

■ LRU Cache

Combine HashMap + Doubly LinkedList for $O(1)$ get and put

● Linked List

■ MEDIUM

■ Intuition First

LRU = Least Recently Used. When cache is full and you add new item, evict the one used least recently. Think of it as a 'recently used' line: most recent at front, least recent at back. Every get/put moves that item to the front. When full: remove from back.

■ Data Structure Choice

HashMap alone can't maintain order. LinkedList alone is $O(n)$ for get. COMBINE: HashMap for $O(1)$ lookup (key→node), DoublyLinkedList for $O(1)$ insertion and removal anywhere. Together: $O(1)$ both.

■ Visual — Structure

HEAD ↔ [most recent] ↔ ... ↔ [least recent] ↔ TAIL

HashMap: key → node pointer

Doubly LinkedList: maintains access order

get(key):

1. Find node via HashMap
2. Move node to HEAD (just after dummy head)
3. Return node.val

put(key, val):

1. If exists: update value, move to HEAD
2. If not exists:
 - a. Create new node, insert at HEAD
 - b. Add to HashMap
 - c. If over capacity: remove TAIL node + from HashMap

■ Java Code

```
1 class LRUCache {
2     private Map<Integer, Node> map = new HashMap<>();
3     private int capacity;
4     private Node head = new Node(0, 0); // dummy head
5     private Node tail = new Node(0, 0); // dummy tail
6
7     public LRUCache(int capacity) {
8         this.capacity = capacity;
```

```

9     head.next = tail; tail.prev = head;
10 }
11 public int get(int key) {
12     if (!map.containsKey(key)) return -1;
13     Node node = map.get(key);
14     remove(node); addToHead(node);
15     return node.val;
16 }
17 public void put(int key, int value) {
18     if (map.containsKey(key)) remove(map.get(key));
19     Node node = new Node(key, value);
20     addToHead(node); map.put(key, node);
21     if (map.size() > capacity) {
22         map.remove(tail.prev.key);
23         remove(tail.prev);
24     }
25 }
26 private void remove(Node n) {
27     n.prev.next = n.next; n.next.prev = n.prev;
28 }
29 private void addToHead(Node n) {
30     n.next = head.next; n.prev = head;
31     head.next.prev = n; head.next = n;
32 }
33 class Node { int key, val; Node prev, next;
34     Node(int k, int v) { key=k; val=v; } }
35 }

```

■ ■ Complexity

Complexity	Value	What it means
Time	O(1)	Both get and put are O(1) — HashMap lookup + DLL pointer ops
Space	O(capacity)	Stores up to 'capacity' nodes

■ ■ Common Mistakes

■ ■ Doubly vs Singly

Must use DOUBLY linked list. Removing a node requires updating both prev.next and next.prev. Singly linked list can't do this in O(1).

■ ■ Store key in Node

When evicting from tail, you need to remove the key from HashMap. Store the key in the Node class — don't just store val.

■ Quick Revision

✓	HashMap: key $\rightarrow$ node ($O(1)$ lookup)
✓	Doubly LinkedList: maintains recency order
✓	Head side = most recent, Tail side = least recent
✓	get: HashMap lookup + move node to head
✓	put: add to head + evict tail.prev if over capacity
✓	Node must store key (for HashMap removal on eviction)
✓	Time $O(1)$ for both, Space $O(\text{capacity})$

■ Visual Thinking

Think of the cache as a VIP queue. Most important (recent) people are at the front. When you interact with anyone, they jump to the front. When the queue is full and a newcomer arrives, the person at the very back (least important) gets removed. The HashMap is the guest list — you can find anyone instantly.