Notes on hashtables and writing a hashcode() method

The Map interface

A Map (aka dictionary or associative array) as an association between “keys” and “values”. The “keys” have to be unique, so that given a key, you can determine the corresponding value. Examples?

(ID number, student record),
(URL, cached page).
(Editor attribute, preference)

In literal terms it’s just a set of (key, value) pairs, called “entries” in the map. Any implementation of a set can be easily turned into an implementation of Map. Logically you might think of it as a “table of values”.

In Java, the basic interface is Map. Note it doesn’t extend Collection. There are two type parameters, a type for the keys, and a type for the values. There is a nested interface Entry. It is somewhat more complex than collection, the main issue being that there are really three collections involved:

The set of entries
The set of keys
The collection of values

We’ll investigate Map using a simplified interface that captures the main ideas.

```java
public interface SimpleMap<K, V> {
    public V put(K key, V value);
    public V get(K key);
    public V remove(K key);
    public boolean containsKey(K key);
    public int size();
    public Iterator<K> keyIterator();
}
```

The main methods are put(), get(), remove(), and containsKey(). We’ll also require the ability to iterate over the keys, so there is a method keyIterator() that is *not* in the Java Map API (instead you use keySet().iterator()).

get() – returns the value for key, or null if it isn’t there
put() – adds (key, value) to the map, possibly replacing an existing value for key. Returns the old value (which is null if the key wasn’t there)
remove() – removes the key and its associated value, returns the value (which is null if the key wasn’t there)
containsKey() – returns True if the key is in the map

There are many possible implementations. The most important case is a data structure called a hashtable, which we’ll come back to in detail. The standard implementation in the Java APIs is called HashMap (avoid using the legacy class “Hashtable”). When it is important to iterate over the keys in order, a binary search tree is normally used. In the Java APIs, this style of implementation is used for TreeMap, based on a balanced BST called a Red-Black tree.

Let’s start with a really simple implementation: we can just use a list of (key, value) pairs. We can decide to disallow null keys. There is no need to re-implement a list, we can just “compose” one from the APIs and delegate the work to the underlying List.

First, we need a type for the entries.

```java
public class ListBasedMap<K, V> implements SimpleMap<K, V> {
    private List<MapEntry> list = new LinkedList<MapEntry>();

    /**
     * Each entry has a key and a value.
     */
    private class MapEntry {
        public K key;
        public V value;

        public MapEntry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

Then for a put() operation, just do a linear search. If we find the key, we update the value and return the old value. Note we’re using equals() on the key, not the Entry! If we don’t find the key, add an entry and return null.

```java
@Override
public V put(K key, V value) {
    if (key == null) throw new IllegalArgumentException();
    for (MapEntry entry : list) {
        if (entry.key.equals(key)) {
```
V ret = entry.value;
entry.value = value;
return ret;
}

list.add(new MapEntry(key, value));
return null;
}

Other operations are similar. (For the iterator, just compose the List iterator but have the next() method return the key rather than the entry.) The get/put/remove/containsKey operations are all O(n).

For a sorted map, we can use our BSTSet. Now the Entry objects have to be Comparable, so we have a small amount of additional work to do in the Entry class.

public class BSTMap<K extends Comparable<? super K>, V>
implements SimpleMap<K, V> {
    private BSTSet<MapEntry> entrySet = new BSTSet<MapEntry>();

    private class MapEntry implements Comparable<MapEntry> {
        public K key;
        public V value;

        @Override
        public int compareTo(MapEntry rhs) {
            return key.compareTo(rhs.key);
        }

        public MapEntry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}

The get/put/remove/containsKey ops are typically O(log n) but O(n) in the worst case.

The idea of a hashtable

If the keys were integers, you could implement a Map using an array of values: given key k, store the associated value in the array cell with index k. Then adding and looking up entries would be O(1), since you now have random access based on the array index.
But of course, the keys aren’t always integers. The idea of a hashtable is to try to make a lookup table for arbitrary key/value pairs that “acts” as though you had random access into an array. The ideas are as follows:

1. From each key compute an integer, called its hash code, and then use the integer as an index into an array of values.
2. Take the hash code modulo the array size to get an index into an array. (You want the array to be bounded in size - ideally it would be not much bigger than the total number of entries).
3. Store the key and the value in a linked list of entries at the array index.

See illustration below.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Hash function</th>
<th>Hashcode</th>
<th>Array index</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td></td>
<td>-348297134</td>
<td>6102</td>
<td>key</td>
</tr>
<tr>
<td>Lisa Smith</td>
<td></td>
<td>892346104</td>
<td>6103</td>
<td>value</td>
</tr>
<tr>
<td>Sam Doe</td>
<td></td>
<td>998987430</td>
<td>6104</td>
<td></td>
</tr>
<tr>
<td>Sandra Dee</td>
<td></td>
<td></td>
<td>9999</td>
<td></td>
</tr>
</tbody>
</table>

The point of item (3) is that it is possible that two different keys end up with the same hash code, or that two different hash codes result in the same array index. That means you’ll try to store two values at the same array index. This is called a collision. Using a linked list of entries allows you to store multiple values at the same array index. Traditionally these lists are called buckets.

The standard implementation in the Java libraries is called HashMap. The Set version is called HashSet. You’ll also notice that there is an implementation of the Map interface called Hashtable, which is leftover from Java 1 and generally should not be used.

**Hash Functions**

In Java, every object has a method hashcode() that is defined in Object. When you go to look up “John Smith” in a HashMap or HashSet, the library does the following:

1. calls the built-in hashcode() function to get a hash code
2. takes a modulus of the hash code to find the appropriate bucket, and
3. searches the list for an entry whose key is equal to “John Smith” in the sense of the equals() method.

4. It is important to note the use of the equals() method to identify the right entry. **It is absolutely crucial that if two keys are equal, then they must have the same hash code.**

If you override equals(), you must also override hashcode()

Of course, we always want the lists at each bucket to be very, very short (ideally length 1), since the whole point of the exercise is to get something like O(1) lookup times. Getting the keys nicely spread out into the buckets depends largely on the quality of the hash function. The idea is to try to ensure that two different keys are likely to end up with different hash codes. As a consequence, a good hash function will produce a value that incorporates **all** the data in the key. **It is highly desirable that if two keys are NOT equal, then they have different hash codes.**

A standard pattern for computing hashcodes is outlined below.

**How to write a pretty good hashcode() method**

```
let result be a nonzero value
for each instance variable v that is used in the equals() method
    let c be an integer hashcode for v  (*)
    result = result * 31 + c
return result
```

*Result* is an int variable, and the computation is freely allowed to overflow. There is nothing particularly special about the number 31 except that it is prime.

For the line (*), if v is an object, then c is just v.hashcode(). If v is a primitive value, it is converted to an int, if necessary, according to the table below. If v is an array, then the c should be the result of applying the same rules to the individual elements.
<table>
<thead>
<tr>
<th>Type of instance variable v</th>
<th>Conversion to int</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>short, char, byte</td>
<td>( c = (\text{int}) v; )</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>( c = (v ? 0 : 1); )</td>
<td>if ( v ) is true then 0 else 1</td>
</tr>
<tr>
<td>float</td>
<td>( c = \text{Float.floatToIntBits}(v); )</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>( c = (\text{int})(v ^ (v &gt;&gt;&gt; 32)); )</td>
<td>XOR of lower 32 bits with upper 32 bits</td>
</tr>
<tr>
<td>double</td>
<td>( \text{long x} = \text{Double.doubleToLongBits}(v); ) ( c = (\text{int})(x ^ (x &gt;&gt;&gt; 32)); )</td>
<td>Same as for long</td>
</tr>
</tbody>
</table>

For example, the `hashCode()` method for `String` works something like this:

```java
if (s.length() == 0) return 0;
int result = s.charAt(0);
for (int i = 1; i < s.length(); ++i)
{
    result = result * 31 + s.charAt(i);
}
return result;
```

### Rehashing and the load factor

If a hashtable is implemented using linked lists for the buckets, as illustrated here, you can add new entries to a hashtable indefinitely without expanding the array. However, the performance will deteriorate, because the bucket lists will get longer and longer. A better strategy is to expand the array and rehash the elements, that is, insert all the entries into the expanded array. How full should the table be allowed to be, before you expand the array? Normally this is controlled by a parameter called the load factor. When the ratio (number of entries) / (number of buckets) is greater than the load factor, the table is expanded. Java uses a default load factor of .75, and if exceeded the table size is expanded so that the number of buckets is roughly twice the number of entries. A higher load factor decreases the space overhead, but increases the time to look up an entry.

### Ordering and iteration

Most hashtable implementations, including `HashMap`, make no guarantees as the order of elements; in particular, it does not guarantee that the order will remain constant over time. If you frequently need to access the elements in a particular order, consider using a `TreeMap` or `TreeSet` instead.