LRU Cache Implementation: Hash Table + Doubly Linked List

Data Structures and Algorithms Analysis

June 1, 2025

1 Abstract

This document analyzes the implementation of a Least Recently Used (LRU) Cache using a combination of Hash Table and Doubly Linked List data structures. The implementation achieves O(1) time complexity for both GET and PUT operations through strategic use of these complementary data structures.

2 Problem Statement

Design a data structure that supports:

- **GET(key)**: Retrieve value and mark as most recently used
- **PUT(key, value)**: Insert/update key-value pair and mark as most recently used
- **Capacity Management**: Remove least recently used item when capacity is exceeded
- **Time Constraint**: Both operations must execute in O(1) time

3 Data Structure Design

3.1 Hybrid Architecture

The LRU Cache employs a hybrid approach combining two fundamental data structures:

$$\text{LRU Cache} = \text{Hash Table} + \text{Doubly Linked List} \tag{1}$$

- **Hash Table**: $\text{cache} : \text{key} \to \text{Node}$
- **Doubly Linked List**: Maintains access order with $O(1)$ insertion/deletion

3.2 Node Structure

Each cache entry is represented as:

$$\text{Node} = \{key, value, prev, next\} \tag{2}$$

Where:

- key: Hash table key for reverse lookup during eviction
- $value$: Associated data value
- $prev, next$: Doubly linked list pointers

3.3 Sentinel Nodes

The implementation uses dummy head and tail nodes:

$$\text{head} \leftrightarrow \text{Node}_1 \leftrightarrow \text{Node}_2 \leftrightarrow \cdots \leftrightarrow \text{Node}_n \leftrightarrow \text{tail} \tag{3}$$

Benefits:

- Eliminates null pointer checks
- Simplifies insertion/deletion operations
- head.next = Most Recently Used (MRU)
- tail.prev = Least Recently Used (LRU)

4 Algorithm Implementation

4.1 Core Operations

4.1.1 Node Removal

Algorithm 1 Remove Node from Doubly Linked List

Require: Node n to be removed

1: $n.prev.next \leftarrow n.next$

2: $n.next.prev \leftarrow n.prev$

Time Complexity: $O(1)$ - Direct pointer manipulation

4.1.2 Node Insertion at Front

Time Complexity: $O(1)$ - Constant pointer updates

Algorithm 2 Insert Node After Head (Most Recent Position)

Require: Node n to be inserted

$n.next \leftarrow head.next$
$n.prev \leftarrow head$
$head.next.prev \leftarrow n$
$head.next \leftarrow n$

Algorithm 3 GET(key) - Retrieve and Update Access Order

Require: key to retrieve

if $key \in cache$ **then**
  $node \leftarrow cache[key]$
  _remove($node$) {Remove from current position}
  _insert_at_front($node$) {Move to MRU position}
  return $node.value$
else
  return -1 {Key not found}
end if

4.2 GET Operation

Analysis:

$$T_{GET} = T_{hash\_lookup} + T_{remove} + T_{insert} \tag{4}$$
$$= O(1) + O(1) + O(1) = O(1) \tag{5}$$

4.3 PUT Operation

Algorithm 4 PUT(key, value) - Insert/Update with Capacity Management

Require: $key, value$ to insert/update

if $key \in cache$ **then**
  _remove($cache[key]$) {Remove existing node}
else if $|cache| = capacity$ **then**
  $lru \leftarrow tail.prev$ {Get LRU node}
  _remove(lru) {Remove from list}
  delete $cache[lru.key]$ {Remove from hash table}
end if
$new\_node \leftarrow$ Node($key, value$)
_insert_at_front(new_node) {Add as MRU}
$cache[key] \leftarrow new\_node$ {Update hash table}

Analysis:

$$T_{PUT} = T_{hash\_lookup} + T_{remove} + T_{insert} + T_{hash\_update} \tag{6}$$
$$= O(1) + O(1) + O(1) + O(1) = O(1) \tag{7}$$

5 Data Structure Analysis

5.1 Why This Combination Works

Operation	Hash Table Only	Linked List Only	Hybrid Approach
Key Lookup	O(1)	O(n)	O(1)
Insert at Front	N/A	O(1)	O(1)
Remove Arbitrary	N/A	O(n)	O(1)
Find LRU	O(n)	O(1)	O(1)
Space Complexity	O(n)	O(n)	O(n)

5.2 Hash Table Contribution

- **Fast Access**: $O(1)$ key-to-node mapping
- **Existence Check**: Instant determination if key exists
- **Direct Reference**: Eliminates linear search in linked list

5.3 Doubly Linked List Contribution

- **Order Maintenance**: Preserves access sequence naturally
- **Efficient Repositioning**: Move nodes without traversal
- **LRU Identification**: Tail pointer gives instant LRU access
- **Bidirectional Navigation**: Previous pointer enables $O(1)$ removal

6 Complexity Analysis

6.1 Time Complexity

$$\text{GET}(key) : O(1) \tag{8}$$
$$\text{PUT}(key, value) : O(1) \tag{9}$$
$$\text{Eviction} : O(1) \tag{10}$$

6.2 Space Complexity

$$\text{Hash Table} : O(n) \text{ entries} \tag{11}$$
$$\text{Linked List} : O(n) \text{ nodes} \tag{12}$$
$$\text{Total Space} : O(n) \text{ where } n = \text{cache capacity} \tag{13}$$

6.3 Auxiliary Space

$$\text{Per Node} : O(1) \text{ - constant pointers and data} \tag{14}$$
$$\text{Sentinel Nodes} : O(1) \text{ - fixed overhead} \tag{15}$$

7 Key Design Decisions

7.1 Why Doubly Linked List vs Singly Linked List?

- **Removal Operation**: Requires access to previous node
- **Singly Linked**: $O(n)$ to find previous node
- **Doubly Linked**: $O(1)$ direct access via *prev* pointer

7.2 Why Hash Table vs Array?

- **Key Space**: Keys may not be contiguous integers
- **Dynamic Range**: Hash table handles arbitrary key ranges
- **Memory Efficiency**: No space waste for unused indices

7.3 Sentinel Node Benefits

$$\text{Without Sentinels:} \quad \text{Special cases for empty list, single node} \tag{16}$$
$$\text{With Sentinels:} \quad \text{Uniform operations, simplified logic} \tag{17}$$

8 Implementation Walkthrough

8.1 Example Execution

Consider LRU Cache with capacity = 2:

Step 1: PUT(1, 10)

$$\text{cache} = \{1 \to \text{Node}(1, 10)\} \tag{18}$$
$$\text{Order: head} \leftrightarrow \text{Node}(1, 10) \leftrightarrow \text{tail} \tag{19}$$

Step 2: PUT(2, 20)

$$\text{cache} = \{1 \to \text{Node}(1, 10), 2 \to \text{Node}(2, 20)\} \tag{20}$$
$$\text{Order: head} \leftrightarrow \text{Node}(2, 20) \leftrightarrow \text{Node}(1, 10) \leftrightarrow \text{tail} \tag{21}$$

Step 3: GET(1)

$$\text{Move Node}(1, 10) \text{ to front} \tag{22}$$
$$\text{Order: head} \leftrightarrow \text{Node}(1, 10) \leftrightarrow \text{Node}(2, 20) \leftrightarrow \text{tail} \tag{23}$$

Step 4: PUT(3, 30) - Triggers Eviction

$$\text{Remove LRU: Node}(2, 20) \tag{24}$$
$$\text{cache} = \{1 \rightarrow \text{Node}(1, 10), 3 \rightarrow \text{Node}(3, 30)\} \tag{25}$$
$$\text{Order: head} \leftrightarrow \text{Node}(3, 30) \leftrightarrow \text{Node}(1, 10) \leftrightarrow \text{tail} \tag{26}$$

9 Advantages and Applications

9.1 Advantages

- **Optimal Time Complexity**: O(1) for all operations
- **Memory Efficient**: Linear space usage
- **Cache-Friendly**: Exploits temporal locality
- **Scalable**: Performance independent of cache size

9.2 Real-World Applications

- **CPU Caches**: Hardware cache replacement
- **Operating Systems**: Page replacement algorithms
- **Database Systems**: Buffer pool management
- **Web Caching**: Browser and CDN caching
- **Application Caching**: Redis, Memcached implementations

10 Conclusion

The LRU Cache implementation demonstrates the power of combining complementary data structures:

- **Hash Table**: Provides O(1) random access
- **Doubly Linked List**: Maintains O(1) ordered operations
- **Synergy**: Each structure compensates for the other's limitations

This hybrid approach achieves the theoretical optimum for LRU cache operations, making it a cornerstone algorithm in systems programming and cache design.