Java Articles

Search for: Search ... Search

TUTORIALS JAVA ANDROID DESIGN PATTERNS SPRING HIBERNATE CAMEL MULE WEB DATA STRUCTURES GUAVA HAZELCAST MONGODB ACTIVEMQ QUARTZ RXJAVA OTHERS UNIT TESTING

LinkedHashMap

0

BY RAM SATISH ON JUNE 14, 2012 JAVA COLLECTIONS, JAVA MAP

In this article, we will look into the internal workings of LinkedHashMap.

LinkedHashMap vs. HashMap

`LinkedHashMap` is a `HashMap` that also defines the iteration ordering using an additional data structure, a double linked list. By default, the iteration order is same as insertion-order. It can also be the order in which its entries were last accessed so it can be easily extended to build LRU cache.

Data Structure

The data structure of `LinkedHashMap` extends that of `HashMap`.

MORE POSTS

Spring testing using @ContextConfiguration or @ContextHierarchy

Spring AOP Before Advice Annotation Example

Android Google Plus Integration – Initial Setup

Determining caller class using StackTrace Elements

Mockito Mock Object’s Default Return Values

Spring PropertyPathFactoryBean Example

Spring ApplicationEvent Example

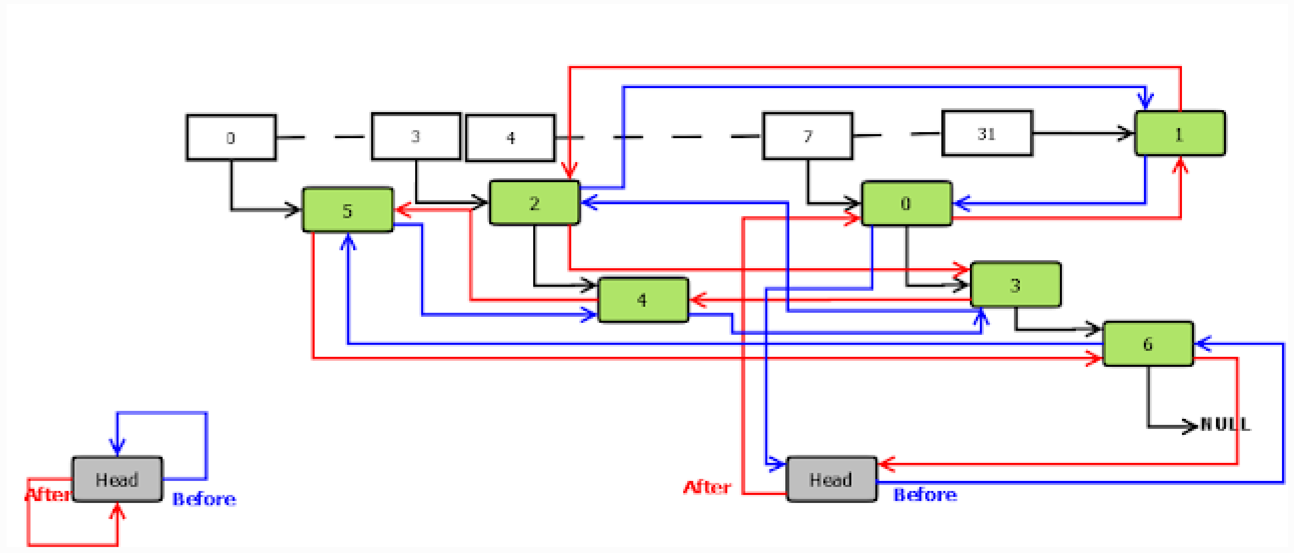
Spring AOP After Advice Annotation Example

In HashMap, the data structure is based on array and linked list. An entry finds its location in the array based on its hash value. If an array element is already occupied, the new entry replaces the old entry and the old entry is linked to the new one.

In `HashMap`, there is no control on the iteration order.

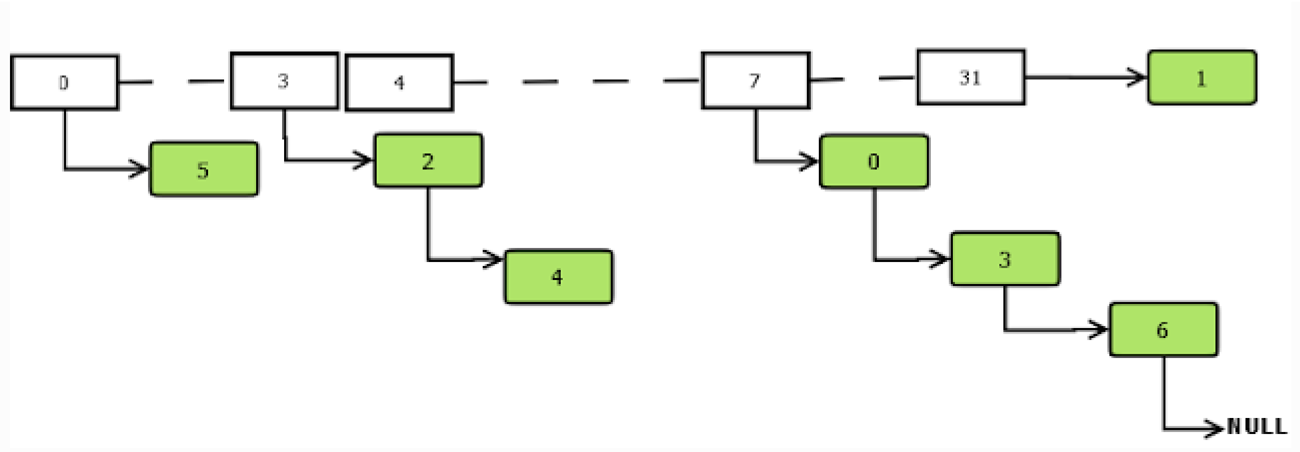
In `LinkedHashMap`, the iteration order is defined, either by the insertion order or access order.

`LinkedHashMap` differs from `HashMap` in that it maintains a doubly-linked list running through all of its entries. The below one is a modified example of the above data structure. It defines the iteration ordering based on the order in which keys were inserted into the map. In order to do so, the entry element is extended to keep track of the after and before element. A zero size `LinkedHashMap` contains just the Head element with before and after pointing to itself.



LinkedHashMap data strutcture

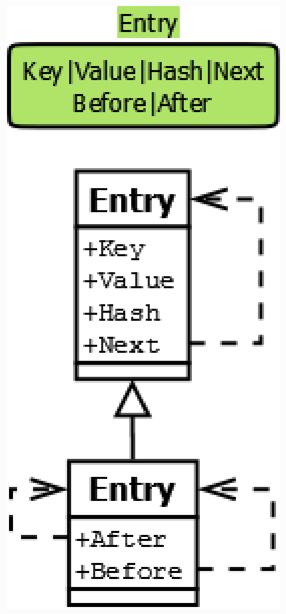
Below is the `HashMap` data structure:



HashMap data structure

Entry

`LinkedHashMap's` `Entry` extends the `HashMap's` `Entry` so it also inherits the same properties key, value, hash and the next Entry sharing the index. Other than these, it also has couple of additional properties to maintain the double-linked list, after and before entries.



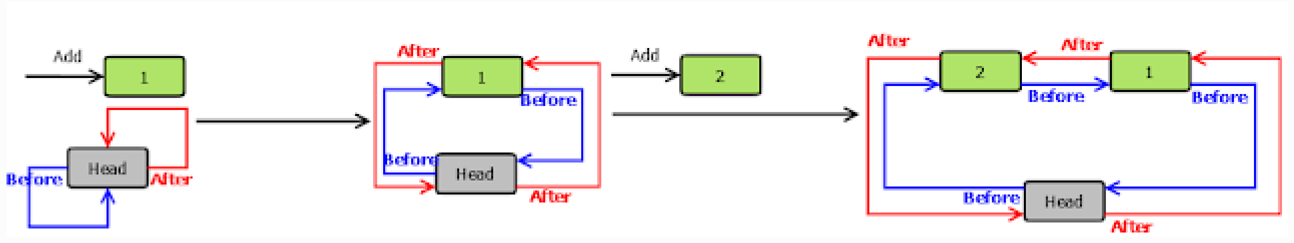
LinkedHashMap Entry class

New Entry

`LinkedHashMap` inherits `HashMap` so its internal data structure is same as that of `HashMap`. Apart from that it also maintains a double-linked list which is circularly linked via the sentinel node called head. Each node contains references to the previous and to the next node . A new node is always added to the end of the list. In order to do so, the last node's and the header node's links have to be adjusted.

1. The new node's next reference will point to the head.
2. The new node's previous reference will point to the current last node.
3. The current last node's next reference will point to the new node instead of head.
4. Head's previous reference will point to the new node.

```
after  = head;
before = head.before;
before.after = this;
after.before = this;
```



Double linked-list

Performance is likely to be just slightly below that of `HashMap`, due to the added expense of maintaining the linked list.

Access Ordered

A special `LinkedHashMap(capacity, loadFactor, accessOrderBoolean)` constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently. Invoking the put or get method results in an access to the corresponding entry. If the enclosing Map is access-ordered, it moves the entry to the end of the list; otherwise, it does nothing.

```java
public void testLinkedHashMap() {
    LinkedHashMap lru = new LinkedHashMap(16, 0.75f, true);
    lru.put("one", null);
    lru.put("two", null);
    lru.put("three", null);

    Iterator itr = lru.keySet().iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }

    System.out.println("** Access one, will move it to end **");
    lru.get("one");

    itr = lru.keySet().iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }

    System.out.println("** Access two, will move it to end **");
    lru.put("two", "two");

    itr = lru.keySet().iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

```
Result:
one
two
three
** Access one, will move it to end **
two
three
one
** Access two, will move it to end **
three
one
two
```

Thus in access-ordered linked hash maps, merely querying the map with get is a structural modification.

Iterator

In `HashMap`, the iterator has to traverse through each table element and the element's own linked list, requiring time proportional to its *capacity*.
In `LinkedHashMap`, it simply has to traverse through its own double-linked list thus requires time proportional to the *size* of the map and not its capacity so `HashMap` iteration is likely to be more expensive.

Transfer

Re-sizing is supposed to be faster as it iterates through its double-linked list to transfer the contents into a new table array.

```
void transfer(HashMap.Entry[] newTable) {
    int newCapacity = newTable.length;
    for (Entry e = header.after; e != header; e = e.after) {
        int index = indexFor(e.hash, newCapacity);
        e.next = newTable[index];
        newTable[index]= e;
    }
}
```

Contains Value

`containsValue()` is Overridden to take advantage of the faster iterator.

LRU Cache

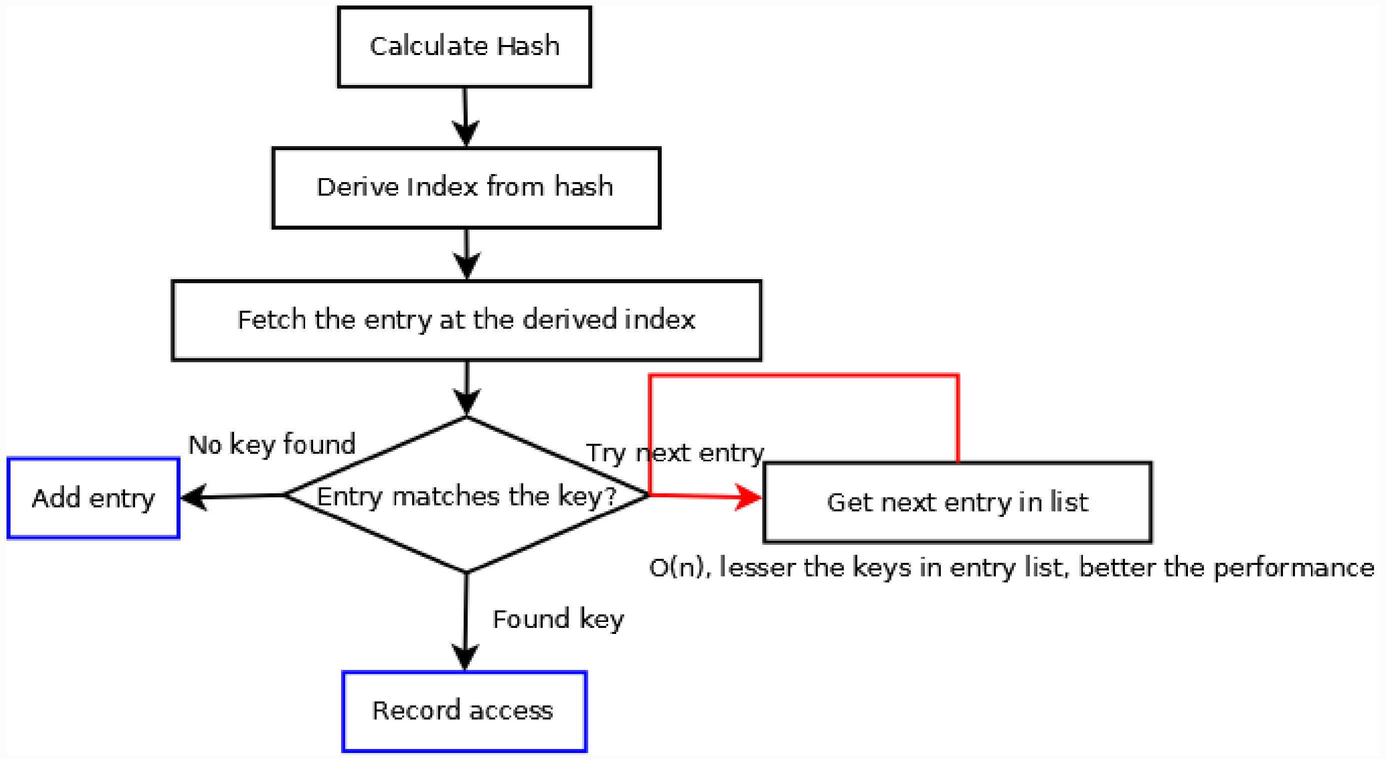
If access ordered is true, order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). This kind of map is well-suited to building LRU caches. The `removeEldestEntry(Entry)` method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map. For example,

```
protected boolean removeEldestEntry(Map.Entry eldest) {
    return size() > maxCacheSize;
}
```

The eldest entry is returned by `header.after`. The default implementation of `removeEldestEntry()` returns false.

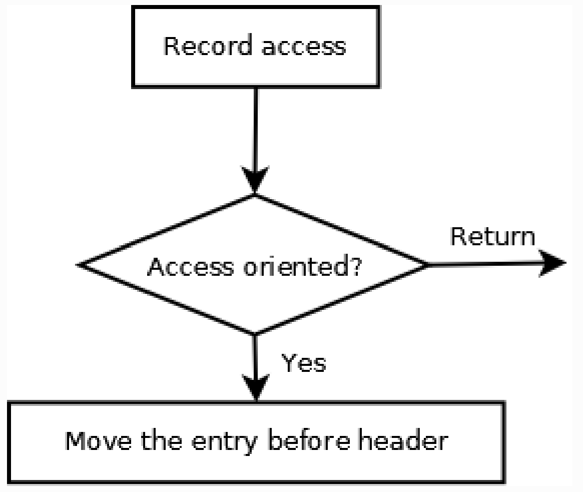
Access Order

The main reason why one prefers `LinkedHashMap` over `HashMap` is that it can retain the order in which the elements are accessed. Below is the basic flow that a `HashMap` goes through to put a new entry. The blue boxes, 'Add Entry' and 'Record Access' are the ones `LinkedHashMap` overrides.



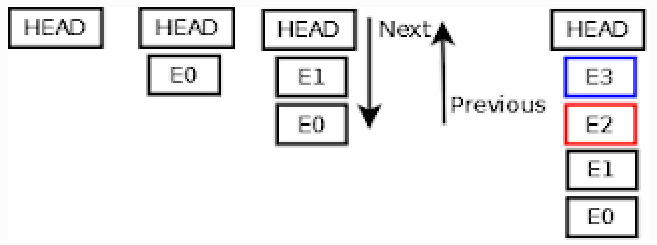
LinkedHashMap

It overrides 'Record Access' to record the access order. If the user is interested in the access order, it updates its double linked list.



LinkedHashMap

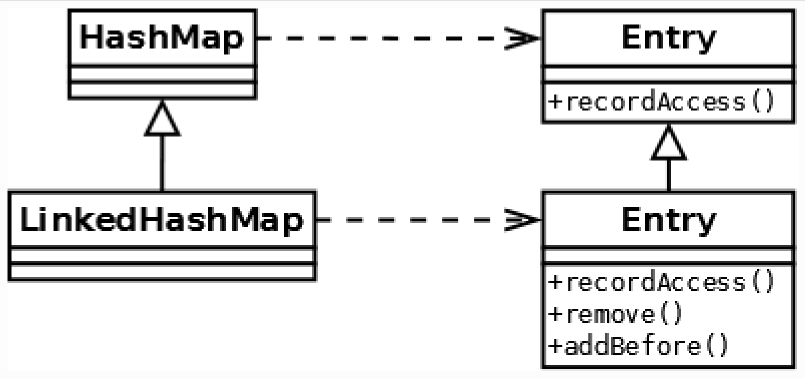
Below entry picture shows how the entry moves up the linked list. Head's next entry will point to the latest entry accessed.



LinkedHashMap

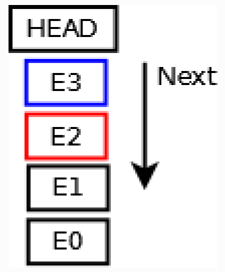
If E2 is accessed again, HashMap identifies the entry and then calls record access.

The record access is overridden in LinkedHashMap. Below is the class diagram where LinkedHashMap extends HashMap's entry to override the recordAccess.



LinkedHashMap

Double linked list vs Single Linked List



LinkedHashMap

Suppose we want to remove an entry E2.

If we have a single linked list, E3's next should point to E1 which means if want to eliminate E2, we need to know its previous entry. If it is a single linked list, we will end up traversing the entire list to search the entry previous to E2, thus the performance would in the O(n)

In case of double linked list, the previous pointer in E2 will take us to E3 in O(1).

LEAVE A REPLY

You must be logged in to post a comment.