UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 7

Doubly Linked Lists

Submitted by:
Enverzo, Kyle Andrey D.

Instructor:
Engr. Maria Rizette H. Sayo

August, 23, 2025

I. Objectives

Introduction

A doubly linked list is a type of linked list data structure where each node contains three components:

Data - The actual value stored in the node
Previous pointer - A reference to the previous node in the sequence
Next pointer - A reference to the next node in the sequence.

This laboratory activity aims to implement the principles and techniques in:

- Writing algorithms using Linked list
- Writing a python program that will perform the common operations in a Doubly linked list
- A doubly linked list is particularly useful when you need frequent bidirectional traversal or easy deletion of nodes from both ends of the list.

II. Methods

- Using Google Colab, type the source codes below:

```
class Node:
    """Node class for doubly linked list"""
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None


class DoublyLinkedList:
    """Doubly Linked List implementation"""


    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0


    def is_empty(self):
        """Check if the list is empty"""
        return self.head is None


    def get_size(self):
        """Get the size of the list"""
```

```
    return self.size

def display_forward(self):
    """Display the list from head to tail"""
    if self.is_empty():
        print("List is empty")
        return

    current = self.head
    print("Forward: ", end="")
    while current:
        print(current.data, end="")
        if current.next:
            print(" ↔ ", end="")
        current = current.next
    print()

def display_backward(self):
    """Display the list from tail to head"""
    if self.is_empty():
        print("List is empty")
        return

    current = self.tail
    print("Backward: ", end="")
    while current:
        print(current.data, end="")
        if current.prev:
            print(" ↔ ", end="")
        current = current.prev
    print()

def insert_at_beginning(self, data):
    """Insert a new node at the beginning"""
    new_node = Node(data)

    if self.is_empty():
        self.head = self.tail = new_node
```

```
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

        self.size += 1
        print(f"Inserted {data} at beginning")

    def insert_at_end(self, data):
        """Insert a new node at the end"""
        new_node = Node(data)

        if self.is_empty():
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

        self.size += 1
        print(f"Inserted {data} at end")

    def insert_at_position(self, data, position):
        """Insert a new node at a specific position"""
        if position < 0 or position > self.size:
            print("Invalid position")
            return

        if position == 0:
            self.insert_at_beginning(data)
            return
        elif position == self.size:
            self.insert_at_end(data)
            return

        new_node = Node(data)
        current = self.head
```

```
        # Traverse to the position
        for _ in range(position - 1):
            current = current.next

        # Insert the new node
        new_node.next = current.next
        new_node.prev = current
        current.next.prev = new_node
        current.next = new_node

        self.size += 1
        print(f"Inserted {data} at position {position}")

    def delete_from_beginning(self):
        """Delete the first node"""
        if self.is_empty():
            print("List is empty")
            return None

        deleted_data = self.head.data

        if self.head == self.tail:  # Only one node
            self.head = self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None

        self.size -= 1
        print(f"Deleted {deleted_data} from beginning")
        return deleted_data

    def delete_from_end(self):
        """Delete the last node"""
        if self.is_empty():
            print("List is empty")
            return None

        deleted_data = self.tail.data
```

```
    if self.head == self.tail:  # Only one node
        self.head = self.tail = None
    else:
        self.tail = self.tail.prev
        self.tail.next = None

    self.size -= 1
    print(f"Deleted {deleted_data} from end")
    return deleted_data

def delete_from_position(self, position):
    """Delete a node from a specific position"""
    if self.is_empty():
        print("List is empty")
        return None

    if position < 0 or position >= self.size:
        print("Invalid position")
        return None

    if position == 0:
        return self.delete_from_beginning()
    elif position == self.size - 1:
        return self.delete_from_end()

    current = self.head

    # Traverse to the position
    for _ in range(position):
        current = current.next

    # Delete the node
    deleted_data = current.data
    current.prev.next = current.next
    current.next.prev = current.prev

    self.size -= 1
```

```
    print(f"Deleted {deleted_data} from position {position}")
    return deleted_data

def search(self, data):
    """Search for a node with given data"""
    if self.is_empty():
        return -1

    current = self.head
    position = 0

    while current:
        if current.data == data:
            return position
        current = current.next
        position += 1

    return -1

def reverse(self):
    """Reverse the doubly linked list"""
    if self.is_empty() or self.head == self.tail:
        return

    current = self.head
    self.tail = self.head

    while current:
        # Swap next and prev pointers
        temp = current.prev
        current.prev = current.next
        current.next = temp

        # Move to the next node (which is now in prev due to swap)
        current = current.prev

    # Update head to the last node we processed
    if temp:
```

```
            self.head = temp.prev

        print("List reversed successfully")

    def clear(self):
        """Clear the entire list"""
        self.head = self.tail = None
        self.size = 0
        print("List cleared")

# Demonstration and testing
def demo_doubly_linked_list():
    """Demonstrate the doubly linked list operations"""
    print("=" * 50)
    print("DOUBLY LINKED LIST DEMONSTRATION")
    print("=" * 50)

    dll = DoublyLinkedList()

    # Insert operations
    dll.insert_at_beginning(10)
    dll.insert_at_end(20)
    dll.insert_at_end(30)
    dll.insert_at_beginning(5)
    dll.insert_at_position(15, 2)

    # Display
    dll.display_forward()
    dll.display_backward()
    print(f"Size: {dll.get_size()}")
    print()

    # Search operation
    search_value = 20
    position = dll.search(search_value)
    if position != -1:
        print(f"Found {search_value} at position {position}")
    else:
```

```
        print(f"{search_value} not found in the list")
    print()

    # Delete operations
    dll.delete_from_beginning()
    dll.delete_from_end()
    dll.delete_from_position(1)

    # Display after deletions
    dll.display_forward()
    print(f"Size: {dll.get_size()}")
    print()

    # Insert more elements
    dll.insert_at_end(40)
    dll.insert_at_end(50)
    dll.insert_at_end(60)

    # Display before reverse
    print("Before reverse:")
    dll.display_forward()

    # Reverse the list
    dll.reverse()

    # Display after reverse
    print("After reverse:")
    dll.display_forward()
    dll.display_backward()
    print()

    # Clear the list
    dll.clear()
    dll.display_forward()

# Interactive menu for user to test
def interactive_menu():
    """Interactive menu for testing the doubly linked list"""
```

```
dll = DoublyLinkedList()

while True:
    print("\n" + "=" * 40)
    print("DOUBLY LINKED LIST MENU")
    print("=" * 40)
    print("1. Insert at beginning")
    print("2. Insert at end")
    print("3. Insert at position")
    print("4. Delete from beginning")
    print("5. Delete from end")
    print("6. Delete from position")
    print("7. Search element")
    print("8. Display forward")
    print("9. Display backward")
    print("10. Reverse list")
    print("11. Get size")
    print("12. Clear list")
    print("13. Exit")
    print("=" * 40)

    choice = input("Enter your choice (1-13): ")

    if choice == '1':
        data = int(input("Enter data to insert: "))
        dll.insert_at_beginning(data)

    elif choice == '2':
        data = int(input("Enter data to insert: "))
        dll.insert_at_end(data)

    elif choice == '3':
        data = int(input("Enter data to insert: "))
        position = int(input("Enter position: "))
        dll.insert_at_position(data, position)

    elif choice == '4':
        dll.delete_from_beginning()
```

```
elif choice == '5':
    dll.delete_from_end()

elif choice == '6':
    position = int(input("Enter position to delete: "))
    dll.delete_from_position(position)

elif choice == '7':
    data = int(input("Enter data to search: "))
    pos = dll.search(data)
    if pos != -1:
        print(f"Element found at position {pos}")
    else:
        print("Element not found")

elif choice == '8':
    dll.display_forward()

elif choice == '9':
    dll.display_backward()

elif choice == '10':
    dll.reverse()

elif choice == '11':
    print(f"Size: {dll.get_size()}")

elif choice == '12':
    dll.clear()

elif choice == '13':
    print("Exiting...")
    break

else:
    print("Invalid choice! Please try again.")
```

```
if __name__ == "__main__":
    # Run the demonstration
    demo_doubly_linked_list()

    # Uncomment the line below to run interactive menu
    # interactive_menu()
```

- Save your source codes to GitHub

Answer the following questions:

1. What are the three main components of a Node in the doubly linked list implementation, and what does the __init__ method of the DoublyLinkedList class initialize?
2. The insert_at_beginning method successfully adds a new node to the start of the list. However, if we were to reverse the order of the two lines of code inside the else block, what specific issue would this introduce? Explain the sequence of operations that would lead to this problem:

```
def insert_at_beginning(self, data):
    new_node = Node(data)

    if self.is_empty():
        self.head = self.tail = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

    self.size += 1
```

3. How does the reverse method work? Trace through the reversal process step by step for a list containing [A, B, C], showing the pointer changes at each iteration

```
def reverse(self):
    if self.is_empty() or self.head == self.tail:
        return

    current = self.head
    self.tail = self.head

    while current:
        temp = current.prev
        current.prev = current.next
        current.next = temp
```

```
        current = current.prev

    if temp:
        self.head = temp.prev
```

III. Results

Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals, books, magazines, reports, or web articles please cite and reference them using the IEEE format.

Please take note of the styles on the style ribbon as these would serve as the style format of this laboratory report. The body style is Times New Roman size 12, line spacing: 1.5. Body text should be in Justified alignment, while captions should be center-aligned. Images should be readable and include captions. Please refer to the sample below:

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

Figure 1 Problem 1

```python
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

Figure 2 Problem 1

In Figure 1, a Node of a doubly linked list is shown, which has three main components: the data that stores the node's value, the prev pointer that links to the previous node, and the next pointer that links to the next node. In Figure 2, the DoublyLinkedList class is initialized, where the head is set to None because there is no first node yet, the tail is set to None because there is no last node, and the size is set to 0, indicating that the list starts empty. Together, Figures 1 and 2 illustrate how individual nodes are structured and how the overall list begins in an empty state.

```
[13] # Question number 2
     # correct order
     def insert_at_beginning(self, data):
         new_node = Node(data)  # create a new node

         if self.is_empty():
             self.head = self.tail = new_node  # list was empty
         else:
             new_node.next = self.head   # 1. new_node points to current head
             self.head.prev = new_node   # 2. old head points back to new_node
             self.head = new_node        # 3. update head to new_node

         self.size += 1
```

```
[15] # Question number 2
     # Reverse Order

     def insert_at_beginning(self, data):
         new_node = Node(data)  # create a new node

         if self.is_empty():
             self.head = self.tail = new_node  # list was empty
         else:
             self.head.prev = new_node   # old head points back to new_node first
             new_node.next = self.head   # new_node points forward to old head second
             self.head = new_node        # update head to new_node

         self.size += 1
```

Figure 3 Problem 2

In Figure 3 If you reverse the first two lines, the current head's prev pointer would temporarily point to a new node that hasn't yet linked forward to the head, creating an inconsistent state; this can lead to broken backward/forward links in the list, so the new node must first point to the old head before updating the old head's prev.

```python
# Question Number 3

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

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

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def reverse(self):
        if not self.head or self.head == self.tail:
            return

        current = self.head
        self.tail = self.head  # new tail is original head
        prev_node = None

        while current:
            # swap prev and next
            current.prev, current.next = current.next, current.prev
            prev_node = current
            current = current.prev  # move to original next

        # update head to last node processed
        if prev_node:
            self.head = prev_node

    def print_list(self):
        print("List from head to tail:")
        current = self.head
        while current:
            prev_data = current.prev.data if current.prev else None
            next_data = current.next.data if current.next else None
            print(f"[Prev: {prev_data} <- {current.data} -> Next: {next_data}]"
            current = current.next

# --- Test the code ---
dll = DoublyLinkedList()
dll.append("A")
dll.append("B")
dll.append("C")

print("Original list:")
dll.print_list()

dll.reverse()

print("\nReversed list:")
dll.print_list()
```

Figure 3 Problem 2

The Figure 3 implements a doubly linked list and demonstrates how to reverse it in place. Each node in the list has two pointers: prev pointing to the previous node and next pointing to the next node. The reverse method works by iterating through the list and swapping these two pointers for every node. It starts from the head of the list, temporarily stores the last node processed, and moves forward along the original next pointer, which becomes the new prev after swapping. Once all nodes have been processed, the head of the list is updated to the last node processed, which becomes the new head. For example, in the list [A, B, C], after reversal, C becomes the head, A becomes the tail, and the pointers are updated such that C.next = B, B.next = A, and A.next = None, while the prev pointers correctly point in the reverse direction (B.prev = C, A.prev = B). The print_list method is enhanced to show both prev and next pointers for each node, making it

clear how the internal structure of the list changes during reversal. This approach reverses the list efficiently in place without creating any new nodes.

IV. Conclusion

When inserting a new node at the beginning of a doubly linked list, the **order of operations is crucial**. The correct sequence is: first, set the new node's next pointer to the current head; second, update the old head's prev pointer to the new node; and finally, update the list's head pointer. Reversing the first two steps can temporarily break the links, causing traversal errors or a corrupted list. Following the correct order ensures that the list remains consistent and all nodes are properly connected.

References

[1] GeeksforGeeks, "Insertion in a Doubly Linked List," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/introduction-and-insertion-in-a-doubly-linked-list/. [Accessed: 23-Aug-2025].

[2] GeeksforGeeks, "Reverse a Doubly Linked List," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/reverse-a-doubly-linked-list/. [Accessed: 23-Aug-2025].

[3] GeeksforGeeks, "Insert a Node at a Specific Position in Doubly Linked List," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/insert-a-node-at-a-specific-position-in-doubly-linked-list/. [Accessed: 23-Aug-2025].

[4] GeeksforGeeks, "Deletion at Beginning (Removal of First Node) in a Doubly Linked List," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/deletion-at-beginning-removal-of-first-node-in-a-doubly-linked-list/. [Accessed: 23-Aug-2025].

[5] GeeksforGeeks, "Deletion at End (Removal of Last Node) in a Doubly Linked List," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/deletion-at-end-removal-of-last-node-in-a-doubly-linked-list/. [Accessed: 23-Aug-2025].