(/)

How to Implement LRU Cache in Java

Last modified: August 3, 2021

by Arash Ariani (https://www.baeldung.com/author/arashariani/)

Algorithms (https://www.baeldung.com/category/algorithms/)

1. Overview

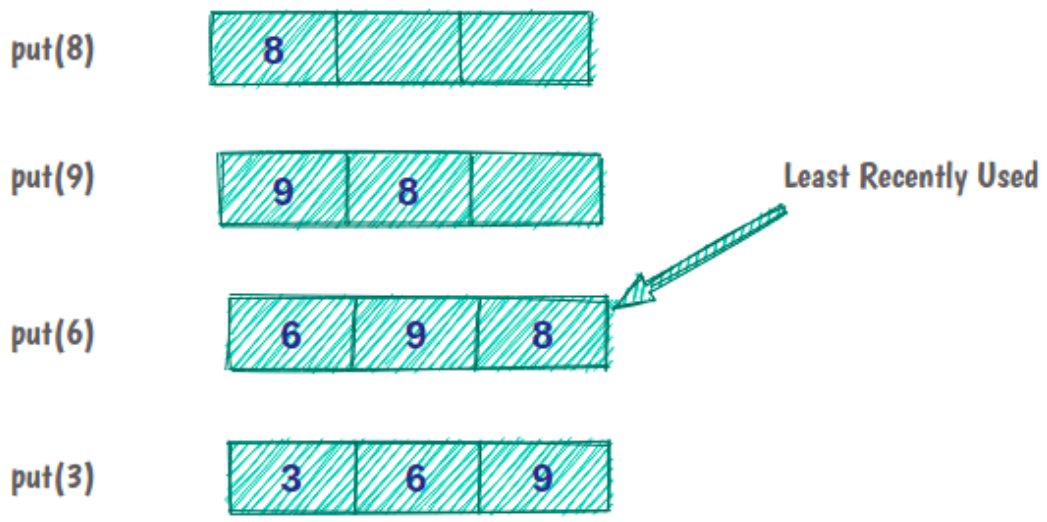
In this tutorial, we're going to learn about the LRU cache (https://en.wikipedia.org/wiki/Cache_(computing)) and take a look at an implementation in Java.

2. LRU Cache

The Least Recently Used (LRU) cache is a cache eviction algorithm that organizes elements in order of use. In LRU, as the name suggests, the element that hasn't been used for the longest time will be evicted from the

cache.

For example, if we have a cache with a capacity of three items:



(/wp-content/uploads/2021/07/Screenshot-from-2021-07-03-14-30-34-1.png)
Initially, the cache is empty, and we put element 8 in the cache. Elements 9 and 6 are cached as before. But now, the cache capacity is full, and to put the next element, we have to evict the least recently used element in the cache.

Before we implement the LRU cache in Java, it's good to know some aspects of the cache:

- All operations should run in order of O(1)
- The cache has a limited size
- It's mandatory that all cache operations support concurrency
- If the cache is full, adding a new item must invoke the LRU strategy

2.1. Structure of an LRU Cache

Now, let's think about a question that will help us in designing the cache.

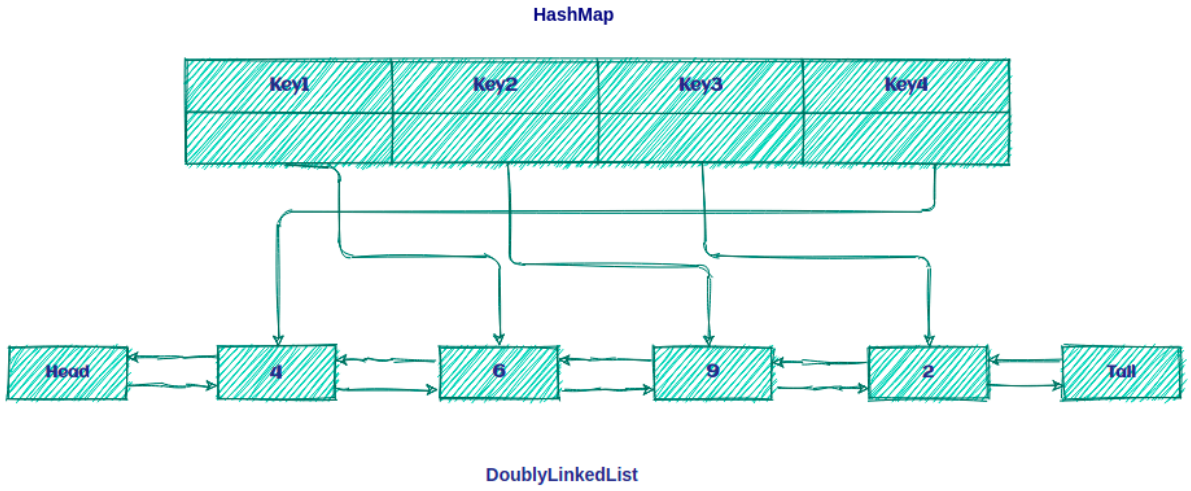
How can we design a data structure that could do operations like reading, sorting (temporal sorting), and deleting elements in constant time?

It seems that to find the answer to this question, we need to think deeply about what has been said about LRU cache and its features:

- In practice, LRU cache is a kind of ***Queue (/java-queue)*** — if an element is reaccessed, it goes to the end of the eviction order
- This queue will have a specific capacity as the cache has a limited size. Whenever a new element is brought in, it is added at the head of the

queue. When eviction happens, it happens from the tail of the queue.
- Hitting data in the cache must be done in constant time, which isn't possible in *Queue*! But, it is possible with Java's ***HashMap* (/java-hashmap)** data structure
- Removal of the least recently used element must be done in constant time, which means for the implementation of *Queue*, we'll use ***DoublyLinkedList (https://en.wikipedia.org/wiki/Doubly_linked_list)*** instead of ***SingleLinkedList (/java-linkedlist)*** or an array

So, the LRU cache is nothing but a combination of the *DoublyLinkedList* and the *HashMap* as shown below:



(/wp-content/uploads/2021/07/Screenshot-from-2021-07-09-02-10-25-1.png)
The idea is to keep the keys on the *Map* for quick access to data within the *Queue*.

2.2. LRU Algorithm

The LRU algorithm is pretty easy! If the key is present in *HashMap*, it's a cache hit; else, it's a cache miss.

We'll follow two steps after a cache miss occurs:

1. Add a new element in front of the list.
2. Add a new entry in *HashMap* and refer to the head of the list.

And, we'll do two steps after a cache hit:

1. Remove the hit element and add it in front of the list.
2. Update *HashMap* with a new reference to the front of the list.

Now, it's time to see how we can implement LRU cache in Java!

3. Implementation in Java

First, we'll define our *Cache* interface:

```
public interface Cache<K, V> {
    boolean set(K key, V value);
    Optional<V> get(K key);
    int size();
    boolean isEmpty();
    void clear();
}
```

Now, we'll define the *LRUCache* class that represents our cache:

```
public class LRUCache<K, V> implements Cache<K, V> {
    private int size;
    private Map<K, LinkedListNode<CacheElement<K,V>>> linkedListNodeMap;
    private DoublyLinkedList<CacheElement<K,V>> doublyLinkedList;

    public LRUCache(int size) {
        this.size = size;
        this.linkedListNodeMap = new HashMap<>(maxSize);
        this.doublyLinkedList = new DoublyLinkedList<>();
    }
   // rest of the implementation
}
```

We can create an instance of the *LRUCache* with a specific size. In this implementation, we use *HashMap* collection for storing all references to *LinkedListNode*.

Now, let's discuss operations on our *LRUCache*.

3.1. Put Operation

The first one is the *put* method:

```
public boolean put(K key, V value) {
    CacheElement<K, V> item = new CacheElement<K, V>(key, value);
    LinkedListNode<CacheElement<K, V>> newNode;
    if (this.linkedListNodeMap.containsKey(key)) {
        LinkedListNode<CacheElement<K, V>> node =
this.linkedListNodeMap.get(key);
        newNode = doublyLinkedList.updateAndMoveToFront(node, item);
    } else {
        if (this.size() >= this.size) {
            this.evictElement();
        }
        newNode = this.doublyLinkedList.add(item);
    }
    if(newNode.isEmpty()) {
        return false;
    }
    this.linkedListNodeMap.put(key, newNode);
    return true;
 }
```

First, we find the key in the *linkedListNodeMap* that stores all keys/references. If the key exists, a cache hit happened, and it's ready to retrieve *CacheElement* from the *DoublyLinkedList* and move it to the front.

After that, we update the *linkedListNodeMap* with a new reference and move it to the front of the list:

```
public LinkedListNode<T> updateAndMoveToFront(LinkedListNode<T> node, T
newValue) {
    if (node.isEmpty() || (this != (node.getListReference()))) {
        return dummyNode;
    }
    detach(node);
    add(newValue);
    return head;
}
```

First, we check that the node is not empty. Also, the reference of the node must be the same as the list. After that, we *detach* the node from the list and add *newValue* to the list.

But if the key doesn't exist, a cache miss happened, and we have to put a new key into the *linkedListNodeMap*. Before we can do that, we check the list size. If the list is full, we have to *evict* the least recently used element from the list.

3.2. Get Operation

Let's take a look at our *get* operation:

```
public Optional<V> get(K key) {
   LinkedListNode<CacheElement<K, V>> linkedListNode =
this.linkedListNodeMap.get(key);
   if(linkedListNode != null && !linkedListNode.isEmpty()) {
       linkedListNodeMap.put(key,
this.doublyLinkedList.moveToFront(linkedListNode));
       return Optional.of(linkedListNode.getElement().getValue());
   }
   return Optional.empty();
 }
```

As we can see above, this operation is straightforward. First, we get the node from the *linkedListNodeMap* and, after that, check that it's not null or empty.

The rest of the operation is the same as before, with just one difference on the *moveToFront* method:

```
public LinkedListNode<T> moveToFront(LinkedListNode<T> node) {
    return node.isEmpty() ? dummyNode : updateAndMoveToFront(node,
node.getElement());
}
```

Now, let's create some tests to verify that our cache works fine:

```
@Test
public void addSomeDataToCache_WhenGetData_ThenIsEqualWithCacheElement(){
    LRUCache<String,String> lruCache = new LRUCache<>(3);
    lruCache.put("1","test1");
    lruCache.put("2","test2");
    lruCache.put("3","test3");
    assertEquals("test1",lruCache.get("1").get());
    assertEquals("test2",lruCache.get("2").get());
    assertEquals("test3",lruCache.get("3").get());
}
```

Now, let's test the eviction policy:

```
@Test
public void
addDataToCacheToTheNumberOfSize_WhenAddOneMoreData_ThenLeastRecentlyDataW
illEvict(){
    LRUCache<String,String> lruCache = new LRUCache<>(3);
    lruCache.put("1","test1");
    lruCache.put("2","test2");
    lruCache.put("3","test3");
    lruCache.put("4","test4");
    assertFalse(lruCache.get("1").isPresent());
 }
```

4. Dealing With Concurrency

So far, we assumed that our cache was just used in a single-threaded environment.

To make this container thread-safe, we need to synchronize all public methods. Let's add a *ReentrantReadWriteLock* (/java-thread-safety#reentrant-locks) and *ConcurrentHashMap* (/java-concurrent-map) into the previous implementation:

```
public class LRUCache<K, V> implements Cache<K, V> {
    private int size;
    private final Map<K, LinkedListNode<CacheElement<K,V>>>
linkedListNodeMap;
    private final DoublyLinkedList<CacheElement<K,V>> doublyLinkedList;
    private final ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();

    public LRUCache(int size) {
        this.size = size;
        this.linkedListNodeMap = new ConcurrentHashMap<>(size);
        this.doublyLinkedList = new DoublyLinkedList<>();
    }
// ...
}
```

We prefer to use a reentrant read/write lock rather than declaring methods as *synchronized* (/java-thread-safety#synchronized-statements) because it gives us more flexibility in deciding when to use a lock on reading and writing.

4.1. *writeLock*

Now, let's add a call to *writeLock* in our *put* method:

```
public boolean put(K key, V value) {
  this.lock.writeLock().lock();
   try {
       //..
   } finally {
       this.lock.writeLock().unlock();
   }
}
```

When we use *writeLock* on the resource, only the thread holding the lock can write to or read from the resource. So, all other threads that are either trying to read or write on the resource will have to wait until the current lock holder releases it.

This is very important to prevent a deadlock (/cs/os-deadlock). If any of the operations inside the *try* block fails, we still release the lock before exiting the function with a *finally* block at the end of the method.

One of the other operations that needs *writeLock* is *evictElement*, which we used in the *put* method:

```
private boolean evictElement() {
    this.lock.writeLock().lock();
    try {
        //...
    } finally {
        this.lock.writeLock().unlock();
    }
}
```

4.2. *readLock*

And now it's time to add a *readLock* call to the *get* method:

```
public Optional<V> get(K key) {
    this.lock.readLock().lock();
    try {
        //...
    } finally {
        this.lock.readLock().unlock();
    }
}
```

It seems exactly what we have done with the *put* method. The only difference is that we use a *readLock* instead of *writeLock*. So, this distinction between the read and write locks allows us to read the cache in parallel while it's not being updated.

Now, let's test our cache in a concurrent environment:

```
@Test
public void
runMultiThreadTask_WhenPutDataInConcurrentToCache_ThenNoDataLost() throws
Exception {
    final int size = 50;
    final ExecutorService executorService =
Executors.newFixedThreadPool(5);
    Cache<Integer, String> cache = new LRUCache<>(size);
    CountDownLatch countDownLatch = new CountDownLatch(size);
    try {
        IntStream.range(0, size).<Runnable>mapToObj(key -> () -> {
            cache.put(key, "value" + key);
            countDownLatch.countDown();
       }).forEach(executorService::submit);
       countDownLatch.await();
    } finally {
        executorService.shutdown();
    }
    assertEquals(cache.size(), size);
    IntStream.range(0, size).forEach(i -> assertEquals("value" +
i,cache.get(i).get()));
}
```

5. Conclusion

In this tutorial, we learned what exactly an LRU cache is, including some of its most common features. Then, we saw one way to implement an LRU cache in Java and explored some of the most common operations.

Finally, we covered concurrency in action using the lock mechanism.

As usual, all the examples used in this article are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/data-structures).