FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico N°2

AlgoGram

2° CUATRIMESTRE 2021

GRUPO N° G15

INTEGRANTES:

Ian Klaus von der Heyde 107638
Valentín Schneider 107964

Implementamos 3 TDA

❑ TDA Post:

```
struct post {
    char* creador;
    size_t id_creador;
    size_t id;
    char* contenido;
    size_t likes;
    hash_t* usuarios_like;
};
```

cada post tiene un id (único), el id y nombre del usuario que lo creó, su contenido y la cantidad de likes que recibió. Además de lo anterior cuenta con un hash de nombres de usuarios para guardar información acerca de las cuentas que le dieron like.

Primitivas:

- post_crear
- post_destruir
- post_likear
- post_ver_id
- post_ver_cant_likes
- post_ver_contenido
- post_ver_creador
- post_ver_id_creador
- post_dio_like

❑ TDA Usuario:

```
struct post_feed {
    size_t id_duenio_del_heap;
    post_t* post;
};

struct usuario {
    size_t id;
    char* nombre;
    heap_t* feed;
};
```

cada usuario cuenta con un id (único), un nombre y un heap (único para cada usuario), utilizado para moderar el orden de visualización de los post pendientes a ver de ese usuario. Ese heap guarda estructuras simples llamadas post_feed, que a su vez contienen el id del usuario al cual pertenece ese heap y el post en sí.

Primitivas:

- usuario_crear:
- usuario_destruir:
- usuario_agregar_feed:
- usuario_ver_siguiente_feed:
- usuario_ver_nombre:
- usuario_ver_id:

❑ TDA Algogram:

Complejidad de los comandos:

```c
struct algogram {
    usuario_t* usuario_loggeado;
    size_t cant_post;
    arreglo_t* posts;
    hash_t* usuarios;
    void** nombres_usuarios;
};
```

```c
struct post_feed {
    size_t id_duenio_del_heap;
    post_t* post;
};
struct usuario {
    size_t id;
    char* nombre;
    heap_t* feed;
};
```


```c
struct algogram {
    usuario_t* usuario_loggeado;
    size_t cant_post;
    arreglo_t* posts;
    hash_t* usuarios;
    void** nombres_usuarios;
};
```


Siendo ***u*** la cantidad total de usuarios y ***p*** la cantidad total de post:

- Para la estructura de algogram utilizamos un hash con el fin de que loggearse sea O(1) ya que lo único que hace la primitiva ***login*** es pedir una entrada al usuario O(1), verificar que efectivamente ese usuario exista y lo carga como usuario loggeado ***O(1)***. Para cargar el usuario loggeado simplemente nos guardamos la referencia a ese usuario.

- Por su parte, ***logout*** también es ***O(1)*** ya que lo único que hace es borrar la referencia a ese usuario.

- Para publicar un post pedimos entrada y lo creamos, el costo de la operación esta en cargar ese post en el feed de los otros usuarios. Para hacer eso iteramos por todos (O(u)), guardamos el post en el feed de cada uno, que nos cuesta O(log (p)) porque utilizamos un heap como feed. Entonces ***publicar*** nos queda ***O(u * log (p)).***

- ***Ver siguiente feed*** solamente cuesta desencolar del heap feed que mencionamos arriba, asi que cuesta ***O(log p)*** (utilizamos un heap para poder mostrar las publicaciones en el orden pedido).

- ***Para likear un post***, simplemente guardamos en el hash de usuarios_like el nombre del usuario que le dio like. Por esto nos queda ***O(1)***. En este caso no utilizamos un heap porque al hacer esto nos quedaba muy alta la complejidad de mostrar likes. Y con esta forma es mas complejo el inicio del programa pero mas eficiente likear cada post. (Nos pareció un mejor trade off que tarde mas en iniciar pero una vez iniciado es mucho mas eficiente)
- ***Para mostrar likes***: iteramos por todos los usuarios (ordenados alfabéticamente) y solo mostramos aquellos usuarios que hayan dado like al post. Por lo que nos quedó ***O(u)***

Estructuras: