Trabajo Práctico Nº2

“ALGOGRAM”

Algoritmos y Programación II, curso Buchwald.
Facultad de Ingeniería de la Universidad de Buenos Aires.

Alumnos**:**

Gerez, Facundo Nahuel. 109429.
Jackunas, Mauro Nicolás. 109723.

Corrector asignado:

Baldi Morales, Tomás.

Presentación:

En este informe vamos a explicar el diseño del Trabajo Práctico Nº2 de Algoritmos y Programación II. Vamos a detallar, el por qué elegimos las estructuras usadas y qué beneficios aporta en la complejidad del programa.

El Trabajo Práctico tiene como temática las redes sociales, tenemos que conseguir que el usuario, a través de comandos, pueda interactuar con esa red y tener su propia experiencia según su afinidad con los otros usuarios.

A continuación, vamos a desarrollar la idea que tuvimos para programar cada comando y qué herramientas utilizamos.

ALGOGRAM

Estructura del Programa:

Nuestra implementación la separamos en dos módulos, la de la Red Social y la de los Usuarios, y después esta el main interactuando con el TDA Red Social; en la primera estarán los métodos de la funcionalidad de los comandos de la red social dicha y en el segundo estará la lógica con los datos y funciones de los usuarios (a medida que detallamos las funcionalidades de Algogram, explicaremos cómo se componen los usuarios).

Explicación de los comandos:

Login:

Como condición en la consigna del trabajo la complejidad de este comando debe ser constante, O(1). En esta función usamos un diccionario, porque a partir de una clave, el nombre de usuario, podemos ingresar a la red social. Este diccionario será guardado en la estructura de la red, ya que será un registro de todos los usuarios, y todas las primitivas van a tener acceso a él, tendrá como claves el nombre y sus valores serán la estructura de cada usuario. Esta incluirá el nivel el cual se obtiene según la línea en la que se encuentra en el archivo de usuarios (define la afinidad), y su propio feed (el feed y el nivel son explicados en ***Publicar***).

El TDA que nos permite usar un diccionario y respeta la complejidad constante pedida, es el Hash. Gracias a sus propias primitivas podemos saber si un usuario pertenece al registro en forma O(1).

Logout:

Con un puntero al nombre del activo actual de la red guardado podemos validar rápidamente si hay alguien conectado. Esta primitiva queda con complejidad constante, debido a que solamente hay una comparación y una reasignación de una variable (en el peor caso), el puntero hacia el usuario pasará a nil (nil cumple el rol de desconección).

Publicar:

En el archivo de Usuarios hay dos estructuras: la del usuario mismo y los posteos. Nosotros pensamos a cada posteo como una estructura a la cual todos pueden acceder excepto el usuario que la creo, de esta forma será más fácil y rápido ver quien lo posteo, su cantidad de ‘likes’, su id y prioridad.

Para la afinidad entre usuarios lo solucionamos guardando la posición en el archivo de cada uno,y haciendo módulo de la resta entre niveles podemos saber la distancia entre cada uno. Entonces, el feed personal de cada usuario donde el siguiente post en aparecer depende de quién lo publicó y su afinidad con el usuario actual, para ello usamos el TDA Heap para definirlo, donde guardaremos los punteros a los posteos como valores con una función que compara la afinidad del usuario actual con los posteos agregados al Heap.

Además, pensamos en usar punteros para que el posteo sea unico y no múltiples posteos en cada feed de cada usuario, pero estos punteros vienen de donde está guardado este posteo que es en la misma Red Social, el cual tiene el rol como de una nube la cual lo almacena en un array para si se necesita acceder a este por el Id, este pueda ser en O(1).

La lógica de la primitiva es la siguiente: iteramos el registro de usuarios y por cada uno de ellos publicamos en sus feeds el nuevo posteo. La iteración lo hacemos con el iterador externo del diccionario, que tendrá como complejidad *Nu* (*Nu* siendo la cantidad de usuarios) porque lo repetiremos para todo el registro; y la publicación se agrega en la cola de prioridad, y por su condición de Heap[1] su primitiva de Encolar es Log(*Np*) (*Np* la cantidad de posteos). Con esto respetamos la consigna de la complejidad de la primitiva: O(*Nu**Log(*Np*)).

Ver Próximo Post:

Gracias de haber guardado el feed del usuario como una Cola de Prioridad, simplemente lo desencolamos y retornamos el mensaje.

En la primitiva de la red también usamos una primitiva de diccionario para tener acceso al usuario actual.

Por ser primitiva del Heap, el desencolar es Log(Np) y la del diccionario por ser del Hash es constante. Por ser el más pesado, la complejidad final de la primitiva es Log(Np). Respetando la consigna.

Dar y Mostrar Like de un Post:

En la estructura de la red social existe un arreglo que guardará todos los posteos, esto lo definimos así (además de facilitar el acceso a la información de los posteos) porque la posición en este arreglo será igual a la ID del posteo.

Los Likes pueden ser guardados en un arreglo o, para hacer lo más efectivo posible, en un diccionario (en este caso Hash al ser constante). Pero la impresión debe ser por orden alfabético. Entonces, el TDA que nos permite guardar datos en el orden que queremos es el ABB (árboles binarios de búsqueda). Cada post tendrá su propio diccionario ordenado, debido a que por cada uno será diferente quién le dio 'Me gusta'.

La primitiva de Dar Like será Log(*Nmg*) (siendo *Nmg* la cantidad de personas que le dieron 'Me Gusta' al posteo). Porque la primitiva de Guardar del ABB[2] es Log(N), y el resto de acciones son constantes. Y al ser un diccionario, nos permite asegurarnos que si un usuario vuelve a darle 'like' a un post no se sumará uno extra.

[1] En el Heap, se guardan la ubicación de los hijos y padres, por ende se ordena sin necesidad de recorrer toda la cola.

[2] El árbol solo recorrerá una rama, evitando pasar por todos sus nodos.

La última primitiva, usará los iteradores provenientes del diccionario, que nos permitirá recorrer[3] de manera ordenada los likes dados al post, imprimiendo paso a paso. Sólo tendrá órdenes constantes, tales como imprimir o asignación de valores, pero estas se repiten por el Iterador Externo del ABB *Nmg* veces. Por estas razones, la primitiva tendrá complejidad O(*Nmg*).

Conclusión:

Creamos este diseño teniendo en mente a las redes existentes. Esta red a pesar de estar limitada por la solo escritura y el no poder ver un posteo anterior, terminamos convencidos por esta estructura debido que consideramos a la Red Social como el que almacena y mueve los registros y Posts como si estuvieran en una nube, asi como tambien nos mantenga logueados al pasar de Posteo en Posteo; y a los usuarios que manejan únicamente la visión de estos posteos según su afinidad con otros, y la habilidad de postear para otros. Además, el posteo es un paquete o un elemento/dato que solo contiene la información importante para su visualización, y por ende no tiene otra función más que el ser visto desde cualquier punto de los logueados.

El trabajo nos ayudó a entender cómo funcionan las páginas como estas, es como una forma muy primitiva de aplicaciones web como Twitter, nos abre las puertas para tareas más complejas con herramientas muy útiles como lo son los diccionarios y la cola de prioridad.

[3] Recorrido InOrden.