LFU Cache

Problem Statement: Design and implement a data structure for a Least Frequently Used (LFU) cache.

Implement the LFUCache class with the following functions:

LFUCache(int capacity): Initialize the object with the specified capacity.

int get(int key): Retrieve the value of the key if it exists in the cache; otherwise, return -1.
void put(int key, int value): Update the value of the key if it is present in the cache, or insert the key if it is not already present. If the cache has reached its capacity, invalidate and remove the least frequently used key before inserting a new item. In case of a tie (i.e., two or more keys with the same frequency), invalidate the least recently used key.

A use counter is maintained for each key in the cache to determine the least frequently used key. The key with the smallest use counter is considered the least frequently used.

When a key is first inserted into the cache, its use counter is set to 1 due to the put operation. The use counter for a key in the cache is incremented whenever a get or put operation is called on it. Ensure that the functions get and put run in O(1) average time complexity.

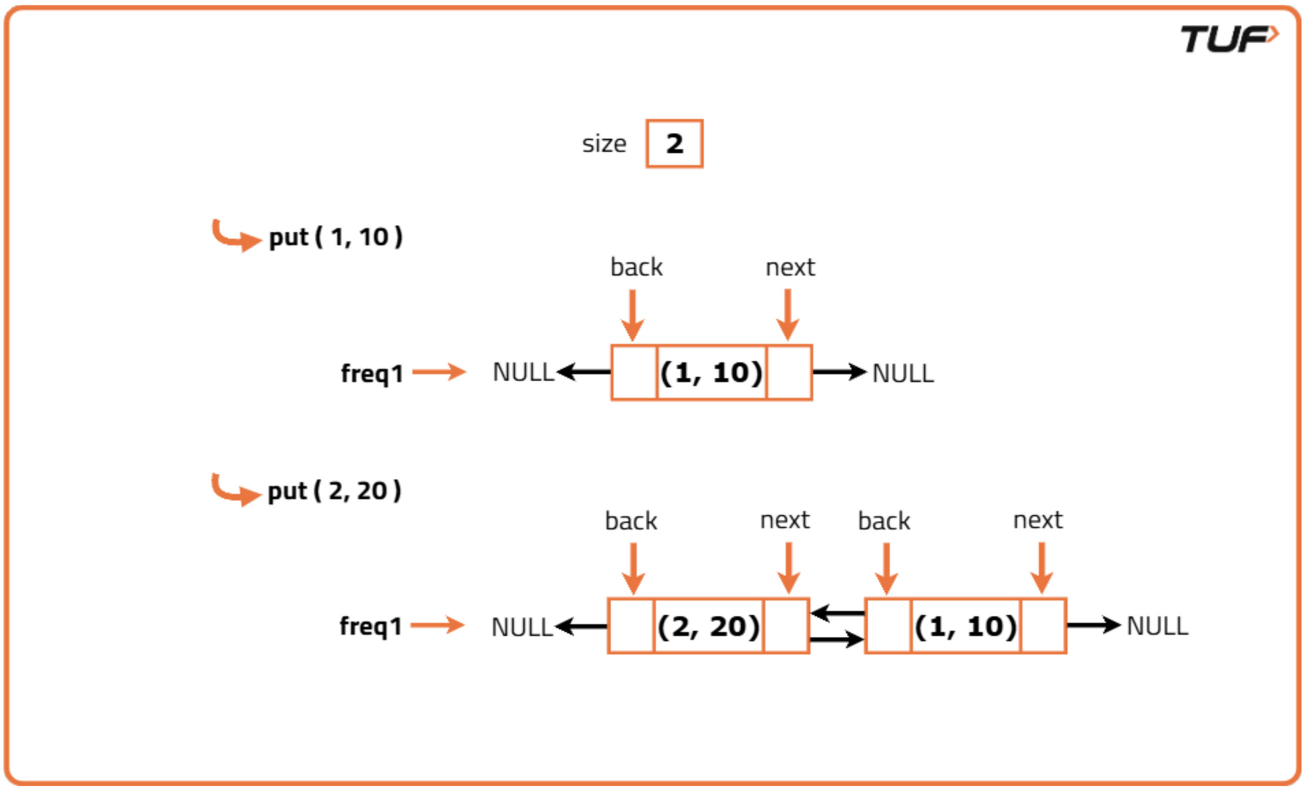
Examples

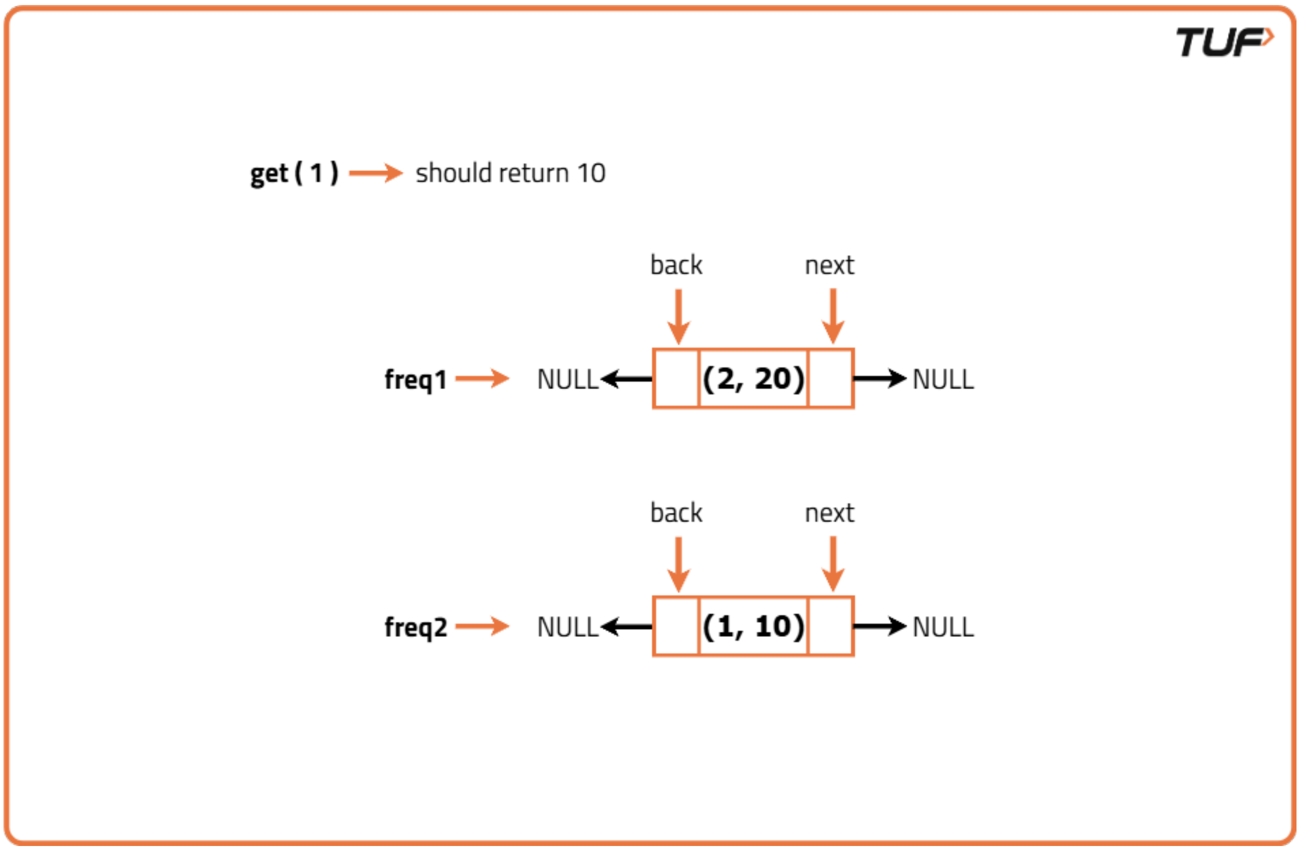
```
Example 1:
Input: ["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get",
"get", "get"]
Output: [null, null, null, 1, null, -1, 3, null, -1, 3, 4]
Explanation:
LFUCache lfu = new LFUCache(2); Initializing cache with capacity of 2.
lfu.put(1, 1); Cache becomes: [1, _], cnt(1)=1.
lfu.put(2, 2); Cache becomes: [2, 1], cnt(2)=1, cnt(1)=1.
lfu.get(1); Returns 1. Cache becomes: [1, 2], cnt(2)=1, cnt(1)=2.
lfu.put(3, 3); Cache evicts 2 as it has the lowest frequency. Cache becomes: [3,
1], cnt(3)=1, cnt(1)=2.
lfu.get(2); Returns -1 (not found).
lfu.get(3); Returns 3. Cache becomes: [3, 1], cnt(3)=2, cnt(1)=2.
lfu.put(4, 4); Cache evicts 1 (LRU). Cache becomes: [4, 3], cnt(4)=1, cnt(3)=2.
lfu.get(1); Returns -1 (not found).
lfu.get(3); Returns 3. Cache becomes: [3, 4], cnt(4)=1, cnt(3)=3.
lfu.get(4); Returns 4. Cache becomes: [4, 3], cnt(4)=2, cnt(3)=3.

Example 2:
Input: ["LFUCache", "put", "put", "put", "put", "put", "get", "get", "get",
"get", "get"]
Output: [null, null, null, null, null, null, 3, 4, 5, -1, -1]
Explanation:
LFUCache lfu = new LFUCache(3); Initializing cache with capacity of 3.
lfu.put(5, 7); Cache becomes: [5], cnt(5)=1.
lfu.put(4, 6); Cache becomes: [4, 5], cnt(4)=1, cnt(5)=1.
lfu.put(3, 5); Cache becomes: [3, 4, 5], cnt(3)=1, cnt(4)=1, cnt(5)=1.
lfu.put(2, 4); Cache evicts 5 as it has the lowest frequency. Cache becomes: [2,
3, 4], cnt(2)=1, cnt(3)=1, cnt(4)=1.
lfu.put(1, 3); Cache evicts 4 as it has the lowest frequency. Cache becomes: [1,
2, 3], cnt(1)=1, cnt(2)=1, cnt(3)=1.
lfu.get(1); Returns 3. Cache becomes: [1, 2, 3], cnt(1)=2, cnt(2)=1, cnt(3)=1.
lfu.get(2); Returns 4. Cache becomes: [2, 1, 3], cnt(1)=2, cnt(2)=2, cnt(3)=1.
lfu.get(3); Returns 5. Cache becomes: [3, 2, 1], cnt(1)=2, cnt(2)=2, cnt(3)=2.
lfu.get(4); Returns -1 (not found).
lfu.get(5); Returns -1 (not found).
```

Approach

Algorithm

- Initialize the cache with a specified capacity
- Create dummy head and tail nodes for each frequency list to simplify edge case handling
- For updating frequency, when a node is accessed or updated:
    - Remove the node from its current frequency list
    - If this was the only node in the minimum frequency list, increment the minimum frequency
    - Increment the node's frequency and add it to the front of the list corresponding to the new frequency
    - Update both hash maps (keyNode and freqNode) to reflect the changes
- For the "get" operation, check if the key exists in the keyNode map:
    - If present, retrieve the node, update its frequency, and return its value
    - If not present, return -1
- For the "put" operation:
    - If the key already exists in keyNode, update its value, update its frequency, and return
    - If not present, check if the cache is at capacity
    - If at capacity, remove the least frequently used node (node with the minimum frequency and least recent usage)
    - Create a new node for the key-value pair, set its frequency to 1, and add it to the front of the frequency list for nodes with frequency 1
    - Update both hash maps to reflect the addition of the new node





TUF
get (1) → should return 10
back
next
freq1 → NULL ← (2, 20) → NULL
back
next
freq2 → NULL ← (1, 10) → NULL

Code

```
import java.util.HashMap;
import java.util.Map;

/* To implement a node in doubly linked list that will store data items */
class Node {
    int key, value, cnt;
    Node next;
    Node prev;

    Node(int _key, int _value) {
        key = _key;
        value = _value;
        cnt = 1;
    }
}

// To implement the doubly linked list
class List {
    int size; // Size
    Node head; // Dummy head
    Node tail; // Dummy tail

    // Constructor
    List() {
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
        size = 0;
    }

    // Function to add node in front
    void addFront(Node node) {
        Node temp = head.next;
        node.next = temp;
        node.prev = head;
        head.next = node;
        temp.prev = node;
        size++;
    }

    // Function to remove node from the list
    void removeNode(Node delnode) {
        Node prevNode = delnode.prev;
        Node nextNode = delnode.next;
        prevNode.next = nextNode;
        nextNode.prev = prevNode;
        size--;
    }
}

// Class to implement LFU cache
class LFUCache {
    private Map<Integer, Node> keyNode; // Hashmap to store the key-nodes pairs
    private Map<Integer, List> freqListMap; // Hashmap to maintain the lists
with different frequencies
    private int maxSizeCache; // Max size of cache
    private int minFreq; // To store the frequency of least frequently used
data-item
    private int curSize; // To store current size of cache

    // Constructor
    public LFUCache(int capacity) {
        maxSizeCache = capacity;
        minFreq = 0;
        curSize = 0;
        keyNode = new HashMap<>();
        freqListMap = new HashMap<>();
    }

    // Method to update frequency of data-items
    private void updateFreqListMap(Node node) {
        // Remove from Hashmap
        keyNode.remove(node.key);

        // Update the frequency list hashmap
        freqListMap.get(node.cnt).removeNode(node);

        // If node was the last node having its frequency
        if (node.cnt == minFreq && freqListMap.get(node.cnt).size == 0) {
            // Update the minimum frequency
            minFreq++;
        }

        // Creating a dummy list for next higher frequency
        List nextHigherFreqList = new List();

        // If the next higher frequency list already exists
        if (freqListMap.containsKey(node.cnt + 1)) {
            // Update pointer to already existing list
            nextHigherFreqList = freqListMap.get(node.cnt + 1);
        }

        // Increment the count of data-item
        node.cnt += 1;

        // Add the node in front of higher frequency list
        nextHigherFreqList.addFront(node);
        // Update the frequency list map
        freqListMap.put(node.cnt, nextHigherFreqList);
        keyNode.put(node.key, node);
    }

    // Method to get the value of key from LFU cache
    public int get(int key) {
        // Return the value if key exists
        if (keyNode.containsKey(key)) {
            Node node = keyNode.get(key); // Get the node
            int val = node.value; // Get the value
            updateFreqListMap(node); // Update the frequency
            // Return the value
            return val;
        }
        // Return -1 if key is not found
        return -1;
    }

    // Method to insert key-value pair in LFU cache
    public void put(int key, int value) {
        /* If the size of Cache is 0,
        no data-items can be inserted */
        if (maxSizeCache == 0) {
            return;
        }

        // If key already exists
        if (keyNode.containsKey(key)) {
            // Get the node
            Node node = keyNode.get(key);
            // Update the value
            node.value = value;
            // Update the frequency
            updateFreqListMap(node);
        } else {
            // If cache limit is reached
            if (curSize == maxSizeCache) {
                // Remove the least frequently used data-item
                List list = freqListMap.get(minFreq);
                keyNode.remove(list.tail.prev.key);

                // Update the frequency map
                freqListMap.get(minFreq).removeNode(list.tail.prev);
                // Decrement the current size of cache
                curSize--;
            }
            // Increment the current cache size
            curSize++;

            // Adding new value to the cache
            minFreq = 1; // Set its frequency to 1

            // Create a dummy list
            List listFreq = new List();

            // If the list already exists
            if (freqListMap.containsKey(minFreq)) {
                // Update the pointer to already present list
                listFreq = freqListMap.get(minFreq);
            }

            // Create the node to store data-item
            Node node = new Node(key, value);

            // Add the node to dummy list
            listFreq.addFront(node);

            // Add the node to Hashmap
            keyNode.put(key, node);

            // Update the frequency list map
            freqListMap.put(minFreq, listFreq);
        }
    }

}

// Main class containing the main method
public class Main {
    public static void main(String[] args) {
        // LFU Cache
        LFUCache cache = new LFUCache(2);

        // Queries
        cache.put(1, 1);
        cache.put(2, 2);
        System.out.print(cache.get(1) + " ");
        cache.put(3, 3);
        System.out.print(cache.get(2) + " ");
        System.out.print(cache.get(3) + " ");
        cache.put(4, 4);
        System.out.print(cache.get(1) + " ");
        System.out.print(cache.get(3) + " ");
        System.out.print(cache.get(4) + " ");
    }
}
```

Complexity Analysis

Time Complexity: O(N), where N is the number of queries on the LFU cache. Each get and put method takes an average of constant time, making the overall complexity O(N).

Space Complexity: O(cap), where cap is the capacity of the LFU cache. The cache can store a maximum of cap data items, taking O(cap) space.