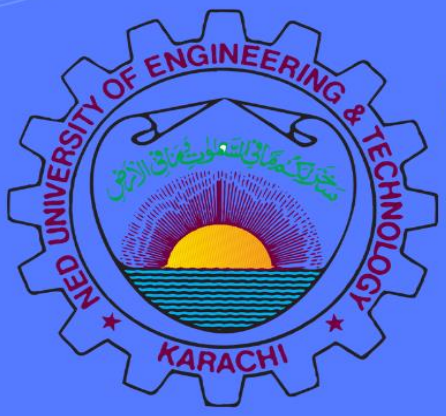


Ned University Of Engineering & Technology

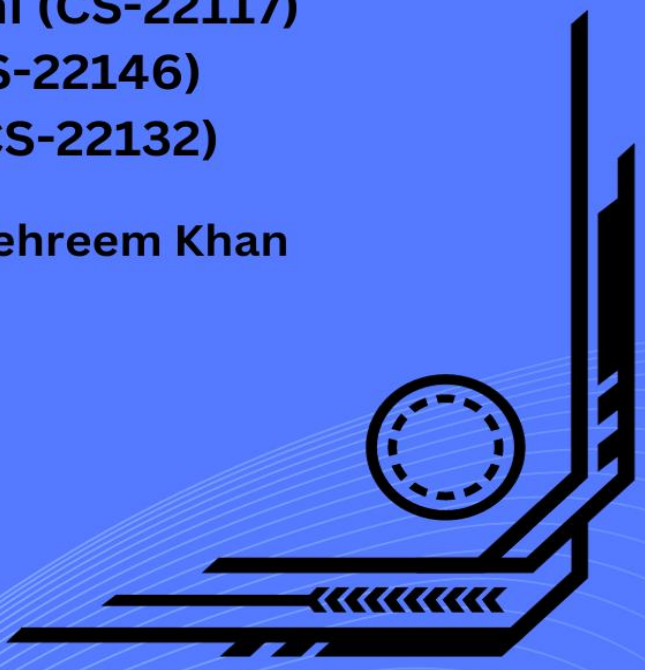
DSA OEL

“LRU CACHE”

G4-9

Moatasim Qureshi (CS-22117)
Hamza Ali (CS-22146)
Awab Shuja (CS-22132)

Submitted to Ms Tehreem Khan



DATA STRUCTURES & ALGORITHMS

S.E. (CIS) OEL REPORT

Project Group ID: G4-9

Moatasim Qureshi	CS-22117
Muhammad Awab Shuja	CS-22132
Hamza Ali	CS-22146

BATCH: 2022

Jan 2024

Department of Computer and Information Systems Engineering

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

CONTENTS

PROBLEM DESCRIPTION:

Design a data structure in Python that adheres to the constraints of a Least Recently Used (LRU) cache. Implement the LRUCache class with the following functionalities:

- LRUCache(int capacity): Initialize the LRU cache with a positive size capacity.
- int get(int key): Return the value of the key if it exists; otherwise, return -1.
- void put(int key, int value): Update the value of the key if it exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity, evict the least recently used key. Each call to put and get functions is counted as a reference.

METHODOLOGY:

Data Structure Used:

The implementation uses a linked list to represent the structure of the LRUCache. Each node in the linked list holds a key-value pair, and the list is manipulated to maintain the order of most recently used keys.

LRU Cache Overview:

A Least Recently Used (LRU) cache is a type of cache that maintains a limited number of items and evicts the least recently used item when the capacity is reached. The goal is to store the most recently and frequently used items, optimizing for fast access.

Overall Strategy:

I. Linked List Representation:

- Nodes in the linked list represent key-value pairs.
- The order of nodes reflects the order of usage, with the most recently used node at the rear.

II. Put Operation: When putting a new key-value pair:

- If the cache is empty, create a new node and set it as both the front and rear.
- If the key exists, update the value and move the node to the rear.
- If the key doesn't exist, add a new node to the rear and evict the front node if the capacity is exceeded.

III. Get Operation: When getting the value for a key:

- If the key exists, move the corresponding node to the rear.
- If the key doesn't exist, increment the cache miss count and return -1.

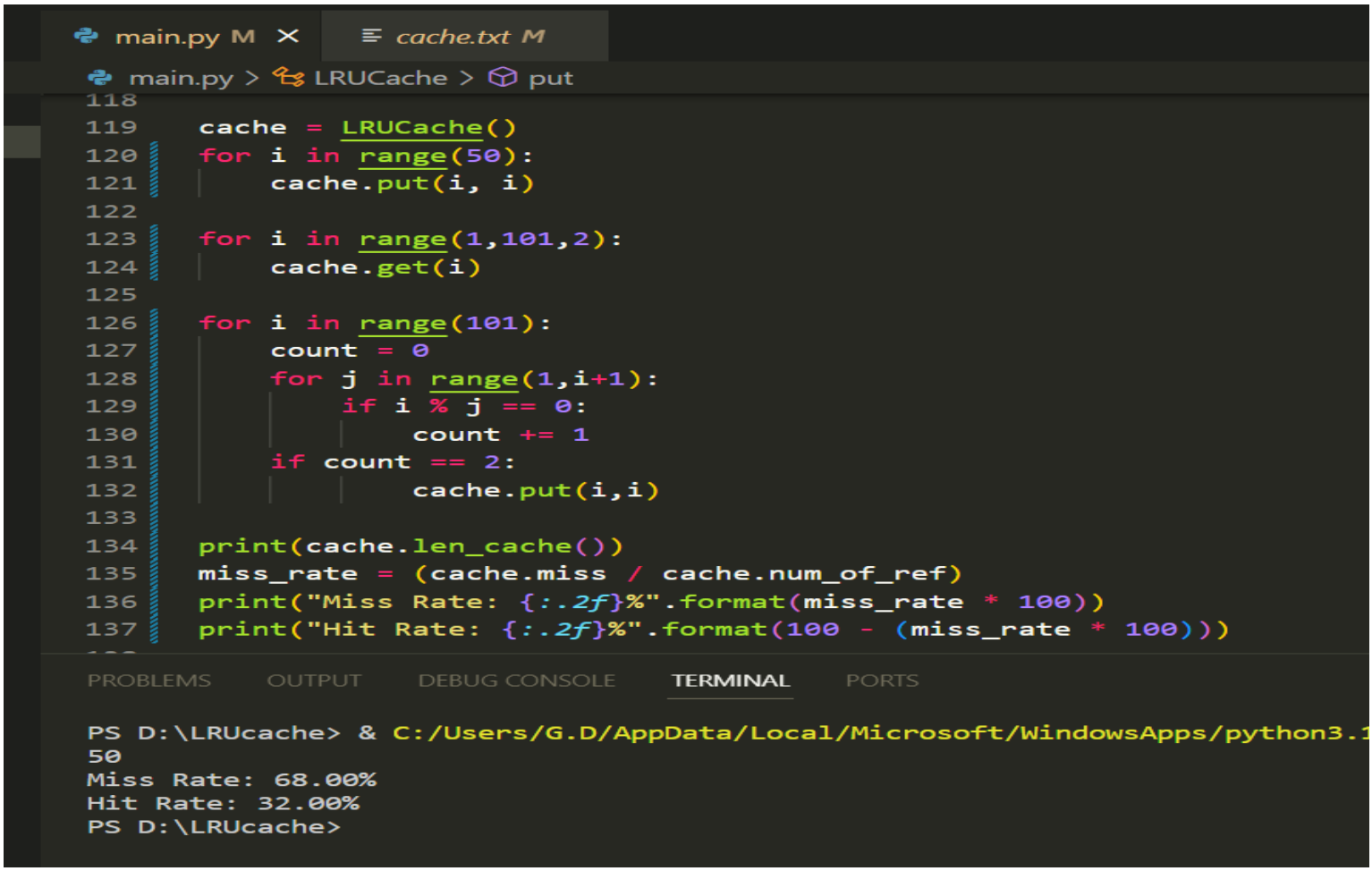
IV. Cache Eviction:

- Occurs when the cache capacity is reached.
- Evicts the front node, ensuring the cache size doesn't exceed the specified capacity. V. File Write Method:
- The file_write method writes the current state of the cache to a file after every modification.

Complexity:

Both space and time complexity will be O notation of n 0(n).

RESULTS:


main.py M
cache.txt M
main.py > LRUCache > put
118
119 cache = LRUCache()
120 for i in range(50):
121 cache.put(i, i)
122
123 for i in range(1,101,2):
124 cache.get(i)
125
126 for i in range(101):
127 count = 0
128 for j in range(1,i+1):
129 if i % j == 0:
130 count += 1
131 if count == 2:
132 cache.put(i,i)
133
134 print(cache.len_cache())
135 miss_rate = (cache.miss / cache.num_of_ref)
136 print("Miss Rate: {:.2f}%".format(miss_rate * 100))
137 print("Hit Rate: {:.2f}%".format(100 - (miss_rate * 100)))
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS D:\LRUcache> & C:/Users/G.D/AppData/Local/Microsoft/WindowsApps/python3.1
50
Miss Rate: 68.00%
Hit Rate: 32.00%
PS D:\LRUcache>

cache.txt

```
1  (22,22)
2  (24,24)
3  (26,26)
4  (28,28)
5  (30,30)
6  (32,32)
7  (34,34)
8  (36,36)
9  (38,38)
10 (40,40)
11 (42,42)
12 (44,44)
13 (46,46)
14 (48,48)
15 (1,1)
16 (9,9)
17 (15,15)
18 (21,21)
19 (25,25)
20 (27,27)
21 (33,33)
22 (35,35)
23 (39,39)
24 (45,45)
25 (49,49)
26 (2,2)
27 (3,3)
28 (5,5)
29 (7,7)
30 (11,11)
```

```
30 (11,11)
31 (13,13)
32 (17,17)
33 (19,19)
34 (23,23)
35 (29,29)
36 (31,31)
37 (37,37)
38 (41,41)
39 (43,43)
40 (47,47)
41 (53,53)
42 (59,59)
43 (61,61)
44 (67,67)
45 (71,71)
46 (73,73)
47 (79,79)
48 (83,83)
49 (89,89)
50 (97,97)
51
```