Fundamentos de Estructuras de Datos y Algoritmos

Tarea 3

Danilo Bonometti

I. Introducción

Los Tipos de Datos Abstractos (TDA) asociativos almacenan pares de datos y proveen una interfaz para referenciar uno de estos datos (valor) a través de otro (llave). Estos tipos de datos poseen propiedades que los hace ideales para trabajar con asociaciones de datos, ya que almacenarlos en estructuras básicas como arreglos o listas, haría que el manejo de estos tuviera un orden de complejidad superior.

En este trabajo se compara el rendimiento de las implementaciones de tres TDA: Árbol Binario Búsqueda, Tabla Hash con Hashing Abierto y una con Hashing Cerrado, para leer una base de datos de registros de usuarios de Twitter que comprenden siete campos distintos. Las implementaciones deben poder ingresar los registros a la estructura utilizando como llave la ID y el Nombre de usuario, de manera de comprender los beneficios de usar una estructura sobre la otra. Como se verá del trabajo realizado, las estructuras de arboles son considerablemente más eficientes al momento de insertar datos, sin embargo las búsquedas son mucho más lentas que en las tablas hash. También podremos ver que entre las tablas hay una gran diferencia en la búsqueda de datos que se encuentran almacenados en estas versus los datos que no están almacenados y como el manejo de la búsqueda hace que la tabla hash de hashing cerrado se mucho mejor para buscar llaves numéricas que no están en la tabla.

II. Estructuras de Datos

A. *Árbol Binario Búsqueda*

Los árboles son un tipo de dato abstracto que organiza los datos almacenados en un esquema de nodos padres que mantienen referencias a nodos hijos y a la vez a su propio nodo padre, comenzando desde un nodo raíz. Una especialización de estos son los árboles binarios de búsqueda, los cuales mantienen solo dos nodos hijos (izquierda y derecha) por cada nodo padre.

Los árboles binarios de búsqueda mantienen una sola invariante, que el hijo izquierdo de un nodo $node$ cualquiera cumpla $\text{left.key} \leq \text{node.key}$ y que el derecho cumpla $\text{node.key} \leq \text{right.key}$, la relación debe ser estricta para uno de los lados dependiendo del criterio que se quiera utilizar. Para las operaciones de búsqueda e inserción, se debe recorrer a lo más hasta el nodo mas lejano al nodo raíz (altura del árbol) por lo que para una altura h el costo de inserción está acotado por O(h). Para la eliminación se tienen tres casos posibles: El nodo a eliminar no tiene hijo izquierdo, entonces es reemplazado por su hijo derecho; El nodo a eliminar no tiene hijo derecho, entonces se reemplaza por su hijo izquierdo; El nodo a eliminar tiene hijo izquierdo y derecho. Para este último caso se busca el sucesor del nodo a eliminar (s). Sí el nodo derecho es s, entonces lo reemplaza, de lo contrario s está lo más a la izquierda en el subárbol derecho, luego se intercambia el nodo derecho por s y en caso de que s tenga un nodo hijo derecho, este se vuelve hijo del anterior padre de s. De esta forma se tiene que la inserción a lo más va a recorrer hasta la altura del árbol y su complejidad también es O(h).

B. *Tabla Hash*

Las tablas hash son arreglos de datos de tamaño fijo que almacenan elementos con una única llave como indice y que tienen un tiempo promedio O(1) para operaciones de búsqueda, inserción o eliminación ya que solo dependen de si pueden ser indexadas en la tabla o no. Estas estructuras indexan las llaves mediante una función *hash* que cuando está bien diseñada, idealmente, mapea cada llave a un indice distinto dentro del arreglo de la tabla. Hay muchas formas de diseñar funciones hash, pero para soluciones versátiles en las que no se conoce el universo de llaves a almacenar no se puede asegurar que una hash no mapeara dos llaves al mismo indice, lo cual es conocido como una colisión. Una buena función hash idealmente genera una distribución de llaves lo más uniforme posible. Los tipos de tablas hash se clasifican basadas en como manejan las colisiones. A medida que las tablas se llenan de datos aumenta la posibilidad de que se generen colisiones. Una forma de controlar esto, es considerar un factor de carga de la tabla y al momento de sobrepasar un cierto umbral aumentar el espacio y hacer un rehashing de los datos para distribuirlos en base al nuevo tamaño de la tabla.

1) *Hashing Abierto*

También conocido como *Closed Addressing* o *Separate Chainning* (SC), es una estrategia donde las colisiones no se resuelven, si no que se utilizan listas en cada indice del arreglo de la tabla para anidar los datos ingresados que produzcan la misma hash para sus llaves. Esta es una estrategia permite que la inserción sea cercana a O(1), sin embargo la búsqueda y eliminación estarán determinadas por el tamaño de la lista a la que se ingresen, por lo que se diseñan para intentar asegurar que el tamaño de las listas no sea muy grande y mantener la complejidad constante, ya sea limitando el tamaño de la lista u ocupando una función hash con una distribución lo suficientemente buena para que las llaves no colisionen tanto.

2) *Hashing Cerrado*

También conocido como *Open Addressing* (OA), es una estrategia donde las colisiones son manejadas encontrando

otro indice en la tabla para la llave que genera la colisión basándose en un criterio en particular. Existen distintos criterios para encontrar un nuevo indice, entre estos están *Linear Probing* (LP), *Quadratic Probing* y *Double Hashing*, estudiados en clases. Linear Probing busca linealmente el siguiente indice inmediatamente disponible en la tabla, Quadratic Probing busca utilizando una función cuadrática y Double Hashing busca un siguiente indice basado en una segunda función hash con un rango más acotado.

Para llaves numéricas no hay mucha dificultad en implementar una función hash adecuada, sin embargo si se desean utilizar llaves de ristras es necesario codificarlas de manera de obtener una representación única de éstas. Una forma de codificarlas es de la misma forma en que se codifican los números en cualquier sistema numérico, es decir, asociar un valor a los caracteres según un alfabeto de valores y multiplicarlos por un valor asociado a su posición en la ristra. Por ejemplo, para un alfabeto que le asocie a cada letra un valor del 1 al 26 y la ristra *asdf* se tendría que su valor codificado es $e = a \cdot 26^3 + s \cdot 26^2 + d \cdot 26^1 + f \cdot 26^0$. Esto le daría una representación única a la ristra *asdf*, sin embargo, un nombre de usuario en Twitter puede contener 15 caracteres entre alfanuméricos y guiones bajos (“_”) lo que significa que para codificar cualquier ristra se necesita un alfabeto de 63 caracteres. Para esto son necesarios $\log_2 63^{15} \sim 92$ bits. Como actualmente la palabras de un procesador son de 64 bits, se necesita acotar el conjunto de entrada de alguna forma. Un método es simplemente tomar una muestra más pequeña de caracteres de la ristra original. Si se toma $\log_2 63^c = 64$ se tiene que $c \sim 10.7$ por lo que a lo más se pueden representar ristras de 10 caracteres usando el alfabeto para los nombres de usuario en Twitter.

La función hash implementada es:

$$\text{hash}(k, M) = k_{\text{mod}} M$$

Donde k es la llave del dato a ingresar y M es el tamaño de la tabla. Para la implementación de las tablas se utilizó un tamaño base de 11 elementos y cuando sea necesario aumentar el tamaño se toma el siguiente valor primo al doble del tamaño actual de la tabla. Esto se hace para mejorar la distribución de las llaves de modo de no introducir sesgo al momento de calcular sus hash, ya que utilizando otro tipo de valores es más posible que se den comportamientos cíclicos. Se consideró un límite para el factor de carga de 90% para SC y 50% para OA. También para OA se probó la implementación de LP.

Para todas las estructuras se considera que si una llave ya fue insertada, inserciones siguientes de esa llave son descartadas. Esto significa que de todas formas se tiene el costo de indexar la llave en la estructura.

III. Datos

Los datos utilizados son 21070 registros de usuarios de twitter con 7 campos cada uno: Universidad a la que sigue el usuario, ID, Nombre, Cantidad de tweets, Cantidad de cuentas a la que sigue, Cantidad de cuentas que lo siguen y Fecha de creación de la cuenta.

El equipo de prueba utilizado tiene un procesador Ryzen 7 1700 de 8 núcleos, memorias cache L1d de 256 KiB, L1i de 512 KiB, L2 de 4 MiB, L3 de 16 MiB y 16 GiB de memoria RAM.

IV. Resultados

A. Tamaño

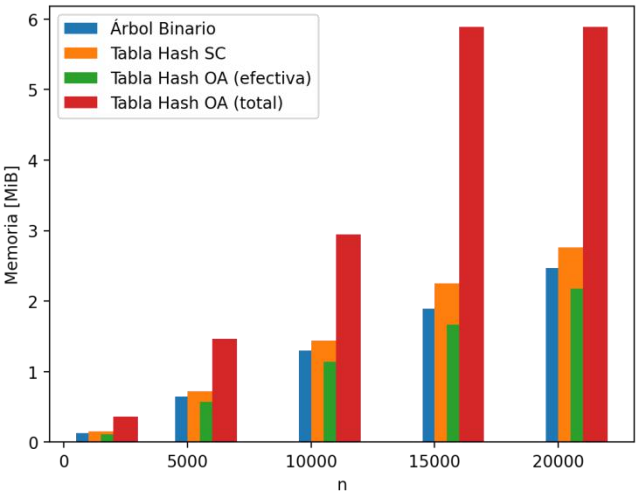

Figura 1. Tamaño de las estructuras de datos en MiB.

En la figura 1 se puede ver el cambio en tamaño de la estructuras a medida que se insertan los datos. Para la tabla hash OA se muestra la memoria efectiva utilizada y el tamaño total de la tabla. Se puede ver claramente la subutilización del espacio en OA, por lo que no es amigable a aplicaciones donde el ahorro de espacio es crítico. También se puede observar que a pesar de las diferentes implementaciones las estructuras no poseen grandes diferencias en términos espacio efectivo utilizado, sin embargo cabe mencionar que debido a que las implementaciones dependen de estructuras de la librería estándar, no es posible acceder a todos los tipos de datos contenidos dentro de los nodos de las listas utilizadas por lo que está representación es solo aproximada.

B. Inserción

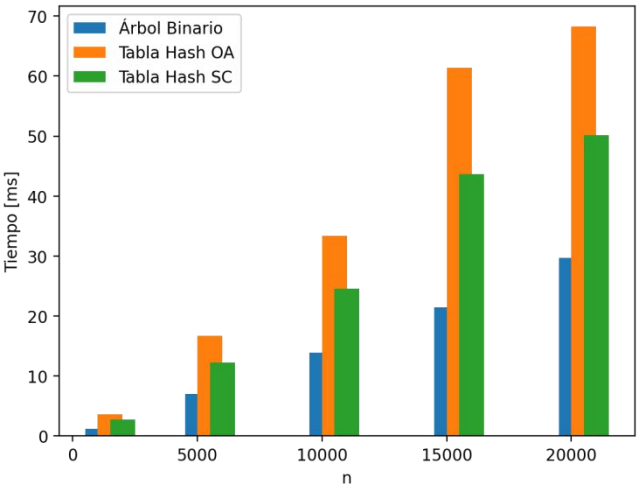

Figura 2. Escalabilidad para llaves de ID de usuario.

En las figuras 2 y 3 se puede observar el tiempo de ingreso para diferentes cantidades de datos y como la diferencia entre los tiempos de inserción de las estructuras aumenta a medida que aumentan los datos ingresados, independiente del valor de la llave utilizada. Se puede observar como todas las estructuras tienen un aumento de alrededor de 10 mili segundos al utilizar llaves de ristras, esto se debe a que las comparaciones en el árbol binario se hacen carácter a carácter y en las tablas se hace la codificación de la llave.

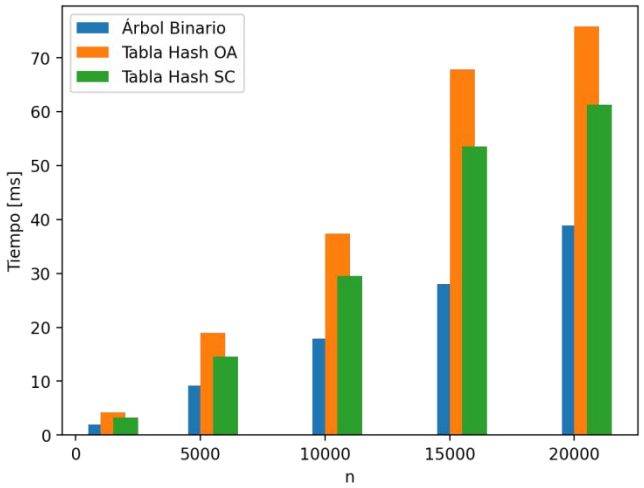

Figura 3. Escalabilidad para llaves de nombre de usuario.

También se puede observar como al llegar a pasar de 15000 a 20000 usuarios el crecimiento del tiempo es bastante menor para las tablas, sin embargo, para el árbol el crecimiento se mantiene contante debido a la naturaleza lineal de su complejidad.

C. Búsqueda

Se consideraron 1 millón de búsquedas con incrementos mil elementos para las pruebas.

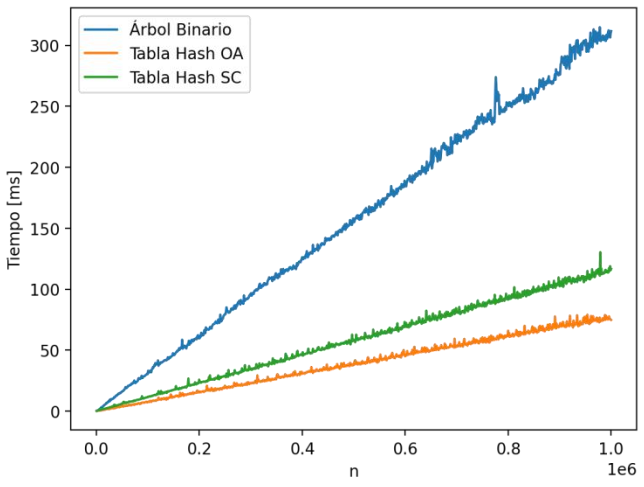

Figura 4. Búsqueda por ID de usuario almacenado.

En la figura 4 se puede apreciar al momento de referenciar datos que están almacenados en la estructura, las tablas hash superan por mucho en rendimiento al árbol binario. En particular la tabla hash con OA rinde mucho mejor que las otra estructuras estudiadas y solo demora al rededor de 70 mili segundos en hacer una búsqueda de 1 millón de elementos. También se puede ver como el árbol binario demora casi 6 veces más que OA.

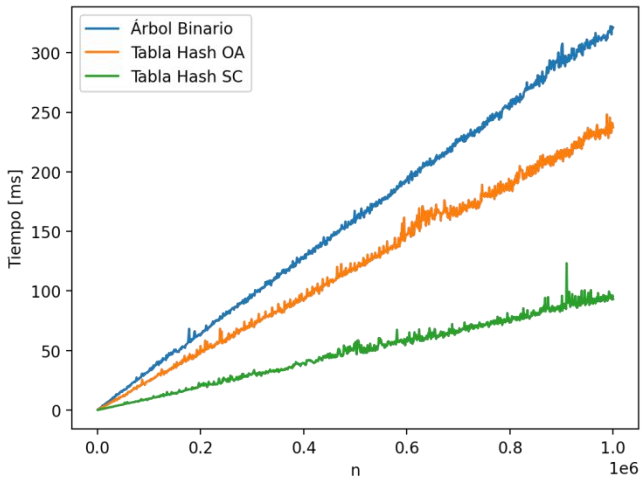

Figura 5. Búsqueda por ID de usuario no almacenado.

En la figura 5 se observa que el comportamiento es bastante distinto cuando se buscan elementos que no están almacenados en las estructuras. En particular se puede observar como el rendimiento de la tabla con OA es mucho menor comparado con el caso anterior. Esto se debe principalmente a que al utilizar LP como manejo de las colisiones demorará mucho más en darse cuenta que un dato no está en la tabla ya las acumulaciones de datos contiguos impiden que llegue rápidamente a un valor inválido.

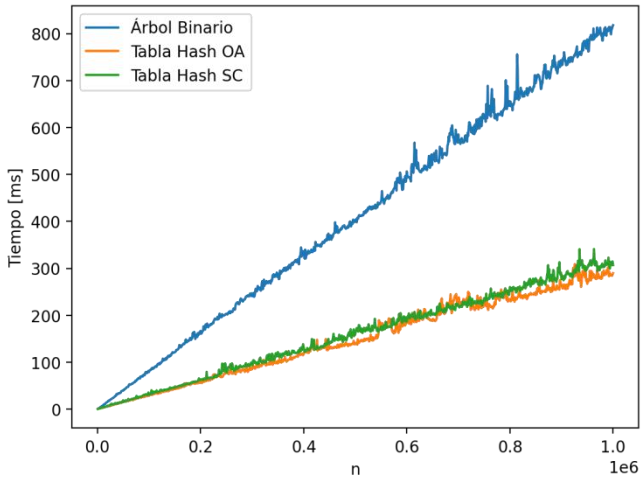

Figura 6. Búsqueda por nombre de usuario almacenado.

La tendencia del rendimiento cambia de gran manera al momento de utilizar una ristra como llave. Se puede observar de la figura 6 que el tiempo de búsqueda llega a tomar más del doble que con las llaves numéricas. También se observa como la ventaja de la tabla con OA sobre la tabla con SC es mínima. Esto se debe a que a las tablas les toma mucho más tiempo

calcular la hash de una llave de ristra ya que debe ser codificada, aumentando de gran manera las constantes asociadas a la búsqueda.

En al figura 7 se puede observar que, de manera similar a lo visto en la figura 5, la tabla hash con OA percibe la misma baja de rendimiento por las mismas razones. Sin embargo, también se observa que, a pesar de esto, para llaves de ristras el rendimiento más afectado es el de el árbol binario, ya que como este usa comparaciones lexicográficas para determinar el valor de una ristra las constantes asociadas a la búsqueda aumentan mucho más que en las tablas hash.

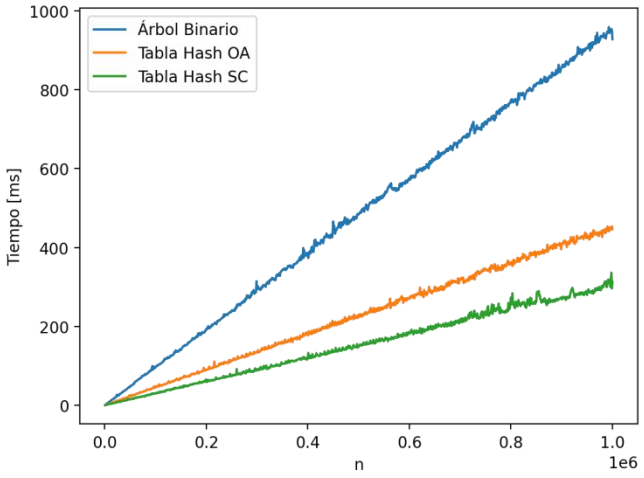

Figura 7. Búsqueda por nombre de usuario no almacenado

V. Conclusiones

De los resultados se puede ver que cada estructura tiene un nicho donde puede dominar sobre las otras. En el caso del árbol binario de búsqueda queda en evidencia que su fuerte está en aplicaciones de almacenamiento donde las consultas a los datos no son necesariamente frecuentes, ya que las constantes al momento de ingresar los datos son bastante bajas, para casos de ID y de nombre. Por otro lado, las tablas hash dominan sobre el árbol binario en términos de búsqueda. Independiente del tipo de tabla la indexación por hash es mucho más rápida para búsquedas, pero su velocidad para ingresar datos se ve mermada por la necesidad de expandir el tamaño de la tabla, y luego re indexar todas las llaves a nuevas posiciones en ésta.

También se pudo ver que utilizar llaves numéricas tiene un gran beneficio en el rendimiento temporal para todas las estructuras, en particular para las tablas hash con OA, ya que al utilizar llaves de ristras se requiere procesamiento adicional para poder manejar su ingreso o su referenciación. Sin embargo, utilizar llaves de ristras tiene muchísimas aplicaciones y como se puede ver de las figuras 6 y 7 las tablas con SC tienen una gran ventaja sobre las otras en este caso, ya que en la búsqueda por llaves insertadas el rendimiento es prácticamente igual al de la tabla con OA, pero en las búsquedas por llaves ausentes la supera por un buen margen. Esto las hace ideales para aplicaciones donde se consultan por muchas llaves de ristras que no necesariamente se encuentran en la tabla.

En términos de memoria, no hay una gran diferencia entre las estructuras ya que en sus implementaciones se utiliza un mismo tipo para almacenar los datos. Sin embargo, las tablas hash con OA ocupan una cantidad considerablemente mayor de memoria ya que por su naturaleza necesitan mantenerse lo suficientemente vacías para que la distribución de llaves sea lo más uniforme posible. Esto se traduce en una subutilización de la memoria reservada para la tabla haciéndolas poco eficientes en este ámbito. Aún así, son un claro ejemplo del intercambio de rendimiento de memoria por velocidad, ya que al mantener la tabla subutilizada se reducen de gran manera las colisiones de llaves. En este caso utilizar tamaños de tablas de números primos reduce la posibilidad de encontrar sesgo en la indexación de llaves por lo que se producen menos colisiones y la tabla debe pasar menos tiempo buscando un espacio vacío donde guardar la llave.

VI. Anexos

Los códigos de la tarea pueden encontrarse en: https://github.com/chromosome/tarea3