DSA

OEL Report

Project Group ID:
Tuba Naushad CS-22021
Bareera CS -22030

Batch: 2022

Date: 12th January, 2024

Contents

Problem definition:

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.
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.

Methodology:

Algorithm:

1. Node Class:
=>Initialize a Node class with attributes key, value, next, and prev.
2. LRU_cache Class:
=>Initialize the LRU_cache class with a given capacity.
=>Create a dictionary (dic) for quick lookup of key-node pairs.
=>Create head and tail sentinel nodes for a doubly linked list, and connect them to form an empty list.
=>Initialize counters (miss and totalcount) for cache statistics.
3. add Method:
=>Accept a node.
=>Get the previous node (p) of the tail.
=>Connect the previous node's next to the new node.
=>Connect the tail's prev to the new node.
=> Connect the new node's next to the tail.
=>Connect the new node's prev to the previous node.
4. remove Method:
=>Accept a node to be removed.
=>Get the previous node (p) and the next node (n) of the given node.
=>Connect the previous node's next to the next node.
=>Connect the next node's prev to the previous node.
5. _str_ Method:
=>Create an empty list (result) to store key-value pairs.
=>Start from the node after the head and traverse the linked list until the tail.
=>Append each node's key-value pair to the result list.
=>Return the string representation of the result list.

6. get Method:

=>Accept a key parameter.

=>Increment the total count.

If the key is in the dictionary:

=>Get the corresponding node (n) from the dictionary.

=>Remove the node from its current position in the linked list.

=>Add the node to the end of the linked list.

=>Return the value of the node.

If the key is not in the dictionary:

=>Increment the miss counter.

=>Return -1.

7. put Method:

=>Accept key and value parameters.

=>Increment the total count.

If the key is in the dictionary:

=>Get the corresponding node (node) from the dictionary.

=>Update the node's value.

=>Remove the node from its current position in the linked list.

=>Add the node to the end of the linked list.

If the key is not in the dictionary:

=>Create a new node with the given key and value.

=>Add the new node to the end of the linked list.

=>Add the new node to the dictionary with the key.

If the dictionary size exceeds the capacity:

=>Increment the miss counter.

=>Get the least recently used node from the front of the linked list.

=>Remove the least recently used node from the linked list.

=>Delete the least recently used key from the dictionary.

Time complexity:

1. Get Operation (Lookup):

Best Case: when the requested key is at the end of the cache.

Worst Case: using a dictionary for direct key-node lookup.

For the both Best case and Worst case: O(1)

2. Put Operation (Insertion/Update):

Best Case: when inserting a new key-value pair without exceeding the capacity.

Worst Case: updating an existing key's value and moving the node to the end of the list. When

For the both Best case and Worst case: O(1)

3. Cache Initialization:

Creating sentinels and initializing variables doesn't depend on the cache size: O(1)

Time Complexity = O(1) + O(1) + O(1) = O(1)

Space complexity:

1. Cache Capacity:

Dictated by the given capacity parameter ‘n’: O(n)

2. Node Objects:

Each ‘n’ unique key requires a node object in the cache: O(n)

3. Dictionary Storage:
The dictionary stores key-node pairs, occupying space proportional to ‘n’ number of unique keys in the cache: O(n)
Space Complexity = O(n) + O(n) + O(n) = O(n)

Results:

```
Odd-numbered keys from 1-50
1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49,
Cache after retrieving odd-numbered keys
[(0, 0), (2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (12, 12), (14, 14), (16, 16), (18, 18), (20, 20), (22, 22), (24, 24), (26, 26), (28, 28), (30, 30), (32, 32),
(34, 34), (36, 36), (38, 38), (40, 40), (42, 42), (44, 44), (46, 46), (48, 48), (1, 1), (3, 3), (5, 5), (7, 7), (9, 9), (11, 11), (13, 13), (15, 15), (17, 17), (19, 19),
(21, 21), (23, 23), (25, 25), (27, 27), (29, 29), (31, 31), (33, 33), (35, 35), (37, 37), (39, 39), (41, 41), (43, 43), (45, 45), (47, 47), (49, 49)]
```

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

```
Cache after filling prime-numbered keys
[(22, 22), (24, 24), (26, 26), (28, 28), (30, 30), (32, 32), (34, 34), (36, 36), (38, 38), (40, 40), (42, 42), (44, 44), (46, 46), (48, 48), (1, 1), (9, 9), (15, 15),
(21, 21), (25, 25), (27, 27), (33, 33), (35, 35), (39, 39), (45, 45), (49, 49), (2, 2), (3, 3), (5, 5), (7, 7), (11, 11), (13, 13), (17, 17), (19, 19), (23, 23),
(29, 29), (31, 31), (37, 37), (41, 41), (43, 43), (47, 47), (53, 53), (59, 59), (61, 61), (67, 67), (71, 71), (73, 73), (79, 79), (83, 83), (89, 89), (97, 97)]
```

```
Total Count: 100
Miss Count: 60
Miss Rate: 60.0
```