INTRODUCTION TO LRU CACHE

An LRU (Least Recently Used) cache is a type of data structure that is used to manage a limited amount of memory efficiently. It keeps track of the order in which elements are accessed, ensuring that the least recently used elements are removed first when the cache reaches its capacity. This approach optimizes memory usage and improves access speed by keeping the most frequently accessed items readily available.

Key Concepts of LRU Cache

- Capacity: The maximum number of items the cache can hold.

- Eviction Policy: The strategy for removing elements from the cache when it is full. For LRU, the least recently used item is evicted.

- Access Order: Maintains the order of item usage to determine which item was least recently used.

LRU Cache Operations

1. Get Operation:

   - Retrieves an item from the cache.

   - If the item is found, it moves to the front (most recently used position).

   - If the item is not found, it returns a predefined "not found" value.

2. Put Operation:

   - Adds a new item to the cache.

   - If the item already exists, it updates the value and moves it to the front.

   - If the cache is full, it evicts the least recently used item before adding the new one.

Data Structures for LRU Cache

An efficient LRU Cache can be implemented using a combination of a doubly linked list and a hash map (or dictionary). The doubly linked list maintains the order of usage, and the hash map provides quick access to the cache items.

1. Doubly Linked List:

- Nodes contain the cache entries.

- Allows for O(1) time complexity for insertion and deletion of nodes.

- Maintains the order of elements, with the head being the most recently used and the tail being the least recently used.

2. Hash Map:

- Maps keys to their corresponding nodes in the doubly linked list.

- Provides O(1) average time complexity for access and updates.

LRU Cache Implementation in Python

Here's a step-by-step implementation of an LRU Cache in Python using a doubly linked list and a hash map.

python

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}   Hash map to store key-node pairs
        self.head = Node(0, 0)   Dummy head
```

```
    self.tail = Node(0, 0)   Dummy tail
    self.head.next = self.tail
    self.tail.prev = self.head

def _add_node(self, node):
    """Add new node right after head"""
    node.prev = self.head
    node.next = self.head.next
    self.head.next.prev = node
    self.head.next = node

def _remove_node(self, node):
    """Remove an existing node from the linked list"""
    prev = node.prev
    next = node.next
    prev.next = next
    next.prev = prev

def _move_to_head(self, node):
    """Move certain node to the head (most recently used)"""
    self._remove_node(node)
    self._add_node(node)

def _pop_tail(self):
    """Pop the least recently used node"""
    res = self.tail.prev
    self._remove_node(res)
    return res

def get(self, key):
    node = self.cache.get(key)
```

```
        if not node:
            return -1   Key not found
        self._move_to_head(node)
        return node.value

    def put(self, key, value):
        node = self.cache.get(key)
        if not node:
            new_node = Node(key, value)
            self.cache[key] = new_node
            self._add_node(new_node)
            if len(self.cache) > self.capacity:
                Pop the tail
                tail = self._pop_tail()
                del self.cache[tail.key]
        else:
            node.value = value
            self._move_to_head(node)

 Example usage:
cache = LRUCache(2)
cache.put(1, 1)
cache.put(2, 2)
print(cache.get(1))   returns 1
cache.put(3, 3)       evicts key 2
print(cache.get(2))   returns -1 (not found)
cache.put(4, 4)       evicts key 1
print(cache.get(1))   returns -1 (not found)
print(cache.get(3))   returns 3
print(cache.get(4))   returns 4
```

Explanation of the Implementation

- Node Class: Defines the structure of the nodes used in the doubly linked list. Each node stores a key, value, and pointers to the previous and next nodes.

- LRUCache Class:

  - Initialization: Sets up the cache with a given capacity and initializes the dummy head and tail nodes to simplify edge case handling.

  - _add_node: Adds a new node right after the head.

  - _remove_node: Removes an existing node from the linked list.

  - _move_to_head: Moves a node to the head, indicating it was recently accessed.

  - _pop_tail: Removes the least recently used node from the tail.

  - get: Retrieves a value from the cache, moving the node to the head if found.

  - put: Adds a key-value pair to the cache. If the cache exceeds its capacity, it removes the least recently used node.

Benefits of LRU Cache

- Efficiency: Ensures O(1) time complexity for both get and put operations.

- Optimal Memory Usage: Automatically manages memory by keeping only the most frequently accessed items.

- Applicability: Useful in various applications, including web caching, database systems, and operating system page replacement policies.

Conclusion

The LRU Cache is a crucial data structure for optimizing memory usage and access speed in systems where resources are limited. By understanding and implementing the LRU Cache, developers can ensure that their applications make efficient use of available memory, improving performance and reliability.