DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING
BACHELORS IN COMPUTER SYSTEMS ENGINEERING
Course Code: CS-218

Course Title: Data Structures & Algorithms

Complex Engineering Problem

SE Batch 2023, Fall Semester 2024
Grading Rubric
TERM PROJECT Group

Members:

Student No.	Name	Roll No.
S1	ZARA AKRAM	CS23104
S2	AREEBA KHAN	CS23110
S3		

CRITERIA AND SCALES				Marks Obtained		
				S1	S2	S3
Criterion 1: Has the student provided the appropriate design of LRU data structure?						
0	1	2	-			
The chosen design is too simple	The design is fit to be chosen for a class project	The choice is different and impressive.	-			
Criterion 2: How good is the programming implementation?						
0	1	2	3			
The project could not be implemented	The project has been implemented partially.	The project has been implemented completely but can be improved.	The project has been implemented completely and impressively			
Criterion 3: How well written is the report?						
0	1	2	-			
The submitted report is unfit to be graded	The report is partially acceptable	The report is complete and concise				
			Total Marks:			

Contents

COMPLEX ENGINEERING PROBLEM

PROBLEM DESCRIPTION:

The task is to design a data structure in Python that follows the constraints of a *Least Recently Used (LRU)* cache and find its time and space complexities.

By Implement the LRUCache class with functionality:

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. Each call to put and get functions is counted a reference.

DATA STRUCTURES ANALYSIS:

To achieve optimal time complexity for get and put operations, the LRU Cache integrates two core data structures:

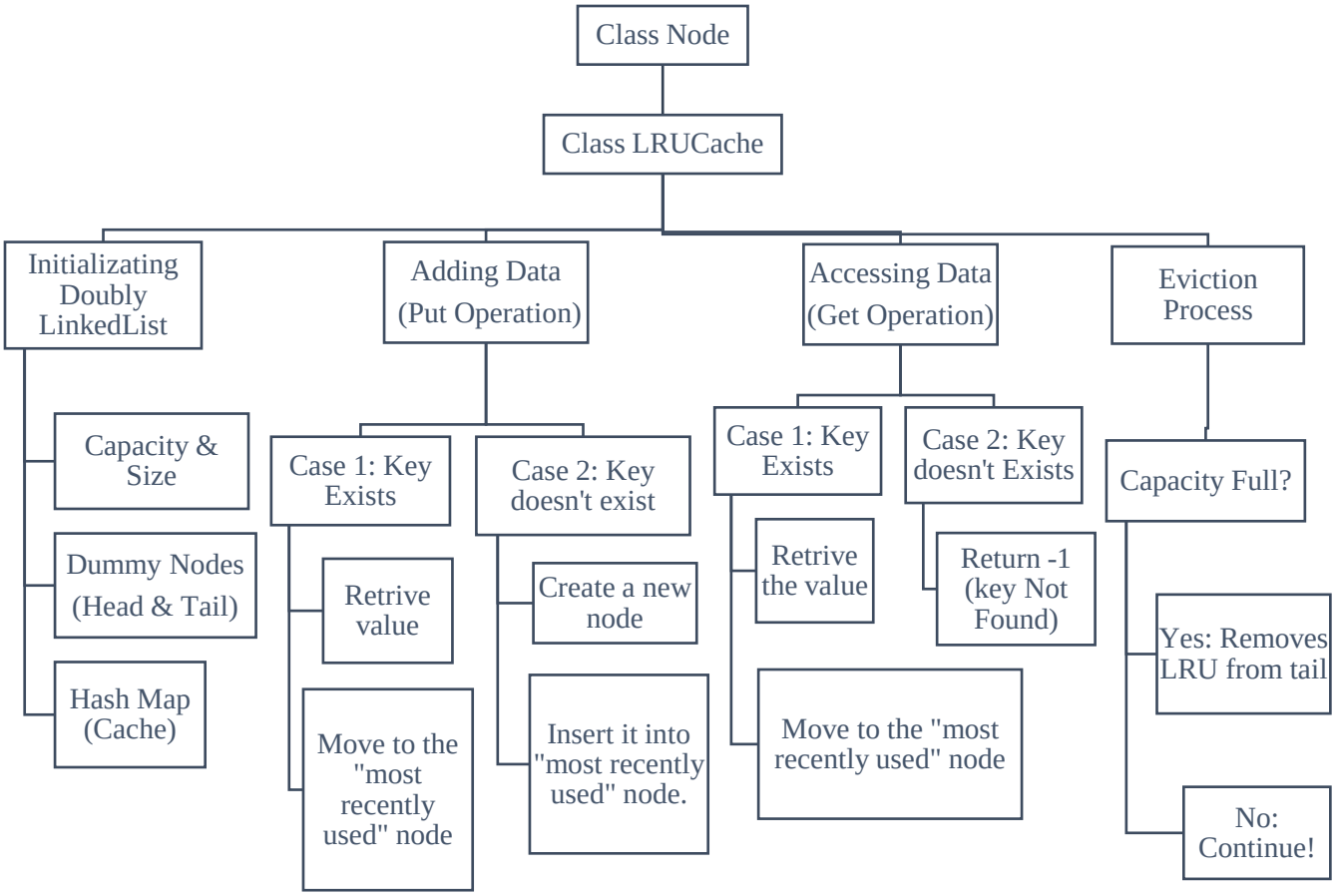
Doubly Linked List:

- Each cache item is represented as a node in a doubly linked list.
- The doubly linked list facilitates constant-time addition and removal of nodes from either end.
- It maintains the order of items, ensuring that the **"most recently used**" item is always at the rightmost position, while the **"least recently used"** item is at the leftmost position.

Hash Map (Dictionary):

- The hash map allows lookup for nodes using their keys.
- It maintains a direct reference to the corresponding linked list node, ensuring quick access without traversal.

FLOW OF THE PROJECT



EXPLANATION:

Initialization:
The LRUCache is initialized with a capacity, dummy head and tail nodes, and a hash map for fast access.

Adding Data:
On put, if the key exists, update its value and move it to the right end. If not, add a new node and evict the least recently used item if the capacity is exceeded.

Accessing Data:
On get, if the key exists, move it to the right end and return its value. If not, return -1.

Eviction Process:
When the cache reaches capacity, the least recently used item is removed from the leftmost position and deleted from the hash map.

CHALLENGING PART FOR PROJECT:

Maintaining LRU Order:
Updating the doubly linked list to reflect the LRU order while ensuring fast access was tricky and required precise pointer management.

Achieving O(1) Complexity:
Ensuring all operations were constant time and that the hash map and linked list stayed synchronized was challenging.

Syncing the Data Structures:
Ensuring the hash map and doubly linked list stayed synchronized was challenging. Any mismatch could lead to incorrect cache behavior, so careful coding and thorough testing were essential.

KEY CONCEPTS LEARNED DURING THE PROJECT:

Doubly Linked Lists and Hash Maps:
We gained valuable insight into using doubly linked lists with hash maps for efficient data structures like the LRU cache, enabling constant-time operations crucial for performance.

Improved Object-Oriented Programming (OOP) Skills:
This project enhanced my OOP skills, particularly in defining classes, managing objects, and structuring code efficiently for better performance and maintainability.

Memory Management:
We improved our understanding of memory management by ensuring efficient deletion of nodes from both the list and hash map.

TIME COMPLEXITY:
MAIN FUNCTIONS:

1. **get(key):**
   Accessing the cache dictionary, removing and inserting nodes into the doubly linked list, and updating the node position all happen in constant time.
   Time Complexity= O(1)

2. **put(key, val):**
   Accessing the cache dictionary, adding or updating a node in the doubly linked list, and handling eviction (if needed) all occur in constant time.
   Time Complexity= O(1)

SUPPORTING FUNCTIONS:

1. **add_value(node):**
   Adds a node to the head of the doubly linked list.
   Time Complexity: O(1)
2. **remove_node(node):**
   Removes a node from the doubly linked list by updating the previous and next pointers.
   Time Complexity: O(1)

3. **move_to_head(node):**
   Removes the node from its current position and adds it to the head of the doubly linked list.
   Time Complexity: O(1)
4. **evict_from_tail():**
   Removes the least recently used node from the tail of the doubly linked list and deletes it from the cache.
   Time Complexity: O(1)

SPACE COMPLEXITY:

1. **Cache Dictionary (cache)**: It stores up to n key-value pairs.

   Space Complexity = O(n).

2. **For the doubly linked list**: It stores n nodes, each of which holds a key, value, and pointers for the doubly linked list.
   Space Complexity = O(n).

Total: O(n) (where n is the number of elements (or keys) stored in the cache).

TEST CASES RUN:

TEST CASE 1:

```
# Initialize the cache with a capacity of 50
cache = LRUCache(50)

# Situation 1: Filling the cache with keys 0 to 49 (100% miss rate)
misses_situation_1 = 0
for i in range(50):
    if cache.get(i) == -1:  # Each get is initially a miss
        misses_situation_1 += 1
    cache.put(i, i)  # Fill the cache

print("\nThe Total Hits are:", 0)
print("The Total Misses are:", misses_situation_1)
print("Cache Keys:", cache.cache.keys())
```

```
The Total Hits are: 0
The Total Misses are: 50
Cache Keys: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1
0, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 3
7, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

TEST CASE 2:

```
# Situation 2: Accessing keys alternately between odd (hits) and even (misses)
misses_situation_2 = 0
hits_situation_2 = 0
for i in range(50):  # Alternate accesses: first 25 hits, then 25 misses
    if i % 2 == 0:  # Even numbers from 0 to 48 should miss
        if cache.get(i + 50) == -1:
            misses_situation_2 += 1
    else:  # Odd numbers from 1 to 49 should hit
        if cache.get(i) != -1:
            hits_situation_2 += 1
        else:
            misses_situation_2 += 1

print("\nThe Total Hits are:", hits_situation_2)
print("The Total Misses are:", misses_situation_2)
print("Cache Keys:", cache.cache.keys())
```

```
The Total Hits are: 25
The Total Misses are: 25
Cache Keys: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

TEST CASE 3:

```
misses_situation_3 = 0
hits_situation_3 = 0
for prime in primes:
    if cache.get(prime) == -1:  # Check for miss before inserting
        misses_situation_3 += 1
    else:
        hits_situation_3 += 1
    cache.put(prime, prime)

print("\nThe Total Hits are:", hits_situation_3)
print("The Total Misses are:", misses_situation_3)
print("Cache Keys:", cache.cache.keys())
```

```
The Total Hits are: 15
The Total Misses are: 10
Cache Keys: dict_keys([1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 2
1, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 53, 59, 6
1, 67, 71, 73, 79, 83, 89, 97])
```

FINAL MISS RATE:

```
# Total retrievals and misses for each situation
total_misses = misses_situation_1 + misses_situation_2 + misses_situation_3
total_hits = hits_situation_2 + hits_situation_3
total_retrievals = total_hits + total_misses

# Calculate and print the final miss rate
miss_rate = (total_misses / total_retrievals) * 100
print(f"\nFinal Miss Rate: {miss_rate:.1f}%")
```

```
Final Miss Rate: 68.0%

Process finished with exit code 0
```