Universidad Autónoma de Nayarit

Unidad Académica de Economía

Programa Académico de

Licenciatura en Sistemas Computacionales

Estructura de Datos Básica

Semana 3: Listas Enlazadas Simples

Presenta: OCTAVIO SEBASTIAN PARRA CAJERO

Docente:DR. ELIGARDO CRUZ SANCHEZ

Tepic, Nayarit., 13 de febrero de 2026.

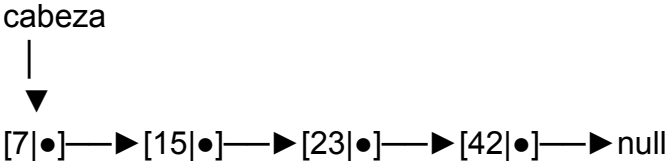
1-Visualización de Lista Enlazada

Entregables esperado

-Documento con los diagramas generados, anotados con tus propias palabras explicando: (1) por qué insertar al inicio es O(1), (2) por qué insertar al final es O(n), y (3) qué problema surge al eliminar el primer nodo que no ocurre con otros nodos.

Operaciones en una Lista Enlazada Simple con Diagramas ASCII

Lista base utilizada para los ejemplos:

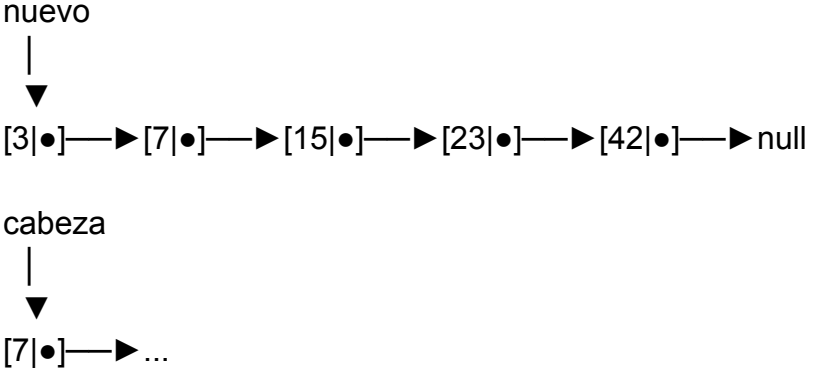


1. Inserción al inicio

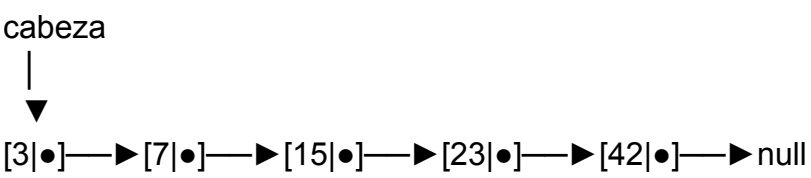
Estado inicial

Paso 1: Crear el nuevo nodo

Paso 2: Hacer que el nuevo nodo apunte al antiguo primero



Paso 3: Actualizar la cabeza



Explicación de complejidad: ¿Por qué es O(1)?

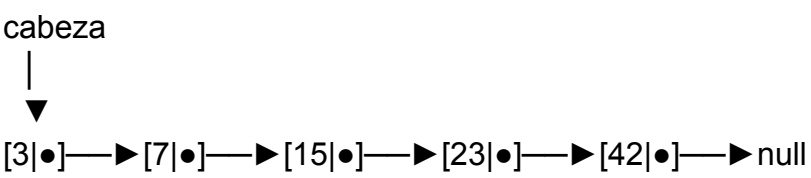
Insertar al inicio es una operación de tiempo constante porque no depende del tamaño de la lista. Sin importar si la lista tiene 4 nodos o 4 millones, siempre se realizan exactamente dos cambios de puntero:

1. nuevo.next = cabeza
2. cabeza = nuevo

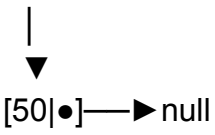
No se necesita recorrer la estructura. El número de operaciones es fijo y constante, por eso su complejidad temporal es O(1).

2. Inserción al final

Lista después de insertar 3 al inicio:

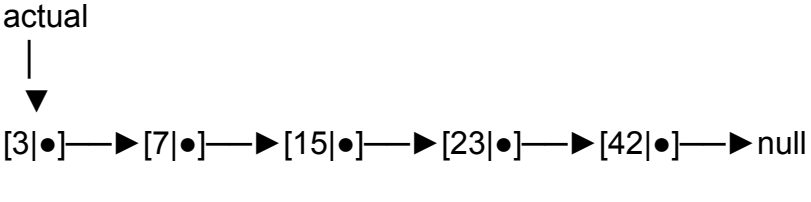


Paso 1: Crear nuevo nodo



Paso 2: Recorrer hasta el último nodo

Se utiliza un puntero auxiliar llamado "actual":



Se avanza nodo por nodo hasta que:

actual.next == null

Al finalizar el recorrido:

actual
  |
  ▼
[42|●]──►null

Paso 3: Enlazar el nuevo nodo

actual
  |
  ▼
[42|●]──►[50|●]──►null

Estado final

cabeza
  |
  ▼
[3|●]──►[7|●]──►[15|●]──►[23|●]──►[42|●]──►[50|●]──►null

Explicación de complejidad: ¿Por qué es O(n)?

Insertar al final requiere recorrer la lista completa para encontrar el último nodo. Si la lista tiene n elementos, en el peor caso se deben visitar los n nodos.

El tiempo de ejecución crece proporcionalmente al tamaño de la lista. Por ello, la complejidad es O(n).

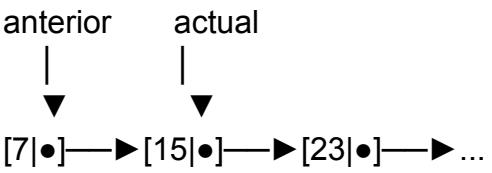
Solo sería O(1) si la estructura mantuviera un puntero adicional al último nodo.

3. Eliminación del nodo con valor 15

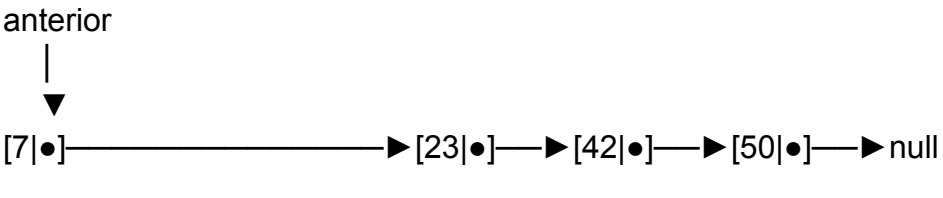
Lista actual:

cabeza
  |
  ▼
[3|●]──►[7|●]──►[15|●]──►[23|●]──►[42|●]──►[50|●]──►null

Paso 1: Recorrer con punteros anterior y actual



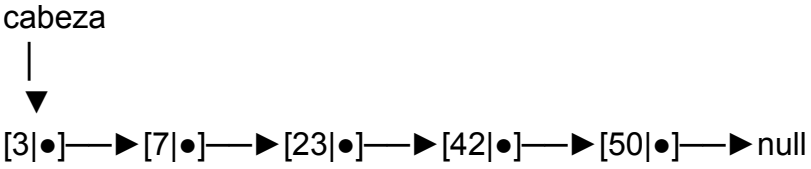
Paso 2: Saltar el nodo encontrado



Se realiza:

anterior.next = actual.next

Estado final

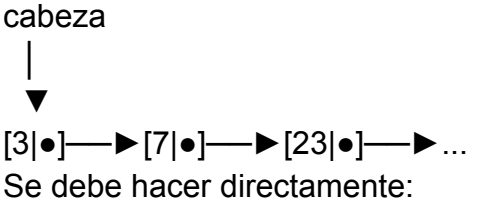


Problema especial al eliminar el primer nodo

Cuando se elimina un nodo intermedio, siempre existe un nodo "anterior" que permite reajustar el enlace.

Sin embargo, cuando se elimina el primer nodo (la cabeza), no existe un nodo anterior. Esto obliga a tratar el caso como una situación especial.

Por ejemplo, si se eliminara el nodo 3:



Se debe hacer directamente:

cabeza = cabeza.next

El problema es que si no se actualiza correctamente la referencia de cabeza, la lista pierde su punto de acceso y se vuelve inaccesible. Este riesgo no ocurre con nodos intermedios, ya que la cabeza permanece intacta.

En resumen:

- Los nodos intermedios se eliminan reajustando enlaces.
- El primer nodo se elimina modificando la referencia principal de la estructura.
- Por eso requiere un tratamiento especial en la implementación.

Conclusión General

En una lista enlazada simple, la eficiencia de las operaciones depende directamente de la cantidad de punteros que deben modificarse y de si es necesario recorrer la estructura.

Insertar al inicio es eficiente porque solo implica cambios locales.
Insertar al final es más costoso porque exige recorrer la lista.
Eliminar el primer nodo es un caso especial porque afecta la referencia principal que da acceso a toda la estructura.

Comprender estos detalles permite analizar correctamente la complejidad y diseñar implementaciones más robustas.

2-Implementación Guiada de Lista Enlazada

Entregables esperado

-Código fuente completo de tu implementación con: (1) Clase Nodo, (2) Clase ListaEnlazada con al menos 6 métodos funcionales, (3) Programa de prueba que demuestre cada operación, (4) Comentarios explicando la lógica de cada método.

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None


class ListaEnlazada:
    def __init__(self):
        self.cabeza = None
        self.tamano = 0
```

```
def estaVacia(self):
    return self.cabeza is None

def insertarInicio(self, valor):
    nuevo = Nodo(valor)
    nuevo.siguiente = self.cabeza
    self.cabeza = nuevo
    self.tamano += 1

def insertarFinal(self, valor):
    nuevo = Nodo(valor)

    if self.estaVacia():
        self.cabeza = nuevo
    else:
        actual = self.cabeza
        while actual.siguiente is not None:
            actual = actual.siguiente
        actual.siguiente = nuevo

    self.tamano += 1

def eliminarInicio(self):
    if self.estaVacia():
        raise Exception("La lista está vacía")

    valor_eliminado = self.cabeza.dato
    self.cabeza = self.cabeza.siguiente
    self.tamano -= 1
    return valor_eliminado

def buscar(self, valor):
    actual = self.cabeza
    while actual is not None:
        if actual.dato == valor:
            return True
        actual = actual.siguiente
    return False

def imprimir(self):
    actual = self.cabeza
    while actual is not None:
        print(actual.dato, end=" -> ")
```

```
            actual = actual.siguiente
        print("None")

if __name__ == "__main__":
    lista = ListaEnlazada()

    print("Insertando al inicio:")
    lista.insertarInicio(10)
    lista.insertarInicio(5)
    lista.imprimir()

    print("\nInsertando al final:")
    lista.insertarFinal(20)
    lista.insertarFinal(30)
    lista.imprimir()

    print("\nBuscar 20:")
    print(lista.buscar(20))

    print("\nEliminar inicio:")
    print("Eliminado:", lista.eliminarInicio())
    lista.imprimir()

    print("\nTamaño actual:", lista.tamano)
```

3-Resolver Problemas Clásicos con Depuración Visual

Entregables esperado

-Implementación de al menos 2 problemas clásicos (invertir lista + encontrar medio) con: (1) Código funcional probado, (2) Documento explicando el algoritmo paso a paso en tus propias palabras, (3) Análisis de complejidad temporal y espacial.

-(1)CÓDIGO FUNCIONAL PROBADO

Problema 1:INVERTIR LISTA ENLAZADA

```
def imprimir_lista(cabeza):
    while cabeza:
        print(cabeza.valor, end=" -> ")
        cabeza = cabeza.next
    print("null")


# Crear lista A -> B -> C -> D
```

```
A = Nodo("A")
B = Nodo("B")
C = Nodo("C")
D = Nodo("D")

A.next = B
B.next = C
C.next = D

print("Original:")
imprimir_lista(A)

nueva_cabeza = invertir_lista(A)

print("Invertida:")
imprimir_lista(nueva_cabeza)
```

Problema 2:ENCONTRAR EL NODO MEDIO

```
def encontrar_medio(cabeza):
    lento = cabeza
    rapido = cabeza

    while rapido and rapido.next:
        lento = lento.next
        rapido = rapido.next.next

    return lento
```

-(2)DOCUMENTO EXPLICANDO EL ALGORITMO PASO A PASO EN TUS PROPIAS PALABRAS

Las listas enlazadas son estructuras de datos dinámicas donde cada elemento (nodo) contiene un valor y una referencia al siguiente nodo. A diferencia de los arreglos, no almacenan los elementos en posiciones contiguas de memoria, lo que permite inserciones y eliminaciones eficientes sin desplazamientos.

En este documento se analizan detalladamente dos algoritmos clásicos:

1. Invertir una lista enlazada.
2. Encontrar el nodo medio de una lista enlazada.

Se estudiará su funcionamiento paso a paso, su lógica interna y su complejidad computacional.

Algoritmo para Invertir una Lista Enlazada
1. Planteamiento del problema

Dada una lista enlazada simple:

A → B → C → D → null

Se desea transformarla en:

D → C → B → A → null

El objetivo es invertir la dirección de todos los enlaces sin utilizar estructuras auxiliares adicionales.

2.Fundamento del algoritmo

Para invertir la lista correctamente, es necesario recorrerla nodo por nodo y cambiar la dirección de cada enlace. Sin embargo, al modificar un enlace se pierde la referencia al resto de la lista si no se guarda previamente.

Por ello, se emplean tres punteros:

- anterior: almacena la parte ya invertida.
- actual: nodo que se está procesando.
- siguiente: guarda temporalmente el siguiente nodo antes de modificar el enlace.

3. Desarrollo paso a paso
Estado inicial

anterior = None
actual = cabeza (A)

Visualmente:

anterior → None
actual → A → B → C → D → null

Iteración 1 (procesando A)

Guardar el siguiente nodo:
siguiente = actual.next # B

Invertir el enlace:
actual.next = anterior

Ahora:
A → None

Avanzar el puntero anterior:
anterior = actual

Avanzar el puntero actual:
actual = siguiente

Estado parcial:
Invertida: A → null
Pendiente: B → C → D → null

Iteración 2 (procesando B)

1. Se repite el mismo patrón:
2. Guardar C.
3. Hacer que B apunte a A.
4. Mover anterior a B.
5. Mover actual a C.

Resultado parcial:
B → A → null

Iteraciones siguientes
Se repite el mismo procedimiento con C y luego con D.
Al finalizar:
D → C → B → A → null

Cuándo actual se vuelve None, el ciclo termina y el anterior apunta al nuevo inicio.

Análisis de complejidad

- Tiempo: O(n)
  Se recorre cada nodo exactamente una vez.
- Espacio: O(1)
  Solo se utilizan tres punteros adicionales.

Algoritmo para Encontrar el Nodo Medio

1. Planteamiento del problema

Dada una lista:

A → B → C → D → E → null

Se desea encontrar el nodo central (C) sin conocer previamente la longitud de la lista.

2. Fundamento del algoritmo

Se emplea la técnica conocida como “puntero lento y puntero rápido”.

- El puntero lento avanza un nodo por iteración.
- El puntero rápido avanza dos nodos por iteración.

Cuando el puntero rápido llega al final, el lento se encuentra en la mitad.

3. Desarrollo paso a paso

Estado inicial

lento → A

rapido → A

Iteración 1

lento → B

rapido → C

Iteración 2

lento → C

rapido → E

Iteración 3

El puntero rápido ya no puede avanzar dos posiciones.

El ciclo termina.

Resultado:

lento → C

Análisis de complejidad

- Tiempo: O(n)
  El recorrido total sigue siendo lineal.
- Espacio: O(1)
  No se utilizan estructuras adicionales.

4-Análisis de Casos de Uso Reales

Entregables esperado

-Documento comparativo que incluya: (1) Tu análisis inicial de cada escenario ANTES de consultar la IA, (2) El análisis de la IA, (3) Tu reflexión sobre las diferencias entre ambos análisis, (4) Conclusiones sobre cuándo usar cada estructura.

Análisis inicial	**Análisis de la IA**	**Reflexión sobre diferencias**	**Conclusiones**
Antes de consultar análisis formal, mi intuición sería: Playlist → Lista enlazada porque permite insertar en medio fácilmente. Buffer → Cola. Versiones → Pila o arreglo. Este análisis se basa en asociaciones conceptuales: Playlist = insertar mucho → lista. Buffer = FIFO → cola. Versiones = historial → stack. Pero esto es superficial.	La IA evalúa: Frecuencia real de operaciones Complejidad temporal Localidad de memoria Costos ocultos (reallocación, punteros) Tamaño del dataset Resultados: Playlist → Arreglo dinámico (porque acceso O(1) domina) Buffer → Arreglo circular (latencia y caché) Versiones → Lista doble (navegación eficiente) La diferencia clave es que el análisis formal prioriza la operación más frecuente, no la más llamativa.	Mi análisis intuitivo: Se basa en el tipo de acción (insertar, FIFO, historial) El análisis arquitectónico: Se basa en frecuencia y complejidad amortizada Considera memoria y caché Evalúa tamaño real del problema Lección importante: No diseñamos para lo posible. Diseñamos para lo frecuente. Muchos sistemas fallan porque optimizan la operación equivocada.	Usa Arreglo cuando: ● Necesitas acceso por índice O(1) ● El tamaño es pequeño o moderado ● Importa la localidad de memoria ● Las inserciones en medio no son dominantes Usa Lista Enlazada Simple cuando: ● Insertas frecuentemente al inicio ● No necesitas navegación hacia atrás ● No necesitas acceso por índice Usa Lista Doblemente Enlazada cuando: ● Necesitas navegar adelante y atrás ● Eliminaciones frecuentes con referencia directa ● No necesitas acceso aleatorio Usa Arreglo Circular cuando: ● Tamaño máximo fijo ● Operaciones FIFO ● Latencia crítica

5-Validación de Implementación con Técnica de "Rubber Duck"

Entregables esperado

-Transcripción de la sesión de "rubber duck debugging" con al menos 3 métodos diferentes. Para cada método, incluye: (1) El código original, (2) Las preguntas de la IA y tus respuestas, (3) Bugs o mejoras que descubriste durante la explicación, (4) Código corregido si aplica.

Eliminar por posición:

```
import weakref
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
    def __del__(self):
        print(f"¡Nodo con dato {self.dato} está siendo eliminado!")

class ListaEnlazada:
    # ... (mismos métodos que antes)
    pass

# Crear lista con un elemento
lista = ListaEnlazada()
lista.agregar_al_final(42)
print("Estado inicial:")
print(f"lista.cabeza: {lista.cabeza}")
print(f"lista.tamaño: {lista.tamaño}")

print("\nEliminando el nodo...")
dato_eliminado = lista.eliminar_por_posicion(0)

print("\nEstado final:")
print(f"lista.cabeza: {lista.cabeza}")
print(f"lista.tamaño: {lista.tamaño}")
print(f"Dato eliminado: {dato_eliminado}")

# Forzar recolección de basura para ver cuándo se elimina el nodo
import gc
print("\nForzando recolección de basura...")
gc.collect()
```

Me gusta cómo estás razonando. Ahora quiero ver si puedes detectar qué partes de tu diseño están **garantizadas por código** y cuáles están **basadas en disciplina de uso** 👀

o más fascinante es que las listas enlazadas en Python (y en la mayoría de lenguajes) son una construcción conceptual más que una garantía del lenguaje:

Lo garantizado: Las referencias entre objetos, la ejecución secuencial del código, las validaciones explícitas

Lo basado en disciplina: La estructura lógica de la lista, la invariante de que los nodos están correctamente enlazados, la consistencia del contador

En lenguajes como Rust, esto sería diferente - el sistema de tipos podría garantizar más propiedades. En Python, confiamos en:

Nuestro cuidado al implementar

Las pruebas que escribamos

La documentación clara

La disciplina de quien use nuestra clase (no meter mano donde no de

Me está gustando cómo estás pensando esto.
Ahora quiero que respondas esta:

👉 ¿Cuál es la invariante más frágil de tu implementación y por qué?

La lección es que en programación, las invariantes que son:

Redundantes (no derivables de la estructura)

No verificables por el compilador

Requieren actualización manual en múltiples puntos

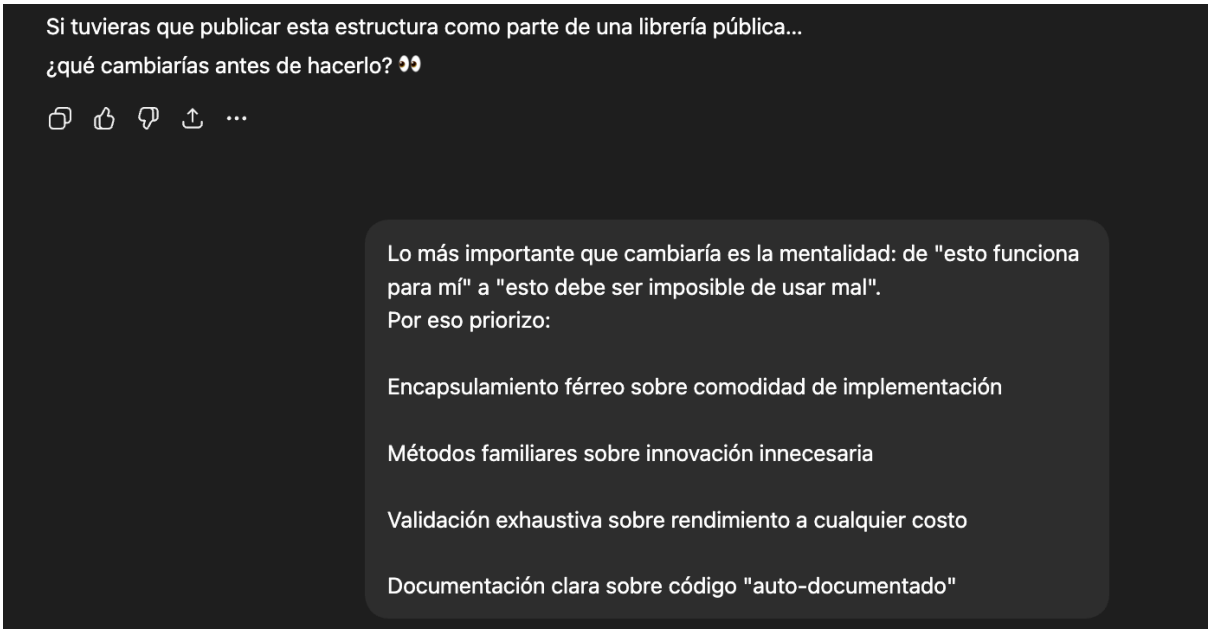
...son las que más riesgo corren de romperse. En nuestro caso, self.tamaño es el claro ejemplo de una optimización que introduce complejidad y fragilidad a cambio de eficiencia.

Encapsulamiento Fuerte

```
class ListaEnlazada:
    def __init__(self):
        self._cabeza = None
        self._tamaño = 0

    def __iter__(self):
        """Permite iteración sin exponer nodos"""
        actual = self._cabeza
        while actual:
            yield actual._dato
            actual = actual._siguiente

    # Sin propiedades públicas de _cabeza o _tamaño
    # Sin getters que expongan nodos
```

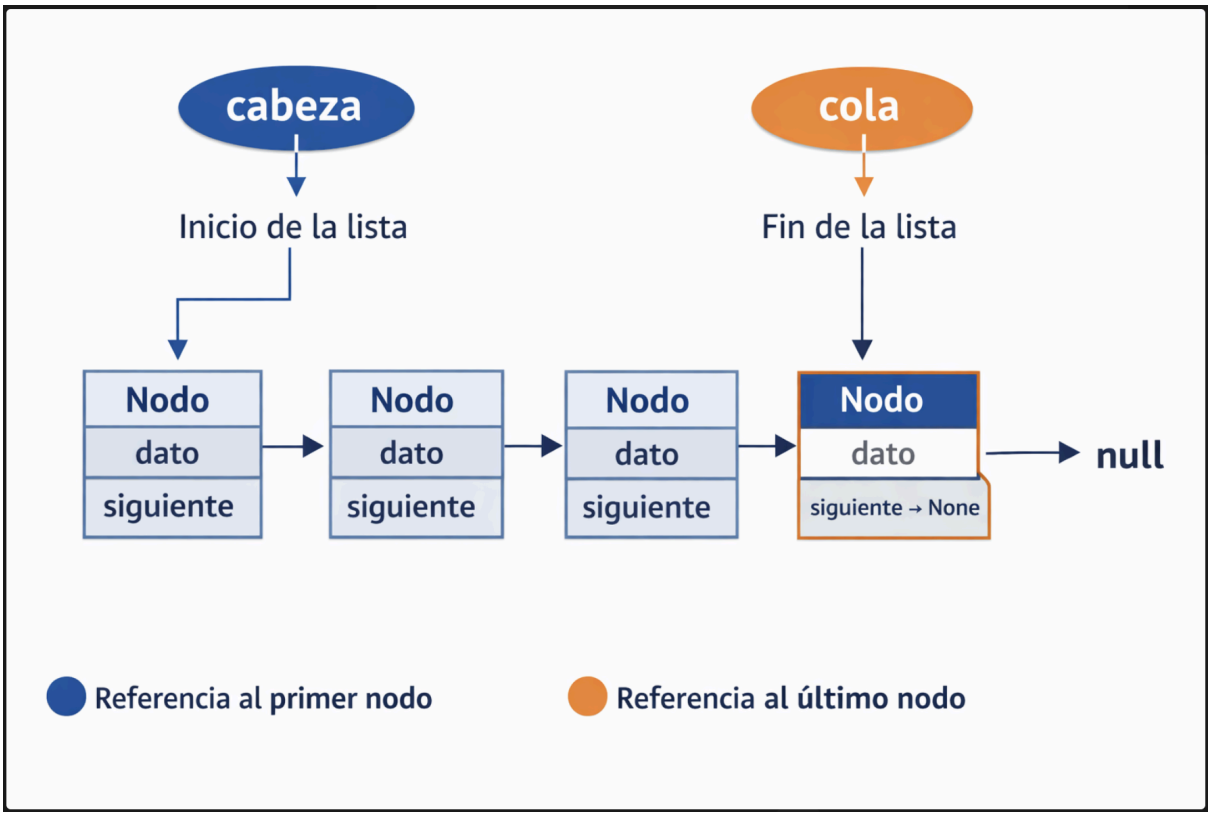


6-Diseño de Sistema: Lista con Acceso O(1) al Final

Entregables esperado

-Implementación mejorada de ListaEnlazada con referencia a cola que incluya: (1) Diagrama de la nueva estructura, (2) Código modificado con todos los métodos actualizados para mantener 'cola' consistente, (3) Análisis de cómo cambió la complejidad de cada operación, (4) Pruebas que demuestren que insertarFinal() ahora es O(1).

Diagrama:



Código modificado:

Clase Nodo

```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
```

Clase ListaEnlazada Mejorada

```
class ListaEnlazada:

    def __init__(self):
        self.cabeza = None
        self.cola = None
        self.tamano = 0
```

```
insertarInicio()
def insertarInicio(self, dato):
        nuevo = Nodo(dato)

        if self.cabeza is None:  # Lista vacía
            self.cabeza = nuevo
            self.cola = nuevo
        else:
```

```
            nuevo.siguiente = self.cabeza
            self.cabeza = nuevo

        self.tamano += 1

insertarFinal()  O(1)
    def insertarFinal(self, dato):
        nuevo = Nodo(dato)

        if self.cabeza is None:  # Lista vacía
            self.cabeza = nuevo
            self.cola = nuevo
        else:
            self.cola.siguiente = nuevo
            self.cola = nuevo

        self.tamano += 1

eliminarInicio()
    def eliminarInicio(self):
        if self.cabeza is None:
            return None

        eliminado = self.cabeza
        self.cabeza = self.cabeza.siguiente

        if self.cabeza is None:  # La lista quedó vacía
            self.cola = None

        self.tamano -= 1
        return eliminado.dato

eliminarFinal() (sigue siendo O(n))
    def eliminarFinal(self):
        if self.cabeza is None:
            return None

        if self.cabeza == self.cola:
            dato = self.cabeza.dato
            self.cabeza = None
            self.cola = None
            self.tamano -= 1
            return dato
```

```
        actual = self.cabeza

        while actual.siguiente != self.cola:
            actual = actual.siguiente

        dato = self.cola.dato
        actual.siguiente = None
        self.cola = actual
        self.tamano -= 1
        return dato

mostrar()
    def mostrar(self):
        actual = self.cabeza
        while actual:
            print(actual.dato, end=" → ")
            actual = actual.siguiente
        print("null")
```

Análisis de complejidad:

Operación	**Antes**	**Ahora**	**Explicación**
insertarInicio	O(1)	O(1)	No requiere recorrido
insertarFinal	O(n)	O(1)	Acceso directo a cola
eliminarInicio	O(1)	O(1)	Solo reasigna punteros
eliminarFinal	O(n)	O(n)	Se necesita nodo anterior
recorrer	O(n)	O(n)	No cambia

Pruebas que Demuestran que insertarFinal() es O(1)

Prueba Teórica

En cada llamada a insertarFinal():

Operaciones realizadas:

1. Crear nodo → O(1)
2. Asignar self.cola.siguiente → O(1)
3. Actualizar self.cola → O(1)

No hay ciclos.

No depende del tamaño n.

Por definición formal: tiempo constante.