Technique: Linked Hash Table

Level: Hard

Implement a data structure for a Least Recently Used (LRU) cache.

Questions to Clarify:
Q. What should we return if an entry wasn't found?
A. Return null

Solution:
Let's see how a Least Recently Used (LRU) cache works. Say we want to add an entry (memory block) A to the cache. If the cache is not full, we simply add the entry. If the cache is full, we need to make space. We remove the entry that was accessed least recently (let's call this LR). This makes space, and now we can put A into the cache.

Remember that in a cache, each entry has a fixed block size. So removing LR frees the exact amount memory to store A.

We need to store cache entries in order of when they were accessed. So if entry X was read a second ago, it should be at the front of our order. Same goes for entries that were added recently.
If we keep blocks in this order, we can easily find the least recently used (LRU) element. It should just be last on the list.

It is also important to access each block quickly. It is a cache after all. So, we need a data structure that has the O(1) access of a hash table and keeps nodes in order. Remember, in a hash table, keys are not stored in order.

A Linked Hash Table is the best data structure for this. It is a Linked List where each Node is stored in a Hash table for quick lookups.

The cache will have the following 2 standard operations:

```
1. read(key)
2. write(key, value)
```

You should check with the interviewer if they want other operations.

When we read an entry in the middle of the linked list, it becomes recently used. So we move it up to the front of the list. To move the node to front in O(1) time, we need to use a doubly linked list. This is because as we need to remove the node from the list before adding it to front. As we saw earlier, removal cannot be done in a singly linked list without the previous node.

Pseudocode:

(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

```
write(key, value)
 if full:
     Remove LRU Node
 add Node to front
read(key)
 remove node
 add node back to the front
 return node's value
```

Test Cases:
Edge Cases: Null Node, Empty Data structure
Base Cases: Single element in Linked Hash Table
Regular Cases: Read/Write, Cache Full/Empty/Not Full

Time Complexity: O(1) for both reads and writes

Space Complexity: O(n) where n is the amount of data in cache

```
public class LRUCache<K, V> {
    // Maps keys to nodes
    HashMap<K, Node<K,V>> map;

    // Linked List variables
    Node<K,V> head;
    Node<K,V> tail;

    // Maximum nodes the cache can hold.
    int capacity;

    public LRUCache(int capacity) {
        this.map = new HashMap<>();
        this.capacity = capacity;
    }

    // Read a value from cache.
    public V read(K key) {
        Node<K,V> node = map.get(key);
        if(node == null)
            return null;

        remove(key); // remove from linked hash table
        add(node.getKey(), node.getValue()); // add back to front
```

```
    return node.getValue();
}

// Write a value to cache.
public void write(K key, V value) {
    if (map.size() == capacity) { // cache is full, evict the head
        remove(head.getKey());
    }

    // In this implementation, we create a new node every time.
    // If you want, you can also move the same node to the end.
    add(key, value);
}

// Removed a node from the Linked Hash Table
private void remove(K key) {
    if (!map.containsKey(key))
        return;
    Node<K,V> toRemove = map.get(key);
    removeFromLinkedList(toRemove);
    map.remove(key);
}

// Add a node to the end of the Linked Hash Table
private void add(K key, V value) {
    Node<K,V> newNode = new Node<>(key, value);
    appendToLinkedList(newNode);
    map.put(key, newNode);
}

/*
 * These are helper functions we use above. Ask the interviewer if they want
 * you to implement these. Most interviewers will not ask you to implement
 * most of these.
 */

// Append Function, same as the technique we applied in earlier section.
// This one is for a Doubly Linked List.
public void appendToLinkedList(Node<K,V> toAdd) {
    if (head == null) {
        head = toAdd;
    } else {
        tail.setNext(toAdd);
        toAdd.setPrev(tail);
    }
    tail = toAdd;
}
```

```
    // Removes a Node from a doubly Linked List. Practice this function as well.
    public void removeFromLinkedList(Node<K,V> toRemove) {
        if (toRemove.getPrev() != null)
            toRemove.getPrev().setNext(toRemove.getNext());

        if (toRemove.getNext() != null)
            toRemove.getNext().setPrev(toRemove.getPrev());

        if (toRemove == head)
            head = toRemove.getNext();

        if (toRemove == tail)
            tail = toRemove.getPrev();
    }
}

// Node that stores Key and Value. It is a doubly linked list node,
// so it stores next and previous node. After implementing the LRUCache
// class, you can ask the interviewer if they want you to implement the Node class.
public class Node<K,V> {
    Node<K,V> next;
    Node<K,V> prev;
    K key;
    V value;

    public Node(K key, V value) {
        super();
        this.key = key;
        this.value = value;
    }

    public Node<K,V> getNext() {
        return next;
    }

    public void setNext(Node<K,V> next) {
        this.next = next;
    }

    public Node<K,V> getPrev() {
        return prev;
    }

    public void setPrev(Node<K,V> prev) {
        this.prev = prev;
    }

    public K getKey() {
        return key;
    }
```

```
    public V getValue() {
        return value;
    }
}
```