**Lab3: Linked lists**

**Capacidades**

* Definir las reglas básicas para la creación de listas enlazadas.

**Seguridad**

* Generar un ambiente seguro.
* Evitar el consumo de alimentos.
* Dejar el ambiente ordenado y limpio.

**Preparación**

* El alumno debe revisar previamente el material cargado.

**Recursos**

* Computadora.

**Instrucciones**

Cada integrante del grupo debe seleccionar un ejercicio diferente y desarrollarlo con la siguiente estructura:

* Nombre del alumno
* Ejercicio a desarrollar
* Prompt engineering (Si aplica)
  + Prompt ingresado y/o captura
  + Análisis del prompt
  + Ajustes del prompt y/o captura
  + Comentarios de los compañeros
* Código (Si aplica)
  + Código desarrollado
  + Análisis del código
  + Captura de la ejecución del código
  + Comentarios de los compañeros
* Ejercicios (Si aplica)
  + Explicar cómo funciona el algoritmo
  + Hacer su diagrama de cómo se ejecuta.
  + Comentarios del problema
  + Hacer 3 casos de prueba

Desarrollar todo el código en inglés

# **Linked Lists Laboratory**

## **Introduction**

Welcome to the Linked Lists Laboratory! In this lab, you'll gain hands-on experience implementing and working with linked lists in Python. This step-by-step guide will help you understand the core concepts of linked lists and their operations.

## **Learning Objectives**

By the end of this lab, you should be able to:

1. Implement a singly linked list data structure
2. Understand the basic operations of linked lists
3. Analyze the time and space complexity of different operations
4. Apply linked lists to solve practical problems

## **File Structure**

Your linked\_list\_lab.py file will have the following structure:

linked\_list\_lab.py

├── class Node

│ ├── \_\_init\_\_()

│ ├── get\_data()

│ ├── set\_data()

│ ├── get\_next()

│ └── set\_next()

│

├── class LinkedList

│ ├── \_\_init\_\_()

│ ├── display()

│ ├── list\_length()

│ ├── insert\_at\_beginning()

│ ├── insert\_at\_end()

│ ├── insert\_at\_position()

│ ├── delete\_from\_beginning()

│ ├── delete\_from\_end()

│ ├── delete\_from\_position()

│ ├── search()

│ ├── get\_nth\_from\_end()

│ ├── clear()

│ ├── has\_cycle()

│ ├── reverse()

│ ├── find\_middle()

│ └── remove\_duplicates()

│

├── merge\_sorted\_lists() (standalone function)

│

├── class Queue

│ ├── \_\_init\_\_()

│ ├── is\_empty()

│ ├── enqueue()

│ ├── dequeue()

│ ├── peek()

│ ├── size()

│ └── display()

│

├── test\_linked\_list()

├── test\_queue()

└── test\_merge\_sorted\_lists()

## **Part 1: The Node Class**

Let's start by defining the basic building block of a linked list: the Node.

class Node:

"""Node in a linked list, stores data and reference to the next node."""

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

def get\_data(self):

return self.data

def set\_data(self, data):

self.data = data

def get\_next(self):

return self.next

def set\_next(self, next\_node):

self.next = next\_node

## **Part 2: The LinkedList Class**

Now, let's implement the LinkedList class that will use our Node class.

class LinkedList:

"""Singly linked list implementation."""

def \_\_init\_\_(self):

self.head = None

self.length = 0

## **Part 3: Basic Operations**

### **Exercise 1: Displaying the List**

Implement a method to display all elements in the list.

def display(self):

"""Return a string representation of the linked list."""

if self.head is None:

return "Empty list"

current = self.head

result = ""

while current is not None:

result += str(current.get\_data()) + " -> "

current = current.get\_next()

return result + "None"

### **Exercise 2: Counting Nodes**

Implement a method to count the number of nodes in the linked list.

def list\_length(self):

"""Count and return the number of nodes in the list."""

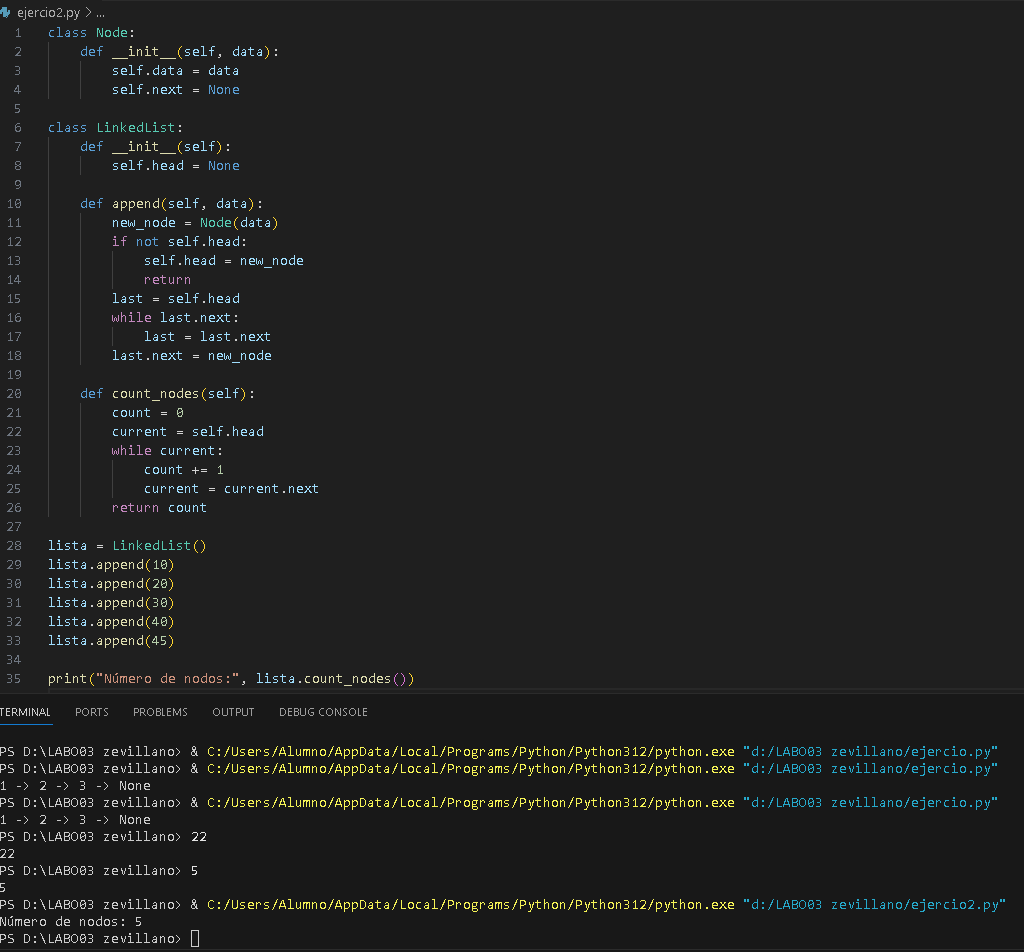
count = 0

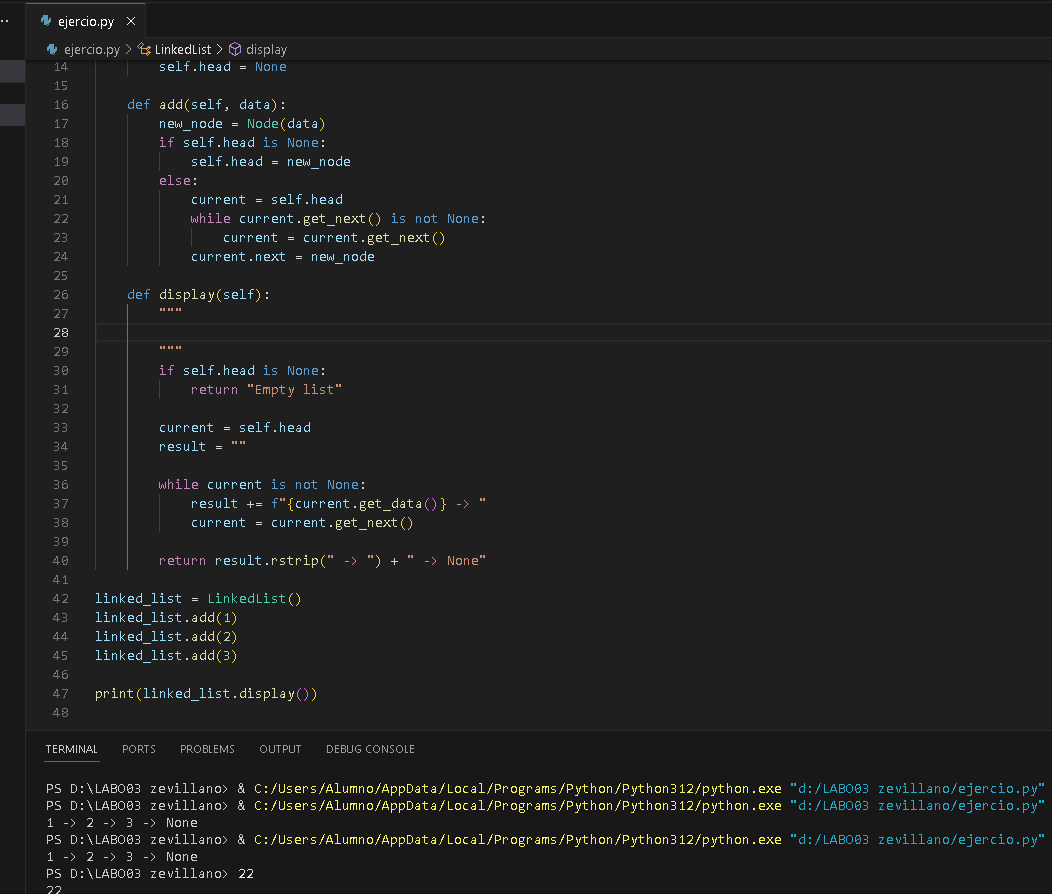
current = self.head

while current is not None:

count += 1

current = current.get\_next()

return count  




### **Exercise 3: Insertion at the Beginning**

Implement a method to insert a new node at the beginning of the list.

def insert\_at\_beginning(self, data):

"""Insert a new node with data at the beginning of the list."""

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

new\_node.set\_next(self.head)

self.head = new\_node

self.length += 1

return True

### **Exercise 4: Insertion at the End**

Implement a method to insert a new node at the end of the list.

def insert\_at\_end(self, data):

"""Insert a new node with data at the end of the list."""

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

current = self.head

*# Traverse to the last node*

while current.get\_next() is not None:

current = current.get\_next()

current.set\_next(new\_node)

self.length += 1

return True

### **Exercise 5: Insertion at a Specific Position**

Implement a method to insert a new node at a specific position in the list.

def insert\_at\_position(self, position, data):

"""Insert a new node at the specified position (0-based)."""

*# Check if position is valid*

if position < 0 or position > self.length:

return False

*# Insert at the beginning*

if position == 0:

return self.insert\_at\_beginning(data)

*# Insert at the end*

if position == self.length:

return self.insert\_at\_end(data)

*# Insert at the middle*

new\_node = Node(data)

current = self.head

count = 0

*# Traverse to the node just before the insertion point*

while count < position - 1:

current = current.get\_next()

count += 1

new\_node.set\_next(current.get\_next())

current.set\_next(new\_node)

self.length += 1

return True

## **Part 4: More Advanced Operations**

### **Exercise 6: Deletion from the Beginning**

Implement a method to delete a node from the beginning of the list.

def delete\_from\_beginning(self):

"""Delete and return the data from the first node."""

if self.head is None:

return None

data = self.head.get\_data()

self.head = self.head.get\_next()

self.length -= 1

return data

### **Exercise 7: Deletion from the End**

Implement a method to delete a node from the end of the list.

def delete\_from\_end(self):

"""Delete and return the data from the last node."""

if self.head is None:

return None

*# If there's only one node*

if self.head.get\_next() is None:

data = self.head.get\_data()

self.head = None

self.length -= 1

return data

current = self.head

*# Traverse to the second-to-last node*

while current.get\_next().get\_next() is not None:

current = current.get\_next()

data = current.get\_next().get\_data()

current.set\_next(None)

self.length -= 1

return data

### **Exercise 8: Deletion from a Specific Position**

Implement a method to delete a node from a specific position in the list.

def delete\_from\_position(self, position):

"""Delete and return data from node at the specified position."""

*# Check if position is valid*

if position < 0 or position >= self.length or self.head is None:

return None

*# Delete from the beginning*

if position == 0:

return self.delete\_from\_beginning()

*# Delete from the end*

if position == self.length - 1:

return self.delete\_from\_end()

*# Delete from the middle*

current = self.head

count = 0

*# Traverse to the node just before the deletion point*

while count < position - 1:

current = current.get\_next()

count += 1

node\_to\_delete = current.get\_next()

data = node\_to\_delete.get\_data()

current.set\_next(node\_to\_delete.get\_next())

self.length -= 1

return data

### **Exercise 9: Searching**

Implement a method to search for a value in the list and return its position.

def search(self, data):

"""Find the position of data in the list, or return -1 if not found."""

if self.head is None:

return -1

current = self.head

position = 0

while current is not None:

if current.get\_data() == data:

return position

current = current.get\_next()

position += 1

return -1

### **Exercise 10: Finding the Nth Node from the End**

Implement a method to find the nth node from the end of the list.

def get\_nth\_from\_end(self, n):

"""Return the data of the nth node from the end (1-based indexing)."""

if n <= 0 or n > self.length or self.head is None:

return None

*# The nth node from the end is the (length-n+1)th node from the beginning*

position = self.length - n

current = self.head

count = 0

while count < position:

current = current.get\_next()

count += 1

return current.get\_data()

### **Exercise 11: Clearing the List**

Implement a method to clear all nodes from the list.

def clear(self):

"""Remove all nodes from the list."""

self.head = None

self.length = 0

return True

## **Part 5: Testing Your Implementation**

Let's test the basic operations we've implemented:

def test\_linked\_list():

"""Test the LinkedList implementation with basic operations."""

my\_list = LinkedList()

print("Created a new linked list")

print(f"List: {my\_list.display()}")

print(f"Length: {my\_list.list\_length()}")

*# Test insertions*

print("\nTesting insertions:")

my\_list.insert\_at\_beginning(5)

print(f"After insert\_at\_beginning(5): {my\_list.display()}")

my\_list.insert\_at\_beginning(10)

print(f"After insert\_at\_beginning(10): {my\_list.display()}")

my\_list.insert\_at\_end(20)

print(f"After insert\_at\_end(20): {my\_list.display()}")

my\_list.insert\_at\_position(1, 15)

print(f"After insert\_at\_position(1, 15): {my\_list.display()}")

print(f"Current length: {my\_list.list\_length()}")

*# Test search*

print("\nTesting search:")

print(f"Position of 15: {my\_list.search(15)}")

print(f"Position of 100: {my\_list.search(100)}")

*# Test deletions*

print("\nTesting deletions:")

deleted = my\_list.delete\_from\_beginning()

print(f"Deleted from beginning: {deleted}")

print(f"After deletion: {my\_list.display()}")

deleted = my\_list.delete\_from\_position(1)

print(f"Deleted from position 1: {deleted}")

print(f"After deletion: {my\_list.display()}")

deleted = my\_list.delete\_from\_end()

print(f"Deleted from end: {deleted}")

print(f"After deletion: {my\_list.display()}")

*# Clear the list*

print("\nTesting clear:")

my\_list.clear()

print(f"After clear: {my\_list.display()}")

print(f"Length after clear: {my\_list.list\_length()}")

*# Run the test*

if \_\_name\_\_ == "\_\_main\_\_":

test\_linked\_list()

# **Part 6: Advanced Challenges (Optional)**

Now that you've implemented and tested the basic linked list operations, here are some more advanced challenges to practice:

### **Challenge 1: Cycle Detection**

Implement a method to detect if a linked list has a cycle (a node points back to a previous node). For this, you can use Floyd's Cycle-Finding Algorithm (also known as the "tortoise and hare" algorithm).

**Outline:**

* Use two pointers: one moving one step at a time (slow) and one moving two steps at a time (fast)
* If they ever meet, there's a cycle
* If fast reaches the end (null), there's no cycle

**Test case:**

*# Create a list with a cycle*

cycle\_list = LinkedList()

cycle\_list.insert\_at\_end(1)

cycle\_list.insert\_at\_end(2)

cycle\_list.insert\_at\_end(3)

cycle\_list.insert\_at\_end(4)

*# Create a cycle by connecting the last node to the second node*

last = cycle\_list.head

while last.get\_next() is not None:

last = last.get\_next()

second = cycle\_list.head.get\_next()

last.set\_next(second)

*# Test cycle detection*

print(f"Has cycle: {cycle\_list.has\_cycle()}")

### **Challenge 2: List Reversal**

Implement a method to reverse the linked list in-place.

**Outline:**

* Use three pointers: previous, current, and next
* Iterate through the list, reversing each link
* Update the head to point to the new first node (previously the last)

**Test case:**

*# Create and reverse a list*

reverse\_list = LinkedList()

for i in range(1, 6):

reverse\_list.insert\_at\_end(i)

print(f"Original list: {reverse\_list.display()}")

reverse\_list.reverse()

print(f"Reversed list: {reverse\_list.display()}")

### **Challenge 3: Finding the Middle Node**

Implement a method to find the middle node of the linked list using only one pass.

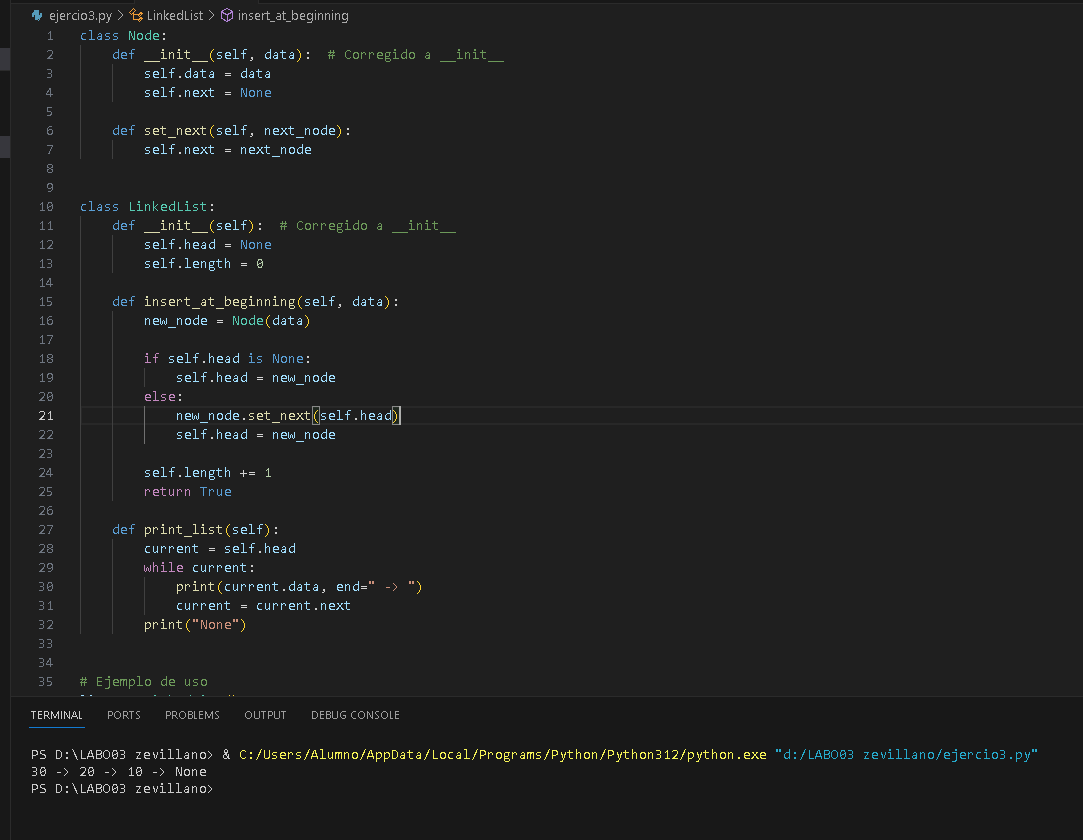
**Outline:**

* Use the "slow and fast pointer" technique
* Slow moves one step at a time, fast moves two steps
* When fast reaches the end, slow is at the middle

**Test case:**

*# Create a list and find the middle*

middle\_list = LinkedList()

for i in range(1, 8): 

middle\_list.insert\_at\_end(i)

print(f"List: {middle\_list.display()}")

print(f"Middle element: {middle\_list.find\_middle()}")

### **Challenge 4: Removing Duplicates**

Implement a method to remove duplicate values from a linked list.

**Outline:**

* Use a set to track values you've seen
* Traverse the list, removing nodes with duplicate values
* Keep track of the previous node to connect after removing a node

**Test case:**

*# Create a list with duplicates*

dup\_list = LinkedList()

for val in [1, 2, 3, 2, 4, 1, 5]:

dup\_list.insert\_at\_end(val)

print(f"List with duplicates: {dup\_list.display()}")

dup\_list.remove\_duplicates()

print(f"List after removing duplicates: {dup\_list.display()}")

### **Challenge 5: Merging Sorted Lists**

Implement a function to merge two sorted linked lists into a single sorted linked list.

**Outline:**

* Create a new result list
* Compare elements from both lists, adding the smaller to the result
* When one list is empty, add all remaining elements from the other list

**Test case:**

*# Create two sorted lists*

list1 = LinkedList()

for val in [1, 3, 5, 7]:

list1.insert\_at\_end(val)

list2 = LinkedList()

for val in [2, 4, 6, 8]:

list2.insert\_at\_end(val)

print(f"List 1: {list1.display()}")

print(f"List 2: {list2.display()}")

merged = merge\_sorted\_lists(list1, list2)

print(f"Merged list: {merged.display()}")

## **Part 7: Practical Linked List Exercises**

Now that you've implemented the basic operations of a linked list, let's look at some practical exercises that use linked lists to solve real problems.

## **Exercise 1: Implementing a Stack using a Linked List**

A Stack is a Last-In-First-Out (LIFO) data structure. Implement a Stack using your LinkedList class:

class Stack:

"""Stack implementation using a linked list (LIFO data structure)."""

def \_\_init\_\_(self):

self.linked\_list = LinkedList()

def is\_empty(self):

"""Check if the stack is empty."""

return self.linked\_list.head is None

def push(self, data):

"""Add an element to the top of the stack."""

self.linked\_list.insert\_at\_beginning(data)

def pop(self):

"""Remove and return the element at the top of the stack."""

return self.linked\_list.delete\_from\_beginning()

def peek(self):

"""Return the top element without removing it."""

if self.is\_empty():

return None

return self.linked\_list.head.get\_data()

def size(self):

"""Return the number of elements in the stack."""

return self.linked\_list.length

def display(self):

"""Display the elements in the stack."""

return self.linked\_list.display()

**Test your Stack implementation:**

def test\_stack():

stack = Stack()

print("Created a new stack")

print(f"Stack: {stack.display()}")

print("\nPushing elements:")

for i in range(1, 6):

stack.push(i)

print(f"Pushed {i}, Stack: {stack.display()}")

print(f"\nTop element (peek): {stack.peek()}")

print(f"Stack size: {stack.size()}")

print("\nPopping elements:")

while not stack.is\_empty():

print(f"Popped: {stack.pop()}, Stack: {stack.display()}")

## **Exercise 2: Implementing a Queue using a Linked List**

A Queue is a First-In-First-Out (FIFO) data structure. Implement a Queue using your LinkedList class:

class Queue:

"""Queue implementation using a linked list (FIFO data structure)."""

def \_\_init\_\_(self):

self.linked\_list = LinkedList()

def is\_empty(self):

"""Check if the queue is empty."""

return self.linked\_list.head is None

def enqueue(self, data):

"""Add an element to the end of the queue."""

self.linked\_list.insert\_at\_end(data)

def dequeue(self):

"""Remove and return the element at the front of the queue."""

return self.linked\_list.delete\_from\_beginning()

def peek(self):

"""Return the front element without removing it."""

if self.is\_empty():

return None

return self.linked\_list.head.get\_data()

def size(self):

"""Return the number of elements in the queue."""

return self.linked\_list.length

def display(self):

"""Display the elements in the queue."""

return self.linked\_list.display()

**Test your Queue implementation:**

def test\_queue():

queue = Queue()

print("Created a new queue")

print(f"Queue: {queue.display()}")

print("\nEnqueuing elements:")

for i in range(1, 6):

queue.enqueue(i)

print(f"Enqueued {i}, Queue: {queue.display()}")

print(f"\nFront element (peek): {queue.peek()}")

print(f"Queue size: {queue.size()}")

print("\nDequeuing elements:")

while not queue.is\_empty():

print(f"Dequeued: {queue.dequeue()}, Queue: {queue.display()}")

## **Exercise 3: Polynomial Representation**

Linked lists can be used to represent polynomials, where each node contains a coefficient and an exponent. Implement a PolynomialTerm class and a Polynomial class to represent polynomials:

class PolynomialTerm:

"""A term in a polynomial with coefficient and exponent."""

def \_\_init\_\_(self, coefficient, exponent):

self.coefficient = coefficient

self.exponent = exponent

def \_\_str\_\_(self):

if self.exponent == 0:

return str(self.coefficient)

elif self.exponent == 1:

return f"{self.coefficient}x"

else:

return f"{self.coefficient}x^{self.exponent}"

class Polynomial:

"""A polynomial represented as a linked list of terms."""

def \_\_init\_\_(self):

self.terms = LinkedList()

def add\_term(self, coefficient, exponent):

"""Add a term to the polynomial."""

*# Skip terms with coefficient 0*

if coefficient == 0:

return

*# Add term to the list*

term = PolynomialTerm(coefficient, exponent)

current = self.terms.head

prev = None

*# Find the right position based on exponent (descending order)*

while current is not None and current.get\_data().exponent > exponent:

prev = current

current = current.get\_next()

*# Check if we already have a term with this exponent*

if current is not None and current.get\_data().exponent == exponent:

*# Add coefficients*

new\_coef = current.get\_data().coefficient + coefficient

if new\_coef != 0:

current.get\_data().coefficient = new\_coef

else:

*# If coefficient becomes 0, remove the term*

if prev is None:

self.terms.head = current.get\_next()

else:

prev.set\_next(current.get\_next())

self.terms.length -= 1

else:

*# Insert the new term at the right position*

new\_node = Node(term)

if prev is None:

new\_node.set\_next(self.terms.head)

self.terms.head = new\_node

else:

new\_node.set\_next(current)

prev.set\_next(new\_node)

self.terms.length += 1

def \_\_str\_\_(self):

"""Return string representation of the polynomial."""

if self.terms.head is None:

return "0"

result = ""

current = self.terms.head

while current is not None:

term = current.get\_data()

*# Add + sign for positive terms (except the first one)*

if result and term.coefficient > 0:

result += " + "

*# Add - sign for negative terms*

elif term.coefficient < 0:

result += " - " if result else "-"

*# Add the term (without the sign if it's negative)*

coef = abs(term.coefficient)

if term.exponent == 0:

result += str(coef)

elif term.exponent == 1:

result += f"{coef}x"

else:

result += f"{coef}x^{term.exponent}"

current = current.get\_next()

return result

**Test your Polynomial implementation:**

def test\_polynomial():

poly = Polynomial()

print("Created an empty polynomial")

print(f"Polynomial: {poly}")

print("\nAdding terms:")

poly.add\_term(3, 2) *# 3x^2*

print(f"After adding 3x^2: {poly}")

poly.add\_term(-2, 1) *# -2x*

print(f"After adding -2x: {poly}")

poly.add\_term(5, 0) *# 5*

print(f"After adding 5: {poly}")

poly.add\_term(1, 2) *# Add to existing term 3x^2 + 1x^2 = 4x^2*

print(f"After adding 1x^2: {poly}")

poly.add\_term(-5, 0) *# Cancel out constant term*

print(f"After adding -5: {poly}")

## **Exercise 4: Sparse Matrix Representation**

A sparse matrix is a matrix with mostly zero values. It can be efficiently represented using linked lists to store only non-zero elements. Each non-zero element is represented as a node containing the row index, column index, and value:

class MatrixElement:

"""A non-zero element in a sparse matrix."""

def \_\_init\_\_(self, row, col, value):

self.row = row

self.col = col

self.value = value

def \_\_str\_\_(self):

return f"({self.row}, {self.col}, {self.value})"

class SparseMatrix:

"""A sparse matrix represented using linked lists."""

def \_\_init\_\_(self, rows, cols):

self.rows = rows

self.cols = cols

self.elements = LinkedList()

def set\_element(self, row, col, value):

"""Set the value at position (row, col)."""

if row < 0 or row >= self.rows or col < 0 or col >= self.cols:

raise ValueError("Position out of bounds")

*# If value is 0, remove the element if it exists*

if value == 0:

self.remove\_element(row, col)

return

*# Check if element already exists*

current = self.elements.head

prev = None

while current is not None:

elem = current.get\_data()

if elem.row == row and elem.col == col:

*# Update existing element*

elem.value = value

return

elif (elem.row > row) or (elem.row == row and elem.col > col):

*# Found position to insert (keep sorted by row, then column)*

break

prev = current

current = current.get\_next()

*# Insert new element*

new\_elem = MatrixElement(row, col, value)

new\_node = Node(new\_elem)

if prev is None:

new\_node.set\_next(self.elements.head)

self.elements.head = new\_node

else:

new\_node.set\_next(current)

prev.set\_next(new\_node)

self.elements.length += 1

def get\_element(self, row, col):

"""Get the value at position (row, col)."""

if row < 0 or row >= self.rows or col < 0 or col >= self.cols:

raise ValueError("Position out of bounds")

current = self.elements.head

while current is not None:

elem = current.get\_data()

if elem.row == row and elem.col == col:

return elem.value

current = current.get\_next()

*# If element not found, it's 0*

return 0

def remove\_element(self, row, col):

"""Remove the element at position (row, col)."""

if row < 0 or row >= self.rows or col < 0 or col >= self.cols:

raise ValueError("Position out of bounds")

current = self.elements.head

prev = None

while current is not None:

elem = current.get\_data()

if elem.row == row and elem.col == col:

*# Remove the element*

if prev is None:

self.elements.head = current.get\_next()

else:

prev.set\_next(current.get\_next())

self.elements.length -= 1

return

prev = current

current = current.get\_next()

def display(self):

"""Display the matrix in a readable format."""

result = []

for i in range(self.rows):

row = []

for j in range(self.cols):

row.append(self.get\_element(i, j))

result.append(row)

return result

**Test your SparseMatrix implementation:**

def test\_sparse\_matrix():

matrix = SparseMatrix(4, 4)

print("Created a 4x4 sparse matrix")

*# Set some elements*

matrix.set\_element(0, 0, 1)

matrix.set\_element(1, 1, 2)

matrix.set\_element(2, 2, 3)

matrix.set\_element(3, 3, 4)

matrix.set\_element(0, 3, 5)

print("\nMatrix after adding elements:")

for row in matrix.display():

print(row)

*# Update an element*

matrix.set\_element(0, 0, 10)

*# Remove an element*

matrix.set\_element(1, 1, 0) *# Setting to 0 removes the element*

print("\nMatrix after updates:")

for row in matrix.display():

print(row)

*# Print all non-zero elements*

print("\nNon-zero elements:")

current = matrix.elements.head

while current is not None:

print(current.get\_data())

current = current.get\_next()

## **Part 8: Real-World Applications of Linked Lists**

Linked lists are fundamental data structures used in many real-world applications:

1. **Implementation of other data structures**:
   * Stacks, queues, and hash tables (for collision handling)
   * Adjacency lists for graph representation
   * Symbol tables in compiler design
2. **Operating systems**:
   * Memory management (free lists)
   * Process scheduling queues
   * File system directories
3. **Applications**:
   * Undo functionality in text editors and applications
   * Browser's forward and backward navigation (doubly linked lists)
   * Music player playlists
   * Image galleries for navigating between images
   * Social media feeds for infinite scrolling
   * Text editors for efficient insertion/deletion

## **Part 9: Time Complexity Analysis**

Understanding the performance characteristics of linked lists is crucial for choosing the right data structure for your application:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Linked List** | **Array** | **Dynamic Array** |
| Access by index | O(n) | O(1) | O(1) |
| Insertion/deletion at start | O(1) | O(n) | O(n) |
| Insertion/deletion at end | O(n)\* | O(1) | O(1) amortized |

## **Conclusion**

Linked lists are fundamental data structures that offer flexibility in dynamic memory allocation and efficient insertions/deletions at the beginning. While they have limitations in random access operations, they serve as building blocks for more complex data structures and algorithms.

The skills you've developed in this lab will help you understand not only linked lists but also the principles behind efficient data manipulation and algorithm design. Continue exploring and applying these concepts to enhance your programming toolkit!

**Reto**

Proponer un algoritmo que resuelva un problema del 1 al 8 de la siguiente web <https://adventjs.dev/> e iterar hasta tener 5 estrellas. (Entrar con tus datos)

**Desarrollo**

**Incrementos**

* **Repositorio (0-5)**
* **Código fuente (0-5)**
* **Informe (-10-0)**
* **Expo (0-10)**

**Conclusiones**

Linked lists are data structures composed of nodes, where each node stores a value and a pointer to the next node in the sequence. The first node is called the head, while the last node has a pointer with the value None, indicating the end of the list. Unlike arrays, linked lists allow for efficient insertion and deletion of elements, especially at the beginning or intermediate positions, without having to shift other elements. However, they consume more memory due to the additional pointers, and their access is sequential, meaning they cannot be accessed directly by index, but rather the list must be traversed node by node.

When designing a linked list, it is important to consider fundamental operations such as adding elements to the beginning or end, traversing the list to display its data, and searching for or deleting nodes by adjusting the pointers appropriately. There are different types of linked lists: the singly linked list, where each node points only to the next, and the doubly linked list, in which each node has references to both the previous and next nodes. Although linked lists are flexible and facilitate efficient data manipulation, their main disadvantage is the lack of direct access to elements, which requires traversing the structure sequentially.