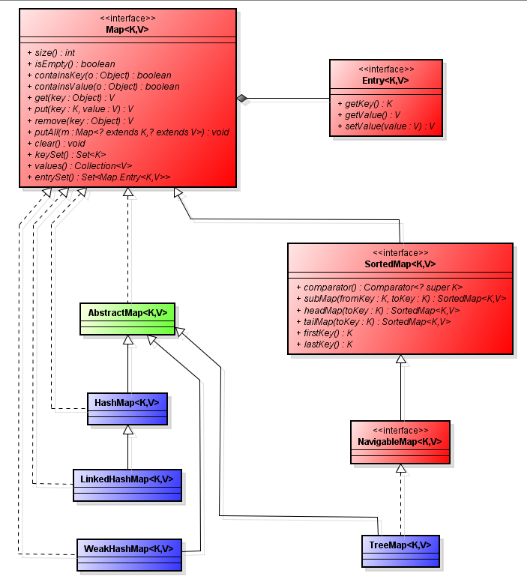
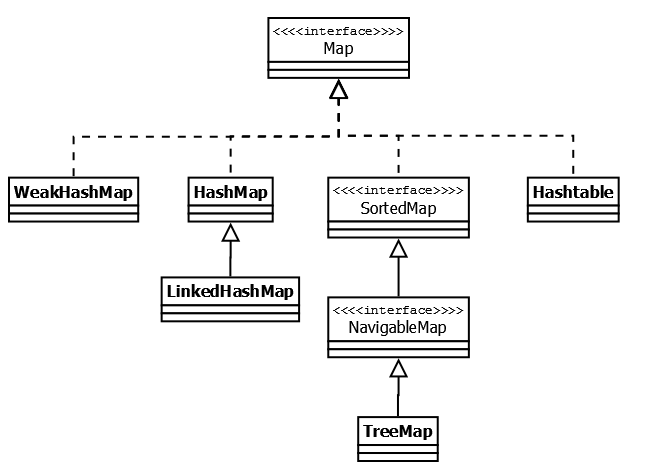
**MAP**

(АССОЦИАТИВНЫЙ МАССИВ.  
ХРАНЕНИЕ ОБЬЕКТОВ В ВИДЕ ПАР: КЛЮЧ-ЗНАЧЕНИЕ)





**HashMap**

Основана на хэш-таблицах, ключи и значения могут быть любого типа в том числе null. Хранение элементов не структурированно, может меняться с течением времени. Не синхронизирован. Состоит из так называемых “корзин”.

Только что созданный объект HashMap содержит в себе:

массив **table** типа **Entry <K,V>[]** - это хранилище ссылок на цепочки значений (связанные списки).

**loadFactor** – поле содержит коэффициент загрузки (стандартно 0.75).

**threshold** – содержит рассчитанное предельное ко-во элементов, при достижении которого происходит увеличение (x2) и переопределение hashMap (capacity \* loadFactor).

**size** - кол-во элементов hashMapa. Изначально состоит из 16 позиций (корзин).

После добавления пары в массив Node [] table, элемент будет состоять из 4 подэлементов – это **hash** (hashCode вычесленный по key), **key** (сам наш ключ в оригинале), **value** (значение, передаётся вторым параметром после key), **next** (указатель на следующий узел под данным ключом).

Можно указать емкость и коэффициент загрузки для нового HashMap, используя конструкторы HashMap(capacity) и HashMap(capacity, loadFactor). Максимальная емкость, которую можно установить, равна половине максимального значения int (1073741824).

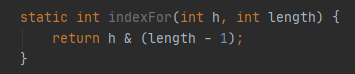
При удалении элементов из hashMap – размер коллекции не уменьшается, как выход можно на основе раздолбаной коллекции – создать новую (передав в параметры прошлую map). Или просто использовать метод подгонки размера.

**Порядок добавления:**

-Ключ проверяется на равенство null. Если true вызывается метод putForNullKey(value);

-Генерируется хэш на основе ключа, используется метод hash(hashCode)в который передаётся key.hashCode(), особенность этого метода в том – что уменьшает кол-во коллиций(примерно 8 при 0.75 loadFactor).

-С помощью метода indexFor(hash, tableLength)определяется позиция из возможных (со старта 16) куда будет помещён объект, hash сжимается с учётом tableLength и уже из возможных выбирается индекс.



**-**Зная индекс мы получаем цепочку элементов привязанных к этой ячейке. HashCode и key нового элемента поочерёдно сравниваются, с HashCode и key элементов из этой цепочки. Если есть совпадение – перезаписывается значение.

-Если НЕТУ – то создаётся ещё одно звено под данным индексом методом addEntry(hash, key, value, index); Новый элемент всегда помещается в начало.

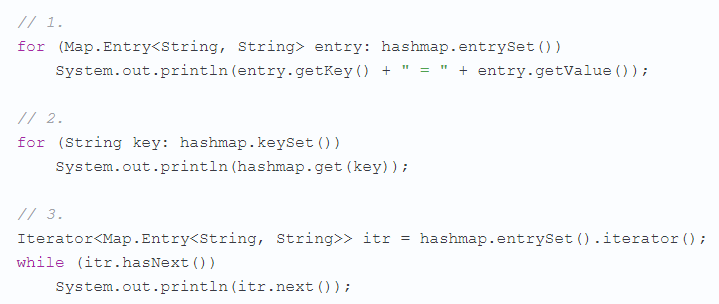
-Если добавляю null элемент – он добавляется в entry[0] позицицю. Это место в массиве предназначено для ключей = null, хэш код для данных ключей не высчитывается, и не определяется место в таблице. Потом поочерёдно просматриваются все элементы в данной цепочке и при совпадении ключа – данные перезаписываются.

Методы **transfer** и **resize**

Методы вызываются в момент увеличения коллекции (при заполнении до предельного размера) – служит для перераспределения всех объектов в hashMap (пересчитывает индексы всех элементов с учётом нового размера и перераспределяет элементы по новому массиву).

**Итераторы**

Все способы получения ключей\значений или того и другого сразу:

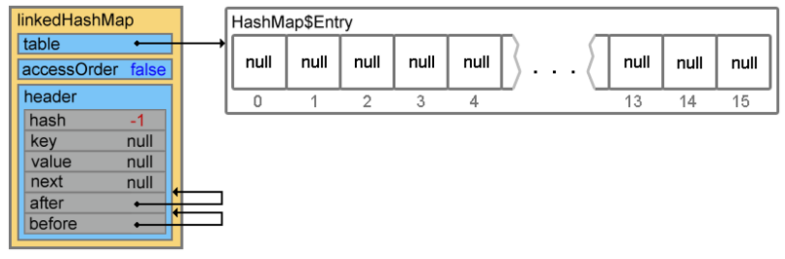


**LinkedHashMap**

Наследуется от HashMap поэтому имеет схожую структуру, но помимо этого при создании обьекта данного класса, внутри появляется 2 доп. свойства:

**header** – голова двусвязного списка, при инициализации указывает сама на себя. Значение = null. Ссылается при добавлении элементов на следующий и предыдущий элемент.

**accessOrder** – указывает каким образом будет осуществляться сортировка элеменов (true – по порядку последнего доступа (последний в коллекции лежит элемент – к которому обращались в последний раз), false – доступ в том порядке в котором добавлялись), влияют на порядок только методы get() и put().



В этой реализации есть метод removeEldestEntry() который вызывается автоматически после вставки элемента в коллекцию (можно этот метод переопределить и сделать свой небольшой кэш (LRU-Cache), в котором менее используемые элементы будут автоматически удаляться accessorder = true), при этом нужно сделать что бы map не увеличивалась loadFactor >1.

**TreeMap**

Используется в случае необходимости хранения данных в структурированном виде, с возможностью навигации по данным. Сортировка происходит по ключу. Может хранить null в качестве ключа – только если не используется компаратор. Под капотом данный класс использует структуру данных называемую “красно-чёрное дерево”. Время доступа к элементам за О(ln(n)). **Красно-чёрное дерево** – это сбалансированное бинарное дерево поиска, частный случай двоичного дерева поиска (ДДП).

Если создали свой собственный класс в качестве ключа, и не реализовали в нём Comparable и не используете Comparator – при попытке внести пару с данным ключём будет выброшено исключение ClassCastException.

Поведение сортировки может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра при создании объекта TreeMap.

Есть дополнительные полезные методы от имплементации NavigableMap.

**Свойства бинарного дерева:**

-Поддеревья тоже являются бинарными деревьями поиска

-Для каждого узла, левый потомок < узла, правый потомок > узла

**Есть условия, которые должны быть соблюдены при постройки красно-чёрного дерева:**

-Корень должен быть окрашен в чёрный цвет

-Листья дерева должны быть чёрного цвета

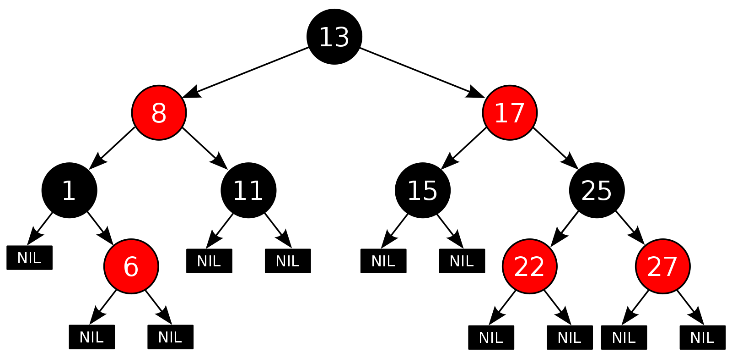
-Красный узел должен иметь 2 чёрных узла

-Чёрный узел может иметь любые дочерние узлы

-Путь от корня к листьям, должен содержать равное число чёрных узлов (чёрная высота), но это не всегда так – если элементов мало (максимальная разница в пути может быть = 1)

-Новые узлы добавляются на места листьев

Вывод: путь через чёрные узлы всегда короче, по этому по наличию чёрных узлов – всегда определяется общий размер дерева (называется этот размер “чёрная высота”).



**Вставка элемента в красно-чёрное дерево:**

-Элемент вставляется на позицию листа и окрашивается в красный цвет

-Происходит проверка свойств и в случае несоответствия перебалансировка

**Перебалансировка**

N – текущий узел.

P – предок.

U – дядя (сосед предка).

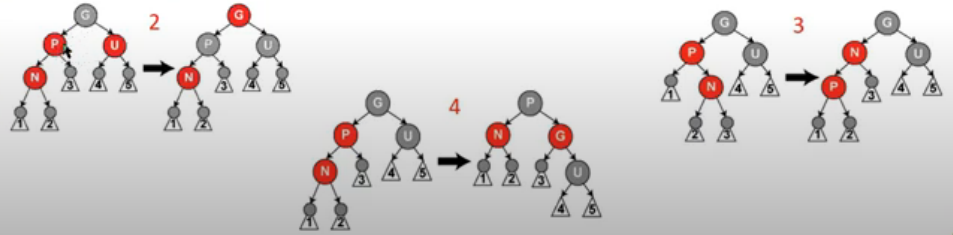
G – дедушка (предок предка)

1. Если N оказывается первым в дереве (корень) – перекрашиваем в чёрный.

2. Если P и U красные, то они оба могут быть перекрашены в чёрный, а G станет красным. Но в таком случае для G проверяется свойство перебалансировки 1.

3. Если P красный, а U – чёрный, при этом N - правый потомок, а Р – левый. В этом случае может быть произведён поворот дереве, меняющий N и P. Но свойство 4 всё ещё нарушено – переход к случаю 4.

4. Если Р – красный, но U – чёрный, при этом N и P – левые потомки. Тогда выполняется поворот дерева на G. А так как Р был красным, значит G был чёрным, потому цвета G и Р меняются местами.



Хорошо рассказывает про перебалансировку: https://www.youtube.com/watch?v=oxBVSD35gk0&list=PLhjky2UFu-C2TvBdCmdnlizX83drnd01J&index=9

**EnumMap**

Ключи в данной реализации – это Enum. Плюс в том что ключей будет = количеству Enum. Компактно и эффективно.

**WeakHashMap**

Слабые ключи. Элемент удаляется из этой реализации map когда ключ больше не используется где то в коде (в других реализациях map всегда есть под капотом ссылка на какой то ключ, что бы garbage collected не добрался). В этой же реализации – при неиспользовании элемент удаляется. Подходит в качестве временного хранилища.

Под ключом собирается какой то объект с разных мест, после его отправки куда либо – он уже не нужен, автоматически удаляется из map. Временное хранилище – с возможностью автоматического очищения.

public class RunnerWeakHashMap {  
 public static void main(String[] args) {  
 Map<Object,String> map = new WeakHashMap<>();  
 Object o1 = new Object();  
 map.put(o1, "myObj");  
  
 o1 = null; //теперь GC получил доступ и может удалить элемент из коллекции  
 System.*gc*();  
  
 for (int i = 0; i <= 1000; i++){  
 if(map.isEmpty()){  
 System.*out*.println("Map is empty! iter = " + i);  
 break;  
 }  
 }  
 }  
}