



UNIVERSITY OF CALOOCAN CITY  
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 7

---

# Doubly Linked Lists

---

*Submitted by:*  
Directo, Hannah Thea B.

*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



Figure 1 Screenshot of Output

**Answers:**

1. In the doubly linked list implementation, each node has three main components: data, prev, and next. The data holds the value or information stored in the node. The prev is a reference (or pointer) to the previous node in the list, while the next is a reference to the next node. These two references allow the list to be traversed in both forward and backward directions. The `_init_` method of the `DoublyLinkedList` class initializes three important attributes: head, tail, and size. Initially, both head and tail are set to `None`, indicating that the list is empty, and size is set to 0, representing the number of nodes in the list.

2. If we reverse the two lines inside the else block of the insert\_at\_beginning method, it can cause a problem with how the nodes are linked. The correct order is to first set new\_node.next = self.head and then update self.head.prev = new\_node. This way, the new node is connected to the current head before the head is updated. If we do it the other way around, we try to set the current head's previous pointer to a node that isn't fully linked yet. This can lead to incorrect or broken connections in the list. The list may not behave correctly during traversal or other operations. Therefore, the sequence matters to maintain proper structure in the doubly linked list.

3. The reverse method works by swapping next and prev pointers of each node to reverse the direction of the list. The method starts at the head(A) and moves node by node, swapping each node's next and prev pointers.

Initial List:

Head: ← A → B → C → Tail

A: prev = None, next = B

B: prev = A, next = C

C: prev = B, next = None

Step 1: Current = A

temp = A.prev → None

A.prev = A.next → B

A.next = temp → None

Current = A.prev → B

B ← A                      A: prev = B, next = None

Step 2: Current = B

temp = B.prev → A

B.prev = B.next → C

B.next = temp → A

Current = B.prev → C

C ← B → A                      B: prev = C, next = A

Step 3: Current = C  
temp = C.prev → B  
c.prev = C.next → None  
C.next = temp → B  
Current = C.prev → None (loop ends)  
C → B                      C: prev = None, next: B

Final:

Head: ← C ↔ B ↔ A → Tail

## IV. Conclusion

In this lab activity, we learned how to use a doubly linked list in Python. A doubly linked list is a type of list where each item connects to both the next and the previous items. We practiced how to add and remove items at the beginning, end, or any position in the list. We also tried moving through the list forward and backward. By writing and testing the code, we better understood how the pointers between nodes work. This helped us see how useful doubly linked lists are when we need flexible ways to store and manage data. This activity helped us improve our coding skills and understand how linked lists work in real programs.