Practical no: 01

Write a Python program to manage the borrowing records of books in a library. Implement the following functionalities:

- Compute the average number of books borrowed by all library members.
- Find the book with the highest and lowest number of borrowings in the library.
- Count the number of members who have not borrowed any books (denoted by a borrow count of 0).
- Display the most frequently borrowed book (i.e., the mode of borrow counts)

Library Management System for Borrowing Records

Program :

```
class Library:
    def __init__(self):
        self.borrowing_records = {}
        self.book_borrow_counts = {}

    def add_borrowing_record(self, member_name, book_title):
        """Add a borrowing record for a member."""
        if member_name not in self.borrowing_records:
            self.borrowing_records[member_name] = []
        self.borrowing_records[member_name].append(book_title)
        if book_title not in self.book_borrow_counts:
            self.book_borrow_counts[book_title] = 0
        self.book_borrow_counts[book_title] += 1

    def compute_average_books_borrowed(self):
        """Compute the average number of books borrowed by all members."""
        total_books = sum(len(books) for books in self.borrowing_records.values())


        total_members = len(self.borrowing_records)
        if total_members == 0:
            return 0
        return total_books / total_members


    def find_highest_and_lowest_borrowed_books(self):
        """Find the book with the highest and lowest number of borrowings."""
```

```
        if not self.book_borrow_counts:
            return None, None
        highest_borrowed = max(self.book_borrow_counts, key=self.book_borrow_counts.get)
        lowest_borrowed = min(self.book_borrow_counts, key=self.book_borrow_counts.get)
        return highest_borrowed, lowest_borrowed


if __name__ == "__main__":
    library = Library()
    library.add_borrowing_record("Alice", "Book A")
    library.add_borrowing_record("Alice", "Book B")
    library.add_borrowing_record("Bob", "Book A")
    library.add_borrowing_record("Bob", "Book C")
    library.add_borrowing_record("Charlie", "Book A")
    library.add_borrowing_record("Charlie", "Book B")
    library.add_borrowing_record("Charlie", "Book C")
    library.add_borrowing_record("Charlie", "Book D")
    average_books = library.compute_average_books_borrowed()
    print(f"Average number of books borrowed by all members: {average_books:.2f}")
    highest, lowest = library.find_highest_and_lowest_borrowed_books()
    if highest and lowest:
        print(f"Book with the highest borrowings: {highest} ({library.book_borrow_counts[highest]}
times)")
        print(f"Book with the lowest borrowings: {lowest} ({library.book_borrow_counts[lowest]}
times)")
    else:
        print("No books have been borrowed yet.")
```

Output :

Average number of books borrowed by all members: 2.67

Book with the highest borrowings: Book A (3 times)

Book with the lowest borrowings: Book D (1 times)

Practical No: 02

Tittle:- Hash Table implementation using Division Method and Chaining.

Program :

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        """Division method hash function"""
        return key % self.size

    def insert(self, key, value):
        """Insert a key-value pair"""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                print(f"Updated key {key} with value {value}.")
                return

        self.table[index].append([key, value])
        print(f"Inserted ({key}, {value}) at index {index}.")

    def search(self, key):
        """Search for a value by key"""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                print(f"Found key {key} with value {pair[1]} at index {index}.")
                return pair[1]
        print(f"Key {key} not found.")
        return None

    def delete(self, key):
        """Delete a key-value pair"""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                self.table[index].remove(pair)
```

```
                print(f"Deleted key {key} from index {index}.")
                return
        print(f"Key {key} not found. Cannot delete.")

    def display(self):
        """Display the hash table"""
        print("\nHash Table:")
        for i, chain in enumerate(self.table):
            print(f"Index {i}: {chain}")
        print()


if __name__ == "__main__":
    ht = HashTable()

    ht.insert(10, "Apple")
    ht.insert(15, "Banana")
    ht.insert(20, "Cherry")
    ht.insert(25, "Mango")
    ht.display()

    ht.search(15)
    ht.search(99)

    ht.delete(20)
    ht.delete(99)
    ht.display()
```

Output :

Inserted (10, Apple) at index 0.

Inserted (15, Banana) at index 5.

Inserted (20, Cherry) at index 0.

Inserted (25, Mango) at index 5.

Hash Table:

Index 0: [[10, 'Apple'], [20, 'Cherry']]

Index 1: []

Index 2: []

Index 3: []

Index 4: []

Index 5: [[15, 'Banana'], [25, 'Mango']]

Index 6: []

Index 7: []

Index 8: []

Index 9: []

Found key 15 with value Banana at index 5.

Key 99 not found.

Deleted key 20 from index 0.

Key 99 not found. Cannot delete.

Hash Table:

Index 0: [[10, 'Apple']]

Index 1: []

Index 2: []

Index 3: []

Index 4: []

Index 5: [[15, 'Banana'], [25, 'Mango']]

Index 6: []

Index 7: []

Index 8: []

Index 9: []

Practical No 03:

Implementing a real-time undo/redo system for a text editing application using a Stack data structure. The system should support the following operations:

- Make a Change: A new change to the document is made.
- Undo Action: Revert the most recent change and store it for potential redo.
- Redo Action: Reapply the most recently undone action.
- Display Document State: Show the current state of the document after undoing or redoing an action.

Output :

```
class TextEditor:
    def __init__(self):
        self.document = []  # List to store each line of the document
        self.undo_stack = []  # Stack to store previous states for Undo
        self.redo_stack = []  # Stack to store undone states for Redo

    # Add a new line/text to the document
    def make_change(self, change):
        self.undo_stack.append(self.document.copy())  # Save current state for Undo
        self.document.append(change)  # Add new line/text
        self.redo_stack.clear()  # Clear Redo stack after a new change
        print(f"\nChange Made: '{change}'")

        # Undo the last change

    def undo(self):
        if not self.undo_stack:
            print("\nNothing to Undo!")
            return
        self.redo_stack.append(self.document.copy())  # Save current state for Redo
        self.document = self.undo_stack.pop()  # Revert to the previous state
        print("\nUndo performed.")

        # Redo the last undone change

    def redo(self):
        if not self.redo_stack:
            print("\nNothing to Redo!")
            return
```

```
        self.undo_stack.append(self.document.copy())  # Save current state for Undo
        self.document = self.redo_stack.pop()  # Reapply the undone change
        print("\nRedo performed.")

    # Display the current document


    def display_state(self):
        print("\n--- Current Document ---")
        if not self.document:
            print("[Empty Document]")
        else:
            for i in range(len(self.document)):  # loop over index numbers
                print(f"{i + 1}: {self.document[i]}")  # i+1 for line number, document[i] for text
    print("------------------------")


# ----------------- Menu Driven Code -----------------
editor = TextEditor()
while True:
    # Display menu
    print("\n--- Menu ---")
    print("1. Add Line/Text")
    print("2. Undo")
    print("3. Redo")
    print("4. Show Document")
    print("5. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        text = input("Enter line/text to add: ")
        editor.make_change(text)  # Add new line/text
    elif choice == '2':
        editor.undo()  # Undo last change
    elif choice == '3':
        editor.redo()  # Redo last undone change
    elif choice == '4':
        editor.display_state()  # Display current document
    elif choice == '5':
        print("Exiting...")
        break
    else:
        print("Invalid choice! Try again.")
```

Output :

--- Menu ---

1. Add Line/Text

2. Undo

3. Redo

4. Show Document

5. Exit

Enter your choice: 1

Enter line/text to add: shardul

Change Made: 'shardul'

--- Menu ---

1. Add Line/Text

2. Undo

3. Redo

4. Show Document

5. Exit

Enter your choice:

Practical No : 04

Title: Perform the Queue Operation

Program :

```
from collections import deque
class EventProcessingSystem:
    def __init__(self):
        self.event_queue = deque()

    def add_event(self, event):
        """Add a new event to the queue"""
        self.event_queue.append(event)
        print(f"Event '{event}' added to queue.")

    def process_next_event(self):
        """Process and remove the oldest event"""
        if self.event_queue:
            event = self.event_queue.popleft()
            print(f"Processing event: {event}")
        else:
            print("No events to process.")

    def display_pending_events(self):
        """Display all events currently in queue"""
        if self.event_queue:
            print("Pending Events:", list(self.event_queue))
        else:
            print("No pending events.")

    def cancel_event(self, event):
        """Cancel a specific event if not yet processed"""
        if event in self.event_queue:
            self.event_queue.remove(event)
            print(f"Event '{event}' canceled.")
        else:
            print(f"Event '{event}' not found or already processed.")

# Example Usage
if __name__ == "__main__":
```

```
system = EventProcessingSystem()
system.add_event("Login Request")
system.add_event("File Upload")
system.add_event("Database Backup")
system.display_pending_events()
system.cancel_event("File Upload")
system.display_pending_events()
system.process_next_event()
system.display_pending_events()
```

Output :

Event 'Login Request' added to queue.

Event 'File Upload' added to queue.

Event 'Database Backup' added to queue.

Pending Events: ['Login Request', 'File Upload', 'Database Backup']

Event 'File Upload' canceled.

Pending Events: ['Login Request', 'Database Backup']

Processing event: Login Request

Pending Events: ['Database Backup']

Practical No 05

Program :

```
class StudentNode:
    def __init__(self, roll, name, marks):
        self.roll = roll
        self.name = name
        self.marks = marks
        self.next = None
        self.prev = None


class StudentRecordManagement:
    def __init__(self):
        self.head = None

    def add_student(self, roll, name, marks):
        new_node = StudentNode(roll, name, marks)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
            new_node.prev = temp
        print(f" Student {name} added successfully.")

    def delete_student(self, roll):
        temp = self.head
        while temp:
            if temp.roll == roll:
                if temp.prev:
                    temp.prev.next = temp.next
                if temp.next:
                    temp.next.prev = temp.prev
                if temp == self.head:
                    self.head = temp.next
                print(f" Student with Roll {roll} deleted.")
                return
            temp = temp.next
        print(" Student not found.")
```

```
def update_student(self, roll, name=None, marks=None):
    temp = self.head
    while temp:
        if temp.roll == roll:
            if name:
                temp.name = name
            if marks is not None:
                temp.marks = marks
            print(f" Student with Roll {roll} updated.")
            return
        temp = temp.next
    print(" Student not found.")

def search_student(self, roll):
    temp = self.head
    while temp:
        if temp.roll == roll:
            print(f"Found -> Roll: {temp.roll}, Name: {temp.name}, Marks: {temp.marks}")
            return temp
        temp = temp.next
    print(" Student not found.")
    return None
def display(self):
    if not self.head:
        print(" No student records available.")
        return
    temp = self.head
    print("\n Student Records:")
    print("RollNo\tName\tMarks")
    while temp:
        print(f"{temp.roll}\t{temp.name}\t{temp.marks}")
        temp = temp.next

def sort_records(self, key="roll", order="asc"):
    if not self.head or not self.head.next:
        return
    swapped = True
    while swapped:
        swapped = False
        temp = self.head
        while temp.next:
            if key == "roll":
                cond = (temp.roll > temp.next.roll)
```

```
            else:
                cond = (temp.marks > temp.next.marks)

            if order == "desc":
                cond = not cond

            if cond:
                temp.roll, temp.next.roll = temp.next.roll, temp.roll
                temp.name, temp.next.name = temp.next.name, temp.name
                temp.marks, temp.next.marks = temp.next.marks, temp.marks
                swapped = True
            temp = temp.next
        print(f" Records sorted by {key} ({order}).")

if __name__ == "__main__":
    system = StudentRecordManagement()

    system.add_student(1, "alex", 85)
    system.add_student(3, "Banty", 75)
    system.add_student(2, "komal", 90)

    system.display()

    system.search_student(2)

    system.update_student(3, marks=80)

    system.delete_student(1)

    system.sort_records(key="marks", order="desc")
    system.display()
```

Output :

Student alex added successfully.

Student Banty added successfully.

Student komal added successfully.

Student Records:

RollNo Name Marks

1 alex 85

3 Banty 75

2 komal 90

Found -> Roll: 2, Name: komal, Marks: 90

Student with Roll 3 updated.

 Student with Roll 1 deleted.

Records sorted by marks (desc).

Student Records:

RollNo Name Marks

2 komal 90

3 Banty 80

Practical No 06

Hash Table implementation using Division Method and Chaining

Program :

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # list of lists for chaining

    def hash_function(self, key):
        """Division method hash function"""
        return key % self.size

    def insert(self, key, value):
        """Insert a key-value pair"""
        index = self.hash_function(key)
        # Check if key already exists; if yes, update it
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                print(f"Updated key {key} with value {value}.")
                return
        # Otherwise, append a new pair
        self.table[index].append([key, value])
        print(f"Inserted ({key}, {value}) at index {index}.")

    def search(self, key):
        """Search for a value by key"""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                print(f"Found key {key} with value {pair[1]} at index {index}.")
                return pair[1]
        print(f"Key {key} not found.")
        return None

    def delete(self, key):
        """Delete a key-value pair"""
        index = self.hash_function(key)
        for pair in self.table[index]:
```

```
            if pair[0] == key:
                self.table[index].remove(pair)
                print(f"Deleted key {key} from index {index}.")
                return
        print(f"Key {key} not found. Cannot delete.")

    def display(self):
        """Display the hash table"""
        print("\nHash Table:")
        for i, chain in enumerate(self.table):
            print(f"Index {i}: {chain}")
        print()

# --- Example Usage ---
if __name__ == "__main__":
    ht = HashTable()

    ht.insert(10, "Apple")
    ht.insert(20, "Banana")
    ht.insert(15, "Cherry")
    ht.insert(25, "Mango")   # collision with 15 (same index)
    ht.display()

    ht.search(15)
    ht.search(99)

    ht.delete(20)
    ht.delete(99)
    ht.display()
```

Output :

Inserted (10, Apple) at index 0.

Inserted (20, Banana) at index 0.

Inserted (15, Cherry) at index 5.

Inserted (25, Mango) at index 5.

Hash Table:

Index 0: [[10, 'Apple'], [20, 'Banana']]

Index 1: []

Index 2: []

Index 3: []

Index 4: []

Index 5: [[15, 'Cherry'], [25, 'Mango']]

Index 6: []

Index 7: []

Index 8: []

Index 9: []

Found key 15 with value Cherry at index 5.

Key 99 not found.

Deleted key 20 from index 0.

Key 99 not found. Cannot delete.

Hash Table:

Index 0: [[10, 'Apple']]

Index 1: []

Index 2: []

Index 3: []

Index 4: []

Index 5: [[15, 'Cherry'], [25, 'Mango']]

Index 6: []

Index 7: []

Index 8: []

Index 9: []