Overview of LinkedHashMap:

1) LinkedHashMap is a class in the Java programming language that extends HashMap.

Example:

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

2) LinkedHashMap maintains a doubly-linked list of its entries in the order they were inserted.

This ensures the elements can be accessed in insertion order.

3) LinkedHashMap shares most characteristics with HashMap but provides additional methods and constructors to maintain the ordering of elements.

For example, LinkedHashMap offers a constructor to specify the initial capacity and the order of iteration:

public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {

super(initialCapacity, loadFactor);

this.accessOrder = accessOrder;

}

- \*\*initialCapacity\*\*: The initial number of buckets. Default value: 16.

- \*\*loadFactor\*\*: The ratio of entries to buckets before resizing occurs. Default value: 0.75.

- \*\*accessOrder\*\*:

- true: Iteration follows access order (from least to most recently accessed).

- false: Iteration follows insertion order (default).

4) Load factor and initial capacity are the same as in HashMap, but LinkedHashMap incurs less penalty for a high initial capacity, as iteration time is unaffected by capacity.

5) LinkedHashMap guarantees predictable iteration order.

6) Duplicate keys are not allowed.

7) Null keys are permitted.

8) LinkedHashMap is not synchronized. If accessed by multiple threads, and at least one of them modifies it structurally, external synchronization is required.

For example:

Map m = Collections.synchronizedMap(new LinkedHashMap<>());

See the class 'LinkedHashMapWithSynchronizedMap' for more details.

9) The constant-time performance of LinkedHashMap is slightly worse than HashMap due to the overhead of maintaining the doubly-linked list. Iteration over LinkedHashMap is O(n), but it tends to perform better than HashMap for iteration, as the time complexity for LinkedHashMap depends only on the number of entries, not the capacity.

See 'LinkedHashMapAndHashMapEfficiency' for more details.

10) LinkedHashMap was introduced in Java 1.4 (released in 2002).

For further details, refer to:

<https://www.baeldung.com/java-linked-hashmap>

You would set the third argument accessOrder to true in the LinkedHashMap constructor when you need the iteration order to reflect the access order, meaning that the map will reorder entries based on the most recent access. This is useful when you want to implement features such as **cache mechanisms**, particularly for **Least Recently Used (LRU)** caching.

**Use cases:**

1. **Implementing an LRU Cache:** When using LinkedHashMap with accessOrder = true, the least recently accessed entries will be iterated first. This is helpful for implementing an LRU cache where you need to remove the least recently accessed elements when the cache reaches its maximum size.

By overriding the removeEldestEntry() method, you can automatically remove the least recently used entry when the map exceeds a certain size.

Example:  
  
LinkedHashMap<Integer, String> lruCache = new LinkedHashMap<Integer, String>(16, 0.75f, true) {

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest) {

return size() > 100; // Remove the eldest entry if the size exceeds 100

}

};

1. **Tracking Usage Frequency:** When you need to track how frequently certain elements are accessed in your system, setting accessOrder = true allows you to iterate over the most recently accessed elements first, which can be useful in analyzing or optimizing the performance based on usage patterns.
2. **Optimizing Resource Allocation:** In applications where resources (e.g., database connections, file handles) are allocated dynamically and need to be freed based on recent usage, maintaining an access order helps identify resources that are rarely accessed and should be closed or recycled.

By using accessOrder = true, you can ensure that your map reflects access patterns rather than just the order of insertion, which is key for these scenarios.

當你需要讓遍歷順序反映存取順序時，可以將 LinkedHashMap 構造函數中的第三個參數 accessOrder 設為 true，這意味著映射中的條目將根據最近的存取重新排序。這在實現像是**快取機制**，特別是\*\*最近最少使用（LRU）\*\*快取時非常有用。

### 使用情境：

1. **實現 LRU 快取：** 當使用 accessOrder = true 的 LinkedHashMap 時，最少存取的條目將優先遍歷。這對於需要在快取達到最大容量時刪除最少存取元素的 LRU 快取非常有幫助。

通過覆寫 removeEldestEntry() 方法，可以在映射超過某個大小時，自動刪除最近最少使用的條目。

範例：

LinkedHashMap<Integer, String> lruCache = new LinkedHashMap<Integer, String>(16, 0.75f, true) {

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest) {

return size() > 100; // 如果大小超過 100，刪除最舊的條目

}

};

1. **追蹤使用頻率：** 當你需要追蹤系統中某些元素的存取頻率時，將 accessOrder 設為 true 可以讓你優先遍歷最近存取的元素，這在分析或根據使用模式優化性能時非常有用。
2. **優化資源分配：** 在需要動態分配資源（如資料庫連線、檔案句柄）的應用中，根據最近的使用情況來釋放資源時，維護存取順序可以幫助識別那些較少使用的資源，進而關閉或回收它們。

通過使用 accessOrder = true，你可以確保映射反映存取模式，而不僅僅是插入順序，這在上述情境中尤為關鍵。

Here is a sample code that demonstrates the use of the LinkedHashMap with the third argument (accessOrder) set to true. This will cause the iteration order to reflect the most recently accessed entries, rather than the insertion order.

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapAccessOrderExample {

public static void main(String[] args) {

// Create LinkedHashMap with accessOrder set to true

LinkedHashMap<Integer, String> lhm = new LinkedHashMap<>(16, 0.75f, true);

// Insert some elements

lhm.put(1, "Apple");

lhm.put(2, "Banana");

lhm.put(3, "Cherry");

lhm.put(4, "Date");

// Print the original order of elements (insertion order)

System.out.println("Original order (insertion):");

for (Map.Entry<Integer, String> entry : lhm.entrySet()) {

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

}

// Access some elements

lhm.get(2); // Access Banana

lhm.get(4); // Access Date

// Print the order of elements after access (access order)

System.out.println("\nOrder after access (access order):");

for (Map.Entry<Integer, String> entry : lhm.entrySet()) {

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

}

}

}

**Explanation:**

* In this code, we create a LinkedHashMap with the third argument (accessOrder) set to true.
* The map initially holds entries in their **insertion order**.
* After accessing elements (in this case, Banana and Date), the order of iteration changes to reflect the **access order**—Banana and Date will be moved to the end of the iteration sequence because they were recently accessed.

**Output:**

makefile

Copy code

Original order (insertion):

1: Apple

2: Banana

3: Cherry

4: Date

Order after access (access order):

1: Apple

3: Cherry

2: Banana

4: Date

In this case, the elements Banana and Date move to the end of the iteration order because they were accessed.

Synchronized LinkedHashMap Example:

Here is a sample code that demonstrates how to use a synchronized LinkedHashMap with Collections.synchronizedMap:

import java.util.Collections;

import java.util.LinkedHashMap;

import java.util.Map;

public class SynchronizedMapDemo {

public static void main(String[] args) {

// Creating a synchronized LinkedHashMap

Map<String, Integer> synchronizedMap = Collections.synchronizedMap(new LinkedHashMap<>());

// Adding elements to the map

synchronizedMap.put("Apple", 3);

synchronizedMap.put("Banana", 5);

synchronizedMap.put("Orange", 2);

// Retrieving and printing elements

synchronized (synchronizedMap) { // Synchronize when iterating over the map

for (Map.Entry<String, Integer> entry : synchronizedMap.entrySet()) {

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

}

}

}

}

**Key Points:**

* The synchronizedMap is thread-safe, but when iterating over the map, you need to explicitly synchronize on the map using synchronized(synchronizedMap) block.
* In the example, a LinkedHashMap is used, which preserves the order of insertion.

In the code sample, I am showing how to create a synchronized LinkedHashMap using Collections.synchronizedMap. Here's a clearer breakdown:

* **Synchronized Map:** The synchronizedMap ensures thread safety when multiple threads access it.
* **Iteration Synchronization:** When you want to iterate over the map (i.e., read through its entries), you need to synchronize manually using synchronized (synchronizedMap) to ensure safe access during iteration. This is necessary because Collections.synchronizedMap only guarantees thread safety for individual operations, like put or get, but not for operations that traverse the map, such as iterating through it.

You can use the synchronizedMap for any application that requires safe concurrent access to a Map, such as in multi-threaded environments. Here are a few examples of practical applications where the code might be used:

**1. Caching Frequently Accessed Data**

* You can use the synchronized LinkedHashMap to maintain a cache of frequently accessed data, ensuring that multiple threads can safely read from and write to the cache without causing data corruption.

**2. Tracking User Activity in Web Applications**

* A synchronized map can store data like active users in a web application where multiple threads handle different user sessions. You can track user activity such as login counts or page visits in a thread-safe manner.

**3. Thread-Safe Access to Configuration Settings**

* In a server-side application, if multiple threads need to access and update shared configuration settings, using a synchronized map ensures that the updates don’t lead to inconsistencies.

**4. Real-Time Stock Price Tracker**

* If you're building a stock price tracker, you can use the synchronized map to store real-time prices of different stocks. Multiple threads fetching and updating stock prices won’t interfere with each other, keeping the data accurate.

**5. Logging Occurrences of Events**

* You can use the synchronized map to count the occurrences of specific events (e.g., error types, API calls) in a multithreaded application. Each thread can log events safely, updating the map without overwriting each other’s data.

**Example: Tracking Active Users in a Web Application**

import java.util.Collections;

import java.util.LinkedHashMap;

import java.util.Map;

public class UserActivityTracker {

private final Map<String, Integer> activeUsers = Collections.synchronizedMap(new LinkedHashMap<>());

// Method to add a user to the active user list

public void userLoggedIn(String username) {

activeUsers.put(username, activeUsers.getOrDefault(username, 0) + 1);

}

// Method to log user activity

public void logUserActivity() {

synchronized (activeUsers) { // Synchronize during iteration

for (Map.Entry<String, Integer> entry : activeUsers.entrySet()) {

System.out.println(entry.getKey() + ": Login Count " + entry.getValue());

}

}

}

public static void main(String[] args) {

UserActivityTracker tracker = new UserActivityTracker();

// Simulate multiple users logging in

tracker.userLoggedIn("user1");

tracker.userLoggedIn("user2");

tracker.userLoggedIn("user1");

// Log current user activity

tracker.logUserActivity();

}

}

In this example, the UserActivityTracker class uses a synchronized LinkedHashMap to track the number of times each user logs in. Multiple threads can call userLoggedIn() concurrently, and the map will be thread-safe. When iterating through the map to log activity, synchronization ensures that the process is safe.