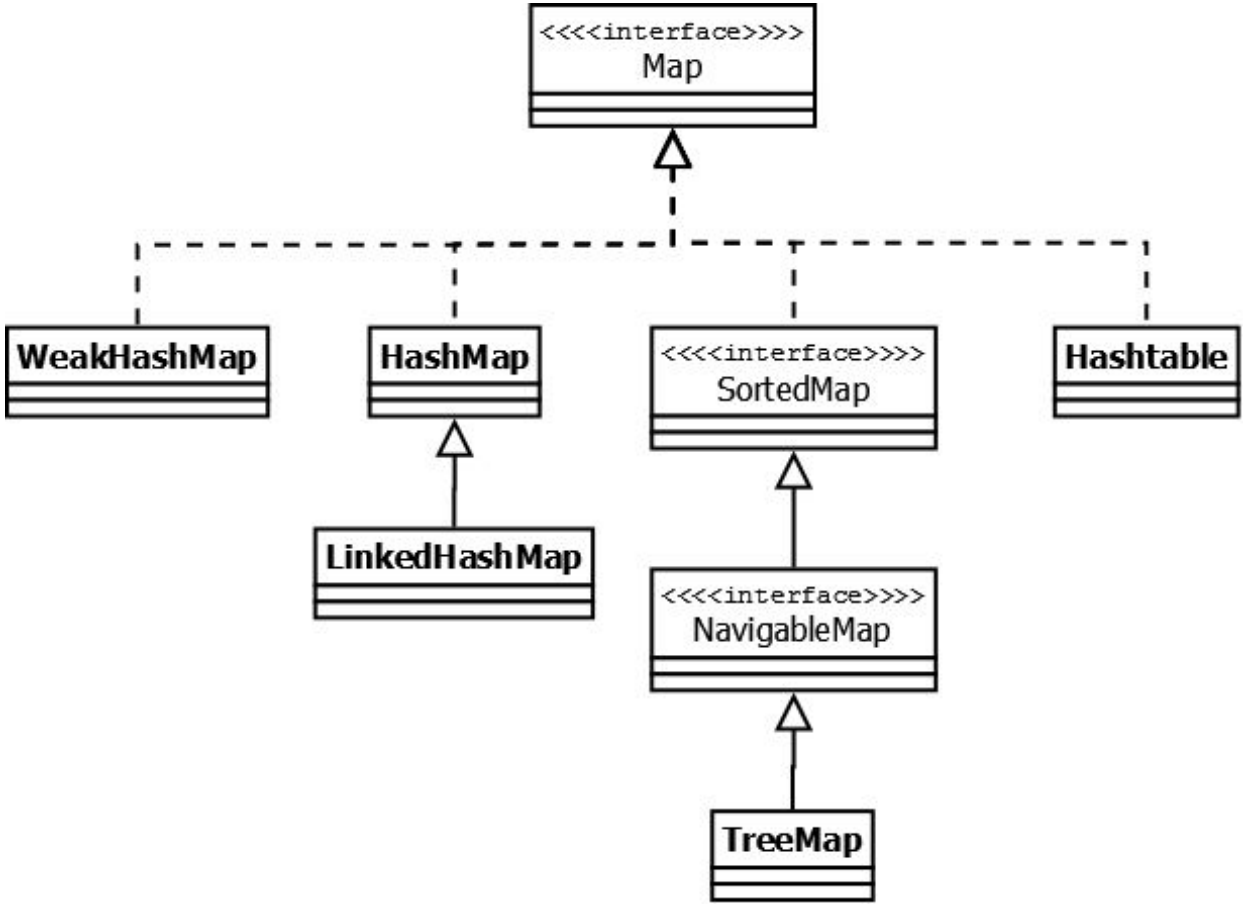


Map

Map. Данный интерфейс находится в составе JDK с версии 1.2 и предоставляет разработчику базовые методы для работы с данными вида «ключ — значение».

Также как и Collection, он был дополнен дженериками в версии Java 1.5 и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые зачастую реализовались в логике приложения (getOrDefault(Object key, V defaultValue), putIfAbsent(K key, V value)).

Map


<<<<interface>>>>
Map
WeakHashMap
HashMap
<<<<interface>>>>
SortedMap
Hashtable
LinkedHashMap
<<<<interface>>>>
NavigableMap
TreeMap

HashMap

Хеш-таблицей называется структура данных, реализующая интерфейс ассоциативного массива (абстрактная модель «ключ – значение» или entry), которая обеспечивает очень быструю вставку и поиск: независимо от количества элементов вставка и поиск (а иногда и удаление) выполняются за время, близкое к константе – O(1). По сути, это обычный массив, где местоположение элемента зависит от значения самого элемента. Связь между значением элемента и его позицией в хеш-таблице задает хеш-функция.

Хеш-функция получает входную часть данных, которую мы называем ключом, а на выходе она выдает целое число, известное как хеш-значение (или хеш-код). Затем, хеш-значение привязывает наш ключ к определенному индексу хеш-таблицы. Для основных операций: вставки, поиска и удаления мы используем одну и ту же хеш-функцию, поэтому эти операции осуществляются довольно быстро. По этой причине важно, чтобы хеш-функция вела себя последовательно и выводила один и тот же индекс для одинаковых входных данных.

HashMap

Класс HashMap наследуется от класса AbstractMap и реализует следующие интерфейсы: Map, Cloneable, Serializable.

За хеш-функцию в Java отвечает метод hashCode(). Реализация по умолчанию hashCode() возвращает значение, которое называется идентификационный хеш (identity hash code). Даже если класс переопределяет hashCode(), вы всегда можете получить идентификационный хеш объекта с помощью вызова System.identityHashCode(Object o). Реализация по умолчанию hashCode() в OpenJDK не имеет ничего общего с адресом памяти, как это порой принято считать.

HashMap

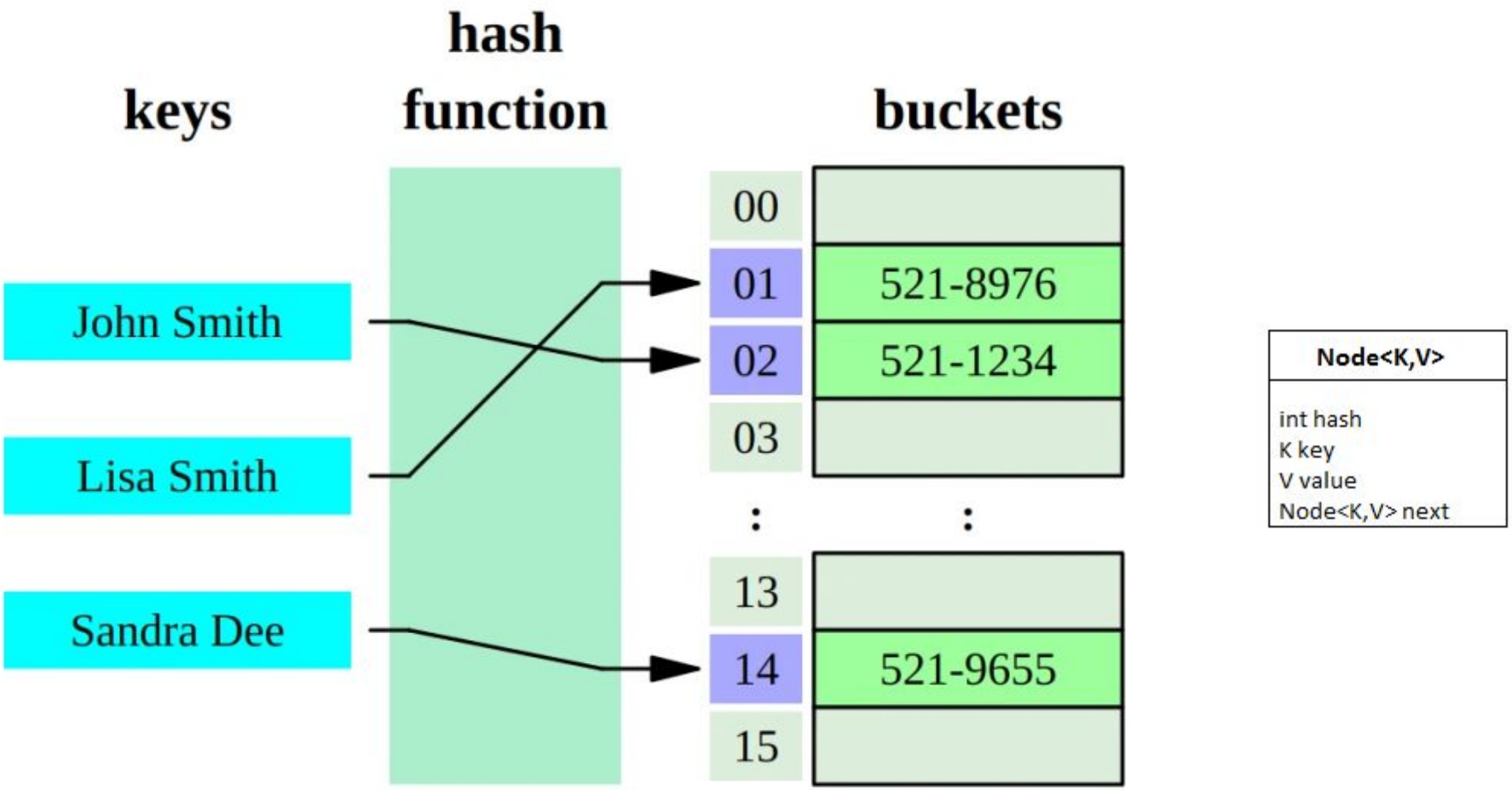
В HashMap хеш-таблица реализована на основе массива (а если точнее — динамического, так как таблица может расширяться) односвязных списков.

По сути, мы получаем хеш-код ключа в результате работы метода hashCode(), который затем модифицируется, а внутри с помощью дополнительного метода полученные значения распределяются по нужным ячейкам.

Элементы массива (ячейки) еще называются корзинами «buckets», которые используются для хранения отдельно взятых узлов.

Каждый из бакетов представляет из себя коллекцию (список или дерево).

Узел представляет собой объект вложенного класса Node (или TreeNode при древовидной структуре). По сути, внутри ячейки массива лежит LinkedList, только список односвязный, либо красное-черное дерево, которое лежит в основе реализации другого класса — TreeMap.


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655
Node<K,V>
int hash
K key
V value
Node<K,V> next

HashMap. Поля класса.

- transient Node < K, V> [] table – сама хеш-таблица, реализованная на основе массива, для хранения пар «ключ-значение» в виде узлов. Здесь хранятся наши Node;
- transient int size — количество пар «ключ-значение»;
- int threshold — предельное количество элементов, при достижении которого размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле (capacity * loadFactor);
- final float loadFactor — этот параметр отвечает за то, при какой степени загруженности текущей хеш-таблицы необходимо создавать новую хеш-таблицу, т.е. как только хеш-таблица заполнилась на 75%, будет создана новая хеш-таблица с перемещением в неё текущих элементов (затратная операция, так как требуется перехеширование всех элементов);
- transient Set< Map.Entry< K,V>> entrySet — содержит кешированный entrySet(), с помощью которого мы можем перебирать HashMap.

HashMap. Константы

- static final int DEFAULT_INITIAL_CAPACITY= 1 << 4 — емкость хеш-таблицы по умолчанию (16);
- static final int MAXIMUM_CAPACITY = 1 << 30 — максимально возможная емкость хеш-таблицы (приблизительно 1 млрд.);
- static final float DEFAULT_LOAD_FACTOR = 0.75f — коэффициент загрузки, используемый по умолчанию;
- static final int TREEIFY_THRESHOLD = 8 — это «порог» количества элементов в одной корзине, при достижении которого внутренний связный список будет преобразован в древовидную структуру (красно-черное дерево).
- static final int UNTREEIFY_THRESHOLD = 6 — если количество элементов в одной корзине уменьшится до 6, то произойдет обратный переход от дерева к связному списку;
- static final int MIN_TREEIFY_CAPACITY = 64 — минимальная емкость (количество корзин) хеш-таблицы, при которой возможен переход к древовидной структуре. Т.е. если в хеш-таблице по крайней мере 64 бакета и в одном бакете 8 или более элементов, то произойдет переход к древовидной структуре.

HashMap. Конструкторы

1. public HashMap() — создает хеш-отображение по умолчанию: объемом (capacity) = 16 и с коэффициентом загруженности (load factor) = 0.75;
2. public HashMap(Map< ? extends K, ? extends V> m) — создает хеш-отображение, инициализируемое элементами другого заданного отображения с той начальной емкостью, которой хватит вместить в себя элементы другого отображения;
3. public HashMap(int initialCapacity) — создает хеш-отображение с заданной начальной емкостью. Для корректной и правильной работы HashMap размер внутреннего массива обязательно должен быть степенью двойки (т.е. 16, 64, 128 и т.д.);
4. public HashMap(int initialCapacity, float loadFactor) — создает хеш-отображение с заданными параметрами: начальной емкостью и коэффициентом загруженности.

Добавление объекта в HashMap

Добавление пары "ключ-значение" осуществляется с помощью метода put(). Рассмотрим шаги, выполняемые при добавлении объекта:

1.Вычисляется хеш-значение ключа введенного объекта. Хэш ключа вычисляет метод static final int hash(Object key), который уже обращается к известному нам методу hashCode() ключа. Для этого используется либо переопределенный метод hashCode(), либо его реализация по умолчанию.

2.Вычисляем индекс бакета (ячейки массива), в который будет добавлен наш элемент:

i = (n - 1) & hash // n – длина массива.

3.Создается объект Node.

4.Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Если в бакете пусто, тогда просто размещаем в нем элемент. Иначе хэш и ключ нового элемента поочередно сравниваются с хешами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается. Если совпадений не найдено, элемент добавляется в конец списка.

Добавление объекта в HashMap. Подробнее

1.Создадим объект класса HashMap:

HashMap < String, Integer> map = new HashMap<>();

2.С помощью метода put() добавим в хеш-отображение первую пару «ключ-значение»:

map.put("KING", 100);

В этот момент внутри вызывается метод *putVal*().

3.С помощью хеш-функции, роль которой играет метод hash, вычисляется хеш-код ключа, внутри которого предварительно вычисляется хеш-код с помощью метода hashCode() (в данном случае класса String), в результате чего мы получаем его «предварительное значение» – 2306967. Может проверить в IDEA с помощью

System.out.println("KING".hashCode());

Полученный хеш-код модифицируется по формуле: (хеш-код) ^ (хеш-код>>> 16), и в результате получаем окончательный хеш-код – 2306996.

Добавление объекта в HashMap. Подробнее

4.Проверяем таблицу на «пустоту»:

if ((tab = table) == null || (n = tab.length) == 0)

где [] tab — сама хеш-таблица: ссылаем tab и table (напомню, что это поле класса HashMap, которое хранит массив для наших элементов) на один и тот же массив, а int n – дополнительная переменная-счетчик.

Так как проверка вернёт true (потому что массив для таблицы не был создан с помощью оператора new в конструкторе), будет вызван метод resize(), который собственно и создаст таблицу размером на 16 элементов. Да-да, конструкторы класса никакой таблицы не создают. Вместо этого всегда происходит вызов метода resize() при первом добавлении элемента. Длина созданной таблицы (считай длина массива) будет записана в переменную n

n = (tab = resize()).length;

которая в дальнейшем используется для вычисления бакета.

Добавление объекта в HashMap. Подробнее

5.Одновременно вычисляем индекс бакета, куда будет помещен наш объект, и проверяем, а есть ли уже в нем элементы. Вычисляем:

i = (n - 1) & hash

i = (16 - 1) & 2306996

i = 4

проверяем:

if ((p = tab[i = (n - 1) & hash]) == null)

Добавление объекта в HashMap. Подробнее

6.Так как в результате проверки мы получим true (в бакете элементов нет), будет сгенерирован объект Node со следующими полями:

{
int hash = 2306996 — сгенерированный хеш-код
String key = {"KING"} — ключ
Integer value = 100 — значение
Node next = null — ссылка на следующий узел
}

Node

hash	2306996
key	"KING"
value	100
next	null

Наш сформированный объект Node будет добавлен в бакет под индексом [4]:
tab[i] = newNode(hash, key, value, null);
tab[4] = newNode(2306996, “KING”, 100, null);
newNode() — это метод, который просто возвращает объект класса Node.

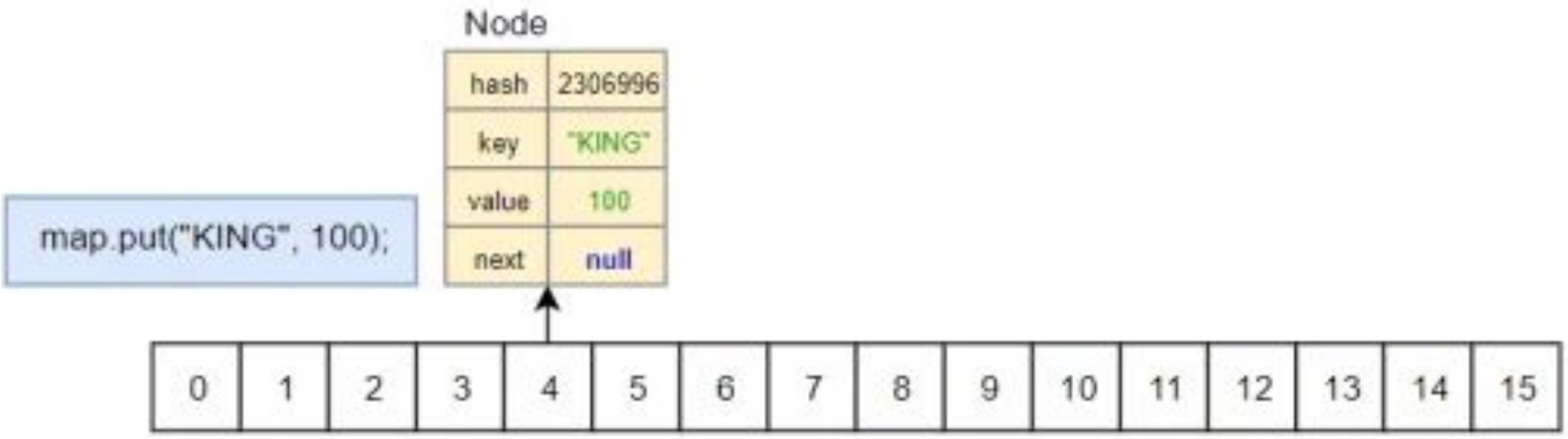
Добавление объекта в HashMap. Подробнее

7.После добавления будет произведена проверка не превышает ли текущее количество элементов параметр threshold:

```
if (++size > threshold)
    resize();
```

Если превышает, тогда будет вызван метод resize() для увеличения размера хеш-таблицы.

На этом метод *putVal*() (соответственно и *put*()) завершит свою работу.




```
public V put(K key, V value)
        hash(Object key)
        index = (n - 1) & hash
```

hash("KING") = 2306996

index = 4

n = 16

Добавление объекта в HashMap. Коллизии

1.С помощью метода put() добавим в хеш-отображение еще одну пару «ключ-значение»:

map.put("BLAKE", 10);

2.Вычисляем "предварительный хеш" – 63281361. Модифицируем его и в результате получаем окончательный хеш-код – 63281940.

3.Так как первая проверка на «пустоту» теперь вернет false (создавать таблицу не надо), сразу вычисляем индекс бакета, куда будет помещен наш объект:

i = (n - 1) & hash
i = (16 - 1) & 63281940
i = 4

4.Бакет под указанным индексом проверяется на наличие в нем элементов и так как условие if ((p = tab[i = (n - 1) & hash]) == null) в этом случае не выполняется (в бакете уже есть элемент), тогда переходим к блоку else.

Добавление объекта в HashMap. Коллизии

5.В первую очередь мы сравниваем добавляемый элемент с первым элементом связного списка внутри бакета:

(p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))

При проверке сначала сравниваются хеши ключей. Если этот «участок» (p.hash == hash) возвращает false, тогда остальная часть условия игнорируется (&&), так как объекты гарантированно разные. Иначе затем сравниваются ключи по ссылке (==) и в случае неравенства, ключи сравниваются уже посредством метода equals(). Сравнение осуществляется в таком порядке во благо производительности.

Добавление объекта в HashMap. Коллизии

Если все три выражения возвращают true, это означает, что ключи равны и новый элемент будет записан в дополнительную переменную, чтобы в дальнейшем с её помощью перезаписать значение у ключа:

```
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null)
    e.value = value;
    afterNodeAccess(e);
     return oldValue;
 }
```

В результате сравнения ключей мы получаем false уже на первом этапе (разные хеши).

Добавление объекта в HashMap. Коллизии

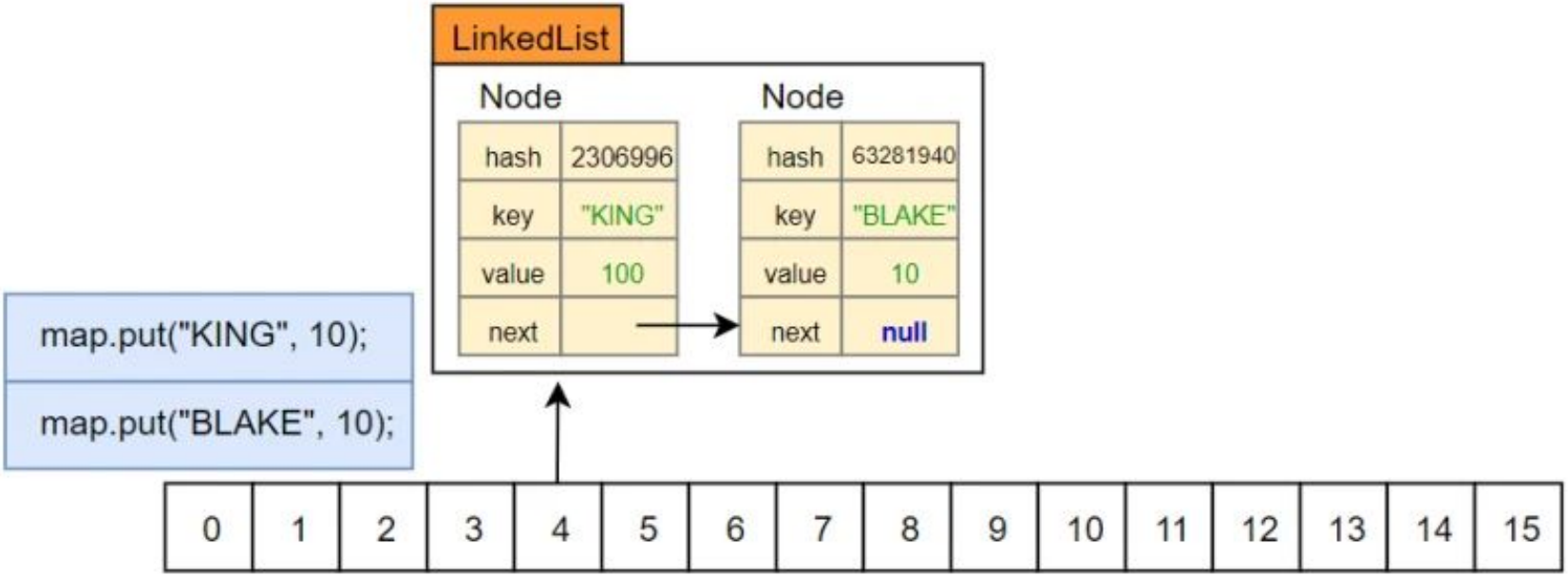
6.Игнорируем условие (p instanceof TreeNode), так как структура в бакете не является древовидной на данном этапе.

7.Далее переходим в цикл for, где в пределах одного бакета проверяем у элементов указатель на следующий элемент next, и если он равен null (значит элемент в списке последний и единственный), добавляем новый элемент Node в конец списка:

```
if ((e = p.next) == null){
        p.next = newNode(hash, key, value, null)
... };
```

Если же при первой итерации указатель не равен null, это говорит о том, что в списке как минимум два элемента, поэтому в таком случае мы переходим к следующему условия и сравниваем ключ добавляемого элемента со всеми ключами элементов в списке (способом, описанным в пятом пункте).

```
if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
```




```
public V put(K key, V value)
        hash(Object key)
        index = (n - 1) & hash
```

hash("BLAKE") = 63281940

index = 4

n = 16

Добавление объекта в HashMap (в пределах одного бакета)

- проверяем с помощью методов hashCode() и equals(), что оба ключа одинаковы.
- если ключи одинаковы, заменить текущее значение новым.
- иначе связать новый и старый объекты с помощью структуры данных "связный список", указав ссылку на следующий объект в текущем и сохранить оба под нужным индексом; либо осуществить переход к древовидной структуре

Добавление объекта в HashMap.

После каждой итерации (добавления нового элемента) в цикле for увеличивается счетчик, который отвечает за количество элементов в бакете:

for (int binCount = 0; ; ++binCount)

До тех пор, пока их количество не станет равно или больше 7:

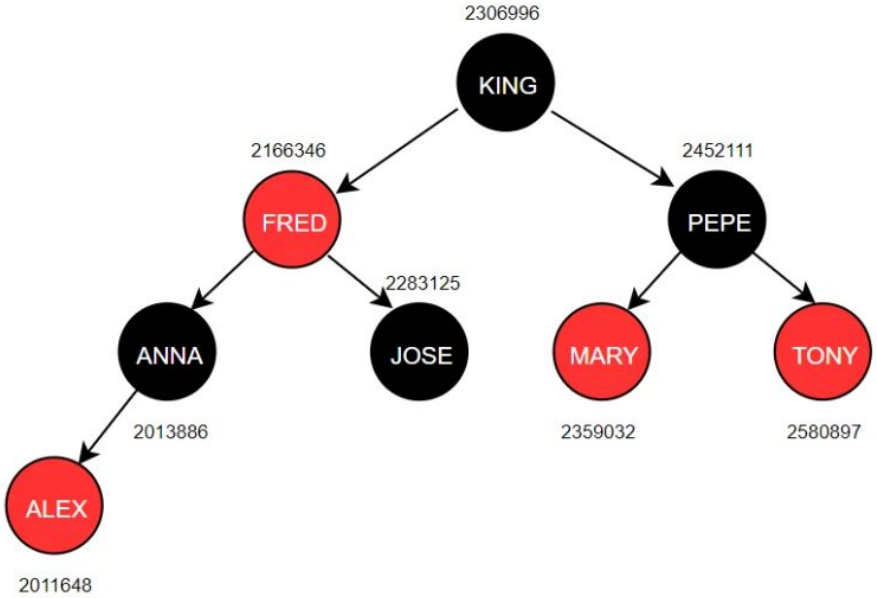
binCount >= TREEIFY_THRESHOLD – 1

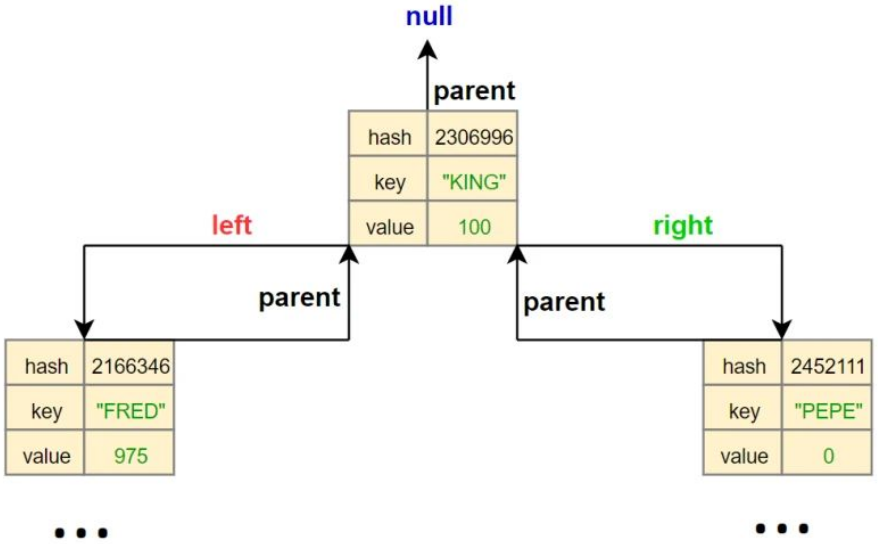
В таком случае произойдет вызов метода treeifyBin() и treeify() для непосредственного построения древовидной структуры. Однако, если количество бакетов в текущей хеш-таблице меньше 64:

if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)

Вместо перехода к древовидной структуре будет вызван метод resize() для увеличения размера хеш-таблицы с перераспределением элементов. treeify() в свою очередь связный список из TreeNode конвертирует в красно-черное дерево. Метод resize() перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.

Вид Красно-Чёрного дерева внутри бакета





Получение элементов (извлечение значения по ключу)

Алгоритм (когда в бакете связный список) можно записать так:

1. Вычислить хэш код ключа.
2. Вычислить индекс бакета.
3. Перейти по индексу и сравнить ключ первого элемента с имеющимся значением. Если они равны – вернуть значение, иначе выполнить проверку для следующего элемента, если он существует.
4. Если следующий объект Node равен null, возвращаем null.
5. Если следующий объект Node не равен null, переходим к нему и повторяем первые три шага до тех пор, пока элемент не будет найден или следующий объект Node не будет равен null.

Временна́я сложность

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

Источники

1. Справочник по Java Collection Framework
2. Подробный разбор класса HashMap
3. https://blog.skillfactory.ru/glossary/hashmap/
4. Видео (JavaRush)

P.S.

Никита Небольсин Уровень 31

Это божественно! Статья обязательна к прочтению всем!

Ann Kudra Уровень 47 EXPERT 3 октября 2023

Мне кажется, для студента на 17ом уровне еще рановато. Оставлю-ка я её в закладках до лучших времен.

Alexey Bril Уровень 4

Благодарю за статью. Получилось уложить хешмапу в голове

Anonymous #2657901 Уровень 1

надо покурить

Ольга Уровень 108 EXPERT 29 ноября 2022

Слушьте, ну на 18 уровне Java Syntax это выглядит просто целенаправленным взрывом мозга студента, ссылка на статью хоть и позиционируется как доп материал, но тем не менее ввергает новичка в уныние с каждым следующим абзацем. Отложу-ка я ее на далёкое потом...