Module 2 of 7

LRU cache

A HashMap of nodes plus a doubly linked list for O(1) get and put — then expiry on a logical clock, and hit statistics that a peek must not disturb.

The LRU cache is the machine-coding question asked most, because one requirement — get and put in O(1) — forces a specific design: a hash map whose values are the nodes of a doubly linked list kept in order of use.

You will code the cache itself, then extend it with expiry: entries carry an expiry time on a logical clock that a `TICK` command advances, so the behaviour is testable without a real clock. The checkpoint adds deletes, a `PEEK` that must not count as a use, and hit, miss and eviction counters. Hidden cases run the evictions that tell a real LRU from a FIFO — a get or an update moving a key to the front, a miss changing nothing — and the edge of expiry.

It is the practice half of the note Design an LRU Cache.

CONTENTS

1	LRU cache	12 min
✓	Checkpoint — LRU cache	30 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

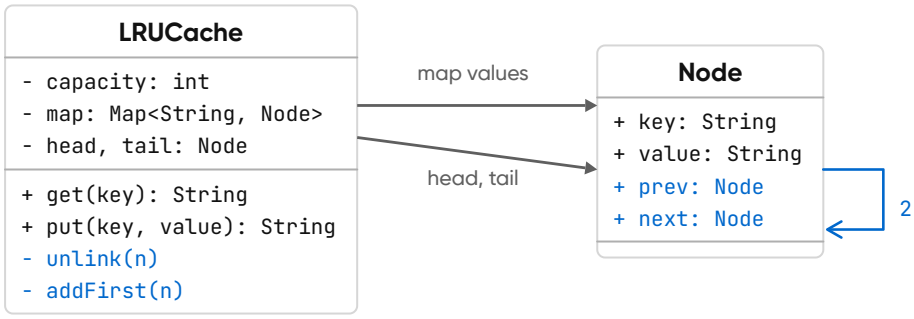
LRU cache

The most asked machine-coding problem, because one line of the brief decides the design: **get and put in O(1)**. The full treatment, with the code in four languages, is the note Design an LRU Cache; this lesson is what you need to code it in twenty minutes.

Two structures, one object

- A `HashMap<String, Node>` finds any entry in O(1) but keeps no order.
- A **doubly linked list** keeps the order of use: most recent after `head`, least recent before `tail`. Unlinking a node you already hold is O(1) because it knows its `prev`.
- The map's values *are* the list's nodes, so a lookup lands on the node to move.
- `head` and `tail` are **sentinels**: never removed, so insert and unlink need no null checks.

An LRU cache is a map of nodes plus a doubly linked list

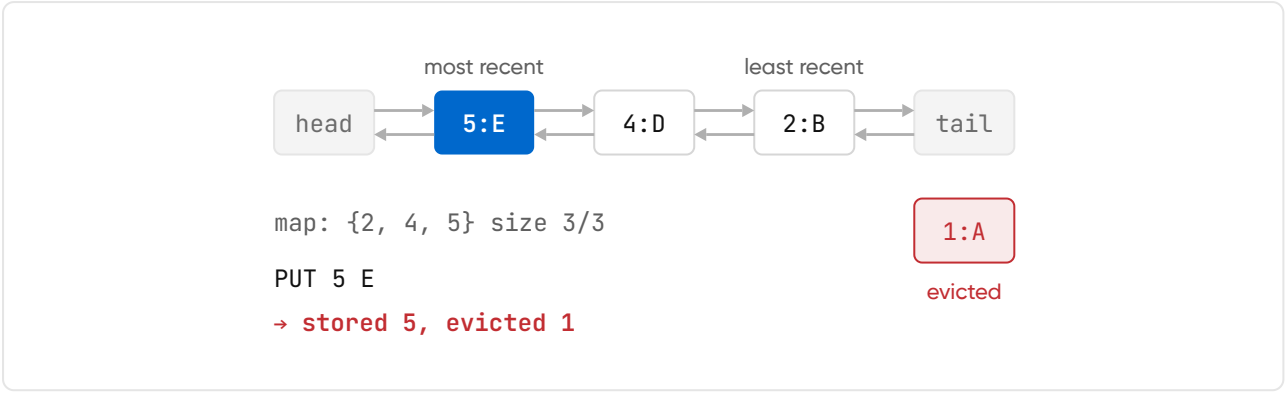


1. One class faces the caller: get and put, a fixed capacity, a HashMap from key to node, and the two sentinel nodes that bound the list.
2. The map's values are the list's own Node objects. A lookup lands on the node, and head.next is always the most recent entry, tail.prev the least.
3. Each Node points both ways. unlink rewires its two neighbours and addFirst puts it after head — six pointer writes, whatever the cache's size.

Watch the order move

Capacity 3. A hit moves its node to the front; a put on a full cache evicts from the back.

LRU cache, capacity 3: hits move to the front, the tail is evicted



1. PUT 1 into an empty cache: a node goes after head and into the map. head and tail are sentinels, never removed, so no null checks.
2. PUT 2 goes in after head too, pushing 1 one place towards the tail. Newest first, oldest last: the list is the order of use.
3. PUT 3 adds a new node after head, so the newest key is always the most recent. The cache holds 3 of 3.
4. GET 1 is a hit. The map hands over node 1, which unlinks itself from position 3 and goes after head: the order is now 1, 3, 2.
5. GET 2 is a hit. The map hands over node 2, which unlinks itself from position 3 and goes after head: the order is now 2, 1, 3.
6. PUT 4 finds the cache full, so the node before tail goes first: key 3, the least recently used, leaves the list and the map. Then 4 goes in after head.
7. GET 3 misses: key 3 was evicted earlier. No node is touched, so the order stays 4, 2, 1.
8. PUT 5 finds the cache full, so the node before tail goes first: key 1, the least recently used, leaves the list and the map. Then 5 goes in after head.

- `get` = map lookup + `unlink` + `addFirst`.
- `put` of a new key into a full cache: unlink `tail.prev`, delete *its key* from the map, then add the new node. Evict first, so the size never passes capacity.
- `put` of an existing key updates the value and counts as a use.
- A **miss changes nothing**.

Time without a clock

The second program adds **expiry**. Never call `System.currentTimeMillis()` in a design you must test: keep a logical `now` that a `TICK` command advances, store `expiresAt` on each node, and treat an entry as gone once `now ≥ expiresAt`.

- A `GET` of an expired entry removes it and says so.
- A `PUT` into a full cache drops expired entries **before** evicting a live one: an expired entry is worth nothing.

The traps

- A **singly** linked list: unlinking needs the predecessor, so every `get` walks the list.
- Forgetting the key in the node: eviction cannot find the map entry to delete.
- Printing from a `HashMap`: list the order by walking the linked list from `head`.

EXERCISE 1

LRU cache

Write an `LRUCache` with O(1) `get` and `put` using a `HashMap` from key to node and a doubly linked list with head and tail sentinels. The first line of input is the capacity (at least 1); then commands, one per line. Keys and values are single words.

Command	Prints
`PUT <key> <value>`	`stored <key>`, or `stored <key>, evicted <old>` when a new key fills a full cache, or `updated <key>` for a key already there
`GET <key>`	the value, or `miss`
`ORDER`	`order: <keys from most to least recently used>`, or `order: empty`
anything else	`error: unknown command <WORD>`

A `GET` hit and any `PUT` make the key the most recently used; a miss and `ORDER` change nothing. A full cache evicts the least recently used key.

INPUT 1

```
2
PUT a 1
PUT b 2
GET a
PUT c 3
GET b
ORDER
```

OUTPUT 1

```
stored a
stored b
1
stored c, evicted b
miss
order: c a
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class Node {
        final String key; // eviction starts from the node, so it must know its key
        String value;
        Node prev, next;

        Node(String key, String value) { this.key = key; this.value = value; }
    }

    static class LRUCache {
        private final int capacity;
        private final Map<String, Node> map = new HashMap<>();
        private final Node head = new Node(null, null); // sentinels: never removed,
        private final Node tail = new Node(null, null);

        LRUCache(int capacity) {
            this.capacity = capacity;
            head.next = tail;
            tail.prev = head;
        }

        boolean contains(String key) { return map.containsKey(key); }

        /** The value, or null on a miss. A hit makes the key the most recent. */
        String get(String key) {
            Node node = map.get(key);
            if (node == null) return null;
            unlink(node);
            addFirst(node);
            return node.value;
        }

        /** Stores the value and makes the key the most recent; answers the evicted
        String put(String key, String value) {
            Node node = map.get(key);
            if (node != null) {
                node.value = value;
                unlink(node);
                addFirst(node);
                return null;
            }
            String evicted = null;
            if (map.size() == capacity) {
                Node last = tail.prev;
                unlink(last);
                map.remove(last.key);
```

```
            evicted = last.key;
        }
        node = new Node(key, value);
        addFirst(node);
        map.put(key, node);
        return evicted;
    }

    /** Keys from most to least recent; reading the order is not a use. */
    List<String> order() {
        List<String> keys = new ArrayList<>();
        for (Node n = head.next; n != tail; n = n.next) keys.add(n.key);
        return keys;
    }

    private void unlink(Node n) {
        n.prev.next = n.next;
        n.next.prev = n.prev;
    }

    private void addFirst(Node n) {
        n.prev = head;
        n.next = head.next;
        head.next.prev = n;
        head.next = n;
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    LRUCache cache = new LRUCache(Integer.parseInt(br.readLine().trim()));
    StringBuilder out = new StringBuilder();
    String line;
    while ((line = br.readLine()) != null) {
        line = line.trim();
        if (line.isEmpty()) continue;
        String[] p = line.split("\\s+");
        switch (p[0]) {
            case "PUT": {
                boolean existed = cache.contains(p[1]);
                String evicted = cache.put(p[1], p[2]);
                out.append(existed ? "updated " + p[1] : "stored " + p[1] + (evi
                break;
            }
            case "GET": {
                String value = cache.get(p[1]);
                out.append(value == null ? "miss" : value);
                break;
            }
```

```
                case "ORDER": {
                    List<String> keys = cache.order();
                    out.append("order: ").append(keys.isEmpty() ? "empty" : String.j
                    break;
                }
                default:
                    out.append("error: unknown command ").append(p[0]);
            }
            out.append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

LRU cache with expiry

Extend the cache with time-to-live on a **logical clock**: time starts at 0 and moves only on `TICK`. An entry put at time `t` with ttl `n` expires at `t + n`, and is expired once `now ≥ t + n`. The first line is the capacity; then commands:

Command	Prints
`PUT <key> <value> <ttl>`	`updated <key>` for a live key already there (new value, new expiry, most recent); otherwise `stored <key>` or `stored <key>, evicted <old>`
`GET <key>`	the value (a hit moves it to the front); `expired` when the key is there but expired — it is removed; `miss` when it is not there
`TICK <n>`	`time <now>` after advancing the clock by `n`
`SIZE`	the number of live entries (expired ones are dropped first)
anything else	`error: unknown command <WORD>`

An expired key that is `PUT` again counts as new. When a new key meets a full cache, first drop every expired entry silently; only if the cache is still full, evict the least recently used live key.

INPUT 1

```
2
PUT a 1 5
PUT b 2 10
TICK 5
GET a
GET b
SIZE
```

OUTPUT 1

```
stored a
stored b
time 5
expired
2
1
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class Node {
        final String key;
        String value;
        long expiresAt;
        Node prev, next;

        Node(String key, String value, long expiresAt) { this.key = key; this.value
    }

    /** An LRU cache whose entries expire on a logical clock: time moves only when t
    static class ExpiringCache {
        private final int capacity;
        private final Map<String, Node> map = new HashMap<>();
        private final Node head = new Node(null, null, 0);
        private final Node tail = new Node(null, null, 0);
        private long now = 0;

        ExpiringCache(int capacity) {
            this.capacity = capacity;
            head.next = tail;
            tail.prev = head;
        }

        long tick(long n) { return now += n; }

        private boolean expired(Node n) { return now >= n.expiresAt; }

        /** Removes the key if it is there and expired; true when it did. */
        boolean dropIfExpired(String key) {
            Node n = map.get(key);
            if (n == null || !expired(n)) return false;
            remove(n);
            return true;
        }

        /** The live value, or null. Call dropIfExpired first to tell an expired key
        String get(String key) {
            dropIfExpired(key);
            Node n = map.get(key);
            if (n == null) return null;
            unlink(n);
            addFirst(n);
            return n.value;
        }
```

```
boolean containsLive(String key) {
    dropIfExpired(key);
    return map.containsKey(key);
}

/** Answers the live key evicted to make room, or null. */
String put(String key, String value, long ttl) {
    dropIfExpired(key);
    Node n = map.get(key);
    if (n != null) {
        n.value = value;
        n.expiresAt = now + ttl;
        unlink(n);
        addFirst(n);
        return null;
    }
    String evicted = null;
    if (map.size() == capacity) {
        purgeExpired(); // an expired entry is worth nothing: drop those bef
        if (map.size() == capacity) {
            Node last = tail.prev;
            remove(last);
            evicted = last.key;
        }
    }
    n = new Node(key, value, now + ttl);
    addFirst(n);
    map.put(key, n);
    return evicted;
}

int size() {
    purgeExpired();
    return map.size();
}

private void purgeExpired() {
    for (Node n = head.next; n != tail; ) {
        Node next = n.next;
        if (expired(n)) remove(n);
        n = next;
    }
}

private void remove(Node n) {
    unlink(n);
    map.remove(n.key);
}
```

```
        private void unlink(Node n) {
            n.prev.next = n.next;
            n.next.prev = n.prev;
        }

        private void addFirst(Node n) {
            n.prev = head;
            n.next = head.next;
            head.next.prev = n;
            head.next = n;
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        ExpiringCache cache = new ExpiringCache(Integer.parseInt(br.readLine().trim(
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) ≠ null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            switch (p[0]) {
                case "PUT": {
                    boolean existed = cache.containsLive(p[1]);
                    String evicted = cache.put(p[1], p[2], Long.parseLong(p[3]));
                    out.append(existed ? "updated " + p[1] : "stored " + p[1] + (evi
                    break;
                }
                case "GET": {
                    if (cache.dropIfExpired(p[1])) {
                        out.append("expired");
                    } else {
                        String value = cache.get(p[1]);
                        out.append(value == null ? "miss" : value);
                    }
                    break;
                }
                case "TICK":
                    out.append("time ").append(cache.tick(Long.parseLong(p[1])));
                    break;
                case "SIZE":
                    out.append(cache.size());
                    break;
                default:
                    out.append("error: unknown command ").append(p[0]);
            }
            out.append('\n');
        }
```

```
        System.out.print(out);
    }
}
```

QUIZ

1. Why does an O(1) LRU cache need a **doubly** linked list?

A. To iterate in both directions when printing

B. Unlinking a node found through the map needs its predecessor, which only a prev pointer gives in O(1)

C. Java has no singly linked list

D. To store two values per node

2. Capacity 2: `PUT a`, `PUT b`, `GET a`, `PUT c`, `GET b`. What does the last GET return?

A. b's value

B. miss

C. a's value

D. An error: the cache is full

3. What do the head and tail **sentinel** nodes buy you?

A. Faster hashing

B. Every real node always has a non-null prev and next, so insert and unlink need no special cases

C. They store the capacity

D. They make the cache thread-safe

4. An expiring cache reads `System.currentTimeMillis()` inside `get`. What is the main design problem?

A. It is too slow

B. Behaviour depends on real time, so tests cannot reproduce an expiry; pass in a clock (or a logical one) instead

C. Milliseconds overflow an int

D. Nothing

ANSWER KEY · LRU CACHE

1. B The map hands you the node; moving it means rewiring its neighbours. With `prev` that is O(1). A singly linked list would have to walk from the head to find the predecessor, making every hit O(n).

2. B `GET a` moved a to the front, leaving b least recent, so `PUT c` evicted b. A FIFO cache would have evicted a instead — this sequence is exactly the test that separates the two.

3. B Without sentinels, adding to an empty list and removing the first or last node each need their own null checks. Two dummy nodes that are never removed make those cases disappear.

4. B Time is an input. A cache that reads a clock you pass in — a logical one advanced by `TICK` here, `java.time.Clock` in production — can be tested at the exact boundary `now == expiresAt`.

CHECKPOINT · 30 MIN

Checkpoint — LRU cache

This checkpoint covers the module: the map-of-nodes plus doubly linked list design, sentinels, eviction order, expiry on a logical clock, and the questions an interviewer asks next (LinkedHashMap, thread safety, LFU).

How it works. Ten questions and one program. You need 70% on the questions and the program accepted to clear the module.

The program adds three things to the cache you built: `DEL`, a `PEEK` that must not count as a use, and hit, miss and eviction counters. Before you code, decide for each command:

- Does it move the node? Does it change the size?
- Which counter, if any, does it touch?

Hidden cases peek at the key that is about to be evicted, delete a key so the next put needs no eviction, and run a capacity of one.

EXERCISE

LRU cache with statistics

Build an `LRUCache` (HashMap + doubly linked list) with counters. The first line is the capacity (at least 1); then commands:

Command	Prints
`PUT <key> <value>`	`stored <key>`, `stored <key>, evicted <old>`, or `updated <key>`
`GET <key>`	the value or `miss`; a hit is a use
`PEEK <key>`	the value or `miss`, **without** making the key more recent
`DEL <key>`	`deleted <key>` or `miss`
`ORDER`	`order: <keys most to least recent>` or `order: empty`
`STATS`	`hits <h>, misses <m>, evictions <e>`
anything else	`error: unknown command <WORD>`

Only `GET` counts towards hits and misses (not `PEEK` or `DEL`); evictions count keys pushed out by a full cache, not deletes.

INPUT 1

```
2
PUT a 1
PUT b 2
PEEK a
PUT c 3
GET a
GET b
STATS
```

OUTPUT 1

```
stored a
stored b
1
stored c, evicted a
miss
2
hits 1, misses 1, evictions 1
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class Node {
        final String key; // eviction starts from the node, so it must know its key
        String value;
        Node prev, next;

        Node(String key, String value) { this.key = key; this.value = value; }
    }

    static class LRUCache {
        private final int capacity;
        private final Map<String, Node> map = new HashMap<>();
        private final Node head = new Node(null, null);
        private final Node tail = new Node(null, null);
        private int hits, misses, evictions;

        LRUCache(int capacity) {
            this.capacity = capacity;
            head.next = tail;
            tail.prev = head;
        }

        boolean contains(String key) { return map.containsKey(key); }

        String get(String key) {
            Node node = map.get(key);
            if (node == null) {
                misses++;
                return null;
            }
            hits++;
            unlink(node);
            addFirst(node);
            return node.value;
        }

        /** The value without making it more recent, and without counting as a hit o
        String peek(String key) {
            Node node = map.get(key);
            return node == null ? null : node.value;
        }

        String put(String key, String value) {
            Node node = map.get(key);
            if (node != null) {
                node.value = value;
```

```
            unlink(node);
            addFirst(node);
            return null;
        }
        String evicted = null;
        if (map.size() == capacity) {
            Node last = tail.prev;
            unlink(last);
            map.remove(last.key);
            evicted = last.key;
            evictions++;
        }
        node = new Node(key, value);
        addFirst(node);
        map.put(key, node);
        return evicted;
    }

    boolean delete(String key) {
        Node node = map.remove(key);
        if (node == null) return false;
        unlink(node);
        return true;
    }

    List<String> order() {
        List<String> keys = new ArrayList<>();
        for (Node n = head.next; n != tail; n = n.next) keys.add(n.key);
        return keys;
    }

    String stats() { return "hits " + hits + ", misses " + misses + ", evictions

    private void unlink(Node n) {
        n.prev.next = n.next;
        n.next.prev = n.prev;
    }

    private void addFirst(Node n) {
        n.prev = head;
        n.next = head.next;
        head.next.prev = n;
        head.next = n;
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    LRUCache cache = new LRUCache(Integer.parseInt(br.readLine().trim()));
```

```
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            switch (p[0]) {
                case "PUT": {
                    boolean existed = cache.contains(p[1]);
                    String evicted = cache.put(p[1], p[2]);
                    out.append(existed ? "updated " + p[1] : "stored " + p[1] + (evi
                    break;
                }
                case "GET": {
                    String value = cache.get(p[1]);
                    out.append(value == null ? "miss" : value);
                    break;
                }
                case "PEEK": {
                    String value = cache.peek(p[1]);
                    out.append(value == null ? "miss" : value);
                    break;
                }
                case "DEL":
                    out.append(cache.delete(p[1]) ? "deleted " + p[1] : "miss");
                    break;
                case "ORDER": {
                    List<String> keys = cache.order();
                    out.append("order: ").append(keys.isEmpty() ? "empty" : String.j
                    break;
                }
                case "STATS":
                    out.append(cache.stats());
                    break;
                default:
                    out.append("error: unknown command ").append(p[0]);
            }
            out.append('\n');
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. In the LRU design, what are the map's values?

A. The cached values themselves

B. The list's nodes, so a lookup lands on the node to move

C. Indexes into an array

D. Timestamps of last use

2. Why must each node store its key, not just its value?

A. For printing

B. On eviction the cache starts from tail.prev and must delete that node's map entry, so the node has to know its key

C. Java requires it for hashCode

D. To support PEEK

3. Capacity 3: `PUT 1`, `PUT 2`, `PUT 3`, `GET 1`, `GET 2`, `PUT 4`. Which key is evicted?

A. 1

B. 2

C. 3

D. 4

4. Does a `GET` that misses change the recency order?

A. Yes, it moves the tail to the front

B. No — no node is touched

C. Yes, it evicts the tail

D. Only in a full cache

5. What does `PUT` of a key already in the cache do to the order?

A. Nothing

B. Moves it to the front, as a use

C. Moves it to the back

D. Evicts the least recent key

6. Which Java class gives an LRU cache in a few lines?

A. TreeMap with a comparator

B. LinkedHashMap with accessOrder = true and removeEldestEntry overridden

C. HashSet

D. ArrayDeque

7. A cache is full and one of its entries has expired. A new key arrives. What should happen?

A. Evict the least recently used live entry

B. Drop the expired entry; no live entry needs to go

C. Refuse the new key

D. Extend the expired entry's life

8. Why should a `PEEK` (or a debug listing) not move the node?

A. It would be O(n)

B. Reading for inspection is not a use by the application; moving it would change which key is evicted next

C. Java forbids it

D. It would double-count evictions

9. What is the simplest correct way to make this LRU cache thread-safe?

A. Make the map a ConcurrentHashMap

B. One lock around get and put, because even get rewires the list

C. Make the fields volatile

D. Nothing: get only reads

10. What is the difference between LRU and LFU eviction?

A. They are the same

B. LRU evicts the entry used longest ago; LFU evicts the entry used least often

C. LFU evicts the newest entry

D. LRU needs counters, LFU does not

ANSWER KEY · CHECKPOINT — LRU CACHE

1. B Storing the node, not the value, is the whole trick: the map finds the node in O(1) and the node unlinks itself in O(1). Storing only values would leave no way to find the entry's place in the order.

2. B Eviction begins at the list, not at the map. The node before the tail is all the cache holds at that moment, and removing its map entry needs the key.

3. C After the puts the order is 3, 2, 1 (most recent first). `GET 1` gives 1, 3, 2; `GET 2` gives 2, 1, 3. The tail holds 3, so `PUT 4` evicts it.

4. B A miss finds no node, so nothing moves. Treating a miss as a use is a common bug that hidden cases catch with a miss followed by a PUT into a full cache.

5. B Writing a key is a use, so it becomes the most recent. No eviction happens because the size does not change.

6. B `new LinkedHashMap<>(16, 0.75f, true)` keeps access order, and `removeEldestEntry` returning `size() > capacity` evicts. Say so in an interview — then build the map and list yourself, which is what is being graded.

7. B An expired entry already has no value to anyone. Dropping expired entries before evicting means the cache never throws out a live key while holding a dead one.

8. B The order reflects how the application uses the cache. Monitoring or debugging reads that move nodes change eviction decisions — a classic heisenbug.

9. B A get changes the list, so it is a write. A ConcurrentHashMap alone leaves the list racing. One lock is correct; sharding by key with a lock and a list per shard trades exact global LRU for throughput.

10. B LRU keeps an order of use; LFU keeps a count of uses (and needs extra structure — frequency buckets — to stay O(1)). LRU adapts faster when the working set shifts.

CodeKairo · Low-Level Design study plan · Module 2: LRU cache. Solve the exercises online to track your progress and earn XP.