HashMapHack

Optimizing HashMap using a “hashcode hack”

Terminology and Background

In the class library a `java.util.HashMap` is implemented as an array of linked lists. Each element of the array represents a **bucket** that contains a pointer to the head of a linked list of map **entries**.

The number of buckets is constrained to be a power of 2 to make later computations fast, and constrained to be at least 2 buckets.

The **hashcode** of an object is a 32-bit signed integer value, determined by invoking the `hashCode()` method.

Each map entry comprises a pointer to the **key**, a pointer to the **value**, the **stored hashcode** of the key, and a pointer to the next map entry in the linked list. The key may be `null`, in which case the key pointer is `null` and the stored hashcode is zero.

See Figure 1 in the HashmapHack-figures.

Adding a map entry

When adding a new map entry to a hashmap containing *numBucket* buckets, we first compute the incoming key's **hashcode**. The bucket index is determined by considering the *numBucket* least significant bits from the key's hashcode.

```
int index = hash & (numBucket – 1);
```

See Figure 2 in the HashmapHack-figures.

Then the new entry is stored at the head of the bucket, and any existing entry at that position is linked to the next pointer of the new entry. The key's hashcode is stored in the entry.

Searching for a map entry

When searching for a map entry by key, the incoming key hashcode is computed, we determine the bucket number (as above) then search the linked list for the matching key. As an optimization, we test equality of the incoming hashcode with the stored hashcode as a fast int-to-int comparison before testing the equality of the incoming key with the stored key, which is an expensive object deference, equals() message send with the incoming key.

```
while (entry != null &&
         (entry.storedKeyHash != incomingHash ||
          !incomingKey.equals(entry.key))) {
   entry = entry.next;
}
```

Finding spare bits in the stored hashcode

The set of possible values for an incoming key's hashcode occupies the full range of a 32 bit signed integer. However, within a given bucket the set of possible hashcode values that may be found in the linked list is smaller. This is because a hashmap with *numBuckets* buckets, where numBuckets is 2^n, the algorithm for adding a new map entry guarantees that all entries within a given buckets have identical bit values in the lowest *n-1* bit positions.

We can use this information to 'steal' up to *n-1* bits from the stored hashcode value to represent optimization information, knowing that we can reconstruct the actual hashcode from the stored hashcode.

Furthermore, this technique does not loose any information contained in the original hashcode.

See Figure 3 in the HashmapHack-figures.

Optimizing for a known key type

Using the spare bits in the stored hashcode we can optimize searching for some known key types. The optimization avoids dereferencing the key object pointer and invoking the equals method for each searched entry with an equal incoming and stored hashcode.

```
while (entry != null &&
         (entry.storedKeyHash != incomingHash ||
          !incomingKey.equals(entry.key))) { <<--- Avoid this test
    entry = entry.next;
}
```

Types that are suitable for optimization are those whose computed hashcode value, and equality are defined by the Java specification, and for which the equality criteria can be encoded in the hashcode. Examples include, Boolean, Byte, Character, Short, Integer, Float, etc.

The value of the *n-1* bits of a stored hashcode within a given bucket can be used to determine the type of the key, and the remaining bits of the stored hashcode are unique to the equality proposition of the key instance.

Optimizing for the Integer key type

The Integer key type is interesting because it is used extensively in the SPECjbb benchmark. If we constrain the *numBuckets* to be >= 2 then we have at least one spare bit to encode the Integer key type in the stored hashcode.

The algorithms for adding a map entry, and searching for a map entry are modified as follows.

When adding a new entry we first compute the incoming key's **hashcode**. The bucket is determined by considering the numBucket least significant bits from the key's hashcode as before.

If the incoming key is an Integer the stored hashcode is computed as `(key.hashcode() | 0x1)`, i.e. the least significant bit is set. If the incoming key is not an Integer the stored hashcode is computed as `(key.hashcode() & 0xFFFFFFFE)`, i.e. the least significant bit is cleared.

```
int index = incomingKeyHash & (numBucket – 1);
if (incomingKey instanceof Integer) {
    entry.storedKeyHash = incomingKeyHash | 0x00000001;
} else {
    entry.storedKeyHash = incomingKeyHash & 0xFFFFFFFE;
}
```

When searching for a map entry by key, the incoming key hashcode is computed, and we determine the bucket number. We then compute the stored hashcode for the incoming key based on it's type (as above) then search the linked list for the matching stored hashcode. Iff an identical stored hashcode is found then we have a key that is equal in type and value, so we can return the value directly. We do not need to dereference the key object pointer or invoke the equals method.

```
if (incomingKey instanceof Integer) {
    int incomingKeyHash = incomingKey.hashCode() | 0x00000001;
    while (entry != null && (entry.storedKeyHash != incomingKeyHash)) {
        entry = entry.next;
    }
} else {
    int incomingKeyHash = incomingKey.hashCode() & 0xFFFFFFFE;
    while (entry != null &&
             (entry.storedKeyHash != storedKeyHash ||
              !incomingKey.equals(entry.key))) {
        entry = entry.next;
    }
}
```

Rehashing the hashmap

The stored hashcodes are never returned to the caller through regular API calls. However, it is necessary to do the inverse operation when rehashing the map, since the stored hashes are only unique within the bucket, and during rehashing the entries may change bucket.

When rehashing from 2^n buckets to 2^m buckets, the stored hashcode is first restored to the original incoming hashcode value by setting the lowest n bits back to the value of the bucket index it is leaving, then computing the new stored hashcode using the lowest *m-1* bits as described above.

e.g. when using a single bit, each entry in the bucket *oldIndex* is rehashed to *newIndex* using

```
int actualHash = (oldIndex & 0x00000001) | (entry.storedKeyHash & 0xFFFFFFFE);
int newIndex = actualHash & (numBuckets - 1);
```

~~-end-~~