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

Splay Trees y ABB's
Tarea 2 - Diseño y análisis de algoritmos

Integrantes: Francisco Almeida.
Patricio Espinoza A.
Scarlett Plaza.

Profesor: Gonzalo Navarro

Auxiliar: Máximo Flores V.
Sergio Rojas H.

Fecha de entrega: 10 de Noviembre de 2024

Índice

1. Introducción

Los ABB's son una estructura de datos ampliamente usada para las aplicaciones que requieren de información ordenada como las bases de datos dada su eficiencia para realizar operaciones clave como la inserción y búsqueda de elementos. En esta oportunidad se implementará un ABB clásico el cual tiene un costo de peor caso de $O(n)$, este árbol se define tal que la raíz corresponde al mayor valor de todos los elementos de su subárbol izquierdo y al menor valor de todos elementos de su subárbol derecho y todos sus subárboles cumplen con la condición de también ser un ABB. En esta estructura la búsqueda de un elemento *x* comienza desde la raíz que contiene un elemento *r*, si esta corresponde al elemento buscado se termina con éxito, en caso contrario se evalúa si *r* es mayor o menor que *x* ($x < r \vee x > r$), y se realiza el mismo procedimiento en el subárbol izquierdo o derecho, respectivamente. La inserción cumple con una dinámica muy similar pues también se compara el elemento a insertar *x* con la raíz, si la raíz es vacía se crea un árbol de un solo nodo con *x* y de ser mayor o menor se realiza el procedimiento de manera recursiva equivalentemente a la forma en que se hace en la búsqueda.

Por otra parte se tienen los Splay Trees, los cuales son un tipo de ABB con la modificación de que las operaciones de lectura, búsqueda e inserción cambian el árbol, esto hace que sus respuestas se mantengan al mismo tiempo que las operaciones se vuelvan más eficientes gracias a estas modificaciones, alcanzando un costo amortizado de $O(\log n)$ sin necesidad de almacenar información de balanceo e inclusive en el caso de nodos con distintas probabilidades llega a un costo amortizado de $O(H)$ con *H* la entropía de estas probabilidades. En este árbol las operaciones de búsqueda e inserción son tales que al nodo que se desea acceder debe quedar siempre en la raíz del árbol, para lo cual se define una función *splay(x)* la cual es la encargada de llevar el nodo accedido *x* a la raíz mediante rotaciones, considerando la notación $z(A, B)$, que corresponde al árbol con elemento de la raíz *z* y con subárbol izquierdo *A* y derecho *B*, *splay(x)* tiene las siguientes operaciones posibles que puede combinar en 0 o más rotaciones dobles consecutivas y puede finalizar o no en una rotación simple.

- Rotaciones Simples:
  - **Zig:** $y(x(A, B), C) \rightarrow x(A, y(B, C))$
  - **Zag:** $y(A, x(B, C)) \rightarrow x(y(A, B), C)$
- Rotaciones dobles:
  - **Zig-zig:** $z(y(x(A, B), C), D) \rightarrow x(A, y(B, z(C, D)))$
  - **Zig-zag:** $z(y(A, x(B, C)), D) \rightarrow x(y(A, B), z(C, D))$
  - **Zag-zig:** $z(A, y(x(B, C), D)) \rightarrow x(z(A, B), y(C, D))$
  - **Zag-Zag:** $z(A, y(B, x(C, D))) \rightarrow x(y(z(A, B), C), D)$

Este trabajo tiene como objetivo principal implementar ambas estructuras y comparar su rendimiento del costo promedio de búsqueda frente a los siguientes experimentos con $N \in 10^6 * \{0.1, 0.2, ..., 1.0\}$ y $M = 100 * N$:

1. Insertar *N* enteros distintos de manera aleatoria y hacer *M* búsquedas de valores escogidos aleatoriamente entre los insertados.
2. Insertar *N* enteros distintos de manera aleatoria pero al hacer la búsqueda de *M* elementos se asignarán probabilidades sesgadas de búsqueda de elementos utilizando una función del tipo $f(i) = \frac{C}{(i+2)^2}$.
3. Insertar *N* elementos distintos de manera ordenada y se hacen *M* búsquedas de valores escogidos aleatoriamente entre los insertados.
4. Antes de insertar los *N* elementos se realiza una copia de estos a otro arreglo que se ordenará y se utilizará para realizar las inserciones, luego se hará la búsqueda de *M* elementos asignando probabilidades sesgadas de búsqueda a los del arreglo original utilizando una función del mismo tipo que en el segundo experimento.

Dentro del informe se presentará la forma en que se implementaron estas estructuras, se mostrarán y analizarán los resultados de los experimentos antes nombrados y se compararán estos resultados con las hipótesis iniciales, la cual se basa en que en que la ventaja del uso de Splay Tree es que el árbol se va modificando según las búsquedas que se hagan, por lo cual al hacerse búsquedas con probabilidad sesgada, este tendrá un mejor rendimiento que el ABB clásico, pero al buscar elementos de manera aleatoria, el ABB clásico tendrá mejor rendimiento pues no hará las operaciones relacionadas a reacomodar el árbol, lo que le dará una ventaja en el tiempo de búsqueda; es por esto que para cada experimento se tienen las siguientes hipótesis:

- Experimento 1: El ABB clásico tendrá mejor rendimiento que el Splay Tree ya que la cantidad de consultas es uniforme y aleatoria, quitando el beneficio de este árbol en cuanto a las consultas más frecuentes, dejándolo con el coste extra de reacomodación de los nodos.
- Experimento 2: El Splay Tree tendrá mejor rendimiento. Esto basado en que la cantidad de búsquedas estará sesgada por la función *f(i)*, provocando que algunos valores se busquen más que otros y por consiguiente el tiempo de búsqueda sea menor y mejor que el del ABB Tree.
- Experimento 3: En este caso al insertarse los elementos de forma ordenada el ABB clásico estará muy desbalanceado teniendo solo nodos derechos, en cambio el Splay Tree se irá modificando con cada búsqueda, por lo cual se espera que este último tenga mejor costo promedio de búsqueda.
- Experimento 4: Por la misma razón del experimento anterior y además tomando en cuenta que se están utilizando probabilidades sesgadas a la última se espera que el Splay Tree tenga un costo promedio de búsqueda menor.

2. Desarrollo

Para la implementación del trabajo se utilizará Java debido a que corresponde a un lenguaje compilado al igual que C o C++ pero donde no es necesario hacer manejo de memoria y recolección de basura, evitando así los errores que esto podría generar. Al ser Java un lenguaje orientado a objetos para la implementación del problema planteado se construyen las clases enunciadas a continuación.

2.1. Node

En esta clase se define la estructura que describe el nodo de un árbol binario, cada nodo se define utilizando un entero key, que corresponde al elemento que se guarda en ese nodo y dos Node, uno izquierdo y uno derecho que corresponden a los hijos de este nodo. Además se define un constructor Node(int item) que crea un nodo con key = item e inicializa los hijos con valor null, lo que indica que inicialmente un nodo no tiene hijos.

2.2. BinarySearchTree

Esta clase es la encargada de representar un ABB clásico, que comienza solo con un Node root = null, es decir define un ABB que no contiene nodos y se define un método constructor que no realiza ninguna acción para que al inicializar un árbol este corresponda a uno vacío. Luego para realizar las operaciones de inserción y búsqueda se definen los siguientes métodos:

- **void insert(int var1)**: Este método es el encargado de insertar el elemento var1 al ABB, para esto se llama a la función auxiliar insertRec pasándole como argumentos la raíz del árbol y la variable.
- **Node insertRec(Node var1, int var2)**: Corresponde a un método auxiliar recursivo que se encarga de insertar el elemento var2 en la posición correcta del árbol, para esto se le entrega esta variable ya mencionada y var1, que corresponde al nodo actual de la recursión. Dados estos valores se evalúa si el nodo actual es nulo, en caso de serlo significa que se llegó a un lugar vacío donde el elemento puede ser insertado, por lo que se define a var1 como un nuevo nodo con key el var2 y se retorna. En caso de que el nodo actual no sea nulo, se evalúa si la variable var2 es menor o mayor al elemento que se encuentra en el nodo actual, si es menor significa que debe insertarse en el subárbol izquierdo, por lo cual se llama recursivamente a insertRec con este, en caso contrario se hace el llamado recursivo con el subárbol derecho, finalmente se retorna el nodo actual preservando la estructura del árbol.
- **boolean search(int var1)**: Este método se encarga de buscar en un ABB el elemento var1, retornando true si fue encontrado y false si no. Para hacer esto se llama a una función auxiliar searchRec con la raíz del árbol y el elemento a buscar.
- **Node searchRec(Node var1, int var2)**: Este es un método auxiliar que se encarga de la búsqueda en el árbol binario de manera recursiva, para hacerlo recibe var1 y var2, que corresponden al nodo actual en la recursión y el elemento a buscar, respectivamente. Dados estos evalúa si la var1 no es un nodo nulo y si su key es igual o no a var2, de cumplirse esta condición significa que se ha encontrado el elemento a buscar, por lo que se retorna el nodo var1, de caso contrario se verifica si var2 es menor al key de var1, de serlo se busca recursivamente en el árbol izquierdo y de no serlo se hace la recursión en el árbol derecho y se retorna el resultado de esta acción.

2.3. SplayTree

Esta clase se encarga de describir un ABB Splay Tree, el cual luego de cada operación de búsqueda o inserción lleva el nodo accedido a la raíz del árbol para mejorar la eficiencia de futuros accesos. Al igual que el ABB clásico este comienza como un nodo raíz nulo. En este caso se definen los siguientes métodos para hacer las inserciones, búsqueda de elementos y la rotación característica de estos árboles:

- **Node rightRotate(Node x)**: Esta función se encarga de generar una rotación hacia la derecha en el árbol que tiene como raíz el nodo x, esta se utilizará para las operaciones que incluyan a Zig, para esto recibe el nodo x, asigna el hijo izquierdo de este a el nodo y, reasigna el nodo izquierdo de x como el nodo derecho de y, y luego asigna como nodo derecho de y al nodo x para finalmente retornar el nodo y.

- **Node leftRotate(Node x)**: Equivalentemente al método anterior este genera una rotación, solo que esta vez hacia la izquierda, y se utilizará en las operaciones que incluyan a Zag, para esto recibe el nodo *x*, asigna el hijo derecho de este al nodo *y*, reasigna el nodo derecho de *x* como el hijo izquierdo del nodo *y*, luego define el nodo izquierdo de *y* como el nodo *x* y retorna el nodo *y*.
- **Node splay(Node root, int key)**: Este método corresponde a la función *splay(x)* mencionada en la introducción, la cual tiene como objetivo hacer las rotaciones necesarias para llevar el nodo accedido a la raíz del árbol, para esto recibe el nodo root que es la raíz actual del árbol y la clave key a buscar para llevar a la raíz. Cualquiera sea la key que se desea llevar a la raíz existen 3 casos, los cuales se abordarán de la siguiente manera:
  - El árbol esta vacío o la clave ya esta en la raíz: Este corresponde al caso base y retorna el nodo root.
  - La key es menor al elemento que está almacenado en root: Este caso corresponde a cuando el elemento buscado se encuentra en el subárbol izquierdo de root, esta situación es posible dividirlo en 3 subcasos:
    - Si no hay subárbol izquierdo: Se retorna la root.
    - Si la key es menor que el elemento almacenado en el hijo izquierdo de root: Este caso corresponde a la operación de zig-zig y para realizarse se redefine al nodo izquierdo del hijo izquierdo de root como la llamada recursiva de splay con el nodo izquierdo del hijo izquierdo de root y el key. Finalmente se define el nodo root como la llamada a rightRotate del nodo root actual. Es decir, este es el caso donde la key se encuentra en el subárbol izquierdo del subárbol izquierdo de la raíz, por lo cual se llama recursivamente a splay para traer key hacia arriba y luego realiza una rotación hacia la derecha de root.
    - Si la key es mayor que el elemento almacenado en el hijo izquierdo de root: Este caso es equivalente a zig-zag y para realizarse se redefine el nodo derecho del hijo izquierdo de root como la llamada recursiva a splay con el nodo derecho del hijo izquierdo de root y la key. Luego se evalúa si el nodo derecho del hijo izquierdo de root es distinto a nulo en cuyo caso se define al hijo izquierdo de root como la llamada a leftRotate con el nodo izquierdo de root. Este caso corresponde a cuando la clave se encuentra en el subárbol derecho del subárbol izquierdo de root, por lo cual se hace un llamado recursivo de splay en el nodo derecho del hijo izquierdo de root para llevar la key hacia arriba y luego se realiza una rotación hacia la izquierda en el subárbol izquierdo de root de ser necesario.

    Finalmente si el nodo izquierdo de root es nulo retorna root y en caso contrario retorna un llamada a rightRotate con el nodo root, es decir, se realiza una rotación zig en caso de ser necesario.
  - La key es mayor que el elemento almacenado en root: Este caso corresponde a cuando el elemento buscado se encuentra en el subárbol derecho de root, al igual que el caso anterior este se puede dividir en 3 subcasos:
    - Si no hay subárbol derecho: se retorna root
    - Si la key es menor que el elemento almacenado en el hijo derecho de root: Este caso corresponde a hacer una rotación zag-zag, es decir, la key se encuentra en el subárbol derecho del subárbol derecho de root, para esto se hace un llamado recursivo a splay con el nodo derecho del hijo derecho de root para llevar key hacia arriba y luego se realiza una rotación izquierda de root. La implementación de esto corresponde a redefinir el nodo derecho del hijo derecho de root como la llamada recursiva de splay con nodo derecho del hijo derecho y la key, y luego redefinir root como el llamado a leftRotate del nodo root actual.
    - Si la key es mayor que el elemento almacenado en el hijo derecho de root: Este caso corresponde con hacer una rotación zag-zig pues la key se encuentra en el subárbol izquierdo del subárbol derecho de root, para hacer esto se hace un llamado recursivo a splay para traer la key hacia arriba y luego se realiza una rotación hacia la derecha del subárbol derecho de root de ser necesario. La implementación de esto es tal que se define el nodo izquierdo del subárbol derecho de root como una llamada recursiva a splay con el nodo izquierdo del subárbol derecho de root y la key, luego se evalúa si el nodo izquierdo del subárbol derecho de root es distinto a null, en

cuyo caso se redefine el hijo derecho de root como una llamada a rightRotate del nodo derecho de root.

Finalmente se realiza una rotación zag de ser necesaria evaluando si el nodo derecho de root es nulo, en cuyo caso se retorna root y en caso contrario se retorna una llamada a leftRotate de root.

- **void insert(int key)**: Este método es el encargado de insertar elementos al Splay Tree, para esto comienza por evaluar el caso base de que el árbol este vacío, en cuyo caso se crea un nuevo nodo como raíz. Luego de esto se lleva el nodo con la clave a la raíz usando splay y si la clave ya existe en la raíz retorna sin hacer nada, en caso contrario se crea un nuevo nodo con la key y se reorganiza el árbol, para implementar la reorganización se evalúa si la key es menor o mayor al elemento almacenado en la raíz, de ser menor se define el hijo derecho del nuevo nodo como la raíz actual, el hijo izquierdo del nuevo nodo como el hijo izquierdo de la raíz y finalmente el nodo izquierdo de la raíz como nulo; de ser mayor se define el nodo izquierdo del nuevo nodo como la raíz, el hijo derecho como el nodo izquierdo de la raíz y el hijo derecho de la raíz como nulo. Luego de esta reorganización se define la nueva raíz como el nuevo nodo.
- **boolean search(int key)**: Corresponde al método de búsqueda en el Splay Tree, para implementarlo se redefine root como el splay del nodo root con la key, lo cual lleva el nodo con la clave buscada a la raíz, y finalmente se retorna la verificación de si la clave está o no en la raíz actual, es decir, retorna true si el elemento buscado está en el árbol y falso si no.

2.4. IterativeBinarySearchTree

Esta clase corresponde a una versión iterativa de implementar los ABB clásicos, por lo cual es equivalente a la clase de BinarySearchTree, esta cuenta con los siguientes métodos:

- **void insert(int key):** Este se encarga de insertar la clave key al árbol, para esto comienza por crear un nuevo nodo que contenga a key, luego si el árbol está vacío se asigna el nuevo nodo como la raíz del árbol, de no estarlo se comienza una búsqueda de la posición donde se insertará el nuevo nodo, para esto se definen los nodos current y parent como la raíz y null respectivamente, luego se recorre el árbol hasta encontrar el lugar adecuado para el nuevo nodo mediante un loop while que se repite hasta que el nodo actual sea nulo, dentro de cada iteración de este se redefine el nodo padre como el actual y el nodo actual como el hijo izquierdo o derecho de si mismo dependiendo si es menor o mayor respectivamente, además si la clave ya existe dentro del árbol se retorna sin hacer la inserción. Una vez que ya se tiene el nodo donde debe ir el elemento, se inserta como el hijo izquierdo o derecho del nodo parent siguiendo la misma condición de dentro del while.
- **boolean search(int key):** Corresponde al método encargado de hacer la búsqueda de key dentro del árbol, esta se inicia desde la raíz de este definiendo un nodo current como la raíz y lo recorre de manera iterativa utilizando un loop while hasta que el nodo actual sea nulo, dentro de cada iteración se evalúa si la key buscada es igual a la clave del nodo current, de serlo significa que el elemento fue encontrado y retorna true, en caso contrario se redefine el nodo current como su hijo izquierdo o derecho si la key buscada es menor o mayor a la clave del nodo actual respectivamente. Finalmente si se llego a un nodo nulo y el elemento no fue encontrado se retorna false.

2.5. IterativeSplayTree

De la misma forma que los IterativeBinarySearchTree esta clase corresponde a una clase equivalente a SplayTree pero que está implementada de manera iterativa en vez de recursiva, esta mantiene los métodos de rotación de SplayTree pero implementa los siguientes métodos:

- **Node splay(int key):** Corresponde al encargado de hacer las rotaciones necesarias para que el nodo accedido se lleve a la raíz del árbol, para esto comienza por evaluar si la raíz del árbol esta vacía o key ya esta en la raíz, en cuyo caso retorna root. Si esto no ocurre define el nodo current como la raíz, dummy que será un nodo auxiliar, leftTreeMax como el nodo dummy y rightTreeMin también como el nodo dummy. Luego hace un bucle while hasta que el nodo actual sea nulo o la llave del nodo actual sea igual a la clave buscada, dentro de cada iteración se evalúa si la key buscada es menor o mayor al elemento del nodo actual y para cada caso se hace lo siguiente:

- key < current.key: Se evalua si el nodo izquierdo del nodo actual es nulo, en cuyo caso se rompe la iteración, en otro caso si key es menor a la clave del hijo izquierdo de current se realiza una rotación a la derecha del nodo actual y se corrobora que el nodo izquierdo de current no sea nulo, luego se conecta el subárbol derecho definiendo el hijo izquierdo de rightTreeMin como el nodo actual, redefiniendo rightTreeMin como el nodo actual y el nodo actual como su hijo izquierdo.
- key > current-key: Se verifica que el hijo derecho del nodo actual no sea nulo, luego si la key es mayor al elemento del hijo derecho del nodo actual, en cuyo caso se realiza una leftRotate al nodo actual y se verifica que su nodo derecho no sea nulo, luego se conecta el subárbol izquierdo definiendo el hijo derecho de leftTreeMax y leftTreeMax como el nodo actual y redefiniendo el nodo actual como su hijo derecho.

Finalmente se reconstruye el árbol definiendo el hijo derecho de leftTreeMax como el hijo izquierdo del actual, el hijo izquierdo de rightTreeMin como el nodo derecho del nodo actual, el nodo izquierdo de current como el hijo derecho de dummy, el nodo derecho de current como el hijo izquierdo de dummy y la raíz como el nodo actual, retornando esta.

- **void insert(int key):** Comienza por verificar si el árbol esta vacío, en cuyo caso inserta el elemento en la raíz, en caso de que si hayan elementos en el árbol utiliza el método splay(key) para llevar a la raíz el nodo donde deber ser insertada la calve, si esta ya esta en la raíz no se hace nada pero en el otro caso se crea un nuevo nodo y se reorganiza el árbol, para esto se evalúa si la key es menor al elemento almacenado en la raíz en cuyo caso define el nodo derecho del nuevo nodo como la raíz, el izquierdo como el hijo izquierdo de la raíz y el nodo izquierdo de la raíz como nulo, en el caso de que sea mayor se define el nodo izquierdo del nuevo nodo como la raíz, el derecho como el hijo derecho de la raíz y el nodo derecho de la raíz como nulo. Finalmente se redefine la raíz como el nuevo nodo.
- **boolean search(int key):** Para buscar este método trae a la raíz la key buscada usando la función splay(key), si la raíz no es nula y el elemento dentro de ella es igual a la key buscada retorna true y en caso contrario retorna false.

2.6. Experimento 1

Es la clase correspondiente a la realización del experiemento 1, es decir, se encarga comparar los rendimientos de ABB clásico y Splay Tree para esto inserta N enteros distintos de manera aleatoria y hacer M búsquedas de valores escogidos entre los insertados con $N \in 10^6 * \{0.1, 0.2, ..., 1.0\}$ y $M = 100 * N$ para cada uno y registra en un archivo CSV la iteración, el N, el M, el tiempo de inserción y búsqueda de ambos árboles, y el costo promedio de búsqueda de ambos árboles, el cual es calculado mediante el tiempo que demora en hacerse las operaciones.

Para generar la implementación del experimento se comienza por definir los factores de N, es decir, se hace un arreglo de $\{0.1, 0.2, ..., 1.0\}$, se define un número random que se utiliza posteriormente para crear los elementos a insertar y se crea una variable iteración para contabilizarlas. Luego se crea un archivo CSV donde se guardarán los resultados.

Ya en a la implementación del experimento se hacen los siguientes pasos para cada iteración sobre los factores:

1. Se definen N como el factor por 10^6 y M como $N * 100$
2. Se inicializan ambos árboles vacíos usando las clases BinarySearchTree y SplayTree.
3. Se generan los N enteros únicos que se utilizarán para la inserción utilizando el número aleatorio antes mencionado y guardando cada uno en una instancia de HashSet para asegurar que no existirán duplicados, luego se traspasan a una lista y esta se desordena para que las inserciones ocurran en un orden aleatorio.
4. Se definen las variables startTime y endTime para registrar el tiempo que tarda la inserción y búsqueda.
5. Se insertan los N elementos en el ABB y se guarda el tiempo tardado en una variable tiempoInsercionABB. Y se hace equivalentemente para el Splay Tree.
6. Se crea una lista de elementos con M copias de elementos del arreglo de números insertados y se desordenan, esta lista será la utilizada para hacer las M búsquedas.

7. Se realizan las M búsquedas en el árbol ABB, se registra el tiempo tardado en esto en una variable tiempoBusquedaABB y se calcula el costo promedio guardándolo en una variable costoPromedioABB. Y se realiza el proceso equivalente para Splay Tree.
8. Se guardan los resultados de la iteración en al archivo CSV.

2.7. Experimento 2

Corresponde a la clase encargada de generar el segundo experimento, es decir, en ella se insertan N enteros distintos de manera aleatoria y se buscan M elementos a los cuales se les asignarán probabilidades sesgadas de búsqueda para ambos árboles y se compara su rendimiento de costo promedio de búsqueda guardando estos datos en un archivo CSV equivalente al del experimento 1 para $N \in 10^6 * \{0.1, 0.2, ..., 1.0\}$ y $M = 100 * N$.

Al igual que en el experimento anterior se comienza por definir una lista de los factores de N, un número aleatorio, un contador de las iteraciones y se crea el archivo CSV.

Adentrándose en la implementación del experimento se tendrán los mismos procedimientos que el experimento 1 a excepción del paso 6 que ahora corresponde a:

6. Se crea la lista de M elementos con probabilidades sesgadas que se utilizará para realizar las búsquedas, para esto se hacen los siguientes pasos:
   - Se calcula la constante de la función de probabilidades utilizando la función calculateC definida al final de la clase, esta dada la cantidad de elementos que fueron insertados calcula que la suma acumulada de las probabilidades, y luego divide 1 entre esta suma para que dada la constante C, la suma acumulada de como resultado 1.
   - Crea una lista para ir guardando los elementos a buscar
   - Itera sobre la cantidad de elementos en N y para cada uno calcula una frecuencia como el valor entero de la función inversa al cuadrado tal que a medida que aumenta el indice de la posición de un elemento de la lista insertada, la probabilidad disminuye y lo multiplica por M para que la escala de la probabilidad se ajuste a M.
   - Agrega a la lista de números a buscar el elemento tantas veces como indique la frecuencia calculada.
   - Desordena la lista de números a buscar

2.8. Experimento 3

Esta clase será la encargada de hacer el tercer experimento, es decir, insertará N elementos distintos de manera ordenada y se harán M búsquedas de valores escogidos aleatoriamente entre los insertados en ambos árboles de búsqueda y compara sus rendimientos del costo promedio.

Para la implementación se hacen los mismos pasos que en la del experimento 1 a excepción de los siguientes pasos donde se hicieron las siguientes modificaciones:

3. Se crean los N enteros únicos de la misma manera que en el experimento anterior pero esta vez se ordenan de menor a mayor.

Otra modificación importante es que en vez de utilizar las clases BinarySearchTree y SplayTree utiliza sus versiones iterativas IterativeBinarySearchTree y IterativeSplayTree.

2.9. Experimento 4

Corresponde a la clase encargada de implementar el experimento 4, es decir, antes de insertar los N elementos se realiza una copia de estos a otro arreglo que se ordenará y se utilizará para realizar las inserciones, luego se hará la búsqueda de M elementos asignando probabilidades sesgadas de búsqueda a los del arreglo original utilizando una función del mismo tipo que en el segundo experimento; para ambos tipos de árboles y se compararán sus resultados.

La implementación se hace de manera muy similar a la del experimento 2 con las siguientes modificaciones:

- Entre el paso 3 y 4 se agrega un nuevo paso que se encarga de hacer una copia del arreglo desordenado de elementos a insertar y ordena los elementos de la copia de menor a mayor, esta será la lista de elementos a insertar y la lista desordenada se utilizará más adelante para crear el arreglo de claves a buscar.
- Se utiliza el paso 6 descrito en el experimento 2 usando el arreglo desordenado.

Es importante destacar que al igual que el experimento 3 en vez de utilizar las clases BinarySearchTree y SplayTree utiliza sus versiones iterativas IterativeBinarySearchTree y IterativeSplayTree.

3. Resultados

Para obtener resultados de los 4 experimentos se utilizaron los métodos antes descritos, los cuales entregan un archivo CSV para cada uno de ellos y que posteriormente fueron procesados en Python para la obtención de gráficos. Estos experimentos fueron ejecutados en un dispositivo con sistema operativo Windows11 con memoria RAM de 16GB, y caches L1: 1.1 MB, L2: 9 MB y L3: 12 MB con los siguientes input para cada experimento:

- Experimento 1 y 2: $N \in 10^6 * \{0.1, 0.2, ..., 1.0\}$ y $M = 100 * N$.
- Experimento 3 y 4: $N \in 10^6 * \{0.01, 0.02, ..., 0.1\}$ y $M = 100 * N$ en este caso se modificó N multiplicando los factores por 0.1 para disminuir el tiempo de ejecución.

A continuación se pueden observar los gráficos resultantes (Figura 1, Figura 2, Figura 3 y Figura 4), para cada experimento se generó un esquema para el tiempo de inserción, de búsqueda y uno del costo promedio de búsqueda con respecto a la cantidad de elementos insertados *N*, cabe recalcar que no se incluyó la variable *M* en estos ya que es una variable que aumenta de forma directamente proporcional a *N*. También es importante notar que para los experimentos 1 y 2 su eje absiso se encuentra en el orden de cientos de miles, mientras que en los experimentos 3 y 4 dado que los factores de *N* fueron multiplicados por 0.1, el eje absiso se encuentra en escala logarítmica para que los datos se apreciaran de mejor manera.

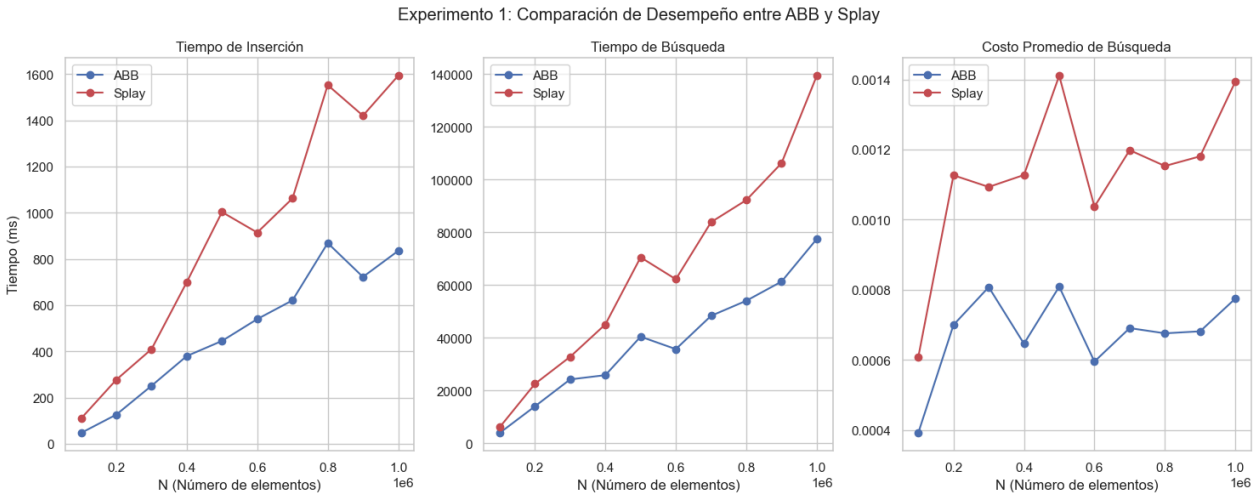

Figura 1: Gráficos resultantes del experimento 1

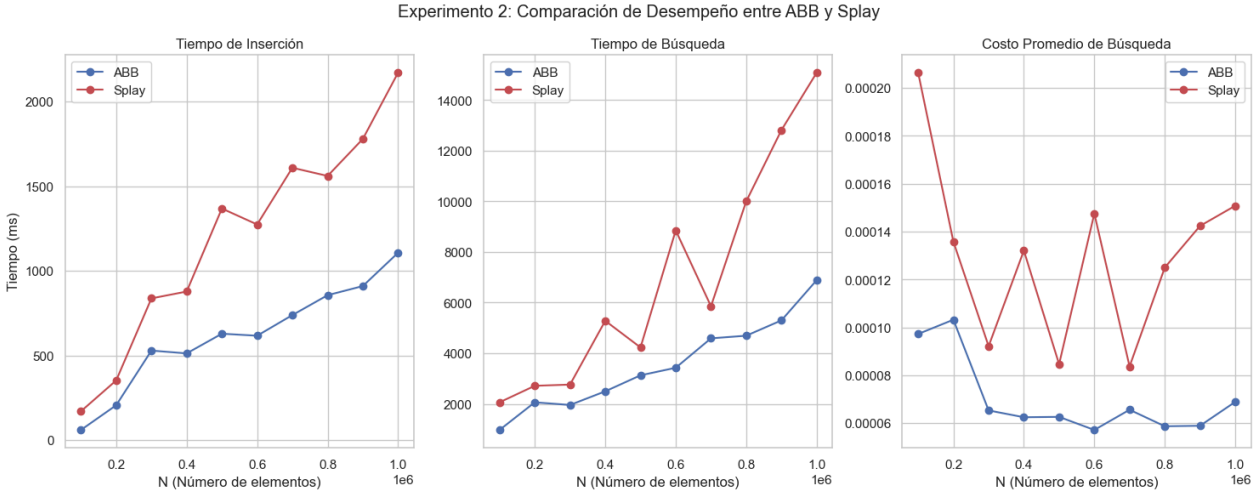

Figura 2: Gráficos resultantes del experimento 2

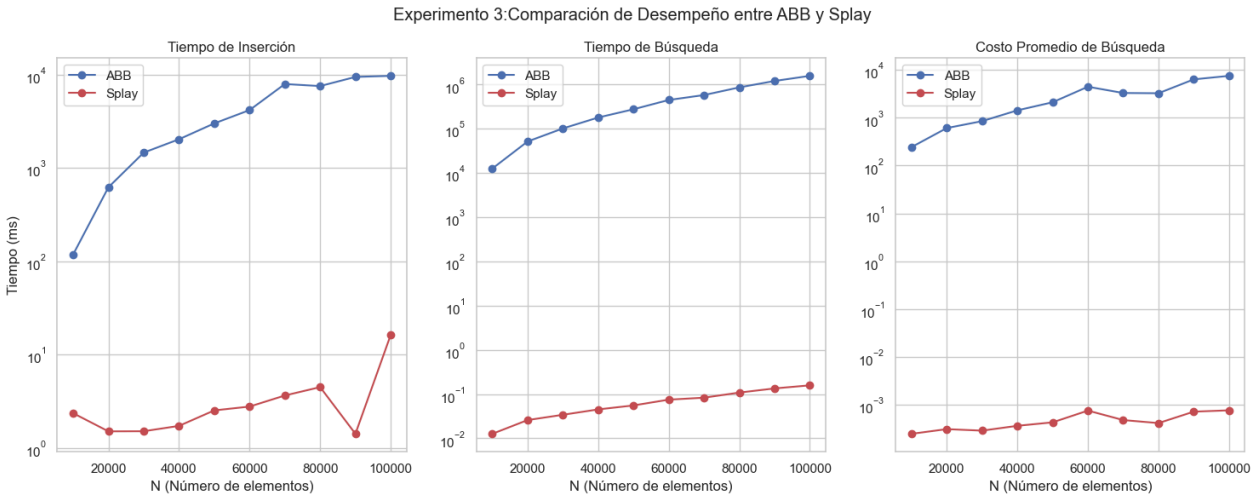

Figura 3: Gráficos resultantes del experimento 3

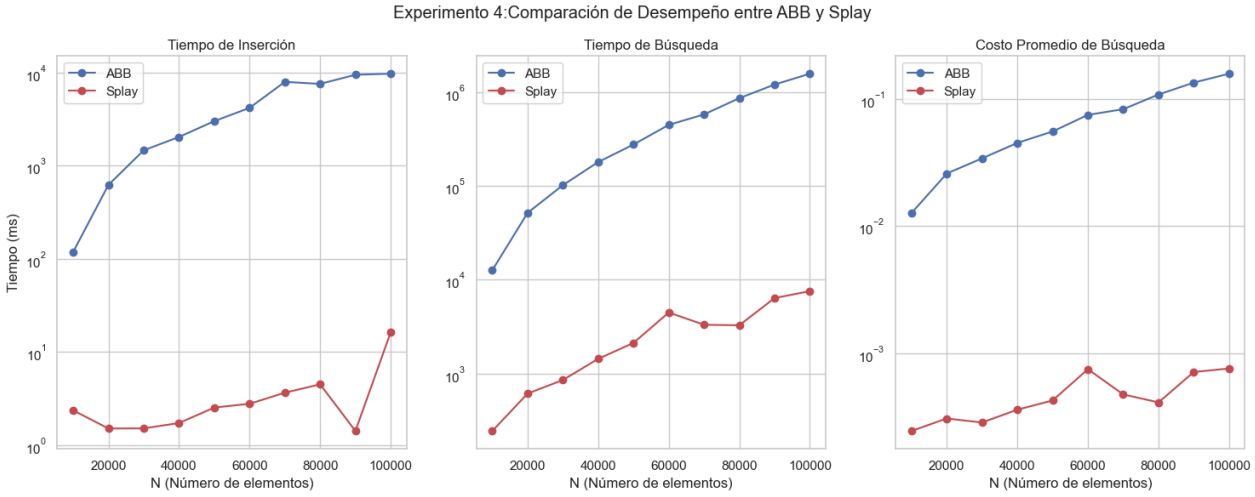

Figura 4: Gráficos resultantes del experimento 4

Con respecto a los gráficos resultantes es posible observar que todos a excepción del experimento 2 comparten que a medida que aumenta la cantidad de elementos insertados también aumenta el tiempo de inserción, el de búsqueda y el costo promedio de búsqueda mientras que en el segundo experimento el costo de búsqueda disminuye al aumentar la cantidad de elementos insertados. Luego visualizando los gráficos para cada uno se tiene que en el primer y segundo experimento (Figura 1 y Figura 2) la curva del Splay Tree se encuentra por sobre la del ABB clásico en todo momento, mientras que para el tercer y cuarto experimento (Figura 3 y Figura 4) la curva del ABB clásico se encuentra por sobre la del Splay Tree en todo momento.

4. Análisis

4.1. Experimento 1

Para el experimento 1 notamos que el Splay Tree es menos eficiente que el ABB. A medida de que aumenta tanto el número de inserciones como el de búsquedas el ABB presenta un menor tiempo para ejecutar ambas operaciones. En general para los gráficos de tiempo de inserción y búsqueda (Figura 1) se observa un comportamiento directamente proporcional, el tiempo llega a su máximo para el N más grande.

El costo promedio de búsqueda también parece ser directamente proporcional salvo entre 0.4 y 0.7 ya que acá alcanza el máximo de costo promedio tanto para el ABB como para el Splay.

4.2. Experimento 2

Para el experimento 2 vemos cosas distintas en los gráficos (Figura 2) en comparación al experimento 1. No obstante a estas diferencias, el ABB claramente sigue teniendo mejor rendimiento que Splay, a parte para el tiempo de inserción y de búsqueda se sigue comportando directamente proporcional.

La diferencia está en como se comporta el costo promedio de búsqueda. Se puede observar que este tiene un comportamiento inversamente proporcional, a medida de que más números se buscan el costo promedio mejora, esto es más evidente para el ABB ya que para Slay pareciera oscilar este comportamiento.

4.3. Experimento 3

Acá ya notamos grandes diferencias a partir de los resultados (Figura 3), primero que todo se tuvo que cambiar la escala del eje Y a una logarítmica, ya que los datos obtenidos por Splay son muy pequeños en comparación a los de Abb en términos de unidades.

En este caso es Splay el que empieza a tener mejor rendimiento que ABB,la diferencia entre el tiempo de inserción, de búsqueda y costo promedio son demasiado grandes, hablamos de 10 elevado a 6 en algunos casos de diferencia de magnitud. Esto es constante a medida de que el N va aumentando.

4.4. Experimento 4

Ocurre exactamente lo mismo que para el experimento 3. Se tuvo que cambiar la escala del eje Y a logarítmica. El rendimiento de Splay es mucho mejor que el de ABB. Una observación importante en este escenario ocurre al comparar los árboles con los resultados obtenidos en el experimento anterior Figura 3, para el árbol ABB no se notan diferencias en el tiempo de búsqueda, pues este simula una lista enlazada y la búsquedas de valores no afectan la estructura del árbol. Aún así, la disminución del costo promedio de búsqueda se puede atribuir a que debido al sesgo de probabilidad números con mejor prioridad en el orden de inserción se buscaron más y ayudaron a que esta métrica disminuya. Por otra parte, para el Splay Tree se observa que el tiempo de búsqueda aumenta, lo que se puede atribuir al desbalance generado en el árbol por el sesgo de probabilidades al buscar, en donde algunas son más frecuentes y generan que el árbol sea re-estructurado en favor de ellas.

Esto se nota a para cada uno de los gráficos (Figura 4), todos son directamente proporcionales.

5. Conclusión

Esta investigación permitió comparar efectivamente el desempeño de los árboles ABB y Splay en distintos escenarios, analizando tiempos de inserción, búsqueda y búsqueda promedio para altas cantidades de datos. La experimentación propuesta permite observar el comportamiento de los árboles, en donde fue posible determinar la veracidad de nuestras hipótesis:

1. Experimento 1: El ABB clásico resultó ser más eficiente que el Splay Tree, coincidiendo con lo planteado de que en búsquedas uniformes y aleatorias el Splay pierde su ventaja de reorganización del árbol y esta se vuelve un costo extra no beneficioso.

2. Experimento 2: En este caso la hipótesis planteada fue incorrecta, el Splay Tree tuvo un desempeño peor al del ABB clásico, el resultado se puede relacionar a que aunque hayan valores que se busquen más que otros, la aleatoriedad en el arreglo de búsqueda B sigue jugando un papel más importante en que estas consultas con mayor *f(i)* no sean seguidas/frecuentes entre sí. Una observación importante a destacar es que aunque la hipótesis fue incorrecta, el Splay Tree si tuvo una mejora en comparación al experimento 1, atribuible a nuestro argumento.

3. Experimento 3: Nuestra hipótesis coincide con los resultados obtenidos, mostrando un mejor desempeño del Splay Tree en comparación al ABB clásico cuando se trabaja con datos insertados en orden, esto gracias a que el Splay Tree mantiene un balance en sus nodos al reorganizarse en cada búsqueda, mientras que el ABB se extiende por sus hijos derechos, quedando desbalanceado y simulando una lista enlazada.

4. Experimento 4: Nuevamente nuestra hipótesis coincide con los resultados obtenidos, mostrando un mejor desempeño del Splay Tree cuando los datos a buscar fueron insertados en orden aunque tenga un sesgo de probabilidad que provoca un desbalance del árbol en favor de aquellos valores más buscados, esto se debe a la misma razón que para el experimento 3, en el que el ABB clásico se termina convirtiendo en una lista enlazada debido a su gran desbalance .

Dentro del trabajo a futuro consideramos que uno de los principales desafíos de esta investigación es el extenso tiempo de ejecución que llevan algunas etapas, lo que nos llevo a realizar los experimentos 3 y 4 con los valores propuestos de N multiplicados por 0.1. Por tanto, proponemos realizar nuevamente los experimentos 3 y 4 considerando los valores $N \in 10^6 * \{0.1, 0.2, ..., 1.0\}$. Asimismo, se propone incluir el uso de técnicas de paralelización para optimizar los tiempos al utilizar más cores del dispositivo de ejecución y ejecutar todos los experimentos usando las versiones iterativas de los árboles para que no haya un sesgo basado en la recursión para los experimentos 1 y 2 al compararlos con los 3 y 4.