Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 1: Estudio de Eficiencia sobre Árboles de Búsqueda

Diseño y Análisis de Algoritmos - CC4102-1

Integrantes: Pablo Gutiérrez
María Antonia Hernández
Sebastián Salinas
Profesor: Gonzalo Navarro
Auxiliar: Antonia Labarca
Ayudantes: Claudio Zamorano
Javier Oliva
Matías Rojas
Nicolás Canales

Fecha de entrega: 11 de noviembre de 2021

Índice de Contenidos

Índice de Figuras

Índice de Tablas

1. Introducción

A partir de distintas distribuciones de datos generadas aleatoriamente, se desea estudiar que implementación de árboles de búsqueda es la más adecuada y eficiente para realizar una secuencia de operaciones, también generada aleatoriamente. La secuencia consiste en operaciones de inserción, búsqueda exitosa y búsqueda infructuosa, siempre comenzando desde un árbol vacío con una inserción.

Para ello, se implementan árboles conocidos, los cuales son: árbol de búsqueda binaria clásico (ABB), un árbol AVL, 3 implementaciones de árboles B, con B=16, B=256 y B=4096, además de un árbol Splay.

Luego, se crean secuencias de operaciones para ser puestas a prueba en los experimentos, estas secuencias poseen 10^6 operaciones. Los valores a utilizar en las secuencias son enteros positivos de 32 bits, lo cual equivale a números entre 0 y 2^{32}. Se crea una secuencia aleatoria donde los elementos a insertar o hacer búsqueda infructuosa se generan de forma aleatoria y sin un orden en específico. Otra secuencia es creciente, donde los elementos son generados aleatoriamente, pero estos elementos tienden a crecer. Y finalmente se crea una secuencia sesgada donde las búsquedas exitosas se eligen de forma no uniforme entre los elementos ya insertados.

En la fase experimental se ejecuta cada secuencia en los distintos tipos de árboles creados. Se miden los tiempos de ejecución en cada caso para luego compara los resultados de distintos árboles, dada una misma secuencia.

Para la realización de los experimentos se utiliza el lenguaje C dada la eficiencia que proporciona. Se ocupa como compilador gcc. Y se trabaja en Debian bajo una máquina virtual *VirtualBox*, de manera de minimizar el tiempo de ejecución.

Todo esto se hace con el objetivo de comprobar si en la práctica, los órdenes aprendidos teóricamente coinciden con los experimentales, para distintos tamaños de árboles. Además de ver que estructura de árbol es más conveniente en las distintas secuencias de operaciones.

2. Implementación

Se implementan distintas versiones de árboles de búsqueda. Cada cual, con una construcción distinta y sus propias características favorables y desfavorables, teniendo así que cada estructura, debido a su comportamiento, tenga un orden característico para realizar operaciones de inserción y búsqueda.

2.1. ABB clásico

Para la implementación de un ABB clásico se utiliza una estructura *ABBNode* la cual consta de:

1. El valor que almacena el nodo.
2. Un puntero al hijo izquierdo (nodo izquierdo).
3. Un puntero al hijo derecho (nodo derecho).

Se crea una función auxiliar *createABBNode* la cual genera un nuevo nodo. En éste se inserta el valor y se inicializan los punteros en *NULL*. Esta es la representación más simple de un ABB con un elemento adentro.

La función *ABB_insert* ingresa un elemento al árbol de manera recursiva, en la cual se decide si se agrega a la izquierda o derecha del árbol de acuerdo a el valor del nuevo elemento a insertar. La función tiene como caso base el llegar a la hoja. Esta implementación no repite elementos en un mismo ABB (lo cual es lo que se menciona en el enunciado cuando se hace el supuesto de que los valores no serán repetidos).

La función *ABB_find* busca un elemento x en un ABB, retornando el puntero al nodo que lo contiene o *NULL* si falla en la búsqueda. Esta función es recursiva siguiendo nuestro árbol de acuerdo al valor que se quiera buscar.

Finalmente se agregan *ABB_preorden*, *ABB_inorden* y *ABB_postorden* para mostrar los ABB's que se generen durante toda la tarea, también pueden usarse para hacer inspección de los experimentos.

Con respecto a los órdenes en esta estructura, se tiene que las operaciones de inserción y búsqueda en un ABB construido a partir de n claves aleatorias requieren $O(logn)$ operaciones en el caso medio y $O(n)$ en el peor de los casos, esto ocurre cuando el árbol está totalmente desbalanceado y se asemeja a una lista secuencial.

2.2. AVL

Para la implementación del AVL se utiliza la estructura *AVLNode* que representa a un nodo de un árbol AVL. Esta estructura consta de 4 elementos, los cuales son:

1. El valor que almacena el nodo.
2. Un puntero al hijo izquierdo (nodo izquierdo).
3. Un puntero al hijo derecho (nodo derecho).
4. La altura actual del nodo.

Se creó la función *createAVLNode*, la cual genera un nodo con el valor dado como parámetro, inicializa los punteros a los subárboles en nulo y coloca la altura actual en 1.

Además, como un AVL es un árbol de búsqueda binaria auto equilibrado donde la diferencia entre las alturas de los subárboles izquierdo y derecho no puede ser más de uno para todos los nodos, se crearon un par de funciones auxiliares para manejar este comportamiento.

- La función *height*, que retorna la altura actual del nodo dado como parámetro.
- La función *max*, que retorna el máximo entre los dos números dados como parámetro.
- Las funciones *leftRotate* y *rightRotate* que se encargan de realizar rotaciones hacia la izquierda y derecha respectivamente, con el objetivo de balancear el árbol según sea el caso y mantener el invariante descrito anteriormente.
- La función *getBalance* que retorna el balance del nodo (la diferencia entre las alturas del hijo izquierdo y el hijo derecho del nodo), para verificar si se cumple el invariante del árbol.

Todas estas funciones auxiliares son utilizadas al momento de realizar la inserción de un elemento con la función *AVL_insert*, para que dado el caso se apliquen las rotaciones necesarias y se cumpla que la diferencia entre las alturas de los subárboles izquierdo y derecho no sea más de uno para todos los nodos. Además, de manera similar a lo descrito para los ABB's, para la búsqueda se utiliza la función *AVL_find*, que busca un elemento x en un AVL, retornando el puntero al nodo que lo contiene o *NULL* si falla en la búsqueda.

Por último, se agregaron las funciones *AVL_preorden*, *AVL_inorden* y *AVL_postorden* para mostrar la estructura de los AVL's que se generan y también para realizar *debugging* de los experimentos.

Dicho esto, debido al comportamiento de la estructura y de los balanceos que esta realiza, se asegura que la altura de un árbol AVL es $O(logn)$. Esto implica que la inserción y la búsqueda en un AVL tiene costo en el peor caso de $O(logn)$.

2.3. Árboles B

Se crean 3 implementaciones de árboles B con B=16, B=256, B=4096, donde todas comparten una base común.

Los nodos de un árbol B poseen:

1. Una lista de llaves de largo B-1 ordenados crecientemente.
2. Un contador para saber cuántas llaves tiene el nodo actualmente.
3. Un puntero al nodo padre.
4. Una lista de punteros a los nodos hijos.

La función *createBTree* genera un nodo con la llave dada e inicializa el contador de llaves en 1.

Para insertar un valor al árbol se usa la función *BTree_insert* la cual recibe la raíz y el valor a insertar.

Los árboles B son árboles de búsqueda auto equilibrados por lo que al agregar un valor debemos asegurar que todas las hojas se encuentren a la misma altura. Además se cumple que un nodo tiene como máximo B hijos, lo cual implica que, un nodo tiene como máximo B-1 llaves.

Dado esto, al momento de insertar se debe encontrar primero la hoja correcta donde poner la llave, de acuerdo a la construcción del árbol, y luego verificar si la hoja excedió los B-1 valores que puede tener. De haberse excedido se hace un llamado a la función *BTree_split* la cual reordena los nodos de forma que el árbol cumpla las propiedades correspondientes. Para reajustar un nodo que llega a las B llaves, se busca la mediana entre las llaves que posee, luego se crea un nuevo nodo el cual posee los valores mayores a la mediana y otro con los valores menores. El valor correspondiente a la mediana sube al nodo padre. En caso de que, al subir al nodo padre, el padre quede con B llaves, se realizará el split recursivamente.

Para la búsqueda se crea la función *BTree_find* la cual recorre desde la raíz del árbol en búsqueda de la llave dada. Para ello revisa el arreglo con las llaves del nodo en el que se encuentra y de ser necesario baja por alguno de los hijos que se encuentra en el arreglo de hijos del nodo. En caso de encontrar la llave retorna el nodo en el que se encuentra. Y de llegar hasta las hojas y no encontrar el nodo, se retorna *NULL*

Finalmente se agrega *BTree_inorder* para poder visualizar mejor que los árboles se generen de forma correcta.

Cabe mencionar que como todas las hojas del árbol B están a un mismo nivel, se tiene que la altura es $O(log_B N)$. Por lo cual, la inserción y la búsqueda es $O(log_B N)$ en los mejores de los casos, cuando una inserción no genera muchos split recursivos y una búsqueda no hace que, con el objetivo de encontrar el valor pedido, se deba iterar el arreglo completo con las llaves de nodos de forma lineal.

2.4. Árboles Splay

Para implementar el Splay Tree se utiliza una estructura de 3 elementos:

1. El valor que almacena el nodo.
2. Un puntero al hijo izquierdo (nodo izquierdo).
3. Un puntero al hijo derecho (nodo derecho).

Se crea una función llamada *createSplayNode* que, como su nombre lo indica, crea un nodo splay que se utilizará para las siguientes implementaciones.

Se implementan las funciones auxiliares *rightRotation* y *leftRotation* las cuales sirven para mandar el valor de un nodo a la rama derecha o izquierda, modificando los demás elementos para que sigan siendo consistentes.

La función *splay_find*, que recibe un doble puntero a un splayNode y el elemento buscado. Esta sigue 4 patrones diferentes al buscar, los cuales incluyen rotaciones de acuerdo a la posición del elemento que se está buscando, estos casos son:

1. El elemento se encuentra dos hijos a la izquierda (*Zig-Zig*).
2. El elemento se encuentra un hijo a la izquierda y otro a la derecha (*Zig-Zag*).
3. El elemento se encuentra un hijo a la derecha y otro a la izquierda (Zag-Zig).
4. El elemento se encuentra dos hijos a la derecha (*Zag-Zag*).

Se asumirán las transformaciones y su orden como materia vista en otros cursos para no extenderse con una explicación. El elemento se encuentra al menos dos niveles bajo la raíz gracias a la recursión que se implementa. Recordemos que, al buscar un elemento, este se modifica para llegar a la raíz del árbol.

Además, se agregan más casos de borde que implican hacer una rotación o no y el caso de un árbol nulo.

Para *splay_insert* se procede a insertar como si fuera un ABB clásico, con la diferencia de que luego se realiza una búsqueda del elemento ingresado y así posicionar en el nodo raíz el valor insertado.

Finalmente se definen funciones para implementar Inorder, preorder y post-orden para el análisis de los experimentos al momento de ejecutarlos en local.

Por la forma en la que se realizan las operaciones es los Splay Trees, éstas tienen un costo amortizado de $O(logn)$ por operación.

3. Experimentación

Como se mencionó en la introducción, el propósito de la tarea consiste en implementar, comparar y estudiar la eficiencia de algunas variantes de los árboles de búsqueda, para así concluir sobre los desempeños que estos tienen sobre distintas distribuciones de datos. Es así que, para realizar los experimentos y obtener los tiempos de desempeño de las distintas estructuras ante una secuencia de operaciones, se generó el script *experiments.c*, el cual se procederá a explicar cómo funciona.

Primero, se genera una secuencia de $n = 10^6$ operaciones, dentro de las cuales se tienen inserciones, búsquedas exitosas y búsquedas infructuosas. Para generar una secuencia válida para los experimentos se debe comenzar por una inserción (los experimentos se inician desde un árbol vacío, se inserta al menos un elemento y se procede con las demás operaciones) y luego continuar con distintas operaciones elegidas al azar basándose en la probabilidad que estas tienen dada por el enunciado:

- Probabilidad de inserción = 1/2
- Probabilidad de búsquedas exitosas = 1/3
- Probabilidad de búsquedas infructuosas = 1/6

Por lo cual, la idea que se siguió para realizar una secuencia de operaciones válida fue generar $n = 10^6$ números aleatorios del 1 al 6, asignando una operación de:

- Inserción si sale un 1 o un 2 o un 3 (probabilidad de 1/2).
- Búsqueda exitosa si sale un 4 o un 5 (probabilidad de 1/3).
- Búsqueda infructuosa si sale un 6 (probabilidad de 1/6)

Cumpliendo de esta forma una secuencia aleatoria de operaciones válida, con las probabilidades pedidas.

Luego, para continuar con los experimentos y solo considerar los tiempos que tarda cada operación en cada uno de ellos, se tomó la decisión de pre computar los valores aleatorios sobre los cuales se realizarán las distintas operaciones de la secuencia. Este pre calculo se realiza en la función *generate_random_values*, la lógica que sigue la función es la de ir guardando en arreglos para los 6 experimentos los valores que se usarán para cada operación de la secuencia.

Para el experimento aleatorio se seleccionan valores aleatorios entre 0 y 2^{32} (verificando que no estén previamente insertados). Por otro lado, para los experimentos crecientes también se generan números aleatorios en ese rango, pero se dividen por 2^{32}, para crear un factor de ponderación el cual es aplicado sobre el valor de k y así terminar generando un número aleatorio entre 0 y k, para posteriormente sumarle la cantidad de nodos que se llevan insertados, haciendo así que los valores seleccionados tiendan a ser crecientes. Por último, para los experimentos sesgados, se generan números aleatorios en el mismo rango antes descrito y a la vez se guardan en arreglos los pesos asociados a estos valores según la función que corresponda ($p(x) = x$, $p(x) = \sqrt{x}$, $p(x) = ln(x)$), para que en operaciones de búsqueda se tenga fácil acceso a estos pesos y se logre elegir de manera más fácil el elemento con mayor probabilidad $p(x)/P$.

Una vez computado los valores que se utilizarán en los experimentos se procede a ejecutarlos mediante la función *run_experiments*, aquí se inicializan los 6 árboles y se les aplican la secuencia de operaciones, que es la misma para cada estructura. Sin embargo, los valores con los cuales se realiza la secuencia para los distintos experimentos son diferentes, ya que como se mencionó antes, para los experimentos se generan valores aleatorios de distintas formas.
Luego de un intervalo de $1000 = 10^3$ operaciones se mide el tiempo que tardó la estructura (sobre la cual se está corriendo el experimento) en realizar las operaciones. Por lo cual la ejecución de los experimentos irá midiendo tiempos de desempeño en intervalos de 1000 operaciones y guardando en archivos de texto los tiempos medidos en cada estructura y en cada experimento, generando de esta forma 36 archivos distintos (6 estructuras sometidas a 6 experimentos) con mediciones de tiempos transcurridos cada 1000 operaciones realizadas.

Ya con los tiempos medidos y almacenados en los archivos de texto se procede a utilizar un script de Python llamado *graficos.py*, el cual se encarga de generar:

- 6 gráficos de tiempo en función de los intervalos de 1000 operaciones (uno por cada experimento).
- Los 6 gráficos anteriores, pero en escala logarítmica para facilitar su comprensión y análisis.
- Los 6 gráficos logarítmicos, pero añadiéndole ajustes de curvas siguiendo el modelo teórico de los tiempos esperados en función de los tamaños de los árboles.

Estos gráficos se pueden apreciar en la sección de presentación de resultados.

4. Presentación de Resultados

Antes de iniciar la presentación de los resultados cabe mencionar nuevamente que luego de realizar los experimentos, se notó un resultado anormal en los tiempos arrojados por el Splay Tree. Estos resultados no son correctos, dado que la implementación posee una falla al momento de hacer ciertas búsquedas que, a nuestro criterio, son aleatorias. Entrando más en detalle y tecnicismos la implementación actual de la estructura pierde la referencia a algunos nodos y termina eliminando algunas veces todos los hijos a la derecha al momento de hacer rotaciones a la izquierda y lo mismo se observa con los hijos de la izquierda con rotaciones a la derecha. Debido a este error la estructura no crece demasiado en cantidad de elementos ya que apenas toca una búsqueda en la secuencia de operaciones es probable que se pierda la referencia a algunos nodos. Este error explica el porqué de los tiempos de ejecución tan bajos, ya que el árbol se mantendrá la mayoría del experimento con una cantidad baja de elementos, provocando que las operaciones sean bastante rápidas y no crezcan en tiempo como las otras estructuras.

Dicho esto, no se considerarán los resultados obtenidos para esta estructura en los análisis, pero se decidió incluirlos de igual manera en algunos gráficos y realizar esta aclaración para demostrar que si bien la implementación de la estructura no está bien hecha, se pudo detectar el error tempranamente antes de realizar conclusiones incorrectas, haciendo que el análisis final de los experimentos no fuera totalmente erróneo.

4.1. Experimento Aleatorio

En el primer experimento se utiliza la secuencia de valores aleatorios, donde se puede observar que el peor desempeño fue por parte del B Tree 4096. Y en cambio el mejor es el resultado del B Tree 16.

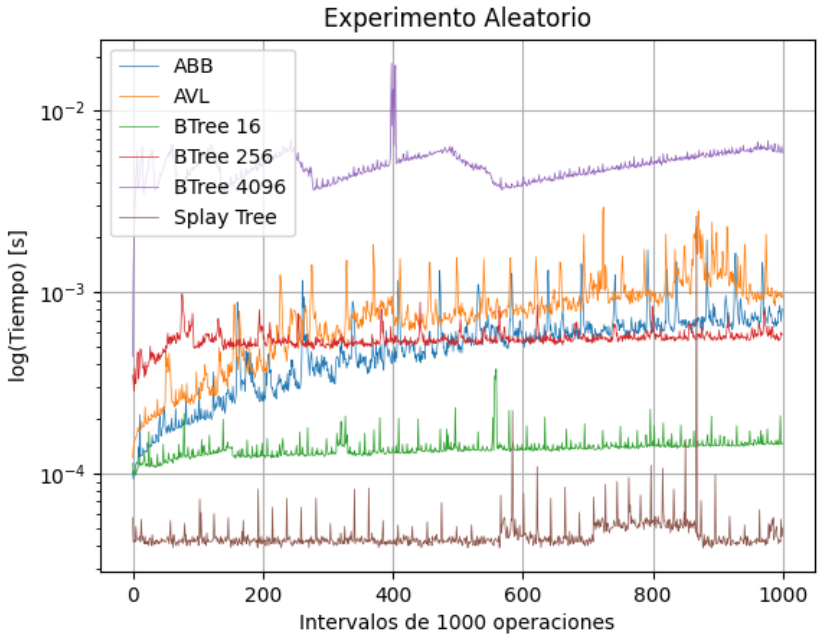

Figura 1: Experimento Aleatorio en escala logarítmica

Además, se calculan los tiempos amortizados de las operaciones en cada árbol con el objetivo de observar cual implementación de árbol es más conveniente cuando hay curvas que se intersectan (empates). Se observa que el tiempo amortizado de ABB es muy similar al de B Tree 256 al realizar 1 millón operaciones, pero al observar la curva podemos notar cual es más conveniente para distintas cantidades de operaciones. El ABB va creciendo de forma logarítmica a medida que se avanza en la secuencia de operaciones, mientras que el B Tree 256 se mantiene con un comportamiento un poco más constante a través de la misma.

Tabla 1: Tiempo amortizado de operaciones en experimento aleatorio

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	5.2×10^{-7}
AVL	8.0×10^{-7}
B Tree 16	1.3×10^{-7}
B Tree 256	5.5×10^{-7}
B Tree 4096	50.5×10^{-7}

También, se hace un ajuste de curvas según el modelo teórico de tiempos para cada árbol. En el caso de la secuencia de valores aleatorios se utiliza una curva logarítmica del tipo :

$$f(x) = (a * log(x + b)) + c$$

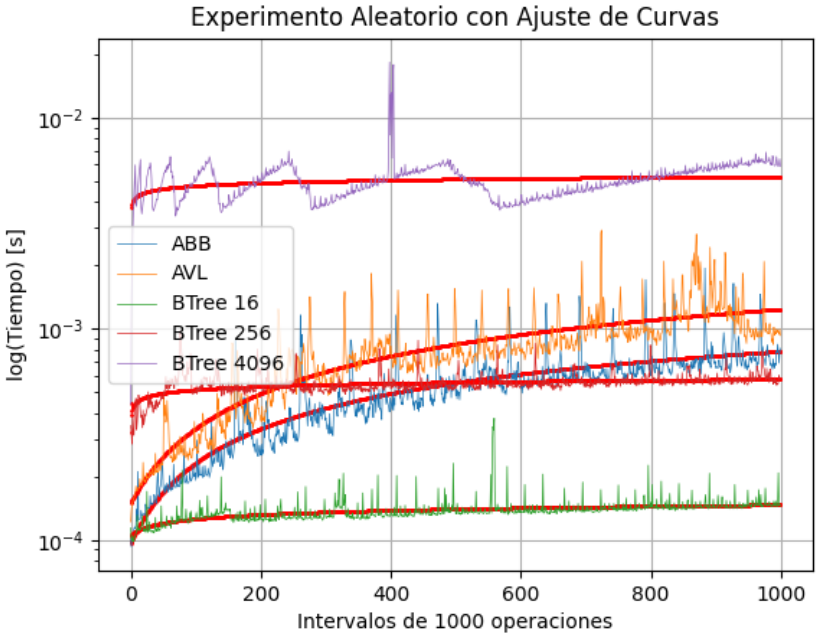

Figura 2: Experimento Aleatorio en escala logarítmica con ajuste de curvas

4.2. Experimentos Crecientes

Luego se experimenta con la secuencia creciente donde los resultados concuerdan con lo esperado teóricamente, dado que el ABB alcanza su peor caso, obteniendo un orden $O(n)$ para sus operaciones. Además, el B Tree 16 continúa siendo la estructura con los mejores resultados.

4.2.1. Experimento Creciente con factor=0.1

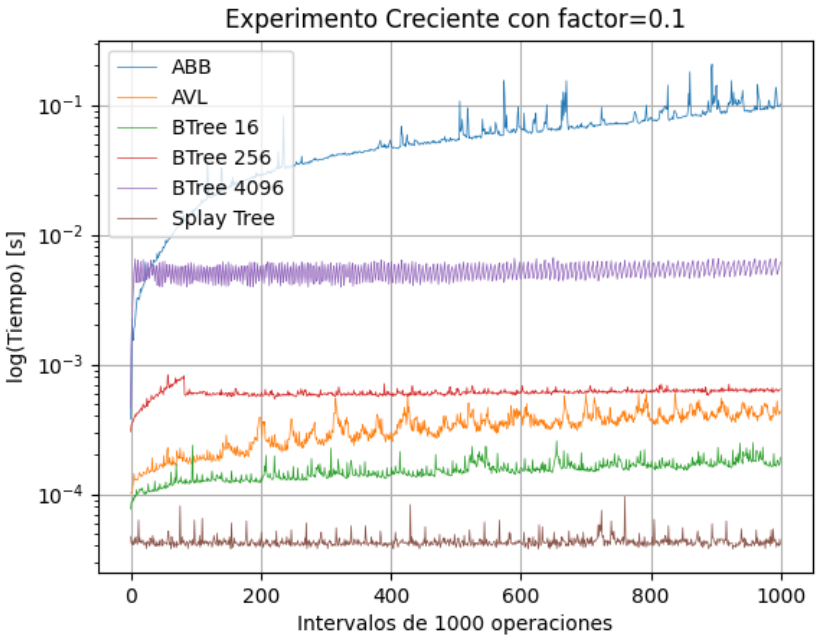

Figura 3: Experimento Creciente con factor=0.1 en escala logarítmica

Con respecto a los tiempos amortizados se nota un gran aumento del tiempo amortizado del ABB con respecto al experimento aleatorio. Pero los demás árboles mantienen tiempos similares, esto debido a que se auto balancean.

Tabla 2: Tiempo amortizado de operaciones en experimento creciente con factor 0.1

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	558.8 x 10^{-7}
AVL	3.2 x 10^{-7}
B Tree 16	1.5 x 10^{-7}
B Tree 256	6.0 x 10^{-7}
B Tree 4096	52.1 x 10^{-7}

Dado que el ABB se encuentra en su peor caso, para ajustar su curva se debe usar una función lineal del tipo:

$$f(x) = a * x + b$$

y las demás funciones utilizan la función logarítmica anteriormente mencionada.

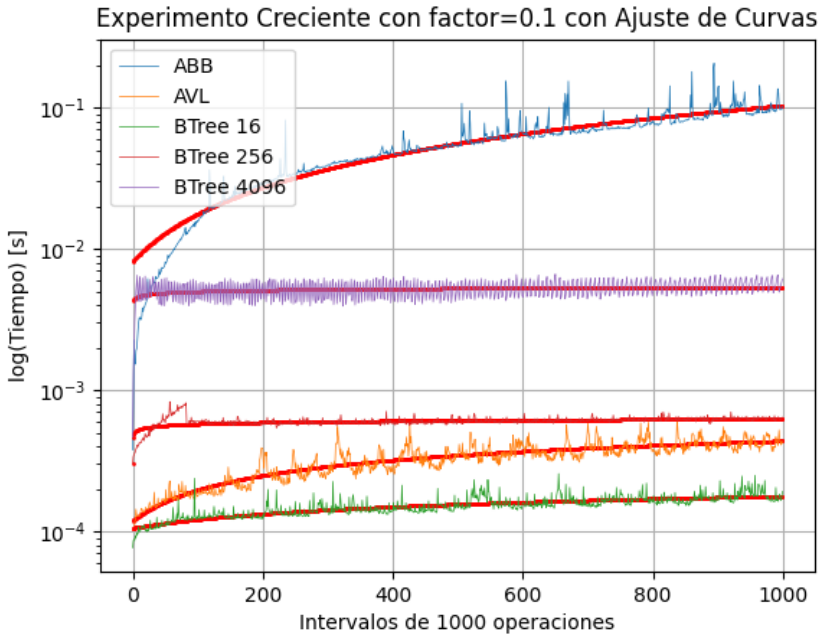

Figura 4: Experimento Creciente con factor=0.1 en escala logarítmica con ajuste de curvas

4.2.2. Experimento Creciente con factor=0.5

Al cambiar el factor y aumentar su valor se obtiene un rango más amplio en el cual se generan los números aleatorios para el experimento. Un factor de 0.5 permite que los valores aleatorios se generen de manera más holgada, lo que hace que los valores que tienden a ser crecientes escalen de forma más rápida, llegando a utilizar números más grandes para realizar las operaciones de forma más rápida que con un factor de 0.1, que es más ajustado. Sin embargo, se observa que al cambiar el factor y aumentar su valor los resultados varían levemente

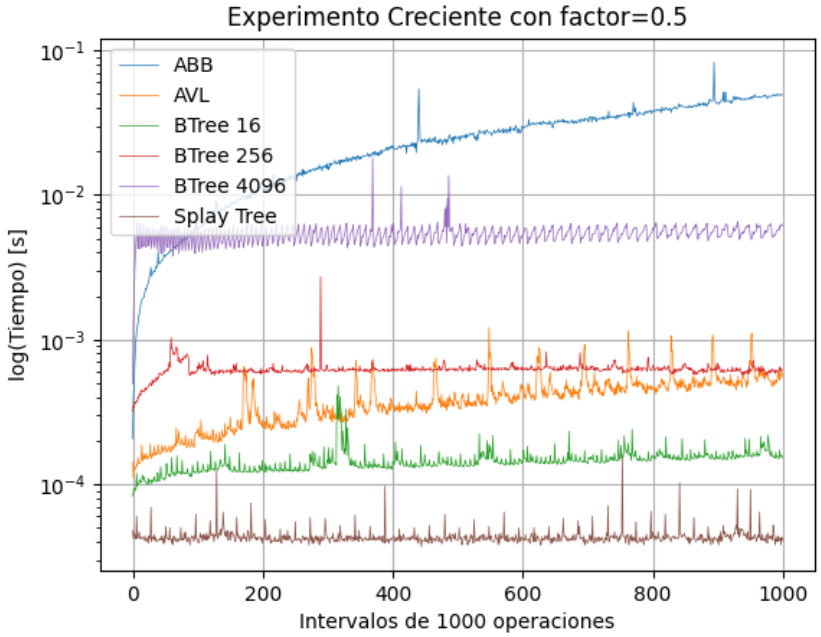

Figura 5: Experimento Creciente con factor=0.5 en escala logarítmica

Además, los tiempos amortizados se mantienen muy similares por excepción del ABB.

Tabla 3: Tiempo amortizado de operaciones en experimento Creciente con factor 0.5

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	250.0 x 10^{-7}
AVL	3.9 x 10^{-7}
B Tree 16	1.4 x 10^{-7}
B Tree 256	6.2 x 10^{-7}
B Tree 4096	53.6 x 10^{-7}

De manera similar al experimento anterior, para el ajuste de curvas se ocupa una función lineal para el ABB y para las demás funciones se utilizan la misma función logarítmica anteriormente mencionada.

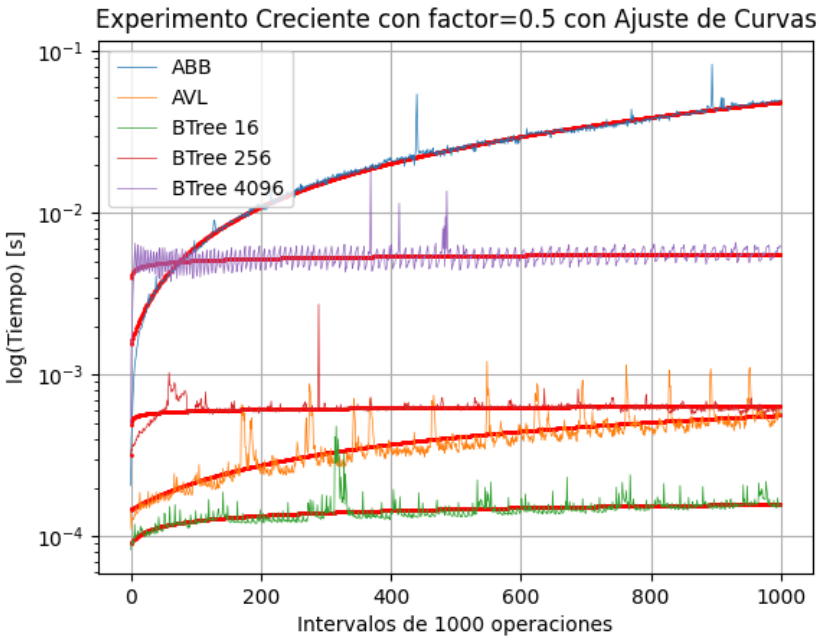

Figura 6: Experimento creciente con factor=0.5 en escala logarítmica con ajuste de curvas

4.3. Experimentos Sesgados

En los experimentos con secuencia sesgada se puede notar que independiente de la función ocupada para el peso el B Tree 256 y B Tree 16 mantienen curvas muy similares, y difieren en el corte que tienen con el eje y. Además, antes de las 600 mil operaciones el ABB es mejore que el B Tree 256, pero luego esto cambia ya que el árbol ABB se comporta de forma más lineal y el B Tree 256 de forma logarítmica.

4.3.1. Experimento Sesgado con $p(x) = x$

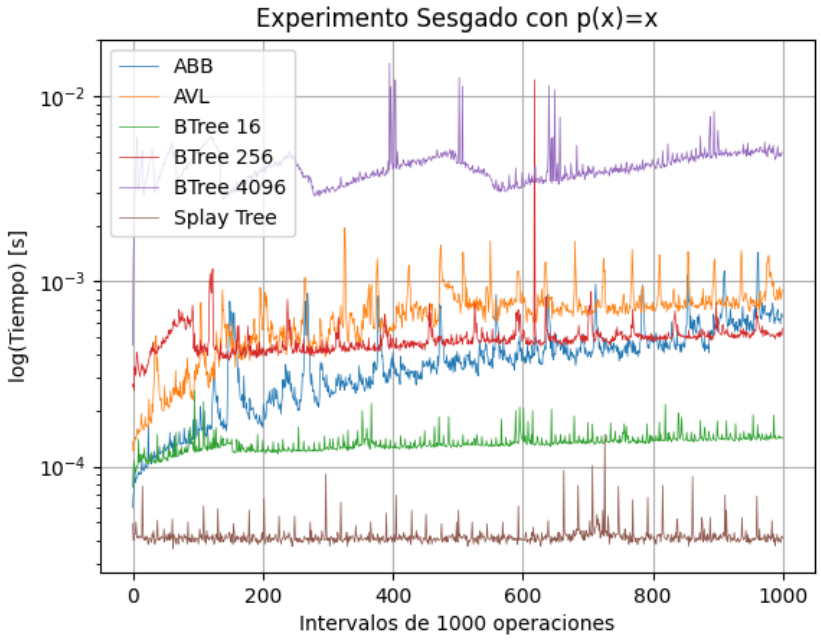

Figura 7: Experimento Sesgado con p(x)=x en escala logarítmica

Nuevamente el B Tree 16 posee los mejores tiempos amortizados.

Tabla 4: Tiempo amortizado de operaciones en experimento sesgado con p(x)=x

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	3.8 x 10^{-7}
AVL	6.6 x 10^{-7}
B Tree 16	1.3 x 10^{-7}
B Tree 256	4.9 x 10^{-7}
B Tree 4096	42.1 x 10^{-7}

Se hace un ajuste de curvas utilizando las funciones lineales y logarítmicas antes mencionadas.

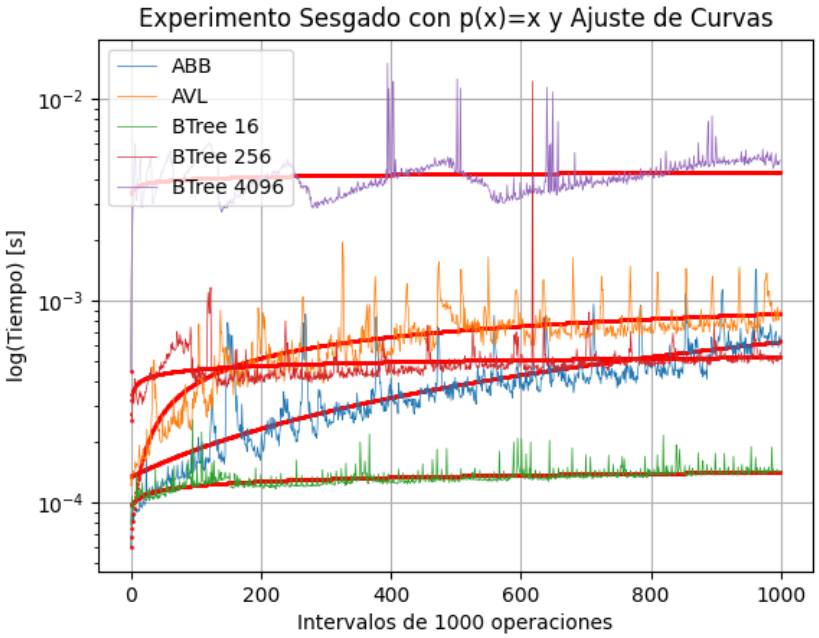

Figura 8: Experimento Sesgado con p(x)=x en escala logarítmica con ajuste de curvas

4.3.2. Experimento Sesgado con $p(x) = \sqrt{x}$

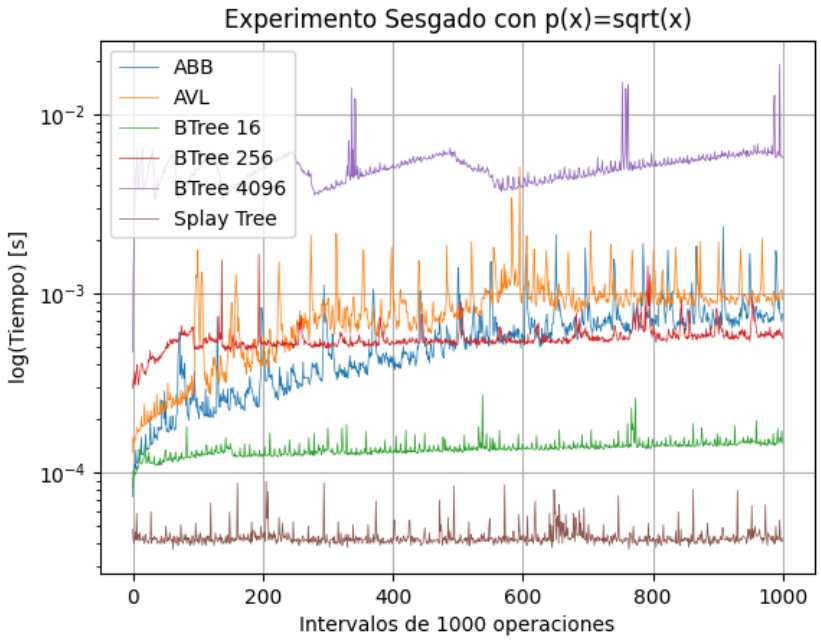

Figura 9: Experimento Sesgado con p(x)=$\sqrt{(x)}$*enescalalogarítmica*

Tabla 5: Tiempo amortizado de operaciones en experimento sesgado con p(x)=$\sqrt{(x)}$

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	5.5 x 10^{-7}
AVL	8.2 x 10^{-7}
B Tree 16	1.3 x 10^{-7}
B Tree 256	5.6 x 10^{-7}
B Tree 4096	50.9 x 10^{-7}

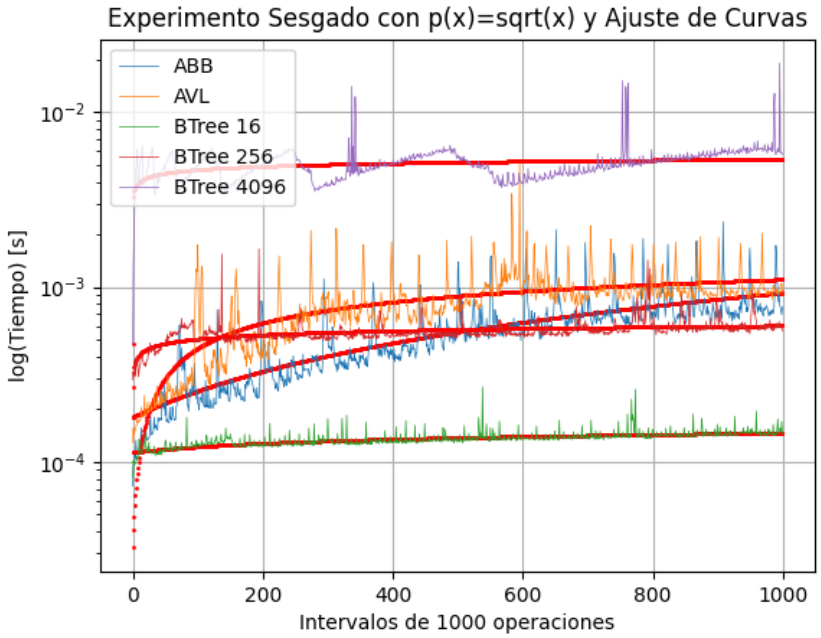

Figura 10: Experimento Sesgado con p(x)=$\sqrt{(}x)enescalalogarítmicaconajustedecurvas$

4.3.3. Experimento Sesgado con $p(x) = ln(x)$

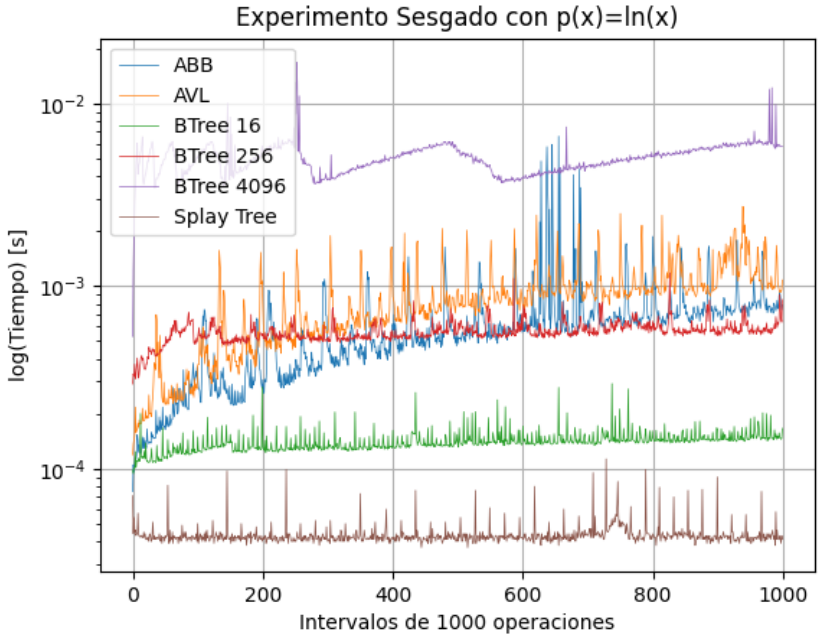

Figura 11: Experimento Sesgado con p(x)=ln(x) en escala logarítmica

Se observa que con $p(x) = ln(n)$, las curvas de tiempos comienzan a solaparse más y es menos claro que árbol conviene en cada caso. Así se puede observar los tiempos amortizados y notar que B Tree 16 continúa siento la mejor opción a realizar el millón de experimentos, pero en caso de menos de 600 mil el ABB sería más conveniente.

Tabla 6: Tiempo amortizado de operaciones en experimento sesgado con p(x)=ln(x)

Tipo de árbol	Tiempo amortizado por operación [s]
ABB	6.1 x 10^{-7}
AVL	8.3 x 10^{-7}
B Tree 16	1.4 x 10^{-7}
B Tree 256	5.6 x 10^{-7}
B Tree 4096	50.2 x 10^{-7}

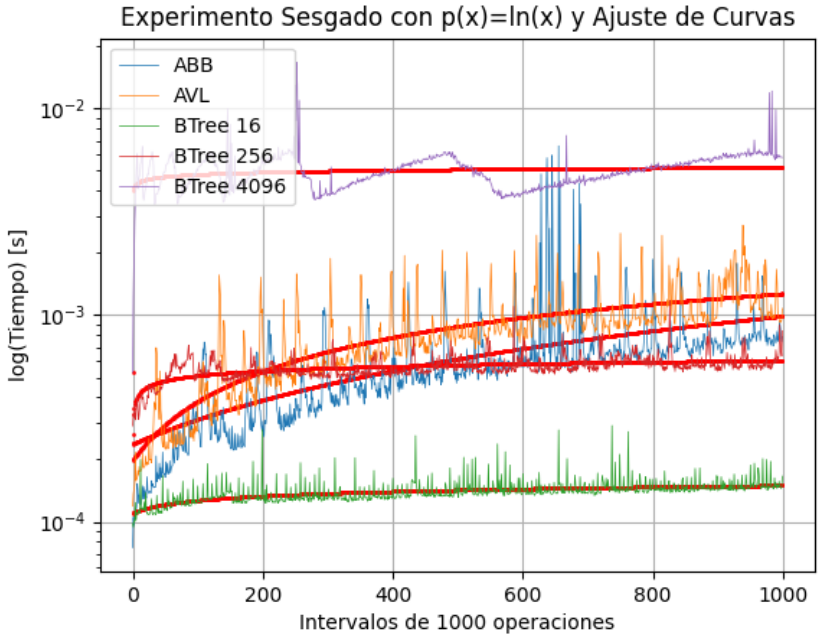

Figura 12: Experimento Sesgado con p(x)=ln(x) en escala logarítmica con ajuste de curvas

Con respecto a las funciones $p(x)$, se sabe que x crece más rápido que $sqrt$, y también que $sqrt$ crece más rápido que ln. Y al observar los gráficos, mientras menos crece la función $p(x)$ más conveniente es una ABB y menos B Tree 16. Junto a esto también se debe considerar la dificultad de implementar un B Tree, lo cual es más complejo que un ABB clásico.

Finalmente se nota que en todos los gráficos el árbol que toma menos tiempo en realizar las operaciones es el B Tree 16. Y en general el que más tiempo demora es el B Tree 4096, con excepción

del caso donde los datos son crecientes, donde es el ABB.

Además, notamos que los tiempos no suelen exceder los 10^{-2} segundos en el caso de los valores aleatorios y sesgados. Pero en el experimento creciente llega hasta un tiempo de 10^{-1} segundos en el caso del ABB.

5. Discusión

Los resultados obtenidos con el Splay Tree son incorrectos dada una mala implementación del árbol antes mencionada por lo cual sus resultados no serán considerados para la discusión.

5.1. Experimento 1: Datos Aleatorios

En la figura 1 tenemos el resultado del experimento aleatorio, en ella se pueden empezar a ver patrones que describen a cada uno de los árboles: el B Tree 4096 es el de mayor costo, con una forma de diente de sierra, que representan los splits. Una versión similar se observa con el B Tree 256 pero a menor escala y de manera mucho menos brusca respecto a lo anterior. Finalizando los B Trees, se tiene que el B Tree 16 es el más eficiente debido a la menor cantidad de valores en cada nodo. Esto dado que, en un B Tree, la búsqueda dentro del nodo se hace de forma lineal, y esta búsqueda se hace tanto cuando se quiere buscar como cuando se inserta un elemento, ya que se debe buscar la ubicación correcta dentro del nodo o el hijo que pueda llevar a esa ubicación.

AVL y ABB tienen una forma creciente de forma logarítmica, debido a que la inserción costará más a medida que va creciendo el árbol y por lo tanto sus alturas van aumentando. El AVL tiende a tomar un poco más de tiempo que el ABB, esto puede deberse a que se debe auto balancear.

Así, si se tiene una secuencia de 1 millón de operaciones con valores aleatorios, la mejor opción es escoger el B Tree 16 por sus tiempos de ejecución. Pero dada la poca diferencia que presenta con el ABB, y que el segundo tiene una implementación mucho más simple ya que no se auto balancea de ninguna manera, debe ser tomado en consideración al momento de escoger el árbol a utilizar.

Además, no se debe generalizar estos resultados para secuencia con mayor cantidad de operaciones, ya que se observa en la Figura 1 como al llegar a las 800 mil operaciones, el ABB deja de tener los segundos mejores tiempos ya que tiene una pendiente de crecimiento mayor.

Finalmente, se concluye que, en caso de tener secuencias de menos de 1 millón de operaciones, lo más recomendado es utilizar un ABB dada la simplicidad de su programación, y la poca diferencia en tiempo de ejecución con respecto a los demás árboles experimentados. Pero en caso de tener una secuencia de mayor magnitud, se puede considerar invertir tiempo en generar un B Tree ya que este le dará un mejor tiempo.

5.2. Experimento 2: Datos Crecientes

Para el experimento con el esquema de operaciones de datos creciente se puede apreciar que el ABB es la estructura que tarda más. Esto se debe a que gracias a la forma en que está hecho el experimento los valores con los que se realizan las operaciones tienden a ser crecientes. Por lo cual, como el ABB no tiene ninguna manera de balancearse o modificar su estructura, luego de un par de inserciones se tiene un nodo raíz el cual tendrá bastantes hijos a la derecha, y en el caso totalmente creciente ninguno será menor que el otro o existirá un nodo con elementos a la izquierda. Esto quiere decir que cada vez que se busque un elemento será mucho más difícil, pues estará a una profundidad mayor en el árbol y se terminará recorriendo la estructura linealmente y no logarítmicamente, realizando prácticamente una búsqueda secuencial. lo mismo pasa con la inserción, con la diferencia de que esta siempre tenderá a agregarse al final de la secuencia de nodos, provocando así que cada vez cueste más tiempo realizar las operaciones, esto explica el muy mal desempeño que tiene este árbol en comparación con los demás al momento de hablar de valores crecientes.

Con el resto de las estructuras se tiene un buen comportamiento, el cual corrobora lo que dice la teoría ya que se puede observar que gracias a que las demás estructuras son árboles autobalanceables no caen en este peor caso en el cual se recorre linealmente y no logarítmicamente.

Entrando más en detalle, por el lado de los árboles B, debido a que la cantidad de datos es creciente se pueden observar dientes de cierra en el B Tree 4096 que son más pequeños y no varían demasiado (en comparación con la imagen anterior) respecto a la cantidad de tiempo. El mismo patrón se puede observar en el B Tree de 256 y para el B Tree de 16, siendo este último el más eficiente de todos los árboles a testear.

Finalmente, vemos como el AVL no sufre lo mismo que el ABB gracias a su balanceo, de esta forma tenemos que el costo de inserción no será tan creciente como el ABB.

Ahora, cuando el factor cambia a 0.5 notamos un comportamiento similar a los valores, con la diferencia de que en general se pueden ver pequeñas variaciones en todas las curvas del gráfico.

Finalmente, llegamos a la conclusión de elegir el BTree 16, o en otro caso AVL, debido a que sus rendimientos son los más eficientes y por diferencia en comparación con los demás árboles, con la particularidad de que el tiempo en AVL va creciendo con más datos, se prefiere el BTree 16, debido a que su crecimiento es demasiado lento, con una pendiente poco inclinada.

5.3. Experimento 3: Datos Sesgados

En las figuras 7, 9 y 11 se observa cómo cambian las curvas para B Tree 4096, ABB y AVL. Esto se debe a la función p(x) la cual le asigna un peso a cada elemento insertado en el árbol.

Cuando $p(x) = x$, corresponde a un caso donde entre muchos números en un árbol se tienden a buscar los de mayor valor. Luego, en el caso de $p(x) = \sqrt{x}$, la probabilidad de buscar datos de gran valor está menos sesgada que en el primer caso. Y finalmente en $p(x) = ln(x)$ la probabilidad cambia muy poco entre valores grandes y pequeños. Esto debe ser tomado en consideración al momento de escoger un árbol, ya que de querer escoger un ABB, por su fácil implementación, en cada caso el rango donde este deja de ser tan efectivo es distinto.

Además, los árboles Splay, al igual que la mayoría de los árboles tienen un costo amortizado de $O(logn)$, pero en caso de secuencias de accesos a nodos con distinta probabilidad, es decir sesgados, su costo amortizado es $O(H)$ con H la entropía de las probabilidades. Dado esto, se hubieran esperado resultados donde, en el caso de $p(x) = x$, que presenta una mayor entropía entre las probabilidades, los tiempos no fueran tan bajos como en el caso de $p(x) = ln(x)$, donde la entropía disminuye considerablemente.

En caso de tener que escoger un árbol, se debe tomar en consideración la cantidad de operaciones que se quiere realizar. Esto debido a que el ABB, AVL y B Tree 256 intersectan sus curvas de tiempo, y para una mayor cantidad de operaciones B Tree 256 puede llegar a ser una mejor opción. Sin embargo, el B Tree 16 sigue siendo la mejor opción independiente de la cantidad de operaciones.

6. Conclusión

Se concluye que nunca es conveniente usar un B Tree 4096, esto debido a que cada tiene entre 2048 y 4095 llaves, y al momento de buscar entre ellas se hace de forma lineal. Esto no general un $O(n)$ dada la forma de árbol general que tiene, pero no llega a ser lo suficientemente eficiente como para compararse con los demás árboles del experimento.

También, los tiempos de los B Tree 16, en todos los experimentos fueron los menores. Esto nos hace entender de mejor manera el por qué son los árboles escogidos para obtener información desde memoria secundaria en los computadores.

Por último se nota que las curvas varían según la secuencia, por lo que, al momento de escoger un árbol, no existe uno mejor en todo ámbito sino que cada uno posee propiedades que le permiten ajustarse mejor a problemas distintos. Además, no se debe escoger solo por el tiempo que se demora en ejecutar una operación sino también en tiempo que se invertirá en programar el algoritmo para el árbol dado.

Se debe mencionar que los resultados obtenidos dependen del lenguaje escogido, la función usada para medir el tiempo, como también el ambiente y el compilador. Estos parámetros deben ser considerados al momento de escoger un árbol y ver cual se acomoda más a las necesidades del problema.