ALGOGRAM

Informe técnico

Alumnos:

Matias Rueda
Lucas Venutti

Corrector:

Federico Brasburg

ESTRUCTURAS AUXILIARES

Las próximas lineas van a explicar la idea general acerca de las estructuras auxiliares que usamos para poder elaborar el TP2.

```
33  typedef struct algogram {
34      hash_t* hash_users;
35      hash_t* hash_comandos;
36      hash_t* hash_posts;
37      char* user_actual;
38      char* comando;
39  }algogram_t;
```

Es la estructura principal del programa, en donde giran todas las ejecuciones.
La idea al usar un **HASH DE USUARIOS** fue la de poder acceder a los usuarios en tiempo constante, usando las cadenas (brindadas por la base de datos de usuarios) como claves, logrando que *login* y *logout* se ejecuten en tiempo constante, pues su respuesta se basa en chequear si un usuario está o no en el *hash de usuarios.*
Cada usuario del hash tenia como valor otra estructura:

```
20  typedef struct datos_user {
21      heap_t* heap_feed_user;
22      size_t id_user;
23  }datos_user_t;
```

La idea de esta estructura era hacer que cada usuario contara con un **HEAP FEED**, debido a que el feed de cada usuario es personalizado, dependiendo del criterio de la *afinidad* para mostrar los posts.
La motivación principal para usar un heap personalizado surge de que el criterio de afinidad determina cuál es el próximo post que debe ver un usuario.
Así, los comandos publicar post y ver próximo post se ejecutan en el tiempo que deben: publicar un post afecta a todos los usuarios excepto al actual, y su acción es encolar el post en ese heap personalizado, dando una complejidad de O(U*log_P), donde U es la cantidad de usuarios y P la cantidad de posts.
Ahora, ver el siguiente en el feed es literalmente desencolar un post del heap del usuario logueado, teniendo una complejidad de O(log_P).

Una observación importante es que, lo que se encola en cada heap no es un post en sí, sino un **BLOQUE** que contiene al post y a la prioridad (afinidad) que tiene dicho post:

```
47  typedef struct bloque_post {
48      int afinidad;
49      post_t* post;
50  }bloque_post_t;
```

Esto se debe a que la estructura de cada post fue pensada con los elementos clásicos que debería tener un post: el *texto* de la publicación, el *usuario* que lo publicó, el *número de ID* que tiene, su *cantidad de likes* e *información* sobre los usuarios que likearon el post.

Entonces, era necesario al momento de comparar dos entradas diferentes, qué post tiene prioridad respecto a otro para cada usuario. Por eso el bloque post tiene calculada la afinidad (desde el momento en que se publica un post y se debe encolar en el heap de los demás usuarios).

Hay que señalar que los posts viven en un **HASH DE POSTS** para poder acceder a ellos de manera constante según su número de ID. Esto se hace para que funciones como *likear un post* y *mostrar likes* tengan una complejidad que dependa solo de la cantidad de usuarios del programa (particularmente, la cantidad de usuarios que likearon un post).

Siendo que la función *mostrar likes* tiene que devolver a todos los usuarios que likearon un post, pero de manera ordenada, la idea fue usar un **ABB DE LIKES**, que comparando según el criterio lexicográfico permite acceder fácilmente a un ordenamiento de ese estilo. Entonces, likear un post accede al post en tiempo constante (por el hash de posts) y mete el nombre del usuario que lo likeó en el abb (contabilizando una complejidad de O(log_U)). Y además, mostrar los likes de un post accede también en O(1) e imprime por pantalla a cada usuario que likeó el post, recorriendo el abb de manera in-order, siendo de complejidad O(U).

Adicionalmente, existe un **HASH DE COMANDOS** para poder verificar un comando y ejecutar su funcionalidad en tiempo constante.