Tp° 2: Algogram

Integrantes: Cesar Moros y Clara Farbiarz

Corrector: Ezequiel Genender

Para realizar este trabajo práctico lo primero que se realizó fue pensar la estructura para luego empezar a escribir en el computador el código. Se empezó a idear las primitivas necesarias, guiandose con las restricciones de complejidad dadas, y luego se creó los registros según las interacciones de las primitivas y pensando en la lógica de donde guardar cada dato.

Algoritmos de los comandos implementados

- Login/Logout: Se comprueba que no hay un usuario ya loggeado, además se tiene que verificar si el usuario ingresado es válido. Para el caso de no loggeado, se planteó que la búsqueda se haga en un hash de usuarios_t, buscando por la clave el nick del usuario, logrando así que la complejidad sea O(1).
- Publicar post: Los posts se guardan en 2 lugares diferentes, en un hash para ubicar los posts con sus respectivo id, autor y contenido, y en una estructura post_afinidad_t que se va a encolar en el heap feed dentro del registro de cada usuario_t, dado que cada usuario tendra su propio feed según su afinidad con el resto de los usuarios. Encolar el post en el heap es O(log(p)) siendo p la cantidad de posts e el feed de cada usuario, que en su peor caso son todos los posts, y ese proceso se hará para el feed de todos los usuarios, logrando la complejidad de O($u$log(p)).
- Ver próximo post en el feed: Se desencola el elemento de mayor prioridad del heap de feed del usuario loggeado, siendo el post_afinidad_t que tendrá mayor afinidad (en este caso, mayor cercanía con con respecto al autor del post en el archivo.txt) por lo que la complejidad es O(log(p)) siendo p la cantidad de posts en el feed del usuario en cuestión, en el peor de los casos, todos los posts del sistema.
- Likear un post: Para likear se penso en un ABB que guarda como clave los nicks de todos los usuarios que likearon el post (y el dato no importa mucho). Guardar, entonces, en un ABB, tendrá complejidad O(log(u)) (siendo u la cantidad de usuarios).
- Mostrar likes: Es recorrer todo el ABB de likes del post con un iterador e imprimir por pantalla cada uno, teniendo esto de complejidad O(u), ya que el peor caso es recorrer todos los usuarios en O(1) c/u.

TDA

sistema.c

Estructuras.

- TDA sistema: Es un tipo de abstracto que contiene al hash de registro de usuarios (usuario_t), un char* del nombre del usuario loggeado (o NULL en su defecto) , el hash de registro de posts (post_t) y la cantidad de posts publicados.

Primitivas:

- sistema_crear(FILE* archivo): Crea el archivo a partir del archivo con los usuarios existentes, e inicializa todos los structs correspondientes de los usuarios, y de los posts
- sistema_orden(sistema_t* sistema, char* orden, FILE* archivo): Recibe la orden que proporcione el usuario y devuelve el resultado por pantalla (además de modificar los structs internos).
- Sistema_destruir(sistema_t* sistema): Destruye el sistema con sus respectivos structs.

usuario.c

Estructuras.

- Registro de usuario (usuario_t): El registro de usuarios contiene su nombre (nick), el orden en el que fue cargado (pos), el feed personal del usuario que le muestra los post según la afinidad que tiene con otros usuarios (registro post_afinidad_t).
- Registro de afinidad de post (post_afinidad t): En este registro está el dato (tipo size_t) de afinidad que tiene el usuario con el autor del post y el registro del post (post_t).

Primitivas:

- usuario_crear(char* nick, size_t pos): Crea el usuario con los datos proporcionados por parametro, su nick, su posicion en el archivo y su feed(heap) vacio.
- usuario_agregar_feed(post_t* post, usuario_t* autor, usuario_t* usuario): Crea el post_afinidad_t para ser encolado en el heap del feed, ordenado por la correspondiente funcion de comparacion de afinidad.
- usuario_feed_vacio(usuario_t* usuario): Devuelve si el feed (heap) del usuario esta vacío.
- void usuario_imprimir_feed(usuario_t* usuario): Desencola del heap el post_afinidad e imprime por pantalla el post de mayor prioridad, en este caso, de mayor afinidad.

- usuario_likear_post(usuario_t* usuario, post_t* post, char* id): Agrega un like del usuario en cuestión en el post objetivo.
- usuario_destruir(usuario_t* usuario): Destruye el usuario junto con el heap y su nick.

post.c

Estructuras.

- Registro de post (post_t): En el registro de post está su contenido (tipo char*), su id (tipo size_t), el ABB con los usuarios que le dieron like (clave char* con el nick de cada usuario y dato NULL) y el nick del usuario que lo publicó (char*).

Primitivas:

- post_crear(char* texto_post, size_t nuevo_id, char* autor): Crea el post a partir de sus parametros: su contenido, su ID, el nick de su autor y un ABB vacio que se llenará de likes
- post_obtener_id(post_t* post): Obtiene el id del post.
- post_obtener_autor(post_t* post): Obtiene el autor del post.
- post_obtener_contenido(post_t* post): Obtiene el contenido del post.
- post_obtener_cantidad_likes(post_t* post): Obtiene la cantidad de likes, equivalente al tamaño del abb de likes.
- post_likear(post_t* post, char* nick): Agrega al abb un like de clave nick del usuario y dato NULL
- void post_imprimir_likes(post_t* post): Recorre todo el ABB através de un iterador a la vez que imprime por pantalla todos los usuarios que dieron like.
- post_destruir(post_t* post): Destruye el post con el ABB de likes, el nick del autor y del contenido.

Otros TDA utilizados:

- hash.c
- heap.c
- abb.c
- pila.c