

## 1. Insertar el valor 3 al INICIO

### Estado ANTES

```
css
cabeza
|
v
[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### Paso 1: Crear el nuevo nodo

```
csharp
nuevo
|
v
[3|•]→ null
```

[Copiar código](#)

### Paso 2: Apuntar el nuevo nodo al antiguo primero

```
css
nuevo
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### Paso 3: Mover `cabeza` al nuevo nodo

```
css
cabeza
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### Estado DESPUÉS

```
css
cabeza
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

🕒 **Complejidad temporal:  $O(1)$**

👉 No importa el tamaño de la lista. Solo se mueven punteros.

# 1. Visualización de lista enlazada

¿Por qué insertar al inicio es  $O(1)$ ?

Porque para agregar un nuevo nodo a la lista solo es necesario generar un puntero que haga referencia al antiguo primer nodo y cambiar la cabeza de la lista. Son solo dos operaciones.

## 2. Insertar el valor 50 al FINAL

### Estado ANTES

```
css
cabeza
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→ null
```

 Copiar código

### Paso 1: Recorrer la lista hasta el final

```
csharp
actual
|
v
[42|•]→ null
```

 Copiar código

### Paso 2: Crear el nuevo nodo

```
csharp
nuevo
|
v
[50|•]→ null
```

 Copiar código

### Paso 3: Apuntar el último nodo al nuevo

```
csharp
[42|•]→[50|•]→ null
```

 Copiar código

### Estado DESPUÉS

```
css
cabeza
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→[50|•]→ null
```

 Copiar código

### Complejidad temporal: $O(n)$

👉 Hay que recorrer **toda la lista** para encontrar el último nodo.

¿Por qué insertar al final es  $O(n)$ ?

Porque se debe recorrer toda la lista hasta llegar al último nodo (se sabe que es el último porque hace referencia a NULL).

### 3. Eliminar el nodo con valor 15

#### Estado ANTES

```
css
cabeza
|
v
[3|•]→[7|•]→[15|•]→[23|•]→[42|•]→[50|•]→null
```

Copiar código

#### Paso 1: Buscar el nodo (manteniendo anterior y actual)

```
css
anterior  actual
|         |
v         v
[7|•]→[15|•]→[23|•]
```

Copiar código

#### Paso 2: Saltar el nodo a eliminar

👉 `anterior.next = actual.next`

```
css
[7|•]→[23|•]
```

Copiar código

#### Paso 3: El nodo 15 queda desconectado

```
csharp
[15|•]→null (liberado)
```

Copiar código

#### Estado DESPUÉS

```
css
cabeza
|
v
[3|•]→[7|•]→[23|•]→[42|•]→[50|•]→null
```

Copiar código

#### ⌚ Complejidad temporal: $O(n)$

👉 Es necesario recorrer la lista para **encontrar** el nodo 15.

Se recorre toda la lista hasta encontrar el nodo deseado, después, se referencia el nodo siguiente al nodo deseado desde el nodo anterior, lo que causa que el nodo deseado quede desconectado.

## Eliminar el PRIMER nodo (valor 7)

En una lista simple, eliminar el primer nodo es **especial** porque **NO** hay **nodo anterior**.

### ♦ ANTES

```
css
cabeza
|
v
[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### ♦ Paso 1: Guardar referencia al nodo a eliminar

```
css
actual
|
v
[7|•]→[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

(esto es opcional conceptualmente, pero útil para entender memoria)

### ♦ Paso 2: Mover `cabeza` al siguiente nodo

👉 `cabeza = cabeza.next`

```
css
cabeza
|
v
[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### ♦ Paso 3: El nodo eliminado queda aislado

```
csharp
[7|•]→ null
```

[Copiar código](#)

## ✅ DESPUÉS

```
css
cabeza
|
v
[15|•]→[23|•]→[42|•]→ null
```

[Copiar código](#)

### 🕒 Complejidad temporal: $O(1)$

👉 No se recorre la lista

👉 Solo se mueve el puntero `cabeza`

¿Qué problema surge al eliminar el primer nodo que no ocurre con otros nodos?

Es esencial cambiar la cabeza de la lista, si se elimina el primer nodo sin hacer eso entonces la lista se corromperá y dejará de funcionar.

# Checkpoint Metacognitivo - Fase Comprende

¿Puedo explicar en mis propias palabras por qué una lista enlazada no permite acceso  $O(1)$  por índice, aunque los arreglos sí?

Porque los elementos están enlazados entre sí y, como las posiciones de memoria no son contiguas y están dispersas, es necesario recorrer cada nodo para dar con el nodo objetivo.

Si me piden insertar un elemento entre el nodo 3 y el nodo 4 de una lista, ¿sé qué punteros debo modificar sin consultar código?

Debo modificar el puntero del nodo 3 y el puntero del nuevo nodo. En la nueva estructura el nodo 3 haría referencia al nuevo nodo y el nuevo nodo haría referencia al nodo 4 antiguo.

¿Entiendo por qué necesito referencia al nodo ANTERIOR para eliminar un nodo en una lista simple?

Sí, es porque para “eliminar” un nodo realmente lo que se debe hacer es desconectarlo de la lista, modificando el puntero del nodo anterior para así lograr la desconexión.

## 2. Implementación Guiada de Lista Enlazada

```
// =====  
// Clase Nodo  
// =====  
class Nodo {  
    constructor(dato) {  
        // Dato que almacenará el nodo  
        this.dato = dato;  
  
        // Referencia al siguiente nodo (inicialmente null)  
        this.siguiente = null;  
    }  
}  
  
// =====  
// Clase ListaEnlazada  
// =====  
class ListaEnlazada {  
    constructor() {
```

```
// Apunta al primer nodo de la lista
this.cabeza = null;

// Lleva la cuenta de cuántos elementos hay
this.tamaño = 0;
}

// -----
// Verifica si la lista está vacía
// -----
estaVacía() {
    // La lista está vacía si no tiene elementos
    return this.tamaño === 0;
}

// -----
// Inserta un valor al inicio de la lista
// -----
insertarInicio(valor) {
    // Se crea un nuevo nodo
    const nuevoNodo = new Nodo(valor);

    // Enlazamos el nuevo nodo con la antigua cabeza
    nuevoNodo.siguiente = this.cabeza;

    // Actualizamos la cabeza
    this.cabeza = nuevoNodo;

    // Aumentamos el tamaño de la lista
    this.tamaño++;
}

// -----
// Inserta un valor al final de la lista
// -----
insertarFinal(valor) {
    // Se crea un nuevo nodo
    const nuevoNodo = new Nodo(valor);

    // Si la lista está vacía, el nuevo nodo es la cabeza
```

```

    if (this.estaVacia()) {
        this.cabeza = nuevoNodo;
    } else { // Si no
        // Se crea una variable para guardar el nodo de la cabeza
actual
        let actual = this.cabeza;

        // Se recorre toda la lista hasta el último nodo
        while (actual.siguiente !== null) {
            actual = actual.siguiente;
        }

        // Enlazamos el último nodo con el nuevo
        actual.siguiente = nuevoNodo;
    }

    // Incrementamos el tamaño de la lista
    this.tamaño++;
}

// -----
// Elimina el primer elemento de la lista
// -----
eliminarInicio() {
    // Si la lista está vacía, no se elimina nada
    if (this.estaVacia()) {
        return null;
    }

    // Se guarda el nodo que será eliminado
    const nodoEliminado = this.cabeza;

    // Movemos la cabeza al siguiente nodo
    this.cabeza = this.cabeza.siguiente;

    // Reducimos el tamaño de la lista
    this.tamaño--;

    // Se devuelve el valor eliminado
    return nodoEliminado.dato;
}

```

```

}

// -----
// Busca un valor dentro de la lista
// -----
buscar(valor) {
    // Se guarda el nodo de la cabeza actual
    let actual = this.cabeza;

    // Se recorre nodo por nodo
    while (actual !== null) {
        // Se compara el dato de cada nodo con el valor buscado
        if (actual.dato === valor) {
            // Si se encuentra, se devuelve true
            return true;
        }
        // Se cambia el valor de la variable al valor del siguiente
nodo
        actual = actual.siguiente;
    }

    // Si se llega al final sin encontrarlo, se devuelve false
    return false;
}

// -----
// Imprime todos los elementos de la lista
// -----
imprimir() {
    // Si la lista está vacía, se muestra un mensaje y sale del método
    if (this.estaVacía()) {
        console.log("Lista vacía");
        return;
    }

    // Se guarda la cabeza actual
    let actual = this.cabeza;
    // Se crea una variable para concatenar los resultados
    let resultado = "";

```



```

    // Se recorre la lista desde la cabeza
    while (actual !== null) {
        // Se concatenan los resultados
        resultado += actual.dato + " -> ";
        // Se pasa al siguiente nodo
        actual = actual.siguiente;
    }

    // Se termina en "null"
    resultado += "null";

    // Imprime el resultado
    console.log(resultado);
}

const lista = new ListaEnlazada();

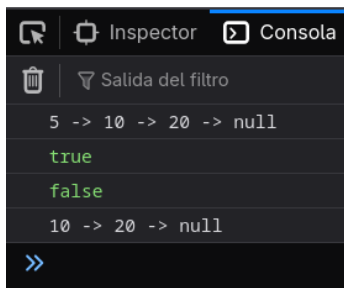
lista.insertarInicio(10);
lista.insertarInicio(5);
lista.insertarFinal(20);

lista.imprimir();           // 5 -> 10 -> 20 -> null

console.log(lista.buscar(10)); // true
console.log(lista.buscar(99)); // false

lista.eliminarInicio();
lista.imprimir();           // 10 -> 20 -> null

```



### 3. Resolver problemas clásicos con depuración visual

#### Algoritmo para invertir lista en JavaScript

```
// =====  
// Definición del nodo  
// =====  
class Nodo {  
    constructor(valor) {  
        this.valor = valor;  
        this.next = null;  
    }  
}  
  
// =====  
// Función para imprimir  
// =====  
function imprimirLista(cabeza) {  
    let actual = cabeza;  
    let resultado = "";  
  
    while (actual !== null) {  
        resultado += `[${actual.valor}] → `;  
        actual = actual.next;  
    }  
  
    resultado += "null";  
    console.log(resultado);  
}  
  
// =====  
// Función para invertir  
// =====  
function invertirLista(cabeza) {  
    let anterior = null;  
    let actual = cabeza;  
    while (actual !== null) { // Se repetirá mientras haya nodos sin  
        procesar
```

```

        let siguiente = actual.next; // Guarda el valor del siguiente nodo
        actual.next = anterior; // Se invierte el puntero que apunta al
siguiente nodo
        anterior = actual; // Mueve el nodo anterior una posición hacia
adelante
        actual = siguiente; // Mueve el nodo actual una posición hacia
adelante
    }
    return anterior; // Regresa la nueva cabeza
}

```

```

// Crear lista: A → B → C → D
let A = new Nodo("A");
let B = new Nodo("B");
let C = new Nodo("C");
let D = new Nodo("D");

// Asigna punteros de nodos siguientes
A.next = B;
B.next = C;
C.next = D;

```

```

let cabeza = A; // Crea la cabeza

console.log("Lista original:");
imprimirLista(cabeza);

let nuevaCabeza = invertirLista(cabeza);

console.log("Lista invertida:");
imprimirLista(nuevaCabeza);

```

```

Lista original:
[A] → [B] → [C] → [D] → null
Lista invertida:
[D] → [C] → [B] → [A] → null
>

```

Complejidad temporal:  $O(n)$ . Complejidad espacial:  $O(1)$

## Algoritmo para encontrar el medio de una lista en JavaScript

```
// =====  
// Definición del nodo  
// =====  
class Nodo {  
  constructor(valor) {  
    this.valor = valor;  
    this.next = null;  
  }  
}  
  
// =====  
// Función para imprimir  
// =====  
function imprimirLista(cabeza) {  
  let actual = cabeza;  
  let resultado = "";  
  
  while (actual !== null) {  
    resultado += `[${actual.valor}] → `;  
    actual = actual.next;  
  }  
  
  resultado += "null";  
  console.log(resultado);  
}  
  
// =====  
// Función para encontrar medio  
// =====  
function encontrarMedio(cabeza) {  
  // Inicializa ambos punteros con la cabeza de la lista  
  let lento = cabeza;  
  let rapido = cabeza;  
  
  // Mientras el puntero rápido pueda avanzar dos posiciones  
  while (rapido !== null && rapido.next !== null) {
```

```

        lento = lento.next; // Puntero lento avanza un nodo
        rapido = rapido.next.next; // Puntero rápido avanza dos nodos
    }

    // Cuando rápido llegue al final, lento estará en el medio
    return lento;
}

// Crear lista: A → B → C → D
let A = new Nodo("A");
let B = new Nodo("B");
let C = new Nodo("C");
let D = new Nodo("D");

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

let cabeza = A;

console.log("Lista:");
imprimirLista(cabeza);

let medio = encontrarMedio(cabeza);

console.log("Nodo medio:", medio.valor);

```

```

Lista:
[A] → [B] → [C] → [D] → null
Nodo medio: C

```

Complejidad temporal:  $O(n)$ , porque necesita recorrer la lista, aunque no la recorra en su totalidad. Complejidad espacial:  $O(1)$ , porque solo utiliza 2 punteros, no crea arreglos nuevos.

## Checkpoint Metacognitivo - Aplica

¿Puedo implementar insertarInicio() sin mirar el código? ¿Qué línea escribiría primero y por qué en ese orden?

Sin consultarlo de nuevo no podría, necesito repasarlo más.

Si mi método eliminar() no funciona para el primer elemento, ¿puedo identificar qué caso especial olvidé?

Confirmar que la lista no esté vacía.

¿Cuánto del código escribí yo versus cuánto copié directamente de la IA sin modificar?

Todo el código fue generado por la IA, pues yo aún no domino el lenguaje de programación que elegí (JavaScript).

## 4. Análisis de casos de uso reales

### Sistema de playlist de música

#### Mi hipótesis

Considerando que el usuario puede saltar a cualquier canción de acuerdo a su número, muy probablemente la mejor estructura de datos sería el arreglo, porque al saltar a cualquier canción está usando el acceso directo por índice, la cual es una característica que solo tienen los arreglos.

#### Análisis de la IA

##### 5 Justificación técnica

Operación	Arreglo dinámico	Lista enlazada
Acceso por índice	$O(1)$	$O(n)$
Agregar al final	$O(1)$ amortizado	$O(1)$
Insertar en medio	$O(n)$	$O(n)$
Eliminar en medio	$O(n)$	$O(n)$

El punto decisivo es:

El acceso por índice es frecuente y debe ser  $O(1)$ .

En una lista enlazada, saltar a la canción 300 es  $O(n)$ .

En un arreglo es acceso directo en memoria contigua.

Además:

- 500 elementos es pequeño → el costo de desplazamiento  $O(n)$  no es crítico.
- Mejor localidad de caché en arreglos → mejor rendimiento real.

🔴 Conclusión: **Arreglo dinámico gana por acceso aleatorio eficiente y mejor performance práctica.**

#### Reflexión

Parece que ambos usamos un razonamiento similar y dimos con la misma respuesta

# Buffer de teclado

## Mi hipótesis

No estoy muy seguro, pero como debe “insertar” los caracteres uno tras otro creo que lo mejor sería usar un arreglo, que puede insertar elementos al final de manera directa  $[O(1)]$ .

## Análisis de la IA

### 4 Recomendación

✓ Arreglo circular (Queue basada en array)

### 5 Justificación técnica

Un arreglo circular permite:

- Insertar:  $O(1)$
- Remover:  $O(1)$
- Sin desplazamientos
- Sin asignaciones dinámicas
- Memoria contigua → excelente para CPU cache

Comparación:

Estructura	Insertar	Remover	Overhead memoria	Latencia
Arreglo circular	$O(1)$	$O(1)$	Bajo	Muy baja
Lista enlazada	$O(1)$	$O(1)$	Alto (punteros)	Mayor

El puntero extra en listas enlazadas:

- Duplica uso de memoria por carácter
- Genera fragmentación
- Empeora cache locality

⚠ En sistemas críticos (drivers, OS, sistemas embebidos), SIEMPRE se usa buffer circular.

## Reflexión

No consideré que la lista enlazada duplica el uso de memoria por carácter y que no aprovecha el caché del procesador, pero al final tuve razón, aunque haya sido a medias.

## Sistema de versiones de documentos

## Mi hipótesis

Creo que también elegiría el arreglo, porque se tienen que insertar las nuevas versiones al final.

## Análisis de la IA

En lista doble:

- Navegación bidireccional natural
- Semántica clara tipo historial
- No necesitamos redimensionamientos

Además, como los documentos son grandes, el costo del puntero extra es irrelevante.

## Reflexión

Como no es necesario el acceso aleatorio, parece ser que lo mejor sería una lista enlazada doble que mantenga en un puntero la posición del último elemento, para así poder acceder fácilmente a la última versión.

## Conclusiones

Siempre que el acceso aleatorio sea muy frecuente, o sea necesario insertar al final, lo mejor será utilizar un arreglo. Pero cuando se mencione un patrón secuencial, lo ideal será usar la lista enlazada.

## Checkpoint metacognitivo - Reflexiona

Si alguien me pregunta "¿es mejor usar arreglo o lista enlazada?", ¿puedo responder "depende" y explicar de qué depende?

En los casos comunes sería capaz de hacerlo.

¿Puedo identificar al menos 2 escenarios donde elegir la estructura incorrecta causaría problemas graves de rendimiento?

Sí, en un escenario donde sea muy importante tener una baja latencia, usar una lista enlazada no sería adecuado. Además, usar una lista enlazada en un sistema con poca memoria sería desastroso por culpa de la sobrecarga de memoria que traen consigo los punteros.

Cuando la IA me dio una recomendación, ¿la acepté automáticamente o cuestioné su razonamiento?

No la cuestioné porque sus explicaciones tuvieron sentido para mí.



# 5. Validación de implementación con técnica Rubber Duck

## Método insertarInicio

```
insertarInicio(valor) {  
    const nuevoNodo = new Nodo(valor);  
    nuevoNodo.siguiente = this.cabeza;  
    this.cabeza = nuevoNodo;  
    this.tamaño++;  
}
```

¿Puedes explicarme qué hace la primera línea de tu código y por qué es necesaria?

```
js  
const nuevoNodo = new Nodo(valor);
```

📄 📌 🗑️ ↻ ⋮

Crea un nuevo nodo de la lista enlazada, es necesario porque el método pretende insertar un nuevo nodo al inicio

Bien, entonces estás creando una nueva instancia para poder insertarla »»

Déjame preguntarte algo para profundizar un poco más:

Cuando haces `new Nodo(valor)`, ¿qué propiedades esperas que tenga ese nodo recién creado exactamente?

Y más importante aún... en ese momento justo después de crearlo, ¿qué valor tiene su propiedad `siguiente`?

< 2/2 > 📄 📌 🗑️ ↻ ⋮

va a contener un valor dado cuando sea llamado el método. Inicialmente siguiente es null

Perfecto, entonces tu `Nodo` inicia con:

- `valor` con el dato que recibes
- `siguiente` en `null`

Ahora vayamos a la siguiente línea:

```
js  
nuevoNodo.siguiente = this.cabeza;
```

Si inicialmente `siguiente` es `null`, ¿por qué necesitas reasignarlo aquí?

Y quiero que pienses en este caso específico:

👉 ¿Qué pasa si la lista está vacía cuando llamas `insertarInicio`?

Explicame qué valor tendría `this.cabeza` en ese momento y qué terminaría guardándose en `nuevoNodo.siguiente`.

📄 📌 🗑️ ↻ ⋮

tendría valor null y se guardaría ese valor

Bien 👉 entonces si `this.cabeza` es `null`, `nuevoNodo.siguiente` también quedaría en `null`.

Ahora quiero que pienses un poco más profundo:

Si la lista **no** está vacía, ¿qué representa exactamente `this.cabeza` en ese momento?

Y cuando haces:



```
js  
nuevoNodo.siguiente = this.cabeza;
```

¿qué relación estás creando entre el nuevo nodo y el resto de la lista?

Describeme la estructura que queda formada en memoria después de esa línea, antes de actualizar

`this.cabeza`.

📄 📌 🗑️ ↻ ⋮

`this.cabeza` es el nodo inicial. Se está moviendo la cabeza una posición hacia adelante

Interesante lo que dices »» pero quiero que lo revisemos con cuidado.

Dices que "se está moviendo la cabeza una posición hacia adelante".

En esa línea todavía **no** has hecho:

```
js  
this.cabeza = nuevoNodo;
```

Entonces pregúntate:

En el momento exacto después de ejecutar

```
js  
nuevoNodo.siguiente = this.cabeza;
```

¿`this.cabeza` ya cambió... o sigue apuntando al mismo nodo de antes?

Y otra cosa más concreta:

¿Quién apunta a quién en ese instante?

- ¿El nodo viejo apunta al nuevo?
- ¿El nuevo apunta al viejo?
- ¿La cabeza ya se movió o todavía no?

Describeme exactamente cómo quedan las referencias en memoria en ese punto.

📄 📌 🗑️ ↻ ⋮

Okey, la cabeza no cambió, lo que quería decir es que el valor de la cabeza se copió al siguiente nodo

Bien, vas mejor 👉 pero quiero afinar algo importante en lo que dijiste:

Dices que "el valor de la cabeza se copió al siguiente nodo".

¿Realmente se está **copiando el valor**... o se está **copiando la referencia al nodo**?

Piensa en esto:

Si modificas después el nodo que era la antigua cabeza, ¿el `nuevoNodo.siguiente` seguiría apuntando al mismo objeto o a una copia independiente?

Explicame qué está pasando realmente a nivel de referencias.



Bien, vas mejor 🙌 pero quiero afinar algo importante en lo que dijiste:

Dices que “el valor de la cabeza se copió al siguiente nodo”.

¿Realmente se está **copiando el valor**... o se está copiando **la referencia al nodo**?

Piensa en esto:

Si modificas después el nodo que era la antigua cabeza, ¿el `nuevoNodo.siguiente` seguiría apuntando al mismo objeto o a una copia independiente?

Explicame qué está pasando realmente a nivel de referencias.

📄 👍 🗑 ⬆ ↺ ...

a uno copia

¿Seguro que es una copia? 🤔

Piensa en esto con calma:

Si fuera una copia real del nodo antiguo, entonces estarías creando **dos nodos distintos en memoria** con el mismo contenido.

Pero en tu código, ¿en qué momento estás creando ese supuesto segundo nodo?

Solo estás usando:

```
js
nuevoNodo.siguiente = this.cabeza;
```

No hay ningún `new` ahí.

Entonces te pregunto de forma más precisa:

Cuando asignas `nuevoNodo.siguiente = this.cabeza`, ¿estás creando un nuevo nodo... o simplemente estás haciendo que ambos apunten al mismo objeto que ya existía?

Y si luego modificas el nodo antiguo (por ejemplo su `valor`), ¿crees que `nuevoNodo.siguiente` vería ese cambio o no? ¿Por qué?

📄 👍 🗑 ⬆ ↺ ...

entonces solo es un puntero

Ahhh 🤔 ahora si estamos afinando el modelo mental.

Entonces, si es solo una referencia (un “puntero” como dices):

Cuando haces:

```
js
nuevoNodo.siguiente = this.cabeza;
```

¿podrías decir que estás:

1. Moviendo la cabeza?
2. Copiando el nodo?



o simplemente estás creando un nuevo nodo con el mismo valor de la cabeza.

Explicamelo con tus propias palabras como si tuvieras que dibujarlo en papel.

Y una más para que lo pienses bien:

Después de esa línea, pero **antes** de hacer

```
js
this.cabeza = nuevoNodo;
```

¿Existe alguna referencia desde la lista hacia `nuevoNodo`?

¿O todavía está “aislado” del resto?

📄 👍 🗑 ⬆ ↺ ...

solo se está conectando el nuevo nodo con el inicio de la lista

Bien 🙌 esa explicación ya es mucho más precisa.

Ahora quiero que pienses en la siguiente línea:

```
js
this.cabeza = nuevoNodo;
```

En este punto si cambia algo importante.

¿Qué pasaría si **olvidaras ejecutar esta línea**?

- ¿La lista seguiría funcionando igual?
- ¿El nuevo nodo quedaría accesible desde la lista?
- ¿O quedaría “perdido” en memoria?

Razona paso a paso qué ocurriría con las referencias en ese caso.

📄 👍 🗑 ⬆ ↺ ...

el nuevo nodo no estaría enlazado

Exacto 🙌, el nuevo nodo quedaría “perdido”, porque ninguna referencia desde la lista apunta a él.

Última línea del método:

```
js
this.tamaño++;
```

¿Por qué crees que es importante actualizar el tamaño aquí?

Y otra cosa: ¿qué pasaría si olvidaras incrementar `this.tamaño`? Piensa en cómo afectaría al resto de operaciones de la lista.

📄 👍 🗑 ⬆ ↺ ...

Sirve para aumentar el tamaño de la lista y conocerlo sin tener que recorrerla toda. Si no se incrementa el tamaño de la lista no sería correcto ⬇

# Método estaVacía()

```
estaVacía() {  
    return this.tamaño === 0;  
}
```

```
js Copiar código  
return this.tamaño === 0;
```

¿Puedes explicarme qué hace exactamente esta línea y por qué revisas `this.tamaño` en lugar de otra cosa, como `this.cabeza`?

📄 📌 🔍 ↺ ↻ ⋮

se comprueba que el tamaño sea estrictamente igual a 0, lo que en caso de ser cierto devuelve el valor true

Exacto 🏆, si `tamaño` es 0, la lista no tiene elementos y el método devuelve `true`.

Ahora quiero que lo pienses un poco más:

¿Hay algún caso en el que la lista **esté realmente vacía** pero `this.tamaño` **no sea 0**?

O dicho de otra forma: ¿confías en que `tamaño` siempre refleje correctamente la cantidad de nodos en la lista?

📄 📌 🔍 ↺ ↻ ⋮

El valor de tamaño está perfectamente controlado con otros métodos como el de insertarInicio

¡Perfecto! ✅ Entonces tu método `estaVacía` depende de que todos los demás métodos mantengan `tamaño` actualizado, y mientras eso se cumpla, funciona correctamente.

Última pregunta de reflexión sobre este método:

Si en algún momento quisieras implementarlo **sin usar** `tamaño`, ¿cómo podrías determinar si la lista está vacía usando solo `cabeza`?

No hace falta que lo codifiques, solo descríbelo con tus palabras.

📄 📌 🔍 ↺ ↻ ⋮

comprobando que cabeza sea o no igual a null

¡Exacto! 🏆 Si `this.cabeza === null`, la lista está vacía; si apunta a un nodo, entonces no lo está.

Así que `estaVacía` podría implementarse de **dos formas**: usando `tamaño` o usando `cabeza`. Ambas son correctas, solo que usando `tamaño` evitas tener que pensar en referencias de nodos y confías en el contador.

Si quieres, podemos hacer un **diagrama rápido de cómo funcionan ambos métodos juntos** para ver la relación entre `cabeza`, `tamaño` y los nodos. ¿Quieres que haga eso?

📄 📌 🔍 ↺ ↻ ⋮

# Método EliminarInicio()

```
eliminarInicio() {  
    if (this.estaVacía()) {  
        return null;  
    }  
    const nodoEliminado = this.cabeza;  
    this.cabeza = this.cabeza.siguiente;  
    this.tamaño--;  
    return nodoEliminado.dato;  
}
```

¡Perfecto! 🧨 Exactamente: al mover la cabeza al siguiente nodo, que en este caso es `null`, la lista queda vacía.

Siguiente línea:

```
js
this.tamaño--;
```

Copiar código

¿Por qué es importante decrementar `tamaño` aquí?

Piensa también: ¿qué pasaría si olvidaras hacer esto y luego quisieras insertar o eliminar más nodos?

📄 🗨️ 🔄 ⬇️ ⌛ ...

Es importante decrementar esa variable para que el tamaño sea siempre correcto. Si olvidara hacerlo entonces no sabría de manera certera cuantos nodos hay en la lista

¡Exacto! ✅ Mantener `tamaño` sincronizado asegura que cualquier operación que dependa del número de nodos funcione correctamente.

Última línea del método:

```
js
return nodoEliminado.dato;
```

Copiar código

Pregunta de reflexión:

¿Por qué devuelves `nodoEliminado.dato` en lugar de devolver el nodo completo?

Piensa en cómo se podría usar este valor desde fuera de la lista y qué implicaciones tendría devolver todo el nodo en vez de solo su dato.

📄 🗨️ 🔄 ⬇️ ⌛ ...

Es para identificar el valor del nodo que fue eliminado

Exacto 🧡, devolver solo el `dato` permite que el código que llama al método sepa qué valor se eliminó, sin exponer la estructura interna de la lista ni sus referencias.

Última reflexión rápida sobre tu método `eliminarInicio`:

¿Ves algún caso en el que este método pueda fallar o comportarse de forma inesperada, o crees que maneja correctamente listas vacías, listas con un nodo y listas con varios nodos?

📄 🗨️ 🔄 ⬇️ ⌛ ...

Lo maneja todo correctamente

¡Exacto! 🧠 Tu `eliminarInicio` ya maneja todos los casos importantes:

- Lista vacía → devuelve `null` sin errores
- Lista con un nodo → la cabeza pasa a `null`, lista queda vacía
- Lista con varios nodos → la cabeza se mueve al siguiente nodo y `tamaño` se actualiza

No hubo bugs o mejoras necesarias.

¡Perfecto! Vamos a diseccionar `eliminarInicio` línea por línea como hicimos antes 🧐

Primera pregunta:

```
js
if (this.estaVacia()) {
  return null;
}
```

Copiar código

¿Puedes explicarme qué hace esta condición y por qué es importante comprobar si la lista está vacía antes de intentar eliminar un nodo?

📄 🗨️ 🔄 ⬇️ ⌛ ...

si el método `estaVacia` devuelve true entonces significa que la lista no tiene ningún nodo. Como no hay nada que eliminar, este método devuelve null

Exacto 🧡, evitas errores intentando acceder a `this.cabeza` cuando no hay nodos.

Siguiente línea:

```
js
const nodoEliminado = this.cabeza;
```

Copiar código

¿Cuál es la idea de guardar la cabeza en esta variable antes de moverla?

Piensa en qué pasaría si **no guardaras esta referencia** y actualizaras `this.cabeza` directamente.

📄 🗨️ 🔄 ⬇️ ⌛ ...

si se actualizara directamente ese dato se perdería por completo

Exacto 🧡, sin guardar la referencia, perderías acceso al nodo que quieres eliminar y también a su `dato`.

Ahora la siguiente línea:

```
js
this.cabeza = this.cabeza.siguiente;
```

Copiar código

Explicame qué está pasando aquí con las referencias.

Y como reflexión: ¿qué ocurre si la lista solo tenía un nodo antes de llamar a `eliminarInicio`?

📄 🗨️ 🔄 ⬇️ ⌛ ...

El nodo que viene después de la cabeza actual pasa a ser la nueva cabeza

¡Sí! 🧡 Exactamente: el segundo nodo (si existe) se convierte en la nueva cabeza.

Ahora piensa en tu reflexión: si la lista solo tenía **un nodo**, ¿qué valor tendría `this.cabeza` después de esta línea?

Y como consecuencia: ¿cómo queda la lista?

# Checkpoint metacognitivo - Valida

¿Mi implementación pasa TODOS los casos de prueba de la secuencia anterior sin errores?

Obvio.

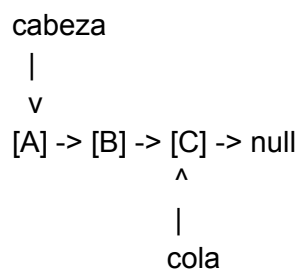
Cuando expliqué mi código a la IA, ¿hubo partes que no pude explicar claramente? ¿Por qué?

Sí, porque me preguntaba sobre características del lenguaje (JavaScript) que simplemente aún no he aprendido.

¿Escribí pruebas yo mismo o solo usé las que me dieron? ¿Puedo pensar en un caso de prueba adicional?

Solo usé las que me dieron.

## 6. Diseño de sistema: Lista con acceso $O(1)$ al final



```
class Nodo {
    constructor(valor) {
        this.valor = valor;
        this.siguiente = null;
    }
}
```

```
class ListaEnlazada {
    constructor() {
        this.cabeza = null;
        this cola = null;
        this.tamaño = 0;
    }
}
```

```
// Insertar al inicio
```

```

insertarInicio(valor) {
    const nuevoNode = new Node(valor);

    if (this.cabeza === null) {
        // Lista vacía
        this.cabeza = nuevoNode;
        this cola = nuevoNode;
    } else {
        nuevoNode.siguiente = this.cabeza;
        this.cabeza = nuevoNode;
    }

    this.tamaño++;
}

// Insertar al final (O(1))
insertarFinal(valor) {
    const nuevoNode = new Node(valor);

    if (this.cabeza === null) {
        // Lista vacía
        this.cabeza = nuevoNode;
        this cola = nuevoNode;
    } else {
        this.cola.siguiente = nuevoNode;
        this.cola = nuevoNode;
    }

    this.tamaño++;
}

// Eliminar al inicio
eliminarInicio() {
    if (this.cabeza === null) return null;

    const valorEliminado = this.cabeza.valor;
    this.cabeza = this.cabeza.siguiente;

    // Si la lista quedó vacía
    if (this.cabeza === null) {

```

```

        this cola = null;
    }

    this.tamaño--;
    return valorEliminado;
}

// Eliminar al final
eliminarUltimo() {
    if (this.cabeza === null) return null;

    // Caso: un solo nodo
    if (this.cabeza === this cola) {
        const valorEliminado = this.cabeza.valor;
        this.cabeza = null;
        this cola = null;
        this.tamaño--;
        return valorEliminado;
    }

    // Caso: más de un nodo
    let actual = this.cabeza;

    // Buscar el nodo anterior al cola
    while (actual.siguiente !== this cola) {
        actual = actual.siguiente;
    }

    const valorEliminado = this cola.valor;
    actual.siguiente = null;
    this cola = actual;

    this.tamaño--;
    return valorEliminado;
}

// Método auxiliar para visualizar la lista
imprimir() {
    let actual = this.cabeza;
    let resultado = "";

```

```
while (actual !== null) {  
    resultado += actual.valor + " -> ";  
    actual = actual.siguiente;  
}  
  
console.log(resultado + "null");  
}
```

Como ya no se necesita recorrer toda la lista desde la cabeza, la complejidad deja de ser  $O(n)$  y pasa a ser  $O(1)$  porque ahora solo se realizan las operaciones de reasignación.

## Checkpoint metacognitivo - Profundiza

¿Puedo identificar al menos una situación en mi vida académica o laboral donde una lista enlazada sería la estructura correcta?

Ahora mismo no se me ocurre nada.

¿Entiendo por qué una referencia a 'cola' hace insertarFinal()  $O(1)$  pero no hace eliminarFinal()  $O(1)$ ?

Sí, es porque los nodos no tienen una referencia al nodo anterior, y por eso es imposible reasignar la cola sin recorrer todo de nuevo.

¿Qué preguntas me quedan pendientes que quiero investigar o preguntar la próxima semana?

Debo repasar el concepto de constructores.