Informe Trabajo Práctico 2

Integrantes:

Bermúdez Nuñez, Juan Pablo. Padrón: 107405

Lazarte, Ezequiel. Padrón: 108063

Ayudante:

Jorge Collinet

Estructuras utilizadas

Basados en el funcionamiento esperado y en las complejidades requeridas para los comandos del programa AlgoGram, decidimos utilizar las siguientes estructuras:

- Heap
- Hash
- ABB
- Vector Dinámico
- Lista enlazada
- Post

TDA POST

¿Por qué un TDA POST? Debido a los comportamientos que presentaba el programa a implementar como *publicar*, *dar like*, *ver likes*, *ver publicaciones*, y observando que la publicación en si tiene bastante información asociada, fue que vimos idóneo el implementar un TDA POST, el cual se podría usar para futuras redes sociales o cualquier otro programa que requiera usar publicaciones.

¿Qué es el TDA POST? Dicha estructura sirve para crear un post a partir de un texto, ID asociado y un autor. Además, mantiene la cantidad de likes y quien da el like según los vaya recibiendo. Para poder mostrar a las personas/usuarios que le hayan dado like al post ordenados alfabéticamente y manteniendo este orden a medida que más personas "likean el post" es que se decidió almacenar los nombres de estos en un ABB, donde la prioridad es por orden alfabético, de esta forma se mantendrán ordenados cada vez que se agrega uno nuevo y a la hora de recorrer el ABB, con un recorrido *inorder*, lo hará por orden alfabético.

Primitivas y complejidades de estas:

- post_crear: O(1), asigna los parámetros correspondientes.

- post_destruir: en O(N), con N la cantidad de nombres en el ABB.

- post_agregar_like: en O(log N), ya que agrega al ABB un nombre y después suma uno a la cantidad de likes, esto último en O(1).

- post_obtener_likes: en O(N), ya que recorre por completo al ABB y al mismo tiempo va agregando, en O(1), los nombres al final de una Lista enlazada, creada previamente, para luego devolverla. Notar que esta lista tendrá los nombres ordenados alfabéticamente.

- post_obtener_texto: en O(1), devuelve el parámetro correspondiente.
- post_obtener_autor: en O(1), devuelve el parámetro correspondiente.
- post_obtener_id: en O(1), devuelve el parámetro correspondiente.
- post_cant_likes: en O(1), devuelve el parámetro correspondiente.

Comandos del programa

A continuación, se explicarán los algoritmos utilizados y las consideraciones tomadas para implementar los comandos de AlgoGram.

1. LOGIN:
   Al comienzo del programa se recibe un archivo con los correspondientes usuarios, el cual recorremos y, por cada usuario, vamos agregando a un HASH que tiene como clave el nombre del usuario, y como dato una estructura que contiene información de los mismos. Decidimos utilizar un HASH para almacenar y obtener en O(1) la información asociada a los usuarios.
   Para el login del usuario hacemos las verificaciones correspondientes en O(1), con el HASH mencionado anteriormente, y luego actualizamos el usuario actual del programa.

2. LOGOUT:
   Verificamos que haya un usuario logeado y, en caso de haberlo, actualizamos el usuario actual.

3. PUBLICAR POST:
   Al publicar un post primero se crea un TDA POST, para luego almacenarlo en un Vector Dinámico que contiene todas las publicaciones creadas hasta el momento.
   Debido a que las publicaciones tienen una importancia distinta para cada usuario, decidimos agregar a cada uno (dentro del HASH de usuarios) un HEAP que corresponde al feed, donde se guarda la afinidad con el autor del post y el id del post. Se priorizan los posts de los usuarios con los que tenga mayor afinidad.
   Utilizamos un Vector Dinámico para almacenar los TDA POST porqué se puede acceder a cualquier elemento en O(1), siempre que se sepa la posición (al agregar los post siempre al final del Vector a medida que se crean, además de ser en O(1),

nos aseguramos que los id's de estos se correspondan con su posición dentro del vector).

Utilizamos un HEAP porque, al publicar un post, "añadimos la publicación" al feed de cada usuario, y con esta estructura implicaría encolar los datos correspondientes en cada feed, lo cual tiene una complejidad de O(U * log P), siendo P la cantidad de publicaciones creadas hasta el momento y U la cantidad de usuarios, ya que encolar un elemento en un HEAP es O(log P) y se realiza para cada usuario.

4. VER PROXIMO POST EN EL FEED:

   Para obtener el próximo post que le corresponde ver al usuario actual, basta con buscar el feed asociado al mismo, dentro del hash de usuarios. Desencolamos del feed para obtener el id del post a mostrar y luego, con este id, obtenemos el post que se encuentra en el Vector. Tras haber obtenido el post, se lo muestra al usuario. Primero obtener el feed del usuario actual en el HASH de usuarios es O(1), luego desencolar del feed tiene una complejidad de O(log P), siendo P la cantidad de publicaciones creadas hasta el momento y obtener un elemento del Vector, conociendo su posición, es O(1). Con lo cual la complejidad del comando es O(log P).

5. LIKEAR UN POST:

   Con un id recibido por el usuario actual, buscamos el post dentro del Vector de publicaciones y le agregamos un like a dicho post, lo cual tiene una complejidad O(log U), siendo U la cantidad de usuarios que le dieron like al post.

6. MOSTRAR LIKES:

   Con un id recibido, buscamos el post dentro del Vector de publicaciones, luego utilizando la primitiva *post_obtener_likes* del TDA-POST, que devuelve una lista ordenada alfabéticamente según el nombre de los usuarios que le dieron like, mostramos los nombres de esta lista.

   Tanto la primitiva *post_obetener_likes,* como mostrar los nombres de la lista, es O(U). Por lo tanto, la complejidad del comando es O(U), siendo U la cantidad de usuarios que le dieron like al post.