DATA STRUCURES AND ALGORITHMS

S.E. (CIS) OEL REPORT

PROJECT GROUP ID: G3-7

MUHAMMAD OWAIS	CS-22080
ZUHAIB NOOR	CS-22081
AYAN KHAN	CS-22083

BATCH: 2022

JAN 2024

Department of Computer and Information Systems Engineering

NED University of Engg. & Tech., Karachi-75270

Table of Contents

1. Problem Description

Design a data structure in Python that follows the constraints of a Least Recently Used (LRU) cache and find its time and space complexities.

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

Example:

Input

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

Output

[null, null, null, 1, null, -1, null, -1, 3, 4]

Constraints:

1 <= capacity <= 50

0 <= key <= 100

0 <= value <=100

Test the above task by filling the full cache using keys 0-49. Retrieve the odd number key values. Fill the cache with prime number keys 0-100. In the end, compute the final miss rate.

2. Methodology

Before jumping on to the methodologies used to solve this problem, let's first discuss **what is LRU cache?**. A **Least Recently Used (LRU) Cache** is a data structure that maintains a limited set of items, where the eviction policy is based on the least recently accessed or used item, ensuring efficient storage and retrieval of recently accessed elements.

LRU Cache Implementation:

The LRU cache is implemented using the Doubly Linked List along with an OrderedDict from the collections module, providing enhanced functionality for tracking key-value pairs.

LRU Class:

Initializes an LRU cache with a specified capacity. It utilizes a doubly linked list (dll) for managing the order of elements and an ordered dictionary (dic) for tracking key-value pairs. Tracks the number of references (track) and the overall length of the cache (length).

put Method: Inserts a new key-value pair into the cache. If the key already exists, updates the value and adjusts the order in the linked list. It evicts the least recently used element if the cache exceeds its capacity.

get Method:

Retrieves the value of a given key from the cache. It updates the order in the linked list based on the accessed key. It can return -1 if the key is not found in the cache.

tra Method:

Prints the elements in the order of their usage, separating the "Least Recently Used" and "Recently Used" portions.

Preservation of Insertion Order:

The insertion order of key-value pairs in the OrderedDict ensures that the LRU cache can easily identify the least recently used element for eviction when the cache reaches its capacity.

Efficient Popitem Operation:

The popitem method in the OrderedDict facilitates the removal of the least recently used element with a time complexity of O(1). This operation is instrumental in maintaining the cache within its specified capacity.

Time complexity:

In worst cases

- The insertion and update operations for both data structures take O(1) time.
- The time complexity of the get method is O(1).
- In the worst case, traversing the elements takes O(n) time, where n is the number of elements in the cache. Therefore, the time complexity of the tra method is O(n).

Therefore, for put and get functions, the overall time complexity is O(1).

In summary, the combination of a Doubly Linked List and an OrderedDict provides a robust solution for implementing the LRU cache, ensuring efficient management of elements and quick identification of the least recently used item for eviction. The OrderedDict, in particular, enhances the overall performance of the cache by preserving the insertion order of key-value pairs. The provided driver program demonstrates the practical application of the LRU cache and showcases its effectiveness in handling various access patterns.

3. Results

For the following driver code, the output looks like this:

```
l = LRU(2)
print()
print()
print(l.put(1,1), end=", ")
print(l.put(2,2), end=", ")
print(l.get(1), end=", ")
print(l.put(3,3), end=", ")
print(l.get(2), end=", ")
print(l.put(4,4), end=", ")
print(l.get(1), end=", ")
print(l.get(3), end=", ")
print(l.get(4))
print()
print()
print()
l.tra()
```

```
None, None, 1, None, -1, None, -1, 3, 4
```

```
Least Recently Used
[3, 3]
[4, 4]
Recently Used
```

For testing constraints:

```
After:                  Before:
Least Recently Used     Least Recently Used
[22, 22]                [1, 1]
[24, 24]                [2, 2]
[26, 26]                [3, 3]
[28, 28]                [4, 4]
[30, 30]                [5, 5]
[32, 32]                [6, 6]
[34, 34]                [7, 7]
[36, 36]                [8, 8]
[38, 38]                [9, 9]
[40, 40]                [10, 10]
[42, 42]                [11, 11]
[44, 44]                [12, 12]
[46, 46]                [13, 13]
[48, 48]                [14, 14]
[50, 50]                [15, 15]
[1, 1]                  [16, 16]
[9, 9]                  [17, 17]
[15, 15]                [18, 18]
[21, 21]                [19, 19]
[25, 25]                [20, 20]
[27, 27]                [21, 21]
[33, 33]                [22, 22]
[35, 35]                [23, 23]
[39, 39]                [24, 24]
[45, 45]                [25, 25]
[49, 49]                [26, 26]
[2, 2]                  [27, 27]
[3, 3]                  [28, 28]
[5, 5]                  [29, 29]
[7, 7]                  [30, 30]
[11, 11]                [31, 31]
[13, 13]                [32, 32]
[17, 17]                [33, 33]
[19, 19]                [34, 34]
[23, 23]                [35, 35]
[29, 29]                [36, 36]
[31, 31]                [37, 37]
[37, 37]                [38, 38]
[41, 41]                [39, 39]
[43, 43]                [40, 40]
[47, 47]                [41, 41]
[53, 53]                [42, 42]
[59, 59]                [43, 43]
[61, 61]                [44, 44]
[67, 67]                [45, 45]
[71, 71]                [46, 46]
[73, 73]                [47, 47]
[79, 79]                [48, 48]
[83, 83]                [49, 49]
[89, 89]                [50, 50]
Recently Used           Recently Used
Miss Rate : 68.0
```