Data Structure and Algorithm

Laboratory Activity No. 7

Doubly Linked Lists

|  |  |
| --- | --- |
| *Submitted by:* | *Instructor:* |
| Polestico,Paul Justine D. | 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:

"""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?

In a doubly linked list, The node has three parts:

The Data, which holds the actual value of information stored in the node.

The Prev,which is a reference to the previous node in the list and

The Next, which is a reference to the next node in the list.

The \_\_init\_\_ method initializes the starting state of the list,just like how you use a key to start the car. It initiliazes the head (the first node in the list) and the tail (the last node in the list.).

* + 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 you reverse the order of the two lines inside the else block of the insert\_at\_beginning method, it may cause incorrect behavior in how the nodes are linked. Specifically, the issue arises because you're trying to update the prev pointer of the current head **before** assigning the new node’s next pointer. In order to fix this issue, you need to define the relationship between nodes before updating their pointers.

* + 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

Iteration 1: Node A

temp = current.prev # temp → None

current.prev = current.next # A.prev → B

current.next = temp # A.next → None

current = current.prev # current → B

After this:

[A] → prev: B, next: None

Iteration 2: Node B

temp = current.prev # temp → A

current.prev = current.next # B.prev → C

current.next = temp # B.next → A

current = current.prev # current → C

After this:

[B] → prev: C, next: A

Iteration 3: Node C

temp = current.prev # temp → B

current.prev = current.next # C.prev → None

current.next = temp # C.next → B

current = current.prev # current → None (loop ends)

After this:

[C] → prev: None, next: B

Final Step

if temp:

self.head = temp.prev # temp → B, temp.prev → C

self.head → C

So now:

- head → C

- tail → A

- New list:

[C] <-> [B] <-> [A]

# Results



Figure 1 Screenshot of program

This program demonstrates the various functions that can used in a doubly linked list. While similar,there are notable differences. The most notable difference is that a doubly linked list can be iterated vice versa due to its bidirectional properties,meaning both ends of a doubly linked list can be manipulated. The next would be that doubly linked lists have twice the amount of nodes compared to singly linked lists. Lastly, Both nodes at the end of each side of a doubly linked list can be deleted or modified using the same functions used in a singly linked list.

Conclusion

Doubly linked lists,similar to singly lnked lists,are also helpful in managing,processing and arranging data in its unique way: Due to its bidirectional properties,both nodes at the end of each side of the list can be modified or deleted,similar to an array. In other words, it is like the singly linked lists with additional properties that can prove to be helpful in certain situations.

**References**

[1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.