Ex. No: 10 24-06-2023	UIT2201 — Programming and Data Structures

Aim:

To execute the following programs and note the output.

PART – A

1. Design and implement Linked List with the following operations. • Isempty • Display • Find • append • Insert by pos • Delete by pos • Insert by specifying previous value. • Delete by specifying previous value.

Code:

```
class ElementNotFoundError(Exception):
    pass


class Node:
    __slots__ = ["item", "next"]

    def __init__(self, item=None, next=None):
        """
        Initialize a Node with an item and next pointer.

        Args:
            item: The item to be stored in the node.
            next: The reference to the next node.
        """
        self.item = item
        self.next = next


class SinglyLinkedList:
    def __init__(self):
        """
        Initialize an empty Singly Linked List.
        """
        self.head = self.tail = Node()
        self.size = 0

    def is_empty(self):
        """
        Check if the linked list is empty.

        Returns:
            True if the linked list is empty, False otherwise.
        """
        return self.head == self.tail

    def display(self):
        """
        Display the items in the linked list.
        """
        if self.is_empty():
```

```
            print("LinkedList is empty.")
        else:
            current = self.head.next
            while current is not None:
                print(current.item, end=" ")
                current = current.next
            print()

    def find(self, item):
        """
        Find the index of a given item in the linked list.

        Args:
            item: The item to be searched in the linked list.

        Returns:
            The index of the item if found, -1 otherwise.
        """
        current = self.head.next
        index = 0
        while current is not None:
            if current.item == item:
                return index
            current = current.next
            index += 1
        return -1

    def append(self, item):
        """
        Append a new node with the specified item to the end of the linked
list.

        Args:
            item: The item to be appended.
        """
        new_node = Node(item)
        self.tail.next = new_node
        self.tail = new_node
        self.size += 1

    def insert(self, pos, item):
        """
        Insert a new node with the specified item at the given position in
the linked list.

        Args:
            pos: The position at which the item should be inserted.
            item: The item to be inserted.

        Raises:
            IndexError: If the position is out of range.
        """
        if pos < 0 or pos > self.size:
            raise IndexError("Index out of range")

        if pos == self.size:
            self.append(item)
            return

        current = self.head
        for _ in range(pos):
```

```
            current = current.next

        new_node = Node(item, current.next)
        current.next = new_node
        self.size += 1

    def delete(self, pos):
        """
        Delete the node at the specified position in the linked list.

        Args:
            pos: The position of the node to be deleted.

        Raises:
            IndexError: If the position is out of range.
        """
        if pos < 0 or pos >= self.size:
            raise IndexError("Index out of range")

        current = self.head
        for _ in range(pos):
            current = current.next

        del_node = current.next
        current.next = del_node.next
        if current.next is None:
            self.tail = current
        del del_node
        self.size -= 1

    def insert_by_value(self, prev_value, item):
        """
        Insert a new node with the specified item after the node containing
the previous value.

        Args:
            prev_value: The previous value after which the new node should
be inserted.
            item: The item to be inserted.

        Raises:
            ElementNotFoundError: If the previous value is not found in the
linked list.
        """
        prev = self.find_prev(prev_value)
        if prev is None:
            raise ElementNotFoundError("Previous value not found")

        new_node = Node(item, prev.next)
        prev.next = new_node
        if prev == self.tail:
            self.tail = new_node
        self.size += 1

    def delete_by_value(self, prev_value):
        """
        Delete the node following the node containing the previous value.

        Args:
            prev_value: The previous value whose next node should be
deleted.
```

```
        Raises:
            ElementNotFoundError: If the previous value is not found in the
linked list.
        """
        prev = self.find_prev(prev_value)
        if prev is None or prev.next is None:
            raise ElementNotFoundError("Previous value not found")

        del_node = prev.next
        prev.next = del_node.next
        if prev.next is None:
            self.tail = prev
        del del_node
        self.size -= 1

    def find_prev(self, item):
        """
        Find the node preceding the node containing the specified item.

        Args:
            item: The item whose preceding node should be found.

        Returns:
            The preceding node if found, None otherwise.
        """
        current = self.head
        while current.next is not None:
            if current.next.item == item:
                return current
            current = current.next
        return None


if __name__ == "__main__":
    # Create a new SinglyLinkedList
    linked_list = SinglyLinkedList()

    # Append elements to the linked list
    linked_list.append(10)
    linked_list.append(20)
    linked_list.append(30)

    # Display the linked list
    print("Linked List:")
    linked_list.display()  # Output: 10 20 30

    # Find an item in the linked list
    index = linked_list.find(20)
    if index != -1:
        print("Found at index:", index)  # Output: Found at index: 1
    else:
        print("Item not found")

    # Insert an item at a specific position
    linked_list.insert(1, 15)
    print("After Insertion:")
    linked_list.display()  # Output: 10 15 20 30

    # Delete an item at a specific position
    linked_list.delete(2)
```

```
    print("After Deletion:")
    linked_list.display()  # Output: 10 15 30

    # Insert an item by specifying the previous value
    try:
        linked_list.insert_by_value(15, 25)
        print("After Insertion by Value:")
        linked_list.display()  # Output: 10 15 25 30
    except ElementNotFoundError:
        print("Previous value not found")

    # Delete an item by specifying the previous value
    try:
        linked_list.delete_by_value(15)
        print("After Deletion by Value:")
        linked_list.display()  # Output: 10 25 30
    except ElementNotFoundError:
        print("Previous value not found")
```

Inputs and Output:

```
Linked List:
10 20 30
Found at index: 1
After Insertion:
10 15 20 30
After Deletion:
10 15 30
After Insertion by Value:
10 25 15 30
After Deletion by Value:
10 25 30
```

2. Implement Linked Stack and Linked Queue

Code:

```
class EmptyStackError(Exception):
    """
    Exception raised when an operation is performed on an empty stack.
    """
    pass


class Node:
    __slots__ = ["item", "next"]
```

```
    def __init__(self, item=None, next=None):
        """
        Initialize a Node with an item and next pointer.

        Args:
            item: The item to be stored in the node.
            next: The reference to the next node.
        """
        self.item = item
        self.next = next


class LinkedStack:
    def __init__(self):
        """
        Initialize an empty stack.
        """
        self.top = None
        self._size = 0

    def is_empty(self):
        """
        Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return self.top is None

    def push(self, item):
        """
        Push an item onto the stack.

        Args:
            item: The item to be pushed onto the stack.
        """
        new_node = Node(item)
        new_node.next = self.top
        self.top = new_node
        self._size += 1

    def pop(self):
        """
        Pop an item from the stack.

        Returns:
            The item that is popped from the stack.

        Raises:
            EmptyStackError: If the stack is empty.
        """
        if self.is_empty():
            raise EmptyStackError("Stack is empty")

        popped_item = self.top.item
        self.top = self.top.next
        self._size -= 1

        return popped_item
```

```
    def peek(self):
        """
        Return the top item of the stack without removing it.

        Returns:
            The top item of the stack.

        Raises:
            EmptyStackError: If the stack is empty.
        """
        if self.is_empty():
            raise EmptyStackError("Stack is empty")

        return self.top.item

    def __len__(self):
        """
        Return the number of items in the stack.

        Returns:
            The number of items in the stack.
        """
        return self._size

    def __getitem__(self, index):
        """
        Get the item at the specified index.

        Args:
            index: The index of the item to retrieve.

        Returns:
            The item at the specified index.

        Raises:
            IndexError: If the index is out of range.
        """
        if index < 0 or index >= self._size:
            raise IndexError("Index out of range")

        current = self.top
        for _ in range(index):
            current = current.next

        return current.item

    def __str__(self):
        """
        Return a string representation of the stack.

        Returns:
            A string representation of the stack.
        """
        if self.is_empty():
            return "Stack: []"

        stack_items = []
        current = self.top
        while current is not None:
            stack_items.append(str(current.item))
            current = current.next
```

```
        return "Stack: [" + ", ".join(stack_items) + "]"

if __name__ == "__main__":
    stack = LinkedStack()

    print("Is the stack empty?", stack.is_empty())  # Output: True

    stack.push(10)
    stack.push(20)
    stack.push(30)

    print("Is the stack empty?", stack.is_empty())  # Output: False

    print("Top item of the stack:", stack.peek())  # Output: 30

    item = stack.pop()
    print("Popped item:", item)  # Output: 30

    print("Length of the stack:", len(stack))  # Output: 2

    print("Item at index 0:", stack[0])  # Output: 20
    print("Item at index 1:", stack[1])  # Output: 10

    print(stack)  # Output: Stack: [20, 10]
```

Output:

```
Is the stack empty? True
Is the stack empty? False
Top item of the stack: 30
Popped item: 30
Length of the stack: 2
Item at index 0: 20
Item at index 1: 10
Stack: [20, 10]
```

Code:

```
class EmptyQueueError(Exception):
    """
    Exception raised when an operation is performed on an empty queue.
    """
    pass


class Node:
    __slots__ = ["item", "next"]
```

```
    def __init__(self, item=None, next=None):
        """
        Initialize a Node with an item and next pointer.

        Args:
            item: The item to be stored in the node.
            next: The reference to the next node.
        """
        self.item = item
        self.next = next


class LinkedQueue:
    def __init__(self):
        """
        Initialize an empty queue.
        """
        self.front = None
        self.rear = None
        self._size = 0

    def is_empty(self):
        """
        Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
        """
        return self.front is None

    def enqueue(self, item):
        """
        Add an item to the rear of the queue.

        Args:
            item: The item to be added to the queue.
        """
        new_node = Node(item)
        if self.is_empty():
            self.front = new_node
            self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
        self._size += 1

    def dequeue(self):
        """
        Remove and return the item from the front of the queue.

        Returns:
            The item removed from the front of the queue.

        Raises:
            EmptyQueueError: If the queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError("Queue is empty")

        removed_item = self.front.item
```

```
        self.front = self.front.next
        self._size -= 1

        if self.front is None:
            self.rear = None

        return removed_item

    def peek(self):
        """
        Return the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue.

        Raises:
            EmptyQueueError: If the queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError("Queue is empty")

        return self.front.item

    def __len__(self):
        """
        Return the number of items in the queue.

        Returns:
            The number of items in the queue.
        """
        return self._size

    def __str__(self):
        """
        Return a string representation of the queue.

        Returns:
            A string representation of the queue.
        """
        if self.is_empty():
            return "Queue: []"

        queue_items = []
        current = self.front
        while current is not None:
            queue_items.append(str(current.item))
            current = current.next

        return "Queue: [" + ", ".join(queue_items) + "]"


if __name__ == "__main__":
    queue = LinkedQueue()

    print("Is the queue empty?", queue.is_empty())  # Output: True

    queue.enqueue(10)
    queue.enqueue(20)
    queue.enqueue(30)

    print("Is the queue empty?", queue.is_empty())  # Output: False
```

```
print("Front item of the queue:", queue.peek())  # Output: 10

item = queue.dequeue()
print("Dequeued item:", item)  # Output: 10

print("Length of the queue:", len(queue))  # Output: 2

print(queue)  # Output: Queue: [20, 30]
```

Output:

```
Is the queue empty? True
Is the queue empty? False
Front item of the queue: 10
Dequeued item: 10
Length of the queue: 2
Queue: [20, 30]
```

==========