Data Structure and Algorithm

Laboratory Activity No. 7

Doubly Linked Lists

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Luminario, Venice Lou Gabrielle M. | Engr. Maria Rizette H. Sayo |

AUGUST 23, 2025

# 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.

# Methods

* Using Google Colab, type the source codes below:

class Node:  
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

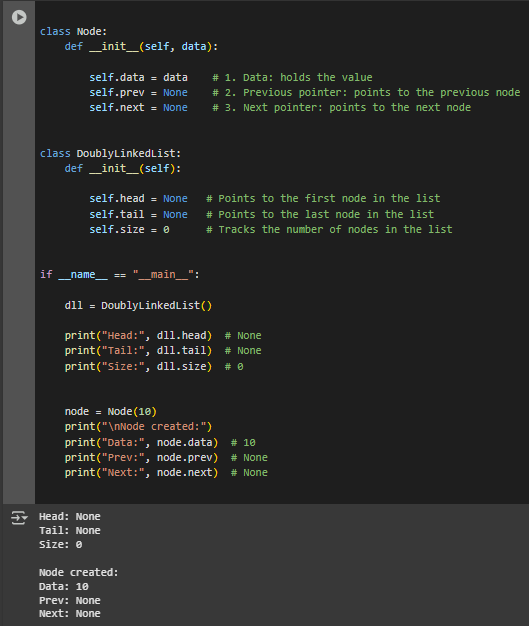
    # interactive\_menu()

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?

A node in a doubly linked list is made up of three parts: data (which stores the value), a pointer to the previous node, and a pointer to the next node. The \_\_init\_\_ function of the DoublyLinkedList class initializes the head and tail pointers to None, indicating that the list is empty, and sets the size to 0 to keep track of the number of nodes in the list.

  
Figure 1 Screenshot of program

* + 1. 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

If we change the order, the code will attempt to update self.head.prev before joining the new node to the existing head. Because new\_node.next has not yet been set, the link between the nodes is incomplete, which may cause the list to break or create problems. The proper procedure is to first refer new\_node.next to the existing head, then change the head's prior, and lastly set the new node as the new head.

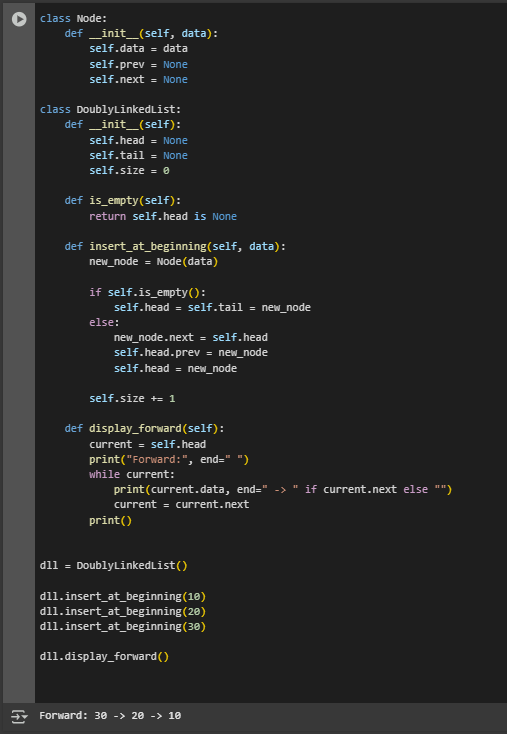


Figure 2 Screenshot of program

* + 1. 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:

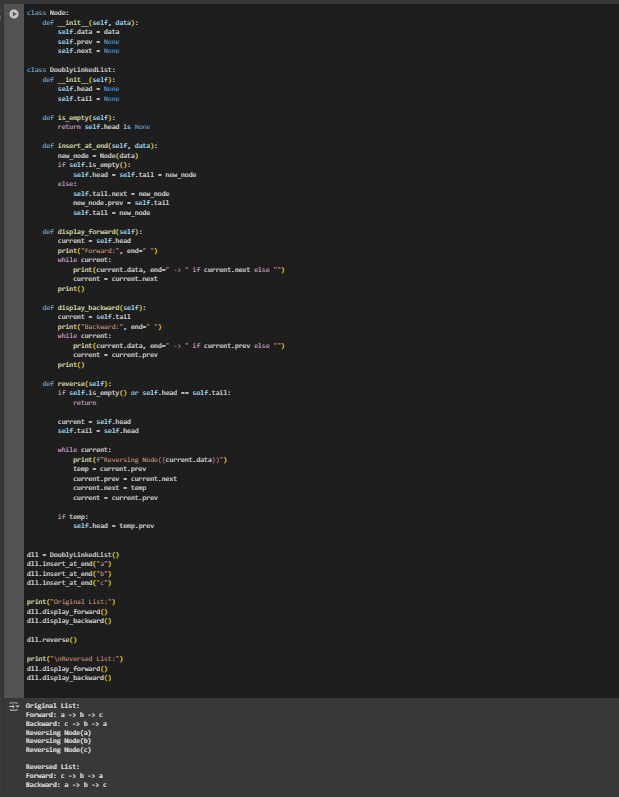
self.head = temp.prev

The reverse function proceeds across the list, flipping the prior and next pointers of each node. It begins with current at the head and assigns self.tail to the old head, as it will be the new tail. For each node, it briefly saves the preceding pointer, swaps prev and next, and then returns to the original next node.

Step 1 (Node A): A.prev (None) is swapped with A.next (B). Now A.prev = B and A.next = None.

* Step 2 (Node B**):** B.prev (A) is swapped with B.next (C). Now B.prev = C and B.next = A.
* Step 3 (Node C): C.prev (B) is swapped with C.next (None). Now C.prev = None and C.next = B.

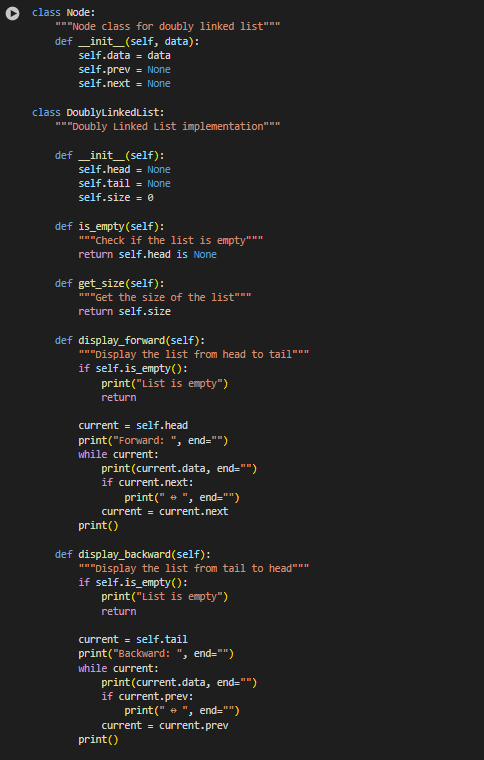
At the end of the traversal, the head pointer is updated to the last processed node, which is C. The new list order becomes [C, B, A], with all pointers correctly reversed.

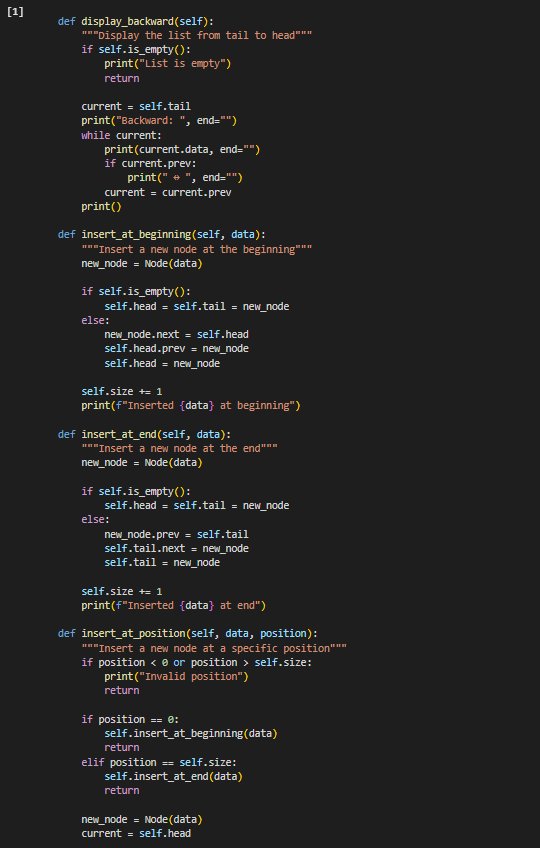
  
Figure 3 Screenshot of program

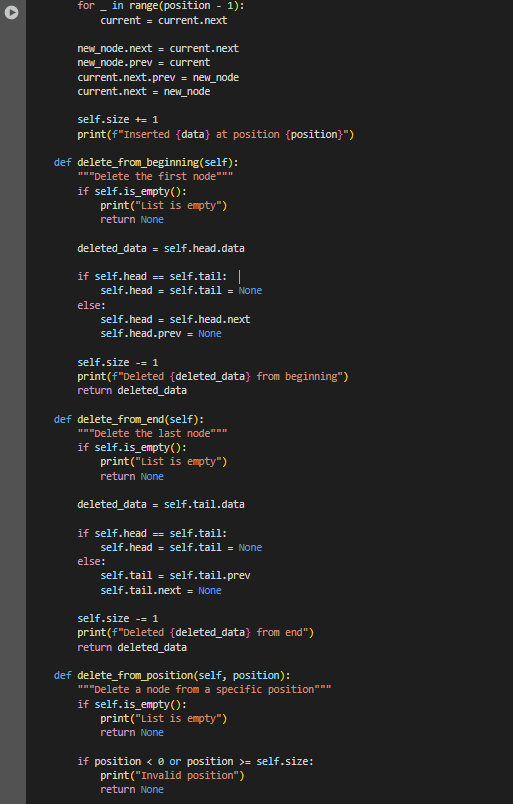
# Results

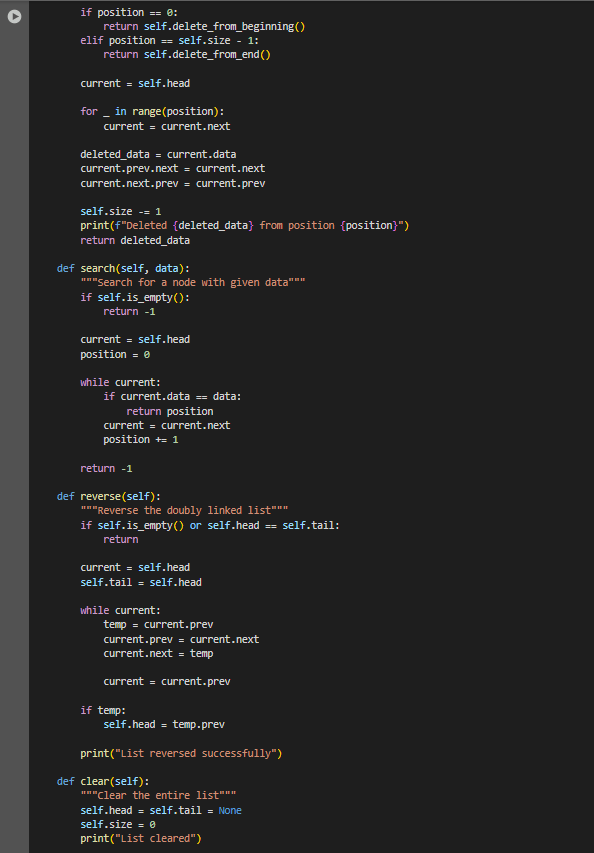
Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals,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

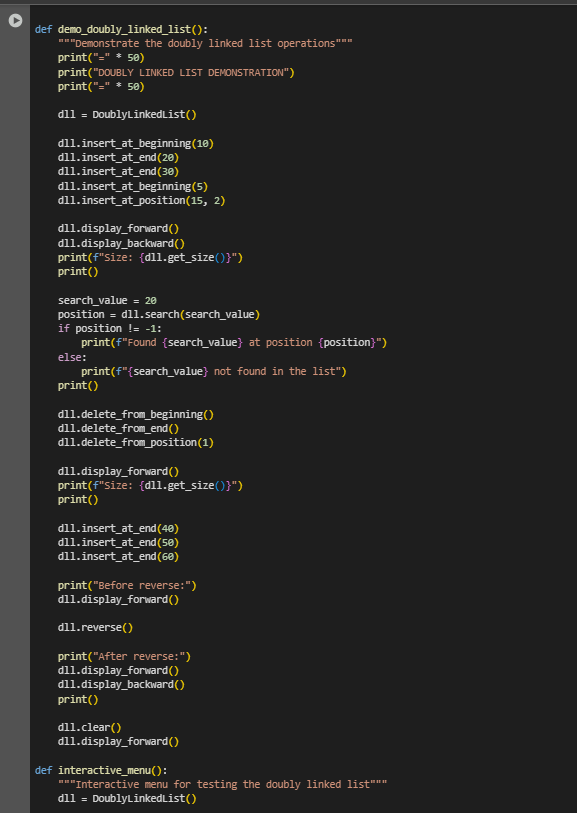
Please take note of the styles on the style ribbon as these would serve as the style format of this laboratory report. The body style is Times New Roman size 12, line spacing: 1.5. Body text should be in Justified alignment, while captions should be center-aligned. Images should be readable and include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

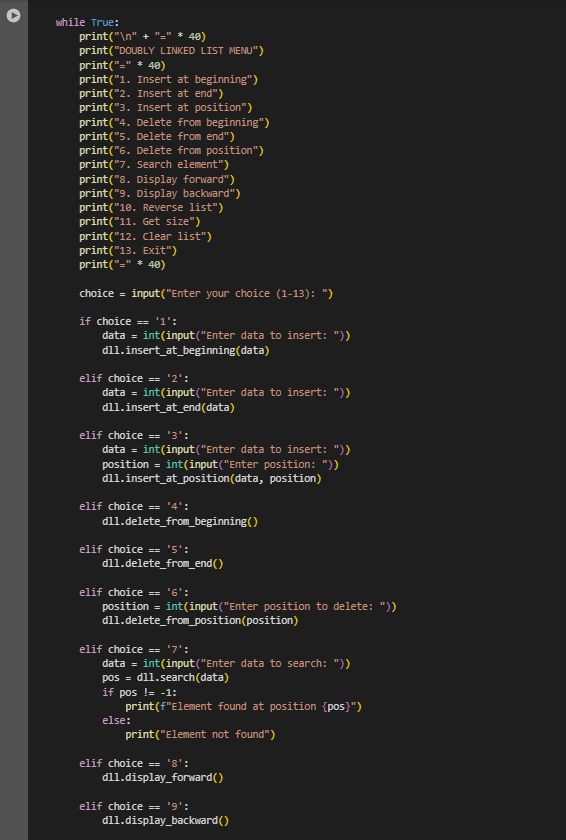


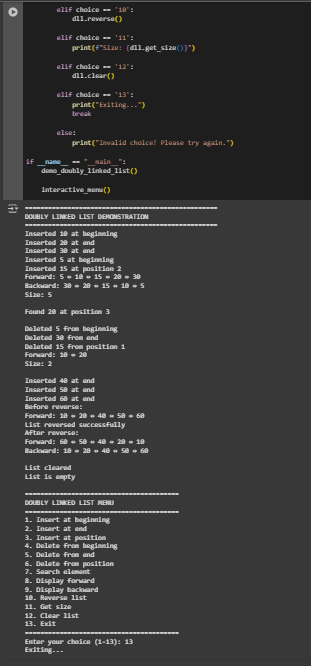












Figures Screenshots of program

This Python program provides a thorough and realistic implementation of a doubly linked list. A doubly linked list is a data structure composed of nodes, each of which holds some data and has two pointers one to the next node and one to the previous node. Unlike singly linked lists, this configuration permits movement in both directions, making some operations more flexible and economical. The code specifies two primary classes: Node, which produces individual members of the list, and DoublyLinkedList, which manages the entire list and actions. It has standard functionality such as inserting nodes at the beginning or end, removing individual nodes, searching for values, and showing the list forwards and backwards. There is also a way for reversing the full list. As the list evolves, the head, tail, and size properties are updated to maintain everything in order. To demonstrate how the list works, the application provides two ways to test it. The first is a sample function that performs a number of predefined operations such as adding, deleting, and reversing nodes to show you how everything works step by step. The second is an interactive menu that allows you to select actions and enter data manually, making it simple to play with and comprehend. Together, these tools bring the notion of doubly linked lists to life in an organized and hands-on manner.

Conclusion:

All things considered, this course provides a comprehensive and useful approach to comprehending doubly linked lists' operation, both conceptually and practically. It transforms a complex data structure into something far more comprehensible and practical by fusing an interactive menu, built-in demos, and a clear code structure.

This way simplifies understanding how nodes are connected, updated, and maintained, whether you're new to linked lists or returning to the basics. It's more than just a technical illustration, it is a effective teaching tool that bridges the gap between understanding a concept and seeing it in action.

**References**

W3Schools. (n.d.). *Python linked lists*. W3Schools. https://www.w3schools.com/python/python\_linked\_lists.asp

Python Software Foundation. (2023). *Python documentation: Classes*. Python.org. <https://docs.python.org/3/tutorial/classes.html>

GeeksforGeeks. (2022, November 28). *Doubly linked list data structure*. GeeksforGeeks. https://www.geeksforgeeks.org/doubly-linked-list/