Algogram

Trabajo Practico Nro. 2

Algoritmos y Programación 2

Catedra Buchwald

Segundo cuatrimestre 2021

107789-Gonzalo Ebbes Tenor

107414-Ianniello Rocio Belen

Corrector: Jasmina Sella Faena

Resumen del proyecto

Programa estilo red social, pero reducido donde se puede ingresar con un usuario, hacer publicaciones, dar likes y ver quienes dieron like a cada publicación.

Análisis del proyecto

El primer paso fue plantearnos las estructuras que creíamos útiles, y cumpliesen con la complejidad pedida, decidimos usar un hash tanto para publicaciones como para usuarios, ya que para acceder a la información era la forma más directa, además de que los usuarios no necesitaban ningún orden particular, y los casos de las publicaciones que necesitábamos saber el orden, nos lo resolvía tener que usar un ID incremental. Además de que, tener las publicaciones en un único lugar central nos cerraba con la idea de que los feeds tengan alguna representación de la publicación, más que la publicación en si, que no cerraba tan bien con la idea del heap que venía naturalmente, al ser algo difícil de comparar por su propia cuenta, ya que una publicación no podía tener su ubicación dentro de todos los feeds guardada. En esto fue el representante de la publicación en cada feed, nos redondeó más la idea, un "Identificador" que contiene los únicos dos valores que sirven para saber cuál publicación es más importante, prioridad entre publicante-receptor, y el ID que desempata en caso de que dos publicaciones tengan misma afinidad, además de identificar completamente a la publicación.

Ya saliendo del par Usuarios-Publicaciones que también nos encaminaba bastante el feed, nos queda en si las partes de los likes, asignar un +like iba a ser bastante sencillo por el ID pero nos quedaba mostrar los likeadores cuando nos lo pidan, si hubiese sido por orden de entrada o sin orden, había varias estructuras posibles, pero al ser alfabético nos resultó natural pensar en el ABB + un in-order con la función print.

En cuanto al cálculo de afinidad, en un principio parecía algo que terminaría siendo engorroso o tal vez un poco más feo de armar, un hash que se inicialice en algún momento, para cada usuario con sus afinidades, no nos cerraba y era algo que venía atado al identificador en los feeds. Dejando eso descansar un rato y sentándonos a definir las demás

estructuras, caímos en que cada usuario podía tener su ID que sería su aparición en la lista, y que obtenerlo sería constante, y pensando esos IDs como dos puntos a una distancia del 0, se resolvió rápido.

Programación

Al inicio del programa recibimos un archivo con los nombres de los usuarios, para almacenarlos usamos la estructura hash, que como se especifica en el segmento anterior como dato contiene un struct usuarios, una vez cargados los datos el usuario ya puede ingresar los comandos deseados.

Operaciones

El programa cuenta con las siguientes operaciones que el usuario podrá realizar, a continuación, se detalla el funcionamiento y la justificación de la implementación de cada una:

- **Login**
  - Funcionamiento: inicia la sesión, primero verifica si ya hay un usuario logueado "*`Error: Ya habia un usuario Loggeado.`*", en segundo lugar, verifica si el usuario ingresado existe en el hash, si existe muestra "*Hola* <usuario>", si el usuario no existe muestra el siguiente error "*Error:* <usuario> *no existente.*"
  - Complejidad: verificar si una clave existe en un hash es O(1), como es lo que se procede hacer, *login* es O(1).
  - Parámetro: *login.*

Logout

  - Funcionamiento: cierra la sesión, se verifica si hay un usuario logueado, si es el caso muestra "*Adiós*" y cierra la sesión, sino muestra el siguiente error "*`Error: no habia usuario Loggeado`*".
  - Complejidad: comparar dos string es O(1), que es lo que se procede hacer, por lo cual *logout* es O(1).
  - Parámetro: *logout.*

- **Publicar un post**
    - Lógica: Lo primero que se hace es comprobar que haya usuario loggeado, si lo hace, se crea un post obteniendo la cantidad de elementos publicados, el cual será el ID, y se recorren todos los usuarios ubicando en su feed, una especie de tupla, conformada por ID y afinidad, que se calcula en el momento.
    - Parámetro: *publicar.*

- **Ver próximo post en el feed**
    - Funcionamiento: Lo primero que se hace es comprobar que el feed tenga algún elemento, y si lo hace, se desencola el "mayor" siendo el que presente numero de afinidad mas bajo, y entre aquellos que empaten, el ID mas bajo. Esa "" tupla "" se desencola y se obtiene el post desde hash_publicaciones, desde el cual obtenemos toda la información a imprimir.
    - Parámetro: *ver_siguiente_feed.*

- **Likear un post**
    - Funcionamiento: el usuario luego de ingresar el parámetro, debe ingresar el id del post que desea likear, luego se verifica si hay un usuario logeado, si es el caso contrario se muestra el siguiente error "*Error: Usuario no Loggeado o Post inexistente*", si hay un usuario logueado, se procede a verificar si el post existe en el hash que contienen los post publicados, en caso de existir se muestra " *Post likeado*", en caso contrario se muestra el siguiente error "*Error: Usuario no loggeado o Post inexistente*".
    - Complejidad: tanto que el usuario ingrese un paramatro, verificar si hay un usuario ingresado, verificar si la publicación pertenece al hash publicación, obtener un dato de un hash como verificar si un usuario pertenece al abb likeadores es O(1), pero el guardar un dato en un abb es O(log u)(con u siendo la cantidad de usuarios), siendo

que las operaciones anteriores son de tiempo constante las despreciamos, por ende la complejidad es O(log u).
    - Parámetro: *likear_post.*

- **Mostrar likes**
    - Funcionamiento: para este parámetro no se necesita que un usuario este logeado, primero se ingresa el parámetro por siguiente se ingresa el número del post, se verifica que el id del post ingresado pertenezca al hash de las publicaciones, si no es el caso se mostrara el siguiente mensaje de error "`Error: Post inexistente o sin likes`." en caso de que si exita se verifica que la cantidad de usuarios exixtentes en el abb que guarda quienes diron like al post sea distinto de 0, si es igual nuevamente se mostrara el mensaje de error anterio, si es el caso contrario se procede a mostrar el mensaje "*El post tiene :* <cantidad de likes > *likes:*" por último se muestra los usuarios que likearon el post.
    - Complejidad: tanto que el usuario ingrese un paramatro, hacer un duplicado del id del post, verificar si el ide pertenece al hash de publicaciones, obtener un dato de un hash como verificar si la cantidad del abb es igual a 0 es O(1), pero recorrer el abb de likeadores es O(u) (con u siendo la cantidad de usuarios) por lo cual al ser las operaciones anteriores de tiempo constante se pueden despreciar, siendo asi la complejidad O(u).
    - Parámetro: *mostrar_likes.*

Dificultades

En contra de lo que esperábamos, las mayores dificultades que tuvimos fueron en los errores de memoria en específico con los getline, que en un primer momento manejamos como una variable que se iba sobre escribiendo, pero necesitábamos muchos frees intermedios y terminamos utilizándolo así casi exclusivamente en las opciones, pero optando por utilizarlo solo para captar lo

ingresado por terminal, copiando la variable a memoria estática al instante y liberando. También dificultades en cuanto a cómo adaptar la idea del identificador + heap por el tema de que el heap codeado en clase, era de máximos, y las pruebas estaban hechas así, y pareciéndonos esquivable ese tema, en un primer instante, la solución era que, los identificadores iban a ser enteros positivos asique si uno, solo para la comparación, comparaba su parte negativa, obtendría un heap que priorizaba los valores absolutos más bajos, en su momento pareció un gran eureka, aunque al momento de codear nos dimos cuenta que por más que sea un heap de máximos, si la función de comparación priorizaba mínimos, todo funcionaba perfectamente y sin tanto rebusque.