How HashMap actually works internally

Buckets, hashing, collisions and treeification -- and why the follow-up question catches people out.

SDET Mastery · ashishautomationlabs.store

Most candidates can say "it stores key-value pairs". Far fewer can explain what happens when two keys land in the same place -- which is exactly where the interview goes next.

The array underneath

A `HashMap` is an array of buckets. When you put a key in, three things happen:

- `hashCode()` is called on the key
- That value is spread, to mix high bits into the low bits
- The result is reduced to an array index

Lookup repeats the same arithmetic, which is why average-case `get()` is constant time -- it is arithmetic, not a search.

Why the hash gets mixed

This is the detail worth remembering:

```
// Simplified from java.util.HashMap
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

The bucket index is computed with a bitmask, not a modulo -- so only the **low** bits of the hash decide where an entry lands. Without XOR-ing the high 16 bits downward, two keys differing only in their upper bits would collide every single time.

When two keys collide

Different keys can produce the same index. When that happens, entries in that bucket form a linked list.

From Java 8 onward, once a bucket holds more than eight entries **and** the table itself is at least 64 slots, that list converts into a balanced tree. Worst-case lookup therefore degrades to O(log n)

rather than O(n) -- which matters when keys are attacker-controlled.

Why interviewers ask

- It shows whether you understand the **equals/hashCode contract** or merely memorised it
- It leads naturally into concurrency, and why `ConcurrentHashMap` exists
- It reveals whether you have ever read JDK source, which separates candidates quickly

> *A real bug, not trivia. Use a mutable object as a key, then mutate it, and its hash changes. The entry is now in the wrong bucket -- still occupying memory, never returned by* `get()`*, and invisible to* `remove()`*. This is why keys should be immutable.*

The follow-up you should expect

"What happens when the map resizes?"

When the entry count exceeds capacity × load factor (0.75 by default), the table doubles and every entry is rehashed into the new array. It is an O(n) operation. If you know roughly how many entries you need, sizing the map up front avoids repeated resizes -- a genuine, measurable win in test-data builders that populate thousands of entries.