Report on CEP Project: LRU Cache Implementation

---

1. Problem Description

The project task was to design and implement a Least Recently Used (LRU) Cache, a commonly required structure in software applications where efficient, quick access to recently used data is essential. An LRU cache maintains its most recently used items, removing the least recently used item when the cache reaches maximum capacity. This method is essential in applications with limited memory or fast data access needs, such as web browsers, operating systems, and database management systems.

The Role of Caching: Caching is widely used across multiple domains:

- **Web Browsers** store recently accessed web pages locally, improving load times for frequently visited sites.
- **Operating Systems** manage memory by caching recently used files, making them quickly accessible.
- **Databases** use caches to store results of recent queries, reducing query response time.

Why Use an LRU Cache? The LRU policy is effective when memory is limited, but high access speed is critical. An LRU cache keeps only the most recently accessed items available in memory, while less frequently used items are removed, thereby balancing storage capacity with fast data retrieval.

2. Project Flow and Data Structures Used

The LRU cache project was carefully structured to maintain optimal time complexity for both retrieving and updating cache entries. Two data structures were combined:

Doubly Linked List:

- Each item in the cache is represented by a node in a doubly linked list, with pointers to both previous and next nodes. This structure enables quick addition or removal of nodes from either end.
- By organizing items in LRU order, the doubly linked list provides efficient node relocation to the "most recently used" position upon access, maintaining the cache's logical structure.

Hash Map:

- A hash map (Python dictionary) links each key to its corresponding node in constant time, allowing efficient access without list traversal.
- This hash map maintains consistency by linking keys to node references, creating a fast and reliable lookup table.

Flow of the Project:

Initialization:

- The cache was initialized with a fixed capacity and two dummy nodes representing the "least recently used" (left) and "most recently used" (right) ends of the cache.

Adding Data:

- When a new item is added, the cache checks if it's at capacity. If so, the least recently used item is removed from the left end. Then, the new item is inserted at the right end of the list.

Accessing Data:

- Each time a cached item is accessed, it's moved to the right end, updating its position as the "most recently used" item.

Eviction Process:

- If the cache reaches capacity, the leftmost node is removed, ensuring the cache does not exceed its set limit.

3. Challenges Faced

The most challenging aspects of this project were:

Maintaining the LRU Order Efficiently:

- Keeping the doubly linked list updated to reflect the LRU order while ensuring efficient node access required careful management of node pointers. Each access or addition involved repositioning nodes to the "most recently used" end.

Ensuring Constant Time Complexity:

- Achieving constant time complexity for cache operations was essential for project success. Synchronizing the hash map and linked list data structures without introducing delays or inconsistencies was challenging.

Managing Edge Cases:

- Edge cases, such as accessing nonexistent keys, handling duplicate keys, and managing cache boundaries, posed unique challenges. Each required additional validations to prevent inconsistencies, memory issues, or runtime errors.

Approaches to Overcome Challenges:

- The group implemented thorough testing to account for various scenarios, including cache hits and misses, capacity limits, and validation of keys. By rigorously testing, we addressed each challenge and enhanced the cache's reliability.

4. New Python Concepts Learned

During the project, the team explored several advanced Python concepts, which enhanced our technical skills:

Assertions for Input Validation:

- We utilized assertions to validate cache capacity and key values, which helped in detecting errors early and maintaining data integrity.

Doubly Linked Lists:

- By using a doubly linked list, we managed quick insertion and deletion from either end, which would be cumbersome with a singly linked list. This structure provided the efficiency required for the cache.

Integrated Use of Hash Maps and Linked Lists:

- The project highlighted the effectiveness of combining multiple data structures to achieve performance goals. By linking the hash map and linked list, we ensured the cache was both fast and reliable.

Memory Management:

- Efficient memory handling, including timely node deletion from the list and hash map, was essential. This experience illustrated the importance of carefully managing memory resources in large-scale applications.

5.Test Case Runs:

Test case #1:

```
76  #drivers code
77  # Create cache with capacity 50
78  cache = LRUCache(50)
79
80  # Fill the cache with keys 0 to 49
81  for i in range(50):
82      cache.put(i, i)
83  print("The Total Hits are: ",cache.hits)
84  print("The Total Misses are: ",cache.misses)
85  cache.print_cache()
86  print()
```

```
The Total Hits are:  0
The Total Misses are:  50
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 #2:

```
88  # Access all odd-numbered keys to update hits/misses
89  for i in range(100):
90      if i%2 !=0:
91          cache.get(i)
92  print("The Total Hits are: ",cache.hits)
93  print("The Total Misses are: ",cache.misses)
94  cache.print_cache()
95  print()
```

```
The Total Hits are:  25
The Total Misses are:  75
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:

```
97   #filling the cache with prime numbers from 0 to 100
98   for i in range(100):
99       count=0
100      for j in range(1,i+1):
101          if i%j==0:
102              count+=1
103      if count ==2:
104          cache.put(i,i)
105  print("The Total Hits are: ",cache.hits)
106  print("The Total Misses are: ",cache.misses)
107  cache.print_cache()
108  print()
```

```
The Total Hits are:  40
The Total Misses are:  85
dict_keys([1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 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, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97])
```

Final Miss Rate:

```
110  # Print the final miss rate
111  print("Final Miss Rate:", cache.missrate(), "%")
112  print()
113
```

```
Final Miss Rate: 68.0 %
```

Time Complexity

1. get(key):

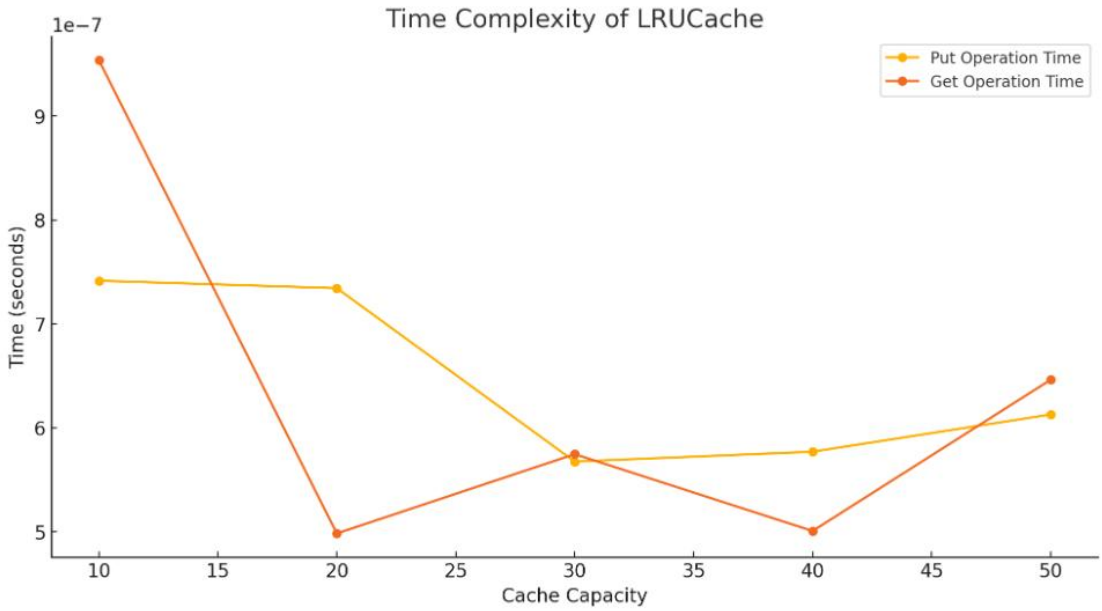
- **Accessing the** cache **dictionary**: O(1)average case.
- **Removing a node from the doubly linked list**: O(1).
- **Inserting a node into the doubly linked list**: O(1).
- **Overall**: O(1).

2. put(key, val):

- **Accessing the** cache **dictionary**: O(1) average case.
- **Removing a node from the doubly linked list**: O(1).
- **Inserting a node into the doubly linked list**: O(1).
- **Adding or updating the** cache **dictionary**: O(1).
- If the cache exceeds capacity:
    - **Removing the least recently used node**: O(1).
    - **Deleting from the** cache **dictionary**: O(1).
- **Overall**: O(1).

Overall Time Complexity: O(1)

3. Time Complexity Graph:



4. Auxiliary operations:

- missrate(): Calculating a division is O(1).
- print_cache(): Printing keys is O(N), where N is the number of keys in the cache.

Space Complexity

1. Data Structures:

- **Cache dictionary (**self.cache**)**: Stores up to N key-node pairs, where N=capacity, requiring O(N) space.
- **Doubly linked list**: At most N nodes, each requiring O(1) space for key, val, prev, and next. Total O(N).

2. Total:

- O(N) space.

3.Space Complexity Graph: