2015

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

Cristian Andrade Muñoz
Javier Liberman Salazar

[DISEÑO Y ANÁLISIS DE ALGORITMOS: DICCIONARIOS EN MEMORIA SECUNDARIA]

Análisis cuantitativo del desempeño de algoritmos y estructuras de almacenamiento en disco (BTree, Hash Extensible y Hash Lineal en 2 versiones), de acuerdo a su acceso a disco y porcentaje de ocupación, sobre el almacenamiento de cadenas de ADN.

Resumen Ejecutivo:

Se asignó la construcción del código fuente para la implementación de algoritmos y estructuras de almacenamiento de texto en memoria secundaria, específicamente estructuras BTree, Hash Extendible y Hash Lineal (en 2 versiones, dependiendo de su método de expansión). 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 almacenamiento externo se utilizó un disco tradicional de 2 Terabytes conectado mediante SATA 3, con bloques de memoria de 4096 bytes.

Para los archivos de búsqueda, se utilizaron archivos de cadenas de ADN) generados a partir de código provisto en ftp://ftp.ncbi.nih.gov/genomes/, específicamente genoma humano, y ADN sintético generado al azar (ambos en su formato de caracteres C, G, A, T).

Los resultados del algoritmo utilizado indican que el algoritmo/estructura con mejor desempeño *over all* es el algoritmo Hash Extensible, mostrando densidad de ocupación cercana al 70%, cercana a los valores experimentales obtenidos por investigaciones exhaustivas, y demostrando ser más eficiente en su acceso a disco para inserción, eliminación y búsqueda de elementos, en este caso, cadenas de caracteres un alfabeto acotado (ADN).

Introducción:

La búsqueda en texto es una tarea del día a día en los tiempos modernos, donde es necesario aplicarla en situaciones desde un procesador de texto básico, hasta en procesos de análisis de datos de gran envergadura, en laboratorios, observaciones astronómicas, servicios de búsqueda de datos online, entre otras. Es necesario medir, en estos casos, el desempeño de los algoritmos utilizados, buscando con ello el más adecuado en cuanto a la tasa de ocupación de los datos ingresados con respecto al tamaño de las estructuras, la cantidad de datos revisados, y la dificultad de implementarlos.

Con el fin de analizar el rendimiento de distintos algoritmos/estructuras de ordenamiento en memoria secundaria, se realizan mediciones de porcentaje de ocupación de páginas, y cantidad accesos a disco en múltiples experimentos, aplicando conceptos y técnicas vistas en clases tanto en la construcción de dichos experimentos como en el análisis de los resultados.

Los algoritmos solicitados a analizar son Hashing Extensible, Hashing Lineal en su versión de expansión y contracción dependiente de la y B-Trees, y su rendimiento será evaluado en una serie de inserciones, búsquedas y eliminaciones, en strings de secuencias de ADN de 15 caracteres, extraídos de muestras reales de genoma humano[1] o generados al azar, en repeticiones de 2^20, 2^21, 2^22, 2^23, 2^24 y 2^25 líneas.

Finalmente, por dificultades en la programación y en el tiempo total empleado en la ejecución, se redujo estas cantidades a tramos de repeticiones de 2^10, 2^11, 2^12, 2^13, 2^14 y 2^15 elementos totales insertados.

[1] Extraidos de ftp://ftp.ncbi.nih.gov/genomes/

[2] http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html

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, por lo que no representa una variable relevante al momento de comparar los algoritmos.
- Por otra parte, debería existir una leve variación entre las ejecuciones sobre ADN real y ADN sintético, debido a las limitaciones que presentan las bases nitrogenadas en su disposición en las cadenas, las cuales no se reflejan en la generación de archivos de genoma sintético.
- Dadas las limitantes de tiempo para la ejecución del programa, tanto por la **velar por la seguridad del hardware utilizado**, se optó por reducir el conjunto de elementos a suministrar a las estructuras, lo cual igualmente debe entregar una muestra representativa de su desempeño. Consideramos B = 4096 y N = n° de elementos en la simulación que, por razones detalladas en la Implementación, redujimos a un rango entre 2^10 y 2^15 elementos.
- Cada string contiene 15 caracteres, los cuales, por implementación, se guardan directamente en el caso de BTrees, y de forma binaria en el caso de los hash (convirtiendo cada letra a una codificación de 2 bits). Esto entrega, para el segundo caso, un aproximado virtual de 128 strings por página en disco.
- A medida que el texto crece, tanto la inserción como la búsqueda sobre la estructura BTree debería tender a O(logB n), mientras que la búsqueda en Hash Extensible y Lineal Versión 1 debería ser constante, dependiendo su tiempo de ejecución sólo en la expansión y contracción de cada una de las versiones lineales.
- Por otra parte, suponemos que las tasas de ocupación se mantendrán sobre el 50% en el caso de todas las estructuras. Sin embargo, experimentalmente se ha demostrado que la ocupación de Hash Extensible se mantiene (en promedio) en torno al 69%, lo cual será revisado con este experimento.

Diseño e Implementación

Todas las implementaciones ocupan la clase **DiskSimulator** para realizar sus operaciones de Input-Output. Al inicializar una nueva instancia de DiskSimulator, éste se encarga de crear un nuevo archivo vacío del tipo **RandomAccessFile**, que se comporta como un arreglo de bytes guardado directamente en disco[2]. Sabiendo que en la máquina de prueba el largo B, o tamaño de página, es **512 bytes**, DiskSimulator provee de las funciones **getNextFreePage** que retorna el índice de una página sin utilizar, **getPage(int pageNumber)** que retorna un arreglo de 512 bytes con los contenidos de la página y **writePage(int pageNumber, byte[] bytes)** que escribe en la página el arreglo de bytes.

Además de esto, todos los algoritmos implementan una interfaz común, **DiskMemoryManager**, que define un protocolo para las operaciones de diccionario y mediciones de la eficiencia.

Métodos

- **String find(String chain)**: Busca en la estructura de datos el String chain. Retorna chain si lo encuentra, sino, retorna null.
- **boolean delete(String chain)**: Busca una cadena en la estructura. Si la encuentra la elimina y retorna true. Si no la encuentra retorna false.
- **void add(String chain)**: Agrega una cadena a la estructura
- **float getOcupation()**: Retorna el porcentaje de ocupación de las páginas de disco de esta estructura, delegando su cálculo a DiskSimulator.
- **int getIOs**: Retorna el número de operaciones IO que se han realizado desde el inicio del programa o desde que se llamó por última vez a resetIOs.
- **void resetIOs**: Resetea el contador de IOs

Se define también las estructuras que están siendo probadas en esta tarea:

[2] http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html

BTree

Consiste en un árbol de datos, en que cada nodo tiene asignada una página en disco. En consecuencia, tiene un almacenamiento de 512 bytes por nodo. Cumple las siguientes invariantes:

- Cada nodo almacena a lo más floor{B/Tamaño de elemento} elementos
- Salvo la raíz todo nodo tiene al menos floor{B/Tamaño de elemento}/2 elementos.
- Un nodo interno con k elementos tiene k+1 hijos
- Todas las claves en el i-ésimo hijo (1<=i<=k+1) están entre, en valor lexicográfico, los valores entre los elementos i-1 e i.
- Todas las hojas están a la misma profundidad.

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.

- **String find(String chain)**: Es la operación más sencilla. Parte buscando en la raíz el elemento, si lo encuentra lo retorna. En caso de no encontrarlo, desciende por el hijo con índice i, donde i es el primer elemento dentro de los contenidos que es mayor lexicográficamente a chain. Si dicho hijo no existe, retorna nulo. Si existe sigue buscando recursivamente.
- **void add(String chain)**: Haciendo un find infructuoso se encuentra el nodo del árbol donde debería insertarse la cadena. En caso de que el find sea exitoso, el árbol ya contiene la cadena y no se debe hacer nada. En este nodo se desea realizar la inserción en el índice i donde i+1 es el primer elemento que es lexicográficamente mayor a chain, o 1 si el nodo está vacío. Sin embargo, antes de realizar la inserción, se verifica que tenga espacio para una inserción en su hoja de disco. Si lo tiene, se inserta. Si no lo tiene se divide la hoja en dos y se promueve la mediana al elemento padre. Ahora si se inserta el nuevo elemento en la hoja que corresponda y se manejan posibles overflow en cadena que puedan haber en los padres. En caso de que la raíz haga overflow, se agrega una nueva raíz con un solo elemento.
- **boolean delete(String chain)**: Es la operación más compleja. Primero se busca con find el nodo donde está el elemento. Si no se encuentra se retorna false. Si se encuentra y está en un nodo interno, se busca el elemento que le sigue

aumentando en 1 el valor lexicográfico de chain. Esto resulta sencillo ya que con el alfabeto limitado de 'A', 'C', 'G', 'T' es trivial implementar una función que encuentre el sucesor o antecesor en caso de no poseer alguno. Se busca aquel elemento, encontrarlo es altamente improbable, pero el find 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 el borrado es más complejo. Se elimina de la hoja, si la hoja mantiene el invariante de tener al menos floor(B/tamaño de los elementos)/2 elementos no hay problema. En caso contrario se debe buscar en los hermanos algún elemento para pedir. En caso de que ningún hermano tenga, se pide un elemento al padre y se fusiona con un hermano.

Hashing extensible:

Consiste en ocupar los bits del hash de los elementos para decidir las páginas de disco donde debo almacenar y buscar. Una buena función de hash debería distribuir aleatoriamente los 0 y 1 en todos los bits. Luego este hash se puede utilizar como un *trie* de búsqueda.

Implementación: Dado que, en total, hay cuatro caracteres diferentes dentro de nuestro alfabeto, una buena función de hash, que distribuye uniformemente, es transformar cada uno de los caracteres en los bits 00, 01, 10 o 11. Se mantiene un árbol en memoria que guarda los primeros n bits de las paginas de disco, donde n es log_2 numero de paginas.

- **String find(String chain)**: La búsqueda es trivial y siempre toma un acceso a disco en caso de que n no haya alcanzado el tamaño de los hash.
- **void add(String chain)**: 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).
- **boolean delete(String chain)**: Esta función también es trivial. Busca a través del *trie* y, llegando a la página que contiene la clave, la elimina.

Hashing Lineal:

Consiste en ocupar los bits del hash de los elementos para decidir las página de disco donde debo almacenar y buscar. A diferencia del hash extensible, que tiene forma de árbol, éste tipo de hash extiende lateralmente (linealmente) su cantidad de hojas, similar a un arreglo de largo variable. La expansión y contracción de su tamaño se puede controlar bajo dos estrategias distintas:

1. Expandir cuando la tasa de ocupación de las hojas es alta; contraer cuando la tasa de ocupación es baja. Esto lleva a que expansión y contracción se evalúen en inserciones y borrados.
2. Expandir cuando el tiempo promedio de búsqueda en la estructura aumente considerablemente; contraer cuando el tiempo promedio de búsqueda se reduzca considerablemente. Esto lleva a que expansión y contracción se evalúen en las búsquedas (las cuales también se realizan cuando se inserta o borra).

Ambos tipos de hash están implementados a partir de una clase padre en común, la cual implementa las funciones para obtener las páginas correspondientes en disco, así como las funciones de expandir y contraer.

En Hash Lineal V1, las funciones destacables son:

- **String find(String chain)**: a través del Bucket, se solicita la página en que está presente la cadena buscada, mediante la función de hash, y luego se procede a buscar en dicha página. Esto conlleva 1 acceso a disco.
- **void add(String chain)**: a través de Bucket, se obtiene la página a disco que debe ser modificada para agregar la cadena. Tras esta operación, se chequea si se debe expandir la estructura, de acuerdo a su ocupación. Si no se expande, conlleva 2 accesos a disco. De lo contrario, requiere 3.
- **void delete(String chain)**: a través de Bucket, se obtiene la página a disco que debe ser modificada para eliminar la cadena. Tras esta operación, se chequea si se debe contraer la estructura, de acuerdo a su ocupación. Si no se contrae, conlleva 2 accesos a disco. De lo contrario, requiere 3.

En Hash Lineal V2, las funciones destacables son:

- **String find(String chain)**: a través del Bucket, se solicita la página en que está presente la cadena buscada, mediante la función de hash, y luego se procede a buscar en dicha página. Esto conlleva 1 acceso a disco. A su vez, se toma el tiempo empleado en encontrar exitosa o infructuosamente la cadena, y se compara con su desempeño anterior. De variar considerablemente, se evaluará expandir o contraer el hash, considerando allí 3 accesos a disco (para dividir una página en 2, o unir dos páginas en 1).
- **void add(String chain)**: a través de Bucket, se obtiene la página a disco que debe ser modificada para agregar la cadena. Tras esta operación, se chequea si se debe expandir la estructura, de acuerdo a su ocupación. Si no se expande, conlleva 2 accesos a disco. De lo contrario, requiere 3.
- **void delete(String chain)**: a través de Bucket, se obtiene la página a disco que debe ser modificada para eliminar la cadena. Tras esta operación, se chequea si se debe contraer la estructura, de acuerdo a su ocupación. Si no se contrae, conlleva 2 accesos a disco. De lo contrario, requiere 3.

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 [] generateChains(boolean o, String [] 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.

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.
- rd = confirmación de que se probará con archivos de ADN real.
- fd = confirmación de que se probará con archivos de ADN sintético.

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 enunciado.

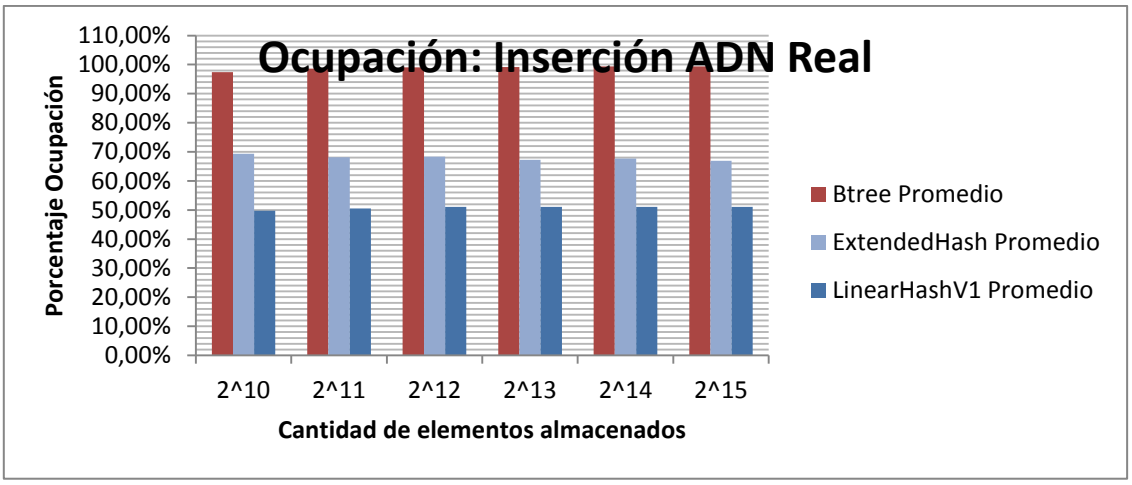
Finalmente, se utiliza la librería Math3 de Apache, en específico la estructura SummaryStatistics para mantener las estadísticas de cada uno de las tomas de muestras:

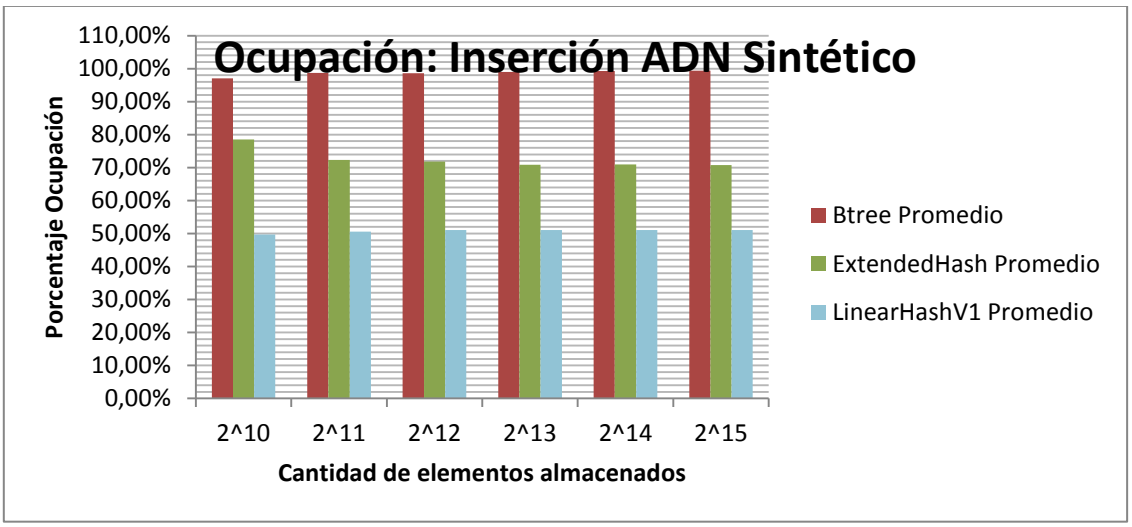
- Porcentaje de ocupación de las estructuras a medida que se llenan, tomadas en 5 ocasiones (inserción de 2^k elementos, con k entre i e I)
- Porcentaje de ocupación de las estructuras a medida que se vacían, tomadas en 5 ocasiones (inserción de 2^k elementos, con k entre I-1 e i)
- Número de operaciones IO:
    - 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:



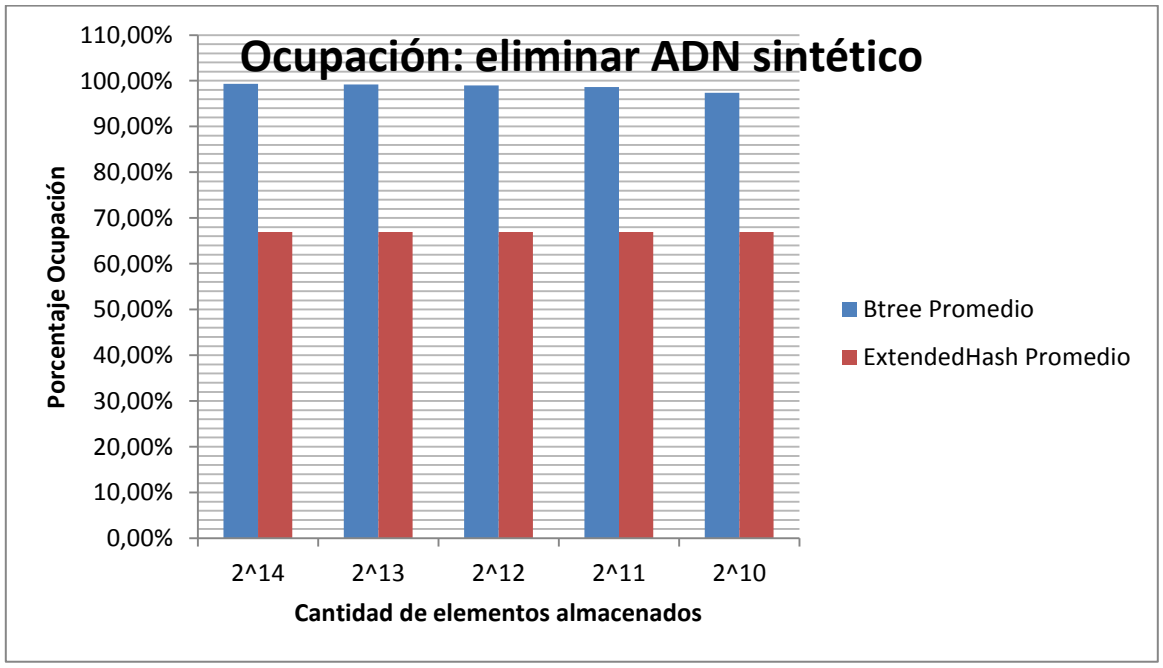


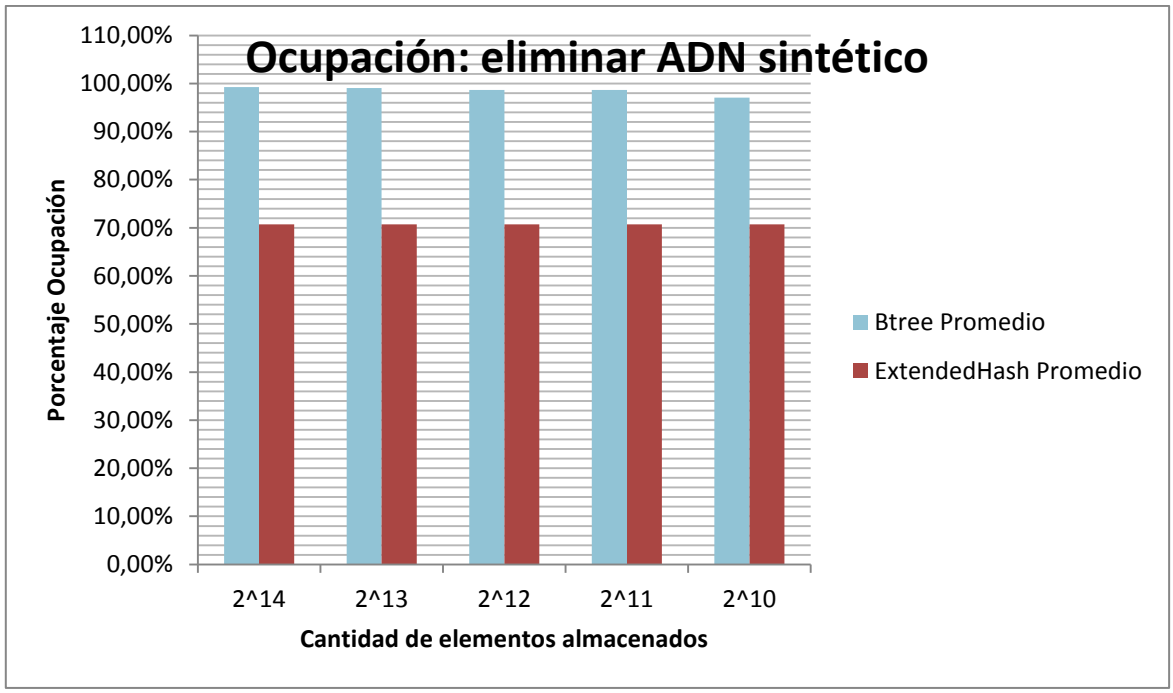
De acuerdo a las pruebas realizadas, BTree muestra una saturación de la estructura tendiente al 100%, dejando muy poco espacio incompleto en las hojas. Por su parte, Hash Extensible se acerca mucho a los resultados experimentales demostrados en investigaciones más exhaustivas (69%), tendiendo a dicho valor a medida que se agregaban más datos; sin embargo, el desempeño para insertar ADN sintético muestra que la saturación de las páginas llegaba al 78%, lo cual decae rápidamente según lo indicado anteriormente. Por otro lado, el desempeño

de Hash Lineal V1 es pobre, bordeando y tendiendo al 51% de saturación de su capacidad.

El error estándar más destacable en la medición es el de 4,45% para Hash Extensible en 2^10 elementos insertados desde ADN sintético, por debajo del límite fijado de 5%. Con un 1,07%, el mayor error estándar de BTree se produce en 2^10 inserciones de ADN real, mientras que Hash Lineal no presenta error porcentual en la muestra.

Ocupación de las estructuras en borrado:



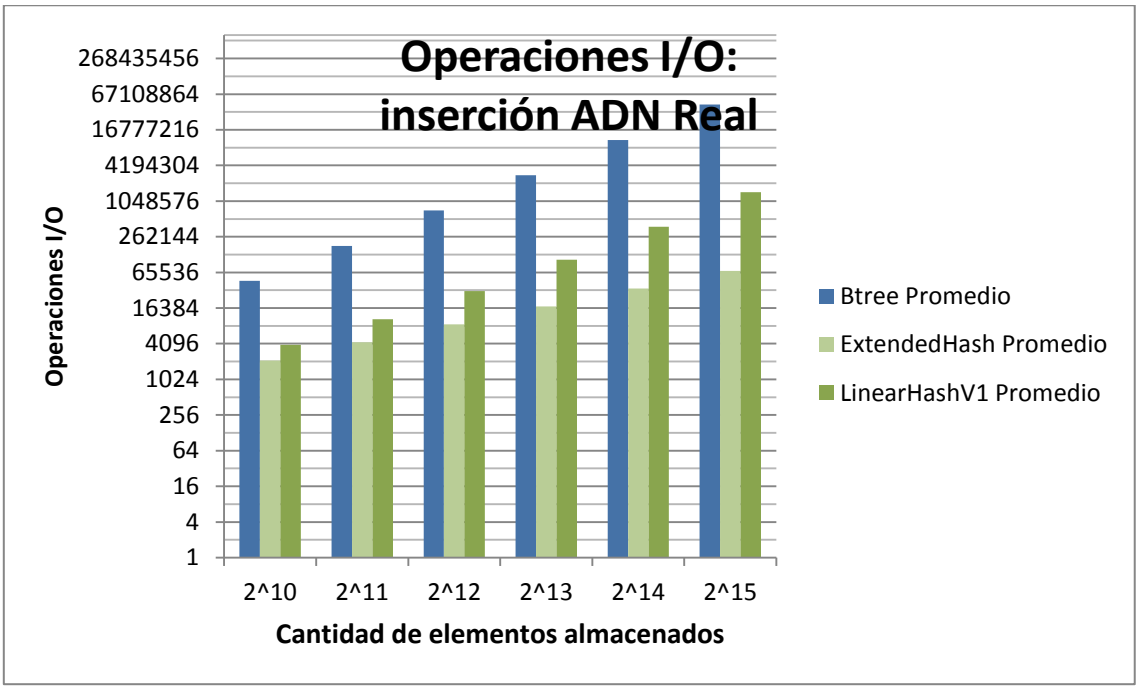


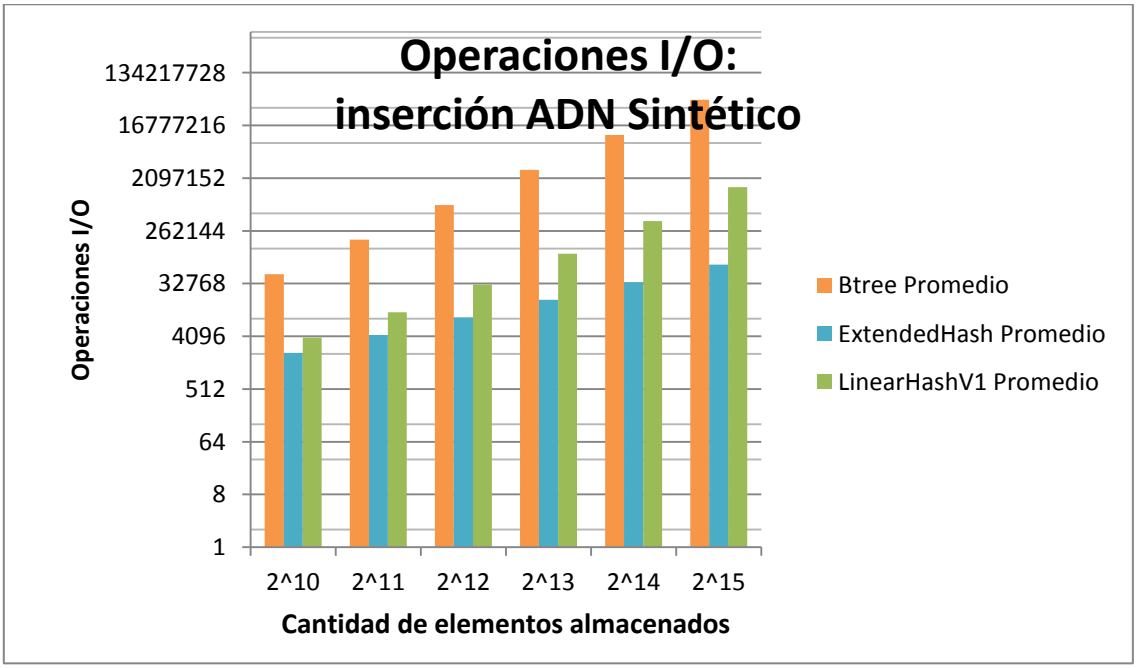
En contraste con la inserción, en la eliminación de datos desde las estructuras se observa una inversión en las curvas de ocupación porcentual. Sin embargo, la tendencia se mantiene: BTree mantiene una ocupación que sobrepasa el 97%, con desarrollos casi idénticos para ADN real y sintético. Para el caso de Hash Extensible, el porcentaje de ocupación se estanca en un 66.9% en ADN real, y 70.7% en ADN sintético, acercándose bastante a los resultados de experimentos formales exhaustivos anteriormente mencionados (69%).

Lamentablemente, para Hash Lineal, experimentalmente tuvimos problemas en la implementación de las funciones de borrado, volviéndose inestable, lo cual nos llevó a no considerarlo en el estudio.

El error estándar para BTree se mantuvo estable bajo el 0,025% en ambos tests, mientras que para Hash Extendido se mantuvo bajo el 1%.

Operaciones I/O en inserción:



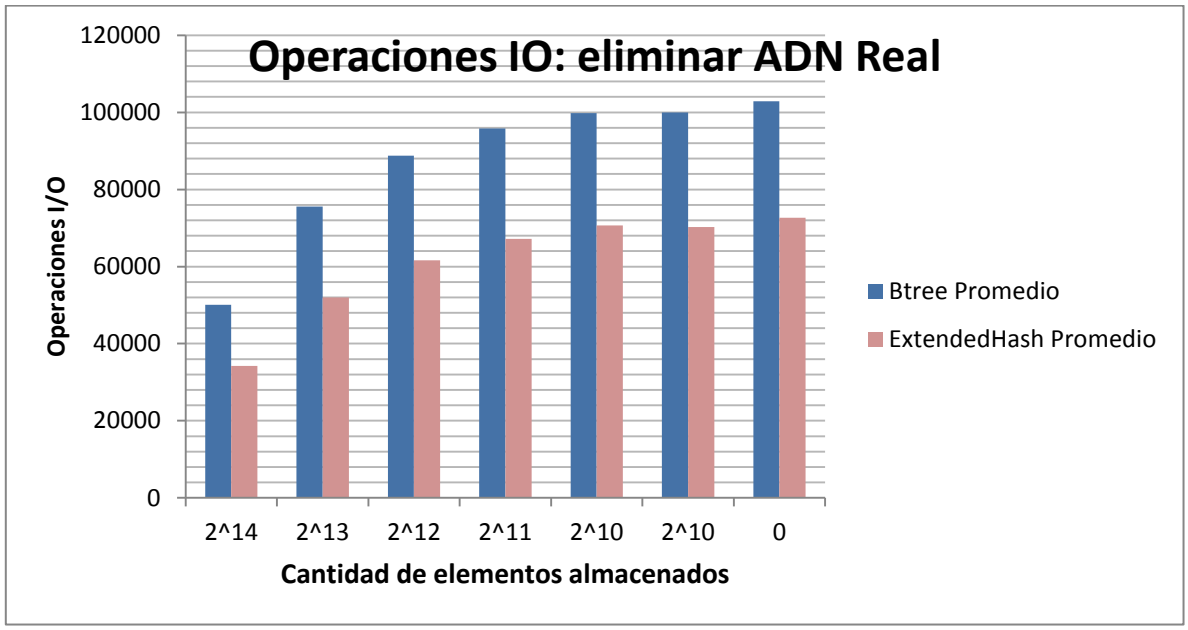


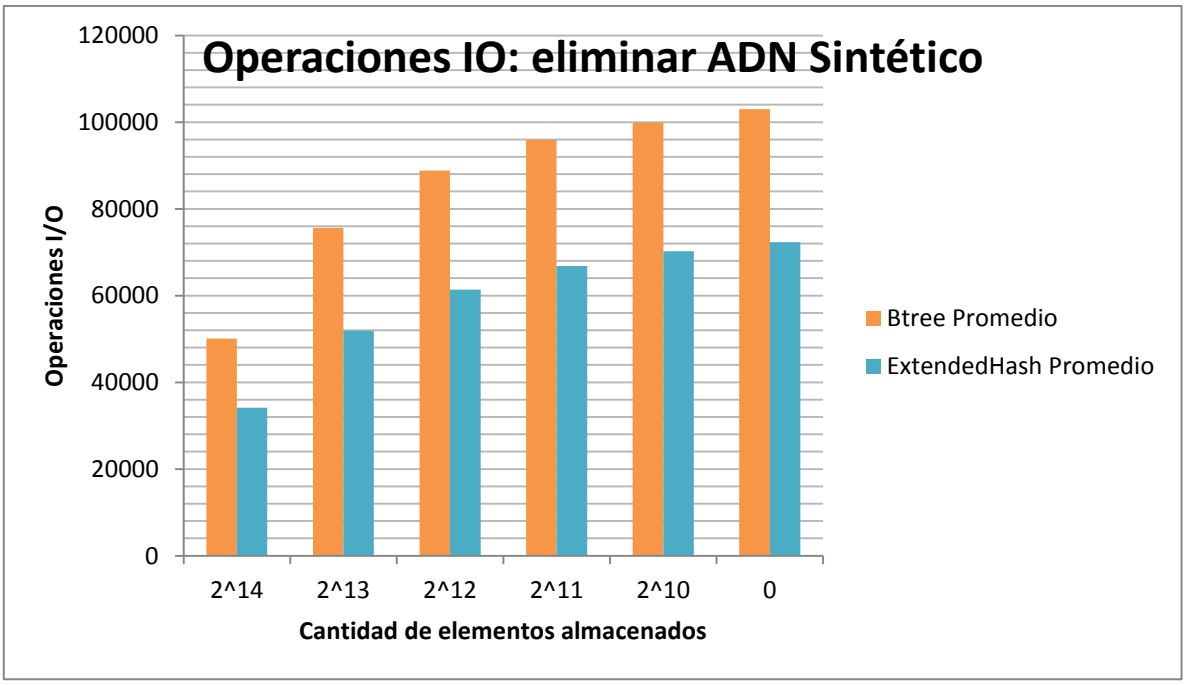
Como muestran los gráficos dibujados en escala logarítmica en base 2, el desempeño de BTree empeora notablemente con respecto a sus competidores, mostrando grandes diferencias entre cada iteración. Por su parte, Hash Extensible y Lineal muestran crecimientos más estables, siendo el primero el que necesita menos operaciones a disco. De cualquier forma, se muestra que la cantidad de

accesos a disco para los casos de Hash se mantienen en torno a lo esperado (entre Orden(2*n) y Orden(3*n), de acuerdo a la necesidad de expansión de cada uno, o sea, entre O(2) y O(3) por consulta).

El error estándar asociado a las pruebas de BTree no supera el 2,2% para casos de ADN real, mientras que para ADN sintético las pruebas muestran un error de porcentual de milésimas.

Operaciones I/O en borrado:



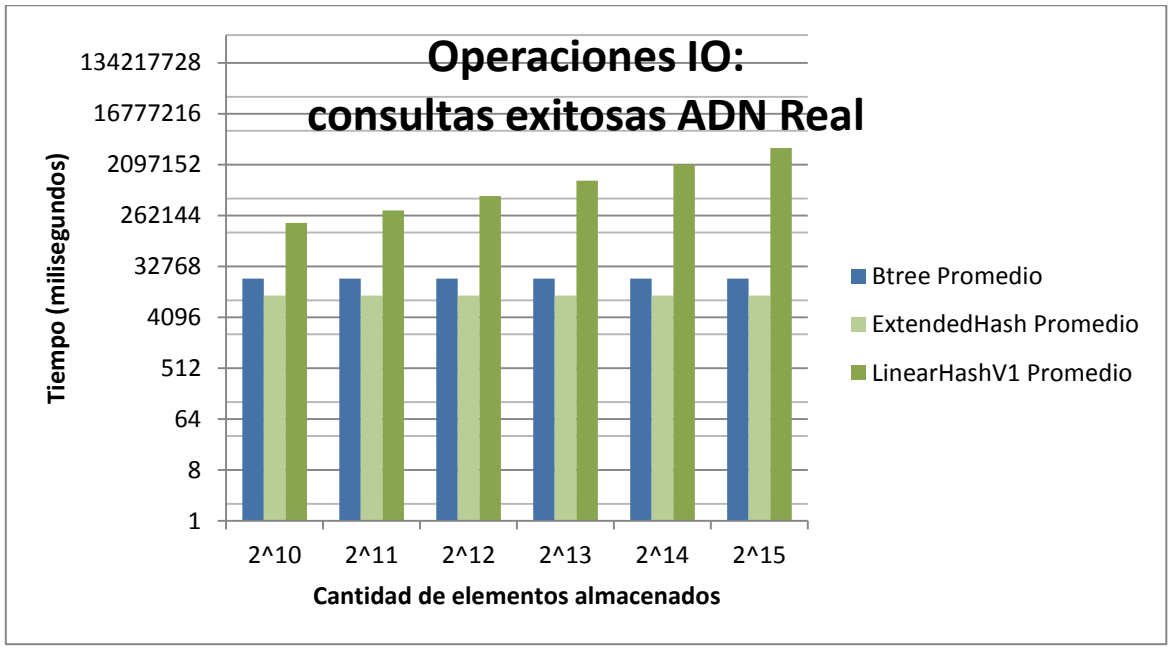


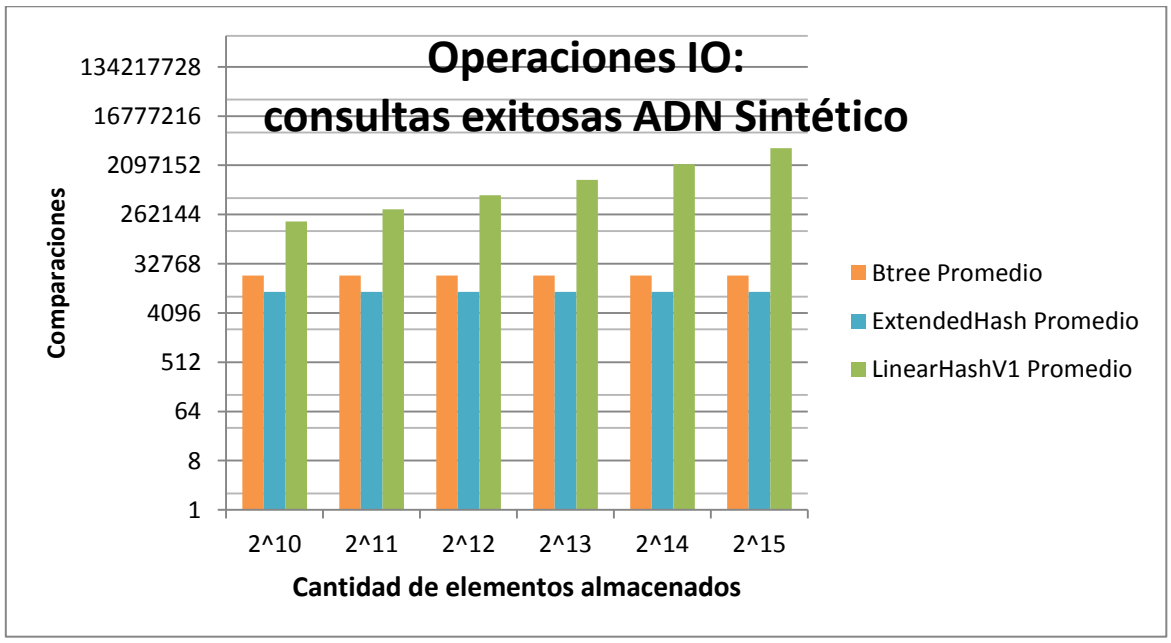
En eliminación, nuevamente tuvimos que excluir el testeo de operaciones entrada y salida para Hash Lineal.

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.

En cuanto al error asociado en las muestras de BTree, se mantiene bajo el 0,08% para ADN real, y se anula para ADN sintético (desempeño estable). Para Hash Extensible, el error asociado se mantiene bajo el 0,1% para ADN real, y 0,04% para ADN sintético.

Operaciones I/O en búsquedas exitosas:

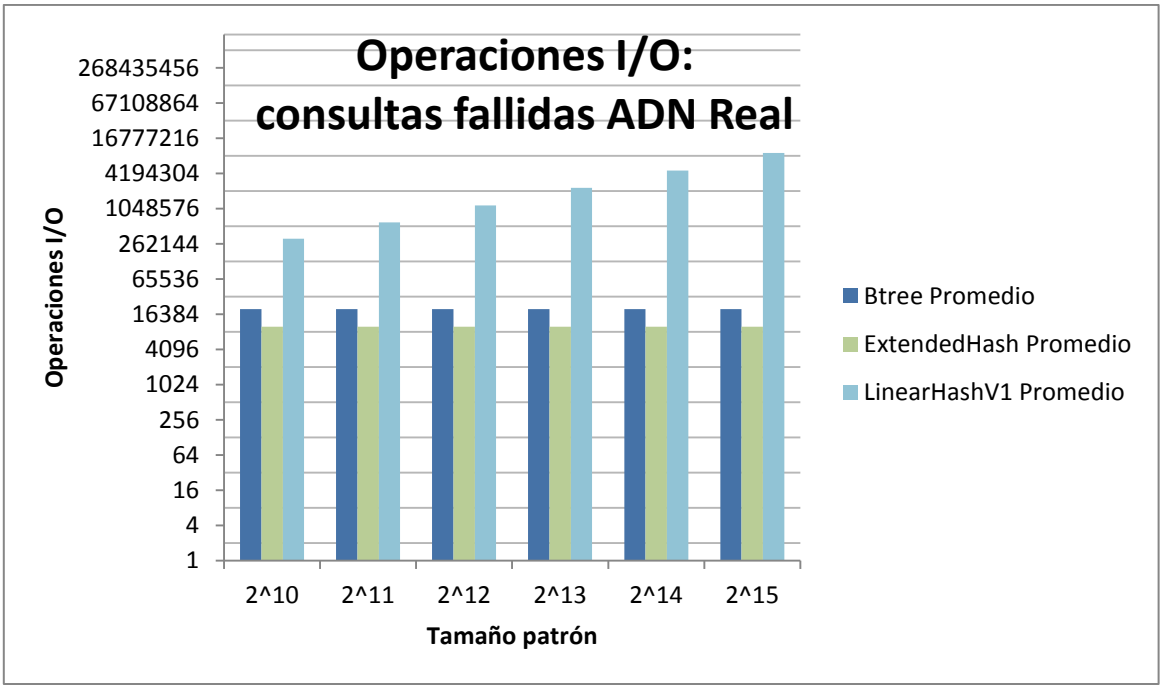


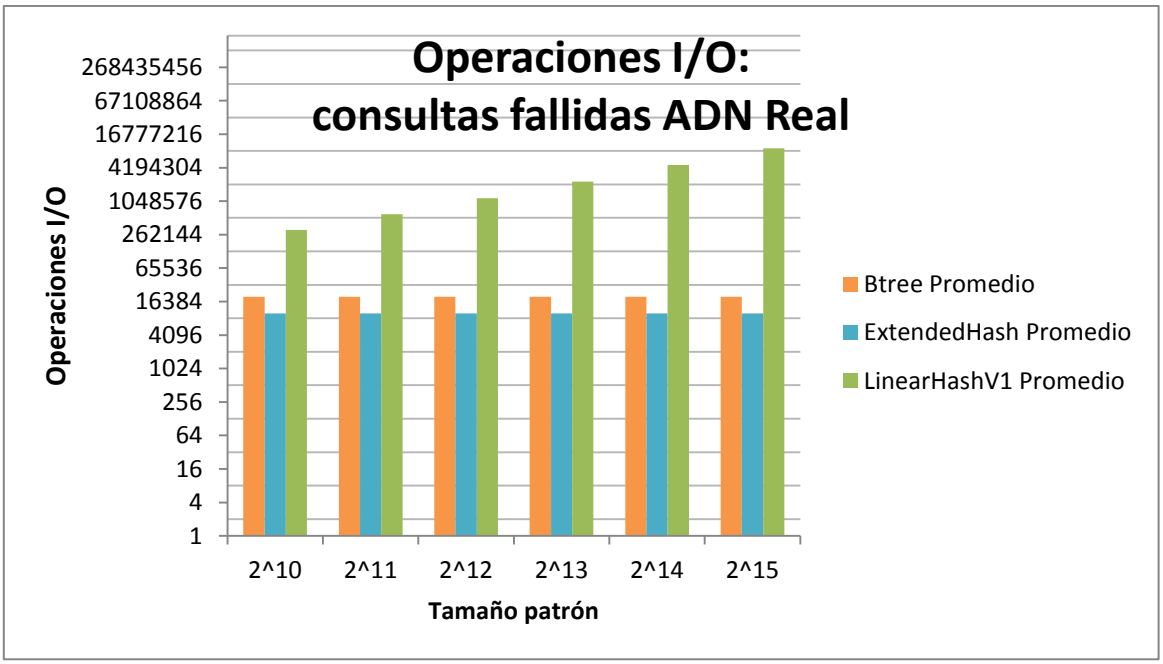


En estas pruebas, se nota que la función de Hash para Hash Lineal V1 presenta problemas, viéndose saturadas de consultas IO a medida que la cantidad de datos almacenados aumenta considerablemente, dando cuenta de la gran cantidad de colisiones generadas por la función de hash. Por su parte, BTree y Hash Extensible muestran un costo constante en cada una de las repeticiones según la cantidad de búsquedas (O(2) y O(1) por búsqueda), lo cual va de la mano a lo esperado.

En cuanto a los errores porcentuales, para BTree y Hash Extensible se mantiene estable en 0%, mientras que para Hash Lineal V1 el error porcentual se mantiene bajo el 4%, siendo aún apropiado para el experimento.

Operaciones I/O en búsquedas “infructuosas”:





Equivalentemente a lo anterior, las consultas infructuosas sobre BTree y Hash extensible cuestan la misma cantidad de accesos a disco, siendo evidente que estas consultas dependen exclusivamente de la altura de la estructura de árbol de ambas. Por su parte, en las búsquedas infructuosas de Hash Lineal V1, se evidencian ciertos casos en que, probablemente, haya calces entre las cadenas generadas al azar para testeo, con aquellas que efectivamente hayan sido guardadas anteriormente en las estructuras, para ADN sintético. Sin embargo, el resultado general también indica que los costos de las consultas dependen de la cantidad de elementos guardados en la estructura, dando cuenta de la gran cantidad de colisiones que genera Hash Lineal V1.

El error asociado a las primeras 2 estructuras vuelve a ser 0%, así como en Hash Lineal para ADN real. Sin embargo, para ADN sintético, hay algunas variaciones que llegan hasta el 0,02%.

Conclusiones

El objetivo de esta tarea es probar la eficiencia en accesos a disco de cada uno de los algoritmos utilizados. La principal meta era limitar la saturación de estas estructuras en su mínimo y máximo, controlando así el desperdicio de espacio (underflow) y la saturación (overflow, malo para las búsquedas). Aumentando la saturación, los algoritmos se asimilan a escribir directamente en disco.

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

Hash Extensible

Hashing extensible fue la estructura más eficiente con respecto a su número de operaciones IO frente a lecturas y escrituras. El problema con esta estructura fue que a partir de las 2^15 inserciones, de acuerdo a la implementación entregada, la estructura no tenía más bits por donde seguir construyendo un trie, por lo que hubo que recurrir a guardar las nuevas cadenas en páginas de overflow. En pruebas externas a esta tarea, a medida que hubo más inserciones, el porcentaje de ocupación mejoró bastante, alcanzando un 90% en 2^21 inserciones. Para los efectos de lo testeado en esta tarea, su funcionamiento denotó un acceso casi constante para las búsquedas exitosas e infructuosas

Hash Lineal V1

En esta variante del hash lineal, se tomó como métrica que la estructura se comprimió cuando la tasa de ocupación era menor al 60% y que se expandiría en el 80%. Estos parámetros fueron modelados viendo los resultados experimentales, considerando que las operaciones de compresión y expansión son las más caras dentro de esta estructura, se dio un rango amplio para realizarlas.

Sin embargo, los test indican que su tasa de ocupación estuvo rondando el 50%, contradiciendo lo anterior. Esto puede dar cuenta de problemas en nuestra implementación, los cuales sin embargo aún permitieron seguir con buena parte de las pruebas ejemplificadas en este informe.

Hash Lineal V2

En esta variante, la métrica de expansión y compresión se determinó como el tiempo de búsqueda. Esta variante resultó altamente ineficiente por la forma en que se probó la implementación. Ésta se probó realizando grandes sets de inserciones, búsquedas y eliminación de datos. Hash Lineal V2 solamente media el tiempo de búsqueda cuando se realizaba esa operación, mas no cuando se realizaba una inserción o un borrado, por lo que la estructura se desbalanceaba. Mientras se realizaba el set de inserciones y búsquedas, se balanceaba correctamente, y se volvía a desbalancear cuando se realizaban los borrados. A pesar de lo altamente ineficiente que resultó esta estructura, no es fácil determinar su real desempeño frente a una implementación real, en que las inserciones, borrados y búsquedas generalmente ocurren de forma intercalada.

BTree

En la implementación del BTree se alcanzaron altos porcentajes de ocupación. Los porcentajes obtenidos son alrededor de un 97%, considerablemente más alto que los mínimos garantizados. Esto llevó a una alta ineficiencia en los accesos a disco durante las inserciones, escapándose exponencialmente de sus competidores.

A pesar de su saturación, su desempeño en las búsquedas y eliminaciones es aceptable, sólo superado por el Hash Extensible.

Finalmente, queda remarcar el desempeño del Hash Extensible en las pruebas, mostrándose eficiente en cuanto a buena parte de los test ejecutados. 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

Porcentaje de ocupación en inserción de elementos:

		2^10	2^10	2^11	2^11	2^12	2^12	2^13	2^13	2^14	2^14	2^15	2^15
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	97,40%	97,07%	98,65%	98,67%	99,00%	98,63%	99,19%	99,03%	99,35%	99,23%	99,34%	99,33%
	Desviación Estándar	1,16%	0,00%	0,50%	0,00%	0,37%	0,01%	0,18%	0,01%	0,09%	0,01%	0,03%	0,00%
	Error Estándar	1,07%	0,00%	0,46%	0,00%	0,33%	0,01%	0,16%	0,01%	0,08%	0,00%	0,02%	0,00%
	95% de datos entre	99,72%	97,07%	99,65%	98,67%	99,74%	98,66%	99,54%	99,04%	99,52%	99,24%	99,40%	99,34%
	95% de datos entre	95,08%	97,07%	97,64%	98,67%	98,26%	98,60%	98,84%	99,01%	99,18%	99,22%	99,29%	99,33%
ExtendedHa	Promedio	69,33%	78,49%	68,09%	72,28%	68,31%	71,81%	67,30%	70,82%	67,69%	70,94%	66,92%	70,73%
	Desviación Estándar	2,68%	3,90%	2,96%	1,29%	1,18%	2,39%	0,58%	1,07%	0,86%	0,71%	0,74%	0,31%
	Error Estándar	3,46%	4,45%	3,89%	1,60%	1,54%	2,97%	0,77%	1,36%	1,14%	0,89%	0,99%	0,39%
	95% de datos entre	74,68%	86,30%	74,01%	74,87%	70,67%	76,58%	68,45%	72,97%	69,41%	72,35%	68,40%	71,35%
	95% de datos entre	63,97%	70,69%	62,17%	69,70%	65,96%	67,03%	66,15%	68,67%	65,96%	69,53%	65,43%	70,10%
LinearHashV	Promedio	49,70%	49,70%	50,57%	50,57%	51,04%	51,04%	51,06%	51,06%	51,07%	51,07%	51,03%	51,03%
	Desviación Estándar	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,01%
	Error Estándar	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,02%
	95% de datos entre	49,70%	49,70%	50,57%	50,57%	51,04%	51,04%	51,06%	51,06%	51,07%	51,07%	51,03%	51,06%
	95% de datos entre	49,70%	49,70%	50,57%	50,57%	51,04%	51,04%	51,06%	51,06%	51,07%	51,07%	51,03%	51,01%

Porcentaje de ocupación en eliminación de elementos:

		2^14	2^14	2^13	2^13	2^12	2^12	2^11	2^11	2^10	2^10
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	99,35%	99,23%	99,19%	99,03%	99,00%	98,63%	98,65%	98,67%	97,40%	97,07%
	Desviación Estánd	0,03%	0,00%	0,03%	0,00%	0,03%	0,00%	0,03%	0,00%	0,03%	0,00%
	Error Estándar	0,0248%	0,0031%	0,0248%	0,0031%	0,0248%	0,0031%	0,0248%	0,0031%	0,0248%	0,0031%
	95% de datos entr	99,40%	99,24%	99,25%	99,03%	99,05%	98,64%	98,70%	98,68%	97,46%	97,08%
	95% de datos entr	99,29%	99,23%	99,14%	99,02%	98,94%	98,62%	98,59%	98,66%	97,35%	97,06%
ExtendedHa	Promedio	66,91%	70,73%	66,91%	70,73%	66,91%	70,72%	66,91%	70,72%	66,91%	70,72%
	Desviación Estánd	0,74%	0,31%	0,74%	0,31%	0,74%	0,31%	0,74%	0,31%	0,74%	0,31%
	Error Estándar	0,99%	0,39%	0,99%	0,39%	0,99%	0,39%	0,99%	0,39%	0,99%	0,39%
	95% de datos entr	68,40%	71,35%	68,40%	71,35%	68,39%	71,34%	68,39%	71,34%	68,39%	71,34%
	95% de datos entr	65,43%	70,10%	65,43%	70,10%	65,43%	70,10%	65,43%	70,10%	65,43%	70,10%
LinearHashV	Promedio	51,03%	51,03%	51,03%	51,03%	51,03%	51,03%	51,03%	51,03%	51,03%	51,03%
	Desviación Estánd	0,00%	0,01%	0,00%	0,01%	0,00%	0,01%	0,00%	0,01%	0,00%	0,01%
	Error Estándar	0,00%	0,02%	0,00%	0,02%	0,00%	0,02%	0,00%	0,02%	0,00%	0,02%
	95% de datos entr	51,03%	51,06%	51,03%	51,06%	51,03%	51,06%	51,03%	51,06%	51,03%	51,06%
	95% de datos entr	51,03%	51,01%	51,03%	51,01%	51,03%	51,01%	51,03%	51,01%	51,03%	51,01%

Operaciones I/O en inserción de elementos:

		2^10	2^10	2^11	2^11	2^12	2^12	2^13	2^13	2^14	2^14	2^15	2^15
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	47339,8	47628	184150,6	185155	726330,2	729796,8	2863722,4	2897623,2	1,13E+07	1,15E+07	4,47E+07	4,61E+07
	Desviación Estánd	372,5032886	0	1123,11544	0	3932,80367	72,0291608	39412,7659	229,867788	213796,515	544,709831	1106282,69	2608,30295
	Error Estándar	0,7038%	0,0000%	0,5455%	0,0000%	0,4843%	0,0088%	1,2310%	0,0071%	1,6896%	0,0042%	2,2131%	0,0051%
	95% de datos entr	48084,80658	47628	186396,831	185155	734195,807	729940,858	2942547,93	2898082,94	11745573,2	11548510,8	46922581,2	46107451,2
	95% de datos entr	46594,79342	47628	181904,369	185155	718464,593	729652,742	2784896,87	2897163,46	10890387,2	11546332	42497450,4	46097018
ExtendedHash	Promedio	2135,4	2125,4	4317,2	4301,8	8676,6	8649,6	17405,6	17351,8	34849,4	34748,4	69772,4	69549,8
	Desviación Estánd	3,286335345	3,84707681	7,1902712	3,42052628	6,18869938	10,0149888	8,56154192	6,37965516	21,7094449	11,6103402	42,7001171	15,8177116
	Error Estándar	0,14%	0,16%	0,15%	0,07%	0,06%	0,10%	0,04%	0,03%	0,06%	0,03%	0,05%	0,02%
	95% de datos entr	2141,972671	2133,09415	4331,58054	4308,64105	8688,9774	8669,62998	17422,7231	17364,5593	34892,8189	34771,6207	69857,8002	69581,4354
	95% de datos entr	2128,827329	2117,70585	4302,81946	4294,95895	8664,2226	8629,57002	17388,4769	17339,0407	34805,9811	34725,1793	69686,9998	69518,1646
LinearHashV1	Promedio	3912,4	3913,8	10545,4	10547,6	31893	31894,6	107204,6	107207	388298	388302,6	1474264,2	1474272,6
	Desviación Estánd	0,547722558	0,4472136	0,89442719	0,89442719	0,70710678	1,51657509	2,19089023	0,70710678	2,34520788	2,50998008	7,7588659	5,6833089
	Error Estándar	0,013%	0,010%	0,008%	0,008%	0,002%	0,004%	0,002%	0,001%	0,001%	0,001%	0,000%	0,000%
	95% de datos entr	3913,495445	3914,69443	10547,1889	10549,3889	31894,4142	31897,6332	107208,982	107208,414	388302,69	388307,62	1474279,72	1474283,97
	95% de datos entr	3911,304555	3912,90557	10543,6111	10545,8111	31891,5858	31891,5668	107200,218	107205,586	388293,31	388297,58	1474248,68	1474261,23

Operaciones I/O en eliminación de elementos:

		2^14	2^14	2^13	2^13	2^12	2^12	2^11	2^11	2^10	2^10	0	0
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	50069	50090	75562	75604	88767	88830	95828	95912	99817	99922	102889	102994
	Desviación Estánda	16,7630546	0	33,5261092	0	50,2891638	0	67,0522185	0	83,8152731	0	83,8152731	0
	Error Estándar	0,02995%	0,00000%	0,03968%	0,00000%	0,05067%	0,00000%	0,06258%	0,00000%	0,07510%	0,00000%	0,07286%	0,00000%
	95% de datos entre	50102,5261	50090	75629,0522	75604	88867,5783	88830	95962,1044	95912	99984,6305	99922	103056,631	102994
	95% de datos entre	50035,4739	50090	75494,9478	75604	88666,4217	88830	95693,8956	95912	99649,3695	99922	102721,369	102994
ExtendedHa	Promedio	34199,6	34122,6	52015,2	51861,2	61638,8	61407,8	67166,4	66858,4	70646	70261	72694	72309
	Desviación Estánda	15,6620561	5,94138031	31,3241121	11,8827606	46,9861682	17,8241409	62,6482242	23,7655212	78,3102803	29,7069016	78,3102803	29,7069016
	Error Estándar	0,041%	0,016%	0,054%	0,020%	0,068%	0,026%	0,083%	0,032%	0,099%	0,038%	0,096%	0,037%
	95% de datos entre	34230,9241	34134,4828	52077,8482	51884,9655	61732,7723	61443,4483	67291,6964	66905,931	70802,6206	70320,4138	72850,6206	72368,4138
	95% de datos entre	34168,2759	34110,7172	51952,5518	51837,4345	61544,8277	61372,1517	67041,1036	66810,869	70489,3794	70201,5862	72537,3794	72249,5862
LinearHashV	Promedio	1879	1878,8	3758	3757,6	5637	5636,4	7516	7515,2	9395	9394	9395	9394
	Desviación Estánda	0	0,4472136	0	0,89442719	0	1,34164079	0	1,78885438	0	2,23606798	0	2,23606798
	Error Estándar	0,000%	0,021%	0,000%	0,021%	0,000%	0,021%	0,000%	0,021%	0,000%	0,021%	0,000%	0,021%
	95% de datos entre	1879	1879,69443	3758	3759,38885	5637	5639,08328	7516	7518,77771	9395	9398,47214	9395	9398,47214
	95% de datos entre	1879	1877,90557	3758	3755,81115	5637	5633,71672	7516	7511,62229	9395	9389,52786	9395	9389,52786

Operaciones I/O en búsquedas exitosas de elementos:

		2^10	2^10	2^11	2^11	2^12	2^12	2^13	2^13	2^14	2^14	2^15	2^15
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
	Desviación Estánd	0	0	0	0	0	0	0	0	0	0	0	0
	Error Estándar	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	95% de datos entre	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
	95% de datos entre	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
ExtendedHa	Promedio	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	Desviación Estánd	0	0	0	0	0	0	0	0	0	0	0	0
	Error Estándar	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	95% de datos entre	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	95% de datos entre	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
LinearHashV	Promedio	194348,2	196547,6	320895,4	329019	578215,2	591993	1087578	1126356,4	2117184,8	2199755,8	4125456,8	4295374,6
	Desviación Estánd	4492,12396	3506,18615	8561,82547	8056,11597	15947,7122	19929,6187	34429,2176	41147,4014	77343,7918	78603,1603	151172,537	188712,252
	Error Estándar	2,07%	1,60%	2,39%	2,19%	2,47%	3,01%	2,83%	3,27%	3,27%	3,20%	3,28%	3,93%
	95% de datos entre	203332,448	203559,972	338019,051	345131,232	610110,624	631852,237	1156436,44	1208651,2	2271872,38	2356962,12	4427801,87	4672799,1
	95% de datos entre	185363,952	189535,228	303771,749	312906,768	546319,776	552133,763	1018719,56	1044061,6	1962497,22	2042549,48	3823111,73	3917950,1

Operaciones I/O en búsquedas infructuosas de elementos:

		2^10	2^10	2^11	2^11	2^12	2^12	2^13	2^13	2^14	2^14	2^15	2^15
		RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA	RealDNA	FakeDNA
Btree	Promedio	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
	Desviación Estánda	0	0	0	0	0	0	0	0	0	0	0	0
	Error Estándar	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%
	95% de datos entre	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
	95% de datos entre	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000	20000
ExtendedHa	Promedio	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	Desviación Estánda	0	0	0	0	0	0	0	0	0	0	0	0
	Error Estándar	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%
	95% de datos entre	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
	95% de datos entre	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
LinearHashV	Promedio	320000	320000	610000	609988,8	1190000	1190000	2360000	2360000	4699920,2	4700000	9390000	9388850,6
	Desviación Estánda	0	0	0	25,0439613	0	0	0	0	178,438225	0	0	2162,60348
	Error Estándar	0,0000%	0,0000%	0,0000%	0,0037%	0,0000%	0,0000%	0,0000%	0,0000%	0,0034%	0,0000%	0,0000%	0,0206%
	95% de datos entre	320000	320000	610000	610038,888	1190000	1190000	2360000	2360000	4700277,08	4700000	9390000	9393175,81
	95% de datos entre	320000	320000	610000	609938,712	1190000	1190000	2360000	2360000	4699563,32	4700000	9390000	9384525,39

Estas tablas se encuentran presentes en el archivo adjunto “Resultados Tarea 2.xlsx”, para su revisión más cómoda y exhaustiva.