Trabajo Práctico N°2

Algoritmos y programación II: Buchwald

2C 2021

Grupo 41:

- Joaquin Rivero (padrón 106032).
- Santiago Langer (padrón 107912).

Correctora:

- Jasmina Sella Faena

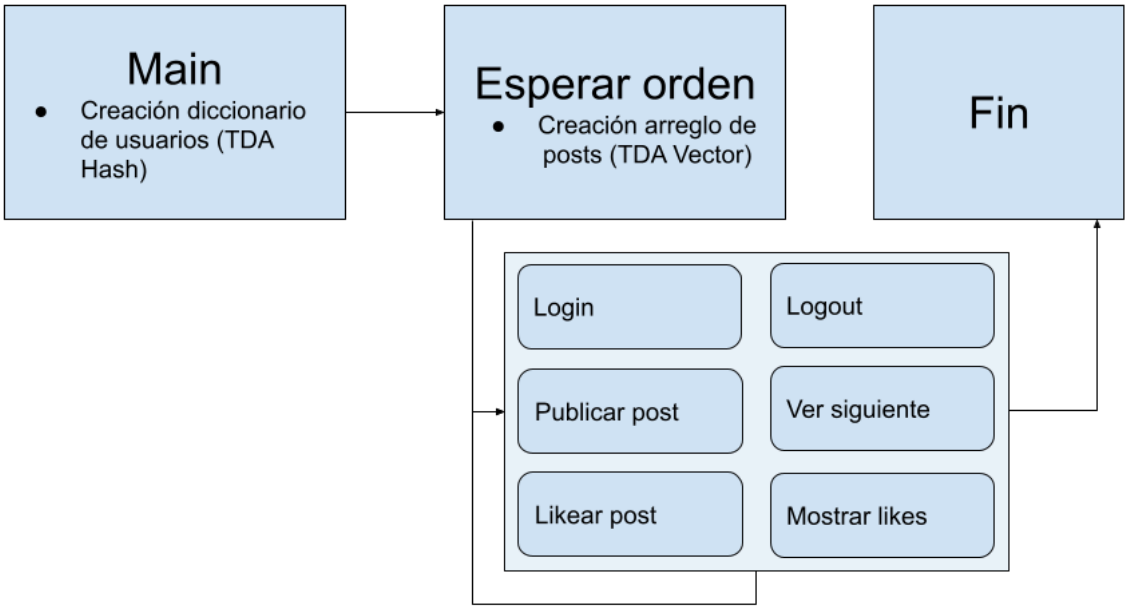
Facultad de Ingeniería, Universidad de Buenos Aires.

Buenos Aires, Argentina.

Algogram es una red social donde los usuarios pueden comunicarse mediante posts, teniendo una experiencia personalizada para cada uno: el feed de cada usuario muestra primero los posts de los usuarios con los que se tiene una relación más cercana. En este informe haremos una explicación de la implementación de esta red social.

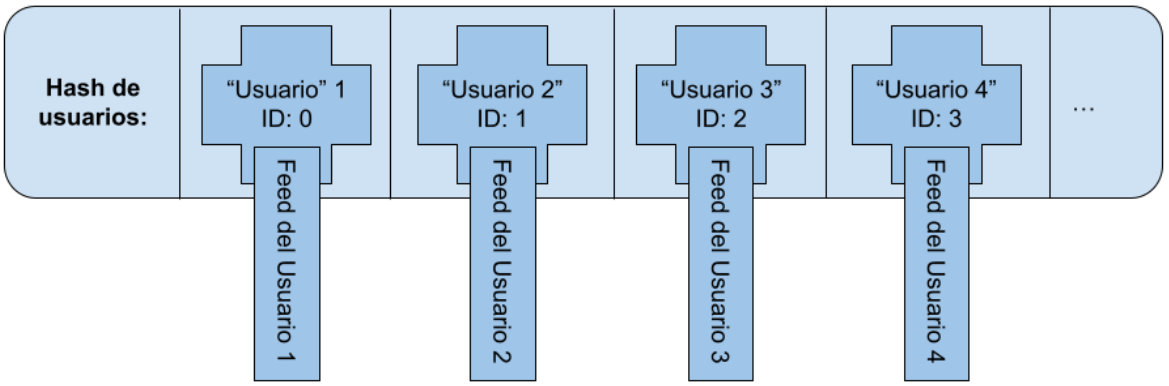
La red social debía tener dos componentes elementales, usuarios y posts. Cada uno con su propias características: Los usuarios debían tener feeds propias según su afinidad con otros usuarios, y los posts debían retener contenido y recibir likes. Enfrentamos estos problemas creando una TDA para los usuarios (con un TDA Heap para el feed), y una TDA para los posts(con un TDA ABB para los likes).

Antes de comenzar a programarlo, nuestro equipo se tomó un tiempo para diseñar el proyecto debido a que el Trabajo Práctico *Algogram* presentaba un desafío principal: la baja complejidad que debían cumplir los comandos *"Ver próximo post"* y *"Likear un post"*.
Ver próximo post debía ser $O(log(P))$ y *likear un post* debía ser $O(log(U))$. Además, el resto de operaciones también tenían un máximo de complejidad posible en su implementación (véase Anexo para conocer la complejidad de cada comando).
Frente a estas necesidades, decidimos aprovechar al máximo las exigencias de complejidad de ciertos comandos cargándolas con más operaciones, mientras simplificamos qué debían hacer los comandos que menos carga podían permitirse.



A continuación se explicarán las decisiones tomadas para cada una de las seis operaciones que debía cumplir la red social:

- *Login*: Entrar a la cuenta del usuario debía ser $O(1)$, por lo que decidimos utilizar un diccionario (implementado con una función de hashing) para guardar todos los usuarios creados en la red social. Para maximizar la velocidad de todas las operaciones, creamos el diccionario y guardamos todos los usuarios antes de comenzar a solicitarle comandos al usuario.

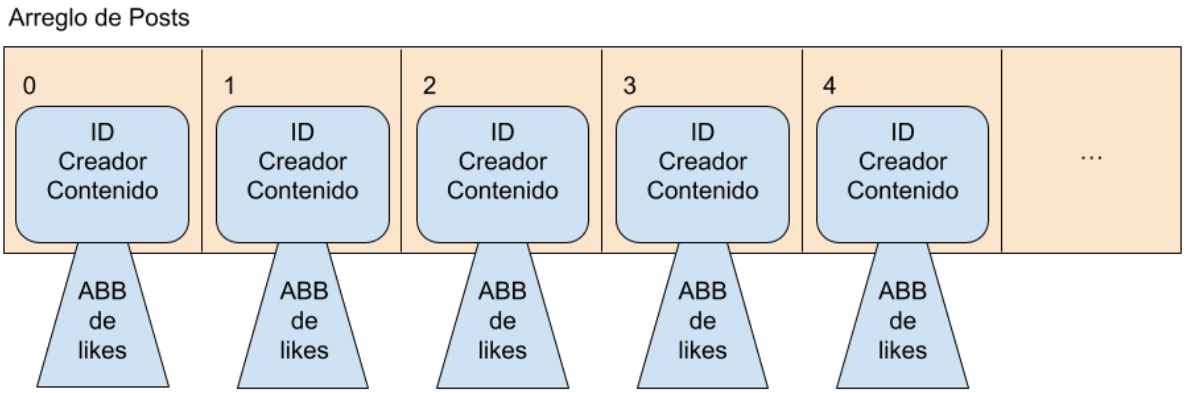


- Con el diccionario ya previamente creado, loguearse era $O(1)$: revisaba que el usuario solicitado estuviese guardado en el hash y luego lo guardaba en una variable con el único usuario logueado en el programa.

- *Logout*: La función logout también debía cumplir con velocidad constante, por lo que simplemente revisa si había algún usuario logueado en el momento, y vacía la variable que retiene al único usuario logueado.

- *Ver próximo post*: Las decisiones tomadas por nuestro equipo giraron alrededor de las necesidades de este comando, el más exigente de todos. Como debíamos retornar posts en un orden determinado y con complejidad $O(log(P))$, optamos por utilizar una cola de prioridad (mediante un TDA Heap) para el feed de cada usuario, que desencola en orden en $O(log(P))$ tiempo.
  A cada post, cuando es creado y publicado en la función *Publicar Post* le asignamos un número de prioridad según la afinidad del usuario que lo publicaba y el usuario que lo recibía en su feed. Así, mientras el usuario logueado mira su feed, le aparecen primero los posts de sus usuarios preferidos.

- *Publicar un post*: Publicar era la función que más margen de diseño nos daba en cuanto a orden de complejidad. Por esto, decidimos utilizarla como el comando que más operaciones realizase, aligerando la carga sobre el resto.
  La función, primero, crea un post y lo agrega a un arreglo de posts (implementado con un TDA Vector) previamente creado. Luego, para cada usuario, encolamos el puntero a ese post en la cola de prioridad de cada usuario. Así, toda la función lograba cumplir con la exigencia de orden de complejidad $O(U * log(P))$.



- *Likear un post*: Permitir que un usuario likease un post podría haber sido $O(1)$ de no ser por una importante necesidad de la función *mostrar_likes*: los usuarios que likearon debían aparecer en orden alfabético. Por esta razón, elegimos un TDA que se pudiese leer en orden y que guardase sus elementos en el orden que necesitamos. Un ABB y un Heap cumplían con estas dos necesidades, pero decantamos por el primero ya que con el segundo se perdían los usuarios que habían likeado luego de su primera lectura, mientras que el ABB guarda el próximo usuario que quisiese ver los likes de un mismo post ya visto por otro usuario, podría hacerlo.

- *Mostrar likes*: Reiterando y ampliando sobre lo ya explicado en “Likear un post”, mostrar likes debía retornar una lista de usuarios que habían likeado (teniendo en cuenta su exigencia de complejidad $O(U)$, pero su segunda exigencia de que fuese en orden y legible múltiples veces nos llevó a utilizar el ABB, que al ser leído in-order, retorna sus elementos alfabéticamente (como lo requerían las exigencias de red social).

Durante la implementación, encontramos una serie de dificultades en nuestro diseño inicial o alternativas para mejorarlo. Por ejemplo, encontramos que era mucho más organizado compartimentar la creación de usuarios y posts en TDA nuevos. Por esta razón, creamos los TDA Usuario y el TDA Post, con sus primitivas correspondientes.

Habiendo sido explicadas las partes, a continuación se hará un corto acompañamiento de ejemplo de la ejecución del programa:
Primero, se ejecuta el programa. Se abre el archivo recibido por parámetro y se crean todos los usuarios, que son guardados en el diccionario de usuarios. También se crea el arreglo de posts, donde serán guardados todos los posts creados durante la ejecución del programa.
Luego, el programa espera a recibir comandos hasta que se pide cerrar el programa. La red Algogram recibe un comando, por ejemplo login, y ejecuta su función. La función logout recibe el nombre del usuario que desea conectarse, revisa si existe en el diccionario de usuarios y lo guarda a la variable de usuario actualmente conectado. Posteriormente, por ejemplo, el usuario escribe la orden “Publicar” y se ejecuta la función. La función *publicar post* pide el contenido del post, y una vez recibido crea el post y lo guarda junto a su

prioridad en cada feed de todos los usuarios del programa. La prioridad varía según en qué feed se está guardando el nuevo post.
Luego de publicar, ese mismo usuario decide likear su propio post, así que ingresa el comando *"*likear post" y se llama a la función correspondiente. La función solicita el número ID de la función a likear, accede a su posición en el arreglo de posts y suma el usuario likeante al ABB de likes del post.
El usuario, finalmente, cierra su sesión llamando a *logout*. La función *logout* vacía la variable del usuario actualmente conectado.

ANEXO

- Complejidades:
  - Login: $O(1)$
  - Logout: $O(1)$
  - Publicar post: $O(U * log(P))$
  - Ver próximo post: $O(log(P))$
  - Likear un post: $O(log(U))$
  - Mostrar likes: $O(U)$

- Referencias:
  - U = cantidad de usuarios
  - P = cantidad de posts