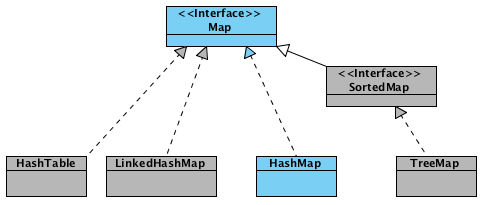
Структуры данных в картинках. HashMap

[JAVA](http://habrahabr.ru/hub/java/)\*

Приветствую вас, хабрачитатели!  
  
Продолжаю попытки визуализировать структуры данных в Java. В предыдущих сериях мы уже ознакомились с [ArrayList](http://habrahabr.ru/blogs/java/128269/) и [LinkedList](http://habrahabr.ru/blogs/java/127864/), сегодня же рассмотрим HashMap.  
  
  
  
*HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Разрешение коллизий осуществляется с помощью*[*метода цепочек*](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0#.D0.9C.D0.B5.D1.82.D0.BE.D0.B4_.D1.86.D0.B5.D0.BF.D0.BE.D1.87.D0.B5.D0.BA)*.*

Создание объекта

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

Footprint{Objects=2, References=20, Primitives=[int x 3, float]}  
Object size: 120 bytes  
  
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. Словом, ничего необычного.

// Инициализация хранилища в конструкторе

// capacity - по умолчанию имеет значение 16

table = new Entry[capacity];

[[http://habrastorage.org/storage1/f5998744/554eeee7/3d597647/2f404c04.png](http://habrastorage.org/storage1/c7aef3f0/b31342f1/a6194b46/1c4a2384.png)](http://habrastorage.org/storage1/c7aef3f0/b31342f1/a6194b46/1c4a2384.png)  
Вы можете указать свои емкость и коэффициент загрузки, используя конструкторы **HashMap(capacity)** и **HashMap(capacity, loadFactor)**. Максимальная емкость, которую вы сможете установить, равна половине максимального значения **int** (1073741824).

Добавление элементов

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

Footprint{Objects=7, References=25, Primitives=[int x 10, char x 5, float]}  
Object size: 232 bytes  
  
При добавлении элемента, последовательность шагов следующая:

1. Сначала ключ проверяется на равенство null. Если это проверка вернула true, будет вызван метод **putForNullKey(value)**(вариант с добавлением null-ключа рассмотрим чуть [позже](http://habrahabr.ru/post/128017/#putForNullKey)).
2. Далее генерируется хэш на основе ключа. Для генерации используется метод **hash(hashCode)**, в который передается**key.hashCode()**.
3. static int hash(int h)
4. {
5. h ^= (h >>> 20) ^ (h >>> 12);
6. return h ^ (h >>> 7) ^ (h >>> 4);
7. }

Комментарий из исходников объясняет, каких результатов стоит ожидать — *метод****hash(key)****гарантирует что полученные хэш-коды, будут иметь только ограниченное количество коллизий (примерно 8, при дефолтном значении коэффициента загрузки).*  
  
В моем случае, для ключа со значением **''0''** метод **hashCode()** вернул значение 48, в итоге:

h ^ (h >>> 20) ^ (h >>> 12) = 48

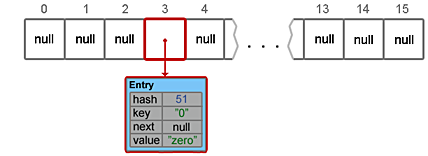
h ^ (h >>> 7) ^ (h >>> 4) = 51

1. С помощью метода **indexFor(hash, tableLength)**, определяется позиция в массиве, куда будет помещен элемент.
2. static int indexFor(int h, int length)
3. {
4. return h & (length - 1);
5. }

При значении хэша **51** и размере таблице **16**, мы получаем индекс в массиве:

h & (length - 1) = 3

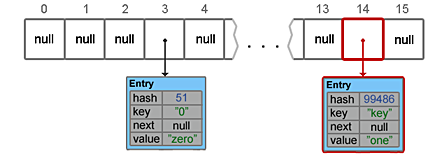
1. Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.
2. if (e.hash == hash && (e.key == key || key.equals(e.key)))
3. {
4. V oldValue = e.value;
5. e.value = value;
7. return oldValue;
8. }
9. Если же предыдущий шаг не выявил совпадений, будет вызван метод **addEntry(hash, key, value, index)** для добавления нового элемента.
10. void addEntry(int hash, K key, V value, int index)
11. {
12. Entry<K, V> e = table[index];
13. table[index] = new Entry<K, V>(hash, key, value, e);
14. ...
15. }

[[](http://habrastorage.org/storage1/517a990b/09fe6798/cf03fb7f/88d6f1dd.png)](http://habrastorage.org/storage1/517a990b/09fe6798/cf03fb7f/88d6f1dd.png)

Для того чтобы продемонстрировать, как заполняется HashMap, добавим еще несколько элементов.

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

Footprint{Objects=12, References=30, Primitives=[int x 17, char x 11, float]}  
Object size: 352 bytes

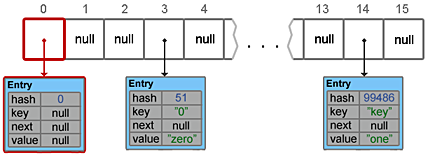
1. Пропускается, ключ не равен null
2. **''key''.hashCode()** = 106079
3. h ^ (h >>> 20) ^ (h >>> 12) = 106054
4. h ^ (h >>> 7) ^ (h >>> 4) = 99486
5. Определение позиции в массиве
6. h & (length - 1) = 14
7. Подобные элементы не найдены
8. Добавление элемента  
     
   [[](http://habrastorage.org/storage1/912edc07/a8ee5362/c52fe95f/1fea7b76.png)](http://habrastorage.org/storage1/912edc07/a8ee5362/c52fe95f/1fea7b76.png)

hashmap.put(null, null);

Footprint{Objects=13, References=33, Primitives=[int x 18, char x 11, float]}  
Object size: 376 bytes  
  
Как было сказано выше, если при добавлении элемента в качестве ключа был передан null, действия будут отличаться. Будет вызван метод **putForNullKey(value)**, внутри которого нет вызова методов **hash()** и **indexFor()** (потому как все элементы с null-ключами всегда помещаются в **table[0]**), но есть такие действия:

1. Все элементы цепочки, привязанные к **table[0]**, поочередно просматриваются в поисках элемента с ключом null. Если такой элемент в цепочке существует, его значение перезаписывается.
2. Если элемент с ключом null не был найден, будет вызван уже знакомый метод **addEntry()**.

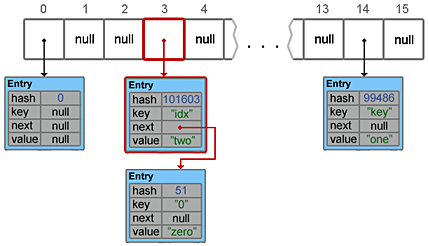
addEntry(0, null, value, 0);

[[](http://habrastorage.org/storage1/a1d8de60/ae91ecf5/b79fe023/33b07c6a.png)](http://habrastorage.org/storage1/a1d8de60/ae91ecf5/b79fe023/33b07c6a.png)

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

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes  
  
Теперь рассмотрим случай, когда при добавлении элемента возникает коллизия.

1. Пропускается, ключ не равен null
2. **''idx''.hashCode()** = 104125
3. h ^ (h >>> 20) ^ (h >>> 12) = 104100
4. h ^ (h >>> 7) ^ (h >>> 4) = 101603
5. Определение позиции в массиве
6. h & (length - 1) = 3
7. Подобные элементы не найдены
8. Добавление элемента
9. // В table[3] уже хранится цепочка состоящая из элемента ["0", "zero"]
10. Entry<K, V> e = table[index];
11. // Новый элемент добавляется в начало цепочки
12. table[index] = new Entry<K, V>(hash, key, value, e);

[[](http://habrastorage.org/storage1/9cd24c82/9dd96244/dc1dd8aa/72923c80.png)](http://habrastorage.org/storage1/9cd24c82/9dd96244/dc1dd8aa/72923c80.png)

Resize и Transfer

Когда массив **table[]** заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов. Как вы сами можете убедиться, ничего сложного в методах **resize(capacity)** и **transfer(newTable)** нет.

void resize(int newCapacity)

{

if (table.length == MAXIMUM\_CAPACITY)

{

threshold = Integer.MAX\_VALUE;

return;

}

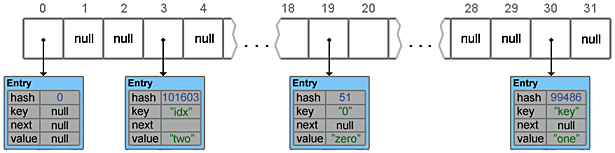
Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int)(newCapacity \* loadFactor);

}

Метод **transfer()** перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.  
  
Если в исходный **hashmap** добавить, скажем, еще 15 элементов, то в результате размер будет увеличен и распределение элементов изменится.  
  
[[](http://habrastorage.org/storage1/e96dd8c1/e5029883/e46934bc/8d1fca4b.png)](http://habrastorage.org/storage1/e96dd8c1/e5029883/e46934bc/8d1fca4b.png)

Удаление элементов

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 (хотя, как сказал один мой коллега — "*А может оно и не надо?*").  
  
Небольшой тест, для демонстрации того что написано выше. Исходный объект занимает 496 байт. Добавим, например, 150 элементов.  
  
Footprint{Objects=768, References=1028, Primitives=[int x 1075, char x 2201, float]}  
Object size: 21064 bytes  
  
Теперь удалим те же 150 элементов, и снова замерим.  
  
Footprint{Objects=18, References=278, Primitives=[int x 25, char x 17, float]}  
Object size: 1456 bytes  
  
Как видно, размер даже близко не вернулся к исходному. Если есть желание/потребность исправить ситуацию, можно, например, воспользоваться конструктором **HashMap(Map)**.

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

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes

Итераторы

HashMap имеет встроенные итераторы, такие, что вы можете получить список всех ключей **keySet()**, всех значений **values()** или же все пары ключ/значение **entrySet()**. Ниже представлены некоторые варианты для перебора элементов:

// 1.

for (Map.Entry<String, String> entry: hashmap.entrySet())

System.out.println(entry.getKey() + " = " + entry.getValue());

// 2.

for (String key: hashmap.keySet())

System.out.println(hashmap.get(key));

// 3.

Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();

while (itr.hasNext())

System.out.println(itr.next());

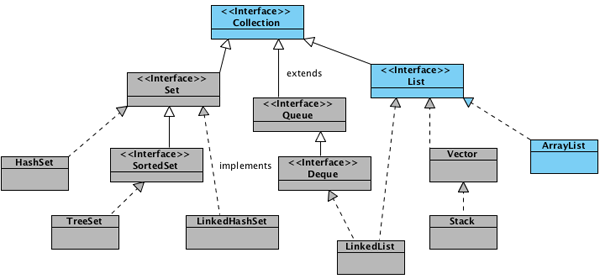
Стоит помнить, что если в ходе работы итератора HashMap был изменен (без использования собственным методов итератора), то результат перебора элементов будет непредсказуемым. 

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.

Структуры данных в картинках. ArrayList

[JAVA](http://habrahabr.ru/hub/java/)\*

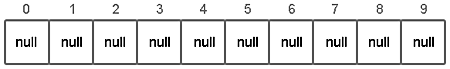
Приветствую вас, хабралюди!  
  
Взбрело мне в голову написать несколько статей, о том как реализованы некоторые структуры данных в Java. Надеюсь, статьи будут полезны визуалам (картинки наше всё), начинающим java-визуалам а также тем кто уже умеет писать new ArrayList(), но слабо представляет что же происходит внутри.  
  
  
  
Сегодня поговорим о ArrayList-ах  
  
*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.*

Создание объекта

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

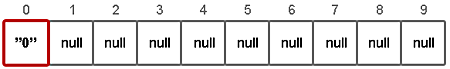
Только что созданный объект list, содержит свойства **elementData** и **size**.  
  
Хранилище значений **elementData** есть ни что иное как массив определенного типа (указанного в generic), в нашем случае **String[]**. Если вызывается конструктор без параметров, то по умолчанию будет создан массив из 10-ти элементов типа Object (с приведением к типу, разумеется).

elementData = (E[]) new Object[10];

[[](http://habrastorage.org/storage1/ddb168b8/4a0ff0d5/43889eac/c437e2fb.png)](http://habrastorage.org/storage1/ddb168b8/4a0ff0d5/43889eac/c437e2fb.png)  
Вы можете использовать конструктор **ArrayList(capacity)** и указать свою начальную емкость списка.

Добавление элементов

list.add("0");

[[](http://habrastorage.org/storage1/3720b293/cf78ff93/c9343de2/f9e8c1b7.png)](http://habrastorage.org/storage1/3720b293/cf78ff93/c9343de2/f9e8c1b7.png)  
  
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;

Весь метод **ensureCapacity(minCapacity)** рассматривать не будем, остановимся только на паре интересных мест. 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

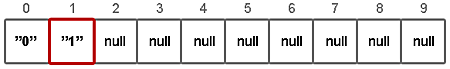
elementData = (E[])new Object[newCapacity];

// oldData - временное хранилище текущего массива с данными

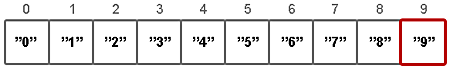
System.arraycopy(oldData, 0, elementData, 0, size);

Ниже продемонстрирован цикл, поочередно добавляющий 15 элементов:

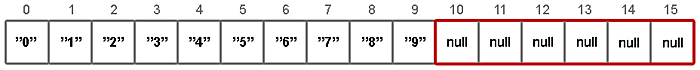
list.add("1");

[[](http://habrastorage.org/storage1/44742fc7/7d6f5cb5/d75ad62e/d5b70d56.png)](http://habrastorage.org/storage1/44742fc7/7d6f5cb5/d75ad62e/d5b70d56.png)  
**...**

list.add("9");

[[](http://habrastorage.org/storage1/6a399826/5504984d/2d6e3d4a/502b0fef.png)](http://habrastorage.org/storage1/6a399826/5504984d/2d6e3d4a/502b0fef.png)

list.add("10");

При добавлении 11-го элемента, проверка показывает что места в массиве нет. Соответственно создается новый массив и вызывается **System.arraycopy()**.  
  
[[](http://habrastorage.org/storage1/e599f79f/7a50da6d/c4ee905d/79f11337.png)](http://habrastorage.org/storage1/e599f79f/7a50da6d/c4ee905d/79f11337.png)  
После этого добавление элементов продолжается  
  
[[](http://habrastorage.org/storage1/652cfaca/aaf41ba4/4fe0b427/7170a7bc.png)](http://habrastorage.org/storage1/652cfaca/aaf41ba4/4fe0b427/7170a7bc.png)  
**...**

list.add("14");

[[](http://habrastorage.org/storage1/c848d485/82158846/8d02bb6b/e8e21fa8.png)](http://habrastorage.org/storage1/c848d485/82158846/8d02bb6b/e8e21fa8.png)

Добавление в «середину» списка

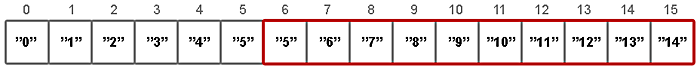
list.add(5, "100");

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);

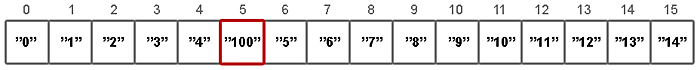
2) подготавливается место для нового элемента с помощью **System.arraycopy()**;

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

[[](http://habrastorage.org/storage1/b68ec001/369d8f50/7a73cd2c/6632cbaf.png)](http://habrastorage.org/storage1/b68ec001/369d8f50/7a73cd2c/6632cbaf.png)  
3) перезаписывается значение у элемента с указанным индексом.

elementData[index] = element;

size++;

[[](http://habrastorage.org/storage1/48afc31a/8207337a/7d7d71f2/2ff2ae09.png)](http://habrastorage.org/storage1/48afc31a/8207337a/7d7d71f2/2ff2ae09.png)  
  
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 элементов

Удалять элементы можно двумя способами:  
— по индексу **remove(index)**  
— по значению **remove(value)**  
  
С удалением элемента по индексу всё достаточно просто

list.remove(5);

Сначала определяется какое количество элементов надо скопировать

int numMoved = size - index - 1;

затем копируем элементы используя **System.arraycopy()**

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

уменьшаем размер массива и забываем про последний элемент

elementData[--size] = null; // Let gc do its work

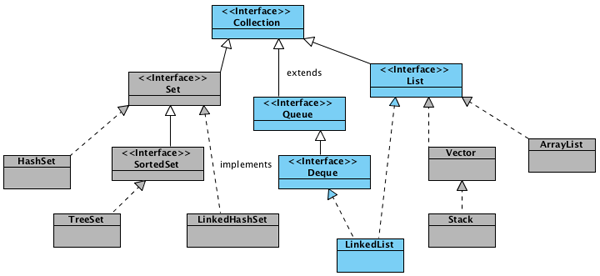
При удалении по значению, в цикле просматриваются все элементы списка, до тех пор пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.  
  
**Дополнение 1:** Как верно заметил [MikeMirzayanov](http://habrahabr.ru/users/mikemirzayanov/), при удалении элементов текущая величина capacity не уменьшается, что может привести к своеобразным утечкам памяти. Поэтому не стоит пренебрегать методом **trimToSize()**.

Итоги

— 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;  
— Не синхронизирован.

Структуры данных в картинках. LinkedList

[JAVA](http://habrahabr.ru/hub/java/)\*

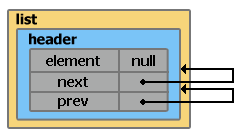
Приветствую вас, хабражители!  
  
Продолжаю начатое, а именно, пытаюсь рассказать (с применением визуальных образов) о том как реализованы некоторые структуры данных в Java.  
  
  
  
В прошлый раз мы говорили об [ArrayList](http://habrahabr.ru/blogs/java/128269/), сегодня присматриваемся к LinkedList.  
  
*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.*

Создание объекта

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

Footprint{Objects=2, References=4, Primitives=[int x 2]}  
Object size: 48 bytes  
  
Только что созданный объект list, содержит свойства **header** и **size**.  
  
**header** — псевдо-элемент списка. Его значение всегда равно **null**, a 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.

header.next = header.prev = header;

[[](http://habrastorage.org/storage1/58a98d34/957cbd74/50ccce5f/1fb95972.png)](http://habrastorage.org/storage1/58a98d34/957cbd74/50ccce5f/1fb95972.png)

Добавление элементов

list.add("0");

Footprint{Objects=5, References=8, Primitives=[int x 5, char]}  
Object size: 112 bytes  
  
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;

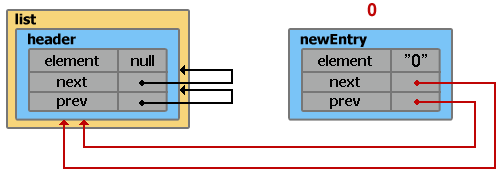
this.prev = prev;

}

}

Каждый раз при добавлении нового элемента, по сути выполняется два шага:  
  
1) создается новый новый экземпляр класса **Entry**

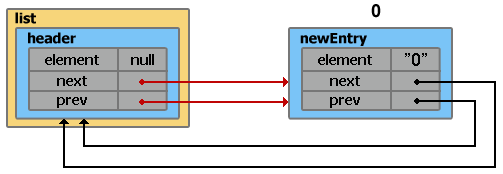
Entry newEntry = new Entry("0", header, header.prev);

[[](http://habrastorage.org/storage1/789c9810/a8612e8a/20246617/7fe32bf0.png)](http://habrastorage.org/storage1/789c9810/a8612e8a/20246617/7fe32bf0.png)  
  
2) переопределяются указатели на предыдущий и следующий элемент

newEntry.prev.next = newEntry;

newEntry.next.prev = newEntry;

size++;

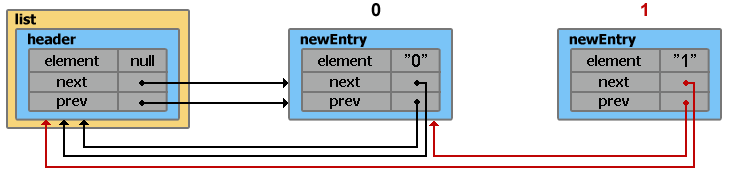
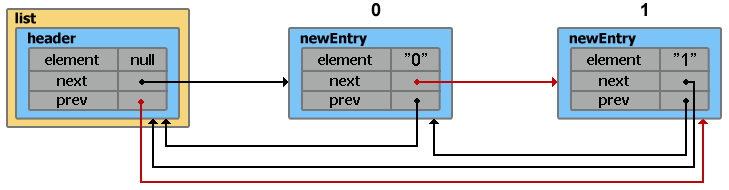
[[](http://habrastorage.org/storage1/315ee28a/4707101c/cf104547/b9c7abb4.png)](http://habrastorage.org/storage1/315ee28a/4707101c/cf104547/b9c7abb4.png)  
  
Добавим еще один элемент

list.add("1");

Footprint{Objects=8, References=12, Primitives=[int x 8, char x 2]}  
Object size: 176 bytes  
  
1)

// header.prev указывает на элемент с индексом 0

Entry newEntry = new Entry("1", header, header.prev);

[[](http://habrastorage.org/storage1/6605581f/f23b97d5/f4c7c489/843a7bbc.png)](http://habrastorage.org/storage1/6605581f/f23b97d5/f4c7c489/843a7bbc.png)  
2)  
[[](http://habrastorage.org/storage1/98df3406/6e28bf9c/d29f486d/10538c08.png)](http://habrastorage.org/storage1/98df3406/6e28bf9c/d29f486d/10538c08.png)

Добавление элементов в «середину» списка

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))

Метод **entry(index)** пробегает по всему списку в поисках элемента с указанным индексом. Направление обхода определяется условием **(index < (size >> 1))**. По факту получается что для нахождения нужного элемента перебирается не больше половины списка, но с точки зрения асимптотического анализа время на поиск растет линейно — O(n).

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;

}

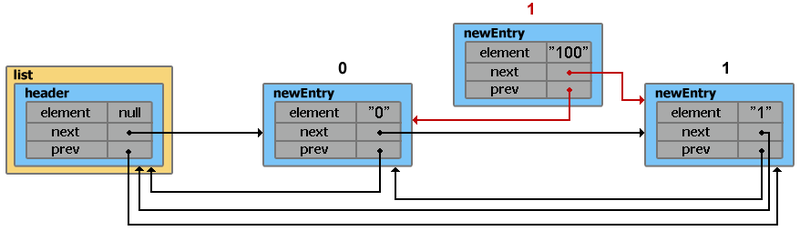
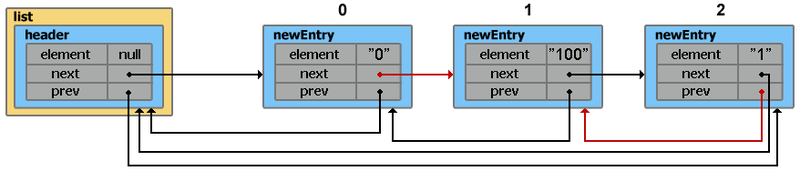
Как видно, разработчик может словить **IndexOutOfBoundsException**, если указанный индекс окажется отрицательным или большим текущего значения **size**. Это справедливо для всех методов где в параметрах фигурирует индекс.

list.add(1, "100");

Footprint{Objects=11, References=16, Primitives=[int x 11, char x 5]}  
Object size: 248 bytes  
  
1)

// entry указывает на элемент с индексом 1, entry.prev на элемент с индексом 0

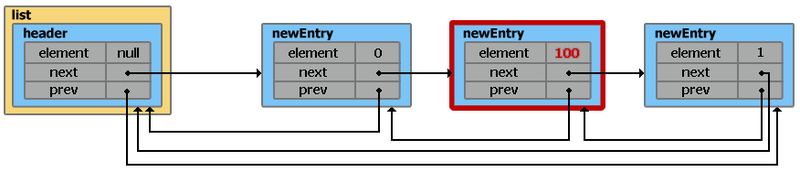
Entry newEntry = new Entry("100", entry, entry.prev);

[[](http://habrastorage.org/storage1/a483eb3d/de1e97b2/9cb9e02b/00357dba.png)](http://habrastorage.org/storage1/a483eb3d/de1e97b2/9cb9e02b/00357dba.png)  
2)  
[[](http://habrastorage.org/storage1/08d59af3/d6dff7a8/64adb35a/2e77f923.png)](http://habrastorage.org/storage1/08d59af3/d6dff7a8/64adb35a/2e77f923.png)

Удаление элементов

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).  
  
Рассмотрим удаление по значению

list.remove("100");

Footprint{Objects=8, References=12, Primitives=[int x 8, char x 2]}  
Object size: 176 bytes  
  
Внутри метода **remove(value)** просматриваются все элементы списка в поисках нужного. Удален будет лишь первый найденный элемент.  
  
В общем, удаление из списка можно условно разбить на 3 шага:  
  
1) поиск первого элемента с соответствующим значением  
  
[[](http://habrastorage.org/storage1/f7d31fab/03e93ecc/84cdb55c/208aa1d7.png)](http://habrastorage.org/storage1/f7d31fab/03e93ecc/84cdb55c/208aa1d7.png)  
2) переопределяются указатели на предыдущий и следующий элемент

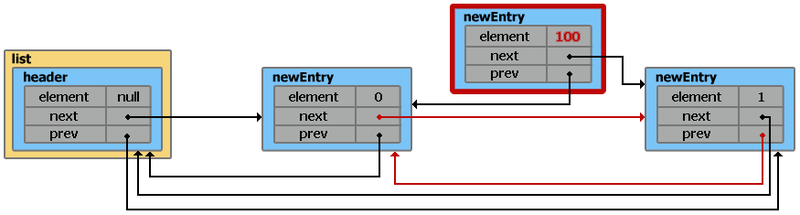
// Значение удаляемого элемента сохраняется

// для того чтобы в конце метода вернуть его

E result = e.element;

e.prev.next = e.next;

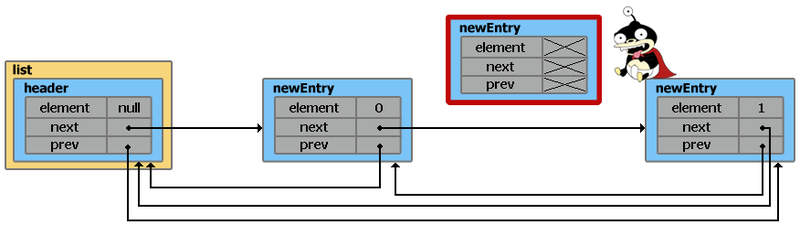
e.next.prev = e.prev;

[[](http://habrastorage.org/storage1/bacd09e2/245376c3/b8064ef7/54c54e67.png)](http://habrastorage.org/storage1/bacd09e2/245376c3/b8064ef7/54c54e67.png)  
3) удаление указателей на другие элементы и предание забвению самого элемента

e.next = e.prev = null;

e.element = null;

size--;

[[](http://habrastorage.org/storage1/3307bec1/7ccc7da0/c4547ce0/e8bfbb1f.png)](http://habrastorage.org/storage1/3307bec1/7ccc7da0/c4547ce0/e8bfbb1f.png)

Итераторы

Для собственноручного перебора элементов можно воспользоваться «встроенным» итератором. Сильно углубляться не буду, процессы протекающие внутри, очень похожи на то что описано выше.

ListIterator<String> itr = list.listIterator();

Приведенный выше код поместит указатель в начало списка. Так же можно начать перебор элементов с определенного места, для этого нужно передать индекс в метод **listIterator(index)**. В случае, если необходимо начать обход с конца списка, можно воспользоваться методом **descendingIterator()**.  
  
Стоит помнить, что **ListIterator** свалится с **ConcurrentModificationException**, если после создания итератора, список был изменен не через собственные методы итератора.  
  
Ну и на всякий случай примитивный пример перебора элементов:

while (itr.hasNext())

System.out.println(itr.next());

Итоги

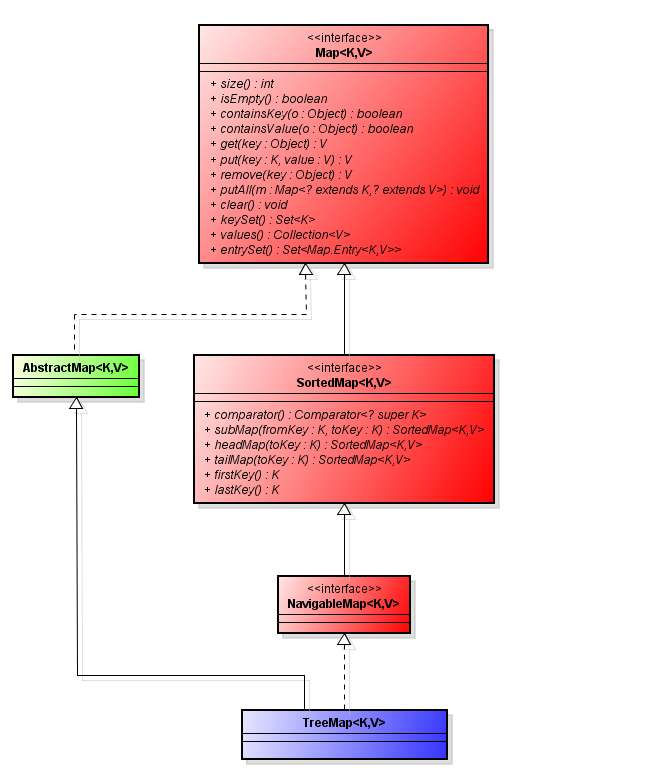
— Из LinkedList можно организовать стэк, очередь, или двойную очередь, со временем доступа O(1);  
— На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);  
— Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-оберки;  
— Не синхронизирован.

Класс TreeMap, его устройство и способ применения.

headtailСтатья

|  |  |
| --- | --- |
| категория [Java](http://www.quizful.net/category/java) | |
| дата | 13.06.2013 |
| автор | [vovanok](http://www.quizful.net/user/vovanok) |
| голосов | 25 |

Введение

В интернете можно часто встретить вопросы про то, что такое TreeMap, как он устроен, и как им пользоваться. В данной статье я постараюсь кратко ответить на эти вопросы.  
  
Класс **TreeMap**расширяет класс **AbstractMap**и реализует интерфейс **NavigatebleMap**. Он создает коллекцию, которая для хранения элементов применяет **дерево**. Объекты сохраняются в **отсортированном**порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.  
  
На рисунке 1 показана иерархия предков TreeMap.  
  
*Рисунок 1*   
  
TreeMap основан на Красно-Черном дереве, вследствие чего TreeMap сортирует элементы по ключу в естественном порядке или на основе заданного вами компаратора.TreeMap гарантирует скорость доступа**log(n)** для операций containsKey, get, put и remove.

Пример

Давайте рассмотрим простой пример использования TreeMap

Map treeMap = **new** TreeMap<>();

treeMap.put("Bruce", "Willis");

treeMap.put("Arnold", "Schwarzenegger");

treeMap.put("Jackie", "Chan");

treeMap.put("Sylvester", "Stallone");

treeMap.put("Chuck", "Norris");

**for**(Map.Entry e : treeMap.entrySet()){

    System.out.println(e.getKey()+" "+ e.getValue());

}

Вывод на консоль:  
  
Arnold Schwarzenegger  
Bruce Willis  
Chuck Norris  
Jackie Chan  
Sylvester Stallone  
  
Как видим, элементы отсортированы по ключу (Chuck Norris не первый :О).  
  
Чтобы получить ключи и значения нужно использовать методы keySet() и values().  
  
При попытке добавить null-элемент в TreeMap происходит исключение NullPointerException.

Конструкторы

В классе **TreeMap** присутствуют следующие конструкторы:  
  
1. TreeMap( )   
  
2. TreeMap(Comparator comp)   
  
3. TreeMap(Map m)   
  
4. TreeMap(SortedMap sm)  
  
**Первый конструктор** создает коллекцию, в которой все элементы отсортированы в натуральном порядке их ключей.  
  
**Второй конструктор** создаст пустую коллекцию, элементы в которой будут отсортированы по закону, который определен в передаваемом компараторе.  
  
**Третий конструктор** создаст TreeMap на основе уже имеющегося Map.  
  
**Четвертый конструктор** создаст TreeMap на основе уже имеющегося SortedMap , элементы в которой будут отсортированы по закону передаваемой SortedMap .  
  
Обратите внимание на то, что для **сортировки**используются **ключи**, а не значения. 

Синхронизация

TreeMap **не синхронизирован**. По этому, если имеется множество потоков, работающих с данной коллекцией и хотя бы один поток может её изменять , то коллекцию нужно синхронизировать внешне.  
  
Можно воспользоваться стандартным механизмом синхронизации, при котором в качестве монитора передаться ссылка объекта, инкапсулирующий данную коллекцию. Если такого объекта нет, можно воспользоваться методом[synchronizedSortedMap](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#synchronizedSortedMap(java.util.SortedMap)).  
Например:  
  
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

Создание собственного порядка сортировки

Для определения собственного порядка следования элементов в коллекции, нужно реализовать интерфейс Comparator и передать объект своего компаратора в качестве конструктора TreeMap

**public** **interface** Comparator {

**public** **int** compare (Object object1, Object object2);

**public** **boolean** equals (Object object);

**}**

Или реализовать интерфейс Comparable у класса, который будет использоваться в качестве ключа.

**public** **interface** Comparable {

**public** **int** compareTo (Object objectToCompare);

}

Классы Integer, String, Double и т.п. реализуют интерфейс Comparable. Если вы создали собственный класс для ключей и не реализовали интерфейс Comparable (и не используете Comparator), то при попытке добавления объекта в коллекцию будет брошено исключение **java.lang.ClassCastException**!  
  
*Пример:*

**public** **class** MyCustomKey **implements** Comparable{

**private** **int** value;

**public** MyCustomKey(**int** value){

**this**.value = value;

    }

**public** **int**compareTo (MyCustomKey key){

**int** comparison =0;

         // Note:

        // Return -1 if this.value < key.value

        // Return  0 if this.value = key.value

        // Return  1 if this.value > key.value

        // …. Реализация сравнения….

**return**(comparison);

     }

**public** **int**hashCode(){

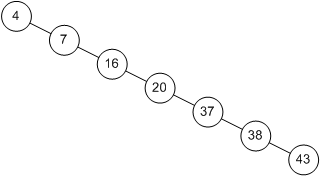
**return**(**this**.value \*199);

     }

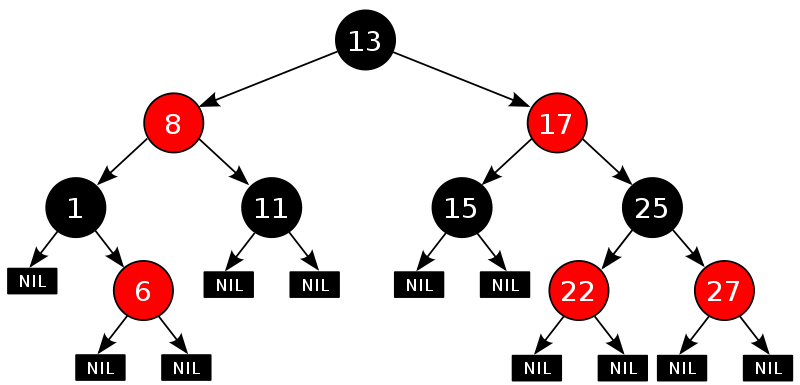
}

Частая ошибка при использовании TreeMap, в том, что для класса ключа не определяется метод hashcode(). Если в данном случае использовать метод map.get(new MyCustomKey()), то результат может быть непредсказуем. Поэтому всегда рекомендуется реализовать метод hashCode() для класса ключей.

Структура данных

Как уже было сказано, **TreeMap** для хранения элементов применяет красно-черное дерево. **Красно-черное дерево** это частный случай двоичного дерева поиска.  
  
**Двоичным деревом поиска (ДДП)** называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя. Вершины, не имеющие потомков, называются листами. Подразумевается, что каждой вершине соответствует элемент или несколько элементов, имеющие некие ключевые значения, в дальнейшем именуемые просто ключами.  
  
ДДП позволяет выполнять следующие основные операции:  
  
· Поиск вершины по ключу.  
  
· Определение вершин с минимальным и максимальным значением ключа.  
  
· Переход к предыдущей или последующей вершине, в порядке, определяемом ключами.  
  
· Вставка вершины.  
  
· Удаление вершины.  
  
У читателя, возможно, возникает вопрос, зачем нужны такие сложности, если можно просто хранить список пар [ключ, значение]. Ответ прост — операции с деревом работают быстрее. При реализации списком все функции требуют O(n) действий, где n — размер структуры. Операции с деревом же работают за O(h), где h — максимальная глубина дерева (глубина — расстояние от корня до вершины). В оптимальном случае, когда глубина всех листьев одинакова, в дереве будет n=2^h вершин. Значит, сложность операций в деревьях, близких к оптимуму будет O(log(n)).  
  
  
Рисунок 2  
  
В худшем случае дерево может принять вид, изображённый на рисунке 1. Сложность операций в таком случае будет как у списка.

Красно-черные деревья

Для того, чтобы ситуации как на рисунке 2 не было, применяют специальные сбалансированные деревья.  
  
Сбалансированные деревья:  
  
1. B-дерево  
2. АВЛ-дерево  
3. Красно-чёрное дерево  
….  
  
Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или черный. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно-чёрным деревьям применяются следующие требования:  
  
1. Узел либо красный, либо чёрный.  
2. Корень — чёрный. (В других определениях это правило иногда опускается. Это правило слабо влияет на анализ, так как корень всегда может быть изменен с красного на чёрный, но не обязательно наоборот).  
3. Все листья(NIL) — черные.  
4. Оба потомка каждого красного узла — черные.  
5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов.  
  
*Минутка философии*  
  
*Некоторые считают, что красно-черные деревья это всего лишь проекция 2-3-4 деревьев в чисто бинарные, где "настоящие узлы" 2-3-4 деревьев — это черные, а подузлы с 3-мя или 4-мя потомками это красные. Если смотреть правила балансировки 2-3-4 деревьев, то они очень простые, а если смотреть правила балансировки красно-черных деревьев, то они выглядят как мантры и заклинания, где ничего непонятно. :)*   
  
  
*Рисунок 3*  
  
Количество черных узлов на ветви от корня до листа называется черной высотой дерева. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу. Чтобы понять, почему это так, рассмотрим дерево с черной высотой 2. Кратчайшее возможное расстояние от корня до листа равно двум – когда оба узла черные. Длиннейшее расстояние от корня до листа равно четырем – узлы при этом покрашены (от корня к листу) так: красный–>черный–>красный–>черный. Сюда нельзя добавить черные узлы, поскольку при этом нарушится свойство 4, из которого вытекает корректность понятия черной высоты.  
  
Поскольку согласно свойству 4 у красных узлов непременно черные наследники, в подобной последовательности недопустимы и два красных узла подряд. Таким образом, длиннейший путь, который мы можем сконструировать, состоит из чередования красных и черных узлов, что и приводит нас к удвоенной длине пути, проходящего только через черные узлы. Все операции над деревом должны уметь работать с перечисленными свойствами. В частности, при вставке и удалении эти свойства должны сохраниться.   
  
Если Вы с первого раза не поняли, как работает красно-черное дерево (как и я), то воспользуйтесь визуализатором по ссылке [красно-черные деревья](http://rain.ifmo.ru/cat/view.php/vis/trees/red-black-2002). Там пошагово рассказывают и показывают операции поиска и вставки элементов.

Если Вам понравилась статья, проголосуйте за нее