2015

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

Cristian Andrade Muñoz

[DISEÑO Y ANÁLISIS DE ALGORITMOS: ÁRBOLES DE BÚSQUEDA BINARIA]

Análisis cuantitativo del desempeño de algoritmos y estructuras de búsqueda binaria.

Resumen Ejecutivo:

Se asignó la construcción del código fuente para la implementación de algoritmos y estructuras de almacenamiento de datos, específicamente estructuras ABB, AVL, Splay Tree y van Emde Boas. Para esto, se utilizó el lenguaje Java, con implementación en Interfaz y Clases para los distintos algoritmos y estructuras, y empaquetamiento en JAR para ejecución headless en un computador de 64 bits, con procesador de 4 núcleos de 2.93GHz, con 6 Gb de RAM a 1333Mhz.

Para los archivos de búsqueda, se utilizaron cadenas de números naturales aleatorios entre 0 y 2^32 – 1, de forma de acercarse a completar un almacenamiento aproximado de 100 mb por estructura.

Los resultados del algoritmo utilizado indican que el algoritmo/estructura con mejor desempeño *over all* es el algoritmo AVL, mostrando costos de inserción y consulta cercanos valores experimentales obtenidos por investigaciones exhaustivas, y demostrando ser más eficiente en su acceso a disco para inserción y búsqueda de elementos, en este caso, números naturales. Para eliminación, el algoritmo más eficiente fue ABB, por su simplicidad al momento de eliminar los elementos.

Introducción:

Dada la gran cantidad de datos que se manipulan en el día a día, es necesario encontrar formas de almacenar y realizar búsquedas sobre ellos de forma fácil y escalable. Los algoritmos de búsqueda binaria han demostrado a lo largo de los años ser una alternativa simple de implementar, y con buena respuesta en cuanto a inserciones, borrados y búsquedas sobre sets de elementos de dimensión razonable, es decir, donde el procesamiento no sea necesario de llevar a disco.

Con el fin de analizar el rendimiento de distintos algoritmos/estructuras de ordenamiento en memoria principal, se realizan mediciones de tiempo de ejecución de operaciones (inserción, búsqueda y borrado), y tamaño efectivo de la estructura con respecto a los datos ingresados en múltiples experimentos, aplicando conceptos y técnicas vistas en clases tanto en la construcción de dichos experimentos como en el análisis de los resultados.

Las estructuras solicitadas a analizar son Árbol tradicional de Búsqueda Binaria (en adelante ABB), Árbol Adelson-Velskii y Landis (en adelante AVL), Splay Tree, o Árbol Biselado (en adelante SPL), y árbol de Van Emde Boas (en adelante VEB), y su rendimiento será evaluado en una serie de inserciones, búsquedas y eliminaciones, en números enteros positivos entre 0 y 2^32-1 (finalmente entre 0 y 12.000.000), extraídos de un archivo generado aleatoriamente o generados al azar, en repeticiones de 2^10 a 2^15 elementos.

Hipótesis:

- Las condiciones de ejecución son iguales para las estructuras testeadas, y no dependen de los conjuntos de datos utilizados. Los resultados deberían tener forma similar en todas las situaciones, y las variaciones que se produzcan van a ser el resultado del desempeño específico de los algoritmos frente a las situaciones de prueba.
- El tamaño del alfabeto es único para todas las ejecuciones (se trabaja sólo con números enteros positivos, y su representación en el sistema también es única), por lo que no representa una variable relevante al momento de comparar los algoritmos.
- Por otra parte, debería existir una mínima diferencia entre las ejecuciones de búsquedas, debido al balanceo de AVL y SplayTree.
- Cada línea en el archivo suministrado contiene 2 enteros positivos, los cuales, por implementación, se guardan directamente en cada árbol, y en forma binaria en el caso de VanEmdeBoas (convirtiendo cada entero a una codificación binaria).
- A medida que los árboles crecen, tanto la inserción como la búsqueda sobre la estructuras debería tender a O(log n) tiempo, y O(n log n), mientras que la búsqueda, inserción y borrado en Van Emde Boas debería costar O(log log (max. cantidad de elementos almacenables)), dependiendo su tiempo de ejecución sólo en la expansión y contracción de su estructura.
- Por otra parte, suponemos que el tamaño en memoria de los árboles dependerá cuasi linealmente de la cantidad de elementos ingresados (al ser elementos individuales enlazados), mientras que en Van Emde Boas dependerá de otros factores (como las variables de construcción de la estructura).

Diseño e Implementación

Todas las implementaciones se basan en la clase *GenericTree*, la cual obliga a realizar la implementación de *void insert(int key, int value)*, GenericNode find(int key), *long size()*, y *void delete(int value)*. De la misma forma, se crea una clase abstracta *GenericNode*, que obliga a implementar funciones de modificación e obtención de la llave del Nodo y su valor (las cuales, lamentablemente, no fueron bien aprovechadas). Se incluye también una clase Generador, que genera los archivos necesarios para almacenar los elementos en los arreglos.

Se definen así las estructuras que están siendo probadas en esta tarea:

ABB

Consiste en un árbol binario de datos, en que cada nodo tiene asignada una llave y un valor. En consecuencia, tiene un almacenamiento de 512 bytes por nodo. Sus costos promedio tienden a O(log n) en búsqueda, inserción y borrado, pero el peor caso puede llegar a ser lineal (por la falta de balanceo).

Cumple las siguientes invariantes:

- Cada nodo almacena desde 0 y hasta 2 nuevos nodos que, en efecto, son subárboles
- Los valores de todas las claves en el hijo izquierdo estarán entre:
  - Si el nodo actual es mayor que su padre, entre su padre y el nodo actual
  - Si es menor, entre 0 y el nodo actual
- De forma contraria, los valores de todas las claves en el hijo derecho estarán entre
  - Si el nodo actual es mayor que su padre, estarán entre Integer.MAX_INT (corregido a 12.000.000, por simplicidad) y el nodo actual
  - Si el nodo actual es menor que su padre, estarán entre el nodo actual y su padre
- Todas las hojas están a profundidad variable, pues no hay balanceo.

Implementación: Para poder realizar las operaciones con mayor facilidad, cada BTree tiene una referencia a su padre y a sus hijos, a excepción de la raíz, que tiene padre nulo.

- **GenericNode find(int key)**: Es la operación más sencilla. Delega en findABB la búsqueda del nodo que corresponde a la llave key. Parte buscando en la raíz el elemento, si lo encuentra lo retorna. En caso de no encontrarlo, desciende por el hijo que le corresponda: si es menor que el valor del nodo actual, desciende por el hijo izquierdo; caso contrario, por el derecho. Si dicho hijo no es encontrado al llegar a una hoja, retorna nulo. Caso contrario, continúa hasta encontrarlo.
- **void insert(int key, int value)**: Realiza una pseudo búsqueda sobre el árbol, encontrando la primera hoja vacía que le corresponda, insertándose allí.
- **Void delete(int key)**: Es la operación más compleja. Primero se busca el nodo donde está el elemento. Si no se encuentra se retorna. Si se encuentra y está en un nodo interno, se busca el elemento que le antecede para reemplazarlo. La operación findABB nos indicará cuál es el elemento más cercano a este en la estructura. Teniendo este elemento, se sobre escribe sobre el cual se quería borrar. En caso de que el elemento se encuentre en un nodo externo, sólo se elimina la hoja, dejando un "árbol vacío".

AVL:

Consiste en un ABB que se balancea ante las operaciones de inserción y borrado. Se mantiene el invariante de que el árbol mantendrá una altura máxima de O(log n) + 1. Sus costos en mejor y peor caso son O(log n) para inserción, borrado y búsqueda.

Implementación:

- **GenericNode find(int key)**: La búsqueda es muy similar a ABB, tomando el mismo costo aproximado O(log n). Delega en findABB(int key) la búsqueda del nodo correspondiente a la llave key.
- **void insert(int key, int value)**: Esta función es más compleja pues, similar a como actúa BTree, se debe buscar la hoja del árbol en que debe ubicarse la cadena. Tras determinar, mediante la función de hash, cuál es la página correspondiente, ésta se obtiene (o genera) en la raíz del árbol de hash, en 2 accesos a disco (o 3 si hay overflow, dividiendo la página en 2 nuevas). Si el nodo ya existía, se reemplaza su valor por value.
- **void delete(int key)**: Nuevamente esta función busca recursivamente el elemento a borrar. De no encontrarlo, retorna. De encontrarlo, debe eliminarlo, con su consiguiente balanceo.

- **Node rebalance(Node r, int Side, int times):** esta función se encarga de rebalancear el nodo r entregado, de forma recursiva. Cada vez que encuentra un problema de balance (una diferencia de altura mayor a 2, entre ramas de un nodo) se realizan operaciones de zig-zag para ordenar y balancearlos: *Side* indica para qué lado se debe realizar el balanceo, y *times* indica qué tipo de rotación debe realizarse (simple o doble).
- **boolean isLeaf(), boolean isEmpty(), int height():** son funciones que facilitan el cálculo de ciertas operaciones. isLeaf indica si el nodo actual tiene hijos no nulos, isEmpty indica si el nodo actual es en realidad un nodo vacío ("hijo de una hoja"), y height calcula *on demand* la altura del árbol.
- **Node extractMin():** para las operaciones de rebalanceo ante borrado, se ocupa esta función para quitar la hoja mínima de un extremo del árbol, para que reemplace al elemento borrado.

A pesar del esfuerzo involucrado en la implementación del balanceo, existen casos borde problemáticos para la eliminación de elementos que no pudieron ser corregidos, por lo que la eliminación de elementos para AVL no se realiza en la ejecución de la clase principal.

SplayTree

Consiste en un árbol binario similar a los anteriores, con una gran diferencia: inserciones, búsquedas y borrados implican una operación *splay*, la cual lleva la llave recién insertada o buscada a la raíz del árbol en base a rotaciones. Esto implica que los casos en que se utilizan muchas veces ciertos valores, el costo de obtenerlos baja, pero empeora los costos para valores que se utilizan poco. De esta forma, sus costos amortizados tienden a O(log n)

En SplayTree, las funciones destacables son:

- **GenericNode find(int key)**: Esta operación requiere que el árbol se reordene en base a la función splay(int key). De encontrarse key en el árbol, quedará en la raíz, siendo fácilmente obtenible. En caso contrario, la raíz quedará con un valor distinto.
- **void insert(int key, int value)**: de forma similar a find, esta función solicita que se realice la operación splay previo a su accionar. De existir el nodo, se reemplaza su valor por value. De no existir, se realiza la inserción del nuevo

nodo, cargando la raíz original a la derecha del nuevo nodo, y el contenido de la izquierda de la raíz original a la izquierda del nuevo nodo.

- **void delete(int key)**: nuevamente se solicita la operación splay para llevar a la raíz el nodo con la llave key. De existir, se elimina y se reordenan las dos ramas que le colgaban, para reemplazarlo. En caso contrario, no se realizan operaciones.
- **void splay(int key):** de forma iterativa, esta función va buscando la aparición de esta llave en el árbol, haciendo las rotaciones necesarias si es que se topa con un nodo que no le sirve. Estas rotaciones respetan el orden numérico de los nodos, realizándose hacia la izquierda o a la derecha cuando corresponda (bajo operaciones de zig-zag).

Van Emde Boas

A pesar de mantener ciertas similitudes con las estructuras anteriores, Van Emde Boas es un árbol, pero no binario. Se caracteriza por guardar los valores mínimos y máximos de la estructura en el nodo raíz, almacenando el resto

- **GenericNode find(int key)**:
- **void insert(int key, int value)**:
- **void delete(int key)**:

Finalmente, por problemas en la implementación de las otras estructuras, se optó por no implementar Van Emde Boas, entregando una versión DummyVanEmdeBoas, que sólo entrega funciones vacías para su llamado en la ejecución de la clase principal.

Generador:

Para generar los archivos, se utilizó la clase Generador, la cual entrega (de acuerdo a la cantidad de repeticiones solicitadas), un número de archivos de acuerdo al origen indicado. Sus funciones son:

- **void generate(int iterations)**: de acuerdo a la cantidad de iteraciones indicadas, se generaba dicho número de archivos de la forma

"fakeDNA#.txt", con # el número de la iteración. En ellos, se incluyen 2^25 líneas de texto en formato ADN, con caracteres escogidos al azar desde un arreglo fijo.

- **void dump(int iteraciones)**: tras obtener un archivo modelo desde la página proporcionada en el enunciado, con ADN humano, se recorre y obtiene de él cadenas de ADN de largo 70, las cuales se inspeccionan para comprobar que no tengan caracteres perdidos (marcados con "N", carácter que no nos sirve). De ser correctas, éstas se dividen en 4, entregándose así 4 nuevas líneas al archivo destino. Debido a las dimensiones del mismo, se itera hasta completar 2^25 líneas, reabriendo el archivo desde 0 de ser necesario. Esto da la certeza de que el set de archivos derivados del original serán distintos.

Main:

Para realizar el set principal de pruebas, se escribió una clase Main, que contiene la creación de las 4 estructuras y la toma de muestras de ocupación y accesos a disco para las operaciones de llenado, consulta satisfactoria e infructuosa, y borrado de las mismas. También se incluye

- **String [] init(String filename, int k)**: esta función obtiene de cada archivo los strings a ingresar a las estructuras, y los inserta en un arreglo de String de largo 2^k, con 2^k <= (cantidad de líneas del archivo).
- **String [] generateInt(boolean o, int [][] chain, int l)**: esta función se encarga de obtener 10.000 cadenas de ADN de largo 15 para las pruebas de búsquedas exitosas o infructuosas. Dependiendo del valor de o, estas cadenas se extraen en orden aleatorio desde el arreglo chain, que tiene largo 2^l, o se generan al azar.
- **void main(String [] args):** es la función principal de la clase, la que efectivamente realiza los experimentos:
    - Inserciones: Se realizaron sets de inserciones de 0 a 2^15 elementos, en bloques de (2^n – 2^(n-1)) elementos (considerando siempre el primer bloque de 0 a 2^10). En estos bloques se tomó el tiempo para la totalidad de las inserciones, y el tamaño de cada estructura tras este set de operaciones.
    - Consultas exitosas: tras las inserciones, se obtiene un arreglo de llaves desde el arreglo de inserción original, y se consulta en cada estructura

por su valor. Se tomaron sets de 10.000 consultas iguales para cada estructura. Aquí también se mide el tiempo que demora este set de operaciones.

- Consultas “infructuosas”: se repite la operación anterior para todas las estructuras, con la diferencia que el arreglo de llaves no se extrae desde el original, si no que se obtiene con llaves al azar, para simular elementos que no estén ingresados.
- Eliminación de elementos: se randomiza la lista de llaves insertadas originalmente, y se eliminan en sets de 2^n a 2^(n-1), desde 15 hasta 10, y luego un set de operaciones de 2^10 hasta 0. Aquí nuevamente se obtiene el tamaño de las estructuras, así como el tiempo que toma ejecutar el set de operaciones.

Atributos de Main:

A la función main de la clase Main se le entregan atributos al momento de la ejecución, obtenidos a través de la librería Commons de Apache; estos argumentos se entregan mediante consola, anteponiendo un “-“ antes de ellos.

- i = log(número mínimo de elementos a ingresar), en este caso 10
- I = log(número máximo de elementos a ingresar), en este caso 15
- iterations = número de iteraciones a probar, en este caso 5.
- -javaagent:lib/classmexer.jar Para la medición de tamaño de las estructuras se utilizó la librería classmexer.jar, la cual debe ser incluída como variable de la máquina virtual de Java.

También se destaca que el número de elementos distintos en la búsqueda exitosa o infructuosa de cadenas en las estructuras es siempre 10.000, fijados por el programador.

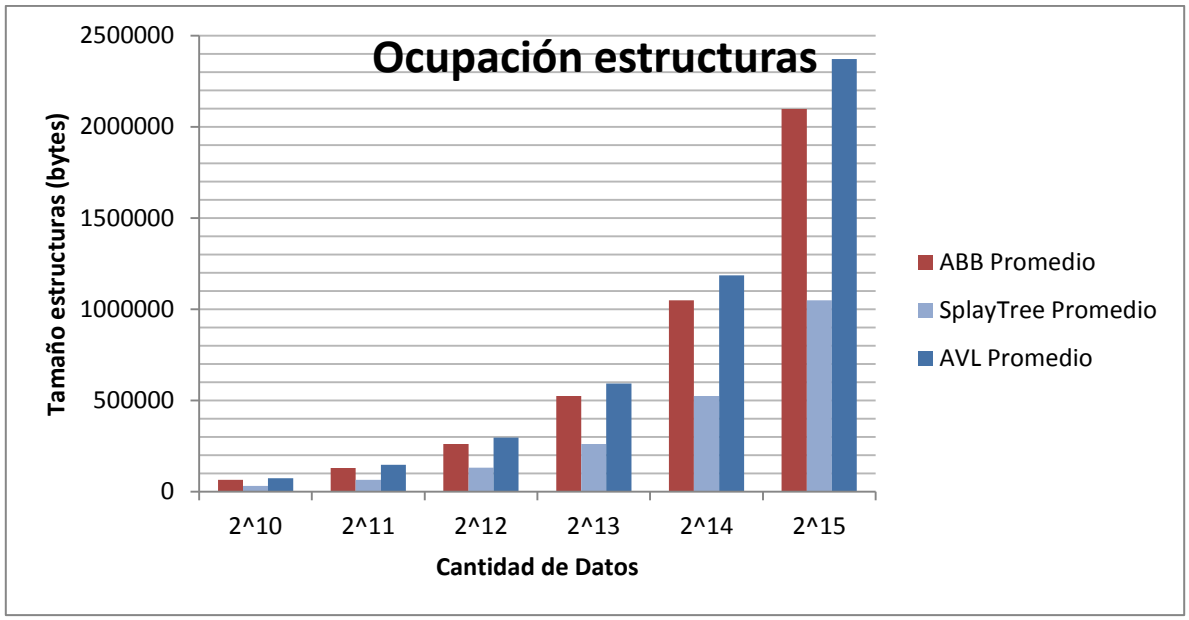
Finalmente, se utiliza la librería Math3 de Apache, en específico la estructura SummaryStatistics para mantener las estadísticas de cada uno de las tomas de muestras:

- Tamaño de las estructuras a medida que se llenan, tomadas en 5 ocasiones (inserción de 2^k elementos, con k entre i e I)
- Tamaño de las estructuras a medida que se vacían, tomadas en 5 ocasiones (inserción de 2^k elementos, con k entre I-1 e i)
- Tiempo de operaciones:
    - Tras insertar 2^k elementos en cada estructura, con k entre i e I, acumulando el conteo desde
    - Tras 10.000 búsquedas exitosas sobre la estructura (buscando cadenas que han sido fehacientemente insertadas), tras cada paso de inserción, habiendo reseteado los contadores de IO
    - Tras 10.000 búsquedas “infructuosas“ sobre la estructura (buscando cadenas generadas al azar, las cuales podrían probabilísticamente estar presentes en la estructura de todas formas), tras cada paso de inserción, habiendo reseteado los contadores de IO
    - Tras haber eliminado (2^k – 2^(k-1)) elementos desde las estructuras, con k entre I e i+1. Es decir, mediciones cuando hay presentes 2^(k-1) elementos en las estructuras
    - Tras haber eliminado la totalidad de los elementos desde las estructuras.

Análisis

De acuerdo a los datos recabados en los análisis realizados, con 5 iteraciones, se obtuvieron los siguientes resultados (detalle de tablas completas en Anexo):

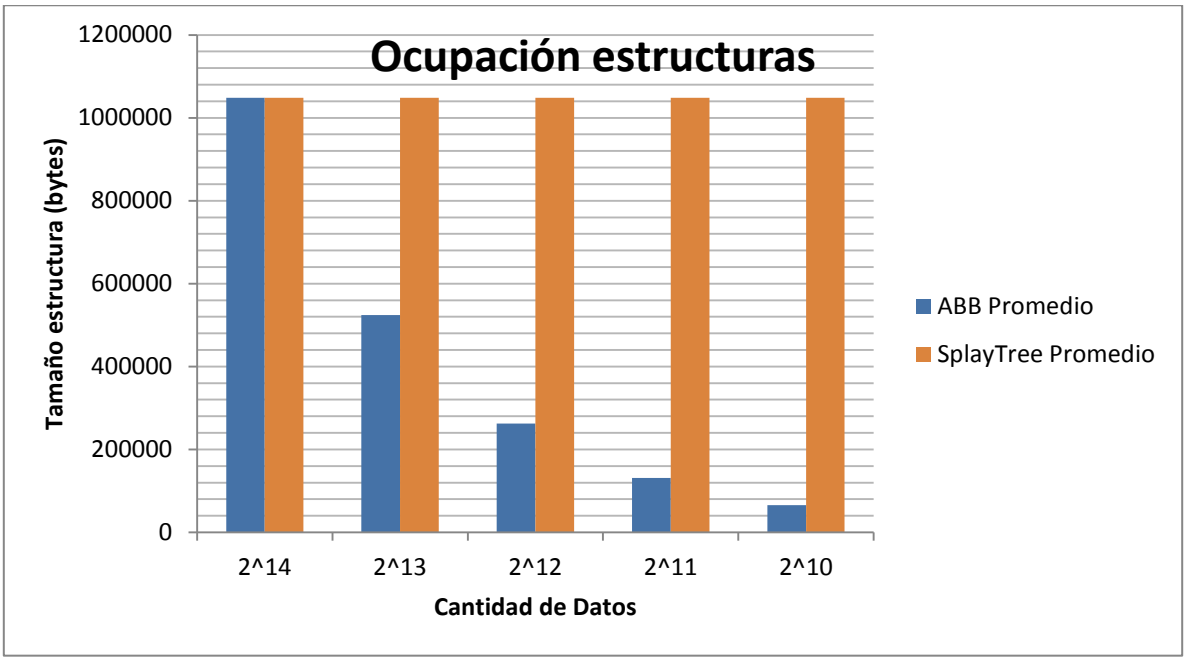
Ocupación de las estructuras en inserción:



De acuerdo a las pruebas realizadas, AVL, SplayTree y ABB muestran una tendencia similar en la necesidad de espacio para cada una de sus estructuras. Se puede asumir que, descontando las constantes de tamaño por implementación (ABB y AVL tienen nodos-subárboles, mientras que SplayTree tiene sólo nodos), el tamaño tiende al esperado O(n log n)

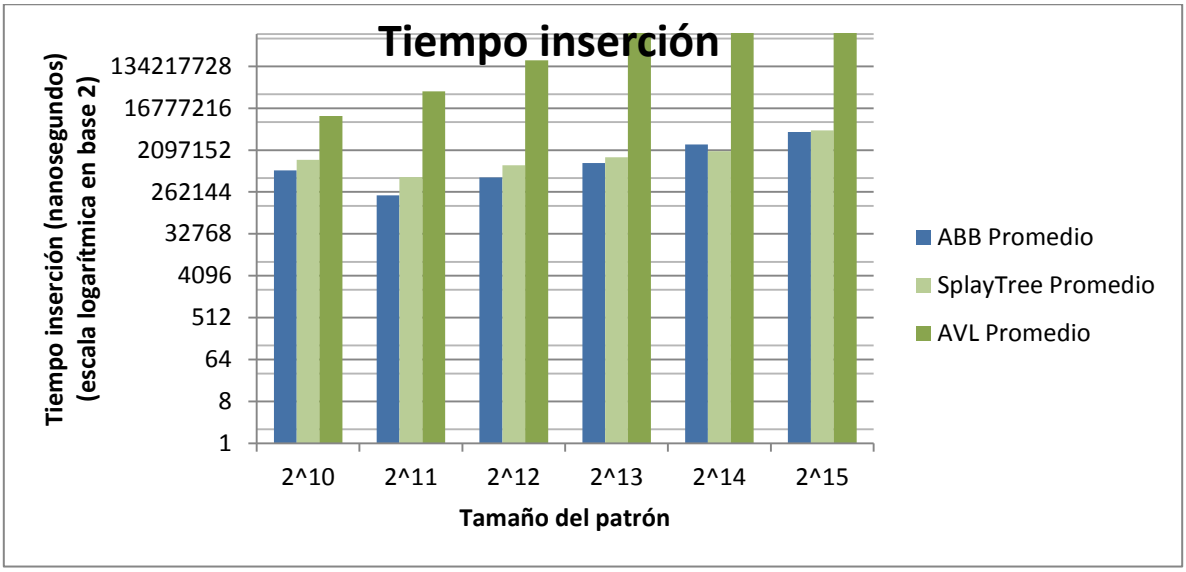
El error estándar más destacable en la medición es el de 0,0% casi constante para ABB y SplayTree, mostrando una independencia del input. Sin embargo, AVL depende de éste, notándose variaciones de hasta 0,1% de error.

Ocupación de las estructuras en borrado:



Lamentablemente, este gráfico expresa posibles errores de implementación de SplayTree, pues se esperaba que su tamaño disminuyera en cuanto se eliminaran elementos. Por su parte, ABB muestra lo esperado, eliminando elementos a ritmo decreciente, constatando su ocupación de O(n log n). El error para esta medición es despreciable.

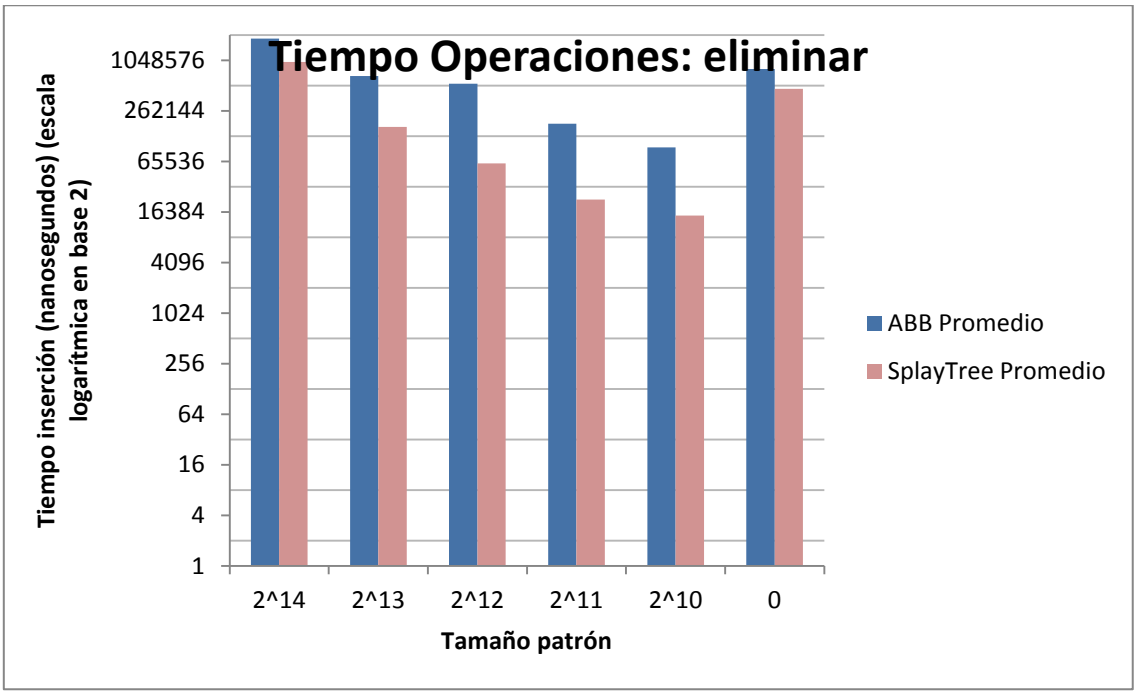
Tiempo Operaciones en inserción:



Como muestran los gráficos dibujados en escala logarítmica en base 2, el desempeño de ABB muestra un crecimiento acorde a la cantidad de elementos ingresados, salvo una anomalía en las primeras inserciones. Para AVL y SplayTree, se nota la necesidad de balancear las estructuras cada vez que se operan (sobre todo AVL).

Dado que las mediciones están en nanosegundos, las variaciones de tiempo en las operaciones es muy alta (hasta 86% de error en SplayTree D:!), lo cual puede sin lugar a dudas anular la validez de este test.

Tiempo Operaciones en borrado:

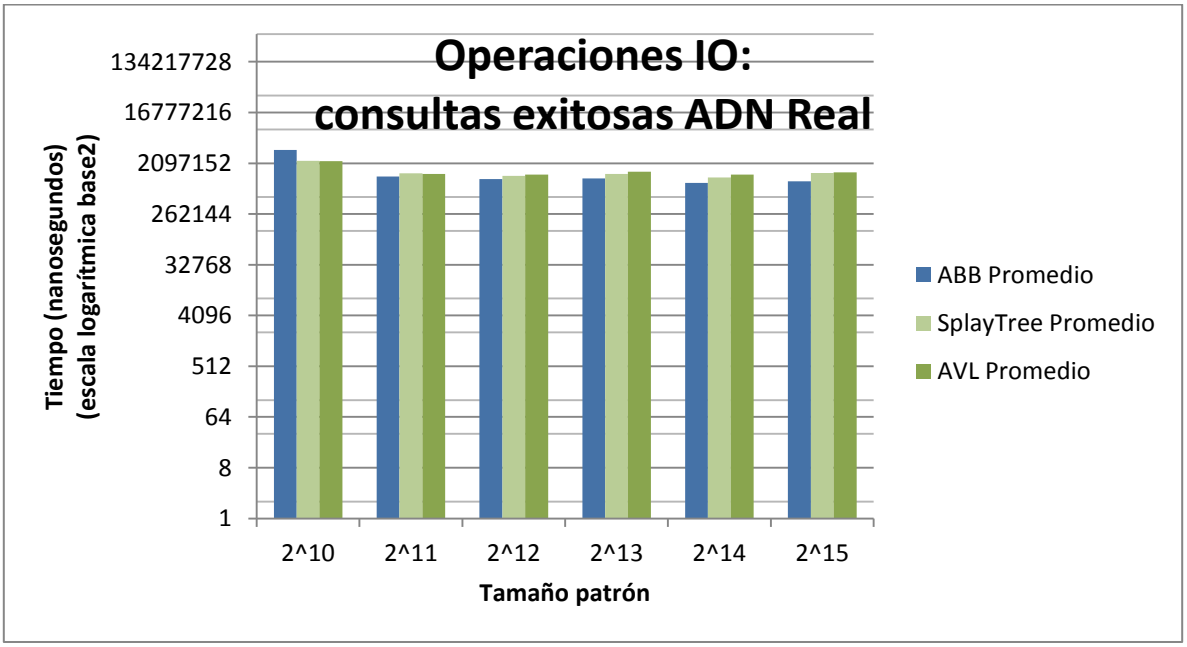


En eliminación, nuevamente tuvimos que excluir el testeo de operaciones entrada y salida para Hash Lineal.

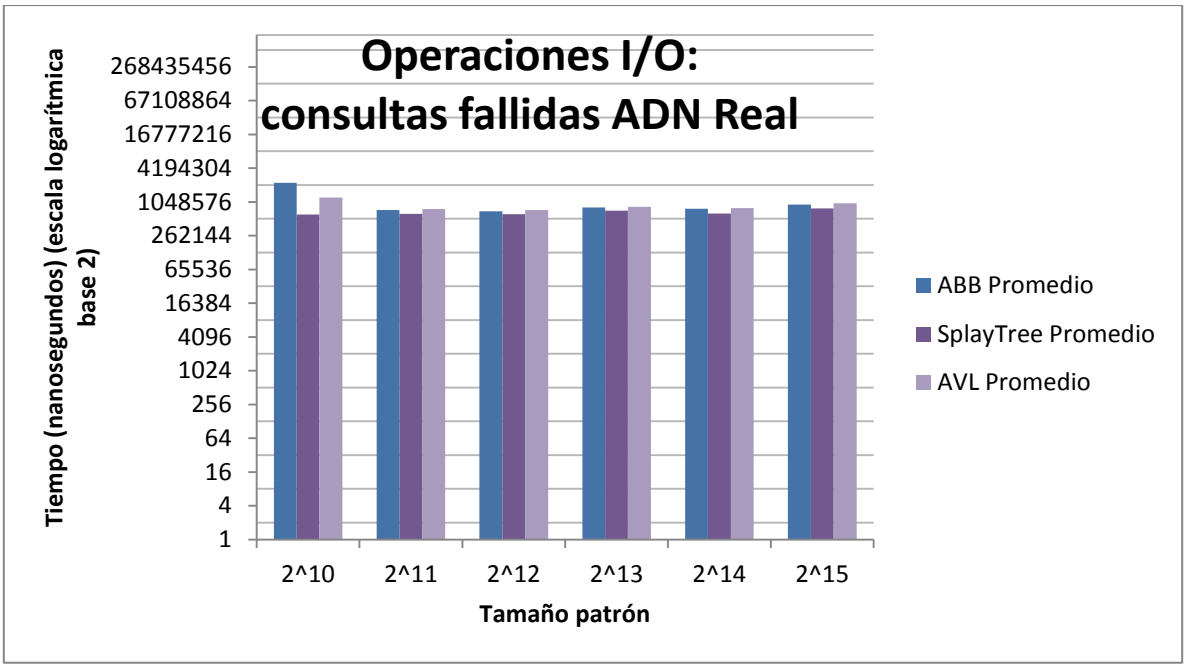
Para ambas muestras, podemos observar un desempeño mucho más acotado en accesos a disco para la cantidad de elementos manejados, sobre todo para BTree. Para el caso de Hash Extendible, se mantiene la tendencia de mantener su desarrollo entre Orden(2*n) y Orden(3*n) operaciones, por la compresión: es decir, entre O(2) y O(3) por operación de eliminación.

Dado que las mediciones están en nanosegundos, las variaciones de tiempo en las operaciones es muy alta (hasta 86% de error en SplayTree D:!), lo cual puede sin lugar a dudas anular la validez de este test.

Tiempo Operaciones en búsquedas exitosas:



Operaciones I/O en búsquedas "infructuosas"



Conclusiones

El objetivo de esta tarea es probar la eficiencia en tiempo de ejecución de operaciones de inserción, borrado y búsqueda en cada uno de las estructuras solicitadas. La principal meta era cuantificar la saturación de estas estructuras en su mínimo y máximo, controlando así el uso de espacio y las demoras en sus ejecuciones (malas distribuciones de elementos son malas para las búsquedas y eliminaciones). Aumentando la saturación, los algoritmos se asimilan a árboles binarios perfectamente balanceados.

Tras realizar mediciones sobre las operaciones de inserción, búsqueda, búsqueda infructuosa, y borrado sobre las estructuras anteriormente mencionadas, se concluye que:

ABB

ABB fue la estructura más acorde a los resultados esperados, tanto por su implementación como por su desempeño particular.

AVL

AVL es la estructura que más entrega sospechas por su implementación, siendo evidente que la función de balanceo tiene problemas y puede haber influido en el resultado final. A pesar de ello, su rendimiento va a la par de los otros algoritmos en las funciones que se pudieron probar.

VanEmdeBoas

Van Emde Boas no se pudo implementar, queda pendiente revisar su funcionamiento.

SplayTree

En la implementación del SplayTree se alcanzaron buenos resultados, siendo muy competitivo con respecto a ABB. Sus tiempos de operación estuvieron a la par de los otros algoritmos, según lo esperado

A pesar de sus características, su desempeño en las búsquedas y eliminaciones es aceptable, sólo superado por el ABB en ciertas situaciones.

Finalmente, queda remarcar el desempeño del SplayTree, mostrándose eficiente en cuanto a buena parte de los test ejecutados, así como competitivo a pesar de su carácter poco intuitivo. Sin embargo, es importante mencionar la necesidad de revisar exhaustivamente la implementación de todos estos algoritmos, por cuanto no pudimos realizar la totalidad de los tests, tanto en operaciones como en cantidad de datos manejados.

Anexo

Las tablas de resultados y sus gráficos se encuentran en el archivo Resultados Tarea 3.xlsx