DEPARTMENT OF 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	SHAMEEN GHYAS	CS002
S2	NABEEHA ASHAR	CS005
S3	MUTAHIR AHMED	CS032

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 completelybut 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				
Total Marks:						

Contents

DATA STRUCTURES AND ALGORITHMS PROJECT

CEP REPORT

TERM PROJECT TITLE:

IMPLEMENTATION OF LRU CACHE DATA STRUCTURE

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.

WORKING OF THE PROJECT:

The project implements an **LRU Cache** using a doubly linked list and dictionary for efficient operations.It extends the implementation with a **Graphical User Interface (GUI)** using tkinter for interactive usage.

1. CORE LRU CACHE IMPLEMENTATION:

DATA STRUCTURE:

1. The DLNode class represents a node in the doubly linked list with key, value, left, and right
pointers for navigation.
2. The LRU class manages the cache, maintaining an order of usage with the doubly linked list and a dictionary for fast key-value lookups.

KEY OPERATIONS:

1. **get(key):**
   a. Searches for the key in the cache.
   b. If the key exists, its node is moved to the front of the list (most recently used).
   c. If not, increments the misses count and returns -1.
2. **put(key, value):**
   a. Adds or updates a key-value pair. Updates move the node to the front, while new entries add a new node.
   b. If the cache exceeds its capacity, the least recently used node (at the tail) is removed.
   c. Increments misses for new entries and accesses for every operation. iii. **traverse():**
   d. Returns the current state of the cache as a list of key-value pairs, in order from most to least recently used.
3. **miss_rate():**
   a. Computes the percentage of cache misses relative to total accesses.

2. GRAPHICAL USER INTERFACE (GUI):

a. The GUI provides an interactive way to test and visualize the LRU cache.

b. **Input Fields and Buttons:**
   1. Key and Value entry fields allow users to specify operations.

   2. Buttons for Put and Get actions trigger respective cache methods.

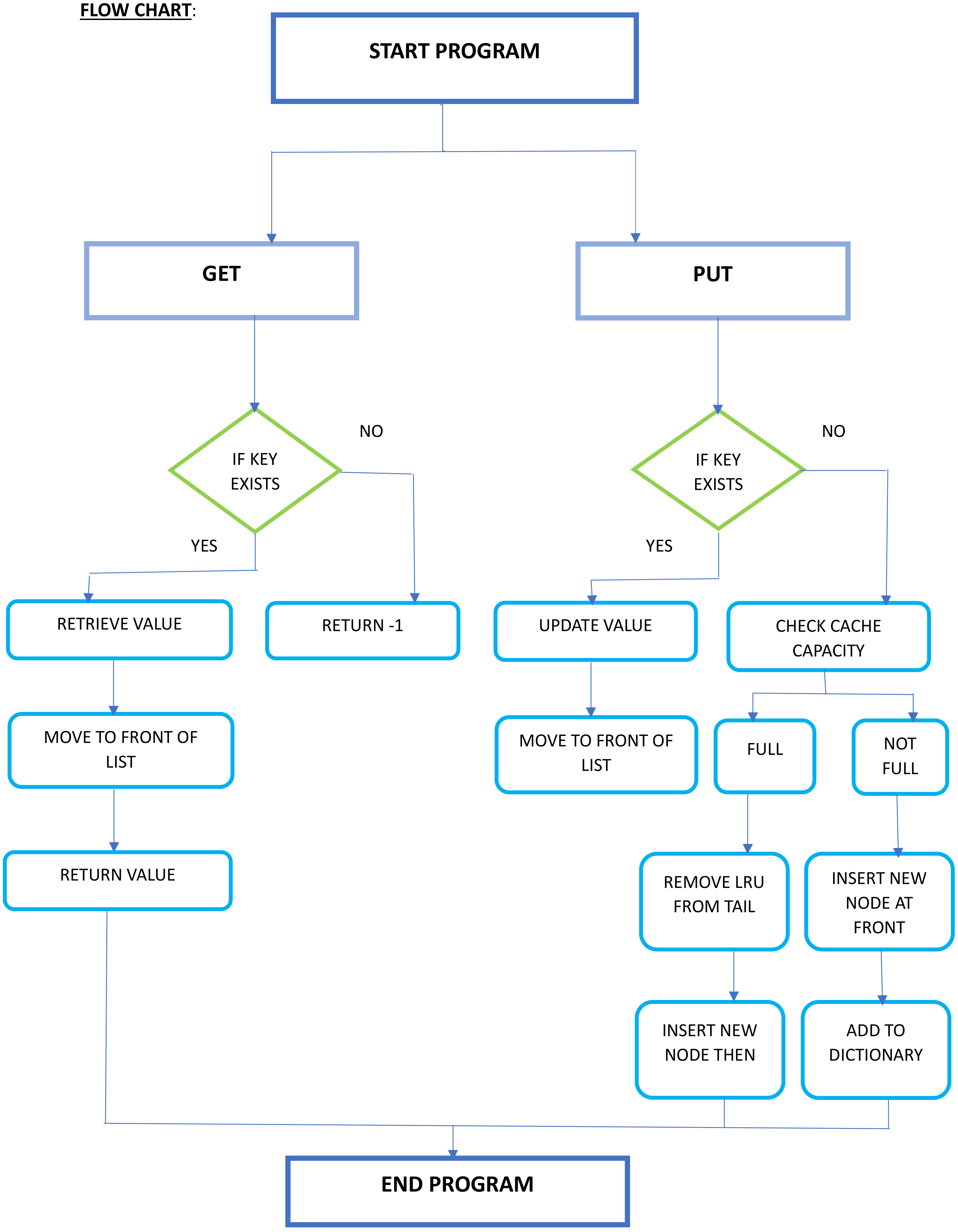
c. **Cache Visualization:**
   1. A canvas dynamically displays the cache's content. Each key-value pair is represented as a rectangle labeled key:value.

   2. Rectangles are displayed in order from most recently used (left) to least recently used (right).

d. **Real-Time Statistics:**
   1. The miss rate is displayed and updated after every operation.

e. **Error Handling:**
   1. Ensures only integers are accepted for keys and values, displaying appropriate error messages for invalid inputs.

FLOW CHART:


START PROGRAM
GET
PUT
IF KEY EXISTS
NO
YES
IF KEY EXISTS
NO
YES
RETRIEVE VALUE
RETURN -1
UPDATE VALUE
CHECK CACHE CAPACITY
MOVE TO FRONT OF LIST
MOVE TO FRONT OF LIST
FULL
NOT FULL
RETURN VALUE
REMOVE LRU FROM TAIL
INSERT NEW NODE AT FRONT
INSERT NEW NODE THEN
ADD TO DICTIONARY
END PROGRAM

COMLEXITY ANALYSIS:

TIME COMPLEXITY ANALYSIS:

1. **DLNODE CLASS**

- **insert(self, node)**:
  Insertion involves updating a constant number of pointers (p.right, q.left, etc.).
  Time Complexity: O(1)
- **delete(self)**:
  Deletion involves updating a constant number of pointers (p.right, r.left, etc.).
  Time Complexity: O(1)

2. **LRU CACHE CLASS:**

1. **Initialization (__init__)**:
   Creating two dummy nodes (head and tail), initializing the cache dictionary, and setting counters are constant-time operations.
   Time Complexity: O(1)
2. **get(self, key)**:
   Key lookup in the dictionary is **O(1)**.
   If the key exists:
   - Remove the node from its current position using delete() (**O(1)**).
   - Insert the node at the head using insert() (**O(1)**).

   If the key does not exist:
   - Increment misses (**O(1)**).

   Overall, the complexity is **O(1)**.

3. **put(self, key, value)**:
   Key lookup in the dictionary is **O(1)**.
   If the key exists:
   • Update the value and move the node to the head (**O(1)** for both).
   If the key does not exist:
   • Increment misses (**O(1)**).
   If the cache is full:
   • Remove the least recently used (LRU) node:
   • Access the node using self.tail.left (**O(1)**).
   • Remove it using delete() (**O(1)**).
   • Remove it from the dictionary (**O(1)**).
   • Insert a new node at the head using insert() (**O(1)**).
   • Add the node to the dictionary (**O(1)**).
   Overall, the complexity is **O(1)**.
4. **traverse(self)**:
   • Traverses through the doubly linked list, iterating through n nodes (where n is the number of items in the
   cache).
   • **Time Complexity: O(n)**
5. **miss_rate(self)**:
   Calculating the miss rate involves simple arithmetic operations on two counters.

- **Time Complexity: O(1)**

Both get and put have O(1) time complexity, making the implementation highly efficient for an LRU cache.

SPACE COMPLEXITY ANALYSIS:

1. DLNODE CLASS

Each node in the doubly linked list stores:

- A key with constant space.
- A value with constant space.
- A pointer to the next node takes constant space.
- A pointer to the previous node takes constant space.

Therefore, the space complexity for each DLNode is **O(1).**

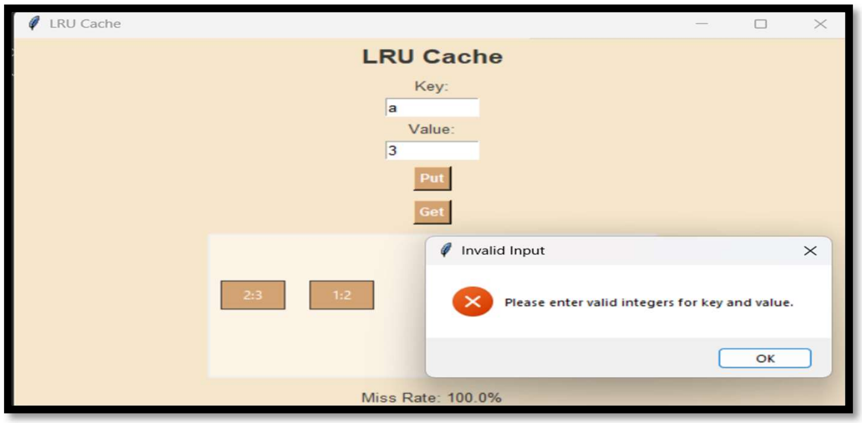
2. LRU CACHE CLASS:

- self.cache dictionary stores the key-value pairs. The number of entries in the cache is limited by the capacity of the LRU cache.
- If the capacity of the cache is **n** then the space used by the dictionary is **O(n).**

Hence, the overall space complexity is **O(n).**

DISTINGUISHING FEATURES OF OUR PROJECT:

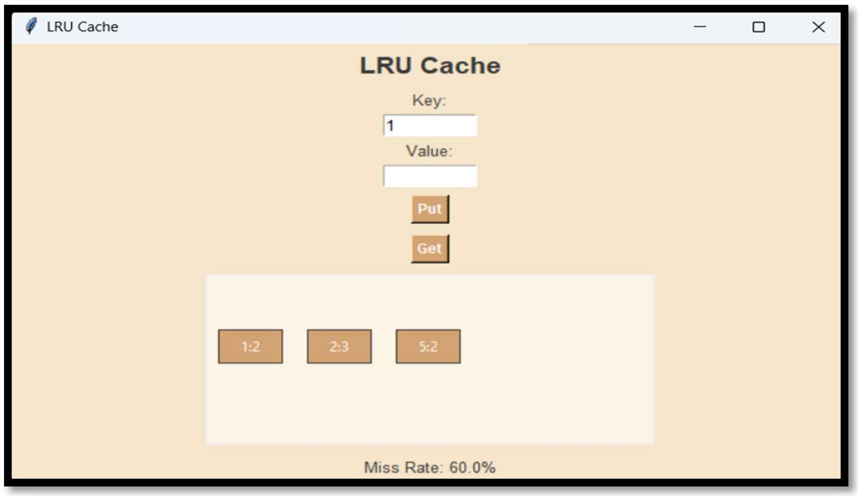
1. **Interactive GUI:**
   - Designed using tkinter, the GUI adds accessibility to the LRU cache by providing a visual and interactive representation of its operations.
   - Users can observe real-time updates in the cache state and miss rate statistics.
2. **Dynamic Cache Visualization:**
   The canvas visually represents the cache's content and order of usage. Users can see how operations like Put and Get affect the cache.
3. **Enhanced Usability:**
   The GUI simplifies experimentation with the LRU cache, making it suitable for demonstrations, teaching, and testing.
4. **Input validation feature:**
   A **message box** appears to alert the user if they enter a non-integer key or value, ensuring input validation and user-friendly error handling.



Please enter valid integers for key and value.

Miss Rate: 100.0%

5. **Miss Rate Tracking:**
Includes a dynamic miss rate tracker that updates in real-time.



MOST CHALLENGING PART:

The most challenging part of this project was managing the eviction of the least recently used (LRU) item while ensuring the cache operates efficiently with O(1) time complexity for both get and put operations. This was particularly difficult when using a combination of doubly linked lists and dictionaries to track the order of usage and maintain the cache's functionality.

NEW THINGS LEARNT IN PYTHON:

1. DATA STRUCTURES:
Learnt how to use data structures like **doubly linked lists** and **dictionaries** together to implement an efficient LRU Cache.

2. TIME AND SPACE COMPLEXITIES:
Gained a deeper understanding of **time** and **space complexity** to ensure the cache operates with **O(1)** time for both get and put operations.

3. EFFIECIENT CODE WRITING:
Focused on writing **efficient code** that handles eviction and updates the cache in constant time.

4. MISS RATE:
Learned how to track and update the **miss rate** in real-time to measure the cache's effectiveness.

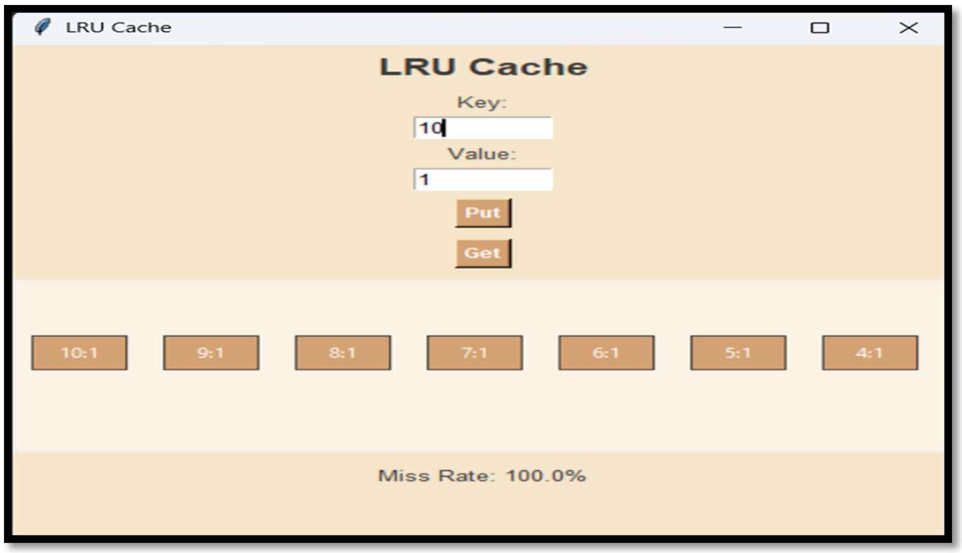
5. UNDERSTANDING LRU CACHE FUNCTIONALTIY:
Learnt how the LRU Cache works, especially how it removes the least recently used data and keeps track of the most recently used data.

TEST CASE RUNS SCREENSHOTS:

TEST CASE 1: PUT FUNCTION

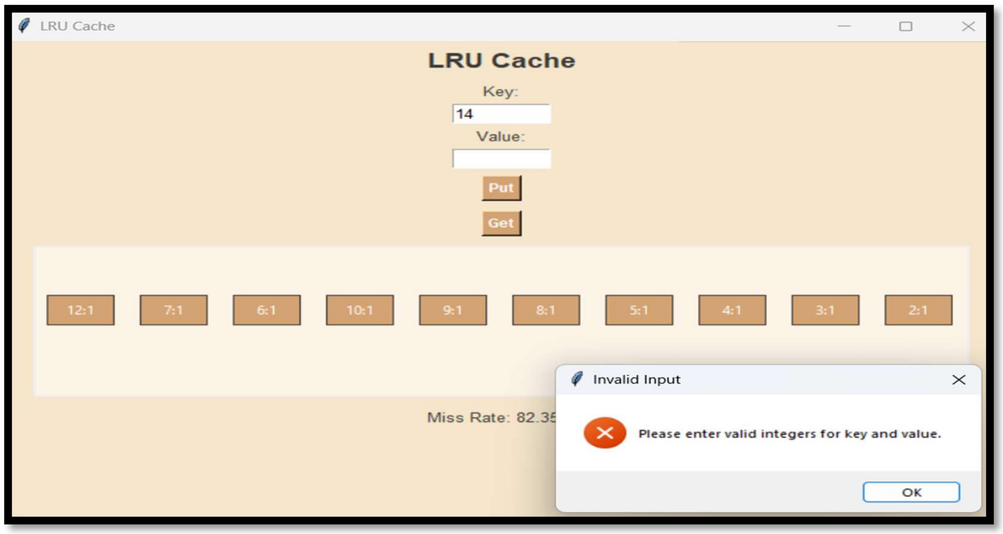
This test case demonstrates the correct functioning of the put function by adding key-value pairs to the cache. If key exists, it updates the existing key and if either key or value is not entered, it returns a message box. It verifies that new items are added, and the cache evicts the least recently used item when the capacity is exceeded.

FOR VALID KEY-VALUE PAIR:



Miss Rate: 100.0%

FOR INVALID KEY-VALUE PAIR:



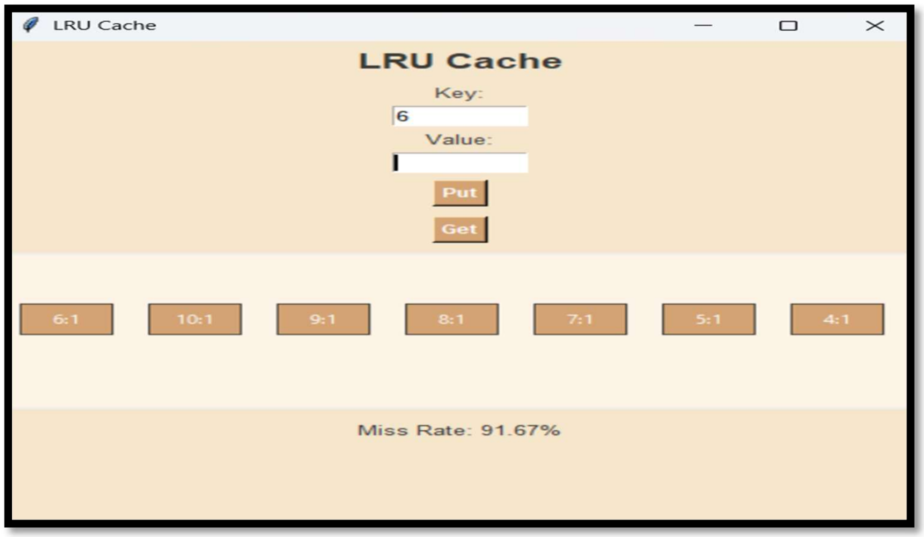
Miss Rate: 82.35

Please enter valid integers for key and value.

TEST CASE 3: CACHE UPDATE WITH ACCESS:

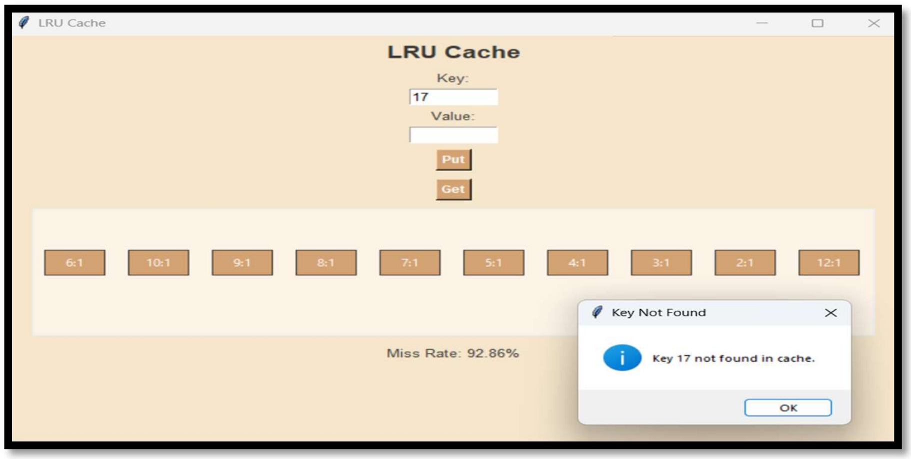
This test case checks the get function, ensuring it returns the correct value when a valid key is requested and a message box appears for non-existent keys. It verifies the cache's ability to retrieve stored data accurately.

FOR EXISTING KEYS:



Miss Rate: 91.67%

FOR NON-EXISTING KEYS:

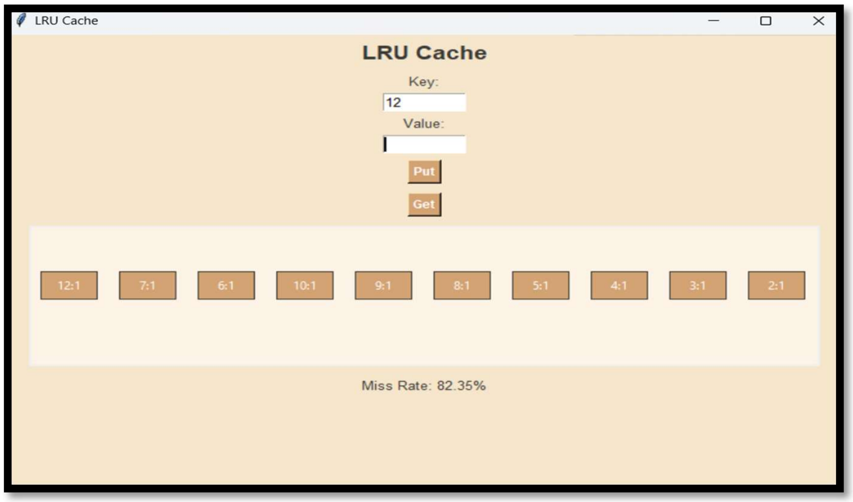


Miss Rate: 92.86%

Key 17 not found in cache.

TEST CASE 3: CACHE UPDATE WITH ACCESS:

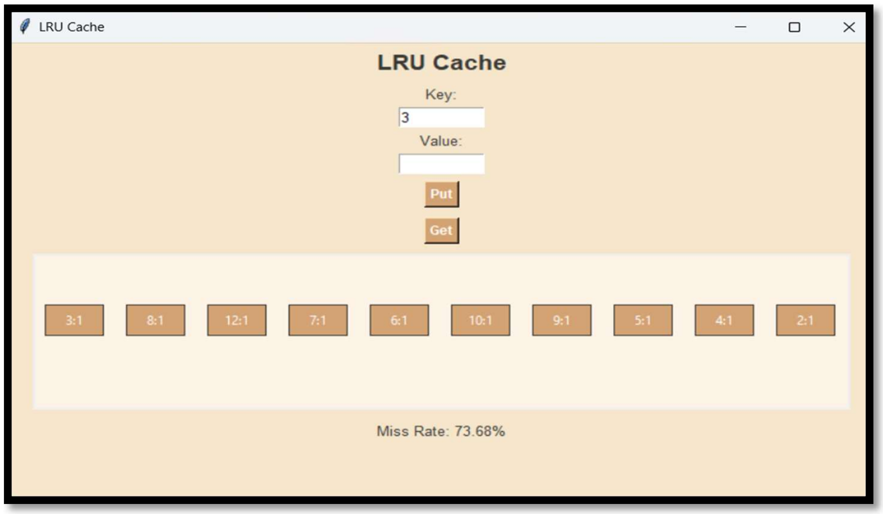
This test case demonstrates how the cache updates its contents based on key access patterns. It shows how the recently accessed keys are moved to the front, while the least recently used keys are evicted when the cache reaches its capacity, ensuring the correct implementation of the LRU eviction policy.



Miss Rate: 82.35%

TEST CASE 4: MISS RATE TRACKER

This test case checks the functionality of the miss rate tracker, ensuring that the miss rate is updated correctly as keys are accessed and evicted from the cache. It verifies that the tracker reflects the cache's performance over time.



Miss Rate: 73.68%

LIST OF REFERENCES:

1) Books for DSA using python provided by teacher.
2) Class Lectures
3) Internet