Trabajo Práctico 2
de Algoritmos II

Grupo:	*(Individual)*
Integrantes:	Nestor Fabian Palavecino Arnold
Ayudante asignado:	Lucía Capón Paul
Período:	2do Cuatrimestre 2021
Institución:	FIUBA

Análisis del problema

- **Login:** El requisito de guardar la persona que se loguea en el sistema en O(1) es casi trivial (es simplemente asignar un valor a una variable), por lo que realizar esta tarea no ofreció mayores complicaciones.

- **Logout:** Idem login.

- **Publicar post:** para realizar esta tarea en *O(U * log P)*, siendo U la cantidad de usuarios, y P la cantidad de publicaciones/posts, tenemos que realizar U tareas con complejidad *O(log P)*. Podría haber utilizado un ABB para guardar los posts. Pero luego se menciona el requisito de mostrar las publicaciones del feed del usuario logueado siguiendo un orden de prioridad. Luego, la elección resulta clara. Heap es el camino a seguir para resolver este problema. Encolar un post en un feed cualquiera cuesta *O(log P)*. Luego, repetir este paso U-1 veces (porque el usuario logueado no va a ver su propio post en su feed), nos deja la complejidad total en *O((U-1) * log P)*, es decir, *O(U * log P)*.

- **Ver próximo post:** siguiendo la explicación anterior, si utilizamos un heap como feed para guardar los posts, posteriormente obtener el siguiente post cuesta *O(log P),* siendo P la cantidad de posts dentro del heap del feed del usuario actual.

- **Likear post:** este caso estuvo más complicado. Porque parecía fácil. El requisito era "dar un like" en *O(log U)*. Pero utilizando un arreglo, un vector dinámico o un hash, la complejidad podría haber sido *O(1)*. Mucho mejor. Sin embargo, para guardar los likes decidí utilizar un ABB, que efectivamente, cumple con el requisito pedido en las especificaciones del trabajo práctico. En el punto siguiente se explica el motivo. Cada post guarda en su interior un ABB con las personas que le dieron like como clave, y nada como valor (porque no hacía falta).

- **Mostrar likes:** siguiendo con la explicación anterior, lo que me permitió el utilizar un ABB para guardar los likes de un post fue poder obtener de manera ordenada en *O(U)* los nombres de las personas que le dieron like a ese post, ya que el requisito fundamental de mostrar los likes era que los nombres de los usuarios estuvieran en orden alfabético. Podría haber utilizado vector dinámico y mergesort, y listo. Pero se pedía una complejidad *O(U)*. Y utilizando un vector o un hash la complejidad del ordenamiento ascendía, por lo menos, hasta *O(U * log U)*. Y esta fue la razón que me hizo decantarme por ABB. Obtener los nombres ordenados se convierte en algo tan simple como recorrer *in-order* el árbol binario de búsqueda, guardando las claves en una estructura auxiliar con una función "visitar". Recorrer in-order el ABB cuesta *O(U),* siendo U la cantidad de usuarios que dieron like a ese post. La estructura auxiliar para obtener los likes es una cola, ya que, insertar y obtener los elementos es tan simple como encolar y desencolar, respectivamente, y, aprovechando la propiedad FIFO de la cola, los elementos salen en el mismo orden en el que ingresaron a la cola, el alfabético, el cual era el que se pedía. Y el orden se mantiene en *O(U)*.

Estructuras utilizadas

Para desarrollar este trabajo práctico, tuve que:

- Crear un nuevo TDA, **vd_t**, una variación del vector dinámico que hicimos al principio de la cursada, pero ahora con la posibilidad de aceptar datos en formato **void*** en vez de int. Otra mejora aplicada sobre el vector dinámico es la posibilidad de especificar una función de destrucción de dato a la hora de eliminar la estructura.

- Utilizar **todas** las estructuras de datos que vimos hasta ahora, directa o indirectamente:
    a. *ABB* para guardar los likes de un post
    b. *Cola* para mostrar los likes de un post
    c. *Heap* para guardar y mostrar el feed de los usuarios
    d. *Hash* para guardar la información de los usuarios asociada a su nombre
    e. *Lista* (indirectamente) para usar el TDA Hash
    f. *Pila* (indirectamente) para usar el TDA Heap
    g. *Vector* (indirectamente), en forma del TDA nuevo que mencionamos antes.

- Crear nuevas estructuras de datos que no llegan del todo a ser TDAs:

    a. ***Struct algogram / algogram_s:*** contiene todos los datos de la red social. Basicamente, pasar un *algogram_s** como parámetro implica pasar todos los datos necesarios "de un saque". Esto simplificó muchisimo el desarrollo del trabajo.

    b. ***Struct usuario / usuario_s:*** guarda el ID asociado a ese usuario y su respectivo feed de publicaciones/posts. No llega a ser un TDA porque esta implementación de un usuario es específica de este TP, y no contiene todos los elementos necesarios para ser independiente (por ejemplo, el nombre del usuario aquí no se guarda).

    c. ***Struct post / post_s:*** guarda toda la información relacionada a un post/publicación. Esta estructura era el candidato perfecto a ser un TDA, el TDA "Post", pero como le pasamos un *algogram_s** como parámetro a casi todas sus funciones, no cumple con la condición de ser genérico (porque *algogram_s* es una implementación especifica de este TP). Por eso, Post se quedó en estructura auxiliar.

    d. ***Struct post_autor_distancia / postdist_s:*** podemos considerar a esta estructura como un *Wrapper* de la estructura Post, que, adicionalmente, contiene un dato que representa la distancia en el archivo de usuarios, entre el usuario que guarda este post en su feed, y el usuario que lo publicó originalmente. Esto se utiliza para comparar dos posts dentro del heap del cual el feed de publicaciones está conformado, y saber cual se le mostrará primero al usuario logueado.

Diseño de la solución

Muestro a continuación un esquema con las principales estructuras de datos utilizadas: