DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING
BACHELORS IN COMPUTER SYSTEMS ENGINEERING
Course Code: CS-218

Course Title: Data Structures & Algorithms

Complex Engineering Problem

SE Batch 2023, Fall Semester 2024
Grading Rubric
TERM PROJECT
Group Members:

Student No.	Name	Roll No.
S1	Muhammad Umer Ahmed	CS-23095
S2	Malik Hashir Shakeel	CS-23096
S3	Muhammad Hassin Saghir	CS-23129

CRITERIA AND SCALES				Marks Obtained		
				S1	S2	S3
Criterion 1: Has the student provided the appropriate design of LRU data structure?						
0	1	2	-			
The chosen design is too simple	The design is fit to be chosen for a class project	The choice is different and impressive.	-			
Criterion 2: How good is the programming implementation?						
0	1	2	3			
The project could not be implemented	The project has been implemented partially.	The project has been implemented completely but can be improved.	The project has been implemented completely and impressively			
Criterion 3: How well written is the report?						
0	1	2	-			
The submitted report is unfit to be graded	The report is partially acceptable	The report is complete and concise				

Problem Description

This class implements an **LRU (Least Recently Used) Cache**, which stores key-value pairs and automatically removes the least recently used items when the cache reaches its capacity. The implementation provides efficient operations through the use of well-chosen data structures.

Key Operations

1. **`add(self, node):=> O(1)`**
   - Adds a node just after the dummy head, marking it as the most recently used item.
2. **`put(self, key, value):=> O(1)`**
   - Adds or updates a key-value pair in the cache.
   - If adding the new entry exceeds the cache's capacity, the least recently used item is removed automatically.
3. **`resize(self, new_capacity):=> O(n)`**
   - Updates the cache's maximum capacity.
   - If the current number of items exceeds the new capacity, evicts the least recently used items.
4. **`miss_rate(self):=> O(1)`**
   - Calculates and returns the ratio of cache misses to total accesses.
   - If there are no accesses, returns `0.0`.
5. **`remove(self, node):=> O(1)`**
   - Removes a node from the doubly linked list by updating its neighbors to skip over it.

This implementation ensures high efficiency, with both **`get`** and **`put`** operations executed in **O(1)** time complexity.

Flow of the Project

To implement the LRU Cache, two primary data structures were combined:

1. Hashmap (Python Dictionary)

- A dictionary maps keys to their corresponding nodes in the doubly linked list.
- Enables instant lookup of nodes for efficient retrieval and updates.

2. Doubly Linked List

- Maintains the access order of cache items:
  - **Most Recently Used (MRU):** Items accessed or added recently are moved to the left end of the list.
  - **Least Recently Used (LRU):** Items that haven't been accessed for the longest time remain at the right end of the list.

How It Works

1. Inserting or Accessing Data

- When a key is accessed or a new key-value pair is inserted:
  - The `add()` function ensures the node is moved or added to the front of the linked list, marking it as the **most recently used**.

2. Evicting the Least Recently Used Item

- If the cache exceeds its capacity:
    - The `remove()` function removes the **rightmost node** (the least recently used item).
    - The corresponding key is also deleted from the hashmap.

3. Tracking Cache Efficiency

- **Cache Hits and Misses:**
    - Counters for cache hits (successful lookups) and cache misses (unsuccessful lookups) are maintained.
    - The `miss_rate()` function calculates the ratio of misses to total accesses, providing insights into the cache's performance.

Most Challenging Part

The most challenging aspect of this project was ensuring that the **hashmap** and the **doubly linked list** worked seamlessly together while maintaining optimal performance. Specific challenges included:

Key Challenges

- **Synchronizing the Two Data Structures**
  Ensuring that every time a key was added, updated, or removed in the hashmap, the doubly linked list reflected these changes accurately.
- **Handling Edge Cases**
  Managing scenarios like accessing a non-existent key or evicting the least recently used item when the cache reached its limit was occasionally tricky.
- **Managing Pointers in the Doubly Linked List**
  Ensuring the `prev` and `next` pointers of nodes were updated correctly, especially when adding or removing nodes at the boundaries (leftmost or rightmost).

Any New Concepts Learned

While working on this project, several new concepts and techniques in Python were explored:

1. Using Helper Functions

- Breaking down tasks into smaller helper functions like `add()` and `remove()` improved the readability and maintainability of the code.
- This approach enhanced my ability to organize and structure larger projects effectively.

2. Managing a Doubly Linked List

- Although familiar with standard lists, working with a doubly linked list was a new experience.
- Learned to use `prev` and `next` pointers to manage node connections and handle edge cases efficiently.

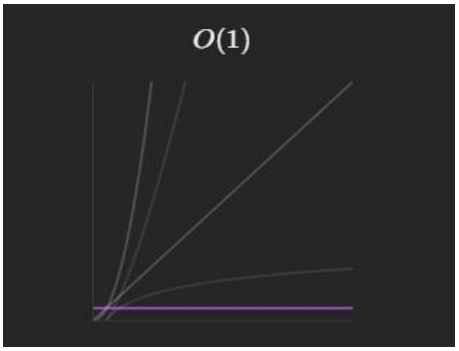
3. Efficient Use of Dictionaries

- This project demonstrated the power of dictionaries, not only for storing values but also for linking keys to more complex objects like nodes.

Class Time and Space Complexity

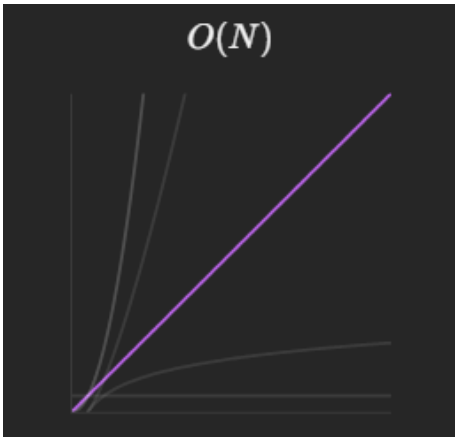
Time Complexity (Per Operation)

1. **Average Case: O(1)**
   - Most operations, including get and put, are O(1) due to the use of a dictionary for lookup and a doubly linked list for managing the order



2. **Worst Case (during resize): O(n)**

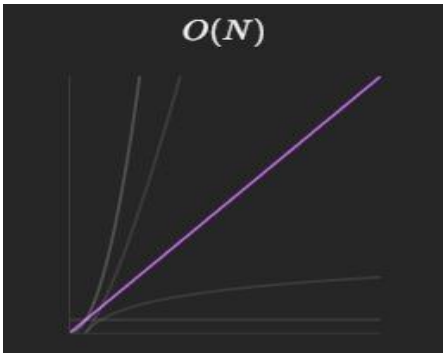
   - Resizing involves evicting multiple nodes.



Space Complexity:

Total: O(n)

- The dictionary _cache stores n key-node pairs.
- The doubly linked list contains n+2 nodes (including the dummy head and tail)



Test Case 1 (Miss Rate Calculation)

TestCase.py > ...

```python
from LRUCache import LRUCache

def test():
    cache = LRUCache(50)

    for i in range(50):
        cache.put(i, i)

    print("Retrieving values for odd keys:")
    for i in range(1, 100, 2):
        value = cache.get(i)
        print(f"Key: {i}, Value: {value}")

    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
    print("\nFilling the cache with prime numbers:")
    for prime in primes:
        cache.put(prime, prime)

    final_miss_rate = cache.miss_rate()
    print(f"\nAccesses: {cache._accesses}, Misses: {cache._misses}")
    print(f"\nFinal Miss Rate: {final_miss_rate:.2%}")

test()
```

OUTPUT

```
Accesses: 125, Misses: 85

Final Miss Rate: 68.00%
PS E:\LRU Cache>
```

Test Case 2 (Test some basic cache operations)

TestCase2.py > ...

```python
import unittest
from LRUCache import LRUCache

class TestLRUCacheBasic(unittest.TestCase):
    def setUp(self):
        self.cache = LRUCache(3)

    def test_basic_operations(self):
        self.cache.put(1, 90)
        self.cache.put(2, 80)
        self.cache.put(3, 70)
        self.assertEqual(self.cache.get(1), 90)
        self.assertEqual(self.cache.get(2), 80)

        # Test LRU eviction
        self.cache.put(4, 60)
        self.assertEqual(self.cache.get(3), -1)
        self.assertEqual(self.cache.get(4), 60)

        self.cache.put(2, 50)
        self.assertEqual(self.cache.get(2), 50)
        self.assertEqual(self.cache.get(1), 90)

if __name__ == "__main__":
    unittest.main()

```

OUTPUT

```
PS E:\LRU Cache> python -u "e:\LRU Cache\TestCase2.py"
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
PS E:\LRU Cache>
```

Test Case 3(Test advanced cache operations)

TestCase3.py > ...

```python
import unittest
from LRUCache import LRUCache

class TestLRUCacheAdvanced(unittest.TestCase):
    def setUp(self):
        self.cache = LRUCache(5)

    def test_advanced_features(self):
        for i in range(1, 6):
            self.cache.put(i, i * 10)

        # Ensure all items are accessible
        for i in range(1, 6):
            self.assertEqual(self.cache.get(i), i * 10)

        # Test resizing the cache
        self.cache.resize(3)
        self.assertEqual(len(self.cache._cache), 3)

        # Check if the least recently used items were evicted
        self.assertEqual(self.cache.get(1), -1)
        self.assertEqual(self.cache.get(2), -1)

        # Test invalid inputs
        with self.assertRaises(ValueError):
            self.cache.put(-1, 50)
        with self.assertRaises(ValueError):
            self.cache.put(1, -50)
        with self.assertRaises(ValueError):
            self.cache.resize(0)

if __name__ == "__main__":
    unittest.main()
```

OUTPUT

```
PS E:\LRU Cache> python -u "e:\LRU Cache\TestCase3.py"
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
PS E:\LRU Cache>
```

Distinguishing Features

The LRU Cache implementation stands out due to the following distinguishing features:

1. Dynamic Cache Resizing

- The cache supports dynamic resizing to accommodate varying storage needs.
- Users can adjust the cache capacity at runtime, ensuring flexibility without the need to restart or reinitialize the cache.
- The system automatically evicts the least recently used items if the current number of items exceeds the updated capacity.

2. Command-Line Interface (CLI)

- A user-friendly CLI interface enables seamless interaction with the cache.
- Users can easily perform operations such as adding, retrieving, or resizing the cache directly from the command line.
- Real-time feedback and error handling provide a smooth experience, even for non-technical users.

These features make the LRU Cache not only efficient but also highly adaptable and accessible.

Conclusion

The implementation of the LRU Cache provided an excellent opportunity to explore efficient data structure design and problem-solving techniques. By combining a hashmap for quick lookups and a doubly linked list for maintaining access order, the cache ensures that both `get` and `put` operations execute in O(1) time complexity. This efficient design is particularly well-suited for real-world applications where caching mechanisms play a critical role in improving performance.

Throughout this project, careful attention was given to synchronizing the two data structures, handling edge cases, and maintaining consistent cache behavior. These challenges fostered a deeper understanding of how to integrate multiple data structures seamlessly while ensuring reliability and efficiency.

In addition to honing technical skills, the project provided insights into breaking down complex tasks into manageable helper functions, understanding linked list operations, and leveraging Python's dictionaries for advanced use cases. The practical experience gained in implementing and optimizing the LRU Cache highlights the importance of well-structured code and efficient algorithms in software development.

Overall, this project not only fulfilled its functional requirements but also served as a valuable learning experience in algorithmic design and performance optimization.