Estructuras de datos dinámicas en C

Guía teórica y práctica: listas simplemente enlazadas, doblemente enlazadas y circulares

Las listas enlazadas son estructuras dinámicas que almacenan elementos en nodos conectados mediante punteros. A diferencia de los arreglos, no requieren reservar un bloque contiguo de memoria ni fijar obligatoriamente su tamaño al inicio.

Esta guía presenta los conceptos, las estructuras en C, las operaciones fundamentales, sus complejidades y programas completos de ejemplo.

1. Conceptos fundamentales

1.1. ¿Qué es un nodo?

Un nodo contiene el dato y uno o más punteros a otros nodos. Por ejemplo, un nodo de una lista simplemente enlazada:

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

1.2. Memoria dinámica

malloc() reserva memoria durante la ejecución. Siempre se debe comprobar que la reserva tuvo éxito. free() libera la memoria cuando el nodo deja de utilizarse.

```
Nodo *nuevo = malloc(sizeof(Nodo));

if (nuevo == NULL) {
    printf("Error al reservar memoria\\n");
    return;
}

nuevo->dato = 25;
nuevo->siguiente = NULL;

/* Cuando ya no se necesite: */
free(nuevo);
```

El operador -> permite acceder a los campos de una estructura mediante un puntero. Después de llamar a free(), no se debe volver a acceder al nodo liberado.

2. Lista simplemente enlazada

Cada nodo contiene un dato y un puntero al siguiente. La cabeza identifica el primer nodo y el último nodo apunta a NULL.

Representación: cabeza → [10] → [20] → [30] → NULL

2.1. Estructura

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

Nodo *cabeza = NULL;
```

2.2. Operaciones y complejidad

Operación	Descripción	Complejidad
Insertar al inicio	Agrega antes de la cabeza.	O(1)
Insertar al final	Recorre hasta el último nodo si no hay cola.	O(n)
Buscar	Examina nodos hasta encontrar el dato.	O(n)
Recorrer	Visita todos los nodos.	O(n)
Eliminar por valor	Busca y elimina la primera coincidencia.	O(n)
Liberar la lista	Libera todos los nodos.	O(n)

2.3. Código completo

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

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

Nodo *crearNodo(int dato) {
    Nodo *nuevo = malloc(sizeof(Nodo));
    if (nuevo == NULL) {
        printf("Error al reservar memoria.\n");
        exit(EXIT_FAILURE);
    }
    nuevo->dato = dato;
    nuevo->siguiente = NULL;
    return nuevo;
}

void insertarInicio(Nodo **cabeza, int dato) {
    Nodo *nuevo = crearNodo(dato);
    nuevo->siguiente = *cabeza;
    *cabeza = nuevo;
}

void insertarFinal(Nodo **cabeza, int dato) {
    Nodo *nuevo = crearNodo(dato);
    if (*cabeza == NULL) {
        *cabeza = nuevo;
        return;
    }
    Nodo *actual = *cabeza;
    while (actual->siguiente != NULL)
        actual = actual->siguiente;
    actual->siguiente = nuevo;
}

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

Nodo *buscar(Nodo *cabeza, int dato) {
    Nodo *actual = cabeza;
    while (actual != NULL) {
        if (actual->dato == dato) return actual;
        actual = actual->siguiente;
    }
    return NULL;
}

int eliminar(Nodo **cabeza, int dato) {
    Nodo *actual = *cabeza;
    Nodo *anterior = NULL;

    while (actual != NULL && actual->dato != dato) {
        anterior = actual;
        actual = actual->siguiente;
    }
    if (actual == NULL) return 0;

    if (anterior == NULL)
        *cabeza = actual->siguiente;
    else
        anterior->siguiente = actual->siguiente;

    free(actual);
    return 1;
}

void liberarLista(Nodo **cabeza) {
    Nodo *actual = *cabeza;
    while (actual != NULL) {
        Nodo *temporal = actual;
```

```
        actual = actual->siguiente;
        free(temporal);
    }
    *cabeza = NULL;
}

int main(void) {
    Nodo *cabeza = NULL;

    insertarInicio(&cabeza, 20);
    insertarInicio(&cabeza, 10);
    insertarFinal(&cabeza, 30);
    insertarFinal(&cabeza, 40);

    printf("Lista inicial:\n");
    recorrer(cabeza);

    printf("El valor 30 %s\n",
           buscar(cabeza, 30) ? "existe" : "no existe");

    if (eliminar(&cabeza, 20))
        printf("Se elimino el valor 20.\n");

    recorrer(cabeza);
    liberarLista(&cabeza);
    return 0;
}
```

3. Lista doblemente enlazada

Cada nodo contiene dos enlaces: anterior y siguiente. Por ello, se puede recorrer la lista desde la cabeza hacia adelante y desde la cola hacia atrás.

Representación: NULL ← [10] ⇄ [20] ⇄ [30] → NULL

3.1. Estructura

```
typedef struct NodoDoble {
    int dato;
    struct NodoDoble *anterior;
    struct NodoDoble *siguiente;
} NodoDoble;
```

3.2. Operaciones y complejidad

Operación	Descripción	Complejidad
Insertar al inicio	Actualiza cabeza y enlace anterior.	O(1)
Insertar al final	O(1) si se mantiene una cola.	O(1)*
Recorrer adelante/atrás	Sigue siguiente o anterior.	O(n)
Buscar	Examina nodos hasta hallar el dato.	O(n)
Eliminar nodo localizado	Reconecta los vecinos.	O(1)*
Eliminar por valor	Busca y después elimina.	O(n)

* La eliminación de un nodo localizado es O(1) si se tiene su puntero y se actualizan correctamente cabeza y cola.

3.3. Código completo

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

typedef struct NodoDoble {
    int dato;
    struct NodoDoble *anterior;
    struct NodoDoble *siguiente;
} NodoDoble;

NodoDoble *crearNodo(int dato) {
    NodoDoble *nuevo = malloc(sizeof(NodoDoble));
    if (nuevo == NULL) {
        printf("Error al reservar memoria.\n");
        exit(EXIT_FAILURE);
    }
    nuevo->dato = dato;
    nuevo->anterior = NULL;
    nuevo->siguiente = NULL;
    return nuevo;
}

void insertarInicio(NodoDoble **cabeza,
                    NodoDoble **cola, int dato) {
    NodoDoble *nuevo = crearNodo(dato);
    if (*cabeza == NULL) {
        *cabeza = *cola = nuevo;
        return;
    }
    nuevo->siguiente = *cabeza;
    (*cabeza)->anterior = nuevo;
    *cabeza = nuevo;
}

void insertarFinal(NodoDoble **cabeza,
                   NodoDoble **cola, int dato) {
    NodoDoble *nuevo = crearNodo(dato);
    if (*cola == NULL) {
        *cabeza = *cola = nuevo;
        return;
    }
    nuevo->anterior = *cola;
    (*cola)->siguiente = nuevo;
    *cola = nuevo;
}

void recorrerAdelante(NodoDoble *cabeza) {
    for (NodoDoble *p = cabeza; p != NULL; p = p->siguiente)
        printf("%d <-> ", p->dato);
    printf("NULL\n");
}

void recorrerAtras(NodoDoble *cola) {
    for (NodoDoble *p = cola; p != NULL; p = p->anterior)
        printf("%d <-> ", p->dato);
    printf("NULL\n");
}

NodoDoble *buscar(NodoDoble *cabeza, int dato) {
    NodoDoble *p = cabeza;
    while (p != NULL) {
        if (p->dato == dato) return p;
        p = p->siguiente;
    }
    return NULL;
}

int eliminar(NodoDoble **cabeza,
             NodoDoble **cola, int dato) {
    NodoDoble *p = buscar(*cabeza, dato);
    if (p == NULL) return 0;

    if (p->anterior != NULL)
        p->anterior->siguiente = p->siguiente;
    else
        *cabeza = p->siguiente;

    if (p->siguiente != NULL)
        p->siguiente->anterior = p->anterior;
```

```
    else
        *cola = p->anterior;

    free(p);
    return 1;
}

void liberarLista(NodoDoble **cabeza, NodoDoble **cola) {
    NodoDoble *p = *cabeza;
    while (p != NULL) {
        NodoDoble *temporal = p;
        p = p->siguiente;
        free(temporal);
    }
    *cabeza = NULL;
    *cola = NULL;
}

int main(void) {
    NodoDoble *cabeza = NULL, *cola = NULL;

    insertarInicio(&cabeza, &cola, 20);
    insertarInicio(&cabeza, &cola, 10);
    insertarFinal(&cabeza, &cola, 30);
    insertarFinal(&cabeza, &cola, 40);

    printf("Adelante:\n");
    recorrerAdelante(cabeza);
    printf("Atras:\n");
    recorrerAtras(cola);

    eliminar(&cabeza, &cola, 30);
    printf("Despues de eliminar 30:\n");
    recorrerAdelante(cabeza);

    liberarLista(&cabeza, &cola);
    return 0;
}
```

4. Lista circular simplemente enlazada

El último nodo apunta al primero, en vez de apuntar a NULL. En este ejemplo se mantiene un puntero cola, y el primer nodo se obtiene con cola->siguiente.

Representación: [10] → [20] → [30] ─┐ y [30] vuelve a [10].

4.1. Estructura y casos

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

/* Lista vacia: cola == NULL */
/* Un nodo: cola->siguiente == cola */
```

4.2. Operaciones y complejidad

Operación	Descripción	Complejidad
Insertar al inicio	Actualiza el primer enlace.	O(1)
Insertar al final	Actualiza cola y el enlace circular.	O(1)
Recorrer	Termina cuando vuelve al inicio.	O(n)
Buscar	Visita como máximo una vuelta.	O(n)
Eliminar por valor	Busca y reconecta el ciclo.	O(n)
Liberar	Rompe el ciclo y libera cada nodo.	O(n)

4.3. Código completo

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

typedef struct NodoCircular {
    int dato;
    struct NodoCircular *siguiente;
} NodoCircular;

NodoCircular *crearNodo(int dato) {
    NodoCircular *nuevo = malloc(sizeof(NodoCircular));
    if (nuevo == NULL) {
        printf("Error al reservar memoria.\n");
        exit(EXIT_FAILURE);
    }
    nuevo->dato = dato;
    nuevo->siguiente = NULL;
    return nuevo;
}

void insertarInicio(NodoCircular **cola, int dato) {
    NodoCircular *nuevo = crearNodo(dato);
    if (*cola == NULL) {
        nuevo->siguiente = nuevo;
        *cola = nuevo;
        return;
    }
    nuevo->siguiente = (*cola)->siguiente;
    (*cola)->siguiente = nuevo;
}

void insertarFinal(NodoCircular **cola, int dato) {
    NodoCircular *nuevo = crearNodo(dato);
    if (*cola == NULL) {
        nuevo->siguiente = nuevo;
        *cola = nuevo;
        return;
    }
    nuevo->siguiente = (*cola)->siguiente;
    (*cola)->siguiente = nuevo;
    *cola = nuevo;
}

void recorrer(NodoCircular *cola) {
    if (cola == NULL) {
        printf("Lista vacia.\n");
        return;
    }
    NodoCircular *inicio = cola->siguiente;
    NodoCircular *p = inicio;
    do {
        printf("%d -> ", p->dato);
        p = p->siguiente;
    } while (p != inicio);
    printf("(vuelve al inicio)\n");
}

NodoCircular *buscar(NodoCircular *cola, int dato) {
    if (cola == NULL) return NULL;
    NodoCircular *inicio = cola->siguiente;
    NodoCircular *p = inicio;
    do {
        if (p->dato == dato) return p;
        p = p->siguiente;
    } while (p != inicio);
    return NULL;
}

int eliminar(NodoCircular **cola, int dato) {
    if (*cola == NULL) return 0;

    NodoCircular *p = (*cola)->siguiente;
    NodoCircular *anterior = *cola;

    do {
        if (p->dato == dato) {
            if (p == anterior) {
                *cola = NULL; /* Un solo nodo */
            } else {
```

```
                anterior->siguiente = p->siguiente;
                if (p == *cola) *cola = anterior;
            }
            free(p);
            return 1;
        }
        anterior = p;
        p = p->siguiente;
    } while (p != (*cola)->siguiente);

    return 0;
}

void liberarLista(NodoCircular **cola) {
    if (*cola == NULL) return;

    NodoCircular *inicio = (*cola)->siguiente;
    (*cola)->siguiente = NULL; /* Romper el ciclo */

    NodoCircular *p = inicio;
    while (p != NULL) {
        NodoCircular *temporal = p;
        p = p->siguiente;
        free(temporal);
    }
    *cola = NULL;
}

int main(void) {
    NodoCircular *cola = NULL;

    insertarInicio(&cola, 20);
    insertarInicio(&cola, 10);
    insertarFinal(&cola, 30);
    insertarFinal(&cola, 40);

    printf("Lista circular:\n");
    recorrer(cola);

    printf("El valor 30 %s\n",
           buscar(cola, 30) ? "existe" : "no existe");

    eliminar(&cola, 10);
    recorrer(cola);

    liberarLista(&cola);
    return 0;
}
```

5. Comparación de las estructuras

Característica	Simple	Doble	Circular
Punteros por nodo	1	2	1
Recorrido hacia adelante	Sí	Sí	Sí
Recorrido hacia atrás	No	Sí	No
Enlace del último	NULL	NULL	Al primero
Memoria por nodo	Menor	Mayor	Menor
Insertar al inicio	O(1)	O(1)	O(1)
Insertar al final	O(n)*	O(1)*	O(1)*

* Las complejidades de inserción al final suponen las implementaciones descritas: la lista simple no conserva cola, la doble conserva cola y la circular conserva cola.

6. ¿Cuándo utilizar cada estructura?

Lista simplemente enlazada

Conviene cuando los datos se recorren principalmente en una dirección: secuencias de tareas, registros o estructuras auxiliares.

Lista doblemente enlazada

Es útil en historiales de navegación, listas de reproducción y funciones de deshacer/rehacer, donde se necesita avanzar y retroceder.

Lista circular

Es apropiada para turnos de jugadores, planificación cíclica de procesos y rotación de tareas.

7. Compilar y ejecutar

Guarda cada programa en un archivo independiente. Con GCC, compila cada uno así:

```
gcc lista_simple.c -o lista_simple
gcc lista_doble.c -o lista_doble
gcc lista_circular.c -o lista_circular
```

En Linux o macOS ejecuta ./lista_simple, ./lista_doble y ./lista_circular. En Windows con MinGW ejecuta lista_simple.exe, lista_doble.exe y lista_circular.exe.

8. Ejercicios propuestos

• Agregar una operación para contar los nodos de cada lista.

• Implementar la inserción en una posición indicada por el usuario.

• Eliminar todas las apariciones de un número, no solo la primera.

• Crear funciones para calcular la suma y el promedio de los datos.

• Implementar una lista circular para administrar los turnos de cuatro jugadores.

Recomendación didáctica: comenzar por la lista simplemente enlazada para comprender struct, punteros, memoria dinámica y modificación de enlaces. Después estudiar la doblemente enlazada y, finalmente, la circular, prestando especial atención a las condiciones de terminación y a la liberación de memoria.