



Act 2.3

Importancia y eficiencia de listas doblemente ligadas

Santiago Amir Rodríguez González
A01739942

Reflexión sobre el proyecto de las listas doblemente ligadas en la bitácora

Ya había hecho esta actividad con vectores, así que sabía cómo estructurar el problema: leer el archivo, descomponer cada línea, ordenar y hacer búsquedas por rango. Esta vez el único cambio fue la estructura de datos. Al intentar implementar QuickSort con listas doblemente ligadas me encontré con que era mucho más difícil y complejo sin acceso directo a índices. Por eso cambié a Merge Sort, que resultó ser mucho más natural para trabajar con listas ligadas. No necesitaba pivotes complicados; simplemente dividía la lista por la mitad, ordenaba recursivamente y fusionaba.

La lectura y descomposición del archivo funcionó igual que antes. La función `tokenizer` ya estaba lista, y separar la IP octeto por octeto también. El cambio real estuvo en la gestión de memoria: crear nodos correctamente, asegurar que ambos apuntadores (`prev` y `next`) estuvieran bien enlazados. Con Merge Sort en listas aprendí a dividir correctamente usando el método de la tortuga y la liebre. Una vez que lo dominé, fusionar las sublistas fue natural.

La verdadera ventaja de dos apuntadores apareció en el despliegue. Cuando necesitaba mostrar resultados de forma descendente, simplemente recorría hacia atrás usando `prev` sin invertir la lista. Con vectores habría sido más complicado.

Reflexionando: las listas doblemente ligadas demostraron ser útiles para este caso específico, no porque sean "mejores" en general, sino porque encajaban bien con los requisitos. La navegación bidireccional fue práctica para el despliegue descendente. Las

desventajas son obvias: gastan el doble de memoria por nodo, las búsquedas siguen siendo $O(n)$, y requieren más cuidado con memory leaks.

Comparando con vectores: los vectores son más eficientes en memoria y QuickSort es más natural. Un árbol binario habría sido teóricamente óptimo pero excesivo. Las listas doblemente ligadas sacrifican eficiencia de memoria por flexibilidad de navegación, y eso fue exactamente lo que necesitaba aquí.

La conclusión es que cambiar de vectores a listas no fue radical en la lógica, sino en la ejecución. Resultaron adecuadas para este problema porque encajaron con los requisitos de navegación bidireccional y con Merge Sort. Aprendí que la elección de estructura de datos no es absoluta, sino relativa al contexto y los algoritmos que la acompañan.