CAPÍTULO

1

INTRODUCCIÓN

✧ ✧ ✧

Extraído del capítulo 1 de **Data Structures and Algorithms Made Easy**, págs. 16–19

El objetivo de este capítulo es explicar la importancia del análisis de algoritmos, sus notaciones, sus relaciones y la resolución de tantos problemas como sea posible. Primero, concentrémonos en comprender los elementos básicos de los algoritmos, la importancia del análisis de algoritmos y, luego, avancemos lentamente hacia los otros temas mencionados antes. Después de completar este capítulo, usted debería ser capaz de encontrar la complejidad de cualquier algoritmo dado (especialmente de las funciones recursivas).

1.1 Variables

Antes de pasar a la definición de variables, relacionémoslas con antiguas ecuaciones matemáticas. Todos nosotros hemos resuelto muchas ecuaciones matemáticas desde la infancia. Como ejemplo, considere la siguiente ecuación:

$$x^2 + 2y - 2 = 1$$

No tenemos que preocuparnos por el uso de esta ecuación. Lo importante que necesitamos entender es que la ecuación tiene nombres (x e y), que contienen valores (datos). Eso significa que los nombres (x e y) son marcadores de posición para representar datos. De manera similar, en la programación de ciencias de la computación necesitamos algo para contener datos, y las *variables* son la forma de hacerlo.

1.2 Tipos de datos

En la ecuación mencionada antes, las variables x e y pueden tomar cualquier valor, como números enteros (10, 20), números reales (0.23, 5.5), o simplemente 0 y 1. Para resolver la ecuación, necesi-

tamos relacionarlas con el tipo de valores que pueden tomar, y *tipo de dato* es el nombre usado en la programación de ciencias de la computación para este propósito. Un *tipo de dato* en un lenguaje de programación es un conjunto de datos con valores predefinidos. Ejemplos de tipos de datos son: integer, floating point, unit number, character, string, etc.

La memoria de la computadora está completamente llena de ceros y unos. Si tenemos un problema y queremos codificarlo, es muy difícil proporcionar la solución en términos de ceros y unos. Para ayudar a los usuarios, los lenguajes de programación y los compiladores nos proporcionan tipos de datos. Por ejemplo, *integer* ocupa 2 bytes (el valor real depende del compilador), *float* ocupa 4 bytes, etc. Esto dice que en memoria estamos combinando 2 bytes (16 bits) y llamándolo un *integer*. De manera similar, combinamos 4 bytes (32 bits) y lo llamamos un *float*. Un tipo de dato reduce el esfuerzo de codificación. En el nivel superior, hay dos tipos de tipos de datos:

- Tipos de datos definidos por el sistema (también llamados tipos de datos *primitivos*)
- Tipos de datos definidos por el usuario

Tipos de datos definidos por el sistema (tipos de datos primitivos)

Los tipos de datos definidos por el sistema se llaman tipos de datos *primitivos*. Los tipos de datos primitivos proporcionados por muchos lenguajes de programación son: int, float, char, double, bool, etc. La cantidad de bits asignada a cada tipo de dato primitivo depende de los lenguajes de programación, del compilador y del sistema operativo. Para el mismo tipo de dato primitivo, distintos lenguajes pueden usar tamaños diferentes. Dependiendo del tamaño de los tipos de datos, el total de valores disponibles (dominio) también cambiará.

Por ejemplo, *"int"* puede ocupar 2 bytes o 4 bytes. Si ocupa 2 bytes (16 bits), entonces el total de valores posibles va de −32,768 a +32,767 (-2^{15} a $2^{15}-1$). Si ocupa 4 bytes (32 bits), entonces los valores posibles están entre −2,147,483,648 y +2,147,483,647 (-2^{31} a $2^{31}-1$). Lo mismo ocurre con otros tipos de datos.

Tipos de datos definidos por el usuario

Si los tipos de datos definidos por el sistema no son suficientes, entonces la mayoría de los lenguajes de programación permiten a los usuarios definir sus propios *tipos de datos definidos por el usuario*. Buenos ejemplos de tipos de datos definidos por el usuario son: estructuras en *C/C++* y clases en *Java*. Por ejemplo, en el fragmento que aparece debajo, estamos combinando muchos tipos de datos definidos por el sistema y llamando al tipo de dato definido por el usuario con el nombre *"newType"*. Esto da más flexibilidad y comodidad al tratar con la memoria de la computadora.

```
struct newType {
  int data1;
  float data2;
  ...
  char data;
};
```

1.3 Estructuras de datos

Con base en la discusión anterior, una vez que tenemos datos en variables, necesitamos algún mecanismo para manipular esos datos y resolver problemas. Una *estructura de datos* es una manera particular de almacenar y organizar datos en una computadora para que puedan usarse eficientemente. Una *estructura de datos* es un formato especial para organizar y almacenar datos. Los tipos generales de estructuras de datos incluyen arrays, archivos, listas enlazadas, pilas, colas, árboles, grafos, etc.

Dependiendo de la organización de los elementos, las estructuras de datos se clasifican en dos tipos:

1) *Estructuras de datos lineales:* se accede a los elementos en un orden secuencial, pero no es obligatorio almacenar todos los elementos secuencialmente. *Ejemplos:* listas enlazadas, pilas y colas.
2) *Estructuras de datos no lineales:* los elementos de esta estructura de datos se almacenan o se accede a ellos en un orden no lineal. *Ejemplos:* árboles y grafos.

1.4 Tipos de Datos Abstractos (ADTs)

Antes de definir los Tipos de Datos Abstractos, consideremos la visión diferente de los tipos de datos definidos por el sistema. Todos sabemos que, por defecto, todos los tipos de datos primitivos (int, float, etc.) admiten operaciones básicas como suma y resta. El sistema proporciona las implementaciones para los tipos de datos primitivos. Para los tipos de datos definidos por el usuario también necesitamos definir operaciones. La implementación de estas operaciones puede hacerse cuando realmente queremos usarlas. Eso significa que, en general, los tipos de datos definidos por el usuario se definen junto con sus operaciones.

Para simplificar el proceso de resolución de problemas, combinamos las estructuras de datos con sus operaciones y llamamos a esto *Tipos de Datos Abstractos* (ADTs). Un ADT consta de *dos partes:*

1) Declaración de datos
2) Declaración de operaciones

Los ADTs de uso común *incluyen:* listas enlazadas, pilas, colas, colas de prioridad, árboles binarios, diccionarios, conjuntos disjuntos (Union and Find), tablas hash, grafos y muchos otros. Por ejemplo, una pila usa el mecanismo LIFO (Last-In-First-Out) mientras almacena los datos en estructuras de

datos. El último elemento insertado en la pila es el primer elemento que se elimina. Sus operaciones comunes son: crear la pila, insertar un elemento en la pila, extraer un elemento de la pila, encontrar el tope actual de la pila, encontrar el número de elementos en la pila, etc.

Mientras se definen los ADTs, no se preocupe por los detalles de implementación. Estos aparecen solo cuando queremos usarlos. Diferentes clases de ADTs se adaptan a diferentes clases de aplicaciones, y algunos están altamente especializados para tareas específicas. Al final de este libro, habremos recorrido muchos de ellos y usted estará en condiciones de relacionar las estructuras de datos con el tipo de problemas que resuelven.

CAPÍTULO 2

Punteros y memoria dinámica en C

Harvey M. Deitel y Paul J. Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed. (2004), caps. 7, 10, 12 y 14; K. N. King, **C Programming: A Modern Approach**, 2.ª ed., caps. 11, 17 y 19.

Este capítulo introduce una idea que será necesaria para estudiar nodos y listas: un programa puede trabajar con direcciones, reservar espacio durante su ejecución y liberar ese espacio cuando deja de necesitarlo. La progresión va desde una variable ordinaria hasta un bloque de memoria administrado explícitamente.

La traducción base de este bloque se mantiene separada de las decisiones editoriales. Los ejemplos están reescritos en C moderno: no convierten explícitamente el resultado de `malloc`, incluyen `<stdlib.h>`, comprueban los fallos de asignación y usan `size_t` para cantidades y capacidades.

2.1 Punteros, direcciones y desreferenciación

La memoria principal puede imaginarse como una sucesión de bytes. Cada byte tiene una dirección que permite distinguirlo de los demás. Una variable ocupa uno o más bytes; la dirección de la variable es la dirección de su primer byte.

Un puntero es un objeto cuyo valor puede ser una dirección. Si `p` contiene la dirección de `x`, decimos que `p` apunta a `x`. El puntero y el objeto apuntado son cosas distintas: cambiar `p` cambia la dirección almacenada en `p`, mientras que cambiar `*p` cambia el objeto al que `p` apunta.

El operador `&` obtiene la dirección de un objeto. El operador `*`, usado como operador de indirección o desreferenciación, accede al objeto ubicado en la dirección almacenada en un puntero.

Objeto	Dirección	Contenido
x	`0x1000`	42
p	`0x2000`	apunta a `0x1000`

Tabla 1: Redibujo conceptual de referencia directa e indirecta.

Redibujo propio basado en Deitel, cap. 7, fig. 7.1, págs. 234–235, y King, cap. 11, págs. 241–244.

Una declaración como `int *p;` reserva espacio para un puntero a `int`, pero no inicializa ese puntero para que apunte a un `int` válido. Antes de aplicar `*p`, `p` debe contener una dirección válida. Una forma segura de expresar que todavía no apunta a un objeto es inicializarlo con `NULL`:

```c
#include <stddef.h>

int value = 42;
int *p = &value;
int *not_ready = NULL;

printf("%d\n", *p);       /* accede a value */
/* *not_ready;            ERROR: no hay objeto apuntado */
```

`NULL` representa un puntero nulo, no una dirección que el programa pueda desreferenciar. El valor de un puntero nulo puede comprobarse antes de usarlo. Un puntero no inicializado, en cambio, contiene un valor indeterminado y no debe leerse ni desreferenciarse.

> **Procedencia:** Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed., cap. 7, secciones 7.1–7.4, págs. 233–247; King, **C Programming: A Modern Approach**, 2.ª ed., cap. 11, secciones 11.1–11.3, págs. 241–247.

2.2 Punteros como argumentos

C pasa los argumentos por valor. Una función que recibe un `int` recibe una copia del entero; modificar esa copia no modifica la variable del llamador. Para que una función pueda modificar el objeto original, se le pasa su dirección y se declara un parámetro puntero al tipo correspondiente.

```c
static void intercambiar(int *a, int *b)
{
    if (a == NULL || b == NULL) {
        return;
    }

    int temporal = *a;
    *a = *b;
    *b = temporal;
}

int izquierda = 3;
int derecha = 8;
intercambiar(&izquierda, &derecha);
```

Dentro de `intercambiar`, `*a` es un alias para `izquierda` y `*b` es un alias para `derecha`. No se trata de una llamada por referencia incorporada al lenguaje: el mecanismo es pasar por valor una dirección y utilizar la indirección para acceder al objeto original.

El llamador conserva la propiedad de `izquierda` y `derecha`. La función solo recibe permiso para modificar esos objetos mientras sus direcciones sigan siendo válidas. Esta distinción será importante

más adelante, cuando una función reciba un puntero a memoria dinámica y deba documentarse quién puede liberarla.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 11, sección 11.4, págs. 247–251; Deitel, cap. 7, secciones 7.4–7.6, págs. 238–250.

2.3 `sizeof` y estructuras

El operador `sizeof` produce el tamaño, en bytes, de un tipo o de una expresión. Su resultado tiene tipo `size_t`, definido por las bibliotecas estándar. El tamaño de un tipo depende de la implementación; por eso no conviene sustituirlo por una cantidad escrita a mano.

```c
#include <stdio.h>
#include <stddef.h>

struct muestra {
    char etiqueta;
    int valor;
};

struct muestra dato = {.etiqueta = 'A', .valor = 97};

printf("%zu\n", sizeof dato);
printf("%zu\n", sizeof dato.valor);
```

Una estructura reúne miembros que pueden tener tipos diferentes. El compilador puede insertar bytes de relleno entre miembros para respetar las reglas de alineación de la máquina. Por eso `sizeof(struct muestra)` puede ser mayor que la suma intuitiva de `sizeof(char)` y `sizeof(int)`.

byte	0	1	2–3
miembro	`etiqueta`	relleno posible	`valor`

Tabla 2: Redibujo conceptual de alineación y relleno en una estructura.

Redibujo propio basado en Deitel, cap. 10, fig. 10.1, pág. 358. La ubicación exacta del relleno depende de la implementación.

Los operadores de acceso se eligen según el operando. Para una variable de estructura se usa el punto, `dato.valor`. Para un puntero a estructura se usa la flecha, `ptr->valor`. La expresión `ptr->valor` equivale a `(*ptr).valor`; los paréntesis son necesarios porque el punto tiene mayor precedencia que `*`.

```c
struct muestra *ptr = &dato;
printf("%c %d\n", dato.etiqueta, dato.valor);
printf("%c %d\n", ptr->etiqueta, ptr->valor);
```

Expresión	Operador	Lectura
`dato.valor`	`.`	miembro de la estructura `dato`
`ptr->valor`	`->`	miembro del objeto apuntado por `ptr`
`(*ptr).valor`	`.`	misma operación, escrita en dos pasos

Tabla 3: Redibujo conceptual de los operadores `.` y `->`.

Redibujo propio basado en Deitel, cap. 10, fig. 10.2, págs. 359–360.

> **Procedencia:** Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed., cap. 7, secciones 7.7–7.10, págs. 250–258, y cap. 10, secciones 10.2–10.4, págs. 356–360.

2.4 Estructuras autorreferenciadas

Una estructura no puede contener una instancia completa de sí misma: eso produciría un tamaño infinito. Sí puede contener un puntero a otra instancia del mismo tipo, porque el tamaño de un puntero es conocido sin conocer el tamaño del objeto al que apunta.

```
struct nodo {
    int dato;
    struct nodo *siguiente;
};
```

El miembro `siguiente` es una liga. Si contiene `NULL`, representa que no hay otro nodo después. Esta declaración define el tipo; todavía no reserva un nodo ni una lista. La reserva ocurre cuando se declara una variable o cuando se solicita memoria dinámica.

nodo A	liga	nodo B
`dato = 10`	⟶	`dato = 20`
`siguiente`	`siguiente = NULL`	`fin`

Tabla 4: Redibujo conceptual de dos nodos enlazados y el `NULL` final.

Redibujo propio basado en Deitel, cap. 12, fig. 12.1, pág. 423, y fig. 12.2, pág. 425.

La estructura autorreferenciada es el puente entre punteros y listas, pero en este capítulo solo se presenta el modelo. Las operaciones de insertar, buscar y eliminar pertenecen al Bloque 3 — Listas enlazadas.

> **Procedencia:** Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed., cap. 10, págs. 356–360, y cap. 12, sección 12.2, pág. 423; King, cap. 17, sección 17.5, págs. 424–426.

2.5 Memoria dinámica y heap

Las variables automáticas suelen vivir mientras se ejecuta el bloque que las contiene. Un array declarado con una longitud fija conserva esa longitud durante toda su vida. La asignación dinámica permite solicitar un bloque de memoria durante la ejecución, cuando el programa ya conoce cuánto espacio necesita.

Ese almacenamiento se obtiene del **heap** o almacén dinámico. La biblioteca no conoce el tipo de los datos que se guardarán allí, por lo que las funciones de asignación devuelven un `void *`, un puntero genérico que C puede convertir implícitamente al asignarlo a un puntero de objeto. El programa debe conservar el puntero y administrar el ciclo de vida del bloque.

El ciclo básico es:

1) calcular la cantidad de bytes;
2) reservar el bloque;
3) comprobar el resultado;
4) inicializar y usar los objetos;
5) liberar el bloque exactamente una vez cuando ya no sea necesario.

solicitar	inicializar	usar	liberar
`malloc`	`ptr->dato = 10`	`ptr->dato`	`free(ptr)`

Tabla 5: Redibujo conceptual del ciclo de vida de una reserva.

Redibujo propio basado en Deitel, cap. 12, sección 12.3, págs. 423–424, y King, cap. 17, secciones 17.1 y 17.4, págs. 414–416 y 422–424.

2.6 `malloc`, `NULL` y comprobación de errores

`malloc` reserva la cantidad de bytes indicada y deja su contenido sin inicializar. La forma idiomática de calcular el espacio para un objeto es usar el tipo de la expresión que se va a almacenar, no repetir el nombre del tipo:

```c
#include <stdint.h>
#include <stdlib.h>

size_t cantidad = 4;
if (cantidad > SIZE_MAX / sizeof(int)) {
    return EXIT_FAILURE; /* overflow de tamaño */
}

int *valores = malloc(cantidad * sizeof *valores);
if (valores == NULL) {
    return EXIT_FAILURE; /* no se obtuvo el bloque */
}

for (size_t i = 0; i < cantidad; i++) {
    valores[i] = 0;
}

free(valores);
valores = NULL;
```

No se debe escribir `int *valores = (int *) malloc(...)` en C. El cast no es necesario y puede ocultar la falta de `<stdlib.h>`. `sizeof *valores` sigue siendo correcto aunque más adelante cambie el tipo del puntero.

Un fallo de asignación no debe tratarse como un caso imposible. Si `malloc` devuelve `NULL`, no existe un bloque que pueda desreferenciarse ni liberarse como si se hubiera obtenido. El programa puede terminar, informar el error o propagarlo al llamador, según el contrato de la función.

> **Procedencia:** Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed., cap. 12, sección 12.3, págs. 423–424; King, cap. 17, sección 17.1, págs. 414–416.

2.7 `free` y ciclo de vida

`free` devuelve al sistema un bloque obtenido previamente por `malloc`, `calloc` o `realloc`. Liberar no borra una copia del objeto: invalida el acceso a ese bloque. Después de `free(ptr)`, el valor de `ptr` ya no debe usarse para acceder al objeto.

Una práctica útil cuando el puntero sigue en alcance es asignarle `NULL` inmediatamente después de liberar:

```c
struct nodo *nodo = malloc(sizeof *nodo);
if (nodo == NULL) {
    return EXIT_FAILURE;
}

nodo->dato = 7;
free(nodo);
nodo = NULL;
```

Asignar `NULL` evita que esa variable conserve una dirección que ya no representa almacenamiento válido. No corrige copias del mismo puntero que existan en otros lugares: la responsabilidad de ownership debe estar definida para que solo una parte del programa libere cada bloque.

> **Procedencia:** Deitel, cap. 12, sección 12.3, págs. 423–424; King, cap. 17, sección 17.4, págs. 422–424.

2.8 Fugas y punteros colgantes

Hay una fuga (**memory leak**) cuando el programa pierde toda posibilidad de llegar a un bloque que todavía no fue liberado. El bloque permanece ocupado, pero ya no puede reutilizarse. Un caso típico es sobrescribir el único puntero que lo guardaba:

```c
int *leak = malloc(sizeof *leak);
if (leak != NULL) {
    *leak = 10;
    leak = NULL; /* ERROR: se perdió la dirección sin llamar a free */
}
```

Un puntero colgante (**dangling pointer**) conserva la dirección de un bloque después de que ese bloque fue liberado. Desreferenciarlo es un **use-after-free** y produce comportamiento indefinido. Liberarlo otra vez es un **double-free**, también inválido:

```c
int *p = malloc(sizeof *p);
if (p != NULL) {
    free(p);
    /* *p = 4;  ERROR: use-after-free */
    /* free(p);  ERROR: double-free */
    p = NULL;
}
```

Estos fragmentos están marcados como errores conceptuales; no forman parte de un programa correcto. La corrección consiste en mantener una única responsabilidad de liberación, no perder el puntero antes de liberar y dejar de usarlo después de `free`.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 17, sección 17.4, págs. 422–424; Deitel, cap. 12, sección 12.3, págs. 423–424.

2.9 `calloc`

`calloc` reserva espacio para una cantidad de elementos y un tamaño por elemento. A diferencia de `malloc`, inicializa el bloque a cero. Al igual que con `malloc`, hay que verificar el resultado y liberar el bloque cuando deja de ser necesario.

```c
#include <stdint.h>
#include <stdlib.h>

size_t cantidad = 8;
if (cantidad > SIZE_MAX / sizeof(int)) {
    return EXIT_FAILURE;
}

int *contadores = calloc(cantidad, sizeof *contadores);
if (contadores == NULL) {
    return EXIT_FAILURE;
}

/* contadores[0] ... contadores[7] comienzan en cero */
free(contadores);
contadores = NULL;
```

La forma con dos argumentos deja expresada la cantidad de elementos y evita calcular manualmente el producto en la llamada. Sigue siendo necesario validar tamaños provenientes de una entrada externa y definir qué significa una cantidad igual a cero para la función que la recibe.

> **Procedencia:** Deitel, **Cómo programar en C/C++ y Java**, 4.ª ed., cap. 14, sección 14.11, pág. 494; King, cap. 17, secciones 17.1 y 17.3, págs. 414–416 y 420–422.

2.10 `realloc` y arrays dinámicos

`realloc` cambia el tamaño de un bloque previamente reservado. Puede ampliar el bloque en el mismo lugar o moverlo a otra ubicación. Si la operación falla, devuelve `NULL` y el bloque original sigue siendo válido. Por eso no se debe sobrescribir directamente el único puntero con el resultado.

```c
#include <stdint.h>
#include <stdlib.h>

size_t vieja_cantidad = 4;
size_t nueva_cantidad = 8;
int *valores = malloc(vieja_cantidad * sizeof *valores);
if (valores == NULL) {
    return EXIT_FAILURE;
}

if (nueva_cantidad > SIZE_MAX / sizeof *valores) {
    free(valores);
    return EXIT_FAILURE;
}

int *temporal = realloc(valores, nueva_cantidad * sizeof *temporal);
if (temporal == NULL) {
    /* valores sigue apuntando al bloque original */
    free(valores);
    valores = NULL;
    return EXIT_FAILURE;
}

valores = temporal;
/* Solo valores puede usarse ahora para acceder al bloque. */
free(valores);
valores = NULL;
```

El patrón separa tres estados: antes del intento, durante el intento y después de aceptarlo. Si `realloc` mueve el bloque, cualquier puntero que apuntara a un elemento anterior puede quedar invalidado; deben conservarse índices o recalcularse las direcciones después de actualizar el puntero principal.

antes	→	**realloc**	→	**después**
`valores`	bloque A	`temporal`	bloque B	`valores = temporal`

Tabla 6: Redibujo conceptual: `realloc` puede devolver una dirección distinta.

Redibujo propio basado en Deitel, cap. 14, sección 14.11, pág. 494, y King, cap. 17, sección 17.3, págs. 420–422.

Un array dinámico necesita, como mínimo, un puntero al almacenamiento, una cantidad de elementos válidos y una capacidad reservada. La cantidad no debe superar la capacidad. Una política común es duplicar la capacidad cuando se agota, aunque la política concreta forma parte del diseño del contenedor.

```c
#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>

struct vector_int {
    int *datos;
    size_t cantidad;
    size_t capacidad;
};

static int reservar(struct vector_int *v, size_t capacidad_nueva)
{
    if (v == NULL || capacidad_nueva <= v->capacidad) {
        return 1;
    }
    if (capacidad_nueva > SIZE_MAX / sizeof *v->datos) {
        return 0;
    }

    int *temporal = realloc(
        v->datos, capacidad_nueva * sizeof *temporal);
    if (temporal == NULL) {
        return 0;
    }

    v->datos = temporal;
    v->capacidad = capacidad_nueva;
    return 1;
}

static void destruir(struct vector_int *v)
{
    if (v != NULL) {
        free(v->datos);
        v->datos = NULL;
        v->cantidad = 0;
        v->capacidad = 0;
    }
}
```

La función `reservar` no destruye el bloque original si el nuevo tamaño no puede obtenerse. `destruir` concentra el ownership del almacenamiento y deja explícito que, después de la destrucción, el vector está vacío. Separar la representación interna de las operaciones públicas también permite cambiar un array fijo por uno dinámico sin cambiar la idea de la interfaz; King muestra este criterio al presentar una pila con distintas representaciones.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 17, secciones 17.1 y 17.3–17.4, págs. 414–424, y cap. 19, sección 19.4, págs. 497–499; Deitel, cap. 14, sección 14.11, pág. 494.

2.11 Arrays fijos, VLA y arrays dinámicos

Los tres modelos no deben confundirse:

Modelo	Duración y tamaño	Liberación
Array fijo	El tamaño forma parte de la declaración; su duración depende del lugar de declaración.	Automática o estática; no se usa `free`.
VLA	El tamaño se conoce en tiempo de ejecución, pero queda fijo durante la vida del array y suele estar en la pila.	Automática; no se usa `free`.
Array dinámico	El bloque se reserva en el heap y puede redimensionarse con `realloc`.	El owner llama a `free` exactamente una vez.

```c
#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>

void comparar_modelos(size_t n)
{
    if (n == 0 || n > SIZE_MAX / sizeof(int)) {
        return;
    }

    int fijo[10];
    int vla[n];             /* VLA: C99; opcional en implementaciones C11 */
    int *dinamico = malloc(n * sizeof *dinamico);

    if (dinamico == NULL) {
        return;
    }

    fijo[0] = 0;
    vla[0] = 0;
    dinamico[0] = 0;
    free(dinamico);
}
```

El ejemplo supone `n > 0`; una función real debe validar esa precondición antes de declarar el VLA y antes de indexar cualquiera de los arrays. Un VLA no es una reserva que pueda conservarse después de salir de la función y tampoco se libera con `free`. Un array dinámico sí puede sobrevivir a la función que lo creó, siempre que su ownership pase de forma explícita al llamador.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 17, sección 17.1, pág. 414, y cap. 17, sección 17.3, págs. 420–422; nota editorial de adaptación para distinguir la duración automática del VLA de la reserva en heap.

2.12 Strings dinámicos

Una string de C es una sucesión de caracteres terminada por el carácter nulo `\0`. Si el texto contiene `n` caracteres visibles, la reserva necesita al menos `n + 1` bytes. El byte adicional no es opcional: las funciones de `<string.h>` lo utilizan para encontrar el final.

```c
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

static char *duplicar(const char *texto)
{
    if (texto == NULL) {
        return NULL;
    }

    size_t longitud = strlen(texto);
    if (longitud == SIZE_MAX) {
        return NULL;
    }

    char *copia = malloc((longitud + 1) * sizeof *copia);
    if (copia == NULL) {
        return NULL;
    }

    memcpy(copia, texto, longitud + 1);
    return copia;
}

char *mensaje = duplicar("hola");
if (mensaje == NULL) {
    return EXIT_FAILURE;
}

puts(mensaje);
free(mensaje);
mensaje = NULL;
```

La función devuelve un puntero a un objeto creado dentro de ella, pero ese objeto vive en el heap y no desaparece al retornar. El llamador recibe el ownership del resultado y debe liberarlo. Esto es diferente de devolver la dirección de un array local, que dejaría un puntero colgante tan pronto como terminara la función.

Para leer texto no se usa `gets`: esa función no recibe el tamaño del buffer y puede escribir fuera de sus límites. `fgets` permite establecer una capacidad máxima; después hay que decidir cómo tratar el salto de línea y si el texto fue truncado.

Cuando se guardan varias strings, un array de `char *` contiene punteros, no los caracteres mismos. Si cada string fue reservada por separado, cada elemento posee un bloque que debe liberarse por separado:

```
char *items[2] = {duplicar("uno"), duplicar("dos")};

for (size_t i = 0; i < 2; i++) {
    free(items[i]);
    items[i] = NULL;
}
```

En código de producción hay que ocuparse también del fallo parcial: si la segunda duplicación falla, se libera la primera antes de devolver el error. El principio es el mismo: cada bloque tiene un owner y se libera exactamente una vez.

0	1	2	3	4
h	o	l	a	\0

Tabla 7: Redibujo conceptual de una string dinámica y su terminador.

Redibujo propio basado en King, cap. 17, sección 17.2, págs. 416–420.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 17, sección 17.2, págs. 416–420; Deitel, cap. 7, secciones 7.8–7.10, págs. 250–258.

2.13 Puente hacia nodos y listas enlazadas

Un nodo enlazado combina los dos conceptos centrales de este capítulo: una estructura con datos y un puntero que guarda la dirección del siguiente nodo. Un nodo puede reservarse así:

```
struct nodo *crear_nodo(int dato)
{
    struct nodo *nuevo = malloc(sizeof *nuevo);
    if (nuevo == NULL) {
        return NULL;
    }

    nuevo->dato = dato;
    nuevo->siguiente = NULL;
    return nuevo; /* ownership pasa al llamador */
}
```

Si el llamador obtiene un nodo, pasa a ser responsable de destruirlo cuando ya no lo necesite. Para una lista, esa responsabilidad se extiende a todos los nodos: antes de liberar un nodo hay que conservar el puntero al siguiente nodo, porque después de `free` ya no es válido leer `actual->siguiente`.

```
void destruir_nodos(struct nodo *actual)
{
    while (actual != NULL) {
        struct nodo *siguiente = actual->siguiente;
        free(actual);
        actual = siguiente;
    }
}
```

Esta función muestra el ciclo de destrucción, no las operaciones completas del ADT. El Bloque 3 desarrollará las interfaces de insertar, recorrer, buscar y eliminar; allí se comparará el enfoque que retorna el nuevo `head` con la alternativa de pasar un puntero doble.

> **Procedencia:** King, **C Programming: A Modern Approach**, 2.ª ed., cap. 17, secciones 17.5–17.6, págs. 424–439; Deitel, cap. 12, secciones 12.2–12.4, págs. 423–424 y siguientes. La función de destrucción está reescrita para conservar el enlace antes de liberar cada nodo.

CAPÍTULO 3

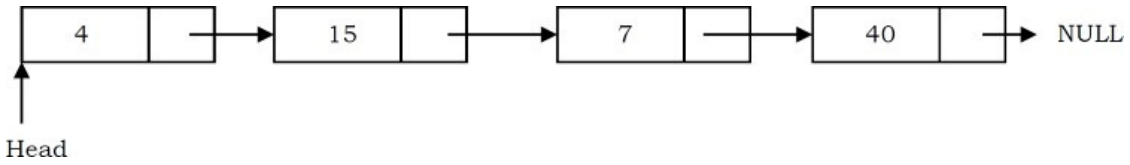
Listas enlazadas

Extraído del capítulo 3 de **Data Structures and Algorithms Made Easy**, págs. 74–88

3.1 ¿Qué es una lista enlazada?

Una lista enlazada es una estructura de datos que se utiliza para almacenar colecciones de datos. Una lista enlazada tiene las siguientes propiedades:

- Los elementos sucesivos están conectados mediante punteros.
- El último elemento apunta a `NULL`.
- Puede crecer o reducirse durante la ejecución de un programa.
- Puede alcanzar la longitud que sea necesaria, hasta que se agote la memoria del sistema.
- No desperdicia espacio de memoria, aunque necesita algo de memoria adicional para los punteros. Asigna memoria a medida que la lista crece.



3.2 ADT de listas enlazadas

Las siguientes operaciones convierten a las listas enlazadas en un ADT:

Operaciones principales de las listas enlazadas

- **Insertar:** inserta un elemento en la lista.
- **Eliminar:** elimina y devuelve el elemento situado en la posición especificada de la lista.

Operaciones auxiliares de las listas enlazadas

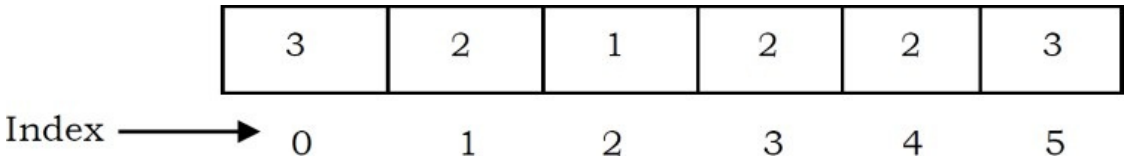
- **Eliminar la lista:** elimina todos los elementos de la lista (descarta la lista).
- **Contar:** devuelve el número de elementos de la lista.
- **Encontrar el n-ésimo nodo desde el final de la lista.**

3.3 ¿Por qué listas enlazadas?

Existen muchas otras estructuras de datos que hacen lo mismo que las listas enlazadas. Antes de analizar las listas enlazadas, es importante comprender la diferencia entre las listas enlazadas y los arrays. Ambas estructuras se utilizan para almacenar colecciones de datos y, como se usan con el mismo propósito, necesitamos diferenciar sus usos. Es decir, en qué casos son adecuados los arrays y en qué casos son adecuadas las listas enlazadas.

3.4 Descripción general de los arrays

Se asigna un bloque de memoria para todo el array, que contiene sus elementos. Los elementos del array pueden accederse en tiempo constante utilizando como subíndice el índice del elemento particular.



¿Por qué el acceso a los elementos de un array toma tiempo constante?

Para acceder a un elemento de un array, se calcula la dirección del elemento como un desplazamiento respecto de la dirección base del array. Se necesita una multiplicación para calcular qué debe sumarse a la dirección base para obtener la dirección de memoria del elemento. Primero se calcula el tamaño de un elemento de ese tipo de dato y luego se lo multiplica por el índice del elemento para obtener el valor que debe sumarse a la dirección base.

Este proceso requiere una multiplicación y una suma. Como estas dos operaciones toman tiempo constante, podemos decir que el acceso a un array puede realizarse en tiempo constante.

Ventajas de los arrays

- Son simples y fáciles de usar.
- El acceso a los elementos es más rápido (acceso constante).

Desventajas de los arrays

- Preasignan toda la memoria necesaria y desperdician espacio para los índices del array que están vacíos.
- **Tamaño fijo:** el tamaño del array es estático (se especifica antes de usarlo).
- **Asignación de un bloque:** para asignar el array al comienzo, a veces puede no ser posible obtener memoria para el array completo, si su tamaño es grande.
- **Inserción compleja basada en la posición:** para insertar un elemento en una posición dada, puede ser necesario desplazar los elementos existentes. Esto crea una posición para insertar el nuevo elemento en el lugar deseado. Si la posición en la que queremos agregar un elemento está al principio, la operación de desplazamiento es más costosa.

Arrays dinámicos

Un array dinámico (también llamado array expandible, array redimensionable, tabla dinámica o lista de arrays) es una estructura de datos de lista, de tamaño variable y acceso aleatorio, que permite agregar o quitar elementos.

Una forma simple de implementar arrays dinámicos consiste en comenzar con un array de tamaño fijo. En cuanto ese array se llena, se crea un array nuevo con el doble del tamaño del array original. De manera similar, se reduce a la mitad el tamaño del array si los elementos que contiene son menos de la mitad.

La reserva, el crecimiento y la liberación del almacenamiento dinámico ya se explicaron en el Capítulo 2 — Punteros y memoria dinámica en C. Aquí se conserva únicamente la comparación del original; no se repite su implementación. En la cursada, el array dinámico se denomina **vector**.

Nota: el libro indica que la implementación de arrays dinámicos se verá en los capítulos sobre pilas, colas y hashing.

Ventajas de las listas enlazadas

Las listas enlazadas tienen ventajas y desventajas. La ventaja de las listas enlazadas es que pueden expandirse en tiempo constante. Para crear un array, debemos asignar memoria para una cantidad determinada de elementos. Para agregar más elementos al array cuando está lleno, debemos crear un array nuevo y copiar el array anterior en el nuevo. Esto puede tomar mucho tiempo.

Podemos evitarlo asignando mucho espacio inicialmente, pero entonces podríamos asignar más espacio del necesario y desperdiciarlo. Con una lista enlazada, podemos comenzar con espacio para un único elemento asignado y agregar elementos nuevos fácilmente, sin necesidad de copiar ni reasignar.

Problemas de las listas enlazadas (desventajas)

Las listas enlazadas presentan varios problemas. La principal desventaja es el tiempo de acceso a los elementos individuales. Un array permite acceso aleatorio, lo que significa que toma $O(1)$ acceder a cualquier elemento del array. En el peor caso, las listas enlazadas toman $O(n)$ para acceder a un elemento de la lista. Otra ventaja de los arrays respecto del tiempo de acceso es la localidad espacial en memoria. Los arrays se definen como bloques contiguos de memoria, por lo que cada elemento está físicamente cerca de sus vecinos. Esto aprovecha considerablemente los métodos modernos de caché de la CPU.

Aunque la asignación dinámica de almacenamiento es una gran ventaja, el costo adicional de almacenar y recuperar datos puede marcar una diferencia importante. A veces las listas enlazadas son difíciles de manipular. Si se elimina el último elemento, el anteúltimo debe cambiar su puntero para contener una referencia `NULL`. Para ello, es necesario recorrer la lista hasta encontrar el anteúltimo enlace y establecer su puntero en una referencia `NULL`.

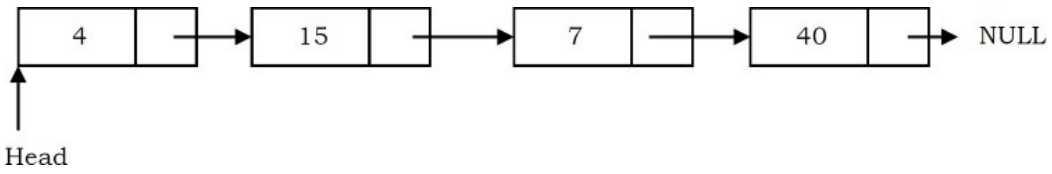
Por último, las listas enlazadas desperdician memoria debido a las referencias adicionales.

3.5 Comparación de listas enlazadas con arrays y arrays dinámicos

Parámetro	Lista enlazada	Array	Array dinámico (**vector**)
Indexación	$O(n)$	$O(1)$	$O(1)$
Inserción/eliminación al principio	$O(1)$	$O(n)$, si el array no está lleno (para desplazar los elementos)	$O(n)$
Inserción al final	$O(n)$	$O(1)$, si el array no está lleno	$O(1)$, si no está lleno; $O(n)$, si está lleno
Eliminación al final	$O(n)$	$O(1)$	$O(n)$
Inserción en el medio	$O(n)$	$O(n)$, si el array no está lleno (para desplazar los elementos)	$O(n)$
Eliminación en el medio	$O(n)$	$O(n)$, si el array no está lleno (para desplazar los elementos)	$O(n)$
Espacio desperdiciado	$O(n)$ (para los punteros)	0	$O(n)$

3.6 Listas simplemente enlazadas

En general, "lista enlazada" significa una lista simplemente enlazada. Esta lista consta de varios nodos, donde cada nodo tiene un puntero `next` al elemento siguiente. El enlace del último nodo de la lista es `NULL`, lo que indica el final de la lista.



La siguiente es una declaración de tipo para una lista enlazada de enteros:

```
struct ListNode {
    int data;
    struct ListNode *next;
};
```

El Capítulo 2 ya presentó punteros, estructuras autorreferenciadas y la administración de memoria dinámica. Aquí se usan esos conceptos para manipular los enlaces; no se vuelve a explicar su funcionamiento.

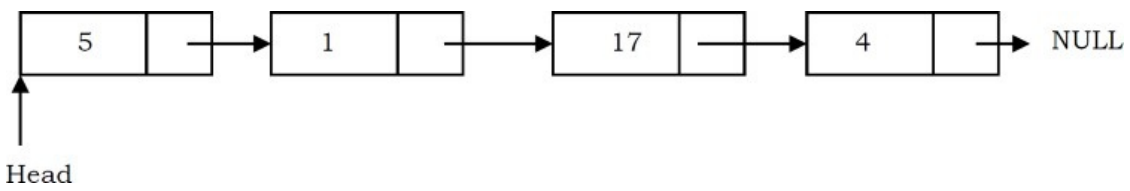
Operaciones básicas sobre una lista

- Recorrer la lista.
- Insertar un elemento en la lista.
- Eliminar un elemento de la lista.

Recorrido de la lista enlazada

Supongamos que `head` apunta al primer nodo de la lista. Para recorrer la lista hacemos lo siguiente:

- Seguimos los punteros.
- Mostramos el contenido de los nodos (o contamos los nodos) a medida que los recorremos.
- Nos detenemos cuando el puntero `next` apunta a `NULL`.



La función `ListLength()` recibe como entrada una lista enlazada y cuenta el número de nodos que contiene. La función siguiente puede utilizarse para imprimir los datos de la lista junto con una función de impresión adicional.

```
int ListLength(const struct ListNode *head)
{
    const struct ListNode *current = head;
    int count = 0;

    while (current != NULL) {
        count++;
        current = current->next;
    }

    return count;
}
```

Complejidad temporal: $O(n)$, para recorrer una lista de tamaño n.

Complejidad espacial: $O(1)$, para crear una variable temporal.

Inserción en una lista simplemente enlazada

La inserción en una lista simplemente enlazada tiene tres casos:

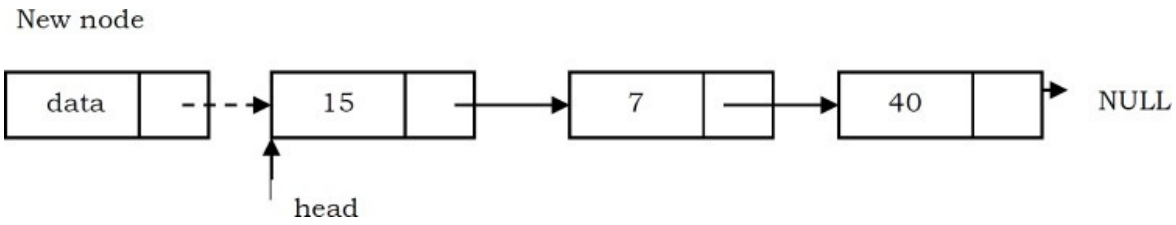
- Insertar un nodo nuevo antes de `head` (al principio).
- Insertar un nodo nuevo después de `tail` (al final de la lista).
- Insertar un nodo nuevo en el medio de la lista (una ubicación aleatoria).

Nota: para insertar un elemento en la lista enlazada en una posición p, supongamos que, después de insertar el elemento, la posición de este nuevo nodo es p. Las posiciones de los ejemplos son 1--based: la primera posición es 1.

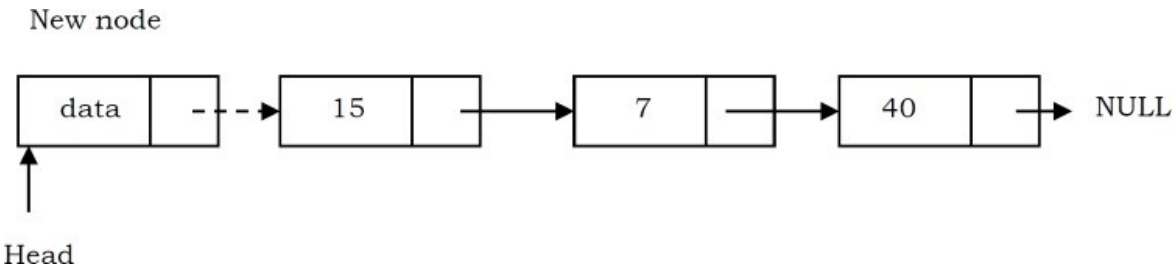
Insertar un nodo al principio de una lista simplemente enlazada

En este caso, se inserta un nodo nuevo antes del nodo que actualmente es `head`. Solo es necesario modificar un puntero `next` (el puntero `next` del nodo nuevo), y puede hacerse en dos pasos:

- Actualizar el puntero `next` del nodo nuevo para que apunte al `head` actual.



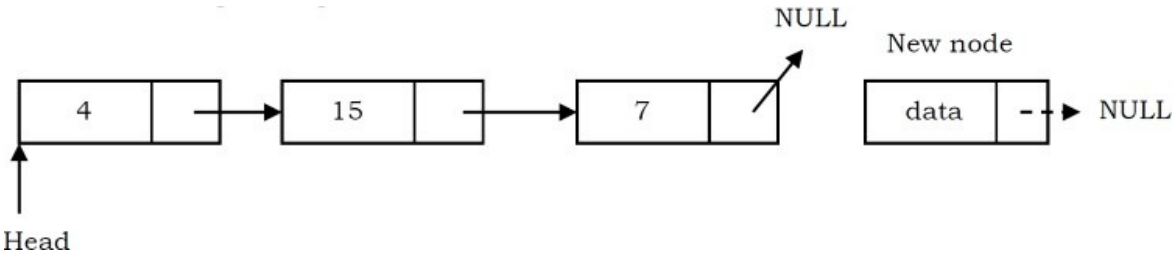
- Actualizar el puntero `head` para que apunte al nodo nuevo.



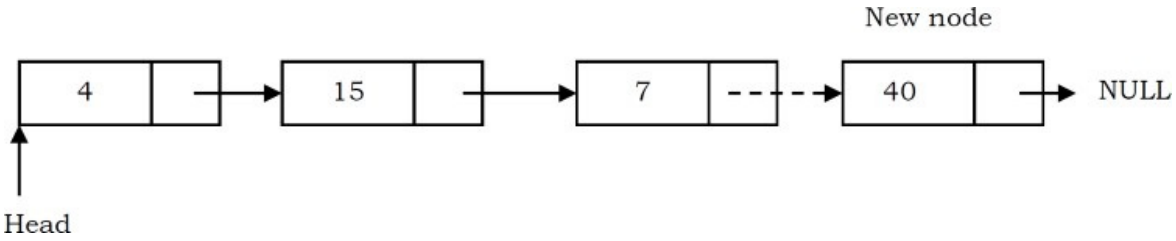
Insertar un nodo al final de una lista simplemente enlazada

En este caso, debemos modificar dos punteros `next` (el puntero `next` del último nodo y el puntero `next` del nodo nuevo).

- El puntero `next` del nodo nuevo apunta a `NULL`.



- El puntero `next` del último nodo apunta al nodo nuevo.

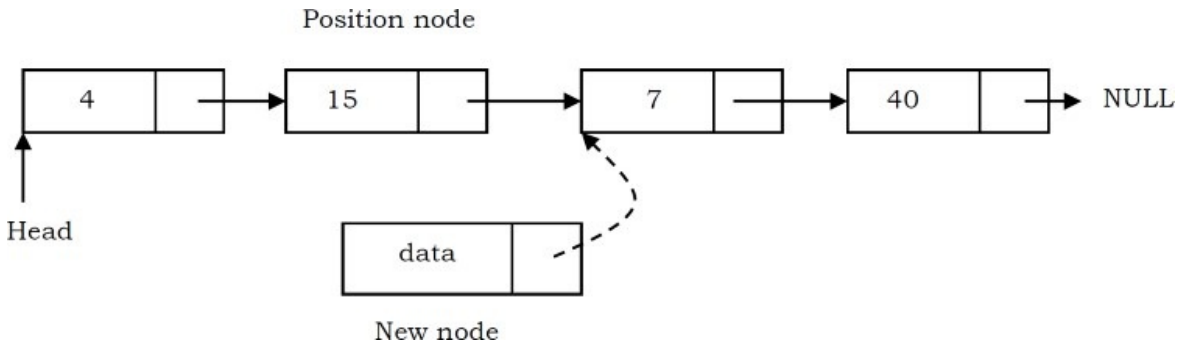


Insertar un nodo en el medio de una lista simplemente enlazada

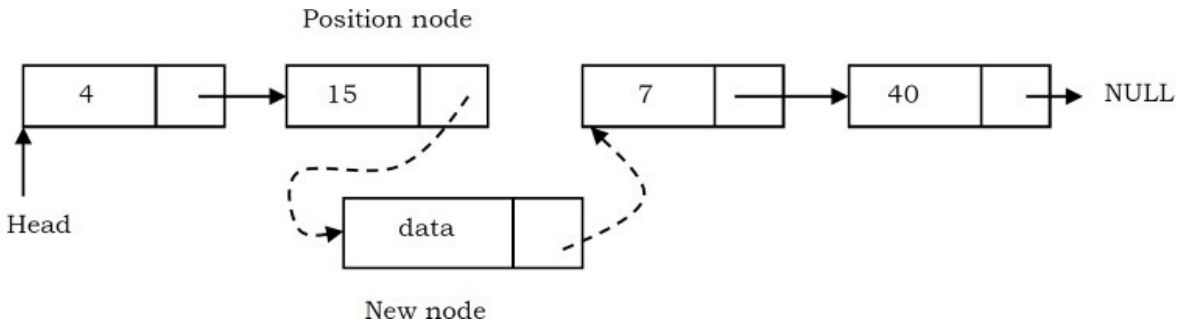
Supongamos que se nos da una posición en la que queremos insertar el nodo nuevo. En este caso también debemos modificar dos punteros `next`.

- Si queremos agregar un elemento en la posición 3, nos detenemos en la posición 2. Es decir, recorremos dos nodos e insertamos el nodo nuevo. Para simplificar, supongamos que

el segundo nodo se llama nodo de posición. El nodo nuevo apunta al siguiente nodo de la posición en la que queremos agregarlo.



- El puntero `next` del nodo de posición ahora apunta al nodo nuevo.



Para las operaciones que modifican la lista, la interfaz de la cursada recibe `head` por valor y retorna el nuevo `head`. Así, la función puede devolver un `head` distinto cuando la inserción ocurre al principio.

```c
#include <stdlib.h>

struct ListNode *InsertInLinkedList(
    struct ListNode *head, int data, int position)
{
    if (position < 1) {
        return head;
    }

    struct ListNode *newNode = malloc(sizeof *newNode);
    if (newNode == NULL) {
        return head;
    }
    newNode->data = data;

    if (position == 1) {
        newNode->next = head;
        return newNode;
    }

    struct ListNode *positionNode = head;
    for (int k = 1;
         positionNode != NULL && k < position - 1;
         k++) {
        positionNode = positionNode->next;
    }

    if (positionNode == NULL) {
        free(newNode);
        return head;
    }

    newNode->next = positionNode->next;
    positionNode->next = newNode;
    return head;
}

/* Ejemplo de uso: la asignación conserva el head actualizado. */
head = InsertInLinkedList(head, data, position);
```

La función acepta insertar al final cuando `position` es una posición posterior al último nodo. Si la posición no es válida o falla la reserva, conserva el `head` original. La reserva y la liberación se muestran aquí solo para completar la operación; su ciclo de vida se explicó en el Capítulo 2.

Nota: podemos implementar por separado las tres variantes de la operación de inserción.

Complejidad temporal: $O(n)$, ya que, en el peor caso, podríamos necesitar insertar el nodo al final de la lista.

Complejidad espacial: $O(1)$, sin contar el nodo nuevo; se crean variables temporales auxiliares.

Alternativa: pasar un puntero doble al `head`

El libro presenta el código de esta operación recibiendo `struct ListNode **head`. En ese enfoque, el llamador pasa la dirección de su variable `head` y la función puede escribir sobre esa variable mediante `*head`. La idea de enlazar nodos es la misma; cambia la interfaz.

```
void InsertInLinkedListWithDoublePointer(
    struct ListNode **head, int data, int position)
{
    if (head != NULL) {
        *head = InsertInLinkedList(*head, data, position);
    }
}

/* Uso alternativo: */
InsertInLinkedListWithDoublePointer(&head, data, position);
```

En la cursada se prefiere la primera forma: `head = InsertInLinkedList(head, data, position)`. El doble puntero queda como alternativa para leer el ejemplo del libro, especialmente el de la pág. 87.

Eliminación en una lista simplemente enlazada

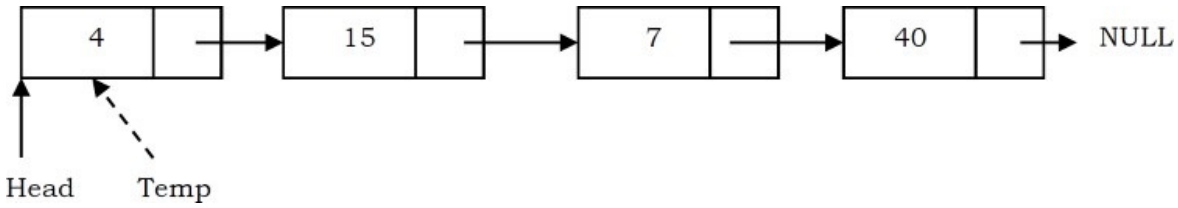
Al igual que en la inserción, aquí también hay tres casos:

- Eliminar el primer nodo.
- Eliminar el último nodo.
- Eliminar un nodo intermedio.

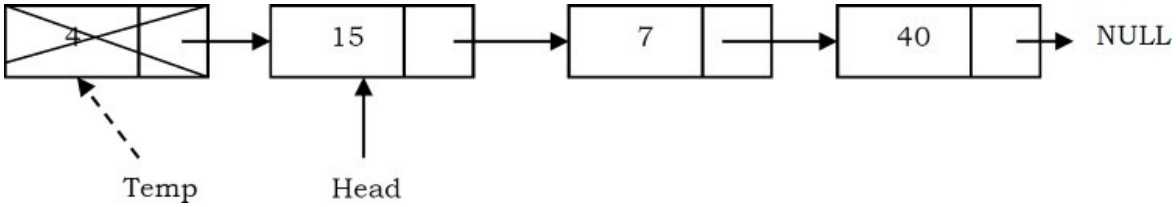
Eliminar el primer nodo de una lista simplemente enlazada

Se elimina el primer nodo (el nodo que actualmente es `head`). Puede hacerse en dos pasos:

- Crear un nodo temporal que apunte al mismo nodo que `head`.



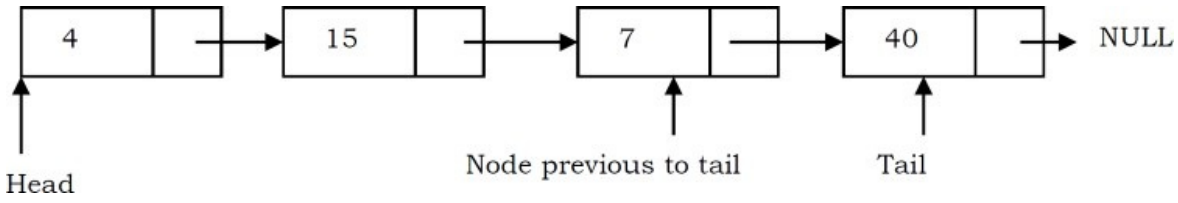
- Ahora, mover el puntero de `head` al nodo siguiente y descartar el nodo temporal.



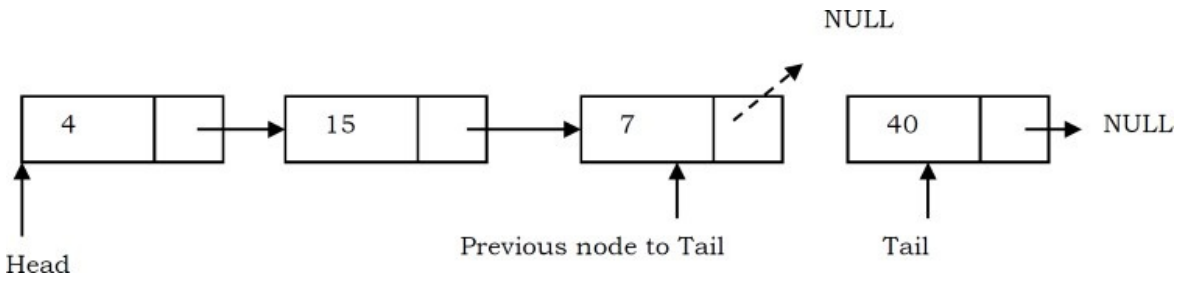
Eliminar el último nodo de una lista simplemente enlazada

En este caso se elimina el último nodo. Esta operación es algo más complicada que eliminar el primero, porque el algoritmo debe encontrar un nodo que sea anterior al nodo final. Puede hacerse en tres pasos:

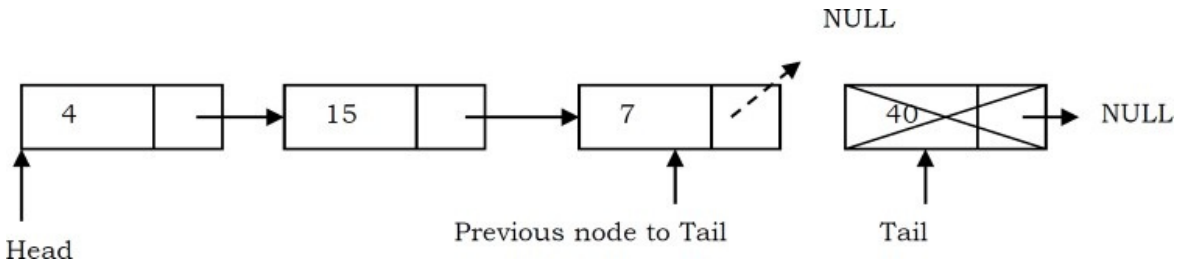
- Recorrer la lista y mantener también la dirección del nodo anterior. Cuando lleguemos al final de la lista, tendremos dos punteros: uno que apunta al nodo final y otro que apunta al nodo anterior al nodo final.



- Actualizar el puntero `next` del nodo anterior con `NULL`.



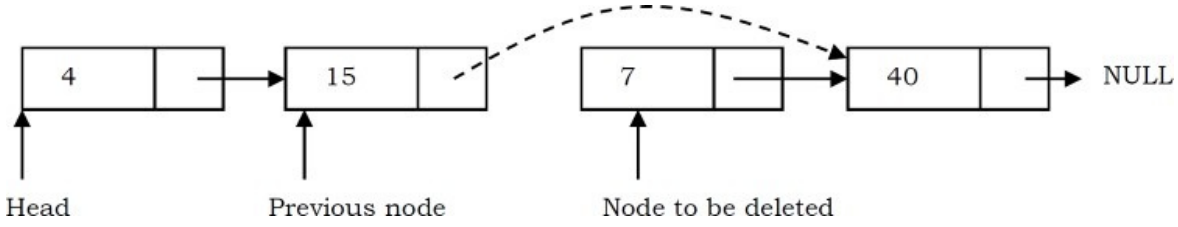
- Descartar el nodo final.



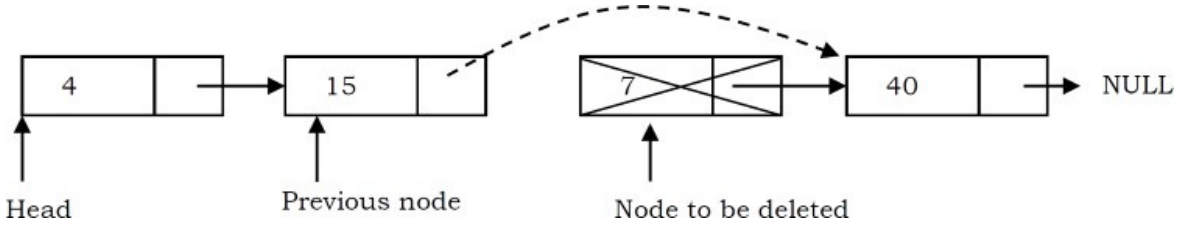
Eliminar un nodo intermedio de una lista simplemente enlazada

En este caso, el nodo que se eliminará siempre está ubicado entre otros dos nodos. Los enlaces de `head` y `tail` no se actualizan. Esta eliminación puede hacerse en dos pasos:

- Al igual que en el caso anterior, mantener el nodo anterior durante el recorrido. Una vez que encontremos el nodo que se eliminará, cambiar el puntero `next` del nodo anterior por el puntero `next` del nodo que se eliminará.



- Descartar el nodo actual que se eliminará.




```c
struct ListNode *DeleteNodeFromLinkedList(
    struct ListNode *head, int position)
{
    if (head == NULL || position < 1) {
        return head;
    }

    if (position == 1) {
        struct ListNode *temporary = head;
        head = head->next;
        free(temporary);
        return head;
    }

    struct ListNode *previous = head;
    for (int k = 1;
         previous->next != NULL && k < position - 1;
         k++) {
        previous = previous->next;
    }

    if (previous->next == NULL) {
        return head;
    }

    struct ListNode *current = previous->next;
    previous->next = current->next;
    free(current);
    return head;
}

/* Ejemplo de uso: */
head = DeleteNodeFromLinkedList(head, position);
```

La función devuelve el mismo `head` cuando la lista está vacía o la posición no existe. Cuando se elimina la primera posición devuelve el nodo siguiente. Esta adaptación conserva la operación del libro y evita modificar la variable del llamador mediante un puntero doble.

Complejidad temporal: $O(n)$. En el peor caso, podríamos necesitar eliminar el nodo al final de la lista.

Complejidad espacial: $O(1)$, para una variable temporal.

Eliminar una lista simplemente enlazada

Esta operación almacena el nodo actual en una variable temporal y libera el nodo actual. Después de liberar el nodo actual, se avanza al nodo siguiente con una variable temporal y se repite este proceso para todos los nodos.

```c
struct ListNode *DeleteLinkedList(struct ListNode *head)
{
    struct ListNode *iterator = head;

    while (iterator != NULL) {
        struct ListNode *next = iterator->next;
        free(iterator);
        iterator = next;
    }

    return NULL;
}

/* Ejemplo de uso: */
head = DeleteLinkedList(head);
```

Complejidad temporal: $O(n)$, para recorrer la lista completa de tamaño n.

Complejidad espacial: $O(1)$, para crear una variable temporal.

> **Procedencia:** Narasimha Karumanchi, **Data Structures and Algorithms Made Easy**, capítulo 3, secciones 3.1–3.6, págs. impresas 74–88. Fuente local: `source/Data-Structures-and-Algorithms-Narasimha-Karumanchi.pdf`. Texto contrastado mediante `pdftotext -layout` y extracción estructurada con `pdftohtml -xml`. Los diagramas incluidos son recortes de las figuras de las págs. 74, 77, 79, 81–86 y 88; el código fue reescrito en Typst y adaptado a la interfaz de la cursada (`head` → nuevo `head`).

CAPÍTULO 4

Algoritmos y análisis

Extraído del capítulo 1 de **Data Structures and Algorithms Made Easy**, págs. 19–25

1.5 ¿Qué es un algoritmo?

Consideremos el problema de preparar una tortilla. Para preparar una tortilla, seguimos los pasos que se indican a continuación:

1) Obtener la sartén.
2) Obtener el aceite.
    1) ¿Tenemos aceite?
        1) Si la respuesta es sí, ponerlo en la sartén.
        2) Si la respuesta es no, ¿queremos comprar aceite?
            1) Si la respuesta es sí, salir a comprarlo.
            2) Si la respuesta es no, podemos terminar.
3) Encender la cocina, etc.

Lo que estamos haciendo es proporcionar, para un problema dado (preparar una tortilla), un procedimiento paso a paso para resolverlo. La definición formal de un algoritmo puede expresarse de la siguiente manera:

> **Un algoritmo** son instrucciones paso a paso y no ambiguas para resolver un problema dado.

En el estudio tradicional de los algoritmos hay dos criterios principales para juzgar sus méritos: la **corrección** (¿el algoritmo da una solución al problema en un número finito de pasos?) y la **eficiencia** (¿cuántos recursos, en términos de memoria y tiempo, necesita para ejecutarse?).

Nota: no es necesario demostrar cada paso del algoritmo.

1.6 ¿Por qué analizar los algoritmos?

Para ir de la ciudad **A** a la ciudad **B**, puede haber muchas formas de hacerlo: en avión, en autobús, en tren o también en bicicleta. Dependiendo de la disponibilidad y la conveniencia, elegimos la que más nos conviene. De manera similar, en ciencias de la computación hay múltiples algoritmos disponibles para resolver el mismo problema (por ejemplo, un problema de ordenamiento tiene muchos algoritmos, como ordenamiento por inserción, ordenamiento por selección, quicksort y muchos más). El análisis de algoritmos nos ayuda a determinar qué algoritmo es más eficiente en términos del tiempo y el espacio consumidos.

1.7 Objetivo del análisis de algoritmos

El objetivo del análisis de algoritmos es comparar algoritmos (o soluciones), principalmente en términos del tiempo de ejecución, pero también en términos de otros factores (por ejemplo, memoria, esfuerzo del desarrollador, etc.).

1.8 ¿Qué es el análisis del tiempo de ejecución?

Es el proceso de determinar cómo aumenta el tiempo de procesamiento a medida que aumenta el tamaño del problema (tamaño de la entrada). El tamaño de la entrada es la cantidad de elementos de la entrada y, según el tipo de problema, la entrada puede ser de distintos tipos. Estos son los tipos de entrada más comunes:

- Tamaño de un array.
- Grado de un polinomio.
- Cantidad de elementos de una matriz.
- Cantidad de bits en la representación binaria de la entrada.
- Vértices y aristas de un grafo.

1.9 Cómo comparar algoritmos

Para comparar algoritmos, definamos algunas medidas objetivas:

¿Tiempos de ejecución? No es una buena medida, ya que los tiempos de ejecución son específicos de una computadora particular.

¿Cantidad de instrucciones ejecutadas? No es una buena medida, ya que la cantidad de instrucciones varía según el lenguaje de programación y el estilo de cada programador.

¿Solución ideal? Supongamos que expresamos el tiempo de ejecución de un algoritmo dado como una función del tamaño de entrada n (es decir, $f(n)$) y comparamos estas distintas funciones correspondientes a los tiempos de ejecución. Este tipo de comparación es independiente del tiempo de la máquina, del estilo de programación, etc.

1.10 ¿Qué es la tasa de crecimiento?

La tasa a la que aumenta el tiempo de ejecución en función de la entrada se denomina **tasa de crecimiento**. Supongamos que usted va a una tienda a comprar un automóvil y una bicicleta. Si su

amigo lo ve allí y le pregunta qué está comprando, por lo general usted responde que está comprando un automóvil. Esto se debe a que el costo del automóvil es alto en comparación con el costo de la bicicleta (se aproxima el costo de la bicicleta al costo del automóvil).

$$\text{Costo total} = \text{costo del automóvil} + \text{costo de la bicicleta}$$

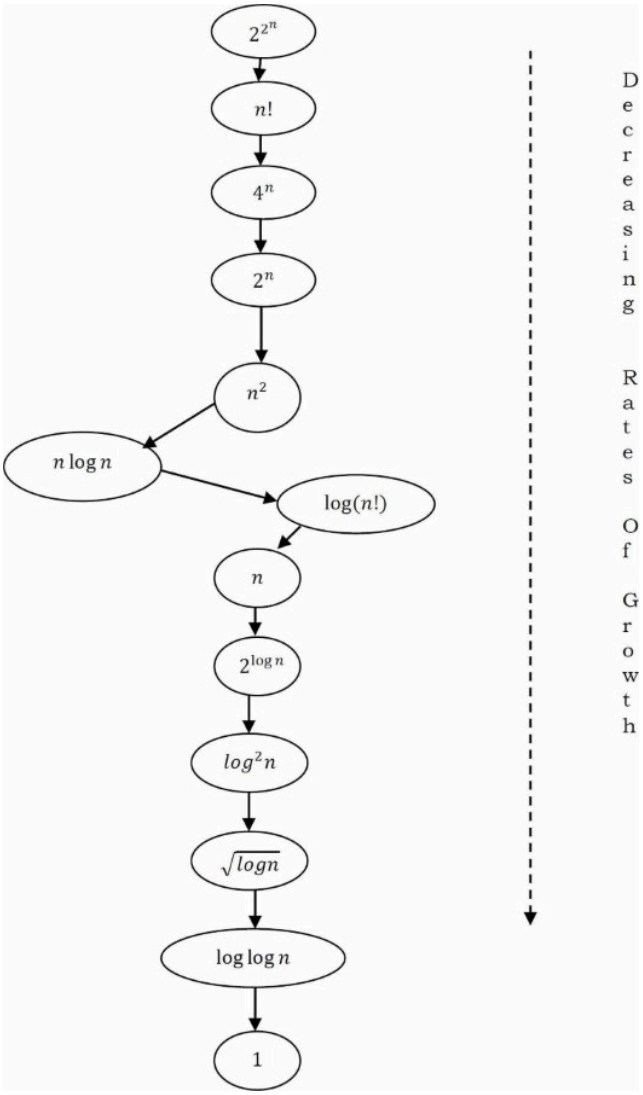
$$\text{Costo total} \approx \text{costo del automóvil} \quad (\text{aproximación})$$

Para el ejemplo anterior, podemos representar el costo del automóvil y el costo de la bicicleta en términos de funciones y, para una función dada, ignorar los términos de orden inferior, que son relativamente insignificantes (para un valor grande del tamaño de entrada n). Por ejemplo, en el caso siguiente, n^4, $2n^2$, $100n$ y 500 son los costos individuales de alguna función y se aproximan a n^4, ya que n^4 es la tasa de crecimiento más alta.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 Tasas de crecimiento de uso común

El diagrama siguiente muestra la relación entre las distintas tasas de crecimiento.



A continuación se presenta la lista de tasas de crecimiento que aparecerán en los capítulos siguientes.

Complejidad temporal	Nombre	Ejemplo
1	Constante	Agregar un elemento al principio de una lista enlazada
$\log n$	Logarítmica	Encontrar un elemento en un array ordenado
n	Lineal	Encontrar un elemento en un array no ordenado
$n \log n$	Lineal logarítmica	Ordenar n elementos mediante «divide y vencerás» — mergesort
n^2	Cuadrática	Camino más corto entre dos nodos de un grafo
n^3	Cúbica	Multiplicación de matrices
2^n	Exponencial	El problema de las Torres de Hanoi

1.12 Tipos de análisis

Para analizar el algoritmo dado, necesitamos saber con qué entradas el algoritmo tarda menos (funciona mejor) y con qué entradas tarda más. Ya hemos visto que un algoritmo puede representarse mediante una expresión. Esto significa que representamos el algoritmo con varias expresiones: una para el caso en el que tarda menos y otra para el caso en el que tarda más.

En general, el primer caso se denomina **mejor caso** y el segundo se denomina **peor caso** para el algoritmo. Para analizar un algoritmo necesitamos algún tipo de sintaxis, y esta constituye la base del análisis/notación asintótico. Hay tres tipos de análisis:

- **Peor caso**
  - Define la entrada para la cual el algoritmo tarda mucho (el tiempo más lento para completar la operación).
  - Es la entrada para la cual el algoritmo se ejecuta más lentamente.
- **Mejor caso**
  - Define la entrada para la cual el algoritmo tarda menos (el tiempo más rápido para completar la operación).
  - Es la entrada para la cual el algoritmo se ejecuta más rápidamente.
- **Caso promedio**
  - Proporciona una predicción sobre el tiempo de ejecución del algoritmo.
  - Ejecutar el algoritmo muchas veces, usando muchas entradas diferentes provenientes de alguna distribución que genere esas entradas; calcular el tiempo total de ejecución (sumando los tiempos individuales) y dividirlo por la cantidad de pruebas.
  - Supone que la entrada es aleatoria.

Cota inferior $\leq$ tiempo promedio $\leq$ cota superior

> **Nota - EDA.** En el curso nos enfocaremos principalmente en la notación del peor caso.

Para un algoritmo dado, podemos representar los casos mejor, peor y promedio mediante expresiones. Por ejemplo, sea $f(n)$ la función que representa el algoritmo dado.

$$f(n) = n^2 + 500$$

para el peor caso

$$f(n) = n + 100$$

para el mejor caso

De manera similar, se representa el caso promedio. La expresión define las entradas para las cuales el algoritmo tarda el tiempo de ejecución (o utiliza la memoria) promedio.

1.13 Notación asintótica

Una vez que tenemos las expresiones para los casos mejor, promedio y peor, para los tres casos necesitamos identificar las cotas superior e inferior. Para representar estas cotas superior e inferior

necesitamos algún tipo de sintaxis, y ese es el tema de la discusión siguiente. Supongamos que el algoritmo dado está representado mediante la función $f(n)$.

1.14 Notación Big-O [función de cota superior]

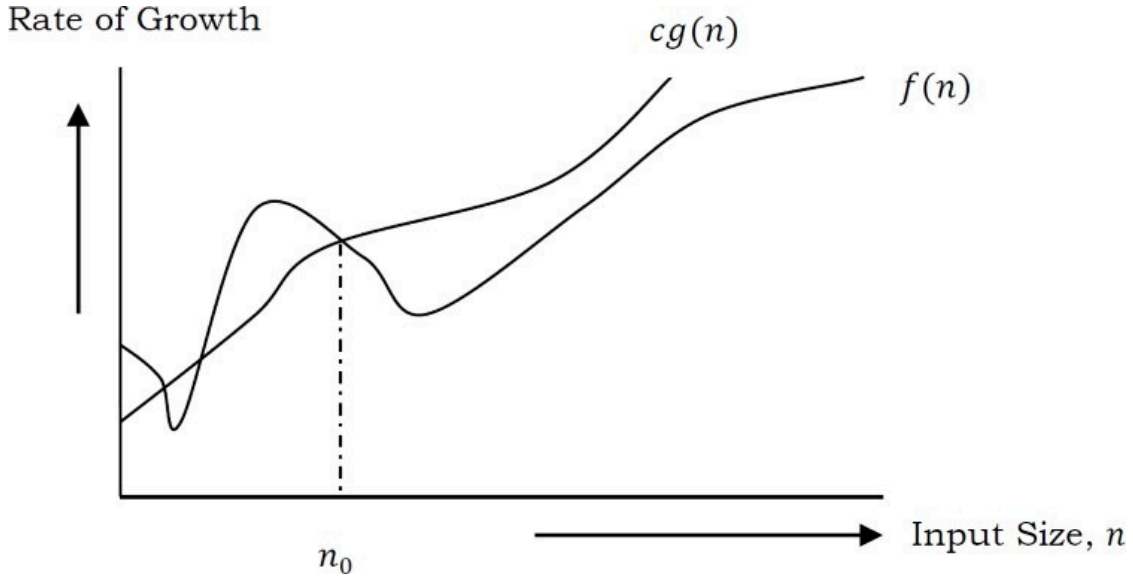
Esta notación proporciona la **cota superior ajustada** de la función dada. Generalmente se representa como $f(n) = O(g(n))$. Esto significa que, para valores grandes de n, la cota superior de $f(n)$ es $g(n)$. Por ejemplo, si $f(n) = n^4 + 100n^2 + 10n + 50$ es el algoritmo dado, entonces n^4 es $g(n)$. Esto significa que $g(n)$ proporciona la tasa máxima de crecimiento de $f(n)$ para valores grandes de n.

Veamos la notación O con un poco más de detalle. La notación O se define como:

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0$$
$$\text{tal que}$$
$$0 \leq f(n) \leq cg(n)$$
$$\text{para todo}$$
$$n > n_0\}$$

$g(n)$ es una cota superior asintótica ajustada para $f(n)$. Nuestro objetivo es dar la tasa de crecimiento más pequeña $g(n)$ que sea mayor o igual que la tasa de crecimiento del algoritmo dado $f(n)$.

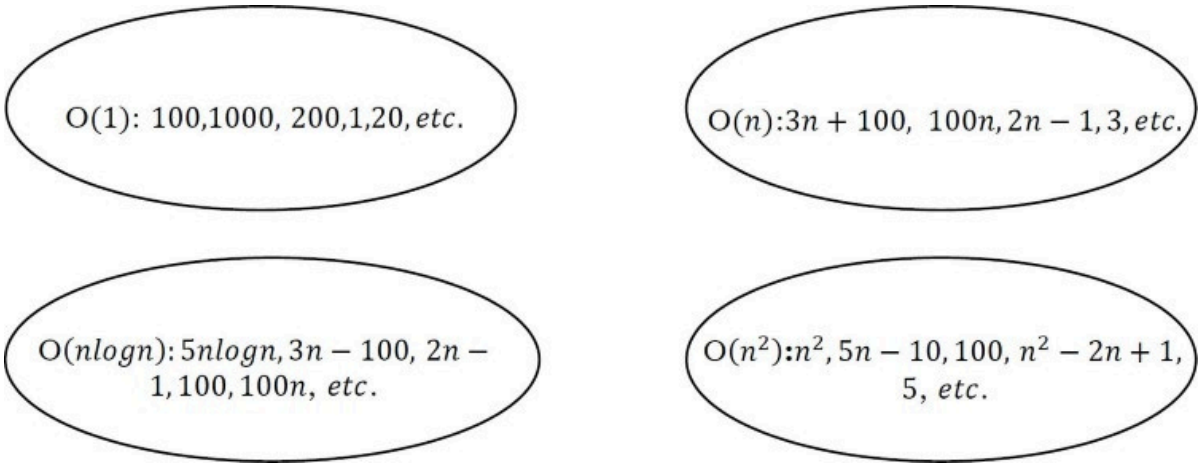
Generalmente descartamos los valores pequeños de n. Esto significa que la tasa de crecimiento para valores pequeños de n no es importante. En la figura, n_0 es el punto a partir del cual debemos considerar la tasa de crecimiento de un algoritmo dado. Por debajo de n_0, la tasa de crecimiento podría ser diferente. n_0 se denomina umbral de la función dada.



Visualización de Big-O

$O(g(n))$ es el conjunto de funciones con un orden de crecimiento menor o igual que el de $g(n)$. Por ejemplo, $O(n^2)$ incluye $O(1)$, $O(n)$, $O(n \log n)$, etc.

Nota: analice los algoritmos solamente para valores grandes de n. Esto significa que, por debajo de n_0, no nos importa la tasa de crecimiento.



Ejemplos de Big-O

Ejemplo 1. Encuentre la cota superior para $f(n) = 3n + 8$.

Solución: $3n + 8 \leq 4n$, para todo $n \geq 8$.

$$\therefore 3n + 8 = O(n) \quad \text{con } c = 4 \text{ y } n_0 = 8$$

CAPÍTULO 5

Recursión y backtracking

Extraído del capítulo 2 de **Data Structures and Algorithms Made Easy**, págs. 62–67

2.1 Introducción

En este capítulo veremos uno de los temas importantes, la **recursión**, que se utilizará en casi todos los capítulos, y también su concepto relacionado, el **backtracking**.

2.2 ¿Qué es la recursión?

Toda función que se llama a sí misma se denomina **recursiva**. Un método recursivo resuelve un problema llamando a una copia de sí mismo para trabajar sobre un problema más pequeño. Esto se denomina **paso recursivo**. El paso recursivo puede dar lugar a muchas más llamadas recursivas de este tipo.

Es importante asegurarse de que la recursión termine. Cada vez, la función se llama a sí misma con una versión ligeramente más sencilla del problema original. La secuencia de problemas más pequeños debe converger finalmente en el **caso base**.

2.3 ¿Por qué recursión?

La recursión es una técnica útil tomada de las matemáticas. El código recursivo generalmente es más corto y fácil de escribir que el código iterativo. Por lo general, los bucles se convierten en funciones recursivas cuando se compilan o interpretan.

La recursión es más útil para tareas que pueden definirse en términos de subtareas similares. Por ejemplo, los problemas de ordenamiento, búsqueda y recorrido suelen tener soluciones recursivas simples.

2.4 Formato de una función recursiva

Una función recursiva realiza una tarea, en parte, llamándose a sí misma para ejecutar las subtareas. En algún momento, la función encuentra una subtarea que puede realizar sin llamarse a sí misma.

Este caso, en el que la función no vuelve a llamarse, se denomina **caso base**. El otro, en el que la función se llama a sí misma para realizar una subtarea, se denomina **caso recursivo**. Todas las funciones recursivas pueden escribirse con el siguiente formato:

```
if(test for the base case)
    return some base case value
else if(test for another base case)
    return some other base case value
// the recursive case
else
    return (some work and then a recursive call)
```

Como ejemplo, consideremos la función factorial: $n!$ es el producto de todos los enteros entre n y 1. La definición del factorial recursivo es la siguiente:

$$n! = 1, \quad \text{si } n = 0$$
$$n! = n \cdot (n-1)!, \quad \text{si } n > 0$$

Esta definición puede convertirse fácilmente en una implementación recursiva. Aquí, el problema consiste en determinar el valor de $n!$, y el subproblema consiste en determinar el valor de $(n-1)!$. En el caso recursivo, cuando n es mayor que 1, la función se llama a sí misma para determinar el valor de $(n-1)!$ y lo multiplica por n.

En el caso base, cuando n es 0 o 1, la función simplemente devuelve 1. Esto se ve de la siguiente manera:

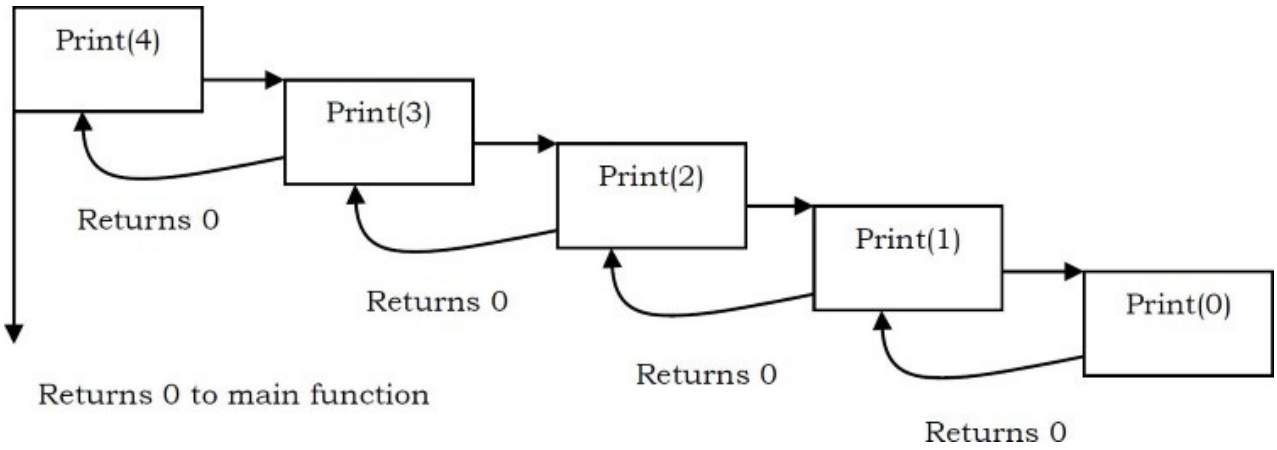
```
// calculates factorial of a positive integer
int Fact(int n) {
    if(n == 1)   // base cases: fact of 0 or 1 is 1
        return 1;
    else if(n == 0)
        return 1;
    else        // recursive case: multiply n by (n - 1) factorial
        return n*Fact(n-1);
}
```

2.5 Recursión y memoria (visualización)

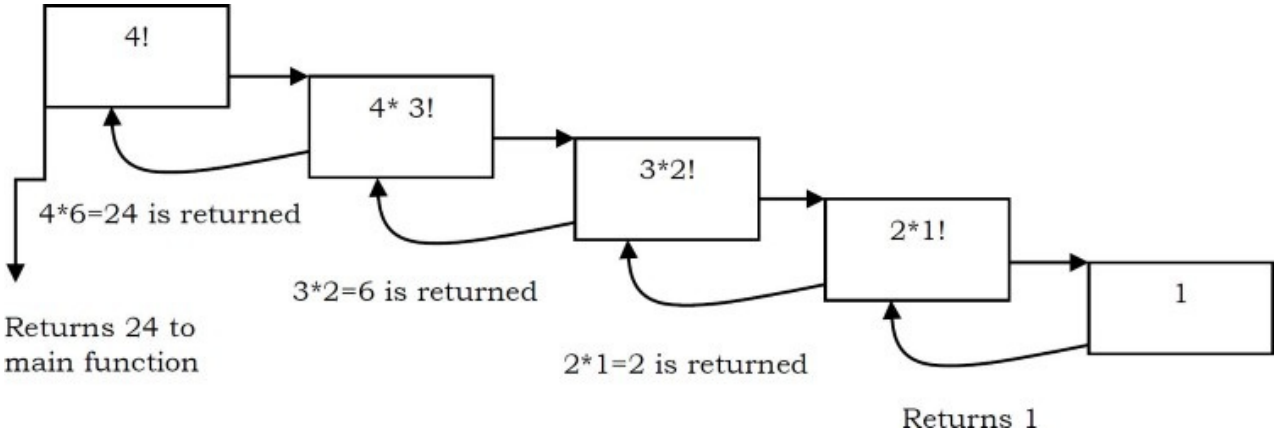
Cada llamada recursiva crea una copia nueva de ese método (en realidad, solo de las variables) en memoria. Cuando un método termina (es decir, devuelve algún dato), la copia de ese método que retorna se elimina de la memoria. Las soluciones recursivas parecen simples, pero visualizarlas y rastrearlas lleva tiempo. Para comprenderlo mejor, consideremos el siguiente ejemplo.

```c
// print numbers 1 to n backward
int Print(int n) {
    if(n == 0) // this is the terminating base case
        return 0;
    else {
        printf("%d",n);
        return Print(n-1);       // recursive call to itself again
    }
}
```

Para este ejemplo, si llamamos a la función `Print` con $n = 4$, visualmente nuestras asignaciones de memoria podrían verse así:



Ahora consideremos nuestra función factorial. La visualización de la función factorial con $n = 4$ se verá así:



2.6 Recursión frente a iteración

Al hablar de recursión, la pregunta básica que viene a la mente es: ¿qué forma es mejor: la iteración o la recursión? La respuesta depende de lo que intentemos hacer. Un enfoque recursivo refleja el problema que intentamos resolver. Un enfoque recursivo simplifica la resolución de un problema

que quizá no tenga una respuesta evidente. Sin embargo, la recursión agrega una sobrecarga en cada llamada recursiva (necesita espacio en el marco de pila).

Recursión

- Termina cuando se alcanza un caso base.
- Cada llamada recursiva requiere espacio adicional en el marco de pila (memoria).
- Si obtenemos una recursión infinita, el programa puede quedarse sin memoria y producir un desbordamiento de pila.
- Las soluciones para algunos problemas son más fáciles de formular recursivamente.

Iteración

- Termina cuando se demuestra que una condición es falsa.
- Cada iteración no requiere espacio adicional.
- Un bucle infinito podría ejecutarse para siempre, ya que no se crea memoria adicional.
- Las soluciones iterativas para un problema no siempre son tan evidentes como una solución recursiva.

2.7 Notas sobre recursión

- Los algoritmos recursivos tienen dos tipos de casos: casos recursivos y casos base.
- Todo caso de una función recursiva debe terminar en un caso base.
- En general, las soluciones iterativas son más eficientes que las soluciones recursivas [debido a la sobrecarga de las llamadas a funciones].
- Un algoritmo recursivo puede implementarse sin llamadas a funciones recursivas utilizando una pila, pero normalmente da más trabajo del que vale la pena. Esto significa que todo problema que puede resolverse recursivamente también puede resolverse iterativamente.
- Para algunos problemas no existen algoritmos iterativos evidentes.
- Algunos problemas se adaptan mejor a soluciones recursivas, mientras que otros no.

2.8 Ejemplos de algoritmos de recursión

- Serie de Fibonacci, cálculo de factoriales.
- Mergesort, quicksort.
- Búsqueda binaria.
- Recorridos de árboles y muchos problemas de árboles: inorden, preorden y postorden.
- Recorridos de grafos: DFS [búsqueda en profundidad] y BFS [búsqueda en anchura].
- Ejemplos de programación dinámica.
- Algoritmos de divide y vencerás.
- Torres de Hanoi.
- Algoritmos de backtracking [se analizarán en la siguiente sección].

2.9 Recursión: problemas y soluciones

En este capítulo cubrimos algunos problemas con recursión y analizaremos el resto en otros capítulos. Cuando termine de leer el libro completo, encontrará muchos problemas de recursión.

Problema 1. Analizar el rompecabezas de las Torres de Hanoi.

Solución. Las Torres de Hanoi son un rompecabezas matemático. Consta de tres varillas (o clavijas o torres) y una cantidad de discos de distintos tamaños que pueden deslizarse sobre cualquier varilla. El rompecabezas comienza con los discos en una varilla, ordenados de manera ascendente por tamaño, con el más pequeño arriba, formando así una figura cónica. El objetivo del rompecabezas es mover toda la pila a otra varilla, respetando las siguientes reglas:

- Solo se puede mover un disco a la vez.
- Cada movimiento consiste en tomar el disco superior de una de las varillas y deslizarlo sobre otra varilla, encima de los demás discos que ya puedan estar presentes en ella.
- No se puede colocar ningún disco encima de un disco más pequeño.

Algoritmo:

- Mover los $n-1$ discos superiores de la torre **Source** a la torre **Auxiliary**.
- Mover el disco n de la torre **Source** a la torre **Destination**.
- Mover los $n-1$ discos de la torre **Auxiliary** a la torre **Destination**.
- La transferencia de los $n-1$ discos superiores de la torre **Source** a la torre **Auxiliary** puede considerarse nuevamente un problema nuevo y resolverse de la misma manera. Una vez que resolvemos las Torres de Hanoi con tres discos, podemos resolverlas con cualquier cantidad de discos mediante el algoritmo anterior.

CAPÍTULO 6

Listas doblemente enlazadas

Extraído del capítulo 3 de **Data Structures and Algorithms Made Easy**, págs. 88–96

3.7 Listas doblemente enlazadas

La ventaja de una lista doblemente enlazada (también llamada lista enlazada de dos vías) es que, dado un nodo de la lista, podemos desplazarnos en ambas direcciones. Un nodo de una lista simplemente enlazada no puede eliminarse a menos que tengamos el puntero a su predecesor. En una lista doblemente enlazada, en cambio, podemos eliminar un nodo incluso si no tenemos la dirección del nodo anterior, ya que cada nodo tiene un puntero `prev` que apunta al nodo anterior y permite retroceder.

Las principales desventajas de las listas doblemente enlazadas son:

- Cada nodo necesita un puntero adicional, por lo que requiere más espacio.
- Insertar o eliminar un nodo toma un poco más de tiempo, debido a la mayor cantidad de operaciones sobre punteros.

Al igual que en una lista simplemente enlazada, implementaremos las operaciones de una lista doblemente enlazada. Si se comprenden las operaciones de las listas simplemente enlazadas, las operaciones de las listas doblemente enlazadas resultan evidentes.

La siguiente es una declaración de tipo para una lista doblemente enlazada de enteros:

```
struct DLLNode {
    int data;
    struct DLLNode *next;
    struct DLLNode *prev;
};
```

En los diagramas de la fuente, los punteros `prev` y `next` aparecen como enlaces izquierdo y derecho, respectivamente. Aquí se conservan los nombres de campo del código de la fuente.

Inserción en una lista doblemente enlazada

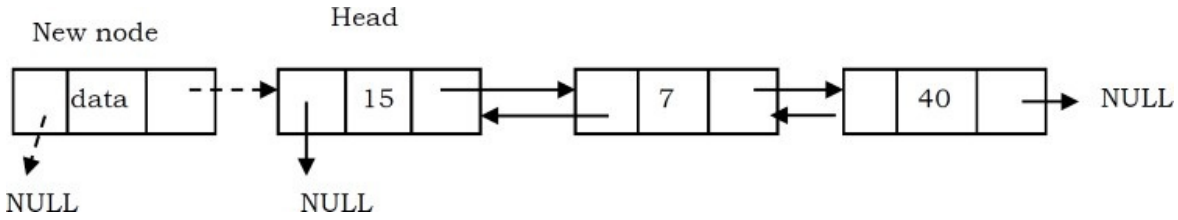
La inserción en una lista doblemente enlazada tiene tres casos, al igual que en una lista simplemente enlazada:

- Insertar un nodo nuevo antes de `head`.
- Insertar un nodo nuevo después de `tail` (al final de la lista).
- Insertar un nodo nuevo en el medio de la lista.

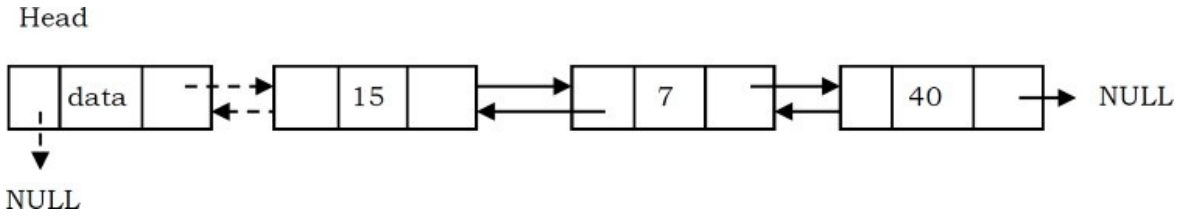
Insertar un nodo al principio de una lista doblemente enlazada

En este caso, el nodo nuevo se inserta antes del nodo `head`. Deben modificarse los punteros anterior y siguiente, y esto puede hacerse en dos pasos:

- Actualizar el puntero `next` del nodo nuevo para que apunte al `head` actual (el enlace punteado de la figura siguiente) y hacer que el puntero `prev` del nodo nuevo sea `NULL`.



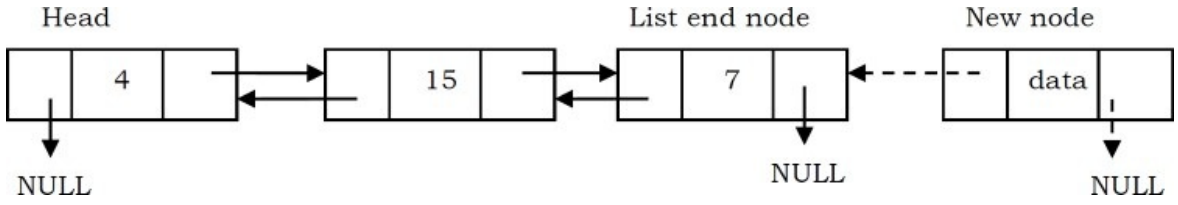
- Actualizar el puntero `prev` del nodo `head` para que apunte al nodo nuevo y hacer que el nodo nuevo sea el `head`.



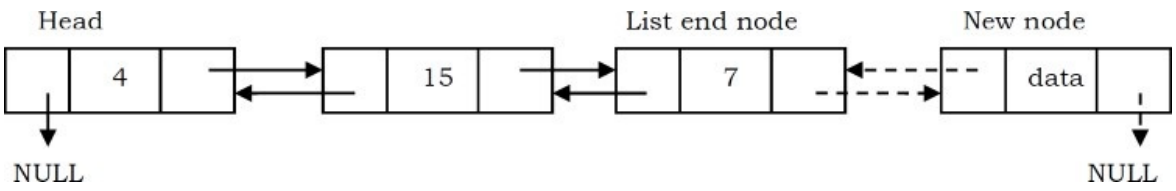
Insertar un nodo al final de una lista doblemente enlazada

En este caso, recorremos la lista hasta el final e insertamos el nodo nuevo.

- El puntero `next` del nodo nuevo apunta a `NULL` y el puntero `prev` apunta al final de la lista.



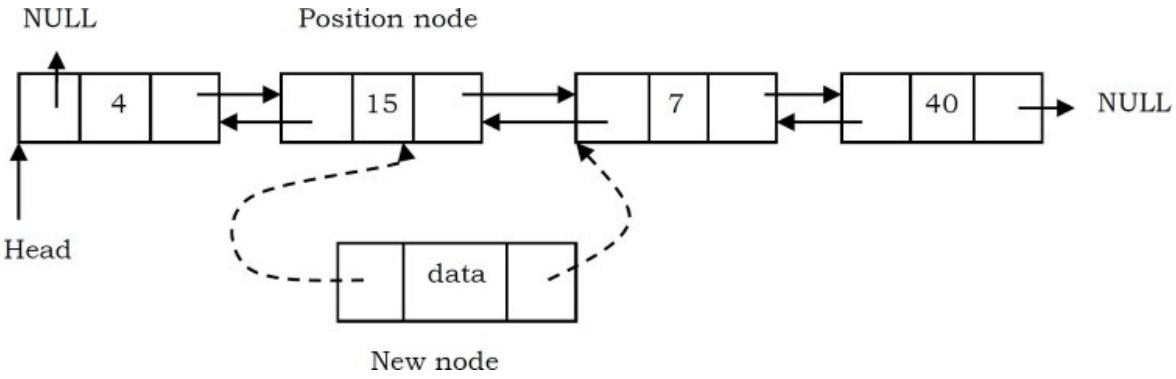
- Actualizar el puntero `next` del último nodo para que apunte al nodo nuevo.



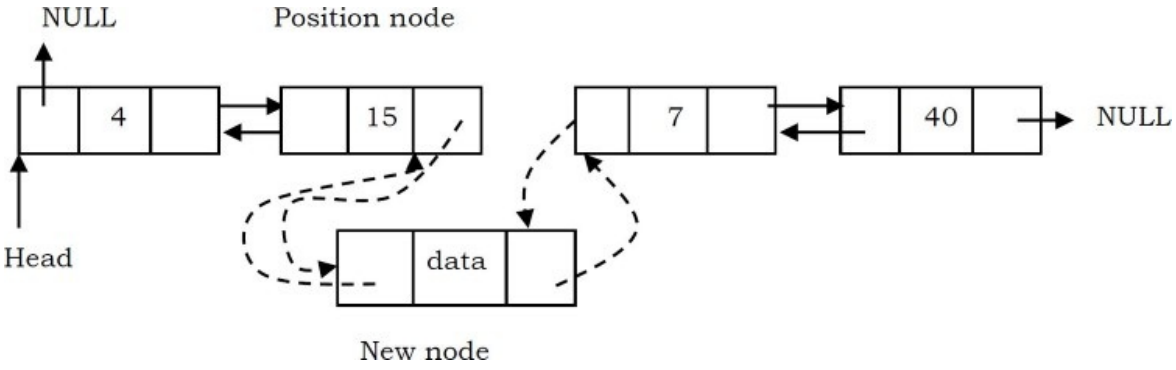
Insertar un nodo en el medio de una lista doblemente enlazada

Como en las listas simplemente enlazadas, recorremos la lista hasta el nodo de posición e insertamos allí el nodo nuevo.

- El puntero `next` del nodo nuevo apunta al siguiente nodo del nodo de posición donde queremos insertarlo. Además, el puntero `prev` del nodo nuevo apunta al nodo de posición.



- El puntero `next` del nodo de posición apunta ahora al nodo nuevo y el puntero `prev` del siguiente nodo del nodo de posición apunta al nodo nuevo.



La función original de la fuente recibe `struct DLLNode **head`, porque necesita actualizar el puntero al primer elemento en la función llamadora. En la cursada usamos la interfaz `head` → nuevo `head`: la función recibe el `head` actual y devuelve el nuevo cuando la inserción ocurre al principio. La operación sobre los enlaces es la misma y no se duplica la implementación del doble puntero presentada en el Capítulo 3.

```c
#include <stdlib.h>

struct DLLNode *DLLInsert(
    struct DLLNode *head, int data, int position)
{
    if (position < 1) {
        return head;
    }

    struct DLLNode *newNode = malloc(sizeof *newNode);
    if (newNode == NULL) {
        return head;
    }
    newNode->data = data;

    if (position == 1) {
        newNode->next = head;
        newNode->prev = NULL;
        if (head != NULL) {
            head->prev = newNode;
        }
        return newNode;
    }

    struct DLLNode *positionNode = head;
    for (int k = 1;
         positionNode != NULL && k < position - 1;
         k++) {
        positionNode = positionNode->next;
    }

    if (positionNode == NULL) {
        free(newNode);
        return head;
    }

    newNode->next = positionNode->next;
    newNode->prev = positionNode;
    if (positionNode->next != NULL) {
        positionNode->next->prev = newNode;
    }
    positionNode->next = newNode;
    return head;
}

/* Ejemplo de uso: conservar el head que devuelve la función. */
head = DLLInsert(head, data, position);
```

La función acepta insertar al final cuando `position` es la posición posterior al último nodo. Si la posición no es válida o falla la reserva, conserva el `head` original.

Complejidad temporal: $O(n)$. En el peor caso, puede ser necesario insertar el nodo al final de la lista.

Complejidad espacial: $O(1)$, para crear una variable temporal, sin contar el nodo nuevo.

Eliminación en una lista doblemente enlazada

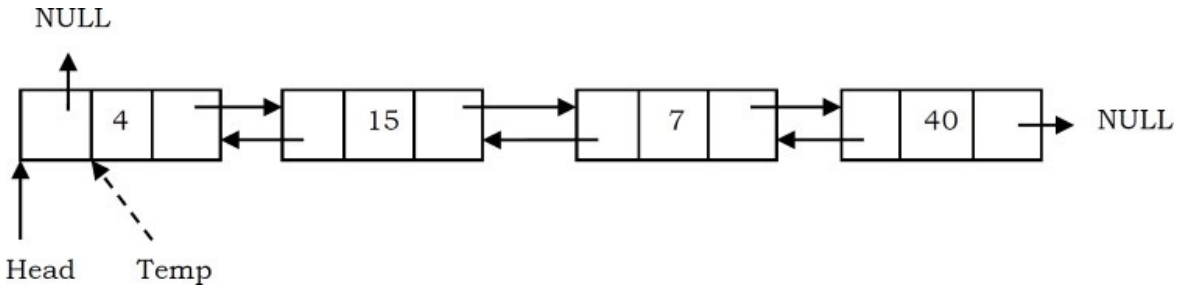
Al igual que en la eliminación de una lista simplemente enlazada, aquí tenemos tres casos:

- Eliminar el primer nodo.
- Eliminar el último nodo.
- Eliminar un nodo intermedio.

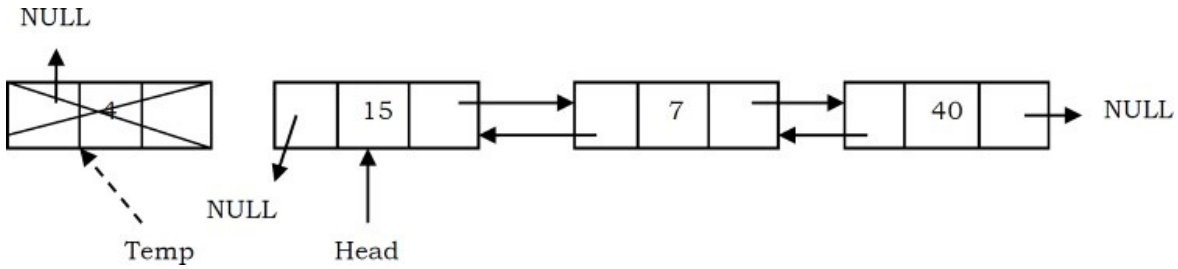
Eliminar el primer nodo de una lista doblemente enlazada

En este caso se elimina el primer nodo, el nodo que actualmente es `head`. Puede hacerse en dos pasos:

- Crear un nodo temporal que apunte al mismo nodo que `head`.



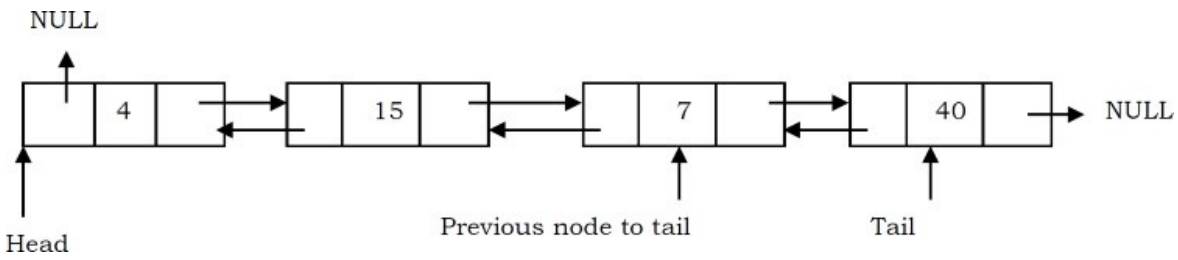
- Mover el puntero de `head` al nodo siguiente, cambiar a `NULL` el puntero `prev` del nuevo `head` y descartar el nodo temporal.



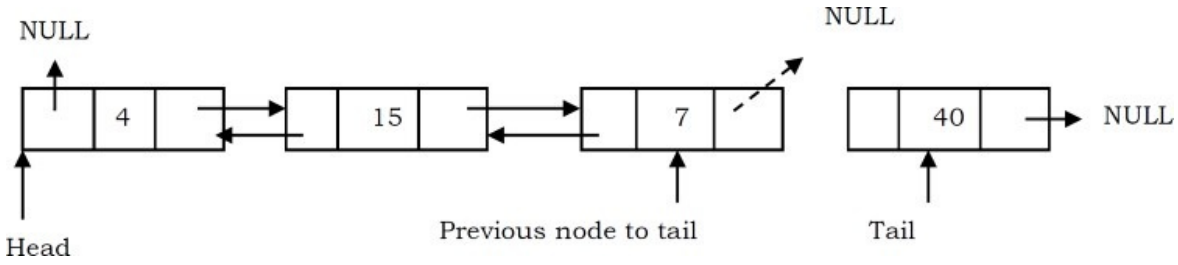
Eliminar el último nodo de una lista doblemente enlazada

Esta operación es un poco más complicada que eliminar el primero, porque el algoritmo primero debe encontrar el nodo anterior al `tail`. Puede hacerse en tres pasos:

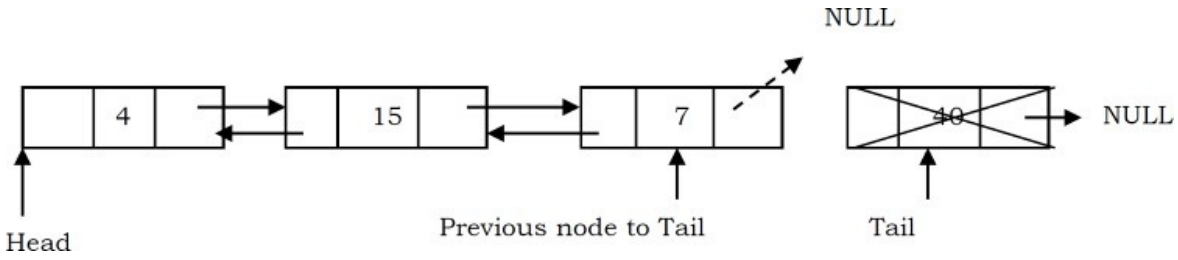
- Recorrer la lista y mantener también la dirección del nodo anterior. Cuando lleguemos al final de la lista, tendremos dos punteros: uno que apunta al `tail` y otro que apunta al nodo anterior al `tail`.



- Actualizar a `NULL` el puntero `next` del nodo anterior al `tail`.



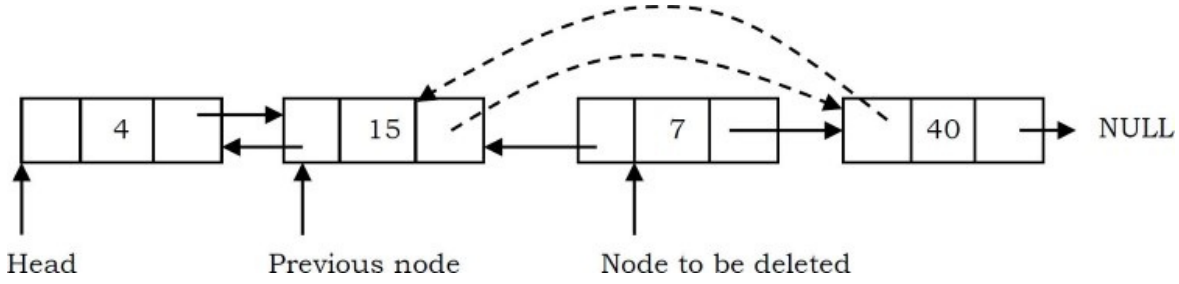
- Descartar el nodo `tail`.



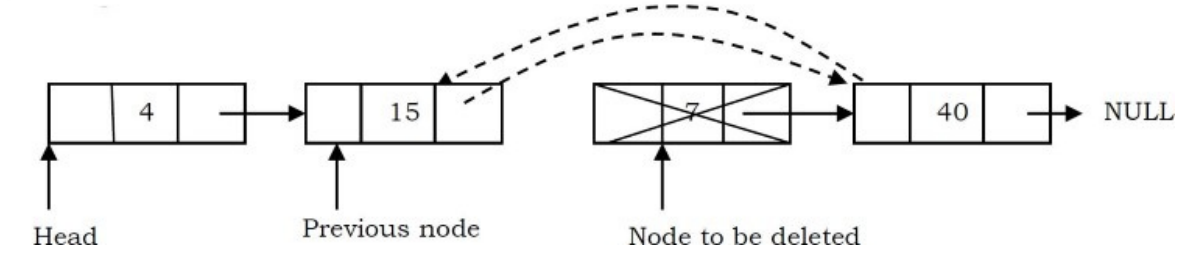
Eliminar un nodo intermedio de una lista doblemente enlazada

En este caso, el nodo que se eliminará siempre está ubicado entre otros dos nodos, y los enlaces `head` y `tail` no se actualizan. La eliminación puede hacerse en dos pasos:

- Al igual que en el caso anterior, mantener el nodo anterior durante el recorrido. Cuando encontremos el nodo que se eliminará, cambiar el puntero `next` del nodo anterior para que apunte al siguiente nodo del nodo que se eliminará.



- Descartar el nodo actual que se eliminará.



Complejidad temporal: $O(n)$, para recorrer la lista completa de tamaño n.

Complejidad espacial: $O(1)$, para crear una variable temporal.

> **Procedencia:** Narasimha Karumanchi, **Data Structures and Algorithms Made Easy**, capítulo 3, sección 3.7, págs. impresas 88–96.