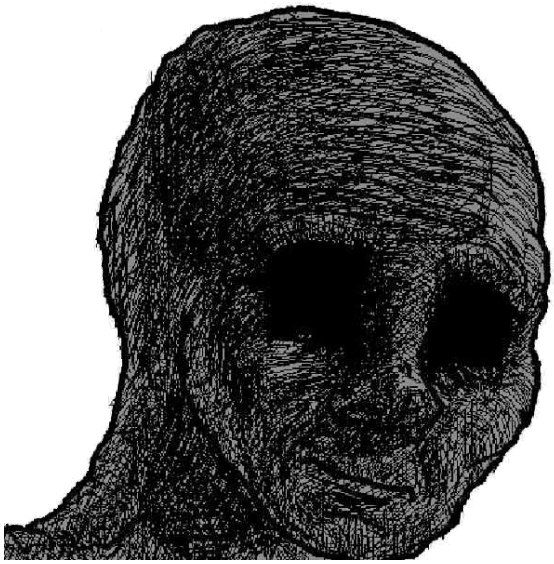
Tarea 1

CC4102-1 Análisis y diseño de algortimos



Profesor: Gonzalo Navarro

Estudiantes: Nicolás García
Javier Lavados
Lung Pang

Introducción

El fin de este trabajo es estudiar y analizar el desempeño de 6 estructuras de datos frente a cantidades de datos crecientes y con experimentos que varían en operaciones y probabilidades de apariciones. Las estructuras elegidas para este trabajo son variantes de árboles binarios, las cuales van desde la forma más simple (ABB) hasta estructuras que aseguran peor caso (AVL y BTrees) y que modifican la estructura por operaciones recientes (Splay Tree).

Por otro lado, los experimentos realizados para el estudio se basan en la aplicación de n (1.000.000) operaciones aleatorias (inserción y búsqueda) con valores de datos entre 0 y 2^{32}. Cada experimento variará la probabilidad de ocurrencia de sus valores con el fin de observar el comportamiento de cada estructura sometidas a distintas formas de datos, las cuales variarán desde datos equiprobables, hasta datos crecientes y sesgados.

En cada experimento se midieron los tiempos promedios de ejecución por cada 1.000 operaciones de cada estructura, lo cuál permite una visualización gráfica de los tiempos que toma cada estructura en función de las operaciones realizadas hasta el momento. Cada experimento fué repetido una única vez, debido al tiempo que supone la ejecución de estos tests y la generación de unos 36.000 puntos a graficar.

Los resultados de los experimentos arrojaron que las estructuras con peor desempeño son el árbol de busqueda binaria ABB y BTree de tamaño B=4096, pero bajo distintos tipos y probabilidades de datos.

Por un lado, el árbol de búsqueda binaria presenta mal rendimiento frente a experimentos con datos crecientes. Debido a que los arboles de busqueda binaria no aseguran peor caso, estos permiten la creación de árboles cargados y con operaciones que tienden a tiempos lineales en vez de logarítmicos.

En el caso del BTree con B=4096, se obtienen peores tiempos en experimentos de datos aleatorios equiprobables y con datos sesgados bajo funciones de probabilidad. Este tipo de árboles tienen una gran cantidad de datos guardados en un único nodo, lo cuál hace que sus operaciones tarden más en recorrer cada nodo y descartar grandes cantidades de datos por cada iteración.

A continuación se detalla tanto la implementación de las estructuras estudiadas, los experimentos a los cuales se sometieron, y sus resultados junto con las conclusiones obtenidas por cada experimento.

Implementación

Las estructuras ocupadas para la realización de este trabajo son variantes de árboles binarios, por lo que es necesario detallar su funcionamiento, estructura, e implementación para una mejor comprensión y análisis de los resultados obtenidos en la experimentación. A continuación, se muestran las distintas estructuras utilizadas con una descripción de su implementación:

I **ABB (Árbol de Búsqueda Binaria clásico)**

Un ABB es una estructura de datos recursiva que se puede caracterizar en forma inductiva:

- El árbol nulo es un ABB,
- Un nodo de valor v del cual cuelgan dos ABBs $\{\text{left}, \text{right}\}$, donde $\forall a \in \text{left}, a < v$ y $\forall b \in \text{right}, b > v$. En otras palabras, todos los valores guardados en el hijo izquierdo son menores que v, y todos los valores guardados en el hijo derecho son mayores que v.

Así, podemos definir un ABB como la estructura que contiene los siguientes datos:

```
typedef struct abb{
    long value;             // Valor almacenado en el nodo
    struct abb *left;       // ABB izquierdo, con valores menores a abb.valor
    struct abb *right;      // ABB derecho, con valores mayores a abb.valor
} Abb;
```

Esta definición es la clásica y que será base para el resto de los árboles ocupados en este trabajo.

I **Inserción**

La inserción de un valor v en un ABB A es simple y podemos dividirlo en dos casos:

1) Si A es nulo, insertamos inmediatamente,
2) Si A no es nulo, entonces vemos su relación con el valor contenido en A:
   a) Si $v < A.value$, entonces insertamos recursivamente en el ABB izquierdo de A,
   b) si no, insertamos recursivamente en el ABB derecho de A.

II **Búsqueda**

La búsqueda de un valor v en un ABB A es equivalente a la inserción:

1) Si A es nulo, entonces v no está en A y tenemos una búsqueda infructuosa,
2) Si A no es nulo, y $A.value = v$, entonces lo encontramos y retornamos A.
3) Si A no es nulo y $A.value \neq v$, entonces vemos su relación con el valor contenido en A:
   a) Si $v < A.value$, entonces buscamos recursivamente en el ABB izquierdo de A,
   b) si no, buscamos recursivamente en el ABB derecho de A.

La implementación en lenguaje C se encuentra adjunta a este trabajo, con el nombre `Abb.c`

II AVL (Árbol de Búsqueda Binaria Autobalanceable)

Un AVL es un ABB que se autobalancea respecto a la altura de sus sub-arboles con el fin de asegurar el peor caso de búsqueda e inserción. Esto hace que la altura total de un AVL sea siempre $log(n)$, evitando sub-árboles cargados. La definición recursiva de un AVL es exactamente la misma que la de un ABB, pero lo que cambia es la definición de la estructura, ya que esta debe contener ahora la altura del árbol completo:

```
typedef struct avl{
    long value;             // Valor almacenado en el nodo
    struct avl *left;       // AVL izquierdo, con valores menores a avl.valor
    struct avl *right;      // AVL derecho, con valores mayores a avl.valor
    int height;             // Altura total del AVL
} Avl;
```

Es claro que la operación de Búsqueda e Inserción en un AVL es exactamente igual que en un ABB, y que estos difieren en la actualización de la altura de cada sub-arbol:

I Inserción

Como se mencionó anteriormente, la primera parte de la inserción en un AVL es equivalente a la de un ABB (esto es, se inserta recursivamente hasta llegar a un arbol nulo y crearlo). Pero además de esto debemos ahora actualizar la altura del sub-árbol insertado y la de sus padres recursivamente:

- Primero, actualizamos la altura actual del arbol a la altura máxima entre sus hijos + 1 (por el nodo insertado recientemente).
- Ahora debemos verificar las alturas de sus hijos y ver si necesitamos balancearlas. Esta condición depende si la diferencia de sus alturas es mayor a 1 (los sub-árboles difieren en dos niveles, por lo que están desbalanceados)
  1) Si están desbalanceados hacia la derecha, entonces dependiendo si el valor insertado es mayor o menor rotamos el árbol derecho o izquierdo (respectivamente) como cabezal y adjustamos los hijos para mantener la definición recursiva.
  2) El caso contrario es análogo, ya que se adjusta de igual manera pero como un espejo.

  Este split es recursivo, ya que es posible que la raíz también se desbalancee dependiendo de los datos insertados al árbol.

Como se mencionó anteriormente, la primera parte de la inserción en un AVL es equivalente a la de un ABB (esto es, se inserta recursivamente hasta llegar a un arbol nulo y crearlo). Pero además de esto debemos ahora actualizar la altura del sub-árbol insertado y la de sus padres recursivament

II Búsqueda

Se busca un elemento igual que en un ABB, por lo que no se hondará en su implementación.

La implementación en lenguaje C se encuentra adjunta a este trabajo, con el nombre `Avl.c`

III B-Tree

A diferencia de las estructuras mostradas anteriormente, los B Trees permiten nodos con más de un único valor con el fin de evitar tantos llamados recursivos para encontrar valores guardados. Así, podemos entender un B-Tree como la estructura recursiva definida como:

- El árbol nulo es un B-Tree,
- Un nodo con a lo más B valores guardados en una lista V y con a lo más $B+1$ B-Trees hijos guardados en una lista Children, donde $\forall c_i \in$ Children y $\forall v \in c_i, V_{i-1} < v < V_i$. En otras palabras, todos los valores guardados en el sub-árbol c_i son mayores que el valor $i-1$, y menores que el valor i de la lista de valores V.

Cada vez que se inserta un valor este se hace en una hoja (nodo sin hijos), y cuando este alcance el máximo de valores B se dividirá en dos partes (operación split), donde el valor central sube al padre, y el nodo se divide en dos de aproximadamente $B/2$ valores guardados.

La forma de este árbol asegura el peor caso como $log(B/n)$, y existen distintas formas de implementarlo algorítmicamente. En este trabajo se utilizará la siguiente definición de estructura:

```
typedef struct btree{
    long Values[B];             // B valores almacenados en un vector
    int count;                  // Contador de la cantidad actual de valores almacenados
    struct btree *C[B+1];       // B+1 B-trees como hijos del nodo
    struct btree *parent;       // Referencia al padre para la operacion split
} BTree;
```

Naturalmente las operaciones de inserción y búsqueda cambian ligeramente en su implementación, debido a que ahora estamos trabajando con vectores de valores y árboles, y además introducimos la operación split para poder dividir los nodos llenos:

I Búsqueda

La búsqueda de un valor v en un Btree A es similar a la de un ABB, pero adaptada a una lista:

1) Si el nodo es nulo, tenemos una búsqueda infructuosa.
2) Si no, verificamos de izquierda a derecha los valores guardados en el nodo con las siguientes afirmaciones:
   a) Si el valor en la posicion i es igual que v, entonces tenemos una búsqueda exitosa, y retornamos el nodo.
   b) Si el valor en la posicion i es mayor, entonces buscamos recursivamente en el hijo c_i.

   Si ya buscamos en todos los valores guardados en el nodo y no se cumplieron ninguna de las afirmaciones anteriores, entonces buscamos recursivamente en el último hijo guardado en el nodo.

II Inserción

La inserción de un valor v en un Btree A incluye una nueva operación split sobre los nodos. Para esto, consideremos el siguiente algoritmo:

- Al igual que en la búsqueda en un Btree, buscamos recursivamente el valor a insertar v, pero en vez de retornar un nodo, en este caso terminamos una vez que el nodo consultado no tiene hijos (hoja de un btree) y guardamos la última posición consultada k
- Ya obtenida la posición exacta donde insertar el valor v, simplemente movemos todos los valores de k hasta el último en una posición a la derecha, para finalmente insertar en la posición k el valor v.

- Ya insertado el valor, debemos consultar si el nodo está lleno. Si este tiene B valores guardados, entonces debemos hacer split del nodo, si no, retornamos.

III **Split**

La operación split, como se dijo anteriormente, divide un nodo en dos utilizando un valor pivote, el cuál después sube al nodo padre. Para esta operación, consideremos el siguiente algoritmo:

- Primero, calculamos el pivote como el valor central del nodo (valor en la posición floor($B/2$)).
- Luego creamos un nuevo Btree, el cuál contendrá todos los valores mayores al valor pivote, y además sus nodos hijos.
- Por cada valor guardado en el nodo:
  1) Si el valor es igual al pivote, solo lo borramos y seguimos.
  2) Si el valor es mayor al pivote, entonces debemos moverlo al nuevo arbol junto con el hijo a su izquierda, y seguimos.

  Finalmente movemos el último hijo guardado en el nodo al último del nuevo nodo.
- Ya habiendo dividido el nodo original, es momento de subir el pivote al nodo padre:
  1) Si el padre es nulo, entonces estamos en la raíz del árbol. Simplemente creamos un nuevo nodo raíz con valor igual al pivote, y dejamos como hijos el nodo modificado y el nodo creado con valores mayores al pivote.
  2) Si el padre no es nulo, entonces debemos insertar el pivote en el padre. Para esto buscamos la posición exacta en el padre, y movemos todos los valores e hijos de este hacia la derecha.
     a) Si el padre ahora está lleno, hacemos split recursivamente sobre el nodo padre.
     b) Si no está lleno, la inserción ha terminado y retornamos.

La implementación en lenguaje C se encuentra adjunta a este trabajo, con el nombre `Btree.c`

IV Splay Tree o Árbol Biselado

Un Splay Tree es un ABB que va rotando y subiendo nodos a medida que se busca (exitosa o infructuosamente) y que se inserta. Dado que es un ABB modificado, su definición recursiva es la misma, solo que cambia su implementación al tener que incluir la referencia al padre del nodo para la realización de las rotaciones de nodos.

```
typedef struct splay{
    long value;             // Valor almacenado en el nodo
    struct splay *left;     // SplayTree izquierdo, con valores menores a splay.valor
    struct splay *right;    // SplayTree derecho, con valores mayores a splay.valor
    struct splay *parent;   // Referencia al padre del nodo
} Splay;
```

Es claro que la operación de Búsqueda e Inserción en un Splay Tree es exactamente igual que en un ABB, y que estos difieren en las rotaciones necesarias para actualizar el árbol en base al nodo insertado/buscado. Para realizar las rotaciones y reacomodar el nodo x a la raíz se usa la función *splay(x)* explicada en el apunte.

I Inserción

Como se mencionó anteriormente, la primera parte de la inserción en un Splay Tree es equivalente a la de un ABB (esto es, se inserta recursivamente hasta llegar a un arbol nulo y crearlo). Pero además de esto debemos ahora mover el nodo creado como raíz del árbol, y rotar los hijos del padre para mantener la definición recursiva:

1) Si el valor del padre es mayor que el del insertado, entonces todo se va a la izquierda de la nueva raíz.
2) Si el valor del padre es menor que el del insertado, entonces todo se va a la derecha de la nueva raíz.

II Búsqueda

La búsqueda en un Splay Tree es equivalente a la de un ABB, pero además de esto debemos ahora mover un nodo a la raíz, dependiendo si la búsqueda fué exitosa o infructuosa:

- Si la búsqueda del elemento x es exitosa, lo movemos hacia la raíz y rotamos el árbol correspondiente llamando a la función `splay(x)`.
- Si no, movemos el último elemento accedido a la raíz y rotamos el árbol correspondiente usando *splay(x)*.

La implementación en lenguaje C se encuentra adjunta a este trabajo, con el nombre `Splay.c`

Experimentación

Para generar la secuencia de n ($n = 10^6$), se utilizó el método `random.sample(range, n)` del módulo `random` de python. Así, se programa el archivo `value_rng.py` que genera la secuencia aleatoria n, donde `range` corresponde al rango de números posibles (en nuestro caso range = `range(0, 2^32)`) y `n` la cantidad de números aleatorios a generar. La secuencia generada se guarda en un archivo `.txt` el cual posteriormente será utilizado en `test.c` para realizar los experimentos.

Para la experimentación, `test.c` genera los 6 árboles: ABB, AVL, Splay, BTree 16, BTree 256, BTree 4098 descritos anteriormente e inicialmente vacíos. Utilizando la secuencia ya mencionada, se realizan 6 experimentos que varían las probabilidades de cada valor. Esos experimentos son: Aleatorio (con valores equiprobables), Creciente (con factor k=0,1 y 0,5) y Sesgado (con f(x) = x, srqt(x) y ln(x)).

Para poder estudiar correctamente el desempeño de cada estructura en función de las operaciones, es necesario medir el tiempo que toma cada operación individualmente y guardar estos tiempos para posteriormente graficarlos. Lo anterior, sin embargo, no es eficiente, y aumentaría el tiempo de ejecución enormemente, por lo que una alternativa más razonable es medir el tiempo promedio por cada sección de 1.000 operaciones, y generar entonces 1.000 datos por cada estructura y por cada experimento.

Como las operaciones de inserción, búsqueda exitosa y búsqueda infructuosa se tienen probabilidades distintas, 0.5, 0.33 y 0.17 respectivamente, podemos decir que la cantidad de inserciones, búsqueda exitosa e infructousa serán $n*0{,}5$, $n*0{,}33$ y $n*0{,}17$ respectivamente, estos valores los insertamos en un arreglo llámese `operacion[]` donde habrán $n*0{,}5$ 0s, $n*0{,}33$ 1s y $n*0{,}17$ 2s los cuales se colocarán de manera aleatoria dentro de esta. Ahora para ver qué número le corresponderá cada operación, se elegirá un número aleatorio del arreglo de la secuencia de valores y se colocará en otro arreglo, llámese `params[]` que guardará el valor en la misma posición que la de la operación. i.e si toca una operación en la posición j, elegiremos un valor de la lista de valores y la copiaremos en `params` en la posición j.

Para el caso del experimento Creciente lo que se hace es equivalente, pero al elegir un valor del arreglo de valores se multiplica por $\frac{k}{2^{32}}$, para transformarlo en un numero entre 0 y k para luego sumarle la cantidad de nodos

Para el del experimento Sesgado por otro lado, al elegir un valor lo multiplicamos por su función de probabilidad $P(x) \in \{x, sqrt(x), ln(x)\}$. Esta función de probabilidad se va acumulando por cada valor (i.e. $x = x + P(x)$) con el fin de evitar listas de tamaños extremadamente grandes, por lo que al elegir un valor aleatorio, este se busca en la lista de probabilidades acumuladas y se relaciona con su valor real para realizar la operación.

Recorremos el arreglo de las operaciones, vemos cual va tocando y como tenemos la lista de valores a insertar, buscar exitosa e infructuosamente, iteramos de `i=0 to n` y por cada `i` sacaremos su operación en `operacion[i]` y el valor en `params[i]`. Luego ejecutamos la operación con su valor según corresponda.

Para calcular el tiempo que el arbol candidato se demora en realizar en un intervalo, se utiliza la función `clock()` para calcular el tiempo total del intervalo y seguidamente lo escribimos en un `.txt` para guardar los resultados para posteriormente graficar los resultados obtenidos.

Una vez con los resultados obtenidos se procede a graficar y los resultados obtenidos se presentan a continuación.

Presentación de Resultados

El experimento descrito anteriormente se realizó solo una vez debido al tiempo de ejecución, calculando el tiempo promedio cada 1000 operaciones 1000 veces. A continuación se muestran las estadísticas más relevantes por cada experimento, junto con los boxplots de los tiempos por cada estructura, y gráficos con ajustes de curvas logarítmicas y también en escala logarítmica:

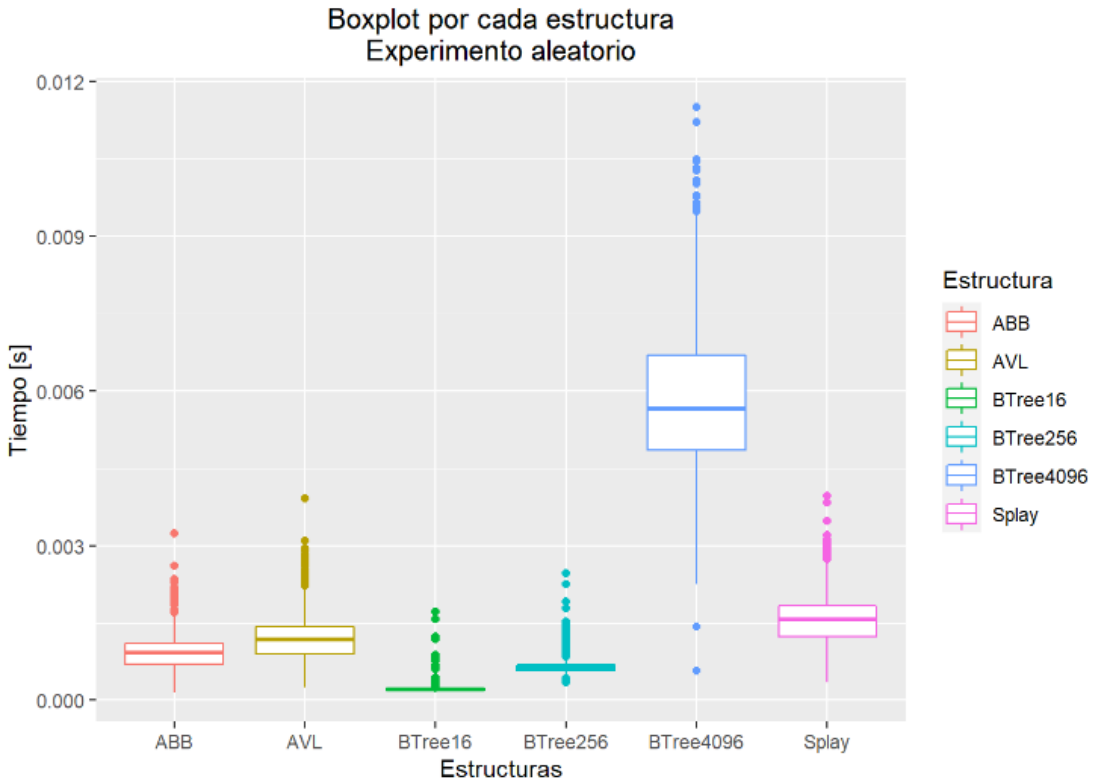
I Experimento Aleatorio:

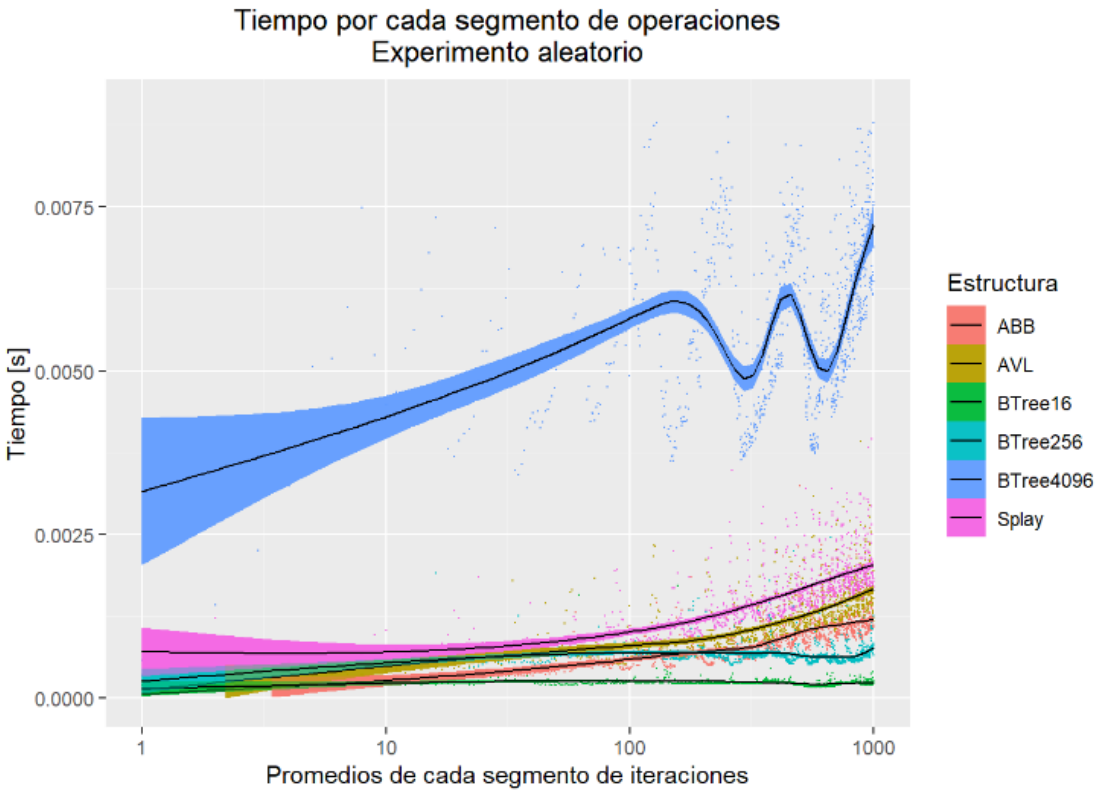
Del experimento aleatorio se observa a partir de los gráficos mostrados a continuación que el árbol que toma más tiempo en realizar las operaciones es el arbol BTree 4096 mientras que el de menor tiempo es el arbol BTree 16.
Por otro lado, el ABB, AVL, Splay tienen comportamiento similar durante el experimento.

Estadísticas importantes

```
"ABB: MEAN =  0.000930327 , SD =  0.000348079455553439"
"AVL: MEAN =  0.001216248 , SD =  0.00044697059546373 9"
"Splay: MEAN =  0.001579597 , SD =  0.00051553154837349"
"BTree16: MEAN =  0.000237074 , SD =  0.000103498736767714"
"BTree256: MEAN =  0.000668438 , SD =  0.000169946809543675"
"BTree4096: MEAN =  0.00580662 , SD =  0.00133740359053053"
```

Boxplots de las estructuras

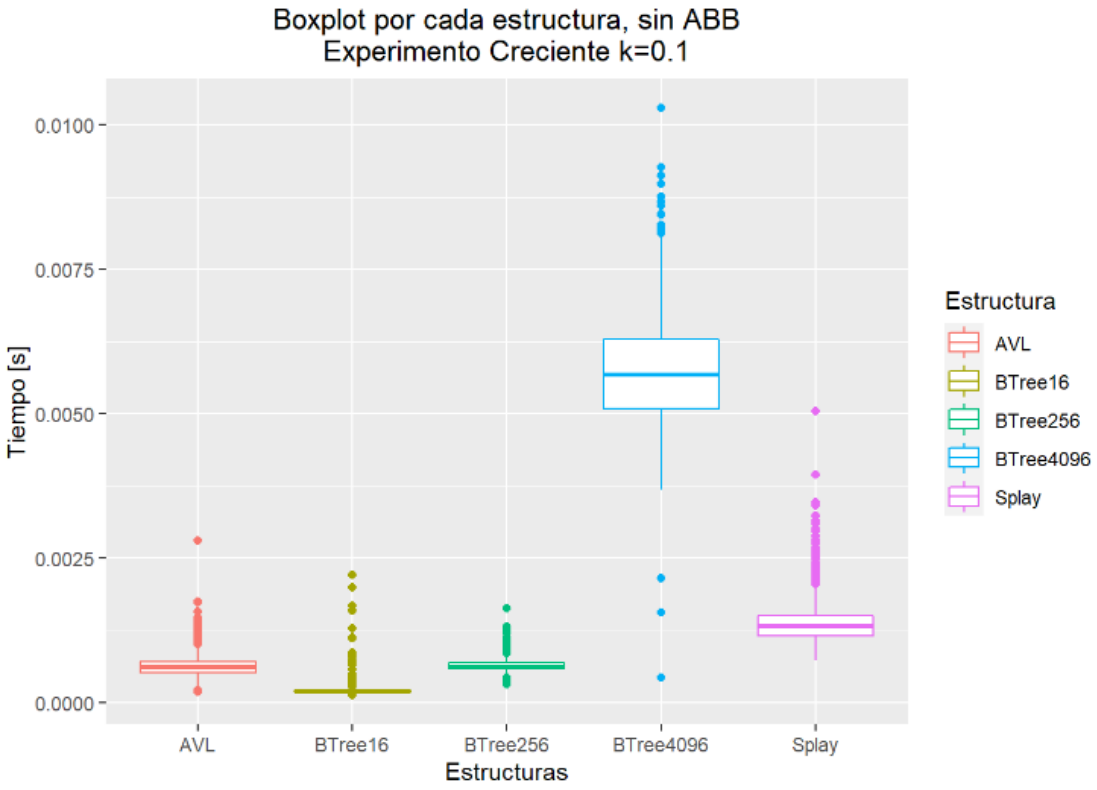


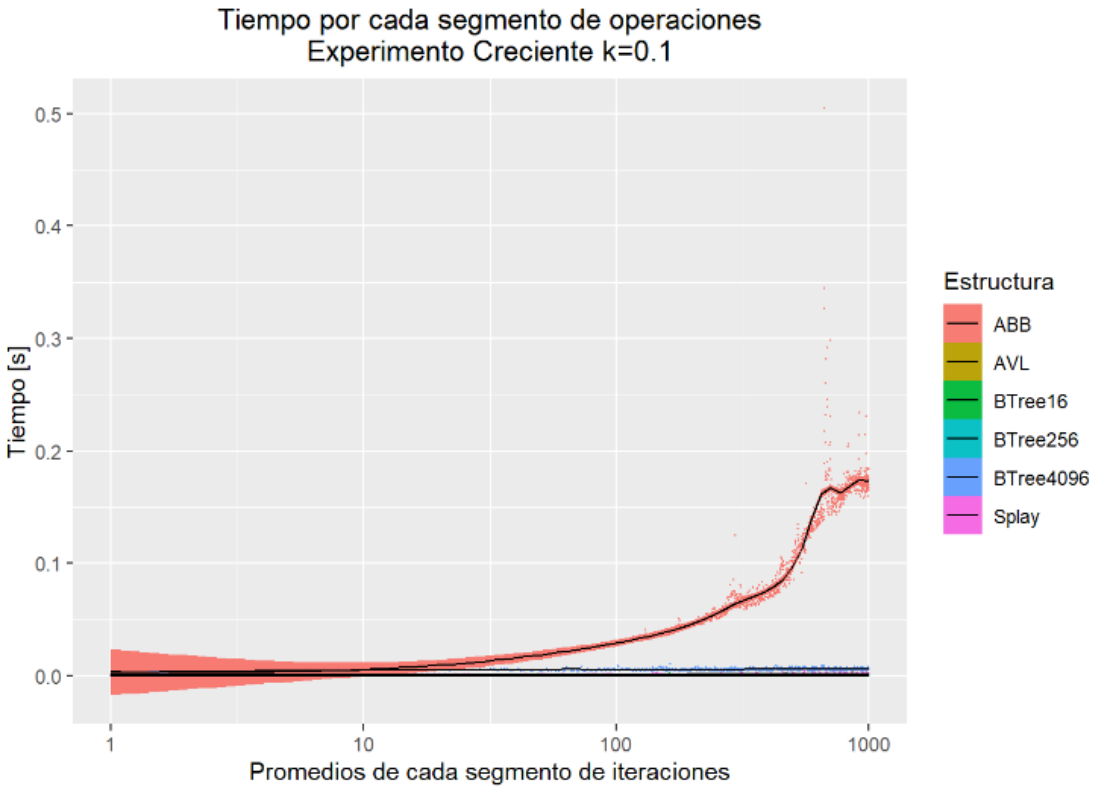

Tiempo por cada segmento de operaciones
Experimento aleatorio
Tiempo [s]
0.0075
0.0050
0.0025
0.0000
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

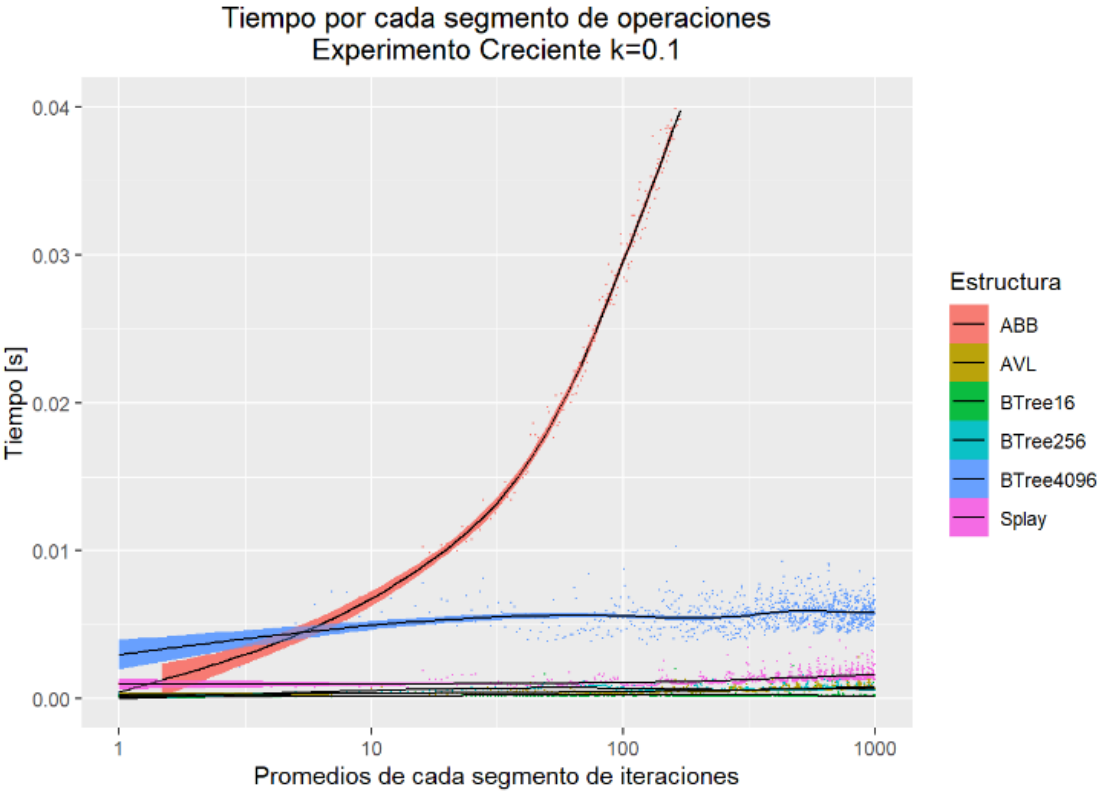
II Experimento Creciente con $k = 0{,}1m$:

Del experimento creciente $k = 0{,}1m$ se observa a partir de los gráficos que el árbol que toma más tiempo en realizar las operaciones es el ABB mientras que el de menor tiempo sigue siendo el arbol BTree 16. Notar que en las 3 variaciones del BTree, el AVL y el Splay, el tiempo no aumenta a medida que aumenten las operaciones, como el ABB, pues se mantienen "estables".

```
"ABB: MEAN =  0.105542098 , SD =  0.0597515321009998"
"AVL: MEAN =  0.000647077 , SD =  0.000206721171421105"
"Splay: MEAN =  0.001396472 , SD =  0.0004030516640083"
"BTree16: MEAN =  0.000219507 , SD =  0.00013064235674687"
"BTree256: MEAN =  0.00065965 , SD =  0.000121809955115192"
"BTree4096: MEAN =  0.005747077 , SD =  0.000944166761724297"
```



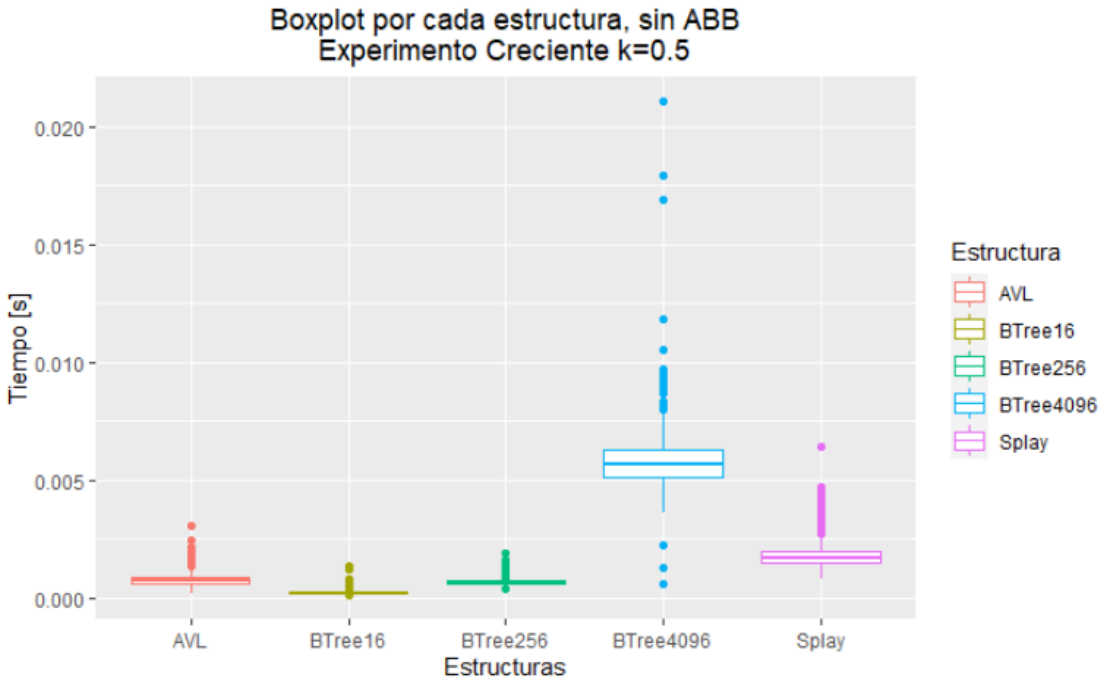

Tiempo por cada segmento de operaciones
Experimento Creciente k=0.1
Tiempo [s]
0.5
0.4
0.3
0.2
0.1
0.0
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

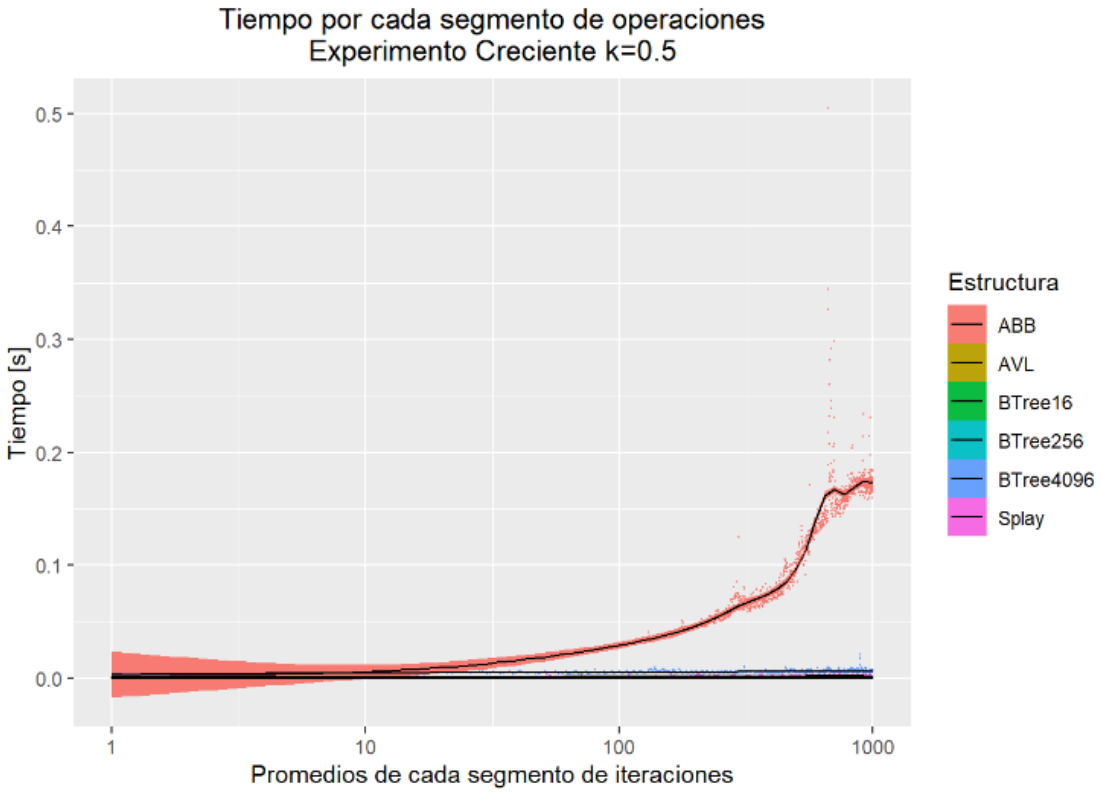

Tiempo por cada segmento de operaciones
Experimento Creciente k=0.1
Tiempo [s]
0.04
0.03
0.02
0.01
0.00
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

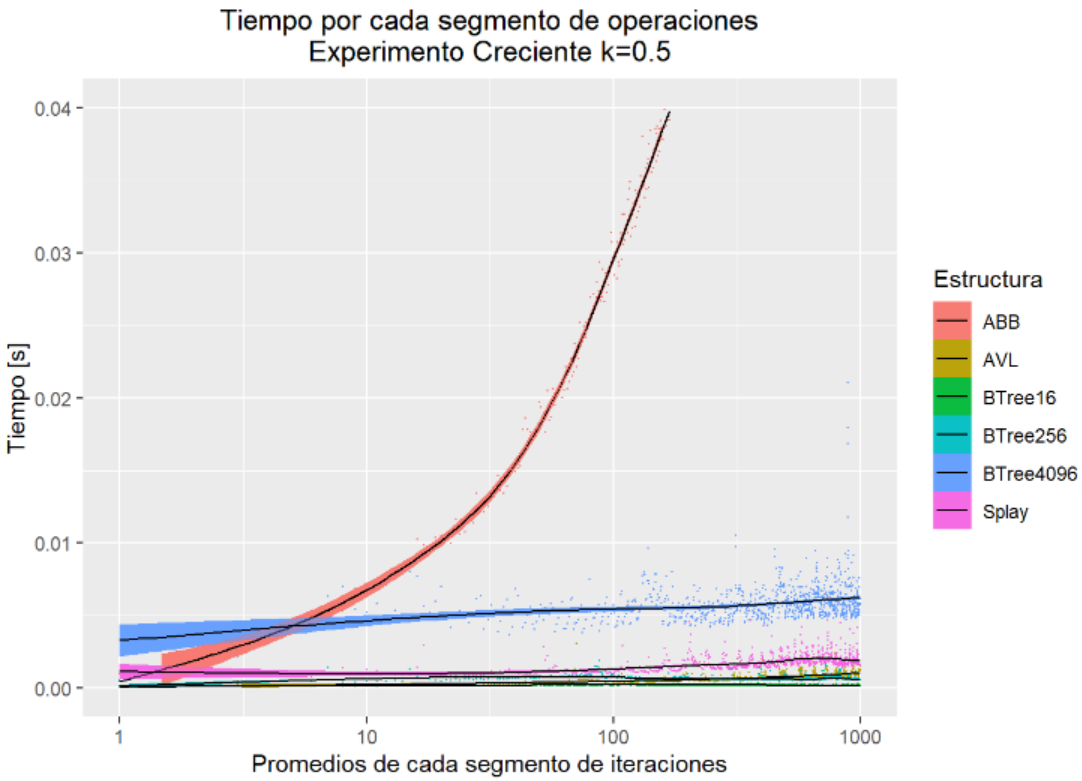
III Experimento Creciente con $k = 0{,}5m$:

Del experimento creciente $k = 0{,}5m$ se observa a partir de los gráficos que el árbol que toma más tiempo en realizar las operaciones es el ABB mientras que el de menor tiempo sigue siendo el arbol BTree 16. Ahora, sin considerar el ABB, el segundo más lento le correspondería al BTree 4096. Además podemos notar que los árboles Btree 256 y AVL se comportan de forma similar, demorando más que el árbol BTree 16 pero menos que el árbol Splay que muestra un tiempo mayor.

```
"ABB: MEAN =  0.04265281 , SD =  0.0200915517989425"
"AVL: MEAN =  0.0007641 , SD =  0.000311144035151906"
"Splay: MEAN =  0.001778053 , SD =  0.00055461030713807"
"BTree16: MEAN =  0.000210127 , SD =  7.29064209601405e-05"
"BTree256: MEAN =  0.000659842 , SD =  0.000140676212746977"
"BTree4096: MEAN =  0.005832377 , SD =  0.00123320118635892"
```



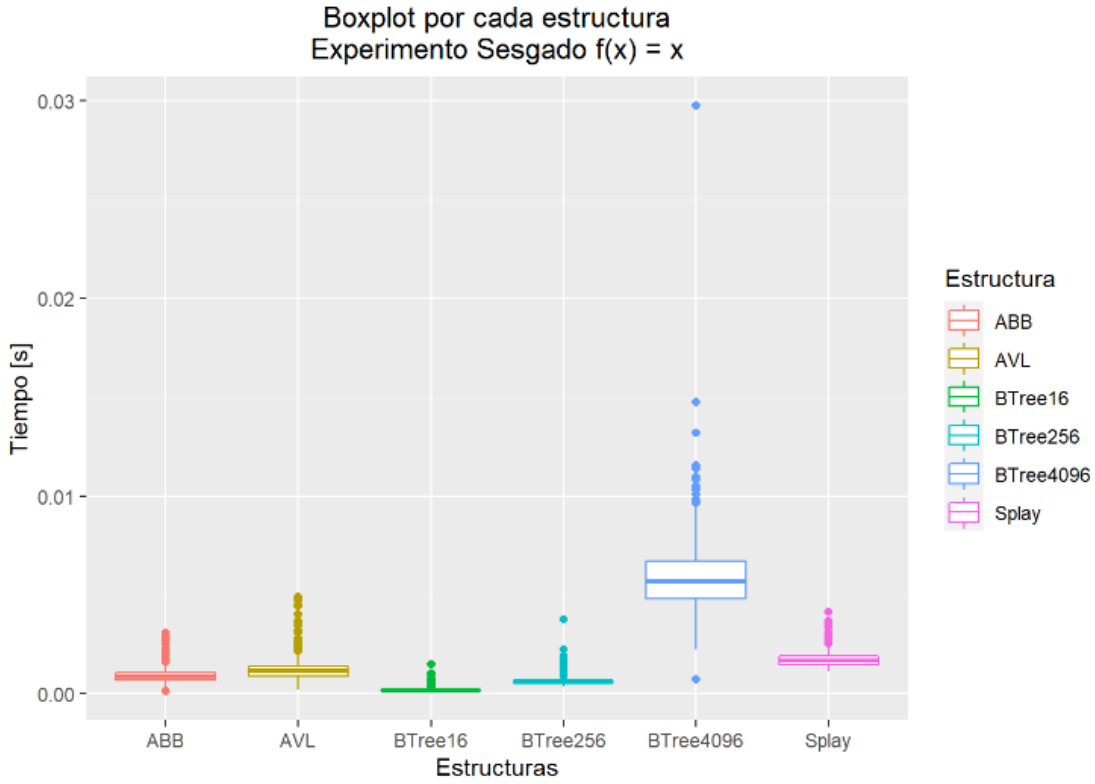

Tiempo por cada segmento de operaciones
Experimento Creciente k=0.5
Tiempo [s]
0.5
0.4
0.3
0.2
0.1
0.0
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

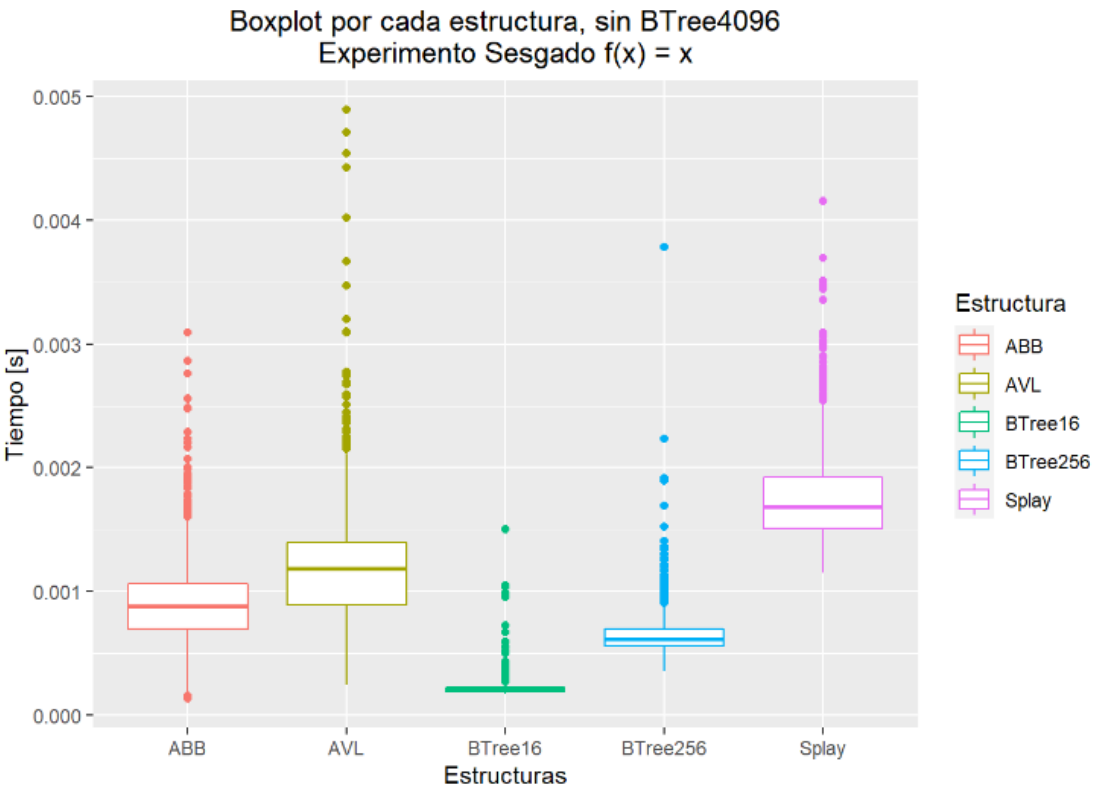

Tiempo por cada segmento de operaciones
Experimento Creciente k=0.5
Tiempo [s]
0.04
0.03
0.02
0.01
0.00
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

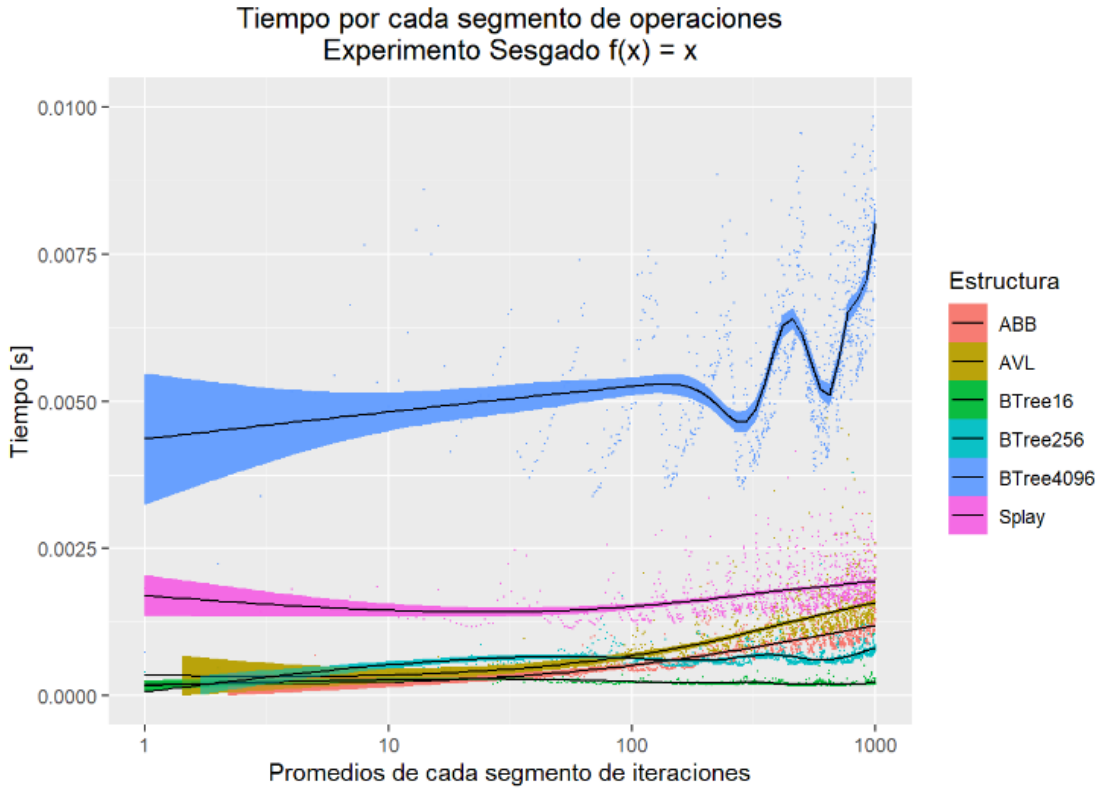
IV Experimento sesgado con $p(x) = x$:

Del experimento sesgado con funcion peso igual $p(x) = x$ se observa a partir de los gráficos que el árbol que toma más tiempo en realizar las operaciones es el arbol BTree 4096 mientras que el de menor tiempo sigue siendo el arbol BTree 16.
En cuanto a los otros arboles, estos muestran comportamientos similares.

```
"ABB: MEAN =  0.000901977 , SD =  0.00036564149679047"
"AVL: MEAN =  0.001204548 , SD =  0.00051011259792183"
"Splay: MEAN =  0.001775908 , SD =  0.00039892323232315794"
"BTree16: MEAN =  0.00021806 , SD =  8.16068138826169e-05"
"BTree256: MEAN =  0.000663951 , SD =  0.00019650256450453"
"BTree4096: MEAN =  0.005887827 , SD =  0.00162321232601478"
```



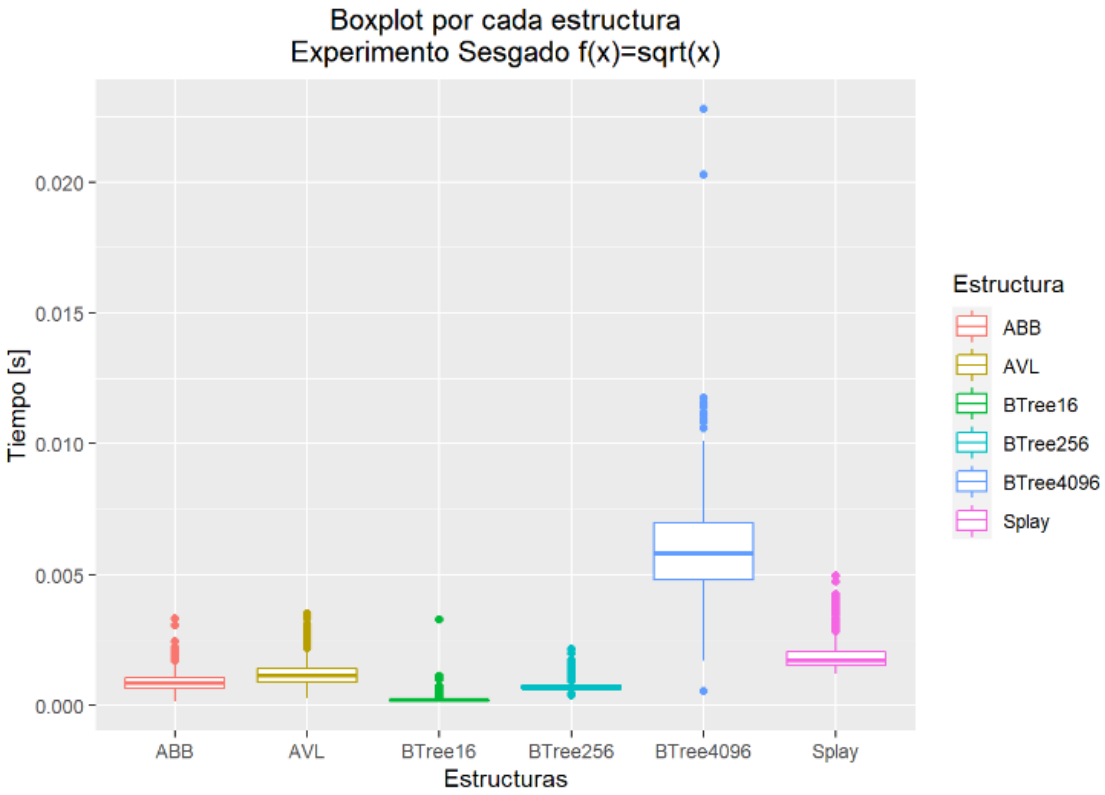

Boxplot por cada estructura, sin BTree4096
Experimento Sesgado f(x) = x
Tiempo [s]
0.005
0.004
0.003
0.002
0.001
0.000
ABB
AVL
BTree16
BTree256
Splay
Estructuras
Estructura
ABB
AVL
BTree16
BTree256
Splay

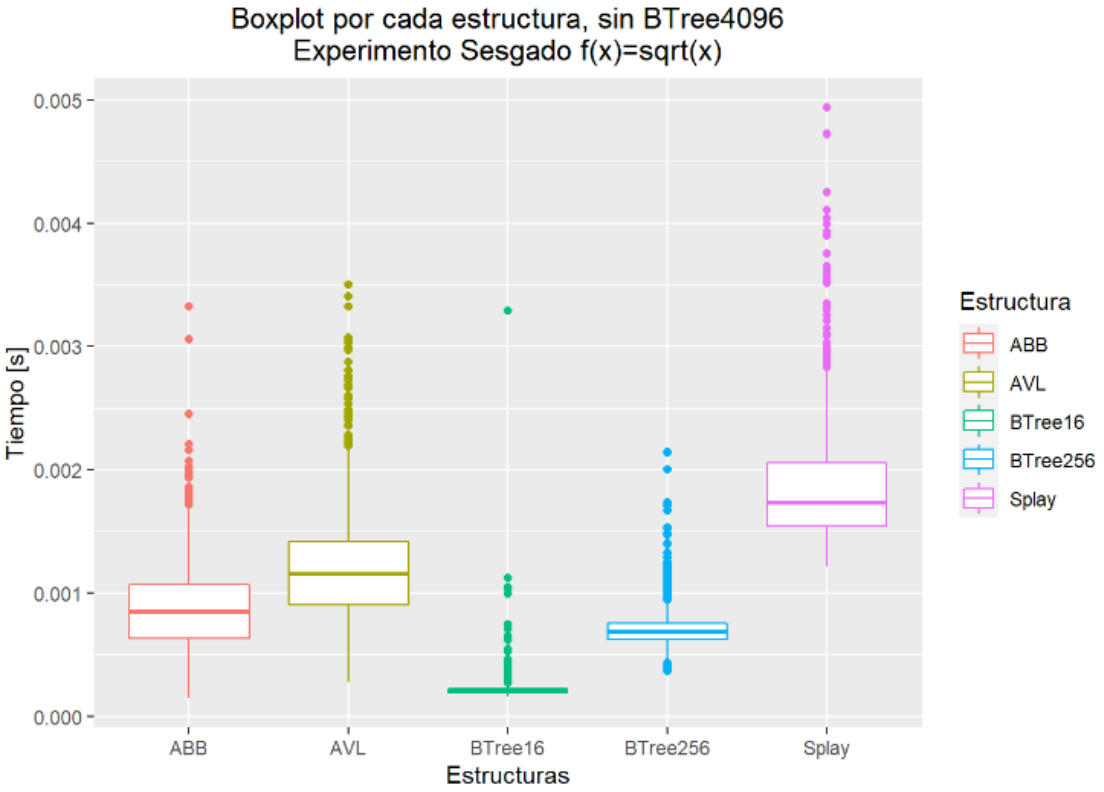

Tiempo por cada segmento de operaciones
Experimento Sesgado f(x) = x
Tiempo [s]
0.0100
0.0075
0.0050
0.0025
0.0000
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

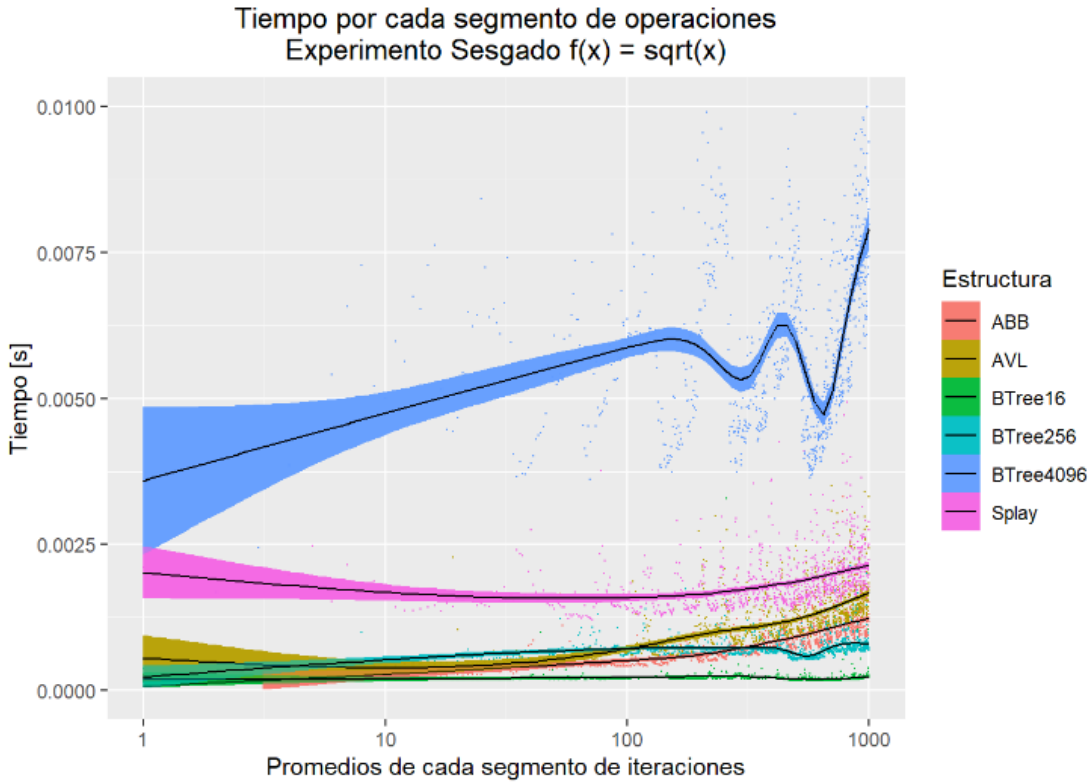
V Experimento sesgado con $p(x) = \sqrt{x}$:

Del experimento sesgado con funcion peso igual $p(x) = \sqrt{x}$ se observa a partir de los gráficos que el árbol que toma más tiempo en realizar las operaciones es el arbol BTree 4096 mientras que el de menor tiempo sigue siendo el arbol BTree 16.
En cuanto a los otros arboles, estos muestran comportamientos similares.

```
"ABB: MEAN =  0.000884197 , SD =  0.000360665184855104"
"AVL: MEAN =  0.001200664 , SD =  0.000492648306053576"
"Splay: MEAN =  0.001865372 , SD =  0.000485580139305231"
"BTree16: MEAN =  0.000219922 , SD =  0.000122061029042077"
"BTree256: MEAN =  0.000721659 , SD =  0.000187344833944803"
"BTree4096: MEAN =  0.006016262 , SD =  0.00166918716480422"
```



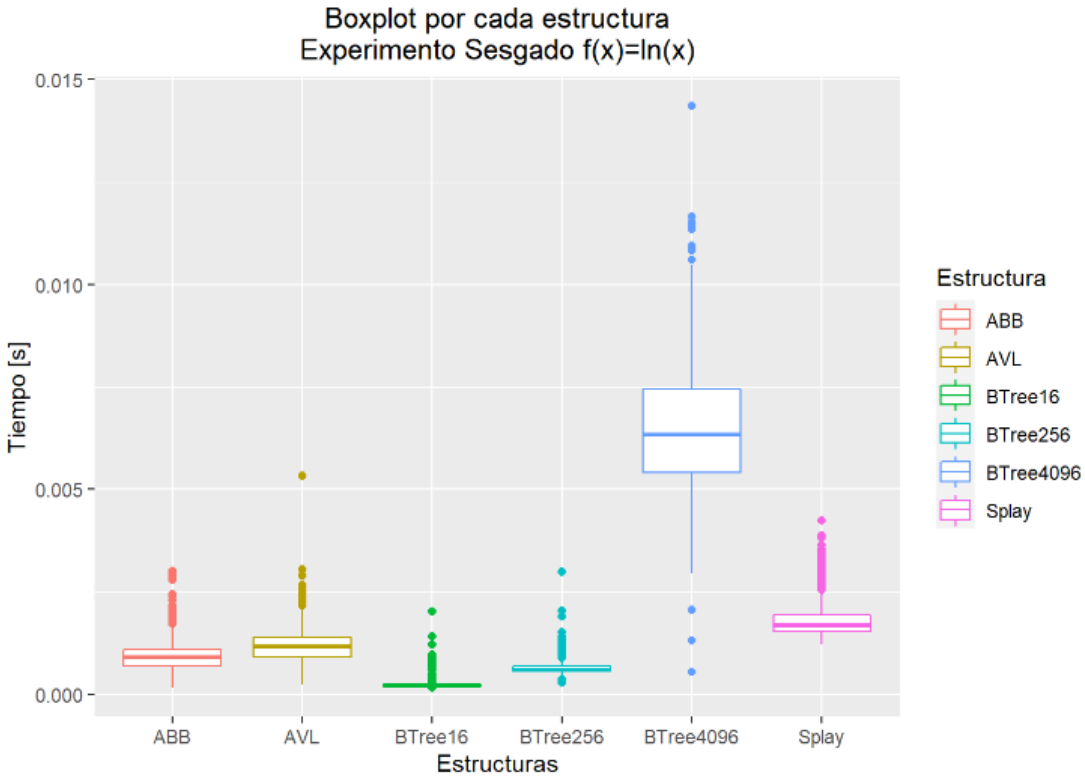

Boxplot por cada estructura, sin BTree4096
Experimento Sesgado f(x)=sqrt(x)
Tiempo [s]
0.005
0.004
0.003
0.002
0.001
0.000
ABB
AVL
BTree16
BTree256
Splay
Estructuras
Estructura
ABB
AVL
BTree16
BTree256
Splay

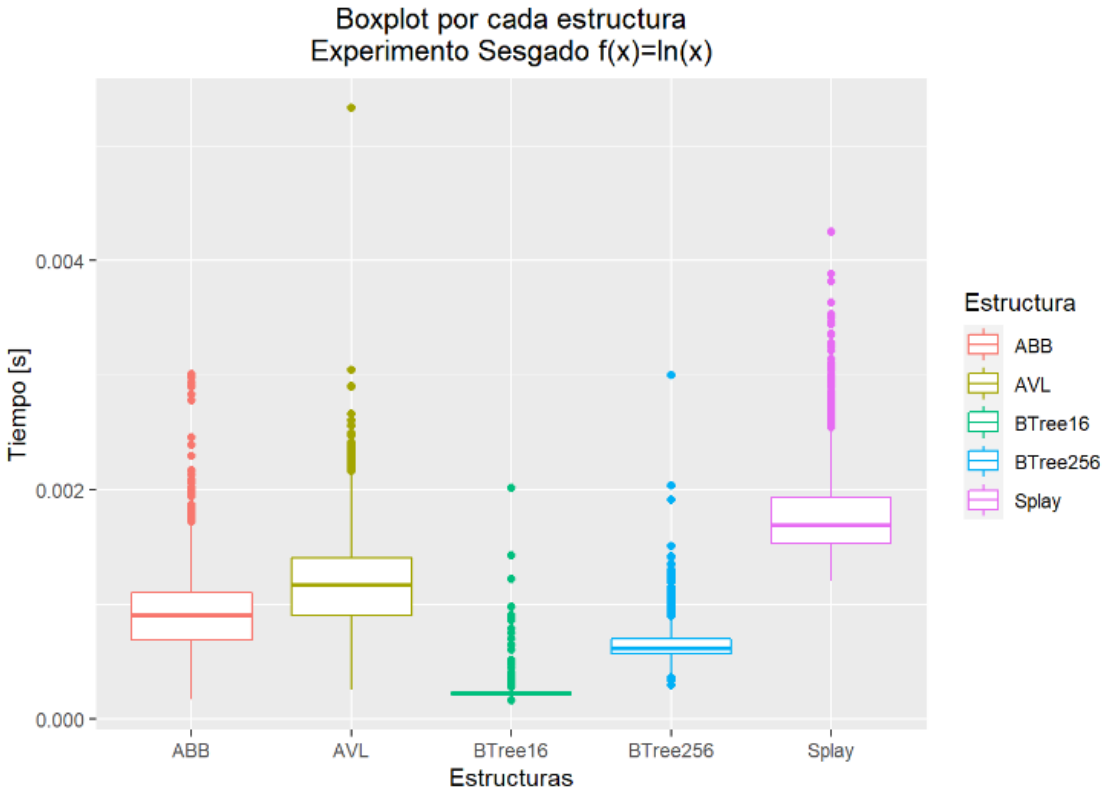

Tiempo por cada segmento de operaciones
Experimento Sesgado f(x) = sqrt(x)
Tiempo [s]
0.0100
0.0075
0.0050
0.0025
0.0000
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

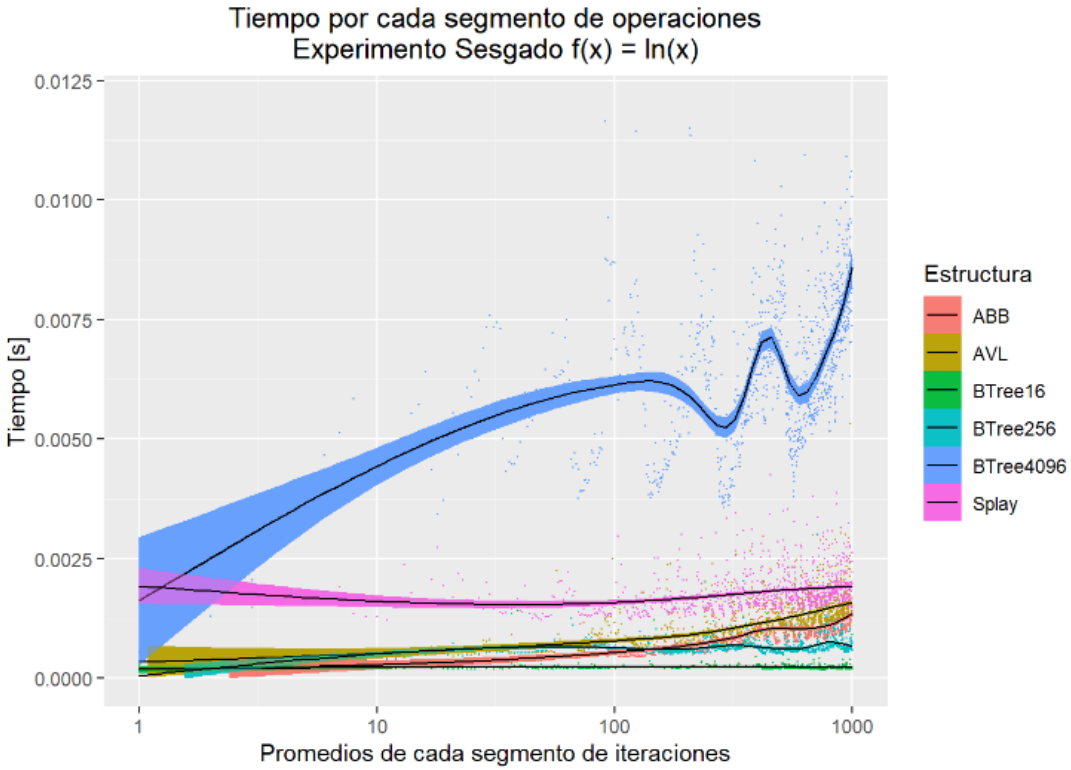
VI Experimento sesgado con $p(x) = \ln x$:

Del experimento sesgado con funcion peso igual $p(x) = \ln x$ se observa a partir de los gráficos que el árbol que toma más tiempo en realizar las operaciones es el arbol BTree 4096 mientras que el de menor tiempo sigue siendo el arbol BTree 16.
En cuanto a los otros arboles, estos muestran comportamientos similares.

```
"ABB: MEAN =  0.000936846 , SD =  0.000389173234036065"
"AVL: MEAN =  0.001194411 , SD =  0.00043715648671027 7"
"Splay: MEAN =  0.001800395 , SD =  0.000413390017152424"
"BTree16: MEAN =  0.000237688 , SD =  9.81557599390337e-05"
"BTree256: MEAN =  0.000666389 , SD =  0.000177742234657329"
"BTree4096: MEAN =  0.006447387 , SD =  0.00150761245369675"
```




Boxplot por cada estructura
Experimento Sesgado f(x)=ln(x)
Tiempo [s]
0.004
0.002
0.000
ABB
AVL
BTree16
BTree256
Splay
Estructuras
Estructura
ABB
AVL
BTree16
BTree256
Splay


Tiempo por cada segmento de operaciones
Experimento Sesgado f(x) = ln(x)
Tiempo [s]
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
1
10
100
1000
Promedios de cada segmento de iteraciones
Estructura
ABB
AVL
BTree16
BTree256
BTree4096
Splay

Discusión

Experimento 1. Datos Aleatorios

El peor desempeño obtenido es el BTree4096, este ocurre debido a que cada nodo tiene un máximo de 4096 elementos, por lo que para realizar una inserción o búsqueda tendrá que recorrer linealmente el nodo pasando por a lo sumo 4096 elementos, y descartando menos datos por cada iteración.
El segundo peor desempeño, por otro lado, es el splay tree. Como el experimento es aleatorio y el splay mueve cada vez el elemento buscado/insertado a la raíz, el árbol tendrá que re-arreglar el árbol invocando cada vez la función `splay(x)` y buscando desde la última inserción o busqueda cada vez, lo cuál en datos aleatorios no tiene ningún tipo de ventaja sobre el resto de estructuras.

Además, en el gráfico de tiempos se puede observar claramente en el caso de BTree4096 los momentos donde este realiza la operación split. Esas subidas y bajadas que se hacen cada vez más largas se deben a que los nodos más profundos del árbol se van llenando más y más a medida que se opera sobre este, haciendo que los tiempos que en los que se tarda el árbol en insertar y buscar sean cada vez más lineales. Luego estos ascensos de tiempos promedios caen drásticamente de 0.008s a un aproximado de 0.004s al momento de hacer los splits de forma recursiva, los cuales disminuyen a la mitad los tamaños de los nodos más externos y bajan los tiempos de búsqueda.

Experimento 2. Datos Crecientes

Para ambos experimentos crecientes, el ABB es el que más tiempo tarda. Esto es debido a que los elementos siempre serán mayores al anterior tanto para la búsqueda como para las inserciones, y como este árbol no se re-arregla el ABB tiende a tener forma de lista al estar insertando valores siempre a la derecha, lo cuál por cada elemento nuevo el árbol tendrá que recorrer la estructura linealmente y no logarítmicamente.

El resto de las estructuras tiene un buen comportamiento al estar siempre asegurando el peor caso en tiempo logarítmico, mientras que para el caso del Splay Tree tendremos que cada nuevo elemento buscado o insertado estará siempre hacia el lado derecho y relativamente cerca del nodo raíz.

Para estos experimentos no se realizaron boxplots con ABB incluido, debido a que esta estructura tiene una tendencia lineal lo cuál aumentaría enormemente su distribución y estadisticas de resumen respecto al resto.

Experimento 3. Datos Sesgados

De manera análoga al experimento aleatorio, en el Btree 4096 se tendrá que recorrer en cada nodo a lo sumo 4096 valores. Como en este experimento, las 3 funciones $p(x)$ son crecientes, para el caso del Splay, mientras más grande sea el valor, más probabilidad tendrá de ser buscado, y como el Splay siempre mueve el ultimo elemento buscado o insertado a la raíz, es esperable que arroje los tiempos que se muestran en el gráfico.
Por parte del AVL, como independiente de la función o el elemento, siempre luego de una inserción la estructura cumple el invariante, es decir siempre mantiene una altura mayor o igual a 1 para los subárboles de este.

Conclusiones

Como se menciona en la sección anterior, existen casos donde el árbol de peor rendimiento es el BTree 4096 y en otros el ABB, mientras que el AVL, Splay y las otras variantes del BTree siempre se mantienen dentro del mismo rango. En los casos donde el ABB demora más en realizar las operaciones es esperado debido a que en los experiementos crecientes, como los elementos son crecientes a la hora de insertar siempre se insertarán como hijo derecho del nodo dando como resultado una especie de lista enlazada. Por lo que cuando aumenten los elementos en el árbol la búsqueda e inserción tendrán tiempo lineal $O(n)$. Teóricamente dicha complejidad correspondiente al peor caso del ABB. Esto se reflejado en el gráfico ajustado en los experimentos crecientes ya que están modelados en escala logarítmica y al hacer ´zoom‘ se puede apreciar que el gráfico del ABB tiende a ser exponencial, esto quiere decir que la teoría se cumple con los datos experimentales, pues insertar y buscar están tendiendo una complejidad de $O(n)$.

De manera análoga, cuando el BTree 4096 es el más lento pasa por el mismo argumento debido a que se tendrán que recorrer a lo sumo 4096 elementos por nodo es decir sería una búsqueda lineal.

Por otra parte, el AVL asegura costo de peor caso $O(logn)$, debido a que siempre asegura una altura logarítmica, esto también se puede ver reflejado en los gráficos de tiempo, debido a que, en particular, en los experiemntos crecientes el tiempo de ejecución de las operaciones no aumenta de manera abrupta como sí lo hace el ABB.

Cabe destacar que en todos los experimentos realizados el árbol con menor tiempo de ejecución fue el BTree 16, esto también es esperado, como el tamaño de cada nodo es de 16 no tiene que recorrer una gran cantidad de valores a la hora de insertar o buscar como en el caso del 4096. Además, esta estructura de datos se utiliza principalmente en memoria secundaria, por lo que está diseñado con el motivo de poder soportar grandes cantidades de operaciones, en particular las inserciones y búsquedas toman tiempo $O(log_B \frac{n}{m})$. Por lo que en este caso al estar operando con 1 millón de operaciones, se puede concluir que esta estructura efectivamente se comporta a lo estudiado teóricamente.