FACULTAD DE INGENIERIA
Universidad de Buenos Aires

ALGORITMOS Y PROGRAMACIÓN II

TRABAJO PRACTICO N° 2

INTEGRANTES: - Alejo Fábregas N.º 106160
- Celeste de Benedetto N.º 108082

NÚMERO DE GRUPO: Grupo N.º 29

AYUDANTE: Martín Buchwald

Introducción

Al comenzar con este trabajo práctico, lo primero que hicimos fue tomarnos un tiempo para plantear un diseño general del programa, considerando las distintas funcionalidades de la red social, las complejidades temporales requeridas para cada comando, y las estructuras de datos que tenemos disponibles. Además, contemplamos la necesidad de crear nuevos TDAs o estructuras específicamente para este trabajo. De los TDAs vistos en clase, utilizamos Hash, Heap y ABB. Además, creamos tres nuevos TDAs, Algogram, Usuario y Post. En el resto del informe describiremos en más detalle por qué decidimos esto, y como lo implementamos.

Diseño general

En este proceso debatimos cómo debería ser el programa y elaboramos un borrador de las estructuras y funcionalidades que necesitamos.

Lo primero que diseñamos e implementamos fue un main del programa, que se encuentra en el archivo *tp2.c*. En este main validamos el archivo de usuarios pasado por parámetro, lo procesamos para obtener los usuarios de la red social y luego el programa se queda a la espera que el usuario ingrese los comandos para ejecutarlos.

Una vez hecho esto, el siguiente paso fue implementar los comandos de la red social. Para esto, decidimos diseñar e implementar un nuevo TDA llamado AlgoGram, con sus respectivos archivos *algogram.h* y *algogram.c*. Este nuevo TDA contiene las estructuras de datos necesarias para el funcionamiento de la red social, como los usuarios y los posts. Su comportamiento son las funcionalidades principales de la red social: *login*, *logout*, *publicar post*, *ver post*, *likear post*, *mostrar likes*, etc.

Para decidir que estructuras de datos formarían parte de este TDA, pensamos en los datos que necesitamos que contenga (los usuarios y los posts de la red social), las primitivas que le darán su comportamiento y la complejidad exigida para cada una de ellas. En primera instancia decidimos que tendría un hash de usuarios (los nombres de usuario como clave y como dato sus feeds) para acceder a ellos en O(1), una string correspondiente al usuario_loggeado para verificar efectivamente que haya un usuario loggeado y poder acceder a él fácilmente (ver si un usuario está loggeado es un chequeo O(1)), y un contador de los ID de los posts, que se les asigna a medida que se publican. Esta implementación se modificó a medida que avanzamos con el trabajo práctico.

Luego, para modelar el feed de los usuarios decidimos utilizar el TDA Heap. A cada usuario en el hash de usuarios le corresponde (como dato en el hash) un heap que represente su feed. Este heap contiene los posts del feed de cada usuario (que tiene los posts de todos menos los de sí mismo) ordenados según la afinidad. Para obtener un post simplemente se desencola del heap, y de esa manera el post ya visualizado no se encuentra más en el feed del usuario. Con esta implementación publicar un post sería iterar el hash de usuarios, y en el heap de cada uno de ellos encolar el post, por lo que es O(u log(p)). Ver el proximo post del feed seria ir al heap del usuario en el hash (O(1)) y desencolar el post con mayor afinidad, por lo que es O(log (p)).

Respecto a los likes de los posts decidimos que AlgoGram tenga un nuevo hash de posts, que tiene como clave el ID de los posts y como dato al post correspondiente a

ese ID. Esto fue motivado por las funcionalidades *likear_post* y *mostrar_likes*, ya que se necesita acceder en O(1) a los posts por medio de su ID, para cumplir con la complejidad requerida de estas funcionalidades. Para los likes en sí mismos, pensamos en que los posts tengan un ABB de likes, es decir un ABB que contenga a los usuarios que le dieron like a ese post, con una función de comparación que tenga en cuenta el orden alfabético. De esta manera, si hacemos un recorrido in order por el ABB nos va a dar todos los usuarios que le dieron like a ese post en orden alfabético. Para likear se busca en el hash de posts con el ID del post en O(1), y se guarda el usuario que likeó en O(log (u)) en el ABB de likes de ese post, ya que en el peor de los casos todos los usuarios likearon ese post. Para ver los likes de un post, recorremos en in order el ABB, y como vemos todos sus elementos va a ser O(u).

En las etapas iniciales del diseño de este programa pensábamos que los usuarios y los posts serían simples structs que contengan los datos necesarios (los heaps que representan a los feeds y las strings de los posts con su ID, por ejemplo), pero luego advertimos que teníamos que diseñar nuevos TDAs, ya que ellos tienen su propio comportamiento, no son simples contenedores de información.

Desarrollo

En esta etapa, implementamos las estructuras que diseñamos anteriormente, con todas sus primitivas y funciones auxiliares. A medida que avanzamos, nos dimos cuenta que algunos cambios eran necesarios con respecto a lo que habíamos diseñado.

Luego de intentar implementar las primitivas de AlgoGram que utilizan a los usuarios y sus posts, nos dimos cuenta que necesitábamos crear nuevos TDA Post y TDA Usuario, ya que estos tenían su propio comportamiento, y necesitábamos primitivas para modificarlos y utilizarlos, sin acceder a su estructura interna.

El nuevo TDA Post contiene el ID de dicho post, el nombre del usuario que lo posteó, el texto del post y un ABB que contiene a las personas que le dieron like. Con las primitivas se puede ver el ID del post, el nombre del posteador, el contenido del post, se puede likear ese post, o ver quiénes lo likearon y cuántos son. Además, puede imprimir los usuarios que le dieron like a ese post en orden alfabético.

El nuevo TDA Usuario contiene el nombre de dicho usuario, su ID y un heap que representa su feed. Con sus primitivas se pueden ver y guardar posts en el feed del usuario, y se puede ver el ID que le corresponde.

Luego de haber creado estos TDAs, la implementación se hizo más amena, simplemente tuvimos que ajustar el código de las primitivas de AlgoGram para que utilicen correctamente a los nuevos TDAs. Por ejemplo, el hash de usuarios ahora tiene como dato a Usuarios en lugar de sólo tener un heap, y el hash de posts contiene Posts.

A continuación se detalla el funcionamiento del programa luego del desarrollo.

Resultado final

Al iniciar el programa se realiza la lectura del archivo de usuarios que se espera recibir como parámetro – en caso contrario notifica que ocurrió un error –. Una vez abierto, los usuarios que se encuentran en él son guardados en la estructura hash de usuarios de AlgoGram. Luego, el programa se queda esperando a que se ingrese por consola alguno de los comandos explicados a continuación.

Comandos:

Login y logout en O(1) → Primero verificamos que haya un usuario loggeado en AlgoGram, lo que es chequear que la variable *usuario_loggeado* no sea NULL. Luego gracias al hash de usuarios que tenemos en nuestro TDA AlgoGram, podemos saber si el usuario que se quiere loggear es válido chequeando que pertenezca a él, y como ya sabemos, todas las operaciones del hash son O(1). Logout realiza todas operaciones O(1), entre ellas reestablece nuestra variable de *usuario_loggeado* en NULL para que AlgoGram sepa que ya no hay más un usuario loggeado.

Publicar post en O(u log(p)) → Siempre y cuando haya usuario loggeado, éste va a poder publicar un post. Para este comando, obtenemos el texto y creamos un Post, luego recorremos nuestro hash de usuarios y mientras el usuario del hash no sea aquel que lo publicó, utilizamos la primitiva de Usuario *usuario_guardar_feed* para agregar el posteo al feed de cada usuario. Esta primitiva encola en el heap que representa el feed el Post, cuya complejidad es O(log p) siendo p un post, ya que solamente se encola en el heap. Esto se realiza por cada usuario (u veces), por lo que publicar sería O(u(log p)). Además, agregamos el post a nuestro hash de posts en Algogram, operación que cuesta O(1) por ser un hash. Así finalmente logramos la complejidad esperada.

Ver próximo post en el feed en O(log(p)) → La primitiva *ver_post* se encarga de este comando. Primero se busca el usuario loggeado (se requiere que haya uno) en el hash de usuarios. Luego, la primitiva *usuario_ver_post* nos devuelve el post que sigue en el feed de este usuario según su ***afinidad***. Ésta primitiva realiza la operación en O(log (p)), ya que simplemente desencola el post del heap. Con primitivas del Post, muestra su id, sus likes, y el texto publicado.

- ***Afinidad:*** Realizar la función de comparación nos resultó complicado. Finalmente decidimos crear un struct *post_afinidad* en el TDA Usuario que nos sirva como contenedor del post y una variable que representa la afinidad que tiene. Esto no es un TDA ya que el struct es simplemente un contenedor de información que se utiliza solamente en el TDA Usuario, no tiene comportamiento propio. El cálculo de la afinidad lo realizamos al publicar el post, calculando la distancia que hay entre el ID del usuario loggeado, al cual le pertenece el feed, y el ID del usuario que publicó el post.

  afinidad = | id_usuario_loggeado – id_posteador |

  Este struct *post_afinidad* es lo que encolamos en el heap. La función de comparación afinidad accede a la afinidad de cada *post_afinidad* y las compara para determinar el post con menor afinidad, es decir, el mayor de los dos posts.

Likear post en O(log u) → Para likear un post, es necesario estar loggeado. Se obtiene qué usuario quiere likear, recibimos por entrada el ID del post a likear y lo buscamos en nuestro hash de posts en O(1). Una vez obtenido el post y mientras el usuario no lo haya likeado previamente, se llama a *post_likear* que guarda en el ABB de likes del post el usuario que likeó. Esta operación es O(log u) siendo u la cantidad de usuarios, ya que en el peor de los casos todos likearon el post.

Mostrar likes en O(u) → No es necesario estar loggeado para esta operación, sólo precisa recibir por pantalla el ID del post, luego lo busca en el hash de posts en O(1), y una vez que lo obtuvo, llama a la primitiva *post_ver_likes*. Esta primitiva imprime *post_cantidad_likes*, que es la cantidad de elementos del ABB, y hace un recorrido in order de dicho ABB, para imprimir los usuarios que likearon en orden alfabético. Como se recorre todo el ABB y solo se imprimen los usuarios (operación O(1)), la complejidad de mostrar todos los que likearon es O(u), si todos los usuarios likearon el post.

Conclusión

Para realizar este TP nos ayudó mucho pensar y diseñar las estructuras de datos y funciones que necesitamos para hacer el programa, antes de codearlo. A pesar de que nos tomamos varios días para el diseño, en el momento de la implementación descubrimos que algunas cosas se nos pasaron por alto, ya sea porque no sabíamos como seguir, o porque teníamos errores de memoria, y tuvimos que sentarnos nuevamente a pensar una solución antes de seguir con el código. Esto nos muestra lo importante que es pensar antes de codear, sobre todo para proyectos de grande escala.