Least Recently Used (LRU) Cache LeetCode

Design a data structure that follows the constraints of a **Least Recently Used (LRU)** cache.

Least recently used (LRU)

Discards the least recently used items first. **This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item**.

Implement the LRUCache class:

- LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
- int get(int key) Return the value of the key if the key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

Approach 1: Implementation of LRUCache using Queue and Hashmap approach

In this approach, we use a combination of a queue (to maintain the order of recently used keys) and a hashmap (to store key-value pairs). The queue helps us keep track of the least recently used item, and the hashmap allows for quick retrieval of values associated with keys.

1. **LRUCache(int capacity)**
    - Constructor for initializing the LRUCache with a specified capacity.
    - Initializes the capacity, creates a dummy head and tail for the doubly linked list, and initializes an empty hashmap.
2. **get(int key)**
    - Retrieves the value associated with the given key from the cache.
    - If the key is found in the hashmap:
        - Reorganizes the queue to move the accessed key to the front (indicating it's the most recently used).
        - Returns the value associated with the key.
    - If the key is not found, returns -1 to indicate a cache miss.
        - **Time Complexity: O(n)**

        In the worst case, where 'n' is the capacity of the cache, when the key is not found in the cache and you need to reorganize the queue, it takes O(n) time to search for the key in the queue.

- **Space Complexity: O(n)**

  The space complexity is O(n) because you are using a hashmap to store key-value pairs, and a queue to maintain the order of keys.

3. **put(int key, int value)**
   - Inserts or updates a key-value pair in the cache.
   - If the key is found in the hashmap (indicating an update):
     - Reorganizes the queue to move the modified key to the front (indicating it's the most recently used).
   - If the key is not found (indicating an insert):
     - Adds the new key to the front of the queue.
   - If the cache exceeds its capacity:
     - Removes the least recently used key from the queue and the hashmap.
   - Updates the value associated with the key in the hashmap.
   - **Time Complexity: O(n)**

     Similar to get, in the worst case, it takes O(n) time to search for the key in the queue when you need to reorganize it.
   - **Space Complexity: O(n)**

     The space complexity is O(n) because you are using a hashmap to store key-value pairs, and a queue to maintain the order of keys.

Approach 2: Implementation of LRUCache using Doubly Linked List and Hashmap approach

In this approach, we use a doubly linked list to maintain the order of recently used items and a hashmap to store key-value pairs. The doubly linked list provides efficient removal and insertion of nodes for LRU management.

1. **LRUCache(int capacity)**
   - Constructor for initializing the LRUCache with a specified capacity.
   - Initializes the capacity, creates a dummy head and tail for the doubly linked list, and initializes an empty hashmap.
2. **get(int key)**
   - Retrieves the value associated with the given key from the cache.
   - If the key is found in the hashmap:
     - Removes the corresponding node from its current position in the doubly linked list.

        - Moves the node to the front of the linked list (indicating it's the most recently used).
        - Returns the value associated with the key.
    - If the key is not found, returns -1 to indicate a cache miss.
    - **Time Complexity: O(1)**

      Retrieving the value associated with a key from the hashmap is an O(1) operation. Additionally, moving a node to the front of the doubly linked list is also an O(1) operation.
    - **Space Complexity: O(n)**

      The space complexity is O(n) because you are using a hashmap to store key-value pairs, and a doubly linked list to maintain the order of keys.

3. **put(int key, int value)**
    - Inserts or updates a key-value pair in the cache.
    - If the key is found in the hashmap (indicating an update):
        - Updates the value associated with the key.
        - Removes the corresponding node from its current position in the doubly linked list.
        - Moves the node to the front of the linked list (indicating it's the most recently used).
    - If the key is not found (indicating an insert):
        - Creates a new node with the key and value.
            - Inserts the new node to the front of the doubly linked list.
        - Adds the new key-value pair to the hashmap.
    - If the cache exceeds its capacity:
        - Removes the least recently used node from the tail of the linked list.
        - Removes the corresponding key from the hashmap.
    - **Time Complexity: O(1)**

      Inserting or updating a key-value pair in the hashmap is an O(1) operation. Moving a node to the front of the doubly linked list is also an O(1) operation.
    - **Space Complexity: O(n)**

      The space complexity is O(n) because you are using a hashmap to store key-value pairs, and a doubly linked list to maintain the order of keys.