Árboles binarios: *Binary Search Tree*, *AVL Tree* y *Splay tree*.

Alvarado Contreras Rebeca.
Universidad Autónoma de San Luis Potosí, Facultad de Ciencias.
Ingeniería Física, Cómputo de alto rendimiento.
Proyecto 3: Árboles binarios.

***Objetivos*—Implementar las operaciones de búsqueda, inserción y eliminación para tres tipos de árboles binarios: *Binary Search Tree*, *AVL Tree* y *Splay tree*; y comparar el rendimiento entre estos.**

***Palabras clave*—*Binary Search Tree*, *AVL Tree*, *Splay tree*.**

I. Introducción

Los árboles son estructuras de datos compuestas por nodos que contienen un elemento de datos, estos pueden tener como hijos más subárboles. Se dice que son árboles binarios si cada nodo tiene a lo más, 2 subárboles como hijos: un puntero a la "izquierda" y otro a la "derecha". La raíz es el nodo que es la cabeza del todo árbol, y se dice que no tiene padre (es nulo).[1]

Un árbol de búsqueda binario (BST por sus siglas en inglés) es un tipo de árbol binario en el cual los nodos están de alguna manera ordenados: para cada nodo, todos los elementos a la izquierda son menores (o iguales) que el nodo, todos los elementos a la derecha son mayores que el nodo.[1]

A diferencia de las listas con n elementos, en la cual el tiempo de búsqueda es de $O(n)$, y aún más para ordenarla. Con los árboles binarios es posible hacer la búsqueda, inserción y eliminación en un tiempo de $O(\log n)$; por lo que si se tienen 1000 elementos, solo habría que visitar alrededor de 10 nodos.[2]

Sin embargo, lo anterior solo puede ser cierto si el árbol está balanceado, es decir, la altura del árbol izquierdo y derecho es similar (véase Figura 1); si no, puede llegar a comportarse como una lista ligada. Por esto, surge la necesidad de que el árbol se auto-balancee. Esta es justo la característica de los árboles AVL, los cuales mediante rotaciones se balancean, de tal manera que la diferencia de altura entre el árbol izquierdo y derecho no puede ser mayor a 1.[3]

Existen otros tipos de árboles, tales como el árbol biselado o *Splay Tree*, el cual tiene la propiedad de que los elementos recientemente accesados pueden ser accesados de nuevo rápidamente; mediante una operación llamada *splaying* (una sucesión de rotaciones) se lleva el nodo deseado a la cabeza/raíz del árbol.[4]

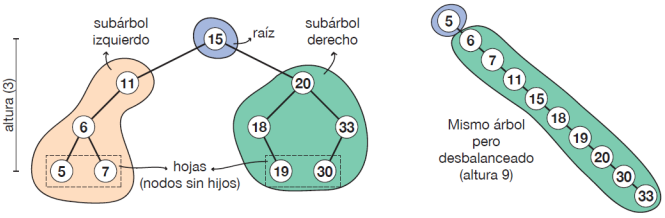

Figura 1: Árbol binario de búsqueda, sus partes y ejemplo de un caso extremo no balanceado. BST

II. Metodología

En este aparatado se habla sobre consideraciones generales, y una breve descripción del programa implementado.

Tanto para el BST como para el árbol AVL se usó recursión para evitar mantener la conexión explícita (un puntero) hacia los padres.

En el caso del árbol AVL, justo después de regresar de la iteración de la recursión se checa la altura del los subárboles izquierdo y derecho, y si es necesario, se hace la rotación apropiada y se regresa el nuevo nodo después de la rotación.

Ahora, referente a la operación de eliminar, la estrategia general fue la siguiente: al encontrar el nodo a eliminar, se manda este nodo y el subárbol izquierdo para buscar el máximo recursivamente (cuando ya no tiene más hijos derechos), se reemplaza el valor del nodo máximo en el nodo a eliminar (solo el valor, ya que no se quieren perder las conexiones que tiene el nodo a eliminar), se libera el nodo máximo y

1 error3 warnings se regresa el hijo izquierdo, En el caso del AVL, también al regreso se asegura de hacer las rotaciones necesarias.

Por otro lado, para el árbol splay no se empleo recursión y se mantuvieron punteros hacia los padres, si no que se "simuló"la recursión con ciclos `while`. Se hizo de esta manera dado que el regreso de la recursión estaría un tanto desperdiciado, no se podría hacer la operación de *splay* durante el regreso (como en los árboles AVL), dado que hay que revisar a los "abuelos"; se tendría que hacer una vez más desde la raíz. Por esa razón. justo después de ciclo while

(por ejemplo, se halló un nodo vacío para inserción) se hace la operación de inserción (o eliminación) y la operación de textitsplay en ese nodo. La implementación se basó en la sugerida por Wikipedia [4].

La operación de *splay* puede ser implementada de varias maneras; por ejemplo, desde la cabeza hasta el nodo deseado (revisando los hijos), o desde el nodo en cuestión hasta la raíz; puede hacerse mediante recursión o ciclos. Esta última fue la que se implementó; ya que como se mencionó, después del ciclo que va desde la raíz hasta el nodo, y con el *splay*, se manda este nodo y se hacen las rotaciones necesarias (revisando los padres y abuelos) hasta que el nodo llegue a la raíz.

Un "error" que surgió mucho en esta implementación, fue el que no se estaban actualizando bien las relaciones entre padres-hijos al hacer rotaciones, además de checar que no sean `NULL`. Por lo que se debe tener mucho cuidado con estas relaciones.

En cuanto a la operación de eliminar para el árbol splay, se ejecuta la operación de *splay* sobre el nodo a borrar, dejando sus hijos izquierdo y derecho. Si existe el hijo izquierdo, se hace un *splay* en el nodo con el elemento más grande, de tal manera que esté en la cabeza y tenga disponible para el hijo derecho (dado que es el máximo, todos los otros nodos son menores y están a la izquierda), ahí es donde se junta el subárbol derecho, como hijo derecho del subárbol izquierdo. Si no existe, el subárbol izquierdo, solo se regresa el árbol derecho.

Para todos los casos, para liberar el árbol, se empleo la estrategia de viajar por el árbol "post-order" liberando cada nodo.

Como referencia y comprobación, se pueden encontrar visualizaciones de los árboles en [5].

III. Resultados y discusión

Para comparar el rendimiento de los tres tipos de árboles, se revisaron tres escenarios, aumentando el número de elementos N:

- Insertar y eliminar elementos ordenados de forma creciente.
- Insertar elementos ordenados de forma creciente y eliminar de manera decreciente.
- Insertar y eliminar elementos aleatorios.

Para la secuencia aleatoria solo se usó `rand()` para que siempre se usara la misma secuencia.

Los resultados para el BST se muestra en la Figura 2. Se esperaba que este tuviera el peor rendimiento por lo menos (en los dos primeros escenarios) debido a que no se balancea y tiene un comportamiento similar al de una lista ligada. Sin embargo, tuvo un aún peor comportamiento, el programa falló con $N = 70000$ para el escenario creciente-creciente y con $N = 45000$ para creciente-decreciente. Esto se atribuye a un error de *stack overflow*, al ser similar a una lista ligada, se tenía que lidiar con alrededor de N recursiones. Por otro lado, en el escenario de números aleatorios, al naturalmente balancearse de alguna manera tuvo tiempos de ejecución satisfactorios.

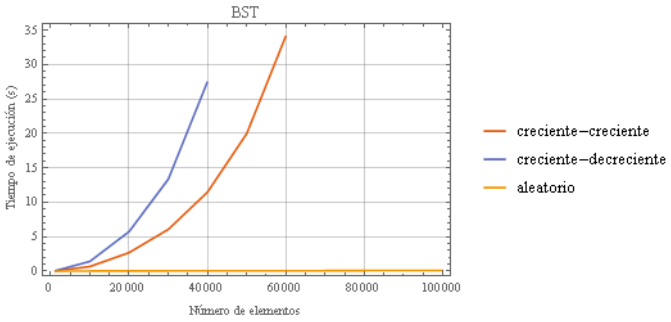

Figura 2: Resultados para el árbol binario de búsqueda (BST).

Los resultados para el árbol AVL se muestra en la Figura 3. Tal y como se esperaba, al ser un árbol auto-balanceado tuvo un mucho mejor comportamiento que el BST, por lo que se lograba el comportamiento de $O(\log n)$. No obstante, no se preveía que el escenario de números aleatorios fuera más lento que los otros, ya que en principio serían menos rotaciones a hacer, a comparación de los otros que se tenían que hacer rotaciones en prácticamente cada iteración. Se atribuye a que al ser aleatorio, muchos elementos no se encontraban, y tenía que buscar en todos los niveles.

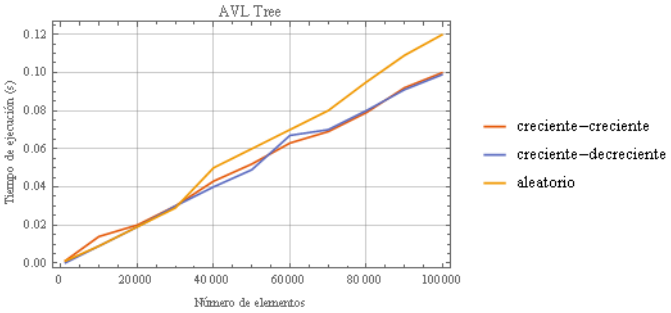

Figura 3: Resultados para el árbol AVL.

Los resultados para el árbol Splay se muestran en la Figura 4. El comportamiento de este fue un tanto sorprendente para los dos primeros escenarios, en donde no se apreciaba gran diferencia en el tiempo de ejecución en el rango de 1k - 50k elementos; incluso al tratar con $N = 1$M se tardó alrededor de 0.5 s y 0.25 s para el primer y segundo escenario. Se le atribuye a la propiedad característica de los árboles splay, rápido acceso a los elementos reciente, y al estar ordenados estaban muy cercanos unos de otros, por lo que no habría que revisar hasta el fondo del árbol. Para el escenario de números aleatorios, tuvo resultados comparables con los otros árboles.

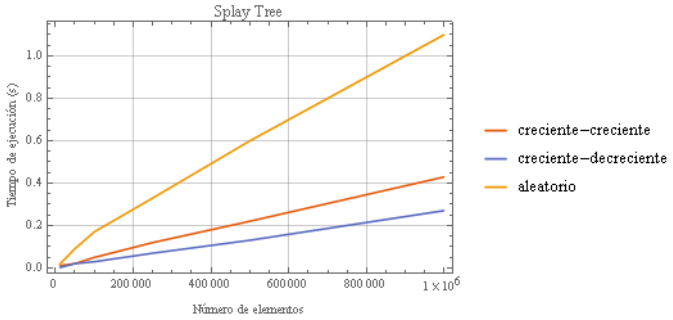

Figura 4: Resultados para el árbol splay.

En la Figura 5 se presenta una comparación entre los tres tipos de árboles para el escenario de números aleatorios. Se observa que el BST es el que muestra un mejor comportamiento, al insertarlo de manera aleatoria se balancea naturalmente sin tener que hacer rotaciones, al contrario del árbol AVL que está balanceado estrictamente, se tienen que hacer operaciones "extras" de rotación. En cuanto al árbol splay, al tener la ventaja de tener rápido acceso a los elementos recientes, al accesar aleatoriamente, se pierde por completo esta ventaja.

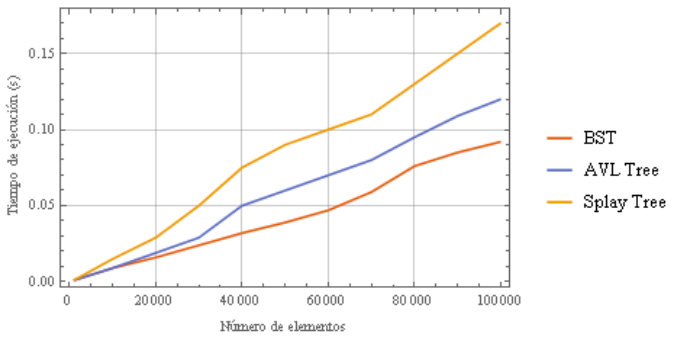

Figura 5: Comparación de resultados para el escenario de números resultados entre los tres tipos de árboles binarios.

Al contrario, en el escenario de creciente-creciente (véase Figura 6), el árbol splay es el mejor entre los tres, que como ya se mencionó, este escenario se aprovecha de la ventaja de rápido acceso de los elementos más recientes.

Pero, tal vez no se pueda tener una comparación tan justa con los otros árboles, dado que tuvieron implementaciones diferentes, el splay se implementó con ciclos, mientras que los otros dos con recursión, la cual puede presentar algunas limitaciones, al tener que manejar en mayor medida el stack.

Cabe mencionar que para una evaluación más justa en el caso de los números aleatorios contra los otros casos, se debió haber usado el mismo conjunto de números insertados para eliminar (solo desordenando de alguna manera la secuencia); y no solo con el comando de `rand()`, provocando que al eliminar no se encontraran muchos elementos. A pesar de esto, los tres árboles se evaluaron bajo las mismas condiciones (misma

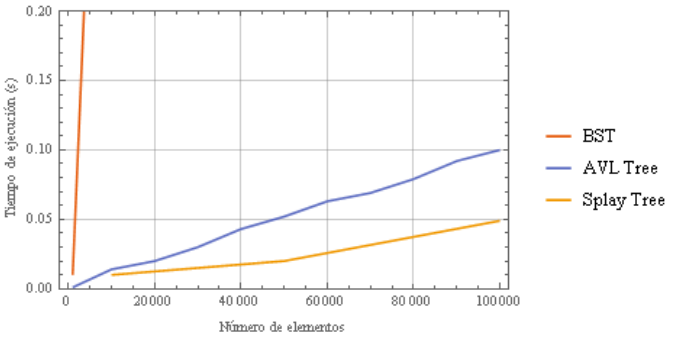

Figura 6: Comparación de resultados para el escenario de creciente-creciente entre los tres tipos de árboles binarios.

secuencia de números aleatorios), por lo que la comparación de estos es en cierta manera más justa.

IV. Conclusión

Los árboles binarios son una estructura de datos eficiente para almacenar elementos y mantenerlos ordenados con un tiempo de ejecución promedio de $O(\log n)$, a pesar de que consuman una mayor cantidad de recursos en cuestión de memoria, por el uso de punteros sobre todo.

Ahora, cada árbol presenta sus ventajas según la situación y la implementación. Los BST son los mejores cuando las entradas son aleatorias, ya que naturalmente se balancean sin hacer las operaciones extras de rotaciones; al contrario de los árboles AVL, que si bien tienen un desempeño satisfactorio, debido a las rotaciones tardan más que los BST (dadas las entradas aleatorias); y los árboles splay fueron los de menor rendimiento en este escenario, debido a que no se aprovecha la propiedad de acceso rápido a elementos recientes.

Por otro lado, en el caso de insertar y eliminar elementos de forma creciente-creciente y creciente-decreciente. Los BST tuvieron un rendimiento deplorable, porque se comportan como listas ligadas, y al estar implementados con recursión, se obtiene errores de *stack overflow*, una implementación con ciclos hubiera sido más estable. Los AVL, tuvieron un comportamiento comparable al caso de los aleatorios. Mientras que los splay, fueron bastante más superiores para elementos ordenados (sobre todo en el caso creciente-decreciente).

En general, los BST son los mejores para entradas aleatorias, los árboles splay para elementos ordenados, mientras que los árboles AVL tienen un comportamiento más balanceado en todos los casos.

En este proyecto, también surgió la situación de usar recursión vs. usar ciclos, que aunque la recursión nos permite una solución más "elegante" al simplificar la situación (en este caso, el no tener que mantener punteros explícitos con los padres), puede tener mayor "sobrecoste"(*overhead*) que un ciclo `while`, y se corre el riesgo de que se presente la situación de *stack overflow*, por lo que para asegurar

estabilidad en el caso de los BST, sería más conveniente usar la segunda opción con ciclos.

Referencias

[1] Nick Parlante. (). "Binary Trees," Stanford CS Education Library, dirección: http://cslibrary.stanford.edu/110/BinaryTrees.html (visitado 14-05-2021).

[2] (). "Binary Trees," Maynooth University Department of Computer Science, dirección: http://www.cs.nuim.ie/~dkelly/CS100-2/Binary%20Trees.htm (visitado 14-05-2021).

[3] *AVL tree*, en *Wikipedia*, Page Version ID: 1015256971, 31 de mar. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=AVL_tree&oldid=1015256971 (visitado 14-05-2021).

[4] *Splay tree*, en *Wikipedia*, Page Version ID: 1009835650, 2 de mar. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Splay_tree&oldid=1009835650 (visitado 14-05-2021).

[5] (). "Binary Search Tree Visualization," University of San Francisco, dirección: https://www.cs.usfca.edu/~galles/visualization/BST.html (visitado 14-05-2021).