Push_swap — Apunte de Clase (Estructuras, Listas, Stack)

Escuela 42 Madrid — Material offline para repaso intensivo

Contenido en español; código, funciones y variables en inglés. Ejercicios con solución tras cada enunciado.

1. Estructuras (struct) en C

Una estructura (struct) agrupa varios datos bajo un mismo tipo. Es un molde que describe cómo es un dato compuesto. La pareja typedef + struct te permite definir un alias legible para ese molde.

1.1. Declaración y typedef

```
typedef struct s_student {
    char  *name;     // cadena
    int    age;      // entero
    float  grade;    // flotante
} t_student;

// Uso
t_student alice;
alice.name  = "Alice";
alice.age   = 21;
alice.grade = 8.7f;
```

Notas clave:

• Usa el operador punto (.) si tienes una variable (no puntero) y la flecha (->) si tienes un puntero a struct.

```
t_student bob;
t_student *p = &bob;
bob.age = 20;      // variable directa -> '.'
p->age  = 20;      // puntero -> '->'
```

1.2. Inicialización en una sola línea

```
t_student alice = {"Alice", 21, 8.7f};              // orden exacto de los campos
t_student bob   = {.name = "Bob", .age = 20};       // designated initializers (otros quedan a 0/
```

Esta inicialización solo es válida en el momento de la declaración.

1.3. Auto-referencia dentro de la estructura

Cuando un struct se referencia a sí mismo, debes usar la MISMA etiqueta de struct en los punteros internos.

```
typedef struct s_node {
    int              value;
    struct s_node   *next;   // MISMA etiqueta
    struct s_node   *prev;   // MISMA etiqueta
} t_node;
```

1.4. Pasar estructuras a funciones

```
typedef struct s_player {
    char *name;
    int   age;
} t_player;

void print_player(const t_player *p) {   // por puntero (no se copia)
    printf("%s (%d)\n", p->name, p->age);
}
```

Ejercicio 1 — Structs básicos

Declara un tipo t_book con title (char*), pages (int) y price (float). Crea una variable e inicialízala en una sola línea. Imprime sus campos.

Solución:

```c
typedef struct s_book {
    char  *title;
    int    pages;
    float  price;
} t_book;

int main(void) {
    t_book b = {"C Notes", 180, 19.90f};
    printf("%s | %d | %.2f\n", b.title, b.pages, b.price);
    return 0;
}
```

2. Listas Enlazadas

Una lista enlazada es una secuencia de nodos conectados por punteros. En una lista doble, cada nodo conoce al siguiente (next) y al anterior (prev).

```
NULL <- [ value | prev | next ] <-> [ value | prev | next ] <-> [ value | prev | next ] -> NULL
```

2.1. Nodo y lista (stack) como tipos

```
typedef struct s_node {
    int              value;
    struct s_node   *next;
    struct s_node   *prev;
} t_node;

typedef struct s_stack {
    t_node *top;       // primer nodo
    t_node *bottom;    // último nodo
    int     size;      // cantidad de nodos
} t_stack;
```

2.2. Crear estructuras en memoria

```
t_stack *init_stack(void) {
    t_stack *s = malloc(sizeof(t_stack));
    if (!s) return NULL;
    s->top = NULL;
    s->bottom = NULL;
    s->size = 0;
    return s;
}

t_node *new_node(int v) {
    t_node *n = malloc(sizeof(t_node));
    if (!n) return NULL;
    n->value = v;
    n->next = NULL;
    n->prev = NULL;
    return n;
}
```

2.3. Invariantes (siempre deben cumplirse)

• Si size == 0: top == NULL y bottom == NULL.

• Si size == 1: top == bottom; top->prev == NULL; bottom->next == NULL.

• En general: top->prev == NULL y bottom->next == NULL.

Ejercicio 2 — Enlazar tres nodos a mano

Crea un stack vacío y tres nodos (4, 7, 9). Enlázalos para formar top→4↔7↔9←bottom. Comprueba que size=3, top->prev==NULL y bottom->next==NULL.

Solución:

```
t_stack *a = init_stack();
t_node *n1 = new_node(4);
t_node *n2 = new_node(7);
t_node *n3 = new_node(9);

a->top = n1;
```

```
a->bottom = n3;
a->size = 3;

n1->next = n2;  n2->prev = n1;
n2->next = n3;  n3->prev = n2;

// checks
// n1->prev == NULL? (sí, por defecto al crear estaba a NULL)
// n3->next == NULL? (sí, por defecto)
```

3. Operaciones básicas sobre la lista/stack

Automatizamos la conexión de punteros con funciones. Estas funciones serán la base para las operaciones de push_swap.

3.1. Insertar arriba (push_front)

```
void push_front(t_stack *s, t_node *node) {
    if (!s || !node) return;
    if (s->size == 0) {
        s->top = node;
        s->bottom = node;
    } else {
        node->next = s->top;
        s->top->prev = node;
        s->top = node;
    }
    s->size++;
}
```

3.2. Insertar abajo (push_back)

```
void push_back(t_stack *s, t_node *node) {
    if (!s || !node) return;
    if (s->size == 0) {
        s->top = node;
        s->bottom = node;
    } else {
        node->prev = s->bottom;
        s->bottom->next = node;
        s->bottom = node;
    }
    s->size++;
}
```

3.3. Quitar arriba (pop_front) — devuelve el nodo

```
t_node *pop_front(t_stack *s) {
    t_node *n;
    if (!s || s->size == 0) return NULL;
    n = s->top;
    s->top = n->next;
    if (s->top)
        s->top->prev = NULL;
    else
        s->bottom = NULL;  // quedó vacío
    n->next = NULL;
    n->prev = NULL;
    s->size--;
    return n;
}
```

3.4. Quitar abajo (pop_back) — devuelve el nodo

```
t_node *pop_back(t_stack *s) {
    t_node *n;
    if (!s || s->size == 0) return NULL;
    n = s->bottom;
    s->bottom = n->prev;
    if (s->bottom)
        s->bottom->next = NULL;
```

```
    else
        s->top = NULL;     // quedó vacío
    n->next = NULL;
    n->prev = NULL;
    s->size--;
    return n;
}
```

3.5. Liberar toda la memoria del stack

```
void free_stack(t_stack *s) {
    t_node *cur = s ? s->top : NULL;
    while (cur) {
        t_node *next = cur->next;
        free(cur);
        cur = next;
    }
    free(s);
}
```

Ejercicio 3 — Construcción incremental

Crea un stack vacío e inserta (por este orden): push_back(10), push_back(20), push_front(5), push_back(30). Dibuja el estado final y el valor de top, bottom y size.

Solución:

Tras las cuatro operaciones: top→5↔10↔20↔30←bottom, size=4.

Ejercicio 4 — Extracción

Con el stack del ejercicio anterior, realiza: pop_front(), pop_back(). ¿Qué valores devolvió cada función y cómo quedó la estructura?

Solución:

pop_front() devuelve 5; la lista queda 10↔20↔30. Luego pop_back() devuelve 30; la lista queda 10↔20; top=10, bottom=20, size=2.

4. Del Stack a las Operaciones de push_swap

push_swap impone un conjunto cerrado de operaciones sobre dos stacks (a y b). Aquí describimos su efecto apoyándonos en nuestras primitivas.

4.1. Operaciones y su intención

Operación	Efecto (intención)
sa/sb	Intercambia los dos primeros elementos del stack.
ss	Aplica sa y sb simultáneamente.
pa	Mueve el primer elemento de b a la cima de a.
pb	Mueve el primer elemento de a a la cima de b.
ra/rb	Rotate: el primero pasa al final.
rr	ra y rb a la vez.
rra/rrb	Reverse rotate: el último pasa al principio.
rrr	rra y rrb a la vez.

4.2. Implementaciones base (plantillas)

Swap arriba (sa/sb) se puede hacer re-enlazando punteros de los dos primeros nodos.

```
void swap_top(t_stack *s) {
    t_node *a, *b, *c;
    if (!s || s->size < 2) return;
    a = s->top;        // 1º
    b = a->next;       // 2º
    c = b->next;       // 3º (puede ser NULL)

    // b sube a top
    s->top = b;
    b->prev = NULL;
    b->next = a;

    // a baja a 2º
    a->prev = b;
    a->next = c;

    if (c) c->prev = a;
    else   s->bottom = a; // si solo había 2, ahora bottom es 'a'
}
```

Push a otro stack (pa/pb) se hace con pop_front del origen y push_front en el destino.

```
void push_from_to(t_stack *src, t_stack *dst) {
    t_node *n = pop_front(src);
    if (n) push_front(dst, n);
}
```

Rotate (ra/rb): pop_front y push_back del mismo stack.

```
void rotate(t_stack *s) {
    t_node *n = pop_front(s);
    if (n) push_back(s, n);
}
```

Reverse rotate (rra/rrb): pop_back y push_front.

```
void reverse_rotate(t_stack *s) {
    t_node *n = pop_back(s);
```

```
    if (n) push_front(s, n);
}
```

Ejercicio 5 — Simulación manual

Con a=[3,2,1] y b=[], aplica: sa; pb; pb; pa. Escribe el estado de a y b tras cada paso.

Solución:

Inicio: a=3,2,1; b=[]. Tras sa: a=2,3,1; b=[]. Tras pb: a=3,1; b=2. Tras pb: a=1; b=3,2. Tras pa: a=3,1,2; b=2.

Ejercicio 6 — Implementa ra/rra con primitivas

Escribe las funciones rotate y reverse_rotate (ya mostradas) y prueba con a=[1,2,3,4]. Comprueba estados: ra→[2,3,4,1], rra→[1,2,3,4].

Solución:

Las implementaciones mostradas arriba son válidas. Con a=[1,2,3,4]: rotate: extrae 1 (pop_front) y lo inserta al final → [2,3,4,1]. reverse_rotate: extrae 4 (pop_back) y lo pone delante → [4,1,2,3].

5. Robustez, errores comunes y depuración

• Coherencia de etiquetas: si declaras 'struct s_node', usa esa etiqueta internamente.

• Flecha vs punto: punteros '->', variables '.'.

• Comprobar NULL antes de desreferenciar.

• Mantener invariantes de top/bottom/size siempre actualizados.

• Liberar memoria: cada malloc debe tener su free. Crea 'free_stack'.

• Debug: imprime estados tras cada operación compleja; añade aserciones si lo ves útil.

Ejercicio 7 — free_stack seguro

Escribe y prueba una función free_stack que libere todos los nodos y después el propio t_stack.

Solución:

```
void free_stack(t_stack *s) {
    if (!s) return;
    for (t_node *cur = s->top; cur; ) {
        t_node *next = cur->next;
        free(cur);
        cur = next;
    }
    free(s);
}
```

Ejercicio 8 — Comprobaciones defensivas

Añade comprobaciones de punteros NULL en push/pop y tests para los casos size=0 y size=1.

Solución:

En el código mostrado, todas las funciones verifican '!s' o 'size == 0' antes de operar. Para size==1, las ramas que ajustan top y bottom a NULL dejan el stack vacío de forma segura.

Cierre y siguientes pasos

Ya dominas la base: structs, listas dobles y operaciones de inserción/extracción. Con estas primitivas puedes implementar directamente las operaciones sa/sb/ss, pa/pb, ra/rb, rra/rrb. El siguiente bloque será parseo de argumentos, validación y primeras estrategias de ordenación para push_swap (3, 5 y N elementos).