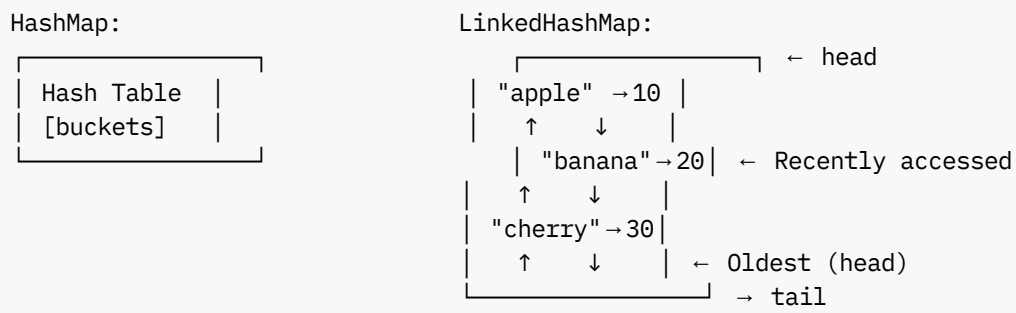
perplexity

How LinkedHashMap Maintains Insertion Order

LinkedHashMap combines the **performance of HashMap** with **insertion order guarantee** by maintaining a **doubly-linked list** that runs **through all entries**.

Internal Structure (The Magic)



Key Insight: Every `Entry` node has **3 pointers**:

```
class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before;  // Previous entry
    Entry<K,V> after;   // Next entry
    // + HashMap's hash/next pointers
}
```

How Insertion Order is Maintained

1. On `put(key, value)`:

```
Before: ["apple":10] → ["cherry":30]
After:  ["apple":10] → ["cherry":30] → ["banana":20]  ← NEW at tail
```

Code Flow:

```
1. HashMap.put() → finds/create bucket
2. Create Entry with before/after pointers
3. Link to tail: tail.after = newEntry; newEntry.before = tail
4. Update tail: tail = newEntry
```

2. Iteration Order:

```
for (Entry e : linkedHashMap.entrySet()) {
    // Follows: head → after → after → ... → tail
    System.out.println(e.key);  // Prints in insertion order!
}
```

LinkedHashMap vs HashMap: Advantages

Feature	HashMap	LinkedHashMap
Order	✕ **Unpredictable**	✓ **Insertion order** (default)
Performance	⚡ **Fastest**	⚡ **Nearly identical**
Memory	24 bytes/entry	**~32 bytes/entry** (+8 bytes for links)
Iteration	✕ **Random order**	✓ **Predictable order**

Access Order Mode (Bonus Feature)

```
// LRU Cache behavior
LinkedHashMap<String, Integer> lru = new LinkedHashMap<>(16, 0.75f, true);
lru.get("apple");  // Moves "apple" to tail (most recently used)
```

Real-World Example

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("C", 3); hashMap.put("A", 1); hashMap.put("B", 2);
// Iteration: C, A, B? B, C, A? → UNPREDICTABLE! 😱

Map<String, Integer> linked = new LinkedHashMap<>();
linked.put("C", 3); linked.put("A", 1); linked.put("B", 2);
// Iteration: ALWAYS C, A, B ✓
```

When to Use LinkedHashMap (vs HashMap)

```
✓ Use LinkedHashMap when:
• JSON serialization (order matters)
• Configuration properties (predictable output)
• Caching (LRU with accessOrder=true)
• Debugging (consistent iteration)

✕ Use HashMap when:
• Pure performance (no order needed)
• Massive datasets (save 25% memory)
```

Performance Impact: Negligible (~2-5% slower) for 99% of use cases.

Bottom Line: LinkedHashMap gives you **order guarantee** with **HashMap performance** at **minimal cost**.

⁂

1. image.jpg
2. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedHashMap.html
3. http://www.w3schools.com/JAVA/java_linkedhashmap.asp
4. https://codegym.cc/groups/posts/linkedhashmap-in-java
5. https://codingstrain.com/java-map-preserve-insertion-order/
6. https://www.scaler.com/topics/linkedhashmap-in-java/
7. image.jpg
8. image.jpg
9. image.jpg
10. https://www.tutorialspoint.com/how-to-keep-the-insertion-order-with-java-linkedhashmap
11. https://www.geeksforgeeks.org/java/how-does-linkedhashmap-maintain-insertion-order/
12. https://www.aicodesnippet.com/java/java-collections-framework/linkedhashmap-example-maintaining-insertion-order.html
13. https://www.geeksforgeeks.org/java/how-to-maintain-insertion-order-of-the-elements-in-java-hashmap/
14. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html