FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Algoritmos y programación II

Cátedra BUCHWALD

TRABAJO PRÁCTICO Nº2
TEMA: Algogram
GRUPO: 22

ALUMNOS:

109599 – Gonzalez, Valentin
109689 – Villalba, Ana Daniela

CORRECTOR:

Federico Brasburg

Definición de TDAs nuevos

TDAs implementados, cada uno con las primitivas necesarias para tanto (1) poder acceder a la información guardada en cada estructura, (2) realizar acciones asociadas a las mismas.

USUARIO
- – – Nombre (1)
- – – VerAfinidad (1)
- – – ActualizarFeed (2)
- – – ProximoPost (1)
- – – Id (1)

PUBLICACION
- – – Texto Publicación (1)
- – – Id (1)
- – – VerPublicador (1)
- – – Likear (2)
- – – MostrarLikes (1)
- – – CantLikes (1)

Originalmente habiamos creado una primitiva Publicar() en el TDA usuario. Para los fines de este trabajo práctico no hacia mucha diferencia tenerla como primitiva o en el main, ya que la información del usuario necesaria era accesible por la primitiva Nombre. Finalmente terminamos delegando la creación de la publicación al programa principal, a través de CrearPublicacion, parte de Publicación, y guardando un puntero al Usuario con toda la información que allí se pueda necesitar. Esta decisión hace más fácil poder manejar todos los datos alojados para poder cambiar la manera de definir la afinidad entre usuarios, de ser necesario.

Excepto las aclaradas a continuación, todas las primitivas son de una complejidad de tiempo constante.
ProximoPost: O(log p) por utilizar desencolar con un heap auxiliar.
ActualizarFeed: O(u log p)
MostrarLikes: O(k), por recorrer el arreglo de nombres de usuarios que likearon.

Tomaremos como referencias en todo el trabajo: p - número de posts, u - total de usuarios del programa, k - número de usuarios que likearon la publicación.

Lectura del archivo de usuarios

Lo resolvimos de una forma bastante similar al anterior trabajo. El programa lee línea por línea el nombre de un usuario y con esto crea un TDA usuario para

cada uno. Luego lo guarda en un diccionario ya creado de usuarios donde la clave es el nombre correspondiente para poder acceder más tarde a estos cuando sea necesario.

Originalmente, al momento de guardarlo en el diccionario se calculaba la afinidad de este usuario con todos los otros usuarios registrados en el programa. La complejidad total de solamente calcular las afinidades para todos los usuarios (en preparar usuarios recorremos el arreglo de usuarios u - complejidad O(u)- llamando a DefinirAfinidad para cada usuario que lo vuelve a recorrer para calcular la afinidad del usuario actual con todo el resto -complejidad O(u)-) sería exponencial. Habiamos considerado que era más conveniente (a pesar de la complejidad horrible) ya que en un uso extenso del programa el costo del inicio del programa se mantendria sin importar cuantas operaciones realicemos entre los usuarios.

Sin embargo, tuvimos problemas al pasar la prueba de volumen del autocorrector (no así de manera local) y decidimos probar la manera que quedó efectivamente representada en el codigo: la afinidad se calcula en cada operación, que sea necesaria (actualizar feeds y ver_siguiente_feed), y al ser de tiempo constante, no afecta en general las complejidades exigidas en este ejercicio.

Complejidad de los comandos

Más allá de aquellas aportadas por los tdas auxiliares (detalladas en el próximo cuadro), todos los comandos utilizan operaciones de tiempo constante.

En el caso especifico de mostrar_likes, además de iterar, se debe recorrer el arreglo resultante de esa iteración pero eso seria igualmente O(2.k) == O (k).

Comando	**Complejidad**	**TDA Auxiliar / primitivas usadas**
LOGIN	O (1)	Hash / Pertenece, Obtener
LOGOUT	O (1)	Hash / Guardar
PUBLICAR*	O(u log p)	Hash / Iterar, Heap / Encolar
VER_SIGUIENTE_FEED	O (log p)	Heap / Desencolar
LIKEAR_POST	O (log k)	ABB / Guardar
MOSTRAR_LIKES	O (k)	ABB / Iterar

**El feed es un arreglo, por lo que tendrá el costo de redimensionar cada tanta cantidad de publicaciones nuevas.*

Detalles de la implementación de los comandos

"*login*": Guardamos todos los usuarios en un diccionario en vez de cualquier otra cosa para que funcione correctamente este comando, ya que requiere que trabaje en tiempo constante, y como las búsquedas, guardados y borrados en un Hash ocurren en tiempo constante, al ingresar este comando solo hace falta comprobar si el usuario ingresado pertenece al diccionario, y realizar lo que corresponda según el resultado dado.

"logout": ya que solo es necesario hacer el seguimiento del último usuario que ingresó, con la creación de una variable que guarde a este mismo usuario es más que suficiente para que este comando funcione en tiempo constante, porque consiste únicamente en actualizar el valor de esa variable. Para representar que no ingresó ningún usuario, hacemos que esa variable valga nulo.

"publicar": empezamos a utilizar las primitivas de los TDAs implementados. Cuando un usuario publique un post se creará un TDA publicación, el cual tiene información de su publicador, su ID, y el mensaje, además de otras primitivas que se utilizarán más tarde. Esta publicación se guardará en un arreglo de publicaciones ya creado para poder asignarles su ID de forma sencilla y en tiempo constante.
Esta es la parte que define la complejidad del algoritmo. Una vez creada la publicación se debe actualizar el feed para todos los usuarios registrados en la aplicación.
El feed de cada usuario nosotros lo implementamos como un Heap de publicaciones. Si bien hubiera sido viable implementarlo como un ABB, tanto por las complejidades de sus primitivas como por facilidad de código no nos pareció conveniente (y también a que tiene mucho más sentido utilizar una cola de prioridad para trabajar con la afinidad, o sea, la prioridad de ver a un usuario ante otro).
Actualizar el feed consiste en, para cada usuario, encolar un puntero a la publicación creada, y como encolar un elemento en un Heap tiene complejidad logarítmica en función del total de elementos, en este caso publicaciones, y realizamos esto para todos los usuarios registrados, termina quedando la complejidad pedida para este comando.

A su vez, no era posible considerar algún otro TDA ya que era necesario poder determinar una función de comparación personalizada según nuestras necesidades en este TP como si desearamos cambiar la forma de definir la afinidad.

"ver_siguiente_feed": consiste básicamente en desencolar el máximo del Heap de publicaciones de un usuario, si el Heap está vació devuelve el error correspondiente. Como desencolar el máximo de un Heap tiene complejidad logarítmica en función de la cantidad de elementos, que son publicaciones, el comando cumple la complejidad indicada.

A su vez, como estamos usando punteros a las publicaciones, debería ser más conveniente a nivel de memoria utilizada.

"likear_post": definimos en el TDA de publicación los likes de cada publicación como un Árbol Binario, donde las claves son los nombres de los usuarios que le dieron like al post, y los valores no son relevantes para esta implementación.
Decidimos implementar este comando con un ABB debido, principalmente, a que los nombres de los usuarios que dieron like se deben mostrar en orden alfabético, y este TDA ordena las claves según una función de comparación, para nuestro caso, es simplemente comparar cadenas, con lo cual lo consideramos una buena herramienta para este caso.
Además likear un post se reduce a guardar en el diccionario el nombre del usuario correspondiente y un valor arbitrario, acción de complejidad logarítmica en función del total de elementos en el árbol, que serían los usuarios que likearon ese post. Por lo tanto también cumple con lo pedido.

"mostrar_likes": funciona con una iteración del ABB de likes de la publicación indicada, guardando todos los nombres pertenecientes al árbol en un arreglo de cadenas para luego imprimirlos uno por uno. Tanto la iteración como la impresión de los nombres son de complejidad lineal, en función de todos los elementos del ABB, los cuales representan los usuarios que likearon el post. Así que tiene la complejidad pedida.