


# Registro/Estructuras


⦿ Parcial

AutoEstudio

## ¿Qué es un registro en C (estructura)?

En C, un **registro** es definido con `struct`, y permite agrupar **diferentes tipos de datos** en una sola unidad lógica.

 `strcpy` se usa en C para **copiar cadenas de texto** (strings) de un lugar a otro.  
En C,  
**las cadenas son arreglos de caracteres, y no puedes asignarlas directamente** con `=` como en otros lenguajes.

 Sintaxis

```
strcpy(destino, origen);
```

- `destino`: el arreglo donde quieres guardar la cadena
- `origen`: la cadena que quieres copiar

✗ Esto NO funciona:

```
char nombre[30];  
nombre = "Luis"; // ✗ Error: no se pueden asignar strings así en C
```

✓ Usamos

`strcpy`:

```
#include <string.h>
```

```
char nombre[30];
```

```
strcpy(nombre, "Luis"); // ✓ Copia "Luis" dentro del arreglo nombre
```

Vamos a hacer un ejemplo de un **registro de estudiante** usando `struct` en C, que incluya los siguientes campos:

- `matricula` (entero)
- `nombre` (cadena)
- `apellido` (cadena)
- `edad` (entero)
- `esRegular` (valor booleano: `1` para regular, `0` para no regular)

## ✓ Código en C: Registro de estudiante

```
#include <stdio.h>
#include <string.h>

// Definimos el tipo de estructura para un estudiante
typedef struct {
    int matricula;
    char nombre[30];
    char apellido[30];
    int edad;
    int esRegular; // 1 = Sí, 0 = No
} Estudiante;

// Función para imprimir la información de un estudiante
void imprimirEstudiante(Estudiante e) {
    printf("Matricula: %d\n", e.matricula);
    printf("Nombre: %s %s\n", e.nombre, e.apellido);
    printf("Edad: %d\n", e.edad);
    printf("¿Es regular?: %s\n", e.esRegular ? "Sí" : "No");
}

int main() {
    // Creamos una variable tipo Estudiante
    Estudiante alumno1; //sin usar struct al principio

    // Llenamos los datos
    alumno1.matricula = 20250123;
    strcpy(alumno1.nombre, "Luis");
    strcpy(alumno1.apellido, "Martínez");
    alumno1.edad = 20;
    alumno1.esRegular = 1;

    // Mostramos los datos
```

```
imprimirEstudiante(alumno1);

return 0;
}
```

## 🧠 Explicación por partes

Parte del código	Qué hace
<code>struct Estudiante</code>	Define un nuevo tipo de dato que agrupa varios campos
<code>char nombre[30];</code>	Guarda una cadena (nombre), hasta 29 caracteres + <code>\0</code>
<code>strcpy()</code>	Copia una cadena a un arreglo de caracteres
<code>esRegular</code>	Se usa como booleano (0 o 1)
<code>imprimirEstudiante()</code>	Función que muestra el contenido del registro

## 🔄 ¿Y si quiero varios estudiantes?

Puedes usar un **arreglo de estructuras**:

```
#define MAX 3

int main() {
    Estudiante alumnos[MAX];

    // Llenamos manualmente 3 estudiantes (puedes usar scanf también)
    alumnos[0] = (struct Estudiante){20250123, "Luis", "Martínez", 20, 1};
    alumnos[1] = (struct Estudiante){20250124, "Ana", "Gómez", 21, 1};
    alumnos[2] = (struct Estudiante){20250125, "Carlos", "Ruiz", 19, 0};

    // Imprimir todos
    int i = 0;
    while (i < MAX) {
        printf("\nEstudiante #%d\n", i + 1);
        imprimirEstudiante(alumnos[i]);
    }
}
```

```

        i++;
    }

    return 0;
}

```

## Para ingresar con teclado

```

#include <stdio.h>
#include <string.h>

// Definimos la estructura con typedef
typedef struct {
    int matricula;
    char nombre[30];
    char apellido[30];
    int edad;
    int esRegular; // 1 = Sí, 0 = No
} Estudiante;

// Función para pedir los datos de un estudiante
void capturarEstudiante(Estudiante *e) {
    printf("Ingrese la matrícula: ");
    scanf("%d", &e->matricula);
    getchar(); // limpiar el salto de línea

    printf("Ingrese el nombre: ");
    fgets(e->nombre, sizeof(e->nombre), stdin);
    e->nombre[strcspn(e->nombre, "\n")] = '\0'; // quitar el salto de línea

    printf("Ingrese el apellido: ");
    fgets(e->apellido, sizeof(e->apellido), stdin);
    e->apellido[strcspn(e->apellido, "\n")] = '\0';
}

```

```

printf("Ingrese la edad: ");
scanf("%d", &e->edad);

printf("¿Es regular? (1 = Sí, 0 = No): ");
scanf("%d", &e->esRegular);
}

// Función para imprimir un estudiante
void imprimirEstudiante(Estudiante e) {
    printf("Matricula: %d\n", e.matricula);
    printf("Nombre: %s %s\n", e.nombre, e.apellido);
    printf("Edad: %d\n", e.edad);
    printf("¿Es regular?: %s\n", e.esRegular ? "Sí" : "No");
}

int main() {
    int n;
    printf("¿Cuántos estudiantes deseas registrar? ");
    scanf("%d", &n);

    Estudiante alumnos[n];
    int i = 0;

    printf("\n--- Captura de estudiantes ---\n");
    while (i < n) {
        printf("\nEstudiante #%d:\n", i + 1);
        capturarEstudiante(&alumnos[i]);
        i++;
    }

    printf("\n--- Lista de estudiantes registrados ---\n");
    i = 0;
    while (i < n) {
        printf("\nEstudiante #%d\n", i + 1);
        imprimirEstudiante(alumnos[i]);
        i++;
    }
}

```

```
}  
  
return 0;  
}
```

# Sumario Pilas y Filas

⊙ Parcial

AutoEstudio

## Pila (stack)

- **LIFO (Last In, First Out):** El último que entra es el primero en salir.
- Se utiliza un arreglo y un entero `top` que indica la cima de la pila.
- Inicialmente, `top = -1` significa pila vacía.
- Se usa para **deshacer acciones, paréntesis, recursión**, etc.

```
#include <stdio.h>  
  
#define MAX 5  
  
// Estructura para la pila  
typedef struct {  
    int datos[MAX]; // Arreglo que guarda los datos  
    int top;        // Índice del tope de la pila  
} Stack;  
  
// Inicializa la pila vacía  
void init(Stack *pila) {  
    pila->top = -1; //pila vacia  
}  
  
// Verifica si la pila está vacía  
int isEmpty(Stack *pila) {  
    return pila->top == -1;  
}  
  
// Verifica si la pila está llena
```

```

int isFull(Stack *pila) {
    return pila->top == MAX - 1;
}

// Agrega un valor a la pila
void push(Stack *pila, int valor) {
    if (!isFull(pila)) {
        pila->top++;
        pila->datos[pila->top] = valor;
    } else {
        printf("La pila está llena\n");
    }
}

// Elimina y retorna el valor del tope
int pop(Stack *pila) {
    if (!isEmpty(pila)) {
        int val = pila->datos[pila->top];
        pila->top--;
        return val;
    } else {
        printf("La pila está vacía\n");
        return -1;
    }
}

// Muestra el valor del tope sin eliminarlo
int peek(Stack *pila) {
    if (!isEmpty(pila)) {
        return pila->datos[pila->top];
    } else {
        return -1;
    }
}

// Imprime el contenido actual de la pila

```

```

void print(Stack *pila) {
    if (isEmpty(pila)) {
        printf("La pila está vacía\n");
    } else {
        printf("Pila: ");
        for (int i = pila->top; i >= 0; i--) {
            printf("%d ", pila->datos[i]);
        }
        printf("\n");
    }
}

```

## Fila circular (Queue)

- **FIFO (First In, First Out):** El primero que entra es el primero en salir.
- Se usan dos índices: `front` y `rear`.
- El arreglo es circular con `%` (módulo): si se llega al final, se vuelve al inicio.
- Condición de **fila vacía**: `front == -1`
- Condición de **fila llena**: `(rear + 1) % MAX == front`

```

#include <stdio.h>

#define MAX 5

// Estructura para la fila circular
typedef struct {
    int datos[MAX]; // Arreglo circular
    int front;      // Índice del primer elemento
    int rear;       // Índice del último elemento
} Queue;

// Inicializa la fila vacía
void initQueue(Queue *q) {

```

```

    q->front = -1;
    q->rear = -1;
}

// Verifica si la fila está vacía
int isEmpty(Queue *q) {
    return q->front == -1;
}

// Verifica si la fila está llena (modo circular)
int isFull(Queue *q) {
    return (q->rear + 1) % MAX == q->front;
}

// Inserta un valor al final de la fila
void enqueue(Queue *q, int valor) {
    if (isFull(q)) {
        printf("La fila está llena\n");
        return;
    }

    if (isEmpty(q)) {
        q->front = 0;
        q->rear = 0;
    } else {
        q->rear = (q->rear + 1) % MAX;
    }

    q->datos[q->rear] = valor;
}

// Elimina y retorna el valor del frente de la fila
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("La fila está vacía\n");
        return -1;
    }

```

```

}

    int val = q->datos[q->front];

    if (q->front == q->rear) {
        // Solo había un elemento
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }

    return val;
}

// Muestra el valor al frente de la fila
int peek(Queue *q) {
    if (!isEmpty(q)) {
        return q->datos[q->front];
    }
    return -1;
}

// Imprime todos los elementos de la fila circular
void printQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Fila vacía\n");
        return;
    }

    printf("Fila: ");
    int i = q->front;
    while (1) {
        printf("%d ", q->datos[i]);
        if (i == q->rear) break;
        i = (i + 1) % MAX;
    }

```

```

    }
    printf("\n");
}

```

## Recursividad

- Una función se llama a sí misma para resolver un problema en partes.
- Siempre necesita:
  1. **Caso base**: condición para detenerse.
  2. **Llamada recursiva**: repetir con menor tamaño.

### Factorial

```

int factorial(int n) {
    if (n <= 1) return 1; // Caso base
    return n * factorial(n - 1); // Paso recursivo
}

```

### Suma de numeros de 1 a n

```

int suma(int n) {
    if (n == 0) return 0;
    return n + suma(n - 1);
}

```




### Recorrer lista enlazada recursivamente

```

void imprimirRecursivo(Nodo *n) {
    if (n == NULL) {
        printf("NULL\n");
        return;
    }
    printf("%d → ", n→valor);
    imprimirRecursivo(n→siguiente);
}

```

## Notas utiles para el examen

-  se usa para **declarar o acceder** a punteros.
-  se usa para acceder a miembros de una estructura a través de un puntero.
-  se usa en fila circular para **volver al inicio del arreglo** (circularidad).

# Listas ligadas simples

⦿ Parcial

AutoEstudio

Una **lista ligada** (o enlazada) es una estructura de datos lineal donde cada elemento (llamado **nodo**) contiene dos partes:

- **Dato** (por ejemplo, un número o una estructura).
- **Apuntador** (puntero) al siguiente nodo de la lista.

A diferencia de los arreglos, las listas ligadas **no tienen un tamaño fijo** y los elementos **no están en posiciones contiguas en memoria**.

## Analogía

Imagina una lista ligada como un tren:

- Cada vagón tiene un número (dato) y un enganche al siguiente vagón (puntero).
- No necesitas saber cuántos vagones hay.
- Puedes agregar o quitar vagones en cualquier parte del tren.

## Definición de un Nodo en C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Nodo {
    int dato;
    struct Nodo* siguiente;
} Nodo;
```

El `typedef` permite luego declarar nodos como `Nodo*`.

## 1. Inserción al inicio de una lista

```
bool insertarInicio(Nodo** cabeza, int valor) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = *cabeza;
    *cabeza = nuevo;
    return true;
}
```

### ¿Qué hace?

Agrega un nuevo nodo al **inicio de la lista**.

### Pasos:

1. Reserva memoria para un nuevo nodo.
2. Si `malloc` falla, retorna `false`.
3. Coloca el valor en el nuevo nodo.
4. Apunta el nuevo nodo al nodo que antes era el primero.
5. Actualiza la cabeza de la lista para que apunte al nuevo nodo.
6. Retorna `true`.

 **Piensa en insertar un libro al principio de una pila.**

 Ejemplo visual:

```
ANTES:  cabeza → [3] → [5] → NULL
NUEVO VALOR: 1
DESPUÉS:  cabeza → [1] → [3] → [5] → NULL
```

## 2. Inserción al final de una lista



```
bool insertarFinal(Nodo** cabeza, int valor) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = NULL;

    if (*cabeza == NULL) {
        *cabeza = nuevo;
        return true;
    }

    Nodo* actual = *cabeza;
    while (actual->siguiente != NULL) {
        actual = actual->siguiente;
    }
    actual->siguiente = nuevo;
    return true;
}
```

### ¿Qué hace?

Agrega un nodo al **final de la lista**.

#### Pasos:

1. Crea un nuevo nodo con `valor`.
2. Si la lista está vacía, lo pone como primer nodo.
3. Si no, recorre la lista hasta llegar al último nodo.
4. El último nodo ahora apunta al nuevo nodo.
5. Retorna `true`.

🔗 Ejemplo:

ANTES: cabeza → [3] → [5] → NULL  
 NUEVO VALOR: 8

DESPUÉS: cabeza → [3] → [5] → [8] → NULL

### 3. Inserción en una posición específica

```
bool insertarEnPosicion(Nodo** cabeza, int valor, int posicion) {
    if (posicion < 0) return false;
    if (posicion == 0) return insertarInicio(cabeza, valor);

    Nodo* actual = *cabeza;
    int i = 0;

    while (actual != NULL && i < posicion - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return false;

    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = actual->siguiente;
    actual->siguiente = nuevo;
    return true;
}
```

### ¿Qué hace?

Agrega un nodo en una **posición específica**.

#### Pasos:

1. Si `posición` es 0, llama a `insertarInicio`.
2. Recorre la lista hasta el nodo anterior a la posición deseada.
3. Si no encuentra la posición válida, retorna `false`.

4. Inserta el nuevo nodo en medio de la lista.

5. Retorna `true`.

🔗 Ejemplo:

```
ANTES:  cabeza → [3] → [5] → NULL
INSERTA: 4 en posición 1
DESPUÉS:  cabeza → [3] → [4] → [5] → NULL
```

⚠ Si la posición no existe, muestra un mensaje de error.

## ⚠ 4. Eliminación al inicio

```
bool eliminarInicio(Nodo** cabeza) {
    if (*cabeza == NULL) return false;
    Nodo* temp = *cabeza;
    *cabeza = temp->siguiente;
    free(temp);
    temp = NULL;
    return true;
}
```

### ¿Qué hace?

Elimina el **primer nodo** de la lista.

#### Pasos:

1. Si la lista está vacía, retorna `false`.
2. Guarda una referencia temporal del primer nodo.
3. Actualiza la cabeza al siguiente nodo.
4. Libera la memoria del nodo eliminado y lo manda a `NULL`.
5. Retorna `true`.

🔗 Ejemplo:

ANTES: cabeza → [3] → [5] → NULL

DESPUÉS: cabeza → [5] → NULL

## ⚠ 5. Eliminación al final

```
bool eliminarFinal(Nodo** cabeza) {
    if (*cabeza == NULL) return false;

    if ((*cabeza)->siguiente == NULL) {
        free(*cabeza);
        *cabeza = NULL;
        return true;
    }

    Nodo* actual = *cabeza;
    while (actual->siguiente->siguiente != NULL) {
        actual = actual->siguiente;
    }

    free(actual->siguiente);
    actual->siguiente = NULL;
    return true;
}
```

### ¿Qué hace?

Elimina el **último nodo**.

#### Pasos:

1. Si la lista está vacía, retorna `false`.
2. Si solo hay un nodo, lo elimina y pone `NULL` como cabeza.
3. Si hay más de uno, recorre hasta el penúltimo.
4. Libera el último nodo.

5. Retorna `true`.

🔗 Ejemplo:

```
ANTES:  cabeza → [3] → [5] → [8] → NULL
DESPUÉS: cabeza → [3] → [5] → NULL
```

## ! 6. Eliminación en una posición específica

```
bool eliminarEnPosicion(Nodo** cabeza, int posicion) {
    if (*cabeza == NULL || posicion < 0) return false;
    if (posicion == 0) return eliminarInicio(cabeza);

    Nodo* actual = *cabeza;
    int i = 0;

    while (actual->siguiente != NULL && i < posicion - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual->siguiente == NULL) return false;

    Nodo* temp = actual->siguiente;
    actual->siguiente = temp->siguiente;
    free(temp);
    temp = NULL;
    return true;
}
```

### ¿Qué hace?

Elimina un nodo en una **posición específica**.

### Pasos:

1. Si la lista está vacía o la posición es inválida, retorna `false`.
2. Si es la posición 0, llama a `eliminarInicio`.
3. Recorre hasta el nodo anterior a la posición.
4. Elimina el nodo apuntado, ajusta los enlaces.
5. Libera la memoria del nodo eliminado.
6. Retorna `true`.

🔗 Ejemplo:

```
ANTES:  cabeza → [3] → [5] → [8] → NULL
ELIMINA en posición 1
DESPUÉS: cabeza → [3] → [8] → NULL
```

## 🔄 7. Recorrer la lista

```
void imprimirLista(Nodo* cabeza) {
    Nodo* actual = cabeza;
    while (actual != NULL) {
        printf("%d → ", actual->dato);
        actual = actual->siguiente;
    }
    printf("NULL\n");
}
```

### ¿Qué hace?

Imprime todos los nodos de la lista en orden.

### Pasos:

1. Recorre desde el primer nodo hasta el último.
2. Imprime cada dato seguido de `>`.
3. Al final imprime `NULL`.

🚩 Ejemplo de salida:

3 → 5 → 8 → NULL

## Ejemplo de uso

```
int main() {
    Nodo* lista = NULL;

    insertarInicio(&lista, 10);
    insertarFinal(&lista, 20);
    insertarEnPosicion(&lista, 15, 1); // Inserta 15 en la posición 1

    recorrerLista(lista); // 10 → 15 → 20 → NULL

    eliminarEnPosicion(&lista, 1); // Elimina el nodo en la posición 1
    recorrerLista(lista); // 10 → 20 → NULL

    eliminarInicio(&lista);
    eliminarFinal(&lista);
    recorrerLista(lista); // NULL

    return 0;
}
```

## aplicandolo para trabajar con estructura de estudiantes

### listas.h

```
#ifndef LISTA_H
#define LISTA_H
```

```
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

// Tipo genérico de nodo usando un tipo definido externamente llamado TIPO_DATO
#define TIPO_DATO Estudiante

typedef struct Nodo {
    TIPO_DATO dato;
    struct Nodo* siguiente;
} Nodo;

// Funciones genéricas
bool insertarInicio(Nodo** cabeza, TIPO_DATO valor) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = *cabeza;
    *cabeza = nuevo;
    return true;
}

bool insertarFinal(Nodo** cabeza, TIPO_DATO valor) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = NULL;

    if (*cabeza == NULL) {
        *cabeza = nuevo;
        return true;
    }

    Nodo* actual = *cabeza;
    while (actual->siguiente != NULL) {
```

```

    actual = actual->siguiente;
}
actual->siguiente = nuevo;
return true;
}

bool insertarEnPosicion(Nodo** cabeza, TIPO_DATO valor, int posicion) {
    if (posicion < 0) return false;
    if (posicion == 0) return insertarInicio(cabeza, valor);

    Nodo* actual = *cabeza;
    int i = 0;

    while (actual != NULL && i < posicion - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return false;

    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (nuevo == NULL) return false;
    nuevo->dato = valor;
    nuevo->siguiente = actual->siguiente;
    actual->siguiente = nuevo;
    return true;
}

bool eliminarInicio(Nodo** cabeza) {
    if (*cabeza == NULL) return false;
    Nodo* temp = *cabeza;
    *cabeza = temp->siguiente;
    free(temp);
    return true;
}

```

```

bool eliminarFinal(Nodo** cabeza) {
    if (*cabeza == NULL) return false;

    if ((*cabeza)->siguiente == NULL) {
        free(*cabeza);
        *cabeza = NULL;
        return true;
    }

    Nodo* actual = *cabeza;
    while (actual->siguiente->siguiente != NULL) {
        actual = actual->siguiente;
    }

    free(actual->siguiente);
    actual->siguiente = NULL;
    return true;
}

bool eliminarEnPosicion(Nodo** cabeza, int posicion) {
    if (*cabeza == NULL || posicion < 0) return false;
    if (posicion == 0) return eliminarInicio(cabeza);

    Nodo* actual = *cabeza;
    int i = 0;

    while (actual->siguiente != NULL && i < posicion - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual->siguiente == NULL) return false;

    Nodo* temp = actual->siguiente;
    actual->siguiente = temp->siguiente;
    free(temp);
}

```

```

    return true;
}

// El recorrido lo dejamos para que el usuario lo defina según su estructura

#endif

```

## listas.c

```

#include <stdio.h>
#include <string.h>
#include "lista.h"

// Definimos el tipo de dato antes de incluir la cabecera
typedef struct {
    int matricula;
    char nombre[50];
    char apellido[50];
    int edad;
    bool esRegular;
} Estudiante;

#undef TIPO_DATO
#define TIPO_DATO Estudiante
#include "lista.h"

// Función para imprimir un estudiante
void imprimirEstudiante(Estudiante e) {
    printf("Matricula: %d\n", e.matricula);
    printf("Nombre: %s %s\n", e.nombre, e.apellido);
    printf("Edad: %d\n", e.edad);
    printf("Regular: %s\n", e.esRegular ? "Sí" : "No");
}

// Función para recorrer lista

```

```

void recorrerLista(Nodo* cabeza) {
    Nodo* actual = cabeza;
    while (actual != NULL) {
        imprimirEstudiante(actual->dato);
        printf("-----\n");
        actual = actual->siguiente;
    }
}

Estudiante crearEstudianteDesdeTeclado() {
    Estudiante e;
    printf("Matricula: "); scanf("%d", &e.matricula);
    printf("Nombre: "); scanf("%[^\\n]", e.nombre);
    printf("Apellido: "); scanf("%[^\\n]", e.apellido);
    printf("Edad: "); scanf("%d", &e.edad);
    int regular;
    printf("¿Es regular? (1: Sí, 0: No): "); scanf("%d", &regular);
    e.esRegular = (regular == 1);
    return e;
}

int main() {
    Nodo* lista = NULL;
    int opcion;

    do {
        printf("\n--- Menú ---\n");
        printf("1. Insertar estudiante al inicio\n");
        printf("2. Insertar estudiante al final\n");
        printf("3. Insertar estudiante en posición\n");
        printf("4. Eliminar estudiante al inicio\n");
        printf("5. Eliminar estudiante al final\n");
        printf("6. Eliminar estudiante en posición\n");
        printf("7. Ver estudiantes\n");
        printf("0. Salir\n");
        printf("Opción: ");
    } while (opcion != 0);
}

```

```

scanf("%d", &opcion);

Estudiante e;
int pos;

switch (opcion) {
    case 1:
        e = crearEstudianteDesdeTeclado();
        insertarInicio(&lista, e);
        break;
    case 2:
        e = crearEstudianteDesdeTeclado();
        insertarFinal(&lista, e);
        break;
    case 3:
        e = crearEstudianteDesdeTeclado();
        printf("Posición: ");
        scanf("%d", &pos);
        insertarEnPosicion(&lista, e, pos);
        break;
    case 4:
        eliminarInicio(&lista);
        break;
    case 5:
        eliminarFinal(&lista);
        break;
    case 6:
        printf("Posición: ");
        scanf("%d", &pos);
        eliminarEnPosicion(&lista, pos);
        break;
    case 7:
        recorrerLista(lista);
        break;
    case 0:
        printf("Saliendo...\n");

```

```

        break;
    default:
        printf("Opción inválida.\n");
    }
} while (opcion != 0);

return 0;
}

```

# Listas ligadas dobles

Parcial AutoEstudio

## ✓ ¿Qué son las listas doblemente ligadas?

Una **lista doblemente ligada** es una estructura de datos donde **cada nodo apunta al siguiente y al anterior**. Esto permite recorrer la lista **en ambos sentidos** (adelante y atrás).

## 📦 Estructura de un nodo:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Estructura de un nodo de lista doblemente ligada
typedef struct Nodo {
    int dato;           // Dato almacenado (puede ser struct en lugar de int)
    struct Nodo* siguiente; // Apunta al siguiente nodo
    struct Nodo* anterior;  // Apunta al nodo anterior
} Nodo;
```

## 📁 Variables de inicio y final de la lista

```
Nodo* inicio = NULL; // Apunta al primer nodo de la lista
Nodo* fin = NULL;    // Apunta al último nodo de la lista
```

## 💠 Función 1: Inserción al inicio

```
bool insertarInicio(Nodo** inicio, Nodo** fin, int valor) {
    Nodo* nuevo = malloc(sizeof(Nodo)); // Reservamos memoria para el nuevo
```

```
    if (!nuevo) return false;           // Si falla la memoria, salimos

    nuevo->dato = valor;                 // Guardamos el dato
    nuevo->anterior = NULL;              // No hay nodo antes, es el nuevo inicio
    nuevo->siguiente = *inicio;          // Su siguiente será el nodo actual de inicio

    if (*inicio != NULL)                 // Si la lista no está vacía
        (*inicio)->anterior = nuevo;    // El nodo que era primero ahora apunta hacia atrás

    else                                 // Si la lista estaba vacía, también es el final
        *fin = nuevo;                   // Si la lista estaba vacía, también es el final

    *inicio = nuevo;                     // Actualizamos el puntero de inicio
    return true;
}
```

## 📌 ¿Qué hace?

- Crea un nuevo nodo al **inicio**.
- Si la lista está vacía, es **inicio y fin**.
- Si no, lo conecta con el nodo anterior y actualiza punteros.

## 💠 Función 2: Inserción al final

```
bool insertarFinal(Nodo** inicio, Nodo** fin, int valor) {
    Nodo* nuevo = malloc(sizeof(Nodo)); // Creamos el nuevo nodo
    if (!nuevo) return false;

    nuevo->dato = valor;
    nuevo->siguiente = NULL;           // No tiene siguiente (es el último)
    nuevo->anterior = *fin;             // Su anterior es el nodo que era último

    if (*fin != NULL)                  // El nodo actual de fin apunta al nuevo
        (*fin)->siguiente = nuevo;    // El nodo actual de fin apunta al nuevo
    else
        *fin = nuevo;
}
```



```

*inicio = nuevo;           // Si estaba vacía, también es el inicio

*fin = nuevo;              // Actualizamos el puntero fin
return true;
}

```

### 📌 ¿Qué hace?

- Crea un nuevo nodo al **final**.
- Si la lista está vacía, también es el inicio.
- Conecta correctamente el nuevo nodo con el anterior.

## ◆ Función 3: Inserción en una posición específica

```

bool insertarEnPosicion(Nodo** inicio, Nodo** fin, int valor, int pos) {
    if (pos < 0) return false;

    if (pos == 0)
        return insertarInicio(inicio, fin, valor); // Insertar al inicio si posición es 0

    Nodo* actual = *inicio;
    int i = 0;

    while (actual != NULL && i < pos - 1) { // Recorremos hasta llegar a la posición
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL)
        return insertarFinal(inicio, fin, valor); // Si llegamos al final, insertamos al final

    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo->dato = valor;

```

```

nuevo->siguiente = actual->siguiente; // El nuevo apunta al siguiente del
nuevo->anterior = actual;           // El nuevo apunta atrás al actual

if (actual->siguiente != NULL)
    actual->siguiente->anterior = nuevo; // El nodo siguiente apunta atrás al
else
    *fin = nuevo;                       // Si no hay siguiente, es el nuevo fin

actual->siguiente = nuevo;              // El nodo actual apunta al nuevo
return true;
}

```

### 📌 ¿Qué hace?

- Inserta un nodo en una posición dada (**0 = inicio**).
- Recorre hasta la posición indicada.
- Ajusta punteros anteriores y siguientes correctamente.

## ◆ Función 4: Eliminación al inicio

```

bool eliminarInicio(Nodo** inicio, Nodo** fin) {
    if (*inicio == NULL) return false; // Lista vacía

    Nodo* temp = *inicio;
    *inicio = temp->siguiente; // Saltamos al siguiente nodo

    if (*inicio != NULL)
        (*inicio)->anterior = NULL; // El nuevo inicio ya no tiene anterior
    else
        *fin = NULL; // Si quedó vacía, fin también es NULL

    free(temp); // Liberamos memoria
    temp = NULL;
}

```

```

return true;
}

```

### 📌 ¿Qué hace?

- Elimina el nodo del **inicio**.
- Si queda vacío, también actualiza el final.
- Libera memoria correctamente.

## ◆ Función 5: Eliminación al final

```

bool eliminarFinal(Nodo** inicio, Nodo** fin) {
    if (*fin == NULL) return false;    // Lista vacía

    Nodo* temp = *fin;
    *fin = temp->anterior;              // Retrocedemos al nodo anterior

    if (*fin != NULL)
        (*fin)->siguiente = NULL;      // El nuevo fin ya no tiene siguiente
    else
        *inicio = NULL;                // Si quedó vacía

    free(temp);
    temp=NULL;
    return true;
}

```

### 📌 ¿Qué hace?

- Elimina el nodo del **final**.
- Actualiza el puntero fin.
- Si queda vacía, inicio también es NULL.

## ◆ Función 6: Eliminación en una posición específica

```

bool eliminarEnPosicion(Nodo** inicio, Nodo** fin, int pos) {
    if (*inicio == NULL || pos < 0) return false;

    Nodo* actual = *inicio;
    int i = 0;

    while (actual != NULL && i < pos) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return false;

    if (actual->anterior != NULL)
        actual->anterior->siguiente = actual->siguiente;
    else
        *inicio = actual->siguiente;    // Si eliminamos el primero

    if (actual->siguiente != NULL)
        actual->siguiente->anterior = actual->anterior;
    else
        *fin = actual->anterior;        // Si eliminamos el último

    free(actual);
    actual = NULL;
    return true;
}

```

### 📌 ¿Qué hace?

- Elimina el nodo en la **posición** `pos`.
- Ajusta enlaces del anterior y siguiente correctamente.
- Actualiza `inicio` y `fin` si es necesario.

## ◆ Función 7: Recorrido hacia adelante

```
void recorrerLista(Nodo* inicio) {
    Nodo* actual = inicio;
    while (actual != NULL) {
        printf("%d ", actual->dato);
        actual = actual->siguiente;
    }
    printf("\n");
}
```

### 📌 ¿Qué hace?

- Recorre e imprime los datos desde el inicio hasta el final.

## ◆ Función 8: Recorrido hacia atrás

```
void recorrerReversa(Nodo* fin) {
    Nodo* actual = fin;
    while (actual != NULL) {
        printf("%d ", actual->dato);
        actual = actual->anterior;
    }
    printf("\n");
}
```

### 📌 ¿Qué hace?

- Recorre e imprime desde el final hasta el inicio.

## Ejemplo de prueba usando int

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Definición de nodo
typedef struct Nodo {
    int dato;
    struct Nodo* siguiente;
    struct Nodo* anterior;
} Nodo;

// Variables globales para inicio y fin
Nodo* inicio = NULL;
Nodo* fin = NULL;

// Funciones
bool insertarInicio(int valor) {
    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo->dato = valor;
    nuevo->anterior = NULL;
    nuevo->siguiente = inicio;

    if (inicio != NULL)
        inicio->anterior = nuevo;
    else
        fin = nuevo;

    inicio = nuevo;
    return true;
}

bool insertarFinal(int valor) {
    Nodo* nuevo = malloc(sizeof(Nodo));
```

```

if (!nuevo) return false;

nuevo->dato = valor;
nuevo->siguiente = NULL;
nuevo->anterior = fin;

if (fin != NULL)
    fin->siguiente = nuevo;
else
    inicio = nuevo;

fin = nuevo;
return true;
}

bool insertarEnPosicion(int valor, int pos) {
    if (pos < 0) return false;
    if (pos == 0) return insertarInicio(valor);

    Nodo* actual = inicio;
    int i = 0;
    while (actual != NULL && i < pos - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return insertarFinal(valor);

    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo->dato = valor;
    nuevo->siguiente = actual->siguiente;
    nuevo->anterior = actual;

    if (actual->siguiente != NULL)

```

```

        actual->siguiente->anterior = nuevo;
    else
        fin = nuevo;

    actual->siguiente = nuevo;
    return true;
}

bool eliminarInicio() {
    if (inicio == NULL) return false;

    Nodo* temp = inicio;
    inicio = inicio->siguiente;

    if (inicio != NULL)
        inicio->anterior = NULL;
    else
        fin = NULL;

    free(temp);
    temp = NULL;
    return true;
}

bool eliminarFinal() {
    if (fin == NULL) return false;

    Nodo* temp = fin;
    fin = fin->anterior;

    if (fin != NULL)
        fin->siguiente = NULL;
    else
        inicio = NULL;

    free(temp);

```

```

    temp = NULL;
    return true;
}

bool eliminarEnPosicion(int pos) {
    if (inicio == NULL || pos < 0) return false;

    Nodo* actual = inicio;
    int i = 0;
    while (actual != NULL && i < pos) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return false;

    if (actual->anterior != NULL)
        actual->anterior->siguiente = actual->siguiente;
    else
        inicio = actual->siguiente;

    if (actual->siguiente != NULL)
        actual->siguiente->anterior = actual->anterior;
    else
        fin = actual->anterior;

    free(actual);
    actual = NULL;
    return true;
}

void recorrerLista() {
    Nodo* actual = inicio;
    printf("Lista: ");
    while (actual != NULL) {
        printf("%d ", actual->dato);

```

```

        actual = actual->siguiente;
    }
    printf("\n");
}

void recorrerReversa() {
    Nodo* actual = fin;
    printf("Lista (reversa): ");
    while (actual != NULL) {
        printf("%d ", actual->dato);
        actual = actual->anterior;
    }
    printf("\n");
}

// Función principal con menú
int main() {
    int opcion, valor, posicion;

    do {
        printf("\n--- MENU LISTA DOBLE ---\n");
        printf("1. Insertar al inicio\n");
        printf("2. Insertar al final\n");
        printf("3. Insertar en posicion\n");
        printf("4. Eliminar al inicio\n");
        printf("5. Eliminar al final\n");
        printf("6. Eliminar en posicion\n");
        printf("7. Mostrar lista\n");
        printf("8. Mostrar reversa\n");
        printf("0. Salir\n");
        printf("Selecciona una opcion: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                printf("Valor a insertar: ");

```

```

        scanf("%d", &valor);
        insertarInicio(valor);
        break;
case 2:
    printf("Valor a insertar: ");
    scanf("%d", &valor);
    insertarFinal(valor);
    break;
case 3:
    printf("Valor a insertar: ");
    scanf("%d", &valor);
    printf("Posicion: ");
    scanf("%d", &posicion);
    insertarEnPosicion(valor, posicion);
    break;
case 4:
    eliminarInicio();
    break;
case 5:
    eliminarFinal();
    break;
case 6:
    printf("Posicion a eliminar: ");
    scanf("%d", &posicion);
    eliminarEnPosicion(posicion);
    break;
case 7:
    recorrerLista();
    break;
case 8:
    recorrerReversa();
    break;
case 0:
    printf("Saliendo...\n");
    break;
default:

```

```

        printf("Opcion invalida.\n");
    }
} while (opcion != 0);

return 0;
}

```

## Aplicado a estudiantes

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

// Estructura Estudiante
typedef struct {
    int matricula;
    char nombre[30];
    bool regular;
} Estudiante;

// Nodo doblemente ligado con Estudiante
typedef struct Nodo {
    Estudiante dato;
    struct Nodo* siguiente;
    struct Nodo* anterior;
} Nodo;

// Punteros globales
Nodo* inicio = NULL;
Nodo* fin = NULL;

// Función para capturar un estudiante desde teclado

```

```

Estudiante capturarEstudiante() {
    Estudiante e;
    printf("Matricula: ");
    scanf("%d", &e.matricula);
    printf("Nombre: ");
    getchar(); // limpiar buffer
    fgets(e.nombre, sizeof(e.nombre), stdin);
    e.nombre[strcspn(e.nombre, "\n")] = '\0'; // quitar salto de línea
    printf("¿Es regular? (1=Sí, 0=No): ");
    scanf("%d", (int*)&e.regular);
    return e;
}

// Función para imprimir un estudiante
void imprimirEstudiante(Estudiante e) {
    printf("[Matricula: %d, Nombre: %s, Regular: %s] ",
        e.matricula, e.nombre, e.regular ? "Sí" : "No");
}

// Inserción al inicio
bool insertarInicio(Estudiante valor) {
    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;
    nuevo->dato = valor;
    nuevo->anterior = NULL;
    nuevo->siguiente = inicio;

    if (inicio != NULL)
        inicio->anterior = nuevo;
    else
        fin = nuevo;

    inicio = nuevo;
    return true;
}

```

```

// Inserción al final
bool insertarFinal(Estudiante valor) {
    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;
    nuevo->dato = valor;
    nuevo->siguiente = NULL;
    nuevo->anterior = fin;

    if (fin != NULL)
        fin->siguiente = nuevo;
    else
        inicio = nuevo;

    fin = nuevo;
    return true;
}

// Inserción en posición
bool insertarEnPosicion(Estudiante valor, int pos) {
    if (pos < 0) return false;
    if (pos == 0) return insertarInicio(valor);

    Nodo* actual = inicio;
    int i = 0;
    while (actual != NULL && i < pos - 1) {
        actual = actual->siguiente;
        i++;
    }

    if (actual == NULL) return insertarFinal(valor);

    Nodo* nuevo = malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo->dato = valor;
    nuevo->siguiente = actual->siguiente;

```

```

nuevo→anterior = actual;

if (actual→siguiente != NULL)
    actual→siguiente→anterior = nuevo;
else
    fin = nuevo;

actual→siguiente = nuevo;
return true;
}

// Eliminación al inicio
bool eliminarInicio() {
    if (inicio == NULL) return false;

    Nodo* temp = inicio;
    inicio = inicio→siguiente;

    if (inicio != NULL)
        inicio→anterior = NULL;
    else
        fin = NULL;

    free(temp);
    temp = NULL;
    return true;
}

// Eliminación al final
bool eliminarFinal() {
    if (fin == NULL) return false;

    Nodo* temp = fin;
    fin = fin→anterior;

    if (fin != NULL)

```

```

        fin→siguiente = NULL;
    else
        inicio = NULL;

    free(temp);
    temp = NULL;
    return true;
}

// Eliminación en posición
bool eliminarEnPosicion(int pos) {
    if (inicio == NULL || pos < 0) return false;

    Nodo* actual = inicio;
    int i = 0;
    while (actual != NULL && i < pos) {
        actual = actual→siguiente;
        i++;
    }

    if (actual == NULL) return false;

    if (actual→anterior != NULL)
        actual→anterior→siguiente = actual→siguiente;
    else
        inicio = actual→siguiente;

    if (actual→siguiente != NULL)
        actual→siguiente→anterior = actual→anterior;
    else
        fin = actual→anterior;

    free(actual);
    actual = NULL;
    return true;
}

```



```

// Recorrido hacia adelante
void recorrerLista() {
    Nodo* actual = inicio;
    printf("\nLista de estudiantes:\n");
    while (actual != NULL) {
        imprimirEstudiante(actual->dato);
        printf("\n");
        actual = actual->siguiente;
    }
}

// Recorrido hacia atrás
void recorrerReversa() {
    Nodo* actual = fin;
    printf("\nLista en reversa:\n");
    while (actual != NULL) {
        imprimirEstudiante(actual->dato);
        printf("\n");
        actual = actual->anterior;
    }
}

// Menú principal
int main() {
    int opcion, posicion;
    Estudiante e;

    do {
        printf("\n--- MENU LISTA DOBLE DE ESTUDIANTES ---\n");
        printf("1. Insertar al inicio\n");
        printf("2. Insertar al final\n");
        printf("3. Insertar en posicion\n");
        printf("4. Eliminar al inicio\n");
        printf("5. Eliminar al final\n");
        printf("6. Eliminar en posicion\n");
    }
}

```

```

printf("7. Mostrar lista\n");
printf("8. Mostrar reversa\n");
printf("0. Salir\n");
printf("Selecciona una opcion: ");
scanf("%d", &opcion);

switch (opcion) {
    case 1:
        e = capturarEstudiante();
        insertarInicio(e);
        break;
    case 2:
        e = capturarEstudiante();
        insertarFinal(e);
        break;
    case 3:
        e = capturarEstudiante();
        printf("Posicion: ");
        scanf("%d", &posicion);
        insertarEnPosicion(e, posicion);
        break;
    case 4:
        eliminarInicio();
        break;
    case 5:
        eliminarFinal();
        break;
    case 6:
        printf("Posicion a eliminar: ");
        scanf("%d", &posicion);
        eliminarEnPosicion(posicion);
        break;
    case 7:
        recorrerLista();
        break;
    case 8:

```

```

        recorrerReversa();
        break;
    case 0:
        printf("Saliendo...\n");
        break;
    default:
        printf("Opcion invalida.\n");
    }
} while (opcion != 0);

return 0;
}

```

# lista simple circular

☐ Parcial

☒ AutoEstudio

## estructura base

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

typedef struct {
    int dato;
    // Aquí puedes agregar otros campos si usas Estudiante
} Elemento;

typedef struct Nodo {
    Elemento info;
    struct Nodo* siguiente;
} Nodo;

```

## ☒ Inserción al inicio

```

bool insertarInicio(Nodo** final, Elemento nuevoDato) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo->info = nuevoDato;
    if (*final == NULL) {
        nuevo->siguiente = nuevo;
        *final = nuevo;
    } else {

```

```

    nuevo→siguiente = (*final)→siguiente;
    (*final)→siguiente = nuevo;
}
return true;
}

```

## ✓ Inserción al final

```

bool insertarFinal(Nodo** final, Elemento nuevoDato) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    if (!nuevo) return false;

    nuevo→info = nuevoDato;
    if (*final == NULL) {
        nuevo→siguiente = nuevo;
        *final = nuevo;
    } else {
        nuevo→siguiente = (*final)→siguiente;
        (*final)→siguiente = nuevo;
        *final = nuevo;
    }
    return true;
}

```

## ✓ Inserción en posición específica

```

bool insertarEnPos(Nodo** final, Elemento nuevoDato, int pos) {
    if (pos < 0) return false;

    if (*final == NULL || pos == 0)
        return insertarInicio(final, nuevoDato);

    Nodo* actual = (*final)→siguiente;
    int i = 0;

```

```

while (i < pos - 1 && actual != *final) {
    actual = actual→siguiente;
    i++;
}

Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
if (!nuevo) return false;

nuevo→info = nuevoDato;
nuevo→siguiente = actual→siguiente;
actual→siguiente = nuevo;

if (actual == *final)
    *final = nuevo;

return true;
}

```

## ✗ Eliminación al inicio

```

bool eliminarInicio(Nodo** final) {
    if (*final == NULL) return false;

    Nodo* inicio = (*final)→siguiente;

    if (*final == inicio) {
        free(inicio);
        *final = NULL;
    } else {
        (*final)→siguiente = inicio→siguiente;
        free(inicio);
    }
    return true;
}

```

## ❌ Eliminación al final

```
bool eliminarFinal(Nodo** final) {
    if (*final == NULL) return false;

    Nodo* actual = (*final)→siguiente;

    if (actual == *final) {
        free(*final);
        *final = NULL;
    } else {
        while (actual→siguiente != *final) {
            actual = actual→siguiente;
        }
        actual→siguiente = (*final)→siguiente;
        free(*final);
        *final = actual;
    }
    return true;
}
```

## ❌ Eliminación en posición específica

```
bool eliminarEnPos(Nodo** final, int pos) {
    if (*final == NULL || pos < 0) return false;

    if (pos == 0)
        return eliminarInicio(final);

    Nodo* actual = (*final)→siguiente;
    int i = 0;

    while (i < pos - 1 && actual→siguiente != (*final)→siguiente) {
        actual = actual→siguiente;
    }
```

```
        i++;
    }

    Nodo* temp = actual→siguiente;

    if (temp == (*final)→siguiente)
        return eliminarInicio(final);

    if (temp == *final)
        *final = actual;

    actual→siguiente = temp→siguiente;
    free(temp);
    temp = NULL;
    return true;
}
```

## 🔄 Recorrer lista circular

```
void recorrer(Nodo* final) {
    if (final == NULL) {
        printf("Lista vacía\n");
        return;
    }

    Nodo* actual = final→siguiente;
    do {
        printf("%d ", actual→info.dato); // Ajusta si usas estructura Estudiante
        actual = actual→siguiente;
    } while (actual != final→siguiente);
    printf("\n");
}
```

## aplicado a Estudiante

```

int main() {
    Nodo* lista = NULL;
    int opcion, pos;
    Estudiante est;

    do {
        printf("\n--- MENU LISTA CIRCULAR DE ESTUDIANTES ---\n");
        printf("1. Insertar al inicio\n");
        printf("2. Insertar al final\n");
        printf("3. Insertar en posición\n");
        printf("4. Eliminar al inicio\n");
        printf("5. Eliminar al final\n");
        printf("6. Eliminar en posición\n");
        printf("7. Mostrar lista\n");
        printf("0. Salir\n");
        printf("Elige una opción: ");
        scanf("%d", &opcion);
        getchar(); // Limpia buffer de enter

        switch (opcion) {
            case 1:
            case 2:
            case 3:
                printf("Matricula: ");
                scanf("%d", &est.matricula);
                getchar(); // limpia enter

                printf("Nombre: ");
                fgets(est.nombre, sizeof(est.nombre), stdin);
                est.nombre[strcspn(est.nombre, "\n")] = 0; // quitar salto

                printf("¿Es regular? (1=Si, 0=No): ");
                scanf("%d", (int*)&est.regular);
                getchar();

```

```

        if (opcion == 1)
            insertarInicio(&lista, est);
        else if (opcion == 2)
            insertarFinal(&lista, est);
        else {
            printf("Posición: ");
            scanf("%d", &pos);
            insertarEnPos(&lista, est, pos);
        }
        break;

    case 4:
        if (!eliminarInicio(&lista))
            printf("No se pudo eliminar. Lista vacía.\n");
        break;

    case 5:
        if (!eliminarFinal(&lista))
            printf("No se pudo eliminar. Lista vacía.\n");
        break;

    case 6:
        printf("Posición: ");
        scanf("%d", &pos);
        if (!eliminarEnPos(&lista, pos))
            printf("No se pudo eliminar. Posición inválida o lista vacía.\n");
        break;

    case 7:
        recorrerLista(lista);
        break;

    case 0:
        printf("Saliendo...\n");
        break;

```

```
        default:
            printf("Opción inválida. Intenta de nuevo.\n");
        }

    } while (opcion != 0);

    return 0;
}
```