Push_swap — Parte 2: Inserciones, Extracciones y Operaciones

Apunte narrativo — continuación de listas y stacks. Incluye teoría, diagramas, ejercicios y soluciones.

Capítulo 1 — Añadir nodos al stack automáticamente

Necesitamos automatizar la conexión de punteros para añadir nodos. Las funciones push_front y push_back lo hacen.

```
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++;
}
```

```
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++;
}
```

Ejercicio — Inserta nodos y describe el estado final del stack.

Solución: tras push_back(10), push_back(20), push_front(5), push_back(30): top=5, bottom=30, size=4.

Capítulo 2 — Quitar nodos del stack

pop_front y pop_back extraen nodos actualizando los punteros.

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

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

Ejercicio — Realiza pop_front y pop_back sobre el stack anterior. Solución: devuelve 5 y 30; stack final 10↔20.

Capítulo 3 — Operaciones básicas de push_swap

Estas funciones permiten modificar los stacks a y b según las reglas del proyecto.

```
void swap_top(t_stack *s) {
    if (!s || s->size < 2) return;
    t_node *a = s->top;
    t_node *b = a->next;
    a->next = b->next;
    b->prev = NULL;
    b->next = a;
    a->prev = b;
    s->top = b;
    if (a->next)
        a->next->prev = a;
    else
        s->bottom = a;
}
```

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

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

Ejercicio — Simula a=[3,2,1], b=[], aplica sa, pb, pb, pa. Solución: a→[3,2,1]->sa→[2,3,1]->pb→a[3,1],b[2]->pb→a[1],b[3,2]->pa→a[3,1,2],b[2].

Capítulo 4 — Errores comunes y depuración

1. Segfault: acceder a punteros NULL. Solución: comprueba !s y !node. 2. Leaks: libera cada malloc con free_stack. 3. Double free: nunca liberes dos veces el mismo puntero. 4. Mantén size, top y bottom siempre coherentes.

```
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);
}
```

Consejos: imprime el stack tras cada operación mientras desarrollas, usa valgrind para detectar leaks.

Conclusión

Con estas funciones dominas el comportamiento interno de push_swap. La próxima etapa será leer y validar los argumentos y aplicar algoritmos de ordenación. Ahora entiendes cómo se conectan y manipulan los nodos, y cómo las operaciones modifican los stacks.