Trabajo Práctico 2

Red Social AlgoGram

Grupo 20

Integrantes

Alan Ezequiel Valdevenito
Padrón: 107585

Mateo Julián Rico
Padrón: 108127

Corrector

Julián Crespo

Análisis y diseño de la solución. Algoritmos y estructuras de datos utilizados, justificando conforme a los requisitos pedidos en este TP.

¿Que estructuras y TDAs utilizamos?

Implementamos un TDA Vector. La razón por la cual decidimos implementarlo es porque queríamos relacionar el ID de las publicaciones con la posición en la cual se guardan en dicho vector. Además, otra razón por la cual decidimos implementarlo y usarlo es porque al ser un vector podemos acceder a la posición que queramos del mismo y esto solo nos costaría O(1). ¿Qué ventajas nos da por sobre una lista enlazada?, bueno en este caso la ventaja principal es poder acceder a cualquier posición del vector y esto en una lista enlazada no puede hacerse, para ello tendríamos que recorrer toda la lista enlazada y esto nos costaría en complejidad O(n) siendo n la cantidad total de elementos. ¿Qué ventajas nos da por sobre un hash?, en este caso sabemos que en el hash los elementos no se encuentran ordenados (ya que el orden no interesa) entonces justamente nosotros queríamos orden para las publicaciones.

Implementamos también un TDA Red Social. En este TDA lo que hicimos fue relacionar sus primitivas con los comandos que teníamos como objetivo implementar para este TP2. Decidimos implementarlo porque tiene un cierto comportamiento. En este TDA se puede publicar, se puede dar like, se puede loggear, etc. Entonces este TDA nos soluciona el problema de los comandos que son un requisito del TP2 porque como dijimos al principio, sus primitivas están relacionadas con estos comandos.

Por otra parte, los usuarios decidimos que sean un struct en el cual guardamos la información de cada uno de ellos. Decidimos hacerlo de esta forma porque así sería muy fácil tener toda la información de un mismo usuario toda junta (id, nombre, publicaciones que puede ver). Para los post usamos la misma lógica y también decidimos que sean un struct que guarde su información particular (quienes le dieron like, que cantidad de likes tiene, cual es el texto del post, etc). De esta forma logramos tener toda una misma información sobre algo particular (ya sea un usuario o un post) toda en un mismo lugar.

Complejidades de los comandos

➤ Login: debe funcionar en O(1).

➤ Logout: debe funcionar en O(1).

El problema con el que nos encontramos acá es que la complejidad de estos comandos debe ser en tiempo constante. Cómo deben funcionar ambas en O(1) decidimos relacionar a los usuarios con un TDA Hash. Esto nos permite obtenerlos y/o buscarlos en O(1) para poder conectarlos o desconectarlos de la red social.

➤ Publicar Post: debe funcionar en O(u log(p)), siendo u la cantidad de usuarios y p la cantidad de posts que se hayan creado hasta ese momento.

La complejidad O(log (p)) la asociamos a encolar y desencolar de un TDA Heap. Entonces lo que hicimos fue crear un heap para cada usuario en donde vamos a encolar los post que puedan ver según la afinidad entre usuarios y esto cumple con la complejidad. Cada vez que un usuario publica un post se va a encolar en los heap de los demás usuarios y por lo tanto vamos a tener que acceder a los heap de estos (iterándo a los usuarios) lo cual nos va a costar O(u). Por lo tanto, la complejidad final es la pedida.

➤ Ver próximo post en el feed: debe funcionar en O(log(p)).

Como dijimos anteriormente la complejidad O(log(p)) la asociamos a encolar y desencolar de un TDA Heap. Entonces lo que hicimos para que un usuario pueda ver el próximo post en el feed es simplemente desencolar del heap el post que puede ver y esto cumple con la complejidad pedida.

➤ Likear un post: debe funcionar en O(log u).

El problema que nos encontramos a acá, y también uno de los requisitos del TP2, es que un mismo usuario no pueda dar like a una misma publicación dos o mas veces. Entonces como la complejidad debe ser O(log u), decidimos utilizar para cada post un TDA ABB que guarde los nombres de los usuarios que likearon el post. De esta forma al actualizar la cantidad de likes de un post simplemente guardamos el nombre del usuario que dió like en el ABB pero antes de hacer esto lo primero que se hace es buscarlo dentro del ABB para comprobar que no este dando like por segunda vez a un mismo post y una búsqueda en un ABB cuesta O(log u) por lo que cumplimos con la complejidad pedida.

➤ Mostrar likes: debe funcionar en O(u).

En este caso para cumplir con la complejidad lo que hacemos es acceder en O(1) a la variable que guarda la cantidad de likes (esta variable se encuentra en el struct del post como mencionamos anteriormente), luego como los nombres de aquellos usuarios que dieron like se encuentran en un TDA ABB lo que hacemos es iterar este ABB con un recorrido in-order (O(u)) y a cada usuario lo vamos guardando en TDA LISTA para poder mostrárselo al usuario que pide ver los likes. Finalmente, lo que hacemos es iterar este TDA LISTA lo cual nos cuesta también O(u). La complejidad final sería O(2u) pero podemos despreciar esa constante y finalmente decir que la complejidad final será O(u) que es justamente la que se pide.