Boletín 2: Árboles de búsqueda

Jaime Ansorena Carrasco
2020401497

Profesor: José Fuentes Sepúlveda
Ayudante: Leonardo Enrique Lovera Emanuelli

7 de octubre de 2024

1. Resumen

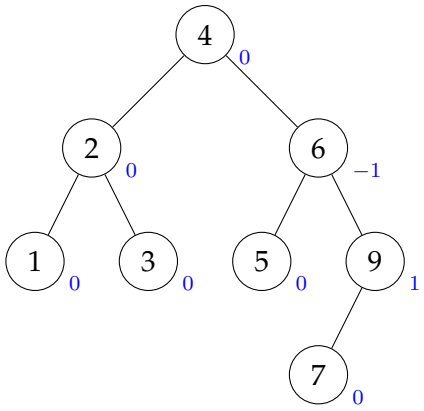
El problema de búsqueda eficiente de datos puede abordarse mediante diferentes estructuras de datos. En este boletín se analizaron cuatro estructuras de búsqueda en árboles: AVL Trees, Red-Black Trees, Splay Trees y Skip Lists. El objetivo principal fue comparar su rendimiento teórico y experimental, evaluando aspectos como balanceo, complejidad de operaciones y adaptabilidad en operaciones de inserción, eliminación y búsqueda. Se consideraron diversos tamaños de entrada para la parte experimental.

Los resultados experimentales muestran que el AVL Tree y el Red-Black Tree son ideales para tiempos de respuesta constantes y eficientes, con el AVL sobresaliendo en búsquedas y el Red-Black Tree en inserciones y eliminaciones. Para consultas frecuentes sobre un conjunto específico de elementos, el Splay Tree es la mejor opción gracias a su autoajuste. En cambio, el Skip List resulta menos eficiente en búsquedas, especialmente en conjuntos de datos grandes.

2. Algoritmos

AVL Tree

El árbol AVL fue desarrollado por Georgii Adelson-Velskii y Yevgeniy Landis, en 1962 en el articulo “An algorithm for the organization of information”. Este árbol fue el primer árbol binario de búsqueda auto-balanceado. Cada nodo se asegura de que la diferencia de altura entre sus subárboles izquierdo y derecho no sea mayor a uno. Esta característica garantiza que el árbol permanezca balanceado. Cada nodo puede almacenar un “factor de balanceo” que indica la diferencia de alturas entre sus subárboles izquierdo y derecho. Cuando se realiza una inserción o eliminación de un nodo que altera esta condición de equilibrio, se aplican rotaciones específicas para restaurar el balance. Estas rotaciones pueden ser simples o dobles.

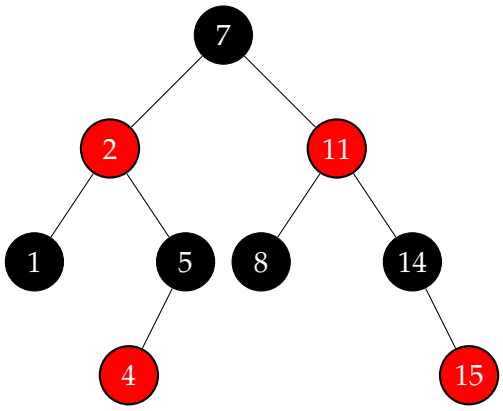


El número en cada nodo indica el factor de balanceo.

Red-Black Tree

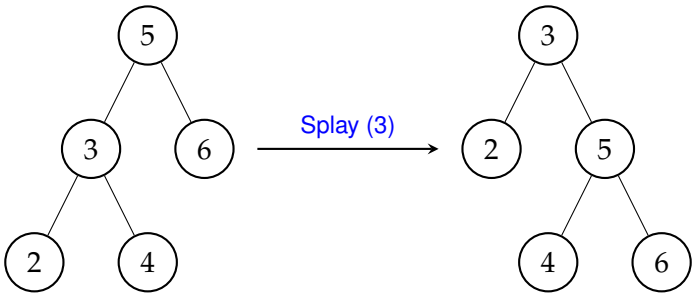
La estructura original fue creada por Rudolf Bayer en 1972, que le dio el nombre de "árboles-B binarios simétricos". Los nodos en este árbol contienen un bit adicional (color), que suele representarse como rojo y negro, y que ayuda a garantizar que el árbol se mantenga siempre "aproximadamente" equilibrado. Cuando el árbol se modifica, la nueva estructura se reorganiza y se "recolorea" para restaurar las propiedades de coloración. Estas propiedades son:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas son negras.
- Todo nodo rojo debe tener dos nodos hijos negros.
- Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.



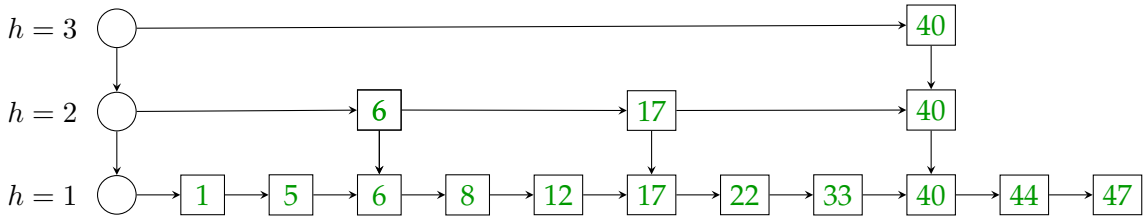
Splay Tree

Los Splay Trees, o árboles de rotación, fueron desarrollados por Daniel Sleator y Robert Tarjan en 1985. Estos árboles binarios de búsqueda se caracterizan por autoajustarse después de cada operación de acceso, moviendo el nodo consultado hacia la raíz a través de una serie de rotaciones. Este comportamiento se denomina splaying, y permite que los elementos más accedidos estén más cercanos a la raíz, mejorando el tiempo de acceso en búsquedas repetidas a los mismos elementos. Los Splay Trees son especialmente útiles en contextos donde ciertos elementos son accedidos con mucha frecuencia, como en caches y algoritmos de compresión de datos.



Skip List

Estructura de datos probabilística desarrollada por William Pugh en 1989, diseñada para ofrecer un rendimiento comparable al de los árboles balanceados. Una Skip List es una lista enlazada de varios niveles, donde cada nivel adicional permite "saltar" sobre una cantidad creciente de elementos. Esto se logra al duplicar probabilísticamente los nodos en varios niveles. Al tener múltiples niveles de enlaces, una Skip List puede simular un árbol de búsqueda balanceado, permitiendo una navegación eficiente a través de saltos en los niveles superiores y búsquedas secuenciales en los niveles inferiores.



3. Análisis Teórico

AVL Tree

- Inserción: $O(\log n)$. La inserción en un AVL Tree puede provocar desajustes en el factor de balanceo, que son corregidos mediante rotaciones simples o dobles. En el peor de los casos, se requiere un número constante de rotaciones, manteniendo el costo total de la inserción en $O(\log n)$, ya que el árbol permanece balanceado.
- Eliminación: $O(\log n)$. Similar a la inserción, la eliminación puede desbalancear el árbol y requerir rotaciones para restaurar el equilibrio. La operación se mantiene en $O(\log n)$ debido al equilibrio del árbol.

- Búsqueda: $O(\log n)$. Dado que el árbol está balanceado, la profundidad máxima del árbol es $O(\log n)$. Esto permite que la búsqueda se mantenga en tiempo logarítmico.

Red-Black Tree

- Inserción: $O(\log n)$. La inserción puede requerir recoloraciones y rotaciones para mantener las propiedades del árbol, pero no es tan estricta como en los AVL Trees. Aunque el balanceo no es perfecto, la profundidad del árbol se mantiene en $O(\log n)$.
- Eliminación: $O(\log n)$. Al igual que en la inserción, la eliminación puede requerir recoloraciones y rotaciones para mantener el equilibrio.
- Búsqueda: $O(\log n)$. La profundidad del árbol también es $O(\log n)$, ya que el equilibrio aproximado garantiza que el camino más largo sea a lo sumo el doble del camino más corto $(2\log(n+1))$.

Splay Tree

- Inserción: Amortizado $O(\log n)$. El tiempo amortizado de muchas inserciones es $O(\log n)$, ya que las rotaciones posteriores reorganizan el árbol para optimizar futuros accesos. Peor caso es $O(n)$, si el árbol se convierte en una lista enlazada.
- Eliminación: Amortizado $O(\log n)$. Similar a la inserción, una eliminación puede ser costosa si el árbol está altamente desbalanceado. Sin embargo, tras el proceso de splaying, el árbol se autoajusta, manteniendo un tiempo amortizado de $O(\log n)$ para múltiples eliminaciones.
- Búsqueda: Amortizado $O(\log n)$. La búsqueda también se beneficia del tiempo amortizado, ya que los elementos más accedidos tienden a estar más cerca de la raíz. Al igual que las otras operaciones, una búsqueda individual puede ser $O(n)$ si el árbol se convierte temporalmente en una lista.

Skip List

- Inserción: $O(\log n)$ en promedio, $O(n)$ en el peor caso. Generalmente, las inserciones son eficientes debido a los múltiples niveles que permiten saltos, lo que facilita el acceso rápido. Sin embargo, en el peor de los casos, si no se puede aprovechar esta estructura multinivel, el costo puede ser $O(n)$, ya que se debe recorrer la lista de manera secuencial.
- Eliminación: $O(\log n)$ promedio, O(n) peor caso. Al igual que la inserción, es $O(\log n)$ en promedio.
- Búsqueda: $O(\log n)$ promedio, $O(n)$ peor caso. En promedio, la búsqueda es eficiente gracias a los niveles que permiten saltar rangos de la lista. Sin embargo, en el peor caso, si todos los nodos están en un solo nivel, la búsqueda se degrada a $O(n)$.

Resumen

Estructura	Inserción	Eliminación	Búsqueda
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Splay Tree	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)^*$
Skip List	$O(\log n)^{**}$	$O(\log n)^{**}$	$O(\log n)^{**}$

** Amortizado*
*** Promedio, $O(n)$ peor caso*

Para las complejidades espaciales, todos los algoritmos presentan $O(n)$, con excepción de la Skip List, que en el peor caso puede alcanzar $O(n \log n)$. Esto se debe a que la Skip List utiliza n elementos en el nivel base, y en el peor de los casos, los elementos se duplican en niveles superiores, resultando en aproximadamente $O(\log n)$ niveles adicionales, cada uno con un subconjunto de los elementos, lo que lleva a la complejidad $O(n \log n)$.

Implementaciones

Las implementaciones de los distintos arboles de búsqueda se obtuvieron de los siguientes repositorios:

- AVL Tree
- Splay Tree
- Skip List
- Red-Black Tree (`std::set`)

4. Resultados experimentales

Inicialización

Se realizaron distintas inicializaciones de acorde a la operación y a la estructura a usar. A continuación se detallan:

insert

Se generó un número aleatorio mediante una distribución uniforme y luego se realizaron n inserciones a cada árbol.

```
for(std::int64_t j = 0; j < n; j++){
    std::int64_t value = u_distr(rng);
    //avl_tree.insert(value);
    //rb_tree.insert(value);
    //sp_tree.insert(value);
    //sl.insert(value);
}
```

delete

Para la operación delete primeramente se pobló el árbol con n elementos. Luego se realizaron n borrados del árbol de una clave aleatoria (distribución uniforme).

```
for(std::int64_t j = 0; j < n; j++){
    std::int64_t value = u_distr(rng);
    //avl_tree.erase(value);
    //rb_tree.erase(value);
    //sp_tree.Delete(value);
    //sl.erase(value);
}
```

search

Para los arboles AVL, Red-Black Tree y Skip List, se realizo un procedimiento similar. Se pobló el árbol con n elementos y luego se realizaron n búsquedas de claves aleatorias.

Para el Splay Tree, se consideraron algunas diferencias debido a sus características. Se llevó a cabo un conjunto de búsquedas con claves que se repiten frecuentemente, otras que se repiten con menor frecuencia y una clave con menor frecuencia que se repite muchas veces. Las claves fueron generadas siguiendo distribuciones de probabilidad Poisson, con un n = 0.5.

```
std::map<std::int64_t, std::int64_t> key_frequency;
std::poisson_distribution<std::int64_t> poisson_distr(0.5);

for (i = 0; i < runs; i++) {
    for (std::int64_t j = 0; j < n; j++) {
        std::int64_t value = poisson_distr(rng);
        sp_tree.find(value); // Search
        key_frequency[value]++; // Frequency
    }
}

// pair vector
std::vector<std::pair<std::int64_t, std::int64_t>>
    freq_vector(key_frequency.begin(), key_frequency.end());

// sort
std::sort(freq_vector.begin(), freq_vector.end(),
    [](const std::pair<std::int64_t, std::int64_t>& a,
      const std::pair<std::int64_t, std::int64_t>& b) {
        //return a.second > b.second; // most frequent first
        return a.second < b.second; // least frequent first
    });

// frecuent keys
//std::vector<std::int64_t> most_frequent_keys;
std::vector<std::int64_t> least_frequent_keys;
std::size_t top_n = n / 2; // top n most frequent keys
```

```
for (std::size_t i = 0; i < top_n && i < freq_vector.size(); ++i) {
    //most_frequent_keys.push_back(freq_vector[i].first);
    least_frequent_keys.push_back(freq_vector[i].first);
}
```

Estos datos fueron generados internamente y no se utilizaron datasets externos.

Se realizaron pruebas sobre cada algoritmo ejecutándolo en 100 repeticiones para cada tamaño de entrada. Se midió tanto el tiempo promedio por operación como la desviación estándar, con el fin de obtener resultados estadísticamente significativos y mitigar el efecto de fluctuaciones de rendimiento aleatorias.

Especificaciones

Los experimentos se realizaron en un equipo de escritorio con CPU i9-9900k que posee 8 núcleos a 3.6 GHz y 32 GB DDR4 a 3600 MHz. Dado que los algoritmos no involucran memoria secundaria, todas las operaciones se realizaron en memoria principal (RAM). El sistema operativo es Linux Mint 22 x86_64 con el kernel de linux 6.8.0-45

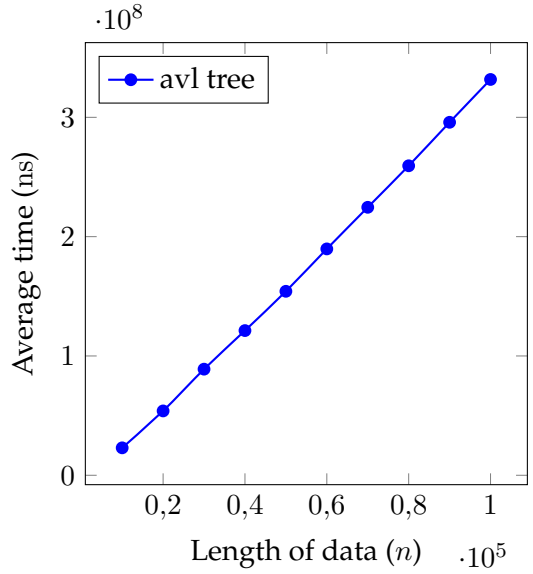
Compilación

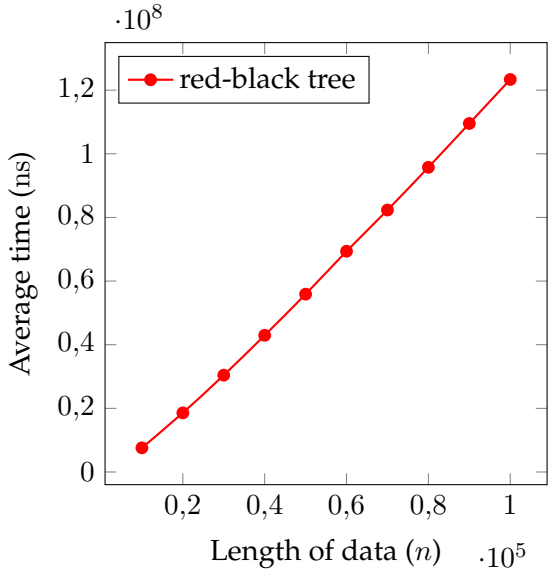
Para automatizar el análisis experimental se utilizó la herramienta `uhr`, disponible en el repositorio del curso. Esta se compiló usando el compilador GNU G++ como:

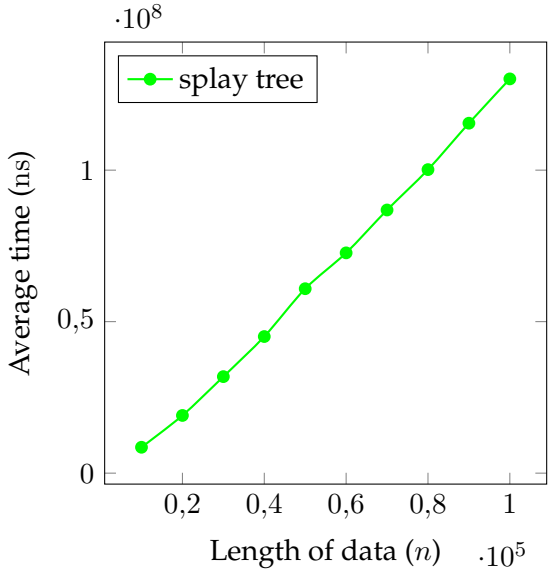
```
g++ -O0 -std=c++11 -Wall -Wpedantic -o <tree>_<op> uhr_<op>.cpp
```

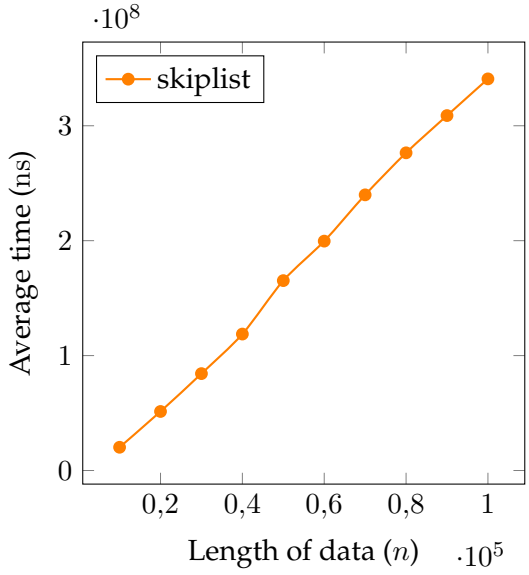
donde `<tree>` corresponde al árbol utilizado y `<op>` a la operación. Cada operación generó distintos ejecutables que luego fueron automatizados mediante un script de bash. El detalle se encuentra en el archivo `run_experiments_<op>.sh`. Para todos los experimentos se utilizaron valores de `LOWER=10000, UPPER=100000 y STEP=10000`

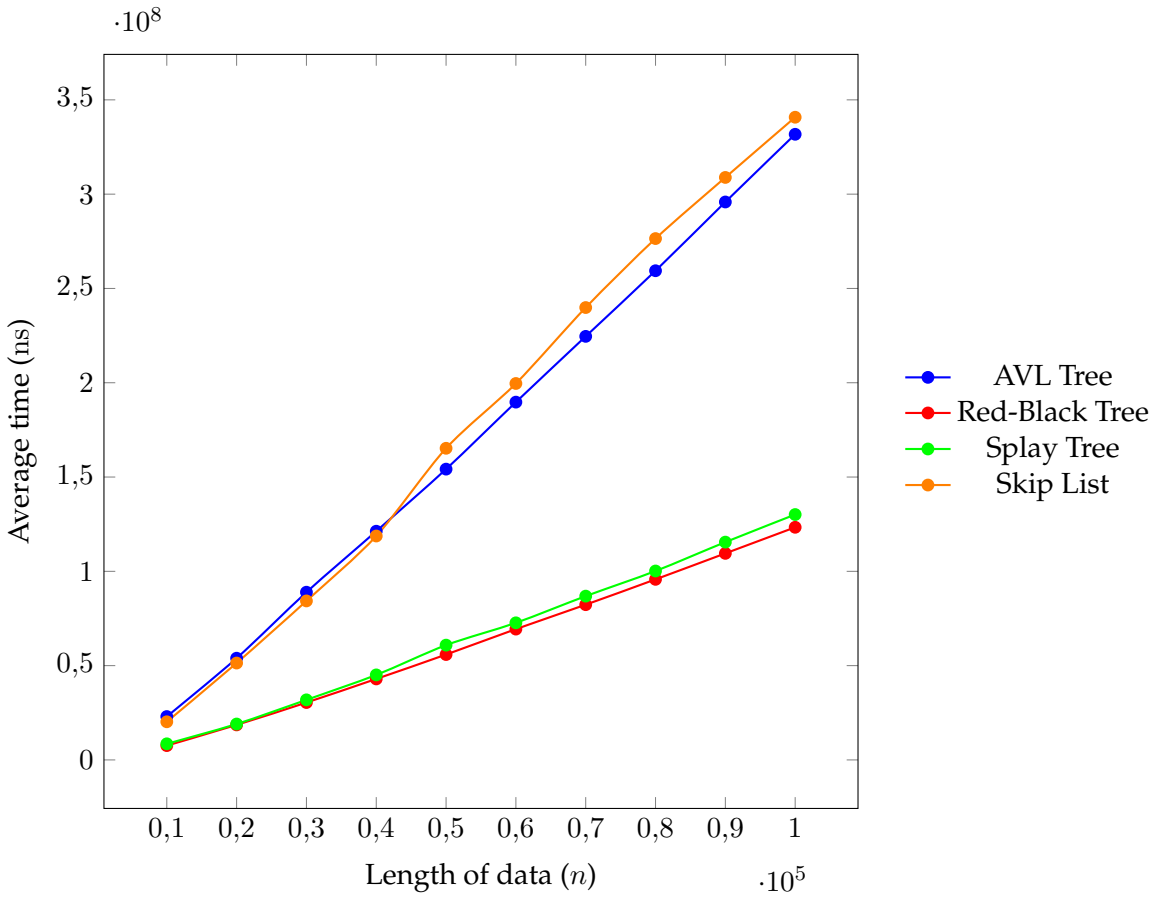
5. Resultados insert





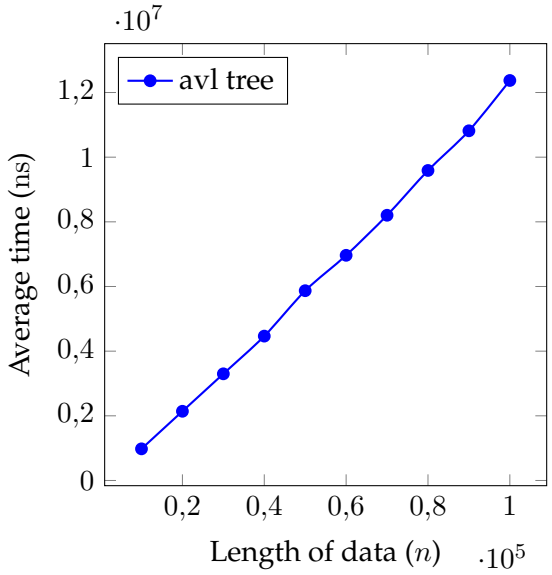


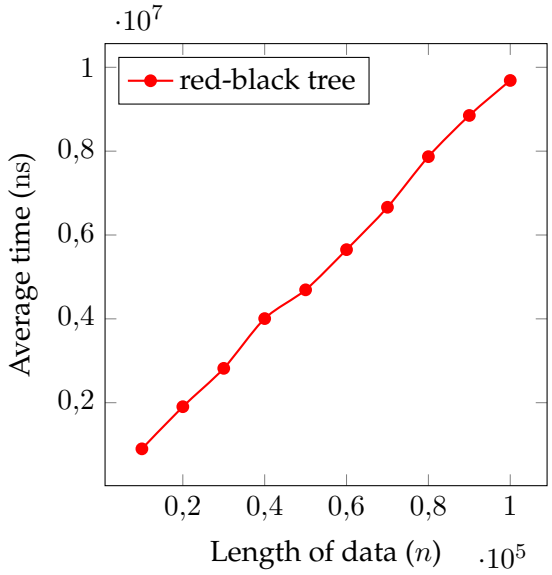


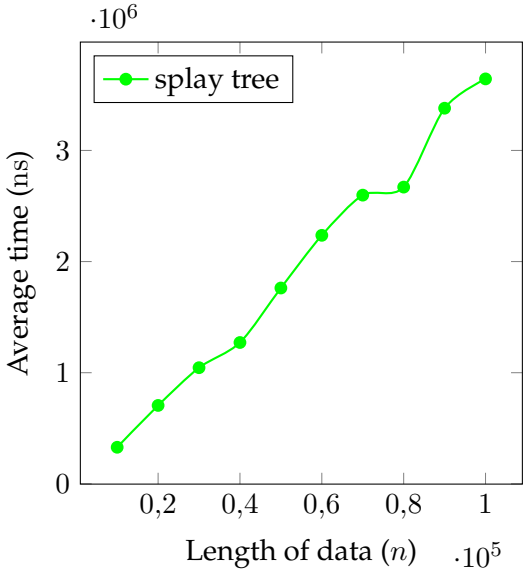


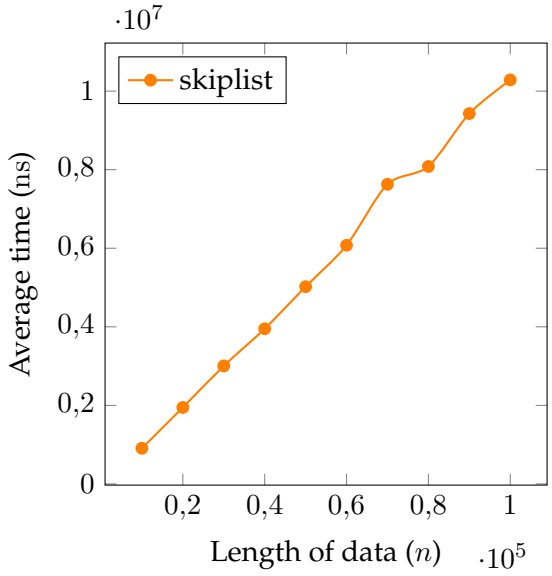
Los resultados son consistentes con el análisis teórico. Se observan dos grupos en el gráfico. Los que presentan mayores tiempos promedios corresponden al AVL y a la Skip List. En el AVL a pesar de que las operaciones son $O(\log n)$ los costos adicionales de rotaciones para mantener el equilibrio pueden hacer que el tiempo aumente. En el caso de la Skip List, como es una estructura probabilista, el peor caso corresponde a $O(n)$, aunque el tiempo promedio observado sigue siendo $O(\log n)$. Por otro lado, el Red-Black Tree y el Splay Tree muestran tiempos de inserción más bajos en promedio. El Red-Black Tree logra mantener un equilibrio relativamente bueno sin la necesidad de tantas rotaciones como el AVL, lo que se refleja en un mejor rendimiento. En el caso del Splay Tree, su mecanismo de autoajuste permite que las claves frecuentemente accedidas se mantengan en posiciones cercanas a la raíz, lo cual favorece tiempos más rápidos en inserciones sucesivas.

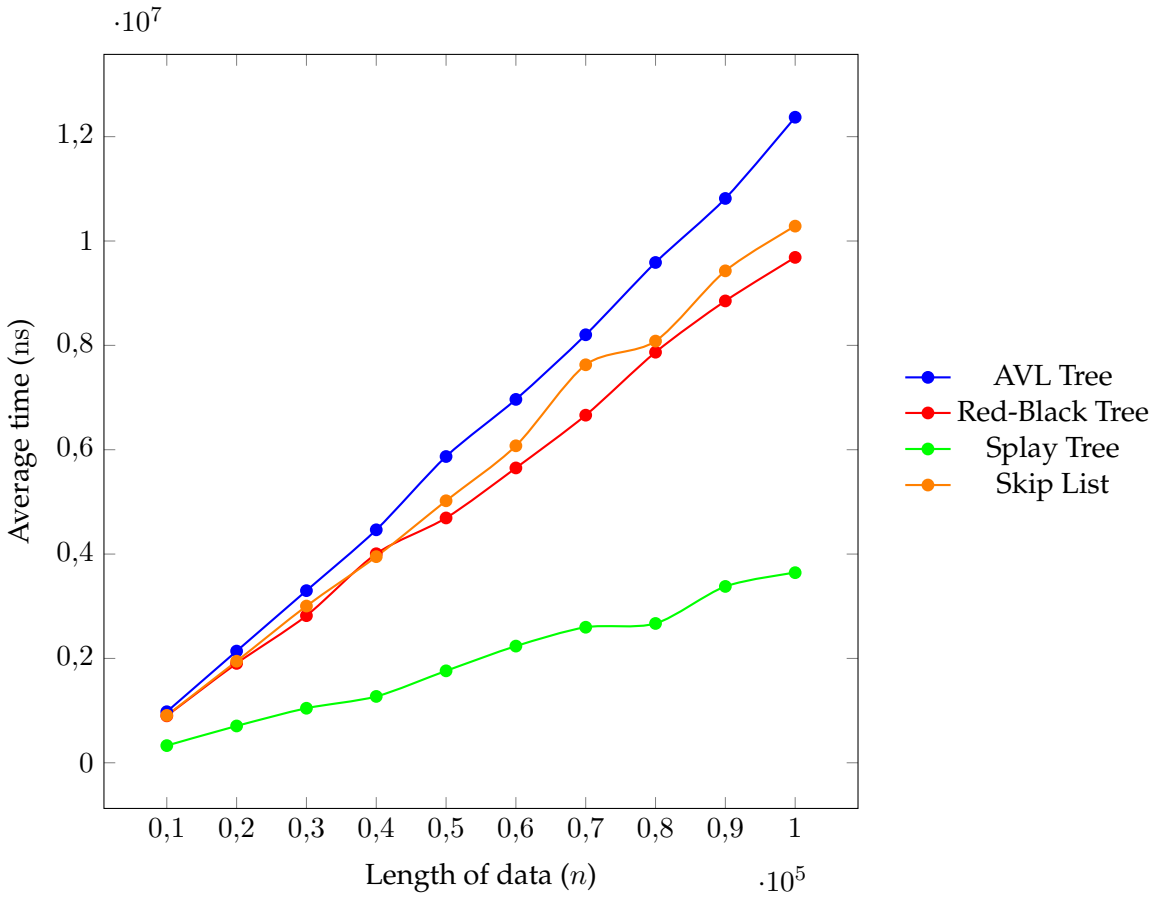
Resultados delete







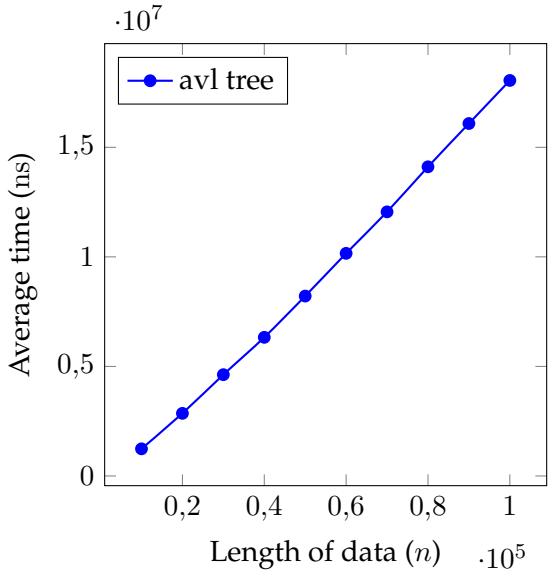


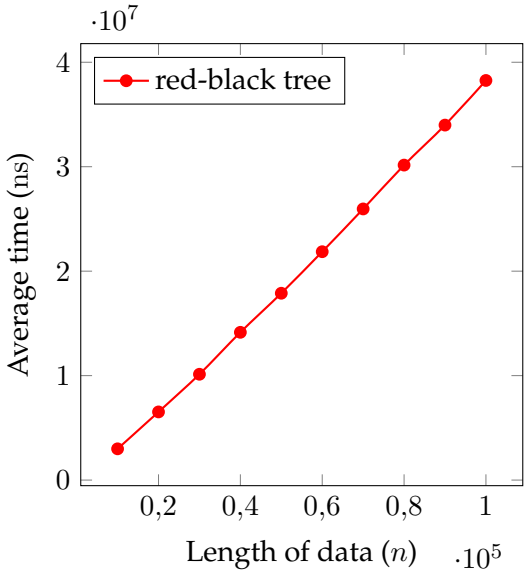


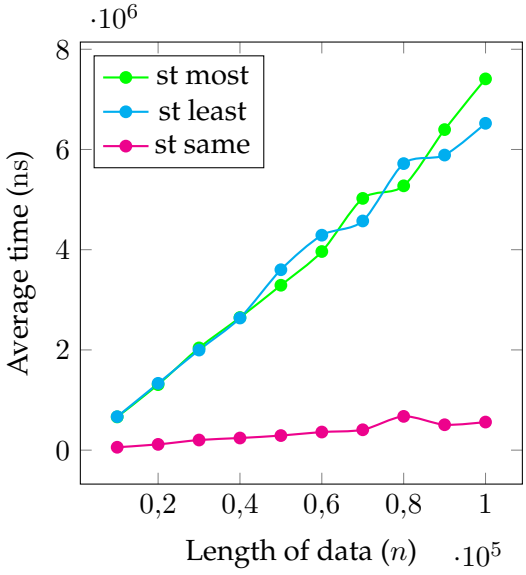
El AVL Tree, Red-Black Tree y Skip List presentan tiempos promedios más altos. En el caso del AVL, al igual que en la inserción, aunque la operación es $O(\log n)$, los costos adicionales de rotaciones para mantener el equilibrio pueden incrementar el tiempo de eliminación. La Skip List, al ser una estructura probabilística, tiene un peor caso de $O(n)$, aunque en promedio se comporta como $O(\log n)$. El Red-Black Tree, con su criterio de equilibrio menos estricto, se comporta de manera similar al AVL pero con menos rotaciones, lo que contribuye a un rendimiento ligeramente mejor.

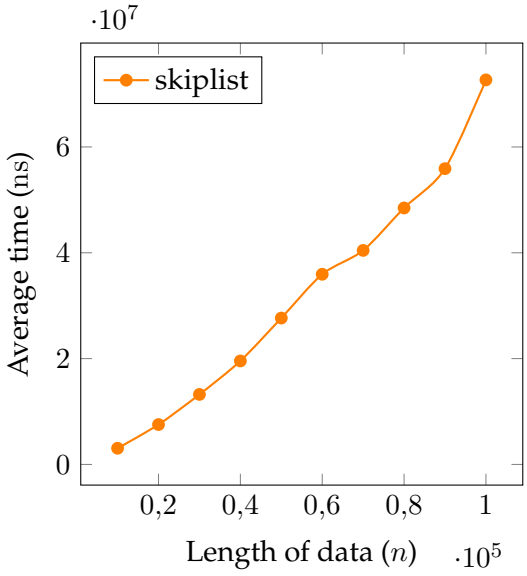
Por otro lado, el Splay Tree muestra un tiempo promedio significativamente más bajo, separándose de los otros tres. Gracias a su autoajuste, las claves accedidas con frecuencia tienden a acercarse a la raíz, lo que permite menores tiempos de eliminación, especialmente cuando se realizan eliminaciones repetidas desde posiciones cercanas a la raíz.

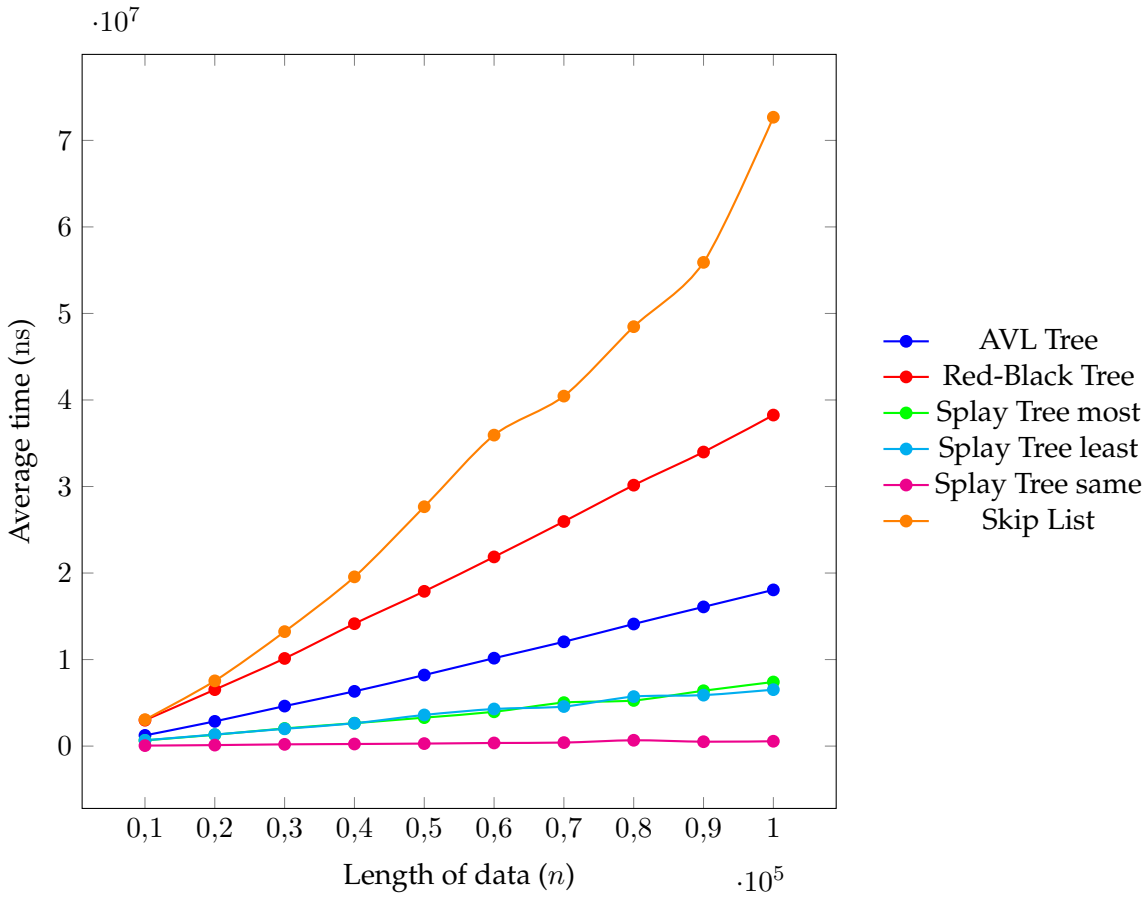
Resultados search











El árbol Skip List tiene tiempos de búsqueda significativamente más altos, destacándose por encima de las otras estructuras a medida que crece el tamaño de los datos. Esto implica que se ve afectado en el peor caso y se aproxima a $O(n)$, reflejando su naturaleza probabilística. Por otro lado, el Red-Black Tree también tiene un tiempo de búsqueda más alto, aunque menor que el Skip List, debido a que al no tener una estructura tan rígida, el camino más largo es a lo sumo el doble del camino más corto $(2\log(n+1))$. El AVL Tree mantiene un rendimiento relativamente eficiente, menor al Red-Black Tree, ya que tiene garantizada una altura de $O(\log n)$, lo cual permite mantener los tiempos de búsqueda bajo control con una mayor regularidad.

En el caso del Splay Tree, se observa que cuando se buscan las claves más frecuentes (Splay Tree most), el rendimiento mejora, ya que su autoajuste trae las claves más buscadas hacia la raíz, facilitando accesos rápidos. La búsqueda de claves menos frecuentes (Splay Tree least) no muestran una diferencia significativa. La búsqueda repetida de la misma clave (Splay Tree same) destaca con un tiempo constante y bajo, ya que el autoajuste minimiza el tiempo necesario al mantener la clave cerca de la raíz.

6. Conclusiones

Los resultados experimentales muestran que el Skip List es la menos eficiente en términos de búsqueda en comparación con las otras estructuras. Para entradas muy grandes, el tiempo promedio de búsqueda en el Skip List es considerablemente mayor, ya que en el peor de los casos su complejidad es $O(n)$. En cambio, el Red-Black Tree y el AVL Tree mantienen un rendimiento mucho más estable, con tiempos de búsqueda notablemente más bajos, ya que ambos garantizan alturas acotadas por $O(\log n)$. El AVL Tree, al mantener una altura más estrictamente balanceada, ofrece un mejor rendimiento en búsquedas. Sin embargo, para las operaciones de inserción y eliminación, el Red-Black Tree suele ser más eficiente, ya que su estructura menos rígida requiere menos rotaciones para mantenerse equilibrado. En estas operaciones, la Skip List también presenta mejores tiempos que el AVL, aunque la diferencia no es significativa.

Por otro lado, el Splay Tree se comporta de manera variable en función del patrón de acceso a las claves. Cuando se busca repetidamente la misma clave o se consultan claves con mucha frecuencia, el Splay Tree demuestra ser significativamente más rápido, llegando a ser hasta cuatro veces más eficiente que otras estructuras en estos casos. Esto se debe a su capacidad de autoajuste, que acerca las claves frecuentemente accedidas a la raíz, optimizando así el tiempo de búsqueda. Si se anticipa que ciertos elementos serán consultados con frecuencia, el Splay Tree es la opción más rápida debido a su habilidad de autoajuste. Sin embargo, para aplicaciones en las que la búsqueda se distribuye de manera uniforme entre los datos, el AVL Tree proporciona una respuesta más consistente y predecible.

En resumen, para aplicaciones que requieren tiempos de respuesta constantes y eficientes en las distintas operaciones, el AVL Tree y el Red-Black Tree son las opciones más sólidas. El AVL Tree destaca en operaciones de búsqueda, gracias a su estricta acotación en altura, mientras que el Red-Black Tree ofrece un mejor rendimiento en inserciones y eliminaciones, debido a su menor número de rotaciones. No obstante, si la estructura de datos debe manejar consultas frecuentes sobre un conjunto específico de elementos, el Splay Tree se convierte en la opción ideal, aprovechando su capacidad de autoajuste para optimizar el acceso repetido a dichas claves.

7. Referencias

1. Implementaciones de algoritmos
2. Apuntes Clase 2, Trees y heaps, Estructuras de Datos y Algoritmos Avanzados, Profesor José Fuentes, Universidad de Concepción, 2024.