Data Structure and Algorithm

Laboratory Activity No. 7

Doubly Linked Lists

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Gabuyo, Ivan love D. | Engr. Maria Rizette H. Sayo |

AUG, 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()

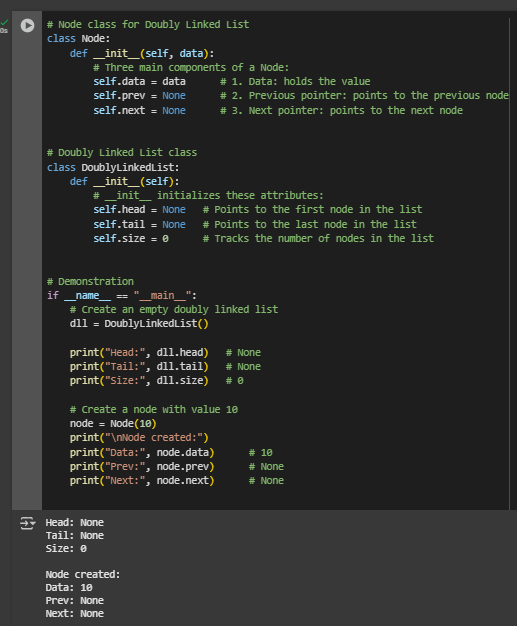
    # Uncomment the line below to run interactive menu

    # 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 has three components: the data, which stores the actual value, (the previous pointer, which references the node before it, and the next pointer, which references the node after it. The \_\_init\_\_ method of the DoublyLinkedList class initializes the linked list by setting the head and tail pointers to None (indicating an empty list) and the size to 0 (tracking how many nodes are currently stored).  
  
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  
answer: If we swapped the order of new\_node.next = self.head and self.head.prev = new\_node, the program would try to access and modify self.head.prev before updating new\_node.next to point to the current head. At this point, self.head still refers to the old head node, but since new\_node.next hasn’t been set yet, the linkage between the new node and the old head would not be established correctly. This would break the connection between nodes, causing either a missing link in the list or a runtime error due to referencing an incorrect or None pointer. The correct sequence is first to link the new node to the current head, then update the head’s prev pointer, and finally move the head reference to the new node.  
A screen shot of a computer

AI-generated content may be incorrect.  
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

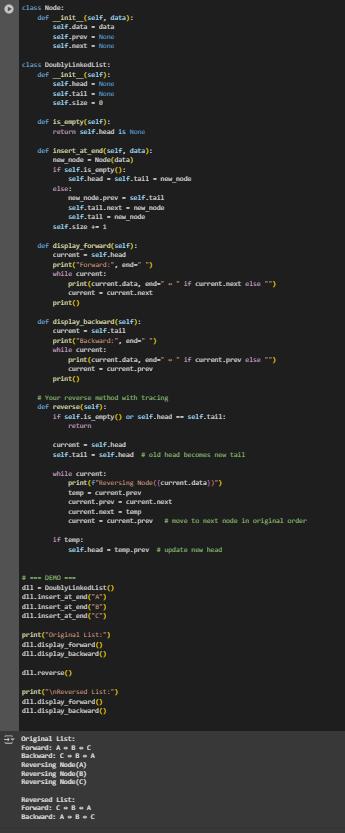
current = current.prev

if temp:

self.head = temp.prev  
answer: The reverse method works by swapping the prev and next pointers of each node as it traverses the list. Initially, current is set to the head, and self.tail is updated to point to the original head (since it will become the new tail). Then, for each node, the algorithm stores the prev pointer in a temporary variable, swaps the prev and next pointers, and moves current to what was originally the 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

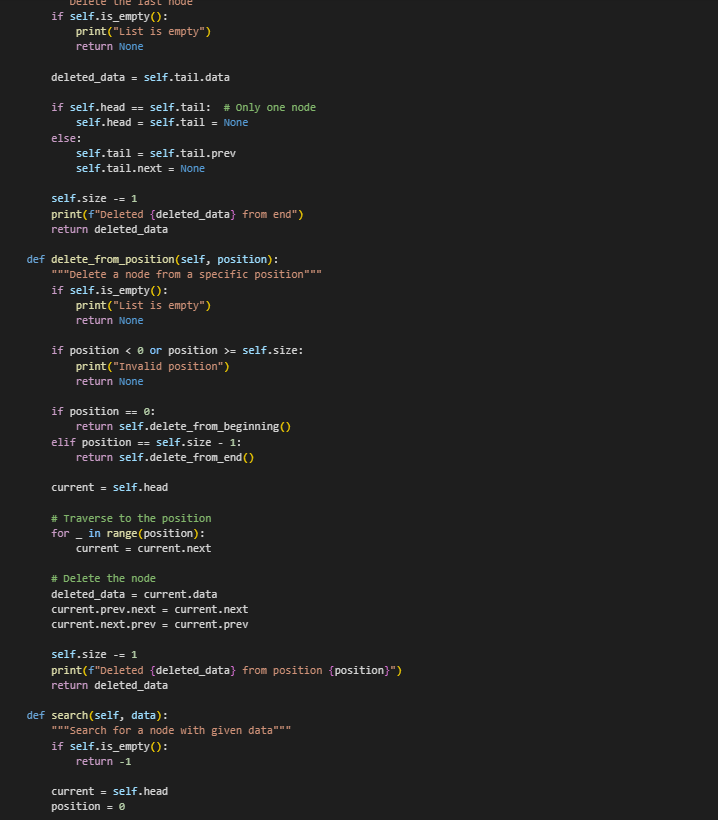
Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals, 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:

A screen shot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect.  A screen shot of a computer program

AI-generated content may be incorrect. A screen shot of a computer program

AI-generated content may be incorrect.  A screenshot of a computer program

AI-generated content may be incorrect.

Figures Screenshots of program

This program is a full implementation of a doubly linked list in Python, which is a data structure made up of nodes where each node has three parts: the data itself, a pointer to the next node, and a pointer to the previous node. Unlike a singly linked list, this design allows movement both forward and backward through the list. The program defines two main classes: Node, which creates individual nodes, and DoublyLinkedList, which manages the overall structure and operations. Within DoublyLinkedList, common list operations such as inserting at the beginning, at the end, or at a specific position are included, along with methods for deleting nodes, searching for values, displaying the list in both directions, and even reversing the entire list. Each operation updates the head (first node), tail (last node), and size attributes to keep the structure accurate as changes are made.

The program also includes two practical ways to test and use the list. The first is a demo\_doubly\_linked\_list() function, which automatically performs a sequence of operations—like inserting nodes, deleting nodes, displaying results, and reversing the list—to show how the structure behaves step by step. The second is an interactive\_menu() function, which allows the user to directly choose operations from a menu and input their own values, making the program more hands-on and user-friendly. By combining these two approaches, the code not only demonstrates how a doubly linked list works in theory but also provides a clear way to experiment with it in practice, reinforcing how linked lists organize and manage data dynamically.

# Conclusion

n this activity, I was able to implement and explore the different operations of a doubly linked list in Python. By writing and testing the code, I saw firsthand how each node connects not just forward but also backward, making the structure more flexible than a singly linked list. Performing insertions, deletions, traversals, searches, and reversals gave me a clearer picture of how pointers work together to keep the list organized. The demonstration and interactive menu made it easier to see the effects of each operation step by step. Overall, this experience gave me a stronger understanding of how doubly linked lists work in practice and why they are useful in situations that require efficient navigation in both directions.

**References**

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/

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