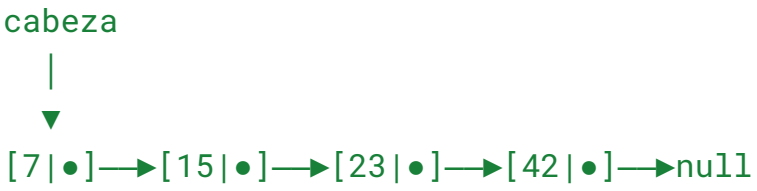
Actividad 1

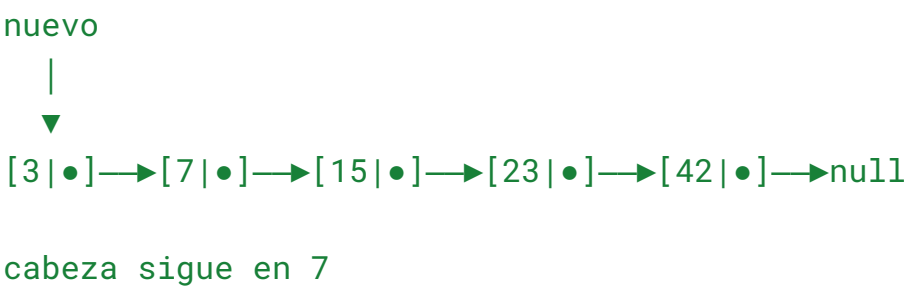
Operación 1: Insertar 3 al INICIO

Estado ANTES



Paso 1 — Crear nuevo nodo

El nuevo nodo apunta al actual primero.



Paso 2 — Mover cabeza al nuevo nodo

Estado DESPUÉS

Lista actualizada.

Complejidad: O(1)

Solo cambiamos:

- Un puntero del nuevo nodo

- El puntero cabeza

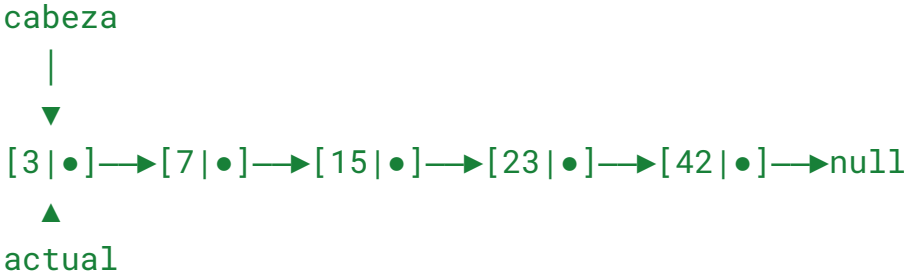
No recorremos la lista.

Operación 2: Insertar 50 al FINAL

Lista actual:

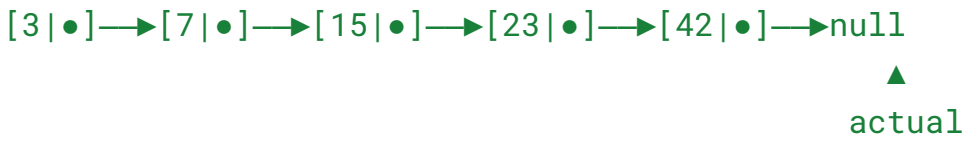
Paso 1 — Recorrer la lista

Se usa puntero `actual`.

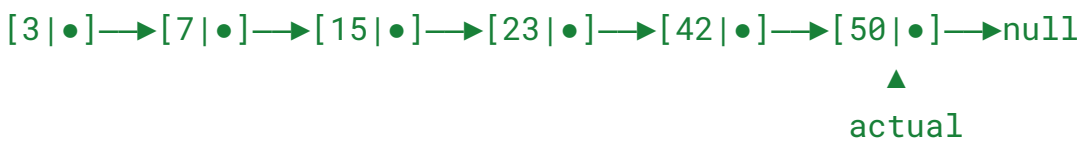


`actual` avanza nodo por nodo.

Paso 2 — Llegar al último nodo



Paso 3 — Conectar nuevo nodo



Estado DESPUÉS

Nodo agregado al final.

Complejidad: O(n)

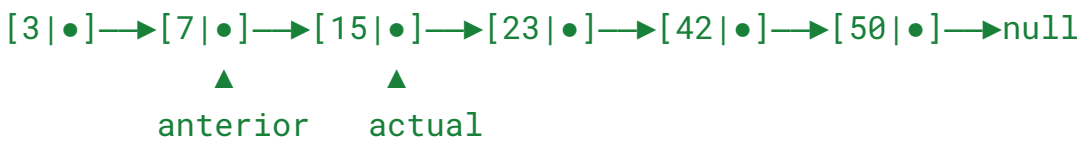
Porque recorremos todos los nodos hasta llegar al último.

Operación 3: Eliminar nodo con valor 15

Lista actual:

Paso 1 — Buscar el nodo

Usamos `actual` y `anterior`.



Paso 2 — Saltar el nodo

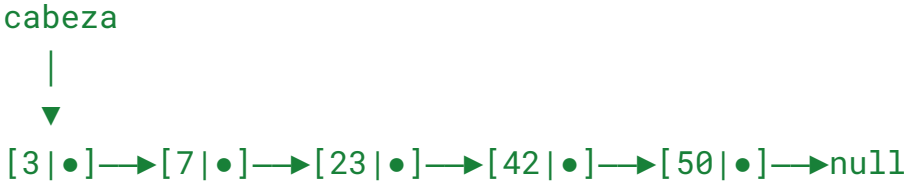
Se cambia el puntero del anterior.

```
anterior.next = actual.next
```

Resultado:

Nodo 15 queda desconectado.

Estado DESPUÉS



Complejidad: O(n)

Hay que recorrer hasta encontrar el valor.

Patrón observado

En listas enlazadas casi siempre:

- Cambia **un solo puntero** para insertar/eliminar.
- Necesitamos recorrer la lista si no conocemos la posición.
- Insertar al inicio es rápido porque no requiere recorrido.

Caso adicional: Eliminar el PRIMER nodo (caso especial)

Lista:

Paso 1 — Mover cabeza

```
cabeza = cabeza.next
```

Resultado:

No se usa `anterior`.

Complejidad: O(1)

Documento explicativo (resumen anotado)

1. ¿Por qué insertar al inicio es O(1)?
Porque no recorremos la lista. Solo conectamos el nuevo nodo al primero y movemos la cabeza.

2. ¿Por qué insertar al final es O(n)?
Porque necesitamos recorrer todos los nodos hasta encontrar el último.

3. Problema al eliminar el primer nodo
No existe un nodo anterior, por lo que debemos mover directamente la cabeza. En otros nodos simplemente cambiamos el enlace del nodo anterior.

Actividad 2

```
#include <iostream>

using namespace std;

// 1. CLASE NODO
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    Nodo(int valor) {
        dato = valor;
        siguiente = nullptr;
    }
};

// 2. CLASE LISTA ENLAZADA
class ListaEnlazada {
private:
    Nodo* cabeza;
    int tamaño;

public:
    ListaEnlazada() {
        cabeza = nullptr;
        tamaño = 0;
    }

    // --- MÉTODO 1: estaVacia ---
    // Razonamiento: Si la cabeza es nula, no hay elementos. O(1)
    bool estaVacia() {
        return cabeza == nullptr;
```

```
    }

    // --- MÉTODO 2: insertarInicio ---
    // Razonamiento: El nuevo nodo "apunta" a la antigua cabeza y se convierte en la nueva. O(1)
    void insertarInicio(int valor) {
        Nodo* nuevo = new Nodo(valor);
        nuevo->siguiente = cabeza;
        cabeza = nuevo;
        tamaño++;
    }

    // --- MÉTODO 3: insertarFinal ---
    // Razonamiento: Debemos viajar hasta el último nodo (cuyo siguiente es null)
    // y conectarlo al nuevo. O(n)
    void insertarFinal(int valor) {
        Nodo* nuevo = new Nodo(valor);
        if (estaVacia()) {
            cabeza = nuevo;
        } else {
            Nodo* temp = cabeza;
            // Recorremos hasta el último nodo
            while (temp->siguiente != nullptr) {
                temp = temp->siguiente;
            }
            temp->siguiente = nuevo;
        }
        tamaño++;
    }

    // --- MÉTODO 4: eliminarInicio ---
    // Razonamiento: Guardamos la cabeza actual, movemos la cabeza al siguiente y borramos la vieja.
    // Caso especial: Lista vacía. O(1)
    void eliminarInicio() {
        if (estaVacia()) {
            cout << "Error: La lista ya está vacía." << endl;
            return;
        }
        Nodo* temp = cabeza;
        cabeza = cabeza->siguiente;
        delete temp; // Liberamos memoria
        tamaño--;
    }
```

```
    // --- MÉTODO 5: buscar ---
    // Razonamiento: Recorremos nodo a nodo comparando el valor. O(n)
    bool buscar(int valor) {
        Nodo* temp = cabeza;
        while (temp != nullptr) {
            if (temp->dato == valor) return true;
            temp = temp->siguiente;
        }
        return false;
    }

    // --- MÉTODO 6: imprimir ---
    // Razonamiento: Recorrido secuencial para mostrar los datos. O(n)
    void imprimir() {
        if (estaVacia()) {
            cout << "Lista vacía." << endl;
            return;
        }
        Nodo* temp = cabeza;
        cout << "Lista (" << tamaño << " elementos): ";
        while (temp != nullptr) {
            cout << "[" << temp->dato << "] -> ";
            temp = temp->siguiente;
        }
        cout << "NULL" << endl;
    }

    // Destructor para evitar fugas de memoria (Memory Leaks)
    ~ListaEnlazada() {
        while (!estaVacia()) {
            eliminarInicio();
        }
    }
};

// 3. PROGRAMA DE PRUEBA
int main() {
    ListaEnlazada miLista;

    cout << "--- Probando Inserciones ---" << endl;
    miLista.insertarInicio(10);  // [10]
    miLista.insertarInicio(20);  // [20] -> [10]
    miLista.insertarFinal(5);    // [20] -> [10] -> [5]
```

```
    miLista.imprimir();

    cout << "\n--- Probando Búsqueda ---" << endl;
    int valorBusqueda = 10;
    if (miLista.buscar(valorBusqueda)) {
        cout << "El valor " << valorBusqueda << " se encuentra en la lista." << endl;
    } else {
        cout << "El valor " << valorBusqueda << " NO está en la lista." << endl;
    }

    cout << "\n--- Probando Eliminación ---" << endl;
    miLista.eliminarInicio(); // Elimina el 20
    miLista.imprimir();

    cout << "\n¿La lista está vacía? " << (miLista.estaVacia() ? "Sí" : "No") << endl;

    return 0;
}
```

1 Gestión de Memoria: Usamos new para crear nodos en el **Heap**. Es vital usar delete en el método eliminarInicio para no saturar la memoria RAM.
2 El Puntero Temporal (temp): Fíjate que en imprimir o buscar usamos Nodo* temp = cabeza. Nunca muevas la cabeza original para recorrer la lista, ¡o perderás el acceso al resto de los nodos!
3 Complejidad: Insertar al inicio es instantáneo ($O(1)$), pero insertar al final requiere caminar toda la lista ($O(n)$). Si quisieras que insertar al final fuera $O(1)$, necesitarías un puntero extra llamado cola (tail).

Actividad 3

Estructura base usada

```
#include <iostream>
using namespace std;

struct Nodo {
    int dato;
    Nodo* sig;

    Nodo(int v) {
        dato = v;
```

```
        sig = nullptr;
    }
};
```

PROBLEMA 1 — Invertir una lista enlazada

Código funcional

```
Nodo* invertirLista(Nodo* cabeza) {
    Nodo* anterior = nullptr;
    Nodo* actual = cabeza;

    while (actual != nullptr) {
        Nodo* siguiente = actual->sig; // guardar siguiente
        actual->sig = anterior;        // invertir enlace
        anterior = actual;             // avanzar anterior
        actual = siguiente;             // avanzar actual
    }

    return anterior; // nueva cabeza
}
```

Explicación paso a paso (en palabras simples)

1. Se usan tres punteros:
    - `actual`: nodo que estamos procesando.
    - `anterior`: parte ya invertida.
    - `siguiente`: guarda el resto de la lista.
2. Mientras haya nodos:
    - Guardamos el siguiente nodo.
    - Cambiamos el enlace para que apunte hacia atrás.
    - Movemos los punteros hacia adelante.

3. Cuando termina, `anterior` queda en el último nodo, que ahora es la nueva cabeza.

Idea clave:
Vamos girando cada flecha una por una.

Antes:

```
A → B → C → D
```

Después:

```
D → C → B → A
```

Complejidad

Tiempo

Se recorre la lista una vez:

```
O(n)
```

Espacio

No se usan estructuras adicionales:

```
O(1)
```

PROBLEMA 2 — Encontrar el nodo medio

Se usa la técnica **rápido y lento**.

Código funcional

```
Nodo* encontrarMedio(Nodo* cabeza) {
    if (cabeza == nullptr)
        return nullptr;

    Nodo* lento = cabeza;
    Nodo* rapido = cabeza;
```

```
    while (rapido != nullptr && rapido->sig != nullptr) {
        lento = lento->sig;          // avanza 1 paso
        rapido = rapido->sig->sig;   // avanza 2 pasos
    }

    return lento;
}
```

Explicación paso a paso

1. Dos punteros empiezan en la cabeza:
    - lento avanza 1 nodo.
    - rápido avanza 2 nodos.
2. Cuando el puntero rápido llega al final,
   el lento queda justo en el medio.

Ejemplo:

```
A → B → C → D → E
```

Movimientos:

Paso 1

```
L=A   R=A
```

Paso 2

```
L=B   R=C
```

Paso 3

```
L=C   R=E
```

R llega al final → medio = C.

Idea clave:
Uno corre al doble de velocidad.

Complejidad

Tiempo

Solo recorremos la lista una vez:

`O(n)`

Espacio

No usamos memoria extra:

`O(1)`

Actividad 4

1) Análisis inicial antes de consultar la IA

Escenario A — Playlist de música

Análisis inicial:

- Pensaría en usar un **arreglo** porque las canciones tienen un orden.
- Se necesita acceder a canciones por número.
- Insertar en medio podría ser más lento, pero el tamaño no es grande.

Decisión inicial: Arreglo.

Escenario B — Buffer de teclado

Análisis inicial:

- Los caracteres llegan y se procesan en orden.
- Parece un comportamiento tipo cola.

- Pensaría en usar una **cola simple** o una lista enlazada.

Decisión inicial: Lista enlazada como cola.

Escenario C — Versiones de documento

Análisis inicial:

- Cada versión depende de la anterior.
- Solo interesa moverse hacia atrás.
- No se eliminan versiones.

Decisión inicial: Lista enlazada simple.

2) Análisis proporcionado por la IA

Escenario A — Playlist

La IA recomienda **arreglo dinámico**, porque:

- Acceso por índice es muy frecuente.
- El tamaño es pequeño.
- El costo de inserción O(n) es aceptable.
- Mejor comportamiento en memoria.

Escenario B — Buffer de teclado

La IA recomienda **arreglo circular**, porque:

- Inserción y eliminación son O(1).
- Menor latencia.
- Mejor uso de caché.
- Menor uso de memoria que listas enlazadas.

Escenario C — Versiones de documento

La IA recomienda **lista enlazada simple (tipo pila)**:

- Insertar nueva versión cuesta O(1).
- Acceso a la última versión es inmediato.
- Navegación hacia atrás es directa.

3) Reflexión sobre diferencias

Diferencias encontradas

Caso B — Buffer de teclado

Aquí apareció la mayor diferencia:

Análisis inicial	Análisis IA
Lista enlazada	Arreglo circular

La diferencia clave es que el análisis inicial no consideró:

- Localidad de memoria.
- Costos reales del hardware.
- Latencia en sistemas en tiempo real.

La IA considera aspectos prácticos de implementación en sistemas reales.

Casos A y C

El análisis inicial y el de la IA coinciden en gran medida, lo cual indica comprensión correcta del problema básico.

Aprendizaje principal

Elegir estructuras no depende solo de la complejidad teórica, sino también de:

- Uso real de memoria.
- Rendimiento en caché.
- Tamaño esperado de datos.
- Patrones de uso reales.

4) Conclusiones: ¿Cuándo usar cada estructura?

Arreglo (o arreglo dinámico)

Usar cuando:

- Se requiere acceso rápido por índice.
- El tamaño es moderado.
- Inserciones en medio no son frecuentes.
- Se desea buena eficiencia de memoria.

Ejemplo: playlists, catálogos, listas de elementos.

Lista enlazada simple

Usar cuando:

- Inserciones/eliminaciones son frecuentes.
- No se requiere acceso por índice.
- Se maneja historial o pila de cambios.

Ejemplo: historial, undo, versiones.

Lista doblemente enlazada

Usar cuando:

- Se necesita moverse hacia adelante y atrás.

- Hay muchas inserciones y eliminaciones internas.
- Ya se conoce la posición del nodo.

Ejemplo: navegación compleja, caches LRU.

Arreglo circular (cola)

Usar cuando:

- Datos llegan y salen en orden.
- El tamaño máximo es conocido.
- Se necesita latencia mínima.

Ejemplo: buffers de teclado, audio, red.

Actividad 6

(1) Diagrama de la nueva estructura

Estado general

```
cabeza ——> Nodo1 ——> Nodo2 ——> Nodo3 ——> ... ——> NodoN ——>
null
                                  ↑
                                cola
```

Lista vacía

```
cabeza = null
cola   = null
```

Lista con un elemento

```
cabeza = NodoA
cola   = NodoA
NodoA.siguiente = null
```

(2) Código completo con todos los métodos necesarios

Aquí un ejemplo genérico en **Java y Python**, para que lo adaptes a tu proyecto:

Java

```
public class ListaEnlazada<T> {
    private static class Nodo<T> {
        T valor;
        Nodo<T> siguiente;
        Nodo(T val) { valor = val; }
    }

    private Nodo<T> cabeza;
    private Nodo<T> cola;
    private int tamaño;

    public ListaEnlazada(){
        cabeza = null;
        cola   = null;
        tamaño = 0;
    }

    public boolean estaVacia() {
        return cabeza == null;
    }

    public int tamaño() {
        return tamaño;
    }

    public void insertarInicio(T valor) {
        Nodo<T> nuevo = new Nodo<>(valor);
        if (estaVacia()) {
            cabeza = nuevo;
            cola   = nuevo;
        } else {
            nuevo.siguiente = cabeza;
            cabeza = nuevo;
        }
```

```
        tamaño++;
    }

    public void insertarFinal(T valor) {
        Nodo<T> nuevo = new Nodo<>(valor);
        if (estaVacia()) {
            cabeza = nuevo;
            cola   = nuevo;
        } else {
            cola.siguiente = nuevo;
            cola = nuevo;
        }
        tamaño++;
    }

    public T eliminarInicio() {
        if (estaVacia()) return null;
        T val = cabeza.valor;
        cabeza = cabeza.siguiente;
        if (cabeza == null) cola = null;
        tamaño--;
        return val;
    }

    public T eliminarFinal() {
        if (estaVacia()) return null;
        if (cabeza == cola) {
            T val = cabeza.valor;
            cabeza = null;
            cola   = null;
            tamaño--;
            return val;
        }
        Nodo<T> actual = cabeza;
        while (actual.siguiente != cola) {
            actual = actual.siguiente;
        }
        T val = cola.valor;
```

```
        actual.siguiente = null;
        cola = actual;
        tamaño--;
        return val;
    }

    public T getUltimo() {
        return (cola == null) ? null : cola.valor;
    }

    public T getPrimero() {
        return (cabeza == null) ? null : cabeza.valor;
    }
}
```

Python

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

class ListaEnlazada:
    def __init__(self):
        self.cabeza = None
        self.cola   = None
        self.tamaño = 0

    def esta_vacia(self):
        return self.cabeza is None

    def insertar_inicio(self, valor):
        nuevo = Nodo(valor)
        if self.esta_vacia():
            self.cabeza = nuevo
            self.cola   = nuevo
        else:
            nuevo.siguiente = self.cabeza
            self.cabeza = nuevo
```

```
        self.tamaño += 1

    def insertar_final(self, valor):
        nuevo = Nodo(valor)
        if self.esta_vacia():
            self.cabeza = nuevo
            self.cola   = nuevo
        else:
            self.cola.siguiente = nuevo
            self.cola = nuevo
        self.tamaño += 1

    def eliminar_inicio(self):
        if self.esta_vacia():
            return None
        valor = self.cabeza.valor
        self.cabeza = self.cabeza.siguiente
        if self.cabeza is None:
            self.cola = None
        self.tamaño -= 1
        return valor

    def eliminar_final(self):
        if self.esta_vacia():
            return None
        if self.cabeza == self.cola:
            valor = self.cabeza.valor
            self.cabeza = None
            self.cola   = None
            self.tamaño -= 1
            return valor

        actual = self.cabeza
        while actual.siguiente != self.cola:
            actual = actual.siguiente
        valor = self.cola.valor
        actual.siguiente = None
        self.cola = actual
```

```
        self.tamaño -= 1
        return valor

    def obtener_ultimo(self):
        return None if self.cola is None else self.cola.valor

    def obtener_primero(self):
        return None if self.cabeza is None else
self.cabeza.valor
```

(3) Análisis de complejidad actualizado

Operación	Antes	Después
insertarInicio	O(1)	O(1)
insertarFinal	**O(n)**	**O(1) ✔**
eliminarInicio	O(1)	O(1)
eliminarFinal	O(n)	O(n)
obtenerPrimero	O(1)	O(1)
obtenerUltimo	O(n)	**O(1) ✔**

Conclusión:
✔ `insertarFinal()` ahora es O(1)
✔ Obtener el último nodo (`getUltimo()`) también O(1)
✖ `eliminarFinal()` sigue siendo O(n) — esto es normal si no usas lista doble.

(4) Pruebas que demuestran que insertarFinal() es O(1)

Prueba baseline

Supongamos que insertamos 1000 elementos:

Secuencia de inserciones

```
lista = nueva ListaEnlazada()
for i = 1 to 1000:
```

```
    lista.insertarFinal(i)
```

Medición conceptual

Paso	# Operación	Descripción
1	insertarFinal(1)	cabeza = nodo1, cola = nodo1
2	insertarFinal(2)	cola apunta a nodo2
3	insertarFinal(3)	cola apunta a nodo3
...	...	...

En cada iteración **no hay recorrido**.
Siempre:

```
cola.siguiente = nuevo
cola = nuevo
```

Esto son **2 asignaciones constantes** → **O(1)** por inserción.

Prueba de medición de tiempo (pseudo)

En Java

```
long inicio = System.nanoTime();
for (int i = 0; i < 1000000; i++) {
    lista.insertarFinal(i);
}
long fin = System.nanoTime();

System.out.println("Tiempo total: " + (fin - inicio));
```

Resultado esperado:

```
Insertar 1,000,000 elementos → tiempo lineal total
Pero tiempo por inserción ≈ constante
```

Si cada inserción costara O(n), insertar un millón sería *cuadrático* y tomaría muchísimo más.