DATA STRUTURES AND ALGORITHMS

(CS-218)

CEP REPORT

CLASS: 2ND YEAR-A **BATCH:** 2022

SUBMITTED BY:

AMINA SHAHZAD (CS-22019)

ANEEBA ZAFAR (CS-22029)

AYESHA TAUFIQUE (CS-22036)

DEPARTMENT: COMPUTER AND INFORMATION SYSTEM ENGINEERING

SUBMITTED TO: MISS HAMEEZA AHMED

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.

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]

Explanation

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1} lRUCache.put(2,

2); // cache is {1=1, 2=2} lRUCache.get(1); // return

1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3} lRUCache.get(2);

// returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found) lRUCache.get(3); // return

3

lRUCache.get(4); // return 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.

ALGORITHM OF THE CODE:

The provided code defines an LRU (Least Recently Used) cache using a Python class LRU. Here's a high-level algorithm for the functionalities implemented in this code:

Algorithm:

1. **updated(dict1, dict2) Function:**
   - Merges two dictionaries (dict1 and dict2) and returns the merged dictionary.
2. **addList(list1, item) Function:**
   - Appends an item to the end of the list1 and returns the updated list.
3. **length(dict) Function:**
   - Calculates and returns the length of the dictionary by counting its keys.
4. **dicGet(item, dic) Function:**
   - Searches for an item in the dictionary dic and returns its associated value.
5. **LRU Class:**

   init(self, size) Method:
   - ▪ Initializes the LRU cache with a specified **‘size’**.
   - ▪ Initializes an empty dictionary **‘self.dict’** to store key-value pairs.
   - ▪ Sets the cache size, cache miss count, and an empty **‘track’** list to track recent accesses.

   put(self, key, item) Method:
   - ▪ Checks if the cache is full.
     - If the cache is full:
       - If the **‘key’** already exists in the cache **(‘key in self.dict’**):
         - ▪ Updates the **‘track’** list to maintain the most recently used order by moving the **‘key’** to the End.
         - ▪ Removes the **‘key’** and its associated value from the cache.
       - If the **‘key’** does not exist in the cache:
         - ▪ Increases the cache miss count.
         - ▪ Removes the least recently used item from the cache **(‘x = self.track.pop(0)’**) and the dictionary.
       - Adds the new **‘key:item’** pair to the cache.
     - If the cache has space available:
       - Increases the cache miss count.
       - Adds the new **‘key:item’** pair directly to the cache.
     - Updates the **‘track’** list to reflect the most recent usage.

   get(self, key) Method:
   - Retrieves the value associated with the given **‘key’** from the cache.
   - If the **‘key’** exists in the cache:
     - ▪ Updates the **‘track’** list to reflect the most recent usage.
     - ▪ Returns the corresponding value.
   - If the **‘key’** doesn’t exist in the cache, increments the cache miss count and returns **‘-1’**.

   str(self) Method:
   - Generates a string representation of the cache and its contents.

Overall Flow:

1. The code initializes an LRU cache l1 with a size of 50.
2. It inserts numbers from 0 to 49 into the cache using l1.put(i, i).

3. Retrieves odd numbers from 1 to 49 using l1.get(i) and prints the cache after each retrieval.
4. Inserts prime numbers from a predefined list k into the cache using l1.put(i, i).
5. Prints the cache content and the miss rate (l1.cache_miss).

This algorithm creates an LRU cache using a dictionary and a list to maintain the order of item accesses, and it follows the LRU eviction policy by removing the least recently used item when the cache is full.

DATA STRUCTURES USED IN THE PROGRAM:

The provided code primarily uses the following data structures:

1. Dictionary (dict): Used for implementing the main data storage of the cache.

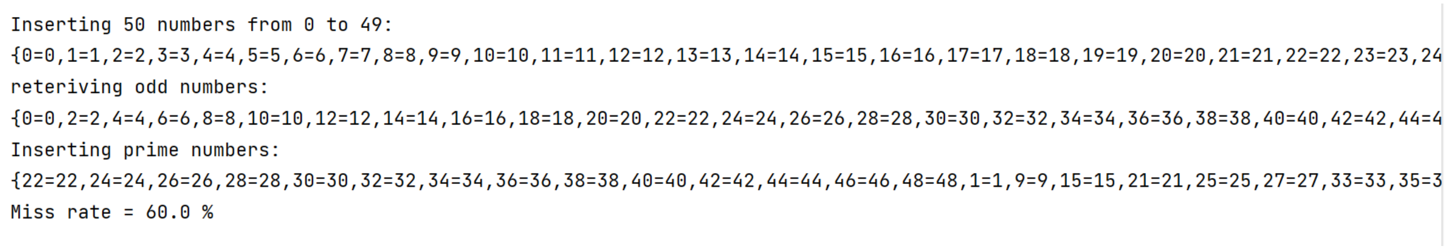
- Stores key-value pairs where keys are used to access corresponding values efficiently.
- In this code, the self.dict dictionary in the LRU class stores the cache items.

2. List (list):

- Utilized to keep track of the order of item accesses or to maintain a queue-like structure.
- Stores the keys or items in the order they were accessed or inserted.
- The self.track list in the LRU class is used to track the order of key accesses in the cache.

These data structures are essential components for managing and organizing the LRU cache system implemented in the provided code. The dictionary serves as the main storage for key-value pairs, while the list helps in maintaining the order of access and implementing the eviction policy based on the least recently used items.

SCREENSHOT OF THE OUTPUT:



```
Inserting 50 numbers from 0 to 49:
{0=0,1=1,2=2,3=3,4=4,5=5,6=6,7=7,8=8,9=9,10=10,11=11,12=12,13=13,14=14,15=15,16=16,17=17,18=18,19=19,20=20,21=21,22=22,23=23,24
reteriving odd numbers:
{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=4
Inserting prime numbers:
{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=3
Miss rate = 60.0 %
```

MOST CHALLENGING PART WHILE WORKING ON THE PROJECT:

Overall the project is easy but there is only one part which is challenging and that is to build the logic of cache miss it has taken a little bit more time as compared to others but we have resolved it afterwards.

INDIVIDUAL CONTRIBUTION OF EACH GROUP MEMBER IN THE PROJECT:

- **Amina Shahzad (CS-22019):**
  She coded the most difficult part of our project that are the functions under the LRU class ‘put’ and ‘get’ that involves the concept of the cache miss. She also coded the test code of the program.
- **Aneeba Zafar (CS-22029):**
  She coded the ‘updated’, ‘addList’ and ‘length’ function. She also made the report of the project including the algorithm for the code.
- **Ayesha Taufique (CS-22036):**
  She coded ‘dicGet’ function and the ‘constructor (__init__ function)’ and ‘string’ function of the LRU class. She also helped in making the test code of the program.