Assignment 14 (Queue, Priority Queue, Doubly Linked List)

> **Name: Mohammed Varaliya**

> **Roll No: 54**

Questions

1. **Queue using LinkedList**
2. **Priority Queue [insertion & deletion of a node]**
3. **Doubly Linked List [insertion & deletion of a node]**

---

1. **Queue using LinkedList**
    a. **Code:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
```

```
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            return "Queue is empty"
        temp = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return temp.data

    def peek(self):
        if self.is_empty():
            return "Queue is empty"
        return self.front.data

    def display(self):
        if self.is_empty():
            print("Queue is empty")
            return
        temp = self.front
        while temp:
            print(temp.data, end=" ")
            temp = temp.next
        print()
```

```
if __name__ == "__main__":

    queue = Queue()
    queue.enqueue(10)
    queue.enqueue(20)
    queue.enqueue(30)
    queue.display()

    print(queue.dequeue())
    queue.display()
```

1. **Explanation:**
    a. **Node Class**: Represents a node of the queue. It holds the data and a reference to the next node.
    b. **Queue Class**: Implements the queue using linked list. It has methods:
        i. `is_empty()`: checks if the queue is empty.
        ii. `enqueue(data)`: Adds an element to the rear of the queue.
        iii. `dequeue()`: Removes an element from the front of the queue.
        iv. `peek()`: Returns the element at the front without removing it.
        v. `display()`: Displays the queue elements.

b. **Output:**

```
10 20 30
10
20 30
```

---

2. **Priority Queue [insertion & deletion of a node]**
    a. **Code:**

```
import heapq
```

```
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def insert(self, data, priority):
        heapq.heappush(self.heap, (priority, data))

    def delete(self):
        if not self.is_empty():
            return heapq.heappop(self.heap)[1]
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.heap[0][1]
        else:
            return "Queue is empty"

    def is_empty(self):
        return len(self.heap) == 0

    def display(self):
        if self.is_empty():
            print("Priority Queue is empty")
        else:
            for priority, data in self.heap:
                print(f"Priority: {priority}, Data:
{data}")

if __name__ == "__main__":

    pq = PriorityQueue()
    pq.insert("Task1", 2)
    pq.insert("Task2", 1)
    pq.insert("Task3", 3)
    pq.display()
```

```
    print(pq.delete())
    pq.display()
```

1. **Explanation:**
    a. **PriorityQueue Class**: Implements the priority queue using Python's `heapq` module, which is a binary heap and provides efficient insertion and deletion.
        i. `insert(data, priority)`: Adds an element with its priority to the heap.
        ii. `delete()`: Removes and returns the element with the highest priority (lowest priority number).
        iii. `peek()`: Returns the highest priority element without removing it.
        iv. `is_empty()`: Checks if the queue is empty.
        v. `display()`: Prints the elements in the queue.
    b. `heapq` automatically maintains the heap property, where the smallest element (based on priority) is at the root.

b. **Output:**

```
Priority: 1, Data: Task2
Priority: 2, Data: Task1
Priority: 3, Data: Task3
Task2
Priority: 2, Data: Task1
Priority: 3, Data: Task3
```

---

3. **Doubly Linked List [insertion & deletion of a node]**

a. **Code:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
```

```
        self.next = None

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

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        new_node.prev = last_node

    def delete_node(self, key):
        if self.head is None:
            return "List is empty"
        current_node = self.head
        while current_node:
            if current_node.data == key:
                if current_node.prev:
                    current_node.prev.next = current_
node.next
                if current_node.next:
                    current_node.next.prev = current_
node.prev
                if current_node == self.head:
                    self.head = current_node.next
                return
            current_node = current_node.next
        return "Node not found"

    def display(self):
        if self.head is None:
            print("List is empty")
```

```
            return
        current_node = self.head
        while current_node:
            print(current_node.data, end=" <-> ")
            current_node = current_node.next
        print("None")

if __name__ == "__main__":

    dll = DoublyLinkedList()
    dll.insert_at_end(10)
    dll.insert_at_end(20)
    dll.insert_at_end(30)
    dll.display()

    dll.delete_node(20)
    dll.display()
```

1. **Explanation:**
    a. **Node Class**: Each node has a `data` field and two pointers: `prev` (points to the previous node) and `next` (points to the next node).
    b. **DoublyLinkedList Class**: Implements the doubly linked list with the following methods:The doubly linked list allows for easier deletion since each node knows its predecessor.
        i. `insert_at_end(data)`: Inserts a new node with the specified data at the end of the list.
        ii. `delete_node(key)`: Deletes the node with the specified data.
        iii. `display()`: Displays the nodes in the list, showing the links between them.

b. **Output:**

```
10 <-> 20 <-> 30 <-> None
10 <-> 30 <-> None
```