(95.12) ALGORITMOS Y PROGRAMACIÓN II

TRABAJO PRÁCTICO N°2: AlgoGram

1° cuatrimestre 2022

Fecha de entrega: 17/06/2022

Corrector asignado: Santiago Bellido

Grupo: G16

Alumnos:

- **MAMANI, Matías Hernán - 94765 -** mmamani@fi.uba.ar
- **MANUEL, Pablo Eluney - 94962 -** pmanuel@fi.uba.ar

OBJETIVO

Implementar en lenguaje C una red social denominada “AlgoGram” que cuente con las siguientes funcionalidades:

- Login y logout de usuarios que se encuentren en un archivo en formato de texto plano (.txt).
- Cada usuario debe ser capaz de publicar un post y ver los posts publicados anteriormente por otros usuarios (feed), mostrándolos según cierto grado de afinidad definido por la cercanía que exista entre el usuario que lo publicó y el usuario logueado en el archivo de usuarios.
- Likear un post indicado mediante su identificador ID.
- Mostrar la cantidad de likes y que usuarios se lo dieron a un post también mediante su identificador ID.

Estas funcionalidades deben cumplir ciertos requerimientos de complejidad algorítmica indicados en el enunciado.
La red social debe ser implementada utilizando distintos tipos de datos abstractos estudiados en la materia y creando nuevos tipos de datos a partir de ellos si así se requiriese.

ANÁLISIS

En base a las complejidades pedidas para cada comando se realizó el análisis de las estructuras de datos que serían idóneas para cada uno.

Para el *login* y *logout* al pedirse con complejidad *O(1)* es necesario utilizar una tabla de *hash* donde podamos contar con todos los usuarios y al intentar establecer sesión con uno validar si este existe o no.
Publicar Post se pide con complejidad *O(u log(p))* siendo *“u”* la cantidad de usuarios y *“p”* la cantidad de posts creados, por lo tanto en este caso dado que los posts deben tener orden de prioridad por afinidad entre usuarios, se optó por utilizar una estructura *heap* ya que encolar los posts en esta estructura es de complejidad *O(log(p))*. El complemento de complejidad faltante *O(u)* indica que se debe iterar esta operación de encolar en todos los usuarios diferentes del que publicó.
El comando *Ver Próximo Post* debe tener una complejidad temporal *O(log(p))*, la de *Likear Post* debe ser *O(log(u))*, y la de *Mostrar Likes* debe ser *O(u)*. En vista de estos requerimientos de complejidad, se decidió utilizar una tabla de *hash* para almacenar los posts por ID donde los datos serán el texto de la publicación, la cantidad de likes, el usuario que publicó y *árboles binarios de búsqueda* con un iterador interno *in-order* para rescatar los datos de los usuarios que dieron *like* a una publicación de forma ordenada alfabéticamente.

Habiendo descrito este análisis, se optó por implementar tres *Tipo de Datos Abstracto (TDAs)* que nos facilitaría el desarrollo del presente trabajo práctico:

- ***TDA usuarios***
- ***TDA sesion***
- ***TDA posts***

DISEÑO

A continuación se desarrolla el diseño de los tres TDAs implementados.

1) *TDA usuarios:*

El *TDA usuarios* se encargará de almacenar todos los usuarios que existen en la red social (a partir de un archivo *.txt*) en un estructura *hash* dado que tenemos que tener acceso a cada usuario de complejidad *O(1)* para el login.
Además, cuando un usuario publique un post guardará en cada uno de los otros usuarios el ID del post en un estructura *heap* con prioridad en base a la afinidad entre sendos usuarios correspondiendo a una complejidad *O(u log(p))*. De este modo, cuando un usuario quiera ver el siguiente post en su *feed* solamente será necesario desencolar el ID del post del *heap* cumpliendo con la complejidad *O(log(p))* pedia. Para lograr todo esto, en el dato del *hash* descrito anteriormente tendremos un *struct* con la posición relativa del usuario en el archivo y un puntero a una estructura *heap* que en cada posición almacenará otro *struct* con el ID del post y el grado de afinidad entre usuarios.
El siguiente diagrama muestra la estructura del *TDA usuarios* partiendo del *typedef struct usuarios usuarios_t*.

```
typedef struct {
        hash_t *hash;
} usuarios_t;
```

Tabla *hash*

CLAVE	**DATO**
“chorch”	dato_hash_t* dato_hash;
“mondi”	dato_hash_t* dato_hash;
…	…

```
typedef struct {
        size_t pos;
        heap_t *heap;
} dato_hash_t;
```

heap

dato_heap_t *dato_heap	…	…	…	…	…	…	…

```
typedef struct {
        ssize_t id;
        size_t afinidad;
} dato_heap_t;
```

Este TDA cuenta con las siguientes primitivas:

- ***usuarios_t *usuarios_crear(void)***: crea y retorna un *TDA usuarios*. Devuelve *NULL* en caso de error.
- ***bool usuarios_guardar(usuarios_t *usuarios, const char *usuario, size_t pos)***: guarda un usuario y su posición relativa en el TDA. En caso de lograrse la operación exitosamente devuelve *true* o *false* en otro caso.
- ***bool usuarios_pertenece(usuarios_t *usuarios, const char *usuario)***: valida si el usuario existe en el TDA, retorna *true* en caso afirmativo y *false* en otro caso.
- ***bool usuarios_publicar(usuarios_t *usuarios, ssize_t id, const char *usuario)***: guarda el ID del post publicado por un usuario en los demás usuarios con orden de prioridad en base a la afinidad entre ambos. Retorna *true* en caso de lograrlo exitosamente, y *false* en otro caso.
- ***ssize_t usuarios_ver_sig_feed(usuarios_t *usuarios, const char *usuario)***: retorna el ID del post siguiente para ver en el *feed* de un usuario provisto por parámetro. En caso de error o de que no haya más posts para mostrar en el *feed* retorna -1.
- ***void usuarios_destruir(usuarios_t *usuarios)***: destruye el *TDA usuarios*.

2) *TDA sesion:*

El *TDA sesion* guardará la sesión activa en la red social. Para ello, consta simplemente de un *typedef struct sesion sesion_t* que contiene el string del usuario activo y *NULL* en caso de que no haya usuario loggeado. Es importante notar que todas las primitivas de este TDA tienen complejidad *O(1)* por lo que utilizarlo no afecta la complejidad del programa.
El *TDA sesion* cuenta con las siguientes primitivas:

- ***sesion_t *sesion_crear(void)***: crea y retorna un *TDA sesion*. Devuelve *NULL* en caso de error.
- ***bool sesion_login(sesion_t *sesion, const char *usuario)***: guarda en el TDA el usuario provisto por parámetro estableciendo la sesión del mismo. Retorna *true* en caso exitoso y *false* en otro caso.
- ***void sesion_logout(sesion_t *sesion)***: remueve la sesión activa.
- ***bool sesion_esta_loggeado(sesion_t *sesion)***: valida si hay sesión activa alguna. Retorna *true* en caso afirmativo y *false* en otro caso.
- ***char* sesion_obtener_usuario(sesion_t *sesion)***: obtiene y devuelve el usuario de la sesión activa. En caso de no haber, devuelve *NULL*.
- ***void sesion_destruir(sesion_t *sesion)***: destruye el *TDA sesion*.

3) *TDA posts:*

Las funcionalidades de publicar, likear y ver posteos se encuentran estrechamente relacionada entre sí, por lo que se decidió implementar un tipo de dato abstracto denominado *posts* cuyas primitivas permitieran la implementación de estas funcionalidades y mantener de forma diferenciada todo lo concerniente a un posteo con el fin de permitir un debugueo más rápido en caso de errores, y facilitar su corrección y el agregado de código adicional si fuera necesario.

Se decidió crear un tipo de dato *struct* denominado *post_t* que contiene un generador de IDs que permite asignar estos a los posts que se vayan creando y un puntero a una tabla *hash*. Dicha tabla *hash* consta de una clave que será el ID del post y un dato que será un *struct dato_hash_t* que contiene información de cada posteo, como ser el texto del post en formato string, la cantidad de likes del mismo, el usuario que lo publicó y un puntero a un *ABB* que a su vez contendrá los nombres de los usuarios que dieron *like*.

Este TDA consta de las siguientes primitivas:

- ***posts_t* posts_crear(void)***: crea y retorna un *TDA posts*. Devuelve *NULL* en caso de error.
- ***bool posts_pertenece(posts_t* posts, const char* clave_id)***: indica si el ID corresponde a un post publicado. Retorna *true* en caso afirmativo, *false* en otro caso.
- ***ssize_t posts_publicar(posts_t* posts, const char* texto, const char * usuario)***: asigna un ID a un post y lo inserta en la tabla *hash*, guarda como datos el texto del post y el usuario que lo publicó. Devuelve el ID del post generado, o -1 en caso de error.
- ***bool posts_ver_siguiente_feed(posts_t* posts, ssize_t id, char** usuario, size_t* cant_likes, char** texto)***: al proveer un ID de un post devuelve por parámetro el usuario que lo publicó, el texto y la cantidad de likes. El valor de retorno indica *true* si la operación fue exitosa o *false* en otro caso.
- ***bool posts_likear(posts_t* posts, const char *clave_id, const char* usuario)***: agrega un like a un post indicado mediante su ID y guarda en el *árbol binario de búsqueda* que usuario realizó dicha acción. Retorna *true* en caso de éxito o *false* en otro caso.
- ***lista_t* posts_mostrar_likes(posts_t* posts, const char *clave_id, size_t* cant_likes)***: a partir del ID de un post devuelve por parámetro la cantidad de likes que posee y retorna una lista con los usuarios que le dieron like ordenados alfabéticamente. En caso de que la cantidad de likes del posts sea cero devuelve una lista vacía.
- ***void posts_destruir(posts_t* posts)***: destruye el *TDA posts*.

El siguiente diagrama muestra la estructura del *TDA posts* partiendo del *typedef struct posts posts_t*.

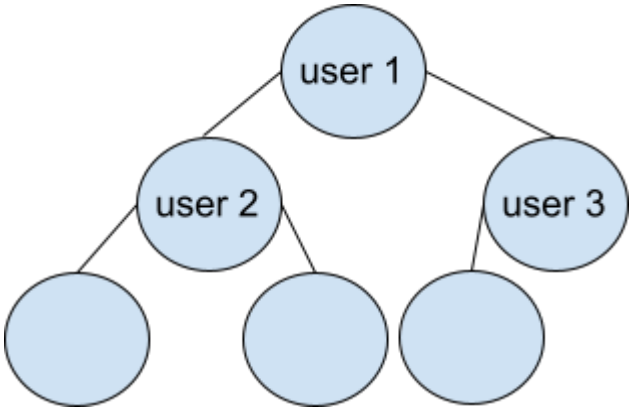
```
typedef struct posts {
        size_t generador_id;
        hash_t * hash;
} posts_t;
```

Tabla *hash*

CLAVE	**DATO**
“0”	dato_hash_t* dato_hash;
“1”	dato_hash_t* dato_hash;
…	…

```
typedef struct dato_hash {
        char* texto;
        char* usuario;
        size_t cant_likes;
        abb_t *abb;
} dato_hash_t;
```

ABB con los usuarios que le dieron like al post



DESARROLLO

Una vez implementados los TDAs descritos en la sección anterior, el desarrollo del programa principal queda de manera muy simple y concisa contando básicamente de cuatro partes:

1. Creación de los TDAs

Consiste en la ejecución de la función ***bool crear_TDAs(usuarios_t** usuarios, sesion_t** sesion, posts_t** posts)*** la cual se encarga de la creación de los tres TDAs retornado *true* si se logró exitosamente. En caso de error, se retorna *false* y se aborta la ejecución del programa.

2. Carga de archivo

Esta parte consiste en la ejecución de la función ***bool cargar_archivo(usuarios_t *usuarios, const char *nombre_archivo)*** la cual se encarga de abrir el archivo de usuarios (provisto como argumento de la ejecución del programa) y guardarlos en el *TDA usuarios*. Finalmente se cierra correctamente el archivo, y se retorna *true* en caso de éxito o se retorna *false* y se cierra el programa en caso de error.

3. Lógica principal (lectura del *stdin*)

La lógica principal del programa se lleva a cabo mediante la ejecución de la función ***bool algogram(usuarios_t *usuarios, sesion_t *sesion, posts_t *posts)*** la cual se encarga de ir leyendo los comandos ingresados en el standard input (*stdin*). En caso de que el comando sea inválido, se ignora y se procede a una próxima lectura. En caso de que se ingrese un comando válido, se llama a una función secundaria de nombre homónimo al comando que será la encargada de resolver lo requerido mediante el uso de las primitivas de los TDAs pertinentes. Cuando se cierra el *stdin*, la función finaliza retornando *true*.

4. Destrucción de los TDAs

Esta sección consiste en la destrucción de los tres TDAs mediante la función ***void destruir_TDAs(usuarios_t *usuarios, sesion_t *sesion, posts_t *posts)***.

Además de las cuatros secciones descritas (y sus respectivas funciones principales), se cuenta con varias funciones secundarias homónimas a los comandos pedidos que ayudan a realizar el procesamiento de manera más modularizada y prolija.

Para el manejo de mensajes a mostrar por consola, se optó por utilizar un arreglo de *strings* (*status_msj[ ]*) que se indexa mediante el uso de un tipo de datos enumerativo (*enum status*) según corresponda en cada comando.

CONCLUSIONES

- Tal cual como sugirió el cuerpo docente, el hecho de sentarse a pensar la solución en términos de las estructuras conocidas (y la complejidad de sus primitivas) para diseñar TDAs propios primero y luego hacer el programa principal, simplificó mucho la tarea de desarrollar este trabajo práctico.
- Implementar TDAs que cumplan funciones específicas permite debuggear y resolver problemas de manera aislada al programa principal, así como también agregar eventuales mejoras cuando se necesite.
- El manejo de mensajes mediante un arreglo de *strings* indexado por tipo enumerativo nos permite visualizar todos los mensajes juntos y simplificar la tarea de agregar nuevos.