Informe TP2 - Algogram

Trabajo Práctico n° 2

Algoritmos y programación II, curso Buchwald

Entrega individual

- **Nombre**: Santiago Nicolás Reynoso Dunjo
- **DNI**: 43104340
- **Padrón**: 107051

Ayudante asignada

- **Nombre**: Fiona González Lisella

Análisis y diseño de la solución

Planteamiento de estructuras

Desde un principio se trató la idea de crear un ***TDA*** por cada entidad que tenga acciones que realizar dentro del programa.

Luego del análisis de la problemática a resolver (crear una red social) se decidió que las entidades principales serían los **usuarios** y los **posts**.

Lo primero que se realizó es la creación de las **interfaces** a respetar para la entidad `Usuario` y para la entidad `Post`. Estas incluyeron en un principio las acciones básicas que se deberían implementar para su posterior uso.

A medida que el desarrollo avanzaba, se crearon más funcionalidades que beneficiarían la lectura y complejidad del código.

Al tener las estructuras iniciales casi completas, surgió una nueva necesidad que era la de **regular**, **mantener** y **organizar** en **conjunto** estas estructuras creadas. Como consecuencia, se crearon los *TDAs* correspondientes a las entidades `Posts` y `Usuarios` que ayudarían a poder solucionar esta necesidad dicha**.**

Estructuras de datos utilizadas

En el proyecto se utilizaron diferentes estructuras de datos que se pueden dividir en tres subgrupos.

TDAs implementados en la cursada utilizados de manera directa

- `ABB`: Se utilizó un diccionario ordenado implementado con un ***árbol de búsqueda binaria*** para poder hacer un manejo de los **likes** en los **posts**.
- `Hash`: Se utilizó un diccionario implementado con un ***hash*** para poder realizar operaciones de consulta e inserción con una complejidad igual a `O(1)`. La utilidad de esta estructura de datos se ve principalmente a la hora de almacenar los **usuarios registrados,** los **posts creados** y, a la hora de consultar si están dados de alta o no.
- `Heap`: Se utilizó una cola de prioridad implementada con un ***heap*** para poder establecer diferentes prioridades a los posts dados de alta por otros usuarios.

TDAs implementados en la cursada utilizados de manera indirecta

- `Pila`: Esta estructura de datos es utilizada indirectamente debido a que la necesita el TDA `ABB` para poder recorrer de manera eficiente sus elementos en un recorrido ***in-order***.
- `Lista`: Esta estructura de datos es utilizada indirectamente debido a que la necesita el TDA `Hash` ya que este está implementado como un ***hash abierto*** el cual internamente utiliza **listas** para lidiar con las **colisiones**.

TDAs implementados en base a requerimientos y entidades del problema

`Usuario`

Estructura de datos implementada para poder realizar acciones de manera rápida y sin repetición de código ante cada usuario registrado. Cada usuario se da de alta con un `id` y, un `index` manejado por la estructura de datos padre `Usuarios`.

Las principales acciones de este TDA son:

- Agregar un post al *feed*
- Saber si hay más posts para ver
- Ver el siguiente post en el *feed*

El *feed* de cada usuario es una **cola de prioridad** implementada como un `Heap` que define las prioridades de los posts a partir de una función de comparación que, internamente, utiliza al `index` recibido.

Este `index` define las prioridades de los *posts* en base a **"distancias"** entre los usuarios, siendo estas el elemento principal a la hora de comparar. Las "distancias" entre los usuarios viene dada por su posición en la lista recibida al inicio del programa.

Si la "distancia" de un `usuario2` a un `usuario1` es **menor** a la "distancia" de un `usuario3` al `usuario1`, significa que el `usuario2` tiene **mayor prioridad** que el `usuario3`.

De esta manera, cada vez que se crea un usuario se determina una función de comparación que se utilizará a medida que se agreguen posts al *feed*.

`Usuarios`

Estructura de datos implementada para

- **Leer** el archivo recibido inicialmente con los usuarios dados de alta
- **Guardar** a cada uno de los usuarios dentro de un diccionario
- **Asignarles** un `index` (utilizado para la definición de las prioridades)
- **Ver** de manera rápida si un usuario está dado de alta o no
- **Guardar un nuevo post** en el *feed* de cada usuario.

Este TDA con estas acciones permiten trabajar de manera ordenada con una cantidad `n` de usuarios.

En el momento de la creación, al inicio del programa, se lee el archivo de usuarios proveído y se los guarda. Internamente almacena a estos usuarios leídos dentro de un diccionario implementado con un `Hash` el cual nos permite realizar consultas e inserciones de manera rápida con una complejidad igual a `O(1)`.

Al contar con esta estructura y su iterador, podemos realizar acciones para todos los usuarios de una manera eficiente y sencilla (utilizado para guardar un nuevo post en el *feed* de cada usuario).

`Post`

Estructura de datos implementada principalmente para manejar el comportamiento de los *likes*. Internamente se utiliza un diccionario ordenado implementado con un árbol de búsqueda binaria el cual administra los likes que dan los usuarios.

Las principales acciones de este TDA son:

- Guardar un like de un usuario en un post
- Mostrar los likes que tiene un post
- Mostrar la información del post (autor, contenido y likes)

La utilización de este árbol con la función de comparación `strings.compare` de los paquetes nativos de `Go` permite guardar los *likes* en `O(log n)` siendo `n` la cantidad de *likes* en el árbol y, luego, poder visualizar de manera ordenada el contenido del árbol con una complejidad igual a `O(n)`.

`Posts`

Estructura de datos implementada para

- Almacenar los posts creados por los usuarios
- Darles un `id` para así poder llevar un orden
- Proporcionar acciones fáciles y rápidas cuando se necesite interactuar con los posts dados de alta.

Este TDA almacena los posts creados dentro de un diccionario implementado con un `Hash` lo cual permite que las inserciones y consultas se realizen con una complejidad igual a `O(1)`.

Sirve como un mediador para el acceso a la información de los posts y el guardado de likes.

Análisis de complejidad para las acciones requeridas

En base a las reestricciones impuestas por los estudios de mercado adjuntados en la consigna, a continuación, se justificarán las complejidades de las acciones del programa derivadas de las acciones de las estructuras de datos utilizadas.

Login

- Evaluación de errores `O(1)`.
- Obtener el usuario *loggeado* `O(1)` → Se busca si la clave pertenece en el `Hash`.

Complejidad del **login**: `O(1)`.

Logout

- Evaluación de errores `O(1)`.
- Poner en nulo la variable que almacena al usuario loggeado `O(1)`.

Complejidad del **logout**: `O(1)`.

Publicar un post

- Evaluación de errores `O(1)`.
- Guardar post `O(1)` → Se guarda un nuevo post en el diccionario de posts. Al utilizar un `Hash` es `O(1)`.
- Guardar en el *feed* de cada usuario `O(u log(p))`
  - `u` porque debemos realizar esta acción para todos los usuarios.
  - `log(p)` porque la acción que debemos realizar es encolar en un `Heap` que contiene una cantidad `p` de posts.

Complejidad de **publicar un post**: `O(u log p)` siendo `u` la cantidad de usuarios registrados y `p` la cantidad de *posts* en la red social.

Ver siguiente post

- Evaluación de errores `O(1)`.
- Ver al post con mayor prioridad del *feed* `O(log(p))` → En un `Heap`, desencolar el elemento con mayor prioridad cuesta `O(log(n))`.

Complejidad de **ver siguiente post**: `O(log p)` siendo `p` la cantidad de *posts* en la red social.

Likear un post

- Evaluación de errores `O(1)`.
- Guardar like en post `O(log` u_p `)` → Al tener un `ABB` manejando los *likes* de un post, el guardado de un linke cuesta `O(log n)`.

Complejidad de **likear un post**: `O(log` u_p `)` siendo u_p la cantidad de usuarios que le dieron *like* al post.

Mostrar likes

- Evaluación de errores `O(1)`.
- Mostrar los *likes* de un post `O(` u_p `)` → El recorrido ***in-order*** de un `ABB` permite ver los *likes* de manera ordenada. El recorrer todos los elementos en un `ABB` cuesta `O(n)`.

Complejidad de **mostrar likes**: `O(` u_p `)` siendo u_p la cantidad de usuarios que le dieron *like* al post.