FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TP2
Algogram

Materia: Algoritmos y Programación II

Periodo: 2C 2021

Integrantes: Ignacio Rodriguez Justo y Ezequiel Martin Tosto Valenzuela

Ayudante: Jorge Collinet

Al comienzo decidimos reunirnos a definir cuáles iban a ser las estructuras utilizadas, los TDAs que íbamos a necesitar y las primitivas a desarrollar.
Para mejor organización y escalabilidad del proyecto decidimos, en primer lugar, crear los siguientes archivos: *"algogram.c"*, *"usuario.c"* y *"publicacion.c"*, y el *"tp2.c"* que se iba a encargar de manejar los archivos y hacer los llamados necesarios para el funcionamiento del programa.

Cada **algogram** iba a contener:

- **hash de usuarios (de tipo usuario_t*)**
- **usuario actual (de tipo usuario_t*)**
- **hash de todas las publicaciones**

Usuario iba a contener:

- **Un texto (char*) nombre**
- **Heap feed (contiene publicaciones)**
- **Número ID**

Cada **publicación** contiene:

- **Usuario creador de la publicación (tipo usuario_t*)**
- **ABB de likes (nombres de usuarios que likearon)**
- **Número cantidad de likes**
- **El texto de la publicación (char*)**
- **El ID (int)**

Ahora vamos a explicar la decisión detrás de cada TDA auxiliar utilizado hablando sobre la complejidad de cada comando.

1. **Login**

Como la complejidad del login tenía que ser *O(1)* decidimos utilizar un **hash** para guardar los usuarios de la red social, ya que **hash_obtener** tiene como complejidad *O(1)* y de esa forma conseguimos el usuario con el que quiere logear Barbara.

2. **Logout**

Su complejidad es *O(1)* porque simplemente lo que hacemos es cambiar el atributo **usuario_actual** de algogram a NULL.

3. **Publicar post**

La complejidad al publicar una publicación es de *O(u*log(p))*, **u** siendo la cantidad de usuarios y **p** la cantidad de posts creados. Lo que hacemos al guardar primero que nada es guardar la publicación en el hash de publicaciones totales, lo cual es *O(1)*. Y luego iteramos todos los usuarios, *O(u)*, y por cada usuario calculamos la afinidad con el usuario creador de la publicación y luego encolamos en su **heap feed**, lo cual tiene como complejidad *O(log(p))*, por ende la complejidad del todo el comando es *O(u*log(p))*.

4. **Ver próximo post en el feed**

Tiene como complejidad *O(log(p))* **p** siendo la cantidad de publicaciones en el feed del usuario, esto es porque desencolamos el heep de feed del usuario actualmente logeado, desencolar es complejidad *O(log(n))*. Por eso mismo utilizamos el TDA heap para guardar las publicaciones, ya que utilizando un método de comparación podríamos tener ordenados dependiendo de la afinidad, así que creamos un struct llamado publicacion_afinidad que contenía la publicación (publicacion_t*) y una afinidad, que se calculaba sacando la distancia entre usuarios.

5. **Likear un post**

Ya que los likes necesitaban estar ordenados en orden alfabético (los nombres de los usuarios que likearon dicho post utilizamos un ABB, entonces cuando hacemos abb_guardar la complejidad de la operación termina siendo *O(log(u))*.

6. **Mostrar likes**

En mostrar likes utilizamos un método muy útil del TDA ABB el cual nos permite, utilizando una función auxiliar, imprimir todos los nombres de los usuarios en orden, esto siendo *O(u)* siendo **u** la cantidad de usuarios que likearon la publicación.

Tuvimos algunos inconvenientes durante el desarrollo del trabajo práctico, pero la verdad, gracias al diseño previo a empezar se nos facilitó mucho implementar funcionalidades y solucionar problemas, ya que el *esqueleto* del programa lo teníamos armado de antemano. Nos pareció divertido el trabajo práctico, pudimos implementar los TDAs vistos anteriormente, diseñar de antemano un programa con múltiples structs y pensar en base a las complejidades de cada comando.